TLP **3** (4 & 5): 393–424, 2003.
DOI: 10.1017/S1471068403001832 Printed in the United Kingdom

Answer sets for consistent query answering in inconsistent databases

MARCELO ARENAS*

Pontificia Universidad Catolica de Chile, Departamento de Ciencia de Computacion, Santiago, Chile
(*e-mail:* marenas@ing.puc.cl)

LEOPOLDO BERTOSSI

School of Computer Science, Carleton University, Ottawa, Canada
(*e-mail:* bertossi@scs.carleton.ca)

JAN CHOMICKI

Department of Computer Science and Engineering, State University of New York at Buffalo, Buffalo, NY, USA
(*e-mail:* chomicki@cse.buffalo.edu)

Abstract

A relational database is *inconsistent* if it does not satisfy a given set of integrity constraints. Nevertheless, it is likely that most of the data in it is consistent with the constraints. In this paper we apply logic programming based on answer sets to the problem of retrieving consistent information from a possibly inconsistent database. Since consistent information persists from the original database to every of its minimal repairs, the approach is based on a specification of database repairs using *disjunctive logic programs with exceptions*, whose answer set semantics can be represented and computed by systems that implement stable model semantics. These programs allow us to declare persistence by default of data from the original instance to the repairs; and changes to restore consistency, by exceptions. We concentrate mainly on logic programs for binary integrity constraints, among which we find most of the integrity constraints found in practice.

KEYWORDS: databases, answer set programming, integrity constraints, consistency

1 Introduction

Integrity Constraints (IC) capture an important normative aspect of every database application, whose aim is to guarantee the consistency of its data. However, it is very difficult, if not impossible, to always have a consistent database instance. Databases may become inconsistent with respect to a given set of integrity constraints. This may happen due, among others, to the following factors: (1) Certain ICs cannot

* Current address: University of Toronto, Department of Computer Science, Toronto, Canada. E-mail: marenas@cs.toronto.edu.

be expressed/maintained by existing DBMSs; (2) transient inconsistencies caused by the inherent non-atomicity of database transactions, (3) delayed updates of a datawarehouse, (4) integration of heterogeneous databases, in particular with duplicated information; (5) inconsistency with respect to *soft* integrity constraints, where transactions in violation of their conditions are not prevented from executing; (6) legacy data on which one wants to impose semantic constraints; (7) the consistency of the database will be restored by executing further transactions; and (8) user constraints than cannot be checked or maintained.

Independently of the cause of inconsistency, the inconsistent database may be the only source of data available, and we may still want or need to use it for a number of reasons. Restoring the consistency of the database may not be an option since that may require permissions we don't have, lead to the loss of useful information, or be a complex and non-deterministic process. Under such circumstances, one faces the natural problem of characterizing and retrieving the *consistent* information from the database. Most likely, most of the information in the database is still consistent, and the database can still provide us with correct answers to certain queries, making the problem of determining what kinds of queries and query answers are consistent with the integrity constraints a worthwhile effort.

The problem of defining and retrieving consistent information from an inconsistent relational database has been studied in the context of relational databases. The basic approach is based on the intuition that the information that is consistent, despite the inconsistency of the database as a whole, is the one that is invariant under all sensible ways in which the consistency of the database is restored. More precisely, an answer to a query is consistent if it is obtained as an answer every time the query is posed to a minimally repaired version of the original database (Arenas *et al.*, 1999).

Example 1

Assume we have the following database instance *Salary*:

Salary	*Name*	*Amount*
	V.Smith	5000
	V.Smith	8000
	P.Jones	3000
	M.Stone	7000

and *FD* is the functional dependency $Name \rightarrow Amount$, meaning that *Name* functionally determines *Amount*, that is violated by the table *Salary*. Actually the tuples participating in this violation are those with the value *V.Smith* in attribute *Name*.

When we ask about the tuples that are consistent wrt the FD, we should retrieve only (*P.Jones*, 3000) and (*M.Stone*, 7000), because those tuples should stay in any reasonable way in which we restore consistency.

If we want to consider only repaired versions of the original instance that minimally differ from the original instance, in the sense that the set of inserted or deleted tuples (to restore inconsistency) is minimal under set inclusion, the possible *repairs* of the inconsistent database are

$Salary_1$	*Name*	*Amount*
	V .Smith	5000
	P .Jones	3000
	M .Stone	7000

$Salary_2$	*Name*	*Amount*
	V .Smith	8000
	P .Jones	3000
	M .Stone	7000

We can see that only tuples (*P .Jones*, 3000) and (*M .Stone*, 7000) can be found in both repairs. □

In this paper, we address the problem of retrieving consistent information when general first order queries are posed to an inconsistent relational database. Since the consistent information in the database is the one that persists across all repairs, we solve this problem by using logic programs with answer sets semantics to specify in a compact manner the class of repairs of the inconsistent instance.

Although the consistent answers are defined in terms of minimally repaired version of the database, we are not interested in restoring consistency, in particular, in computing the repairs of the database: Repairs are used as an auxiliary notion in order to give a model-theoretic characterization of the consistent answers to queries. Actually, it is easy to find situations where exponentially many repairs of an inconsistent database exist (Arenas *et al*., 2001).

A possible computational mechanism for retrieving consistent answers, first introduced in Arenas *et al.* (1999) and Celle and Bertossi (2000), is as follows: Given a first-order query Q and an inconsistent database instance r, instead of explicitly computing all the repairs of r and querying all of them, a new query $T(Q)$ is computed and posed to r, the only available database. The answers to the new query are expected to be the consistent answers to Q. Such an iterative operator for query transformation was introduced and analyzed with respect to soundness, completeness and termination in Arenas *et al.* (1999) and Celle and Bertossi (2000).

Nevertheless, the query rewriting approach has some limitations. The iterative operator introduced in Arenas *et al.* (1999) and Celle and Bertossi (2000) works for some particular classes of queries and constraints, e.g. for queries that are conjunctions of literals and universal integrity constraints, but completeness is lost when it is applied to disjunctive or existential queries. The methodology for obtaining consistent answers that we present in this paper can applied to any first-order query instead.

Furthermore, the notion of consistent answer introduced in Arenas *et al.* (1999) is a model theoretic notion, that is complemented by a computational mechanism. Nevertheless, that approach is not based on or accompanied by a *logical specification* of the class of all the repairs of a given database instance relative to a fixed set of ICs. Such a specification is another contribution of this paper, namely a specification expressed as a disjunctive logic program with answer set semantics. The database repairs correspond to the intended models or answer sets of the program.

In this paper, we are motivated mainly by the possibility of retrieving consistent answers to general first-order queries, extending the possibilities we developed in Arenas *et al.* (1999). However, the logical specification could be also used to (1) Reason about all database repairs, in particular about consistent query answers,

(2) derive specialized algorithms for consistent query answering, (3) analyze complexity issues related to consistent query answering, and (4) obtain the intended models of the specification, i.e. the database repairs, allowing us to analyze different ways to restore the consistency of the database. That is, a mechanism for computing database repairs could be used for conflict resolution.

Notice that consistent answers are non-monotonic in the sense that adding information to the original database, may cause loosing previous consistent answers. In consequence, a non-monotonic semantics for the specification (or its consequences) should be expected.

A preliminary version of this paper appeared in Arenas *et al.* (2000a), where extended disjunctive logic programs with exceptions where introduced and applied to the specification of database repairs and to retrieve consistent answers to general first-order queries. This paper extends Arenas *et al.* (2000a), addressing several new issues, among which we find (1) a detailed analysis of the correspondence between e-answer sets and database repairs for binary integrity constraints, (2) application of the *DLV* system (Eiter *et al.*, 1998) to obtain database repairs and consistent answers, (3) extensions of the methodology to more general universal constraints and to referential integrity constraints, (4) an analysis of the applicability of the disjunctive well-founded semantics to consistent query answering, and (5) the use of weak constraints to capture database repairs based on minimal *number* of changes.

This paper is structured as follows. In section 2 we introduce the notions of database repair and consistent answer to a query, and the query language. Section 3 introduces extended disjunctive logic programs with exceptions. In section 4, the main section of the paper, we present the repair programs for binary integrity constraints, and show how to consistently evaluate queries. In section 5 we show some examples using the *DLV* system to obtain database repairs and consistent answers. In section 6 we illustrate how to handle referential integrity constraints. In section 7 we analyze the well-founded interpretation as an approximation to the set of consistent answers, and we identify cases where it provides the exact solution. In section 8 we show how database repairs based on minimal number of changes can be specified by introducing weak constraints in the repair programs. In section 9, we draw conclusions, we sketch some extensions, e.g. to the case of general universal ICs, we also mention open issues, and discuss related work.

2 Consistent query answers

A database schema can be represented by a typed language $\mathscr{L}$ of first-order predicate logic, that contains a finite set of predicates and a fixed infinite set of constants D. A relational database instance r can be seen as an interpretation or a first order structure for $\mathscr{L}$, whose domain is also D (the interpretation of each constant is the constant itself), and the predicates have finite extensions. In what follows, a database instance r will be represented, in a natural way, as a finite set of ground atoms (the atoms true in r).

The active domain of a database instance r is the set of those elements of D that explicitly appear in r. The active domain is always finite and we denote it by $Act(r)$.

We may also have a set of built-in (or evaluable) predicates, like equality, order relations, arithmetical relations, etc. In this case, we have the language $\mathscr{L}$ possibly extended with those predicates. In all database instances, for a given schema, each of these predicates has a fixed and possibly infinite extension. Since we defined database instances as finite sets of ground atoms, we are not considering those built-in atoms as members of database instances.

In addition to the database schema and instances, we may also have a set of integrity constraints IC expressed in language $\mathscr{L}$. These are first-order formulas which the database instances are expected to satisfy. If a database instance r satisfies IC in the standard model-theoretic sense, what is denoted by $r \models IC$, we say that it is consistent (wrt IC), otherwise we say it is inconsistent. In any case, we will assume from now on that IC is logically consistent set of first-order sentences.

The original motivation in Arenas *et al.* (1999) was to consistently answer first-order queries. We shall call them *basic queries* and define them by the grammar

$$B \ ::= \ Atom \mid B \wedge B \mid \neg B \mid \exists x \ B.$$

One way of explicitly asking at the object level about the consistent answers to a first-order query consists in introducing a new logical operator $\mathscr{K}$, in such a way that $\mathscr{K}\varphi(\bar{x})$, where $\varphi(\bar{x})$ is a basic query, asks for the values of $\bar{x}$ that are consistent answers to $\varphi(\bar{x})$ (or whether φ is consistently true, i.e. true in all repairs, when φ is a sentence). The *$\mathscr{K}$-queries* are similarly defined:

$$A \ ::= \ \mathscr{K}B \mid A \wedge A \mid \neg A \mid \exists x. \ A.$$

In this paper, we concentrate mostly on answering *basic $\mathscr{K}$-queries* of the form $\mathscr{K}B$, where B is a basic query.

Definition 1

(a) (Arenas *et al.*, 1999) Given a database instance r and a set of integrity constraints, IC, a *repair* of r wrt IC is a database instance r', over the same schema, that satisfies IC and such that $r\Delta r' = (r \setminus r') \cup (r' \setminus r)$, the symmetric difference of r and r', is minimal under set inclusion.

(b) (Arenas *et al.*, 1999) A tuple $\bar{t}$ is a *consistent answer* to a first-order query $Q(\bar{x})$, or equivalently, an answer to the query $\mathscr{K}Q(\bar{x})$, in a database instance r iff $\bar{t}$ is an answer to query $Q(\bar{x})$ in every repair r' of r wrt IC. In symbols:

$$r \models \mathscr{K}Q[\bar{t}] \quad \Longleftrightarrow \quad r' \models Q[\bar{t}] \quad \text{for every repair } \ r' \ \text{ of } \ r.$$

(c) If Q is a general $\mathscr{K}$-query, then $r \models Q$ is defined inductively as usual, (b) being the base case.

Example 2

(Example 1 continued.) For the inconsistent database and the given FD, $\bar{t}_3 = (P.Jones, 3000)$ is a consistent answer to the query $Salary(\bar{x})$, i.e. $r \models \mathscr{K}Salary(x, y)[(P.Jones, 3000)]$, but $r \not\models \mathscr{K}Salary(x, y)[(V.Smith, 8000)]$. It also holds $r \models \mathscr{K}(Salary(V.Smith, 5000) \vee Salary(V.Smith, 8000))$, and $r \models \mathscr{K}\exists X(Salary(V.Smith, X) \wedge X > 4000)$.

Computing consistent answer through generation of all possible repairs is not a natural and feasible alternative (Arenas *et al.* 2001). Instead, an approach based on querying the available, although inconsistent, database is much more natural. This rewriting approach introduced in Arenas *et al.* (1999) is not complete for disjunctive or existential queries, like $\exists Y\, Salary(V.Smith, Y)$ in Example 2. We would like to be able to obtain consistent answers to basic $\mathscr{K}$-queries at least.

Notice that the definition of consistent query answer depends on our definition of repair. In section 8.1 we will consider an alternative definition of repair based on minimal *number* of changes instead of minimal *set* of changes.

3 Logic programs with exceptions

Logic Programs with Exceptions (LPEs) (Kowalski and Sadri 1991) have default rules whose consequences can be overridden by the consequences of exception rules. They turn out to be the right formalism for specifying the database repairs: by default everything persists from the original database instance to any of its repairs, except for the changes that are necessary to restore the consistency.

LPEs as introduced in Kowalski and Sadri (1991) consist of definite clauses, whose head and body contain literals of the form $A, \neg A$, where A is an atom and $\neg$ is classical negation. In the bodies, literals may be affected by weak negation, *not* (negation as failure). In a LPE there are *default* rules, which are clauses with positive heads, and *exception* rules, which are clauses with negative heads. To capture the intuition that exceptions have priority over defaults, in Kowalski and Sadri (1991) a new semantics was introduced based on *e-answer sets*. It is defined as follows.

First, instantiate the program Π in the database domain, making it ground. Now, let S be a set of ground literals $S = \{L, \ldots\}$. This S is a candidate to be a model, a guess to be verified, and accepted if properly justified.

Next, generate a new set of ground rules ${}^{S}\Pi$ according to the following steps:

(1) Delete every rule in Π containing $not\, L$ in the body, with $L \in S$.
(2) Delete from the clauses every condition $not\, L$ in the body, when $L \notin S$.
(3) Delete every default rule having a positive conclusion A with $\neg A \in S$.

The result is a ground extended logic program without *not*. Now, S is an *e-answer set* of the original program if S is the smallest set of ground literals, such that: (a) For any clause $L_0 \longleftarrow L_1, \ldots, L_m$ in ${}^{S}\Pi$, if $L_1, \ldots, L_m \in S$, then $L_0 \in S$; (b) if S contains two complementary literals, then S is the set of all literals.

The e-answer sets are the intended models of the original program. Above, (1), (2) are as in the *answer sets semantics* for extended logic programs (Gelfond and Lifschitz 1991), but now (3) gives an account of exceptions.

To specify database repairs, we need to extend the LPEs and their semantics as presented in Kowalski and Sadri (1991), considering *Disjunctive Logic Programs with Exceptions* (DLPEs), that contain also negative defaults, i.e. defaults with negative conclusions that can be overridden by positive exceptions, and extended disjunctive exceptions, i.e. rules of the form

$$L_1 \vee \cdots \vee L_k \longleftarrow L_{k+1}, \ldots, L_r, not\, L_{r+1}, \ldots, not\, L_m,$$

where the L_is are literals.[1] The e-answer semantics is extended as follows. The ground program is pruned according to a modified rule (3) above:

(3') Delete every (positive) default having a positive conclusion A, with $\neg A \in S$; and every (negative) default having a negative conclusion $\neg A$, with $A \in S$.

Applying (1), (2) and (3') to the ground program, we are left with a ground disjunctive logic program without *not*. If the candidate set of literals S belongs to $\alpha({}^S\Pi)$, the set of minimal models of program ${}^S\Pi$, then we say that S is an *e-answer set*.

This semantics does not capture priorities between defaults, and, in principle, there could be conflicting defaults. In this case, the semantics seems to allow that defaults override other defaults, without preferences for any of them. For example, the program containing only the defaults $p \leftarrow not\, q$ and $\neg p \leftarrow not\, r$ (without exception rules), has two e-answer sets, namely $\{p\}$ and $\{\neg p\}$. In any case, in our applications of logic programs with exceptions, due to the kind of defaults we will use (see Definition 6), such a situation will never appear, because the potentially conflicting defaults apply to mutually exclusive cases.

Finally, we take advantage of the existence of a one to one correspondence between the e-answer sets of a DLPE and the answer sets of an extended disjunctive logic program (Gelfond and Lifschitz 1991 (see section 4.1, Remark 1).

4 Logic programs for CQA

We shall use DLPEs for specifying database repairs and answering basic $\mathscr{K}$-queries. Given a set of ICs and an inconsistent database instance r, the first step consists of writing a *repair program*, $\Pi(r)$, having as the e-answer sets the repairs of the original database instance. $\Pi(r)$ captures the fact that when a database instance r is repaired most of the data persists, except for some tuples. In consequence, default rules are introduced: everything persists from the instance r to the repairs. It is also necessary to introduce exception rules: everything persists, as stated by the defaults, unless the ICs are violated and have to be satisfied.

Next, if a first-order query is posed with the intention of retrieving all and only its consistent answers, then a *query program*, that expresses the query, is run together with the repair program.

In this section we introduce the DLPEs for specifying database repairs, and give a careful analysis of those programs for consistent query answering wrt Binary Integrity Constraints (BICs), i.e. they are universally quantified sentences of the form $\overline{\forall}(L_1 \vee L_2 \vee \varphi)$, where $\overline{\forall}$ denotes the universal closure, L_1, L_2 are literals associated to the database schema; φ is a first-order formula containing only built-in predicates[2] and free variables appearing in L_1, L_2.

[1] In our application scenario we will need disjunctive exceptions rules, but not disjunctive defaults.

[2] Built-in predicates have a fixed extension in every database, in particular, in every repair; so they are not subject to changes.

We have three possibilities for BICs in terms of the sign of literals in them, namely the universal closures of:

$$p_1(\bar{x}_1) \vee p_2(\bar{x}_2) \vee \varphi; \quad p_1(\bar{x}_1) \vee \neg q_1(\bar{y}_1) \vee \varphi; \quad \neg q_1(\bar{y}_1) \vee \neg q_2(\bar{y}_2) \vee \varphi, \tag{1}$$

where the $p_i(\bar{x}_i), q_j(\bar{y}_j)$ are database atoms. BICs with one database literal plus possibly a formula containing built-ins are called *unary ICs*.

Several interesting classes of ICs (Abiteboul *et al.*, 1995) used in database praxis can be represented by BICs: (a) Range constraints, e.g. $P(x, y) \rightarrow x > 5$; (b) Full inclusion dependencies; (b) functional dependencies (see Example 1), etc. Nevertheless, for referential ICs, like in $P(x, y) \rightarrow \exists z Q(x, z)$, we need existential quantifiers or Skolem functions (Fitting, 1996). They are considered in section 6.

4.1 Finite domain databases

In this section we will momentarily depart from our assumption that databases have an infinite domain D (see section 1), and will analyze the case of finite domain databases. The reason is that in the general case, we will be interested in *domain independent* BICs, for which only the active domain is relevant (and finite).

4.1.1 The change program

To introduce the repair programs $\Pi(r)$ and analyze their behavior, we will concentrate first on the sub-program that does not contain defaults rules. This program, denoted by $\Pi_\Delta(r)$, is responsible for the changes (but not for persistence).

Splitting the program in this way makes the analysis easier. Furthermore, keeping $\Pi_\Delta(r)$, but using different form of defaults, we can capture different kinds of repairs. In section 4.1.2, we will introduce defaults leading to our notion of repair based on minimal set of changes (Definition 1). In section 8.1, we will use other defaults that lead to repairs based on minimal number of changes.

Definition 2
Given a set of BICs IC and an instance r, the *change program*, $\Pi_\Delta(r)$, contains the following rules:

1. Facts: (a) For every ground database atom $p(\bar{a}) \in r$, the fact $p(\bar{a})$.
   (b) For every a in D, the fact $dom(a)$.
2. For each IC of the forms in (1), respectively, the triggering rule

$$\begin{aligned} p'_1(\bar{X}_1) \vee p'_2(\bar{X}_2) &\longleftarrow dom(\bar{X}_1, \bar{X}_2),\ not\ p_1(\bar{X}_1),\ not\ p_2(\bar{X}_2), \overline{\varphi} \\ p'_1(\bar{X}_1) \vee \neg q'_1(\bar{Y}_1) &\longleftarrow dom(\bar{X}_1),\ not\ p_1(\bar{X}_1), q_1(\bar{Y}_1),\ \overline{\varphi} \\ \neg q'_1(\bar{Y}_1) \vee \neg q'_2(\bar{Y}_2) &\longleftarrow q_1(\bar{Y}_1), q_2(\bar{Y}_2), \overline{\varphi} \end{aligned}$$

3. For an IC of the first form in (1), the *pair* of stabilizing rules

$$\begin{aligned} p'_1(\bar{X}_1) &\longleftarrow dom(\bar{X}_1), \neg p'_2(\bar{X}_2), \overline{\varphi} \\ p'_2(\bar{X}_2) &\longleftarrow dom(\bar{X}_2), \neg p'_1(\bar{X}_1), \overline{\varphi} \end{aligned}$$

For an IC of the second form in (1), the pair of stabilizing rules

$$p'_1(\bar{X}_1) \longleftarrow dom(\bar{x}_1), q'_1(\bar{Y}_1), \overline{\varphi}$$
$$\neg q'_1(\bar{Y}_1) \longleftarrow dom(\bar{Y}_1), \neg p'_1(\bar{X}_1), \overline{\varphi}$$

For an IC of the third form in (1), the pair of stabilizing rules

$$\neg q'_1(\bar{Y}_1) \longleftarrow dom(\bar{Y}_1), q'_2(\bar{Y}_2), \overline{\varphi}$$
$$\neg q'_2(\bar{Y}_2) \longleftarrow dom(\bar{Y}_2), q'_1(\bar{Y}_1), \overline{\varphi}.$$

The primed versions $(p', \ldots)$ of the original database predicates $(p, \ldots)$ stand for the database predicates in the repairs. In these rules, $dom(.,.)$ is an abbreviation for the conjunction of memberships to dom of all the individual variables; and $\overline{\varphi}$, an abbreviation for a representation of the negation of φ. Depending on its syntax, it may be necessary to unfold the formula $\overline{\varphi}$ into additional program rules, but $\overline{\varphi}$ will usually be a conjunction of literals. □

Example 3

Consider the inclusion dependencies $IC : \{\forall xy\ (P(x,y) \rightarrow Q(x,y)), \forall xy\ (Q(x,y) \rightarrow R(x,y))\}$ and the inconsistent database instance $r = \{P(a,b), Q(a,b)\}$. The program $\Pi_\Delta(r)$ contains the following clauses:

1. *Facts*: $P(a,b),\ Q(a,b)$.
2. *Triggering exceptions*: $\neg P'(X,Y) \vee Q'(X,Y) \leftarrow P(X,Y),\ not\ Q(X,Y).$
   $\neg Q'(X,Y) \vee R'(X,Y) \leftarrow Q(X,Y),\ not\ R(X,Y).$
   Each of these rules represent the two possible ways to repair the corresponding IC, separately: The first rule says that in order to "locally" repair the first IC, either eliminate (X,Y) from P or insert (X,Y) into Q. The semantics of these DLPEs gives the disjunction an exclusive interpretation. In this example, due to the form of the ICs, we do not need domain predicates.
3. *Stabilizing exceptions*: $Q'(X,Y) \leftarrow P'(X,Y);\quad \neg P'(X,Y) \leftarrow \neg Q'(X,Y).$
   $R'(X,Y) \leftarrow Q'(X,Y);\quad \neg Q'(X,Y) \leftarrow \neg R'(X,Y).$
   These rules state that eventually the ICs have to be satisfied in the repairs. They are necessary if, like in this example, there are interacting ICs and local repairs alone are not sufficient. Propagation of changes are required beyond the first triggering step. Since the ICs can be repaired by either deleting or inserting a tuple, the contrapositive versions of the ICs are needed.

Notice that for BICs, the stabilizing rules in $\Pi_\Delta(r)$ do not contain disjunctions in the heads.

Definition 3

A *model* of a DLPE, Π, is a set of ground literals, S, that does not contain complementary literals and satisfies Π in the usual logical sense, but with weak negation interpreted as not being an element of S.

Definition 4
Given a model S of $\Pi_\Delta(r)$, we define the database instance corresponding to S by

$$I(S) = \{p(\bar{a}) \mid p'(\bar{a}) \in S\} \cup \{p(\bar{a}) \mid p(\bar{a}) \in S \text{ and } \neg p'(\bar{a}) \notin S)\}.$$

Notice that, for a given model S of the change program, $I(S)$ merges in one new instance all the positive primed tuples with all the old, non primed tuples that persisted, i.e. that their negative primed version do not belong to the model. Since there are no persistence defaults in $\Pi_\Delta(r)$, persistence is captured explicitly in $I(S)$.

Proposition 1
Given a database instance r and a set of BICs IC, if S is a model of $\Pi_\Delta(r)$, then $I(S)$ satisfies IC.

Definition 5
Given database instances r and r' over the same schema and domain, we define

$$S(r, r') = \{p(\bar{a}) \mid r \models p(\bar{a})\} \cup \{p'(\bar{a}) \mid r' \models p(\bar{a})\} \cup \{\neg p'(\bar{a}) \mid r' \not\models p(\bar{a})\}$$
$$\cup \{dom(a) \mid a \in D\}. \qquad \square$$

$S(r, r')$ collects the maximal consistent set of literals that can be obtained from two database instances, e.g. the original instance and a repair. The atoms corresponding to the second argument are primed. Negative literals corresponding to the first argument are not considered, because weak negation will be applied.

Proposition 2
Given a database instance r and a set of BICs IC, if r' satisfies IC, then $S(r, r')$ is a model of $\Pi_\Delta(r)$.

This result tells us that subsets of $S(r, r')$ could be potential models of the change program. $S(r, r')$ can be a large model, in the sense that the difference between r and r' may not be minimal.

Proposition 3
For BICs, the change program $\Pi_\Delta(r)$ has an answer set; and all the answer sets are consistent, i.e. they do not contain complementary literals.[3]

4.1.2 The repair program

Program $\Pi_\Delta(r)$ gives an account of changes only. The fact that repairs contain data that persists from the original instance is captured with persistence defaults.

Definition 6
The repair program $\Pi(r)$ consists of the rules in program $\Pi_\Delta(r)$ (Definition 2) plus the following two rules for each predicate p in the original database:

4. Persistence *defaults*:

$$p'(\bar{X}) \longleftarrow p(\bar{X}); \qquad \neg p'(\bar{X}) \longleftarrow dom(\bar{X}), \; not \; p(\bar{X}). \qquad \square$$

[3] In $\Pi_\Delta(r)$ there are no defaults. In consequence, we can talk about answer sets as in Gelfond and Lifschitz (1991) instead of e-answer sets (Kowalski and Sadri, 1991).

Example 4
(example 3 continued) We have the following persistence defaults:

4. $P'(X,Y) \leftarrow P(X,Y)$; $\quad \neg P'(X,Y) \leftarrow dom(X,Y),\ not\ P(X,Y)$
   $Q'(X,Y) \leftarrow Q(X,Y)$; $\quad \neg Q'(X,Y) \leftarrow dom(X,Y),\ not\ Q(X,Y)$.
   $R'(X,Y) \leftarrow R(X,Y)$; $\quad \neg R'(X,Y) \leftarrow dom(X,Y),\ not\ R(X,Y)$.

   This means that, by default, everything from r is put into a repair r' and nothing else.

In this program rules 2 and 3 have priority over rule 4. It is possible to verify that the e-answer sets of the program are the expected database repairs: $\{P'(a,b), Q'(a,b), \underline{R'(a,b)}, P(a,b), Q(a,b), \ldots\}, \{\underline{\neg P'(a,b)}, \underline{\neg Q'(a,b)}, P(a,b), Q(a,b), \ldots\}$. The underlined literals represent the insertion of $R(a,b)$ in one repair and the deletion of of both $P(a,b)$ and $Q(a,b)$, in the other one, respectively. The original atoms remain, because there are no rules that can change them. The literals not shown explicitly in these e-answer sets are the negative literals, e.g. $\neg P'(a,a), \neg Q'(b,a)$, inherited from the original instance with the negative defaults.

Remark 1
As shown in Kowalski and Sadri (1991), the program $\Pi(r)$, which has an e-answer semantics, can be transformed into a disjunctive extended logic program with answer set semantics, by transforming the persistence defaults in Definition 6, respectively, into

4'. Persistence *rules*:

$$p'(\bar{X}) \longleftarrow p(\bar{X}),\ not\ \neg p'(\bar{X}); \qquad \neg p'(\bar{X}) \longleftarrow dom(\bar{X}),\ not\ p(\bar{X}),\ not\ p'(\bar{X}).$$

As shown in Gelfond and Lifschitz (1991), the resulting program can be further transformed into a disjunctive normal program with a stable model semantics. For the one to one correspondence between answer sets and stable models, we can interchangeably talk about (e-)answer sets and stable models.

Proposition 4
Given a database instance r over a finite domain, and a set of BICs IC, if S_M is an answer set of $\Pi_\Delta(r)$, then $S = S_M \cup \{p'(\bar{a}) \mid p(\bar{a}) \in S_M$ and $\neg p'(\bar{a}) \notin S_M\} \cup \{\neg p'(\bar{a}) \mid p(\bar{a}) \notin S_M$ and $p'(\bar{a}) \notin S_M\}$ is an answer set of $\Pi(r)$.

The following lemma says that whenever we build an answer set S with literals taken from $S(r,r')$, and r' satisfies the ICs and is already as close as possible to r, then in S we recover r' only. The condition that S is contained in $S(r,r')$ makes sure that its literals are taken from the right, maximal set of literals.

Lemma 1
Let r and r' be database instances over the same schema and domain, and IC, a set of BICs. Assume that $r' \models IC$ and the symmetric difference $\Delta(r,r')$ is a minimal element under set inclusion in the set $\{\Delta(r,r^*) \mid r^* \models IC\}$. Then, for every answer set S of $\Pi_\Delta(r)$ contained in $S(r,r')$, it holds $r' = I(S)$.

Theorem 1
If $\Pi(r)$ is the program $\Pi_\Delta(r)$ plus rules 4'., for a finite domain database instance r and a est of BICs IC, it holds:

1. For every repair r' of r wrt IC, there exists an answer set S of $\Pi(r)$ such that $r' = \{p(a) \mid p'(a) \in S\}$.
2. For every answer set S of $\Pi(r)$, there exists a repair r' of r wrt IC such that $r' = \{p(a) \mid p'(a) \in S\}$.

In the case of finite domain databases, the domain can be and has been declared. In this situation, we can handle any set of binary ICs, without caring about their safeness or domain independence (Ullman, 1988).

Example 5
Consider $D = \{a, b, c\}, IC = \{\forall x p(x)\}$ and the inconsistent instance $r =, \{p(a)\}$. $\Pi(r)$ contains the default rules $p'(X) \longleftarrow p(X), not\ \neg p'(X); \neg p'(X) \longleftarrow dom(X), not\ p(X), not\ p'(X)$; the triggering exception $p'(X) \longleftarrow dom(X), not\ p(X)$, the stabilizing exception $p'(X) \longleftarrow dom(X)$; and the facts $dom(a), dom(b), dom(c), p(a)$. The only answer set is $\{dom(a), dom(b), dom(c), p(a), p'(a), p'(b), p'(c)\}$, that corresponds to the only repair $r' = \{p(a), p(b), p(c)\}$.

The IC requires that every element in the finite domain D belongs to table p; and this can be achieved. However, with an infinite domain D, we could not obtain a finite program nor an instance with a table p containing finitely many tuples.

4.2 Infinite domain databases

Now we consider ICs that are *domain independent*, for which checking their satisfaction in an instance r can be done considering the elements of the finite active domain $Act(r)$ only (Ullman, 1988). The IC in Example 5 is not domain independent.

For domain independent BICs all previous lemmas and theorems still hold if we have an infinite domain D. To obtain them, all we need to do is to use a predicate $act_r(x)$, standing for the active domain $Act(r)$ of instance r, instead of predicate $dom(x)$. This is because, for domain independent BICs, the database domain can be considered to be $Act(r)$. Furthermore, in this case we can omit the *dom* facts and goals from $\Pi(r)$. In consequence, we have the following theorem.

Theorem 2
For a set of domain independent binary integrity constraints and a database instance r, there is a one to one correspondence between the answers sets of the repair program $\Pi(r)$ and the repairs of r.

4.3 Evaluating basic $\mathscr{K}$-queries

The specification of database repairs we have obtained provides the underpinning of a general method of evaluating a basic $\mathscr{K}$-query of the form $\beta \equiv \mathscr{K}\alpha$, where α is a basic query.

First, from α, that is expressed in terms of the database predicates in $\mathscr{L}$, we obtain a stratified logic program $\Pi(\alpha)$ (this is a standard construction (Lloyd, 1987;

Abiteboul *et al.*, 1995)) in terms of the new, primed predicates introduced in $\Pi(r)$. One of the predicate symbols, $Answer_\alpha$, of $\Pi(\alpha)$ is designated as the query answer predicate. Second, determine all the answers sets $S_1, \ldots, S_k$ of the logic program $\Pi = \Pi(\alpha) \cup \Pi(r)$. Third, compute the intersection $r_\beta = \bigcap_{1 \leqslant i \leqslant k} S_i/Answer_\alpha$, where $S_i/Answer_\alpha$ is the extension of $Answer_\alpha$ in S_i. The set of tuples r_β is the set of answers to β, or equivalently, the set of consistent answers to α, in r.

Example 6
(example 4 continued) Consider the query for the consistent answers to $\alpha_1(x)$: $(P(x, a) \vee Q(a, x))$, in the database instance. This query can be transformed into the query program $\Pi(\alpha_1)$ containing the rules $Answer_{\alpha_1}(X) \longleftarrow P'(X, a)$, and $Answer_{\alpha_1}(X) \longleftarrow Q'(a, X)$.

To obtain consistent answers it is necessary to evaluate the query goal $Answer_{\alpha_1}(X)$ wrt the program obtained by combining $\Pi(r)$, already obtained in Examples 3 and 4, and program $\Pi(\alpha_1)$. Each of the answer sets of the combined program will contain a set of ground $Answer_{\alpha_1}$-atoms. The arguments of the $Answer_{\alpha_1}$-atoms that are present simultaneously in all the answer sets will be the consistent answers to the original query.

As a second example, consider the query $\alpha_2(y) : \exists x Q(x, y)$. In order to obtain the consistent answers, we keep $\Pi(r)$ as before, but we run it in combination with the new query program $\Pi(\alpha_2) : Answer_{\alpha_2}(Y) \longleftarrow Q'(X, Y)$.

Notice that consistent answers to a query are those that can be obtained from the repair program plus the query program under the *cautious* or *skeptical* answer set semantics for the combined logic program: what is true of the program is what is true of all its answer sets. In section 5 we give computational examples.

The program $\Pi = \Pi(\alpha) \cup \Pi(r)$, where α is a first order query, is naturally split into $\Pi(\alpha)$ and $\Pi(r)$, but also split in the precise sense introduced in Lifschitz and Turner (1994) as follows: the set U of literals consisting of all the primed database literals, $(\neg)p'(\bar{t})$ plus and all the non primed database literals, $(\neg)p(\bar{t})$ appearing in $\Pi(r)$, form a *splitting set* for Π, because whenever a literal in U appears in a head of a rule in Π, all the literals in the body of that rule also appear in U. U splits Π precisely into the two expected parts, $\Pi(r)$ and $\Pi(Q)$, because the literals in U do not appear in heads of rules of $\Pi(\alpha)$ (for $\Pi(\alpha)$ the literals in U act as extensional literals).

As a consequence of this splitting, we know from Lifschitz and Turner (1994), that every answer set of Π can be represented as the union of an answer set of $\Pi(r)$ and an answer set of $\Pi(\alpha)$, where each answer set for $\Pi(r)$ acts as an extensional database for the computation of the answer sets of $\Pi(\alpha)$. Since program $\Pi(\alpha)$ is stratified, for each answer set of $\Pi(r)$, there will only one answer set for $\Pi(\alpha)$.

5 Computational examples

In this section we will assume that, according to Remark 1, the repair programs are given as extended disjunctive logic programs with answer set semantics. In consequence, we can use any implementation for that semantics. In particular, we

will give examples of the application of the *DLV* system (Eiter *et al*., 1998) to the computation of database repairs and consistent query answers.

5.1 Computing database repairs with DLV

Example 7
Consider the schema $Emp(Name, SSN)$, and the functional dependencies $Name \rightarrow SSN$, $SSN \rightarrow Name$, stating that each person should have just one SSN and different persons should have different SSNs. The following is an inconsistent instance:

Emp	*Name*	*SSN*
	Irwin Koper	677-223-112
	Irwin Koper	952-223-564
	Mike Baneman	334-454-991

The following *DLV* program corresponds to the repair program. In it, the repaired, primed version of table *Emp* is now denoted by *emp_p*:

```
% domains of the database
dom_name("Irwin Koper"). dom_name("Mike Baneman"). dom_number("677-223-112").
dom_number("952-223-564"). dom_number("334-454-991").

% initial database
emp("Irwin Koper","677-223-112"). emp("Irwin Koper","952-223-564").
emp("Mike Baneman","334-454-991").

% default rules
emp_p(X,Y)  :- emp(X,Y), not -emp_p(X,Y).
-emp_p(X,Y) :- dom_name(X), dom_number(Y), not emp(X,Y), not emp_p(X,Y).

% triggering rules
-emp_p(X,Y) v -emp_p(X,Z) :- emp(X,Y), emp(X,Z), Y!=Z.
-emp_p(Y,X) v -emp_p(Z,X) :- emp(Y,X), emp(Z,X), Y!=Z.

% stabilizing rules.
-emp_p(X,Y) :- emp_p(X,Z), dom_number(Y), Y!=Z.
-emp_p(Y,X) :- emp_p(Z,X), dom_name(Y), Y!=Z.
```

If *DLV* is asked to compute the answer sets, we obtain two of them, corresponding to the two possible repairs:

Emp	*Name*	*SSN*
	Irwin Koper	952-223-564
	Mike Baneman	334-454-991

Emp	*Name*	*SSN*
	Irwin Koper	677-223-112
	Mike Baneman	334-454-991

To pose the query $Emp(X, Y)$?, asking for the consistent tuples in table *Employee*, we add a new query rule to the program: `answer(X,Y) :- emp_p(X,Y).`
Now, the two answer sets contain `answer`-literals, namely

```
{..,answer("Irwin Koper","952-223-564"),answer("Mike Baneman","334-454-991")}
{..,answer("Irwin Koper","677-223-112"),answer("Mike Baneman","334-454-991")}
```

There is only one ground `answer`-atom in the intersection of the answer sets of the new program. Then, the only consistent answer is the tuple: `X="Mike Baneman", Y="334-454-991"`.

6 Referential integrity constraints

In this section, we show how to extend the specifications of repairs given for binary integrity constraints to Referential Integrity Constraints (RICs). This can be done via an appropriate representation of existential quantifiers as program rules.

Consider the *RIC*: $\forall \bar{x}\,(P(\bar{x}) \rightarrow \exists \bar{y}\, R(\bar{x}, \bar{y}))$, and the inconsistent database instance $r = \{P(\bar{a}), P(\bar{b}), R(\bar{b}, \bar{a})\}$. We assume that there is an underlying database domain D. The repair program has the persistence default rules

$$P'(\bar{X}) \leftarrow P(\bar{X}); \quad \neg P'(\bar{X}) \leftarrow dom(\bar{X}),\ not\ P(\bar{X});$$

$$R'(\bar{X}, \bar{Y}) \leftarrow R(\bar{X}, \bar{Y}); \quad \neg R'(\bar{X}, \bar{Y}) \leftarrow dom(\bar{X}, \bar{Y}),\ not\ R(\bar{X}, \bar{Y}).$$

In addition, it has the triggering exception rule

$$\neg P'(\bar{X}) \vee R'(\bar{X}, null) \leftarrow P(\bar{X}), not\ aux(\bar{X}), \tag{2}$$

with $aux(\bar{X}) \leftarrow R(\bar{X}, \bar{Y})$; $null \notin D$; and the stabilizing exception rules

$$\neg P'(\bar{X}) \leftarrow \neg R'(\bar{X}, null), not\ aux'(\bar{X}), \tag{3}$$

$$R'(\bar{X}, null) \leftarrow P'(\bar{X}), not\ aux'(\bar{X}); \tag{4}$$

with $aux'(\bar{X}) \leftarrow R'(\bar{X}, \bar{Y})$.

The variables in this program range over D, that is, they do not take the value *null*. This is the reason for the first literal in clause (3). The last literal in clause (4) 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 answer sets are the expected ones, namely delete $P(\bar{a})$ or insert $R(\bar{a}, null)$.

It would be natural to include here the functional dependency $\bar{X} \rightarrow \bar{Y}$ on R, expressing that $\bar{X}$ is a primary key in R and a foreign key in P. This can be done without problems, actually the two constraints would not interact, that is, repairing one of them will not cause violations of the other one.

Finally, if only elimination of tuples were considered admissible changes, but not introduction of null values, then the triggering exception (2) would have to be changed into $\neg P'(\bar{X}) \leftarrow P(\bar{X}), not\ aux(\bar{X})$.

[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. Alternatively, conditions of the form $X \neq \text{null}$ can be placed in the bodies.

6.1 Referential ICs and strong constraints

It is possible to use *DLV* to impose preferences on repairs via an appropriate representation of constraints. For RICs, for example, preference for introduction of null values or for a cascade policy can be captured.

Example 8
(Example 7 continued.) Consider the same schema and *FD*s as before, but now we have the following instance:

Emp	*Name*	*SSN*
	Irwin oper	677-223-112
	Irwin Koper	952-223-564
	Mike Baneman	952-223-564

The *DLV* repair program is as in Example 7, but with the facts:

```
dom_number("677-223-112").  dom_number("952-223-564").
emp("Irwin Koper","677-223-112"). emp("Irwin Koper","952-223-564").
emp("Mike Baneman","952-223-564").
```

If *DLV* is run with this program as input, we obtain two answer sets:

```
{..,emp_p("Irwin Koper","677-223-112"),-emp_p("Irwin Koper","952-223-564"),
emp_p("Mike Baneman","952-223-564"),-emp_p("Mike Baneman","677-223-112")}
{..,-emp_p("Irwin Koper","677-223-112"),emp_p("Irwin Koper","952-223-564"),
-emp_p("Mike Baneman","952-223-564"),-emp_p("Mike Baneman","677-223-112")}
```

corresponding to the database repairs:

Emp	*Name*	*SSN*
	Irwin Koper	677-223-112
	Mike Baneman	952-223-564

Emp	*Name*	*SSN*
	Irwin Koper	952-223-564

Adding the query rule `answer(X):- emp_p(X,Y).`, we can ask for those persons who have a SSN. Two answer sets are obtained:

```
{..,answer("Irwin Koper"),answer("Mike Baneman")},{..,answer("Irwin Koper")}
```

From them, we can -consistently- say that only Irwin Koper has a SSN.

Let us now extend the schema with a unary table *Person*(*Name*), whose contents, together with the original contents of table *Emp*, is

Person	*Name*
	Irwin Koper
	Mike Baneman

If we want every person to have a SSN, we may impose the RIC $\forall\, x(Person(x) \rightarrow \exists\, yEmp(x, y))$, stating that every person must have a SSN, that we saw how to repair at the beginning of this section, either by introducing null values or by cascading deletions.

We may not want any of these two options (we do not want null values in the key *SSN*) or we do not want to delete any employees (in this case, M. Baneman from *Person*). An alternative is to use *DLV*'s possibility of specifying *strong constraints*, that have the effect of pruning those answer sets that do not satisfy them. This can be done in *DLV* by introducing the denial `:- dom_name(X), not has_ssn(X).`, with `has_ssn(X) :- emp_p(X,Y).`. The answer sets of the original program that do not satisfy the ICs are filtered out; and now, only one repair is obtained:

```
{..,emp_p("Irwin Koper","677-223-112"),-emp_p("Irwin Koper","952-223-564"),
emp_p("Mike Baneman","952-223-564"),-emp_p("Mike Baneman","677-223-112"),
has_ssn("Irwin Koper"), has_ssn("Mike Baneman"), answer("Irwin Koper"),
answer("Mike Baneman")}
```

In it, every person has a SSN (according to the `has_ssa` predicate). As expected, the answers to the original query are `X="Irwin Koper"` and `X="Mike Baneman"`. Notice that strong constraints differ from the database integrity constraints in that they are not used in the generation of repairs, but only at a final step where some repairs are discarded. Furthermore, strong constraints are constraints on the answer sets, but not directly on the semantics of the database.

7 Well-founded consistent answers

The intersection of all answer sets of a extended disjunctive logic program contains the *well-founded interpretation* for such programs (Leone *et al*., 1997), which can be computed in polynomial time in the size of the ground program. This interpretation may be partial and not necessarily a model of the program. Actually, it is a total interpretation if and only if it is the only answer set.

In Leone *et al*. (1997) it is shown how to compute the answer sets of a program starting from the well-founded interpretation. This is what *DLV* basically does, but instead of starting from the well-founded interpretation, it starts from the also efficiently computable set of *deterministic consequences* of the program, that is still contained in the intersection of all answer sets, and in its turn, contains the well-founded interpretation (Calimieri *et al*., 2002). Actually, *DLV* can be explicitly asked to return the set of deterministic consequences of the program,[5] and it can be also used as an approximation from below to the intersection of all answer sets.

On the other side, in the general case, computing the stable model semantics for disjunctive programs is Π_2^p-complete in the size of the ground program.[6]

The well-founded interpretation, $W_{\Pi(r)} = \langle W^+, W^-, W^u \rangle$, of program $\Pi(r)$, where W^+, W^-, W^u are the sets of true positive, true negative, and unknown literals, resp.,

[5] By means of its option `-det`.

[6] See Dantsin *et al*. (2001) for a review of complexity results in logic programming.

is given by the least fixpoint $\mathscr{W}^{\omega}_{\Pi(r)}(\emptyset)$ of operator $\mathscr{W}_{\Pi(r)}$, that maps interpretations to interpretations (Leone *et al.*, 1997). More precisely, assuming that we have the ground instantiation of the repair program $\Pi(r)$, $\mathscr{W}_{\Pi(r)}(I)$ is defined on interpretations I that are sets of ground literals (without pairs of complementary literals) by: $\mathscr{W}_{\Pi(r)}(I) := T_{\Pi(r)}(I) \cup \neg.GUS_{\Pi(r)}(I)$.

Intuitively, $T_{\Pi(r)}$ is the immediate consequence operator that declares a literal true whenever there is ground rule containing it in the head, the body is true in I and the other literals in the (disjunctive) head are false in I. $\neg.GUS_{\Pi(r)}(I)$ denotes the set of complements of the literals in $GUS_{\Pi(r)}(I)$, being the latter the largest set of unfounded literals, those that definitely cannot be derived from the program and the set I of assumptions; in consequence their complements are declared true.

The intersection of all answer sets of $\Pi(r)$ is

$$Core(\Pi(r)) := \bigcap \{S \mid S \text{ is an answer set of } \Pi(r)\}.$$

Interpretation $W_{\Pi(r)}$, being a subset of $Core(\Pi(r))$, can be used as an approximation from below to the core, but can be computed more efficiently than all database repairs, or their intersection, in the general case. However, it is possible to identify classes of ICs for which $W_{\Pi(r)}$ coincides with $Core(\Pi(r))$.

Proposition 5
For a database instance r, and a set of ICs containing functional dependencies and unary ICs only, the $Core(\Pi(r))$ of program $\Pi(r)$ coincides with the set of true ground literals in $W_{\Pi(r)}$, the well-founded interpretation of program $\Pi(r)$.

Results like the previous one can be established using the repair programs introduced in section 4.1, for finite database domains D. Then, the results are known to still hold for infinite domain databases, but domain independent integrity constraints, like the ones in Proposition 5.

As corollary of Proposition 5, we obtain that, for FDs and unary constraints, $Core(\Pi(r))$ can be computed in polynomial time in the size of the ground instantiation of $\Pi(r)$, a result first established in Arenas *et al.* (2001) for FDs. The core alone can be used to consistently answer non-existential conjunctive queries. Furthermore, in Arenas *et al.* (2001), for the case of functional dependencies, conditions on queries are identified under which one can take advantage of computations on the core to answer aggregate queries more efficiently.

As the following example shows, for other BICs, the core of the repair program may not coincide with the well-founded interpretation.

Example 9
Consider the BICs $IC = \{q \vee r,\ s \vee \neg q,\ s \vee \neg r\}$ and the empty database instance. The program $\Pi(r)$ contains

Triggering rules: $q' \vee r' \longleftarrow not\ q, not\ r; \quad s' \vee \neg q' \longleftarrow q, not\ s; \quad s' \vee \neg r' \longleftarrow r, not\ s.$

Stabilizing rules: $q' \longleftarrow \neg r'; \quad r' \longleftarrow \neg q'; \quad s' \longleftarrow q'; \quad \neg q' \longleftarrow \neg s;$
$s' \longleftarrow r'; \quad \neg r' \longleftarrow \neg s'.$

Persistence rules: $q' \longleftarrow q, not\ \neg q'; \quad s' \longleftarrow s, not \neg s'; \quad r' \longleftarrow r, not\ \neg r';$
$\neg q' \longleftarrow not\ q, not\ q'; \quad \neg s' \longleftarrow not\ s, not\ s'; \quad \neg r' \longleftarrow not\ r, not\ r'.$

The answer sets are: $\{q', s', \neg r'\}$ and $\{\neg q', s', r'\}$. Then $Core(\Pi(r)) = \{s'\}$, but for $W_{\Pi(r)}$, one has $W^+ \cup W^- = \emptyset$.

The results obtained so far in this section apply to the repair program $\Pi(r)$. Nevertheless, when we add an arbitrary query program $\Pi(\alpha)$ to $\Pi(r)$, then it is possible that the new core properly extends the well-founded interpretation of the extended program, even for FDs.

Example 10
Consider $r = \{P(a,b), P(a,c)\}$, with the FD, $P(X,Y)$: $X \rightarrow Y$, and the query $\alpha(x)$: $\exists y\ P(x,y)$. The combined Π program is:

$dom(a).\quad dom(b).\quad dom(c).\quad P(a,b).\quad P(a,c).$
$Answer(X) \leftarrow P'(X,Y)$
$P'(X,Y) \leftarrow P(X,Y), not\ \neg P'(X,Y).$
$\neg P'(X,Y) \leftarrow dom(X), dom(Y), not\ P(X,Y), not\ P'(X,Y).$
$\neg P'(X,Y) \vee \neg P'(X,Z) \leftarrow P(X,Y), P(X,Z), Y \neq Z.$
$\neg P'(X,Y) \leftarrow dom(Y), P'(X,Z), Y \neq Z.$

The answer sets are $S_1 = \{Answer(a), P'(a,b), P(a,b), P(a,c), \ldots\}$ and $S_2 = \{Answer(a), P'(a,c), P(a,b), P(a,c), \ldots\}$. The well-founded interpretation is $W_\Pi = \langle W^+, W^-, W^u\rangle$, with $W^+ = \{P(a,b), P(a,c), dom(a), \ldots\}$, $W^- = \{\neg P'(a,a), \ldots\}$, and $W^u = \{P'(a,b), P'(a,c), Answer(a)\}$. In particular, $Answer(a) \in Core(\Pi)$, but $Answer(a) \notin W^+$.

We know, by complexity results presented in Arenas *et al.* (2001) for functional dependencies that, unless $P = NP$, consistent answers to first-order queries cannot be computed in polynomial time. In consequence, we cannot expect to compute $Core(\Pi)$ of the program that includes the query program by means of the well-founded interpretation of Π alone.

8 An alternative semantics

As discussed in Arenas *et al.* (1999), our database repairs can be obtained as the revision models corresponding to the "possible model approach" introduced in Winslett (1988) and Chou and Winslett (1994) in the context of belief update. When the database instance (a model) is updated by the set of ICs, a new set of models is generated, the database repairs. Winslett's revision models, as our repairs, are based on minimal *set* of changes wrt the original model.

8.1 Cardinality-based repairs and weak constraints

In Dalal (1988), again in the context of belief revision/update, an alternative notion of revision model based on minimal *number* of changes is introduced.

Definition 7
Given a database instance r, an instance r' is a *Dalal repair* of r wrt IC iff $r' \models IC$ and $|\Delta(r,r')|$ is a minimal element of $\{|\Delta(r,r^*)| \mid r^* \models IC\}$. □

We could give a definition of *Dalal consistent answer* exactly in the terms of Definition 1, but replacing "repair" by "Dalal repair". We can also specify Dalal repairs using the same repair programs we had in section 4, but with the persistence defaults replaced by *weak constraints* (Buccafurri *et al*., 2000). The latter will not be imposed on the original database, but rather on the answer sets of the change program, $\Pi_\Delta(r)$, that is responsible for the changes, and was introduced in section 4.1.1.

Weak constraints are of the form $\Leftarrow L_1, \ldots, L_k, not\ L_{k+1}, \ldots, not\ L_n$, where the L_i's are literals. These constraints are added to an extended disjunctive program, with the effect that only those answer sets that minimize the *number* of violated ground instantiations of the weak constraints are kept.

In order to capture Dalal repairs, we need very simple weak constraint. The program $\Pi^D(r)$ that specifies the Dalal repairs of a database instance r wrt a set of BICs consists of program $\Pi_\Delta(r)$ of section 4.1.1 (rules 1–3) plus

4". For every database predicate p, the weak constraints

$$\begin{aligned} &\Leftarrow \quad p'(\bar{X}),\ not\ p(\bar{X}), \\ &\Leftarrow \quad \neg p'(\bar{X}),\ p(\bar{X}). \end{aligned} \tag{5}$$

These constraints say that the original database and a repair are expected to coincide. Since they are weak constraints, they allow violations, but only a minimum *n*umber of tuples that belong to the repair and not to the original instance, or the other way around, are be accepted.

The results for the change program $\Pi_\Delta(r)$ still hold here. In consequence, the program $\Pi^D(r)$ will have answers sets that correspond to repairs that are minimal both under set inclusion and number of changes, i.e. only answer sets corresponding to Dalal repairs.

Example 11
Let $D = \{a\}$, $r = \{p(a)\}$ and $IC = \{\neg p(x) \vee q(x), \neg q(x) \vee r(x)\}$. $\Pi^D(r)$ contains

Facts: $dom(a)$. $p(a)$.

Triggering exceptions: $\neg p'(X) \vee q'(X) \longleftarrow p(X),\ not\ q(X)$
$\neg q'(X) \vee r'(X) \longleftarrow q(X),\ not\ r(X)$

Stabilizing exceptions: $q'(X) \longleftarrow p'(X)$; $\neg p'(X) \longleftarrow \neg q'(X)$
$r'(X) \longleftarrow q'(X)$; $\neg q'(X) \longleftarrow \neg r'(X)$

Weak constraints: $\Leftarrow p'(X),\ not\ p(X)$; $\Leftarrow q'(X),\ not\ q(X)$; $\Leftarrow r'(X),\ not\ r(X)$;
$\Leftarrow \neg p'(X), p(X)$; $\Leftarrow \neg q'(X), q(X)$; $\Leftarrow \neg r'(X), r(X)$.

Weak constraints are implemented in *DLV*,[7] that run on this program returns the answer set $\{dom(a), p(a), \neg p'(a)\}$, corresponding to the empty database repair, but not the other Winslett's repair $\{dom(a), p(a), q'(a), r'(a)\}$, whose set of changes wrt r has two elements, whereas the first repair differs from r by one change only.

[7] They are specified by $:\sim$ *Conj*., where *Conj* is a conjunction of (possibly negated) literals. See *DLV*'s user manual in http://www.dbai.tuwien.ac.at/proj/dlv/man.

Notice that in Example 11, from the change program $\Pi_\Delta(r)$, without the weak constraints, we obtain the eventually discarded answer set $\{dom(a), p(a), q'(a), r'(a)\}$, that only implicitly contains $p'(a)$. The reason is that now we do not have the persistence rules that cause $p(a)$ to persist in the database as $p'(a)$. In consequence, we have to interpret these answer sets to establish the correspondence between them and the repairs. This is done in Theorem 3 via the interpretation I of Definition 4. In consequence, for BICs and finite domain databases we have

Theorem 3
Given a (finite domain) database instance r and a set of BICs IC:

1. For every Dalal repair r' of r wrt IC, there exists an answer set S of $\Pi^D(r)$ such that $I(S) = r'$.
2. For every answer set S of $\Pi^D(r)$, there exists a Dalal repair r' of r wrt IC such that $I(S) = r'$.

As with Winslett's repairs, the theorem still holds for infinite domain databases when the BICs are domain independent.

Instead of interpreting the answer sets due to the only implicit presence of primed literals caused by the lack of persistence defaults, when we pose queries expecting consistent answers, we may transform the original query according to the following table:

original query	query in the program
$p(\bar{x})$	$\mathtt{query_p}(\bar{\mathtt{X}}) \leftarrow \mathtt{p'}(\bar{\mathtt{X}})$. $\mathtt{query_p}(\bar{\mathtt{X}}) \leftarrow \mathtt{p}(\bar{\mathtt{X}}),\ \mathtt{not}\ \neg\mathtt{p'}(\bar{\mathtt{X}})$.
$\neg p(\bar{x})$	$\mathtt{query_{\neg p}}(\bar{\mathtt{X}}) \leftarrow \neg\mathtt{p'}(\bar{\mathtt{X}})$. $\mathtt{query_{\neg p}}(\bar{\mathtt{X}}) \leftarrow dom(\bar{\mathtt{X}}),\ \mathtt{not}\ \mathtt{p}(\bar{\mathtt{X}}),\ \mathtt{not}\ \mathtt{p'}(\bar{\mathtt{X}})$.

That is, everywhere in the original query, we replace p and $\neg p$ by $query_p$, and $query_{\neg p}$, respectively, and we add the rules on the right-hand side of the table.

Finally, as an alternative, we could avoid interpreting answer sets or transforming queries, and explicitly obtain the Dalal repairs, by imposing the weak constraints on the repair program $\Pi(r)$, that contains the default rules.

9 Conclusions

We have presented a general methodology to consistently answer first order queries posed to relational databases that violate given ICs. We have restricted ourselves mainly to the case of binary integrity contraints, i.e. universal ICs containing at most two database literals. However the methodology can be extended to universal ICs with a larger number of database literals. Facts, persistence and triggering exceptions rules are as before, but the number of stabilizing rules grows according to the number of subsets of database literals in each IC. We sketch the solution by means of an example.

Example 12
Consider $r = \{P(a), Q(a), R(a)\}$ and the ternary integrity constraints $IC = \{\neg P(x) \vee \neg Q(x) \vee R(x), \neg P(x) \vee \neg Q(x) \vee \neg R(x), \neg P(x) \vee Q(x) \vee \neg R(x), P(x) \vee \neg Q(x) \vee \neg R(x), \neg P(x) \vee Q(x) \vee R(x), P(x) \vee \neg Q(x) \vee R(x), P(x) \vee Q(x) \vee \neg R(x)\}$. The repair program $\Pi(r)$ contains the usual persistence defaults for P, Q, R, and triggering exception rules, e.g. for the first IC in IC:

$$\neg P'(x) \vee \neg Q'(x) \vee R'(x) \quad \leftarrow \quad P(x), Q(x), not\ R(x).$$

We also need the stabilizing rules, e.g. for the first IC

$$\begin{aligned} \neg P'(x) \vee \neg Q'(x) &\leftarrow \neg R'(x), \\ \neg P'(x) \vee R'(x) &\leftarrow Q'(x), \\ \neg Q'(x) \vee R'(x) &\leftarrow P'(x); \end{aligned} \tag{6}$$

but also for the first IC:

$$\begin{aligned} \neg P'(x) &\leftarrow Q'(x), \neg R'(x), \\ R'(x) &\leftarrow P'(x), Q'(x), \\ \neg Q'(x) &\leftarrow P'(x), \neg R'(x). \end{aligned} \tag{7}$$

In this case we obtain as answer set the only repair, namely the empty instance, represented by $\{P(a), Q(a), R(a), \neg P'(a), \neg Q'(a), \neg R'(a)\}$. Using rules (7) as the only stabilizing rules, without using the disjunctive stabilizing rules (6), the empty repair cannot be obtained.

Extending the current methodology to relational databases with view definitions should be straightforward.

9.1 Ongoing and future work

There are several open issues that deserve further investigation, among them: (a) Analyze conditions under which simpler and optimized programs can be obtained; (b) a more detailed treatment of referential ICs (and other existential ICs); (c) identification of other classes of ICs for which the well-founded interpretation and the intersection of all database repairs coincide; and (d) representation of preferences for certain kinds of repair actions. In principle, the preferences could be captured by choosing the right disjuncts in the triggering rules.

The approach to CQA is based on the specification of all repairs, where each of them completely restores the consistency of the database, independently from the query that is posed and from the fact that it might have nothing to do with some of the violated ICs. This approach work well if the repairs are stored and different queries are posed after that. However, it would be useful to specify and compute "repairs" that partially restore the consistency of the database, only wrt the ICs that are relevant to the query. Possibly appropriate grounding techniques could be used in this case.

The repair programs we presented materialize the closed-world assumption by explicitly producing the negative primed literals. This is due to the persistence default

rules. In practical applications this should and could be avoided by restoring, via the program, the *implicit* closed world assumption applied to the repairs.

We have not addressed the problem of obtaining query answers to general $\mathscr{K}$-queries. The method we presented for basic $\mathscr{K}$-queries needs to be combined with some method of evaluating first-order queries. For example, safe-range first-order queries (Abiteboul *et al*., 1995) can be translated to relational algebra. The same approach can be used for $\mathscr{K}$ queries with the subqueries of the form $\mathscr{K}\alpha$ 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.

There are several interesting open issues related to computational implementation of the methodology we have presented.

The existing implementations of stable models semantics are based on grounding the rules, what, in database applications, may lead to huge ground programs. Some "intelligent" grounding techniques have been implemented in *DLV*. Furthermore, those implementations are geared to computing stable models, possibly only one or some of them, whereas consistent query answering requires, at least implicitly, having all stable models, or the "relevant parts" of all of them. In particular, this opens the interesting issue of having the construction of (the relevant parts of) the stable models guided by the query, because query answering is our primary goal, but not the computation of repairs. Current query evaluation methodologies under the stable model semantics, specially for disjunctive programs, are completely insensitive to the query at hand. The goal is to avoid irrelevant computations.

In database applications, posing and answering open queries (with variables) is more natural and common that answering ground queries. However, existing implementations of stable model semantics are better designed to do the latter.

It would be useful to implement a consistent query answering system based on the interaction of our repairs logic programs with relational DBMS. For this purpose, some functionalities and front-ends included in *DLV*'s architecture (Eiter *et al*., 2000) could be used. Trying to push most of the computation to the DBMS seems to be the right way to proceed.

9.2 Related work

Work on inconsistency handling has been done for long time and by different communities, e.g. philosophical and non-classical logic, knowledge representation, logic programming, databases, software specification, etc. We mention only some related work that has, or may have, some relation to our notions of repair and consistent answer, or are based on some form of logic programming.

There are several similarities between our approach to consistency handling and those followed by the belief revision/update community. As already mentioned in section 8, database repairs coincide with the revised models defined in Winslett (1988). The treatment there is mainly propositional, but a preliminary extension to first order knowledge bases can be found in Chou and Winslett (1994). Those papers concentrate on the computation of the models of the revised theory, i.e. the repairs

in our case, but not on query answering. The revision of a database instance by the ICs produces new database instances, the repairs of the original database.

Nevertheless, our motivation and starting point are quite different from those of belief revision. We are not interested in computing the repairs *per se*, but in answering queries, hopefully using the original database as much as possible. If this is not possible, we look for methodologies, as our logic programming approach, for representing and querying simultaneously and implicitly all the repairs of the database.

Bry (1997) 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 (thus Bry's notion of consistent query answer is weaker than ours). Moreover, Bry's approach is entirely proof-theoretic and does not provide a computational mechanism to obtain consistent answers to first-order queries.

Several papers studied the problem of making inferences from a possibly inconsistent, propositional or first-order, knowledge base. The basic idea is to infer the classical consequences of all maximal consistent subsets of the knowledge base (Lozinskii, 1994; Baral *et al*., 1991), or all *most consistent* models of the knowledge base (Kifer and Lozinskii, 1992; Arieli and avron, 1999) (where the order on models is defined on the base of atom annotations drawing values from a lattice or a bi-lattice). This provides a non-monotonic consequence relation but the special role of the integrity constraints (whose truth cannot be given up) is not captured. Also, the issue of processing general first-order queries is not considered.

Now we briefly review specification and logic programming based approaches to consistency handling in databases. In this direction, the closest approach to ours was presented, independently, in Greco *et al*. (2001) (see also Greco and Zumpano, 2000, 2001). There, disjunctive programs are used to specify the minimal sets of changes, under set inclusion, that lead to database repairs in the sense of Arenas *et al*., (1999). The authors present a compact schema for generating repair programs for general universal integrity constraints. The application of such a schema leads to programs that involve essentially all possible disjunctions of database literals in the heads, ending up with programs like the one in Example 12. They concentrate mainly on producing the set of changes, rather than the repaired databases explicitly. In particular, they do no have persistence rules in the program. In consequence, the program cannot be used directly to obtain consistent answers. An interpretation of the results, possibly like the one introduced in section 8 would be necessary. They also introduce "repair constraints" to specify preferences for certain kinds of repairs.

The annotated predicate logic introduced in Kiffer and Lozinskii (1992) was applied in Arenas *et al*. (2000) to the task of computing consistent query answers via a specification of the database repairs. The specification was used to derive algorithms for consistently answering some restricted forms of first order queries and to obtain some complexity results. As expected, the database repairs correspond to certain minimal models of the specification. This approach is based on a non-classical logic, and computing consistent answers from it is not straightforward. The

specification methodology was extended from universal ICs to referential ICs in Barcelo and Bertossi (2002).

There are several proposals for language constructs extending stratified Datalog programs with the purpose of specifying nondeterministic queries. Essentially, the idea is to construct a maximal subset of a given relation that satisfies a given set of functional dependencies. Since there is usually more than one such subset, the approach yields nondeterministic queries in a natural way. Clearly, maximal consistent subsets, choice models in Giannotti *et al.* (1997), correspond to our repairs in the case of functional dependencies. Stratified Datalog with choice (Giannotti *et al.*, 1997) combines enforcing functional dependencies with inference using stratified Datalog programs. Answering queries in all choice models ($\forall G$-queries (Greco *et al.*, 1995)) corresponds to our notion of computation of consistent query answers for first-order queries.

The *revision programs* (Marek and Truszczynski, 1998) are logic programs for updating databases, and could be used to restore consistency, and then to compute database repairs. The rules in those programs allow explicitly declaring how to enforce the satisfaction of an integrity constraint, rather than explicitly stating the ICs, e.g. $in(a) \leftarrow in(a_1), \ldots, in(a_k), out(b_1), \ldots, out(b_m)$ has the intended procedural meaning of inserting the database atom a whenever $a_1, \ldots, a_k$ are in the database, but not $b_1, \ldots, b_m$. They also give a declarative, stable model semantics to revision programs. Preferences for certain kinds of repair actions can be captured by declaring the corresponding rules in program and omitting rules that could lead to other forms of repairs. Revision programs could be used, as the programs in Greco *et al.* (2001), to obtain consistent answers, but not directly, because they give an account of changes only.

Blair and Subrahmanian (1989) introduced *paraconsistent logic programs*. They have a non-classical semantics, inspired by paraconsistent first-order semantics. In Kifer and Subrahmanian (1992), general annotated logic programs are presented. Their lattice-based semantics is also non-classical. Atoms in clauses have annotations, as in Kifer and Lozinskii (1992), but now annotations may also contain variables and functions, providing a stronger representation formalism. Implementation of annotated logic programs and query answering mechanisms are discussed in Leach and Lu (1996). In Subrahmanian (1994), annotated programs are further generalized, in order to be used for amalgamating databases, resolving potential conflicts between integrated data. For this purpose the product of the lattices underlying each database is constructed as the semantic basis for the integrated database. Conflict resolutions and preferences are captured by means of function-based annotations. Other approaches to paraconsistent logic programming are discussed in Damasio and Moniz-Pereira (1998).

In Barcelo and Bertossi (2002, 2003), starting from the lattice and specification introduced in Arenas *et al.* (2000b), logic programs containing annotations as arguments (as opposed to annotated programs that contain annotated atoms) were used to specify database repairs and compute consistent answers to queries. The approach works for general universal ICs and referential ICs. The logic programs have stable model semantics. The cost of using annotations as extra arguments in

the program is balanced by the fact that the program contains only a linear number of rules, what is not the case if the number of literals per IC grows beyond two (see Example 12). In consequence, for ICs that are non binary, the approach in Barcelo and Bertossi (2003) should be more convenient.

Acknowledgments

Work supported by FONDECYT Grant 1000593, NSF Grant INT-9901877/ CONICYT Grant 1998-02-083, NSF Grant IIS-0119186, Carleton University Start-Up Grant 9364-01, NSERC Grant 250279-02. L. Bertossi holds a Faculty Fellowship of the Center for Advanced Studies, IBM Toronto Lab. We are grateful to Francisco Orchard for informative paper presentations, discussions, and experiments with *DLV*. We are grateful to Nicola Leone for kindly answering all our questions about *DLV*. We appreciate comments received from anonymous reviewers, that have helped us to substantially improve the presentation.

Appendix: Proofs

Proof of Proposition 1

Consider an arbitrary element in IC. Assume that this element is of the form $p(\bar{x}) \vee \neg q(\bar{y}) \vee \varphi$ (the proof is analogous for binary constraints containing either two positive literals or two negative literals). We have to prove that $I(S)$ satisfies any instantiation of this formula, say $p(\bar{a}) \vee \neg q(\bar{b}) \vee \varphi$. We consider two cases.

(I) If r does not satisfy this ground constraint, then S satisfies the body of the ground triggering rule: $p'(\bar{a}) \vee \neg q'(\bar{b}) \leftarrow dom(\bar{a}),\ not\ p(\bar{a}),\ q(\bar{b}), \overline{\varphi}$. Thus, $p'(\bar{a}) \in S$ or $\neg q'(\bar{b}) \in S$. If $p'(\bar{a}) \in S$, then $I(S) \models p(\bar{a})$, and if $\neg q'(\bar{b}) \in S$, then $q'(\bar{b}) \notin S$ and, therefore, $I(S) \models \neg q(\bar{b})$. In any case, $I(S) \models p(\bar{a}) \vee \neg q(\bar{b}) \vee \varphi$.

(II) If r satisfies the ground constraint, then r satisfies φ, $p(\bar{a})$ or $\neg q(\bar{b})$. In the first case, $I(S) \models \varphi$ and, therefore, $I(S) \models p(\bar{a}) \vee \neg q(\bar{b}) \vee \varphi$. Thus, assume that $r \not\models \varphi$ and $r \models p(\bar{a})$ or $r \models \neg q(\bar{b})$.
By contradiction, assume that $I(S) \not\models p(\bar{a}) \vee \neg q(\bar{b})$. If $r \models p(\bar{a})$, then $p(\bar{a}) \in S$ and, therefore, $\neg p'(\bar{a}) \in S$, by definition of $I(S)$. But in this case S satisfies the body of the ground stabilizing rule: $\neg q'(\bar{b}) \leftarrow dom(\bar{b}), \neg p'(\bar{a}), \overline{\varphi}.$ and, therefore, $\neg q'(\bar{b}) \in S$. We conclude that $I(S) \models \neg q(\bar{b})$, a contradiction. If $r \models \neg q(\bar{b})$, then $\neg q(\bar{b}) \notin S$ and, therefore, $q'(\bar{b}) \in S$, by definition of $I(S)$. But in this case S satisfies the body of the ground stabilizing rule: $p'(\bar{a}) \leftarrow dom(\bar{a}), q'(\bar{b}), \overline{\varphi}.$ and, hence, $p'(\bar{a}) \in S$. We conclude that $I(S) \models p(\bar{a})$, again a contradiction.

Proof of Proposition 2

To prove that $S(r, r')$ satisfies $\Pi_\Delta(r)$, we need to consider only the four different types of ground stabilizing rules (the satisfaction of the other rules follows from the fact that r' satisfies IC).

If $S(r,r')$ satisfies the body of the rule $q'(\bar{b}) \leftarrow dom(\bar{b}),\ p'(\bar{a}),\ \overline{\varphi}$, then r' must satisfy $dom(\bar{b})$, $p(\bar{a})$ and $\neg\varphi$. But $r' \models q(\bar{b}) \vee \neg p(\bar{a}) \vee \varphi$, since $\forall\bar{x}\forall\bar{y}(q(\bar{x}) \vee \neg p(\bar{y}) \vee \varphi) \in IC$, and, therefore, $r' \models q(\bar{b})$. Thus, $q'(\bar{b}) \in S(r,r')$.

Analogously, it is possible to prove that $S(r,r')$ satisfies the remaining types of ground stabilizing rules.

Proof of Proposition 3

From the previous proposition, we know that the change program has models; so it is a consistent program. If the program has a consistent (i.e. non trivial) answer set, they are all consistent (Lifschitz and Turner, 1994). Now we show how to obtain such consistent answer sets. The program can be split into two subprograms (Lifschitz and Turner, 1994). The first one contains the domain and database facts plus the rules $p^\star(\bar{X}) \leftarrow not\ p(\bar{X})$. The second one containing the stabilizing rules and the triggering rules modified by replacing the literals of the form *not p* in the bodies by by $p^\star$.

The first program is stratified and has one (consistent) answer set. The second subprogram does not contain weak negation, it is a positive program in that sense, and its minimal models coincide with its answer sets. By a result in Lifschitz and Turner (1994), the original program has as answer sets the unions of the answer sets of the first program and the answer sets of the second one, where the atoms $p^\star$ are treated as extensional database predicates for the computation of the answer sets of the second subprogram.

Proof of Proposition 4

Let S'_M be the set added to S_M. It is easy to verify that ${}^S\Pi(r) \trianglerighteq^{S_M} \Pi_\Delta(r)$. Then, since S_M is an answer set of $\Pi_\Delta(r)$, to prove that S is an answer set of $\Pi(r)$, it suffices to prove (I) and (II) below.

(I) $S'_M \subseteq \cap\alpha({}^S\Pi(r))$. Let $l(\bar{a})$ be an element of S'_M. If $l(\bar{a}) = p'(\bar{a})$, then $p(\bar{a}) \in S_M$ and $\neg p'(\bar{a}) \notin S_M$, and, therefore, $p(\bar{a})$ and $p'(\bar{a}) \leftarrow p(\bar{a})$ are rules in ${}^S\Pi(r)$. Thus, $p'(\bar{a})$ is in $\cap\alpha({}^S\Pi(r))$. If $l(\bar{a}) = \neg p'(\bar{a})$, then $p(\bar{a}) \notin S_M$ and $p'(\bar{a}) \notin S_M$, and, therefore, $\neg p'(\bar{a}) \leftarrow dom(\bar{a})$ is a reduced ground persistence rule in ${}^S\Pi(r)$. Thus, $\neg p'(\bar{a})$ is in $\cap\alpha({}^S\Pi(r))$.

(II) From S'_M is not possible to deduce an element that is not in S by using the stabilizing rules.
Assume that $q'(\bar{Y}) \leftarrow dom(\bar{Y}), p'(\bar{X}), \overline{\varphi}$ is a rule in $\Pi_\Delta(r)$, and $q'(\bar{b}) \leftarrow dom(\bar{b})$, $p'(\bar{a})$ is a rule in ${}^S\Pi(r)$. If $p'(\bar{a}) \in S'_M$, we need to show that $q'(\bar{b}) \in S$. By contradiction, suppose that $q'(\bar{b}) \notin S$. Then $q'(\bar{b}) \notin S_M$ and $q'(\bar{b}) \notin S'_M$, and, therefore, $q(\bar{b}) \notin S_M$ or $\neg q'(\bar{b}) \in S_M$, by definition of S'_M. If $q(\bar{b})$ is not in S_M, then given that $p'(\bar{a}) \in S'_M$, S_M satisfies the body of the rule: $q'(\bar{b}) \vee \neg p'(\bar{a}) \leftarrow dom(\bar{b}), p(\bar{a}),\ not\ q(\bar{b}), \overline{\varphi}$. But, this implies that $q'(\bar{b}) \in S_M$, a contradiction, or $\neg p'(\bar{a}) \in S_M$, also a contradiction (since $p'(\bar{a}) \in S'_M$). Otherwise, if $\neg q'(\bar{b}) \in S_M$, then by using the rule $\neg p'(\bar{a}) \leftarrow dom(\bar{a}), \neg q'(\bar{b})$, we can conclude that $\neg p'(\bar{a})$ is in S_M, a contradiction.

Analogously, it is possible to prove the same property for any other type of stabilizing rule.

Proof of Lemma 1
Let S be a answer set of $\Pi_\Delta(r)$ such that S is a subset of $S(r, r')$. First, we prove that $\Delta(r, I(S)) \subseteq \Delta(r, r')$. If $p(\bar{a}) \in \Delta(r, I(S))$, then one of the following cases holds.

(I) $r \models p(\bar{a})$ and $I(S) \not\models p(\bar{a})$. In this case, $p(\bar{a}) \in S$ and $p'(\bar{a}) \notin S$. Thus, by definition of $I(S)$ we conclude that $\neg p'(\bar{a}) \in S$ and, therefore, $\neg p'(\bar{a}) \in S(r, r')$. But this implies that $r' \not\models p(\bar{a})$. Thus, $p(\bar{a}) \in \Delta(r, r')$.

(II) $r \not\models p(\bar{a})$ and $I(S) \models p(\bar{a})$. In this case, $p(\bar{a}) \notin S$ (S is a minimal model and $p(a)$ does not need to be in S if it was not in r). Thus, by definition of $I(S)$ we conclude that $p'(\bar{a}) \in S$ and, therefore, $p'(\bar{a}) \in S(r, r')$. But this implies that $r' \models p(\bar{a})$. Thus, $p(\bar{a}) \in \Delta(r, r')$.

Hence, $\Delta(r, I(S)) \subseteq \Delta(r, r')$. But, by Proposition 1, $I(S)$ satisfies IC, and therefore, $\Delta(r, I(S))$ must be equal to $\Delta(r, r')$, since $\Delta(r, r')$ is minimal under set inclusion in $\{\Delta(r, r^*) \mid r^* \models IC\}$. Then, we conclude that $I(S) = r'$.

Proof of Theorem 1
We shall prove the first part of this theorem. The second one can be proved analogously.

Given a repair r' of r, by Lemma 1, $r' = I(S_M)$, where S_M is an answer set of $\Pi_\Delta(r)$, with $S_M \subseteq S(r, r')$. Define S from S_M as in Proposition 4. Then, S is an answer set of $\Pi(r)$. By construction of S, $I(S) = I(S_M)$. Furthermore, $I(S) = \{p(a) \mid p'(a) \in S\}$.

Proof of Proposition 5
Since it is always the case that $W_{\Pi(r)} \subseteqq Core(\Pi(r))$ (Leone *et al.*, 1997), we only need to show that $Core(\Pi(r)) \subseteqq W_{\Pi(r)}$. In consequence, it is necessary to check that whenever a literal $(\neg)p'(a)$ belongs to $Core(\Pi(r))$, where a is tuple of elements in the domain D and p is a database predicate, $(\neg)p'(a)$ can be fetched into $\mathscr{W}^n_{\Pi(r)}(\emptyset)$ for some finite integer n.

For each literal L in the original database r, and its primed version L' and each answer set S, either L' or its complement $\overline{L'} \in S$.[8] We will do the proof by cases, considering for a literal $L' : (\neg)p'(a)$ contained in $Core(\Pi(r))$ all the possible transitions from the original instance to the core: (a) negative to positive, i.e. $\neg p(a) \in r$, and $p'(a) \in Core(\Pi(r))$, (b) positive to positive. (c) negative to negative. (d) positive to negative. We will prove only the first two cases, the other two are similar. For each case, again several cases have to be verified according to the different ground program rules that could have made $p'(a)$ get into $Core(\Pi(r))$.

(a) Assume $p'(a) \in Core(\Pi(r))$, and $p(a) \notin r$. To prove: $p'(a) \in W_{\Pi(r)}$.

Since FDs can only produce deletions $p'(a)$ has to be true due to an unary constraint that was false for $p(a)$: $(p(a) \vee \varphi(a)) \in IC_D$ is false, with $\varphi(a)$ is false, where

[8] Actually only positive literals appear in r, but we are invoking the CWA. All the literals in the original instance will belong to $Core(\Pi(r))$.

IC_D is the instantiation of the ICs in the domain D. In the ground program we find the rule $p'(a) \longleftarrow dom(a), \neg\varphi(a)$. The second subgoal becomes true of $\emptyset$. Since $dom(a) \in \mathscr{W}^1_{\Pi(r)}(\emptyset)$, we obtain $p'(a) \in \mathscr{W}^2_{\Pi(r)}(\emptyset)$.

(b) Assume that $p'(a) \in Core(\Pi(r))$ and $p(a) \in r$.

This means that $p(a)$ persisted from the original instance to every answer set.

1. There is $(p(a) \vee \varphi(a)) \in IC_D$ with $\varphi(a)$ false. Then the ground program has a rule $p'(a) \longleftarrow dom(a), \neg\varphi(a)$. The body becomes true, $dom(a)$ gets into WFS after the first step, then, as in case (a), $p'(a) \in \mathscr{W}^2_{\Pi(r)}(\emptyset)$.
2. There is no ground constraint as in item 1, i.e. there is no $(p(a) \vee \varphi(a)) \in IC_D$ or the $\varphi(a)$'s are true. In this case, there is no applicable rule of the form $p'(a) \longleftarrow dom(a), \neg\varphi(a)$ in the ground program.
   Since rules associated to FDs delete tuples only, we must have obtained $p'(a)$ via a default rule $p'(a) \longleftarrow dom(a), p(a), not\ \neg p'(a)$ and the unfoundedness of $\neg p'(a)$ in the ground program. If the $\mathscr{W}_{\Pi(r)}$ operator declares $\neg p'(a)$ unfounded, then $p'(a)$ will belong to $W_{\Pi(r)}$. So, we have to concentrate on the unfoundedness of $\neg p'(a)$.
   (a) We can never get $\neg p'(a)$ from rules of the form $\neg p'(a) \longleftarrow dom(a), \neg\psi(a)$, obtained from unary ICs. If this were the case, we would have $\neg p'(a) \in Core(\Pi(r))$, what is not possible, since $p'(a) \in Core(\Pi(r))$.
   (b) $\neg p'(a)$ cannot be obtained via the default rule

$$\neg p'(a) \longleftarrow dom(a), not\ p(a), not\ p'(a),$$

   because it has the second subgoal false.
   (c) $\neg p'(a)$ cannot be obtained via a possible unfoundedness of $p'(a)$, because $p'(a)$ belongs to answer sets.
   (d) We are left with rules associated to FDs. Assume that

$$(\neg p(a) \vee \neg p(b) \vee c = d) \in IC_D. \tag{8}$$

   i If $c = d$, the associated triggering rule $\neg p'(a) \vee \neg p'(b) \longleftarrow p(a), p(b), c \neq d$. cannot be applied.
   ii If $c \neq d$, then in principle the triggering rule could be applied, but since $p'(a)$ belongs to all answer sets, without being forced to by a unary ICs, it must be case that $p(b)$ is false (otherwise, some repairs would get $p'(a)$ and others $p'(b)$, but not $p'(a)$).
   For the same reason, there is no $(p(b) \vee \chi(b)) \in IC_D$ with $\chi(b)$ false, because this would force $p'(b)$ to be true via the corresponding triggering rule; and this in its turn would force $\neg p'(a)$ to be true (to be in every answer set) due to the FD. This is not possible, because $p'(a)$ is already in $Core(\Pi(r))$.
   In consequence, the rule

$$\neg p'(a) \vee \neg p'(b) \longleftarrow p(a), p(b), c \neq d$$

   cannot be applied.

Now, we have to analyze the stabilizing rule $\neg p'(a) \longleftarrow p'(b), c \neq d$ associated to (8).

i If $c = d$, the rule does not apply.

ii If $c \neq d$, we have (as above) $p(b) \notin r$. Then, $\neg p(b) \in \mathscr{W}^1_{\Pi(r)}(\emptyset)$. Furthermore, $p'(b)$ cannot be obtained from the default $p'(b) \longleftarrow dom(b), p(b), not \neg p'(b)$, because $p(b)$ is false. We already saw that $p'(b)$ cannot be obtained from a rule $p'(b) \longleftarrow dom(b), \neg\chi(b)$.

In consequence, $p'(b)$ is unfounded, i.e. $\neg p'(b) \in \mathscr{W}^2_{\Pi(r)}(\emptyset)$, then, from the stabilizing rule, $\neg p'(a)$ turns out to be unfounded too: $p'(a) \in \mathscr{W}^3_{\Pi(r)}(\emptyset)$.

The two remaining cases that we will not prove are: (c) $p(a) \notin r$ and $\neg p'(a) \in Core(\Pi(r))$; (d) $p(a) \in r$ and $\neg p'(a) \in Core(\Pi(r))$. It is possible to show that always $Core(\Pi(r)) \subseteq \mathscr{W}^3_{\Pi(r)}(\emptyset)$.

References

Abiteboul, S., Hull, R. and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Arenas, M., Bertossi, L. and Chomicki, J. 1999. Consistent query answers in inconsistent databases. *Proceedings ACM Symposium on Principles of Database Systems (ACM PODS'99)*, pp. 68–79. ACM Press.

Arenas, M., Bertossi, L. and Chomicki, J. 2000. Specifying and querying database repairs using logic programs with exceptions. In: H. L. Larsen, J. Kacprzyk, S. Zadrozny and H. Christiansen (Eds.), *Flexible Query Answering Systems. Recent Developments*, pp. 27–41. Springer.

Arenas, M., Bertossi, L. and Kifer, M. 2000. Applications of annotated predicate calculus to querying inconsistent databases. *'Computational Logic – CL 2000'. Stream: 6th International Conference on Rules and Objects in Databases (DOOD'2000): LNAI 1861*, pp. 926–941. Springer.

Arenas, M., Bertossi, L. and Chomicki, J. 2001. Scalar aggregation in FD-inconsistent databases. *Database Theory – ICDT 2001 (Proceedings International Conference on Database Theory, ICDT'2001): LNCS 1973*, pp. 39–53. Springer.

Arieli, O. and Avron, A. 1999. A model-theoretic approach for recovering consistent data from inconsistent knowledge bases. *Journal of Automated Reasoning 22*, 2, 263–309.

Baral, C., Minker, J. and Kraus, S. 1991. Combining multiple knowledge bases. *IEEE Transactions on Knowledge and Data Engineering 3*, 2, 208–221.

Barcelo, P. and Bertossi, L. 2002. Repairing databases with annotated predicate logic. *Proceedings Ninth International Workshop on Non-Monotonic Reasoning (NMR'2002)*, pp. 160–170. Morgan Kaufmann.

Barcelo, P. and Bertossi, L. 2003. Logic programs for querying inconsistent databases. *Proceedings Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003): LNCS 2562*, pp. 208–222. Springer.

Blair, H. A. and Subrahmanian, V. S. 1989. Paraconsistent logic programming. *Theoretical Computer Science 68*, 135–154.

Bry, F. 1997. Query answering in information systems with integrity constraints. *Proceedings First IFIP TC11 Working Conference on Integrity and Internal Control in Information Systems*, pp. 113–130. Chapman & Hall.

Buccafurri, F., Leone, N. and Rullo, P. 2000. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering 12*, 5, 845–860.

Calimeri, F., Faber, W., Leone, N. and Pfeifer, G. 2002. Pruning operators for answer set programming systems. In: S. Benferhat and E. Giunchiglia (Eds.), *Proceedings Ninth International Workshop on Non-Monotonic Reasoning (NMR'2002)*, pp. 200–209. Morgan Kaufmann.

Celle, A. and Bertossi, L. 2000. Querying inconsistent databases: algorithms and implementation. *'Computational Logic – CL 2000'. Stream: 6th International Conference on Rules and Objects in Databases (DOOD'2000): LNAI 1861*, pp. 942–956. Springer.

Chou, T. and Winslett, M. 1994. A model-based belief revision system. *Journal of Automated Reasoning 12*, 157–208.

Dalal, M. 1988. Investigations into a theory of knowledge base revision: preliminary report. *Proceedings Seventh National Conference on Artificial Intelligence (AAAI'88)*, pp. 475–479.

Damasio, C. V. and Moniz-Pereira, L. 1998. A survey on paraconsistent semantics for extended logic programs. In: D. M. Gabbay and Ph. Smets (Eds.), *Handbook of Defeasible Reasoning and Uncertainty Management Systems, Vol. 2*, pp. 241–320. Kluwer Academic.

Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys 33*, 3, 374–425.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G. and Scarcello, F. 1998. The knowledge representation system DLV: progress report, comparisons, and benchmarks. *Proceedings International Conference on Principles of Knowledge Representation and Reasoning (KR98)*, Trento, Italy. Morgan Kaufman.

Eiter, T., Faber, W., Leone, N. and Pfeifer, G. 2000. Declarative problem-solving in DLV. In: J. Minker (Ed.), *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer.

Fitting, M. 1996. *First Order Logic and Automated Theorem Proving. 2nd ed.* Texts and Monographs in Computer Science, Springer.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In: R. A. Kowalski and K. A. Bowen (Eds.), *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pp. 1070–1080. MIT Press.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 9, 365–385.

Giannotti, F., Greco, S., Sacca, D. and Zaniolo, C. 1997. Programming with non-determinism in deductive databases. *Annals of Mathematics and Artificial Intelligence 19*, 3–4.

Greco, S., Sacca, D. and Zaniolo, C. 1995. Datalog queries with stratified negation and choice: from P to D^P. *Proceedings International Conference on Database Theory*, pp. 82–96. Springer.

Greco, S. and Zumpano, E. 2000. Querying inconsistent databases. *Proceedings 7th International Conference on Logic for Programming and Automated Reasoning (LPAR'2000): LNCS 1955*, pp. 308–325.

Greco, S. and Zumpano, E. 2001. Computing repairs for inconsistent databases. *Proceedings Third International Symposium on Cooperative Database Systems for Advanced Applications (CODAS01)*, Beijing, China.

Greco, G., Greco, S. and Zumpano, E. 2001. A logic programming approach to the integration, repairing and querying of inconsistent databases. In: Ph. Codognet (Ed.), *Proceedings 17th International Conference on Logic Programming (ICLP'01): LNCS 2237*, pp. 348–364. Springer.

Kowalski, R. and Sadri, F. 1991. Logic programs with exceptions. *New Generation Computing 9*, 387–400.

Kifer, M. and Lozinskii, E. L. 1992. A logic for reasoning with inconsistency. *Journal of Automated Reasoning 9*, 2, 179–215.

Kifer, M. and Subrahmanian, V. S. 1992. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming 12*, 4, 335–368.

Leach, S. M. and Lu, J. J. 1996. Query processing in annotated logic programming: theory and implementation. *Journal of Intelligent Information Systems 6*, 33–58.

Leone, N., Rullo, P. and Scarcello, F. 1997. Disjunctive stable models: unfounded sets, fixpoint semantics, and computation. *Information and Computation 135*, 2, 69–112.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In: P. van Hentenryck (Ed.), *Proceedings Eleventh International Conference on Logic Programming*, pp. 23–37. MIT Press.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Springer Verlag.

Lozinskii, E. L. 1994. Resolving contradictions: a plausible semantics for inconsistent systems. *Journal of Automated Reasoning 12*, 1, 1–32.

Marek, V. W. and Truszczynski, M. 1998. Revision programming. *Theoretical Computer Science 190*, 2, 241–277.

Subrahmanian V. S. 1994. Amalgamating knowledge bases. *ACM Transactions on Database Systems 19*, 2, 291–331.

Ullman, J. 1988. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press.

Winslett, M. 1988. Reasoning about action using a possible model approach. *Proceedings Seventh National Conference on Artificial Intelligence (AAAI'88)*, pp. 89–93.