Logic Programs for Querying Inconsistent Databases

Pablo Barceló
University of Toronto
Department of Computer Science
Toronto, Canada.
pablo@cs.toronto.edu

Leopoldo Bertossi
Carleton University
School of Computer Science
Ottawa, Canada.
bertossi@scs.carleton.ca

Abstract. Consistent answers from a relational database that violates a given set of integrity constraints (ICs) are characterized as ordinary answers that can be obtained from every minimally repaired version of the database (a repair). Repairs can be specified and interpreted as the stable models of a simple disjunctive normal logic program with database predicates extended with appropriate annotation arguments. In consequence, consistent query answers can be obtained by running a query program in combination with the repair program under the cautious or skeptical stable model semantics. In this paper we show how to write repair programs for universal and referential ICs; we establish their correctness and show how to run them on top of the *DLV* system.

1 Introduction

Integrity constraints (ICs) capture the semantics of a relational database, establishing its correspondence with the application domain that the database is modeling. However, it is not unusual for a database instance to become inconsistent with respect to a given, intended set of ICs. This could happen due to different factors, being one of them the integration of several data sources. The integration of consistent databases may easily lead to an inconsistent integrated database.

A natural problem in databases consists in retrieving answers to queries that are "consistent" with the given ICs, even when the database as a whole does not satisfy those ICs. Very likely "most" of the data is still consistent. The notion of consistent answer to a first order (FO) query was defined in [2], where also a computational mechanism for obtaining consistent answers was presented. Intuitively speaking, a ground tuple $\bar{t}$ to a first order query $Q(\bar{x})$ is *consistent* in a, possibly inconsistent, relational database instance DB if it is an (ordinary) answer to $Q(\bar{x})$ in every minimal repair of DB, i.e. in every database instance over the same schema and domain that differs from DB by a minimal (under set inclusion) set of inserted or deleted tuples.

Computing consistent query answers is a natural problem in DBs. Apart from applications in data integration [10], we also foresee interesting applications in the context of intelligent information systems, where a particular user might impose his/her particular view of the semantics of the database by querying the database through his/her user ICs, that are not necessarily maintained by the DB central administration. This user could specify his/her own constraints as queries are posed, by means of a new, extra SQL statement or a new option in the usual menu for interacting with the DB.

The mechanism presented in [2] for consistent query answering (CQA) has some limitations in terms of the ICs and queries it can handle. In [4], a more general methodology based on logic programs with a stable model semantics was introduced. More general queries could be considered, but ICs were restricted to be "binary", i.e. universal with at most two database literals (plus built-ins). Independently, a similar methodology based on logic programs for CQA wrt arbitrary universal ICs was presented in [23].

For CQA we need to deal with all the repairs of a database, but hopefully in a compact, succinct manner, without having to compute all of them explicitly. Actually, the database repairs corresponds to just an auxiliary conceptual notion used to characterize what is relevant to us, the consistent answers. In consequence, a natural approach consists in providing a manageable logical specification of the class of database repairs, that treats them as a whole. The specification must include information about the database and the ICs.

In this paper we show how to specify the database repairs by means of simple classical disjunctive normal programs with a stable model semantics. The database predicates in these programs contain annotations as extra arguments. In their turn, the annotations are inspired by the theories written in annotated predicate logic that specify database repairs as presented in [3, 8]. Nevertheless, the programs are classical, as opposed to annotated or paraconsistent logic programs [11, 25]. The *coherent* stable models of the program turn out to correspond to the database repairs. The logic programs introduced in [4, 23] to specify database repairs may contain an exponential number of rules depending on the number of database literals appearing in the ICs. However, the programs presented here contain only a linear number of rules.

With this approach we reach two goals. The first goal consists in obtaining a computable specification of all the possible minimal sets of changes required to restore the consistency of a theory corresponding to the positive information explicitly stored in a relational database. However, we are not interested in computing database repairs, neither in repairing in any way the inconsistent database. Actually, the main, second goal consists in providing a general computational mechanism to obtain the consistent answers to a first order query. They can be obtained by "running" the combination of the repair program and a query program under the skeptical stable model semantics that sanctions as true what is true of every stable model. The less a logic programming implementation explicitly computes all stable models in order to answer a query, the better. We have experimented with *DLV*, an implementation of the disjunctive stable model semantics [19].

The methodology presented here works for arbitrary first order queries and arbitrary universal ICs, what considerable extends the cases that could be handled in [2, 4, 3]. We also show how to apply the methodology in the presence of referential integrity constraints [1].

2 Preliminaries

2.1 Database repairs and consistent answers

We consider a fixed relational database schema $\Sigma = (D, P, B)$, consisting of a fixed, possibly infinite, database domain $D = \{c_1, c_2, ...\}$, a fixed set of database predicates $P = \{p_1, \ldots, p_n\}$ with fixed arities, and a fixed set of built-in predicates $B = \{e_1, \ldots, e_m\}$. This schema determines a first order language $\mathcal{L}(\Sigma)$.

A database instance over Σ is a finite collection DB of facts of the form $p(c_1, ..., c_n)$, where p is a predicate in P and $c_1, ..., c_n$ are constants in D. A built-in predicate has a fixed and the same extension in every database instance, not subject to any changes.

An *integrity constraint* (IC) is an implicitly quantified clause of the form

$$\bigvee_{i=1}^{n} \neg p_i(\bar{t}_i) \vee \bigvee_{j=1}^{m} q_j(\bar{s}_j) \vee \varphi, \tag{1}$$

where, p_i and q_j are predicates in P, $\bar{t}_i, \bar{s}_j$ are tuples containing constants and variables, and φ is a formula containing predicates in B only.

We will assume that DB and IC, separately, are consistent theories. Nevertheless, it may be the case that $DB \cup IC$ is inconsistent. Equivalently, if we associate in the natural way to DB a first order structure, also denoted with DB, i.e. by applying the closed world assumption (CWA) that makes false any ground atom not explicitly appearing in the set of atoms DB, it may happen that DB, as a structure, does not satisfy the IC. We denote with $DB \models_\Sigma IC$ the fact that the database satisfies IC. In this case we say that DB is consistent wrt IC; otherwise we say DB is inconsistent.

As in [2], we define that the *distance* between two database instances DB_1 and DB_2 is their symmetric difference $\Delta(DB_1, DB_2) = (DB_1 - DB_2) \cup (DB_2 - DB_1)$.

Now, given database instances DB, possibly inconsistent wrt IC, we say that the instance DB' is a *repair* of DB iff $DB' \models_\Sigma IC$ and $\Delta(DB, DB')$ is minimal under set inclusion in the class of instances that satisfy IC [2], that is, there is no instance DB'' such that $DB'' \models_\Sigma IC$ and $\Delta(DB, DB'') \subsetneq \Delta(DB, DB')$.

Example 1. Consider the relational schema $Book(author, name, publYear)$ and a database instance $DB = \{Book(kafka, metamorph, 1915),\ Book(kafka, metamorph, 1919)\}$. We also have the functional dependency [1] $FD\colon author, name \rightarrow publYear$, that can be expressed by $IC\colon \neg Book(x, y, z) \vee \neg Book(x, y, w) \vee z = w$.

DB is inconsistent with respect to IC. The original instance has two possible repairs, namely $DB_1 = \{Book(kafka, metamorph, 1915)\}$ and $DB_2 = \{Book(kafka, metamorph, 1919)\}$. □

Let DB be a database instance, possibly not satisfying a set IC of integrity constraints. Given a query $Q(\bar{x})$ to DB, we say that a tuple of constants $\bar{t}$ is a *consistent answer* [2], denoted $DB \models_c Q(\bar{t})$, if for every repair DB' of DB, $DB' \models_\Sigma Q(\bar{t})$. If Q is a closed formula, i.e. a sentence, then *true* is a *consistent answer* to Q, denoted $DB \models_c Q$, if for every repair DB' of DB, $DB' \models_\Sigma Q$.

Example 2. (example 1 continued) Query Q_1: $Book(kafka, metamorph, 1915)$ does not have *true* as a consistent answer, because it is not true in every repair. Query $Q_2(y)$: $\exists x \exists z Book(x, y, z)$ has $y = metamorph$ as a consistent answer. Query $Q_3(x)$: $\exists z Book(x, metamorph, z)$ has $x = kafka$ as a consistent answer. □

Notice that repairs are obtained by insertion/deletion of whole relational tuples, and we do not specify a preference for any particular kind of repairs. Repairs are just used to characterize the consistent answers.

Annotated Predicate Calculus was introduced in [24]. It constitutes a non classical logic where classical inconsistencies may be accommodated without trivializing reasoning. Its syntax is similar to that of classical logic, except for the fact that atoms are annotated with values drawn from a truth-values lattice. In [3], in order to embed the database and the ICs into a single consistent theory, a particular lattice was introduced. It contains the four usual truth values: $\mathbf{t}, \mathbf{f}$ (classical true and false), $\top$ (inconsistent), $\bot$ (unknown) plus $\mathbf{t_c}$, $\mathbf{t_d}$, $\mathbf{f_d}$, $\mathbf{f_c}$, $\mathbf{t_a}$, $\mathbf{f_a}$. It was also shown that there is a one to one correspondence between some minimal models of the annotated theory and the repairs of the inconsistent database for universal ICs. This was extended to existential ICs in [8].

The values $\mathbf{t_c}$, $\mathbf{f_c}$ are used to annotate what is needed for constraint satisfaction: If the IC is written in disjunctive normal form, positive, resp. negative, literals are annotated with $\mathbf{t_c}$, resp. $\mathbf{f_c}$. The values $\mathbf{t_d}$ and $\mathbf{f_d}$ represent the truth values according to the original database: An atom in (outside) the DB is annotated with $\mathbf{t_d}$ ($\mathbf{f_d}$). Finally, $\mathbf{t_a}$ and $\mathbf{f_a}$ are considered *advisory* truth values, to solve conflicts between the original database and the ICs, that have to be satisfied by the former. Conflicts are always solved in favor of the ICs, as only the database instance can be changed to restore consistency. That is, $lub(\mathbf{t_d}, \mathbf{f_c}) = \mathbf{f_a}$, meaning that in case an atom is true at the data level, but false at the IC level, it will get the derived annotation $\mathbf{f_a}$, expressing that it is advised to make it false, i.e. to delete it from the database. Similarly, $lub(\mathbf{f_d}, \mathbf{t_c}) = \mathbf{t_a}$ is an indication that the literal which receives the annotation $\mathbf{t_a}$ must be inserted into the database.

In this paper, instead of explicitly using a lattice, the annotations will be new constants in the language, to be used in an extra argument introduced in the database atoms. The rules 3. in Definition 1 will capture the relationships between the annotations we described above.

3 Logic Programming Specification of Repairs

In this section we will consider ICs of the form (1). Our aim is to specify database repairs using classical first order logic programs. However, those programs will be suggested by the non classical annotated theory. In order to accommodate annotations in this classical framework, we replace each predicate $p(\bar{x}) \in P$ by a new predicate $p(\bar{x}, \cdot)$, with an extra argument for annotations. This defines a new FO language, $\mathcal{L}(\Sigma)^{an}$, for annotated $\mathcal{L}(\Sigma)$.

Definition 1. *The repair logic program, $\Pi(DB, IC)$, for DB and IC, is written with predicates from $\mathcal{L}(\Sigma)^{an}$ and contains the following clauses:*

1. *For every atom $p(\bar{a}) \in DB$, $\Pi(DB, IC)$ contains the fact $p(\bar{a}, \mathbf{t_d})$.*
2. *For every predicate $p \in P$, $\Pi(DB, IC)$ contains the clauses:*

   $p(\bar{x}, \mathbf{t}^\star) \leftarrow p(\bar{x}, \mathbf{t_d})$. $\quad p(\bar{x}, \mathbf{t}^\star) \leftarrow p(\bar{x}, \mathbf{t_a})$.
   $p(\bar{x}, \mathbf{f}^\star) \leftarrow p(\bar{x}, \mathbf{f_a})$. $\quad p(\bar{x}, \mathbf{f}^\star) \leftarrow$ *not* $p(\bar{x}, \mathbf{t_d})$.,

   where $\mathbf{t}^\star, \mathbf{f}^\star$ are new, auxiliary elements in the domain of annotations.
3. *For every constraint of the form (1), $\Pi(DB, IC)$ contains the clause:*

   $$\bigvee_{i=1}^{n} p_i(\bar{t_i}, \mathbf{f_a}) \vee \bigvee_{j=1}^{m} q_j(\bar{s_j}, \mathbf{t_a}) \quad \longleftarrow \quad \bigwedge_{i=1}^{n} p_i(\bar{t_i}, \mathbf{t}^\star) \wedge \bigwedge_{j=1}^{m} q_j(\bar{s_j}, \mathbf{f}^\star) \wedge \bar{\varphi},$$

   where $\bar{\varphi}$ represents the negation of φ. □

Intuitively, the clauses in 3. say that when the IC is violated (the body), then DB has to be repaired according to one of the alternatives shown in the head. Since there may be interactions between constraints, these single repairing steps may not be enough to restore the consistency of DB. We have to make sure that the repairing process continues and stabilizes in a state where all the ICs hold. This is the role of the clauses in 2. containing the new annotations $\mathbf{t}^\star$, that groups together those atoms annotated with $\mathbf{t_d}$ and $\mathbf{t_a}$, and $\mathbf{f}^\star$, that does the same with $\mathbf{f_d}$ and $\mathbf{f_a}$. Notice that the annotations $\mathbf{t}^\star$, $\mathbf{f}^\star$, obtained through the combined effect of rules 2. and 3., can be fed back into rules 3. until consistency is restored. This possibility is what allows us to have just one program rule for each IC.

Example 3 shows the interaction of a functional dependency and an inclusion dependency. When atoms are deleted in order to satisfy the functional dependency, the inclusion dependency could be violated, and in a second step it should be repaired. At that second step, the annotations $\mathbf{t}^\star$ and $\mathbf{f}^\star$, computed at the first step where the functional dependency was repaired, will detect the violation of the inclusion dependency and trigger the corresponding repairing process.

Example 3. (example 1 continued) We extend the schema with the table *Eurbook*(*author*, *name*, *publYear*), for European books. Now, DB also contains the literal *Eurbook*(*kafka*, *metamorph*, *1919*)}. If in addition to the ICs we had before, we consider the set inclusion dependency $\forall xyz\ (Eurbook(x, y, z) \rightarrow Book(x, y, z))$, we obtain the following program $\Pi(DB, IC)$:

1. $EurBook(kafka, metamorph, 1919, \mathbf{t_d})$. $\quad Book(kafka, metamorph, 1919, \mathbf{t_d})$.
$Book(kafka, metamorph, 1915, \mathbf{t_d})$.

2. $Book(x, y, z, \mathbf{t}^\star) \leftarrow Book(x, y, z, \mathbf{t_d})$. $\quad Book(x, y, z, \mathbf{t}^\star) \leftarrow Book(x, y, z, \mathbf{t_a})$.
$Book(x, y, z, \mathbf{f}^\star) \leftarrow Book(x, y, z, \mathbf{f_a})$. $\quad Book(x, y, z, \mathbf{f}^\star) \leftarrow$ *not* $Book(x, y, z, \mathbf{t_d})$.
$Eurbook(x, y, z, \mathbf{t}^\star) \leftarrow Eurbook(x, y, z, \mathbf{t_d})$.

$Eurbook(x, y, z, \mathbf{t}^\star) \leftarrow Eurbook(x, y, z, \mathbf{t_a})$.
$Eurbook(x, y, z, \mathbf{f}^\star) \leftarrow Eurbook(x, y, z, \mathbf{f_a})$.
$Eurbook(x, y, z, \mathbf{f}^\star) \leftarrow$ not $Eurbook(x, y, z, \mathbf{t_d})$.

3. $Book(x, y, z, \mathbf{f_a}) \vee Book(x, y, w, \mathbf{f_a}) \ \leftarrow \ Book(x, y, z, \mathbf{t}^\star), Book(x, y, w, \mathbf{t}^\star),$
$z \neq w.$
$Eurbook(x, y, z, \mathbf{f_a}) \vee Book(x, y, z, \mathbf{t_a}) \ \leftarrow \ Eurbook(x, y, z, \mathbf{t}^\star), Book(x, y, z, \mathbf{f}^\star).$
□

For our programs, that contain negation as failure, we will consider the stable models semantics. A model $\mathcal{M}$ is a stable model of a disjunctive program P iff it is a minimal model of $P^{\mathcal{M}}$, where $P^{\mathcal{M}} = \{A_1 \vee \cdots \vee A_n \leftarrow B_1, \cdots, B_m \mid A_1 \vee \cdots \vee A_n \leftarrow B_1, \cdots, B_m, not\ C_1, \cdots, not\ C_k$ is a ground instance of a clause in P and $\mathcal{M} \not\models C_i$ for $1 \leq i \leq k\}$ [20, 21].

Definition 2. *A Herbrand model $\mathcal{M}$ is* coherent *if it does not contain both $p(\bar{a}, \mathbf{t_a})$ and $p(\bar{a}, \mathbf{f_a})$.* □

Example 4. (example 3 continued) The coherent stable models of the program presented in Example 3 are:

$\mathcal{M}_1 = \{Book(kafka, metamorph, 1919, \mathbf{t_d}),\ Book(kafka, metamorph, 1919, \mathbf{t}^\star),$
$Book(kafka, metamorph, 1915, \mathbf{t_d}),\ Book(kafka, metamorph, 1915, \mathbf{t}^\star),$
$Book(kafka, metamorph, 1915, \mathbf{f_a}),\ Book(kafka, metamorph, 1915, \mathbf{f}^\star),$
$Eurbook(kafka, metamorph, 1919, \mathbf{t_d}),\ Eurbook(kafka, metamorph, 1919, \mathbf{t}^\star)\};$

$\mathcal{M}_2 = \{Book(kafka, metamorph, 1919, \mathbf{t_d}),\ Book(kafka, metamorph, 1919, \mathbf{t}^\star),$
$Book(kafka, metamorph, 1919, \mathbf{f_a}),\ Book(kafka, metamorph, 1919, \mathbf{f}^\star),$
$Book(kafka, metamorph, 1915, \mathbf{t_d}),\ Book(kafka, metamorph, 1915, \mathbf{t}^\star),$
$Eurbook(kafka, metamorph, 1919, \mathbf{t_d}),\ Eurbook(kafka, metamorph, 1919, \mathbf{t}^\star),$
$Eurbook(kafka, metamorph, 1919, \mathbf{f_a}),\ Eurbook(kafka, metamorph, 1919, \mathbf{f}^\star)\}.$ □

The stable models of the program will include the database contents with its original annotations ($\mathbf{t_d}$). Every time there is an atom in a model annotated with $\mathbf{t_d}$ or $\mathbf{t_a}$, it will appear annotated with $\mathbf{t}^\star$. From these models we should be able to "read" database repairs. Every stable model of the logic program has to be interpreted. In order to do this, we introduce two new annotations, $\mathbf{t}^{\star\star}, \mathbf{f}^{\star\star}$, in the last arguments. The first one groups together those atoms annotated with $\mathbf{t_a}$ and those annotated with $\mathbf{t_d}$, but not $\mathbf{f_a}$. Intuitively, they correspond to those annotated with $\mathbf{t}$ in the models of $\mathcal{T}(DB, IC)$. A similar role plays the other new annotation wrt the "false" annotations. These new annotations will simplify the expression of the queries to be posed to the program. Without them, instead of simply asking $p(\bar{x}, \mathbf{t}^{\star\star})$ (for the tuples in p in a repair), we would have to ask for $p(\bar{x}, \mathbf{t_a}) \vee (p(\bar{x}, \mathbf{t_d}) \wedge \neg p(\bar{x}, \mathbf{f_a}))$. The interpreted models can be easily obtained by adding new rules.

Definition 3. *The interpretation program $\Pi^\star(DB, IC)$ extends $\Pi(DB, IC)$ with the following rules:*

$$p(\bar{a}, \mathbf{f}^{\star\star}) \ \leftarrow \ p(\bar{a}, \mathbf{f_a}). \qquad p(\bar{a}, \mathbf{f}^{\star\star}) \ \leftarrow \ not\ p(\bar{a}, \mathbf{t_d}),\ not\ p(\bar{a}, \mathbf{t_a}).$$

$$p(\bar{a}, \mathbf{t}^{\star\star}) \leftarrow p(\bar{a}, \mathbf{t_a}). \qquad p(\bar{a}, \mathbf{t}^{\star\star}) \leftarrow p(\bar{a}, \mathbf{t_d}),\ not\ p(\bar{a}, \mathbf{f_a}). \qquad \Box$$

Example 5. (example 4 continued) The coherent stable models of the interpretation program extend

$\mathcal{M}_1$ with $\{Eurbook(kafka, metamorph, 1919, \mathbf{t}^{\star\star}),$
$Book(kafka, metamorph, 1919, \mathbf{t}^{\star\star}), Book(kafka, metamorph, 1915, \mathbf{f}^{\star\star})\}$;

$\mathcal{M}_2$ with $\{Eurbook(kafka, metamorph, 1919, \mathbf{f}^{\star\star}),$
$Book(kafka, metamorph, 1919, \mathbf{f}^{\star\star}), Book(kafka, metamorph, 1915, \mathbf{t}^{\star\star})\}$. $\Box$

From an interpretation model we can obtain a database instance.

Definition 4. *Let $\mathcal{M}$ be a coherent stable model of program $\Pi^\star(DB, IC)$. The database associated to $\mathcal{M}$ is $DB_{\mathcal{M}} = \{p(\bar{a}) \mid p(\bar{a}, \mathbf{t}^{\star\star}) \in \mathcal{M}\}$.* $\Box$

Theorem 1 establishes the one-to-one correspondence between coherent stable models of the program and the repairs of the original instance.

Theorem 1. *If $\mathcal{M}$ is a coherent stable model of $\Pi^\star(DB, IC)$, and $DB_{\mathcal{M}}$ is finite, then $DB_{\mathcal{M}}$ is a repair of DB with respect to IC. Furthermore, the repairs obtained in this way are all the repairs of DB.* $\Box$

Example 6. (example 5 continued) The following database instances obtained from Definition 4 are the repairs of DB:

$$DB_{\mathcal{M}_1} = \{Eurbook(kafka, metamorph, 1919),\ Book(kafka, metamorph, 1919)\},$$
$$DB_{\mathcal{M}_2} = \{Book(kafka, metamorph, 1915)\}. \qquad \Box$$

3.1 The query program

Given a first order query Q, we want the consistent answers from DB. In consequence, we need those atoms that are simultaneously true of Q in every stable model of the program $\Pi(DB, IC)$. They are obtained through the query $Q^{\star\star}$, obtained from Q by replacing, for $p \in P$, every positive literal $p(\bar{s})$ by $p(\bar{s}, \mathbf{t}^{\star\star})$ and every negative literal $\neg p(\bar{s})$ by $p(\bar{s}, \mathbf{f}^{\star\star})$. Now $Q^{\star\star}$ can be transformed into a query program $\Pi(Q^{\star\star})$ by a standard transformation [27, 1]. This query program will be run in combination with $\Pi^\star(DB, IC)$.

Example 7. For the query $Q(y) : \exists z Book(kafka, y, z)$, we generate $Q^{\star\star}(y) : \exists z Book(kafka, y, z, \mathbf{t}^{\star\star})$, that is transformed into the query program clause $Answer(y) \leftarrow Book(kafka, y, z, \mathbf{t}^{\star\star})$. $\Box$

4 Computing from the Program

The database repairs could be computed using an implementation of the disjunctive stable models semantics like *DLV* [19], that also supports denial constraints as studied in [12]. In this way we are able to prune out the models that are not coherent, imposing for every predicate p the constraint $\leftarrow p(\bar{x}, \mathbf{t_a}), p(\bar{x}, \mathbf{f_a})$.

Example 8. Consider the database instance $\{p(a)\}$ that is inconsistent wrt the set inclusion dependency $\forall x\ (p(x) \rightarrow q(x))$. The program $\Pi^\star(DB, IC)$ contains the following clauses:

1. Database contents: $p(a, \mathbf{t_d})$.
2. Rules for the closed world assumption:

   $p(x, \mathbf{f^\star}) \leftarrow\ not\ p(x, \mathbf{t_d}).\qquad q(x, \mathbf{f^\star}) \leftarrow\ not\ q(x, \mathbf{t_d}).$
3. Annotation rules:

   $p(x, \mathbf{f^\star}) \leftarrow p(x, \mathbf{f_a}).\qquad p(x, \mathbf{t^\star}) \leftarrow p(x, \mathbf{t_a}).\qquad p(x, \mathbf{t^\star}) \leftarrow p(x, \mathbf{t_d}).$
   $q(x, \mathbf{f^\star}) \leftarrow q(x, \mathbf{f_a}).\qquad q(x, \mathbf{t^\star}) \leftarrow q(x, \mathbf{t_a}).\qquad q(x, \mathbf{t^\star}) \leftarrow q(x, \mathbf{t_d}).$
4. Rule for the IC: $p(x, \mathbf{f_a}) \vee q(x, \mathbf{t_a}) \leftarrow p(x, \mathbf{t^\star}), q(x, \mathbf{f^\star})$.
5. Denial constraints for coherence

   $\leftarrow p(\bar{x}, \mathbf{t_a}), p(\bar{x}, \mathbf{f_a}).\qquad \leftarrow q(\bar{x}, \mathbf{t_a}), q(\bar{x}, \mathbf{f_a}).$
6. Interpretation rules:

   $p(x, \mathbf{t^{\star\star}}) \leftarrow p(x, \mathbf{t_a}).\qquad p(x, \mathbf{t^{\star\star}}) \leftarrow p(x, \mathbf{t_d}),\ not\ p(x, \mathbf{f_a}).$
   $p(x, \mathbf{f^{\star\star}}) \leftarrow p(x, \mathbf{f_a}).\qquad p(x, \mathbf{f^{\star\star}}) \leftarrow\ not\ p(x, \mathbf{t_d}),\ not\ p(x, \mathbf{t_a}).$
   $q(x, \mathbf{t^{\star\star}}) \leftarrow q(x, \mathbf{t_a}).\qquad q(x, \mathbf{t^{\star\star}}) \leftarrow q(x, \mathbf{t_d}),\ not\ q(x, \mathbf{f_a}).$
   $q(x, \mathbf{f^{\star\star}}) \leftarrow q(x, \mathbf{f_a}).\qquad q(x, \mathbf{f^{\star\star}}) \leftarrow\ not\ q(x, \mathbf{t_d}),\ not\ q(x, \mathbf{t_a}).$

Running program $\Pi^\star(DB, IC)$ with *DLV* we obtain two stable models:

$\mathcal{M}_1 = \{p(a, \mathbf{t_d}), p(a, \mathbf{t^\star}), q(a, \mathbf{f^\star}), q(a, \mathbf{t_a}), p(a, \mathbf{t^{\star\star}}), q(a, \mathbf{t^\star}), q(a, \mathbf{t^{\star\star}})\}$,

$\mathcal{M}_2 = \{p(a, \mathbf{t_d}), p(a, \mathbf{t^\star}), p(a, \mathbf{f^\star})), q(a, \mathbf{f^\star}), p(a, \mathbf{f^{\star\star}}), q(a, \mathbf{f^{\star\star}}), p(a, \mathbf{f_a})\}$.

The first model says, through its atom $q(a, \mathbf{t^{\star\star}})$, that $q(a)$ has to be inserted in the database. The second one, through its atom $p(a, \mathbf{f^{\star\star}})$, that $p(a)$ has to be deleted. □

The coherence denial constraints did not play any role in the previous example, we obtain exactly the same model with or without them. The reason is that we have only one IC; in consequence, only one step is needed to obtain a repair of the database. There is no way to obtain an incoherent stable model due to the application of the rules 1. and 2. in Example 8 in a second repair step.

Example 9. (example 8 continued) Let us now add an extra set inclusion dependency, $\forall x\ (q(x) \rightarrow r(x))$, keeping the same instance. One repair is obtained by inserting $q(a)$, what causes the insertion of $r(a)$. The program is as before, but with the additional rules

$r(x, \mathbf{f^\star}) \leftarrow not\ r(x, \mathbf{t_d}).\quad r(x, \mathbf{f^\star}) \leftarrow r(x, \mathbf{f_a}).\quad r(x, \mathbf{t^\star}) \leftarrow r(x, \mathbf{t_a}).$
$r(X, \mathbf{t^\star}) \leftarrow r(X, \mathbf{t_d}).\quad r(x, \mathbf{t^{\star\star}}) \leftarrow r(x, \mathbf{t_a}).\quad r(x, \mathbf{t^{\star\star}}) \leftarrow r(x, \mathbf{t_d}), not\ r(x, \mathbf{f_a}).$
$r(x, \mathbf{f^{\star\star}}) \leftarrow r(x, \mathbf{f_a}).\quad r(x, \mathbf{f^{\star\star}}) \leftarrow not\ r(x, \mathbf{t_d}), not\ r(x, \mathbf{t_a}).$
$q(x, \mathbf{f_a}) \vee r(x, \mathbf{t_a}) \leftarrow q(x, \mathbf{t^\star}), r(x, \mathbf{f^\star}).\qquad \leftarrow r(x, \mathbf{t_a}), r(x, \mathbf{f_a}).$

If we run the program we obtain the expected models, one that deletes $p(a)$, and a second one that inserts both $q(a)$ and $r(a)$. However, if we omit the coherence denial constraints, more precisely the one for table q, we obtain a third model, namely $\{p(a,\mathbf{t_d}), p(a,\mathbf{t}^\star), q(a,\mathbf{f}^\star), r(a,\mathbf{f}^\star), q(a,\mathbf{f_a}), q(a,\mathbf{t_a}), p(a,\mathbf{t}^{\star\star}),$ $q(a,\mathbf{t}^\star), q(a,\mathbf{t}^{\star\star}), q(a,\mathbf{f}^{\star\star}), r(a,\mathbf{f}^{\star\star})$, that is not coherent, because it contains both $q(a,\mathbf{f_a})$ and $q(a,\mathbf{t_a})$, and cannot be interpreted as a repair of the original database. □

Notice that the programs with annotations obtained are very simple in terms of their dependency on the ICs. As mentioned before, consistent answers can be obtained "running" a query program together with the repair program $\Pi^\star(DB, IC)$, under the skeptical stable model semantics, that sanctions as true what is true of all stable models.

5 Programs with Referential ICs

So far the repair programs have been for universal ICs. Now we also want to consider referential ICs (RICs) of the form $p(\bar{x}) \rightarrow \exists y(q(\bar{x}', y))$, where $\bar{x}' \subseteq \bar{x}$. It is assumed that the variables range over an underlying database domain D, that does not include the value *null*, nevertheless, a RIC can be repaired by insertion of the null value, say $q(\bar{a}, \mathit{null})$, or by cascaded deletion. If the repair is by introduction of *null*, it is expected that this change will not propagate through other ICs, e.g. a set inclusion dependency like $\forall \bar{x}(q(\bar{x}', y) \rightarrow r(\bar{x}', y))$. The program should not detect such inconsistency wrt this IC. This can be easily avoided at the program level by appropriately qualifying the values of variables in the disjunctive repair clause for the other ICs, like the set inclusion IC above.

The program $\Pi^\star(DB, IC)$ is then extended with the following formulas:

$$p(\bar{x}, \mathbf{f_a}) \vee q(\bar{x}', \mathit{null}, \mathbf{t_a}) \leftarrow p(\bar{x}, \mathbf{t}^\star),\ \mathit{not}\ \mathit{aux}(\bar{x}'),\ \mathit{not}\ q(\bar{x}', \mathit{null}, \mathbf{t_d}). \quad (2)$$

$$\mathit{aux}(\bar{x}') \leftarrow q(\bar{x}', y, \mathbf{t_d}),\ \mathit{not}\ q(\bar{x}', y, \mathbf{f_a}). \quad (3)$$

$$\mathit{aux}(\bar{x}') \leftarrow q(\bar{x}', y, \mathbf{t_a}). \quad (4)$$

Intuitively, clauses (3) and (4) detect if the formula $\exists y(q(\bar{a}', y){:}\mathbf{t} \vee q(\bar{a}', y){:}\mathbf{t_a}))$ is satisfied by the model. If this is not the case, and $p(\bar{a}, \mathbf{t}^\star)$ belongs to the model, and $q(\bar{a}', \mathit{null})$ is not in the original instance, i.e. there is a violation of the RIC, then, according to rule (2), the repair is done either by deleting $p(\bar{a})$ or inserting $q(\bar{a}', \mathit{null})$.

Example 10. Consider the database instance $\{p(\bar{a})\}$ and the following set of ICs: $p(x) \rightarrow \exists y q(x, y)$, $q(x, y) \rightarrow r(x, y)$. The program $\Pi^\star(DB, IC)$ is written in *DLV* as follows (`ts, tss, ta`, etc. stand for $\mathbf{t}^\star, \mathbf{t}^{\star\star}, \mathbf{t_a}$, etc.):

Database contents

```
domd(a).    d(a,td).     p(X,td) :- d(X,td), domd(X).
```

Rules for CWA

```
p(X,fs) :- domd(X), not p(X,td).
q(X,Y,fs) :- domd(X), domd(Y), not q(X,Y,td).
r(X,Y,fs) :-  not r(X,Y,td), domd(X), domd(Y).
```

Annotation rules

```
p(X,fs) :- p(X,fa), domd(X).    p(X,ts) :- p(X,ta), domd(X).
p(X,ts):- p(X,td), domd(X).
q(X,Y,fs) :- q(X,Y,fa), domd(X), domd(Y).
q(X,Y,ts) :- q(X,Y,ta), domd(X), domd(Y).
q(X,Y,ts) :- q(X,Y,td), domd(X), domd(Y).
r(X,Y,fs) :- r(X,Y,fa), domd(X), domd(Y).
r(X,Y,ts) :- r(X,Y,ta), domd(X), domd(Y).
r(X,Y,ts) :- r(X,Y,td), domd(X), domd(Y).
```

Rules for the ICs

```
aux(X) :- q(X,Y,td), not q(X,Y,fa), domd(X), domd(Y).
aux(X) :- q(X,Y,ta), domd(X), domd(Y).
p(X,fa) v q(X,null,ta) :- p(X,ts), not aux(x), not q(X,null,td), domd(X).
q(X,Y,fa) v r(X,Y,ta) :- q(X,Y,ts), r(X,Y,fs), domd(X), domd(Y).
```

Interpretation rules

```
p(X,tss) :- p(X,ta), domd(X).   p(X,tss) :- p(X,td), not p(X,fa), domd(X).
p(X,fss) :- p(X,fa), domd(X).   p(X,fss) :- domd(X), not p(X,td), not p(X,ta).
q(X,Y,tss) :- q(X,Y,ta), domd(X),domd(Y).
q(X,Y,tss) :- q(X,Y,td), not q(X,Y,fa), domd(X), domd(Y).
q(X,Y,fss) :- q(X,Y,fa), domd(X), domd(Y).
q(X,Y,fss) :- not q(X,Y,td), not q(X,Y,ta), domd(X), domd(Y).
r(X,Y,tss) :- r(X,Y,ta), domd(X),domd(Y).
r(X,Y,tss) :- r(X,Y,td), not q(X,Y,fa), domd(X), domd(Y).
r(X,Y,fss) :- r(X,Y,fa), domd(X), domd(Y).
r(X,Y,fss) :- not r(X,Y,td), not r(X,Y,ta), domd(X), domd(Y).
```

Rules for interpreting null values

```
q(X,null,tss) :- q(X,null,ta).
q(X,null,tss) :- q(X,null,td), not q(X,null,fa).
r(X,null,tss) :- r(X,null,ta).
r(X,null,tss) :- r(X,null,td), not r(X,null,fa).
```

Denial constraints

```
:- p(X,ta), p(X,fa).  :- q(X,Y,ta), q(X,Y,fa).  :-r(X,Y,ta),r(X,Y,fa).
```

The models obtained are:

```
{domd(a), d(a,td), p(a,td), p(a,ts), p(a,fs), p(a,fss), p(a,fa),
          q(a,a,fs), r(a,a,fs), q(a,a,fss), r(a,a,fss)}

{domd(a), d(a,td), p(a,td), p(a,ts), p(a,tss), q(a,null,ta),
          q(a,a,fs), r(a,a,fs), q(a,a,fss), r(a,a,fss), q(a,null,tss)},
```

corresponding to the database instances $\emptyset$ and $\{p(a),\ q(a,null)\}$. The program does not consider the inclusion dependency $q(x,y) \rightarrow r(x,y)$ to be violated by the insertion of the tuple $q(a,null)$. If the fact $q(a,null)$ is added to the instance. Then, the clauses `e(a,null,td).` `q(X,null,td) :- e(X,null,td), domd(X).` are part of the program. In this case, the program considers that the instance $\{p(a),\ q(a,null)\}$ does not violate the RIC, what is reflected through its only model

```
{domd(a), d(a,td), e(a,null,td), p(a,td), p(a,ts), q(a,null,td), p(a,tss),
 q(a,a,fs), r(a,a,fs), q(a,a,fss), r(a,a,fss), q(a,null,tss)}.
```
$\square$

If we want to impose the policy of repairing the violation of a RIC just by deleting tuples, then, rule (2) should be changed by

$$p(\bar{x}, \mathbf{f_a}) \leftarrow p(\bar{x}, \mathbf{t}^{\star}),\ \ not\ aux(\bar{x}'),\ \ not\ q(\bar{x}', null, \mathbf{t_d}),$$

saying that if the RIC is violated, then the fact $p(\bar{a})$ that produces such violation must be deleted.

Notice that in this section we have been departing from the definition of repair given in section 2, in the sense that repairs now are obtained by deletion of tuples or insertion of null values only, the usual ways to maintain RICs. However, if the instance is $\{p(\bar{a})\}$ and IC contains only $p(\bar{x}) \rightarrow \exists y q(\bar{x}, y)$, then $\{p(\bar{a}), q(\bar{a}, b)\}$, with $b \in D$, will not be obtained as a repair, because it will not be captured by the program.

If we insist in keeping the original definition of repair, i.e. allowing $\{p(\bar{a}), q(\bar{a}, b)\}$ to be a repair for every element $b \in D$, clause (2) could be replaced by:

$$p(\bar{x}, \mathbf{f_a}) \vee q(\bar{x}', y, \mathbf{t_a}) \leftarrow p(\bar{x}, \mathbf{t}^{\star}),\ \ not\ aux(\bar{x}'),\ \ not\ q(\bar{x}', null, \mathbf{t_d}), choice(\bar{x}', y). \tag{5}$$

where $choice(\bar{X}, \bar{Y})$ is the static non-deterministic choice operator [22] that selects one value for attribute tuple $\bar{Y}$ for each value of the attribute tuple $\bar{X}$. In equation (5), $choice(\bar{x}', y)$ selects one value from the domain. Then, this rule forces the one to one correspondence between stable models and repairs.

6 Conclusions

We have presented a general treatment of consistent query answering for first order queries and ICs. In doing so, we have also shown how to specify database repairs by means of classical disjunctive logic programs with stable model semantics. Those programs have annotations as new arguments. In consequence, consistent query answers can be obtained by "running" a query program together with the specification program. Finally, we showed how to run the programs using the *DLV* system. Our treatment of referential ICs considerably extends what has been sketched in [4, 23].

The problem of consistent query answering was explicitly presented in [2], where also the notions of repair and consistent answer were formally defined. In addition, a methodology for consistent query answering based on a rewriting of the original query was developed (and further investigated and implemented in [13]). Basically, if we want the consistent answers to a FO query expressed in, say SQL2, a new query in SQL2 can be computed, such that its usual answers from the database are the consistent answers to the original query. That methodology has a polynomial data complexity, and that is the reason why it works for some restricted classes of FO ICs and queries, basically for non existentially quantified conjunctive queries [1]. Actually, in [14] it is shown that the problem of CQA is coNP-complete for simple functional dependencies and existential queries. Furthermore, in [8], the problem of CQA is formulated as a problem of non-monotonic reasoning, more precisely of minimal entailment, whose complexity, even in the propositional case, can be Π_2^P-complete [17].

Under those circumstances, it makes sense to apply techniques from logic programming, given its success in formalizing and implementing complex nom-monotonic reasoning tasks [7]. The problem then is to come up with the best logic programming specification and the best way to use them, so that the computational complexity involved does not go beyond the theoretical lower bound. Consistent query answering from relational databases is a new and natural application domain for logic programs, and answer set programming, in particular.

6.1 Implementation issues

Implementation and applications are important directions of research. The logic programming environment will interact with a DBMS, where the inconsistent DB will be stored. As much of the computation as possible should be pushed into the DBMS instead of doing it at the logic programming level. Furthermore, whenever possible, materialization of negative (absent) data should be avoided.

The problem of developing query evaluation mechanisms from disjunctive logic programs that are guided by the query, most likely containing free variables and then expecting a set of answers, like magic sets [1], deserves more attention from the logic programming and database communities. The current alternative relies on finding those ground query atoms that belong to all the stable models once they have been computed via a ground instantiation of the original program. In [18] intelligent grounding strategies for pruning in advance the instantiated program have been explored and incorporated into *DLV*. It would be interesting to explore to what extent the program can be pruned from irrelevant rules and subgoals using information obtained by querying the database.

As shown in [6], there are classes of ICs for which the intersection of the stable models of the repair program coincides with the well-founded semantics, which can be computed more efficiently than the stable model semantics. It could be possible to take advantage of this efficient "core" computation for consistent query answering if ways of modularizing or splitting the whole computation into a core part and a query specific part are found. Such cases were identified in [5] for FDs and aggregation queries.

The logic programs could be optimized in several senses. In some cases, the resulting programs turn out to be "head cycle free" (HCF) [9]. Basically, a program is HCF if there are no cycles in the associated graph that shows an arrow from a predicate p to a predicate q if there is a rule where q appears in the disjunction in the head and p appears positive in the body. Example 8 shows a HCF program.

HCF programs can be transformed into non disjunctive normal programs, that have better complexity properties [26]. Such transformations can be justified or discarded on the basis of a careful analysis of the intrinsic complexity of consistent query answering [14]. If the original program can be transformed into a normal program, then also other efficient implementations could be used for query evaluation, e.g. *XSB* [29], that has been already successfully applied in the context of consistent query answering via query transformation, with non-existentially quantified conjunctive queries [13].

6.2 Related work

In [23], a general methodology based on disjunctive logic programs with stable model semantics is used for specifying database repairs wrt universal ICs. In their approach, preferences between repairs can be specified. The program is given through a schema for rule generation.

Independently, in [4] a specification of database repairs for binary universal ICs by means of disjunctive logic programs with a stable model semantics was presented. Those programs contained both "triggering" rules and "stabilizing" rules. The former trigger local, one-step, changes, and the latter stabilize the chain of local changes in a state where all the ICs hold. The same rules, among others, are generated by the schema in [23].

The programs presented here also work for the whole class of universal ICs, but they are much simpler and shorter than those presented in [23, 4]. Actually, the schema presented in [23] and the extended methodology sketched in [4], both generate an exponential number of rules in terms of the number of ICs and literals in them. Instead, in the present work, due to the simplicity of the program, that takes full advantage of the relationship between the annotations, a linear number of rules is generated.

There are several similarities between our approach to consistency handling and those followed by the belief revision/update community. Database repairs coincide with revised models defined by Winslett in [30]. The treatment in [30] is mainly propositional, but a preliminary extension to first order knowledge bases can be found in [15]. 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. Comparing our framework with that of belief revision, we have an empty domain theory, one model: the database instance, and a revision by a set of ICs. 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, possibly posing a modified query. If this is not possible, we look for methodologies for representing and querying simultaneously and implicitly all the repairs of the database. Furthermore, we work in a fully first-order framework.

Another approach to database repairs based on logic programming semantics consists of the *revision programs* [28]. The rules in those programs explicitly declare 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 $b_1, \ldots, b_m$. Also a declarative, stable model semantics is given 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.

In [11, 25] paraconsistent and annotated logic programs, with non classical semantics, are introduced. However, in [16] some transformation methodologies for paraconsistent logic programs [11] are shown that allow assigning to them extensions of classical semantics. Our programs have a stable model semantics.

Acknowledgments: Work funded by DIPUC, FONDECYT Grant 1000593, Carleton University Start-Up Grant 9364-01, NSERC Grant 250279-02. We are grateful to Alberto Mendelzon, Marcelo Arenas, and Nicola Leone for useful conversations.

References

1. Abiteboul, S.; Hull, R. and Vianu, V. "Foundations of Databases". Addison-Wesley, 1995.
2. Arenas, M.; Bertossi, L. and Chomicki, J. "Consistent Query Answers in Inconsistent Databases". In *Proc. ACM Symposium on Principles of Database Systems (ACM PODS'99)*, 1999, pp. 68–79.
3. Arenas, M.; Bertossi, L. and Kifer, M. "Applications of Annotated Predicate Calculus to Querying Inconsistent Databases". In *'Computational Logic - CL2000' Stream: 6th International Conference on Rules and Objects in Databases (DOOD'2000).* Springer Lecture Notes in Artificial Intelligence 1861, 2000, pp. 926–941.
4. Arenas, M.; Bertossi, L. and Chomicki, J. "Specifying and Querying Database Repairs using Logic Programs with Exceptions". In *Flexible Query Answering Systems. Recent Developments*, H.L. Larsen, J. Kacprzyk, S. Zadrozny, H. Christiansen (eds.), Springer, 2000, pp. 27–41.
5. Arenas, M.; Bertossi, L. and Chomicki, J. Scalar Aggregation in FD-Inconsistent Databases. In *Database Theory - ICDT 2001*, Springer, LNCS 1973, 2001, pp. 39 – 53.
6. Arenas, M.; Bertossi, L. and Chomicki, J. Answer Sets for Consistent Query Answers. Submitted in 2001. (CoRR paper cs.DB/0207094)
7. Baral, Ch. Knowledge Representation, Reasoning and Declarative Problem Solving with Answer Sets. Cambridge University Press. To appear.
8. Barcelo, P. and Bertossi, L. Repairing Databases with Annotated Predicate Logic. In *Proc. Nineth International Workshop on Non-Monotonic Reasoning (NMR'2002), Special session: Changing and Integrating Information: From Theory to Practice*, S. Benferhat and E. Giunchiglia (eds.), 2002, pp. 160–170.
9. Ben-Eliyahu, R. and Dechter, R. "Propositional Semantics for Disjunctive Logic Programs". *Annals of Mathematics in Artificial Intelligence*, 1994, 12:53-87.
10. Bertossi, L., Chomicki, J., Cortes, A. and Gutierrez, C. "Consistent Answers from Integrated Data Sources". In 'Flexible Query Answering Systems', Proc. of the 5th International Conference, FQAS 2002. T. Andreasen, A. Motro, H. Christiansen, H. L. Larsen (eds.). Springer LNAI 2522, 2002, pp. 71–85.
11. Blair, H.A. and Subrahmanian, V.S. "Paraconsistent Logic Programming". *Theoretical Computer Science,* 1989, 68:135-154.
12. Buccafurri, F.; Leone, N. and Rullo, P. "Enhancing Disjunctive Datalog by Constraints". *IEEE Transactions on Knowledge and Data Engineering*, 2000, 12(5):845-860.

13. Celle, A. and Bertossi, L. "Querying Inconsistent Databases: Algorithms and Implementation". In 'Computational Logic - CL 2000', J. Lloyd et al. (eds.). Stream: 6th International Conference on Rules and Objects in Databases (DOOD'2000). Springer Lecture Notes in Artificial Intelligence 1861, 2000, pp. 942–956.
14. Chomicki, J. and Marcinkowski, J. "On the Computational Complexity of Consistent Query Answers". Submitted in 2002 (CoRR paper cs.DB/0204010).
15. Chou, T. and Winslett, M. A Model-Based Belief Revision System. *Journal of Automated Reasoning*, 1994, 12:157–208.
16. Damasio, C. V. and Pereira, L.M. "A Survey on Paraconsistent Semantics for Extended Logic Programas". In *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, Vol. 2, D.M. Gabbay and Ph. Smets (eds.), Kluwer Academic Publishers, 1998, pp. 241–320.
17. Eiter, T. and Gottlob, G. Propositional Circumscription and Extended Closed World Assumption are Π_2^p-complete. Theoretical Computer Science, 1993, 114, pp. 231-245.
18. Eiter, T., Leone, N., Mateis, C., Pfeifer, G. and Scarcello, F. "A Deductive System for Non-Monotonic Reasoning". Proc. LPNMR'97, Springer LNAI 1265, 1997, pp. 364–375.
19. Eiter, T.; Faber, W.; Leone, N. and Pfeifer, G. "Declarative Problem-Solving in DLV". In *Logic-Based Artificial Intelligence*, J. Minker (ed.), Kluwer, 2000, pp. 79–103.
20. Gelfond, M. and Lifschitz, V. "The Stable Model Semantics for Logic Programming". In *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, R. A. Kowalski and K. A. Bowen (eds.), MIT Press, 1988, pp. 1070–1080.
21. Gelfond, M. and Lifschitz, V. "Classical Negation in Logic Programs and Disjunctive Databases". *New Generation Computing*, 1991, 9:365–385.
22. Giannotti, F.; Greco, S.; Sacca, D. and Zaniolo, C. Programming with Nondeterminism in Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 1997, 19(3-4).
23. Greco, G.; Greco, S. and Zumpano, E. "A Logic Programming Approach to the Integration, Repairing and Querying of Inconsistent Databases". In *Proc. 17th International Conference on Logic Programming, ICLP'01*, Ph. Codognet (ed.), LNCS 2237, Springer, 2001, pp. 348–364.
24. Kifer, M. and Lozinskii, E.L. "A Logic for Reasoning with Inconsistency". *Journal of Automated reasoning*, 1992, 9(2):179-215.
25. Kifer, M. and Subrahmanian, V.S. "Theory of Generalized Annotated Logic Programming and its Applications". *Journal of Logic Programming*, 1992, 12(4):335-368.
26. Leone, N.; Rullo, P. and Scarcello, F. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation. *Information and Computation*, 1997, 135(2):69-112.
27. Lloyd, J.W. "Foundations of Logic Programming". Springer Verlag, 1987.
28. Marek, V.W. and Truszczynski, M. "Revision Programming". *Theoretical Computer Science*, 1998, 190(2):241–277.
29. Sagonas, K.F.; Swift, T. and Warren, D.S. XSB as an Efficient Deductive Database Engine. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, ACM Press, 1994, pp. 442-453.
30. Winslett, M. Reasoning about Action using a Possible Models Approach. In *Proc. Seventh National Conference on Artificial Intelligence (AAAI'88)*, 1988, pp. 89–93.