Consistent Query Answering under Inclusion Dependencies

Loreto Bravo and **Leopoldo Bertossi**
Carleton University
School of Computer Science
Ottawa, Canada.
{lbravo,bertossi}@scs.carleton.ca

Abstract

For several reasons a database may not satisfy certain integrity constraints (ICs), for example, when it is the result of integrating several independent data sources. However, most likely most of the information in it is still consistent with the ICs; and could be retrieved when queries are answered. Consistent answers with respect to a set of ICs have been characterized as answers that can be obtained from every possible minimal repair of the database. In this paper we show and analyze how specify those repairs using disjunctive logic program with a stable model semantics in the presence of referential ICs. In this case, repairs are obtained by introduction of null values that do not propagate through other constraints, which makes the problem of consistent query answering decidable. We also present results about cases where the implementation of consistent query answering can be made more efficient due to the fact that the program can be simplified into a non-disjunctive program. Finally, we discuss several research issues around the implementation of system for retrieving consistent answers to queries from a DBMS.

1 Introduction

In databases, integrity constraints (ICs) capture the semantics of the application domain and help maintain the correspondence between this domain and its model provided by the database when updates on the database are performed. However, commercial database management systems (DBMSs) provide limited autonomic support to database maintenance, that is, to the process of keeping the database contents consistent with respect to certain ICs. Except for very restricted classes of integrity constraints that can be internally handled if the user has declared them together with the database schema; in general integrity constraints are maintained by means of user-defined triggers or mechanisms specified at the application level.

There are several reasons for a database to be or become inconsistent with respect to a given set of integrity constraints [9]. This could happen in several situations:

- The most common case is a DBMS that does not have a mechanism to maintain the satisfaction of certain ICs. The available DBMSs are able to maintain by themselves important classes of ICs, but not, e.g., the full class of first-order ICs.
- When data of different sources are being integrated, either virtually or under a materialized approach. In this case, even if the independent data sources are consistent with respect to certain ICs, the global integrated system might be inconsistent with respect to other global ICs [11, 12].
- If new constraints are to be imposed on a pre existing database, i.e., legacy data.
- Soft or user constraints that might be considered only when queries are answered, but without being enforced by the system.

Copyright © 2004 Loreto Bravo and Leopoldo Bertossi. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

In several cases it can be difficult, impossible or undesirable to repair the database in order to restore consistency [9]. The process may be too expensive; useful data may be lost; or it is not clear how to restore the consistency, for example, if extra information is needed. Furthermore, a user who wants to impose new constraints may have no permission to make changes on the data. In the case of data integration, the access to the sources might be restricted.

In those situations, possibly most of the data is still consistent and can be retrieved when queries are posed to the database. In [2] consistent data is characterized as the data that is invariant under all minimal restorations of consistency; i.e., as data that is present in all minimally repaired versions of the original instance (the *repairs*). In particular, an answer to a query is defined as consistent when it can be obtained as a standard answer to the query from *every possible* repair.

We envision the next DBMSs providing much more flexible and user friendly mechanisms for dealing with semantic constraints. In this direction, the system should allow the user to provide a set of ICs as another input to the query answering process, in such a way that those ICs are taken into account as answers to the query are computed. Those ICs could be entered as a another clause in a query expressed in an enhanced version of SQL, something like

```
SELECT      Name, Salary            (⋆)
FROM        Employee
WHERE       Position = 'manager'
CONSIST/W   FD: Name -> Salary;
```

where, for some reasons, the specified functional dependency (FD), which requests that attribute Name functionally determines attribute Salary, is not been maintained by the DBMS. The returned answers from the database should be only those that are consistent with FD. For example, if the underlying database is

Employee	Name	Salary	Position
	John	*55,000*	*manager*
	Peter	*50,000*	*manager*
	John	*60,000*	*manager*
	Ken	*40,000*	*secretary*

the only (consistent) answer returned by the system would be the tuple *(Peter, 50,000)*. This is because the only minimal repairs of the database are the instances

Employee1	Name	Salary	Position
	John	*55,000*	*manager*
	Peter	*50,000*	*manager*
	Ken	*40,000*	*secretary*

and

Employee2	Name	Salary	Position
	Peter	*50,000*	*manager*
	John	*60,000*	*manager*
	Ken	*40,000*	*secretary*

which are obtained by deleting each time only one of the conflicting tuples; and the only tuple that is an (usual) answer to the query ($\star$) above (but without the consistency clause in the last line) in both repaired instances is *(Peter, 50,000)*.

With the same original database, if now the query is

```
SELECT      Name
From        Employee
WHERE       Position = 'manager'
CONSIST/W   FD: Name -> Salary;
```

the (consistent) answers are *(John)*, *(Peter)*, because these two names are returned as usual answers in both repairs.

We can see in this example that computing consistent query answers is different from data cleaning. We do not get rid of the tuples in the original database that participate in a violation of integrity constraints. In particular, in this case we do not lose the information about the existence of an employee named John. We can see that in consistent query answering (CQA), we could see (some of) the ICs as constraints on the query answers rather than on states of the database.

In [2, 14, 25, 3, 5], some mechanisms have been developed for CQA, that is, for retrieving consistent answer when queries are posed to an inconsistent database. All those mechanisms, in different degrees, work only with the original, inconsistent database, without restoring its consistency. Ideally, inconsistencies are solved at query time and the query is posed to the original database. For example, the (consistent) answers to the query ($\star$) can be obtained posing as a standard SQL query the following rewriting of ($\star$)

```
SELECT      Name, Salary
FROM        Employee
WHERE       Position = 'manager'
            AND NOT EXISTS (
  SELECT *
  FROM   Employee E
  WHERE  E.Name = Name AND
         E.Salary <> Salary);
```

which retrieves those employees, with their salaries, for which there is not other employee with the same name, but different salary. The usual answers to this query from the original database will be the consistent answers to query ($\star$). No repair is needed to answer this query. Unfortunately, such first-order query rewriting based methodology provably works only for restricted classes of queries and constraints [6].

Since mechanisms that compute the consistent answers at query time without calculating the repairs [2, 14, 16] are restricted to some very limited classes of queries and constraints, more general methodologies, like the one we present in this paper, require more expressive languages to formulate the rewritings of the original queries [6]. Sometimes it is necessary to use Datalog extended with non stratified negation and disjunctive heads [1, 20]. In those cases, although the above mentioned repairs are intended to be an auxiliary concept to define the right semantics for consistent query answers, they become also an auxiliary intermediate computational step that, for complexity reasons, has to be reduced to a minimum.

In [5, 6] an algorithm is presented that deals with more general class of queries and constraints, e.g., all universal ICs and first-order queries. It is based on specifying the repairs of the database by using disjunctive logic programs with stable model semantics [20]. The complexity of this approach is higher than the complexity of the restricted ones, but the reason is that it matches the intrinsic complexity of the problem of computing consistent answers, which is provably a hard computational problem [15, 13].

In this paper we extend the methodology presented in [5, 6] in order to handle referential integrity constraints via introduction of null values that do not propagate through other ICs. The extended methodology is investigated and optimized, which allows to obtain lower complexity for some classes of ICs.

2 Repairs and Consistent Answers

We will consider a fixed relational schema $\Sigma = (\mathcal{U}, \mathcal{R} \cup \mathcal{B})$ where $\mathcal{U}$ is the possibly infinite database domain, $\mathcal{R}$ is a fixed set of database predicates, and $\mathcal{B}$ is a fixed set of built-in predicates.

A database instance can be seen as a finite collection D of ground atoms of the form $P(c_1, ..., c_n)$, where P is a predicate in $\mathcal{R}$ and $c_1, ..., c_n$ are constants in $\mathcal{U}$. Built-in predicates have a fixed and same extension in every database instance, not subject to changes.

In the following we will express integrity constraints and queries in the first-order language of relational calculus, and the latter sometimes as Datalog rules [1]. Database relations will be represented as sets of ground atoms, and not as tables as above.

A *universal integrity constraint* is a any first-order sentence that is logically equivalent to a sentence of the form

$$\bar{\forall}(\bigvee_{i=1}^{m} \neg P_i(\bar{x}_i) \vee \bigvee_{j=1}^{n} Q_j(\bar{y}_j) \vee \varphi), \qquad (1)$$

where $\bar{\forall}$ is a prefix of universal quantifiers, $P_i, Q_j \in \mathcal{R}$, and φ is a formula containing built-in atoms from $\mathcal{B}$ only. Notice that (1) is logically equivalent to

$$\bar{\forall}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \ \rightarrow \ \bigvee_{j=1}^{n} Q_j(\bar{y}_j) \vee \varphi). \qquad (2)$$

Example 1 For a database schema $\{Emp(id, dept),\ People(id)\}$ some universal ICs can be defined, for example, the functional dependency (FD) *Emp*: $id \rightarrow dept$ can be expressed by $\forall id\ dept_1\ dept_2\ (Emp(id, dept_1) \wedge Emp(id, dept_2) \rightarrow dept_1 = dept_2)$; and the full inclusion dependency (IND) $Emp[id] \subseteq People[id]$, by $\forall\ id\ dept\ (Emp(id, dept) \rightarrow People(id))$. We can see that the common universal ICs found in database praxis do not need the disjunction in the RHS of (2). □

A *referential integrity constraint* (RIC) is a sentence of the form

$$\forall \bar{x}\ (P(\bar{x}) \rightarrow \exists \bar{y}\ Q(\bar{x}', \bar{y})), \qquad (3)$$

where $\bar{x}' \subseteq \bar{x}$ and $P, Q \in \mathcal{R}$.

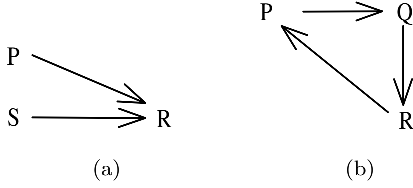

Figure 1: Directed graphs for Example 3.

Example 2 For a database schema $\{Emp(id, dept),\ People(id, name)\}$, in order to represent the IND $Emp[id] \subseteq People[id]$ that states that employees are people, we use the RIC: $\forall id\ dept\ (Emp(id, dept) \rightarrow \exists name\ \ People(id, name))$. Here $\bar{x} = (id, dept)$, $\bar{x}' = (id)$, and $\bar{y} = (name)$. □

These classes of ICs include those most common in the database praxis. We assume we have a fixed set IC of ICs that is logically consistent in the sense that is it is possible to find a database that satisfies them.

A set of RICs is said to be *acyclic* if there are no cycles in the directed graph whose vertices correspond to the relations in $\mathcal{R}$, and an edge from P to R corresponds to a RIC of the form (3).

Example 3 The RICs $\forall x(P(x) \rightarrow \exists y R(x, y))$ and $\forall x(S(x) \rightarrow \exists y R(x, y))$ are *acyclic* since there are no cycles in the directed graph as shown in Figure 1(a). On the other hand, the set of RICs: $\{\forall xz(P(x, z) \rightarrow \exists y Q(x, y))$, $\forall xy(Q(x, y) \rightarrow \exists z R(z, x))$, $\forall xy(R(x, y) \rightarrow \exists z P(x, z))\}$ is cyclic, as shown in Figure 1(b). □

A database instance D is inconsistent if it does not satisfy a given set IC of ICs. In the absence of null values it is clear when this happens, but if they are present or allowed in D, they should be treated as a special constant. Their presence in a tuple means that there are unknown values for the corresponding attributes; i.e., we have incomplete information. Since we do not have precise information about them, we will consider that no inconsistencies arise due to their presence. This leads to the following definition of consistency in the presence of the null value *null*:

Definition 1 [6] For a database instance D, whose domain $\mathcal{U}$ may contain the constant *null*, and a set of integrity constraints $IC = IC_U \cup IC_R$, where IC_U is a set of universal integrity constraints and IC_R is a set of referential integrity constraints, we say that D satisfies IC iff:

1. For each sentence in IC_U of the form $\overline{\forall}\varphi$, where $\bar{\forall}$ is a prefix of universal quantifiers and φ is a a quantifier-free formula, $\varphi[\bar{a}] \in D$ for every ground tuple $\bar{a}$ of elements in $(\mathcal{U} - \{null\})$, and
2. For each sentence in IC_R of the form (3), if $P(\bar{a}) \in D$, with $\bar{a}$ a ground tuple of elements in $(\mathcal{U} - \{null\})$, there exists a tuple $\bar{b}$ of constants in $\mathcal{U}$ for which $Q(\bar{a}', \bar{b}) \in D$. □

Intuitively, this means that a universal IC holds if it is satisfied by non-null values, and a RIC is satisfied considering only non-null values for universally quantified variables and any value for existentially quantified variables.

Example 4 Given a universal IC $\forall xy(P(x, y) \rightarrow R(x, y))$ and a RIC $\forall x(T(x) \rightarrow \exists y P(x, y))$, the database instance $D_1 = \{P(a, d),\ R(a, d),\ T(a),\ T(b),\ P(b, null)\}$ is consistent. The universal constraint is satisfied even in the presence of $P(b, null)$ since the incomplete information does not generate inconsistencies. □

If a database D is inconsistent with respect to a set of constraints IC, a *repair* of D is a new database with the same schema as D, that satisfies IC, and minimally differs from the original database under set inclusion of tuples. These repairs can be obtained from the original repair by adding or deleting tuples [2].

Example 5 Given a database with two tables and one tuple each: $\{P(a, b), R(c, e)\}$, and the universal IC $\forall xy(P(x, y) \rightarrow R(x, y))$; there are two ways of minimally repairing the database: add the tuple (a, b) to table R or delete (a, b) from table P, i.e., the repairs are $D_1 = \{P(a, b), R(c, e)R(a, b)\}$ and $D_2 = \{R(c, e)\}$. The database instance $D_3 = \{P(a, b), R(c, e), R(a, b), P(e, d)\}$ satisfies the ICs, but is not a repair because it unnecessarily adds the tuple $P(e, d)$. □

Example 5 shows how the repairs for universal ICs can be obtained. For RICs the process is different because of the presence of existential variables.

Example 6 Given a database $\{T(a)\}$ and the ICs $\forall x(T(x) \rightarrow \exists y P(x,y))$. One way of repairing the database is by deleting the tuple $T(a)$, corresponding to repair $D_1 = \{\}$. Another way would be to add a tuple $P(a,d)$ where d is any value in the database universe. In latter case, we would have as many repairs as elements in the domain. Instead of this second alternative, we will consider only one possible insertion based repair: $D_2 = \{T(a), P(a, null)\}$. The null value in the second repair represents the fact that we know that there is a tuple in P with first argument a, but the second value is unknown. □

In order to formalize the concept of minimal repair, the distance between databases is defined as follows:

Definition 2 [2] Let D, D' be database instances over the same schema and domain. The *distance*, $\Delta(D, D')$, between D and D' is the symmetric difference $\Delta(D, D') = (D \setminus D') \cup (D' \setminus D)$. □

Example 7 Consider the databases D_1, D_2 and D_3 in example 5. The distances between each of them and the original database are $\Delta(D, D_1) = \{R(a,b)\}$, $\Delta(D, D_2) = \{P(a,b)\}$ and $\Delta(D, D_3) = \{R(a,b), P(e,d)\}$. The elements in each Δ correspond to the elements added into or deleted from D to obtain D_i. □

In order to determine which databases are closer to the original one when repairing it, we define a partial order:

Definition 3 [6] Let D, D', D'' be database instances over the same schema and domain $\mathcal{U}$. It holds that $D' \leq_D D''$ iff:

1. For every atom $P(\bar{a}) \in \Delta(D, D')$, with $\bar{a} \in (\mathcal{U} - \{null\})$,[1] it holds that $P(\bar{a}) \in \Delta(D, D'')$, and
2. For every atom $Q(\bar{a}', null) \in \Delta(D, D')$, it holds that $Q(\bar{a}', null) \in \Delta(D, D'')$ or $Q(\bar{a}, b) \in \Delta(D, D'')$, for some $\bar{b} \in (\mathcal{U} - \{null\})$. □

Example 8 (example 7 continued) From the distances we confirm that the instances that minimally differ from D are D_1 and D_2. Instance D_3 is not minimal, because $D_1 \leq_D D_3$, with $D_1 \neq D_3$. □

[1]That $\bar{a} \in (\mathcal{U} - \{null\})$ means that each of the elements in tuple $\bar{a}$ belongs to $(\mathcal{U} - \{null\})$.

Using this partial order, we are now in position of formally define the *repairs* of an inconsistent database:

Definition 4 Given a database instance D and a set of universal and referential ICs, IC, a repair of D with respect to IC is a database instance D' over the same schema and domain (plus possibly *null* if it was not in the domain of D), such that D' satisfies IC and D' is $\leq_D$-*minimal* in the class of database instances that satisfy IC. We denote by $Rep(D)$ the set of repairs of D. □

In the absence of null-value based repairs, definitions 3 and 4 coincide with those given in [2], where RICs were not considered. The repairs of violations of universal ICs are obtained by either deleting or adding an atom without *null*. The repairs of violations of referential ICs are obtained by either deleting the atom that is generating the inconsistency or by adding an atom with a *null* value. In particular, if the instance is $\{P(\bar{a})\}$ and IC contains only $\forall \bar{x}(P(\bar{x}) \rightarrow \exists y Q(\bar{x}, y))$, then $\{P(\bar{a}), Q(\bar{a}, null)\}$ will be a repair, but not $\{P(\bar{a}), Q(\bar{a}, b)\}$ for any $b \in \mathcal{U}, b \neq null$. In [3, 5, 13] repairs with values other than null have been considered.

Example 9 Consider the universal IC $\forall xy(P(x,y) \rightarrow R(x,y))$, together with the RIC $\forall x(T(x) \rightarrow \exists y P(x,y))$, and an inconsistent database $D = \{P(a,b), T(c)\}$ with domain $\mathcal{U} = \{a,b,c,u\}$. The repairs of D are:

i	D_i	$\Delta(D, D_i)$
1	$\{P(a,b), R(a,b), T(c), P(c, null)\}$	$\{R(a,b), P(c, null)\}$
2	$\{P(a,b), R(a,b)\}$	$\{T(c), R(a,b)\}$
3	$\{T(c), P(c, null)\}$	$\{P(a,b), P(c, null)\}$
4	$\emptyset$	$\{P(a,b), T(c)\}$

We can see that in the first repair the atom $P(c, null)$ does not propagate through the universal constraint to $R(c, null)$. The instance $D_5 = \{P(a,b), R(a,b), T(c), P(c,a)\}$, where $P(c,a)$ has been introduced in order to satisfy the referential IC, does satisfy IC, but is not a repair because $\Delta(D, D_1) \leq_D \Delta(D, D_5) = \{R(a,b), P(c,a)\}$. □

If a database D is consistent with respect to a set of ICs, then it is its only repair.

Definition 5 [2] Given a database instance D, a set of universal and referential ICs IC, and a first-order query $Q(\bar{x})$, we say that a ground tuple $\bar{t}$ is a *consistent answer* to Q with respect to IC iff for every $D' \in Rep(D)$, D' satisfies Q with variables $\bar{x}$ replaced by $\bar{t}$ (denoted $D' \models Q[\bar{t}]$). □

Example 10 Given the IC $\forall x(T(x) \rightarrow \exists y P(x,y))$, the inconsistent database $D = \{P(a,d),\ R(a,d),\ T(a),\ T(b),\ R(b,e)\}$, and the queries $Q_1 : P(x,y)$ and $Q_2 : \forall xy\ (P(x,y) \rightarrow \exists z R(x,z))$. $Rep(D) = \{\ \{P(a,d),\ R(a,d),\ T(a),\ R(b,e)\},\ \{P(a,d),\ R(a,d),\ T(a),\ T(b),\ R(b,e),\ P(b,null)\}\}$. The consistent answer for Q_1 is $P(a,d)$ since that is the only element of P in both repairs. For Q_2 the consistent answer is *Yes* since the formula is satisfied in both repairs. □

3 Repair Logic Programs

The repairs of a relational database can be specified as stable models of disjunctive logic programs [24]. Once the specification has been given, in order to obtain consistent answers to a, say, first-order query Q, the latter is transformed into a query written as a logic program, which is a standard process [29, 1]. Next, this query program is "run" together with the program that specifies the repairs. This evaluation can be implemented on top of, e.g., *DLV*, a logic programming system that computes according to the stable models semantics [21, 28].

The repair programs introduced in [5, 6] use annotation constants with the intended semantics shown in the table below.

Annot.	Atom	Tuple $P(\bar{a})$ is...
$\mathbf{t_d}$	$P(\bar{a}, \mathbf{t_d})$	a database fact
$\mathbf{f_d}$	$P(\bar{a}, \mathbf{f_d})$	not a database fact
$\mathbf{t_a}$	$P(\bar{a}, \mathbf{t_a})$	advised to be true
$\mathbf{f_a}$	$P(\bar{a}, \mathbf{f_a})$	advised to be false
$\mathbf{t^\star}$	$P(\bar{a}, \mathbf{t^\star})$	true or becomes true
$\mathbf{f^\star}$	$P(\bar{a}, \mathbf{f^\star})$	false or becomes false
$\mathbf{t^{\star\star}}$	$P(\bar{a}, \mathbf{t^{\star\star}})$	true in the repair

The intuitive idea behind these annotations is simple. We can think of each ground atom, say $P(\bar{c})$, in the database as annotated with the constant $\mathbf{t_d}$ in an extra argument. In consequence, we have the atom $P(\bar{c}, \mathbf{t_d})$ as a fact of the program. On the other side, if an atom $P(\bar{c})$ does not belong to the database, we have the fact $P(\bar{c}, \mathbf{f_d})$ in the program[2]. Now, when a violation of an IC happens, which can be expressed as the condition in the body of a program rule, then the disjunctive head of the same rule tells us how to restore consistency by deleting or inserting tuples. These recommendations are captured by means of the constants $\mathbf{t_a}, \mathbf{f_a}$, for making an atom true or false (or inserting or deleting it), resp. For example, if the IC is the full inclusion dependency $\forall x(P(x) \rightarrow Q(x))$, then we could have the program rule

$$P(x, \mathbf{f_a}) \vee Q(x, \mathbf{t_a}) \ \leftarrow\ P(x, \mathbf{t_d}), Q(x, \mathbf{f_d}), \quad (4)$$

which in its body detects a violation of the IC (the tuple $P(x)$ is in the database, but not the tuple $Q(x)$). Its head advises then to either delete $P(x)$ or insert $Q(x)$ in order to restore consistency.

The problem is that there might be another IC, whose satisfaction is being restored by inserting, say $P(e)$, which is obtained by deriving the atom $P(e, \mathbf{t_a})$. If we repair in this way, but the tuple $Q(e)$ is not in the database, then there will be a new violation. The rule (4), with its body written in terms of the tuples in (outside) the original database, cannot be used to keep repairing. That is why we need to pass to intermediate notations $\mathbf{t^\star}, \mathbf{f^\star}$, that can be used to detect violations due to the original database values or to those obtained by the local repair steps. Then, instead of the program rule (4) we use the program rule

$$P(x, \mathbf{f_a}) \vee Q(x, \mathbf{t_a}) \ \leftarrow\ P(x, \mathbf{t^\star}), Q(x, \mathbf{f^\star}). \quad (5)$$

The tuples with annotations $\mathbf{t^\star}$ are those obtained collecting those annotated with $\mathbf{t_d}$ or $\mathbf{t_a}$, which can be expressed by means of a new program rule. Similarly for annotation $\mathbf{f^\star}$. Finally, the atoms that can be found in a repair are those that became annotated with $\mathbf{t_a}$ or were originally annotated with $\mathbf{t_d}$, but did not become annotated with $\mathbf{f_a}$. Again, this can be expressed by program rules.

Definition 6 [6] The *repair program*, $\Pi(D, IC)$, of D with respect to IC contains the following clauses:

[2] Actually, these atoms can be defined by rules of the form $P(\bar{x}, \mathbf{f_d}) \leftarrow dom(\bar{x}),\ not\ P(\bar{x}, \mathbf{t_d})$, where a domain predicate stores the admissible values for the tuples involved. This materialization of the closed world assumption [31] can be avoided, getting rid of annotation $\mathbf{f_d}$. Actually, the program in Definition 6 does not use it.

1. $dom(a)$ for each constant $a \in (\mathcal{U} - \{null\})$[3].
2. Fact $P(\bar{a}, \mathbf{t_d})$ for every $P(\bar{a}) \in D$.
3. For every predicate $P \in \mathcal{R}$, the clauses
$P(\bar{x}, \mathbf{t}^\star) \leftarrow P(\bar{x}, \mathbf{t_d}),\ dom(\bar{x})$.[4]
$P(\bar{x}, \mathbf{t}^\star) \leftarrow P(\bar{x}, \mathbf{t_a}),\ dom(\bar{x})$.
$P(\bar{x}, \mathbf{f}^\star) \leftarrow P(\bar{x}, \mathbf{f_a}),\ dom(\bar{x})$.
$P(\bar{x}, \mathbf{f}^\star) \leftarrow dom(\bar{x}),\ not\ P(\bar{x}, \mathbf{t_d})$.[5]
4. For every global universal IC of form (1) the clause:

$$\bigvee_{i=1}^{n} P_i(\bar{x}_i, \mathbf{f_a}) \vee \bigvee_{j=1}^{m} Q_j(\bar{y}_j, \mathbf{t_a}) \ \leftarrow\ \bigwedge_{i=1}^{n} P_i(\bar{x}_i, \mathbf{t}^\star), \bigwedge_{j=1}^{m} Q_j(\bar{y}_j, \mathbf{f}^\star), dom(\bar{x}),\ \bar{\varphi};$$

where $\bar{x}$ is the tuple of all variables appearing in database atoms in the rule, and $\bar{\varphi}$ is a conjunction of built-ins equivalent to the negation of φ.
5. For every referential IC of form (3) the clauses:

$$P(\bar{x}, \mathbf{f_a}) \vee Q(\bar{x}', null, \mathbf{t_a}) \leftarrow P(\bar{x}, \mathbf{t}^\star), not\ aux(\bar{x}'), not\ Q(\bar{x}', null, \mathbf{t_d}), dom(\bar{x}).$$

$$aux(\bar{x}') \leftarrow Q(\bar{x}', y, \mathbf{t_d}),\ not\ Q(\bar{x}', y, \mathbf{f_a}),\ dom(\bar{x}', y).$$

$$aux(\bar{x}') \leftarrow Q(\bar{x}', y, \mathbf{t_a}),\ dom(\bar{x}', y).$$

6. For every predicate $P \in \mathcal{R}$, the interpretation clauses:
$P(\bar{x}, \mathbf{t}^{\star\star}) \ \leftarrow\ P(\bar{x}, \mathbf{t_a})$.
$P(\bar{x}, \mathbf{t}^{\star\star}) \ \leftarrow\ P(\bar{x}, \mathbf{t_d}),\ not\ P(\bar{x}, \mathbf{f_a})$.
7. For every predicate $P \in \mathcal{R}$, the program denial constraint: $\leftarrow\ P(\bar{a}, \mathbf{f_a}),\ P(\bar{a}, \mathbf{t_a})$. $\Box$

A logic program like this, that contains non stratified negation [1], has a stable model semantics [24]. The stable models of the program are the intended models of the program, and they sanction what is true with respect to the program. In general, a program like this will have several stable models.

Rules in 4. and 5. are the most important ones; they specify how the database is to be repaired when a violation of the IC is detected (in the body, i.e., the RHS, of the rule). The disjunction in the head (LHS) of a rule specifies the alternative ways to repair. An atom annotated with $\mathbf{t_a}$ indicates that there is an advice (what the **a** stands for) to make it true, i.e., to insert it into the database; whereas an atom annotated with $\mathbf{f_a}$ indicates an advice to make it false, i.e., to delete it from the database. The annotation constant $\mathbf{t}^\star$ is used in the bodies to give feedback to the repair rules in case there are interacting ICs. At the end, we are only interested in the atoms annotated with $\mathbf{t}^{\star\star}$ in the stable models of the repair program, since they correspond to the data elements in the repairs.

Definition 7 Let $\mathcal{M}$ be a stable model of program $\Pi(D, IC)$.
(a) The database associated to $\mathcal{M}$ is $D_{\mathcal{M}} = \{P(\bar{a}) \mid P(\bar{a}, \mathbf{t}^{\star\star}) \in \mathcal{M}\}$.
(b) $SM(D)$ is the set of databases associated to (the stable models of) $\Pi(D, IC)$. $\Box$

Example 11 (example 9 continued) The repair program $\Pi(D, IC)$ is the following:
1. $dom(a)$. $dom(b)$. $dom(c)$. $dom(u)$.
2. $P(a, b, \mathbf{t_d})$. $T(c, \mathbf{t_d})$.
3. $P(x, y, \mathbf{t}^\star) \leftarrow P(x, y, \mathbf{t_a}),\ dom(x),\ dom(y)$.
$P(x, y, \mathbf{t}^\star) \leftarrow P(x, y, \mathbf{t_d}),\ dom(x),\ dom(y)$.
(similarly for R and T)
4. $P(x, y, \mathbf{f_a}) \vee R(x, y, \mathbf{t_a}) \leftarrow P(x, y, \mathbf{t}^\star),\ R(x, y, \mathbf{f_a}),\ dom(x),\ dom(y)$.
$P(x, y, \mathbf{f_a}) \vee R(x, y, \mathbf{t_a}) \ \leftarrow\ P(x, y, \mathbf{t}^\star),\ not\ R(x, y, \mathbf{t_d}),\ dom(x),\ dom(y)$.
5. $T(x, \mathbf{f_a}) \vee P(x, null, \mathbf{t_a}) \leftarrow T(x, \mathbf{t}^\star), not\ aux(x), not\ P(x, null, \mathbf{t_d}),\ dom(x)$.
$aux(x) \leftarrow P(x, y, \mathbf{t_d}),\ not\ P(x, y, \mathbf{f_a}),\ dom(x, y)$.
$aux(x) \leftarrow P(x, y, \mathbf{t_a}), dom(x, y)$.
6. $P(x, y, \mathbf{t}^{\star\star}) \leftarrow P(x, y, \mathbf{t_a})$.
$P(x, y, \mathbf{t}^{\star\star}) \leftarrow P(x, y, \mathbf{t_d}),\ not\ P(x, y, \mathbf{f_a})$.
(similarly for R and T)
7. $\leftarrow P(x, y, \mathbf{t_a}), P(x, y, \mathbf{f_a})$. (also for R, T)

Only rules 4. and 5. depend on the ICs. Rules 4. corresponds to the universal ICs. They are obtained by unfolding the annotation $\mathbf{f}^\star$ used in 4. in Definition 6 into its definition given in 3. in the same program. The rules in 5. correspond to the referential IC. These rules say how to repair the inconsistencies. Rules 2. contain the database atoms. Rules 7. are denial program constraints to discard models that contain an atom annotated with both $\mathbf{t_a}$ and $\mathbf{f_a}$. The program has four stable models:
$\mathcal{M}_1 = \{dom(a), dom(b), dom(c), dom(u), P(a, b, \mathbf{t_d}), P(a, b, \mathbf{t}^\star), T(c, \mathbf{t_d}), T(c, \mathbf{t}^\star), aux(a), T(c, \mathbf{f_a}), \underline{P(a, b, \mathbf{t}^{\star\star})}, R(a, b, \mathbf{t_a}), R(a, b, \mathbf{t}^\star), \underline{R(a, b, \mathbf{t}^{\star\star})}\}$
$\mathcal{M}_2 = \{dom(a), dom(b), dom(c), dom(u), P(a, b, \mathbf{t_d}), P(a, b, \mathbf{t}^\star), T(c, \mathbf{t_d}), T(c, \mathbf{t}^\star), aux(a), \underline{T(c, \mathbf{t}^{\star\star})}, P(c, null, \mathbf{t_a}), \underline{P(c, null, \mathbf{t}^{\star\star})}, \underline{P(a, b, \mathbf{t}^{\star\star})}, R(a, b, \mathbf{t_a}), R(a, b, \mathbf{t}^\star), \underline{R(a, b, \mathbf{t}^{\star\star})}\}$
$\mathcal{M}_3 = \{dom(a), dom(b), dom(c), dom(u), P(a, b, \mathbf{t_d}), P(a, b, \mathbf{t}^\star), T(c, \mathbf{t_d}), T(c, \mathbf{t}^\star), aux(a), \underline{T(c, \mathbf{t}^{\star\star})}, P(c, null, \mathbf{t_a}), \underline{P(c, null, \mathbf{t}^{\star\star})}, P(a, b, \mathbf{f_a})\}$

[3] Since we want that atoms with *null* values do not generate inconsistencies we would need to add the literal $x \neq null$ to every rule with the predicate $dom(x)$. To avoid this $dom(null)$ is not included in the program.

[4] If $\bar{x} = (x_1, \ldots, x_n)$, we abbreviate $dom(x_1) \wedge \cdots \wedge dom(x_n)$ with $dom(\bar{x})$.

[5] Actually, as illustrated by Example 11, we can always get rid of annotation $\mathbf{f}^\star$.

$\mathcal{M}_4 = \{dom(a),\ dom(b),\ dom(c), dom(u),\ P(a,\ b,\ \mathbf{t_d}), P(a, b, \mathbf{t}^\star), T(c, \mathbf{t_d}), T(c, \mathbf{t}^\star), aux(a), T(c, \mathbf{f_a}), P(a,\ b,\ \mathbf{f_a})\}$

The databases associated to the program are obtained from the models by selecting the atoms annotated with $\mathbf{t}^{\star\star}$ (the underlined atoms): $D_1 = \{P(a,b), R(a,b)\}$, $D_2 = \{T(c),\ P(c, null),\ P(a,b),\ R(a,b)\}$ and $D_3 = \{T(c), P(c, null)\}$, $D_4 = \emptyset$. These repairs coincide with those obtained in example 9. □

When we have the general class of universal and referential ICs, it holds that for every repair of a database with respect to a set of ICs, there exists a model $\mathcal{M}$ of $\Pi(D, IC)$ such that its database associated is the repair, that is, all the repairs can be obtained from the repair program. In example 11 every stable model of the repair program corresponds to a repair. However, there are cases, c.f. example 12, where the database instance corresponding to a model is not a repair of the original database.

Example 12 The database instance $\{Emp(bill, ann), Emp(paul, john), Emp(john, john)\}$ stores the name of an employee with the one of his/her boss. The cyclic RIC: $\forall xy(Emp(x,y) \rightarrow \exists z Emp(y,z))$ states that each is boss is also an employee. The repairs are: $D_1 = \{Emp(paul, john), Emp(john, john)\}$ and $D_2 = \{Emp(bill, ann), Emp(ann, null), Emp(paul, john), Emp(john, john)\}$.

The repair program $\Pi(D, IC)$ is:

$dom(bill).\ \ dom(ann).\ \ dom(paul).\ \ dom(john).$
$Emp(bill, ann, \mathbf{t_d}).\ \ \ Emp(paul, john, \mathbf{t_d}).$
$Emp(john, john, \mathbf{t_d}).$
$Emp(x,y,\mathbf{t}^\star) \leftarrow Emp(x,y,\mathbf{t_d}),\ dom(x),\ dom(y).$
$Emp(x,y,\mathbf{t}^\star) \leftarrow Emp(x,y,\mathbf{t_a}),\ dom(x),\ dom(y).$
$Emp(x,y,\mathbf{f_a}) \vee Emp(y,null,\mathbf{t_a}) \leftarrow Emp(x,y,\mathbf{t}^\star),$
$\quad not\ Emp(y,null,\mathbf{t_d}),\ not\ aux(y),$
$\quad dom(x),\ dom(y).$
$aux(y) \leftarrow Emp(y,z,\mathbf{t_d}),\ not\ Emp(y,z,\mathbf{f_a}),$
$\quad dom(y),\ dom(z).$
$aux(y) \leftarrow Emp(y,z,\mathbf{t_a}),\ dom(y), dom(z).$
$Emp(x,y,\mathbf{t}^{\star\star}) \leftarrow Emp(x,y,\mathbf{t_d}), not\ Emp(x,y,\mathbf{f_a}).$
$Emp(x,y,\mathbf{t}^{\star\star}) \leftarrow Emp(x,y,\mathbf{t_a}).$
$\leftarrow Emp(x,y,\mathbf{t_a}), Emp(x,y,\mathbf{f_a}).$

The stable models of the program are:

$\mathcal{M}_1 = \{dom(bill),\ dom(ann),\ dom(paul),\ dom(john),\ Emp(bill,\ ann,\ \mathbf{t_d}),\ Emp(bill,\ ann,\ \mathbf{t}^\star),\ Emp(paul,\ john,\ \mathbf{t_d}),\ Emp(paul,\ john,\ \mathbf{t}^\star),\ Emp(john,\ john,\ \mathbf{t_d}),\ Emp(john,\ john,\ \mathbf{t}^\star),\ \underline{Emp(bill, ann, \mathbf{t}^{\star\star})},\ \underline{Emp(paul, john, \mathbf{t}^{\star\star})},\ \underline{Emp(john, john, \mathbf{t}^{\star\star})},\ Emp(ann,\ null,\ \mathbf{t_a}),\ aux(john),\ \underline{Emp(ann, null, \mathbf{t}^{\star\star})},\ aux(bill),\ aux(paul)\}.$

$\mathcal{M}_2 = \{dom(bill),\ dom(ann),\ dom(paul),\ dom(john),\ Emp(bill,\ ann,\ \mathbf{t_d}),\ Emp(bill,\ ann,\ \mathbf{t}^\star),\ Emp(paul,\ john,\ \mathbf{t_d}),\ Emp(paul,\ john,\ \mathbf{t}^\star),\ Emp(john,\ john,\ \mathbf{t_d}),\ Emp(john,\ john,\ \mathbf{t}^\star),\ Emp(bill,\ ann,\ \mathbf{f_a}),\ \underline{Emp(paul, john, \mathbf{t}^{\star\star})},\ \underline{Emp(john, john, \mathbf{t}^{\star\star})},\ aux(john),\ aux(paul)\}.$

$\mathcal{M}_3 = \{dom(bill),\ dom(ann),\ dom(paul),\ dom(john),\ Emp(bill,\ ann,\ \mathbf{t_d}),\ Emp(bill,\ ann,\ \mathbf{t}^\star),\ Emp(paul,\ john,\ \mathbf{t_d}),\ Emp(paul,\ john,\ \mathbf{t}^\star),\ Emp(john,\ john,\ \mathbf{t_d}),\ Emp(john,\ john,\ \mathbf{t}^\star),\ \underline{Emp(bill, ann, \mathbf{t}^{\star\star})},\ Emp(paul,\ john,\ \mathbf{f_a}),\ Emp(john,\ john,\ \mathbf{f_a}),\ Emp(ann,\ null,\ \mathbf{t_a}),\ \underline{Emp(ann, null, \mathbf{t}^{\star\star})},\ aux(bill)\}.$

$\mathcal{M}_4 = \{dom(bill),\ dom(ann),\ dom(paul),\ dom(john),\ Emp(bill,\ ann,\ \mathbf{t_d}),\ Emp(bill,\ ann,\ \mathbf{t}^\star),\ Emp(paul,\ john,\ \mathbf{t_d}),\ Emp(paul,\ john,\ \mathbf{t}^\star),\ Emp(john,\ john,\ \mathbf{t_d}),\ Emp(john,\ john,\ \mathbf{t}^\star),\ Emp(bill,\ ann,\ \mathbf{f_a}),\ Emp(paul,\ john,\ \mathbf{f_a}),\ Emp(john,\ john,\ \mathbf{f_a})\}.$

The databases associated to the first two models correspond to the repairs, but not the last two, which are consistent with the IC, but have unnecessarily deleted $Emp(john, john)$. This happens because this deletion, corresponding to the presence of the atom $Emp(john, john, \mathbf{f_a})$ in the models, stably satisfies the rules

$Emp(john, john, \mathbf{f_a}) \vee Emp(john, null, \mathbf{t_a}) \leftarrow$
$\quad Emp(john, john, \mathbf{t}^\star)\ not\ aux(john),$
$\quad not\ Emp(john, null, \mathbf{t_d}),\ dom(john).$
$aux(john) \leftarrow Emp(john, john, \mathbf{t_d}),$
$\quad not\ Emp(john, john, \mathbf{f_a}),\ dom(john).$
$aux(john) \leftarrow Emp(john, john, \mathbf{t_a}),\ dom(john).$

This happens because there is a cycle that involves $Emp(john, john)$, without this tuple participating in the violation of an IC. These rules are satisfied if $Emp(john, john, \mathbf{f_a})$ belong to the model or not. □

In general, $SM(D) \supseteq Rep(D)$. If (and only if) the RICs are cyclic, the inclusion may be proper. However, all the elements of $SM(D)$ satisfy the ICs.

The repair program computes exactly the repairs for universal and *acyclic* referential ICs; i.e., we have a one-to-one correspondence between the repairs and the databases associated to the models of the repair program, i.e., $SM(D) = Rep(D)$.

Example 13 Consider a database instance $\{P(a,b,b), Q(b,c), Q(a,a)\}$, and the cyclic set of RICs: $\{\forall xyz(P(x,y,z) \rightarrow \exists v Q(y,v)), \forall xy(Q(x,y) \rightarrow \exists uw P(u,w,x))\}$. We would expect two ways to restore the consistency of the database, by deleting $Q(b,c)$ or adding the tuple $P(null, null, a)$, but the databases associated to the stable models of the repair program $\Pi(D, IC)$ are $D_{\mathcal{M}_1} = \{P(a,b,b),\ Q(a,a),\ Q(b,c),\ P(null, null, a)\}$, $D_{\mathcal{M}_2} = \{P(a,b,b),\ Q(b,c)\}$, $D_{\mathcal{M}_3} = \{Q(a,a),\ P(null, null, a)\}$ and $D_{\mathcal{M}_4} = \emptyset$. Only the first two are repairs. □

In summary, we have the following:

- D satisfies the ICs for every $D \in SM(D)$.
- $Rep(D) \subseteq SM(D)$.
- For universal and acyclic referential ICs, $Rep(D) = SM(D)$.

4 Consistent Query Answering

The repair program $\Pi(D, IC)$ computes exactly the repairs of the database for universal and acyclic RICs, and a superset when universal and cyclic RICs are considered. We want to use this specification in order to compute the consistent answers from a database with respect to a set of ICs.

We will first concentrate in the case of universal and acyclic RICs. In this case, in order to compute the consistent answers to a query Q, we need to collect the answers that we receive simultaneously from all the stable models of the program $\Pi(D, IC)$. This can be done by first replacing every atom $P(\bar{x})$ of the query by $P(\bar{x}, \mathbf{t}^{\star\star})$. This will force to apply the query over the atoms that belong to every repair. This new query can be transformed into a query program $\Pi(Q)$ by a standard transformation [29, 1]. If this query program is run in combination with $\Pi(D, IC)$, the consistent answers to the query will be obtained.

Example 14 (example 10 continued) Queries Q_1 and Q_2 transformed into query programs are:

$\Pi(Q_1)$: $Ans(x,y) \leftarrow P(x,y,\mathbf{t}^{\star\star})$.

$\Pi(Q_2)$: $Ans \leftarrow\ not\ aux_2$.

$aux_2 \leftarrow P(x,y,\mathbf{t}^{\star\star}),\ not\ aux_1(x)$.

$aux_1(x) \leftarrow R(x,z,\mathbf{t}^{\star\star})$.

If we run $\Pi(D, IC) \cup \Pi(Q_1)$ we get (only the relevant part is shown): $\mathcal{M}_1 = \{\ldots, P(a,d,\mathbf{t}^{\star\star}),\ R(a,d,\mathbf{t}^{\star\star}),\ T(a,\mathbf{t}^{\star\star}),\ R(b,e,\mathbf{t}^{\star\star}),\ Ans(a,d)\}$, and $\mathcal{M}_2 = \{\ldots, P(a,d,\mathbf{t}^{\star\star}),\ R(a,d,\mathbf{t}^{\star\star}),\ T(a,\mathbf{t}^{\star\star}),\ T(b,\mathbf{t}^{\star\star}),\ R(b,e,\mathbf{t}^{\star\star}),\ P(b,null,\mathbf{t}^{\star\star}),\ Ans(a,d),\ Ans(b,null)\}$. The only Ans tuple in both repairs is (a,d), and therefore it is the only consistent answer to the query.

If we do the same for query Q_2, we get the following: $\mathcal{M}_1 = \{\ldots, P(a,d,\mathbf{t}^{\star\star}), R(a,d,\mathbf{t}^{\star\star}), T(a,\mathbf{t}^{\star\star}), R(b,e,\mathbf{t}^{\star\star}), aux_1(a), aux_1(b), Ans\}$, and $\mathcal{M}_2 = \{\ldots, P(a,d,\mathbf{t}^{\star\star}),\ R(a,d,\mathbf{t}^{\star\star}),\ T(a,\mathbf{t}^{\star\star}),\ T(b,\mathbf{t}^{\star\star}),\ R(b,e,\mathbf{t}^{\star\star}),\ P(b,null,\mathbf{t}^{\star\star}),\ aux_1(a),\ aux_1(b),\ Ans\}$. Since Ans is in both repairs the answer to the boolean query is *Yes*. □

Summarizing, in order to compute consistent answers under universal and acyclic RICs, the repair program has to be run with a query program, which will allow us to extract the answers that are true in all the stable models. We have successfully experimented with CQA based on specification of database repairs using the *DLV* system [21].

For cyclic constraints things are more complex. If repairs of RICs are obtained by adding arbitrary elements of the domain, the problem of consistent query answering becomes undecidable [13]. It is possible to prove that under our null values based repairs, the same problem is decidable. We still have the problem of obtaining undesirable models for the programs, those that do not correspond to repairs. We are currently extending the logic program based specification of repairs with local tests for minimality [30]. Using logic programs with priorities [10, 19] is an alternative, due to the fact that, even though all stable models are minimal, the minimality of those that do not correspond to repairs is related to the auxiliary predicates and not to the database predicates (c.f. example 12). In consequence, giving higher priorities to the latter seems to be the right approach. In [15] repairs of RICs by only tuple deletions are investigated. In this case, the problem becomes decidable, but as complex as the evaluation of disjunctive logic programs under the skeptical stable model semantics [20, 18].

5 Optimizations

Sometimes, the repair programs may be transformed into equivalent non-disjunctive programs, i.e., having the same stable models, and then, also specifying the same repairs. This is the case when the disjunctive programs are head-cycle-free [7] (see below). Non-disjunctive logic programs have lower computational complexity than general disjunctive programs [18, 27].

The *dependency graph* of a ground (or fully instantiated) disjunctive program Π is defined as a directed graph where each literal L in the program (i.e., atom or negation of atom) is a node; and there is an arch from L to L' iff there is a rule in which L appears positive in the body and L' appears in the head. Π is *head-cycle-free* (HCF) iff its dependency graph does not contain directed cycles that go through two literals that belong to the head of the same rule.

A disjunctive program Π is HCF if its ground version is HCF. If this is the case, Π can be transformed into a non-disjunctive normal program $sh(\Pi)$ with the same stable models [7]. The non-disjunctive version is obtained by replacing every disjunctive rule of the form: $\bigvee_{i=1}^{n} P_i(\bar{x}_i) \leftarrow \bigwedge_{j=1}^{m} Q_j(\bar{y}_j)$ by the n rules $P_i(\bar{x}_i) \leftarrow \bigwedge_{j=1}^{m} Q_j(\bar{y}_j) \wedge \bigwedge_{k \neq i}$ *not* $P_k(\bar{x}_k)$, $i = 1, ..., n$.

Transformations of this kind can be justified or discarded on the basis of a careful analysis of the intrinsic complexity of consistent query answering [15]. If the original program can be transformed into a non-disjunctive normal program, then also other efficient implementations could be used for query evaluation, e.g., *XSB* [32], that has been applied in interaction with an IBM DB2 DBMS in the context of consistent query answering via first-order query transformation, but only for non-existentially quantified conjunctive queries and limited classes of universal ICs [14].

In [6] it was proved that for the class of universal ICs, the repair programs are HCF, but there no results were reported on referential ICs. Now, we have been able to identify, on the basis of a general test to be applied to a set of ICs containing acyclic RICs, some useful classes of ICs for which the specification program becomes HCF. For example this is the case when IC only contains *denial constraints*; i.e., formulas of the form (1) without positive literals (e.g., FDs and range constraints fall in this class); plus *acyclic referential integrity constraints*. In consequence, for *acyclic foreign key constraints* the repair program becomes HCF.

Example 15 (example 11 continued) The program is HCF and therefore it can be transformed into a normal program by shifting one by one the literals in the disjunctive head to the body in negated form. In this case, the program $sh(\Pi(D, IC))$ is obtained from $\Pi(D, IC)$ by replacing rules in 4. and 5. by:

4. $P(x, y, \mathbf{f_a}) \leftarrow P(x, y, \mathbf{t}^\star),\ R(x, y, \mathbf{f_a}),\ dom(x),$ $\quad not\ R(x, y, \mathbf{t_a}),\ dom(y).$
$R(x, y, \mathbf{t_a}) \leftarrow P(x, y, \mathbf{t}^\star),\ R(x, y, \mathbf{f_a})\ dom(x),$ $\quad not\ P(x, y, \mathbf{f_a}),\ dom(y).$
$R(x, y, \mathbf{t_a}) \leftarrow P(x, y, \mathbf{t}^\star),\ not\ R(x, y, \mathbf{t_d}),$ $\quad not\ P(x, y, \mathbf{f_a})\ dom(x),\ dom(y).$
$P(x, y, \mathbf{f_a}) \leftarrow P(x, y, \mathbf{t}^\star),\ not\ R(x, y, \mathbf{t_d}),$ $\quad not\ R(x, y, \mathbf{t_a})\ dom(x),\ dom(y).$

5. $P(x, null, \mathbf{t_a}) \leftarrow T(x, \mathbf{t}^\star), not\ aux(x),\ dom(x),$ $\quad not\ T(x, \mathbf{f_a}),\ not\ P(x, null, \mathbf{t_d}).$
$T(x, \mathbf{f_a}) \leftarrow T(x, \mathbf{t}^\star), not\ aux(x),\ dom(x),$ $\quad not\ P(x, null, \mathbf{t_a}),\ not\ P(x, null, \mathbf{t_d}).$
$aux(x) \leftarrow P(x, y, \mathbf{t_d}),\ not\ P(x, y, \mathbf{f_a}).$
$aux(x) \leftarrow P(x, y, \mathbf{t_a}).$

The stable models of this program coincide with those of the original program $\Pi(D, IC)$. □

6 Implementation Issues

We are currently working on the implementation of a system for computing consistent query answers on the basis of the repair programs. The details will be presented somewhere else; however we can discuss here a few general issues.

It is clear that any implementation must optimize several processes that participate in consistent query answering. This is because, query answering from disjunctive logic programs has a rather high complexity [18, 20]. However, by using logic programs we are not exceeding the intrinsic complexity of the problem of consistent query answering. In other words, in the general case, the program evaluation and consistent query answering have the same complexity (actually, they are Π_2^P-complete in data complexity) [15, 13]. What is important is to be able to both identify those cases where the complexity of CQA is lower, and optimize

the program in order to match this lower complexity (as the class identified in section 5, for which the complexity of both consistent query answering and the evaluation of the non-disjunctive program can be brought down to the class *NP*). In a similar spirit, determining classes of ICs and queries for which the lower complexity *well-founded semantics* of logic programs [33] can be used is also interesting.

The interaction of a logic programming system and a DBMS is another and important source of complexity. Evaluation of stable models should be avoided whenever possible, trying to obtain as much direct information from the original database as possible. A first step in this direction would be to detect, when a query is to be consistently answered, if the inconsistencies in the data (if any) are relevant to the query at hand. A *consistency check* could determine if the query can be answered directly from the database or the repair program has to be used.

Building a component which decides if the database is consistent or not is simple: The queries

$$C(\overline{x}) \leftarrow P_1(\overline{x}_1), \ldots, P_m(\overline{x}_m), not\ Q_1(\overline{y}_1), \cdots,\ not\ Q_n(\overline{y}_n), \bar{\varphi},$$

for each universal integrity constraint of the form (1), and

$$C'(\overline{x}) \leftarrow P_1(\overline{x}) \wedge\ not\ aux(\overline{x})$$
$$aux(\overline{x}) \leftarrow Q(\overline{x}', y),$$

for each referential integrity constraint of the form (3), will detect the tuples participating in violations of the ICs. Predicates C and C' can be defined, stored and updated as SQL views in the the database, which would reduce the overhead of recomputing them.

If violations that are relevant to the query are detected (actually detecting the relevant ones is another interesting issue), a generator of the repair program, or the programs themselves, should be called. Notice that the repair programs depend on the ICs and not on the query, so, they can be reutilized. Next, a stable model generator, such as *DLV*, has to be used. Finally, the query can be evaluated by running it with the repair program in the logic programming environment.

Since the logic programming environment has to interact with the DBMS, which stores the facts of the program, it is important to avoid unnecessary data extraction. In this sense, it becomes relevant to determine those tables and portions of them which are involved in violation of ICs and relevant to the query. Some interesting ideas in this direction can be found in [22].

It is important to emphasize that we are not interested in the repairs *per se*. In principle, they are used as auxiliary means to characterize the consistent answers. In [2], for restricted classes of queries and ICs, it was possible to compute consistent answers by query the original database alone. However appealing to, complete or partial, computation of repairs, becomes necessary in other cases. In these circumstances, we must minimize their computations or the amount of data involved in the process. The emphasis should be on query answering and not on the computation of the repairs.

It is a problem that the state of art research and implementations of stable model semantics of logic programs are strong at computing (some) stable models, but not at query answering. Full instantiation of the program should be avoided as much as possible; and the system should allow to pose open queries. In this direction, recent research on *magic sets* techniques for disjunctive logic programs under stable model semantics is encouraging [17, 26]. They would allow to reduce the amount of data participating in query processing.

Another direction worth being explored consists in caching previous results of consistent query answering, trying to reuse them when new queries for consistent answers are received; this would avoid running more than desired a stable model generator/evaluator.

7 Conclusions

In this paper we have presented research results that go in the direction of providing mechanisms, to be implemented as as part of the core of a DBMS, that would allow a user to specify, together with a query, a set of integrity constraints -that are not necessarilly maintained by the DBMS- in such a way that the answers to the queries obtained from the system are consistent with the given semantic constraints.

Our approach uses some techniques from the area knowledge representation. At the current state of this line of research, the methodology provably works for any class of first-order ICs that contains universal constraints and acyclic referential constraints.

The current approach considers null-value based repairs under referential integrity constraints. Null values have a special treatment with respect to satisfaction of ICs, and as a consequence, they do not propagate in the repair process. In [3, 5, 13], repairs of RICs using normal domain values are considered. This, under cyclic sets of RICs, may lead to undecidability of consistent query answering. It would be interesting to study some sort of mixed approach, and also the possibility of limited propagation of null values. This is a direction that requires further investigation.

The general complexity of our approach does not exceed the intrinsic complexity of the problem of obtaining consistent query answers; however, as previously discussed, still much exciting research has to be done in terms of optimizing many aspects of the mechanisms, and implementing them in real DBMSs.

Some of the concepts and techniques developed for consistently querying single relational databases, like those presented here, have found applications in the context of virtual data integration. There, global integrity constraints are not maintained, and answers to global queries that are consistent with those constrains are expected to be returned by the mediator [11, 12].

Connections between consistent query answering, virtual data integration and query answering in peer-to-peer data exchange systems are established in [8]. Query answering from a peer has to consider the data exchange constraints and trust relationships with the other peers in the system.

Consistent query answering seem to have natural connections with the area of data exchange, where the main problem is to transfer data from a source database to a target schema that may be different from the schema of the source. In consequence, mappings have to be specified in order to establish the relationship between the data at the source and the data at the target [23], and the process of data transfer has to respect the formulas that express those mappings. However, there are some differences with CQA that deserve further investigation. For example, it is usually the case that for a given source instance, and in contrast with CQA, there are infinite instances at the target that are "solutions" to the problem. The typical syntactic form of the exchange constraints used to express the mappings causes that any superset of a solution is also a solution, whereas database repairs are always minimal.

On the other side, in data exchange some techniques have have been developed to show that some queries over the target schema are not rewritable as a queries that, over a materialized target instance, give a result that is semantically equivalent with the source [4]. It is possible that some of those techniques could be used to show that the consistent answers to some queries cannot be expressed as a first-order views over the underlying instance.

Acknowledgements: Research supported by NSERC Grant 250279-02, and IBM CAS (Toronto Lab.) and CITO awards. We are grateful to Mauricio Vines for his support with the implementation efforts of consistent query answering, and to Marcelo Arenas and Pablo Barcelo for some useful remarks. We also appreciate technical and stylistic comments by anonymous referees.

About the Authors: Loreto Bravo is a Bachelor in Sciences of Engineering and a Civil Transportation Engineer from the Catholic University of Chile (PUC). She is a graduate student and PhD candidate in Computer Science at Carleton University. She participates in a CITO/IBM-CAS (Toronto) Student Internship Program. Leopoldo Bertossi is a full professor of computer science at Carleton University. Before he was a professor at the Catholic University of Chile (PUC). He has held visiting academic positions at the universities of Toronto, Wisconsin-Milwaukee and Marseille (Luminy), and Technical of Berlin. He is a Faculty Fellow of the IBM Center for Advanced Studies, Toronto Lab.; and a member of the NSERC Computing and Information Sciences Grant Selection Committee (GSC 330), 2002 -2005.

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. Symposium on Principles of Database Systems (PODS 99)*, ACM Press, 1999, pp. 68–79.

[3] Arenas, M., Bertossi, L. and Chomicki, J. Answer Sets for Consistent Query Answers. Theory and Practice of Logic Programming, 2003, 3(4-5), pp. 393-424.

[4] Arenas, M., Barcelo, P., Fagin, R. and Libkin, L. Locally Consistent Transformations and Query Answering in Data Exchange. In *Proc. Symposium on Principles of Database Systems (PODS 04)*, ACM Press, 2004, pp. 229-240.

[5] Barcelo, P. and Bertossi, L. Logic Programs for Querying Inconsistent Databases. In *Proc. Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 03)*. Springer LNCS 2562, 2003, pp. 208–222.

[6] Barcelo, P., Bertossi, L. and Bravo, L. Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets. In 'Semantics of Databases', Springer LNCS 2582, 2003, pp. 1–27.

[7] Ben-Eliyahu, R. and Dechter, R. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 1994, 12:53-87.

[8] Bertossi, L. and Bravo, L. Query Answering in Peer-to-Peer Data Exchange Systems. In *Proc. International Workshop on Peer-to-Peer Computing & DataBases (P2P&DB 04)*. To appear in Springer LNCS. Also in CORR repository under cs.DB/0401015.

[9] Bertossi, L. and Chomicki, J. Query Answering in Inconsistent Databases. In 'Logics for Emerging Applications of Databases', J. Chomicki, G. Saake and R. van der Meyden (eds.), Springer, 2003.

[10] Brewka, G. and Eiter, T. Preferred Answer Sets for Extended Logic Programs. *Artificial Intelligence*, 1999, 109(1-2):297-356.

[11] Bravo, L. and Bertossi, L. Logic Programs for Consistently Querying Data Integration Systems. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI 03)*, Morgan Kaufmann, 2003, pp. 10–15.

[12] Bravo, L. and Bertossi, L. Disjunctive Deductive Databases for Computing Certain and Consistent Answers to Queries from Mediated Data Integration Systems. To appear in *Journal of Applied Logic*.

[13] Cali, A., Lembo, D. and Rosati, R. On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. In *Proc. Symposium on Principles of Database Systems (PODS 03)*, ACM Press, 2003, pp. 260-271.

[14] 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 00). Springer LNAI 1861, 2000, pp. 942–956.

[15] Chomicki, J. and Marcinkowski, J. Minimal-Change Integrity Maintenance Using Tuple Deletions. arXiv.org paper cs.DB/0212004. To appear in *Information and Computation*.

[16] Chomicki, J., Marcinkowski, J. and Staworko, S. Hippo: A System for Computing Consistent Answers to a Class of SQL Queries. In *Advances in Database Technology - EDBT 2004*, Springer LNCS 2992, 2004, pp. 841-844.

[17] Cumbo, C., Faber, W., Greco, G. and Leone, N. Enhancing the Magic-Set Method for Disjunctive Datalog Programs. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 04)*, 2004. To appear.

[18] Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 2001, 33(3): 374–425.

[19] Delgrande, J., Schaub, T. and Tompits, H. A Framework for Compiling Preferences in Logic Programs. *Theory and Practice of Logic Programming*, 2003, 3(2):129-187.

[20] Eiter, T., Gottlob, G. Complexity Aspects of Various Semantics for Disjunctive

Databases. In *Proc. Symposium on Principles of Database Systems (PODS 93)*, ACM Press, 1993, pp. 158-167.

[21] 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.

[22] Eiter, T., Fink, M., Greco, G. and Lembo, D. Efficient Efficient Evaluation of Logic Programs for Querying Data Integration Systems. In *Proc. 19th International Conference on Logic Programming (ICLP 03)*, Springer LNCS 2916, 2003, pp. 163-177.

[23] Fagin, R., Kolaitis, P., Miller, R.J. and Popa, L. Data Exchange: Semantics and Query Answering. In *Proc. International Conference on Database Theory (ICDT 03)*, Springer LNCS 2572, 2003, pp. 207–224..

[24] Gelfond, M. and Lifschitz, V. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 1991, 9:365–385.

[25] Greco, G., Greco, S. and Zumpano, E. A Logic Programming Approach to the Integration, Repairing and Querying of Inconsistent Databases. In *Proc. International Conference on Logic Programming (ICLP 01)*, Springer LNCS 2237, 2001, pp. 348–364.

[26] Greco, G., Greco, S., Trubtsyna, I. and Zumpano, E. Optimization of Bound Disjunctive Queries with Constraints. arXiv.org paper cs.LO/0406013. To appear in *Theory and Practice of Logic Programming.*

[27] Leone, N., Rullo, P. and Scarcello, F. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation. *Information and Computation*, 1997, 135(2):69-112.

[28] Leone, N. et al. The DLV System for Konwledge Representation and Reasoning. arXiv.org paper cs.LO/0211004. To appear in *ACM Transactions on Computational Logic.*

[29] Lloyd, J.W. *Foundations of Logic Programming.* Second ed., Springer-Verlag, 1987.

[30] Niemela, I. Implementing Circumscription Using a Tableau Method. In *Proc. European Conference on Artificial Intelligence (ECAI 96)*, 1996, pp. 80–84.

[31] Reiter, R. On Closed World Databases. In *Logic and Databases*, H. Gallaire and J. Minker (eds.), Plenum Press, 1978, pp. 55-76.

[32] Sagonas, K.F., Swift, T. and Warren, D.S. XSB as an Efficient Deductive Database Engine. In *Proc. International Conference on Management of Data (SIGMOD 94)*, ACM Press, 1994, pp. 442-453.

[33] Van Gelder, A., Ross, K.A., Schlipf, J.S. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *Proc. Symposium on Principles of Database Systems (PODS 88)*, ACM Press, 1988, pp. 221-230.