Maintaining and Restoring Database Consistency with Update Rules

Sofian Maabout*
University of Antwerp (UIA)
maabout@uia.ac.be

Abstract

Maintaining database consistency is one of the major applications of active databases. One of the problems encountered in this case is the non termination of rules execution. Then, the general solution is to undo the modifications and roll back to the initial database which is *assumed* to be consistent. In this paper, we present an algorithm that generates update rule programs from constraints expressions. The semantics of these rules is given by means of a logic program model, namely the well founded model. The derived programs can serve to maintain and restore database consistency. Maintaining database consistency assumes its consistency before user updates are requested while restoring consistency does not consider this hypothesis.

1 Introduction

Active databases use ECA rules (Event, Condition, Action) in order to specify which actions must be triggered when some event and condition are met. Database integrity management is one of the most prominent application fields of active databases. The intuitive idea is that whenever some violating updates arise, if integrity constraints are violated, then the system must achieve some correcting actions in order to restore database consistency. A large amount of research has been done in the integration of active rules in this area, see e.g. [CFPT94, CW90, LML97, ST96, UD92]. We also refer to [AART97, DDS98, ALS95, MT98, TO95] among works that have considered logic programming paradigm in the context of integrity constraints management.

Ceri & Widom[CW90] consider integrity constraints (IC) expressed as SQL queries with possibly aggregates. The derived rules from these IC guarantee database consistency whenever rules execution terminates. Schewe & Thalheim [ST96] characterize a class of IC for which there exists a program of active rules that restores database consistency without invalidating users updates when it is possible, e.g. let $p(a) \Rightarrow q(a)$ be an IC and let $\Delta = \emptyset$ be a database instance where we want to insert $p(a)$. This transition violates the IC. In order to restore consistency either $q(a)$ is inserted or $p(a)$ is deleted. The first alternative preserves the effect of the user transition, thus it is *preferred*. Inserting $p(a)$ is in this spirit, a *repairable* transition because there exists a consistent database state containing $p(a)$. Notice that the transaction

*Author's address: University of Antwerp (UIA). Departement Wiskund en Informatica. Universiteitplein 1. B-2610 Antwerpen (Wilrijk). Belgium

$+p(a); -q(a)$ is not repairable. [ST96] does not address the problem of termination. Ludäscher *et al* [LML97, LM98] consider the particular case of referential IC's as those proposed in SQL92 standard. They give a translation of these IC's into logic programs and show that the well founded semantics [vGRS91] may be used in order to maintain database consistency. They use *Statelog* programs which are Datalog programs where predicates are augmented with a state attribute that may contain the successor function $+1$. Because of this use of functions, the programs in *Statelog* must deal with infinite models. Among the works on the termination of rules execution, we refer to [AWH92, BCP95] and the recent work [BDR98]

In this paper, we propose an algorithm that derives logic rules from declarative expression of IC's. Although we use well founded semantics _which is based on a three valued logic_ we show that the models we obtain are always bi-valued. This paper is an extension of [BM97] where we have considered incomplete databases.

Motivating example. Consider the following IC:

$$c_1 \equiv \forall X : p(X) \Rightarrow q(X)$$
$$c_2 \equiv \forall X : r(X) \Rightarrow \neg q(X)$$

and assume that we have the following update rules

$$\rho_1 : p(X), \neg q(X) \longrightarrow +q(X)$$
$$\rho_2 : r(X), q(X) \longrightarrow -q(X)$$

ρ_1 maintains c_1 and the second c_2. If $p(a)$ and $p(b)$ are both inserted in the empty (coherent) database, then the rules are fired in order to restore consistency. ρ_1 is executed first then ρ_2 becomes executable and after executing it, ρ_1 becomes again executable. Thus, we enter a non terminating rule execution. The generally adopted solution in this case, is to stop rules execution and roll back to the initial state of the database, and since this state is a coherent one, so we have a consistent final state.

In this paper, we try to avoid the consistency of the initial state hypothesis. Doing so, we must find a way to associate active rules to IC's whose execution never enter non terminating loops.

Paper organization. The first section introduces the concepts used throughout the paper. Then, we propose an algorithm that generates a set of update rules from a set of IC's. We show that these programs can maintain database consistency. Then, we refine the algorithm in order to obtain rules that can maintain **and** restore database consistency.

The ability to restore database consistency is quite useful in a context where IC's can be added during the life of the database.

2 Preliminaries

A database Δ is a set of facts. The constraints we consider in this paper are those that can be expressed by denials. A *denial* is a rule of the form

$$p_1(\tilde{X}_1), \ldots, p_m(\tilde{X}_m), \neg q_1(\tilde{Y}_1), \ldots, \neg q_n(\tilde{Y}_n) \longrightarrow \texttt{inc} \tag{1}$$

This rule must be *safe* in a sense that variables appearing in negative literals must also appear in a positive literal. `inc` is a 0-ary predicate, hence a denial can be seen as a boolean query. If $\mathcal{C}$ is a set of such IC's, then a database Δ satisfies $\mathcal{C}$ iff $\Delta \cup \mathcal{C} \not\models \texttt{inc}$ or equivalently, $\Delta \cup \mathcal{C} \models \neg\texttt{inc}$.

Now suppose that there exists $c \in \mathcal{C}$ such that $\Delta \models body(\mathcal{C})$. This means that Δ is inconsistent. To restore consistency, either we insert some tuples in the predicates q_j or we delete some ones from the predicates p_i. Thus, from the denial (1), we may derive one rule among the $m+n$ rules of the form

$$p_1(\tilde{X}_1), \ldots, p_{i-1}(\tilde{X}_{i-1}), p_{i+1}(\tilde{X}_{i+1}), \ldots, p_m(\tilde{X}_m), \neg q_1(\tilde{Y}_1), \ldots, \neg q_n(\tilde{Y}_n) \longrightarrow -p_i(\tilde{X}_i)$$
or
$$p_1(\tilde{X}_1), \ldots, p_m(\tilde{X}_m), \neg q_1(\tilde{Y}_1), \ldots, \neg q_{j-1}(\tilde{Y}_{j-1}), \neg p_{j+1}(\tilde{X}_{j+1}), \ldots \neg q_n(\tilde{Y}_n) \longrightarrow +q_j(\tilde{Y}_j)$$

Example 1 Let us consider the referential IC:

$$employee(X, Y), \neg dept(Y) \longrightarrow \texttt{inc}$$

From this denial, we can derive two update rules

$$employee(X, Y) \longrightarrow +dept(Y) \text{or} \neg dept(Y) \longrightarrow -employee(X, Y)$$

Notice that the second rule is not safe nor range restricted; Y and X do not appear in any positive literal in the body of the rule. But this is not a problem since we consider active domain semantics [AHV95]. The problem of domain independence of update rules has been addressed in [BM97]. □

The example above shows how we can derive update rules from denials. We don't give this straightforward algorithm here. The interested reader may find the details in [BM97] where we considered a more general class of IC's.

Now, we recall some basic definitions we use throughout the paper. We consider a set P of normal (base) predicates and SP the set of signed or update predicates such that $p \in P \Rightarrow \{+p, -p\} \subseteq SP$. The definitions of facts, atoms and literals are the classical ones [CGT90]. $Lit(P)$ (resp. $Lit(SP)$) is the set of literals obtained by using the predicates in P (resp. SP). An update rule ρ is a Datalog rule where $head(\rho)$ is an atom in $Lit(SP)$ and $body(\rho) \cap Lit(SP) = \emptyset$. An update program is a set of update rules. A partial interpretation I is a set of ground literals such that, if $pos(I)$ and $neg(I)$ are respectively the set of positive and negative literals of I, and if $\neg.neg(I) = \{p : \neg p \in neg(I)\}$, then $pos(I) \cap \neg.neg(I) = \emptyset$. Let $\mathcal{A}$ be a set of atoms, then I is *total* with respect to $\mathcal{A}$ if for all $p \in \mathcal{A}$, $\{p, \neg p\} \cap I \neq \emptyset$. I is conflicting if there exists p such that $\{+p, -p\} \subseteq I$.

Definition 1 (Entailment) I entails a ground positive literal p (denoted $I \models p$) if $I \supseteq \{p, \neg -p\}$ or $I \supseteq \{+p\}$. Conversely, $I \models \neg p$ iff $I \supseteq \{\neg p, \neg +p\}$ or $I \supseteq \{-p\}$. I satisfies a ground rule ρ ($I \models \rho$) iff whenever $I \models body(\rho)$, then $Head(\rho) \in I$. $Inst(\mathcal{P})$ is the set of all ground rules obtained from the program $\mathcal{P}$ by replacing all variables by the constants appearing in $\mathcal{P}$. I is a model of $\mathcal{P}$ if I satisfies every rule in $Inst(\mathcal{P})$.

□

Definition 2 (Update) An update δ is a set of ground update literals. A **user update** is a non conflicting set of ground update atoms.

□

Definition 3 (Operator $\mathcal{T}$) Let I be an interpretation and $\mathcal{P}$ an update program.

- $T^{\in}_{\mathcal{P}}(I) = \{head(\rho) \;:\; \rho \in inst(\mathcal{P}) \text{ and } I \models body(\rho)\}$.
- $NF_{\mathcal{P}}(I) = \{\ell \;:\; \forall \rho \in inst(\mathcal{P}), head(\rho) = \ell \Rightarrow I \models \neg body(\rho)\}$.

The operator $\mathcal{T}$ is defined by the composition of these two operators:

$$\mathcal{T}_{\mathcal{P}}(I) = I \cup \mathcal{T}^{\in}_{\mathcal{P}}(I) \cup \neg.NF_{\mathcal{P}}(I)$$

□

It is easy to see that $\mathcal{T}$ is monotone, so the sequence $(\mathcal{T}^n_{\mathcal{P}})_{n\geq 0}$ defined by

$$\begin{array}{rcl} \mathcal{T}^0_{\mathcal{P}}(I) & = & I \\ \mathcal{T}^n_{\mathcal{P}}(I) & = & \mathcal{T}^{n-1}_{\mathcal{P}}(I) \end{array}$$

admits a limit. This limit is called *the update model of* $\mathcal{P}$ and is denoted by $\mathcal{M}_I(\mathcal{P})$ or simply by $\mathcal{M}(\mathcal{P})$ when I is understood.

Intuitively, the update process can be described as follows: given a user update δ, an initial database Δ, and an update program $\mathcal{P}$:

1. The application of δ gives a new state (set of facts) Δ_{user},
2. compute the update model $\mathcal{M}_{\Delta_{user}}(\mathcal{P})$. This contains a set of update literals which we call "derived updates", then
3. the application of the derived updates to Δ_{user} gives a new database Δ_{final}.

Formally, consider Δ, a user update δ and a program $\mathcal{P}$.

- $\Delta_{user} = \Delta \cup \{p \mid +p \in \delta\} - \{p \mid -p \in \delta\}$.

- Let $\mathcal{H}$ be the herbrand base of $\mathcal{P}$. Then, $\texttt{Apply}(\delta, \Delta, \mathcal{P}) = \Delta_{final}$ where
  - if $\mathcal{M}_{\Delta_{user}}(\mathcal{P})$ is total wrt $\mathcal{H}$ and not conflicting then $\Delta_{final} = \{\Delta_{user} \cup \{p \mid +p \in \mathcal{M}_{\Delta_{user}}(\mathcal{P})\} \setminus \{p \mid -p \in \mathcal{M}_{\Delta_{user}}(\mathcal{P})\}$,
  - otherwise $\Delta_{final} = \Delta$.

Example 2 Let $\Delta = \{r, s\}, \delta = \{+p, +q\}$ and $\mathcal{P}$ be the program

$$\begin{array}{rcl} p, r & \longrightarrow & -s \\ q, s & \longrightarrow & -r \end{array}$$

$\Delta_{user} = \{p, q, r, s\}$ and it is embedded into $\mathcal{P}$ by adding the rules

$$\begin{array}{llll} \longrightarrow & p; & \longrightarrow & q \\ \longrightarrow & r; & \longrightarrow & s \end{array}$$

Let's call the new program by $\mathcal{P}_{user}$. Now, let us compute the update model

$$\begin{array}{rcl} T^{\in}_{\mathcal{P}_{user}}(\emptyset) & = & \{p, q, r, s\} \\ NF_{\mathcal{P}_{user}}(\emptyset) & = & \{\neg{-p}, \neg{-q}, \neg{+p}, \neg{+q}, \neg{+r}, \neg{+s}\} \\ I_1 & = & T^{\in}_{\mathcal{P}_{user}}(\emptyset) \bigcup NF_{\mathcal{P}_{user}}(\emptyset) \end{array}$$

$$\begin{array}{rcl} T^{\in}_{\mathcal{P}_{user}}(I_1) & = & \{p, q, r, s\} \\ NF_{\mathcal{P}_{user}}(I_1) & = & \{\neg{-p}, \neg{-q}, \neg{+p}, \neg{+q}, \neg{+r}, \neg{+s}\} \\ I_2 & = & I_1 \end{array}$$

I_2 is the fixpoint thus it is the update model. It is not total because both $-s$ and $-r$ are "unknown". So $\Delta_{final} = \Delta$. One should notice that for example, even if $+p$ does not appear in $\mathcal{P}_{user}$, it is considered as being in the herbrand base. indeed, as soon as p, $+p$ or $-p$ appears in $\mathcal{P}_{user}$, $\{p, +p, -p\}$ is in the "extended herbrand base" of $\mathcal{P}_{user}$.

□

The following proposition characterizes the cases where the derived rules from IC's can maintain database consistency.

Proposition 1 Let $\mathcal{C}$ be a set of constraints and $\mathcal{P}_{\mathcal{C}}$ its associated update program. Then, for each user update δ and initial database Δ, if $\mathcal{M}$ is total and not conflicting then $\mathtt{Apply}(\delta, \Delta, \mathcal{P}_{\mathcal{C}})$ is consistent.

□

From the definition of `Apply`, if $\mathcal{M}$ is not total or is conflicting then $\mathtt{Apply}(\delta, \Delta, \mathcal{P}_{\mathcal{C}}) = \Delta$. Thus, if Δ is not initially consistent, we end up with a non consistent database. So, $\mathcal{P}_C$ can maintain database consistency but in some cases, it cannot restore it.

The fact that the update model of $\mathcal{P}$ can be not total is due to the presence of negative cycles in the dependency graph of the program $\bar{\mathcal{P}}$ defined below.

Definition 4 Let $\mathcal{P}$ be an update program. $\bar{\mathcal{P}}$ is the program obtained from $\mathcal{P}$ by:

1. if $head(\rho) = +p(X)$ then add $\neg{-p(X)}$ into $body(\rho)$,
2. if $head(\rho) = -p(X)$ then add $\neg{+p(X)}$ into $body(\rho)$,
3. replace each $p(X) \in body(\rho)$ by $(p(X) \wedge \neg{-p(X)}) \vee +p(X)$, and
4. replace each $\neg p(X) \in body(\rho)$ by $(\neg p(X) \wedge \neg{+p(X)}) \vee -p(X)$. □

Lemma 1 If $\bar{\mathcal{P}}$ is stratified, then $\forall \delta, \Delta : \mathcal{M}_{\Delta_{user}}(\mathcal{P})$ is total and not conflicting.

□

In fact, we can distinguish between two kinds of negative cycles: negatively recursive programs (NR) and potentially conflicting (PC) programs. NR programs are those containing negative cycles after transformations 3. and 4. in the definition above. PC programs are those containing negative cycles after transformations 1. and 2. These are illustrated by the following examples:

Example 3 Let us consider the following sets of IC's:

$$\mathcal{C}_1 = \left\{ \begin{array}{rcl} p, \neg q & \longrightarrow & \texttt{inc} \\ r, q & \longrightarrow & \texttt{inc} \end{array} \right\} \text{ and } \mathcal{C}_2 = \left\{ \begin{array}{rcl} r, p, q & \longrightarrow & \texttt{inc} \\ \neg s, q, p & \longrightarrow & \texttt{inc} \end{array} \right\}$$

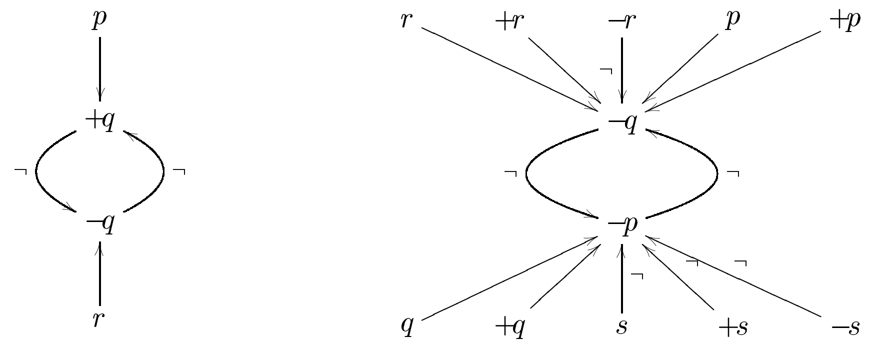
We first derive respectively $\mathcal{P}_{\mathcal{C}_1}$ and $\mathcal{P}_{\mathcal{C}_2}$ such that:

$$\mathcal{P}_{\mathcal{C}_1} = \left\{ \begin{array}{ccc} p & \longrightarrow & +q \\ r & \longrightarrow & -q \end{array} \right\} \text{ et } \mathcal{P}_{\mathcal{C}_2} = \left\{ \begin{array}{lcc} r, p & \longrightarrow & -q \\ \neg s, q & \longrightarrow & -p \end{array} \right\}$$

$\mathcal{P}_{\mathcal{C}_1}$ is a PC program; let $\mathcal{P}_1$ be the program obtained from $\mathcal{P}_{\mathcal{C}_1}$ after transformations 1 and 2. $\mathcal{P}_1$ contains a cycle where $+q$ depends on $\neg -q$ and $-q$ depends on $\neg +q$.
$\mathcal{P}_{\mathcal{C}_2}$ is a NR program because the program $\mathcal{P}_2$ obtained from $\mathcal{P}_{\mathcal{C}_2}$ after transformations 3 and 4 presents a negative cycle where $-q$ depends on $\neg -p$ and $-p$ depends on $\neg -q$.

$$\mathcal{P}_1 = \left\{ \begin{array}{lcc} p, \neg -q & \longrightarrow & +q \\ r, \neg +q & \longrightarrow & -q \end{array} \right\} \quad \text{and} \quad \mathcal{P}_2 = \left\{ \begin{array}{lcc} r, \neg -r, p, \neg -p & \longrightarrow & -q \\ +r, p, \neg -p & \longrightarrow & -q \\ r, \neg -r, +p & \longrightarrow & -q \\ +r, +p & \longrightarrow & -q \\ \neg s, \neg +s, q, \neg -q & \longrightarrow & -p \\ -s, q, \neg -q & \longrightarrow & -p \\ \neg s, \neg +s, +q & \longrightarrow & -p \\ -s, +q & \longrightarrow & -p \end{array} \right\}$$

The dependency graphs of these programs are



□

Now one may wonder whether for each $\mathcal{C}$ there exists $\mathcal{P}_{\mathcal{C}}$ such that $\bar{\mathcal{P}}_{\mathcal{C}}$ is stratified. The following example shows that this is not the case.

Example 4 Consider the functional dependency $X \longrightarrow Y$ expressed by the denial

$$c : p(X, Y), p(X, Y'), Y \neq Y' \longrightarrow \texttt{inc}$$

The predicate p is *repeated* in c. The only update rule we may infer is

$$\rho_c : p(X, Y), Y \neq Y' \longrightarrow -p(X, Y')$$

which is negatively recursive:

$$\bar{\rho_c} = \left\{ \begin{array}{rcl} p(X,Y), \neg{-p}(X,Y), Y \neq Y' & \longrightarrow & -p(X,Y') \\ +p(X,Y), Y \neq Y' & \longrightarrow & -p(X,Y') \end{array} \right.$$

$\square$

Definition 5 Let c be a constraint. p is a *repeated* predicate in c if there exists two literals in c both constructed from p and both are positive or negative. $\square$

In the following, we assume that none of the IC's we consider contain repeated predicates.

An other case where it is not possible to derive programs that guarantee total non conflicting update models is the case where the IC's are themselves inconsistent, i.e. there does not exist an instance Δ that satisfies $\mathcal{C}$. e.g,

$$\mathcal{C} = \left\{ \begin{array}{lrcl} c_1 : & p(X), q(X) & \longrightarrow & \texttt{inc} \\ c_2 : & \neg p(a) & \longrightarrow & \texttt{inc} \\ c_3 : & \neg q(a) & \longrightarrow & \texttt{inc} \end{array} \right\}$$

It is generally admitted that "interesting" practical IC's are those satisfied at least by the empty database. From the particular form of the IC's considered in this paper, a sufficient condition such that the empty database would be coherent is to require that the body of c contains at least one positive literal. c_2 and c_3 above are not satisfied by the empty database. In the following, we suppose that all IC's satisfy this condition.

3 Deriving programs from constraints

In this section, we will first give an algorithm that derives a first program $\mathcal{P}^1_{\mathcal{C}}$ which is not negatively recursive. Then, we present a way to transform $\mathcal{P}^1_{\mathcal{C}}$ into $\mathcal{P}_{\mathcal{C}}$ such that for each Δ and each update δ, $\mathcal{M}_{\Delta_{user}}(\mathcal{P}_{\mathcal{C}})$ is bivalued.

3.1 Non negatively recursive programs

In order to derive non negatively recursive programs, we will use the notion of IC graph.

Definition 6 (Constraints graph) Let $\mathcal{C}$ be a set of constraints. $\mathcal{G}_{\mathcal{C}}$ is a non oriented graph $\langle V, E \rangle$ where V is the set of IC's and $(c_i, c_j) \in E$ labeled with p iff the predicate p appears in both c_i and c_j with the same sign, i.e. positive or negative. If p appears in opposite signs the edge is labeled with $\neg p$.

$\square$

Example 5 Let

$$\mathcal{C} = \left\{ \begin{array}{lll} c_1 : & p(X), q(X) & \longrightarrow \texttt{inc} \\ c_2 : & r(X), \neg q(X) & \longrightarrow \texttt{inc} \end{array} \right\}$$

$\mathcal{G}_\mathcal{C}$ is given below:

$$\begin{tikzcd} c_1 \arrow[rr, dash, "\neg q"] & & c_2 \end{tikzcd}$$

Notice that from $\mathcal{C}$, we can derive a potentially conflicting program i.e. one rule that inserts into Q and another whose action is a deletion from Q.

□

Lemma 2 Let $\mathcal{C}$ be a set of constraints and $\mathcal{G}_\mathcal{C}$ its associated graph. If $\mathcal{G}_\mathcal{C}$ does not contain a negative edge, then each $\mathcal{P}_\mathcal{C}$ is not potentially conflicting.

□

Example 6 Let

$$\mathcal{C} = \left\{ \begin{array}{lll} c_1: & p(X), q(X), r(X) & \longrightarrow \texttt{inc} \\ c_2: & s(X), q(X), \neg u(X) & \longrightarrow \texttt{inc} \\ c_3: & v(X), \neg u(X), r(X) & \longrightarrow \texttt{inc} \end{array} \right\}$$

The corresponding graph is

$$\begin{tikzcd} c_1 \arrow[rr, dash, "q"] \arrow[dr, dash, "r"'] & & c_2 \arrow[dl, dash, "u"] \\ & c_3 & \end{tikzcd}$$

Notice that the graph contains a cycle. Notice also that we can derive a negatively recursive program from $\mathcal{C}$, namely the program:

$$\mathcal{P}_\mathcal{C} = \left\{ \begin{array}{lll} p(X), r(X) & \longrightarrow & -q(X) \\ s(X), q(X) & \longrightarrow & +u(X) \\ v(X), \neg u(X) & \longrightarrow & -r(X) \end{array} \right\}$$

The dependency graph of $\bar{\mathcal{P}}_\mathcal{C}$ contains the negative cycle

$$\begin{tikzcd} -r \arrow[dr, bend right, "\neg"'] & & +u \arrow[ll, bend right, "\neg"'] \\ & -q \arrow[ur, bend right, "\neg"'] & \end{tikzcd}$$

□

Lemma 3 If $\mathcal{G}_\mathcal{C}$ does not contain cycles, then $\mathcal{P}_\mathcal{C}$ is negatively recursive only if there are constraints with repeated predicates.

□

Definition 7 (p_Clique) A *p_Clique* in $\mathcal{G}_\mathcal{C}$ is a clique where all edges are labeled with p. □

All IC's that belong to the same p_clique, can be maintained by issuing the same operation (insertion or deletion) into p. In Example 6, $\{c_1, c_2, c_3\}$ is a clique, but not a p_clique. However, $\{c_1, c_2\}$ is a q_clique. If we decide to maintain these two IC's by performing deletions from q, then this is equivalent to delete the edge between c_1 and c_2, the so reduced graph does not contain any remaining cycle. So whatever the action we decide to perform for maintaining c_3, the obtained program is not negatively recursive. This gives an idea of how $\mathcal{G}_\mathcal{C}$ is used in order to derive non negatively recursive programs.

1st algorithm. The following algorithm derives non negatively recursive programs but which may contain conflicting rules.

Input: A set of constraints $\mathcal{C}$.
Output: A program $\mathcal{P}_\mathcal{C}$ non negatively recursive.
Begin
- $\mathcal{P}_\mathcal{C} := \emptyset$;
- Let $\mathcal{G}_\mathcal{C}$ be the graph of $\mathcal{C}$;
- **While** $\mathcal{G}_\mathcal{C}$ contains connected vertices **Do**:
  - Choose a maximal p_clique CL
    - **For each** $c \in CL$ **Do**
      - Derive a rule $\rho(c)$ which updates p;
      - $\mathcal{P}_\mathcal{C} := \mathcal{P}_\mathcal{C} \cup \{\rho(c)\}$;
      - Remove c from $\mathcal{G}_\mathcal{C}$
  - **End For**
    - **For each** $c \in \mathcal{G}_\mathcal{C}$ that depends on p **Do**
      - Derive a rule $\rho(c)$ which updates p;
      - $\mathcal{P}_\mathcal{C} := \mathcal{P}_\mathcal{C} \cup \{\rho(c)\}$;
      - Remove c from $\mathcal{G}_\mathcal{C}$
  - **End For**
- **End While**
- **For each** remaining constraint c **Do**
  - Derive a rule $\rho(c)$;
  - $\mathcal{P}_\mathcal{C} := \mathcal{P}_\mathcal{C} \cup \{\rho(c)\}$;
  - Remove c from $\mathcal{G}_\mathcal{C}$
- **End For**
- **Return** $\mathcal{P}_\mathcal{C}$

End

Proposition 2 Let $\mathcal{C}$ be a set of IC's. Then $\mathcal{P}_\mathcal{C}$ the output of the above algorithm is not negatively recursive. □

3.2 Managing conflicting rules

In this section, we show how to transform a potentially conflicting but non negatively recursive program in such a way that it insures a total well founded model.

We first consider the simple case of two rules and illustrate it with the following example:

Example 7 Let $\mathcal{C} = \{p(X), \neg q(X) \longrightarrow \mathtt{inc};\ r(X), q(X) \longrightarrow \mathtt{inc}\}$ and

$$\mathcal{P_C} = \left\{ \begin{array}{l} p(X) \longrightarrow +q(X) \\ r(X) \longrightarrow -q(X) \end{array} \right\}$$

$\mathcal{P_C}$ is potentially conflicting. Consider Δ, where

p
a
b

r
a
c

q
a
b
c

$\mathcal{M}_\Delta(\mathcal{P_C}) = \Delta \cup \{\neg p(c), \neg r(b), +q(b), -q(c), \neg -q(b), \neg +q(c)\}$. The literals $+q(a)$ and $-q(a)$ are unknown. It is easy to see that $\mathcal{C}$ is equivalent to:

$$\mathcal{C}' = \left\{ \begin{array}{ll} p(X), \neg r(X), \neg q(X) & \longrightarrow \texttt{inc} \\ r(X), \neg p(X), q(X) & \longrightarrow \texttt{inc} \\ p(X), r(X) & \longrightarrow \texttt{inc} \end{array} \right\}$$

From $\mathcal{C}'$, we derive the program:

$$\mathcal{P_C}' = \left\{ \begin{array}{lll} p(X), \neg r(X) & \longrightarrow & +q(X) \\ r(X), \neg p(X) & \longrightarrow & -q(X) \\ r(X) & \longrightarrow & -p(X) \end{array} \right\}$$

The obtained database by using this program is $\Delta' = \{p(b), r(a), r(c), q(b)\}$. Notice that:

1. Δ' satisfies $\mathcal{C}$,
2. The IC's maintained by $\mathcal{P_C}'$ are exactly the same as those maintained with $\mathcal{P_C}$; i.e. the expression

$$\forall X : [\underline{(p(X) \Rightarrow r(X) \vee q(X))} \wedge \underline{(r(X) \wedge q(X) \Rightarrow p(X))} \wedge \underline{(r(X) \wedge p(X) \Rightarrow)}]$$

$\mathcal{C}'$, is equivalent to

$$\forall X : [\underline{(p(X) \Rightarrow q(X))} \wedge \underline{(r(x) \wedge q(X) \Rightarrow)}].$$

Finally, notice that $\mathcal{P_C}'$ is not the only program we may derive. Indeed, with

$$\mathcal{P_C}'' = \left\{ \begin{array}{lll} p(X), \neg r(X) & \longrightarrow & +q(X) \\ r(X), \neg p(X) & \longrightarrow & -q(X) \\ p(X) & \longrightarrow & -r(X) \end{array} \right\}$$

we obtain $\Delta'' = \{p(a), p(b), r(c), q(a), q(b)\}$ which is also consistent.

$\square$

3.2.1 The case of two conflicting rules

Consider the IC's

$$\mathcal{C}_1 = \left\{ \begin{array}{lcl} \ell_1, \ldots, \ell_m, p(\tilde{X}), c_1(\tilde{X_1}) & \longrightarrow & \texttt{inc} \\ m_1, \ldots, m_n, \neg p(\tilde{X}), c_2(\tilde{X_2}) & \longrightarrow & \texttt{inc} \end{array} \right\}$$

where ℓ_i and m_j are base literals, and c_i is a conjunction of interpreted literals (e.g. $X = 3, Y > 20, X = Y, X \neq Y$). From these IC's, we derive the rules

$$\mathcal{P}_{\mathcal{C}_1} = \left\{ \begin{array}{lcl} \rho_a : \quad \ell_1, \ldots, \ell_m, c_1(\tilde{X_1}) & \longrightarrow & -p(\tilde{X}) \\ \rho_b : \quad m_1, \ldots, m_n, c_2(\tilde{X_2}) & \longrightarrow & +p(\tilde{X}) \end{array} \right\}$$

These rules may, à priori, introduce conflicts. Now Consider the IC's:

$$\mathcal{C}_2 = \left\{ \begin{array}{lcl} \ell_1, \ldots, \ell_m, c_1(\tilde{X_1}) & \longrightarrow & p'(\tilde{X}') \\ m_1, \ldots, m_n, c_2(\tilde{X_2}) & \longrightarrow & p''(\tilde{X}'') \\ p'(\tilde{X}'), \neg p''(\tilde{X}''), p(\tilde{X}) & \longrightarrow & \texttt{inc} \\ p''(\tilde{X}''), \neg p'(\tilde{X}'), \neg p(\tilde{X}) & \longrightarrow & \texttt{inc} \\ \ell_1, \ldots, \ell_m, c_1(\tilde{X_1}), m_1, \ldots, m_n, c_2(\tilde{x_2}) & \longrightarrow & \texttt{inc} \end{array} \right\}$$

Where $\tilde{X}'$ is the set of variables in $\tilde{X_1}$ and $\tilde{X}'' = \tilde{x}$. From this new set of IC's, we derive

$$\mathcal{P}_{\mathcal{C}_2} = \left\{ \begin{array}{lrcl} \rho_1 : & \ell_1, \ldots, \ell_m, c_1(\tilde{X}) & \longrightarrow & p'(\tilde{X}') \\ \rho_2 : & m_1, \ldots, m_n, c_2(\tilde{X}) & \longrightarrow & p''(\tilde{X}'') \\ \rho_3 : & p'(\tilde{X}'), \neg p''(\tilde{X}'') & \longrightarrow & -p(\tilde{X}) \\ \rho_4 : & p''(\tilde{x}''), \neg p'(\tilde{x}') & \longrightarrow & +p(\tilde{X}) \\ \rho_5 : & m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_n, c_2(\tilde{X}), \ell_1, \ldots, \ell_m, c_1(\tilde{X}) & \longrightarrow & op|m_i| \end{array} \right\}$$

Where $op|m_i| = \left\{ \begin{array}{ll} +(\neg m_i) & \text{if } m_i \text{ is negative} \\ -m_i & \text{otherwise} \end{array} \right\}$

Lemma 4 $\mathcal{C}_1$ is equivalent to $\mathcal{C}_2$. □

The body of the rules ρ_3 and ρ_4 in $\mathcal{P}_{\mathcal{C}_2}$, cannot both be true (they mutually exclude each other). Thus, we cannot be in a situation where we want to insert and delete the same tuple from p. Is this enough to say that we have avoided all unknown updates? In other words, can we be sure that the new program is not negatively recursive? The answer to this question depends on the rule ρ_5. Indeed, the new program is negatively recursive only if ρ_5 itself is.

Lemma 5 ρ_5 cannot be negatively recursive. □

Poof: First, notice that ρ_5 cannot be negatively recursive because of a dependency between m_i and an m_j, otherwise this would mean that m_j and m_i have the same sign which contradicts the fact that we don't have repeated predicates.

However, if there exists an ℓ_j s.t. ℓ_j and m_i can be unified (i.e. there exists a substitution θ with $\ell_j\theta = m_i\theta$), then ρ_5 is negatively recursive. Notice that in this case, ℓ_i and m_j are built from the same predicate. We can assume without loss of generality that for every predicate p and every litteral appearing in denials bodies of the form $p(\tilde{X})$ the terms appearing in $\tilde{X}$ are only variables and always the same, i.e. $p(X, a)$ can be replaced by $p(X, Y) \wedge Y = a$ and $p(X, X)$

is replaced by $p(X,Y) \wedge X = Y$. Notice that this assumption is no longer true when accepting repeated predicates. Hence, ℓ_i and m_i are identical. Which means that the condition $\ell_i \wedge m_j$ is equivalent to m_j. This allows us to eliminate ℓ_j from the body of ρ_5. This way, it is not negatively recursive any more. ■

3.3 The case of multiple conflicting rules

In the previous section, we have shown how to solve the problem of conflicting updates where we have only two rules. In this section, we generalize that method to the case of multiple rules. Consider the program:

$$\begin{array}{llcl}
\rho_p : & \ell_1, \ldots, \ell_m, c_p(\tilde{X}_p) & \longrightarrow & -p(\tilde{X}) \\
\hline
\rho_{p+1} : & m^1_1, \ldots, m^1_{n_1}, c_{p+1}(\tilde{X}_{p+1}) & \longrightarrow & +p(\tilde{X}) \\
 & \ldots & & \\
\rho_{p+q} : & m^q_1, \ldots, m^q_{n_1}, c_{p+q}(\tilde{X}_{p+q}) & \longrightarrow & +p(\tilde{X})
\end{array}$$

The rule ρ_p conflicts with the other rules. The solution is to proceed in a way such that the condition (i.e. body) of ρ_p and those of the other rules cannot be true at the same time. For this purpose, we transform the program as follows:

$$\begin{array}{lrcl}
\rho'_p : & \ell_1, \ldots, \ell_m, c_p(\tilde{X}_p) & \longrightarrow & p'(\tilde{X}) \\
\hline
\rho'_{p+1} : & m^1_1, \ldots, m^1_{n_1}, c_{p+1}(\tilde{X}_{p+1}) & \longrightarrow & p''(\tilde{X}) \\
\ldots & & & \\
\rho'_{p+q} : & m^q_1, \ldots, m^q_{n_q}, c_{p+q}(\tilde{X}_{p+q}) & \longrightarrow & p''(\tilde{X}) \\
\hline\hline
\rho''_p : & \ell_1, \ldots, \ell_m, c_p(\tilde{X}_p), \neg p''(\tilde{X}) & \longrightarrow & -p(\tilde{X}) \\
\hline
\rho''_{p+1} : & m^1_1, \ldots, m^1_{n_1}, c_{p+1}(\tilde{X}_{p+1}), \neg p'(\tilde{X}) & \longrightarrow & +p(\tilde{X}) \\
 & \ldots & & \\
\rho''_{p+q} : & m^q_1, \ldots, m^q_{n_q}, c_{p+q}(\tilde{X}_{p+q}), \neg p'(\tilde{X}) & \longrightarrow & +p(\tilde{X}) \\
\hline
\end{array}$$

The only dependencies generated by these rules are those already existing in the initial program. Thus, if the first program is initially non negatively recursive, then the new one is also non negatively recursive.

The remaining case is that where the condition of the rule which makes an insertion and the condition of one of the rules making a deletion are both true, i.e we should consider the IC's of the form

$$c_i : body(\rho_p), body(\rho_{p+i}) \longrightarrow \texttt{inc}$$

To maintain this IC, we must derive a rule while preserving the property of non negative recursion. The following may arise:

- Suppose that there exists a literal M in $body(c_i)$ s.t. there exists ρ with $body(\rho) \ni L$ where L and M are two literal from the same predicate Q. This means that we can update Q in order to maintain c_i without creating a negative cycle.
- Suppose now that their does not exist such an M. It is easy to see that this contradicts the hypothesis that the initial program was non negatively recursive.

In this section, we have first proposed an approach that allows to derive update programs that are not negatively recursive. Then, we have presented a method that solves the problem of potentially conflicting rules. The combination of the two gives programs that satisfy the following proposition.

Proposition 3 Let $\mathcal{C}$ be a set of constraints and $\mathcal{P}_\mathcal{C}$ the program obtained after the two steps described in this section. Then, for each update δ and instance Δ, the update model $\mathcal{M}_{\Delta_{user}}(\mathcal{P}_\mathcal{C})$ is total and non conflicting.

□

So, we can use $\mathcal{P}_\mathcal{C}$ to maintain and restore database consistency. Concerning the complexity of this process is however exponential. The first algorithm is in linear time w.r.t the number of IC's, while the second part of the derivation is exponential.

Now we address the well known property of minimal change [KM91]. Let δ be a user update submitted to the database Δ. Δ_{user} may be seen as the database the user wants to get by his/her update. However, Δ_{user} may be not consistent. So in order to restore consistency, Δ_{user} should be updated. Let Δ_{final} be the coherent database obtained after updating Δ_{user}. Intuitively, the minimal change property says that the "quantity of change" we make to Δ_{user} to reach a consistent database must be as little as possible. Formally,

Definition 8 Δ_{final} fulfills the minimal change property iff $\forall \Delta'$coherent : $\Delta_{user} \div \Delta' \subseteq \Delta_{user} \div \Delta_{final} \Rightarrow \Delta_{user} \div \Delta' = \Delta_{user} \div \Delta_{final}$, where $\div$ is the symmetrical set difference.

□

Proposition 4 If $\mathcal{M}_{\Delta_{user}}(\mathcal{P}_\mathcal{C})$ is total and not conflicting (which is the case with the programs we derive), then Δ_{final} fulfills the minimal change property.

□

4 Expressiveness

Let's now address the expressive power of the update semantics with update rules. An update μ in a language $\mathcal{L}$ can be considered as a sequence of two operations ϕ, ψ where ϕ is a query and ψ is an assignment. Hence, $\mathcal{L} \equiv \mathcal{Q}; FO^{:=}$, where $\mathcal{Q}$ is a query language and $FO^{:=}$ is first order logic with destructive assignment. For instance, consider the operation of deleting all nodes for which there is a cycle passing through. We first compute the nodes that are not in any cycle (the query part), then we assign this set to the set of nodes (the assignment part). Thus, the expressiveness of an update language is fully determined by it's query language part. In our update semantics, we first we derive updates, then we apply them. The expressiveness of the first part is that of *Fixpoint* [AHV95] since we use well founded semantics which is equivalent to *Fixpoint*. Notice that the combination of the two parts is a subset of the *While* language[1]. Let $\mathcal{U}$ denote our update semantics. Then

$$\mathcal{U} \subset While$$

since $\mathcal{U}$ clearly belongs in PTIME while the language $While$ is in PSPACE and it is conjectured that PTIME $\neq$ PSPACE.

5 Conclusion

In this paper, we have proposed an algorithm that derives update programs from IC's. These programs can not only maintain database consistency but also restore it. This is of a particular interest when IC's can be added dynamically.

In Example 4 we saw that it is not possible to restore the functional dependency constraint while guaranteeing the minimal change property with a deterministic language (the case of well founded Datalog). This example suggests the use of the *choice* construct [GPSZ97] which introduces a kind of controlled non determinism while leaving the computational complexity in PTIME which is not the case of the stable semantics[GL88] used by [MT98]. In fact, there are two kinds of non determinism: tuple and predicate non determinism. We solved the latter during the update rules derivation.

As future research we would like to investigate is the following problem. let's say that an update language $\mathcal{L}$ can *maintain* a class of IC's $\mathcal{C}$, denoted $\mathfrak{M}_{\mathcal{L}}^{\mathcal{C}}$ iff for all $C \subseteq \mathcal{C}$, there exists a program $\mathcal{P} \in \mathcal{L}$ such that for all Δ satisfying C and for each update δ, if Δ_{user} is defined as before and $\mathcal{P}(\Delta_{user})$ is the final database, then $\mathcal{P}(\Delta_{user}) \div \Delta_{user}$ is minimal. We say that $\mathcal{L}$ can *restore* $\mathcal{C}$, denoted $\mathfrak{R}_{\mathcal{L}}^{\mathcal{C}}$ if the above definition holds when Δ is not supposed consistent and $\delta = \emptyset$. In other words, $\mathcal{P}$ can restore database consistency with minimal change. For instance, for the update rule

$$p(X,Y), Y \neq Y' \longrightarrow \neg p(X,Y')$$

derived from the functional dependency, the update models that can restore database consistency with minimal change are exactly those which are stable models. Giving general relationships between $\mathcal{L}$ and $\mathcal{C}$ could be an interesting direction of future research.

In [FP97] the authors address the following problem: given a set of IC's $\mathcal{C}$, for each $c \in C$ there may be many ways to maintain c. To express his/her preferences, the DB administrator assigns a weight to each alternative, the problem is to find a program $\mathcal{P}$ which is guaranteed to terminate while maximizing the global weight. The authors show that this is a NP hard problem. In our paper, we have not considered user preferences when deriving programs. It could be interesting to use the change metrics between the instances generated by two programs as the distance between them and use this definition to generate programs which are among the closest to those proposed by the user.

[1] Actually, *While* can be considered as both a query and an update language.

Aknowledgment

The author would thank the referees for their careful reading and fruitful comments which helped to improve the presentation of this work.

References

[AART97] D. Aquilino, P. Asirelli, C. Renso, and F. Turini. Applying restriction constraints to deductive databases. *Annals of Mathematics and Artificial Intelligence*, 19(I-II):3–25, 1997.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison-Wesley, 1995.

[ALS95] M. Halfeld Ferrari Alves, D. Laurent, and N. Spyratos. Update rules in datalog programs. In *Proceedings of LPNMR'95, Logic Programming and Non Monotonic Reasoning*, LNAI 928. Springer Verlag, 1995.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: Termination, confluence and observable determinisme. In *Proceedings of the ACM Sigmod conference*, San Diego, USA, June 1992.

[BCP95] E. Baralis, S. Ceri, and S. Paraboshi. Run-time detection of non-terminating active rule systems. In T. Ling, A. Mendelzon, and L. Vieille, editors, *Proceedings of DOOD conference*, volume 1013 of *LNCS*, pages 38–54. Springer, 1995.

[BDR98] J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proceedings of PODS conference*, 1998.

[BM97] N. Bidoit and S. Maabout. A model theoretic approach to update rule programs. In *Proceedings of ICDT Conference*, LNCS. Springer, 1997.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboshi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM TODS*, 19(2):367–422, 1994.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases.* Springer, 1990.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th VLDB conference*, 1990.

[DDS98] M. Dekhtyar, A. Dikovsky, and N. Spyratos. On Logically Justified Updates. In *Proceedings of JICSLP conference.* MIT Press, 1998.

[FP97] P. Fraternali and S. Paraboschi. Ordering and selecting production rules for constraint maintenance. Complexity and heuristique solution. *IEEE TKDE*, 9(1):173–178, 1997.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *Proceedings of the 5th international symposium on logic programming.* MIT Press, 1988.

[GPSZ97] F. Giannotti, D. Pedreshi, D. Saccà, and C. Zaniolo. Programming with Non-determinism in Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 19(3-4), 1997.

[KM91] H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledgebase and revisiting it. In *Proceedings of Intrl. Conf. on Knowledge Represention and Reasoning (KR'91)*, 1991.

[LM98] B. Ludäscher and W. May. Referential actions: From logical semantics to implementation. In *Proceedings of EDBT'98 conference*, volume 1377 of *LNCS*, Valencia, Spain, Mar. 1998.

[LML97] B. Ludäscher, W. May, and G. Lausen. Referential actions as logic rules. In *Proceedings of PODS conference*, 1997.

[MT98] V. Marek and M. Truszczyǹski. Revision programming. *Theor. Computer Science(TCS)*, 190(2):241–277, 1998.

[ST96] K. D. Schewe and B. Thalheim. Active consistency enforcement for repairable database transitions. In *Proceedings Int. Workshop on foundation of models and languages for databases and objects: Integrity in Databases*, Sept. 1996.

[TO95] E. Teniente and A. Olivé. Updating knowledge bases while maintaining their consistency. *VLDB journal*, 4(2):193–241, 1995.

[UD92] S.D. Urban and M. Desideiro. CONTEXT: a CONnsTraint EXplanation Tool. *Data & Knowledge Enginieering*, 8:153–183, 1992.

[vGRS91] A. van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38, 1991.