Well-Founded Views in Constraint Databases: Incremental Materialization and Maintenance

James Lu* Bertram Ludäscher† Joachim Schü ‡ V.S. Subrahmanian§

Abstract

In this paper, we study the problem of incremental maintenance of materialized views when the view definitions are allowed to contain: (a) constraints, and (b) well-founded modes of negation. A materialized view in our framework consists of a set of *constrained non-ground tuples.* We develop incremental algorithms to compute such non-ground materialized well-founded views, extending previous results of Gottlob et. al. and Eiter et. al. With respect to (b), previous attempts at view maintenance have been limited to the case when stratified negation is used. In contrast, in this paper, we allow arbitrary (stratified and/or unstratified) usage of negation. Recursive views are also allowed. We develop a view maintenance algorithm called **WF-Red** for handling updates in such cases. **WF-Red** takes two parameters as input – a list of **insertions** and a list of **deletions**. To handle insertions, the list of **deletions** is kept empty and vice-versa. The algorithm correctly handles both insertions and deletions to the materialized view.

1 Introduction

View Materialization (VM) is a well studied technique in databases. Given a database DB, users often define views of the database – such a view is merely a query specified in the language of the database. Thus, views may be defined in languages such as SQL, or just as easily in deductive logical languages such as Datalog. *View Materialization* corresponds to the full evaluation of a view/query, and storing the result as a relation. When the same query (or another query that involves this materialized view) is subsequently posed, processing can often be performed with a considerable increase in speed by consulting the materialized view. A problem arises, however, when changes occur in one or more of the base relations upon which this view is defined. In such cases, the materialized view must be updated to incorporate the changes that occur in the base relation – this is the problem of *View Maintenance.*

View Maintenance is a subject that has been studied by many [1, 3, 5, 6, 8, 9, 10, 11, 15, 16, 17, 18, 21, 22, 25, 26, 27, 28, 29, 30, 31, 32, 33, 35, 23]. Every view maintenance technique in existence depends fundamentally upon the following key parameters:

*Computer Science Dept., Bucknell University, Lewisburg, PA 17837, USA. jameslu@bucknell.edu. Supported by the NSF under Grant CCR9225037.

†Institut für Informatik, Universität Freiburg. ludaesch@informatik.uni-freiburg.de.

‡Institut für Algorithmen und Kognitive Systeme, Universität Karlsruhe, schue@ira.uka.de.

§Institute for Advanced Computer Studies, Institute for Systems Research, Department of Computer Science, University of Maryland, College Park, Maryland 20742. vs@cs.umd.edu.

- **How is a view defined ?** For example, a view may be defined in terms of a simple `SELECT ..... FROM ..... WHERE .....` SQL query or in terms of a Datalog query of the form $\mathtt{A} \leftarrow \mathtt{B_1}, \ldots, \mathtt{B_n}$. In general, there are many different ways of specifying a view.

- **How is the materialized view stored ?** Most often, a materialized view is stored as a set of relations; however, there are differences – for example, in [23] materialized views are sets of *constrained* tuples.

- **What kinds of updates are allowed?** Several techniques only allow updates (insertions/deletions) to base relations. However, it is often the case that derived relations (views) are defined in terms of other derived relations. Under these circumstances, it may also be desirable to handle updates to derived relations as well.

Our work differs from previous works on this topic in the following ways:

- We allow view definitions to be sets of Datalog queries extended with constraints and non-monotonic negation. A typical view has the following form.

$$A(\vec{X}) \quad \leftarrow \quad \Xi \mid B_1(\vec{X}_1), \ldots, B_n(\vec{X}_n), \mathbf{not}(C_1(\vec{Y}_1)), \ldots, \mathbf{not}(C_m(\vec{Y}_m))$$

  Each of the $A(..)$, $B(..)$, and $C(..)$'s are atoms, Ξ is a constraint – a first order formula over an appropriate domain – and **not** represents well-founded negation [36]. *To our knowledge, this is the first paper in which view maintenance is studied when view definitions are allowed to contain well-founded modes of negation.* Moreover, not only do we study well-founded views, we also study well-founded views when constraints are present.

- In our framework, we will allow materialized views to be *sets of constrained atoms*. [23] was the first paper to study the case when materialized views consisted of constrained atoms. However, in contrast to [23], we study the problem of maintaining such materialized views when well-founded negation is allowed to occur, and no assumption of stratification [2] is made.

As is customary, we allow the operations of insertions and deletions to modify the materialized view.

The next section summarizes the basic syntax and semantics of constrained view definitions. Section 3 examines the computation of the well-founded materialized view, while Section 4 investigates techniques for maintaining a previously computed materialized view.

2 Constrained View Definitions

In this section, we will provide a formal definition of a "view." A *constraint domain* can be thought of as a triple (D, F, R) where D is a set (whose elements can be thought of as objects of interest within that domain), F is a set of functions (partial or total) on domain D (perhaps of different arities), and R is a set of relations on D.

A good example of a constraint domain is the triple $(\mathbf{R}, \{+, -, \times, \div\}, \{\leq, =, \geq\})$ where $\mathbf{R}$ is the set of real numbers, the functions of interest are the ordinary arithmetic functions, and the relations of interest are the usual equalities and inequalities.

A *Constrained View Definition (CVD)* is a set of formulas of the form:

$$A(\vec{X}) \quad \leftarrow \quad \Xi \mid B_1(\vec{X}_1), ..., B_n(\vec{X}_n), \mathbf{not}(C_1(\vec{Y}_1)), ..., \mathbf{not}(C_m(\vec{Y}_m)) \tag{1}$$

Expression 1 is said to *define* the **derived relation** A. Ξ is called the *constraint part* of the above formula, $A(\vec{X})$ is called the head, and $B_1, \ldots, B_n, C_1, \ldots, C_m$ is called the *body*. The relations $B_1, \ldots, B_n, C_1, \ldots, C_m$ in the body are either derived relations (i.e. ones that are defined within the CVD, perhaps by another rule) or **base** relations stored in a relational database which the CVD accesses.

Example 2.1 Consider the following relational database which contains two relations, `emp` and `payroll`.

Name	Dept	Title
john	personnel	manager
ed	marketing	manager
lisa	personnel	director
mary	technical	manager
frank	marketing	assistant
tina	technical	engineer
oliver	personnel	secretary
diane	marketing	secretary

Relation `emp`

Name	Salary
john	76000
ed	61000
lisa	98000
mary	59000
frank	38000
tina	51000
oliver	21000
diane	19000

Relation `payroll`

A user may define a CVD on these two base relations of the following form.

$$\texttt{veryrich(X)} \quad \leftarrow \quad \texttt{Sal} \geq \texttt{90000} \mid \texttt{emp(X,Dept,Title)}, \texttt{sal(X,Sal)}. \tag{2}$$

$$\texttt{rich(X)} \quad \leftarrow \quad \texttt{Sal} \geq \texttt{70000} \mid \texttt{emp(X,Dept,Title)}, \texttt{sal(X,Sal)}, \mathbf{not}(\texttt{veryrich(X)}). \tag{3}$$

These views define rich people, and very rich people who work for the company. In particular, Rule 3 uses non-monotonic negation to determine those who are rich.

In addition to the above example, constraint domains could access data over different data sources – [24] shows how constraint domains may be used for the purposes of implementing mediators for integrating information from heterogeneous data sources.

2.1 Well-Founded Semantics for Constraint View Definitions

In this section, we briefly describe how the well-founded semantics for deductive databases proposed by Van Gelder, Ross and Schlipf [36] may be extended to the case of constrained databases. One of the fundamental problems with the original formulation of the well-founded semantics is that ground instances of all atoms are computed and stored as tuples in a relational DBMS. In contrast, when considering constraints, such grounding is not feasible. For example, an atom $A(X)$ that is true whenever $\texttt{X} \geq 3$ leads to infinitely many ground instances which obviously cannot be stored as a flat, materialized relation. Recent advances in constraint databases by Jaffar and Lassez[19] and Kanellakis et. al. [20] show that in such cases, it is better to admit tuples that contain variables which obey relevant constraints. Using this intuition, Gottlob et. al. [14] and Eiter et. al. [12] have defined a version of the well-founded semantics of deductive databases that accommodates constraints. We recapitulate the essential definitions below.

If $A(\vec{Y})$ is an atom, and Ξ is a constraint whose variables are drawn from those in $\vec{Y}$, then $A(\vec{Y}) \leftarrow \Xi$ is called a **constrained-atom**. We use the notation $[A(\vec{Y}) \leftarrow \Xi]$ to denote the set of all instances of $A(\vec{Y})$ that satisfy the constraint Ξ. A **constrained-interpretation**, abbreviated c-interpretation, is a finite set of constrained-atoms. Note that since constraints may be disjunctive, there is no loss of generality in assuming that for any predicate symbol A, at most one constrained atom of the form $A(\vec{Y}) \leftarrow \Xi$ is contained in any c-interpretation.

Suppose $A(\vec{Y}) \leftarrow \Xi$ is a constrained atom, and CI is a constrained interpretation. $A(\vec{Y}) \leftarrow \Xi$ *is said to be true in* CI iff there exists a constrained-atom $A(\vec{X}) \leftarrow \Xi'$ in CI such that $((\vec{X} = \vec{Y}) \wedge \Xi) \rightarrow \Xi'$, i.e. every solution of the first constraint is also a solution of the second.[1]

A **constraint database** consists of (1) a constrained view definition C, (2) an *extensional database* (EDB) consisting of relationsl that are referred to in the bodies of rules in C, and (3) a constraint domain Σ. The **constraint program** P defined by a constraint database is of the form $\{A(\vec{t_j}) \mid A \text{ is a base relation } \} \cup C$.

Definition 2.2 (Negation-Free Case) The following fixpoint operator T_P over interpretation corresponds to the mapping given in [13] for propositional logic programs. Let P and I be respectively, a constraint-program and a constrained-interpretation.

$$\begin{aligned} T_P(I) \quad = \quad & \{A(\vec{X}) \leftarrow \Xi \mid \text{There is a clause } A(\vec{X_0}) \leftarrow \Xi_0 | B_1(\vec{t_1}), \ldots, B_n(\vec{t_n}) \text{ in P} \\ & \forall 1 \leq i \leq n \;:\; \exists B_i(\vec{X_i}) \leftarrow \Xi_i \in I, \text{ which share no variables and the constraint} \\ & \Xi = \Xi_0 \wedge \Xi_1 \wedge \ldots \wedge \Xi_n \wedge \{\vec{X_1} = \vec{t_1}\} \wedge \ldots \wedge \{\vec{X_n} = \vec{t_n}\} \wedge \{\vec{X} = \vec{X_0}\} \text{ is solvable}\} \end{aligned}$$

Given two c-interpretations I and J, we say that $I \leq J$ holds just in case the set of ground atoms true in I is a subset of the set of ground atoms true in J. Then as T_P is monotone with respect to $\leq$ (see Theorem 1), it is guaranteed a least fixpoint which we denote $lfp(T_P)$.

It is well-known [4] that the well-founded semantics can be defined in terms of the operator used by Gelfond and Lifschitz[13] to define stable models. However, that transformation applies only to fully grounded views – something that is impossible to do in practice. The following defines a non-ground Gelfond-Lifschitz transformation for CVD's.

Definition 2.3 (Negation Case) Let P be a constraint program and J a constrained-interpretation.

$$\begin{aligned} CGL(P, J) \quad = \quad & \{A(\vec{X}) \leftarrow \Theta' \mid B_1(\vec{X_1}), \ldots, B_n(\vec{X_n}) \mid \\ & A(\vec{X}) \leftarrow \Theta \mid B_1(\vec{X_1}), \ldots, B_n(\vec{X_n}), \mathbf{not}(C_1(\vec{X_1})), \ldots, \mathbf{not}(C_m(\vec{X_m})) \in P \\ & \forall i \in \{1, \ldots, m\} \text{ if } C_i(\vec{X_i'}) \leftarrow \Xi_i \in J \text{ then } \Theta' = (\Theta \wedge \neg \Xi_i \wedge \{\vec{X_i} = \vec{X_i'}\}) \\ & \text{is solvable}\} \end{aligned}$$

[1]Without loss of generality, variables that appear in different constraint atoms of a c-interpretation is assumed to be standardized apart.

To provide a better understanding of the algorithms, we present the win/move program from [36] as a continuing example in this paper.

Example 2.4 Suppose $J = \{win(X) \leftarrow X = b \vee X = c\}$ and P is the constraint program that contains the single constraint clause

$$win(X) \leftarrow move(X, Y), \mathbf{not}(win(Y)).$$

Then $CGL(P, J) = \{win(X) \leftarrow Y \neq b \wedge Y \neq c \mid move(X, Y)\}$. Although the condition that J contains at most one constrained atom associated with any predicate is a syntactic condition, it is necessary to capture the intended meaning of the above transformation. Suppose J had consisted instead of the (semantically equivalent) set of atoms $\{win(X) \leftarrow X = b, win(X) \leftarrow X = c\}$. The above transformation then leads to $\{win(X) \leftarrow Y \neq b \| move(X, Y); win(X) \leftarrow Y \neq c \parallel move(X, Y)\}$ which is not the intended meaning of the transformation.

Given any constraint-program P and c-interpretation I, let $\Gamma_P(I) = lfp(T_{CGL(P,I)})$. The next theorem follows immediately.

Theorem 1 *(Eiter et. al. [12])*

1. *T_P is monotone, i.e. $I \leq J \Rightarrow T_P(I) \leq T_P(J)$.*
2. *Γ_P is anti-monotone, i.e. $I \leq J \Rightarrow \Gamma_P(J) \leq \Gamma_P(I)$.*
3. *Γ_P^2 has a least fixpoint, $lfp(\Gamma_P^2)$ and a greatest fixpoint $gfp(\Gamma_P^2)$.*

The **constrained well-founded semantics (CWFS)** associated with the constraint program P is defined as:

1. Ground atom A is true in $CWFS(P)$ iff A is true in $lfp(\Gamma_P^2)$.
2. Ground atom A is false in $CWFS(P)$ iff A is not true in $gfp(\Gamma_P^2)$.
3. Otherwise, A is undefined with respect to $CWFS(P)$.

Theorem 2 *(Eiter et. al. [12]) The above definition of CWFS coincides with the standard definition of well-founded semantics [36] when no constraints are present.*

3 Incremental Materialization of CVDs

In this section, we will develop a way of materializing well-founded views. First, we will develop a sound and complete approach – later, this approach will be refined to include incremental computation steps. The basic approach follows two basic steps:

Step 1 (Preprocessing) Associate with a constraint program P, a labeled dependency graph $DG(P)$ as follows: the nodes in the dependency graph are predicates. There is a (directed) edge from predicate p to predicate q labeled $+$ (resp. $-$) if there is a rule in P that defines p and q (resp. $\mathbf{not}(q)$) occurs in the body.

The graph $DG(P)$ induces an acyclic graph $CC(P)$ via its strongly connected components (SCCs) as follows: the nodes in the graph are the SCCs of $DG(P)$. There is an arc from component C_1 to C_2 labeled $+$ (resp. $-$) just in the case there is an arc labeled $+$ (resp. $-$) in $DG(P)$ from a vertex in C_1 to a vertex in C_2.

Figure 1 shows $DG(P)$ and $CC(P)$ for the following simple constraint program P.

(1) $\mathtt{p(X)} \leftarrow \mathbf{not}(\mathtt{q(X)})$	(2) $\mathtt{q(X)} \leftarrow \mathbf{not}(\mathtt{p(X)})$
(3) $\mathtt{r(X)} \leftarrow \mathtt{p(X)}$	(4) $\mathtt{r(X)} \leftarrow \mathtt{q(X)}$
(5) $\mathtt{s(X)} \leftarrow (\mathtt{X} = \mathtt{a}) \vee (\mathtt{X} = \mathtt{b})$	

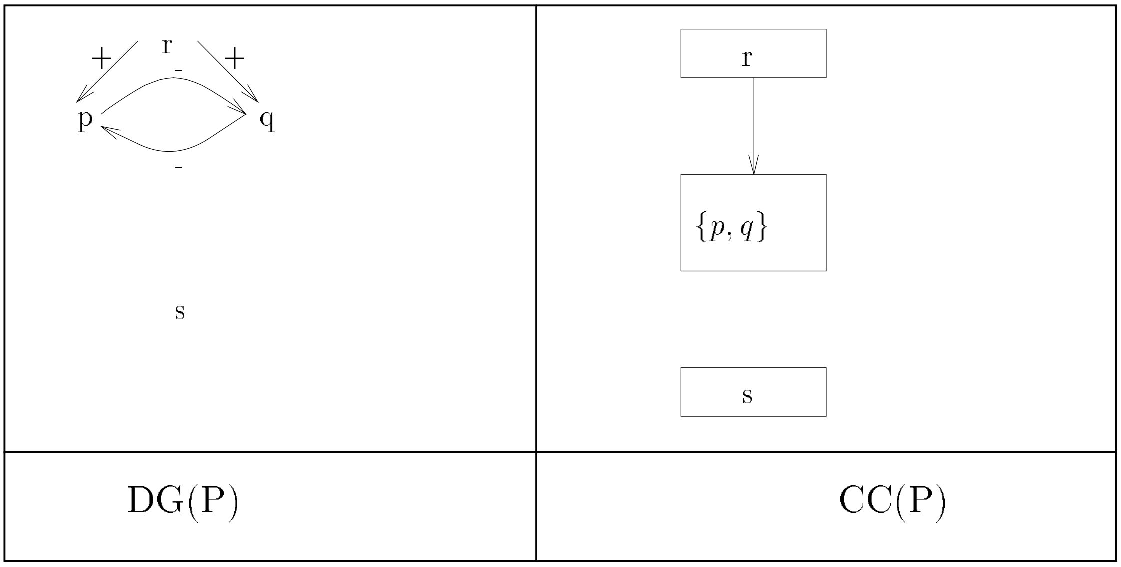

Figure 1: $DG(P)$ and $CC(P)$

Every rule in P is assigned a *Rule Stratum Number* (RSN) as follows: topologically sort the graph $CC(P)$; the i'th component in the topological sort is assigned number i. If a rule in P defines the relation p, and if p occurs in an SCC that has been assigned the number i by the topological sort, then the RSN of this rule is i.

For example, if we consider the constraint program P discussed above (see also Figure 1), then a topological sorting of this is $\mathtt{s}, \{\mathtt{p}, \mathtt{q}\}, \mathtt{r}$. This means that the rules defining `p` and `q` are in stratum 2, and the rules defining `r` are in stratum 3. The "base" relation `s` is in stratum 1. *Note that RSNs can be assigned to all constraint programs, including ones that are not stratified in the sense of Apt. et. al. [2]. In this paper, we will mean by the "stratum" of a predicate/rule, the stratum assigned by the above topological sorting.* In particular, within a given stratum, recursion may occur through negation as is seen in the example depicted by Figure 1.

Step 2 (Materialization) The derived predicates defined in the CVD are materialized in the order specified by the RSNs. The materialized well-founded view is represented as two sets of non-ground atoms, i.e. the *true* and the *undefined* atoms. The evaluation procedure for rules in higher strata which depend on such 3-valued relations can be extended in a straightforward way: Instances of bodies with true and at least one undefined literal produce atoms which are themselves undefined, provided they have not been derived as true before. With this extension to 3-valued interpretations, the materialization proceeds as usual using a semi-naive strategy to avoid redundant computations.

Remark 1 *First we present the basic view materialization algorithm, which we subsequently refine so as to facilitate incremental computations. However, in light of the above discussion on stratification, the interesting case that our algorithms must handle is the case of a single non-stratified SCC. Hence, in order to simplify the discussion, we assume for the remainder of the paper that all algorithms and discussions are given with respect to a single SCC. It is straightforward to extend these algorithms to DG(P)'s that contain multiple SCCs simply by iterating through the SCCs according to the ordering given by the RSN. The materialized well-founded views of the lower SCCs can be regarded as defining the EDB of the higher SCCs. In the algorithms that follows, such an EDB will be represented in the inputs.*

Before the basic algorithm is presented, we need the following definitions.

Definition 3.1 **(T'_P)** Given c-interpretations I and J, the binary operator T'_P is given by the following.

$$T'_P(I, J) = T_{CGL(P,J)}(I)$$

The operator T'_P will be used to calculate both the true and the undefined constraint atoms of a well-founded materialized view in Algorithm 3.3 below.

The following operations on sets of c-interpretations are needed in subsequent algorithms.

Definition 3.2 (Operations on C-Interpretations) If I, J are c-interpretations, then:

1. $\neg I = \{A(\vec{X}) \leftarrow \neg\Xi \mid A(X) \leftarrow \Xi \in I\}$
2. $I \setminus J = \{A(\vec{X}) \leftarrow \Xi \wedge \neg\Xi' | A(X) \leftarrow \Xi \in I$ and $B(X) \leftarrow \Xi' \in J$ and $[A(X) \leftarrow \Xi] \cap [B(X) \leftarrow \Xi'] \neq \emptyset\}$.
3. $I \cap J = \{A(\vec{X}) \leftarrow \Xi \wedge \Xi') | A(\vec{X}) \leftarrow \Xi \in I$ and $A(\vec{X}) \leftarrow \Xi' \in J$ and $[A(\vec{X}) \leftarrow \Xi] \cap [A(\vec{X}) \leftarrow \Xi'] \neq \emptyset\}$.
4. $I \cup J = \{A(\vec{X}) \leftarrow \Xi \wedge \Xi' | A(\vec{X}) \leftarrow \Xi \in I$ and $A(\vec{X}) \leftarrow \Xi' \in J\}$

The inputs U_1 and O_0 to the basic algorithm below represent the true atoms and the true or undefined atoms from lower strata, as described in Remark 1. The names are chosen to indicate that the first set represents an underestimate, and the second set represents an overestimate. Together they define the well-founded materialized view of the lower strata in which the true, the undefined, and false atoms are given respectively by U_1, $O_0 \setminus U_1$, and the set of all other atoms in the language not in O_0. Again, the CVD under consideration only contains rules defining the current SCC.

Algorithm 3.3 (Non-stratified CVD)

Input:	U_1 = true atoms from lower strata
	O_0 = true or undefined atoms from lower strata
Output:	The materialized well-founded view of the CVD

$k := 1$

repeat *outer fixpoint*
  $O_k := O_0$; $\Delta O := U_1$
  while $\Delta O \neq \emptyset$ **do begin** *inner fixpoint 1*
    $O_k := O_k \cup \Delta O$
    $\Delta O := T'_P(O_k, U_k) \setminus O_k$
  end
  $U_{k+1} := \emptyset$; $\Delta U := U_1$
  while $\Delta U \neq \emptyset$ **do begin** *inner fixpoint 2*
    $U_{k+1} := U_{k+1} \cup \Delta U$
    $\Delta U := T'_P(U_{k+1}, O_k) \setminus U_{k+1}$
  end
  $k := k + 1$
until $[U_k] = [U_{k-1}]$

output $(True, Undef) := (U_{k-1}, O_{k-1} \setminus U_{k-1})$

In the above algorithm, it is clear that the indices for U_k and O_k are superfluous. Only the "previous" and "current" values of U and O have to be maintained. However, for the algorithms presented in Section 4, it is useful to remember for each true atom, the first k where it appeared in an underestimate, and for each false atom, the first k where it was removed from the overestimate. Adorning atoms in this way introduces only a minor overhead. The advantage of such adornments is that all U_k and O_k can be stored *implicitly* in place of the sets U (the finally true atoms) and O_1 (the first overestimate).[2]

Theorem 3 *Algorithm 3.3 is correct.*

3.1 Optimizations

To avoid unnecessary computations several optimizations are possible.

Incremental computation of underestimates: A first improvement is to start each computation of an underestimate U_k using the value of the previous underestimate U_{k-1}, thereby avoiding the complete recomputation of U_{k-1}. This optimization can be implemented by replacing the statement $\Delta U := U_1$ by $\Delta U := U_k$ before the inner fixpoint 2. Since the second inner fixpoint iteration is cumulative (adding consequences does not change the result of the iteration), this modification is sound.

The inner fixpoint iterations are monotonically increasing and correspond to computing the minimal model of the negation free constraint program $GL(P, I)$. Therefore the well-known technique of semi-naive evaluation can be applied to inner fixpoints. We now describe the underlying idea of semi-naive evaluation using an incremental operator T_P^δ. The implementation of semi-naive evaluation based on T_P^δ is straightforward and we omit its discussion

[2]This is due to the fact that the sequences U_k and O_k are monotonically increasing and decreasing, respectively.

in this paper (see e.g. [34]). The essential idea is that whenever a new tuple is added from the head of a rule in an iteration, at least one literal in the body of the rule must have been newly produced from the previous iteration. The extra information is carried as a new input, ΔI, in T'_P.

Definition 3.4 (T^δ_P) Given c-interpretations I, J, and ΔI, the operator T^δ_P is given by the same definition as T'_P, but with the additional requirement that any rule

$$A(\vec{X}_0) \leftarrow \Xi_0 | B_1(\vec{t}_1), \ldots, B_n(\vec{t}_n)$$

that belongs to $CGL(P, J)$ which is used in computing $T_{CGL(P,J)}(I)$ must satisfy the condition that *for at least one* B_i of the body, there is a c-atom $B_i(\vec{X}) \leftarrow \Xi$ in ΔI and the constraint $\Xi_0 \wedge \Xi \wedge \{\vec{X} = \vec{t}_i\}$ is solvable.

Notice that the above fixpoint operator T^δ_P is incremental – only atoms whose status has changed (i.e. are in ΔI) trigger rules. Based on this new fixcpoint operator, we arrive at the following improved version of the inner fixpoint 2:

$U_{k+1} := \emptyset; \quad \Delta U := U_k$

while $\Delta U \neq \emptyset$ **do begin**			*inner fixpoint 2*
U_{k+1}	$:=$	$U_{k+1} \cup \Delta U$	
ΔU	$:=$	$T^\delta_P(\Delta U, U_{k+1}, O_k) \setminus U_{k+1}$	
end			

Initially, $\Delta U = U_k$. In other words, the complete previous underestimate is regarded as new. This can be further improved since the first new underestimates which are produced are due to a change in the previous *over*estimate, i.e. they must use at least one tuple that was eliminated in the preceding iteration for overestimates. We exploit this information using the following operator:

Definition 3.5 ($T^{\delta\neg}_P$) Given c-interpretations I, J, and ΔI, the operator $T^{\delta\neg}_P$ is given by the same definition as T'_P, but with the additional requirement that any rule

$$A(\vec{X}_0) \leftarrow \Xi_0 | B_1(\vec{t}_1), \ldots, B_n(\vec{t}_n), \mathbf{not}(C_1(\vec{s}_1)), \ldots, \mathbf{not}(C_m(\vec{s}_m))$$

that belongs to P which is used in computing $CGL(P, J)$ must satisfy the condition that *for at least one* C_j of the body, there is a c-atom $C_j(\vec{X}) \leftarrow \Xi$ in ΔJ and the constraint $\Xi_0 \wedge \mathbf{not}(\Xi) \wedge \{\vec{X} = \vec{s}_j\}$ is solvable.

The intuition behind the incrementality of $T^{\delta\neg}_P$ is that initially only tuples from $\Delta J := O_{k-1} \backslash O_k$ give rise to new tuples in U_{k+1}, so we can initialize ΔU with $T^{\delta\neg}_P(O_{k-1} \backslash O_k, U_k, O_k)$ before the inner fixpoint 2.

Incremental computation of overestimates: We would like to use the same ideas as for the incremental computation of underestimates. However, since overestimates are decreasing, we cannot use a corresponding initialization for O_k as we have used for U_k, viz the value of the previous outer iteration. Instead we initialize O_k to the value U_k of the preceding underestimate and use the semi-naive scheme in the inner fixpoint iteration. The final algorithm is given below:

Algorithm 3.6 (Incremental computation of non-stratified SCC)

Input:	U_1 = true atoms from lower strata
	O_0 = true or undefined atoms from lower strata
Output:	The materialized well-founded view of the CVD

$k := 1$

repeat *outer fixpoint*

$O_k := \emptyset; \quad \Delta O := U_k$

while $\Delta O \neq \emptyset$ **do begin** *inner fixpoint 1*

$O_k := O_k \cup \Delta O$

$\Delta O := T_P^{\delta}(\Delta O, O_k, U_k) \setminus O_k$

end

$U_{k+1} := U_k$

$\Delta U := T_P^{\delta^-}(O_{k-1} \setminus O_k, U_k, O_k)$ (*)

while $\Delta U \neq \emptyset$ **do begin** *inner fixpoint 2*

$U_{k+1} := U_{k+1} \cup \Delta U$

$\Delta U := T_P^{\delta}(\Delta U, U_{k+1}, O_k) \setminus U_{k+1}$

end

$k := k + 1$

until $[U_k] = [U_{k-1}]$

output $(True, Undef) := (U_{k-1}, O_{k-1} \setminus U_{k-1})$

We illustrate the difference between Algorithms 3.3 and 3.6 by the following example.

Example 3.7 Consider the non-stratified view P

$$win(X) \leftarrow move(X,Y), \mathbf{not}(win(Y))$$

together with the EDB given by the *move*-graph

$$move = \{(a,b),(b,c),(c,a),(a,d),(d,e),(d,f),(f,g),(g,h),(h,i)\}$$

For instance $CGL(P, \{win(a)\}) = \{win(X) \leftarrow Y \neq a \mid move(X,Y), \mathbf{not}(win(Y))\}$. Below, the bars correspond to the alternating sequence of under- and overestimates (which is the same for both algorithms). The steps between the bars indicate the computations in the inner fixpoints. *Note that only the atoms in* **boldface** *are produced by the incremental Algorithm 3.6.* Since $O^{k-1} \setminus O_k = \emptyset$ in the last outer iteration, Algorithm 3.6 does not enter the last inner fixpoint computation.

$$
\begin{array}{rcl}
 & & \boxed{move}\ U_1 \\
\Delta O_{1,1} & = & \boldsymbol{move} \\
\Delta O_{1,2} & = & \boldsymbol{\{win(a), win(b), win(c), win(d), win(f), win(g), win(h)\}} \\
\Delta O_{1,3} & = & \{\} \\
 & & \boxed{move \cup \{win(a), win(b), win(c), win(d), win(f), win(g), win(h)\}}\ O_1 \\
\Delta U_{2,1} & = & \boldsymbol{\{win(d), win(h)\}} \\
\Delta U_{2,2} & = & \{\} \\
 & & \boxed{move \cup \{win(d), win(h)\}}\ U_2 \\
\Delta O_{2,1} & = & move \\
\Delta O_{2,2} & = & \boldsymbol{\{win(a), win(b), win(c),} win(d)\boldsymbol{, win(f),} win(h)\boldsymbol{\}} \\
\Delta O_{2,3} & = & \{\} \\
 & & \boxed{move \cup \{win(a), win(b), win(c), win(d), win(f), win(h)\}}\ O_2 \\
 & (*) & \boldsymbol{O_{k-1} \setminus O_k = \{win(g)\}} \\
\Delta U_{3,1} & = & \{win(d), \boldsymbol{win(f)}, win(h)\} \\
\Delta U_{3,2} & = & \{\} \\
 & & \boxed{move \cup \{win(d), win(f), win(h)\}}\ U_3 \\
\Delta O_{3,1} & = & move \\
\Delta O_{3,2} & = & \{\boldsymbol{win(a), win(b), win(c)}, win(d), win(f), win(h)\} \\
\Delta O_{3,3} & = & \{\} \\
 & & \boxed{move \cup \{win(a), win(b), win(c), win(d), win(f), win(h)\}}\ O_3 \\
 & (*) & \boldsymbol{O_{k-1} \setminus O_k = \{win(g)\}} \\
\Delta U_{4,1} & = & \{win(d), win(f), win(h)\} \\
\Delta U_{4,2} & = & \{\} \\
 & & \boxed{move \cup \{win(d), win(f), win(h)\}}\ U_4
\end{array}
$$

The well-founded model in terms of $\Gamma_P(I)$ is $\Gamma_P^{2k+1} = O_i$ and $\Gamma_P^{2k+2} = U_i$ with $k = 0, 1, \ldots$.

Theorem 4 *Algorithm 3.6 is correct.*

4 Incremental view maintenance

For the problem of view maintenance, we will examine an algorithm that is able to handle set-semantics and duplicate semantics for maintaining a materialized view which has been computed by the algorithms in the previous section. In traditional view maintenance algorithms, typically two kinds of transactions occur:

- **Insertions:** A previously false atom is now inserted into the materialized view.
- **Deletions:** An atom that was true is now deleted from the materialized view.

However, in the case of well-founded semantics, an atom's truth value may have been previously unknown, but the update may now assign a definite value (either true, corresponding to an insertion; or false, corresponding to a deletion) to it now. Furthermore, *after materialization*, atoms which are true may become unknown and vice versa and analogously atoms which have been false may become unknown and vice versa. In all, six possible scenarios exist. The following propositional examples illustrate four of the more interesting cases:

- In $\{a \leftarrow b, b \leftarrow \neg a, a \leftarrow d, d\}$ the deletion of d causes b to become unknown in the new materialized view.
- In $\{a \leftarrow b, b \leftarrow \neg a, a \leftarrow \neg d, d\}$ the deletion of d causes b to become false, whereas previously it was unknown.
- In $\{a \leftarrow \neg b, b \leftarrow a, a \leftarrow d, d\}$ the deletion of d causes b to become unknown, whereas previously it was true.
- In $\{a \leftarrow \neg b, b \leftarrow a, a \leftarrow \neg d, d\}$ the deletion of d results in a new materialized view where b is true whereas in the old view, where b had been unknown.

In the rest of this section, we will first (Section 4.1) define a declarative semantics for insertion and deletion w.r.t. materializations of CVDs. We will then (Section 4.2) develop algorithms that efficiently maintain views. As usual, the declarative semantics serves as the yardstick against which the correctness of these algorithms are measured.

4.1 Declarative Semantics

4.1.1 Semantics of Deletion

Suppose P is a constraint program and let (M, U) be materialized well-founded view where M is the set of true constrained atoms, and U is the set of unknown constrained atoms. Let $A(\vec{X}) \leftarrow \Theta$ be a constrained atom whose instances are to be deleted from M. Let Del be the set $\{A(\vec{Y}) \leftarrow \Xi \mid A(\vec{X}) \leftarrow \Lambda$ is a constrained atom in M and $\Xi = (\Theta \wedge (\vec{X} = \vec{Y}) \wedge \Lambda)$ is satisfiable for some instance of $\vec{X}\}$.

The meaning of deletions is specified by a constraint program P' whose least fixpoint does not contain *any* instances of Del. In the propositional case this simply means that all clauses with an atom which should be deleted in its head are deleted. In the constraint logic case, this amounts to a rewriting of the constraint part of each clause which has an atom of Del in its head. The rewritten program is denoted P^{new}, and is composed of the following two sets of constraint clauses.

1. Constraint clauses of the form
$$\begin{aligned} A(\vec{X}) \quad \leftarrow \quad & \Xi \wedge \neg\Xi' \wedge (\vec{X} = \vec{Y}) \mid B_1(\vec{X}_1), \ldots, B_n(\vec{X}_n), \\ & \mathbf{not}(C_1(\vec{Y}_1)), \ldots \mathbf{not}(C_m(\vec{Y}_m)) \end{aligned}$$
where there is a constraint clause of the form
$$A(\vec{X}) \leftarrow \Xi \mid B_1(\vec{X}_1), \ldots, B_n(\vec{X}_n), \mathbf{not}(C_1(\vec{Y}_1)), \ldots, \mathbf{not}(C_m(\vec{Y}_m))$$
in the original program P such that $A(\vec{Y}) \leftarrow \Xi'$ is a constraint atom in Del and $[A(\vec{X}) \leftarrow \Xi] \cap [A(\vec{Y}) \leftarrow \Xi'] \neq \emptyset$
2. Any other constraint clause from P where the condition of item 1 is not met.

Definition 4.1 (Meaning of Deletion) Suppose P is a constraint program with an associated materialized view (M, U), and $A(\vec{X}) \leftarrow \Theta$ is a constraint atom whose instances are to be deleted from M. Let Del be the set of constraint atom associated with $A(\vec{X}) \leftarrow \Theta$ as defined above. Then the materialized view after deletion is the CWFS associated with the constraint program P^{new}.

4.1.2 Semantics of Insertion

Suppose $A(\vec{X}) \leftarrow \Theta$ is to be inserted into the materialized view (M, U) of a given constraint program. Then, for each $A(\vec{Y}) \leftarrow \Xi$ in M, the atom $A(\vec{X}) \leftarrow \Theta \wedge \neg\Xi$ is placed in a set *Add*. As in the case of deletion, the declarative semantics of view maintenance is given by the least fixpoint of a new constraint program. As we will assume that no simultaneous insertion and deletion of constraint atoms may occur, we can denote, without ambiguity, this new program by overloading the symbol P^{new}. Formally, P^{new} is comprised of the following constraint clauses.

$$\begin{array}{ll} P & \cup \;\; Add \\ & \cup \;\; \{A(\vec{X}) \leftarrow \Xi \wedge \neg\Theta \mid B_1(\vec{X}_1), \ldots, B_n(\vec{X}_n), \mathbf{not}(C_1(\vec{Y}_1)), ..., \mathbf{not}(C_m(\vec{Y}_m)) \mid \\ & \quad\;\; A(\vec{X}) \leftarrow \Gamma \mid B_1(\vec{X}_1), \ldots, B_n(\vec{X}_n), ..., \mathbf{not}(C_m(\vec{Y}_m)) \in P, n > 0 \\ & \quad\;\; A(\vec{X}) \leftarrow \Theta \in M\} \end{array}$$

Definition 4.2 (Meaning of Insertion) Suppose P is a constraint program, (M, U) is its materialized view, and $A(\vec{X}) \leftarrow \Theta$ is a constrained atom whose instances are to be added to M. Let *Add* be the set of constraint atoms associated with $A(\vec{X}) \leftarrow \Theta$ as defined above. Then the new materialized view after the appropriate insertion is the CWFS associated with the constraint program P^{new}.

It is important to emphasize here again that no addition and deletion of constraint atoms may be performed simultaneously. Hence, in the algorithms that we present next, P^{new} will denote either the rewritten program given in Section 4.1.1, or the one given in Section 4.1.2, but not *both*.

4.2 View Maintenance Algorithms

4.2.1 Deletion Algorithms

The following definition adapts the semi-naive computation given in [34] to perform incremental view maintenance in CVDs. When deleting a constraint atom $A(\vec{X}) \leftarrow \Xi$, we use $\delta^-(A(\vec{X})) \leftarrow \Xi$ to denote an overestimate of the instances of $A(\vec{X}) \leftarrow \Xi$ that are to be deleted. Conversely, we use $\delta^+(A(\vec{X})) \leftarrow \Theta$ to represent an underestimate of the atoms to be inserted.

Definition 4.3 (Constrained δ-clauses) Given a constraint program P, we associate, with each constraint clause $\mathcal{C}$, $(n+m)$ δ^--clauses and $(n+m)$ δ^+-clauses as follows. Suppose $\mathcal{C}$ is of the form

$$A(\vec{X}) \leftarrow \Xi | B_1(\vec{X}_1), \ldots, B_n(\vec{X}_n), \mathbf{not}(C_1(\vec{Y}_1)), \ldots, \mathbf{not}(C_m(\vec{Y}_m))$$

The δ^--clauses are as follows.

$$\begin{array}{lll} \delta^-(A(\vec{X})) & \leftarrow & \Xi | B_1^{new}(\vec{X}_1), \ldots, B_{k-1}^{new}(\vec{X}_{k-1}), \delta^-(B_k(X_k)), \\ & & B_{k+1}^{new}(\vec{X}_{k+1}), \ldots, B_n^{new}(\vec{X}_n), \mathbf{not}(C_1(\vec{Y}_1)), \ldots, \mathbf{not}(C_m(\vec{Y}_m)) \\ \delta^-(A(\vec{X})) & \leftarrow & \Xi | B_1^{new}(\vec{X}_1), \ldots, \ldots, B_n^{new}(\vec{X}_n) \\ & & \mathbf{not}(C_1(\vec{Y}_1)), \ldots, \mathbf{not}(C_{l-1}(\vec{Y}_{l-1})), \delta^+(C_l(\vec{Y}_l)), \\ & & \mathbf{not}(C_{l+1}(\vec{Y}_{l+1})), \ldots, \mathbf{not}(C_m(\vec{Y}_m)) \end{array}$$

The δ^+-clauses are as follows.

$$
\begin{array}{rcl}
\delta^+(A(\vec{X})) & \leftarrow & \Xi | B_1^{new}(\vec{X}_1), \ldots, B_{k-1}^{new}(\vec{X}_{k-1}), \delta^+(B_k(X_k)), \\
 & & B_{k+1}^{new}(\vec{X}_{k+1}), \ldots, B_n^{new}(\vec{X}_n), \mathbf{not}(C_1(\vec{Y}_1)), \ldots, \mathbf{not}(C_m(\vec{Y}_m)) \\
\delta^+(A(\vec{X})) & \leftarrow & \Xi | B_1^{new}(\vec{X}_1), \ldots, \ldots, B_n^{new}(\vec{X}_n) \\
 & & \mathbf{not}(C_1(\vec{Y}_1)), \ldots, \mathbf{not}(C_{l-1}(\vec{Y}_{l-1})), \delta^-(C_l(\vec{Y}_l)), \\
 & & \mathbf{not}(C_{l+1}(\vec{Y}_{l+1})), \ldots, \mathbf{not}(C_m(\vec{Y}_m))
\end{array}
$$

The problem of incremental view maintenance in well-founded semantics based materialized views is complicated by the fact that when evaluating the δ clauses using methods similar to the DRed algorithm of Gupta et. al. [16], a (negated) constrained atom in the body of a clause may not have been completely evaluated at that stage. Hence a proper computation of overestimates at once is not feasible. The essential idea of our algorithm is to incrementally maintain Γ_P^k for each $k \geq 1$ by adapting the delete-rederive algorithm [6, 16] developed for safe stratified programs. Since Γ_P^k is the least fixpoint of a *negation-free* program, the Gelfond-Lifschitz transform of P with respect to Γ_P^{k-1} is a negation-free program that can be used to rederive relevant constrained atoms. Furthermore, the application of other techniques for views with a finite number of derivation trees, such as counting [16], become feasible. ¿From an implementation point of view, the required extra bookkeeping is minimal since in addition to maintaining a set of true and unknown atoms, it is necessary only to keep a copy of Γ_P^1 and then to remember when atoms are removed from Γ_P^1 for the first time. Furthermore we need to record the smallest k such that an atom becomes true in Γ^k.

We will develop an algorithm, $\mathbf{WF-RED}$, which performs both insertion and deletion of constraint atoms (cf. Algorithm 4.5). The algorithm works as follows. Given a set, *Del*, of constrained-atoms to be deleted, and a set, *Add*, of constrained-atoms to be inserted,[3] **WF-RED** will, for each stratum defined by the graph $CC(P)$, perform the following three steps.

Step 1: Compute an overestimate of the atoms to be deleted. This overestimate will include atoms that have some syntactic dependency upon atoms in *Del*.

Step 2: Determine which of the atoms that are possibly to be deleted have independent justifications (and hence do not really have to be deleted).[4]

Step 3: Finally, insertions will be perform based on two factors. The first is when certain atoms may have been deleted in previous strata, thus causing atoms to become true by virtue of these deletions. The second is when the insertions may be directly implied by other insertions within the current stratum.

We first present an algorithm, **Red**, which deals only with *negation-free* constraint programs. It will be used for Step 2 in the above sequence by the **WF-RED** algorithm. **Red** takes as input, the materialized view I_P associated with P, and a set of constrained-atoms that are *possibly* to be deleted. It then determines which of these possibly-to-be-deleted atoms can be rederived.

[3] Recall that one of the two sets must be empty since simultaneous insertions and deletions are not allowed.

[4] For example, an atom a may be derivable by two rules $b \rightarrow a$ and $c \rightarrow a$. Had b been in *Del*, then a would appear in the overestimate in Step 1 because a depends on b; however, a is independently supported by the rule $c \rightarrow a$ and hence, it has a justification that is independent of b.

Algorithm 4.4 (Incremental rederivation (Red))

Input: P^{pos} = a *negation-free* constrained program

$I^{pd} = \{\delta^-(A_1(X_1)) \leftarrow \Xi_1, \ldots, \delta^-(A_k(X_k)) \leftarrow \Xi_k\}$, a set of possibly to be deleted constrained atoms

I^{true} a set of true constrained atoms

Output: A set ρ containing those constrained atoms whose instances rederive some of the instances of atoms in I^{pd}.

$\rho := \emptyset$
for all $\delta^-(A_i(\vec{X}_i)) \leftarrow \Xi_i$ in I^{pd} **begin**
  if there is a clause $\mathcal{C} = A_i(\vec{X}_i') \leftarrow \Xi_0 \mid B_1(\vec{X}_1), \ldots, B_{n_i}(\vec{X}_{n_i})$ in P
  then begin
    $\Xi := \Xi_0$
    for all $B_j(\vec{X}_j)$ **begin**
      if there is a $B_j(\vec{X}') \leftarrow \Xi'$ in I^{true} **then**
        $\Xi := \Xi \wedge \Xi' \wedge \{\vec{X}_j = \vec{X}'\}$
      else goto *:label*
    end for
    if $[A_i(\vec{X}_i') \leftarrow \Xi] \cap [A_i(\vec{X}_i) \leftarrow \Xi_i] \neq \emptyset$ **then**
      $\rho := \rho \cup \{\delta^-(A_i(\vec{X}_i')) \leftarrow \Xi\}$
  endif
end for *:label*
return ρ

The condition $[A_i(\vec{X}_i') \leftarrow \Xi] \cap [A_i(\vec{X}_i) \leftarrow \Xi_i]$ implies the satisfiability of the constraint Ξ. In a subsequent step all consequences of atoms obtained from the above algorithm need to be computed using a semi-naive bottom-up evaluation with the δ^+-rules from Definition 4.3.

The full algorithm for incremental view maintenance is shown below. It is important to remember that *either Del* or *Add* is initially empty, which means there are no *simultaneous* update requests allowed.

Algorithm 4.5 (WF-Red Algorithm for Incremental materialized view maintenance)

Input: P = constrained clauses defining the materialized view.

Del = a set of δ^--constrained atoms to be deleted from the materialized view.

Add = a set of δ^+-constrained atoms to be inserted into the materialized view.

$\Gamma_P^1, \Gamma_P^2, \ldots, \Gamma_P^n$ = a sequence of materialized least fixpoints of the Gelfond-Lifschitz transformed programs (inner fixpoints).

Output: The well-founded materialized view $WFM_{P^{new}}$ = (true,unknown), of P^{new}.

$\Delta^- := Del, \Delta^+ := Add$ % *Either* $Del = \emptyset$ *or* $Add = \emptyset$ k

$k := 1$

$I_0 := \emptyset, J_0 := \emptyset$

P_{δ^-} := The set of constrained δ^--clauses obtained from P according to Definition 4.3

$P_{\delta^+}^{new}$:= The set of constrained δ^+-clauses obtained from P^{new} according to Definition 4.3

$P^{new}_{\delta^-}$:= The set of constrained δ^--clauses obtained from P^{new} according to Definition 4.3

repeat
  If $k \leq n$ **then** *% Outer fixpoint already reached ?*
    $I_k := \Gamma^k_P(\emptyset)$
  else $I_k := \Gamma^n_P(\emptyset)$

% Compute overestimate of atoms to be deleted
$P_{CGL} := CGL(P_{\delta^-}, I_{k-1})$
$\Delta^- := Del \cup$ An overestimate of atoms to be deleted obtained by a *semi-naive* bottom-up evaluation of P_{CGL} with $\delta^-(A(\vec{X})) \leftarrow \Xi \in \Delta^-, \delta^+(B(\vec{Y})) \leftarrow \Theta \in \Delta^+$

% Rederive
$P_{CGL} := CGL(P^{new}_{\delta^-}, J_{k-1})$
$BRed := \mathbf{Red}(\Delta^- \setminus Del, CGL(P^{new}, J_{k-1}), I_k)$
Red := All atoms to be re-inserted by a *semi-naive* bottom- up evaluation of P_{CGL} with $\delta^-(A(\vec{X})) \leftarrow \Xi \in BRed \cap \Delta^-$
$\Delta^- := \Delta^- \setminus (Red \cup BRed)$

% Insert atoms becoming true due to atoms deleted in Γ^{k-1}_P
$P_{CGL} := CGL(P^{new}_{\delta^+}, J_{k-1})$
Δ^+ := All atoms to be inserted by a *semi-naive* bottom-up evaluation of P_{CGL} with $\delta^-(A(\vec{X})) \leftarrow \Xi \in \Delta^-$, $\delta^+(B(\vec{Y})) \leftarrow \Theta \in \Delta^+$

  $\Gamma^k_{P^{new}} := \Gamma^k_P \setminus \Delta^- \cup \Delta^+$ % $\Gamma^k_{P^{new}} := T_{P_{CGL}} \uparrow \omega(\Gamma^k_P \setminus \Delta^-)$ (*)
  $J_k := \Gamma^k_{P^{new}}(\emptyset)$
  $k := k + 1$
until $J_{k-1} = J_{k-2}$ **or** ($k \geq 3$ **and** $J_{k-1} = J_{k-3}$)

output $(True, Undef) := (J_{k-2}, J_{k-1} \setminus J_{k-2})$

In the above algorithm, the sets Δ^- and Δ^+ represent the difference between the inner fixpoint Γ^k_P and $\Gamma^k_{P^{new}}$. Therefore, when computing the new Δ^- and Δ^+ sets using a semi-naive bottom-up evaluation, the old $\delta^{+/-}$-base facts are not to be included in the new Δ^-, Δ^+ (in contrast to the original semi-naive bottom-up evaluation). This is also the reason why the set BRed is explicitly added to the set of consequences of atoms in BRed. It should also be noted that the case when both sets Δ^- and Δ^+ are non-empty occurs only for $k > 1$. In particular there is never a mutual dependency between atoms in these sets. Therefore the algorithm expects as input *Del either* a set of atoms to be deleted *or* a set of atoms to be inserted and is not capable of simultaneous updates. In the line marked with (*) $\Gamma_{P^{new}}$ is computed from sets of $\delta^{+/-}$-atoms omitting $\delta^{-/+}$.

Example 4.6 (Win-Move Revisited) The following four δ-clauses are generated according to Definition 4.3:

$$\delta^-(win(X)) \leftarrow \delta^-(move(X,Y)), \neg win(Y) \quad (4)$$
$$\delta^-(win(X)) \leftarrow move^{new}(X,Y), \delta^+(win(Y)) \quad (5)$$

$$\delta^+(win(X)) \leftarrow \delta^+(move(X,Y)), \neg win(Y) \quad (6)$$
$$\delta^+(win(X)) \leftarrow move^{new}(X,Y), \delta^-(win(Y)) \quad (7)$$

The algorithm starts with $\Delta^- = \{\delta^-(move(a,b))\}, \Delta^+ = \emptyset$. Bold-faced atoms are the atoms that are new in a materialized view.

$$\begin{array}{lrl}
k=1 \ : & CGL(P_{\delta^-}, \emptyset) & = \{\ \delta^-(win(X)) \leftarrow \delta^-(move(X,Y)), \\
& & \quad \delta^-(win(X)) \leftarrow move^{new}(X,Y), \delta^+(win(Y))\ \} \\
& \Delta^- & = \{\ \delta^-(win(a)), \delta^-(\mathbf{move(a,b)})\ \} \setminus \{\delta^-(win(a))\ \}, \\
& & \quad \text{because of } move(a,d) \\
& \Delta^+ & = \emptyset \\
k=2 \ : & CGL(P_{\delta^-}, \Gamma_P^1) & = \{\ \delta^-(win(X)) \leftarrow Y \neq a \wedge Y \neq b \neq Y \wedge Y \neq c \wedge \\
& & \quad Y \neq d \wedge Y \neq f \wedge Y \neq g \wedge Y \neq h \mid \delta^-(move(X,Y)), \\
& & \quad \delta^-(win(X)) \leftarrow move^{new}(X,Y), \delta^+(win(Y))\ \} \\
& \Delta^- & = \{\ \delta^-(\mathbf{move(a,b)})\ \} \\
& \Delta^+ & = \emptyset \\
k=3 \ : & CGL(P_{\delta^-}, \Gamma_P^2) & = \{\ \delta^-(win(X)) \leftarrow Y \neq d \wedge Y \neq h \mid \delta^-(move(X,Y)), \\
& & \quad \delta^-(win(X)) \leftarrow move^{new}(X,Y), \delta^+(win(Y))\ \} \\
& \Delta^- & = \{\ \delta^-(\mathbf{win(a)}), \delta^-(\mathbf{move(a,b)})\ \} \\
& \Delta^+ & = \emptyset \\
k=4 \ : & CGL(P_{\delta^-}, \Gamma_P^2) & = \{\ \delta^-(win(X)) \leftarrow Y \neq b \wedge Y \neq c \neq Y \wedge Y \neq d \wedge \\
& & \quad Y \neq f \wedge Y \neq h \mid \delta^-(move(X,Y)), \\
& & \quad \delta^-(win(X)) \leftarrow move^{new}(X,Y), \delta^+(win(Y))\ \} \\
& \Delta^- & = \{\ \delta^-(\mathbf{move(a,b)})\ \} \\
& CGL(P_{\delta^+}^{new}, \Gamma_{P^{new}}^3) & = \{\ \delta^+(win(X)) \leftarrow Y \neq b \wedge Y \neq c \neq Y \wedge Y \neq d \wedge \\
& & \quad Y \neq f \wedge Y \neq h \mid \delta^+(move(X,Y)), \\
& & \quad \delta^+(win(X)) \leftarrow move^{new}(X,Y), \delta^-(win(Y))\ \} \\
& \Delta^+ & = \{\ \delta^+(\mathbf{win(c)})\ \} \\
k=5 \ : & CGL(P_{\delta^-}, \Gamma_P^2) & = \{\ \delta^-(win(X)) \leftarrow Y \neq c \wedge Y \neq d \wedge \\
& & \quad Y \neq f \wedge Y \neq h \mid \delta^-(move(X,Y)), \\
& & \quad \delta^-(win(X)) \leftarrow move^{new}(X,Y), \delta^+(win(Y))\ \} \\
& \Delta^- & = \{\ \delta^-(\mathbf{win(a)}), \delta^-(\mathbf{win(b)}), \delta^-(\mathbf{move(a,b)})\ \} \\
& \Delta^+ & = \emptyset
\end{array}$$

The solution returned by the algorithm is $(\Gamma_P^5 \setminus \{move(a,b), win(a), win(b)\}) = (move \setminus \{move(a,b)\} \cup \{win(c), win(d), win(f), win(h)\}, \emptyset)$. The algorithm is incremental in two different ways. First, the inner fixpoints Γ_{Pnew}^k are maintained by the delete-rederive technique. Second the algorithm propagates the differences between Γ_P^k and Γ_{Pnew}^k incrementally across the outer fixpoints by means of the δ-clauses of Definition 4.3. For instance, when $k=3$, $\delta^-(win(a))$ is computed – in the next iteration of the outer-fixpoint computation, this causes the insertion of $\delta^+(win(c))$. Intuitively, whenever an atom gets deleted, the Gelfond-Lifschitz transform in the next iteration may delete *fewer* clauses from the program, and hence, some new atoms may be derived in the next step. When the view contains duplicates, we may use the above algorithm without any changes except that the box which contains the rederivation part may be dropped.

Theorem 5 *The* **WF-Red** *is correct, i.e.*

1. *If it is invoked to accomplish a deletion (i.e. $Add = \emptyset$ and $Del \neq \emptyset$), then the solution $WFM_{P^{new}}$ returned by* **WF-Red** *concides with semantics of Definition 4.1.*

2. *If it is invoked to accomplish an insertion (i.e. $Add \neq \emptyset$ and $Del = \emptyset$), then the solution $WFM_{P^{new}}$ returned by* **WF-Red** *concides with semantics of Definition 4.2.*

5 Conclusions and Related Work

Whenever an update occurs to a view, whether materialized or not, the problem of how to reflect the update correctly by changing the base tables appropriately is called the *view update problem* and has been discussed extensively for relational [3, 8, 10], deductive [3, 26, 32, 33], and object-oriented [9, 25, 29] databases. In particular, the view itself — or, to be more precise, its definition — is affected by the update. This kind of update affecting the view's definition is typically not treated within the view update literature. One exception are deductive databases, where the addition or deletion of rules to the definition of an intensional predicate is discussed [32]. However, they neither materialize nor preprocess the view for efficiency reasons. In contrast, when efficiency is desired, we are often interested in *materializing* the view so as to store frequently asked queries (and hence answer such queries efficiently without having to resort to a complex sequence of relational algebra and/or deductive operations). The resulting problem — preserving the consistency of the view — is called *view maintenance* and has been discussed for, e.g., for (extended) relation [5, 17, 31] and deductive databases [22, 18, 15, 35, 30].

Though the above cited references comprehensively address the view maintenance problem in the setting of relational databases and in the case of object-oriented databases [9, 21], *there has been very little work when view definitions can contain well-founded modes of negation in them* – in particular, such efforts have been limited to stratified negation [16]. In contrast, our approach deals with view maintenance when view definitions can include not just stratified, but also unstratified negation. The examples given in this paper (program P at the beginning of Section 3 and the Win-Move example) involve non-stratified negation as well as recursive view definitions. Furthermore, our notion of a *materialized view* allows *constrained-tuples* to occur in materialized views. This is particularly important for view maintenance is situations where, for instance, $\mathtt{p(X)}$ is true for all $\mathtt{X} \geq \mathtt{3}$ – in such a case, the constrained tuple $\mathtt{p(X)} \leftarrow \mathtt{X} \geq \mathtt{3}$ forms a succinct representation of an infinite set of ground-tuples.

One use of such constraints for view maintenance in heterogeneous mediated environments was studied in [24, 23]. This paper extends the [23] framework to apply to constrained view definitions containing well-founded negation in them. Furthermore, constraint databases have been applied to several other domains such as temporal reasoning and spatial reasoning. Our techniques are therefore applicable in principle to these paradigms as well.

References

[1] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 73–84, 1993.

[2] K. Apt and H. Blair and A. Walker. (1988) *Towards a Theory of Declarative Knowledge*, in: Foundations of Deductive Databases and Logic Programming (ed. J. Minker), pps 89–148, Morgan Kaufmann.

[3] F. Bancilhon and N. Spyratos. Update semantics and relational views. *ACM Trans. on Database Systems*, 6(4):557–575, 1981.

[4] C. Baral and V.S. Subrahmanian. Stable and Extension Class Theory for Logic Programs and Default Theories, *Journal of Automated Reasoning*, 8, pps 345–366, 1992.

[5] J. Blakeley, N. Coburn, and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, 1989.

[6] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *17th VLDB*, 1991.

[7] Stefano Ceri and Jennifer Widom. Deriving Incremental Production Rules for Deductive Data. IBM RJ 9071, IBM Almaden, 1992.

[8] C.C. Cosmadakis and C.H. Papadmitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984.

[9] U. Dayal. Queries and views in a object-oriented databases. In *Int. Workshop on Database Programming Languages*, 1989.

[10] U. Dayal and P. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. on Database Systems*, 7(3), 1982.

[11] U. Dayal and H. Hwang. View definition and generalization for database integration in a multi-database system. *IEEE Trans. Software Eng.*, SE-10(6):628–644, 1984.

[12] T. Eiter, G. Gottlob, J. Lu and V.S. Subrahmanian. (1994) *A Constraint-based Representation of Stable and Well-Founded Semantics*, Tech. Report, TU-Wien.

[13] M. Gelfond and V. Lifschitz. (1988) *The Stable Model Semantics for Logic Programs*, Proc. 1988 Intl. Conf. on Logic Programming (eds. K. Bowen and R. Kowalski), MIT Press.

[14] G. Gottlob, S. Marcus, A. Nerode and V.S. Subrahmanian. (1994) *Non-Ground Stable and Well-Founded Semantics*, to appear in Theoretical Computer Science.

[15] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting Solutions to the View Maintenance Problem. In *Workshop on Deductive Databases, JICSLP*, 1992.

[16] A. Gupta, I.S. Mumick and V.S. Subrahmanian. Maintaining Views Incrementally, *Proc. 1993 ACM SIGMOD Conf. on Management of Data*, Washington, DC.

[17] E. Hanson. A performance analysis of view materialization strategies. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 440–453, 1987.

[18] John V. Harrison and Suzanne Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICSLP*, 1992.

[19] J. Jaffar, J.-L. Lassez. *Constraint Logic Programming* In. Proceedings of Fourteenth Annual ACM Symposium on Principles of Programming Languages, pp. 111-119. ACM, New York, USA, 1987

[20] P. Kanellakis, G. Kuper and P. Revesz. (1990) *Constraint Query Languages*, Proc. 9th ACM Symp. on Principles of Database Systems, pps 299-313.

[21] A. Kemper, C. Kilger, G. Moerkotte. *Function Materialization in Object Bases: Design, Realization, and Evaluation* IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.4, August 1994

[22] V. Küchenhoff. On the efficient computation of the difference between consecutive database states. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 478–502, 1991.

[23] J. Lu, G. Moerkotte, J. Schü and V.S. Subrahmanian. (1995) Efficient Maintenance of Materialized Mediated Views, *Proceedings of the ACM SIGMOD Annual Conference*, May 1995, San Jose, CA.

[24] J. Lu, A. Nerode and V.S. Subrahmanian. (1993) Hybrid Knowledge Bases, *IEEE Transactions on Knowledge and Data Engineering*, to appear.

[25] C. Medeiros and J.-C. Mamou. Interactive manipulation of object-oriented views. In *Proc. IEEE Conference on Data Engineering*, 1991.

[26] G. Moerkotte and P.C. Lockemann. Reactive consistency control in deductive databases. *ACM Trans. on Database Systems*, 16(4):670–702, 1991.

[27] A. Motro. Superviews: Virtual integration of multiple databases. *IEEE Trans. Software Eng.*, 13(7):785–798, 1987.

[28] I. S. Mumick. *Query Optimization in Deductive and Relational Databases.* Ph.D. Thesis, Stanford University, CA 94305, 1991.

[29] M. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 1991.

[30] Oded Shmueli and Alon Itai. *Maintenance of Views.* In *Sigmod Record*, 14(2):240-255, 1984.

[31] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 281–290, 1990.

[32] E. Teniente. Updating knowledge bases while maintaining their consistency. *VLDB Journal*, 1994. to appear.

[33] A. Tomasic. View update translation via deduction and annotation. In *ICDT*, pages 338–352, 1988.

[34] J. D. Ullman. (1989) *Principles of Database and Knowledge-Base Systems, Volume II*, Computer Science Press, New York.

[35] Toni Urpi and Antoni Olive. A Method for Change Computation in Deductive Databases. In *18th VLDB*, pages 225–237, 1992.

[36] A. Van Gelder and K. Ross and J. Schlipf. *The Well-Founded Semantics for General Logic Programs*, Journal of the ACM, 38, 3,pps 620–650.