View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Analysing the Process of Enforcing Integrity Constraints

Enric Mayol
Ernest Teniente

Universitat Politècnica de Catalunya
Facultat d'Informàtica
Pau Gargallo 5
E-08028 Barcelona - Catalonia
e-mail: [mayol | teniente]@lsi.upc.es

Abstract

Two different approaches have been traditionally considered for dealing with the process of integrity constraints enforcement: integrity constraints checking and integrity constraints maintenance. However, while previous research in the first approach has mainly addressed efficiency issues, research in the second approach has been mainly concentrated in being able to generate all possible repairs that falsify an integrity constraint violation. Moreover, the methods proposed up to date are only concerned with handling one of the approaches in an isolated manner, without taking into account the strong relationship between the problems to be solved in both cases.

In this paper we address efficiency issues during the process of integrity constraints maintenance. In this sense, we propose a technique which improves efficiency of existing methods by defining the order in which maintenance of integrity constraints should be performed. Moreover, we use also this technique for being able to handle in an integrated way the integrity constraints enforcement approaches mentioned above.

KEYWORDS: deductive database, updating, integrity checking, integrity maintenance

June 1996

1. Introduction

Deductive database updating has attracted a lot of research during last years (see for example [Abi88, Win90]). In general, several problems may arise when updating a deductive database. One of the most important problems is that of *enforcing database consistency*. A deductive database is called consistent if it satisfies a set of integrity constraints. When performing an update, deductive database consistency may be violated. That is, the update, together with the current content of the deductive database, may falsify some integrity constraint.

The classical approach to deal with this problem has been to develop methods for *checking* whether a given update violates an integrity constraint (see [Oli91, CGMD94] and the references therein). When a violation is detected, the transaction is rolled back in its entirety. That is, the update request is rejected and, in this case, the user intention cannot be satisfied. We will refer to this approach as *integrity checking*.

In some cases, this solution may not be satisfactory because the user may not know which additional changes are needed in order to satisfy all the integrity constraints. Then, a second approach for dealing with integrity constraints satisfaction [CW90, KM90, ML91, CFPT92, Wüt93, TO95, Dec96] consists of trying to repair constraints violations by performing additional updates that restore consistency of the deductive database. In this case, it is guaranteed that the state resulting from applying the update does not violate any integrity constraint and that it satisfies the update requested by the user. We will refer to this approach as *integrity maintenance*.

Up to the present, the main effort of the research in integrity maintenance has been devoted to define methods for handling repairs in an effective way. These methods are mainly concerned with being able to generate all possible repairs when an integrity constraint violation is detected. However, little attention has been paid to efficiency issues (even though efficiency is known to be one of the most important factors of success for practical databases).

Both integrity constraint enforcement policies, integrity checking as well as integrity maintenance, are reasonable [Win90]. The correct choice of a policy for a particular integrity constraint depends on the semantics of the integrity constraint and of the deductive database. Thus, a Deductive Database Management System should allow to define constraints to be checked as well as constraints to be maintained, and should also be able to handle them appropriately. However, the integration of both integrity constraint enforcement policies is not an easy task since most of the existing methods are only concerned with handling one of the policies in an isolated manner, without taking into account the strong relationship between the problems to be solved in both cases.

This paper aims at improving integrity constraints enforcement by considering the two aspects just mentioned before. First, we propose a technique for improving efficiency of existing integrity

maintenance methods by defining the order in which maintenance of integrity constraints should be performed. This technique is based on the definition of a graph which explicitly states all relationships between integrity constraints and repairs. Our technique is directly applicable to the methods we have proposed in the past for updating consistent deductive databases [MT95, TO95] and it could be be easily adapted for improving efficiency of most of the existing integrity constraints maintenance methods. Second, we present an approach for integrating the treatment of integrity constraints to be checked and integrity constraints to be maintained. This approach is based on incorporating also in the previous graph the information corresponding to the integrity constraints to be checked, and considering its relationship with constraints to be maintained and their corresponding repairs.

Our approach is based on a set of rules (proposed in [Oli91, UO92]) that define the precise difference between two consecutive database states, by explicitly stating the exact insertions, deletions and modifications induced by the application of a transaction. We will use these rules for obtaining the graph which contains the relationships between constraints and repairs.

Our previous work in the field has been mainly devoted to the definition of a sound and complete method for updating deductive databases while maintaining their consistency [TO95] and on considering efficiency issues on the treatement of view updates [MT93, MT95]. This paper extends our previous work by considering efficiency issues in the treatement of integrity constraints maintenance and also by proposing an approach for incorporating in a single method integrity checking and integrity maintenance.

This paper is organised as follows. Next section reviews basic concepts of deductive databases. Section 3, which is based on [UO92], reviews the concepts of event, transition rules and event rules. In Section 4 we propose to use a graph to maintain integrity constraints in an efficient way. We distinguish the case that integrity constraints are defined by only base predicates (section 4.1) and integrity constraints defined by either base and derived predicates (section 4.2). In Section 5 we propose a mechanism to execute that graph. Section 6 describes a proposal to combine integrity constraints checking and integrity constraints maintenance policies. Finally, at Section 7 we relate our approach to other relevant previous work and in Section 8 we summarise our conclusions.

2. Deductive Databases

In this section, we briefly review some definitions of the basic concepts related to deductive databases [Llo87, Ull88] and present our notation. Throughout the paper, we consider a first order language with a universe of constants, a set of variables, a set of predicate names and no function symbols. We will use names beginning with a capital letter for predicate symbols and constants (with the exception that constants are also permitted to be numbers) and names beginning with a lower case letter for variables.

A *term* is a variable symbol or a constant symbol. If P is an m-ary predicate symbol and t_1, ..., t_m are terms, then P(t_1, ..., t_m) is an *atom*. The atom is *ground* if every t_i (i = 1, …, m) is a constant. A *literal* is defined as either an atom or a negated atom.

A *fact* is a formula of the form: P(t_1, ..., t_m) ← , where P(t_1, ..., t_m) is a ground atom.

A *deductive rule* is a formula of the form:

$$P \leftarrow L_1 \wedge ... \wedge L_n \qquad \text{with } n \geq 1$$

where P is an atom denoting the conclusion, and L_1,...,L_n are literals representing conditions. Any variable in P, L_1, ..., L_n is assumed to be universally quantified over the whole formula. A derived predicate P may be defined by means of one or more deductive rules.

An *integrity constraint* is a closed first-order formula that the deductive database is required to satisfy. We deal with constraints in *denial* form:

$$\leftarrow L_1 \wedge ... \wedge L_n \qquad \text{with } n \geq 1$$

where the L_i are literals and all variables are assumed to be universally quantified over the whole formula. More general constraints can be transformed into this form by first applying the range form transformation [Dec89] and then using the procedure described in [LT84].

For the sake of uniformity, we associate to each integrity constraint an inconsistency predicate Ic_n, with or without terms, and thus they have the same form as the deductive rules. We call them *integrity rules*. Then, we rewrite the former denial as:

$$Ic_n \leftarrow L_1 \wedge ... \wedge L_m \quad \text{with } m \geq 1$$

We assume that each predicate (base or derived) has a non-null vector of arguments, **k**, that form a key for that predicate. We have then two types of predicates: those, P(**k**,**x**), with key and non-key arguments and those, P(**k**), with only key arguments; where both **k** and **x** are vectors.

To enforce the concept of key we assume that associated to each P(k,x) there is a key integrity constraint that we define as: ← P(k,x) ∧ P(k,x') ∧ x ≠ x'. Underlined arguments of predicates will correspond to their key arguments. Key integrity constraints do not need to be explicitly defined since they are implicitly handled by our update method.

A *deductive database* D is a triple (EDB,IDB,IC), where EDB is a set of facts, IDB a set of deductive rules and IC a set of integrity constraints. The set EDB of facts is called the *extensional* part of the database and the set of deductive rules and integrity constraints is called the *intensional* part.

In this paper we assume that key arguments of predicates appearing in the body of a deductive rule are a subset of key arguments of the predicate defined by that rule. That is, we deal with the *universal key case*.

We also assume that deductive database predicates are partitioned into base and derived (view) predicates. A base predicate appears only in the extensional part and (eventually) in the body of deductive rules. A derived predicate appears only in the intensional part. Any database can be defined in this form [BR86]. We deal with *stratified* databases [Llo87] and, as usual, we require the database to be *allowed* [Llo87]; that is, any variable that occurs in a deductive rule has an occurrence in a positive condition of an ordinary predicate.

3. The Augmented Database [UO92]

Our approach to improving efficiency of current integrity maintenance methods is based on a set of rules that define the difference between two consecutive database states. This set of rules together with the original database D compose the Augmented Database [UO92]. The Augmented Database explicitly defines the insertions, deletions and modifications induced by a transaction consisting of a set of updates to the extensional part of the database. In this section, we will review the main concepts of the Augmented Database. We refer the reader to [UO92] for a further description of these concepts.

The definition of the Augmented Database is strongly based on the concept of *event*. For each predicate P of a given deductive database D, a distinguished *insertion event predicate ιP*, *deletion event predicate δP*, and *modification event predicate μP* are used to define the precise difference of deducible facts of consecutive database states.

More precisely, rules about ιP, δP and μP in the Augmented Database (called *event rules*) define exactly the facts about P that are effectively inserted, deleted or modified in the extension of P by some transaction T. The definition of ιP, δP and μP depends on the definition of P in D, but it is independent of any transaction T and of the extensional part of D. A more formal declarative definition of ιP, δP and μP is given by the following equivalences:

$$\forall \mathbf{k},\mathbf{x}\ (\iota P(\underline{\mathbf{k}},\mathbf{x}) \leftrightarrow P^n(\underline{\mathbf{k}},\mathbf{x}) \wedge \neg\, \exists \mathbf{y} P^o(\underline{\mathbf{k}},\mathbf{y}))$$
$$\forall \mathbf{k},\mathbf{x}\ (\delta P(\underline{\mathbf{k}},\mathbf{x}) \leftrightarrow P^o(\underline{\mathbf{k}},\mathbf{x}) \wedge \neg\, \exists \mathbf{y} P^n(\underline{\mathbf{k}},\mathbf{y}))$$
$$\forall \mathbf{k},\mathbf{x},\mathbf{x}'(\mu P(\underline{\mathbf{k}},\mathbf{x},\mathbf{x}') \leftrightarrow P^o(\underline{\mathbf{k}},\mathbf{x}) \wedge P^n(\underline{\mathbf{k}},\mathbf{x}') \wedge \mathbf{x} \neq \mathbf{x}')$$

where P^o refers to predicate P evaluated in the old state of the database (before the application of T), P^n refers to predicate P evaluated in the new state of the database and **k**, **x**, **y** and **x'** are vectors of variables.

If P is a base predicate, ιP, δP and μP facts represent insertions, deletions and modifications of base facts, respectively. Therefore, we assume from now on that a transaction T consists of an unspecified set of base event facts. If P is a derived predicate, ιP, δP and μP facts represent induced

insertions, induced deletions and induced modifications, respectively. If P is an inconsistency predicate, then ιP facts represent violations of an integrity constraint. For inconsistency predicates, δP and μP facts are not defined since we assume that the database is consistent before the update.

For instance, if Std(st-id,name) is a predicate (denoting that the student identified by st-id is named name), ιStd(1,John) denotes an insertion event corresponding to predicate Std: Std(1,John) is true after the application of T and it was false before. On the other hand, δStd(2,Mary) denotes a deletion event: Std(2,x) is false for all possible values of x after the application of T and Std(2,Mary) was true before. Finally, μStd(2,Mary,Sue) denotes a modification event: Std(2,Mary) was true before the application of T while Std(2,Sue) is true after this application.

Given a deductive database D = (EDB, IDB, IC), the Augmented Database associated to D is a triple A(D) = (EDB, IDB*, IC*), where IDB* contains deductive rules of D and event rules associated to these deductive rules, while IC* contains the integrity rules of D and their associated insertion event rules.

A more precise description and discussion of the procedure for automatically deriving an Augmented Database from database clause definitions can be found in [UO92]. The following example illustrates the concept of Augmented Database.

Example 3.1: Consider the following deductive database which contains three base predicates P(k,x), T(k,x), R(k,x) and two integrity constraints Ic1(k,x) and Ic2(k,x).

$$Ic1(k,x) \leftarrow P(k,x) \wedge \neg T(k,x)$$
$$Ic2(k,x) \leftarrow R(k,x) \wedge \neg P(k,x)$$

Integrity constraint Ic1(k,x) states that facts of predicate P(k,x) can not hold if related facts T(k,x) do not also hold. While in a similar way, integrity constraint Ic2(k,x) prevents to be true a fact of predicate R(k,x) to be false the associated fact P(k,x).

If we apply the definition of insertion event rules to predicates Ic1 and Ic2, we get the following equivalences:

$$\forall k,x\ (\iota Ic1(k,x) \leftrightarrow Ic1^n(k,x) \wedge \neg \exists y\ Ic1^o(k,y))$$
$$\forall k,x\ (\iota Ic2(k,x) \leftrightarrow Ic2^n(k,x) \wedge \neg \exists y\ Ic2^o(k,y))$$

And after simplifying these two rules by applying the procedure defined in [UO92], we get the following insertion event rules of our example:

(C_1) $\iota Ic1\ (k, x) \leftarrow P(k, x) \wedge \neg \delta P(k, x) \wedge \neg \mu P(k, x, a) \wedge \delta T(k, x)$
(C_2) $\iota Ic1\ (k, x) \leftarrow P(k, x) \wedge \neg \delta P(k, x) \wedge \neg \mu P(k, x, a) \wedge \mu T(k, x, b)$
(C_3) $\iota Ic1\ (k, x) \leftarrow \iota P(k, x) \wedge \neg T(k, x) \wedge \neg \iota T(k, x) \wedge \neg \mu T(k, a, x)$
(C_4) $\iota Ic1\ (k, x) \leftarrow \iota P(k, x) \wedge \delta T(k, x)$

(C_5) $\iota Ic1\ (\underline{k}, x) \leftarrow \iota P(\underline{k}, x) \wedge \mu T(\underline{k}, x, a)$
(C_6) $\iota Ic1\ (\underline{k}, x) \leftarrow \mu P(\underline{k}, a, x) \wedge \neg\ T(\underline{k}, x) \wedge \neg\ \iota T(\underline{k}, x) \wedge \neg\ \mu T(\underline{k}, b, x)$

(C_7) $\iota Ic2\ (\underline{k}, x) \leftarrow R(\underline{k}, x) \wedge \neg\ \delta R(\underline{k}, x) \wedge \neg\ \mu R(\underline{k}, x, a) \wedge \delta P(\underline{k}, x)$
(C_8) $\iota Ic2\ (\underline{k}, x) \leftarrow R(\underline{k}, x) \wedge \neg\ \delta R(\underline{k}, x) \wedge \neg\ \mu R(\underline{k}, x, a) \wedge \mu P(\underline{k}, x, b)$
(C_9) $\iota Ic2\ (\underline{k}, x) \leftarrow \iota R(\underline{k}, x) \wedge \neg\ P(\underline{k}, x) \wedge \neg\ \iota P(\underline{k}, x) \wedge \neg\ \mu P(\underline{k}, a, x)$
(C_{10}) $\iota Ic2\ (\underline{k}, x) \leftarrow \iota R(\underline{k}, x) \wedge \delta P(\underline{k}, x)$
(C_{11}) $\iota Ic2\ (\underline{k}, x) \leftarrow \iota R(\underline{k}, x) \wedge \mu P(\underline{k}, x, a)$
(C_{12}) $\iota Ic2\ (\underline{k}, x) \leftarrow \mu R(\underline{k}, a, x) \wedge \neg\ P(\underline{k}, x) \wedge \neg\ \iota P(\underline{k}, x) \wedge \neg\ \mu P(\underline{k}, b, x)$

Rules C_1 to C_{12} define all possible ways of inserting facts about predicate Ic1($\underline{k}$,x) and Ic2($\underline{k}$,x). These rules deserve special attention since they define all possible situations in which database consistency is violated by the application of some transaction. For instance, rule C_9 states that database consistency will be violated if a transaction T inserts a fact R($\underline{K}$, X) and no insertion nor modification is performed by transaction T about the fact P($\underline{K}$, X). It is not difficult to see that database consistency would be violated by the application of T since in the new state fact R($\underline{K}$, X) would be true while fact P($\underline{K}$, X) would be false.

Due to the absense of derived predicates, no insertion, deletion nor modification event rule of derived predicate appears in this example. An example of these event rules will be given in Section 4.2

4. Structuring the Process of Integrity Constraints Maintenance

In general, integrity constraints of a database are very interrelated because they have some predicates in common. These predicates shared among several integrity constraints may appear explicitly in their definition as well as implicitly because they participate in the definition of a certain derived predicate that appears explicitly in that definition. For this reason, the consistency maintenance activity uses to be very complex since, for instance, repairs of an integrity constraint may correspond to violations of other integrity constraints; or since an already repaired integrity constraint could be violated again by the repair of another integrity constraint. This situation is aggravated by the fact that even simple integrity constraints can be violated through several operations and also because often a multitude of repair actions exists.

The methods proposed so far for integrity constraints maintenance [KM90, ML91, CFPT92, Wüt93, TO95, Dec96] have been mainly concerned with the generation of a complete set of repairs of integrity constraints violations, but little attention has been paid in the past to efficiency issues. Thus, for instance, when a constraint is repaired all other constraints are checked for consistency even though they were already satisfied prior to the repair and they could not be violated by the performed repair.

In this section we propose a technique for determining the order in which integrity constraints should be handled to minimize the number of times that an integrity constraint must be reconsidered by an integrity constraints maintenance method. This minimization provides two important advantages. First, it is useful for minimizing the number of recomputations of whether a given constraint is violated, minimizing in this way its associated cost. Second, and most important one, it is helpful for ensuring that a repair of a certain integrity constraint Icj is performed only when it is guaranteed that have been performed all repairs of other constraints that could induce a violation of Icj.

Our technique is based on the definition of a graph, the *Precedence Graph*, which explicitly states all relationships between repairs and potential violations of integrity constraints. Information provided by this graph is directly applicable to the methods we have proposed in the past for handling consistent updates in deductive databases (reported in [MT93, MT95, TO95]) and it could be easily adapted to be applicable to the other existing methods.

To obtain the Precedence Graph we only need to take into account syntactical information associated to the definition of each integrity constraint and, thus, we do not need to consider the contents of the EDB nor the transaction to be applied to the database. Therefore, we generate the Precedence Graph at definition time, and we delay to run time to test whether potential dependencies defined in the graph correspond to real violations.

As we said in the previous section, we assume that the database is consistent before the application of a transaction T. Then, violations of database consistency due to the transaction are produced because some insertion event rule associated to an integrity constraint becomes true. Moreover, repairs of the constraint are defined by the violated insertion event rule, since a repair corresponds to an additional update that will falsify the efect of T on the corresponding event rule. For this reason, in the rest of the paper we will refer to the insertion event rules of an integrity constraint as the *conditions* of that integrity constraint.

In order to state dependencies between integrity constraints more precisely, we will consider the conditions associated to an integrity constraint instead of the own integrity constraint definition. Thus, the Precedence Graph will state all relationships between repairs and potential violations of these conditions.

Example 4.1: Conditions associated to integrity constraint Ic1 of the example 3.1 are the following:

Name	Condition
C_1	$\leftarrow P(\underline{k}, x) \wedge \neg \delta P(\underline{k}, x) \wedge \neg \mu P(\underline{k}, x, a) \wedge \delta T(\underline{k}, x)$
C_2	$\leftarrow P(\underline{k}, x) \wedge \neg \delta P(\underline{k}, x) \wedge \neg \mu P(\underline{k}, x, a) \wedge \mu T(\underline{k}, x, b)$

C_3	$\leftarrow \iota P(\underline{k}, x) \wedge \neg T(\underline{k}, x) \wedge \neg \iota T(\underline{k}, x) \wedge \neg \mu T(\underline{k}, a, x)$
C_4	$\leftarrow \iota P(\underline{k}, x) \wedge \delta T(\underline{k}, x)$
C_5	$\leftarrow \iota P(\underline{k}, x) \wedge \mu T(\underline{k}, x, a)$
C_6	$\leftarrow \mu P(\underline{k}, a, x) \wedge \neg T(\underline{k}, x) \wedge \neg \iota T(\underline{k}, x) \wedge \neg \mu T(\underline{k}, b, x)$

Note that each condition describes a situation to be avoided to ensure that an update does not violate integrity constraint Ic1. Therefore, ensuring that no condition holds we guarantee that no integrity constraint is violated. In the following we will refer to each condition by its identifier C_i (i=1..n).

A dependency from a given condition C_i to another condition C_j in the Precedence Graph indicates that a repair of C_i is a potential violation of C_j and, thus, that C_i should be handled before than C_j.

In the following subsections we describe in detail how to obtain the Precedence Graph. We will start by analysing which events are involved in each condition and in which way. To perform this analysis, we use the *Dependency Graph of Events* [Cos95]. Information provided by this graph allows us to identify dependencies between conditions that are used to set up later the Precedence Graph. Finally, two kinds of optimisations are proposed to remove non feasible dependencies obtaining a graph which states more precisely dependencies between conditions.

4.1 Precedence Graph with Flat Integrity Constraints

In order to simplify the presentation, we will first define how to build the Precedence Graph for the case of flat integrity constraints. That is, we will assume for the moment that predicates appearing in the definition of an integrity constraint are restricted to be base predicates.

Before explicitly stating the relationship between repairs and potential violations of conditions associated to integrity constraints, we need to analyse which events are related to each condition and in which way. This information is provided by the Dependency Graph of Events [Cos95] which explicitly states the relationship between events of base and derived predicates with respect to conditions. In section 4.1.1 we review the main concepts regarding this graph.

4.1.1 Dependency Graph of Events [Cos95]

A *Dependency Graph of Events* is a triad <V,C,E> where V and C are two sets of nodes and $E \subseteq (VxV) \cup (VxC)$ is a set of directed edges. Each node $v \in V$ has associated an event, each node $c \in C$ has associated a condition and each edge $e \in E$ is marked positive or negative.

In the Dependency Graph of Events there are two kinds of edges:

There exists an edge e=(v', v) where v' and v $\in$ V, if there is a rule in the Augmented Database A(D) with event v as a head, and there is an event w in the body of the rule that coincides with v'. This edge is marked positively (resp. negatively) if w is positive (resp. negative) in the rule.

There exists an edge e=(v', c) where v' $\in$ V and c $\in$ C, if there is an event w which coincides with v' in the body of the condition associated to node c. This edge is marked positively (resp. negatively) if w is positive (resp. negative) in the rule.

If v (or c) and v' are nodes in the Dependency Graph of Events, we say that:

a) v depends on v' if there is a path from v' to v.

b) v depends evenly (resp. oddly) on v' if there is a path from v' to v containing an even (resp. odd) number of negative edges.

Example 4.2: Consider the condition C_3 of the integrity constraint Ic1 of example 3.1:

$$\leftarrow \iota P(\underline{k}, x) \wedge \neg T(\underline{k}, x) \wedge \neg \iota T(\underline{k}, x) \wedge \neg \mu T(\underline{k}, a, x)$$

In the body of this rule there is one positive event $\iota P(\underline{k}, x)$ and two negative ones: $\iota T(\underline{k}, x)$ and $\mu T(\underline{k},a,x)$. Therefore, in the Dependency Graph of Events there will be one edge marked positively from node ιP to node C_3, and two edges marked negatively from node ιT and node μT to node C_3. It is also easy to see that node C_3 depends evenly on ιP, and it depends oddly on ιT and μT. Thus, the part of the Dependency Graph of Events corresponding to condition C_3 is the following, where black arrows correspond to edges marked positively and grey arrows correspond to negative ones:

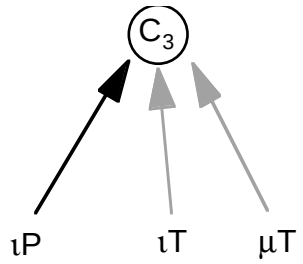


Fig.1. Dependency Graph of Events corresponding to condition C_3

By considering all the conditions of example 3.1, we obtain the following Dependency Graph of Events:

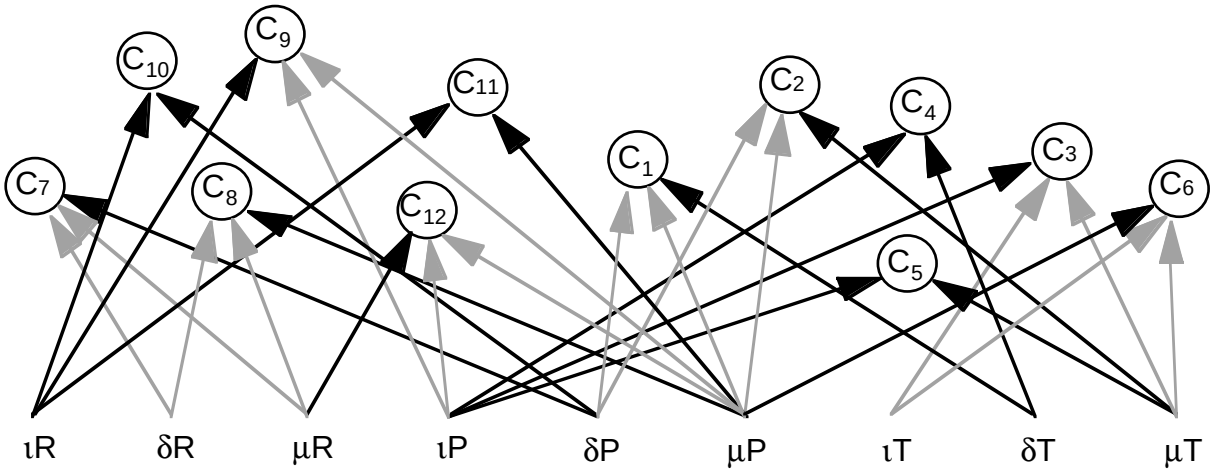


Fig.2. Dependency Graph of Events of our example

In addition to stating which events are related to each condition, the Dependency Graph of Events is also used for defining the concepts of *Checking* and *Generation Conditions* and of *Potential Violation* and of *Repair*.

1) For our purposes it will be useful to distinguish between *Checking Conditions*, those conditions that once violated may never be repaired, and *Generation Conditions*, those that once violated by an update can be falsified by performing additional updates of base facts. Conditions can be classified into one of these classes by a syntactic analysis of the Dependency Graph of Events.

   A condition C_i is a *Checking Condition* if it that does not depend oddly on any base event (all dependencies on base events are even).

   A condition C_i is a *Generation Condition* if it depends oddly on at least one base event.

   Note that from the above definitions it follows that a condition must be either a Checking or a Generation Condition and that it may never be both.

   Checking Conditions of the example are C_4, C_5, C_{10} and C_{11}, while the rest correspond to Generation Conditions. For instance, condition C_3 is a Generation Condition because if it holds, an insertion or a modification of a fact of predicate T(k,x) will falsify it.

2) The Dependency Graph of Events allows also to identify which events could violate a condition, thus being a *Potential Violation* of this condition, and which events may be used as a *Repair* of a condition when it is violated.

   An event Ev is a *Potential Violation* of a condition C_i if the condition depends evenly on it. Note that if an event Ev is a potential violation then when Ev is true an insertion event fact of the inconsistency predicate associated to the condition C_i could be induced and thus integrity constraint could be violated.

At definition time we can not ensure that at run time an event Ev will correspond to a real violation, since the database must also satisfy other requirements which may not be completely verified at this moment. In particular, the rest of literals of the condition must be also true for some concrete value. This is the reason why we talk about potential violations.

An event Ev is a *Repair* of a condition C_i if the condition depends oddly on it. Note that, in this case, if a condition C_i holds and the event Ev (repair) occurs then condition C_i becomes falsified (repaired).

Let us consider again the condition C_3 of the example:

$$\leftarrow \iota P(\underline{k}, x) \wedge \neg T(\underline{k}, x) \wedge \neg \iota T(\underline{k}, x) \wedge \neg \mu T(\underline{k}, a, x)$$

It is not difficult to see that event $\iota P(\underline{k}, x)$ is a potential violation of C_3 because it could make C_3 true depending on the rest of literals of the condition. Notice also that C_3 depends evenly on $\iota P(\underline{k}, x)$. Events $\iota T(\underline{k}, x)$ and $\mu T(\underline{k}, a, x)$ are two different repairs of condition C_3. This condition depends oddly on them, and if one of these events occurs, then the condition will be false.

4.1.2 Dependencies Between Conditions

The Dependency Graph of Events does not provide any information about which repairs of a condition become potential violations of other conditions, even though it provides the basis for determining this information. In this section we explain how to identify a dependency between two conditions by looking at the Dependency Graph of Events. This identification will be a key point for obtaining later the Precedence Graph.

Given two conditions C_i and C_j, there is a *dependency between C_i and C_j* if there exists an event Ev that is a repair of C_i and a potential violation of C_j. Note that a dependency between these two conditions explicitly states that a repair of a condition C_i may induce a violation of C_j.

Since the information about repairs and potential violations is provided by the Dependency Graph of Events, this graph will be the basis for determining dependencies between conditions. To do that, we begin by identifying all relevant conditions to each event. We will obtain for each event Ev two different sets: the set of conditions for which Ev is a repair and the set of conditions for which Ev is a potential violation. Notice that if at least one of these sets is empty no dependencies between conditions exist due to this event since either no condition is repaired by it or no condition is potentially violated by it.

Given two conditions C_i and C_j, and an event Ev which is a repair of C_i and a potential violation of C_j, the dependency between C_i and C_j is depicted as follows:

$$C_i \rightarrow C_j \qquad \text{with respect to Ev}$$

Example 4.3: Consider the deletion event $\delta P(\underline{k}, x)$. Conditions C_7 and C_{10} depend evenly on this event, while conditions C_1 and C_2 depend oddly on it. Then, event $\delta P(\underline{k}, x)$ defines the following dependencies between conditions:

$C_1 \rightarrow C_7$	with respect to $\delta P(\underline{k}, x)$
$C_1 \rightarrow C_{10}$	with respect to $\delta P(\underline{k}, x)$
$C_2 \rightarrow C_7$	with respect to $\delta P(\underline{k}, x)$
$C_2 \rightarrow C_{10}$	with respect to $\delta P(\underline{k}, x)$

All these dependencies state that event $\delta P(\underline{k}, x)$ is a repair of the condition at the left hand side of the arrow and, at the same time, it is a potential violation of the condition at the right hand side.

In the rest of the paper we will write a set of dependencies in a compact way. For example, we will write the previous set of dependencies as:

$C_1, C_2 \rightarrow C_7, C_{10}$	with respect to $\delta P(\underline{k}, x)$

4.1.3 Precedence Graph

Once we know how to identify a dependency between two conditions with respect to an event we are in the position to define the Precedence Graph and to explain how it can be obtained from this set of dependencies.

A *Precedence Graph* for a set C of conditions, is a triad <N,G,E> where N is a finite number of nodes, G is a finite number of subgraphs and $E \subseteq (NxN) \cup (GxN)$ is a set of directed edges. Each node $n \in N$ has associated a condition $c \in C$ and each edge $e \in E$ is labelled with one event.

In the Precedence Graph there are two kinds of edges:

A directed edge e=(n,n') labelled with an event Ev where n and $n' \in N$, states that event Ev corresponds to a repair for the condition associated to node n and it is, at the same time, a potential violation to the condition associated to n'.

A directed edge e=(g,n') labelled with an event Ev where $g \in G$ and $n' \in N$, states that event Ev corresponds to a repair for some condition of the subgraph g and it is, at the same time, a potential violation to the condition associated to n'.

A subgraph G is considered as an special case of node of the Precedence Graph. It corresponds to a cyclic precedence subgraph, where there exists a path from a node to itself. A precedence subgraph of conditions describes that a repair of a condition C_i could induce potential violations to other conditions, repairs of which are also potential violations of the same C_i. Therefore, we can not ensure that C_i is falsified until all conditions (nodes) of the subgraph are false.

If there exists a dependency between condition C_i and condition C_j with respect to event Ev, we establish an edge from node C_i to node C_j labelled by Ev. When a node of the graph corresponds to a subgraph G, dependencies between a node C_i member of the subgraph and a node C_j outside the subgraph are transformed into an edge between the subgraph G and the node C_j, with the same label.

To obtain the Precedence Graph we have to identify all dependencies between conditions with respect to insertion, deletion and modification events of each relevant predicate to all integrity constraints. To do that, we proceed in three steps:

Step 1: Consider dependencies between conditions of the same integrity constraint. Identify dependencies between conditions of each integrity constraint independently of the rest of integrity constraints.

Step 2: Consider dependencies between conditions of different integrity constraints. Identify dependencies between conditions of each integrity constraint and conditions of the rest of integrity constraints.

Step 3: Collect and integrate all dependencies identified in the previous steps into a common graph.

In the rest of this subsection, we explain how to proceed to build the Precedence Graph of the database example 3.1.

Step 1: We begin by analysing the first integrity constraint Ic1. Conditions associated to Ic1 are C_1 to C_6. To identify dependencies between them we proceed as we have explained in section 4.1.2 and we obtain the following set of dependencies:

$C_1, C_2 \rightarrow C_6$	with respect to $\mu P(\underline{k}, x, y)$
$C_3, C_6 \rightarrow C_2, C_5$	with respect to $\mu T(\underline{k}, x, y)$

It is not difficult to detect a recurrent set of dependencies composed by $C_2 \rightarrow C_6$ and $C_6 \rightarrow C_2$. Then, in the corresponding Precedence Graph, we will group these dependencies into a subgraph. Dependency between node C_6 and node C_5 is transformed into an edge from the subgraph to the node C_5 with the same label μT. Fig. 3 shows the Precedence Graph relative to the first integrity constraint Ic1 alone. Note that nodes corresponding to Checking Conditions are filled in grey to differentiate them from Generation Conditions.

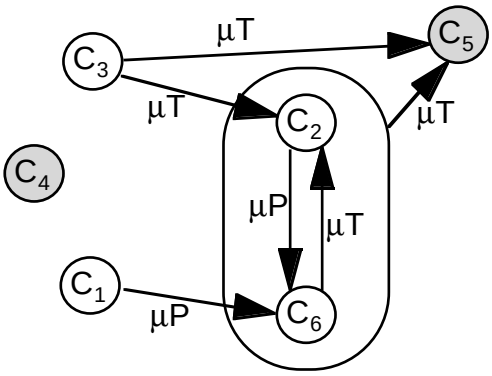


Fig.3. Precedence Graph of integrity constraint Ic1

To obtain the set of dependencies relative to the second integrity constraint Ic2, we proceed in a similar way as for Ic1. Fig. 4 shows the Precedence Graph associated to integrity constraint Ic2.

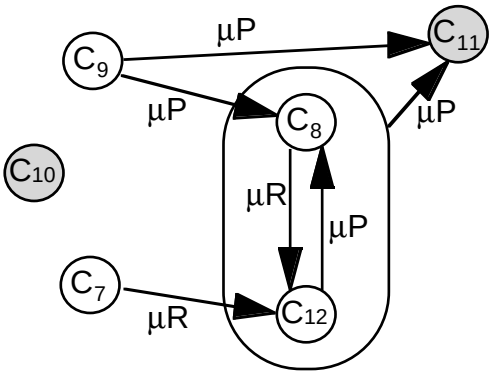


Fig.4. Precedence Graph of integrity constraint Ic2

If we compare Dependency Graphs of Conditions of integrity constraints Ic1 and Ic2, we can see that they have the same structure. They only differ in the condition names (nodes) and events (edge labels). The reason of these similarities is that syntactic definitions of both integrity constraints have the same structure: they are defined by two base predicates, the first one is positive and the second one is negative.

Step 2: In step one, we have obtained dependencies between conditions of each integrity constraint independently of the rest of integrity constraints. Now, as a second step, we will identify which dependencies exist between conditions of integrity constraint Ic1 (C_1...C_6) and conditions of integrity constraint Ic2 (C_7...C_{12}). The resulting set of dependencies is the following:

$C_9, C_{12} \rightarrow C_3, C_4, C_5$	with respect to ιP(k, x)
$C_1, C_2 \rightarrow C_7, C_{10}$	with respect to δP(k, x)
$C_9, C1_2 \rightarrow C_6$	with respect to μP(k, x, y)
$C_1, C_2 \rightarrow C_8, C_{11}$	with respect to μP(k, x, y)

Fig. 5 shows the Precedence Graph that states dependencies between conditions associated to integrity constraint Ic1 and conditions associated to integrity constraint Ic2.

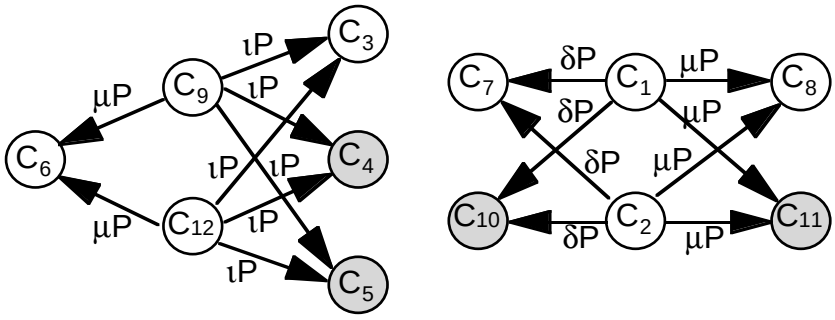


Fig.5. Graph of dependencies between integrity constraints (Ic1- Ic2)

In fact, the partial graphs we have drawn at step 1 and step 2 (shown in figures 3, 4 and 5), do not need to necessary be generated. In general, we only need to obtain the set of their dependencies to obtain the global Precedence Graph. We have drawn graphically these partial Precedence Graphs to make more clear our explanations.

Step 3: As the third step of the process to obtain the Precedence Graph, we have to collect and integrate all subsets of dependencies identified at step 1 and step 2.

These dependencies are:

$C_1, C_2 \rightarrow C_6$	with respect to $\mu P(\underline{k}, x, y)$
$C_3, C_6 \rightarrow C_2, C_5$	with respect to $\mu T(\underline{k}, x, y)$
$C_7, C_8 \rightarrow C_{12}$	with respect to $\mu R(\underline{k}, x, y)$
$C_9, C_{12} \rightarrow C_8, C_{11}$	with respect to $\mu P(\underline{k}, x, y)$
$C_9, C_{12} \rightarrow C_3, C_4, C_5$	with respect to $\iota P(\underline{k}, x)$
$C_1, C_2 \rightarrow C_7, C_{10}$	with respect to $\delta P(\underline{k}, x)$
$C_9, C_{12} \rightarrow C_6$	with respect to $\mu P(\underline{k}, x, y)$
$C_1, C_2 \rightarrow C_8, C_{11}$	with respect to $\mu P(\underline{k}, x, y)$

Notice that this set of dependencies contains those cycles already identified in previous steps between conditions C_2 and C_6, and between C_8 and C_{12}. But now, when considering all dependencies together, two additional cycles have appeared: $C_{12} \rightarrow C_3 \rightarrow C_2 \rightarrow C_8 \rightarrow C_{12}$ and $C_2 \rightarrow C_7 \rightarrow C_{12} \rightarrow C_6 \rightarrow C_2$. But since these cycles share some conditions then, they must be grouped into a unique cyclic subgraph.

The global Precedence Graph of the example 3.1 is shown in Fig.6.

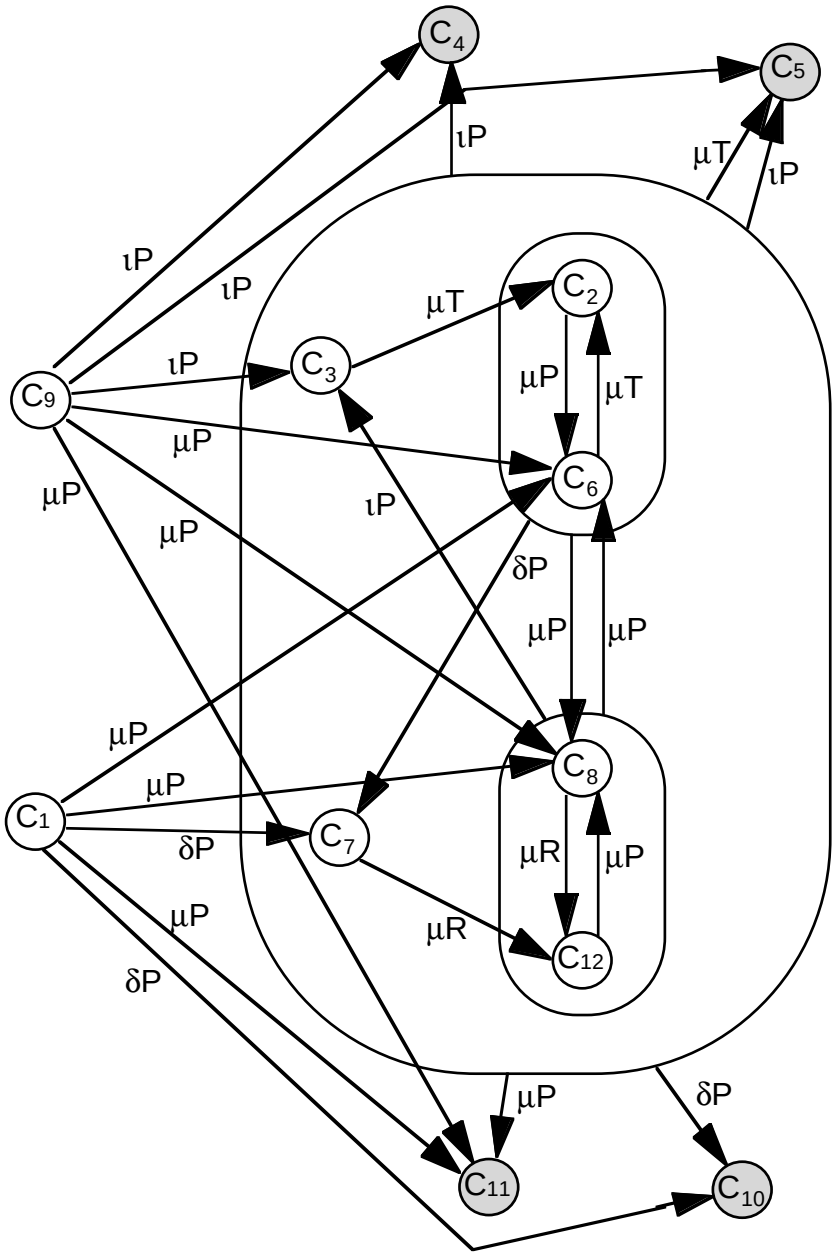


Fig.6. Precedence Graph of example 3.1

As we have said previously, the Precedence Graph is useful to state and manage more easily relationships between integrity constraints. It shows an overall view of these relationships as dependencies between conditions. Using this graph, we can identify alternative repairs for each condition that is violated, and at the same time, it allows to determine which conditions could be affected by the repair. The most important advantages of using the Precedence Graph are that it is useful to minimize the number of recomputations to check whether a given constraint is violated; and also that it is helpful for ensuring that a repair of a certain integrity constraint Icj is performed only when it is guaranteed that all repairs of other constraints that could induce a violation of Icj have been performed.

4.1.4 Optimizations

Each dependency drawn into the Precedence Graph, states that a repair of one condition is a potential violation of another condition. If we analyse in more detail these dependencies, we can detect that some of them are never achievable at run-time since the requirements of each condition are incompatible. The more accurately we can make this decision, more efficiency we will be gaining for integrity constraints maintenance. In this section, we propose two different optimizations that allow us to eliminate non-reachable dependencies, obtaining a more precise Precedence Graph. These optimizations can also be applied at definition time since we only need to take into account definition of events and syntactical information of conditions.

The first optimization that we propose is used to eliminate dependencies between conditions associated to the same integrity constraint (Step 1 of our procedure). This optimization is based on the analysis of the necessary requirements that the contents of the database should satify to ensure that a repair of a condition is a potential repair of another condition. Hence, it restricts the set of conditions that could be violated by the repair of another condition.

The second optimization is addressed to reject non-feasible dependencies drawn into the Precedence Graph, independently if they are associate to the same or to different integrity constraints (Steps 1 and 2 of our procedure). This optimization is based on the fact that the definition of insertion, deletion and modification events given in section 3 prevents two different kinds of event about the same predicate to hold at the same time. This optimization is aimed to detect if such situation could be introduced by a dependency.

a) *Database Requirements Optimization*

As we said in section 2, in this paper we restrict deductive databases to the universal key case. Under this assumption, we can ensure that when a condition associated to an integrity constraint is violated, the corresponding repair (event) with a key k can not violate another condition of the same integrity constraint with another key k', $k' \neq k$.

We know that a condition is violated if and only if all their database requirements hold. Then, a repair of a condition could violate another condition only if both conditions do not have contradictory database requirements. This situation allows us to eliminate dependencies between two conditions with different database requirements, because both conditions will not be achievable at the same time. This criterion is only applicable to dependencies between conditions of the same integrity constraint since for conditions of different integrity constraints we can not guarantee that conditions with different database requirements are always non-achievable at the same time.

For each condition associated to an integrity constraint, we must identify which is the set of requirements of the database contents needed to violate that condition. We are only interested in two kinds of requirements: the set of concrete facts that must hold and those predicates that can not have

any true fact into the database. To do that, we distinguish two kinds of literals in the body of the condition: the set of literals that refer to base or derived predicates and the set of positive events. Literals that refer to base and derived predicates state which is the necessary contents of the database to allow the condition to be violated. Positive events that appear into the body of conditions permit to derive additional database requirements using the event definitions presented in section 3, as it is shown by the following rules:

$\forall k, x \quad (\iota P(\underline{k}, x) \rightarrow \neg \exists y\ P(\underline{k}, y))$
$\forall k, x \quad (\delta P(\underline{k}, x) \rightarrow P(\underline{k}, x))$
$\forall k, x, x' \quad (\mu P(\underline{k}, x, x') \rightarrow P(\underline{k}, x))$

Now, we will show how to apply this optimization to integrity constraint Ic1 of example 3.1. We begin by analysing database requirements of its associated conditions C_1 to C_6. Consider for instance condition C_3:

$$\leftarrow \iota P(\underline{k}, x) \wedge \neg T(\underline{k}, x) \wedge \neg \iota T(\underline{k}, x) \wedge \neg \mu T(\underline{k}, a, x)$$

Satisfaction of this condition requires event $\iota P(\underline{k}, x)$ to hold. Therefore, from the definition of insertion event, C_3 requires $P(\underline{k}, z)$ to be false for all possible values of z, that is, it requires $\neg P(\underline{k}, z)$. On the other hand, literal $\neg T(\underline{k}, x)$ is not considered a database requirement because it only prevents predicate t to be true for only one specific value 'x' (the value for which the event $\iota P(\underline{k}, x)$ holds); while we are only interested in requirements stating either concrete facts that must hold or predicates that must have an empty extension.

In a similar way we obtain the following database requirements of the rest of conditions associated to Ic1:

$C_1 \quad \rightarrow P(\underline{k}, x) \wedge T(\underline{k}, x)$
$C_2 \quad \rightarrow P(\underline{k}, x) \wedge T(\underline{k}, x)$
$C_3 \quad \rightarrow \neg P(\underline{k}, z)$
$C_4 \quad \rightarrow \neg P(\underline{k}, z) \wedge T(\underline{k}, x)$
$C_5 \quad \rightarrow \neg P(\underline{k}, z) \wedge T(\underline{k}, x)$
$C_6 \quad \rightarrow P(\underline{k}, y)$

Notice that conditions C_1, C_2 and C_6 have compatible database requirements, and the group of conditions C_3, C_4 and C_5 also have compatible database requirements. Database requirements of the first group and the requirements of the second one are contradictory with respect to predicate $P(\underline{k}, x)$. This means that dependencies between a condition of one group and another condition of the other can be eliminated because they correspond to non-feasible dependencies. In the example, we eliminate two dependencies of Ic1:

$C_1 \rightarrow C_6 \qquad C_2 \rightarrow C_6 \qquad$ with respect to $\mu P(\underline{k}, x, y)$

~~$C_3 \rightarrow C_2$~~ $C_3 \rightarrow C_5$ with respect to $\mu T(\underline{k}, x, y)$
$C_6 \rightarrow C_2$ ~~$C_6 \rightarrow C_5$~~ with respect to $\mu T(\underline{k}, x, y)$

If we apply this optimization to the set of dependencies between conditions associated to the second integrity constraint Ic2, we obtain this new set of dependencies:

$C_7 \rightarrow C_{12}$ $C_8 \rightarrow C_{12}$ with respect to $\mu R(\underline{k}, x, y)$
~~$C_9 \rightarrow C_8$~~ $C_9 \rightarrow C_{11}$ with respect to $\mu P(\underline{k}, x, y)$
$C_{12} \rightarrow C_8$ ~~$C_{12} \rightarrow C_{11}$~~ with respect to $\mu P(\underline{k}, x, y)$

b) *Event Exclusiveness Optimization*

The purpose of this optimization is to identify in which situations, or upon which assumptions, a dependency will not be reachable at run time. This optimization relies on the analysis of each dependency in the Precedence Graph, independently if it relates conditions of the same or of different integrity constraints.

For each dependency $C_i \rightarrow C_j$, we analyse which are the set of events that must occur and the set of events that must not occur to induce a violation of condition C_i. After that, and given the repair of condition C_i that is a potential violation of C_j, we identify the events that must hold and those that must not hold to violate condition C_j. If we find some contradiction between events of C_i and events of C_j, it means that this dependency will not be reachable at run time, and we eliminate it from de Precedence Graph. Otherwise, we keep it on the graph.

There are two different situations we may detect:

b.1) Mutually exclusive events that must hold at the same time.

By the definition of events, some events are mutually exclusive in the sense that they may not happen at the same time. Therefore, if a dependency requires two mutually exclusive events to occur together, this dependency must be rejected because it will not be reachable at run time.

Using the event definitions, we establish the following relation of exclusivness between events:

$\forall k, x\ (\iota P(\underline{k}, x) \rightarrow \neg\ \exists y\ (\iota P(\underline{k}, y) \wedge x \neq y))$
$\forall k, x\ (\delta P(\underline{k}, x) \rightarrow \neg\ \exists y\ (\mu P(\underline{k}, x, y) \wedge x \neq y))$
$\forall k, x, x'\ (\mu P(\underline{k}, x, x') \rightarrow \neg\ \delta P(\underline{k}, x))$
$\forall k, x, x'\ (\mu P(\underline{k}, x, x') \rightarrow \neg\ \exists y\ (\mu P(\underline{k}, x, y) \wedge x' \neq y))$

This set of implications states that two insertion event facts of the same predicate with the same key, but with different non-key arguments, are mutually exclusive and prohibited. It is also forbidden to combine deletion and modification events of the same fact. Finally, two modifications of a fact to two different new values are also prohibited.

Suppose, for example, dependency $C_9 \rightarrow C_{11}$ with respect to event $\mu P(\underline{k}, x, y)$. Definitions of these conditions are:

$C_9 \quad \leftarrow \iota R(\underline{k}, x) \wedge \neg P(\underline{k}, x) \wedge \neg \iota P(\underline{k}, x) \wedge \neg \mu P(\underline{k}, a, x)$
$C_{11} \quad \leftarrow \iota R(\underline{k}, x) \wedge \mu P(\underline{k}, x, a)$

Events involved in each condition are:

C_9	Repair	C_{11}
$\iota R(\underline{k}, x)$	$\mu P(\underline{k}, a, x)$	$\mu P(\underline{k}, a, x)$
$\mu P(\underline{k}, a, x)$		**$\iota R(\underline{k}, a)$**

To have condition C_9 violated, it is necessary that event $\iota R(\underline{k}, x)$ holds. We repair it by means of the event $\mu P(\underline{k}, a, x)$ with $a \neq x$. To violate condition C_{11} with the event $\mu P(\underline{k}, a, x)$, it is necessary that event $\iota R(\underline{k}, a)$ also occurs. But it is not possible because $\iota R(\underline{k}, a)$ and $\iota R(\underline{k}, x)$ are two mutually exclusive events because the values of a and x must be different.

A similar reasoning can be applied to detect that dependency $C_3 \rightarrow C_5$ may never hold at run time.

b.2) Two opposite assumptions.

A more general contradiction could appear when we have to assume two contradictory events. That is, we have to assume that an event must occur and must be prevented at the same time.

Consider dependency $C_6 \rightarrow C_2$ due to the event $\mu T(\underline{k},x,y)$. Definitions of these conditions are:

$C_6 \quad \leftarrow \mu P(\underline{k}, a, x) \wedge \neg T(\underline{k}, x) \wedge \neg \iota T(\underline{k}, x) \wedge \neg \mu T(\underline{k}, b, x)$
$C_2 \quad \leftarrow P(\underline{k}, x) \wedge \neg \delta P(\underline{k}, x) \wedge \neg \mu P(\underline{k}, x, a) \wedge \mu T(\underline{k}, x, b)$

Requeriments involved in each condition are:

C_6	Repair	C_2
$\mu P(\underline{k}, a, x)$ $\rightarrow P(\underline{k}, a)$	$\mu T(\underline{k}, b, x)$	$\mu T(\underline{k}, b, x)$
$\neg T(\underline{k}, x)$		$P(\underline{k}, b) \rightarrow$ **b=a**
$\mu T(\underline{k}, b, x)$		$\neg$ **$\mu P(\underline{k}, b, c)$**

To have condition C_6 violated, it is necessary that event $\mu P(\underline{k}, a, x)$ occurs and that fact $T(\underline{k}, x)$ is false. Event $\mu P(\underline{k}, a, x)$ to hold requires fact $P(\underline{k}, a)$ to be true. A repair of condition C_6 is the event $\mu T(\underline{k}, b, x)$. It can only violate condition C_2 if fact $P(\underline{k}, b)$ is true and event $\mu P(\underline{k}, b, c)$ does not to occur. Notice that a and b must be equal (a=b) to allow $P(\underline{k}, b)$ to be true. Note also that condition C_2 forces not to occur any event like $\mu P(\underline{k}, b, c)$, where c is a free variable. But,

this is contradictory with the fact that event $\mu P(\underline{k}, a, x)$ is necessary to violate C_6. Therefore, we can eliminate dependency between C_6 and C_2 from the Precedence Graph.

A similar reasoning can be applied to remove dependency $C_{12} \rightarrow C_8$.

The final set of dependencies that results from the application of the Database Requeriments and the Event Exclusiveness Optimizations is the following:

$C_1, C_2 \rightarrow C_6$	with respect to $\mu P(\underline{k}, x, y)$
$C_7, C_8 \rightarrow C_{12}$	with respect to $\mu R(\underline{k}, x, y)$
$C_9, C_{12} \rightarrow C_3, C_4, C_5$	with respect to $\iota P(\underline{k}, x)$
$C_1, C_2 \rightarrow C_7, C_{10}$	with respect to $\delta P(\underline{k}, x)$
$C_9, C_{12} \rightarrow C_6$	with respect to $\mu P(\underline{k}, x, y)$
$C_1, C_2 \rightarrow C_8, C_{11}$	with respect to $\mu P(\underline{k}, x, y)$

We show in Fig.7 the optimized version of the Precedence Graph. Note that, in this example, the optimizations applied permit to eliminate dependencies that are responsible of cycles between conditions.

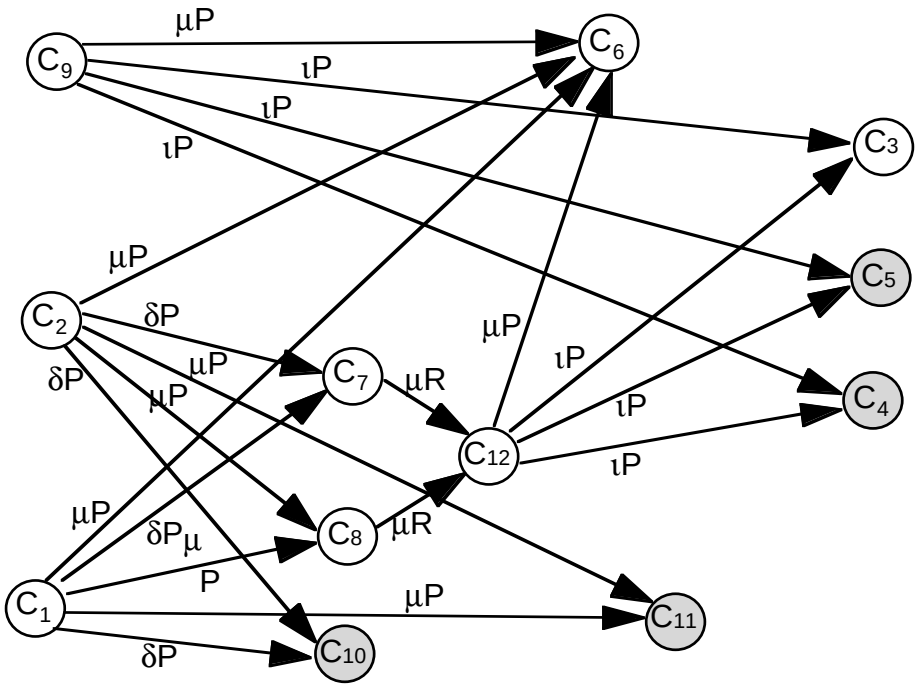


Fig.7. Final Precedence Graph of example 3.1

4.2 Precedence Graph with Non-Flat Integrity Constraints

In this section we explain how to build the Precedence Graph for the case of non-flat integrity constraints, that is, when literals appearing in the body of an integrity constraint may be derived predicates as well as base ones. The procedure for obtaining the Precedence Graph is similar to the

procedure explained in section 4.1 but in the presence of derived predicates we have to consider some specific aspects not considered in the previous case. We begin by introducing the database example we will use in this section.

Example 4.5: Consider the following database containing two integrity constraints: Ic1 and Ic2, and where Ic1 is defined upon the derived predicate P(k, x).

$P(k,x) \leftarrow Q(k,x) \wedge \neg T(k,x)$

$Ic1(k,x) \leftarrow S(k,x) \wedge \neg P(k,x)$
$Ic2(k,x) \leftarrow Z(k,x) \wedge Q(k,x)$

The corresponding Augmented Database A(D) is the following:

(I_1) $\iota P(k,x) \leftarrow Q(k,x) \wedge \neg \delta Q(k,x) \wedge \neg \mu Q(k,x,a) \wedge \delta T(k,x)$
(I_2) $\iota P(k,x) \leftarrow Q(k,x) \wedge \neg \delta Q(k,x) \wedge \neg \mu Q(k,x,a) \wedge \mu T(k,x,b)$
(I_3) $\iota P(k,x) \leftarrow \iota Q(k,x) \wedge \neg T(k,x) \wedge \neg \iota T(k,x) \wedge \neg \mu T(k,a,x)$
(I_4) $\iota P(k,x) \leftarrow \iota Q(k,x) \wedge \delta T(k,x)$
(I_5) $\iota P(k,x) \leftarrow \iota Q(k,x) \wedge \mu T(k,x,a)$
(I_6) $\iota P(k,x) \leftarrow \mu Q(k,a,x) \wedge \neg T(k,x) \wedge \neg \iota T(k,x) \wedge \neg \mu T(k,b,x) \wedge T(k,a)$

(D_1) $\delta P(k,x) \leftarrow \delta Q(k,x) \wedge \neg T(k,x)$
(D_2) $\delta P(k,x) \leftarrow \mu Q(k,x,a) \wedge \neg T(k,x) \wedge T(k,a) \wedge \neg \delta T(k,a) \wedge \neg \mu T(k,a,c)$
(D_3) $\delta P(k,x) \leftarrow \mu Q(k,x,a) \wedge \neg T(k,x) \wedge \iota T(k,a) \wedge \neg \delta T(k,a) \wedge \neg \mu T(k,a,c)$
(D_4) $\delta P(k,x) \leftarrow \mu Q(k,x,a) \wedge \neg T(k,x) \wedge \mu T(k,b,a) \wedge \neg \delta T(k,a) \wedge \neg \mu T(k,a,c)$
(D_5) $\delta P(k,x) \leftarrow Q(k,x) \wedge \iota T(k,x) \wedge \neg \mu Q(k,x,a)$
(D_6) $\delta P(k,x) \leftarrow Q(k,x) \wedge \mu T(k,a,x) \wedge \neg \mu Q(k,x,b)$

(M_1) $\mu P(k,x,x') \leftarrow \mu Q(k,x,x') \wedge \neg T(k,x') \wedge \neg \iota T(k,x') \wedge \neg \mu T(k,a,x') \wedge \neg T(k,x)$
(M_2) $\mu P(k,x,x') \leftarrow \mu Q(k,x,x') \wedge \delta T(k,x')$
(M_3) $\mu P(k,x,x') \leftarrow \mu Q(k,x,x') \wedge \mu T(k,x',a)$

(C_1) $\iota Ic1(k,x) \leftarrow S(k,x) \wedge \neg \delta S(k,x) \wedge \neg \mu S(k,x,a) \wedge \delta P(k,x)$
(C_2) $\iota Ic1(k,x) \leftarrow S(k,x) \wedge \neg \delta S(k,x) \wedge \neg \mu S(k,x,a) \wedge \mu P(k,x,b)$
(C_3) $\iota Ic1(k,x) \leftarrow \iota S(k,x) \wedge \neg P(k,x) \wedge \neg \iota P(k,x) \wedge \neg \mu P(k,a,x)$
(C_4) $\iota Ic1(k,x) \leftarrow \iota S(k,x) \wedge \delta P(k,x)$
(C_5) $\iota Ic1(k,x) \leftarrow \iota S(k,x) \wedge \mu P(k,x,a)$
(C_6) $\iota Ic1(k,x) \leftarrow \mu S(k,a,x) \wedge \neg P(k,x) \wedge \neg \iota P(k,x) \wedge \neg \mu P(k,b,x)$

(C_7) $\iota Ic2(k,x) \leftarrow Z(k,x) \wedge \neg \delta Z(k,x) \wedge \neg \mu Z(k,x,a) \wedge \iota Q(k,x)$
(C_8) $\iota Ic2(k,x) \leftarrow Z(k,x) \wedge \neg \delta Z(k,x) \wedge \neg \mu Z(k,x,a) \wedge \mu Q(k,b,x)$
(C_9) $\iota Ic2(k,x) \leftarrow \iota Z(k,x) \wedge Q(k,x) \wedge \neg \delta Q(k,x) \wedge \neg \mu Q(k,x,a)$
(C_{10}) $\iota Ic2(k,x) \leftarrow \iota Z(k,x) \wedge \iota Q(k,x)$
(C_{11}) $\iota Ic2(k,x) \leftarrow \iota Z(k,x) \wedge \mu Q(k,a,x)$

(C_{12}) $\iota Ic2(\underline{k},x) \leftarrow \mu Z(\underline{k},a,x) \wedge Q(\underline{k},x) \wedge \neg \delta Q(\underline{k},x) \wedge \neg \mu Q(\underline{k},x,b)$
(C_{13}) $\iota Ic2(\underline{k},x) \leftarrow \mu Z(\underline{k},a,x) \wedge \iota Q(\underline{k},x)$
(C_{14}) $\iota Ic2(\underline{k},x) \leftarrow \mu Z(\underline{k},a,x) \wedge \mu Q(\underline{k},b,x)$

4.2.1 Dependency Graph of Events

To build the Dependency Graph of Events of a database with some non-flat integrity constraint, we proceed in the same way as we have described in section 4.1.1, but considering also relationships of derived events.

If there are derived events in the body of a condition, we must also include into the Dependency Graph of Events the positive and negative edges corresponding to the relationships of these derived events with respect to events of their underlying predicates. To obtain them, we apply the mechanism defined in section 4.1.1 to the insertion, deletion and modification event rules of those derived predicates. In some cases, it may happen that there exists two edges one marked positively and the other negativaly that connect the same nodes. For instance, it is not difficult to see by looking at the modification event rules M_1 and M_3 that there exists a negative and a positive dependency of event $\mu P(\underline{k},x,x')$ with respect to event $\mu T(\underline{k},x,x')$.

In Fig.8, we show the Dependency Graph of Events of example 4.5. Black arrows correspond to edges marked positively and grey arrows correspond to negative edges.

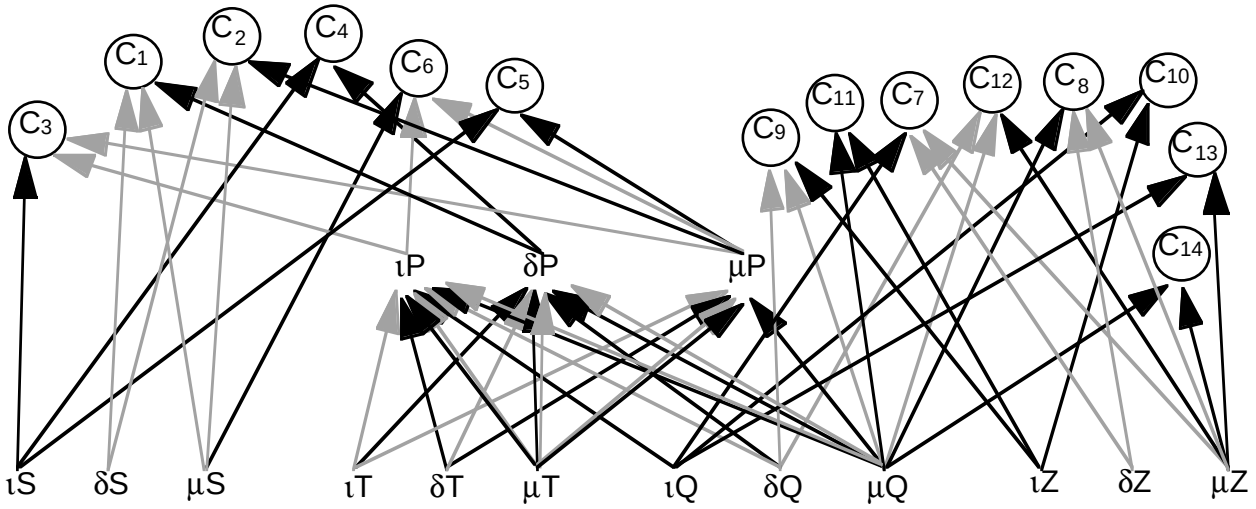


Fig.8. Dependency Graph of Events of example 4.5

Note that in this case it is also possible to distinguish between Checking Conditions and Generation Conditions by using the same criteria than in the case of flat integrity constraints. In the above Dependency Draph of Events we can classify conditions C_{10}, C_{11}, C_{13} and C_{14} as Checking Conditions, while the rest of conditions correspond to Generation Conditions.

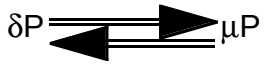
Definitions of potential violation and repair of a condition are also applicable to the case of non-flat integrity constraints. Moreover, we can also use the concept of repair of a condition to define the repair of a derived event. Thus, we have that a base event Ev is a repair of a derived event if this derived event depends oddly on Ev. That is, if the derived event holds and the event Ev (repair) occurs then, the derived event becomes falsified (repaired). For instance, in the previous graph we have that event $\iota T(\underline{k}, x)$ is a repair for derived events $\iota P(\underline{k}, x)$ and $\mu P(\underline{k}, x, y)$ and that it is also a repair for conditions C_2 and C_5.

In addition to the concepts already defined in section 4.1.2, we need to define a new kind of relationship between events of the same derived predicate. Given two events Ev1 and Ev2 of the same derived predicate D and an event Ev such that Ev1 depends oddly on Ev and Ev2 depends evenly on Ev, we say that *Ev1 may become Ev2* since repairing Ev1 by means of Ev may induce Ev2. This new relationship is included in the Dependency Graph of Events as a non labelled edge drawn as a double arrow from node Ev1 to node Ev2. This relationship is needed because if we have two conditons C_i and C_j that depend evenly (resp. oddly) on events Ev1 and Ev2, it may happen that a repair of condition C_i (resp. C_j) could violate C_j (resp. C_i). The notion that an event Ev1 may become Ev2 will allow us to identify the dependency between conditions C_i and C_j when we will generate the Precedence Graph.

Example 4.6: Consider the deletion and modification event rules D_2 and M_2 of derived predicate $P(\underline{k},x)$ of example 4.5:

$$\delta P(\underline{k},x) \leftarrow \mu Q(\underline{k},x,a) \wedge \neg T(\underline{k},x) \wedge T(\underline{k},a) \wedge \neg \delta T(\underline{k},a) \wedge \neg \mu T(\underline{k},a,c)$$
$$\mu P(\underline{k},x,a) \leftarrow \mu Q(\underline{k},x,a) \wedge \delta T(\underline{k},a)$$

In these rules we have that $\mu P(\underline{k},x,a)$ depends evenly on event $\delta T(\underline{k},a)$, while $\delta P(\underline{k},x)$ depends oddly on the same event. Therefore, repairing the derived event $\delta P(\underline{k},x)$ by means of event $\delta T(\underline{k},a)$ will induce the fact $\mu P(\underline{k},x,a)$. In the same way, we can also identify the simetric relationship from μP to δP looking at the first modification event rule (M1) and at the third deletion event rule (D3) of predicate $P(\underline{k},x)$. Then, we represent in the Dependency Graph of Events these two relationships by the following edges:



This kind of relationship may appear only between deletion and modification events of the same derived predicate. It is not possible to fulfil an insertion event by the repair of a deletion or modification event (or viceversa) because these events have opposite database requirements.

4.2.2 Dependencies Between Conditions

The meaning of a dependency between two conditions is the same independently if the event that determines the dependency is a base or a derived event because, even though a derived event does not correspond to a physical repair of the condition, it is induced by this repair. However, the way of identifying dependencies between conditions differs from the case of flat integrity constraints since now we have to decide first with respect to which events (predicate events) do we want to establish the dependency. In the previous case this problem did not exist since we always had conditions defined only by means of base events.

In the non-flat integrity constraint case, we have different alternatives with respect to which are the events considered for identifying dependencies between conditions. In our approach, we express dependencies between conditions with respect to base or derived events involved, explicitly or implicitly, in the definition of an integrity constraint. Then, for each integrity constraint, we must determine with respect to which predicate events do we have to define dependencies between conditions. The set of predicates considered for defining dependencies are the *meeting predicates*:

A base or a derived predicate is a *meeting predicate* with respect to a set of integrity constraints if one of these three cases holds (these cases must be considered in the same order as described below):

- it is a base predicate that explicitly appears into some integrity constraint definition.
- it is a derived predicate whose underlying predicates are not meeting predicates.
- it is a base predicate that defines a derived predicate which is not a meeting predicate.

Example 4.7: Consider the database example 4.5. The corresponding meeting predicates of the integrity constraint Ic1 are S(k,x) and P(k,x). Note that predicates Q(k,x) and T(k,x) are not considered meeting predicates because they do not appear explicitly in the body of Ic1 and do not define any other predicate than P(k,x). Then, all dependencies between conditions associated to Ic1 will be defined only with respect to events of predicates S(k,x) and P(k,x). On the other hand, if we consider integrity constraints Ic1 and Ic2 together, the meeting predicates will be S(k,x), Z(k,x), Q(k,x) and T(k,x). Notice that in this case S(k,x), Z(k,x) and Q(k,x) appear explicitly in the body of some integrity constraint and then P(k,x) is not a meeting predicate because it is defined by the meeting predicate Q(k,x). Predicate T(k,x) must be also considered a meeting predicate because defines the non-meeting predicate P(k,x).

Insertion, deletion and modification events of a meeting predicate are called meeting events.

Now, by taking into account meeting events, we can identify dependencies between conditions from the Dependency Graph of Events. We have three criterions to identify dependencies between conditions:

a) We define a dependency from each condition that depends oddly on a meeting event to each condition that depends evenly respect to the same event. This is the same criteria as for the flat integrity constraint case.

b) For each derived meeting event, we also state a dependency between each Generation Condition that depends evenly on the event to each condition that depends oddly on the same event.

   For example, if we consider the integrity constraint Ic1 alone $\mu P(\underline{k}, x, y)$ is a meeting event and conditions C_2 and C_6 depend evenly and oddly on it, respectively. Since condition C_2 is a Generation Condition we identify a dependency from condition C_2 to condition C_6 with respect to the event $\mu P(\underline{k}, x, y)$.

c) For each double edge from a derived meeting event Ev1 to a derived meeting event Ev2, we define a dependency with respect to the event Ev2 from each condition that depends evenly on Ev1 to each condition that depends evenly on Ev2. We also establish a dependency with respect to the event $\neg$ Ev1 from conditions that depends oddly on Ev2 to conditions that depends oddly on Ev1.

   Note that in this later case, an edge marked with a negative event Ev1 denotes that repairing a condition through event Ev2 may induce the falsification of event Ev1.

   Events $\delta P(\underline{k}, x)$ and $\mu P(\underline{k}, x, y)$ are derived meeting predicates respect to integrity constraint Ic1 and there exists a double edge from δP to μP. Then, we can identify a dependency between conditions C_4 and C_5 with respect to event $\mu P(\underline{k}, x, y)$.

4.2.3 Precedence Graph

To generate the Precedence Graph, we consider exactly the three steps defined in section 4.1.3 for the flat integrity constraints case. Now, the only difference is that to identify a dependency between two conditions we will apply the criterions defined in the previous section. In the following example we show how to obtain dependencies between conditions of example 4.5.

Step 1: We begin by considering dependencies between conditions associated only to integrity constraint Ic1 (C_1...C_6). Dependencies we can derive from the Dependency Graph of Events are the following:

$C_1, C_2 \rightarrow C_6$ with respect to $\mu S(\underline{k}, x, y)$
$C_3, C_6 \rightarrow C_2, C_5$ with respect to $\mu P(\underline{k}, x, y)$

$C_2, C_5 \rightarrow C_3, C_6$ with respect to $\mu P(\underline{k}, x, y)$

$C_2, C_5 \rightarrow C_1, C_4$ with respect to $\delta P(\underline{k}, x)$

$C_1, C_4 \rightarrow C_2, C_5$ with respect to $\mu P(\underline{k}, x, y)$

Notice that the first group of dependencies are generated by applying the criteria a). The second group corresponds to dependencies obtained upon criteria b). And the last group of dependencies are obtained taking into account the third criteria c).

The second integrity constraint is flat, so we only can apply criteria a) and we obtain the following set of dependencies:

$C_7, C_8 \rightarrow C_{12}, C_{13}, C_{14}$ with respect to $\mu Z(\underline{k}, x, y)$
$C_9, C_{12} \rightarrow C_8, C_{11}, C_{14}$ with respect to $\mu Q(\underline{k}, x, y)$

Step 2: Since when considering constraints Ic1 and Ic2 together all meeting predicates are base predicates we can only apply criteria a); and we obtain the following set of dependencies:

$C_3, C_6 \rightarrow C_7, C_{10}, C_{13}$ with respect to $\iota Q(\underline{k}, x)$
$C_9, C_{12} \rightarrow C_1, C_3, C_4, C_6$ with respect to $\delta Q(\underline{k}, x)$
$C_1, C_3, C_4, C_6 \rightarrow C_8, C_{11}, C_{14}$ with respect to $\mu Q(\underline{k}, x, y)$
$C_9, C_{12} \rightarrow C_1, C_2, C_3, C_4, C_5, C_6$ with respect to $\mu Q(\underline{k}, x, y)$

Step 3. In previous Steps 1 and 2 we have obtained the complete set of dependencies between conditions of example 4.5. The graph resulting from considering these dependencies between conditions is too complex and difficult to understand. Therefore, we will not draw it until applying in next section all possible optimizations.

4.2.4 Optimizations

Optimizations of the Precedence Graph proposed in section 3.1.4 are also applicable in this case, but we have to make some precisions.

The Database Requirements optimization is directly applicable. We only have to distinguish that when we require a base predicate to be true (false) it means that it will hold (do not hold) in the database; while when we require a derived predicate to be true (false) it means that it must be (not be) deducible given the contents of the database. Note that if the derived predicate is defined by more than one deductive rule, all of them must be considered.

The Event Exclusiveness optimization is also applicable in this case. We must only take into account that a derived event could be defined by several deductive rules, and that their corresponding insertion, deletion and modification events may also be defined by more than one event rule. This fact forces us to consider all these rules at the same time. In particular, to ensure that a derived fact will not be true, all deduction rules that could induce it must be taken into account. The same consideration must be done to ensure that a derived event will not be fulfilled.

Example 4.8: As an example of both optimizations, let us consider the dependency between conditions C_3 and C_5 with respect to the derived event $\mu P(\underline{k}, x, y)$:

$C_3 \rightarrow C_5$ with respect to $\mu P(\underline{k}, x, y)$

Respect to the database requirements optimization, this dependency is reachable because it has compatible database requirements:

$(C_3) \quad \rightarrow \neg S(\underline{k}, z)$
$(C_5) \quad \rightarrow \neg S(\underline{k}, z) \wedge P(\underline{k}, x)$

Let us see for both conditions what are the set of necessary events to violate them and we will try to detect some contradiction that allows us to discard this dependency.

C_3	Repair	C_5
$\iota S(\underline{k}, x)$	$\mu P(\underline{k}, a, x)$	$\mu P(\underline{k}, a, x)$
$\mu P(\underline{k}, a, x)$		**$\iota S(\underline{k}, a)$**

To violate condition C_3 it is necessary that event $\iota S(\underline{k}, x)$ holds. To repair it, we must induce event $\mu P(\underline{k}, a, x)$ by means of a modification event rule of predicate $P(\underline{k}, x)$. Condition C_5 would become violated by events $\iota S(\underline{k}, a)$ and $\mu P(\underline{k}, a, x)$. But it will not be possible because events $\iota S(\underline{k}, x)$ and $\iota S(\underline{k}, a)$ are mutually exclusive. Then, dependency $C_3 \rightarrow C_5$ with respect $\mu P(\underline{k}, x, y)$ is not feasible and we can reject it.

Consider the set of dependencies $C_9 \rightarrow C_1, C_3, C_4, C_6$ with respect to event $\delta Q(\underline{k}, x)$. To repair condition C_9 we must assume that fact $Q(\underline{k}, x)$ and event $\iota Z(\underline{k}, x)$ hold and the repair is the base event $\delta Q(\underline{k}, x)$. To determine if the repair $\delta Q(\underline{k}, x)$ would violate conditions C_1, C_3, C_4 or C_6, we must see which derived events could be induced or not by the repair. Looking all the events rules of predicate $P(\underline{k}, x)$, we can state that the repair of C_9 could induce $\neg \iota P(\underline{k}, x)$ (insertion event rules I_1 and I_2) and induce $\delta P(\underline{k}, x)$ (deletion event rule D_1). By the fact that the repair can dismiss event $\iota P(\underline{k}, x)$, then conditions C_3 and C_6 could be violated. In the same way, by the induction of event $\delta P(\underline{k}, x)$, it is possible to violate C_1 and C_4. Then, the set of dependencies from C_9 to C_1, C_3, C_4 and C_6 with respect $\delta Q(\underline{k}, x)$ can not be eliminated.

The final set of dependencies that can not be rejected by the proposed simplifications is:

$C_1, C_2 \rightarrow C_6$ with respect to $\mu S(\underline{k}, x, y)$
$C_1 \rightarrow C_2$ with respect to $\mu P(\underline{k}, x, y)$
$C_4 \rightarrow C_5$ with respect to $\mu P(\underline{k}, x, y)$
$C_2 \rightarrow C_1$ with respect to $\delta P(\underline{k}, x)$
$C_5 \rightarrow C_4$ with respect to $\delta P(\underline{k}, x)$
$C_3, C_6 \rightarrow C_7, C_{10}, C_{13}$ with respect to $\iota Q(\underline{k}, x)$

$C_9, C_{12} \rightarrow C_1, C_3, C_4, C_6$ with respect to $\delta Q(\underline{k}, x)$
$C_3, C_6 \rightarrow C_8, C_{11}, C_{14}$ with respect to $\mu Q(\underline{k}, x, y)$
$C_9, C_{12} \rightarrow C_1, C_2, C_3, C_4, C_5, C_6$ with respect to $\mu Q(\underline{k}, x, y)$

The final version of Precedence Graph obtained after the application of the proposed optimizations is shown in Fig.9.

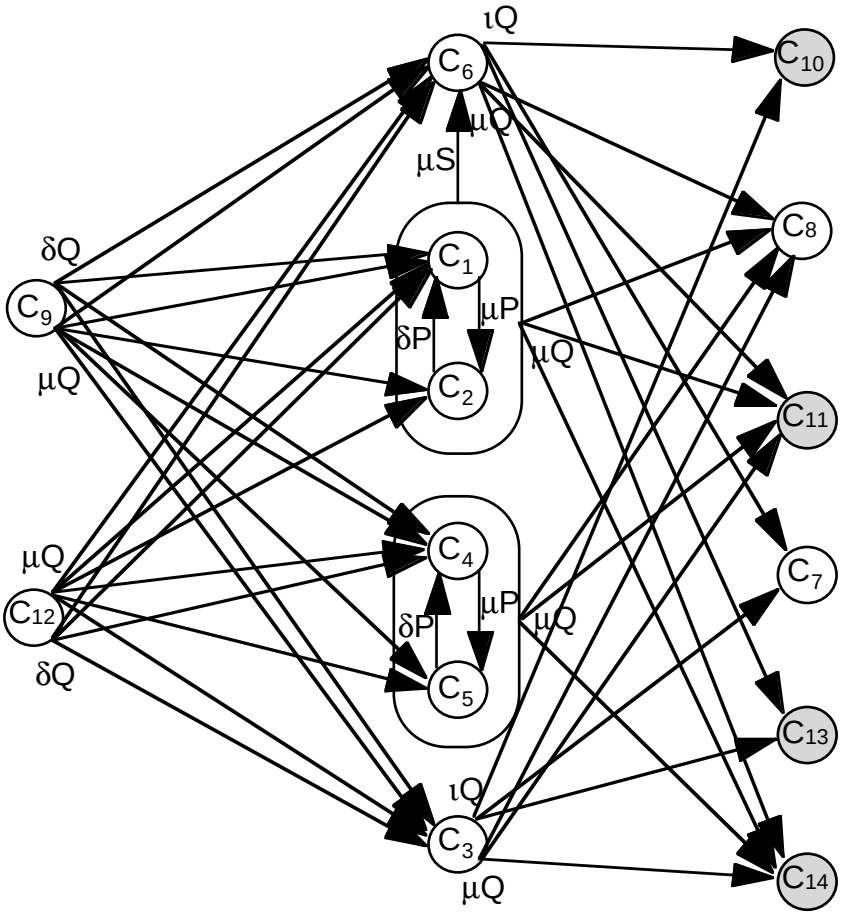


Fig.9. Precedence Graph of the example 4.5

The existence of cycles between conditions of the Precedence Graph may indicate that the process of integrity maintenance does not terminate. However, it is important to note that the existence of a cycle between a set of conditions does not necessarily imply that dealing with conditions of the cycle should be performed forever. On the contrary, a cycle in the Precedence Graph does not correspond in general to an infinite loop at execution time.

As an example, consider the cycle between conditions C_4 and C_5 of Figure 9. This cycle states that a repair of the condition C_4 via the literal δP may induce an event μP which is a potential violation of C_5; and also that a repair of C_5 via μP may also induce an event δP which is a potential violation of C_4. However, if we look at the modification event rules of example 4.5, there exists only one possible repair of condition C_5 (via rule M.1) that may induce an event δP. Thence, at execution time,

it is garanteed that the cycle between conditions C_4 and C_5 will be considered almost one time, which ensures termination of that cycle. This situation will be clarified by the example we will explain in the following section. A similar comment applies to the cycle between conditions C_1 and C_2.

5. Execution of the Precedence Graph

The purpose of the database consistency maintenance is to ensure that a transaction to be applied to a database does not violate any integrity constraint. Therefore, given a transacion T, the outcome of the integrity constraints maintenance process will be either the same transaction T if no constraint is violated by T, or otherwise a new transaction T' (which is a superset of T) that satisfies all the integrity constraints of the database.

In our approach, information about integrity constraints is provided by the set of their associated conditions which are represented in the Precedence Graph. Therefore, integrity constraints maintenance is concerned with guaranteeing that all conditions of the Precedence Graph remain true after the application of a transaction.

To specify the integrity constraints maintenance process, we must define how and when a condition should be processed. The own structure of the Precedence Graph implicitly defines different possible orders to check its conditions, but we need to determine a proper order to check them. To do that, we use the set of events that compose the transaction T and the facts of the extensional database (EDB) to detect which potential violations become real violations and then which conditions must be repaired.

The mechanism we propose to maintain all conditions of the Precedence Graph is based on the execution rules of Petri nets [Pet81]. In our approach, we use tokens to mark nodes whose associated condition is potentially violated by the transaction. Therefore, to maintain all conditions of the Precedence Graph, we have to visit only those nodes that contain a token. For each node, we check if the associated condition is violated and we repair it if this is the case. Finally, the token is dropped from the repaired condition and it is sent to each node whose condition could be violated by this repair.

In the following, we give a more detailed description of the mechanism for maintaining integrity constraints by using our Precedence Graph:

Step 0: *Mark all nodes of the graph.* To ensure that all conditions of the Precedence Graph G will be enforced, mark each node of the graph by one token. Go to Step 1.

Step 1: *Pick up next node C.* If all nodes of the graph G are unmarked, go to step 4. Otherwise, select a marked node C. Select with priority nodes that correspond to Checking Conditions instead of Generation Conditions. If different candidates exist, choose first nodes without (in_degree of node equal to 0) or with all predecessors already visited. If node C belongs to a

subgraph SG of the graph G we are considering, start the mechanism again considering only nodes of subgraph SG. If not, go to Step 2 considering the same graph G.

In general, it is more efficient to begin enforcing Checking Conditions than Generation Conditions. This is because when a Checking Condition is violated, it can not be repaired and the consistency maintenance process will finish rejecting the transaction T, without considering the rest of conditions of the Precedence Graph.

A node C will be a good candidate to be visited if all nodes whose repairs can violate C have been already visited. This precaution permits to ensure that a node will be visited only once (if it does not belong to a subgraph).

Step 2: *Check violation of a condition.* Verify if the condition associated to node C is violated. If not, unmark the node C and go to Step 1.

In the case that condition C is violated and it is a Generation Condition, go to Step 3 to repair it. If the violated condition C is a Checking Condition, go back to the last node Cp whose condition has been repaired, restore the same marked nodes before Cp was repaired and go to Step 3 to find an alternative repair.

Note that, to improve efficiency of the whole process, it will be better to use an integrity constraints checking method [Oli91, CGMD94].

Step 3: *Repair a condition.* Repair the condition associated to node C and include the repair into the transaction T. Notice that if the repair is a derived event, a view updating method must be applied to translate it into base events. Given the repair, compute the derived events that could be induced by it and send a token to each node that can be violated by the repair or by the induced derived events. Finally, unmark node C and go to Step 1.

In the case that it is not possible to repair the condition associated to node C, come back to the last condition Cp we have repaired, restore the same marks on nodes before Cp was repaired and try to find an alternative repair of Cp going to Step 3.

Step 4: *End of the consistency maintenance process.* If the graph we are considering is a subgraph SG of a more general graph G, we have finished the execution of the mechanism restricted to the sugbraph SG. Now, mark all nodes outside the subgraph that could be violated by any repair performed of conditions of nodes inside the subgraph. Then, continue at Step 1 the execution of the mechanism with respect to the more general graph G.

If the gaph we have been considering is not a subgraph, finish the process of consistency maintenance giving the transaction T as a solution that maintains the consistency of all conditions of the Dependency Graph.

At steps 2 and 4 of this mechanism, we distinguish between whether a node belongs to a subgraph SG or not. This is needed because after repairing a condition inside a subgraph (cycle), we can not ensure that we will not need to visit it again until all nodes of the subgraph have been unmarked. So, we can not select nodes outside the subgraph until all nodes of the subgraph become unmarked to avoid visiting several times the same node if it is not necessary.

Notice that, in the particular case in which the Precedence Graph does not contain any subgraph (it is acyclic), the proper order to visit all nodes coincides with the topological sort of a graph [Wil79]. But, in a general case, topological sort is not applicable to our Precedence Graph because it could contain cycles. In this latter case, a possible way to apply the topological sort would be to remove cycles by the substitution of subgraphs of the Precedence Graph by condensed nodes. Then, we could apply the topological sort to the acyclic Precedence Graph, and condensed nodes must be handled individually.

To exemplify how the mechanism we have defined goes on, we will consider the Precedence Graph shown in Fig.10, which is a subset of the final graph of example 4.5. Therefore, note that this graph is only useful to illustrate how to execute the mechanism and it is not useful to enforce the global consistency of the example. We are going to explain the execution of the Precedence Graph assuming that the initial transaction T that we consider is composed by the base event fact $\iota Z(\underline{A}, 1)$ and that the contents of the EDB is $\{Q(\underline{A}, 1), S(\underline{A}, 1)\}$.

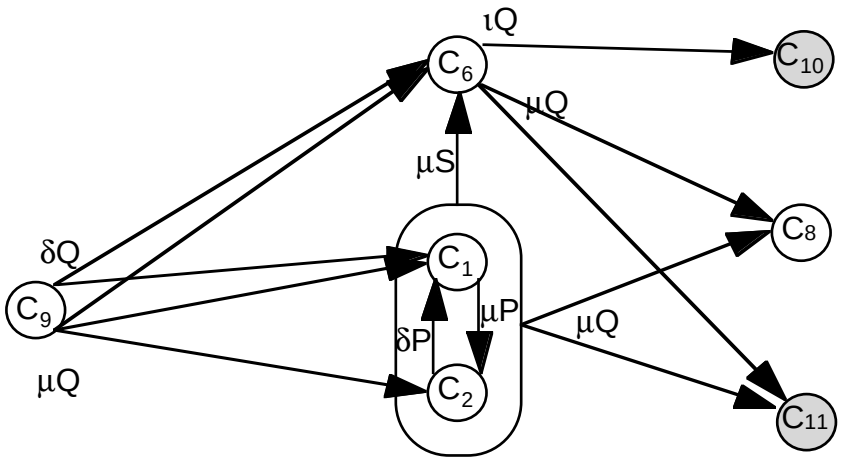


Fig.10. Precedence Graph to execute

The set of event rules of the Augmented Database A(D) involved in this Precedence Graph are the following:

$$P(\underline{k},x) \leftarrow Q(\underline{k},x) \wedge \neg T(\underline{k},x)$$

$$Ic1(\underline{k},x) \leftarrow S(\underline{k},x) \wedge \neg P(\underline{k},x)$$
$$Ic2(\underline{k},x) \leftarrow Z(\underline{k},x) \wedge Q(\underline{k},x)$$

(I_1) $\quad \iota P(\underline{k},x) \leftarrow Q(\underline{k},x) \wedge \neg \delta Q(\underline{k},x) \wedge \neg \mu Q(\underline{k},x,a) \wedge \delta T(\underline{k},x)$

(I_2) $\iota P(\underline{k},x) \leftarrow Q(\underline{k},x) \wedge \neg\, \delta Q(\underline{k},x) \wedge \neg\, \mu Q(\underline{k},x,a) \wedge \mu T(\underline{k},x,b)$
(I_3) $\iota P(\underline{k},x) \leftarrow \iota Q(\underline{k},x) \wedge \neg\, T(\underline{k},x) \wedge \neg\, \iota T(\underline{k},x) \wedge \neg\, \mu T(\underline{k},a,x)$
(I_4) $\iota P(\underline{k},x) \leftarrow \iota Q(\underline{k},x) \wedge \delta T(\underline{k},x)$
(I_5) $\iota P(\underline{k},x) \leftarrow \iota Q(\underline{k},x) \wedge \mu T(\underline{k},x,a)$
(I_6) $\iota P(\underline{k},x) \leftarrow \mu Q(\underline{k},a,x) \wedge \neg\, T(\underline{k},x) \wedge \neg\, \iota T(\underline{k},x) \wedge \neg\, \mu T(\underline{k},b,x) \wedge T(\underline{k},a)$

(D_1) $\delta P(\underline{k},x) \leftarrow \delta Q(\underline{k},x) \wedge \neg\, T(\underline{k},x)$

(M_1) $\mu P(\underline{k},x,x') \leftarrow \mu Q(\underline{k},x,x') \wedge \neg\, T(\underline{k},x') \wedge \neg\, \iota T(\underline{k},x') \wedge \neg\, \mu T(\underline{k},a,x') \wedge \neg\, T(\underline{k},x)$
(M_2) $\mu P(\underline{k},x,x') \leftarrow \mu Q(\underline{k},x,x') \wedge \delta T(\underline{k},x')$
(M_3) $\mu P(\underline{k},x,x') \leftarrow \mu Q(\underline{k},x,x') \wedge \mu T(\underline{k},x',a)$

(C_1) $\iota Ic1(\underline{k},x) \leftarrow S(\underline{k},x) \wedge \neg\, \delta S(\underline{k},x) \wedge \neg\, \mu S(\underline{k},x,a) \wedge \delta P(\underline{k},x)$
(C_2) $\iota Ic1(\underline{k},x) \leftarrow S(\underline{k},x) \wedge \neg\, \delta S(\underline{k},x) \wedge \neg\, \mu S(\underline{k},x,a) \wedge \mu P(\underline{k},x,b)$
(C_6) $\iota Ic1(\underline{k},x) \leftarrow \mu S(\underline{k},a,x) \wedge \neg\, P(\underline{k},x) \wedge \neg\, \iota P(\underline{k},x) \wedge \neg\, \mu P(\underline{k},b,x)$

(C_8) $\iota Ic2(\underline{k},x) \leftarrow Z(\underline{k},x) \wedge \neg\, \delta Z(\underline{k},x) \wedge \neg\, \mu Z(\underline{k},x,a) \wedge \mu Q(\underline{k},b,x)$
(C_9) $\iota Ic2(\underline{k},x) \leftarrow \iota Z(\underline{k},x) \wedge Q(\underline{k},x) \wedge \neg\, \delta Q(\underline{k},x) \wedge \neg\, \mu Q(\underline{k},x,a)$
(C_{10}) $\iota Ic2(\underline{k},x) \leftarrow \iota Z(\underline{k},x) \wedge \iota Q(\underline{k},x)$
(C_{11}) $\iota Ic2(\underline{k},x) \leftarrow \iota Z(\underline{k},x) \wedge \mu Q(\underline{k},a,x)$

In the following table, we summarize the execution of the Precedence Graph of Figure 10 with respect to the initial transaction T = { $\iota Z(\underline{A}, 1)$ }.

Rows of this table represent different states of the ongoing of the mechanism. In each row, we indicate the node that we are visiting with a bold mark (**V**) and we also indicate tokens (X) of the rest of nodes. The column named Transaction T indicates which events belong to the transaction and each new inclusion on T is denoted in italic.

	C_1	C_2	C_6	C_8	C_9	C_{10}	C_{11}	Transaction T
1	X	X	X	X	X	**V**	X	ιZ(A,1)
2	X	X	X	X	X		**V**	ιZ(A,1)
3	X	X	X	X	**V**			ιZ(A,1), *δQ(A,1)*
4	X	**V**	X	X				ιZ(A,1), δQ(A,1)
5	**V**		X	X				ιZ(A,1), δQ(A,1), *μS(A,1,2)*
6			**V**	X				ιZ(A,1), δQ(A,1)
7	**V**		X	X				ιZ(A,1), δQ(A,1), *δS(A,1)*
8			**V**	X				ιZ(A,1), δQ(A,1), δS(A,1)
9				**V**				ιZ(A,1), δQ(A,1), δS(A,1)
10								**ιZ(A,1), δQ(A,1), δS(A,1)**

Initially, a token is assigned to each node of the Precedence Graph. At stages 1 and 2, we select Checking Conditions C_{10} and C_{11}, respectively. They are not violated because even though transaction T contains event ιZ(A,1), T does not contain any deletion nor modification event on predicate Q(A, 1). Then, we can drop their corresponding tokens.

At stage 3 we select node C_9 because it is a node that does not have any incoming edge. It corresponds to a Generation Condition that is violated by transaction T. To repair it, we have two alternatives: δQ(A,1) or μQ(A,1,y). We decide to repair it by means of the first option and, therefore, we include event fact δQ(A,1) into the transaction. Notice that by this repair, an event δP(A,1) is induced by the rule D1. To continue, token of node C_9 is dropped and we send a token to nodes C_1 and C_6.

After the repair of C_9, there are four different candidates to be visited: C_1, C_2, C_6 and C_8. We must selected C_2 because it is the candidate with less incoming edges and all previous nodes had been already visited. His token is removed because it is not violated. Notice that C_2 belongs to a subgraph of the Dependency Graph, then in the next stage, we are forced to choose a marked node of the same subgraph: node C_1. This condition is violated because event δP(A,1) was induced by some previous repair. To repair it, event μS(A,1,2) is included into the transaction. We unmark node C_1. Nodes C_1 and C_2 do not have any token, then the treatement of the subgraph is finished. To continue, we mark C_6 because could be violated by the repair of C_1 and we continue with the rest of the marked nodes of the graph.

Notice that as we have explained in the previous section, cyclic subgraphs do not always correspond to a cycle that at execution time loops infinitely. This is the case of the subgraph composed by conditions C_1 and C_2. In this case, the repair of C_1 does not induce any μP(k,x,x') event.

At stage 6, any node of the subgraph remains marked, then we can select nodes outside the subgraph. We choose node C_6. Notice that this condition C_6 is violated by event $\mu S(\underline{A},1,2)$ and because fact $P(\underline{A},2)$ is not true. It is not possible to repair it by means the insertion $\iota P(\underline{A},2)$ nor the modification event $\mu P(\underline{A},z,2)$. We must come back to the situation previous to the moment we made the last repair. Therefore, at stage 7 we have the same marked graph of stage 5, but now, we repair condition C_1 with the event $\delta S(\underline{A},1)$ instead of event $\mu S(\underline{A},1,2)$. We drop token of C_1 and continue at stage 8.

At immediate stages 8 and 9, nodes C_6 and C_8 are selected. Both are not violated and their token can be dropped.

At the last stage 10, all nodes of the Precedence Graph are unmarked so the execution of the graph is completed. We have obtained a transaction $T'=\{\iota Z(\underline{A},1), \delta Q(\underline{A},1), \delta S(\underline{A},1)\}$ that ensures the consistency of all conditions of this Dependency Graph.

6. Joining Integrity Constraints Maintenance and Checking

Up to now we have been concerned with improving efficiency of the integrity constraints maintenance process. Thus, we have considered for the moment that a deductive database contains only integrity constraints to be maintained. However, it may be interesting for a Deductive Database Management System to distinguish between integrity constraints to be checked and integrity constraints to be maintained since, as we have argued in the introduction, both constraints enforcement policies are reasonable [Win90]. Therefore, it is important to develop methods able to handle appropriately both policies.

Unfortunately, little work has been performed in the past in this direction since the methods proposed up to date are devoted either to one or to the other policy. In this section we will present a first proposal of integration of both integrity constraint enforcement policies which is based on incorporating also in the Precedence Graph the information related to the integrity constraints to be checked.

Integrity constraints to be checked correspond to those constraints that when violated by a transaction involve rolling back of the transaction. Thus, by the own definition of its enforcement policy, no possible repair exists for this kind of integrity constraints. Therefore, all conditions associated to an integrity constraint to be checked correspond to Checking Conditions. For this reason, the only relevant information to be taken into account when incorporating these conditions into the Precedence Graph is the information regarding their potential violations. By considering only this information, the incorporation of these conditions into the Precedence Graph can be performed as explained in section 4.

The procedure for incorporating integrity constraints to be checked into the Precedence Graph can be summarized as follows:

- Consider that all conditions associated to an integrity constraint to be checked correspond to Checking Conditions.
- Obtain the Precedence Graph as explained in section 4, but considering only the dependencies due to potential violations in the case of Checking Conditions.

The following example illustrates our approach.

Example 6.1: Consider again the database of example 4.5, but considering now an additional integrity constraint Ic3, for which we want to apply the integrity constraints checking policy, defined as follows:

$$Ic3(\underline{k},x) \leftarrow R(\underline{k},x) \wedge \neg Z(\underline{k},x)$$

Rules to be added to A(D) due to this new integrity constraint are the following:

(C_{15}) $\iota Ic3(\underline{k},x) \leftarrow R(\underline{k},x) \wedge \neg \delta R(\underline{k},x) \wedge \neg \mu R(\underline{k},x,a) \wedge \delta Z(\underline{k},x)$
(C_{16}) $\iota Ic3(\underline{k},x) \leftarrow R(\underline{k},x) \wedge \neg \delta R(\underline{k},x) \wedge \neg \mu R(\underline{k},x,a) \wedge \mu Z(\underline{k},x,b)$
(C_{17}) $\iota Ic3(\underline{k},x) \leftarrow \iota R(\underline{k},x) \wedge \neg Z(\underline{k},x) \wedge \neg \iota Z(\underline{k},x) \wedge \neg \mu Z(\underline{k},a,x)$
(C_{18}) $\iota Ic3(\underline{k},x) \leftarrow \iota R(\underline{k},x) \wedge \delta Z(\underline{k},x)$
(C_{19}) $\iota Ic3(\underline{k},x) \leftarrow \iota R(\underline{k},x) \wedge \mu Z(\underline{k},x,a)$
(C_{20}) $\iota Ic3(\underline{k},x) \leftarrow \mu R(\underline{k},a,x) \wedge \neg Z(\underline{k},x) \wedge \neg \iota Z(\underline{k},x) \wedge \neg \mu Z(\underline{k},b,x)$

Conditions C_{15} to C_{20} are considered as Checking Conditions since they are associated to an integrity constraint to be checked. Moreover, from the resulting Dependency Graph of Events, we obtain new dependencies related to conditions of the integrity constraint Ic3 in addition to the dependencies already obtained in section 4.2.3:

$C_7, C_8 \rightarrow C_{15}, C_{18}$ with respect to $\delta Z(\underline{k}, x)$
$C_7, C_8 \rightarrow C_{16}, C_{19}$ with respect to $\mu Z(\underline{k}, x, y)$

Note that the conditions C_{17} nor C_{20} may never be violated by repairs of Ic1 or Ic2 since repairs of these integrity constraints do not involve insertions nor modifications of predicate R. On the other hand, none of the conditions C_{15} to C_{20} should be repaired since they correspond to an integrity constraint to be checked. Therefore, no dependency exists from these conditions to conditions associated to Ic1 nor to Ic2.

In the following figure we show the Precedence Graph of example 6.1.

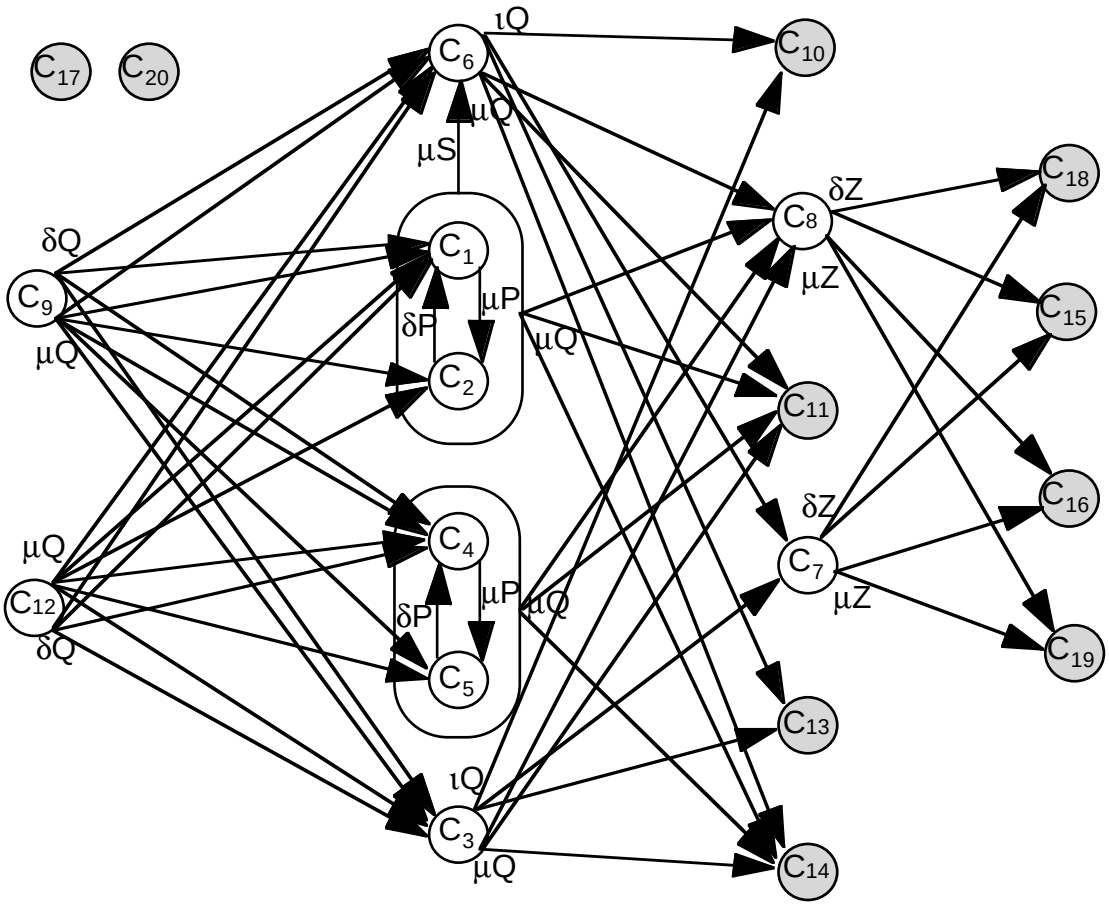


Fig.11 Precedence Graph of Example 6.1

Note that this graph is the same as the graph of Fig.9, but with the inclusion of conditions C_{15} to C_{20}. Conditions C_{17} and C_{20} which are not connected to the rest, are those that may never be violated by repairs of other conditions.

Now, we have obtained a Precedence Graph which includes conditions regarding constraints to be maintained as well as constraints to be checked. Therefore, by executing this graph as we have explained in Section 5, we will be able to deal at the same time with both kinds of integrity constraints.

Little work has been devoted in the past to try to join in a single method the treatment of integrity constraints to be checked and of integrity constraints to be maintained. As far as we know, the only proposal towards this direction is that of [CHM95] which presents a method that follows the integrity constraints checking philosophy, but makes some exceptions by using certain constraints to suggest new updates. However, we believe that much more effort must be devoted to this field if we want to exploit to the maximum the relationship between these two kinds of constraints enforcement polices. For instance, it should be interesting to study how to use the information needed for repairing one constraint for making more efficient the process of checking the violation of other constraints.

7. Relation with Previous Work

Related work can be divided into two groups. The first one comprises the research performed in the field of integrity constraints maintenance related to analyzing and structuring the repair process [Ger93a, Ger93b, Ger94]. The second group contains the research performed in the field of active databases devoted to static analysis techniques for predicting termination and confluence of a given set of active database rules [BW94, KU94, AHW95]. In sections 7.1 and 7.2 we are respectively concerned with the relation between our approach and the techniques proposed in both groups.

7.1 Analyzing and Structuring Reactions

As we said in the introduction, little attention has been devoted to efficiency issues by the methods proposed up to date for integrity constraints maintenance. a significative exception is the work performed by Gertz in this direction [Ger93a, Ger93b, Ger94].

Gertz proposes to carry out at definition time the analysis and the specification of reactions on constraint violations. In this sense, he provides a declarative specification language for reactions on violations suitable to express several integrity constraints enforcement policies. He describes how to obtain, once the integrity constraints and their corresponding reactions have been specified by the designer, a dependency graph which expresses the relationship between repairs and potential violations of integrity constraints. Finally, he presents also a procedure for deriving integrity enforcing triggers from this dependency graph. Execution of these triggers guaranties that a transaction applied to a database maintains the integrity constraints.

Several differences exist between Gertz's proposal and ours. The first one is related to the way of handling integrity constraints maintenance. Gertz proposes the designer to explicitly specify reactions to integrity constraints violations, while we consider these reactions to be automatically generated from the definition of the integrity constraints. Thus, looking for dependencies between integrity constraints is more complex in our approach since they are not explicitly stated and have to be implicitly derived from the integrity constraints definition.

Another important difference refers to the expresiveness of the definition language considered in both proposals. Gertz's proposal is restricted to databases without deductive rules, thus considering only flat integrity constraints (i.e. constraints that are defined only by means of base predicates); and it is restricted also to integrity constraints in Implicative Normal Form (which does not allow negation in the body of a constraint). On the contrary, we handle deductive rules as well as non-flat integrity constraints and we allow negation to appear in the body of the rules and of the constraints (in fact, the only requirements we impose on the database are those of allowedness and stratification which are much more general than Gertz requirements). Thus, our technique can be applied in more cases than Gertz's technique. It is also worth to mention the additional complexity of our approach due to the fact that we have to take the definition of derived predicates into account.

Finally, if more than one dependency exists between two integrity constraints, Gertz forces to the designer to weight all possible reactions to indicate which reaction should be considered with priority. Thus, it is garanteed that at execution time only one repair is considered for a concrete violation of an integrity constraint. On the contrary, we take into account all possible repairs of a given integrity constraint definition. Thus, we will be able to restore database consistency in cases where Gertz is not able to do it since the designer may not have appropiately weighted the repairs of integrity constraints.

7.2 Static Analysis Techniques for Active Database Rules

Work related to ours has also been developed in the field of active databases. Rules in active databases can be very difficult to program due to the unstructured and unpredictable nature of rule processing. For this reason, a significant amount of work has been devoted to statically analyzing sets of active database rules to predict in advance aspects of rule behaviour such as *termination* (that is, if the rules are guaranteed to terminate) or *confluence* (that is, if the rules are guaranteed to produce a unique final database state). See for instance [BW94, KU94, AHW95] and the references therein.

The techniques proposed in the mentioned references are based on the definition of a graph which explicitly states the relationship between the activation of rules. Construction of this graph is based on predicting how a database query (i.e. a rule condition) can be affected by the execution of a data modification operation (i.e. a rule action). Thus, this graph will contain and edge from an active rule r_1 to another active rule r_2 if the action performed by r_1 may activate r_2.

In addition to the different framework of active databases versus deductive databases, there exist two important differences between this work and the one we present in this paper. First, static analysis techniques for active database rules consider that the update performed by the execution of a rule is explicitly stated, while we have to determine the possible ways in which a violated condition may be repaired. That is, we need first to analyse integrity constraints for determining the update performed by the repair of a condition. Second, work in active databases restricts the updates appearing in the action part of a rule to be base fact updates, while we consider also view updates. It is worth to mention the additional complexity of our approach motivated by the need to translate view updates into updates of base facts.

8 Conclusions

In this paper we have presented a technique for improving efficiency of the integrity constraints maintenance process. This technique is based on the definition of a graph, the Precedence Graph, which explicitly states the relationship between repairs of an integrity constraint and potential violations of other integrity constraints. We have also proposed an algorithm for executing the Precedence Graph which aims to minimize the number of times that an integrity constraint must be considered during the maintenance process.

We have also shown how our technique could be extended to takeg into account integrity constraints to be checked in addition to integrity constraints to be maintained. Thus, we have integrated into a single method both integrity constraint enforcement policies, what constitutes in our opinion an important advance in this field.

Our technique is directly applicable to the methods we have proposed in the past for updating consistent deductive databases [MT93, MT95, TO95] and it could be easily adapted for improving efficiency of most of the other existing integrity constraints maintenance methods.

Acknowledgements

We are grateful to D. Costal, A. Olivé, J. A. Pastor, C. Quer, M. R. Sancho, J. Sistac and T. Urpí for many useful comments and discussions. This work has been partially supported by the CICYT PRONTIC program project TIC94-0512.

References

[Abi88] Abiteboul, S. "Updates, a New Frontier", Int. Conf. on Database Theory (ICDT'88), Springer, 1988, pp. 1 - 18.

[AHW95] Aiken, A.; Hellerstein, J.M.; Widom, J. "Static Analysis Techniques for Predicting the Behavior of Active Database Rules", ACM Transactions on Database Systems, Vol. 20, Nº 1, Mrach 1995, pp. 3-41.

[BR86] Bancilhon, F.; Ramakrishnan, R, "An Amateur's Introduction to Recursive Query Processing", Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington D.C., 1986.

[BW94] Baralis, E.; Widom, J. "An Algebraic Approach to Rule Analysis in Expert Database Systems", Proc. of the 20th VLDB Conference, Santiago, Chile, 1994, pp. 475-486.

[CGMD94] Celma, M; García, C.; Mata, L.; Decker, H. "Comparing and Synthesising Integrity Checking Methods for Deductive Databases", Int. Conf. on Data Engineering (ICDE'94), Houston (Texas), 1994, pp. 214-222.

[CHM95] Chen, I.A.; Hull, R.; McLeod, D. "An Execution Model for Limited Ambiguity Rules and Its Application to Derived Data Update". ACM Transactions on Database Systems, Vol. 20, Nº 4, December 1995, pp. 365-413.

[Cos95] Costal, D. "Un mètode de planificació basat en l'actualització de vistes en bases de dades deductives", PhD Thesis, Barcelona, 1995 (in catalan).

[CW90] Ceri, S.; Widom, J. "Deriving Production Rules for Constraint Maintenance", Proc. of the 16th VLDB Conference, Brisbane, Australia, 1990, pp. 566-577.

[CFPT92] Ceri, S.; Fraternali, P.; Paraboschi, S.; Tanca, L. "Integrity Maintenance Systems: an architecture", Third Int. Workshop on the Deductive Approach to Information Systems and Databases, Roses, Catalonia, 1992, pp. 327-344.

[Dec89] Decker, H. "The Range Form of databases or: How to avoid Floundering", Proc. 5th ÖGAI, Springer-Verlag, 1989.

[Dec96] Decker, H. "An Extension of SLD by Abduction and Integrity Maintenance for View Updating in Deductive Databases", To appear in Joint International Conference and Symposium on Logic Programming (JICSLP'96), Bonn (Germany), 1996.

[Ger93a] Gertz, M. "On Specifying the Reactive Behavior on Constraint Violations", Informatik-Berichte 2/93, Institut für Informatik, Universität Hannover, 1993.

[Ger93b] Gertz, M.; Lipeck, U.W. "Deriving Integrity Maintaining Triggers from Transaction Graphs", International Conference on Data Engineering (ICDE'93), Vienne, 1993, pp. 22-29.

[Ger94] Gertz, M. "Specifying Reactive Integrity Control for Active Databases", Research Issiues on Data Engineering: Active Databases (RIDE-ADS'94), Houston, Texas, 1994, pp. 62-70.

[KM90] Kakas, A.; Mancarella, P. "Database Updates through Abduction", Proc. of the 16th VLDB Conference, Brisbane, Australia, 1990, pp. 650-661.

[KU94] Karadimce, A.P.; Urban, S.D. "Conditional Term Rewriting as a Formal Basis for Analysis of Active Database Rules" Research Issiues on Data Engineering: Active Databases (RIDE-ADS'94), Houston, Texas, 1994, pp. 156-162.

[Llo87] Lloyd, J.W. "Foundations on Logic Programming", 2nd edition, Springer, 1987.

[LT84] Lloyd, J.W.; Topor, R.W. “Making Prolog More Expressive”. Journal of Logic Programming, 1984, No. 3, pp. 225-240.

[ML91] Moerkotte, G; Lockemann, P.C. "Reactive Consistency Control in Deductive Databases", ACM Transactions on Database Systems, Vol. 16, No. 4, December 1991, pp. 670-702.

[MT93] Mayol, E.; Teniente, E. " Incorporating Modification Requests in Updating Consistent Knowledge Bases", Fourth Int. Workshop on the Deductive Approach to Information Systems and Databases, Lloret de Mar, Catalonia, 1993, pp. 335-360.

[MT95] Mayol, E.; Teniente, E. "Towards an Efficient Method for Updating Consistent Deductive Databases", Basque International Workshop on Information Techlogogy (BIWIT'96): Data Management Systems, IEEE Computer Society Press, San Sebastian, Spain, 1996, pp. 113-122.

[Oli91] Olivé, A. "Integrity Checking in Deductive Databases", Proc. of the 17th VLDB Conference, Barcelona, Catalonia, 1991, pp. 513-523.

[Pet81] Peterson, J.L. "Petri Net Theory and the Modeling of Systems", Prentice-Hall Inc., 1981.

[TO95] Teniente, E.; Olivé. A. "Updating Knowledge Bases while Maintaining their Consistency", The VLDB Journal, Vol. 4, Num. 2, 1995, pp. 193-241.

[Ull88] Ullman, J.D. "Principles of Database and Knowledge-Base Systems", Computer Science Press, New York, 1988.

[UO92] Urpí, T.; Olivé, A. "A Method for Change Computation in Deductive Databases", Proc. of the 18th VLDB Conference, Vancouver, 1992, pp. 225-237.

[Wil79] Wilson, R.J. "Introduction to Graph Theory", 2on Edition, Longman Editors, 1979.

[Win90] Winslett, M. "Updating Logical Databases", Cambridge Tracts in Theoretical Computer Science 9, 1990.

[Wüt93] Wüthrich, B. "On Updates and Inconsistency Repairing in Deductive databases", Int. Conf. on Data Engineering, Vienna, 1993, pp. 608 - 615.