Consistency Preserving Updates in Deductive Databases

Enric Mayol
Ernest Teniente

[mayol | teniente]@lsi.upc.es
Dept.de Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya (UPC)

Abstract

Several problems may arise when a deductive database is updated. The problems that are addressed in this paper are those of integrity constraint maintenance *and* view updating. *In this sense, we define a method that tackles both problems in an integrated way and that it is sound and complete. We also propose an architecture for our method to deal with both problems efficiently.*

Improvement of efficiency during the integrity constraint maintenance is based on a technique that determines the order in which integrity constraints should be handled. This technique is based on the generation of a graph that states the relationships between potential violations and potential repairs of integrity constraints. This order reduces significantly the number of times that each integrity constraint has to be considered after any integrity constraint repair. To improve efficiency during view updating, we propose to perform an initial analysis of the update request to reduce the number of database accesses and to explore only the relevant alternatives that may lead to valid solutions of an update request.

Furthermore, a detailed comparison considering effectiveness and efficiency issues is also provided with respect to other methods that also deal with integrity constraint maintenance and view updating.

KEYWORDS: Deductive Databases, Updating, Integrity Constraints Maintenance

1. Introduction

Most databases, like relational or deductive ones, allow the definition of intentional information like views or integrity constraints. Intentional information is defined by means of rules that allow the deduction of new data (i.e. intentional data) based in data explicitly stored in the database (i.e. extensional data, like tuples in a relational database or base facts in a deductive one). Views and integrity constraints are the most traditional types of intentional information. *Views* are defined by means of deductive rules that allow the definition of view (derived) facts from stored (base) facts, while *integrity constraints* state conditions that must be satisfied by each state of the database.

Several problems may arise when updating a deductive database [TU95]. A well-known problem 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 contents of the database, may falsify some integrity constraint. There are several approaches to resolve this conflict [Win90]. All of them are reasonable and the correct approach to be considered depends on the semantics of the integrity constraints and of the database. The best known approaches are integrity constraint checking and integrity constraint maintenance.

The classical approach to deal with this problem is that of *integrity constraint checking* [GL90, Oli91, GCMD94, Sel95]. It is concerned with detecting whether a given update violates some integrity constraint, but it is the most conservative approach since it rejects the requested update when any integrity constraint becomes violated. The main drawback of this approach is that the user may be completely lost regarding additional changes to be made to satisfy the integrity constraints.

An alternative approach, aimed at overcoming this limitation, is that of *integrity constraint maintenance* [KM90, ML91, Wüt93, CFPT94, Ger94, CST95, TO95, Dec97, LT97, Maa98, Sch98]. It tries to identify additional updates (i.e. repairs) to add to the original request, to guarantee that integrity constraints do not become violated.

Views provide several advantages like favoring logical data independence or improving expressiveness of integrity constraints definition[1]. However, since the view extension is completely defined by the application of deductive rules to the contents of the database, changes requested on a view must always be translated into changes of the stored base facts.

The problem of appropriately translating updates of a set of derived facts into appropriate updates of the underlying base facts is known as *view updating* [GL90, Wüt93, CHM95, CST95, TO95, Dec97, LT97]. In general, several translations that satisfy the requested update exist. Each translation defines a possible transaction that, if applied to the current database, would satisfy the requested update.

View updating and integrity constraint maintenance problems are strongly related. An update obtained as the translation of a view update request could violate an integrity constraint. On the other hand, when integrity constraints are defined through derived

[1] One can argue that, from a theoretical point of view, constraints involving views can be reduced to constraints on base predicates only, by applying view definition. However, this is not always possible. Consider for example integrity constraint $\leftarrow Q(x) \wedge \neg P(x)$ assuming that P(x) is defined by $P(x) \leftarrow R(x,y) \wedge \neg S(y)$.

predicates, to repair them may require the request of a view update. In both cases, a repair of an integrity constraint must always satisfy the rest of the integrity constraints, and moreover, it must preserve the requested (view) update. Therefore, view updating and integrity constraint maintenance problems can only be dealt satisfactorily in an integrated way. The following example illustrates the interrelationship between both problems.

Example 1.1: Consider a database that contains a derived predicate Doctor(p) that defines that a person p is a doctor if s/he has written a PhD-Thesis and has passed the PhD-exam. Derived predicate ResCert(p) states that a person has a research certificate if s/he has written a PhD-Thesis.

Integrity constraint Ic1 states that it is not possible to have a research certificate without been author of some good research paper. Ic2 states that it is not allowed to be a professor and not to be a doctor at the same time. Meanwhile, Ic3 establishes that is not possible to pass satisfactorily the PhD-exam if you make some errors during the examination. Extensional database contains the fact PassEx(Bob).

$Doctor(p) \leftarrow PhD(p) \wedge PassEx(p)$ $\quad\quad$ $PassEx(Bob)$

$ResCert(p) \leftarrow PhD(p)$

$Ic1(p) \leftarrow ResCert(p) \wedge \neg GoodPap(p)$ $\quad\quad$ $Ic2(p) \leftarrow Prof(p) \wedge \neg Doctor(p)$

$Ic3(p) \leftarrow Errors(p) \wedge PassEx(p)$

The insertion of the fact Prof(Bob) into this database violates integrity constraint Ic2. To repair it, it is necessary to translate the request of inserting derived fact Doctor(Bob) into the insertion of base fact PhD(Bob). Notice that in this case, the integrity constraint maintenance problem requires solving a view update request.

Moreover, insertion of PhD(Bob) violates integrity constraint Ic1 since fact ResCert(Bob) is induced. To repair Ic1, base fact GoodPap(Bob) must be inserted. Notice that in this case, translations of a view update request must also maintain integrity constraints satisfied.

Assume now, a request to insert facts Doctor(Bob) and Errors(Bob) at the same time. As before, insertion of fact Doctor(Bob) is achieved by the insertion of fact PhD(Bob). Integrity constraint Ic1 and Ic3 are violated and they must be repaired with the insertion of fact GoodPap(Bob), and the deletion of fact PassEx(Bob), respectively. In this sense, the set of insertion of facts PhD(Bob), GoodPap(Bob) and Errors(Bob), and the deletion of fact PassEx(Bob) maintain integrity constraints satisfied. But notice that, they do not allow the satisfaction of the initial request of inserting the derived fact Doctor(Bob), since fact PassEx(Bob) is deleted (to satisfy Ic3) and it must remain true to induce derived fact Doctor(Bob). In this case, there is no way to satisfy the initial update request and the integrity constraints at the same time by changing only database facts. This kind of situations can only be detected by methods that deal with both problems in an integrated way.

In last years, several methods have been proposed to deal with both problems. Some of them [KM90, Wüt93, TO95, Dec97] consider view updating and integrity constraint maintenance without imposing significant restrictions on the integrity constraints they can handle. Other methods either impose significant restrictions on the constraints they can deal with, like [CHM95, CST95, LT97]; consider an integrity constraint checking approach

[GL90] or do not tackle the view updating problem at all [ML91, CFPT94, Ger94, Maa98, Sch98]. However, most of these methods present important limitations regarding correctness and completeness. In some cases, they obtain solutions that could not satisfy the initial update request, and in other cases, they can not obtain some of the existing correct solutions to an update request.

Another important weakness of most of the methods proposed in the past for integrity maintenance and/or view updating relies on the fact that they do not consider explicitly efficiency issues. They are more oriented with the generation of a complete set of repairs of integrity constraint violations and/or the complete set of translations to a view update request. In the integrity constraint maintenance field, few methods like [CFPT94, Ger94] take into account efficiency issues in their proposals. These methods consider an explicit order to maintain integrity constraints to reduce the number of times each integrity constraint must be considered. The rest of methods we know in the field check again all constraints for consistency when a constraint is repaired, although they were already satisfied prior to the repair and they could not become violated by the repair. This situation is illustrated in the following example.

Example 1.2: Assume a database that contains the following three integrity constraints Ic1, Ic2 and Ic3, and the update request to insert fact Employee(Ann).

$$Ic1(p) \leftarrow Worker(p) \wedge \neg HasSalary(p)$$

$$Ic2(p) \leftarrow Contracted(p) \wedge \neg Worker(p)$$

$$Ic3(p) \leftarrow Employee(p) \wedge \neg Contracted(p)$$

Consider a method that handles integrity constraints in the sequential order (i.e. Ic1, Ic2, Ic3). To maintain all these integrity constraints, this method will check constraints in the following order {Ic1, Ic2, Ic3, Ic1, Ic2, Ic1, Ic2, Ic3}. The obtained solution is composed by insertion of facts Employee(Ann), Contracted(Ann), Worker(Ann) and HasSalary(Ann). Observe that, the repair of Ic3 is obtained after having checked {Ic1, Ic2, Ic3}. Therefore, the process must check again Ic1 and Ic2 to detect new constraint violations due to the insertion of Contracted(Ann). This insertion violates integrity constraint Ic2 and it must be repaired by inserting fact Worker(Ann). The process restarts again checking all constraints. Finally, by checking integrity constraints in this order have required to check eight integrity constraints.

By the way, if we take into account the interaction among repairs and possible violations of constraints, it is not difficult to see that to obtain the previous solution it is enough to maintain integrity constraints in this order {Ic3, Ic2, Ic1}. The idea is that there is an implicit order Ic3 -> Ic2 -> Ic1 of dealing with these constraints. The insertion of Employee(Ann) can only violate Ic3 and its repair can only violate Ic2. The repair of Ic2 can only violate Ic1 and its repair does not violate neither Ic2 nor Ic3. Finally, we only have checked three integrity constraints.

Therefore, taking into account this information would help to improve the efficiency of the integrity maintenance process by not considering again integrity constraints already checked. In presence of views, this process is more difficult since view definitions must be taken into account to identify this order.

A second difficulty that could appear when views are considered is the need to translate in an efficient way view repair requests during the process of integrity maintenance. Existing methods for view updating have paid little attention to efficiency issues. Thus, for instance, they do not care about exploring alternatives that do not lead to valid translations or performing unnecessary accesses to the extensional database facts.

In this paper, we propose a new method that addresses, in an integrated way, the problems of view updating and integrity constraint maintenance. Given a view update request, the main goal of our method is to obtain all possible ways to satisfy both the update request and all integrity constraints. Moreover, we address efficiency issues during this process. In this sense, we propose a technique for determining the order in which integrity constraints should be handled and a technique for translating view update requests efficiently.

This paper is organized as follows. Section 2 and 3 review the concept of deductive database and the concept of Augmented Database (A(D)) proposed in [UO92]. In Section 4, we provide the definition of our method, and we prove its soundness and completeness. In Section 5, we compare the effectiveness of our method with respect to other relevant work in the field. Section 6 describes the architecture we propose for our method, putting special attention in describing the techniques we introduce to improve the efficiency of the method. In Section 7, we provide a comparison of our method with respect to some methods that explicitly consider efficiency issues in their definition. Finally, Section 8 presents our conclusions.

2. Deductive Databases

In this section, we briefly review some definitions of the basic concepts related to deductive databases [Llo87] 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.

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: $P(t_1, ..., t_m) \leftarrow$, where $P(t_1, ..., t_m)$ is a ground atom. We assume that each m-ary predicate has a subset of arguments t_i (i=1, ..., k with $1 \leq k \leq m$) that form the *key*.

A *deductive rule* is a formula of the form[2]: $P(\underline{t_1,...,t_k},t_{k+1},...,t_m) \leftarrow L_1 \wedge ... \wedge L_n$, with $n \geq 1$, where $P(\underline{t_1,...,t_k},t_{k+1},...,t_m)$ is an atom denoting the conclusion, and $L_1,...,L_n$ are literals. Any variable in $P(\underline{t_1,...,t_k},t_{k+1},...,t_m)$, L_1, ..., L_n is assumed to be universally quantified over the whole formula. A derived predicate 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: $\leftarrow L_1 \wedge ... \wedge L_m$, with $m \geq 1$, where each L_i is a literal and all variables are assumed to be universally quantified over the formula. More general constraints like foreign keys or functional dependencies can be transformed into this form by using [LlT84].

[2] Underlined arguments correspond to the key arguments of that predicate

For the sake of uniformity, we associate an inconsistency predicate Ic_n, with or without terms to each integrity constraint. Then, we would rewrite the former denial as: $Ic_n \leftarrow L_1 \wedge ... \wedge L_m$, with $m \geq 1$. Note that an inconsistency predicate will be true only if the corresponding constraint is violated. We assume also that the database contains a distinguished inconsistency predicate Ic defined by n rules $Ic \leftarrow Ic_j$ (j=1..n). That is, one rule for each integrity constraint Ic_j of the database. Note that Ic will only hold in those database states in which some integrity constraint is violated.

To enforce the concept of key, we assume that there is a key integrity constraint associated to each predicate $P(\underline{t_1,...,t_k},t_{k+1},...,t_m)$ defined in the following form:

$$Ic_k \leftarrow P(\underline{t_1,...,t_k},t_{k+1},...,t_m) \wedge P(\underline{t_1,...,t_k},t'_{k+1},...,t'_m) \wedge [t_{k+1},...,t_m] \neq [t'_{k+1},...,t'_m]$$

These constraints are not explicitly defined in the database schema since they are implicitly handled by our method.

A *deductive database* D is a triple (EDB, IDB, IC), where EDB is a set of base 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.

We 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 database to be *allowed* [Llo87]; that is, any variable that occurs in the body of a deductive rule has an occurrence in a positive literal of a database predicate.

Example 2.1: The following deductive database will be used thought the paper:

Works(Mercè, UPC) — Ic1(p,n) ← IdNum(p,n) ∧ ¬Contracted(p)

Emp(p,c) ← Works(p,c) ∧ Cont(p,c) — Ic2(p,c) ← Emp(p,c) ∧ ¬Lab_age(p)

Contracted(p) ← Cont(p,c)

This database contains four base predicates and two derived predicates:

Works(p, c) states that a person p works in a company c.

Cont(p, c) states that a person p has a contract with the company c.

IdNum(p, n) states that a person p has the working identification number n.

Lab_age(p) states that a person p is in the legal labour age.

Empl(p, c) states that a person p is an employee in a company c if he/she works and has a contract in c.

Contracted(p) states that a person p is contracted if it has a contract with some company.

It contains also two integrity constraints: Ic1 states that persons that have an identification number must be contracted. Integrity constraint Ic2 states that it is not possible to be an employee if s/he is not in the legal labour age.

3. Augmented Database A(D)

The proposed method is based on a set of rules that define the exact difference between two consecutive database states. A transition between two consecutive states is defined by the application of a given transaction T that consists of a set of base fact updates. This set of rules, together with the original database D, form the Augmented Database A(D).

In this section, we review the Augmented Database definition proposed in [UO92] and partially reformulated in [May00].

3.1 Events

Let T be a transaction, D be a deductive database and P be a predicate in D. Moreover, D^n corresponds to the updated database and P^n denote predicate P evaluated in D^n. We say that T induces a transition from D (the old state) to D^n (the new state). We assume for the moment that T consists of an unspecified set of base facts to be inserted, deleted and/or modified.

Due to the presence of deductive rules and integrity constraints, the application of T may induce other updates on some derived or inconsistency predicates. We formalize all changes on database predicates with the concept of *event*. Formally, we associate to each database predicate P, an *insertion event predicate ιP*, a *deletion event predicate δP* and a *modification event predicate* μP.

Definition 3.1: Let P($\underline{\mathbf{k}}$, **x**) be a database predicate, where **k** and **x** are vectors of variables. We define the *insertion event predicate* ιP($\underline{\mathbf{k}}$, **x**), the *deletion event predicate* δP($\underline{\mathbf{k}}$, **x**) and the *modification event predicate* μP($\underline{\mathbf{k}}$, **x**, **x'**) in the following way[3]:

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

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

$$\forall \mathbf{k}, \mathbf{x}, \mathbf{x'}\ (\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \leftrightarrow P(\underline{\mathbf{k}}, \mathbf{x}) \wedge P^n(\underline{\mathbf{k}}, \mathbf{x'}) \wedge \mathbf{x} \neq \mathbf{x'})$$

Notice that insertion and deletion event predicates are defined also for predicates without non-key arguments P($\underline{\mathbf{k}}$), while modification event predicate is applicable only to predicates with non-key arguments.

From the above definitions, we can deduce which are the *database requirements* necessary to allow an event to occur; when two events can not occur simultaneously, that is, when two events are *mutually exclusive*; and when a fact will be true (or false) in the new state of the database.

Definition 3.2: Let P($\underline{\mathbf{k}}$, **x**) be a database predicate, and ιP($\underline{\mathbf{k}}$, **x**), δP($\underline{\mathbf{k}}$, **x**), μP($\underline{\mathbf{k}}$, **x**, **x'**) the associated insertion, deletion and modification event predicates, respectively. The following rules define the *database requirements* that must be satisfied by the old database to allow an event to occur:

$$\forall \mathbf{k}, \mathbf{x}\ (\iota P(\underline{\mathbf{k}}, \mathbf{x}) \rightarrow \neg \exists\, \mathbf{y}\ P(\underline{\mathbf{k}}, \mathbf{y}))$$

$$\forall \mathbf{k}, \mathbf{x}\ (\delta P(\underline{\mathbf{k}}, \mathbf{x}) \rightarrow P(\underline{\mathbf{k}}, \mathbf{x}))$$

$$\forall \mathbf{k}, \mathbf{x}, \mathbf{x'}\ (\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \rightarrow P(\underline{\mathbf{k}}, \mathbf{x}) \wedge \mathbf{x} \neq \mathbf{x'})$$

Definition 3.3: Let P($\underline{\mathbf{k}}$, **x**) be a database predicate, and ιP($\underline{\mathbf{k}}$, **x**), δP($\underline{\mathbf{k}}$, **x**), μP($\underline{\mathbf{k}}$, **x**, **x'**) the associated insertion, deletion and modification event predicates, respectively. We say that two events are *mutually exclusive* if they can not hold at the same time. This is formalized by the following rules:

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

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

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

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

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

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

[3] k, x, x' and y are vectors of variables and underlined arguments correspond to the key of predicate.

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

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

Definition 3.4: Let $P(\underline{\mathbf{k}}, \mathbf{x})$ be a database predicate; $\iota P(\underline{\mathbf{k}}, \mathbf{x})$, $\delta P(\underline{\mathbf{k}}, \mathbf{x})$, $\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'})$ the associated insertion, deletion and modification event predicates, respectively; and $P^n(\underline{\mathbf{k}}, \mathbf{x})$ the evaluation of predicate $P(\underline{\mathbf{k}}, \mathbf{x})$ in the new state of the database (D^n). The following equivalencies define the transition of a predicate between two consecutive states:

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

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

Example 3.1: Consider the database of Example 2.1 and the request to insert that Mercè has the working identification number 12345. This update is modeled by the base event ιIdNum(Mercè, 12345). Notice that to occur this event, it is necessary that old state of database ensures that any fact IdNum(Mercè, n) holds. Moreover, in the same transition, none of the following events can occur: ιIdNum(Mercè, n), with n≠12345; δIdNum(Mercè, n) or μIdNum(Mercè, n, n') for any value n and n'. If all of these conditions are satisfied, then the key integrity constraint of predicate IdNum(p,n) is correctly maintained, and the effect of the request will be perceived in the new state.

If $P(\underline{\mathbf{k}}, \mathbf{x})$ is a base predicate, $\iota P(\underline{\mathbf{k}}, \mathbf{x})$, $\delta P(\underline{\mathbf{k}}, \mathbf{x})$ and $\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'})$ are *base events* and they represent insertions, deletions and modifications of base facts. Therefore, we will assume that a *transaction T* consists of a set of non-mutually exclusive base event facts, whose database requirements are satisfied.

If $P(\underline{\mathbf{k}}, \mathbf{x})$ is a derived predicate, $\iota P(\underline{\mathbf{k}}, \mathbf{x})$, $\delta P(\underline{\mathbf{k}}, \mathbf{x})$ and $\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'})$ are *derived events* and they represent induced insertions, deletions and modifications, respectively. If $P(\underline{\mathbf{k}}, \mathbf{x})$ is an inconsistency predicate, $\iota P(\underline{\mathbf{k}}, \mathbf{x})$ represents a violation of the corresponding integrity constraint. For inconsistency predicates, $\delta P(\underline{\mathbf{k}}, \mathbf{x})$ and $\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'})$ events are not defined since we assume that the database is consistent before the update.

3.2 Transition rules

A first group of rules that form the Augmented Database A(D) are the *Transition rules*. These rules define the extension of derived and inconsistency predicates in the new state of the database ($P^n(\underline{\mathbf{k}}, \mathbf{x})$) in terms of the extension of the old state and the events that occur during the transition.

Let $P(\underline{\mathbf{k}}, \mathbf{x})$ be a derived or inconsistency predicate of the deductive database defined by m≥1 deductive rules. For our purposes, we rename predicate symbols in the conclusions of these rules by $P_1(\underline{\mathbf{k}}, \mathbf{x})$, ..., $P_m(\underline{\mathbf{k}}, \mathbf{x})$ and add the set of clauses:

$P(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow P_i(\underline{\mathbf{k}}, \mathbf{x}) \qquad i = 1...m$

Consider a deductive rule of predicate $P_i(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow L_1 \wedge .. \wedge L_r$ (r = 1...n), and its evaluation in the new state written in the following form: $P^n_i(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow L^n_1 \wedge ... \wedge L^n_r$. Consider a literal

L^n_j of the above rule that evaluates true (in the new state), therefore, taking into account equivalencies (1) and (2) of definition 3.4, we can determine if this literal has been *unchanged* ($U(L^n_j)$); or *inserted* ($I(L^n_j)$); or *modified* ($M(L^n_j)$) during the transition. The following expressions define this information[4]:

$U(L^n_r) = Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r}) \wedge \neg \delta Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r}) \wedge \neg \mu Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r}, \mathbf{x_r'})$ $\quad L^n_r = Q^n_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r})$

$= \neg Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r}) \wedge \neg \iota Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r}) \wedge \neg \mu Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r'}, \mathbf{x_r})$ $\quad = \neg Q^n_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r})$

$= L_r$ $\quad$ evaluable

$I(L^n_r) = \neg \exists \mathbf{y}\, Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{y}) \wedge \iota Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r})$ $\quad L^n_r = Q^n_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r})$

$= Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r}) \wedge \delta Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r})$ $\quad = \neg Q^n_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r})$

$M(L^n_r) = Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r'}) \wedge \mu Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r'}, \mathbf{x_r}) \wedge \mathbf{x_r'} \neq \mathbf{x_r}$ $\quad L^n_r = Q^n_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r})$

$= Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r}) \wedge \mu Q_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r}, \mathbf{x_r'}) \wedge \mathbf{x_r} \neq \mathbf{x_r'}$ $\quad = \neg Q^n_r(\underline{\mathbf{k}}_{\mathbf{r}}, \mathbf{x_r})$

Using these expressions, we formally define *transition rules*, which state all possible ways to satisfy a derived or inconsistency fact in the new state of the database ($P^n(\underline{\mathbf{k}}, \mathbf{x})$).

Definition 3.5: Let $P(\underline{\mathbf{k}}, \mathbf{x})$ be a derived or inconsistency predicate defined by m≥1 deductive rules. Then, for each predicate $P_i(\underline{\mathbf{k}}, \mathbf{x})$ (i=1 ...m) we define its *transition rules* in the following way:

$P^n_i(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow P^n_{i,j}(\underline{\mathbf{k}}, \mathbf{x})$ $\quad j = 1 \ldots \alpha$

$P^n_{i,j}(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow \wedge^n_{r=1} [U(L^n_r) \mid I(L^n_r) \mid M(L^n_r)]$ $\quad j = 1 \ldots \alpha$

with $\alpha = 3^{nk_i} * 2^{k_i}$, where nk_i is the number of database literals with non-key arguments that appear in the derivation rule of predicate $P_i(\underline{\mathbf{k}}, \mathbf{x})$, while k_i is the number of database literals with only key arguments.

We assume that transition rule for j=1 always corresponds to the following one:

$P^n_{i,1}(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow U(L^n_1) \wedge \ldots \wedge U(L^n_n)$

Example 3.2: Consider the derived predicate Emp(p,c) of Example 2.1. The transition rules of this predicate are the following:

$Emp^n_1(\underline{p},c) \leftarrow Emp^n_{1,j}(\underline{p},c)$ $\quad j = 1 \ldots 9$

$Emp^n_{1,1}(\underline{p},c) \leftarrow Works(\underline{p},c) \wedge \neg\delta Works(\underline{p},c) \wedge \neg\mu Works(\underline{p},c,c') \wedge Cont(\underline{p},c) \wedge \neg\delta Cont(\underline{p},c) \wedge \neg\mu Cont(\underline{p},c,c')$

$Emp^n_{1,2}(\underline{p},c) \leftarrow Works(\underline{p},c) \wedge \neg\delta Works(\underline{p},c) \wedge \neg\mu Works(\underline{p},c,c') \wedge \neg Cont(\underline{p},y) \wedge \iota Cont(\underline{p},c)$

$Emp^n_{1,3}(\underline{p},c) \leftarrow Works(\underline{p},c) \wedge \neg\delta Works(\underline{p},c) \wedge \neg\mu Works(\underline{p},c,c') \wedge Cont(\underline{p},c') \wedge \mu Cont(\underline{p},c',c) \wedge c' \neq c$

$Emp^n_{1,4}(\underline{p},c) \leftarrow \neg Works(\underline{p},y) \wedge \iota Works(\underline{p},c) \wedge Cont(\underline{p},c) \wedge \neg\delta Cont(\underline{p},c) \wedge \neg\mu Cont(\underline{p},c,c')$

$Emp^n_{1,5}(\underline{p},c) \leftarrow \neg Works(\underline{p},y) \wedge \iota Works(\underline{p},c) \wedge \neg Cont(\underline{p},x) \wedge \iota Cont(\underline{p},c)$

$Emp^n_{1,6}(\underline{p},c) \leftarrow \neg Works(\underline{p},y) \wedge \iota Works(\underline{p},c) \wedge Cont(\underline{p},c') \wedge \mu Cont(\underline{p},c',c) \wedge c' \neq c$

$Emp^n_{1,7}(\underline{p},c) \leftarrow Works(\underline{p},c') \wedge \mu Works(\underline{p},c',c) \wedge Cont(\underline{p},c) \wedge \neg\delta Cont(\underline{p},c) \wedge \neg\mu Cont(\underline{p},c,c') \wedge c' \neq c$

$Emp^n_{1,8}(\underline{p},c) \leftarrow Works(\underline{p},c') \wedge \mu Works(\underline{p},c',c) \wedge \neg Cont(\underline{p},y) \wedge \iota Cont(\underline{p},c) \wedge c' \neq c$

$Emp^n_{1,9}(\underline{p},c) \leftarrow Works(\underline{p},c') \wedge \mu Works(\underline{p},c',c) \wedge Cont(\underline{p},c'',c) \wedge \mu Cont(\underline{p},c'',c) \wedge c' \neq c \wedge c'' \neq c$

In some cases, these rules must be syntactically rewritten to ensure they are allowed. This minor transformation consist to introduce auxiliary predicates like in the following example:

$Emp^n_{1,6}(\underline{p},c) \leftarrow \neg \textit{Works(p,y)} \wedge \iota Works(\underline{p},c) \wedge Cont(\underline{p},c') \wedge \mu Cont(\underline{p},c',c) \wedge c' \neq c$

$Emp^n_{1,6}(\underline{p},c) \leftarrow \neg Aux_1(\underline{p}) \wedge \iota Works(\underline{p},c) \wedge Cont(\underline{p},c') \wedge \mu Cont(\underline{p},c',c) \wedge c' \neq c$

$Aux_1(\underline{p}) \leftarrow Works(\underline{p},c)$

[4] These expressions are equivalent to those defined in [UO92].

3.3 Event rules

The second group of rules of the Augmented Database A(D) are the *Event rules*. These rules define the derived and inconsistency events that are induced during the transition in terms of the extension of the old state and the events that occur in the transition.

Definition of events $\iota P(\underline{\mathbf{k}}, \mathbf{x})$, $\delta P(\underline{\mathbf{k}}, \mathbf{x})$ and $\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x}')$ depends on definition of predicate $P(\underline{\mathbf{k}}, \mathbf{x})$ in D but it is independent of any transaction T and of the extension of D. Therefore, for each derived predicate $P(\underline{\mathbf{k}}, \mathbf{x})$ we define the *insertion event rules* $\iota P(\underline{\mathbf{k}}, \mathbf{x})$, the *deletion event rules* $\delta P(\underline{\mathbf{k}}, \mathbf{x})$ and the *modification event rules* $\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x}')$, which exactly state the insertions, deletions and modifications of facts $P(\underline{\mathbf{k}}, \mathbf{x})$ that are induced during the transition.

3.3.1 Insertion Event Rules

Given a derived or inconsistency predicate $P(\underline{\mathbf{k}}, \mathbf{x})$ defined by m≥1 derivation rules, and considering the definition of the insertion event predicate $\iota P(\underline{\mathbf{k}}, \mathbf{x})$, we can state that:

$$\iota P(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow P^n_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \neg\exists\, \mathbf{y}\, P_1(\underline{\mathbf{k}}, \mathbf{y}) \wedge \ldots \wedge \neg\exists\, \mathbf{y}\, P_i(\underline{\mathbf{k}}, \mathbf{y}) \wedge \ldots \wedge \neg\exists\, \mathbf{y}\, P_m(\underline{\mathbf{k}}, \mathbf{y}) \qquad i = 1 \ldots m$$

Notice that the conjunction $P^n_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \neg\exists \mathbf{y} P_i(\underline{\mathbf{k}}, \mathbf{y})$ corresponds to the definition of the insertion event of predicate $P_i(\underline{\mathbf{k}}, \mathbf{x})$. Therefore, we obtain this set of rules:

$$\iota P(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow \iota P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \neg\exists \mathbf{y} P_1(\underline{\mathbf{k}}, \mathbf{y}) \wedge \ldots \wedge \neg\exists \mathbf{y} P_{i-1}(\underline{\mathbf{k}}, \mathbf{y}) \wedge \neg\exists \mathbf{y} P_i(\underline{\mathbf{k}}, \mathbf{y})^5 \wedge \neg\exists \mathbf{y} P_{i+1}(\underline{\mathbf{k}}, \mathbf{y}) \wedge \ldots \wedge \neg\exists \mathbf{y} P_m(\underline{\mathbf{k}}, \mathbf{y}) \qquad i = 1 \ldots m$$

$$\iota P_i(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow P^n_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \neg\exists\, \mathbf{y}\, P_i(\underline{\mathbf{k}}, \mathbf{y}) \qquad i = 1 \ldots m$$

This set of rules correspond to the *insertion event rules* of predicates $P(\underline{\mathbf{k}}, \mathbf{x})$ and $P_i(\underline{\mathbf{k}}, \mathbf{x})$, respectively.

Notice that, when the predicate $P(\underline{\mathbf{k}}, \mathbf{x})$ corresponds to the inconsistency predicate Ic, the insertion event rules ιIc specify all possible ways to induce a violation of any integrity constraint. These rules play an important role in our method since they will be used to maintain integrity constraints satisfied.

Example 3.3: Consider the derived predicate Emp(p,c) of Example 2.1. The insertion event rules of this predicate are the following:

$$\iota Emp(\underline{p},c) \leftarrow \iota Emp_1(\underline{p},c) \wedge \neg Emp_1(\underline{p},c)$$

$$\iota Emp_1(\underline{p},c) \leftarrow Emp^n_1(\underline{p},c) \wedge \neg Emp_1(\underline{p},c)$$

3.3.2 Deletion Event Rules

Given a derived predicate $P(\underline{\mathbf{k}}, \mathbf{x})$ defined by m≥1 derivation rules, and considering the definition of the deletion event predicate $\delta P(\underline{\mathbf{k}}, \mathbf{x})$, we can state that:

$$\delta P(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \neg\exists\, \mathbf{y}\, P^n_1(\underline{\mathbf{k}}, \mathbf{y}) \wedge \ldots \wedge \neg\exists\, \mathbf{y}\, P^n_i(\underline{\mathbf{k}}, \mathbf{y}) \wedge \ldots \wedge \neg\exists \mathbf{y} P^n_m(\underline{\mathbf{k}},\mathbf{y}) \qquad i = 1 \ldots m$$

Notice that conjunction $P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \neg\exists \mathbf{y} P^n_i(\underline{\mathbf{k}}, \mathbf{y})$ corresponds to the definition of the deletion event of predicate $P_i(\underline{\mathbf{k}}, \mathbf{x})$, and substituting literal $\neg\exists \mathbf{y} P^n_j(\underline{\mathbf{k}}, \mathbf{y})$ by equivalence (2) we obtain this set of rules:

$$\delta P(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow \wedge^m_{i=1} [P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \delta P_i(\underline{\mathbf{k}}, \mathbf{x}) \mid \neg\exists\, \mathbf{y}\, P_i(\underline{\mathbf{k}}, \mathbf{y}) \wedge \neg\, \iota P_i(\underline{\mathbf{k}}, \mathbf{z})]^6$$

$$\delta P_i(\underline{\mathbf{k}}, \mathbf{x}) \leftarrow P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \neg\exists\, \mathbf{y}\, P^n_i(\underline{\mathbf{k}}, \mathbf{y}) \qquad i = 1 \ldots m$$

---

[5] This literal is redundant, but for technical reasons we maintain it in the body of the insertion event rule.

[6] Notice that this definition generates 2^m-1 deletion event rules

This set of rules correspond to the *deletion event rules* of predicates $P(\underline{\mathbf{k}}, \mathbf{x})$ and $P_i(\underline{\mathbf{k}}, \mathbf{x})$, respectively.

Notice that for inconsistency predicates deletion event rules are not defined, since we assume that the old state is consistent (predicate Ic evaluates false).

Example 3.4: The deletion event rules of derived predicate Emp(p,c) of Example 2.1 are the following:

$$\delta Emp(\underline{p},c) \leftarrow Emp_1(\underline{p},c) \wedge \delta Emp_1(\underline{p},c)$$
$$\delta Emp_1(\underline{p},c) \leftarrow Emp_1(\underline{p},c) \wedge \neg Emp^n{}_1(\underline{p},c')$$

3.3.3 Modification Event Rules

Given a derived predicate $P(\underline{\mathbf{k}}, \mathbf{x})$ defined by m≥1 derivation rules, and considering the definition of the modification event predicate $\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'})$, we can state that:

$$\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \leftarrow P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge P^n{}_h(\underline{\mathbf{k}}, \mathbf{x'}) \wedge \mathbf{x} \neq \mathbf{x'} \qquad i, h = 1 \ldots m$$

Notice that conjunction $P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge P^n{}_i(\underline{\mathbf{k}}, \mathbf{x'}) \wedge \mathbf{x} \neq \mathbf{x'}$ corresponds to the definition of the modification event of predicate $P_i(\underline{\mathbf{k}}, \mathbf{x})$. By assuming key integrity constraints satisfied and substituting literals $P^n{}_j(\underline{\mathbf{k}}, \mathbf{y})$ by equivalence (1), we obtain this set of rules[7]:

$$\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \leftarrow \wedge^m{}_{i=1} [(P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \delta P_i(\underline{\mathbf{k}}, \mathbf{x})) \mid (P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \mu P_i(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \wedge \mathbf{x} \neq \mathbf{x'})]$$
$$\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \leftarrow \wedge^m{}_{i=1} [(P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \delta P_i(\underline{\mathbf{k}}, \mathbf{x})) \mid (\neg \exists \mathbf{y}\ P_i(\underline{\mathbf{k}}, \mathbf{y}) \wedge \iota P_i(\underline{\mathbf{k}}, \mathbf{x'}) \wedge \mathbf{x} \neq \mathbf{x'}))]$$
$$\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \leftarrow \wedge^m{}_{i=1} [(P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \mu P_i(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \wedge \mathbf{x} \neq \mathbf{x'}) \mid (\neg P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \neg \iota P_i(\underline{\mathbf{k}}, \mathbf{z}))]$$
$$\mu P(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \leftarrow \wedge^m{}_{i=1} [(P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge \mu P_i(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \wedge \mathbf{x'} \neq \mathbf{x})]$$
$$\mu P_i(\underline{\mathbf{k}}, \mathbf{x}, \mathbf{x'}) \leftarrow P_i(\underline{\mathbf{k}}, \mathbf{x}) \wedge P^n{}_i(\underline{\mathbf{k}}, \mathbf{x'}) \wedge \mathbf{x} \neq \mathbf{x'} \qquad i = 1 \ldots m$$

These rules correspond to the *modification event rules* of predicates $P(\underline{\mathbf{k}}, \mathbf{x})$ and $P_i(\underline{\mathbf{k}}, \mathbf{x})$.

Notice that for inconsistency predicates modification event rules are not defined for the same reason that neither deletion event rules are defined.

Example 3.5: The modification event rules of derived predicate Emp(p,c) of Example 2.1 are the following:

$$\mu Emp(\underline{p},c,c') \leftarrow Emp_1(\underline{p},c) \wedge \mu Emp_1(\underline{p},c,c') \wedge c' \neq c$$
$$\mu Emp_1(\underline{p},c,c') \leftarrow Emp_1(\underline{p},c) \wedge Emp^n{}_1(\underline{p},c') \wedge c' \neq c$$

3.4 Augmented Database A(D)

The Augmented Database A(D) is an extension of the original database D with the transition rules and the event rules associated to derived and inconsistency predicates.

Definition 3.6: Given a deductive database D, the *Augmented Database A(D)* of D consists of D, its transition rules and its event rules.

Description of the basic procedure for automatically deriving and simplifying an Augmented Database, and syntactical properties of A(D) can be found in [UO92].

Example 3.6: Given the database D of Example 2.1, this is the Augmented Database A(D) after applying the simplifications proposed in [UO92].[8]

[7] Notice that 1st, 2nd and 3rd definitions generate 2^m-2 modification event rules, while the fourth only one.
[8] In this example transition rules are not necessary and could be eliminated.

(F1)[9] Works(Mercè, UPC)

(R1) $Emp(p,c) \leftarrow Works(p,c) \wedge Cont(p,c)$

(R2) $Contracted(p) \leftarrow Cont(p,c)$

(R3) $Ic1(p,n) \leftarrow IdNum(p,n) \wedge \neg Contracted(p)$

(R4) $Ic2(p,c) \leftarrow Emp(p,c) \wedge \neg Lab\_age(p)$

(R01) $Ic \leftarrow Ic1(p,n)$

(R02) $Ic \leftarrow Ic2(p,c)$

(I1) $\iota Emp(p,c) \leftarrow Works(p,c) \wedge \neg\delta Works(p,c) \wedge \neg Aux1(p,c) \wedge \neg Aux5(p) \wedge \iota Cont(p,c)$

(I2) $\iota Emp(p,c) \leftarrow Works(p,c) \wedge \neg\delta Works(p,c) \wedge \neg Aux1(p,c) \wedge Cont(p,c1) \wedge \mu Cont(p,c1,c) \wedge c1 \neq c$

(I3) $\iota Emp(p,c) \leftarrow \neg Aux6(p) \wedge \iota Works(p,c) \wedge Cont(p,c) \wedge \neg\delta Cont(p,c) \wedge \neg Aux2(p,c)$

(I4) $\iota Emp(p,c) \leftarrow \neg Aux6(p) \wedge \iota Works(p,c) \wedge \neg Aux5(p) \wedge \iota Cont(p,c)$

(I5) $\iota Emp(p,c) \leftarrow \neg Aux6(p) \wedge \iota Works(p,c) \wedge Cont(p,c1) \wedge \mu Cont(p,c1,c) \wedge c1 \neq c$

(I6) $\iota Emp(p,c) \leftarrow Works(p,c1) \wedge \mu Works(p,c1,c) \wedge c1 \neq c \wedge Cont(p,c) \wedge \neg\delta Cont(p,c) \wedge \neg Aux2(p,c)$

(I7) $\iota Emp(p,c) \leftarrow Works(p,c1) \wedge \mu Works(p,c1,c) \wedge c1 \neq c \wedge \neg Aux5(p) \wedge \iota Cont(p,c)$

(I8) $\iota Emp(p,c) \leftarrow Works(p,c1) \wedge \mu Works(p,c1,c) \wedge c1 \neq c \wedge Cont(p,c2) \wedge \mu Cont(p,c2,c) \wedge c2 \neq c \wedge c1 \neq c2$

(D1) $\delta Emp(p,c) \leftarrow Works(p,c) \wedge \delta Works(p,c) \wedge Cont(p,c)$

(D2) $\delta Emp(p,c) \leftarrow Works(p,c) \wedge \mu Works(p,c,c1) \wedge c1 \neq c \wedge Cont(p,c) \wedge \neg\mu Cont(p,c,c1)$

(D3) $\delta Emp(p,c) \leftarrow Works(p,c) \wedge Cont(p,c) \wedge \delta Cont(p,c)$

(D4) $\delta Emp(p,c) \leftarrow Works(p,c) \wedge Cont(p,c) \wedge \mu Cont(p,c,c1) \wedge c1 \neq c \wedge \neg\mu Works(p,c,c1)$

(M1) $\mu Emp(p,c1,c2) \leftarrow Works(p,c1) \wedge \mu Works(p,c1,c2) \wedge Cont(p,c1) \wedge \mu Cont(p,c1,c2) \wedge c1 \neq c2$

(I9) $\iota Contracted(p) \leftarrow \neg Aux5(p) \wedge \iota Cont(p,c)$

(D5) $\delta Contracted(p) \leftarrow Cont(p,c) \wedge \delta Cont(p,c)$

(C01) $\iota Ic \leftarrow \iota Ic1(p,n)$

(C02) $\iota Ic \leftarrow \iota Ic2(p,c)$

(C1) $\iota Ic1(p,n) \leftarrow IdNum(p,n) \wedge \neg\delta IdNum(p,n) \wedge \neg Aux3(p,n) \wedge Contracted(p) \wedge \delta Contracted(p)$

(C2) $\iota Ic1(p,n) \leftarrow \neg Aux7(p) \wedge \iota IdNum(p,n) \wedge \neg Contracted(p) \wedge \neg\iota Contracted(p)$

(C3) $\iota Ic1(p,n) \leftarrow \neg Aux7(p) \wedge \iota IdNum(p,n) \wedge Contracted(p) \wedge \delta Contracted(p)$

(C4) $\iota Ic1(p,n) \leftarrow IdNum(p,n) \wedge \mu IdNum(p,n,n1) \wedge n1 \neq n \wedge Contracted(p) \wedge \delta Contracted(p)$

(C5) $\iota Ic2(p,c) \leftarrow Emp(p,c) \wedge \neg\delta Emp(p,c) \wedge \neg Aux4(p,c) \wedge Lab\_age(p) \wedge \delta Lab\_age(p)$

(C6) $\iota Ic2(p,c) \leftarrow \neg Aux8(p) \wedge \iota Emp(p,c) \wedge \neg Lab\_age(p) \wedge \neg\iota Lab\_age(p)$

(C7) $\iota Ic2(p,c) \leftarrow \neg Aux8(p) \wedge \iota Emp(p,c) \wedge Lab\_age(p) \wedge \delta Lab\_age(p)$

(C8) $\iota Ic2(p,c) \leftarrow Emp(p,c) \wedge \mu Emp(p,c,c1) \wedge c1 \neq c \wedge Lab\_age(p) \wedge \delta Lab\_age(p)$

(A1) $Aux1(p,c) \leftarrow \mu Works(p,c,c1)$

(A2) $Aux2(p,c) \leftarrow \mu Cont(p,c,c1)$

(A3) $Aux3(p,n) \leftarrow \mu IdNum(p,n,n1)$

(A4) $Aux4(p,c) \leftarrow \mu Emp(p,c,c1)$

(A5) $Aux5(p) \leftarrow Cont(p,c)$

(A6) $Aux6(p) \leftarrow Works(p,c)$

(A7) $Aux7(p) \leftarrow IdNum(p,n)$

(A8) $Aux8(p) \leftarrow Emp(p,c)$

[9] Rules and facts of the A(D) are identified by a label (between parenthesis) to refer them more easily: Fact (F), derivation Rule (R), Insertion event rule (I), Deletion event rule (D), Modification event rule (M), insertion of event rule of an inconsistency predicate (C), derivation rule of an Auxiliary predicate (A).

4. Definition of our Method

The purpose of our method is to update a deductive database and to enforce, at the same time, that all integrity constraints remain satisfied.

The method proposed in this paper extends the Events Method [TO95] in two different directions. At definition level, we introduce the modification update as a new basic update operator, in addition to the insertion and deletion updates already considered in [TO95]. Moreover, key integrity constraints are enforced by the own definition of the method. The second extension relies in the introduction of some techniques to improve efficiency of the method. Issues on the first direction are considered in this section, meanwhile efficiency issues are considered in Section 6. A preliminary definition of our method has been presented in [MT00].

Given an update request that may contain base and/or derived updates, our method automatically translates it into *all* possible transactions such that, when applied to the database, they satisfy the requested update and guarantee that the integrity constraints remain satisfied.

Definition 4.1: An *update request* u is a conjunction of positive and/or negative base and/or derived event facts[10]. Positive event facts correspond to updates to perform, while negative events correspond to updates that must be prevented.

Example 4.1: Consider the database of Example 2.1 and assume an update request consisting in the modification of Peter as employee of the company UAB to the UPC, and the prevention to delete that Peter is a contracted person. This request corresponds to the following conjunction $u = \mu Emp(Peter, UAB, UPC) \wedge \neg \delta Contracted(Peter)$.

Definition 4.2: A *transaction* T is a set of non-mutually exclusive positive base event facts whose database requirements are satisfied.

Definition 4.3: Given a deductive database D, its Augmented Database A(D) and an update request u, a transaction T_i is a *solution of* u if:

$$A(D) \cup T_i \models u$$

$$A(D) \cup T_i \not\models \iota Ic$$

The first condition states that the update request is a logical consequence of the database updated according to T_i, while the second condition states that no integrity constraint will be violated in the updated database since no insertion of Ic could be induced by T_i. Note that, since we assume that no integrity constraint is violated in D, ensuring that ιIc is not induced is enough to guarantee that all integrity constraints do not become violated in the updated database. Therefore, to ensure both conditions it is sufficient to consider the extended update request $u \wedge \neg \iota Ic$.

Definition 4.4: A solution T_i is a *minimal solution* of u if no proper subset of T_i is also a solution of u.

Example 4.2: Consider the update request u of Example 4.1, the database of Example 2.1 and this extensional database: Works(Peter, UAB), Cont(Peter, UAB), Lab_age(Peter). One

[10] The order of the literals in the conjunction is not relevant since all of them must be satisfied.

solution to the update request u is T = {μWorks(Peter, UAB, UPC), μCont(Peter, UAB, UPC), ιIdNum(Peter, 4562)} since it satisfies the update request and does not violate any integrity constraint. However, notice that this solution is not minimal because the subset T'= T- {ιIdNum(Peter, 4562)} is also a solution.

Given a deductive database D where no integrity constraint is violated, its Augmented Database A(D) and an update request u, our method is aimed at obtaining all minimal solutions T_i of u. Each T_i is obtained by having some failed SLDNF derivation of A(D) ∪ {← u ∧ ¬ιIc } succeed. This is achieved by including in the set T_i each positive base event fact selected during the failed derivation. At the end, we have that there is an SLDNF refutation of ← u ∧ ¬ιIc by considering A(D) ∪ T_i as input set.

Different ways to make failed derivations succeed correspond to the different solutions T_i of u. If no solution is obtained, it is not possible to satisfy the update request by changing only the extensional database.

4.1 Example

In this section, we use an example to illustrate the steps performed by our method to obtain all solutions. We describe only the most relevant branches of the SLDNF refutations and finitely failed trees. Each step is identified by a number, the selected literal is indicated in bold style, the rule of the method we apply is labeled inside a circle, and rules of the A(D) taken into account are indicated in between parenthesis.

In this example, we consider the Augmented Database A(D) shown in Example 3.6 and the update request u = ιIdNum(Mercè,12345) ∧ ¬ιIc. Transactions that satisfy the update request are obtained by having failed derivations of A(D) ∪ {← ιIdNum(Mercè,12345) ∧ ¬ιIc} succeed. Part of this derivation is shown in Figure 4.1.

Notice that in this example, base event ιIdNum(Mercè,12345) induces a violation of integrity constraint Ic1 because Mercè is not contracted. To repair it, it is necessary to induce the insertion of the derived fact Contracted(Mercè). Therefore, we propose Mercè to be contracted by some company, by means of a view updating process. Notice also that, this repair could induce a violation of integrity constraint Ic2 since the derived fact Emp(Mercè,UPC) could be induced by the repair. In this case, there are three alternative ways to repair this violation.

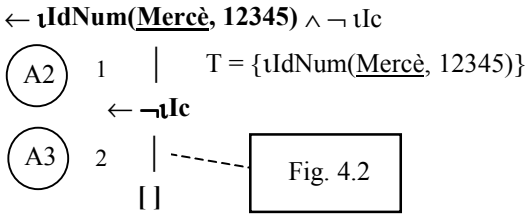


Fig.4.1 Main refutation of Example 4.1

At step 1, the selected literal is a positive base event ιIdNum(Mercè, 12345). Therefore, to get a successful derivation, we must include it in the input set and use it as a new input clause. Therefore, the base event fact is added to the transaction T.

At step 2, the selected literal is ¬ιIc. To get a success for this branch, literal ιIc must not hold. Then, we check that the subsidiary tree rooted by goal ← ιIc fails finitely by means of a subsidiary derivation (shown in Figure 4.2). This check corresponds to guarantee that the current transaction T={ιIdNum(Mercè, 12345)} does not violates any integrity constraint, if this is the case, new updates are proposed to repair this violation.

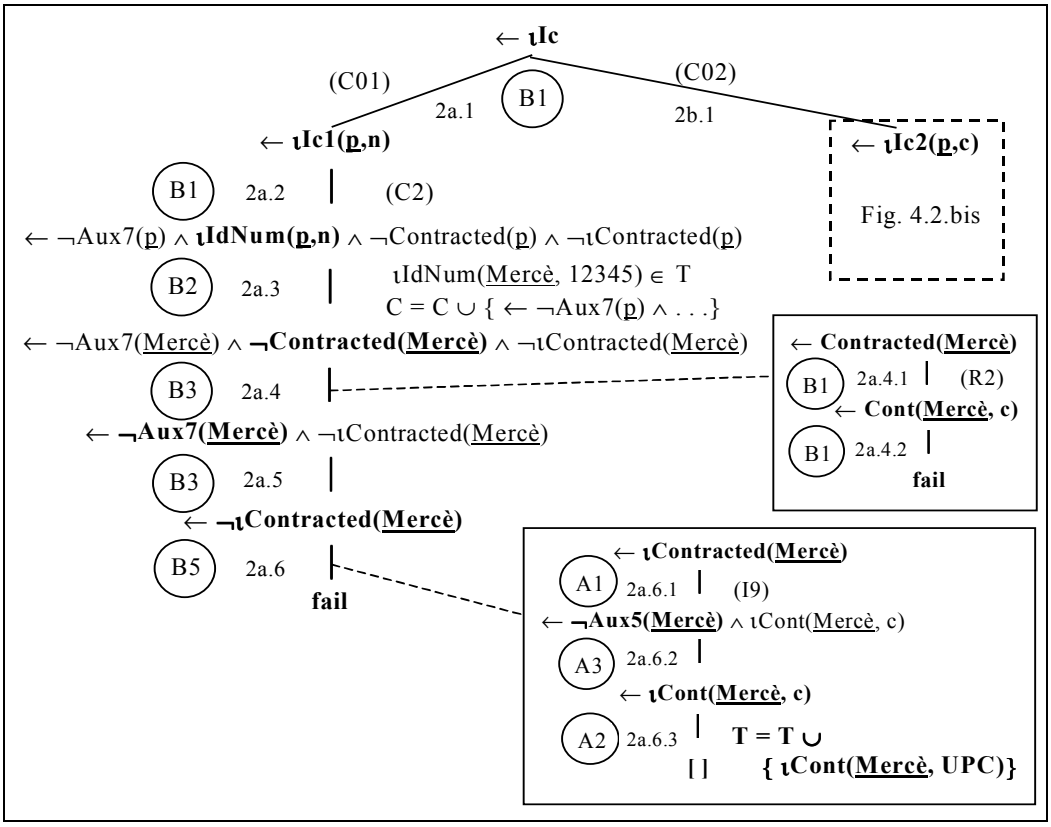


Fig.4.2 Subsidiary tree for step 2

Steps 2a.1, 2a.2 and 2b.1 are SLDNF resolution steps where A(D) acts as input set. At step 2a.2, there are three alternative branches corresponding to rules C1, C3 and C4 of the A(D). All of them fail finitely and they are not shown in the figure.

Step 2a.3 is an SLDNF resolution step where T acts as input set. Notice that the current goal is also included into a set C to ensure that it will remain falsified during the whole derivation process. The auxiliary set C contains conditions that must be false during the whole derivation process. These conditions correspond to some of the goals reached in some previous subsidiary derivation. Hence, before adding any base event to transaction T, we must guarantee that this event does not satisfy any of the conditions of set C.

Steps 2a.4 and 2a.5 correspond to SLDNF resolution steps with the input set A(D).

To get a failure in the step 2a.6, we force to hold the derived event fact ιContracted(Mercè). Notice that it corresponds to repairing the integrity constraint Ic1 by means of a view updating process. This is achieved by means of a subsidiary derivation. Step

2a.6.1 and 2a.6.2 correspond to an SLDNF resolution step where the A(D) acts as input set. In the last step (2a.6.3), to reach the empty goal, we include the base event fact ιCont(Mercè,UPC)[11] into set T. Previously, we have checked database requirements of this event and that conditions of set C remain falsified with the new inclusion into set T.

After this step, the left branch of the subsidiary derivation rooted by goal ← ιIc fails and integrity constraint Ic1 has been repaired.

The right branch of the derivation tree is shown in Figure 4.2.bis. In this branch, we show how the current transaction T (after repairing Ic1) violates integrity constraint Ic2, and how we repair it.

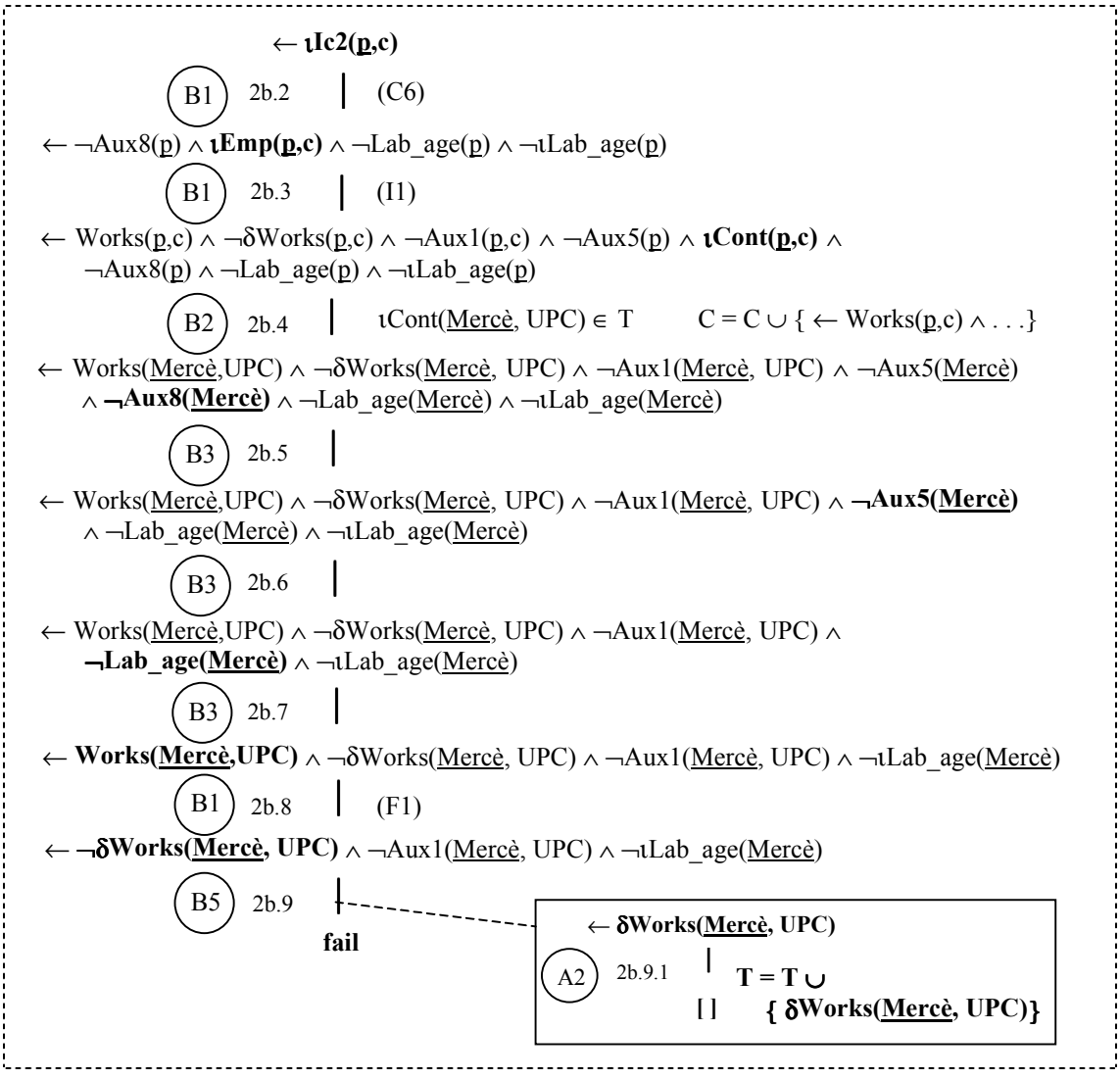


Fig.4.2bis Right branch of subsidiary tree of Figure 4.2

Steps 2b.2, 2b.3 and 2b.4 are SLDNF resolution steps. In steps 2b.2 and 2b.3 the A(D) acts as the input set while, in step 2b.4, the input set corresponds to set T. Notice also, that in this step, similarly to step 2a.3 of Figure 4.2, the current goal is included into set C to prevent

[11] To obtain all possible solutions to the update request ιContracted(Mercè) it would be necessary to consider all possible values of the domain of variable c.

satisfying it with further inclusions of event facts ιCont(p, c) into set T. Alternative branches to steps 2b.2 and 2b.3 are not shown in the figure since they also fail finitely.

Steps 2b.5, 2b.6, 2b.7 and 2b.8 correspond also to SLDNF resolution steps with the input set A(D). Some of them require a subsidiary derivation not shown in the figure.

In the last step (2b.9), to force the failure of the goal, we include the base event fact δWorks(Mercè, UPC) into the input set T. This inclusion is performed in the subsidiary derivation shown inside the box. In step 2b.9.1, the selected literal is a positive base event fact that is included into set T since its database requirements are satisfied and conditions of set C remain falsified. Subsidiary derivations to perform these checks are not shown in the figure.

Finally, all branches to the subsidiary derivation (of Figure 4.2) rooted by goal ← ιIc fails finitely, therefore the initial derivation of Figure 4.1 reaches the empty clause and a solution is obtained. The transaction T={ιIdNum(Mercè, 12345), ιCont(Mercè, UPC), δWorks(Mercè, UPC)} is a minimal solution of the initial update request u = ιIdNum(Mercè,12345) ∧ ¬ιIc since it satisfies the request, it does not violate any integrity constraint and no proper subset of T is also a solution.

Two additional alternatives exist to force the failure of the derivation of Figure 4.2bis, by selecting any of the remaining literals at step 2b.9. These solutions are: T_1={ιIdNum(Mercè, 12345), ιCont(Mercè, UPC), μWorks(Mercè, UPC, UB[12])} and T_2={ιIdNum(Mercè, 12345), ιCont(Mercè, UPC), ιLab_age(Mercè)}. Moreover, there are other solutions to the update request u. They are obtained by considering at step 2a.b.3 a value c≠UPC, such that integrity constraint Ic2 does not become violated. In this case, we obtain a solution like T_3={ιIdNum(Mercè, 12345), ιCont(Mercè, UB)} where UB≠UPC.

4.2 Formalization of our Method

As shown in the previous example, our method consists in the interleaving of two activities. On one side, to satisfy the update request by including base event facts into the set T. On the other, to ensure that the updates induced by events of T are not contradictory with the requested update u, nor with the integrity constraints. These two activities are performed, respectively, during the *Constructive Derivation* and the *Consistency Derivation*.

Let u be an update request. A transaction T is a solution of u if there is a *Constructive Derivation* from (← u∧¬ιIc ∅ ∅) to ([] T C). Positive base events selected during this derivation are included in T since they correspond to the updates needed to satisfy u and to do not violate any integrity constraint. The rest of positive literals correspond to database queries or evaluable predicates. Consistency of negative literals $\neg L_j$ selected during a constructive derivation is verified by considering a subsidiary *Consistency Derivation* from ({←L_j} T C) to ({} T' C').

To define these derivations, we need to introduce the following conventions:

- The transaction T contains the base event facts that are necessary to reach the empty goal, that is, to success a failed SLDNF derivation of A(D) ∪ {← u ∧ ¬ιIc }.

[12] UB is an arbitrary value of the domain.

- The set C contains the set of conditions (goals) that must remain false during all the derivation process to ensure that, events of set T really satisfy the update request u and do not violate any integrity constraint.
- Given a goal G_i of the form $\leftarrow L_1 \wedge ... \wedge L_k$, the expression $G_i \backslash L_j$ refers to the goal obtained by removing the literal L_j from G_i. Notice that if $G_i = \leftarrow L_j$ then $G_i \backslash L_j = []$.
- In a consistency derivation, $F_i = \{H_i\} \cup F_i'$ refers to the set of goals to be falsified. Goal H_i corresponds to the goal of the current branch of the derivation.

Constructive Derivation

A *Constructive Derivation* from $(G_1\ T_1\ C_1)$ to $(G_n\ T_n\ C_n)$ via a safe selection rule R is a sequence: $(G_1\ T_1\ C_1)$, …, $(G_i\ T_i\ C_i)$, …, $(G_n\ T_n\ C_n)$, such that for each $i \geq 1$, G_i has the form $\leftarrow L_1 \wedge \ldots \wedge L_k$ where $R(G_i) = L_j$ and $(G_{i+1}\ T_{i+1}\ C_{i+1})$ is obtained according to one of the following rules:

A1) If L_j is positive and it is not a base event, then $G_{i+1} = S$, $T_{i+1} = T_i$ and $C_{i+1} = C_i$, where S corresponds to: if L_j is a ground evaluable predicate that evaluates to true, then $S = G_i \backslash L_j$; if it is not an evaluable predicate, then S is the resolvent of some clause in A(D) with G_i on the selected literal L_j.

A2) If L_j is a positive base event and there is a substitution σ[13] such that

A21) $L_j\sigma \in T_i$, then $G_{i+1} = G_i \backslash L_j\sigma$, $T_{i+1} = T_i$ and $C_{i+1} = C_i$.

A22) $L_j\sigma \notin T_i$ and

a) $L_j = \iota P(\underline{k}, x)$ and $\neg\, \exists y\, \iota P(\underline{k\sigma}, y) \in T_i$ and $\neg\, \exists z$ such that $P(\underline{k\sigma}, z)$ holds in D

b) $L_j = \delta P(\underline{k}, x)$ and $\neg\, \exists x'\, \mu P(\underline{k\sigma}, x\sigma, x') \in T_i$ and fact $P(\underline{k}, x)\sigma$ holds in D

c) $L_j = \mu P(\underline{k}, x, x')$, $\delta P(\underline{k}, x)\sigma \notin T_i$, $\neg\, \exists y'\, \mu P(\underline{k\sigma}, x\sigma, y') \in T_i$, $(x \neq x')\sigma$ and $P(\underline{k},x)\sigma$ holds in D.

If $C_i = \{\leftarrow Q_1, \ldots, \leftarrow Q_k, \ldots, \leftarrow Q_n\}$ and there are consistency derivations

from $(\{\leftarrow Q_1\}\ \ T_i \cup \{L_j\sigma\}\ \ C_i)$ to $(\{\}\ \ T^1\ \ C^1)$, ...,

from $(\{\leftarrow Q_n\}\ \ T^{n-1}\ \ C^{n-1})$ to $(\{\}\ \ T^n\ \ C^n)$,

then $G_{i+1} = G_i \backslash L_j\sigma$, $T_{i+1} = T^n$ and $C_{i+1} = C^n$.

Notice that if $C_i = \emptyset$ then $G_{i+1} = G_i \backslash L_j\sigma$, $T_{i+1} = T_i \cup \{L_j\sigma\}$ and $C_{i+1} = C_i$.

A3) If L_j is a negative literal and there is a consistency derivation from $(\{\leftarrow \neg L_j\}\ T_i\ C_i)$ to $(\{\}\ T'\ C')$, then $G_{i+1} = G_i \backslash L_j$, $T_{i+1} = T'$ and $C_{i+1} = C'$.

Step A1) is an SLDNF resolution step where A(D) acts as input set.

Step A2) deals with base event literals. In particular, step A21) corresponds to an SLDNF resolution step with the input set T_i. In A22) base event facts are included in T_i when this inclusion does not contradict event exclusiveness, when it satisfies event definition and no condition of set C_i becomes satisfied. Checking falseness of conditions may cause, in some cases, new inclusions into T_i. If the selected base event is not ground, it must be instanced by

[13] Notice that if literal L_j is ground, substitution σ corresponds to the identity substitution.

considering all possible values. In general, there are as many alternatives as possible ways to ground the selected event.

Step A3) ensures the consistency of the selected literal by considering a corresponding consistency derivation for that literal. Again, new events can be included in T_i as result of this subsidiary derivation.

Consistency Derivation

A *Consistency Derivation* from (F_1 T_1 C_1) to (F_n T_n C_n) via a safe selection rule R is a sequence (F_1 T_1 C_1), …, (F_i T_i C_i), …, (F_n T_n C_n), such that for each i≥1, if $H_i = \leftarrow L_1 \wedge \ldots \wedge L_k$ and if F_i has the form $\{H_i\} \cup F'_i$ where $R(H_i) = L_j$ *for some* j=1…k, then (F_{i+1} T_{i+1} C_{i+1}) is obtained according to one of the following rules:

B1) If L_j is positive and it is not a base event, then $F_{i+1}= S'\cup F'_i$, $T_{i+1}= T_i$ and $C_{i+1}= C_i$, where S' corresponds to: if L_j is an evaluable predicate that evaluates to true and k>1, then S'= $\{H_i \backslash L_j\}$, but if it evaluates to false, then S' = Ø; if L_j is not evaluable, then S' corresponds to the set of all resolvents of clauses in A(D) with H_i on the selected literal L_j whenever $[] \notin S'$.

B2) If L_j is a positive base event, S' corresponds to the set of all resolvents of clauses in T_i with H_i on the selected literal L_j. If $[] \notin S'$, then $F_{i+1} = S'\cup F'_i$, $T_{i+1} = T_i$.

If S'=Ø or L_j is not ground then $C_{i+1} = C_i \cup \{H_i\}$, otherwise $C_{i+1} = C_i$.

B3) If L_j is a negative literal and $\neg L_j$ is not a base event, then if k>1 and there is a consistency derivation from ($\{\leftarrow \neg L_j\}$ T_i C_i) to ({} T' C'), then $F_{i+1} = \{H_i \backslash L_j\} \cup F'_i$, $T_{i+1} =$ T' and $C_{i+1}=$ C'.

B4) If L_j is a negative base event and if $\neg L_j \notin T_i$ and k >1, then $F_{i+1} = \{H_i \backslash L_j\} \cup F'_i$, $T_{i+1} = T_i$ and $C_{i+1}= C_i$.

B5) If L_j is a negative literal, then if there is a constructive derivation from ($\leftarrow \neg L_j$ T_i C_i) to ([] T' C'), then $F_{i+1}=F'_i$, $T_{i+1}=$ T' and $C_{i+1}=$ C'.

Step B1) is an SLDNF resolution step where A(D) acts as input set.

Step B2) corresponds to an SLDNF resolution step with the input set T_i, but it may require the addition of the current goal to C_i if the selected literal is a non-ground base event or if it does not belong to T_i. This is required to ensure that this derivation will not be succeeded by further inclusions into T_i.

Steps B3) and B4) go on with the current branch by ensuring that the selected literal L_j is consistent with respect to T_i and C_i.

Step B5) falsifies the current branch by satisfying the literal $\neg L_j$ through a constructive derivation.

Consistency derivations do not depend on the particular order in which literals are selected because, in general, it is necessary to explore all possible ways to falsify a goal H_i since each of them could lead to a different solution.

4.3 Soundness and Completeness of our Method

The proposed method is sound in the sense that, given an update request u, a deductive database D and the augmented database A(D), the method obtains a transaction T that applied to D leaves the new database state D^n such that u holds and integrity constraints are satisfied.

Moreover, the proposed method is also complete. Given a deductive base D, for each possible way to satisfy (in a state D^n) the update request u and the integrity constraints, the proposed method obtains a transaction T_i.

Proofs of both properties assume that SLDNF resolution is sound and complete for stratified databases[14] [Cla78].

4.3.1 Soundness of our Method

Soundness of the proposed method is based on the following Lemma:

Lemma 1: Let D be a deductive database, A(D) the Augmented Database, u an update request and T a solution such that a constructive derivation from (← u ∧ ¬ιIc ∅ ∅) to ([] T C) exists. Then, there is an SLDNF refutation of A(D) ∪ T ∪ {← u ∧ ¬ιIc}.

Lemma 1 relates the constructive derivation from (←u ∧ ¬ιIc ∅ ∅) to ([] T C) of our method to an SLDNF refutation of A(D) ∪ T ∪ {← u ∧ ¬ιIc}. Given that SLDNF resolution has been proved sound for stratified databases [Cla78], then the following theorem follows:

Theorem 1: *(Soundness of our method)*

Let D be a deductive database, A(D) the Augmented Database and u an update request, such that u is not a logical consequence of comp(A(D)). Let T be a solution obtained by our method. Then, u∧¬ιIc is a logical consequence of comp(A(D) ∪ T).

Soundness of our method ensures that if there is a constructive derivation from (← u ∧ ¬ιIc ∅ ∅) to ([] T C), then the database updated according to the transaction T satisfies the update request u and does not violate any integrity constraint. The technical proof is presented in Appendix A.

4.3.2 Completeness of our Method

Completeness of the proposed method is based on completeness of SLDNF for stratified databases and the following Theorem:

Theorem 2: Let D be a deductive database; A(D) the Augmented Database; u an update request and T a minimal solution. If there is an SLDNF refutation of A(D) ∪ T ∪ {←u∧¬ιIc } then, a constructive derivation from (←u∧¬ιIc ∅ ∅) to ([] T C) exists.

Theorem 2 relates an SLDNF refutation of A(D) ∪ T ∪ {← u∧¬ιIc } to the constructive derivation from (←u∧¬ιIc ∅ ∅) to ([] T C) of our method. If we assume that SLDNF resolution is complete, then the following theorem follows:

Theorem 3: (*Completeness of our method*)

Let D be a deductive database; A(D) the Augmented Database; u an update request and T a minimal solution. Since SLDNF resolution is complete for A(D) ∪ T and goal {← u∧¬ιIc},

[14] If a deductive database D is stratified, strict and recursive predicates have only key arguments, then the augmented database A(D) is stratified, call consistent and allowed [UO92].

for any transaction T such that $\mathsf{u} \wedge \neg \iota \mathrm{Ic}$ is a logical consequence of comp(A(D) ∪ T), there is a constructive derivation from ({$\leftarrow \mathsf{u} \wedge \neg \iota \mathrm{Ic}$ } ∅ ∅) to ([] T C).

The technical proofs of both theorems are presented in Appendix B.

5. Related Work

We have compared our method with previous work proposed to deal with view updating and integrity constraint maintenance. In this section, we perform a comparison by considering an effectiveness point of view. Efficiency issues will be considered in detail in Section 7. We show first the main extensions of our method with regards to our precursor, the Events Method [TO95], and we provide afterwards a comparison with the rest of the proposals. A more detailed comparison with related work can be found in [MT99b].

5.1 Comparison with the Events Method [TO95]

Our method is an extension of the Events Method. Concerning to the definition of our method, we have introduced the modification as a new basic update operator in addition to the insertion and the deletion update operators. Additionally, key integrity constraints are implicitly enforced by the own definition of the method, instead of considering them as the rest of integrity constraints explicitly defined in the database schema. The main differences between both methods due to these features are the following:

Basic update operators

The Events Method can not deal with modifications as a basic update operator. A modification update is simulated by a deletion update followed by an insertion update. In our method, the introduction of this new basic update operator has caused some changes on the semantics of the insertion and deletion update operators. Therefore, the meaning of the insertion and the deletion events is not the same in the Events Method than in our method.

Example 5.1: Consider the following deductive database with base predicates Teaches(p,u) and Contract(p,u), they state respectively that a professor p teaches or has a contract in a university u. Derived predicate Visiting(p,u) states that professor p is a visiting professor at the university u if s/he teaches in a university and s/he does not have a contract with it.

Teaches(Joan, UPC) Contract(Joan,UG)

Visiting(p, u) ← Teaches(p, u) ∧ ¬ Contract(p, u)

An update request to add the fact Contract(Ann,UPC) to the EDB corresponds to the base event fact ιContract(Ann,UPC) in both methods. The same occurs with a request for removing the fact Teaches(Joan,UPC) that corresponds to the deletion event δTeaches(Joan,UPC). However, a request for adding the fact Teaches(Joan,UG) corresponds to the insertion event ιTeaches(Joan,UG) in the Events Method[15], while in our method, it corresponds to the modification event μTeaches(Joan,UPC,UG).

[15] Notice that to satisfy the request and not to violate key integrity constraints, the Events Method requires the deletion of fact Teaches(Joan,UPC) so, it includes the event δTeaches(Joan,UPC) into the solution.

Minimal solutions

Considering a modification as a new basic update operator causes also some differences with respect to the minimal solutions that our method and the Events Method obtain. In both cases, a solution T is considered minimal if no proper subset of T is a solution in itself. In some cases, and specifically when a solution contains some modification event fact, a solution that is considered minimal in our method may not be minimal in the Events Method. For this reason, the number of solutions obtained by both methods may differ in these cases.

Example 5.2: Assume the database of Example 5.1 and the update request u = δVisiting(Joan,UPC). In this example, the Events Method obtains two minimal solutions: T_1 = {δTeaches(Joan,UPC)} and T_2 = {ιContract(Joan,UPC), δContract(Joan,UG)}. Applying our method to the same request, we obtain four minimal solutions:

T_1 = {δTeaches(Joan,UPC)} T_2 = {μContract(Joan,UG,UPC)}

T_3 = {μTeaches(Joan,UPC,UG)} T_4 = {μTeaches(Joan,UPC,UO), μContract(Joan,UG,UO)

Notice that solutions of the Events Methods corresponding to T_3 and T_4 are not obtained because solution T_1 is a subset of solutions T_3 and T_4, and therefore they are not minimal.

Maintenance of key integrity constraints

The Events Method implicitly assumes that the key of each predicate is composed by all its attributes. Therefore, to state that a subset of its attributes corresponds to the key of a predicate, it is necessary to define explicitly into the database schema the corresponding key integrity constraint.

Defining explicitly key integrity constraints increases significantly the number of constraints to manage. Moreover, Events Method cannot take advantage of the particular semantics of these constraints since it does not provide any specific treatment for them. In contrast, in our method, we do not need to explicitly define key integrity constraints and its treatment is specifically performed by the own definition of our method.

5.2 Comparison with other Relevant Work

This section is aimed at illustrating, by means of examples, several drawbacks of relevant methods in the view updating and integrity constraint maintenance field. These methods are grouped according to the problem they address. Thus, Section 5.2.1 describes methods that deal with both integrity constraint maintenance and view updating [Wüt93, CST95, LT97, Dec97], while Section 5.2.2 analyses methods that consider only integrity constraint maintenance [CFPT94, Ger94, Sch98, Maa98].

Events Method [TO95] already provides an exhaustive comparison and a clear illustration of the main drawbacks of methods in this field proposed up to then (namely, [Dec90, GL90, KM90]). Therefore, we compare with more recent methods that are not covered in [TO95].

We would like to remark that most of these methods deal only with insertions and deletions as basic updates. Methods that deal also with modifications can not deal with modification on view predicates. Moreover, they do not take into account the information about keys of predicates and, then, they do not include maintenance of key constraints in the management of updates. Finally, we would like to remark that none of these methods has been proved to be, both, correct and complete.

5.2.1 Methods that deal with View Updating and Integrity Constraint Maintenance

We analyze first three methods [Wüt93, CST95, LT97] that follow a similar approach based on an unfolding process. These methods distinguish two steps to obtain the solutions. Given an update request u, the first step is aimed at obtaining a formula F, defined only in terms of base predicates, that characterizes all solutions of the update request. This formula is obtained by incorporating information of integrity constraints in u and by unfolding derived predicates by taking into account their definition. In the second step, the obtained formula F is analyzed to determine the base fact updates to apply to the extensional database.

The other method analyzed in this section [Dec97] is based on the SLDAI resolution procedure, which is an abductive extension of the SLD resolution procedure. The SLDAI procedure is an interleaving of *refutation* and *consistency* derivations. Given an update request, the refutation derivation pursues the empty clause by considering the database contents. During this derivation, new hypotheses are included in the solution set H. Every time a hypothesis is included in H, its consistency is verified by a consistency derivation.

Wüthrich's Method [Wüt93]

This method characterizes an update request as a conjunction of insertions and deletions of base and/or derived facts. A solution is defined by a set of base facts to be inserted (I) and a set of base facts to be deleted (D) that satisfy the requested update. The general approach to draw the solutions follows the two step approach outlined before.

This method has two main drawbacks: it is not complete (i.e. in some cases, it may not obtain some correct solutions) and it does not necessarily generate minimal solutions.

- *Some solutions may not be obtained by the method*. Wüthrich's method implicitly assumes that there is an ordering for dealing with the deductive rules and integrity constraints involved in the update request, which will lead to the generation of a solution. However, this ordering does not always exist. The following example shows that this assumption may cause this method not to obtain all valid solutions.

Example 5.3: Consider that the update u = insert(Edge(A,C)) is requested on the following database:

Node (A) Node (B) Edge (A, B) Edge (B, A)

Ic1 ← Node (x) ∧ ¬ ∃y Edge (x, y) Ic2 ← Node (x) ∧ ¬ ∃z Edge (z, x)

Ic3 ← Edge (x, y) ∧ ¬ Node (x) Ic4 ← Edge (x, y) ∧ ¬ Node (y)

Wüthrich's method may not obtain the solution characterized by the sets I={Edge(A,C), Node(C), Edge(C,D), Node(D), Edge(D,B)} and D=∅, that our method obtains.

- *Generation of non-Minimal Solutions*. In some cases, Wüthrich's method may only generate non-minimal solutions because it does not check whether a base or view fact is already present in the database when suggesting to insert or delete it. This is illustrated in the following example:

Example 5.4: Given the request u = insert(P(A)) and the following database:

S(A, B)

P(x) ← Q(x) ∧ R(x)

R(x) ← S(x, y)

In this example, Wüthrich's method could only obtain the non-minimal solution characterized by sets I={Q(A), S(A,C)} and D=∅, where C is a value given by the user or assigned by default. With our method, we obtain only one minimal solution T={ιQ(A)}, since we take into account the extensional database.

Console, Sapino and Theseider's Method [CST95]

This method restricts the integrity constraints it deals with. It can only handle flat integrity constraints, that is, integrity constraints that can not be defined by derived predicates. Moreover, it considers also two additional restrictions: integrity constraints must be in a denial form with almost two literals in the body; or they must be (non-cyclic) referential integrity constraints. In this sense, our method can be applied to maintain integrity constraints that this method can not enforce properly.

Lobo and Trajcevsky's Method [LT97]

This method presents two different drawbacks:

- *Restrictions on the Integrity Constraints*. This method requires the set of constraints to be *resolution complete*. A set of integrity constraints is resolution complete if it is not possible to derive new (implicit) integrity constraints from the integrity constraints it includes. For instance, integrity constraints Ic1 ← Q(x) ∧ ¬ R(x) and Ic2 ← R(x) ∧ S(x) are not resolution complete since a third integrity constraint can be deduced from them: Ic3 ← Q(x) ∧ S(x). The problem is that, as far as we know, there is no mechanism to derive sets of integrity constraints that are resolution complete.

- *Invalid Solutions*. This method is not always correct since the formula obtained after the unfolding process does not always characterize correct solutions. The following example illustrates this situation.

Example 5.5: Given the update request u = insert(Q(B,2)) and the following database:

S(A, 1)

Q(x, y) ← ¬ P ∧ S(x, y)

P ← S(x, y) ∧ ¬ T(y)

this method would obtain two solutions: S_1={delete(S(A,1)), insert(S(B,2))} and S_2={insert(T(1)), insert(S(B,2))}. However, none of them satisfies the requested update u since insert(S(B,2)) induces P and, hence, it falsifies the requested update insert(Q(B,2)).

On the contrary, our method would obtain the solutions: T_1={δS(A,1), ιS(B,2), ιT(2)} and T2={ιT(1), ιS(B,2), ιT(2)} which are the only ones that satisfy the update request.

Decker's Method [Dec96, Dec97]

The main limitation of this method is that it cannot manage appropriately update requests that involve rules with existential variables. The reason is that refutations flounder when a literal corresponding to a non-ground base predicate is selected, thus impeding to reach the empty clause.

Example 5.6: Given the update request u = insert(P) and the following database:

P ← S(x)

this method does not obtain any solution. On the contrary, our method obtains as many solutions as possible values of x exist to insert S(x).

Moreover, this method does not take into account the base facts during the consistency derivations. Therefore, this method may not obtain correct solutions since it flounders.

Example 5.7: Given the update request u = insert(P) and the following database:

$R(A, B)$

$P \leftarrow Q(A)$

$Ic1 \leftarrow Q(x) \wedge R(x, y) \wedge \neg S(y)$

Decker's method can not obtain any solution to insert fact P. On the contrary, our method would obtain two minimal solutions T1={ιQ(A), δR(A,B)} and T2={ιQ(A), ιS(B)}.

5.2.2 Methods that deal only with Integrity Constraint Maintenance

All these methods [CFPT94, Ger94, Sch98, Maa98] are based on the generation and execution of active rules. This approach is aimed at maintaining integrity constraints through the generation, at compile time, of active rules. These rules are executed at run-time when a certain transaction is applied to the database, in order to guarantee that the integrity constraints remain satisfied. Active rules are generated by taking into account the information provided by the database schema, and their action part contains the updates needed to repair an integrity constraint violation.

Although all these methods present several particularities regarding the generated rules or the language used to define the constraints, they share the same limitations. This is why in this section we only comment on these common drawbacks, instead of describing each method in detail.

- *The update request is not always satisfied*. One of the most common limitations of these methods is that the obtained solutions may not preserve the effect of the requested update. The reason is that they do not take into account the history of database updates needed to enforce database consistency and, thus, they cannot know whether the requested update is undone by the joint effect of these updates. This limitation, which was already identified in [ST96, Sch98], is illustrated in the following example:

Example 5.8 (adapted from [Sch98]): Assume the update request u = insert(Wire(Id1, HB, α)) and the following database:

$Tube(Id1, HB, \beta)$ $\quad\quad$ $Wire(Id5, HB, \alpha)$

$Wire(wire\_id, conn, w\_typ) \rightarrow Tube(tube\_id, conn, t\_typ)$

$Wire(wire\_id, conn, w\_typ) \wedge Tube(tube\_id, conn, t\_typ) \rightarrow wire\_id \neq tube\_id$

Here, methods [CFPT94, Ger94] may obtain a solution S={insert(Wire(Id1,HB,α)), delete(Tube(Id1,HB,β)), delete(Wire(Id1,HB,α)), delete(Wire(Id5,HB,α)} that does not satisfy the original request.

Schewe and Thalheim [ST96, Sch98, ST99] propose a technique to avoid this problem. For instance, in the previous example, this technique would detect that the generated solution does not satisfy the update request, and therefore it would not obtain it.

However, the main drawback of this technique is that, in some cases, it may not obtain solutions that satisfy the request. For instance, in Example 5.8, it would not obtain any

solution. However, there is a correct solution T={ιWire(Id1,HB,α), δTube(Id1,HB,β), ιTube(Id9,HB,ε)} that our method would obtain it.

- *Not all valid solutions can be obtained*. Once the set of active rules for integrity maintenance is generated, these methods [CFPT94, Ger94] define a graph that expresses whether the execution of a certain rule that repairs an integrity constraint could violate another integrity constraint. The presence of cycles in this graph indicates that the process of integrity maintenance could never terminate. To guarantee termination, the database designer should remove some active rules or to define priorities among them. However, the fact of not considering all active rules is equivalent to discarding some potential repair and, thus, not all solutions could be obtained by these methods.

6. Architecture of our Method

Section 6.1 analyses the inefficiency that a direct implementation of the rules that define our method would have. In Section 6.2, we describe a general architecture to improve efficiency of our method. Finally, we describe the techniques we use for improving efficiency in Sections 6.3 and 6.4.

6.1 Lack of Efficiency

As we shown in Section 4.2, our method is an interleaving of constructive and consistency derivations. The purpose of the constructive derivation is to build a transaction T of base event facts that satisfy the update request; while consistency derivations are aimed at ensuring that such transaction satisfies the update request and the integrity constraints.

We analyze efficiency of both derivations separately and we identify the causes of inefficiency of a direct implementation of the rules that define the method. We also introduce the main idea of the techniques proposed to solve this inefficiency.

6.1.1 Efficiency of the Constructive Derivation

The process of translating a request to update a derived predicate P into base updates is based on the event rules that define predicate P. In general, all event rules that define the derived event of predicate P would be considered to find a solution. Nevertheless, not all of them are useful because a database literal of some rule may not hold in the contents of the database or because some necessary base event can not be included into the transaction T.

Example 6.1: Consider the update request $u = \iota Emp(Bob,As)$ and the deductive rules of Example 2.1 but assume now that the extensional database is empty.

The insertion event rules of predicate Emp(p,c) are the following:

(I1) $\iota Emp(p,c) \leftarrow Works(p,c) \wedge \neg\delta Works(p,c) \wedge \neg Aux1(p,c) \wedge \neg Aux5(p) \wedge \iota Cont(p,c)$

(I2) $\iota Emp(p,c) \leftarrow Works(p,c) \wedge \neg\delta Works(p,c) \wedge \neg Aux1(p,c) \wedge Cont(p,c1) \wedge \mu Cont(p,c1,c) \wedge c1 \neq c$

(I3) $\iota Emp(p,c) \leftarrow \neg Aux6(p) \wedge \iota Works(p,c) \wedge Cont(p,c) \wedge \neg\delta Cont(p,c) \wedge \neg Aux2(p,c)$

(I4) $\iota Emp(p,c) \leftarrow \neg Aux6(p) \wedge \iota Works(p,c) \wedge \neg Aux5(p) \wedge \iota Cont(p,c)$

(I5) $\iota Emp(p,c) \leftarrow \neg Aux6(p) \wedge \iota Works(p,c) \wedge Cont(p,c1) \wedge \mu Cont(p,c1,c) \wedge c1 \neq c$

(I6) $\iota Emp(p,c) \leftarrow Works(p,c1) \wedge \mu Works(p,c1,c) \wedge c1 \neq c \wedge Cont(p,c) \wedge \neg\delta Cont(p,c) \wedge \neg Aux2(p,c)$

(I7) $\iota Emp(p,c) \leftarrow Works(p,c1) \wedge \mu Works(p,c1,c) \wedge c1 \neq c \wedge \neg Aux5(p) \wedge \iota Cont(p,c)$

(I8) $\iota Emp(\underline{p},c) \leftarrow Works(\underline{p},c1) \wedge \mu Works(\underline{p},c1,c) \wedge c1 \neq c \wedge Cont(\underline{p},c2) \wedge \mu Cont(\underline{p},c2,c) \wedge c2 \neq c \wedge c1 \neq c2$

(A1) $Aux1(\underline{p},c) \leftarrow \mu Works(\underline{p},c,c1)$ (A5) $Aux5(\underline{p}) \leftarrow Cont(\underline{p},c)$

(A2) $Aux2(\underline{p},c) \leftarrow \mu Cont(\underline{p},c,c1)$ (A6) $Aux6(\underline{p}) \leftarrow Works(\underline{p},c)$

Consider, for instance, the event rule I1. It is easy to see that this rule is not applicable if the database does not contain some fact Works(p,c). The same happens to the rest of rules except for I4, which is the only one that requires the database to be empty to be applicable.

We can avoid considering and exploring event rules that do not provide any solution if we are able to determine this set of rules before starting translation process. Intuitively, this can be done by evaluating the literals of the event rules and consider the contents of the database. The technique we propose in Section 6.4.2.1 (II) is based on this idea.

The second cause of inefficiency relies on the number of accesses performed to the extensional database to evaluate database literals of the event rules. Database literals evaluation in a branch of a derivation tree is performed independently of evaluations performed in other branches. Therefore, accesses to the same extensional fact may be repeated.

Example 6.2: Consider the update request and event rules of Example 6.1. Since each branch of the derivation tree rooted at the goal $\leftarrow \iota Emp(Bob,As)$ does not consider information of other branches, the extensional database accesses performed to obtain all solutions would be the following:

Works(Bob,As)	2 times	(rules I1, I2)	Cont(Bob,As)	2 times	(rules I3, I6)
$\neg$Works(Bob,c)	3 times	(rules I3, I4, I5)	$\neg$ Cont(Bob,c)	3 times	(rules I1, I4, I7)
Works(Bob,c)	3 times	(rules I6, I7, I8)	Cont(Bob,c)	3 times	(rules I2, I5, I8)

In fact, these accesses are performed in order to obtain the extension of base predicates Works(Bob,c) and Cont(Bob,c). Note that, in this case, the same information could be obtained performing only two queries: Works(Bob,c)? and Cont(Bob,c)?.

In this sense, an improvement of efficiency would consist in determining, before starting the translation process, which part of the extensional database requires to be known to obtain a solution to a given update request. Thus, we would reduce the number of accesses to the extensional database by avoiding perform repeated accesses to the same facts and by evaluating database literals taking into account the result of queries performed in other branches. The technique we propose in Section 6.4.2.1 (I) is based on this idea.

6.1.2 Efficiency of the Consistency Derivation

The consistency derivation process consists in getting a failure in each branch of the consistency derivation tree, by taking into account the contents of the transaction T and the extensional database. In some cases, the failure of a branch may require including new events into T and these inclusions may induce again the success of some branches that already failed previously. To prevent this situation, our method includes the goal of each branch that already fails into the condition set C. Thus, when a new event is included into the transaction T, the failure of all conditions of set C must be ensured again.

For this reason, a certain condition of C may be checked several times. First, when the condition is included into C, and later, every time an event is included into the transaction T. Moreover, each condition is checked independently whether its failure may be affected or not by the event included into T.

Example 6.3: Consider the example of Section 4.1. The Consistency Derivation shown in Figure 4.2, starts with the transaction T={ιIdNum(Mercè,12345)}and C=∅. During this process, new events are included into T and new conditions are included into C.

Integrity constraint Ic1 is taken into account before Ic2. Consequently, conditions are considered in the following order: C1, C2 (which is repaired and requires to consider again C1), C3 to C6 (repaired and implies take again into account C1 to C6), C7 and C8. Therefore, our method has ensured the conditions 14 times. However, if we check Ic2 before Ic1, our method would have ensured the conditions 18 times.

This example shows that the number of conditions to be ensured depends on the order in which the conditions are considered. To improve efficiency of the consistency derivation we will propose in Section 6.3 a technique that reduces the number of times that a condition must be ensured. This technique is based on delaying to deal with the conditions until all events needed to satisfy the update request are included into T and on determining an order in which the conditions should be considered. A preliminary version of this technique has been proposed in [MT99a].

6.2 General Architecture of our Method

In this section, we explain the general architecture of our method. Components of this architecture consider the techniques we will propose to solve the inefficiencies we have identified in the previous section. Figure 6.1 shows the architecture of our method, which distinguishes two different environments: the Compile-time and the Run-time environment.

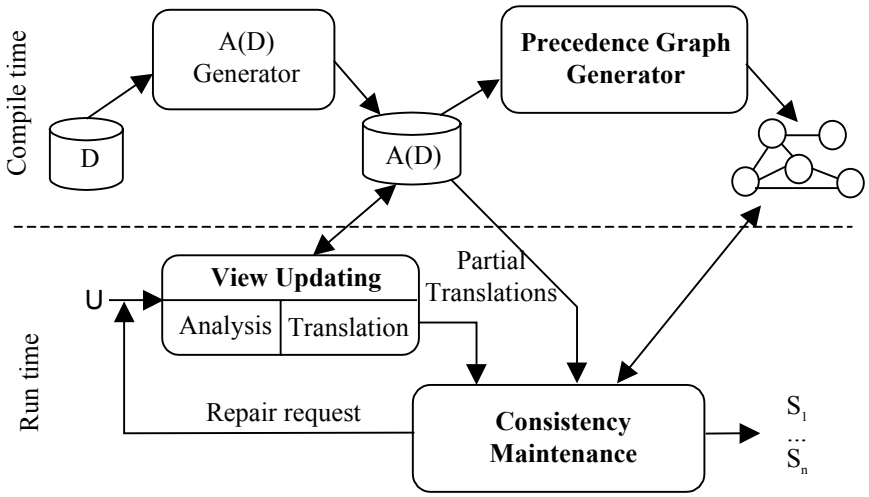


Figure 6.1 General Architecture of our Method

The Compile-time environment contains two components: the Augmented Database Generator and the Precedence Graph Generator. The Augmented Database Generator obtains and simplifies the set of rules that compose the Augmented Database A(D). The Precedence

Graph Generator performs a syntactical analysis of the A(D) and it obtains a directed graph that defines precedence between all the conditions. This graph will be used at Run-time to determine an efficient order to ensure these conditions. The Augmented Database Generator has been explained in Section 3. The Precedence Graph Generator is described in Section 6.3.

The Run-Time Environment obtains all the transactions $S_1 \ldots S_n$ that satisfy an update request u and it contains the view Updating and Consistency Maintenance modules. The View Updating module obtains all events needed to satisfy u and the conditions that must be preserved by these events (partial translations), while the Consistency Maintenance module ensures that those events do not violate any of these conditions and that they do not violate any integrity constraint. Note that some repairs may be requested as a result of this task.

Two steps define the View Updating module: the Analysis and the Translation steps. The Analysis step performs a preparatory work to improve efficiency of the Translation step, while the Translation step obtains all partial translations.

Definition 6.1: A *partial translation* TC=(T,C) of an update request u consists of a transaction T and a set of conditions C.

We have been implemented this architecture on a Sun OS environment using Quintus Prolog.

6.3 Precedence Graph Generator

The Precedence Graph is a directed graph whose nodes correspond to those conditions that will ensure that a transaction is a solution. Edges between nodes are labeled by a base event that corresponds to the update that could repair the condition of the outgoing node and, at the same time, violate the condition of the incoming node. In a sense, edges establish precedence between conditions. This precedence property will be used, at run-time, to determine the proper order in which conditions must be maintained.

Given a partial translation TC=(T,C) of an update request u, the transaction T can not be a solution of u if it violates some integrity constraint or if it does not satisfy the request u. Both situations are detected by our method. The first one is detected when transaction T induces the insertion event fact of the inconsistency predicate Ic. The second one is detected when transaction T makes true some condition of C.

We represent both situations by means of conditions. We distinguish two kinds of conditions: Integrity Constraint Conditions that represent the violation of an integrity constraint, and View Updating Conditions that ensure that a view update request is actually satisfied.

Definition 6.2: A *Condition* is a goal that a transaction must not satisfy to be a solution of an update request. It expresses a situation that must be prevented to actually satisfy the update request or not to violate an integrity constraint.

Definition 6.3: An *Integrity Constraint Condition* is a condition that states when an integrity constraint violation occurs. Its definition corresponds to the body of the insertion event rule of an inconsistency predicate (ιIc_j).

Definition 6.4: Given a partial translation TC=(T,C) and an update request u, a *View Updating Condition* is a condition of C that states a situation that must be prevented to ensure that transaction T actually satisfies the update request u.

Notice that Integrity Constraint Conditions are defined at compile-time, meanwhile View Updating Conditions can only be known at run-time since they depend on the specific update request. However, at compile time, we can distinguish three kinds of View Updating Conditions:

- Conditions that prevent the induction of a base event Ev.

  For each base event Ev, only one condition is defined in the following way: $\leftarrow$ Ev.

- Conditions that prevent the induction of a derived event Ev.

  For each derived event Ev, and for each event rule of Ev, a condition is defined in the following way: $\leftarrow Ev_i$, where Ev_i corresponds to the body of one event rule of the derived event Ev.

- Conditions that prevent induce a transition or auxiliary predicate P.

  For each transition and auxiliary predicate P, and for each transition/auxiliary rule of P, a condition is defined in the following way: $\leftarrow P_i$, where P_i corresponds to the body of one transition/auxiliary rule of predicate P.

Example 6.4: Consider the Augmented Database shown in the Example 3.6. Integrity Constraint Conditions associated to integrity constraints Ic1 and Ic2 are the following[16]:

(C1) $\leftarrow IdNum(\underline{p},n) \wedge \neg\delta IdNum(\underline{p},n) \wedge \neg Aux3(\underline{p},n) \wedge Contracted(\underline{p}) \wedge \delta Contracted(\underline{p})$

(C2) $\leftarrow \neg Aux7(\underline{p}) \wedge \iota IdNum(\underline{p},n) \wedge \neg Contracted(\underline{p}) \wedge \neg\iota Contracted(\underline{p})$

(C3) $\leftarrow \neg Aux7(\underline{p}) \wedge \iota IdNum(\underline{p},n) \wedge Contracted(\underline{p}) \wedge \delta Contracted(\underline{p})$

(C4) $\leftarrow IdNum(\underline{p},n) \wedge \mu IdNum(\underline{p},n,n1) \wedge n1 \neq n \wedge Contracted(\underline{p}) \wedge \delta Contracted(\underline{p})$

(C5) $\leftarrow Emp(\underline{p},c) \wedge \neg\delta Emp(\underline{p},c) \wedge \neg Aux4(\underline{p},c) \wedge Lab\_age(\underline{p}) \wedge \delta Lab\_age(\underline{p})$

(C6) $\leftarrow \neg Aux8(\underline{p}) \wedge \iota Emp(\underline{p},c) \wedge \neg Lab\_age(\underline{p}) \wedge \neg\iota Lab\_age(\underline{p})$

(C7) $\leftarrow \neg Aux8(\underline{p}) \wedge \iota Emp(\underline{p},c) \wedge Lab\_age(\underline{p}) \wedge \delta Lab\_age(\underline{p})$

(C8) $\leftarrow Emp(\underline{p},c) \wedge \mu Emp(\underline{p},c,c1) \wedge c1 \neq c \wedge Lab\_age(\underline{p}) \wedge \delta Lab\_age(\underline{p})$

Moreover, View Updating Conditions that prevent to induce the derived event $\delta Emp(\underline{p},c)$ are the following:

(D1) $\leftarrow Works(\underline{p},c) \wedge \delta Works(\underline{p},c) \wedge Cont(\underline{p},c)$

(D2) $\leftarrow Works(\underline{p},c) \wedge \mu Works(\underline{p},c,c1) \wedge c1 \neq c \wedge Cont(\underline{p},c) \wedge \neg\mu Cont(\underline{p},c,c1)$

(D3) $\leftarrow Works(\underline{p},c) \wedge Cont(\underline{p},c) \wedge \delta Cont(\underline{p},c)$

(D4) $\leftarrow Works(\underline{p},c) \wedge Cont(\underline{p},c) \wedge \mu Cont(\underline{p},c,c1) \wedge c1 \neq c \wedge \neg\mu Works(\underline{p},c,c1)$

Therefore, by considering Integrity Constraint Conditions and View Updating Conditions at compile-time, it is possible to analyze them syntactically in a uniform way and therefore, to build the Precedence Graph.

Nodes of the Precedence Graph have associated an Integrity Constraint Condition or a View Updating Condition[17]. Edges between nodes are labeled by base events. To define the

[16] Conditions are identified by the same label of the event rule that defines it.

precedence between nodes is necessary to identify the events that could violate each condition and the events that could repair them. However, to identify these events, we need first to explicitly state the relationship between base events and their effect on derived events and conditions. All this information can be obtained by a syntactical analysis of the Augmented Database rules.

In Section 6.3.1, we introduce the Dependency Graph of Events that is used to determine the conditions and the derived events that depend on each base event. In Section 6.3.2 we define the concept of precedence between conditions. Finally, in Section 6.3.3 we describe the process to obtain the Precedence Graph based on precedences between conditions.

6.3.1 Dependency Graph of Events

Several derived events (and conditions) may be induced (violated) when applying a transaction into the extensional database. For instance, in Example 3.6, the application of the base event ιWorks may induce the derived event ιEmp and conditions C6 and C7. The effect of transactions on conditions and derived events can be identified by the analysis of the dependencies between derived events and conditions on base events.

Definition 6.5: Let E be an event and C a condition or a derived event. We say that *C directly depends on E* when: if C is a derived event, there is a rule in A(D) with event C in the head and such that E appears in its body; if C is a condition, the event E appears in the body of the condition. In both cases, a direct dependence is *positive* (resp. *negative*) if E is a positive literal (resp. negative).

Example 6.5: Consider the derived event δEmp(p,c) and its derived event rules of the A(D) shown in the Example 3.6. In this example, we can identify direct positive dependencies between δEmp(p,c) and base events δWorks(p,c), μWorks(p,c,c'), δCont(p,c) and μCont(p,c,c'). Moreover, this derived event also depends directly and negatively on base events μWorks(p,c,c') and μCont(p,c,c').

By considering all direct dependencies between events and conditions we can build the Dependency Graph of Events, which explicitly states the relationship between the application of base events to the database and their induced effect.

Definition 6.6: Given a set of conditions and events, a *Dependency Graph of Events* DG is a pair DG = <Nod, Edg> where Nod is a finite number of nodes, Edg $\subseteq$ (Nod x Nod) is a set of directed edges such that each node n $\in$ Nod is labeled with an event or an identifier of a condition. Given two nodes v and v', there is an edge e = (v,v') iff v' directly depends on v. Edges are marked positively (resp. negatively) if the dependence is positive (resp. negative).

Example 6.6: Consider conditions of Example 6.4. Dependencies between them and base events are shown in the Dependency Graph of Events of Figure 6.2. Positive (resp. negative) dependencies are shown by a black (resp. dotted) arrow.

By considering all the information shown in the Dependency Graph of Events, it is possible to identify other kind of dependencies.

---

[17] In the following, we will refer as conditions when we will not distinguish between Integrity Constraint Conditions and View Updating Conditions.

Definition 6.7: Let DG be a and v and v' two nodes of DG. We say that:

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

Dependencies between events and conditions permit to determine potential violations and potential repairs of a condition. Intuitively, a potential violation of a condition C is a base event that when applied to the database may cause C to become true. On the other hand, a potential repair of a condition C is a base event that when applied to the database, may falsify C. This information can be identified by considering the dependencies defined by the Dependency Graph of Events.

Definition 6.8: Let E be an event and C a condition.

- E is a *potential violation* of C if C depends evenly on E.
- E is a *potential repair* of C if E is a base event and C depends oddly on E.

At definition time, we can not ensure that an event will be a real violation of a certain condition since the database must also satisfy other requirements that may be unknown at this moment. This is why we talk about potential violations. We talk about potential repairs since, in general, repairing a condition may require the application of more than one event.

Notice that a base event could be a potential violator and a potential repair of a condition at the same time. This situation appears when a condition depends evenly and oddly on the same base event by two different paths.

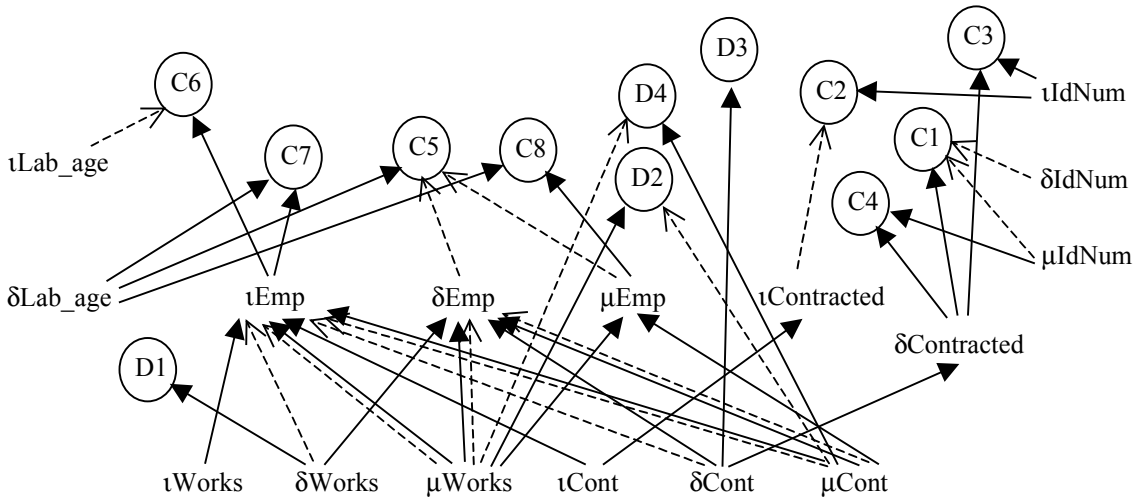


Figure 6.2 Dependency Graph of Events of Example 6.6

Example 6.7: Consider conditions C2 and C6 of Example 6.4 and its representation on the Dependency Graph of Figure 6.2. The potential violator of C2 is the base event ιIdNum since C2 depends evenly on this event. It also depends oddly on base event ιCont, therefore this event is the potential repair of C2. On the other hand, potential violators of C6 are events ιWorks, μWorks, ιCont and μCont, meanwhile its potential repairs are ιLab_age, δWorks, μWorks, δCont and μCont.

It may happen that no potential repair exists for a certain condition. So, we distinguish between two different kinds of conditions. Checking Conditions are those that have no potential repair; while conditions with some potential repair are the Generation Conditions. Each condition can be classified only into one of these two categories. This classification is independent of the distinction between Integrity Constraint and View Updating Conditions.

Definition 6.9: Let C be a condition. C is a C*hecking Condition* if there is no potential repair associated to it. Otherwise, C is a *Generation Condition*.

Example 6.8: Consider the conditions of Example 6.4. Conditions C3, C4, C8, D1, D3 are Checking Conditions since they do not have any potential repair. The rest of conditions are Generation Conditions.

6.3.2 Precedence between Conditions

We are interested on determining the proper order to maintain conditions, such that it reduces the number of times that each condition should be considered. To determine this order, we should identify those conditions whose potential repairs are potential violators of other conditions.

In this way, a condition Ci precedes a condition Cj when repairing condition Ci it is possible to violate condition Cj.

Definition 6.10: Let Ci be a Generation Condition and Cj a condition. We say that *Ci precedes Cj due to event E* (or *there is a precedence relationship between Ci and Cj due to event E*) if there is a base event E such that, at the same time, E is a potential repair of Ci and a potential violator of Cj.

Precedences between conditions are denoted as follows:

Ci, …, Ck → Cj, …, Cm due to E1, …, En

Example 6.9: Consider conditions C2 and C6 of Example 6.4. Condition C2 precedes condition C6 because base event ιCont is a potential repair of C2 and a potential violator of C6. We write this precedence in the following way: C2 → C6 due to ιCont.

Before building the Precedence Graph we may still apply some optimizations and simplify the set of precedence relationships identified by the above definition. Given a precedence Ci → Cj due to E, this optimizations is done by considering also the requirements (mainly database contents and necessary events) that are needed to falsify Ci and to violate Cj by means the event E.

It may happen that we encounter contradictory requirements, which means that this precedence does not really hold and, thus, it may be discarded. Possible contradictions may be to require a fact or an event to be true or false at the same time or require two mutually exclusive events to occur simultaneously [MT97].

Example 6.10: Consider conditions C1 to C8 of Example 6.4. The final set of precedence relationships between conditions after optimizing them is the following:

C2 → C6, C7	due to ιCont	C5, C6, C7 → C1, C3, C4	due to δCont
D2 → C8	due to μCont	C5 → D4, C8	due to μCont
C5 → D1	due to δWorks	C5 → D2, C8	due to μWorks
D4 → C8	due to μWorks	C1 → C4	due to μIdNum

6.3.3 Precedence Graph

By considering all precedence relationships together, we build the Precedence Graph, which explicitly states all relationships among repairs and violations of conditions. In this sense, it defines different orders in which conditions should be handled to reduce the number of times that a condition must be reconsidered. Moreover, the Precedence Graph is used to ensure that a repair of a certain condition C is only performed when it is guaranteed that all repairs of conditions that could induce a violation of C have been already obtained.

Definition 6.11: A *Precedence Graph* PG for a set C of conditions, is a triplet PG=<Nod,Sug,Edg> where Nod is a finite number of nodes, Sug is a set of precedence subgraphs and Edg $\subseteq$ (Nod x Nod) $\cup$ (Sug x Nod) is a set of directed edges. Each node n $\in$ Nod is labeled with a condition identifier, each subgraph is labeled by an adhoc identifier (gi) and each edge is labeled with a base event. There are two kinds of edges:

- an edge e = (n,n') between nodes n and n' labeled with event E, if the condition labeling node n precedes the condition labeling node n' due to event E
- an edge e = (g,n') between subgraph g and node n' labeled with event E, if there is a precedence between a condition of some node of the subgraph g and the node n' due to E

We distinguish subgraphs as a special case of nodes. A subgraph is a group of nodes whose conditions form a cycle. This kind of node reflects the situation that a repair of a condition C may violate other conditions, whose repairs could violate C again. In this case, we can not ensure that a condition of the subgraph is definitely maintained until all nodes of the subgraph are already maintained. This is the reason why a precedence between a condition Ci of a subgraph g and a condition Cj outside the subgraph is represented as a precedence between the subgraph g and the node labeled by Cj.

Example 6.11: The Precedence Graph of Example 6.4 is shown in the Figure 6.3.

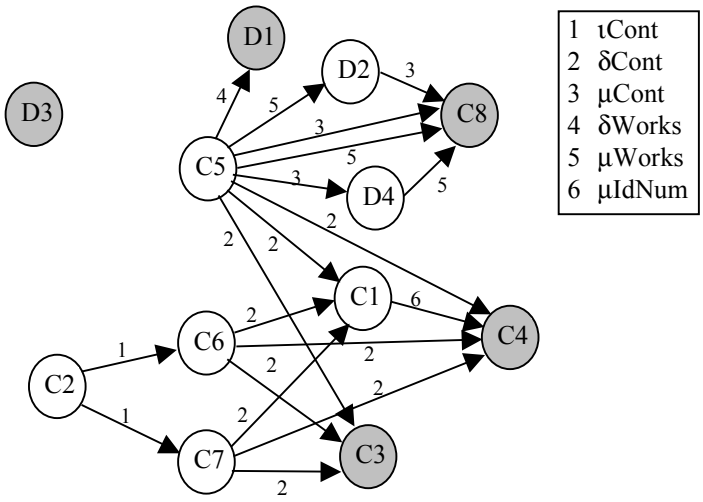


Figure 6.3 Precedence Graph

Several situations may be identified in a Precedence Graph:

- Nodes with a white background (like C5) have associated Generation conditions.
- Nodes with a gray background (like C4) have associated Checking conditions.

- Nodes with no outgoing edges indicate that repairs of its conditions can not violate any other condition.
- Nodes without incoming edges (like C2) indicate that any repair of other conditions can not violate its conditions.

6.4 Run Time Environment

Given an update request u, the run-time environment obtains all solutions that satisfy u. This is done by means of the Consistency Maintenance and View Updating modules.

6.4.1 Consistency Maintenance Module

Given a partial translation TC composed by a transaction T and a condition set C, the purpose of this module is to ensure that transaction T does not violate any condition of C and does not violate any integrity constraint.

To perform consistency maintenance, we determine first the conditions (nodes) of the Precedence Graph that may be violated due to T. Then, we check each of these conditions until all nodes of the Precedence Graph are visited. We use the View Updating Module when we need to repair a condition by means of a derived event.

This process is described in Section 6.4.1.2 but we need to introduce first (see Section 6.4.1.1) some aspects needed to understand it. Section 6.4.1.3 analyzes the efficiency improvement gained with this process.

6.4.1.1 Preliminary aspects

These aspects rely on how the Precedence Graph is considered during the Consistency Maintenance process. First of all, we introduce the distinction between active and non-active nodes; second, we analyze how many conditions may be associated with each node; third, we establish the criteria to select the next node to be considered; and, finally, we analyze how cycles (subgraphs) are managed during this process.

I. Active Nodes

Active nodes correspond to the subset of the Precedence Graph nodes that must be considered when a partial translation TC = (T, C) is taken into account. In fact, we must consider only nodes with View Updating Conditions of C and all nodes corresponding to Integrity Constraint Conditions. These nodes compose the Active Precedence Graph.

Definition 6.12: Given a partial translation TC = (T, C), a node of the Precedence Graph is an *Active Node* if it has associated an Integrity Constraint Condition, or if it has associated a View Updating Condition of set C.

Definition 6.13: Given a Precedence Graph PG and a set of active nodes, an *Active Precedence Graph* is the subgraph of PG composed by only the active nodes and edges between them.

Given a partial translation TC, the associated Active Precedence Graph is the specific graph that will be actually used during all the consistency maintenance process to determine the order of maintaining conditions. Notice that, for different partial translations, different Active Precedence Graphs are obtained.

Notice also that, the Active Precedence Graph is dynamic in the sense that new nodes could be activated during the consistency maintenance process. When a condition requires to be repaired by means of a view update request, new View Updating Conditions could be maintained.

Example 6.12: Consider the Precedence Graph of Example 6.11 and a partial translation TC composed by transaction T={ιIdNum(Mercè,12345)} and set of conditions C=∅. In this case, Active Nodes are those that contain an Integrity Constraint Condition, that is, nodes C1 to C8. Therefore, the Active Precedence Graph associated to TC is the following:

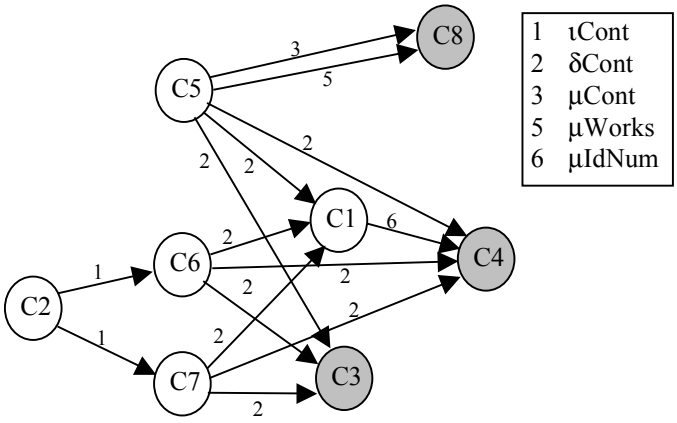


Figure 6.4 Active Precedence Graph

II. Conditions of a Node

A node of the Active Precedence Graph corresponding to a view updating condition may contain the original condition (already present in the Precedence Graph) as well as several particular instantiations of this condition. Those instantiations correspond to particular conditions provided by the partial translations.

Example 6.13: Consider a node (B1) of a Precedence Graph with the View Updating Condition ← ιCont(p,c), and the set of conditions C = {← ιCont(Peter,UPC); ← ιCont(Peter,UAB); ← ιCont(Paul,c)} of a partial translation TC. Then, the active node (B1) of the Active Precedence Graph will also contain three specific conditions:

(B1.1) ← ιCont(Peter,UPC)

(B1.2) ← ιCont(Peter,UAB)

(B1.3) ← ιCont(Paul,c)

III. Node's Selection Criteria

We mark the nodes of the Active Precedence Graph to distinguish nodes that have already been visited, from those that remain to be considered. In principle, all nodes are marked at the beginning of the Consistency Maintenance process. A node is unmarked when its conditions have been successfully maintained. Before selecting the next node to visit, we mark the successors of the current node that could be violated by the performed repairs.

The following criteria define which is the candidate node to deal with, after a certain node has been processed.

1. The candidate node must be marked.
2. The candidate node must have all predecessors unmarked to ensure that a condition C is not maintained until all conditions whose repairs could violate C have been already maintained.
3. If a subgraph (cycle) is selected, the candidate must be the node within the subgraph with less incoming edges.
4. If the current node belongs to a subgraph (cycle), the new candidate must belong to the same subgraph. Nodes outside a subgraph can not be selected until a subgraph has been completely processed.
5. Select preferentially nodes with Checking Conditions. If a Checking Condition is violated by transaction T, it can not be repaired and therefore transaction T must be rejected. Identifying as soon as possible violated checking conditions, we also improve efficiency of consistency maintenance.
6. If several candidates exist, select any of them.

By applying these criteria, we can visit all marked nodes of the Active Precedence Graph in different orders. All of them are equivalent since the number of conditions maintained will be the same.

IV. Cycles

Cycles between conditions are differentiated in the Active Precedence Graph as a node subgraph. When a node subgraph is selected, we must apply a recursive call of the consistency maintenance procedure to the subgraph definition.

The existence of a cycle between nodes in the Active Precedence Graph implies that a node could be visited more than once. The following Lemma and Theorem state that even in the presence of cycles the Consistency Maintenance process terminates when considering finite domains. Proofs are detailed in the Appendix C.

Lemma 6.1: Assuming finite domains and therefore, assuming a finite number of substitutions α_i. Any condition C could be violated with a substitution α_i, be repaired, and become violated again with the same substitution α_i a finite number of times.

Theorem 6.1: Given a transaction T and a set of precedence relationships that define a cycle in the Active Precedence Graph, the process of consistency maintenance of the associated conditions always finishes.

6.4.1.2 Consistency Maintenance Process

Given a Precedence Graph PG and a partial translation TC=(T,C), the purpose of the Consistency Maintenance Process is to obtain all solutions S such that $T \subseteq S$ that satisfy all conditions of set C and all the integrity constraints.

The process of consistency maintenance goes in the following way: given the Active Precedence Graph APG, a node to be visited is selected according to the criteria defined in previous section. If this node does not correspond to a subgraph, conditions of the node are checked. If no condition is violated, the current node is unmarked and a new node is selected. If all nodes are unmarked, the process is finished.

If some condition of the current node is violated, we proceed depending on the type of the condition. If it is a Checking Condition, the transaction T is rejected. If it is a Generation Condition, it must be repaired. In some cases, it may be necessary to use a View Updating procedure to repair it. If the condition can not be repaired, transaction T is rejected. If there is some repair, it is included into transaction T, successors of current node are marked and new nodes may be activated. After that, we continue the process by selecting the next node to visit.

We proceed in a similar way if the selected node corresponds to a subgraph. The treatment of this subgraph finishes by providing the set of pairs (T_g,C_g) that maintain conditions of nodes that form the subgraph (cycle). Before selecting a new node, the corresponding nodes must activated accordingly to C_g. Node subgraph is unmarked and successors are marked according to the performed repairs.

Given a transaction T, to obtain all possible ways to satisfy conditions of the Active Precedence Graph we consider all possible ways to repair a condition.

The function Consistency_Maintenance(T, APG, Cr), described in Appendix D, implements the process of consistency maintenance. T corresponds to the transaction of a partial translation TC=(T,C) for which consistency must be maintained. APG stands for the Active Precedence Graph to be taken into account, in which conditions of set C had been already included. Cr is a technical parameter necessary to manage subgraphs.

Example 6.14: Consider transaction T={ιIdNum(Mercè,12345)}, its Active Precedence Graph APG of Example 6.11 and the definition of its conditions of Example 6.4. Execution of the Consistency_Maintenance function is summarized in the following table. There is a column for each active node of the APG and the last column shows the contents of the transaction T. Each row indicates an execution step. An 'x' indicates that the node is marked and an '**X**' indicates the selected node in each step.

	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**Transaction T**
1	x	x	x	x	**X**	x	x	x	ιIdNum(Mercè,12345)
2	x	x	x	x		x	x	**X**	ιIdNum(Mercè,12345)
3	x	**X**	x	x		x	x		ιIdNum(Mercè,12345), ιCont(Mercè,UPC)
4	x		x	x		x	**X**		ιIdNum(Mercè,12345), ιCont(Mercè,UPC)
5	x		x	x		**X**			ιIdNum(Mercè,12345), ιCont(Mercè,UPC), ιLab_age(Mercè)
6	x		**X**	x					ιIdNum(Mercè,12345), ιCont(Mercè,UPC), ιLab_age(Mercè)
7	**X**			x					ιIdNum(Mercè,12345), ιCont(Mercè,UPC), ιLab_age(Mercè)
8				**X**					ιIdNum(Mercè,12345), ιCont(Mercè,UPC), ιLab_age(Mercè)

At step (1), nodes C2 and C5 may be selected since they do not have any marked predecessor. Node C5 is selected and checked. Since transaction T does not violate C5, the mark of the node is removed and C8 is selected. Its mark is also removed since its condition is not violated.

At step (3), the selected node is C2. In this case, its condition is violated by transaction T. To repair it, the view update ιContracted(Mercè) is requested. Using the View Updating Module (see Section 6.4.2), several partial translations are obtained. One of them corresponds to the transaction R={ιCont(Mercè,UPC)} and the set of conditions C=∅. Node C2 is unmarked, its successors are already marked and no new nodes are activated since C=∅.

At step (4), node C7 is selected and unmarked. At the next step (5), condition of node C6 is violated and it is repaired by the base event ιLab_age(Mercè). Since successors are already marked, we only unmark C6. In the following steps, nodes C3, C1 and C4 are selected in this order and since their conditions are not violated by T, these nodes are unmarked.

After the ninth step, all nodes of the Active Precedence Graph are unmarked. Therefore, transaction T'={ιIdNum(Mercè,12345), ιCont(Mercè,UPC), ιLab_age(Mercè)} is a solution to the initial update request T={ιIdNum(Mercè,12345)} since it satisfies all conditions of the Active Precedence Graph.

Other solutions to the initial request T could be obtained by considering alternative partial translations to the derived repair request of condition C2.

6.4.1.3 Efficiency of the Consistency Maintenance Process

The criteria we use to evaluate the efficiency improvement we obtain by considering the Precedence Graph information is the number of times that each condition is checked.

To perform this evaluation, we consider two different situations: in the first one, we consider a direct implementation of the formalization of our method of Section 4.2. In the second one, we consider the architecture of our method proposed in Section 6.4.1, in which we use the information provided by the Precedence Graph.

Lemma 6.2: Let N be the number of conditions of the condition set C and R the number of repairs needed to obtain a solution by a direct implementation of our method. Then, the number of conditions that must be considered to ensure transaction T is a solution is[18] ((N/2) * R + N).

Lemma 6.3: Let N be the number of conditions of the Active Precedence Graph of a transaction T, and R the number of repairs needed to obtain a solution by taking the information provided by the Active Precedence Graph into account. Then, if the Active Precedence Graph has no cycles, the number of conditions that must be checked to ensure that transaction T is a solution is N.

In the particular case that the Active Precedence Graph has some cycle, a condition of the cycle is checked as many times as loops are done in the cycle. This number is difficult to estimate at compile time, since it depends on the number of repairs performed at each loop, but it is not difficult to see that it will be substantially below the conditions considered by a direct implementation of our method.

From these lemmas, we can conclude that considering the Precedence Graph information, efficiency of the consistency maintenance process is substantially improved by reducing considerably the number of times that a condition is checked.

6.4.2 View Updating Module

The purpose of this module is to obtain all partial translations TC for a given an update request u. It is structured in two different steps: the Analysis and the Translation steps. The Analysis step performs a preparatory work to improve efficiency of the Translation step. This is done by analyzing the update request and identifying the event rules that, with the current

[18] In average, for each repair, we consider that condition set C contains half of the total of conditions (N/2).

contents of the database, may provide a partial translation for u. In the Translation step, we obtain all partial translations by considering the information provided by the Analysis step.

The Analysis and the Translation steps are described, respectively in Sections 6.4.2.1 and 6.4.2.2. In Section 6.4.2.3, we analyze the efficiency improvement obtained in this process.

6.4.2.1 Analysis Step

The main goal of this step is to discard all the event rules that may not lead to a correct partial translation. This is done by querying the contents of the extensional database that is relevant to the requested update.

Moreover, those queries are performed in a way that it will not be necessary to access the EDB again during the translation of the requested update.

I. *Determine the relevant extension of an update request u*

The relevant extension of an event corresponds to the base facts whose truth or falsity is required to be know in order to determine if the event may or may not occur. In a similar way, the relevant extension to an update request u corresponds to the relevant extension of all events in the update request.

Definition 6.14: Let E be an insertion, deletion or modification event of a predicate $P(\underline{k},x)$ defined (directly or indirectly) in terms of a set of base predicates $Q_i(\underline{h_i},y_i)$. Then, the *relevant extension of E* is the set of all base facts $Q_i(\underline{h_i},y_i)\theta$, where θ is the m.g.u. $(k,x)/(h_i,y_i)$.

Definition 6.15: Let u be an update request. Then, the *relevant extension of u* is the union of the relevant extension of the events in u.

Example 6.15: Consider the A(D) of Example 3.6 and the update request u = ιEmp(Bob,As). The extensional database contains only the fact Works(Mercè,UPC) and since the derived predicate Emp(p,c) is defined by the derivation rule $Emp(\underline{p},c) \leftarrow Works(\underline{p},c) \wedge Cont(\underline{p},c)$ the relevant extension of u is composed by base facts Works(Bob,As) and Cont(Bob,As). Notice that both facts are false.

Once we know the relevant extension of a view update request, we may determine the subset of the event rules that are relevant to obtain partial translations.

II. Specialize event rules

The selection of relevant event rules is performed by the specialization of the event rules of the derived events that appear in the update request. This specialization is performed by adapting a transformation proposed in [KS90] to split a general rule into two alternative rules: one specific rule to be applied to a concrete instance t' and a general rule to apply to the rest of values t≠t'.

Definition 6.16: Let E(t') be a derived event fact, where t' is a vector of constants and variables, let Ext be the relevant extension of E(t'). Then, for each event rule (R) $E(t) \leftarrow L_1 \wedge ... \wedge L_n$, we define two Specialized Event Rules (R_1') and (R_2') in the following way:

(R_1') $E(t) \leftarrow L_1 \wedge ... \wedge L_n \wedge t \neq t'$

(R_2') $[E(t) \leftarrow L_i \wedge ... \wedge L_j]\theta$ with $i \geq 1$ and $j \leq n$.

where θ is the m.g.u.(t,t') and where each database literal $L_i\theta$ of rule R_2' that evaluates true with respect to Ext is simplified.

This specialization process is applied also to the event rules of positive and negative derived event facts that appear in the body of rules R_2'. Thus, we obtain a complete specialization of the event rules defining E(t').

Note that if some database literal $L_i\theta$ evaluated with respect to Ext does not hold, rule R_2' is not obtained because it can not provide any partial translation to update request E(t'). Notice also, that an already specialized event rule (R_1') could be specialized again for different values t'.

Specialized event rules (R_1') and (R_2') are equivalent to original event rule R. The first one (R_1') is applicable to any value t≠t', while the second one (R_2') is applicable to the specific value t=t'. Therefore, an event rule R can be substituted by its specialized event rules.

Example 6.16: Consider the update request u = ιEmp(Bob,As), its relevant extension (Example 6.15) and the insertion event rules of ιEmp(p,c) of the A(D) (Example 6.1). After specializing these rules with respect to u and its relevant extension, we obtain the following specialized event rules:

($I1_1$') ιEmp(p,c) ← Works(p,c) ∧ ¬δWorks(p,c) ∧ ¬Aux1(p,c) ∧ ¬Aux5(p) ∧ ιCont(p,c) ∧ p≠Bob

($I2_1$') ιEmp(p,c) ← Works(p,c) ∧ ¬δWorks(p,c) ∧ ¬Aux1(p,c) ∧ Cont(p,c1) ∧ μCont(p,c1,c) ∧ c1≠c ∧ p≠Bob

($I3_1$') ιEmp(p,c) ← ¬Aux6(p) ∧ ιWorks(p,c) ∧ Cont(p,c) ∧ ¬δCont(p,c) ∧ ¬Aux2(p,c) ∧ p≠Bob

($I4_1$') ιEmp(p,c) ← ¬Aux6(p) ∧ ιWorks(p,c) ∧ ¬Aux5(p) ∧ ιCont(p,c) ∧ p≠Bob

($I4_2$') ιEmp(Bob,As) ← ιWorks(Bob,As) ∧ ιCont(Bob,As)

($I5_1$') ιEmp(p,c) ← ¬Aux6(p) ∧ ιWorks(p,c) ∧ Cont(p,c1) ∧ μCont(p,c1,c) ∧ c1≠c ∧ p≠Bob

($I6_1$') ιEmp(p,c) ← Works(p,c1) ∧ μWorks(p,c1,c) ∧ c1≠c ∧ Cont(p,c) ∧ ¬δCont(p,c) ∧ ¬Aux2(p,c) ∧ p≠Bob

($I7_1$') ιEmp(p,c) ← Works(p,c1) ∧ μWorks(p,c1,c) ∧ c1≠c ∧ ¬Aux5(p) ∧ ιCont(p,c) ∧ p≠Bob

($I8_1$') ιEmp(p,c) ← Works(p,c1) ∧ μWorks(p,c1,c) ∧ c1≠c ∧ Cont(p,c2) ∧ μCont(p,c2,c) ∧ c2≠c ∧ c1≠c2 ∧ p≠Bob

In this example, the insertion event rule ($I4_2$') is the only one that could provide a partial translation to the update request. Notice that it does not contain database literals in its body.

The Translation step uses the specialized event rules of the form R_2' to obtain all partial translations that satisfy the update request. Note that the translation step will be performed without accessing against the EDB since there is no database literal in the body of these rules.

6.4.2.2 Translation Step

The purpose of the Translation Step is to obtain all partial translations $TC_i=(T_i,C_i)$ of the update request u. To obtain them, we benefit from the preliminary work performed at the analysis step by considering only the specialized event rules (R_2'). The body of specialized event rules may only contain positive and negative events. Positive base events define the updates that are needed to satisfy the update request u. On the other hand, negative events correspond to updates that must be prevented, since they could dismiss u.

The translation process consists in unfolding the update request by considering specialized event rules until a goal where all positive events are base events. After that, positive base event facts are included into T_i, and definitions of negative literals are included

as conditions in C_i. In Appendix D, we describe the algorithm of the procedure Translate_Update(u, A(D), STC) that implements this process.

Example 6.17: Consider the update request u = ιEmp(Bob,As). Unfolding the initial update request u taking into account the specialized event rules of Example 6.16, we obtain the formula F = ιWorks(Bob,As) ∧ ιCont(Bob,As). At the end, we obtain only one partial translation TC, which contains the transaction T={ιWorks(Bob,As), ιCont(Bob,As)} and the condition set C=∅.

During the translation process, we have not checked whether the transaction T satisfies integrity constraints and conditions of set C since, as we have seen in Section 6.4.1, this task is performed at the Consistency Maintenance module. Consequently, at the end of the view updating process it is not ensured that a transaction T really satisfies neither the update request u nor the integrity constraints. We can only state that events of T are necessary to satisfy u, and conditions of C must be enforced to ensure that T actually satisfies u.

6.4.2.3 Efficiency of the View Updating Process

Two relevant measures have just proposed to evaluate the efficiency improvement of the process of view updating with respect to a direct implementation of our method. First one is to the number of event rules that are taken into account to perform the translation and, the second is the number of times that the extensional database is accessed. Lemmas 6.4 and 6.5, respectively, provide the result of the efficiency comparison with regard to these measures.

To perform this comparison, we have considered a derived predicate without non-key arguments and defined in terms of only base predicates. This simple case is enough to show how the techniques introduced in the View Updating module improve efficiency of the view updating process. The derived predicate P(k) we consider is defined by the following rule P(k) $\leftarrow L_1 \wedge ... \wedge L_n$, with n≥1, where literals $L_1,...,L_n$ refer to base predicates and such that all variables in $L_1,...,L_n$ appear also in vector k.

Notice that, for a derived predicate P(k) without non-key arguments, the A(D) contains 2^n-1 insertion event rules, each one applicable to a different database state. These database states correspond to the different ways of not satisfying the derived fact P(k). On the contrary, the A(D) contains n deletion event rules, all of them applicable to the database state in which fact P(k) is true.

Lemma 6.4: Consider a derived predicate P(k) defined by the following rule P(k) $\leftarrow L_1 \wedge ... \wedge L_n$, with n≥1, where $L_1,...,L_n$ are base literals and such that all variables in $L_1,...,L_n$ appear also in vector k. Then, to obtain all solutions, a direct implementation of the method requires to consider all event rules and n extensional database accesses to each one. That is:

- for an insertion request ιP(k): 2^n-1 event rules and n*(2^n-1) database accesses
- for a deletion request δP(k): n event rules and n^2 database accesses

In the other hand, in the View Updating module, all database accesses are performed during the Analysis step, when the relevant extension of the update request is obtained. In the case of an insertion update requests (ιP(k)), it is necessary to perform only one access for each base fact of the relevant extension. Meanwhile, in the case of requesting a deletion update (δP(k)), it is enough to perform only one database access to query if the derived fact (P(k))

evaluates true. With respect to the number of specialized event rules we consider during the Translation step, we can state that for insertion requests, only one insertion event rule ($\iota P(\underline{k})$) is applicable. On the contrary, for deletion requests ($\delta P(\underline{k})$) all event rules are applicable.

Lemma 6.5: Consider a derived predicate $P(\underline{k})$ defined by the following rule $P(\underline{k}) \leftarrow L_1 \wedge ... \wedge L_n$, with $n \geq 1$, where $L_1,...,L_n$ are base literals and such that all variables in $L_1,...,L_n$ appear also in vector k. Then, to obtain all partial translations the View Updating module requires

- for an insertion request $\iota P(\underline{k})$: 1 event rule and n database accesses
- for a deletion request $\delta P(\underline{k})$: n event rules and 1 database access

By comparing both results, we can state that the preparatory work performed at the Analysis step allows an important reduction in the number of database accesses and the number of event rules to consider during view updating. In this way, we show that efficiency of view updating has been substantially improved.

7. Related Work on Efficiency

To the best of our knowledge, previous work on view updating did not care much about efficiency issues. They are more concerned with proposing effective methods that obtain all possible translations rather than trying to obtain these translations efficiently. Therefore, it is very difficult to provide an efficiency comparison with this line of research since, in our opinion, it would not be fair to compare a declarative specification of a method with a procedural implementation as the one we have proposed in the Section 6.

With respect to integrity constraint maintenance methods, only a few proposals care about efficiency issues [CFPT94, Ger94, FP97]. In fact, [CFPT94, Ger94] follow a similar approach which consists on the automatic generation of production rules (CA rules or ECA rules, respectively) for integrity maintenance. The execution of these rules guarantees that a transaction applied to a database maintains the integrity constraints.

Efficiency is provided in both approaches by defining a graph that expresses whether the execution of a certain production rule R_i that repairs an integrity constraint Ic_j could violate another integrity constraint Ic_k. Nodes of the graph represent integrity constraints while arcs represent production rules that may repair an integrity constraint and violate another one.

Example 7.1: Consider a database defined by the following integrity constraints:

$$Ic1(x) \leftarrow PhD(x) \wedge \neg Grad(x)$$

$$Ic2(x) \leftarrow Prof(x) \wedge \neg PhD(x)$$

$$Ic3(x) \leftarrow Dean(x) \wedge \neg Prof(x)$$

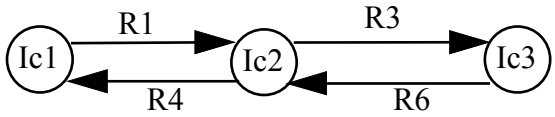
In this example, CA rules and ECA rules generated by [CFPT94] and by [Ger94], respectively, are the following[19]:

[CFPT94]	**[Ger94]**
R1: $PhD(x) \wedge \neg Grad(x) \rightarrow delete(PhD(x))$	R1: $\Delta^{Delete(Grad)}_{Ic1}(x) \wedge (PhD(x)) \succ \neg PhD(x)$

[19] To facilitate the comparison with our work, we do not consider the modification as a basic update operator nor the default production rules defined by [Ger94].

R2: $PhD(x) \wedge \neg Grad(x) \rightarrow insert(Grad(x))$	R2: $\Delta^{Insert(PhD)}_{Ic1}(x) \wedge (\neg Grad(x)) \succ Grad(x)$
R3: $Prof(x) \wedge \neg PhD(x) \rightarrow delete(Prof(x))$	R3: $\Delta^{Delete(PhD)}_{Ic2}(x) \wedge (Prof(x)) \succ \neg Prof(x)$
R4: $Prof(x) \wedge \neg PhD(x) \rightarrow insert(PhD(x))$	R4: $\Delta^{Insert(Prof)}_{Ic2}(x) \wedge (\neg PhD(x)) \succ PhD(x)$
R5: $Dean(x) \wedge \neg Prof(x) \rightarrow delete(Dean(x))$	R5: $\Delta^{Delete(Prof)}_{Ic3}(x) \wedge (Dean(x)) \succ \neg Dean(x)$
R6: $Dean(x) \wedge \neg Prof(x) \rightarrow insert(Prof(x))$	R6: $\Delta^{Insert(Dean)}_{Ic3}(x) \wedge (\neg Prof(x)) \succ Prof(x)$

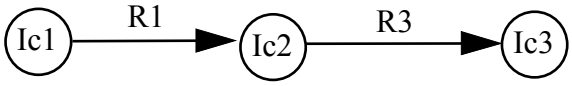
The graph generated by [CFPT94][20] and by [Ger94] is the following:



The presence of cycles in this graph indicates that, with the current set of production rules, the process of integrity maintenance could never terminate. For example, given the initial transaction T={insert(Dean(Bob))}, execution of production rules to maintain the integrity constraints can enter an infinite loop by executing R6 and R3 one after the other.

[CFPT94] and [Ger94] suggest obtaining an acyclic graph to guarantee termination of the process of integrity maintenance. This acyclic graph is obtained by removing arcs from recursive paths [CFPT94] or by giving priorities to active rules [Ger94]. However, by discarding some arcs, they may discard a repair action for a certain integrity constraint. Thus, they loose effectiveness (completeness) because they do not consider all repairs for an integrity constraint violation, and therefore they may not be able to obtain some solutions.

Example 7.2: Consider again Example 7.1 and assume that the generated acyclic graph by both methods is the following:



Now, given the initial transaction T={insert (Dean(Bob))}, the execution of this graph results in only one transaction: T'={insert(Dean(Bob)), delete(Dean(Bob))}. Moreover, observe that in this case, transaction T' is not a valid solution since it does not satisfy the initial update request T.

In our approach, we do not need to restrict the set of possible repairs as these methods do. Our approach is based on defining a more precise graph by splitting each integrity constraint into several conditions. Each one defines a different way of violating and repairing a constraint. Our approach results in a bigger number of conditions to be taken into account, but it allows us to effectively handling a higher number of cases. For instance, by considering the Example 7.1, our method obtains the solution T"={insert(Dean(Bob)), insert(Prof(Bob)), insert(PhD(Bob)), insert(Grad(Bob))} that [CFPT94, Ger94] may not obtain.

Work of [FP97] is devoted to define an stratified order of handling production rules and, indirectly, of handling integrity constraint violations to avoid infinite loops, more precisely than in [CFPT94]. However, it shares with previous work the loose of completeness because it may not consider all possible repairs.

[20] The graph proposed by [CFPT94] is aimed at studying the problem of termination of production rule execution instead of explicitly proposing an order for improving efficiency. However, this order is implicitly provided by the execution mechanism.

8. Conclusions and Further Work

We have proposed a new method that tackles the integrity constraint maintenance and view updating problems in an efficient way. That is, given a consistent deductive database and an update request, our method automatically obtains all possible ways to change the extensional database such that the update request is satisfied and integrity constraints are not violated. We have proposed several techniques to improve efficiency of this process.

Our method deals with three basic update operators: insertions, deletions and modifications on base and derived predicates and it is based on a set of event and transition rules, which explicitly define the insertions, deletions and modifications that could be induced when a transaction is applied. Our method is formally defined as an extension of the SLDNF proof procedure and we have proved that it is sound and complete for stratified databases. That is, obtained translations satisfy the update request (soundness) and all solutions are obtained (completeness).

Efficiency is achieved by making explicit the information required to define an order of handling integrity constraints and by performing some preparatory work before translating a view update. In the first case, we get efficiency by reducing the number of times that each integrity constraint must be reconsidered during the process of integrity maintenance. In the second case, efficiency is gained by considering only those alternatives that are relevant to the request and by reducing the number of accesses to the extensional database.

We have shown that our method improves previous work in the field. After an exhaustive analysis of all methods we know, we can state that our method is more powerful than others because it does not present some of their limitations, and because it deals with certain situations the others cannot handle appropriately.

Moreover, we have shown that our method is one of the first proposals that explicitly incorporates specific techniques to improve efficiency during the integrity constraint maintenance and view updating.

As future work, we would like to analyze how our method could be extended to deal with other database models like relational or object oriented databases. In particular, we would like to analyze how the proposed Precedence Graph could be incorporated to improve efficiency of the integrity constraint maintenance process in a RDBMS. Additionally, we are also considering integrating into our method several integrity constraint enforcement policies. That is, to allow the designer to distinguish between those integrity constraints to be checked and those to be maintained.

With respect to efficiency issues, we plan to take into account the information provided by the original transaction. This information could allow us to detect and select, as soon as possible, such conditions that would not be repaired in any case. Moreover, we are thinking about considering a more clever marking of the graph by incorporating techniques like those proposed in [LS93, LL96] to determine whether a condition could be actually affected by a repair.

Acknowledgments

We are grateful to D. Costal, C. Farré, 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 TIC97-1157.

References

[BR86] Bancilhom, F.; Ramakrisham, R. "An amateur's introduction to recursive query processing strategies", Proc. ACM SIGMOD Conf. On Management of Data, Washington D.C., (1986), pp. 16-52.

[CFPT94] Ceri, S.; Fraternali, P.; Paraboschi, S.; Tanca, L. "Automatic Generation of Production Rules for Integrity Maintenance" ACM Transactions on Database Systems Vol.19 Nº3, (September 1994), pp-367-422.

[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.

[Cla78] Clark, K.L. "Negation as Failure", in Gallaire, H.; Minker, J. "Logic and Databases" Plenum Press, New York, pp. 293-322.

[CST95] Console, L.; Sapino, M.L.; Theseider,D. "The Role of Abduction in Database View Updating", Journal of Intelligent Information Systems, Vol. 4, p.p. 261-280.

[Dec96] Decker, H. "An Extension of SLD by Abduction and Integrity Maintenance for View Updating in Deductive Databases", Joint International Conference and Symposium on Logic Programming JICSLP'96, (September 1996), Bonn (Germany), pp.157-169.

[Dec97] Decker, H. " One Abductive Logic Programming Procedure for two kind of Updates ", Proc. Workshop DINAMICS'97 at Int. Logic Programming Symposium, (September 1997), Port Jefferson, New York.

[FP97] Fraternali, P.; Paraboschi, S. "Ordering and Selecting Production Rules for Constraint Maintenance: Complexity and Heuristic Solution" IEEE Transactions on Knowledge and Data Engineering, Vol. 9, Nº 1, (1997), pp-173-178.

[GCMD94] García, C; Celma, M.; Mota, L.; Decker, H. "Comparing and Synthesizing Integrity Checking Methods for Deductive Databases", 10th Int. Conf. on Data Engineering ICDE'94, Houston, USA, (1994), pp. 214-222.

[Ger94] Gertz, M. "Specifying Reactive Integrity Control for Active Databases", Proceedings of RIDE'94, Houston, Texas, (1994), pp. 62-70.

[GL90] Guessoum, A.; Lloyd, J.W. "Updating Knowledge Bases", New Generation Computing Vol. 8, Num. 1, (1990), pp. 71-89.

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

[KS90] Kowalski, R.; Sadri, F. "Logic Programs with Exceptions", Seventh International Conference of Logic Programming ICLP'90, (1990), pp.598-613.

[LL96] Lee, S.Y.; Ling, T.W. "Further Improvement of Integrity Constraint Checking for Stratified Deductive Databases" Proc. of 22th VLDB Conference, Bombay, India, (September 1996).

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

[LlT84] Lloyd, J. W.; Topor, R. W. “Making Prolog More Expressive”, Journal of Logic Programming (1984), Nº 3, pp. 225-240.

[LT97] Lobo, J.; Trajcevski, G. “Minimal and consistent evolution in knowledge bases” Journal of Applied Non-Classical Logics. Vol. 7. Nº 1-2, (1997), pp 117-146.

[LS93] Levy, A.Y.; Sagiv, Y. “Query Independent of Updates”, Proc. of 19th VLDB Conference, Dublin, Ireland, (1993).

[Maa98] Maabout, S. “Maintaining and Restoring Database Consistency with Update Rules “ Proc. Worksh. DYNAMICS’98 (post conf. Work. Of JICSLP’98), Manchester, (June 98).

[ML91] Moerkotte, G.; Lockemann, P.C. “Reactive Consistency Control in Deductive Databases” ACM Transactions on Database Systems, Vol.16(4), (1991), pp.670-702.

[MT97] Mayol, E.; Teniente, E. “Structuring the Process of Integrity Maintenance”, 8th International Conference on Database and Expert Systems Applications DEXA’97, Toulouse, France, (September 1997), pp. 262-275.

[MT99a] Mayol, E.; Teniente, E. “Addressing Efficiency Issues During the Process of Integrity Maintenance”, 10th International Conference on Database and Expert Systems Applications DEXA’99, Florence, Italy, (September 1999), pp. 270-281.

[MT99b] Mayol, E.; Teniente, E. “A Survey of Current Methods for Integrity Constraint Maintenance and View Updating”, First International Workshop on Evolution and Change in Data Management ECDM’99, Workshop associated to the ER’99, Paris, France, (November 1999), pp. 62-73.

[MT00] Mayol, E.; Teniente, E. “Dealing with Modification Requests During View Updating and Integrity Constraint Maintenance”, First International Symposium on Foundations of Information and Knowledge Systems FoIKS’00, Burg, Germany, (February 2000), pp.192-212.

[May00] Mayol, E. “Actualització consistent de bases de dades deductives”, PhD Thesis, Barcelona, 2000 (written in catalan).

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

[Sch98] Schewe, K.D. “Consistency Enforcement in Entity-Relationship and Object-Oriented Models”, Data & Knowledge Engineering Vol 28, Nº1, (1998), pp.121-140

[Sel95] Seljée, R. “A New Method for Integrity Constraint Checking in Deductive Databases” Data&Knowledge Engineering Vol.15, (1995), pp.63-102.

[ST96] Schewe, K.D.; Thalheim, B. “Active Consistency Enforcement for Repairable Database Transitions”, 6th Int. Workshop on Foundations of Models and Languages for Data and Objects: Integrity in Databases, Dagstuhl, Germany, (September 1996).

[ST99] Schewe, K.D.; Thalheim, B. “Towards a theory of consistency enforcement“, Acta Informatica Vol 36 Num. 2 (1999), pp.97-141

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

[TU95] Teniente, E.; Urpí, T. “A Common Framework for Classifying and Specifying Deductive Database Updating Problems”, 11th Int. Conf. on Data Engineering ICDE, Taipei, Taiwan, (1995), pp. 173-183.

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

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

[Wüt93] Wüthrich, B. “On Updates and Inconsistency Repairing in Knowledge Bases”, Int. Conference on Data Engineering ICDE’93, Vienna, (1993), pp.608-615.

Appendix A. Soundness of our method

In this appendix, we prove the soundness of our method. First of all, we have to define the concepts of constructive and consistency derivations of level k[1].

Definition: Let G be a goal, T and T' transactions and C and C' condition sets. A *consistency derivation of level 0* from ({G} T C) to ({} T' C') is a consistency derivation that does not call any constructive derivation nor any consistency derivation.

Definition: Let G be a goal, T and T' transactions and C and C' condition sets. A *constructive derivation of level 0* from (G T C) to ([] T' C') is a constructive derivation that does not call any consistency derivation, or it calls only consistency derivations of level 0.

Definition: Let G be a goal, T and T' transactions and C and C' condition sets. A *consistency derivation of level k+1* from ({G} T C) to ({} T' C') is a consistency derivation that calls some constructive or consistency derivation of level k.

Definition: Let G be a goal, T and T' transactions and C and C' condition sets. A *constructive derivation of level k+1* from (G T C) to ([] T' C') is a constructive derivation that calls some consistency derivation of level k.

Let u be an update request. Lemma 1 states that there exists an SLDNF refutation of $A(D) \cup T \cup \{\leftarrow u \wedge \neg \iota Ic\}$ for every solution T obtained by our method.

Lemma 1: Let D be a deductive database, A(D) the augmented database, u an update request and T a minimal solution such that there is a constructive derivation from ($\leftarrow u \wedge \neg \iota Ic \varnothing \varnothing$) to ([] T C). Then, an SLDNF refutation of $A(D) \cup T \cup \{\leftarrow u \wedge \neg \iota Ic\}$ exists.

Proof: We have to prove that the steps in the above constructive derivation and the subsidiary consistency derivations correspond to SLDNF resolution steps, where clauses of $A(D) \cup T$ act as input clauses. This proof is performed by induction on the level k of these derivations.

Let G be a goal, T and T' transactions and C and C' condition sets. We first prove that a consistency derivation corresponds to a finitely failed SLDNF tree. This result is used afterwards to prove that a constructive derivation corresponds to an SLDNF refutation.

k = 0

- Let CS be a *consistency derivation* of level 0 from ({G} T C) to ({} T' C'). Then, the SLDNF derivation tree of $A(D) \cup T' \cup \{G\}$ fails finitely[2]. It corresponds to the failure of goal G.
  - Step B1 corresponds to a SLDNF resolution step with input set A(D).
  - Steps B2 and B4 correspond to SLDNF resolution steps with input set T=T'.
  - Steps B3 and B5 are not applicable for k=0.
- Let CT be a *constructive derivation* of level 0 from (G T C) to ([] T' C'). Then, there exists an SLDNF refutation of $A(D) \cup T' \cup \{G\}$. We distinguish two cases:

  1. *No consistency derivation is called*
     - Step A1 corresponds to a SLDNF resolution step with input set A(D).
     - Step A21 corresponds to a SLDNF resolution step with input set T'=T.

[1] Note that the concept of level is different to the concept of rank of SLDNF derivation defined by Lloyd [Llo87].

[2] When G is a set of n>1 goals, the root of the tree is an implicit goal $\leftarrow F$ with n descendant branches, one for each goal H_i in G.

In Step A22 a ground event L_j could be included to T (T' = T∪{$L_j\sigma$}) and no consistency derivation is called (C=∅). Therefore, step A2 corresponds to an SLDNF resolution step with input set T'.

– Step A3 is not applicable in this case.

2. *Some consistency derivation of level 0 is called*

– Step A1 corresponds to a SLDNF resolution step with input set A(D).

– Step A21 corresponds to a SLDNF resolution step with input set T'=T.
In step A22, a ground event L_j could be included to T, and so, a consistency derivation of level 0 is called from (C T∪{$L_j\sigma$} C) to ({} T' C'). We will get a new goal in the constructive derivation if there exists the above consistency derivation. The consistency derivation corresponds to the negation as failure of C with respect to the new input clause $L_j\sigma$. Previous failure of C with respect to T is not altered. Therefore, step A2 correspond to a SLDNF resolution step with input set T'.

– In step A3, it is checked that a consistency derivation of level 0 exists from (←¬L_j T C) to ({} T' C'), that is, the failure of ¬L_j is checked. This corresponds to a negation as failure rule and, therefore, step A3 is a SLDNF resolution step.

k = k

The base case has been proved and we assume the result is true for derivations of level k.

k = k+1

Now, we are going to prove that the Lemma 1 also holds for derivations of level k+1.

a) Let CS be a *consistency derivation* of level k+1 from ({G} T C) to ({} T' C'). Then, the SLDNF derivation tree of A(D) ∪ T' ∪ {G} fails finitely[2].

– Step B1 corresponds to a SLDNF resolution step with input set A(D).

– Step B2 corresponds to a SLDNF resolution step with input set T'. Failure of goal G is ensured:
  · for events that belong to T, by the own definition of this step.
  · for those base events that will be included into T after this step (T'-T). These events will be included in a constructive derivation of level k (step A2). The goal G is included to set C, and therefore, its failure will be ensured by the consistency derivation of step A2.

– Step B3 corresponds to a SLDNF resolution step with the input set A(D) ∪ T'. By induction, the SLDNF derivation tree of A(D) ∪ T' ∪ {←¬L_j} fails finitely since a consistency derivation of level k is called.

– Step B4 corresponds to a SLDNF resolution step with input set T.

– Step B5 corresponds to a SLDNF resolution step with the input set A(D) ∪ T'. By induction, there exists a refutation of A(D) ∪ T' ∪ {←¬L_j} since a constructive derivation of level k is called.

b) Let CT be a *constructive derivation* of level k+1 from (G T C) to ([] T' C'). Then, there exists an SLDNF refutation of A(D) ∪ T' ∪ {G}.

– Step A1 corresponds to a SLDNF resolution step with input set A(D).

– Step A21 corresponds to a SLDNF resolution step with input set T=T'.
In step A22, a ground event L_j could be included to T, and so, a consistency derivation of level k+1 is called from (C T∪{$L_j\sigma$} C) to ({} T' C'). We will get a new goal in the constructive derivation if exists this consistency derivation. As we have proved in c), the consistency derivation of level k+1 or below corresponds to the negation as failure of C with respect to the new input clause $L_j\sigma$. Previous failure of C with respect to T is not altered. Therefore, step A2 correspond to a SLDNF resolution step with input set T'.

– In step A3, it is checked that a consistency derivation of level k+1 exists from (←¬L_j T C) to ({} T' C'), that is, the failure of ¬L_j is checked. This corresponds to a negation as failure rule and, therefore, step A3 is a SLDNF resolution step.

[2] When G is a set of n>1 goals, the root of the tree is an implicit goal ←F with n descendant branches, one for each goal H_i in G.

Theorem 1: (*Soundness of our method*)

Let D be a deductive database, A(D) the Augmented Database and u an update request, such that u is not a logical consequence of comp(A(D)). Let T be a solution obtained by our method. Then, u∧¬ιIc is a logical consequence of comp(A(D) ∪ T).

Proof: Lemma 1 states that there is an SLDNF refutation of A(D) ∪ T ∪ {← u ∧ ¬ιIc} if exists a constructive derivation from (← u ∧ ¬ιIc ∅ ∅) to ([] T C).

By the soundness of the SLDNF resolution, the existence of the SLDNF refutation of A(D) ∪ T ∪ {← u ∧ ¬ιIc } ensures that u∧¬ιIc is a logical consequence of comp(A(D) ∪ T).

Appendix B. Completeness of our method

In this appendix, we prove the completeness of our method. We relate the completeness of our method to that of the SLDNF resolution. Let D be a deductive database, A(D) the augmented database, u an update request and T a minimal solution such that u and the integrity constraints are satisfied in the updated database. Assume that there is an SLDNF refutation of A(D) ∪ T ∪ {← u∧¬ιIc }. We prove in this section that there will be a constructive derivation from (←u∧¬ιIc ∅ ∅) to ([] T C).

We use the concept of rank of SLDNF refutation and rank of finitely failed SLDNF tree as defined in [Llo87].

Lemma 2: Let D be a deductive database; A(D) the augmented database; G and H goals; T, T' and T" transactions; C, C' and C" condition sets. Then the two following results hold:

a) If exists an SLDNF refutation of rank n of A(D) ∪ T ∪ {G} then:
   - for ∀T' such that T' ⊆ T and
   - for each set C' such that, for ∀C_i ∈ C', the SLDNF-tree for A(D) ∪ T ∪ {C_i} fails finitely and has rank n-1

   there is a constructive derivation from (G T' C') to ([] T" C") where:
   - T" ⊆ T and
   - for each condition C_i ∈ C", the SLDNF-tree for A(D) ∪ T" ∪ {C_i} fails finitely and has rank n-1.

b) If exists a finitely failed SLDNF-tree of rank n of A(D) ∪ T ∪ {H} then:
   - for ∀T' such that T' ⊆ T and
   - for each set C' such that, for ∀C_i ∈ C', the SLDNF-tree for A(D) ∪ T ∪ {C_i} fails finitely and has rank n-1

   there is a consistency derivation from ({H} T' C') to ({} T" C") where:
   - T" ⊆ T and
   - for each condition C_i ∈ C"- C', the SLDNF-tree for A(D) ∪ T ∪ {C_i} fails finitely and has rank n.

Proof: the proof is by induction over the rank n of the refutation or the finitely failed trees.

We associate to each SLDNF refutation step a constructive derivation step, and we prove that in any intermediate step we have (G_i T_i C_i) where both conditions of the lemma hold. Initially, G_i=G and T_i=T' and C_i=C'. Last step must reach the empty clause ([] T" C").

To each finitely failed SLDNF-tree derivation step we associate a consistency derivation step, and prove that in any intermediate step we have (F_i T_i C_i) where both conditions of the lemma hold. Notice that F_i is the set of goals of the nodes of the tree F_i = {H_i} ∪ F'_i. Initially, F_i = {H} and T_i=T' and C_i=C'. The last step of each branch of the tree must reach a failure.

n = 0

a) In this case, by the definition of an SLDNF refutation of rank 0, goal G can only contain positive literals and it is not required any finitely failed SLDNF-tree. Therefore, in this case, we have to prove that exists a constructive derivation from (G T' ∅) to ([] T" ∅) where T" ⊆ T.

Let L_j be the selected literal in the refutation. Depending on the type of this literal, we have:

- L_j is a positive literal
  - *L_j is not a base event*: Next goal is (S T_i ∅) and this step corresponds to a SLDNF resolution step. (A1)
  - *L_j is a base event*: Next goal is ($G_i \backslash L_j\sigma$ $T_i \cup \{L_j\sigma\}$ ∅). Notice that event $L_j\sigma$ belongs to T since we assume that there is a refutation of A(D) ∪ T ∪ {G}. Notice that it corresponds to a SLDNF resolution step with T as input set, and that, $T_i \cup \{L_j\sigma\} \subseteq T$. (A2)
- L_j is a negative literal

  This case is not applicable for n=0. (A3)

The derivation ends with the empty clause [].

b) As in the previous case, by the definition of an finitely failed SLDNF-tree of rank 0, goal H can only contain positive literals and it is not required any auxiliary finitely failed SLDNF-tree nor SLDNF refutation. Therefore, in this case, we have to prove that there is a consistency derivation from (F_i T' C') to ({} T" C") where SLDNF-trees of each $C_i \in$ C" fail finitely.

Let H_i be $\leftarrow L_1 \wedge ... \wedge L_k$, and let the selected literal be L_j. Depending on the type of literal L_j, we have:

- L_j is a positive literal
  - *L_j is not a base event*: Next goal is ($S' \cup F_i$' T_i C_i) and this step corresponds to a SLDNF resolution step with [] ∉ S'. (B1)
  - *L_j is a base event*: (B2)
    I. If $L_j \in T_i$ and it is ground, next goal is ($S' \cup F_i$' T_i C_i).
    In this case, S'={$H_i \backslash L_j$} with [] ∉ S', and it corresponds to a SLDNF resolution step with input set T_i.
    II. If $L_j \notin T_i$ and it is ground, next goal is (F_i' T_i $C_i \cup \{H_i\}$).
    In this case, S'=∅. SLDNF-tree for A(D) ∪ T ∪ {H_i} has rank 0 and must fail finitely: for events that belong to T_i it fails since it corresponds to the current branch; for new events that will be included to T, it also fails because it is ensured when we include them into T in an step A2.
    III. If $L_j \in T_i$, it is not ground and they unify by some substitution σ and that [] ∉ S', next goal is ($S' \cup F_i$' T_i $C_i \cup \{H_i\}$).
    In this case, S'={$H_i \backslash L_j\sigma$} with [] ∉ S'. Then, like case II, the SLDNF-tree for A(D) ∪ T ∪ {H_i} has rank 0 and fails finitely.
    IV. If $L_j \notin T_i$ and it is not ground, next goal is (F_i' T_i $C_i \cup \{H_i\}$)
    In this case, S'=∅. Then, like case II, the SLDNF-tree for A(D) ∪ T ∪ {H_i} has rank 0 and fails finitely.
- L_j is a negative literal

  This case is not applicable for n=0. (B3, B4, B5)

The derivation ends with {}.

n = n-1

Assume that the result holds for SLDNF refutations and finitely failed SLDNF-trees of rank n-1. That is, there is a constructive derivation from (G T' C') to ([] T" C") associated to the SLDNF refutation of rank n-1 of A(D) ∪ T ∪ {G}. Moreover, there is a consistency derivation from ({H} T' C') to ({} T" C") associated to the finitely failed SLDNF-tree of rank n-1 for A(D) ∪ T ∪ {H}.

n = n

We are going to prove that it also holds for SLDNF refutations and finitely failed SLDNF-trees of rank n.

a) By the definition of an SLDNF refutation of rank n, goal G contains positive and negative literals. Therefore, in this case, we have to prove that there is a constructive derivation from (G T' C') to ([] T" C") for any T" $\subseteq$ T where SLDNF-trees of each C$\in$C" fails finitely.

Let L_j be the selected literal in the refutation. Depending on the type of this literal, we have:

- L_j is a positive literal
  - *L_j is not a base event*: Next goal is (S T_i C_i) and this step corresponds to a SLDNF resolution step. (A1)
  - *L_j is a base event*: Next goal is: (A2)
    I. If $L_j\sigma \in T_i$, next goal is ($G_i \backslash L_j\sigma$ T_i C_i).
    This case corresponds to a SLDNF resolution step with input set T_i.
    II. If $L_j\sigma \notin T_i$, next goal is ($G_i \backslash L_j\sigma$ T' C')
    In this case, for each C$\in C_i$ it is checked the existence of the consistency derivation of (C $T_i \cup \{L_j\sigma\}$ C_i).
    We assume that for each C$\in C_i$, the SLDNF-tree for A(D) $\cup$ T $\cup$ {C} fails finitely and have rank n-1. Therefore, by induction, there are consistency derivations from (C $T_i \cup \{L_j\sigma\}$ C_i) to ({} T' C') with T' $\subseteq$ T and SLDNF-trees for A(D) $\cup$ T $\cup$ {C} with C$\in$C'-C_i have rank n-1 and fail finitely.
- L_j is a negative literal (A3)
  Next goal is ($G_i \backslash L_j$ T' C') if there is the consistency derivation from ($\{\leftarrow\neg L_j\}$ T_i C_i) to ({} T' C'). The SLDNF-tree for A(D) $\cup$ T $\cup$ $\{\leftarrow\neg L_j\}$ has rank n-1 and fails finitely. Therefore, by induction, exists this consistency derivation.

The derivation ends with the empty clause [].

b) We assume that there is a finitely failed SLDNF-tree of rank n for A(D) $\cup$ T $\cup$ {H}. We have to prove that exists a consistency derivation from ({H} T' C') to ({} T" C") where T" $\subseteq$ T and SLDNF-trees of each C$\in$C" fails finitely.

Let H_i be $\leftarrow L_1 \wedge ... \wedge L_k$, and let the selected literal be L_j. Depending on the type of literal L_j, we have:

- L_j is a positive literal
  - *L_j is not a base event*: Next goal is (S'$\cup F_i$' T_i C_i) and this step corresponds to a SLDNF resolution step with []$\notin$S'. (B1)
  - *L_j is a base event*: (B2)
    I. If $L_j \in T_i$ and it is ground, next goal is (S'$\cup F_i$' T_i C_i).
    In this case, S'=$\{H_i \backslash L_j\}$ with []$\notin$S', and it corresponds to a SLDNF resolution step with input set T_i.
    II. If $L_j \notin T_i$ and it is ground, next goal is (F_i' T_i $C_i \cup \{H_i\}$).
    In this case, S'=$\emptyset$. SLDNF-tree for A(D) $\cup$ T $\cup$ $\{H_i\}$ has rank n and must fail finitely: for events that belong to T_i it fails since it corresponds to the current branch; for new events that will be included to T, it also fails because it is ensured when we include them into T in an step A2.
    III. If $L_j \in T_i$, it is not ground and they unify by some substitution σ and that []$\notin$S', next goal is (S'$\cup F_i$' T_i $C_i \cup \{H_i\}$).
    In this case, S'=$\{H_i \backslash L_j\sigma\}$ with []$\notin$S'. Then, like case II, the SLDNF-tree for A(D) $\cup$ T $\cup$ $\{H_i\}$ has rank n and fails finitely.
    IV. If $L_j \notin T_i$ and it is not ground, next goal is (F_i' T_i $C_i \cup \{H_i\}$).
    In this case, S'=$\emptyset$. Then, like case II, the SLDNF-tree for A(D) $\cup$ T $\cup$ $\{H_i\}$ has rank n and fails finitely.
- L_j is a negative literal
  - *L_j is not a base event*:

I. If there is the consistency derivation from ($\{\leftarrow\neg L_j\}$ T_i C_i) to ($\{\}$ T' C') and k>1 then, next goal is ($\{H_i \backslash L_j\} \cup F_i$' T' C'). (B3)
By the definition of finitely failed SLDNF-tree of rank n, the SLDNF-tree for A(D) $\cup$ T $\cup$ $\{\leftarrow\neg L_j\}$ has rank n-1 and fails finitely. Therefore, by induction, the consistency derivation exists. (B3)

II. If exists the constructive derivation from ($\leftarrow\neg L_j$ T_i C_i) to ([] T' C') then, next goal is (F_i' T' C'). (B5)
By the definition of finitely failed SLDNF-tree of rank n, there is a SLDNF refutation for A(D) $\cup$ T $\cup$ $\{\leftarrow\neg L_j\}$ with rank n-1. Therefore, by induction, the consistency derivation exists.

– *L_j is a base event*:

I. If $\neg L_j \notin T$ and k>1 then, next goal is ($\{H_i \backslash L_j\} \cup F_i$' T_i C_i). (B4)
This step corresponds to a SLDNF resolution step with input set T_i.

II. If $\neg L_j \in T$ then, next goal is (F_i' T_i C_i). (B5)
The existence of the constructive derivation from ($\leftarrow\neg L_j$ T_i C_i) to ([] T_i C_i) is ensured, since it corresponds to a SLDNF resolution step with input set T_i.

The derivation ends with {}.

Lemma 3: Let D be a deductive database; A(D) the Augmented Database; u an update request and T a minimal solution. All refutations (main and auxiliaries) appearing in the SLDNF search space of A(D) $\cup$ T $\cup$ $\{\leftarrow u \wedge \neg\iota Ic\}$ are reached by the constructive derivation from ($\{\leftarrow u \wedge \neg\iota Ic\}$ $\varnothing$ $\varnothing$) to ([] T C).

Proof: We are going to prove the Lemma by contradiction. Let assume that there is an SLDNF refutation for A(D) $\cup$ T $\cup$ $\{\leftarrow G_s\}$ not reached by the constructive derivation from ($\{\leftarrow u \wedge \neg\iota Ic\}$ $\varnothing$ $\varnothing$) to ([] T C).

The above SLDNF refutation, can not be the main refutation. By soundness of the method, to the constructive derivation from ($\{\leftarrow u \wedge \neg\iota Ic\}$ $\varnothing$ $\varnothing$) to ([] T C) it corresponds a SLDNF refutation for A(D) $\cup$ T $\cup$ $\{\leftarrow u \wedge \neg\iota Ic\}$. Therefore, it must be an auxiliary SLDNF refutation.

This auxiliary SLDNF refutation is called by some finitely failed SLDNF-tree for A(D) $\cup$ T $\cup$ $\{H_k\}$. By Lemma 2, there must be a consistency derivation from ($\{H_k\}$ T_k C_k) to ($\{\}$ T' C') that it corresponds to this tree.

If this consistency derivation does not reach goal G_s, it is because in some step of the derivation, a rule of the method has failed the current branch for some selected literal L_j.

Rules that fail the current branch of the consistency derivation are:

(B1) Current branch fails when S'=$\varnothing$. It corresponds to a SLDNF resolution step that does not require any auxiliary derivation.

(B2) Current branch fails if base event $L_j \notin T_k$. This step corresponds to an SLDNF resolution step with input set T_k that fails. Assume that base event fact L_j will be included later into set T-T_k (in a constructive derivation). Then, it is required to continue the consistency derivation to ensure falseness of $\leftarrow H_k$. If this consistency derivation also fails, then sets T and T-$\{L_j\}$ will correspond to different solutions of U. This fact contradicts that T is a minimal solution, and so, such constructive derivation will not be called later.

(B3) This step does not fail the current branch.

(B4) This step does not fail the current branch.

(B5) If there is a constructive derivation from ($\leftarrow\neg L_j$ T_k C_k) to ([] T' C'), by soundness of the method, there is a SLDNF refutation of A(D) $\cup$ T $\cup$ $\{\leftarrow\neg L_j\}$ that can not be A(D) $\cup$ T $\cup$ $\{\leftarrow G_s\}$.

There is not possible to have an SLDNF refutation for A(D) $\cup$ T $\cup$ $\{\leftarrow G_s\}$ not reached by a some main or auxiliary constructive derivation from ($\{\leftarrow u \wedge \neg\iota Ic\}$ $\varnothing$ $\varnothing$) to ([] T C). Therefore, all refutations appearing in the SLDNF search space of A(D) $\cup$ T $\cup$ $\{\leftarrow u \wedge \neg\iota Ic\}$ are reached by the constructive derivation from ($\{\leftarrow u \wedge \neg\iota Ic\}$ $\varnothing$ $\varnothing$) to ([] T C), which proves the Lemma.

Lemma 4: Let D be a deductive database; A(D) the augmented database; u an update request and T a minimal solution. If exists an SLDNF refutation of A(D) ∪ T ∪ {← u∧¬ιIc } then, for each event t ∈ T, t is used in the main refutation or in an auxiliary refutation of A(D) ∪ T ∪ {← u∧¬ιIc }.

Proof: Let assume that t is only used into steps of finitely failed auxiliary trees. In this case, these trees also fail, and set T-{t} becomes a solution. It contradicts that T is a minimal solution and, therefore, all events of T are must be used in some (main or auxiliary) refutation of A(D) ∪ T ∪ {← u∧¬ιIc }.

Theorem 2: Let D be a deductive database; A(D) the augmented database; u an update request and T a minimal solution. If exists an SLDNF refutation of A(D) ∪ T ∪ {← u∧¬ιIc } then, a constructive derivation from ({←u∧¬ιIc } ∅ ∅) to ([] T C) exists.

Proof: Lemma 2 ensures that if there is an SLDNF refutation of A(D) ∪ T ∪ {← u∧¬ιIc } then, a constructive derivation from ({←u∧¬ιIc } ∅ ∅) to ([] T' C') with T' ⊆ T exists. By lemma 3, this constructive derivation reaches the main and all auxiliary refutations of the search space of A(D) ∪ T ∪ {← u∧¬ιIc }. By Lemma 4, all events of T' are used in some of these refutations. Since T is a minimal solution and T' ⊆ T solution T' must includes all events of T, that is, T'=T. Therefore, it is proved that exists the constructive derivation from ({←u∧¬ιIc } ∅ ∅) to ([] T C).

Theorem 3: (*Completeness of our method*)

Let D be a deductive database; A(D) the augmented database; u an update request and T a minimal solution. Since SLDNF resolution is complete for A(D) ∪ T and goal {← u∧¬ιIc}, for any transaction T such that u∧¬ιIc is a logical consequence of comp(A(D) ∪ T), a constructive derivation from ({←u∧¬ιIc } ∅ ∅) to ([] T C) exists.

Proof: It follows from the completeness of the SLDNF resolution that if u∧¬ιIc is a logical consequence of comp(A(D) ∪ T) then there is an SLDNF refutation of A(D) ∪ T ∪ {← u∧¬ιIc }. Therefore, by Theorem 2, it exists a constructive derivation from ({←u∧¬ιIc } ∅ ∅) to ([] T C).

Appendix C. Cycle Termination

In this appendix, we prove that, assuming finite domains, cycles between nodes in the Precedence Graph do not correspond to a loop at execution time. The first Lemma states that an already repaired condition could be violated again with the same substitution a finite number of times. Therefore, cycles between conditions could loop, but always terminate.

Lemma 6.1: Assuming finite domains and therefore, assume a finite number of substitutions α_i. Any condition C could be violated with a substitution α_i, be repaired, and become violated again with the same substitution α_i a finite number of times.

Proof: A condition is a goal of the form $\leftarrow L_1 \wedge \ldots \wedge L_n$. It is defined by a finite number of literals L_i some of them correspond to positive events or negative events. We prove the lemma by considering two situations:

A) Assume that all events that appear in the definition of C are base events. Therefore, this condition has a finite number of potential violators (positive events) and a finite number of potential repairs (negative events).
   In this case, a violation with a substitution α_i of condition C could be induced when transaction T contains a substitution α_i of each positive event of C. To repair it we require to include into transaction T a substitution α_i of any potential repair of C.
   In this case, condition C can not be violated again with the same substitution α_i. The only possible way to violate again condition C is with a new substitution $\alpha_k \neq \alpha_i$.

B) Assume now that in the body of condition C it could appear some derived event. In this case, the number of potential violators and potential repairs is also finite, and they can be obtained by means of the Dependency Graph of Events.
   To violate condition C (with substitution α_i) is necessary that transaction T contains a set of base event facts (with substitution α_i) that induces all positive base/derived events that appear in the definition of C. To repair it, it is only necessary to include into T, a set of base events facts (with substitution α_i) that allows to induce some negated event of C, or that dismisses the induction of some positive event of the C.
   Condition C could be violated again with substitution α_i. It is only necessary to include into transaction T a set of base event facts that dismiss the above repair. That is, a set of events that prevents the induced negative event or that induces the dismissed positive derived event. In any case, since the number of potential repair and potential violators of C is finite then, the times that condition C can be violated with the same substitution α_i is also finite. Obviously, this condition can be violated again with a different substitution $\alpha_k \neq \alpha_i$.

Theorem 6.1: Given a transaction T and a set of precedence relationships that define a cycle in the (Active) Precedence Graph, the process of consistency maintenance of the associated conditions always finishes.

Proof: Let assume that the process of consistency maintenance loops infinitely. Therefore, it is necessary that some condition of the cycle must be violated by T an infinite number of times. This situation is contradictory with respect to Lemma 6.1 since a condition can be violated only a finite number of times with the same substitution. If the number of substitutions is finite, the consistency maintenance process always terminates.

Appendix D. Algorithms

In this appendix, we propose two algorithms: the first one implements part of the Consistency Maintenance Module of the architecture (Consistency_Maintenance(T,A,C)). The second one implements the Translation Step of the View Updating Module (Translate_update(u,A(D),STC)).

Consistency_Maintenance(T, A, C):SS

This function implements the consistency maintenance process by using the information of the Active Precedence Graph A. Parameter T corresponds to the initial transaction. Parameter C is necessary to manage the recursive definition of the function, and it corresponds to a set of conditions.

The algorithm obtains the set (SS) of all alternative ways to satisfy all conditions of the Active Precedence Graph A. If no solution exists, the output will be the empty set.

To obtain all alternative solutions to an update request, the algorithm store in a list (Pending) all alternative ways to repair conditions of a node. Specifically, each element of the list contains the current transaction T (extended with the repair), the current state of the Active Precedence Graph (with new active nodes and new marks) and the set of conditions C. After rejecting a transaction or after obtaining a solution, a new alternative is explored obtaining an element of the list (Pending).

Several functions are used in the definition of the Consistency_Maintenance(T,APG,C) algorithm. These functions are the following:

Obtain_transaction(Pending,T,C,A): it obtains, from the list Pending, a new transaction T to be processed.

Append(Pending,[(T,C,A)]) it stores an alternative transaction T (partially processed) with its Active Precedence Graph A and the set of conditions C.

Select_next_node(A): it selects next node to be visited of the Active Precedence Graph A

Check_node(Node,T): it checks if transaction T violates some condition of Node.

Checking_condition(Node): it returns true when condition of Node is a Checking Condition, false when it is a Generation Condition.

Obtain_all_repair_requests(Node,T): it returns the set of all possible requests to repair conditions of Node. All repairs (partial translations) to each request are obtained. In this process, a View_Updating procedure could be necessary.

Activate_and_mark_nodes(A,TC): given a partial translation TC=(T,C), it activates new nodes of the Active Precedence Graph A and marks successors of current Node.

To_activate(A,C): it selects those conditions of set C whose associated node belongs to the Precedence Graph A. This function is used in presence of subgraphs to determine which conditions of C must be activated at each recursion level.

```
Function Consistency_Maintenance(T,APG,C):SS
  /* input: a transaction T, an Active Precedence Graph APG and the set of conditions C */
  /* output: set of pairs (transaction Ts, condition_set Cs) that Ts satisfies conditions of APG */
SS := ∅;
Pending := [(T, C, APG)];
while Pending ≠ [] do
    Obtain_transaction(Pending,T,C,APG);
    while APG is marked do
        Node := Select_Next_Node(APG);
        if Node is not a subgraph then
            Unmark(Node,APG);
            violated := Ckeck_Node(Node,T);
            if violated then
                if Checking_cond(Node) then
                    Obtain_transaction(Pending,T,C,APG);
                else    /* Generation condition */
                    RepReq := Obtain_all_repair_requests(Node,T);
                    for each Rr_i ∈ RepReq do
                        if Rr_i contain derived fact updates then
                            View_Updating(Rr_i, TC);
                        else    TC := {(Rr_i, ∅)}
                        end if;
                        for each (R_i,C_i) ∈ TC do
                          APG' := Activate_and_mark_nodes(APG,TC);
                          T' := T ∪ R_i; C' := C ∪ C_i;
                          Append(Pending, [(T',C',APG')]);
                        end for each;
                    end for each;
                    Obtain_transaction(Pending,T,C,APG);
                end if;
            end if;
        else /* Node subgraph */
            Unmark(Node,APG);
            SS_g := Consistency_Maintenance(T,Node,C);
            for each S_g ∈ SS_g do                  /* S_g = (T_g, C_g) */
                R := T_g - T;
                C_a := To_activate(APG,C ∪ C_g);
                S_gr := (R, C_a);
                APG' := Activate_and_mark_nodes(APG,S_gr);
                T' := T_g;   C' := C ∪ C_g - Ca;
                Append(Pending, [(T',C',APG')]);
            end for each;
            Obtain_transaction(Pending,T,C,APG);
        end if;
    end while;
SS := SS ∪ {(T, C)}
end while;
Return(SS);
```

Translate_Update(u, A(D), STC)

This algorithm implements the process of translating an update request u into the set STC of all possible partial translations. Each one consists on a transaction T and a set of conditions C. In this process, only the specialized event rules of the A(D) are considered.

```
Algorithm Translate_Update (u, A(D), STC);
    /* input: an update request u and the specialized Augmented Database A(D)*/
    /* output: set of partial translations STC = {(Ti, Ci)} of transaction Ti and condition set Ci */

Unfold (u, A(D), DNF);
for each Disjunct of DNF do
    for each Ei ∈ Disjunct do
        if positive(Ei) then    T := T ∪ {Ei};
        else
            if base_event (Ei) then C := C ∪ {← Ei};
            else
                Get_event_rules (Ei, A(D), EvR);
                for each Ci ∈ EvR do
                    C := C ∪ {← Ci };
                end for each;
            end if;
        end if;
    end for each;
    STC := STC ∪ { (T, C) };
end for each;
Return (STC);
```

Procedure Unfold(u, A(D), DNF) unfolds the update request u into a goal (DNF) in disjunctive normal form by considering the specialized event rules of A(D). Each resulting disjunct D_i of DNF defines a possible way of translating u. For each D_i, base event facts are included in T[21], while negative literals are included as conditions into set C. In the case of a negative derived events or auxiliary/transition predicates E_i, we have a different condition for each possible specialized rule that defines E_i.

[21] If the event is not ground, we could have a transaction for each possible way to instance it.