Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

303

J. W. Schmidt S. Ceri M. Missikoff (Eds.)

Advances in Database Technology – EDBT '88

International Conference on Extending Database Technology
Venice, Italy, March 14–18, 1988
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

Editors

Joachim W. Schmidt
Fachbereich Informatik, Johann Wolfgang Goethe-Universität
Postfach 11 19 32, D-6000 Frankfurt am Main 11, FRG

Stefano Ceri
Dipartimento di Matematica, Universitá di Modena
Via Campi 213/B, I-41100 Modena, Italy

Michele Missikoff
IASI-CNR
Viale Manzoni 30, I-00185 Rome, Italy

CR Subject Classification (1987): D.3.3, E.2, F.4.1, H.2, I.2.1, I.2.4

ISBN 3-540-19074-0 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-19074-0 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210

Contents

Invited Paper

Databases and Logic

Expert System Approaches to Databases

Distributed Databases and Transaction Management

Database Administration

Complex Database Objects

Efficient Data Access

Efficiency by Replicated Data

Data Types and Data Semantics

Special Data

Short Project Papers

Support for Data- and Knowledge-Based Applications

Distributed Database Applications

A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases

François Bry, Hendrik Decker and Rainer Manthey
ECRC, Arabellastr. 17, D - 8000 München 81, West Germany

ABSTRACT *Integrity maintenance methods have been defined for preventing updates from violating integrity constraints. Depending on the update, the full check for constraint satisfaction is reduced to checking certain instances of some relevant constraints only. In the first part of the paper new ideas are proposed for enhancing the efficiency of such a method. The second part is devoted to checking constraint satisfiability, i.e., whether a database exists in which all constraints are simultaneously satisfied. A satisfiability checking method is presented that employs integrity maintenance techniques. Simple Prolog programs are given that serve both as specifications as well as a basis for an efficient implementation.*

1. Introduction

Integrity maintenance methods are intended to guarantee that all integrity constraints remain satisfied after an update, provided they have been satisfied before. In general, not all constraints are relevant to an update but only certain instances of some of them. This depends on which relations are updated. It is sufficient to check whether relevant instances are satisfied in order to guarantee satisfaction of the full constraint set after the update. Since this basic principle for efficient constraint checking was first described in [NICO 79] and [BLAU 81], several authors have proposed extensions for the deductive case, e.g., [DECK 86], [LLOY 86], [KOWA 87] and [LING 87]. In the first part of this paper we present a method that is based on principles common to all proposals mentioned, but introduces several new ideas. We propose to perform the computation of relevant constraint instances independently from any access to the fact base. Fact access is entirely delayed to the evaluation phase and may thus benefit from optimization steps performed during query evaluation. Furthermore we propose to simulate evaluation of constraints over the updated database by means of a simple meta-interpreter. This approach permits to handle recursive rules, provided the database query-answering system has this capacity (e.g., [VIEI 87]).

Apart from preventing constraint violations caused by fact or rule updates, one has to detect inconsistencies when updating the constraint set as well. If a newly introduced constraint is not satisfied in the current database, one can try to enforce it by means of further updates to the factual part of the database. However, any attempt to do so will fail, if the new constraint is not compatible with the already existing ones. Such situations can be characterized by the logical concept of finite satisfiability. A set of formulas is finitely satisfiable if there is at least one finite model that satisfies all formulas in the set. Formulas that are not finitely satisfiable either have no model at all, or all models are infinite and thus not suitable for database purposes. In presence of deduction rules, these logical deficiencies may be due to inherent contradictions between rules and constraints as well. Thus constraint violations observed after a rule update possibly indicate that constraints and rules are no longer finitely satisfiable after the modification.

In contrast to constraint satisfaction which is a decidable property, finite satisfiability of constraints is only semi-decidable, i.e., every algorithm for checking this property may run forever if applied to constraint sets that contain certain "axioms of infinity". In [BRY 86] we have discussed this problem in more detail, and have investigated various possible approaches to it. In this paper we propose a method for checking constraint satisfiability that is closely related to the way constraint satisfaction is handled. The method is based on a proof procedure that we have recently presented to the theorem proving community as well [MANT 87a, MANT 87b]. If applied to a given set of rules and constraints, the method systematically tries to construct a finite set of facts such that all constraints are satisfied in the resulting database. If the procedure succeeds in doing so, a finite model of rules and constraints has been found and finite satisfiability has been demonstrated. The construction process can be viewed as a sequence of successive updates, each of them possibly causing constraint violations that can be efficiently checked by means of the techniques mentioned above. The violated constraint instances determined this way are used for deriving the next updates necessary to enforce the violated instance. Only few authors have till now been concerned with constraint satisfiability. In [KUNG 84] a method is proposed that relies on the same basic principle as ours, but is not complete for finite satisfiability and considerably less efficient. The approach of [LASS 87] is efficiently applicable for propositional rules only.

Besides introducing methods for checking both properties, constraint satisfaction as well as constraint satisfiability, we would like to show that Prolog is a very convenient programming language for the implementation of these methods. We therefore include short Prolog programs in the paper, that on the one hand serve as specifications, on the other hand can be efficiently applied in practice. This is particularly important as several Prolog-DBMS couplings are now available (e.g., [BOCC 86]) that allow to use Prolog for database querying as well.

2. Definitions

A deductive database D consists of three finite sets: a set F of facts, a set R of rules, and a set I of integrity constraints. A fact is a ground atom. A rule is an expression $H \leftarrow B$, where the head H is a positive literal and the body B is a literal or a conjunction of (positive or negative) literals. The only terms occurring in a rule are constants and variables. We assume every rule to be range-restricted, i.e., every variable occurring in H, or in a negative literal in B occurs in a positive literal in B as well.

Constraints are function-free, closed first-order formulas with restricted quantification, i.e., quantified (sub)formulas have one of the forms

$$\exists X_1 ... X_n \, [A_1 \wedge ... \wedge A_m \wedge Q]$$

$$\forall X_1 ... X_n \, [\neg A_1 \vee ... \vee \neg A_m \vee Q]$$

where A_1, ..., A_m are atoms such that every variable X_i occurs in at least one A_j, and where Q is either **true** or **false**, or some formula in which some or all X_i are free. In the Prolog programs quantified formulas are represented as

```
exists([X1,...,Xn], (A1 and...and Am),Q)
forall([X1,...,Xn], (A1 and...and Am),Q)
```

assuming that `and` and `or` have been declared as Prolog infix operators. Furthermore we assume that integrity constraints are expressed in the following normalized form:

- Formulas are rectified, i.e., no two quantifiers in a formula introduce a same variable.
- The scope of each quantifier is reduced as much as possible (miniscope form).
- Implications and equivalences are expressed by means of logical connectives $\wedge$, $\vee$, and $\neg$, and negations occur in front of atoms only (negation normal form).
- $\forall$ is distributed over $\wedge$.

These forms are assumed for obtaining more concise definitions throughout the paper. Negation normal form, e.g., allows to speak directly about complementary literals instead of having to use polarities, and in miniscope form governing relationships between variables and scopes of quantifiers coincide. As far as the expressive power is concerned, neither the restricted quantification form nor the above normalization impose significant restrictions. Note in particular that an expression in relational calculus corresponds to a formula with restricted quantification.

The semantics of integrity constraints - as of queries in general - are defined according to a canonical interpretation in which the true atoms are exactly those that are explicit in F or derivable from F and R. In order to be able to uniquely determine the canonical interpretation, we restrict R to be stratified in the sense of [APT 87]. Constraints are satisfied in D if they are satisfied in the canonical interpretation associated with $F \cup R$.

3. Integrity Maintenance

Let single-fact updates be represented by literals, a positive literal indicating insertion, a negative literal indicating deletion. Throughout this chapter, let U denote a ground single-fact update to a database D and let U(D) denote the updated database.

Definition 1:

If U is a positive literal explicit in D, the updated database U(D) is identical with D. If U is a positive literal not explicit in D, U(D) is D augmented with U.

If U is a negative literal $\neg A$ and if A is explicit in D, then U(D) is D without the fact A. If U is a negative literal $\neg A$ and if A is not explicit in D, then U(D) is identical with D.

The truth value of certain formulas - like, e.g., $\forall X\ p(X)$ or $\exists X\ \neg p(X)$ - depends on the database domain as a whole. Evaluation of such formulas therefore requires that the domain is explicitly stored or computed. This can be extremely inefficient. In order to avoid this problem, the class of domain independent or definite formulas [KUHN 67] has been proposed: A formula C is domain independent if and only if its truth value does not depend on any domain element other than those occurring in the relations that are explicitly mentioned in C. For the efficiency of integrity maintenance methods, it is very desirable that all constraints are domain independent. This permits to evaluate only those constraints in which updated relations occur. Formulas with restricted quantifications are domain independent.

Definition 2:

A constraint C is relevant to an update U iff the complement of U is unifiable with a literal in C.

For constraints with restricted quantifications it is even sufficient to evaluate only certain simplified instances of constraints relevant to an update, in order to prove that these constraints are satisfied in the updated database.

Definition 3:

Let C be an integrity constraint relevant to U. A simplified instance of C is obtained as follows:
Let L denote a literal in C unifiable with the complement of U. Let σ denote a most general unifier (mgu) of L and U, and let τ denote the restriction of σ to those universally quantified variables that are not governed by an existentially quantified variable.

a. partially instantiate C by applying τ
b. simplify the partial instance $C\tau$ by
   - dropping quantifiers for variables grounded by τ
   - replacing $L\tau$ by **false**, in case $L\tau$ is identical with the complement of U, and eventually applying absorption laws (like, e.g., **false** $\vee F \equiv F$)

τ is called the defining substitution of the simplified instance.

Consider the integrity constraint:

$$C_1: \forall X\, [\neg p(X) \vee q(X)]$$

The simplified instance of C_1 associated with the update p(a) is q(a). It is indeed sufficient to evaluate q(a) in order to ensure that C_1 remains satisfied in the updated database. The simplified instance of

$$C_2: \forall XY\ \neg p(X,Y) \vee [\exists Z\, q(X,Z) \wedge \neg s(Y,Z,a)]$$

associated with the update $\neg q(c_1,c_2)$ is

$$\forall Y\ \neg p(c_1,Y) \vee [\exists Z\, q(c_1,Z) \wedge \neg s(Y,Z,a)]$$

The defining substitution binds X to c_1. Instances of C_1 binding X to anyother constant are not affected by the considered update. Note that the existentially quantified variable Z must remain unbound in the simplified instance. Several examples are discussed in [NICO 79] in which this technique was first described.

More than one simplified instance can be obtained from a same integrity constraint. This happens when the complement of U is unifiable with more than one literal in the constraint.

3.1. Relational Databases

Integrity maintenance for relational databases (i.e., databases without deduction rules) is based on the following result:

Proposition 1: [NICO 79]
All constraints are satisfied in U(D) iff they are satisfied in D and every simplified instance of a constraint relevant to U is satisfied in U(D).

In Prolog, the integrity maintenance principle stated in this proposition can be easily implemented as follows. Assume constraints to be stored as Prolog facts `integrity_constraint(Id,C,V)`, where `C` is the constraint, `Id` is its unique identifier and `V` the list of universally quantified variables in `C` that are not governed by an existential one. Furthermore assume that for every literal `L` in a constraint `C` a fact `relevant(Id,L)` has been precomputed, where `Id` denotes the identifier associated with `C`. Simplified instances of constraints that are relevant to `U` can be generated through backtracking by means of:

```
simplified_instance(U,SI) :-
   relevant(Id,U),
   integrity_constraint(Id,C1,V),
   complement(U,UC)
   instantiate(C1,UC),
   integrity_constraint(Id,C2,V),
   simplify(C2,UC,SI).
```

```
complement(not A,A) :- !.
complement(A, not A).

instantiate(forall(_,F1,F2),UC) :-
   !, (complement(UC,U), instantiate(F1,U) ; instantiate(F2,UC)).
instantiate(exists(_,F1,F2),UC) :-
   !, (instantiate(F1,UC) ; instantiate(F2,UC)).
instantiate(F1 and F2,UC) :-
   !, (instantiate(F1,UC) ; instantiate(F2,UC)).
instantiate(F1 or F2,UC) :-
   !, (instantiate(F1,UC) ; instantiate(F2,UC)).
instantiate(L,L).
```

Code for the predicate `simplify` is not given here because it is simple but unsubstantial. The partial instantiation is obtained as follows: Calling `instantiate(C1,UC)` instantiates all variables of `C1`, particularly those in `V`. Since variables in `V` are bound, the subsequent call `integrity_constraint(Id,C2,V)` returns the desired partial instance `C2` respecting the bindings given to `V` before. The set `S` of instances to be evaluated over the updated database is obtained by calling `setof(SI,simplified_instance(U,SI),S)`.

3.2. Deductive Databases: Principles

In presence of deduction rules, an explicit update may induce further logical changes of the database. Induced updates correspond to facts that are either true after the update but not before, or false after the update but true before. They can be characterized as follows:

Definition 4:

Let L denote a ground literal and A a ground atom.

A ($\neg$A, resp.) is directly induced by L over U(D) iff

- there is a deduction rule A' $\leftarrow$ B such that B contains a literal L' unifiable with L (the complement of L, resp.); let τ denote a mgu of L and L'
- A = (A'τ)σ,
  where σ is an answer substitution returned by evaluating (B\L')τ in U(D)
  (B\L' denotes B without L', **true** if B = L')
- A ($\neg$A, resp.) evaluates to false (true, resp.) in D (in U(D), resp.)

A literal is induced by L over U(D) iff it is directly induced by L over U(D) or by a literal induced by L over U(D). Every literal induced by U over U(D) is an update induced by U.

Proposition 1 can be extended to deductive databases by considering all constraints relevant to induced updates too:

Proposition 2:

All constraints are satisfied in U(D) iff they are satisfied in D and every simplified instance of a constraint relevant to U or relevant to an update induced by U is satisfied in U(D).

[**Proof:** (sketched) The property follows easily from Proposition 1 by reduction to the relational case. Consider the canonical interpretation of D as a relational database. Treat the induced updates as explicit updates to this database.]

If integrity maintenance is straightforwardly performed according to Proposition 2 the following would be done: Induced updates are successively computed. As soon as a constraint is relevant to such an update, the corresponding simplified instance is evaluated in the updated database. The methods described in [DECK 86] and [KOWA 87] are of this kind. This approach suffers from two drawbacks. First, all induced updates are computed, even those for which no constraint is relevant. This is for example the case with an update p(a,b) in presence of the deduction rule $r(X) \leftarrow q(X,Y) \wedge p(Y,Z)$ if the predicate r does not occur positively in any constraint. The overhead is considerable if there are a lot of q(X,a)-facts. Second, evaluating all simplified instances independently of each other prevents from applying certain optimizations that a global evaluation would permit. Especially the detection of redundant subqueries can be very useful in this context. Consider for example the following constraint:

$$C_1: \forall X\, [\neg \text{student}(X) \vee \neg \text{enrolled}(X,\text{cs}) \vee \text{attends}(X,\text{ddb})]$$

Assume that there is a deduction rule $\text{enrolled}(X,\text{cs}) \leftarrow \text{student}(X)$ expressing that all students are enrolled in computer science. The update student(jack) yields the following simplified instance

$$S_1: \neg \text{enrolled}(\text{jack},\text{cs}) \vee \text{attends}(\text{jack},\text{ddb})$$

The induced update enrolled(jack,cs) leads to a simplified instance

$$S_2: \neg \text{student}(\text{jack}) \vee \text{attends}(\text{jack},\text{ddb})$$

Evaluating S_1 and S_2 independently requires to evaluate the subquery attends(jack,ddb) twice. A global evaluation, however, could be expected to avoid this redundancy when simultaneously evaluating both instances. Such redundancies, although a bit artificial in this simple example, appear rather frequently in case of transactions consisting of more than one single-fact update.

Instead of applying Proposition 2 straightforwardly, we propose an alternative approach that does not exhibit the above-mentioned drawbacks. This approach does not interleave generation of induced updates and evaluation of simplified instances, but clearly separates two phases: a preparatory one, that does not access the base of facts, and a pure evaluation phase. In the first phase, potential updates are computed, that represent possible ground induced updates. From potential updates and constraints, expressions called update constraints are generated. In a second stage, all update constraints are evaluated. Facts are accessed only during evaluation.

Definition 5:

Let L be a literal and A an atom (both not necessarily ground).

A ($\neg$A, resp.) directly depends on L iff

- there is a deduction rule A' $\leftarrow$ B such that B contains a literal L' unifiable with L (the complement of L, resp.)
- A = A'τ, where τ is a mgu of L' and L (the complement of L, resp.)

A depends on L if and only if A directly depends on L or on a literal that depends on L. Every literal which depends on U is a potential update induced by U.

Every induced update is an instance of a potential update, depending on the facts stored in the database. If for example the database contains a deduction rule r(X) $\leftarrow$ p(X,Y)$\wedge$q(Y,Z) then r(X) ($\neg$r(X), resp.) is a potential update induced by q(a,b) ($\neg$q(a,b), resp.). Note that potential updates are defined without considering any answer substitution, as opposed to induced updates. There may be potential updates no instance of which is an induced update.

Let 'delta' denote a meta-predicate such that delta(U,L) holds if and only if L is satisfied in U(D), but not in D. Similarly, let new(U,F) denote the evaluation of the formula F over the updated database U(D).

Definition 6:

For every constraint C relevant to a literal L the universal closure of the formula $\neg$delta(U,Lτ) $\vee$ new(U,s(C)) is an update constraint for L, where s(C) denotes a simplified instance of C wrt L with defining substitution τ.

Update constraints can be used for integrity maintenance on the basis of the following result:

Proposition 3:

All constraints are satisfied in U(D) iff they are satisfied in D and every update constraint for U or for a potential update induced by U is satisfied in U(D).

[**Proof**: (sketched) By Definitions 3 and 4, all induced updates are instances of potential induced updates. From this remark and from the definition of update constraints, it follows that a simplified instance of a constraint relevant to an induced update is necessarily the right hand side of an instance of an update constraint, the left hand side of which is an induced update. The proposition follows.]

A concept similar to that of a potential update can be found in [LLOY 86]. However, the method proposed in that article does not distinguish between 'new' and 'delta'. Instead of evaluating expressions of the form $\neg$delta(U,L) $\vee$ new(U,s(C)), they evaluate formulas corresponding to $\neg$new(U,L) $\vee$ new(U,s(C)), in our terminology. The resulting loss in efficiency is often considerable. The method described here for single-fact updates has been defined for more general updates, such as transactions and conditional updates [BRY 87]. Rule updates can be treated like conditional updates. However, when defining induced or potential updates one has to respect modifications to the rule set as well.

3.3. Deductive Databases: Implementation

3.3.1. *Computation of update constraints*

According to Definition 5, assume that for every deduction rule A←B and every literal L in B the facts `directly_dependent(L,A,R)` and `directly_dependent(LC,not A,R)` have been computed, where LC is the complement of L and R denotes B\L. The 'dependent' relationship between literals can be represented in terms of `directly_dependent` through

```
dependent(L,U) :-
   directly_dependent(L,U,_).
dependent(L,U) :-
   directly_dependent(L,L1,_), dependent(L1,U).
```

The set P of potential updates induced by U is obtained by calling:

```
setof(L,dependent(L,U),P)
```

In order to stop the generation of potential updates in presence of recursive rules, it is necessary to discard subsumed literals while constructing the set. If the rules are not recursive, this subsumption test is desirable for avoiding redundancies. For every simplified instance SI of C wrt L a Prolog fact `update_constraint(L1,(not delta(U,L1) or new(U,SI)))` can be computed by backtracking over

```
relevant(Id,L),
simplified_instance(L,SI),
assert(update_constraint(L,(not delta(U,L) or new(U,SI))))
```

The set S of queries to be evaluated on the updated database is now obtained by calling:

```
setof(UC,(update_constraint(U,UC);
          dependent(L,U),update_constraint(L,UC)),S)
```

Since it can be determined without querying the facts, this set can be precompiled as well.

3.3.2. *Simulation of the updated state*

Assume that deduction rules H←B are stored as meta-facts `rule(H<-B)`. Let the predicate `evaluate` represent a call to the database query evaluator. Assume in addition that explicitly stored facts can be accessed by means of a predicate `explicit`. The meta-predicate 'new' permitting to simulate the evaluation of formulas in the updated database before the update is actually performed can be implemented as follows:

```
new(U,true).
new(U,not A) :-
  !, not new(U,A).
new(U,A and B) :-
  !, new(U,A), new(U,B).
new(U,A) :-
  U = not V, !,
  (explicit(A), not (A = V)
        ; rule(A<-B), new(U,B)).
new(U,A) :-
  not (U = not V), !,
  (explicit(A) ; A = U
        ; rule(A<-B), new(U,B)).
```

Simulating the updated database with new does not require any specific query evaluator. Although new occurs in bodies of clauses defining it, it is worth noting that new is not recursive as long as no deduction rules of the database are recursive. This applies as well to the procedure delta defined below. In presence of recursive rules it is necessary to dispose of a query evaluator able to handle recursion in order to correctly evaluate new and delta. The other solutions proposed in the literature either require the implementation of an additional query evaluator [KOWA 87], or do not handle recursion [LING 87].

3.3.3. *Implementation of 'delta'*

By definition, delta(U,F) could be implemented by new(U,F), evaluate(not F). However, this direct implementation would in general be extremely inefficient. It would evaluate formulas not relevant to an update twice, once through the meta-predicate 'new', once in the non-updated database. As opposed to that, the following implementation of 'delta' that closely follows Definition 4 exploits that any induced update is necessarily a descendent of the update literal.

```
delta(U,U) :-
   U = not A,
   not evaluate(U), new(U,U).
delta(U,U) :-
   not (U = not A),
   not evaluate(U).
delta(U,A) :-
   A = not B,
   directly_dependent(L,A,R), delta(U,L), new(U,R),
   not evaluate(A), new(U,A).
delta(U,A) :-
   not (A = not B),
   directly_dependent(L,A,R), delta(U,L), new(U,R),
   not evaluate(A).
```

Instead of completely interpreting every delta-expression inside an update constraint, one can as well imagine various degrees of precompilation or macro expansion of delta-calls. This would result in replacing certain delta-expressions by an expression that consists of calls to new and evaluate.

We point out that it is not necessary to dispose of a coupling between Prolog and the DBMS in order to generate update constraints. The respective program refers to rules, constraints and to the update only, but not to facts. As opposed, the procedures new and delta call the database query-evaluator. Provided

the DBMS is efficiently coupled with Prolog, e.g. [BOCC 86], this approach permits to rely on the database query-evaluator.

4. Checking constraint satisfiability

In this chapter we will outline a procedure that, if applied to a set of constraints and rules, systematically attempts to construct a finite set of facts such that all constraints are satisfied in the resulting "database". This sample database is temporary and independent from the set of facts held on secondary storage. The procedure is complete for finite satisfiability as well as for unsatisfiability. If the procedure terminates successfully, finite satisfiability of rules and constraints has been shown, whereas failure indicates unsatisfiability. In case all models of the constraints and rules under consideration are infinite the construction process will not terminate. Because of the semi-decidability of both properties, such cases cannot be avoided. The procedure is based on two main principles:

1. enforcement of violated constraints by means of fact insertions into the sample database so far constructed
2. determination of constraints violated by an insertion by means of techniques introduced in the previous chapter

Initially, the set of facts to be constructed is empty. It is well possible that all constraints are already satisfied in a database without facts. This is the case iff each constraint is a universal formula, i.e., its outermost quantifier is $\forall$. Because of restricted quantification and due to the assumption that $\forall$ has been distributed over $\wedge$, every instance of a universal formula is a disjunction, at least one component of which is a negative literal. In an empty database all negative facts are true, and thus every universal formula is satisfied. This situation arises, e.g., when all constraints are functional or multi-valued dependencies.

The remaining constraints, which are not satisfied in a database without facts, are determined and successively enforced by addition of new facts. Every enforcement step can be viewed as an update in the sense of the previous chapter. Constraint violations caused by these updates can be determined according to the principles discussed and have to be enforced accordingly. Thus integrity checking and database update steps alternate until finally either all constraints are satisfied, or every possible enforcement alternative has led to constraint violations from which no recovery is possible.

As the sample database which is tentatively constructed is comparatively small, it should completely reside in main memory. It is not necessary to take care of separating fact access and update constraint determination as proposed for big secondary storage databases. Thus constraint violations can be determined on the basis of Proposition 2, i.e., by using simplified instances of constraints relevant to actually induced updates instead of potential updates.

Enforcement of violated constraint instances can be achieved by constructively exploiting the induc-

tive definition of the semantics of first-order formulas relative to some interpretation (uniquely represented by a set of positive facts F). In order to satisfy a formula C that is violated in F, do the following:

- if C is a conjunction (disjunction), satisfy all (one of) its immediate subformulas
- if C is $\forall X_1...X_n\,[\neg R \vee Q]$
  ($\neg R$ representing the disjunction of negative literals that restricts $X_1...X_n$) satisfy each instance $Q\sigma$ such that $R\sigma$ is satisfied in F
- if C is $\exists X_1...X_n\,[R \wedge Q]$,
  either satisfy at least one $Q\sigma$ such that $R\sigma$ is satisfied in F, or satisfy $[R \wedge Q]\tau$ where τ instantiates each X_i with a new constant not yet occurring in F
- if C is a positive literal, add C to F

Negative literals that are complementary to a fact in F cannot be satisfied without undoing choices made previously.

For completeness reasons we have to assume that for every rule with negative literals in its body an additional constraint has been introduced: For every rule

$$H \leftarrow A_1, ..., A_n \wedge \neg B_1 \wedge ... \wedge \neg B_m$$

involving free variables $X_1, ..., X_k$ a constraint

$$\forall X_1...X_k\,[\neg A_1 \vee ... \vee \neg A_n \vee B_1 \vee ... \vee B_m \vee H]$$

has to be added. Without this addition certain alternatives that exist for reaching a finite model of the constraint set would never be exploited.

The principle of constructively interpreting the inductive definition of formula semantics is by no means new. It has been proposed independently by several logicians in the early days of automated theorem proving. Their approach has become known as the tableaux method, which has extensively been documented in [SMUL 68]. Our method differs from the original tableaux approach in three points:

1. Instead of fully instantiating universal formulas over the whole domain, we exploit the domain independent-property of restricted quantification that permits to consider only those instantiations that are obtained through evaluation of the restricting literals.
2. In case of existential formulas, the tableaux method considers a single instance only, namely the one obtained through replacing every variable by a newly introduced constant. Consequently, the tableaux method is not complete for finite satisfiability. Only if alternatively the instances obtained through evaluation of the restricting literals are considered too, one can always guarantee that the method stops if finite models exist.
3. Our choice to determine constraints that have to be enforced next in dependence on the most recently introduced facts can be viewed as a special search strategy inside the tableaux approach which may considerably reduce its search space.

While points 1 and 3 are substantial optimizations of the tableaux method, point 2 is an extension of it. This extension is indispensable in the database context, but also leads to a more satisfactory termination behaviour in a theorem proving context. This additional capability has its price: If several constants have to be considered for replacing existential variable, the required case analysis might be fairly expensive. Exploiting domain independence - a property the relevance of which has first been recognized in

the database area - can be related to tendencies in theorem proving to exploit typed logic. However, domain independent formulas, especially those with restricted quantification, are less restrictive than the requirement for full typing.

An advantage of our approach is its close affinity to Prolog. Like for integrity maintenance, we give the basic code for a Prolog implementation making use of predicates introduced in the previous chapter. Note that the predicate `violated` implements the determination of violated constraints wrt to an update according to Proposition 2. The predicate `new_constants` is assumed to bind each element in a list of variables to a newly generated constant. Fact insertion is implemented through assertions to the Prolog main memory database. Evaluation of constraints is assumed to be done by Prolog. The predicate `assume` performs these assertions and automatically undoes them when backtracking to a previous choice point.

Like in the tableaux method, one has to organize the generation of new facts in a level-saturation manner. All constraints violated by the most recent update to the sample database are determined before any further updates are initiated. Such an organization is necessary in order to avoid cases where the database under construction is infinitely extended simply because certain constraints that would allow to stop the generation process are never considered. The parameter `I` attached to most of the Prolog predicates given below indicates the respective generation levels.

```
satisfiable :-
   setof(C,(integrity_constraint(_,C,_), not C),S),
   (S = [] ; enforce_set(0,S),
             satisfiable(1)).


satisfiable(I) :-
   I1 is I - 1,
   setof(C,(generated(I1,A),is_violated(A,C)),S),
   (S = [] ; enforce_set(I,S),
             I2 is I + 1,
             satisfiable(I2)).


is_violated(A,C) :-
   (simplified_instance(A,C) ; delta(A,L), simplified_instance(L,C)),
   not C.


   enforce_set(_,[]).
   enforce_set(I,[H|T]) :-
      H, !, enforce_set(I,T).
   enforce_set(I,[H|T]) :-
      enforce(I,H), enforce_set(I,T).
```

```
enforce(_,true) :- !.
enforce(_,false) :-
   !, fail.
enforce(_,not A) :-
   !, fail.
enforce(I,A and B) :-
   !, enforce(I,A), enforce(I,B).
enforce(I,A or B) :-
   !, (enforce(I,A) ; enforce(I,B)).
enforce(I,exists(Vars,R,Q)) :-
   R, enforce(I,Q).
enforce(I,exists(Vars,R,Q)) :-
   !, new_constants(Vars), enforce(I,R and Q).
enforce(I,forall(Vars,R,Q)) :-
   !, setof(Q,(R and not Q),S), enforce_set(I,S).
enforce(I,A) :-
   assume(A), assume(generated(I,A)).

assume(X) :- assert(X).
assume(X) :- retract(X), !, fail.
```

5. An Example

In this chapter we discuss how satisfiability would be checked by our method if applied to the following example set of rules and constraints:

Rules:

member(X,Y) ← leads(X,Y)

Constraints:

(1) $\forall X\ [\neg \text{employee}(X) \vee \exists Y\ (\text{department}(Y) \text{ and } \text{member}(X,Y))]$
(2) $\forall X\ [\neg \text{department}(X) \vee \exists Y\ (\text{employee}(Y) \wedge \text{leads}(Y,X))]$
(3) $\forall XY\ [\neg \text{member}(X,Y) \vee \forall Z\ (\neg \text{leads}(Z,Y) \vee \text{subordinate}(X,Z))]$
(4) $\forall X\ \neg \text{subordinate}(X,X)$
(5) $\exists X\ \text{employee}(X)$

This example serves at the same time as an explanation of integrity maintenance techniques as described in chapter 3.

Level 0: Initially, only constraint (5) is violated. It is enforced by generating a new constant 'a' and asserting

employee(a)

Level 1: The only constraint violated by this insertion is (1). Enforce its simplified instance $\exists Y$ [department(Y)$\wedge$member(a,Y)] by generating a new constant 'b' and asserting

department(b)
member(a,b)

Level 2: Insertion of department(b) violates constraint (2); the simplified instance to be enforced is $\exists Y$ [employee(Y)$\wedge$leads(Y,b)]. There are two alternative ways how to enforce it:

1st alternative: Evaluate the restricting literal employee(Y) over the available facts. Enforce the resulting instance by asserting

leads(a,b)

Level 3: Constraint (3) is relevant to this insertion, but two simplified instances of it have to be considered:

$$\forall X\ [\neg \text{member}(X,b) \vee \text{subordinate}(X,a)]$$

$$\forall Z\ [\neg \text{leads}(Z,b) \vee \text{subordinate}(a,Z)]$$

the latter because of the induced update member(a,b). Enforcement of both requires to insert

subordinate(a,a)

which directly contradicts constraint (4). No recovery is possible! Backtrack to the last choice point inside level 2 and retract leads(a,b) and subordinate(a,a) on the way back.

2nd alternative: Generate a new constant 'c' and insert

employee(c)

leads(c,b)

Level 3: The only simplified instance of constraint (1) that is relevant to employee(c) is $\exists Y$ [department(Y)$\wedge$member(c,Y)]. This instance is satisfied because member(c,b) is derivable from leads(c,b). Insertion of leads(c,b) results in two violated instances like in the first subcase. Both can be enforced by insertion of

subordinate(c,c)

only, which again is contradictory to constraint (4). No choice remains. On backtracking all facts are retracted and the procedure fails.

The example set has been shown to be unsatisfiable. Finite satisfiability could be achieved by, e.g., transforming constraint (3) into $\forall XY\ [\neg \text{member}(X,Y) \vee \underline{\text{leads}(X,Y)} \vee \forall Z\ (...)]$.

6. Conclusion

An integrity maintenance method and a procedure for checking constraint satisfiability have been proposed. Prolog implementations of both methods have been described.

Our approach to integrity maintenance permits to do more at compile time than other proposals in the literature. Two successive phases, a preparatory one - that does not access the base of facts - and a purely

evaluative one are distinguished. Since the constraints are altogether handed over to the database, evaluation can fully benefit from query optimization techniques. We have proposed a meta-interpreter for simulating query evaluation in the updated database before any update is actually performed. This meta-interpreter can rely on any database query evaluator and handle recursive rules, provided the considered query evaluator has this capacity.

The satisfiability checking method extends an original proof procedure with integrity maintenance techniques. It is complete for unsatisfiability and for finite satisfiability. Besides detecting undesirable situations, such as contradicting rules and constraints, it also permits to recognize the acceptable cases, i.e., rules and constraints which admit a finite model. As far as we know, this is the first time that a practicable procedure is proposed, although the need for such a method has been noticed in the literature. Though remarkably short, the Prolog programs given in this paper are fairly efficient. They appear to be useful in the following respect:

- In Prolog-DBMS couplings:
  Such couplings permit to fully implement the approach described in this paper. In particular, the updated database can be efficiently simulated by means of a meta-interpreter.
- In conventional DBMS:
  Indeed, no coupling with Prolog is needed for the phase without fact access. The meta-interpreter written in Prolog can be used as a specification of an extension to the database query evaluator.
- In Prolog main memory databases:
  Experiments made show that the time saved by the reduction techniques of the integrity maintenance method is significant as soon as base relations contain a few dozen of tuples. Conceivable domains of application are single user databases and expert systems.

Promising efficiency has been observed when testing the satisfiability checking procedure on well-known benchmark examples from the theorem-proving literature. However, this has to be completed with further experiments with deductive database applications. Examples of constraints discussed nowadays in the database literature appear to be very simple with respect to satisfiability checking. As far as integrity maintenance is concerned, further work should be devoted to the constraint evaluation phase. Most of the optimization techniques proposed till now are concerned with conjunctive queries. Since constraints have often a more general syntax, optimization methods for general formulas seem to be desirable.

7. Acknowledgement

We would like to thank Hervé Gallaire and Jean-Marie Nicolas as well as our colleagues at ECRC for providing us with a very stimulating research ambience. The work reported in this article has benefited a lot from it.

8. References

[APT 87] Apt, K.R., Blair, H. and Walker, A.
Towards a theory of declarative knowledge.
In Minker, J. (editor), *Proc. Workshop on Deductive Databases and Logic Programming*. Aug., 1987.

[BLAU 81] Blaustein, B.T.
Enforcing database assertions: Techniques and applications.
PhD thesis, Harvard Univ., 1981.

[BOCC 86] Bocca, J.
On the evaluation strategy of EDUCE.
In *Proc. ACM-SIGMOD Conf. on Management of Data.* May, 1986.

[BRY 86] Bry, F. and Manthey, R.
Checking consistency of database constraints: A logical basis.
In *Proc. 12th VLDB Conf.* Aug., 1986.

[BRY 87] Bry, F.
Maintaining integrity of deductive databases.
Int. Rep. KB-45, ECRC, July, 1987.

[DECK 86] Decker, H.
Integrity enforcement on deductive databases.
In *Proc. 1st Int. Conf. on Expert Database Systems.* Apr., 1986.

[KOWA 87] Kowalski, R., Sadri, F. and Soper, P.
Integrity checking in deductive databases.
In *Proc. 13th VLDB Conf.* Sept., 1987.

[KUHN 67] Kuhns, J.L.
Answering questions by computers - A logical study.
Rand Memo RM 5428 PR, Rand Corp., Santa Monica, Calif., 1967.

[KUNG 84] Kung, C.H.
A temporal framework for information systems specification and verification.
PhD thesis, Univ. of Trondheim, Norway, 1984.

[LASS 87] Lassez, C., McAloon, K. and Port, G.
Stratification and Knowledge Base Management.
In *Proc. 4th Int. Conf. on Logic Programming.* May, 1987.

[LING 87] Ling, T.
Integrity constraint checking in deductive databases using the Prolog not-predicate.
Data & Knowledge Engineering 2, 1987.

[LLOY 86] Lloyd, J.W. and Topor, R.W.
Integrity constraint checking in stratified databases.
Technical Report 86/5, Univ. of Melbourne, May, 1986.

[MANT 87a] Manthey, R. and Bry, F.
A hyperresolution-based proof procedure and its implementation in PROLOG.
In Morik, K. (editor), *Proc. GWAI-87 (German Workshop on Artificial Intelligence).* Sept., 1987.
Springer Verlag IFB 152.

[MANT 87b] Manthey, R. and Bry, F.
SATCHMO: a theorem prover implemented in Prolog.
Technical Report KB-21, ECRC, Nov., 1987.
(submitted to CADE 88).

[NICO 79] Nicolas, J.-M.
Logic for improving integrity checking in relational databases.
Technical Report, ONERA-CERT, Toulouse, France, Feb., 1979.
Also in Acta Informatica 18, 3, Dec. 1982.

[SMUL 68] Smullyan, R.M.
First-order logic.
Springer Verlag, 1968.

[VIEI 87] Vieille, L.
A database-complete proof procedure based on SLD-resolution.
In *Proc. 4th Int. Conf. on Logic Programming.* May, 1987.