An Extensible Framework for Repairing Constraint Violations

M. Gertz, U.W. Lipeck
Institut für Informatik, Universität Hannover
Lange Laube 22, 30159 Hannover, Germany
Phone/Fax: ++49-511-762-4950/-4951
{mg|ul}@informatik.uni-hannover.de

Abstract

In this paper we describe a new approach to repairing violations of integrity constraints in relational databases with null values. By adopting basic concepts from model-based diagnosis, we show how simultaneous reasons for violations of (different) constraints can be determined. These reasons, represented as sets of facts, directly indicate possible repair actions that guarantee to remove the observed violations.

By interleaving the diagnosis of constraint violations and the execution of repair actions, we draw an enumeration schema for possible minimal repair transactions as sequences of repair actions. Each such transaction, when applied to the inconsistent database, guarantees to result in a database consistent with all constraints. In order to enumerate possible repair transactions, repair actions are performed hypothetically using auxiliary relations. This enables the user to query intermediate as well as result states obtained by different repairs in advance.

In order to provide a suitable front-end to the general enumeration schema, we describe various repair strategies which can be imposed by the user. These strategies follow individually specified repair goals and can easily be integrated into the enumeration schema for repair transactions. The proposed strategies range from aspects of minimal change over priorities of stored facts up to the user interaction with the repair process.

Keywords

Constraint Enforcement, Inconsistent Information, Model-Based Diagnosis, Repairing Inconsistencies, Repair Strategies

1 INTRODUCTION

Numerous papers have been written on specifying and maintaining integrity constraints in databases since the first large database conference (Eswaran and Chamberlin 1975, Hammer and McLeod 1975) (for an overview see, e.g., (Grefen and Apers 1993, Widom 1994)). The aim has always been to develop methods that efficiently check integrity constraints for violations. Several proposals on integrity maintenance have been made for relational, deductive and object-oriented databases (Grefen and Apers 1993, Widom 1994, Celma *et al.* 1994, Jeusfeld and Jarke 1991).

Nearly all of these approaches are *passive*, i.e. in case of constraint violations a rollback of the complete transaction is performed. For several applications, however, such a drastic action is insufficient. This holds in particular for nowadays emerging database applications like engineering or design databases (Morgenstern *et al.* 1986, Encarnacao and Lockemann 1990, Buchmann *et al.* 1991), where thus transactions containing a multitude of operations would be undone and a lot of work would get lost. Furthermore, in case of constraint violations the designer has to identify reasons for the violations and possible repairs of the violating transaction by her/himself. This, of course, is not a trivial task since respective applications typically contain numerous complex semantic integrity constraints that describe interdependencies between various relations.

To overcome these problems so-called *active* constraint enforcement methods performing repairing actions have been developed. The topic of repairing constraint violations has recently become a new discipline in the database area. Several proposals have been made to this topic in the context of active databases, e.g., (Urban and Desiderio 1992, Ceri *et al.* 1994, Gertz 1994, Schewe and Thalheim 1994), and deductive databases, e.g., (Moerkotte and Lockemann 1991) (see (Fraternali and Paraboschi 1993) for an extensive overview).

The drawback of these approaches is, however, that they in general realize an autonomous repair of constraint violations. Although the user can choose from automatically derived repairing triggers at compile-time, these triggers are kept fixed at run-time. Once a repair is triggered in an inconsistent database, there is no way to interact with the repair process. Furthermore, often a repair of violations may introduce new violations which are then automatically repaired and so on. Hence it is difficult for the user to identify why what happened. Interesting questions are also what happens if the result state does not reflect the user's intention or the application requirements? How can she/he choose between possible alternative repairs?

With respect not only to the application domains mentioned above, we think that the properties listed below are important for any method performing the repair of constraint violations:

- determination of facts, i.e. *stored tuples* (positive facts) as well as *missing tuples* (negative facts) that contribute to the different constraint violations,
- exposition of *common reasons* for different violations,
- possibility for the user to choose a *repair strategy* following a *repair goal*,
- enumeration of possible repair transactions from which the user chooses *before* executing a repair transaction on the inconsistent database, and
- comparison of the *effect* of different possible repair transactions.

In this paper we describe a general framework which tries to fulfill these requirements and which is extensible concerning further demands on the repair of constraint violations. We propose a sound and complete enumeration schema for possible minimal repair transactions for an inconsistent database. For this we employ basic concepts from model-based diagnosis (Reiter 1987, Hamscher *at al.* 1992). The reason for this is that there is a close relationship between repairing constraint violations in databases and the diagnosis and repair of malfunctioning components of (technical) systems (Gertz and Lipeck 1995). The used concepts provide a diagnostic means to determine minimal sets of facts that contribute to all violations in an inconsistent database. From these sets minimal repair actions are derived that guarantee to remove the observed violations, but which may possibly introduce new violations. By iterating the diagnosis and repair of constraint violations finally minimal repair transactions for the inconsistent database are computed. Iterations are performed using auxiliary relations, thus allowing the user to query different computations and result states of repair transactions.

In contrast to other approaches we additionally consider marked null values. The rationale for this is that often missing facts are reasons for constraint violations and that only some attribute values of these facts are known. Null values provide a suitable means to represent such missing information, e.g., in order to satisfy referential integrity constraints.

The enumeration of all minimal repair transactions, of course, is not well-suited for practical usage, but provides a suitable framework for application independent or application dependent extensions. That is why we propose various repair strategies which can be individually imposed by the user for the computation of possible repairs. Aside from the effect that such strategies reduce the number of possible repairs they are a suitable means to integrate semantic aspects into the repair process. The proposed repair strategies range from minimal change semantics like a minimal undo or consistent completion of a violating transaction up to the user interaction with the repair process.

The paper is structured as follows: In Section 2 we shortly introduce the basic relational concepts and we sketch the main idea of model-based diagnosis and its connection to the repair of constraint violations. Section 3 describes how to collect information about constraint violations and how violations are diagnosed for simultaneous reasons using techniques from model-based diagnosis. In Section 4 we present our algorithm to enumerate alternative possible minimal repair transactions for an inconsistent database. Section 5 presents some repair strategies which can be imposed on the enumeration schema. A complete discussion and formalization of the presented approach can be found in (Gertz 1996).

2 INTEGRITY CONSTRAINTS AND MODEL-BASED DIAGNOSIS

2.1 Constraint Specification and Checking

For our approach we assume an extended relational model with marked null values (Reiter 1984, Imielinski and Lipski 1984). An extended relational database schema is essentially the same as for ordinary relational databases without null values; that is, it contains a set $\mathcal{P} = \{p_1, \ldots p_n\}$ of *base relations*, a collection $\mathcal{D}$ of domains and a set $\mathcal{C}$ of integrity constraints. The difference is that each domain $D_i \in \mathcal{D}$ may contain, aside from *ordinary constants*, a finite set $\{\varepsilon_{i_1}, \ldots, \varepsilon_{i_l}\}$ of *marked null values* (or *special constants*). Marked null values differ from ordinary constants. They denote constants that are incompletely identified in the database; they can denote ordinary constant or completely new ones. Two occurrences of the same marked null value in relations, however, denote the same unknown value. In the sequel we denote a *database state* (or *database* for short) determined by the tuples in the relation instances at a given point of time by B.

Integrity constraints $\mathcal{C}$ are formulated in the relational language that can be associated with a given database schema. In this paper we concentrate on integrity constraints in *implicative normal form.*

Definition 1 (Integrity Constraint) An integrity constraint in *implicative normal form* (INF) is a closed range-restricted formula of the pattern

$$\forall \mathbf{x} : \; (p_1(\mathbf{x}_{p_1}) \wedge \ldots \wedge p_n(\mathbf{x}_{p_n})) \implies \exists \mathbf{y} : \; (q_1(\mathbf{x}_{q_1}, \mathbf{y}_{q_1}) \vee \ldots \vee q_m(\mathbf{x}_{q_m}, \mathbf{y}_{q_m})).$$

Each p_i, q_j is either a *base predicate* corresponding to a base relation, or a comparison operator like $=, <, >$ etc. The notations $\mathbf{x}, \mathbf{y}, \mathbf{x}_{p_i}, \mathbf{y}_{q_j}$ denote domain respecting vectors of variables and/or constants as arguments of the predicates. This class of constraints is very general since it includes foreign key constraints, functional dependencies etc. Several classes of more complex constraints (also called *semantic integrity constraints*) can be transformed into constraints in INF by means of transformation rules similar to those presented in (Lloyd and Topor 1984).

The prevailing logical approach to databases with null values is the model-theoretic approach. For a database B without null values, the model-theoretic approach always succeeds in choosing the least Herbrand model. This is not the case in the presence of null values; there several models of the database may exist, each model describing a database with complete information compatible with B. Roughly speaking, these models are characterized by possible mappings $\xi : \mathcal{E} \rightarrow \mathcal{HB}$ (as part of an interpretation) that assign (ordinary) constants from the Herbrand base $\mathcal{HB}$ to the null values $\mathcal{E}$. It should be clear that in this case a formula can be true in some models of B and false in some others.

For the satisfaction of a constraint C in a database B we adopt an optimistic view as suggested in (Vardi 1986), since requiring a constraint C to be true in all models of a database B with null values would be too restrictive.

Definition 2 (Constraint Satisfaction) An integrity constraint C is said to be *satisfied* in a database B if there exists a model of B (corresponding to an assignment ξ to the null values) such that $B\xi$ is a also model of C. If the constraint C is satisfied in B, this is denoted by $B \models_{\approx} C$. If there does no exist such a model of B, i.e. C is unsatisfiable in B, C is said to be *violated* in B.

In a given database B, a constraint C is checked by evaluating its associated violation query

$$\begin{aligned}\tilde{C} := \{\mathbf{x} \mid\, & p_1(\mathbf{x}_{p_1}) \wedge \ldots \wedge p_n(\mathbf{x}_{p_n}) \wedge \\ & \forall \mathbf{y}_{q_1} : \neg q_1(\mathbf{x}_{q_1}, \mathbf{y}_{q_1}) \wedge \ldots \wedge \forall \mathbf{y}_{q_m} : \neg q_m(\mathbf{x}_{q_m}, \mathbf{y}_{q_m})\}.\end{aligned}$$

If the query evaluates to the empty set, C is said to be satisfied in B, otherwise it is violated and each tuple determined by $\tilde{C}$ is called a *violation* of C. For a query evaluation algorithm in the presence of null values see, e.g., (Reiter 1986). For further issues and problems connected with querying incomplete information see (Abiteboul *et al.* 1991).

2.2 Model-Based Diagnosis

During the last decade *model-based diagnosis* has become a prominent research area in Artificial Intelligence for describing techniques that can be used to identify malfunctioning components of a system (Hamscher *at al.* 1992). One fundamental approach to diagnostic reasoning is called *model-based diagnosis* or *consistency-based diagnosis* (Reiter 1987). Starting point of this approach is a description (a model) of a real-world system. Such a model represents the structure of the system; that is, its components and their interrelations. If now the actual behavior of the system conflicts with the expected behavior of the system a diagnostic task has to be performed. This task comprises identifying those components of the system which, when assumed to function abnormally, will account for the difference between expected and observed behavior.

In model-based diagnosis a *system* is defined as a pair (*SD*, *COMP*), where *SD* is a system description as a set of first order sentences defining how the system components are interrelated and how they normally behave. *COMP* is a finite set of constants denoting the system's components. In the system description a distinguished predicate *ab* on the components is defined whose intended meaning is "*abnormal*". The literal $ab(c)$ holds when a component $c \in COMP$ is behaving abnormally. Typically system descriptions will specify system behavior on the condition that all components are not abnormal. An observation OBS on the system and its components is a finite set of first order sentences. Using the *ab*–predicate REITER (Reiter 1987) characterizes a diagnosis as follows:

Definition 3 (Diagnosis) A diagnosis Δ for $(SD, COMP, OBS)$ is a subset $\Delta \in COMP$ such that

$$SD \cup OBS \cup \{ab(c) \mid c \in \Delta\} \cup \{\neg ab(c) \mid c \in COMP - \Delta\}$$

is consistent. A diagnosis is *minimal* if no proper subset of it is also a diagnosis.

In other words, for a diagnosis Δ the assumption that these components are abnormal, i.e. $ab(c), c \in \Delta$ holds, together with the assumption that all other components are behaving normal, is consistent with the system description SD and the observations OBS. REITER'S subsequent characterization of diagnosis and its computation exploits the notion of *conflict sets.*

Definition 4 (Conflict Set) A *conflict set* for $(SD, COMP, OBS)$ is a set $\{c_1, \ldots, c_n\} \subseteq COMP$ such that $SD \cup OBS \cup \{\neg ab(c_1), \ldots, \neg ab(c_n)\}$ is inconsistent. A conflict set is *minimal* if no proper subset of it is also a conflict set.

A conflict set thus is a set of components that cannot altogether assumed to be not abnormal without leading to an inconsistency with the system description and the observations. The next definition characterizes a *hitting set* for a collection S of sets:

Definition 5 (Hitting Set) A *hitting set* for S is a set $H \subseteq \bigcup_{M \in S} M$, such that $H \cap M \neq \{\,\}$ for each $M \in S$. A hitting set is *minimal* if no proper subset of it is also a hitting set.

In other words, a hitting set for a collection S of critical sets contains at least one component from each $M \in S$. Based on these definitions a diagnosis can be characterized in a way that builds the basis for an efficient computation of diagnoses.

Theorem 6 ((Reiter 1987), p. 67) *A set $\Delta \subseteq COMP$ is a diagnosis for $(SD, COMP, OBS)$ iff Δ is a minimal hitting set for the collection of all conflict sets for $(SD, COMP, OBS)$.*

The computation of minimal diagnoses presented in (Reiter 1987) follows directly from this theorem. All minimal hitting sets corresponding to minimal diagnoses are computed by constructing a *hitting set tree* (HS-tree). Due to space limitations we refer the reader to the respective article (Reiter 1987) for a detailed description and examples of the algorithm.

It turns out that there is a close relationship between the task of diagnosing and repairing malfunctioning components of systems and repairing constraint violations in databases. The following relationships can be drawn:

- specified integrity constraints can be seen as a system description specifying correct system instances
- facts in the database describe the components of an actual system instance
- constraint violations indicate a "misbehavior" of the database contents
- the ultimate goal of repairing constraint violations is to satisfy all integrity constraints by performing repair actions on the inconsistent database

An essential difference to model-based diagnosis, however, is that for repairing constraint violations we do not only reason about the facts stored in an inconsistent database, but also about the facts that are not stored in the database. That is, we do not only consider existing components (positive facts), but also missing components (negative facts) that are necessary to be inserted into the database in order to satisfy, e.g., a referential integrity constraint. Thus for repairing constraint violations the set of system components is not as "simple" as in model–based diagnosis. The diagnostic task on an inconsistent database then can be described as identifying those sets of positive and negative facts that account for the observed constraint violations.

3 DIAGNOSING CONSTRAINT VIOLATIONS

The objective of this section is to describe an approach which allows

- to determine reasons for the constraint violations in an inconsistent database B by computing those possible *minimal sets* of positive and negative facts that account for **all** violations in B,
- to characterize schemas for possible repair actions as sets of modifications that can be associated with such sets of facts.

Instead of reacting separately on each constraint violation in an inconsistent database B, we first require that the result of each violation query $\widetilde{C}$, $C \in \mathcal{C}$, is stored in an auxiliary *violation relation*, denoted by $viol_C$. The schema of a violation relation is determined by the relation schemas of the base predicates occurring in $\widetilde{C}$. E.g., in the context of active databases, storing violations can be done by means of triggers which evaluate the violation queries. For each such violation in $viol_C$ we now want to determine facts, i.e. positive and negative ground literals that contribute to the violation.

Definition 7 (Critical Facts) Let L^+ and L^- denote the lists of base predicates that occur positively, respectively, negatively in $\widetilde{C}$. Given a violation $\mathbf{v} = (a_1, \dots, a_k)$ from a violation relation $viol_C$. The corresponding set $h_\mathbf{v}$ of positive and negative *critical facts* is defined as

$$h_{\mathbf{v}} := \bigcup_{p \in L^+} \{p(\mathbf{a}_p) \mid \mathbf{a}_p = \pi_{attr(p)}(\mathbf{v})\} \cup \bigcup_{q \in L^-} \{\neg q(\mathbf{a}_q, \vec{\varepsilon}_i) \mid \mathbf{a}_q = \pi_{attr(q)}(\mathbf{v})\}$$

where $\vec{\varepsilon}_i$ denotes a vector of marked null values. A fact in $h_{\mathbf{v}}$ is said to be a *critical fact.*

Negative facts (or rather missing tuples) are derived from $viol_C$ by the rightmost union of the equation above. Depending on the number of quantified variables in a base predicate q_j in $\tilde{C}$ *marked null values* are introduced to build the respective critical fact. It is necessary to introduce a new (domain respecting) null value for each quantified variable and each violation instance. The reason for this is, that we have to distinguish critical negative facts participating on different violations. Choosing the same null value for different critical facts would imply the same (unknown) attribute value what, of course, is too restrictive, since the equality of different existentially quantified variables should remain undetermined. In the sequel we denote the collection of all sets of critical facts for the constraints C in a database B by H_C.

It is worth mentioning that in particular negative facts are of interest. They describe (though possibly incompletely) missing facts in the database which are needed, e.g., in order to satisfy a referential integrity constraint.

It is obvious, that each critical fact in $h_{\mathbf{v}}$ is a possible reason for the violation $\mathbf{v}$. Critical facts directly suggest respective repairing modifications. If $p(\mathbf{a}) \in h_{\mathbf{v}}$, then deleting the tuple $\mathbf{a}$ from p, denoted by $del_p(\mathbf{a})$, removes the violation $\mathbf{v}$. Analogously, if $\neg q(\mathbf{c}) \in h_{\mathbf{v}}$, inserting the tuple $\mathbf{c}$ in q, denoted by $ins_q(\mathbf{c})$, removes the violation, too. The objective now is to determine possible minimal sets of modifications that remove **all** the determined violations. A naive approach, of course, would be to take a critical fact from each $h_{\mathbf{v}} \in H_C$, and to perform the associated repair modification, hence removing the violation associated with $h_{\mathbf{v}}$. But this procedure is not very well structured and does not necessarily result in a minimal set of modifications.

In contrast to other approaches to active constraint enforcement we are in particular interested in possible *simultaneous reasons*, i.e. single facts that contribute to more than only one violation. For such facts the associated repairing modifications then remove more than only one violation. We call a minimal set of positive and negative facts whose corresponding modifications remove all violations in an inconsistent database B a *state diagnosis.* The set of modifications corresponding to a state diagnosis is called a *repair action.*

Given a collection H_C of sets of critical facts in an inconsistent database B, minimal state diagnoses and corresponding repair actions can be determined by adopting concepts from model-based diagnosis, namely the computation of *hitting sets* (Reiter 1987). We will give only the main idea here and refer the interested reader to (Gertz 1996) where a complete formalization in the model–theoretic approach with a particular emphasis on null values is given.

It can be shown that each set $h_{\mathbf{v}} \in H_C$ of critical facts determines a conflict set (cf. Definition 4). This is an obvious issue since at least one fact in $h_{\mathbf{v}}$ needs to be modified (i.e. to be inserted or to be deleted) in order to remove the violation $\mathbf{v}$. Basis for the computation of hitting sets and minimal state diagnoses, respectively, are now exactly all sets of critical facts contained in H_C. That is, we can adopt the hitting set algorithm used in model-based diagnosis. Due to possible null values in negative facts, computed hitting sets need to be checked whether two negative literals that contain null values are *null unifiable.* The main idea is to check whether some null values can be reasonably replaced by ordinary constants such that the resulting instance provides more complete information than the negative literals under consideration.

Due to space limitations, we will give only an example here which reflects how state diagnoses and repair actions are chosen for violations of different constraints in a database B, and how negative literals containing null values are unified.

Example 8 Suppose the constraints

$C_1 \equiv \forall x,y,z : p_1(x,y) \wedge p_2(y,z) \implies \exists v : q(v,x)$ and
$C_2 \equiv \forall u,v,w : p_2(u,v) \wedge p_2(v,w) \implies \exists z : q(w,z)$

and the following relation instances in a database B:

p_1	A	B
	b	d
	c	d
	a	b

p_2	B	C
	d	a
	b	f
	g	d

q	B	A
	f	g
	d	g
	b	a

The computation of the violation queries $\widetilde{C_1}$ and $\widetilde{C_2}$ evaluates to the following violations:

$viol_{C_1}$	A	B	C
	b	d	a
	c	d	a

$viol_{C_2}$	B	$C \mid B$	C
	g	d	a

Applying the computation rule for critical facts to these three violations results in

$h_{\mathbf{v}_1} \equiv \{\, p_1(b,d), p_2(d,a), \neg q(\varepsilon_1, b) \,\}$,
$h_{\mathbf{v}_2} \equiv \{\, p_1(c,d), p_2(d,a), \neg q(\varepsilon_2, c) \,\}$,
$h_{\mathbf{v}_3} \equiv \{\, p_2(d,a), p_2(g,d), \neg q(a, \varepsilon_3) \,\}$.

Inter alia, the following minimal state diagnoses Δ_i with their associated repair actions T_{Δ_i} can be determined for $H_C = \{h_{\mathbf{v}_1}, h_{\mathbf{v}_2}, h_{\mathbf{v}_3}\}$:

$\Delta_1 = \{p_2(d,a)\}$ $\quad T_{\Delta_1} = \{del_{p_2}(d,a)\}$
$\Delta_2 = \{p_1(b,d), p_1(c,d), p_2(g,d)\}$ $\quad T_{\Delta_2} = \{del_{p_1}(b,d), del_{p_1}(c,d), del_{p_2}(g,d)\}$
$\Delta_3 = \{\neg q(a,b), \neg q(a,c)\}$ $\quad T_{\Delta_3} = \{ins_q(a,b), ins_q(a,c)\}$

For example, the diagnosis Δ_1 indicates that the single fact $p_2(d,a)$ is a simultaneous reason for all three violations; that is, deleting the tuple (d,a) from the relation p_2 guarantees to remove all three violations. Please note that all state diagnoses and repair actions are minimal, i.e. there exists no proper subset of a determined repair action that removes the observed violations, too. Additionally negative literals containing null values as arguments are reasonably combined into single literals with more complete information (diagnosis Δ_3) in order to fulfill the minimality property of a repair action. Due to the underlying notion of constraint satisfaction (cf. Definition 2), for example, inserting the tuple (a,b) into the relation q would, aside from removing the violation associated with h_{v_3}, also remove the violation associated with h_{v_1}. The unification of respective negative literals can suitably be integrated into the hitting set algorithm as shown in (Gertz 1996).

It is important to note that for an inconsistent database always at least one minimal diagnosis exists which, in the worst case, corresponds to an undo of the violating transaction. It is also obvious that the execution of a repair action T_{Δ_i} on an inconsistent database does not necessarily result in a consistent state. This, of course, is not a drawback since repair actions may require subsequent repairs. Respective considerations have been made in all approaches to repairing constraint violations. In the next section we show that it is nevertheless possible to determine consistency preserving transactions by simulating a "one–step computation".

4 REPAIRING CONSTRAINT VIOLATIONS

We now describe a general enumeration schema for minimal *repair transactions* in an inconsistent database. This schema then serves as the basis for the repair strategies that will be discussed in Section 5. In contrast to a repair action, a repair transaction always guarantees to result in a consistent database; that is, for applying a repair transaction T to an inconsistent database B, denoted by $T(B)$, we have that for the result database B' the condition $B' \models_{\approx} C$ holds. For this, the proposed method tries to enumerate possible repair transactions for an inconsistent database B as illustrated in Figure 1.

$B_{0,1,0}$ denotes the initial inconsistent database obtained by the violating user transaction. Performing a diagnosis on the violations in $B_{0,1,0}$ results, for example, in two different minimal state diagnoses. Executing the associated repair actions on $B_{0,1,0}$ would result in two different databases $B_{1,1,1}$ and $B_{1,2,1}$. Each database $B_{l,i,pre}$ has as subscripts the level l, the number i of the database at that level l and the number *pre* of the database at the previous level which led to $B_{l,i,pre}$ by executing the repair action associated with a diagnosis (i.e. $\Delta_{1,1,1}$ and $\Delta_{1,2,1}$). Assume that in $B_{1,1,1}$ at level 1 again constraints are violated. Again minimal state diagnoses have to be determined.

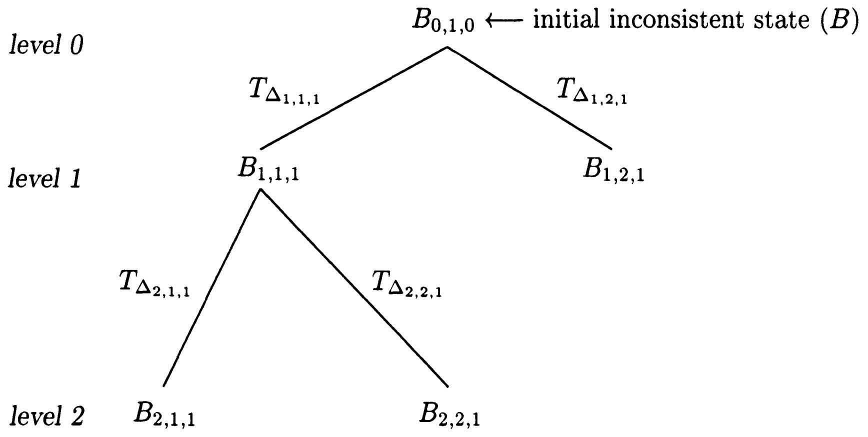

Figure 1 Enumerating Possible Repair Transactions

Iterating this procedure for each database and each diagnosis builds a tree of possible repair actions and databases reachable from $B_{0,1,0}$. Leaves of the tree denote databases which are consistent with all constraints. For example, the state $B_{2,2,1}$ obtained by the sequence of repair actions associated with $\Delta_{1,1,1}$ and $\Delta_{2,2,1}$ is a consistent database. The union of the repair actions associated with $\Delta_{1,1,1}$ and $\Delta_{2,2,1}$ together builds a repair transaction T for $B_{0,1,0}$ whose execution on $B_{0,1,0}$ results in $B_{2,2,1}$.

Instead of executing repair actions on an inconsistent database, respectively, on base relations directly, we store their effect in auxiliary relations, also called *differential relations.* For each base relation p we introduce two relations I_p and D_p as follows.

Definition 9 (Differential Relations)
For each base relation $p\langle A_1, \ldots, A_n\rangle \in \mathcal{P}$ two *differential relations*

$$I_p\langle A_1, \ldots, A_n, \mathit{level}, \mathit{num}, \mathit{pre\text{-}db}\rangle \quad \text{and} \quad D_p\langle A_1, \ldots, A_n, \mathit{level}, \mathit{num}, \mathit{pre\text{-}db}\rangle$$

are added to the database schema, where the attributes *level, num* and *pre-db* range over the domain integer.

These relations are used to represent different *hypothetical* databases as well as different repair actions on these states. For this the additional three attributes reflect the node labelling in the enumeration tree. Given a hypothetical database $B_{l,i,pre}$, using these relations it is now easy to determine which facts would have been inserted and deleted from the base relations in order to obtain this state from the initial inconsistent database $B_{0,1,0}$.

Definition 10 (Hypothetical Extension) For a base relation $p \in \mathcal{P}$ the *hypothetical extension* $\widehat{p}$ in a database $B_{l,i,pre}$ is computed by:

```
$\hat{p} := p$; /* original extension as in the database $B_{0,1,0}$ */
for $l_1 := l$ to 1 do
    /* for a database $i$ at level $l_1$ the pre-state $pre$ is uniquely determined */
    $\hat{p} := \hat{p} \cup \{\mathbf{a} \mid (\mathbf{a}, l_1, i, pre) \in I_p\}$;
    $\hat{p} := \hat{p} - \{\mathbf{a} \mid (\mathbf{a}, l_1, i, pre) \in D_p\}$;
    $i := pre$; /* get the number of the previous state at level $l_1 - 1$ */
end for;
```

In other words, for a given possible database $B_{l,i,pre}$ we determine those tuples which need to be inserted in, respectively, need to be deleted from each base relation $p \in \mathcal{P}$ in $B_{0,1,0}$ by the repair actions leading to that state. A *hypothetical* or *possible database* $B_{l,i,pre}$ thus is the collection of hypothetical extensions of the base relations for that particular database.

It is obvious, that the original integrity constraints $\mathcal{C}$ cannot be used to check violations in hypothetical databases. Instead of a constraint $C \in \mathcal{C}$, we use the corresponding *hypothetical state constraint.*

Definition 11 (Hypothetical State Constraint) A *hypothetical state constraint* for an integrity constraint C has the pattern:

$$C^{\text{hyp}} \equiv \forall \mathbf{x}\colon (\hat{p}_1(\mathbf{x}_{p_1}) \wedge \ldots \wedge \hat{p}_n(\mathbf{x}_{p_n})) \implies \exists \mathbf{y}\colon (\hat{q}_1(\mathbf{x}_{q_1}, \mathbf{y}_{q_1}) \vee \ldots \vee \hat{q}_m(\mathbf{x}_{q_m}, \mathbf{y}_{q_m}))$$

i.e. all predicates in C denoting base relations are replaced by the corresponding predicates denoting hypothetical extensions of the base relation.

In the sequel we denote the set of all hypothetical state constraints corresponding to $\mathcal{C}$ by $\mathcal{C}^{\text{hyp}}$. For these constraints the computation of violations and associated sets of critical facts occurs in the same way as for the original constraints as described in the Sections 2 and 3.

Below the central algorithm describing a sound and complete enumeration schema for all possible minimal repair transactions on an inconsistent database is presented. The algorithm has to be applied to a database B obtained by a user transaction. It performs a breadth-first search for minimal repair transactions through hypothetical databases.

Algorithm 12

```
variables
$l := 0$;                  /* initial (inconsistent) database $B_{0,1,0}$ is at level 0 */
hyp_dbs_at_l := 1; /* $B_{0,1,0}$ is the only hypoth. database at level 0 */
violation_found := false; /* violation in hypothetical database found */
new_dbs := 0;        /* number of hypothetical databases at level $l + 1$ */
diagnoses := 0;     /* number of state diagnoses on a hypoth. database */
$\mathcal{T} := \{\}$;              /* collection of determined repair transactions */
$H_C := \{\}$;             /* collection of sets of critical facts */
```

```
/* Main Loop */
 repeat
   violation_found := false;
   for $i$ := 1 to hyp_dbs_at_l do
     /* consider hypoth. database $B_{l,i,pre}$ ($pre$ is determined by $l$ and $i$) */
     for each $p \in \mathcal{P}$ do
       determine hypothetical extension $\widehat{p}$ for $B_{l,i,pre}$ according to Def. 10;
     end for;

     /* check hypoth. database constraints based on the $\widehat{p}$ 's in $B_{l,i,pre}$ */
     for each $C^{\mathrm{hyp}} \in \mathcal{C}^{\mathrm{hyp}}$ do
       $viol_C = \{\}$;
       insert result of the evaluation of violation query $\widetilde{C}^{\mathrm{hyp}}$ into $viol_C$;
     end for;

     if all violation relations are empty then
       if $l = 0$ then user transaction led to consistent database; exit;
       else
         /* repair transaction leading to $B_{l,i,pre}$ has been determined */
         add_repair_transaction($l, i, pre$);
     else violation_found := true;
       $H_C$ := $\{\}$; /* determine sets of critical facts for database $B_{l,i,pre}$ */
       for each non-empty violation relation $viol_C$ do
         $H_C$ := $H_C \cup \{$ sets of critical facts determined for violations in $viol_C\}$
       end for;

       /* compute number of minimal state diagnoses on $B_{l,i,pre}$ using $H_C$ */
       diagnoses := hitting_sets($H_C, l, i,$ new_dbs);

       /* increase number of hypoth. states to be considered at next level */
       new_dbs := new_dbs + diagnoses;
   end for;
   $l := l + 1$;  hyp_dbs_at_l := new_dbs;  new_dbs := 0;
 until violation_found := false;
```

The structure of the main loop is very simple: Each hypothetical database at a given level is checked for violations of the hypothetical state constraints. For this, the extensions of the hypothetical relations are evaluated in this database. In the case where no hypothetical state constraint is violated, i.e. when $B_{l,i,pre} \models \mathcal{C}^{\mathrm{hyp}}$ holds, a repair transaction has been determined. One can picture this situation as when a leaf in the tree of possible hypothetical databases has been reached (see Figure 1).

The following procedure **add_repair_transaction** checks if a sequence of repair actions leading to the consistent database under consideration builds a minimal repair transaction.

```
procedure add_repair_transaction($l, i, pre$);
  /* sequence of repair actions leading to $B_{l,i,pre}$
     determines repair transaction $T$ */
  /* (1) determine tuples deleted up to $B_{l,i,pre}$ */
  $T_{del} := \bigcup_{p \in \mathcal{P}} \{del_p(\mathbf{a}) \mid \mathbf{a} \in (p - \widehat{p})\}$;
  /* (2) determine tuples inserted up to $B_{l,i,pre}$ */
  $T_{ins} := \bigcup_{q \in \mathcal{P}} \{ins_q(\mathbf{c}) \mid \mathbf{c} \in (\widehat{q} - q)\}$;
  $T := \{T_{ins} \cup T_{del}\}$;

  if there exists a repair transaction $T' \in \mathcal{T}$ such that $T' \subset T$ then
    $T$ is not a minimal repair transaction;
  else if there exists a repair transaction $T' \in \mathcal{T}$ such that $T' \supset T$ then
         drop $T'$ from $\mathcal{T}$
       $\mathcal{T} := \mathcal{T} \cup T$; /* store computed minimal repair transaction */
end.
```

In the case where there exists a non–empty violation relation in the database $B_{l,i,pre}$ under consideration, the diagnostic task is performed on the collection of sets of critical facts determined in $B_{l,i,pre}$. The collection H_C and further information about the current hypothetical database are then passed to the function below.

```
function hitting_sets($H_C, l, i, n$) : $m$;
  invoke the hitting set algorithm with $H_C$ and
  determine the set $\mathcal{G}$ of minimal state diagnoses on $B_{l,i,n}$;
  $m := 0$; /* number of diagnoses */

  for each state diagnosis $\Delta \in \mathcal{G}$ do
    if there exists a positive literal $p(\mathbf{a}) \in \Delta$ such that $\mathbf{a} \in \widehat{p} - p$ or
       there exists a negative literal $\neg q(\mathbf{c}) \in \Delta$ such that $\mathbf{c} \in q - \widehat{q}$
    then /* repair action $T_\Delta$ undoes previous repair(s) leading to $B_{l,i,n}$ */
      do nothing;
    else /* store repair action associated with $\Delta$ in differential relations */
      for each positive literal $p(\mathbf{a}) \in \Delta$ do
        insert $(\mathbf{a}, l+1, n+m+1, i)$ into $D_p$ end for;
      for each negative literal $\neg q(\mathbf{c}) \in \Delta$ do
        insert $(\mathbf{c}, l+1, n+m+1, i)$ into $I_q$ end for;
      $m := m + 1$; /* increase the number of admissible state diagnoses */
  end for;

  return $m$; /* number of diagnoses determined on database $B_{l,i,n}$ */
end.
```

The current number of diagnoses on hypothetical databases at level l is passed to the function **hitting_sets** in order to suitably enumerate further diagnoses and associated repair actions which are stored in the differential relations. The number of admissible diagnoses (i.e. those not performing an undo of previous repairs) on $B_{l,i,n}$ is returned to the main loop in order to increase the total number of hypothetical databases which need to be considered at the next level $l + 1$.

In order to guarantee termination of the algorithm, in the function **hitting_sets** those diagnoses are excluded whose associated repairs would undo a previous repair leading to the state under consideration (hence only the number of "admissible" diagnoses is returned).

(end algorithm 12)

Avoiding the undo of a repair is essential for the termination of the algorithm as well as for its soundness and completeness. Since we do not allow function symbols in our constraint specification language, the only possibility of non-termination of the algorithm is when a repair action is undone and thus possibly non-terminating cycles can be introduced.

Repair cycles are also critical points in active constraint enforcement methods and they have been investigated in several work, e.g., (Aiken *et al.* 1992, Baralis *et al.* 1993). By storing previous repairs, however, with our approach we can check whether a repair action (on the path to a hypothetical database) would be undone by a repair derived from a state diagnosis. Roughly speaking, in this case then the path need not to be considered further since the net-effect of a resulting repair transaction containing an undo is guaranteed to be computed on another path. This result is due to the completeness of the hitting set algorithm for minimal state diagnoses and is shown to be sufficient for the soundness and completeness for enumerating minimal repair transactions (Gertz 1996).

The presented approach for enumerating possible repair transactions has several advantages. First, it provides a well structured method to compute all and only minimal repair transactions. Second, no changes on base relations are necessary; all operations are performed on the auxiliary relations reflecting changes on hypothetical databases. Third, it is possible to inspect derived repair transactions and to check whether or not a specific repair transaction reflects the user's intentions or the applications requirements. Thus the whole process of checking constraint violations, determining reasons and possible repair actions for violations as well as their subsequent effects, i.e. new constraint violations, becomes more visible.

A drawback of the approach, of course, is its computational complexity, which, depending on the number of state diagnoses determined for each possible database, can be exponential. This complexity, however, can be reduced by imposing restrictions on admissible state diagnoses and repair transactions. For this, repair strategies are used which are discussed in the next section.

5 REPAIR STRATEGIES

In the previous section we have presented a general enumeration schema which forms the lowest level of a repairing system that determines possible repair transactions for an inconsistent database. For nearly all applications, however, additional semantic knowledge about the application domain as well as requirements for the repair are present. For a repairing system to be applicable in practice such information must be utilized in order to reduce the search space for possible repair transactions.

The objective of this section now is to outline how the enumeration schema can be extended by ***repair strategies*** that allow to achieve well specified ***repair goals*** in an efficient way. These strategies should be as general as possible, i.e. independent of any specific application domain.

5.1 Aspects on Minimal Change

Up to now, the enumeration of possible repair transactions exclusively utilizes the concept of minimality in a set-oriented manner. The repair strategies which are discussed in the following also all have the common goal to keep "as much information as possible" in the database while determining possible repair transactions. The goal can analogously be formulated as to "perform changes as minimal as possible". In this context, the meaning of "as much information as possible" is subject to the interpretation of a repair transaction as well as to the interpretation of information. Typically, there are two natural questions which a user might want to know in case of a constraint violation by her/his transaction:

- What are possible maximal subsets of operations of her/his transaction which are consistent with the integrity constraints?
- What are possible minimal sets of operations she/he has to perform in addition to the violating transaction in order to obtain a consistent state, while keeping the original transaction?

The notion of a repair transaction up to now neither exploits the knowledge of the state before the violating transaction, nor the contents of the violating transaction itself, i.e. its associated operations. In the sequel we assume that the transaction T performed by the user is represented by a set of insertions into base relations and deletions from base relations, respectively. A transaction can also be considered as a set of positive and negative literals, denoting insertions and deletions, respectively.

Definition 13 (Minimal Undo) Let T be a violating user transaction resulting in an inconsistent database B. A repair transaction T' is said to be an *undo* of T in B if for each operation $ins_p(\mathbf{a}) \in T'$ there exists the operation $del_p(\mathbf{a}) \in T$ and if for each operation $del_q(\mathbf{c}) \in T'$ there exists the operation $ins_q(\mathbf{c}) \in T$; T' is a *minimal undo* iff no proper subset of T' is an undo, too.

In other words, a minimal undo of violating transaction T identifies a minimal (not necessarily unique) subset of operations in T which need to be undone in order to obtain a consistent state. These sets also identify those minimal subsets of operations from T which caused the violations in B. Since for a minimal undo only operations of the violating transaction T can be undone and the rest of the database contents should be kept, we get the following restriction for the computation of possible repair transactions: Only the positive and negative facts associated with the modifications represented by T can contribute to critical facts as the basis for a state diagnosis. From the computational point of view, each fact contained in a set h of critical facts can be removed from that set if the fact does not appear in T. This property can easily be checked in the function **hitting_sets**. It is obvious that this strategy reduces the search space for possible repair transactions since only sets of critical facts with few elements need to be considered for the computation of minimal state diagnoses; that is, the branching factor at each hypothetical database is reduced.

As the contrary to a minimal undo (which, in the worst case, is a complete rollback of the violating transaction), a consistent completion of violating transaction completely "keeps the effect" of the violating transaction.

Definition 14 (Consistent Completion) Let a violating transaction T be given. A repair transaction T' is said to be a *consistent completion* of T if there exists no operation in T' that undoes an operation of T.

In contrast to a minimal undo, for a consistent completion only the facts not inserted or deleted by the violating transaction can contribute to sets of critical facts. Consequently, all facts in sets of critical facts which were introduced by the violating transaction can be removed from these sets in advance.

For a violating transaction, however, not always a consistent completion exists. Assume, for instance, the constraint $\forall x, y : p(x, y) \implies x > y$ and a transaction $T = \{ ins_p(10, 20) \}$. The constraint is violated and the corresponding set of critical facts is $h = \{ p(10, 20) \}$. With regard to a consistent completion of T this fact cannot be a critical fact. Removing $p(10, 20)$ from h results in an empty set h and thus no repair action as a completion of T exists and which can be performed in order to restore the consistency.

The computation of both a minimal undo and a consistent completion can easily be integrated into the algorithm for computing minimal state diagnoses (Gertz 1996). Presuming that the user transaction is suitably represented by a set of positive and negative facts, in the function **hitting_sets** respective facts can be removed from each set of critical facts before applying the computation of hitting sets, respectively, state diagnoses for H_C.

The repair goal to keep "as much information as possible" can also be interpreted in terms of a *counting semantics* for repair transaction. Let $|T|$ denote the number of deletions and insertions contained in a repair transaction T.

Definition 15 A repair transaction T is said to be *counting minimal* if there exists no repair transaction T' such that $|T'| < |T|$, i.e. T' performs less modifications than T in order to obtain a consistent state.

Adopting the counting semantics as a criteria for enumerating possible repair transactions again drastically reduces the number of possible transactions. It resembles a uniform cost search where the path costs are determined by the number of operations leading to a hypothetical database. Once a repair transaction T has been determined at a level l, each hypothetical database at that level need not to be considered further if the overall number of modifications performed by repair actions leading to that state is greater than $|T|$.

It is important to note that the counting semantics can be used in combination with a minimal undo or a minimal completion of a violating transaction. Respective cardinality checks then need to be integrated in the procedure **add_repair_transaction**. The following figure illustrates the restrictiveness of repair transactions obtained by the possible combinations of repair strategies ("$a \rightarrow b$" means that b is more restrictive than a).

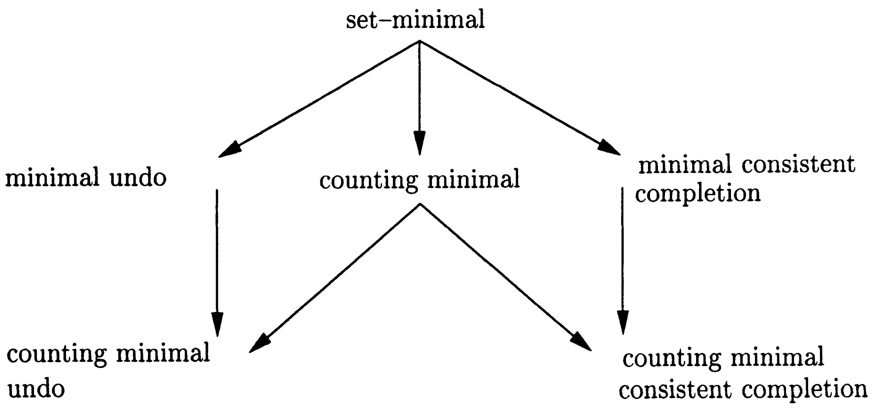
set–minimal
minimal undo
counting minimal
minimal consistent completion
counting minimal undo
counting minimal consistent completion

Figure 2 Restrictiveness of Repair Strategies

Set–minimality is the most permissive strategy that can be utilized for the enumeration of repair transactions. Counting minimal undoes and counting minimal consistent completions are the most restrictive strategies on minimal change that can be adopted to determine possible repair transactions for an inconsistent database.

5.2 Priorities

In this section we shortly sketch how *priorities* or *preferences* can be utilized for the repairing process. This consideration is needed because typically some information in the database is more important than others. This aspect, of course, should suitably be respected by repairing inconsistencies since it allows to describe a partial order on the consistent databases obtained by different repair transactions.

The main idea for a priority based repair is to tag the facts in an inconsistent database; that is, for a fixed $n \in \mathbb{N}$, a number $i \leq n$ is assigned to each positive and negative fact like, e.g., $p(a)^3$. Tagging a database in this way can be done on a tuple level; from the practical point of view, however, it is reasonable to tag facts belonging to the same relation with the same priority. A database obtained in this way is called a database *tagged* by n where n is the lowest priority given to a fact. Given a repair transaction T, the operations contained in T can be grouped according to the priorities of the affected facts:

Definition 16 Let B be an inconsistent database tagged by n. For a given repair transaction T for B, let T^i, $1 \leq i \leq n$, denote the subset of T defined as

$$\begin{aligned} T^i \ := \ & \{ins_p(\mathbf{a}) \mid ins_p(\mathbf{a}) \in T, \neg p(\mathbf{a})^i \in B\} \cup \\ & \{del_q(\mathbf{c}) \mid del_q(\mathbf{c}) \in T, q(\mathbf{c})^i \in B\}. \end{aligned}$$

Definition 17 (Priority Based Repair) Let B be an inconsistent database tagged by n. Let T_1 and T_2 be two minimal repair transactions for B. T_1 is said to accomplish a *better priority based repair* of B than T_2, denoted by $T_1 <_T T_2$, if for some $1 \leq i \leq n$ we have

$$|T_1^j| = |T_2^j| \text{ for each } j \in \{1, \dots, i-1\}, \text{ but } |T_1^i| < |T_2^i|.$$

T_1 and T_2 are said to be *equal* with respect to a priority based repair, denoted by $T_1 =_T T_2$, if we have $|T_1^i| = |T_2^i|$ for each $i \in \{1, \dots, n\}$.

Example 18 Consider the two repair transactions $T_1 = \{del_p(a)\}$ and $T_2 = \{del_q(c), del_q(d)\}$ where all facts in the relation p have the priority 1 and all facts in the relation q have the priority 2. Intuitively this means that the facts contained in the relation p are more important than those contained in q. According to Definition 17, the transaction T_2 accomplishes a better priority based repair since more important information is kept.

In (Gertz 1996) we have furthermore introduced the notion of *weighted priorities* where weights can be assigned to priorities. This allows to numerically compare transactions by evaluating weighted counts of their facts. A *transaction based tagging* furthermore is introduced that allows the user to assign priorities to the operations of her/his violating transaction. Hence it is possible to distinguish between important operations and less important operations.

Assuming that the facts in an inconsistent database are suitably tagged (and assumptions are made for the priorities of facts not contained in the database), checks for a better priority based repair can easily be included in the procedure **add_repair_transaction**.

5.3 Interaction with the Repair Process

The proposed repair strategies up to now all rely on the issue that the user specifies the desired repair goal in advance to the enumeration of possible repair transactions. Presuming a suitable environment, it should also be possible for the user to interact with the repair process. The most trivial way of interaction is that the user inspects the determined state diagnoses and repair actions level by level. She/he then selects those repair actions that reflect her/his intention at most and which should be considered further. This can lead to a drastic pruning of the enumeration tree for repair transactions.

Repair transactions contain only insertions and deletions of tuples, and facts to be inserted may contain null values (cf. Section 3). Given a repair action, the user can replace such null values by ordinary constants, thus obtaining repair transactions with complete information only. Instead of deleting facts as part of repair actions, it is possible to perform updates instead such that some attribute values of the tuples to delete are replaced by marked null values. In (Gertz 1996) we call this the *weak deletion approach.*

Finally the user can add further insertions and deletions to a derived repair action. This, of course, leads to the fact that the computed repair transactions are not minimal anymore. In all these cases of user interaction with the repair process, the proposed algorithm for enumerating repair actions and transactions nevertheless can be used as a guiding tool for the user.

6 SUMMARY

In this paper we have presented a new approach to repairing constraint violations in relational databases. This approach allows to enumerate possible minimal repair transactions for an inconsistent database. Using techniques from model-based diagnosis we have shown how state diagnoses and associated minimal repair actions can be determined. The advantage of the diagnostic approach is that simultaneous reasons (facts) for violations of different constraints can be computed, an important aspect not considered by other approaches to constraint enforcement. In particular the usage of marked null values provides a suitable means to handle missing tuples where not all attribute values are known.

Based on the diagnostic task we have presented a sound and complete algorithm for enumerating possible minimal repair transactions for an inconsistent database. The algorithm performs an iteration of diagnosis and repair of constraint violations in a breadth-first search manner through hypothetical databases. Using hypothetical databases, which can efficiently be represented by means of differential relations, the algorithm permits to query alternative possible repairs as well as alternative consistent result states.

The presented algorithm provides the formal basis for various repair strategies which can be individually imposed by the user previous to the enumeration of possible repair transactions. These strategies, which have not been considered before in related work, are not fixed but can be chosen and combined by the user for different inconsistent databases. The important feature of these strategies is that they all use the same enumeration schema and that they can easily be integrated into the presented algorithm. Extensions by further useful repair strategies and repair goals, of course, need to be discussed.

We have implemented a first prototype of the proposed system on top of the Oracle RDBMS. The diagnostic task is performed by a Prolog system coupled with the database system. Two applications, one from the domain of electrical engineering and one from the domain of rail traffic management, have been implemented using the repair system. Both enumerating repair actions and in particular the usage of repair strategies led to reasonable means for handling constraint violations.

In (Gertz 1996) we have shown that the presented approach can also be extended to temporal databases and deductive databases. Further work includes investigations into the repair of violations of dynamic integrity constraints and the application of the presented approach to object–oriented databases.

REFERENCES

S. Abiteboul, P. Kanellakis, G. Grahne (1991) On the Representation and Querying of Sets of Possible Worlds. *Theoretical Computer Science* **78**, 159–187.

A. Aiken, J. Widom, J. M. Hellerstein (1992) Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In M. Stonebraker (ed.), *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, 59–68, ACM Press, New York.

A. P. Buchmann, R. S. Carrera, M. A. Vazquez-Galindo (1991) Handling Constraints and their Exceptions: An Attached Constraint Handler for Object-Oriented CAD Databases. In K. R. Dittrich, U. Dayal, A. P. Buchmann (eds.), *On Object-Oriented Database Systems*, 65–83, Topics in Information Systems. Springer-Verlag, Berlin.

E. Baralis, S. Ceri, J. Widom (1993) Better Termination Analysis for Active Databases. In N. W. Paton, M. H. Williams (eds.), *Rules in Database Systems, Proceedings of the 1st Int. Workshop in Edinburgh, 1993*, 163–179, Workshops in Computing, Springer-Verlag, London.

S. Ceri, P. Fraternali, S. Paraboschi, L. Tanca (1994) Automatic Generation of Production Rules for Integrity Maintenance. *ACM Transactions on*

Database Systems **19:3** (September 1994), 367–422.

M. Celma, C. Garcia, L. Mota, H. Decker (1994) Comparing and Synthesizing Integrity Checking Methods for Deductive Databases. In M. Rusinkiewicz (ed.), *Proc. of the 10th IEEE CS International Conference on Data Engineering*, 214–222, IEEE Computer Society Press.

K. Eswaran, D. Chamberlin (1975) Functional Specifications of a Subsystem for Data Base Integrity. In D. Kerr (ed.), *Proc. of the 1st International Conference on Very Large Data Bases*, 48–68, Morgan Kaufmann Publishers, Los Altos, CA.

J. L. Encarnacao, P. C. Lockemann (eds.) (1990) *Engineering Databases.* Springer, Berlin, 1990.

P. Fraternali, S. Paraboschi (1993) A Review of Repairing Techniques for Integrity Maintenance. In N. W. Paton, M. H. Williams (eds.), *Rules in Database Systems, Proc. of the 1st Int. Workshop in Edinburgh*, 333–346, Workshops in Computing, Springer-Verlag, London.

P. W. Grefen, P. M. Apers (1993) Integrity Control in Relational Database Systems - An Overview. *Data & Knowledge Engineering* **10:2** (March 1993), 187–223.

M. Gertz (1994) Specifying Reactive Integrity Control for Active Databases. In J. Widom, S. Chakravarthy (eds.), *RIDE'94 - Fourth International Workshop on Research Issues in Data Engineering*, 62–70, IEEE Computer Society Press, Los Alamitos, CA.

M. Gertz (1996) Diagnosis and Repair of Constraint Violations in Database Systems, PhD Thesis, University of Hannover, Hannover, July 1996. (Table of contents available under `ftp.informatik.uni-hannover.de/papers/1996/Ger96a.ps.gz`)

M. Gertz, U. W. Lipeck (1995) A Diagnostic Approach to Repairing Constraint Violations in Databases. In W. Nejdl (ed.), *Sixth International Workshop on Principles of Diagnosis (DX'95), Working Papers, October 2-4, Goslar, Germany*, 65–72, University of Hannover, Hannover.

W. Hamscher, L. Console, J. de Kleer (1992) *Readings in Model-Based Diagnosis.* Morgan Kaufmann Publishers, San Mateo, CA.

M. Hammer, D. McLeod (1975) Semantic Integrity in a Relational Database System. In D. Kerr (ed.), *Proc. of the 1st International Conference on Very Large Data Bases*, 25–47, Morgan Kaufmann Publishers, Los Altos, CA.

T. Imielinski, W. J. Lipski (1984) Incomplete Information in Relational Databases. *Journal of the ACM* **31:4** (October 1984), 761–791.

M. Jeusfeld, M. Jarke (1991) From Relational to Object-Oriented Integrity Simplification. In C. Delobel, M. Kifer, Y. Masunaga (eds.), *Deductive and Object-Oriented Databases — Proceedings DOOD'91*, 460–477, Lecture Notes in Computer Science 566, Springer-Verlag, Berlin.

J. W. Lloyd, R. W. Topor (1984) Making Prolog More Expressive. *Journal of Logic Programming*, 225–240.

M. Morgenstern, A. Borgida, C. Lassez, D. Maier, G. Wiederhold (1986) Constraint-Based Systems: Knowledge About Data. In L. Kerschberg (ed.), *Expert Database Systems: Proc. from the First International Conference*, 23–43, Benjamin/Cummings, Menlo Park, CA.

G. Moerkotte, P. C. Lockemann (1991) Reactive Consistency Control in Deductive Databases. *ACM Transactions on Database Systems* **16:4** (December 1991), 670–702.

R. Reiter (1984) Towards a Logical Reconstruction of Relational Database Theory. In M. L. Brodie, J. Mylopoulos, J. W. Schmidt (eds.), *On Conceptual Modelling*, 191–238. Springer-Verlag, New York.

R. Reiter (1986) A Sound and Sometimes Complete Query Evaluation Algorithm for Relational Databases with Null Values. *Journal of the ACM* **33:2** (April 1986), 349–370.

R. Reiter (1987) A Theory of Diagnosis from First Principles. *Artificial Intelligence* **32**, 57–95. Also in (Hamscher *at al.* 1992).

K.-D. Schewe, B. Thalheim (1994) Achieving Consistency in Active Databases. In J. Widom, S. Chakravarthy (eds.), *RIDE'94 - Fourth International Workshop on Research Issues in Data Engineering*, 71–76, IEEE Computer Society Press, Los Alamitos, CA.

S. D. Urban, M. Desiderio (1992) CONTEXT: A CONstrainT EXplanation Tool. *Data & Knowledge Engineering* **8:2** (May 1992), 153–183.

M. Y. Vardi (1986) On the Integrity of Databases with Incomplete Information. In *Proc. of the 5th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 252–266, ACM Press, New York.

J. Widom (1994) Database Constraint Management *Data Engineering* **17:2** (June 1994), (Special Issue), 2–51.

7 BIOGRAPHY

Michael Gertz is a teaching and research assistant at the Database and Information Systems Group of the Institute for Informatics at the University of Hannover. His research interests include temporal and active databases, multidatabase systems, logical and physical database design, database integrity, database models and languages, and the development of database administration tools.

Udo Lipeck is Professor for Databases and Informations Systems at the University of Hannover. His research interests include formal system specifications (algebraic, logic-based, and object-oriented approaches), non-standard logics (in particular temporal and default logics), conceptual database design, database integrity, deductive databases, and tools for database design and administration.