Integrity Enforcement in Relational Systems with Active and Deductive Capabilities

Laura Mota-Herranz
Dept. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia
lmota@dsic.upv.es

Matilde Celma-Giménez
Dept. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia
mcelma@dsic.upv.es

Abstract
In this paper, we propose a method for integrity enforcement in relational databases with view definition. The method uses the production rule mechanism provided by active database systems. It generates a set of rules at schema specification time; this set of rules repairs the inconsistency produced by some user transaction. The event part of the rules consists of a simple update operation that can violate an integrity constraint; the condition part verifies whether the event actually induces the violation; and, finally the action part is a set of transactions that can repair the violation. The new approach is simple and offers solutions to the problem of missing values and recursive views. The method is sound for all the cases and it never loops.

INTRODUCTION
The main feature of an active system is that the system itself performs certain operations automatically in response to certain events or when certain conditions are satisfied. This behaviour is defined in Event-Condition-Action rules (or ECA-rules) that are usually specified by users or database administrators. On the other hand, the main feature of a deductive system (i.e. a relational system with view definition) is its ability to derive information not explicitly stored using deductive rules. In this paper, we assume a relational system which has active and deductive capabilities ([6]) i.e. a SQL3 system ([8]).

An integrity constraint is a condition that the database is required to satisfy at all times, so a database is not allowed to reach a state in which some integrity constraint is violated [1]. When a transaction containing one or more consistency violating updates occurs, the classical treatment is to roll back the transaction in its entirety. This simple rejection solution is obviously unsatisfactory for most real databases. Instead, the system could try to compensate the violating updates with further updates [2, 4]. These extra updates could be prepared at compile-time, i.e. when the integrity constraint is defined, instantiated at run-time and used to extend the user transaction in order to enforce the database consistency.

Following this approach, usually known as Integrity Enforcement, we propose a method that generates a set of ECA-rules such that the event part of the rules consists of a simple update operation that can violate an integrity constraint; the condition part verifies whether the event actually induces the violation; and, finally the action part is a set of transactions that can repair the violation. These rules, from now on called repair rules, are designed to bring the database into a consistent state, as close as possible to that intended by the transaction supplier.

PROBLEM STATEMENT
Before introducing the problem, we give some assumptions that will be used later [3]:

1. Let \((R, IC)\) be a deductive database schema where \(R\) is a first order language, with disjoint sets of base predicates (or base relations) and derived predicates (or views), and \(IC = \{W_i : W_i\) is a closed formula in \(R, 1 \leq i \leq n\}\) is the set of integrity constraints.

2. Let \(D\) be a database state defined as \(D = EDB \cup IDB\) where \(EDB\) is the extensional database (extensions of base relations) and \(IDB\) is the intensional database (view definitions): \(EDB \subseteq \{A : A\) is a ground atom over a base predicate\} and \(IDB \subseteq \{A \leftarrow L_1 \land \ldots \land L_n : A\) is an atom over a derived predicate and \(L_i\) is a literal\}. And let \(comp(D)\) be the first order theory in \(R\) which represents the state \(D\) in the semantics of the completion.

3. Let \(\{\leftarrow inc_i : inc_i \leftarrow \neg W_i, 1 \leq i \leq n\}\) be the set of integrity constraints in denial form where \(inc_i (1 \leq i \leq n)\) is a predicate symbol of zero arity that does not occur in the database nor in the constraints.

4. Let \(D \cup \{inc_i \leftarrow \neg W_i : 1 \leq i \leq n\}\) be stratified, allowed and strict, (from now on \(D\) will refer to a stratified, allowed and strict normal form of \(D \cup \{inc_i \leftarrow \neg W_i : 1 \leq i \leq n\}\)). Note that if \(D\) is strict there are not two predicate symbols \(p\) and \(q\) such that \(p\) depends on \(q\) both positively and negatively.

5. Let \(D\) be a database state satisfying every integrity constraint \(W_i\) in \(IC\). This means that \(comp(D) \models W_i\).

6. Let a transaction be a set of ground update operations over base predicates. This means that the database will change only due to the insertion (ins) or deletion (del) in the base predicates.

When a transaction \(T\) occurs, the state \(D\) changes to another state \(D'\) which may not satisfy all the integrity constraints in \(IC\). The traditional response of a deductive system to these incorrect transactions is to roll back the transaction. Instead, we propose an alternative approach, in which the database system reacts autonomously to inconsistencies by triggering a set of repair actions capable of progressive elimination of constraint violations until a state \(D''\) is
reached in which every integrity constraint is satisfied or no new repair can be made. When a repair is impossible, an abort is forced yielding state $D$. These repair actions can be encoded, as we will show, in ECA-rules that we have called repair rules.

If $W_i$ is an integrity constraint whose denial form is $\neg \text{inc}$ and taking previous assumptions (from 1 to 6) into account, integrity violation can be formulated in the following way: “$D$ violates the integrity constraint $W_i$ if $\text{comp}(D) \models \text{inc}$”, that is, if the transaction $T$ has inserted the inconsistency atom $\text{inc}$. Therefore the problem of automatically generating repair rules for the constraint $W_i$ can be viewed as a special case of the problem of updating a deductive database $\{(4,5)\}$ and can be solved in two steps:

- The first step consists of computing the set of operations over base predicates that can induce an insertion for the inconsistency atom $\text{inc}$.
- The second step consists of computing a transaction $T^*$, for each operation of the previous set, that can induce the deletion of the inconsistency atom $\text{inc}$.

Therefore, if $O$ is an operation computed in the first step and $T^*$ is its associated transaction computed in the second step, then the following ECA-rule can repair the violation of the integrity constraint $W_i$:

Event: $O$
Condition: $\text{inc}$
Action: $T^*$

This rule must be read as follows: “when the operation $O$ matches an operation in the user transaction, and if $\text{inc}$ is satisfied, then the transaction $T^*$ will be executed”.

For example, let $D$ be a database with the following deductive rules:

1. $p(x) \leftarrow t(x) \land \neg v(x)$
2. $p(x) \leftarrow u(x)$
3. $\text{inc} \leftarrow p(x) \land \neg q(x)$

Where the integrity constraint $W: \forall x \ (p(x) \rightarrow q(x))$ is represented, in denial form, by the body rule 3. It is easy to determine that deletions from predicate $v$ cause the violation of the constraint $W$ by inserting into the predicate $\text{inc}$ if the condition $t(x) \land \neg q(x)$ is satisfied after the operation, and also it is easy to determine that insertions into predicate $q$ can repair the integrity by inducing deletions from the predicate $\text{inc}$. Then we can generate the following ECA-rule for repairing violations over the integrity constraint represented by deductive rule 3:

Event: $\text{del}(v(x))$
Condition: $t(x) \land \neg q(x)$
Action: $\text{ins}(q(x))$

Note that the condition of the rule allows a simplified evaluation of the inconsistency atom $\text{inc}$ since it takes into account the path through which the violation has been induced.

Following these ideas, we are going to illustrate how we can automatically generate the repair rules at schema definition time.

**GENERATION OF THE EVENT AND THE CONDITION OF REPAIR RULES**

In order to obtain the event and the condition of the repair rules, we are going to define the set of all possible updates (Induced Updates) that operations over base predicates can induce over derived predicates. These updates are defined by two sets, $\mathcal{IU}_{\text{POS}}$ and $\mathcal{IU}_{\text{NEG}}$, which catch information respectively about the insertions and deletions over derived predicates as well as over the paths through which the updates are induced. These sets are generated from the deductive rules in $\mathcal{IDB}$ and their elements are 4-tuples, $(P, O, C, L)$, where $P$ (the induced update) is a derived atom, $O$ (the base update) is an operation of the form $\text{ins}(Q)$ or $\text{del}(Q)$, $Q$ being a base atom, $C$ (the updating path) is a conjunction of literals or the special predicate $\text{true}$ and $L$ (the predicate dependencies in the path) is a list of the form $\{q_1 \mid d_{r_1}, \ldots, q_{ic} \mid d_{r_2}\}$ where $q_1$ is a predicate and $d_{r_2}$ is the label of a deductive rule. The inductive definition of these sets is the following:

- $\mathcal{IU}_{\text{POS}, \theta} = \{(P, \text{ins}(A), L_1 \land \ldots \land L_{i-1} \land L_{i+1} \land \ldots \land L_m \ (\text{resp. true}), \{(q, dr_1) \mid (P \leftarrow L_1 \land \ldots \land L_{i-1} \land A \land L_{i+1} \land \ldots \land L_m) \text{ is a variant of a deductive rule with label } dr_1, A \text{ is a base atom with } q \text{ as predicate and } m > 1 \ (\text{resp. } m = 1)\}\} \\
\bigcup \{(P, \text{del}(A), L_1 \land \ldots \land L_{i-1} \land L_{i+1} \land \ldots \land L_m \ (\text{resp. true}), \{(q, dr_1) \mid (P \leftarrow L_1 \land \ldots \land L_{i-1} \land \neg A \land L_{i+1} \land \ldots \land L_m) \text{ is a variant of a deductive rule with label } dr_1, A \text{ is a base atom with } q \text{ as predicate and } m > 1 \ (\text{resp. } m = 1)\}\}\}

- $\mathcal{IU}_{\text{NEG}, \theta} = \{(P, \text{ins}(A), \text{true}, \{(q, dr_1) \mid (P \leftarrow L_1 \land \ldots \land L_{i-1} \land A \land L_{i+1} \land \ldots \land L_m) \text{ is a variant of a deductive rule with label } dr_1, A \text{ is a base atom with } q \text{ as predicate}\}\} \\
\bigcup \{(P, \text{del}(A), \text{true}, \{(q, dr_1) \mid (P \leftarrow L_1 \land \ldots \land L_{i-1} \land \neg A \land L_{i+1} \land \ldots \land L_m) \text{ is a variant of a deductive rule with label } dr_1, A \text{ is a base atom with } q \text{ as predicate}\}\}\}

- $\mathcal{IU}_{\text{POS}, \theta+1} = \{(P_0, \text{E}0, (C \land L_1 \land \ldots \land L_{i-1} \land L_{i+1} \land \ldots \land L_m) \theta, L \land \{(q, dr_1) \mid (P \leftarrow L_1 \land \ldots \land L_{i-1} \land A \land L_{i+1} \land \ldots \land L_m) \text{ is a variant of a deductive rule with label } dr_1, B \text{ is an atom with } q \text{ as predicate, } (A, E, C, L) \in \mathcal{IU}_{\text{POS}, \theta} \text{ and } \theta = \text{mgu}(A,B)\}\} \\
\bigcup \{(P_0, \text{E}0, (L_1 \land \ldots \land L_{i-1} \land \neg B \land L_{i+1} \land \ldots \land L_m) \theta, L \land \{(q, dr_1) \mid (P \leftarrow L_1 \land \ldots \land L_{i-1} \land \neg B \land L_{i+1} \land \ldots \land L_m) \text{ is a variant of a deductive rule with label } dr_1, B \text{ is an atom with } q \text{ as predicate, } (A, E, \text{true}, L) \in \mathcal{IU}_{\text{NEG}, \theta} \text{ and } \theta = \text{mgu}(A,B)\}\}

- $\mathcal{IU}_{\text{NEG}, \theta+1} = \{(P_0, \text{E}0, \text{true}, L \land \{(q, dr_1) \mid (P \leftarrow L_1 \land \ldots \land L_{i-1} \land B \land L_{i+1} \land \ldots \land L_m) \text{ is a variant of a deductive rule with label } dr_1, B \text{ is}...)
an atom with \( q \) as predicate, \((A, E, \text{true}, L) \in IU_{\neg E,q}\) and \( \theta = \text{mgu}(A, B)\)

\[
\bigcup \{(P\theta, E\theta, \text{true}, L + [(g, dr)]) \mid P \leftarrow L_1 \land \ldots \land L_{i-1} \land \neg B \land L_{i+1} \land \ldots \land L_m \text{ is a variant of a deductive rule with label } dr, \ B \text{ is an atom with } q \text{ as predicate}, (A, E, C, L) \in IU_{PO,S} and \theta = \text{mgu}(A, B)\}.
\]

It is important to notice that when an element of \( IU_{\neg E,q-1}\), say \((P, O, \text{true}, L)\), generates an element of \( IU_{PO,S}\) using the deductive rule \( dr\), the conjunction of this new element should include all the literals in the body of \( dr\) in order to ensure that there is no other derivation for \( P\) (see the previous definition). This fact implies that it is not necessary to store the updating path for the elements of \( IU_{\neg E,q}\) because when these elements generate some induced insertion, this path will not be taken into account.

The generation of these sets will finish in a level \( j (j \geq 0)\) such that: \( IU_{PO,S} = \emptyset \) and \( IU_{\neg E,q} = \emptyset \). Such value for \( j \) will always exist if the database has not recursive predicates (i.e. it is hierarchical) but not in the case of stratified databases (i.e. there can be recursive predicates but never through negation). In the latter case, to solve the problem it is necessary to correct the definition of \( IU_{PO,S}\) and \( IU_{\neg E,q}\) in order to detect those induced updates which subsume other induced updates already included in the previous level and to substitute them for other more general updates, therefore:

1. If \((P, O, p(q(x_1), \ldots, x_d)), C, L) \in IU_{PO,S}\) (resp. \((P, O, p(q(x_1), \ldots, x_d)), C, L) \in IU_{\neg E,q}\) where \( O \) is either \( \text{ins} \) or \( \text{del} \) and \( q \) appears in \( L\), then the element must be substituted by \((P, O, p(q(y_1), \ldots, y_d)), P, L)\) (resp. \((P, O, p(q(y_1), \ldots, y_d)), \text{true}, L)\) where \( \{y_1, \ldots, y_d\}\) are new variables.

2. An element of the form \((P, O, p(q(x_1), \ldots, x_d)), C, L)\) where some \((q, dr)\) occurs in \( L\) more than twice will neither be included in \( IU_{PO,S}\) nor in \( IU_{\neg E,q}\).

The definition of the sets taking into account the points 1 and 2 will always finish since we avoid the infinite generation of elements produced by the presence of recursive deductive rules in the schema. Finally, the complete \textit{Induced Updates} are:

\[
IU_{POS} = \bigcup_{n \geq 0} IU_{POS,n} \quad IU_{NEG} = \bigcup_{n > 0} IU_{NEG,n}
\]

The algorithm shown below obtains the repair rules, still without action part, from the \( IU_{POS}\) set:

\textbf{ALGORITHM Generation_of_Repair_Rules}

\begin{itemize}
  \item \textbf{INPUT} \( R: \) set of Repair Rules without action;
  \item \textbf{OUTPUT} \( R: \) set of Repair Rules without action;
  \item \textbf{BEGIN}
    \item \textbf{IF} the predicate of \( P\) is the inconsistency predicate \( inc_d\)
    \item \textbf{THEN} \( R := R \cup \)
\end{itemize}

\textbf{GENERATION OF THE ACTION OF REPAIR RULES}

Once we have shown how it is possible to generate the event and the condition of a repair rule, we are going to illustrate how to find the action. This action will be generated from the condition of the rule. Let us suppose that this condition consists of one derived literal (later we will consider the general case), then if the literal is \( \neg A\) (resp. \( A\)), we have to find the transactions over base literals that can falsify the condition of the rule, by inducing insertions (resp. deletions) in the atom \( A\). These transactions will be the action of the rule.

Let the literal \( \neg p(x)\) be a rule condition and let the deductive rules \( p(x) \leftarrow s(x) \land \neg v(x)\) and \( p(x) \leftarrow t(x) \land u(x)\) be the definition of the predicate \( p\) where \( s, v, t, \) and \( u\) are base predicates. There are four possible transactions for inserting an instance of \( p(x)\):

\begin{itemize}
  \item \textbf{Transaction} \( TIns1(x)\):
    \begin{itemize}
      \item \textbf{IF} \( \neg s(x)\) \text{ \textbf{THEN}} \( \text{ins}(s(x))\)
      \item \textbf{IF} \( v(x)\) \text{ \textbf{THEN}} \( \text{del}(v(x))\)
    \end{itemize}
  \end{itemize}

\begin{itemize}
  \item \textbf{END_IF}.
\end{itemize}

\begin{itemize}
  \item \textbf{Transaction} \( TIns2(x)\):
    \begin{itemize}
      \item \textbf{IF} \( v(x)\) \text{ \textbf{THEN}} \( \text{del}(v(x))\)
      \item \textbf{IF} \( \neg s(x)\) \text{ \textbf{THEN}} \( \text{ins}(s(x))\)
    \end{itemize}
  \end{itemize}

\begin{itemize}
  \item \textbf{END_IF}.
\end{itemize}

\begin{itemize}
  \item \textbf{Transaction} \( TIns3(x)\):
    \begin{itemize}
      \item \textbf{IF} \( \neg t(x)\) \text{ \textbf{THEN}} \( \text{ins}(t(x))\)
      \item \textbf{IF} \( \neg u(x)\) \text{ \textbf{THEN}} \( \text{ins}(u(x))\)
    \end{itemize}
  \end{itemize}

\begin{itemize}
  \item \textbf{END_IF}.
\end{itemize}

\begin{itemize}
  \item \textbf{Transaction} \( TIns4(x)\):
    \begin{itemize}
      \item \textbf{IF} \( \neg u(x)\) \text{ \textbf{THEN}} \( \text{ins}(u(x))\)
      \item \textbf{IF} \( \neg t(x)\) \text{ \textbf{THEN}} \( \text{ins}(t(x))\)
    \end{itemize}
  \end{itemize}

\begin{itemize}
  \item \textbf{END_IF}.
\end{itemize}

These four transactions can be represented as a tree, shown in figure 1, where the root node represents the update request. In this tree, each path from the root to a leaf represents a way of inserting an instance of \( p(x)\). The two paths on the left (resp. on the right) insert an instance of \( p(x)\) using the first (resp. second) deductive rule. The conditions of the nodes represent the conditions of the transaction. Therefore, in this tree, each path represents a transaction.

Transaction \( TIns1\) is represented by the path \( N_0-N_1-N_5\), transaction \( TIns2\) by the path \( N_0-N_2-N_6\), transaction \( TIns3\) by the path \( N_0-N_3-N_7\), and, finally, transaction \( TIns4\) is represented by the path \( N_0-N_4-N_8\). In order to clarify the meaning of these paths, let us process the path \( N_0-N_1-N_5\) which shows how to insert an instance of \( p(x)\) using the first deductive rule \( p(x) \leftarrow s(x) \land \neg v(x)\). Node \( N_1\) specifies that if the first condition of this node,
\[ -s(x), \text{is true for the same instance then the node operation,}\]
\[ ins(s(x)), \text{must be executed. The second condition of N1 checks if it is necessary to visit the next node, that is, if}\]
\[ -v(x) \text{ is true the insertion of the instance of } p(x) \text{ has been achieved; else, i.e. } -v(x) \text{ is false, the processing continue in node N5 executing the operation, } del(v(x)).\]

\[ \text{Transaction } T_{Del1}(x) : \]
\[ \text{IF } s(x) \land -v(x) \text{ THEN del}(s(x)) \text{ END IF; } \]
\[ \text{IF } t(x) \land u(x) \text{ THEN del}(t(x)) \text{ END IF.} \]

\[ \text{Transaction } T_{Del2}(x) : \]
\[ \text{IF } s(x) \land -v(x) \text{ THEN ins}(v(x)) \text{ END IF; } \]
\[ \text{IF } t(x) \land u(x) \text{ THEN del}(t(x)) \text{ END IF.} \]

\[ \text{Transaction } T_{Del3}(x) : \]
\[ \text{IF } s(x) \land -v(x) \text{ THEN del}(s(x)) \text{ END IF; } \]
\[ \text{IF } t(x) \land u(x) \text{ THEN del}(u(x)) \text{ END IF.} \]

\[ \text{Transaction } T_{Del4}(x) : \]
\[ \text{IF } s(x) \land -v(x) \text{ THEN ins}(v(x)) \text{ END IF; } \]
\[ \text{IF } t(x) \land u(x) \text{ THEN del}(u(x)) \text{ END IF.} \]

These transactions can also be represented by the tree shown in figure 2, where the root node represents again the update request. In this tree, each path from the root to a leaf represents a way of deleting a derivation of an instance of \( p(x) \). The two paths on the left (resp. on the right) delete a derivation of an instance of \( p(x) \) which was derivable through the first (resp. second) deductive rule. Again, the conditions of the nodes represent the conditions of the transaction. In this case, a transaction for deleting instances of \( p(x) \) must include a path for each deductive rule defining the view predicate \( p \). Therefore, transaction \( T_{Del1} \) is represented by the paths N0-N1 and N0-N3; transaction \( T_{Del2} \) is represented by N0-N2 and N0-N3; transaction \( T_{Del3} \) is represented by N0-N1 and N0-N4; and, transaction \( T_{Del4} \) is represented by the paths N0-N2 and N0-N3.

Let us show now how to delete an instance of \( p(x) \) processing the paths N0-N1 (in order to ensure that the instance is now not derivable through the first deductive rule) and N0-N3 (in order to ensure that the instance is now not derivable through the second deductive rule). Node N1 specifies that, if the first condition of this node, \( s(x) \land -v(x) \), is true for the same instance then the operation \( del(s(x)) \) must be executed. Node N3 specifies that, if the first condition of this node, \( t(x) \land u(x) \), is true, the operation, \( del(t(x)) \) must be executed.

We will refer to these trees as \( T \)-trees (Transaction trees) [5]. We define this structure more formally in the following section.

**T-trees**

A node of a \( T \)-tree is a tuple with three components, (Pre, Op, Post) where Pre (resp. Post) is the predicate true or a conjunction of literals called Pre-Condition (resp. Post-Condition), Op is \( O(A) \) where \( O \) is either \( ins \) or \( del \) and \( A \) is an atom, or \( Op \) is the special operation abort. The nodes in a \( T \)-tree are linked by hyperarcs. A hyperarc links one node, called predecessor node, with a set of nodes which we call successor clan. A node can have several successor clans.

**Figure 2: Representation of the deletion transactions for \( p(x) \)**

**Figure 3: T-tree with one clan**

**Figure 4: T-tree with two clans**

Processing of a \( T \)-tree is done at run time and is based on the following rules:

1. After visiting a node, it is necessary to pass through a subtree for each clan which is linked to the node.
2. It is possible to go from one node to a node of one of its successor clans whose Pre-Condition holds in the current database state.
3. The process of a path ends when a node is reached whose operation is over a base predicate and whose Post-Condition holds in the current database state.

As we have said before, the action of a repair rule is a \( T \)-tree, which is obtained from the condition of the rule. Let
1. If $n = 1$ and $L_i \neq \text{true}$ then the T-tree has only one node: $(true, abort, true)$.
2. If $n = 1$ and $L_i \neq \text{true}$ then the T-tree has only one node: $(true, del(L_i), true)$.
3. If $n = 1$ and $L_i = \text{true}$ then the T-tree will be generated from the root node: $(true, del(L_i), true)$.
4. If $n = 1$ and $L_i = \text{false}$ then the T-tree will be generated from the root node: $(true, del(L_i), true)$.
5. If $n > 1$ then the T-tree will be generated from the root node: $(true, del(Cond(x_1, ..., x_m)), true)$ where $\{x_1, ..., x_m\}$ are all the variables that occur in $L_1 \wedge ... \wedge L_n$ and $Cond$ is a new predicate defined by the auxiliary deductive rule $Cond(x_1, ..., x_m) \leftarrow L_1 \wedge ... \wedge L_n$.

In the last three cases, the complete T-tree is generated from the root node. Let $A^*$ be the atom $A$ where every variable has been marked (a marked variable behaves as a constant when unifying) and let a restricted $mgu$ be a substitution $\{v_1/l_1, ..., v_n/l_n\}$ where $v_i$ is a non-marked variable and $l_i$ is a constant or a marked variable. Then the set of successor clans of the node $Pretree, Op, Post$ in a T-tree is generated as follows:

1. If $Op = \text{abort}$ then the node has no successor clans.
2. If $Op = O(A), A$ is a base atom and $Post = true$, then the node has no successor clans.
3. If $Op = O(A), A$ is a base atom and $Post$ consists of one literal, $Post = M$, then the node has only one successor clan, $C$, with only one node:
   a) If $M = \neg N$ where $N$ is an atom, then $C = \{(true, del(N), true)\}$.
   b) If $M = N$ where $N$ is an atom, then $C = \{(true, ins(N), true)\}$.
4. If $Op = O(A), A$ is a base atom and $Post = A_1 \wedge ... \wedge A_n \wedge \neg B_1 \wedge ... \wedge \neg B_m$ where $n + m > 1$ and both $A_i (1 \leq i \leq n)$ and $B_j (1 \leq j \leq m)$ are atoms, then the node has only one successor clan, $C$, with $n + m$ nodes:
   $C = \{(-A_i, ins(A_i), A_1 \wedge ... \wedge A_{k-1} \wedge A_{k+1} \wedge ... \wedge A_n) \wedge \neg B_1 \wedge ... \wedge \neg B_m) (1 \leq i \leq n)\}$
   $\cup \{(B_j, del(B_j), A_1 \wedge ... \wedge A_n \wedge \neg B_1 \wedge ... \wedge \neg B_{j-1} \wedge \neg B_{j+1} \wedge ... \wedge \neg B_m) (1 \leq j \leq m)\}$.
5. If $Op = O(A), A$ is a derived atom and there are $p$ deductive rules (variants) $H_i \leftarrow A_{i1} \wedge ... \wedge A_{ik} \wedge \neg B_{i1} \wedge ... \wedge \neg B_{im}$ whose heads unify with $A^*$ through the restricted $mgu$'s $\theta_i$ (where $1 \leq i \leq k$, $n + m_i \geq 1$ and both $A_{ij} (1 \leq i \leq n_i)$ and $B_{ij} (1 \leq j \leq m_i)$ are atoms), then:
   a) If $Op = \text{ins}$, the node has only one successor clan, $C$, with the $\sum_{i=1}^{k}(n_i + m_i)$ nodes:
      $C = \{(-A_i, \theta_i, ins(A_i, \theta_i)), (Post \wedge A_{i1} \wedge ... \wedge A_{i(k-1)} \wedge A_{i(k+1)} \wedge ... \wedge A_{ik} \wedge \neg B_{i1} \wedge ... \wedge \neg B_{im}) (1 \leq i \leq n_i) \cup \{(B_j, del(B_j), (Post \wedge A_{k1} \wedge ... \wedge A_{k(k-1)} \wedge ... \wedge A_{kn} \wedge \neg B_{k1} \wedge ... \wedge \neg B_{km}) (1 \leq j \leq m_i) \wedge (1 \leq k \leq p))\}$.
   b) If $Op = \text{del}$, then the node has $p$ successor clans, $C_i (1 \leq i \leq p)$. Clan $C_i$ has $(n_i + m_i)$ nodes:
      $C_i = \{(A_{i1} \wedge ... \wedge A_{ik} \wedge \neg B_{i1} \wedge ... \wedge \neg B_{im}) \wedge \theta_i, del(A_{i1}, \theta_i), Post \theta_i) (1 \leq i \leq n_i) \cup \{(A_{k1} \wedge ... \wedge A_{kn} \wedge \neg B_{k1} \wedge ... \wedge \neg B_{km}) \wedge \theta_i, ins(B_j, \theta_i), Post \theta_i) (1 \leq j \leq m_i)\}.

6. If $Op = O(A), A$ is a derived atom and there are not deductive rules whose heads unify with $A^*$ then the node has only one successor clan, $C$, with one node: $C = \{(true, abort, true)\}$.

Once we have shown how to obtain a T-tree it is necessary to define how to process it. The following algorithm defines this processing:

**Algorithm T-tree Process**

**Input:**
- $Pre_{root}$, $Op_{root}$, $Post_{root}$: T-tree root;
- $D_{in}$, $D_{initial}$: database states;

**Output:**
- $D_{out}$: database state;

**Begin**

IF $Op_{root} = O^0(A^0)$

THEN

| IF the predicate of $A^0$ is derived |
| THEN |
| | FOR EACH successor clan of the root, $C$, DO |
| | | | Choose a node of $C$, $(Pre^i, O^i(A^0), Post^i)$ |
| | | | such that $Pre^i$ holds in $D_{in}$ |
| | | | $T$-tree Process $(Pre^i, O^i(A^0), Post^i)$ |
| | | | $D_{in}$, $D_{initial}$, $D_{out}$ |
| | | END_IF;
| | IF $Post_{root}$ holds in $D_{out}$ |
| | THEN Stop |
| ELSE |
| | | $D_{out} := D_{out}$ |
| | | | Choose a node from the successor |
| | | | clan, $(Pre^i, O^i(A^0), Post^i)$ such that |
| | | | $Pre^i$ holds in $D_{in}$ |
| | | | $T$-tree Process $(Pre^i, O^i(A^0), Post^i)$ |
| | | | $D_{in}$, $D_{initial}$, $D_{out}$ |
| | | END_IF;
| ELSE |
| | $Op_{root} := \text{abort}$ |
| | $D_{out} := D_{initial}$ |
| END_IF;

**End**
The problem of missing values
A variable in a deductive rule is called existential if it does not appear in the head of the rule. In this section, we are going to consider the presence of these variables in the deductive rules. First, we illustrate the problem, usually called the problem of missing values. Let $p$ be a predicate defined by the deductive rule $p(x) \leftarrow s(x,y) \land t(x,y)$, the T-tree for inserting instances of $p(x)$, is the following:

![T-tree for inserting instances of p(x)](image)

Figure 5: T-tree for inserting instances of $p(x)$ in a database with existential variables

The previous algorithm is not able to process the nodes $N1$ and $N2$, because their operations are not fully instantiated when the processing algorithm reaches them. This problem can appear in nodes with different properties (it is not possible to show an example for each case due to lack of space); in each case we propose a solution. The solution always consists of finding a set of substitutions, $\Theta$, at run time, that will be used to instantiate the operation of the node. When possible, we will use the Pre-Condition and the Post-Condition for computing $\Theta$. Let $(\text{Pre}, \text{O}(\text{A}), \text{Post})$ be the node we are considering and let $O'(\text{A})'$ be the operation of its immediate predecessor node. We have identified the following relevant cases when the operation of the node is not ground:

- Case 1: $\text{A}'$ is a derived atom, $O' = \text{ins}$ and $O = \text{ins}$
- Case 2: $\text{A}'$ is a derived atom, $O' = \text{ins}$ and $O = \text{del}$
- Case 3: $\text{A}'$ is a base atom and $O = \text{ins}$
- Case 4: $\text{A}'$ is a base atom and $O = \text{del}$
- Case 5: $\text{A}'$ is a derived atom and $O' = \text{del}$

Note that:

- In cases 3 and 4, it is non-relevant if $O'$ is ins or del since when the node $(\text{Pre}, \text{O}(\text{A}), \text{Post})$ is reached, this operation (i.e. $O'(\text{A}')$) will have been already executed;
- In case 5, it does not matter if $O$ is inc or del because in both cases the set $\Theta$ is computed in the same way.

Let an allowed conjunction be a conjunction where every variable appears, at least, in a positive literal; then the set of substitutions, $\Theta$, that will be used to instantiate the operation of the node is obtained in each case as follows:

- In cases 1 and 3, $\Theta$ is a set with only one element $\emptyset$. If $\text{Post}$ is an allowed conjunction, then $\emptyset$ can be computed by evaluating $\text{Post}$ in the database. If there is still some missing value, the user will be asked to input it. In any case, $\text{Pre}\emptyset$ must be true in the database.
- In cases 2 and 4, it holds that $\text{Post}$ is always an allowed conjunction at run time and it also holds that every variable in $\text{A}$ appears in $\text{Post}$. $\Theta$ is a set with only one element, $\emptyset$, that can be computed following one of two strategies:
  - a) Evaluating $\text{Post}$ in the database, if $\text{Pre}\emptyset$ is true in the new database;
  - b) Evaluating $\text{Pre}$ in the database.

If both strategies fail, then another node must be selected in the same clan.

- In case 5, the set $\Theta$ will possibly have more than one element and must be computed by evaluating $\text{Pre}$ in the database. All the substitutions obtained with this evaluation must be included in $\Theta$.

The following algorithm is a revised version of the previous one. In it:

- $\emptyset$ represents the identity substitution.
- Instantiate is a procedure which returns a set of substitutions, $\Theta$, to instantiate the operation of the node when appropriate. This set is obtained in accordance with the previous analysis.

ALGORITHM T-tree_Process_2

INPUT

$(\text{Pre}_0\text{root}, \text{O}_\text{root}, \text{Post}_\text{root})$: T-tree root;
$D_{\text{init}}$, $D_{\text{initial}}$: database states;
OUTPUT

$D_{\text{out}}$: database state;
VARIABLES

$C$: successor clan;
$c$: T-tree node;
$\Theta$: set of substitutions;
$\emptyset$: substitution;
BEGIN
IF $\text{O}_\text{root} = \emptyset(\text{A}_\text{root})$
THEN
| IF $\text{A}_\text{root}$ is a derived atom
| THEN
| | | FOR EACH successor clan of the root, $C$,
| | | | DO
| | | | | $\Theta := \emptyset$;
| | | | | WHILE $\Theta = \emptyset$ AND $C$ has still nodes DO
| | | | | | Choose a node, $c$ of $C$,
| | | | | | | $c = (\text{Pre}_c, \text{O}_c(\text{A}_c), \text{Post}_c)$;
| | | | | | IF $\text{A}_c$ is not ground
| | | | | | | THEN
| | | | | | | | $\Theta := \text{Instantiate}((\text{Pre}_c, \text{O}_c(\text{A}_c), \text{Post}_c))$,
The problem of recursive views

A recursive predicate is a predicate which appears in a cycle in the dependency graph of a set of deductive rules. The T-trees for inserting and deleting instances of recursive predicates are obviously infinite as can be seen in the following example.

Let $p$ be a recursive predicate which is defined by the deductive rules:

1. $p(x) \leftarrow v(x,y,z) \land p(y) \land p(z)$
2. $p(x) \leftarrow t(x)$.

The $T$-trees for inserting and deleting instances of $p(x)$, shown in figures 6 and 7, are infinite (the symbol “…” denotes that there are infinite successor clans).

**Figure 6: T-tree for inserting instances of $p(x)$ in a database with recursive predicates**

**Figure 7: T-tree for deleting instances of $p(x)$ in a database with recursive predicates**

To solve this problem it is necessary to include a stop-rule in the generation of $T$-trees. Independently of the stop-rule, the approach is not able to consider all possible transactions in the presence of recursive rules. Since this circumstance cannot be avoided, we have chosen the most simple stop-rule. This stop-rule is the following:

“The node $(Pre, O(A), L_1 \land \ldots \land L_n)$ is forbidden in a $T$-tree if:

- The predicate of $A$ is a recursive predicate and in the path from this node to the root, there is another node $(Pre', O(A'), Post')$ such that the predicate of $A'$ is the same as the predicate of $A$; or
- There exists an $i$ $(1 \leq i \leq n)$ such that the predicate of $L_i$ is recursive and in the path from this node to the root, there is another node $(Pre', O(A'), Post')$ such that the predicate of $A'$ is the same as the predicate of $L_i$.”

Applying this stop-rule, the $T$-trees for inserting and deleting instances of $p(x)$ in the previous example would be the ones shown in figure 8.
And let us consider a repair rule whose event is represented by the following predicate but there are not deductive rules defining it:

\[ \text{condition} \]

In this case, the database of a schema where the predicate \( p(x) \) is a derived predicate but there are not deductive rules defining it:

1. \( \text{ins}(p(x)) \\land \neg p(x) \)
2. \( p(x) \\land s(x) \land \neg r(x) \)

And let us consider a repair rule whose condition is \( q(x) \land \neg p(x) \) and whose action is represented by the T-tree of figure 9.

In this T-tree, the operations \( O \) and \( O' \) are incompatible if their predicates are the same and if the operation of \( O \) is ins (resp. del) and the operation of \( O' \) is del (resp. ins) and let a correct leaf in a T-tree be a node which has not successor clans, whose operation is over a base predicate and whose Post-condition is the special predicate \( true \).

There are four situations in which it is necessary to bound a T-tree:

- There is a node in the T-tree, different from the root node, whose operation is \( abort \).
- The stop-rule has avoided including some nodes in the T-tree.
- There are operations in the T-tree which are incompatible with the event that triggers the repair rule.
- There are nodes in the T-tree which are incompatible with some operation in the user transaction.

The process of bounding the T-tree, \( T \), of a repair rule, \( R \), can be done in four steps:

1. Eliminating from \( T \) the nodes whose operation is incompatible with the event of \( R \). The successor clans of these nodes must also be eliminated.
2. Eliminating from \( T \) the nodes which are in a path that does not finish in a correct leaf.
3. Eliminating from \( T \) the nodes, and its successor clans, which have lost a clan due to the application of the stop-rule when defining the T-tree or due to the application of the previous steps when bounding the T-tree.
4. If \( T \) is empty after applying the previous steps, then the action of \( R \) must be a T-tree with just one node: \( (true, \text{abort}, true) \).

These four steps must be repeated until no node can be eliminated.

Figure 8: T-trees for inserting and deleting instances of \( p(x) \) in a database with recursive predicates after applying the stop-rule.

Bounding a T-tree

In a T-tree can appear paths that are not able to enforce the integrity. For example let the following set of deductive rules be the intensional database of a schema:

- \( \text{inc} \leftarrow q(x) \land \neg p(x) \)
- \( p(x) \leftarrow s(x) \land t(x) \)

And let us consider a repair rule whose event is \( \text{ins}(r(x)) \), whose condition is \( q(x) \land \neg p(x) \) and whose action is represented by the T-tree of figure 9.

In this T-tree, the paths \( N0-N1-N3-N5 \) and \( N0-N1-N4-N6 \) are not repairing paths because they undo the operation that triggers the rule, therefore they must be eliminated.

Figure 9: T-tree with non-repairing paths

In other cases, the problem arises when the operation \( abort \) appears in nodes which are not the T-tree root. This situation is shown in the following example.

Let the following set of deductive rules be the intensional database of a schema where the predicate \( s(x) \) is a derived predicate but there are not deductive rules defining it:

- \( \text{inc} \leftarrow q(x) \land \neg p(x) \)
- \( p(x) \leftarrow s(x) \land t(x) \)

And let us consider a repair rule whose event is \( \text{ins}(q(x)) \), whose condition is \( \neg p(x) \) and whose action is represented by the T-tree:

Figure 10: T-tree with non-repairing paths

In this T-tree, the paths \( N0-N1-N3-N5 \) and \( N0-N2-N4-N6 \) are not repairing paths because both of them include the operation \( abort \).

To avoid the problems shown in these two examples, we propose to bound the T-tree. Bounding a T-tree consists of eliminating the paths in it that cannot repair inconsistencies. Let the operations \( O \) and \( O' \) be incompatible if their predicates are the same and if the operation of \( O \) is ins (resp. del) and the operation of \( O' \) is del (resp. ins) and let a correct leaf in a T-tree be a node which has not successor clans, whose operation is over a base predicate and whose Post-condition is the special predicate \( true \). We have found four situations in which it is necessary to bound a T-tree:

- There is a node in the T-tree, different from the root node, whose operation is \( abort \).
- The stop-rule has avoided including some nodes in the T-tree.
- There are operations in the T-tree which are incompatible with the event that triggers the repair rule.
- There are nodes in the T-tree which are incompatible with some operation in the user transaction.
The \( T \)-trees in figures 9 and 10 would be the following after bounding them:

\[
\begin{align*}
N_0 &= (\text{true}, \text{del}(\text{Cond}(x)), \text{true}) \\
N_2 &= (\text{true}, \text{true}, \text{del}(q(x)), \text{true})
\end{align*}
\]

Figure 11: \( T \)-trees of figures 9 and 10 after bounding

**GENERATION OF REPAIR RULES**

Once we have shown how we can obtain the event, the condition and the action for a repair rule, the algorithm shown below obtains the set of repair rules. Some comments about it are the following:

- The algorithm extends the one shown in section 3.
- The set \( \text{IUPOS}_\text{Reduced} \) is used to eliminate from \( \text{IUPOS} \) the redundant induced updates (see the example in Appendix A).
- The procedure \( \text{Generate}_T \)-tree obtains the action part of the repair rule from its condition following the ideas presented in section 4.

**ALGORITHM Generation_of_Repair_Rules_2**

**INPUT**

\( \text{IUPOS} \): set of Induced Updates; 
\( \text{IDB} \): set of deductive rules;

**OUTPUT**

\( R \): set of Repair Rules;

**VARIABLES**

\( \text{IUPOS}_\text{Reduced} \): set of Induced Updates; 
\( T \): \( T \)-tree;

BEGIN

\[
R := \emptyset;
\]

\[
\text{IUPOS}_\text{Reduced} := \emptyset;
\]

FOR EACH \((P,O,C,L)\) \in \( \text{IUPOS} \) DO

\[
\text{IF there are not an element (P',O',C',L') in \( \text{IUPOS}_\text{Reduced} \) and a variable for variable substitution and }\alpha \text{ such that } P\alpha = P'\alpha, \text{ and } C\alpha = O'\alpha \text{ and } \alpha = C'\alpha \text{ THEN}
\]

\[
\text{IUPOS}_\text{Reduced} := \text{IUPOS}_\text{Reduced} \cup (P,O,C,L);
\]

END_IF

END_FOR;

FOR EACH \((P,O,C,L)\) \in \( \text{IUPOS}_\text{Reduced} \) DO

\[
\text{IF the predicate of } P \text{ is the inconsistency predicate } \text{inc}_D \text{ THEN}
\]

\[
\text{Generate}_T \)-tree\((T,C,\text{IDB}); R := R \cup \{(\text{Event:O, Condition:C, Action:T})\};
\]

END_IF

END_FOR

END.

In a repair rule generated by this algorithm, the event, \( O \), is a dangerous operation for the integrity constraint \( W \), \( C \) is a simplified form of \( W \) and \( T \) compiles a set of transaction which can enforce the integrity of \( W \).

**Operational semantics of a repair rule**

In this section we show how to process, at run time, a repair rule generated be the previous algorithm. Let:

- \( D \) be a deductive database state,
- \( R = (\text{Event: } E, \text{ Condition: } C, \text{ Action: } T) \) be a repair rule,
- \((\text{Pre}, \text{Op}, \text{Post})\) be the root node of \( T \),
- \( Tr = \{O_1, ..., O_n\} \) be a transaction where \( O_i (1 \leq i \leq n) \) is a ground update operation over a base predicate, and
- \( Tr(D) \) be the database state obtained after applying the transaction \( Tr \) to the database state \( D \).

Then the active system has to do the two following tasks:

1. Trigger the repair rule \( R \). If there are an operation in \( Tr \), say \( O \), and a substitution \( \alpha \) such that \( E\alpha = O \) then the rule \( R \) is triggered.
2. Process of the rule \( R \) when it has been selected. This task consist of:

   a) Evaluating the condition \( C \). Let \( \Theta \) be the result of this evaluation, \( \Theta = \{ 0 \mid 0 \) is a computed answer for \( Tr(D) \cup \{ \leftarrow \text{C}\alpha \} \} \).

   b) Executing the action \( T \). If \( \Theta \neq \emptyset \) then for each \( \theta \in \Theta \) the substitution \( \alpha\theta \) is applied to every node in the \( T \)-tree \( T \) and the procedure call \( T\text{-Tree} \text{ Process}_2((\text{Pre}\alpha, \text{Op}\beta, \text{Post}\alpha\beta), Tr(D), D, D') \) is executed.

**SOME RESULTS**

In this section we present three important results of our method. The proofs of these results are not presented in the paper due to lack of space but they can be found in [6].

**Proposition 1:** Property of the elements of \( \text{IUPOS} \).

\( \Rightarrow \) Let \((P, O, C, L)\) be an element of \( \text{IUPOS} \), let \( O' \) be an insertion or deletion operation of a base ground atom and let \( D \) be a database state such that:

- \( \alpha = \text{mg}(O,O') \),
- \( D' \) is the database state obtained by executing \( O' \) in \( D \), and
- there exists an SLDNF-refutation for \( D' \cup \{ \leftarrow \text{C}\alpha \} \) with \( \beta \) as computed answer,

\( \Rightarrow \) Then, there exists an SLDNF-refutation for \( D' \cup \{ \leftarrow \text{C}\alpha \} \) with \( \beta \) as computed answer.

**Corollary 1:** Soundness of the algorithm \( \text{Generation_of_Repair_Rules}_2 \).

\( \Rightarrow \) Let \( R(\text{Event: } O, \text{ Condition: } C, \text{ Action: } ...) \) be a repair rule, associated with the integrity constraint \( W \), obtained by algorithm \( \text{Generation_of_Repair_Rules}_2 \), let \( \text{inc}_W \) be the inconsistency predicate of \( W \), let \( O' \) be an insertion or de-
letion operation of a base ground atom and let \( D \) be a database state such that:

- \( \alpha = \text{mgu}(O, O') \),
- \( D' \) is the database state obtained by executing \( O' \) in \( D \), and
- there exists an SLDNF-refutation for \( D' \cup \{ \leftarrow C \alpha \} \),

\( \Rightarrow \) Then, \( D' \) violates the integrity constraint \( W \).

**Proposition 2:** Soundness of the algorithm \( T\text{-tree}_\text{Process}_2 \).

\( \Rightarrow \) Let \( D \) be a stratified, allowed and strict database and let \( D' \) be the output of the call “\( T\text{-tree}_\text{Process}_2(\text{Pre, ins}(A), \text{Post}, D, D_{\text{initial}}, D') \)” (resp. “\( T\text{-tree}_\text{Process}_2(\text{Pre, del}(A), \text{Post}, D, D_{\text{initial}}, D') \)” where \( A \) is ground and \( \text{Pre} \) is true in \( D \),

\( \Rightarrow \) Then, it holds that \( \text{comp}(D') = \alpha \) (resp. \( \text{comp}(D') = \lnot \alpha \)).

**Proposition 3:** Soundness of a repair rule.

\( \Rightarrow \) Let \( R \) \((\text{Event: } O, \text{Condition: } C, \text{Action: } T)\) be a repair rule obtained by the algorithm \( \text{Generation_of_Repair_Rules}_2 \), let \( (\text{true, } O(A), \text{true}) \) be the root node of \( T \) and let \( O' \) be an insertion or deletion operation of a base ground atom where:

- \( \alpha \) is a substitution such that \( O\alpha = O' \),
- \( D \) is the database state obtained by executing \( O' \) in the state \( D_{\text{initial}} \),
- there exists an SLDNF-refutation for \( D \cup \{ \leftarrow C\alpha \} \) with \( \beta \) as computed answer, and
- \( D' \) is the output of the call “\( T\text{-tree}_\text{Process}_2(\text{true, } O(A\alpha\beta), \text{true}, D, D_{\text{initial}}, D') \)”

\( \Rightarrow \) Then, there exists a finitely failed SLDNF-tree for \( D' \cup \{ \leftarrow C\alpha\beta \} \).

**CONCLUSIONS**

In this paper we have presented a method for generating a set of ECA-rules for integrity enforcement (repair rules) in the context of an active and deductive system i.e. in a SQL3 system. The main features of the method are:

1. The set of repair rules is generated at schema definition time;
2. It works for general deductive databases even in presence of recursive views;
3. The action part of a repair rule embodies all the repairing transactions associated with a violating user update; this is achieved by defining a new data structure, the \( T\)-tree (Transaction Tree);
4. It offers several strategies to solve the problem of missing values;
5. It is sound for all cases and it never loops.
6. The new data structure proposed could be also applied to other problems of database field as Evolution Schema, View Updating or Materialized Views.

**REFERENCES**


