Three Types of Redundancy in Integrity Checking; an optimal solution

R.R. Seljée
e-mail: R.Seljee@everest.nl *

H.C.M. de Swart
e-mail: H.C.M.deSwart@kub.nl

May 29, 1998

Abstract

Known methods for checking integrity constraints in deductive databases do not eliminate all aspects of redundancy in integrity checking. By making the redundancy aspects of integrity constraint checking explicit, independently from any chosen method, it is possible to develop a new method that is optimal with respect to the classified redundancy aspects. We distinguish three types of redundancy and propose an integrity checking method based on revised inconsistency rules.

1 Three Types of Redundancy

A deductive database consists of facts, rules and inconsistency indicators. *Facts* are ground atoms of the form $p(c_1, \ldots, c_n)$ where p is an n-ary predicate symbol and $c_1, \ldots, c_n$ are (individual) constants. A *rule* is a clausal form expression of the form $H_1 \vee \ldots \vee H_k \Leftarrow B_1 \wedge \ldots \wedge B_m \wedge \neg A_1 \wedge \ldots \wedge \neg A_n$ where each H_i, B_j and A_l is an atomic formula. For reasons of simplicity of presentation we restrict ourselves to rules with $k = 1$ and $n = 0$. An *integrity constraint* is a universally quantified closed first-order formula. And an *inconsistency indicator* is the negation of an integrity constraint.

*Supported by a grant from the Cooperation Centre of Tilburg and Eindhoven Universities

So, if $\forall X \forall Y[parent(X,Y) \Rightarrow \neg student(X)]$ is an integrity constraint, then $\exists X \exists Y[parent(X,Y) \wedge student(X)]$ is the corresponding inconsistency indicator, frequently written as $parent(X,Y), student(X)$. The variables in an inconsistency indicator are supposed to be existentially quantified.
Sooner or later, facts, rules or even constraints change. Such a change is called an *update* to the database. We restrict ourselves to updates consisting of ground atoms, representing the insertion of a new fact. A database is called *consistent* if it obeys all its specified integrity constraints.
Following J.M. Nicolas [2] we suppose that the deductive database is consistent before the update. Throughout this paper we will use the following deductive database for illustration purposes.

Example 1 Let D be the deductive database consisting of the following facts, rules and inconsistency indicator.
$F_1 : father(1,10).$
$F_2 : father(1,11).$
$F_3 : child(10,2).$
$F_4 : student(3).$
$R_1 : mother(X,Y) \Leftarrow husband(Z,X), father(Z,Y).$
$R_2 : parent(X,Y) \Leftarrow father(X,Y).$
$R_3 : parent(X,Y) \Leftarrow mother(X,Y).$
$R_4 : mother(X,Y) \Leftarrow child(Y,X).$
$II_1 : parent(X,Y), student(X).$
Let U be the following update.
U: $husband(1,2)$.

Because of the presence of rules, an update may induce other (implicit) changes in the database. For instance, in Example 1 above the update $husband(1,2)$ induces $mother(2,10)$ and $mother(2,11)$ via rule R_1 and consecutively $parent(2,10)$ and $parent(2,11)$ via rule R_3. $Mother(2,Y)$ and parent $(2,Y)$ are called potential updates of U. The database resulting from updating database D by update U is denoted by D_U.

Definition 0 Let D be a deductive database and U an update to D.
i) Let $R : H \Leftarrow B_1 \wedge \ldots \wedge B_m$ be a rule in D and let L be a literal which is unifiable with B_i in R for some i. If γ is the most general unifier of L and B_i, we say that $H\gamma$ *directly depends* on L with respect to R. If σ is a substitution such that $(B_1 \wedge \ldots \wedge B_{i-1} \wedge B_{i+1} \wedge \ldots \wedge B_m)\gamma\sigma$ is true in D_U,

then we say that $(H\gamma)\sigma$ is *directly induced* by L.
ii) A literal *depends on* L iff it directly depends on L or it directly depends on a literal depending on L. And a literal is *induced* by L iff it is directly induced by L or it is directly induced by a literal induced by L.
iii) Each literal depending on update U and U itself is called a *potential update* (with respect to U). And each literal induced by U and U itself is called an *induced update* (with respect to update U).

In this paper we address the problem of checking the integrity of the database after an update U in the most efficient manner; i.e., we want to check as efficiently as possible whether, given a consistent database D and an update U, the updated database D_U is still consistent. From the literature three methods for consistency checking are known: the method based on induced updates, the method based on potential updates and the method based on inconsistency rules [3]. The first two methods are shortly explained in section 1.1 and the third method is described in section 2.1.
In this paper we analyze three types of redundancies and show that the known methods suffer from at least one of them. Next we introduce the method based on revised inconsistency rules and explain why this method is optimal with respect to the redundancies just mentioned.

1.1 Redundancy of the first type

By a redundancy of the first type we mean the redundancy caused by computing *irrelevant* and/or *ineffective* derived updates.

Definition 1 Let U be an update to a database D. An induced or potential update is called *effective* if it is an induced insertion that does not hold in D. Otherwise, it is called *ineffective*.

For instance, in example 1, $mother(2, 10)$ is an induced update of update U: $husband(1, 2)$, but it is ineffective because $mother(2, 10)$ does hold in D because of fact F_3: $child(10, 2)$ and rule R_4. On the other hand, $mother(2, 11)$ is an example of an induced update that is effective. The potential update $mother(2, Y)$ is effective too.

Definition 2 Let U be an update to a database D. An induced or potential update is called *relevant* to the integrity checking of the updated database

D_U if it is relevant to some inconsistency indicator in D. Otherwise, it is called *irrelevant*. (A literal L is relevant to a formula F if it is unifiable with some literal in F.)

For instance, in example 1 the induced updates $mother(2, 10)$ and $mother(2, 11)$ are both irrelevant to II_1. On the other hand, the induced updates $parent(2, 10)$ and $parent(2, 11)$ are both relevant to II_1. When looking at the potential updates derived from update U, we see that $mother(2, Y)$ and $parent(2, Y)$ are derived, of which $mother(2, Y)$ is irrelevant to II_1 but effective and $parent(2, Y)$ is relevant and effective. This is shown in the pyramid of figure 1.

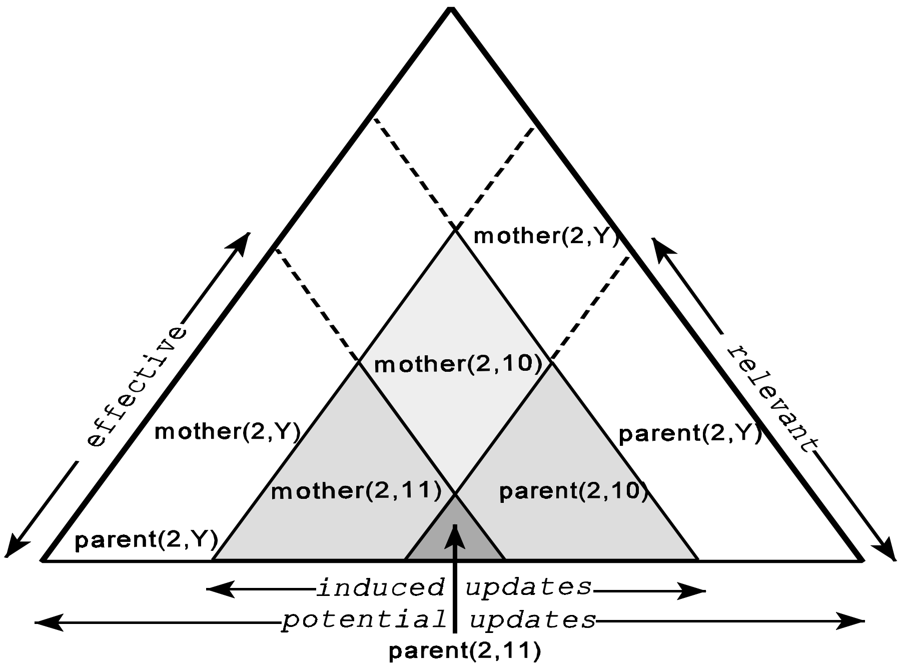


Figure 1: Redundancy of the first type

In this figure triangles are important for its interpretation. The semantics of this pyramid is given in the following way. The whole pyramid consists of derived updates that result from U. In fact, the pyramid consists of potential updates represented by the triangle of which the basis is marked with *potential updates.* Some of them can be instantiated to induced updates. These induced updates are represented by the triangle of which the basis is marked with *induced updates.* However, the potential updates may have other instances, which do not correspond to induced updates. These instances are present in the white area of the pyramid. Now, the derived updates are observed from two other perspectives. On the left side of the pyramid a distinction is made between effective and ineffective derived updates. The effective part of all derived updates is enclosed in the triangle of which the side is marked by *effective.* In the other part of the pyramid all ineffective derived updates are represented. On the right side of the pyramid a distinction is made between relevant and irrelevant derived updates. The relevant part of the derived updates is enclosed in the triangle of which the side is marked by *relevant.* In the other part of the pyramid all irrelevant derived updates are represented. Note that the inner core of the pyramid is the most interesting part with respect to the inconsistency indicator. This part, represented by the smallest triangle in the figure, actually influences the consistency of the database.

In the **method based on induced (potential) updates** three phases are distinguished:
the *generation phase*, in which the induced (potential) updates are generated;
the *selection phase*, in which all induced (potential) updates relevant to the inconsistency indicator are selected;
the *evaluation phase*, in which the induced (potential) instances of the inconsistency indicators, derived in the previous phase, are evaluated.
So, in the method of induced updates, in example 1 the induced updates $parent(2,10)$ and $parent(2,11)$ are selected, while the first one is not effective, and next the following instances of II_1 are evaluated:

$$\begin{array}{l} parent(2,10), student(2); \\ parent(2,11), student(2). \end{array}$$

Because $student(2)$ does not hold in D_U, II_1 is not validated in D_U and D_U remains consistent.
And in the method of potential updates, in example 1 the potential update $parent(2,Y)$ is selected and next the following instance of II_1 is evaluated:

$$parent(2,Y), student(2).$$

Note that in the method based on potential updates the computation of the induced updates is postponed to the evaluation phase. So, we do not spoil any evaluation time for finding instances of irrelevant potential updates, such as instances of $mother(2, Y)$ in example 1.

In the methods based on induced and potential updates, all induced and potential updates are generated before checking the consistency of the updated database. Some or many of these derived updates may be irrelevant and/or ineffective. Consequently, these methods suffer from the redundancy of the first type.
The method based on inconsistency rules, given in Seljée [3] and to be described shortly in section 2.1, does not suffer from this redundancy of the first type.

1.2 Redundancy of the second type

Suppose that in the example given above the $father$-relation does not contain facts for person 1, but only facts for person 3. That is, suppose that the facts F_1 and F_2 are replaced by the facts $father(3, 30)$, $father(3, 31)$ and $father(3, 32)$. Then the update $U : husband(1, 2)$ does not generate new $mother$-facts and therefore also no new $parent$-facts. In other words, there are no induced updates. However, the potential updates in the situation just mentioned are $mother(2, Y)$ and $parent(2, Y)$, of which the latter one is relevant to the inconsistency indicator II_1. So, the following instance of II_1 is evaluated:

$$parent(2, Y), student(2)$$

Since there are no new $parent$-facts that were not present before the update, the evaluation of II_1 is redundant. We call this kind of redundancy a *redundancy of the second type.*
This situation is illustrated in figure 2, where the dotted lines show the part of the database that does not change. Being a literal in II_1, $parent(X, Y)$ is by definition the root literal of a tree $T_{parent(X,Y)}$. Corresponding with the rules R_2 and R_3 there are two OR-branches leaving from the root, one with $mother(X, Y)$ and one with $father(X, Y)$ as child node. Corresponding with rule R_1 the node with $mother(X, Y)$ in its turn has two related (immediate) AND-successors, one with $husband(Z, X)$ and the other with $father(Z, Y)$. Related AND-nodes are indicated by an arc.

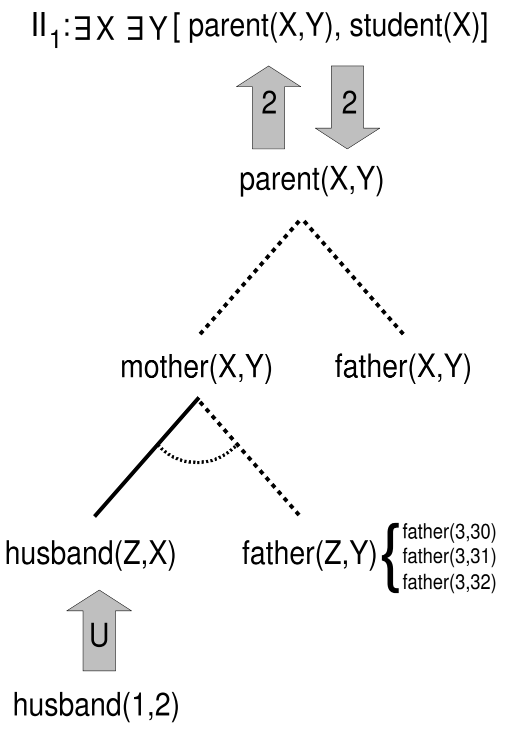


Figure 2: Redundancy of the second type

Note that the occurrence of this type of redundancy is highly dependent on the particular database state and therefore can only be determined at run-time. Therefore, during an inconsistency check it is important to find out with a minimum of database accesses if this situation occurs.

This kind of redundancy does not appear in the method based on induced updates, because by generating the induced updates first, one can pick out those inconsistency indicators that are *really* influenced by the update. It is a serious problem in the method based on potential updates in which a relevant inconsistency indicator may turn out to be irrelevant when actually exploring the database.

1.3 Redundancy of the third type

A redundancy of the first type is a redundancy in the generation of derived updates. A redundancy of the second type is a redundancy in the selection of potential updates relevant to an inconsistency indicator. The third type of redundancy is the redundancy that appears in the evaluation of the selected inconsistency indicators. It involves an evaluation of parts of the database that are not affected by the update.
For instance, in the method based on potential updates the update *husband*$(1, 2)$ may cause an implicit update in the *parent*-relation because of a change in the *mother*-relation.

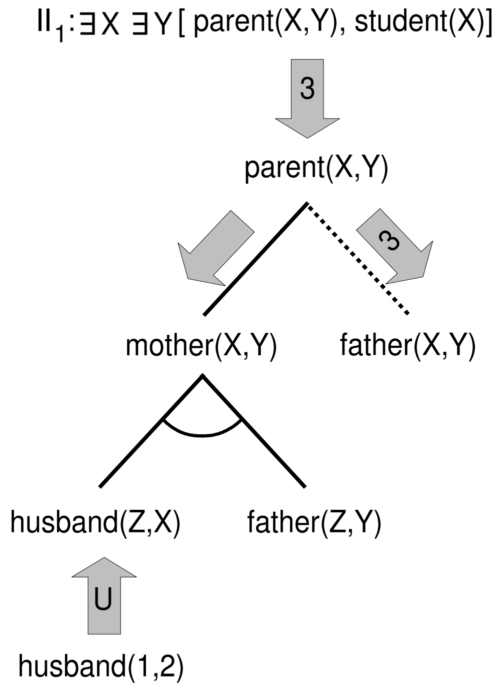


Figure 3: Redundancy of the third type

The resulting evaluation of *parent*$(2, Y)$, *student*(2) will lead to a search for a change of the *mother*-relation as well as the $father$-relation. But the $father$-relation did not change by the update. Therefore, the evaluation of the inconsistency indicator by going into the right branch of our tree is redundant. This situation is shown in figure 3.

Besides a check of an updated branch of such a tree, this could lead to a check of branches which are unchanged. We call this kind of redundancy a *redundancy of the third type*.

Remark Redundancy of the third kind may exist also in the method based on induced updates. For instance, the update *husband*(1, 2) could lead to an induced update in the *parent*-relation, resulting in an evaluation of an induced instance of II_1. This evaluation will search through the *father* definition part of the *parent*-relation, which is clearly not changed.
Redundancy of the third type can lead to an enormous overhead in case of dependency trees, representing the intensional database, which are deeply and widely branched. This is represented in figure 4.

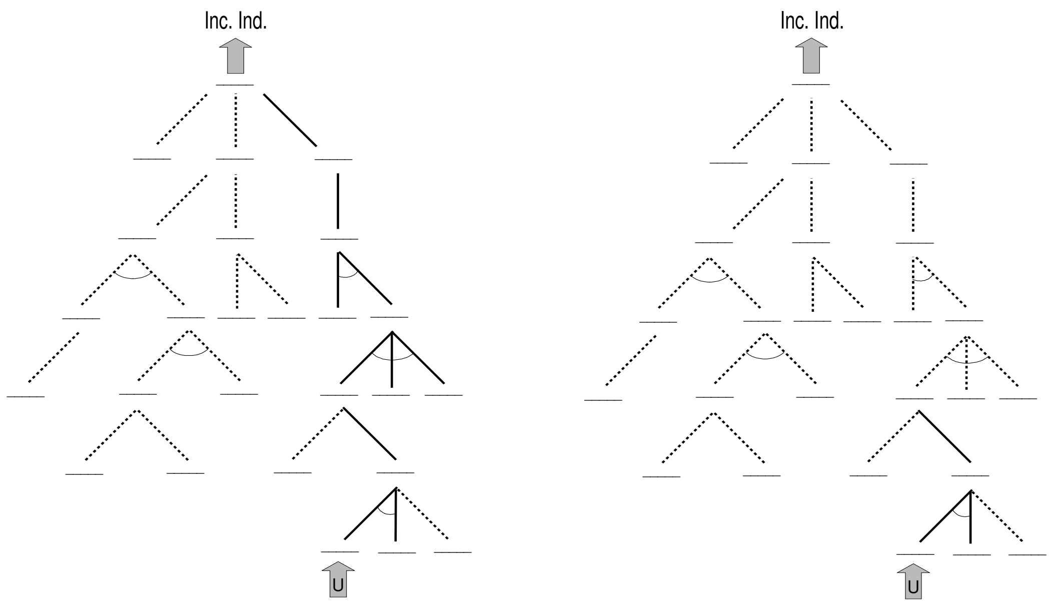


Figure 4: Redundancy of the third type in case of large dependency trees

In this figure the dependency tree is represented by an and/or tree, in which and-nodes are connected by an arc in the branches to the and-node. The parent node of such and-nodes represents a head of a rule and the and-nodes represent the body of that rule. The continuous lines in figure 4 show the influence of the update, i.e., the relations that are updated by the update. The dotted lines show the part of the database that does not change. But evaluating the expression in the top node means that all the branches will be searched for a change, even the dotted branches. Note further that in case of a combination of redundancy of the second and third type, the overhead can become extremely large, as the second picture in figure 4 shows.

1.4 Related research

In Seljée [4] some other kinds of redundancy are considered too: redundancy by duplicates, redundancy by neglecting the relation between updates and redundancy by replacement. The Fact Integrity Constraint Checking System, *FICCS*, presented in the same work and based on revised inconsistency rules (to be presented below), can avoid or minimize all redundancies mentioned above.
Redundancies in the evaluation of inconsistency indicators may be caused by the order of their subgoals and is the concern of query optimization techniques. This is not the kind of redundancy that is studied here. Although, in the literature, a lot of attention is paid to redundancy in query evaluation, which can be found in papers devoted to query optimization, not much fundamental research is done about the causes of redundancies in integrity checking.
However, in [1] Lee and Ling discovered a redundancy caused by evaluable predicates that appear in inconsistency indicators. Their idea is to evaluate these predicates as soon as possible in order to prevent database accesses caused by evaluation of other predicates in indicators. For instance, let

$$II_1\text{: } \mathit{parent}(X, Y),\ \mathit{student}(X),\ \mathit{eval}(X)$$

be an indicator that contains some evaluable predicate *eval*, which only depends on variable X.

Lee and Ling show that their optimization technique can be added to integrity checking methods in order to avoid this kind of redundancy. For instance, in the method of potential updates, first we could select the potential updates that are relevant to the indicator. When a potential update with respect to relation *parent* or *student* exists, which binds X in the indicator, then we should first evaluate *eval* before finding all induced updates that correspond to the potential updates, in order to save database access time when $eval(X)$ does not hold.

In [5] another redundancy concerning integrity checking in the case of aggregate constraints is studied. For instance, when an aggregate constraint I states that a relation R must not exceed the number of n tuples, and a tuple from R is removed, it is awkward to count the number of tuples in R again. Instead of this recount, together with I the number of tuples is stored. Say this number is v. So, checking I corresponds to finding out the number of tuples added to this relation, say m, resulting from the update and computing $v + m$ to see if it does not exceed n, as stated by I. Hence, this constraint can be checked without accessing the database.

2 Revised Inconsistency Rules

The main feature of the proposed method based on inconsistency rules, to be explained below, is that the consistency check itself is completely goal driven. The knowledge of how an arbitrary update may influence inconsistency indicators is represented by so called *inconsistency rules.* These rules are meta-rules that are asserted to the deductive database. By the application of these rules, from any update the relevant instantiated inconsistency indicators, that have to be evaluated in the deductive database, are found in just one step. Therefore, it does not have the disadvantage of generating induced or potential updates that are not relevant to any inconsistency indicator. Hence, redundancy of the first type does not appear in the method based on inconsistency rules.

In order to make this paper self-contained we first discuss the inconsistency rules, presented in Seljée [3]. By using inconsistency rules redundancy of the first type is avoided. Next we introduce revised inconsistency rules in order to minimize redundancy of the second type and of the third type as well.

2.1 Inconsistency rules

Inconsistency rules are derived from *inconsistency trees*, which in turn are derived from *potential update AND/OR trees* (see [4]). In what follows we use example 2 for illustration purposes.

Example 2 Let D be the deductive database consisting of the following facts, rules and inconsistency indicator.
$F_1 : father(1, 10).$
$F_2 : father(1, 11).$
$R_1 : mother(X, Y) \Leftarrow husband(Z, X), father(Z, Y).$
$R_2 : parent(X, Y) \Leftarrow father(X, Y).$
$R_3 : parent(X, Y) \Leftarrow mother(X, Y).$
$R_4 : age\text{-}diff(X, Y, N) \Leftarrow age(X, N_1), age(Y, N_2)$, N is $N_1 - N_2$.
$II_1 : parent(X, Y), age\text{-}diff(X, Y, N), N < 15.$
Let U be the following update.
$U : husband(1, 2).$

For example 2 we construct the potential update AND/OR trees $T_{parent(X,Y)}$ and $T_{age-diff(X,Y,N)}$ for the literals $parent(X, Y)$ and $age\text{-}diff(X, Y, N)$ appearing in the inconsistency indicator II_1, as follows. As before, being a literal in II_1, $parent(X, Y)$ is by definition the root literal of $T_{parent(X,Y)}$. Corresponding with the rules R_2 and R_3 there are two OR-branches leaving from the root, one with $mother(X, Y)$ and one with $father(X, Y)$ as child node. Corresponding with rule R_1 the node with $mother(X, Y)$ in its turn has two related (immediate) AND-successors, one with $husband(Z, X)$ and the other with $father(Z, Y)$. See figure 5. Related AND-nodes are indicated by an arc.

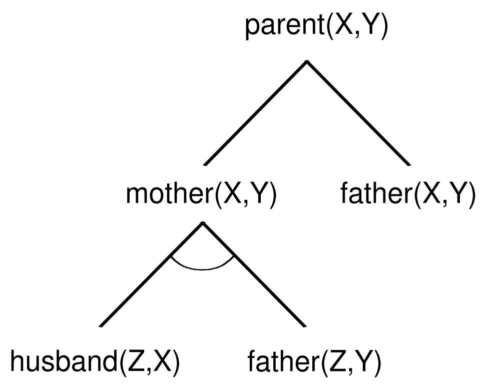
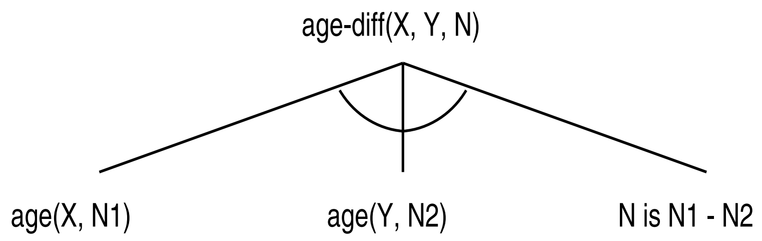


Figure 5: Potential update AND/OR tree for example 2

Similarly, we can construct the potential update AND/OR tree $T_{age-diff(X,Y,N)}$:



A precise definition of potential update AND/OR trees is given below.

Definition 3 Let L be a database literal that appears in an inconsistency indicator. Literal L is the root of a *potential update AND/OR tree*, say T_L. L is called the *root literal* of T_L. We start with L as the first constructed node. Let $\mathcal{N}$ be a constructed node, then the following construction rules are applicable:

1. If $\mathcal{N}$ is unifiable with the head of any rule, then the construction of T_L is a top-down construction which proceeds as follows:
   Let $R : H \longleftarrow B_1 \wedge \cdots \wedge B_m$ be a rule, where H is a positive literal which is unifiable with $\mathcal{N}$ and where $B_1, \ldots, B_m$ are literals. Let σ be the most general unifier of $\mathcal{N}$ and H; then for each j the literal $B_j\sigma$ is an AND-node with respect to rule R of $\mathcal{N}$ only if it is not redundant. If the literal is redundant it is not part of the potential update AND/OR tree again. If more than one rule is applicable, then for each rule there is an OR-branch for the literal $\mathcal{N}$, where each OR-branch ends in a group of related AND-nodes corresponding to the body of the applied rule.
   A literal in the construction process is redundant if

   - it is syntactically the same as some other node in the constructed potential update AND/OR tree so far, or
   - it is syntactically the same as some other node in the constructed potential update AND/OR tree so far, except that both nodes only differ with respect to some variables that do not occur in the root literal.

2. If $\mathcal{N}$ is not unifiable with the head of any rule, then $\mathcal{N}$ does not have any child node, i.e., the construction process stops.

In order to keep things simple and to avoid problems like those with view updates, we assume that only the leaf nodes in potential update AND/OR trees that are base relations in the given database are updatable. Figure 6 shows how in our example 2 the concepts of potential update AND/OR tree and inconsistency indicator interact. An update may instantiate a leaf node of some potential update AND/OR tree. By instantiating the leaf node of a potential update AND/OR tree the root literal of this potential update AND/OR tree is instantiated too.

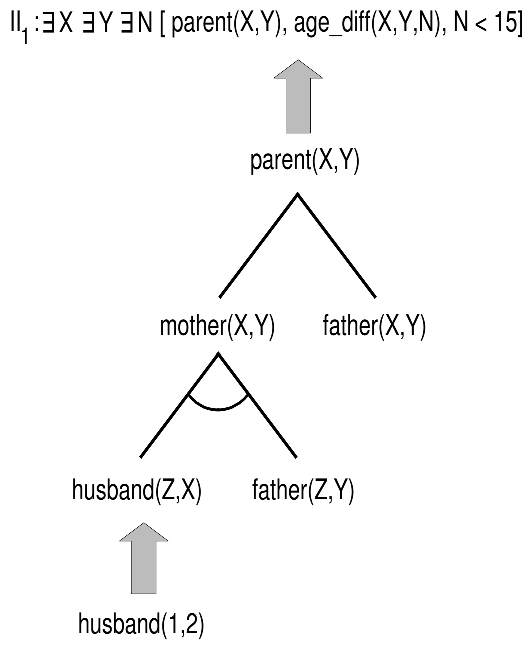


Figure 6: Updatable nodes in the potential update AND/OR tree for *parent* leading to II_1

In figure 6 the substitution $\{X/2,\ Z/1\}$ will instantiate $parent(X, Y)$ in II_1. The instantiated root literal of this potential update AND/OR tree instantiates the inconsistency indicator. The instantiated root literal is a potential update with respect to the update in the leaf node. This is the reason for calling these AND/OR trees *potential update* AND/OR trees.
Instantiation of inconsistency indicators by the updates in the transaction (i.e., a set of updates) can be done in just one step. To express this one-step

instantiation we construct *inconsistency trees*. They are defined using the definition of potential update AND/OR trees.

Definition 4 Let II be an inconsistency indicator. An *inconsistency tree* (also called a *one-level inconsistency tree*, see Seljée [3]) T_{II} is constructed as follows. The root of an inconsistency tree T_{II} is II. $\mathcal{N}$ is a child node of the root (i.e., II) of T_{II} if it is an updatable node of a potential update AND/OR tree, T_L, for some literal L in II. From $\mathcal{N}$ no other nodes are derived.

Figure 7 shows the inconsistency tree for II_1 in example 2. By the way, the update $husband(1, 2)$ does not belong to the inconsistency tree.

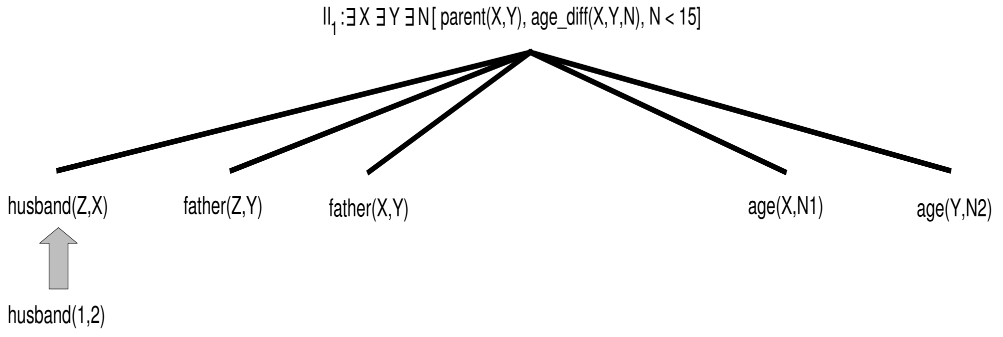


Figure 7: Inconsistency tree for II_1 in example 2

An instantiation of a node implied by some update now leads directly to a potential instance of the inconsistency indicator. Inconsistency trees are only needed to define inconsistency rules.

Definition 5 Let II be an inconsistency indicator, written without the existential quantifiers, and let A be a leaf node of the inconsistency tree T_{II}. Then

$$inconsistent(A) \Leftarrow II$$

is an inconsistency rule, where only the variables in II that do not occur in A are implicitly quantified existentially.

For instance, example 2 yields, among others, the following inconsistency rules:
$\mathit{inconsistent}(\mathit{husband}(Z, X)) \Leftarrow \mathit{parent}(X, Y), \mathit{age\text{-}diff}(X, Y, N), N < 15.$
$\mathit{inconsistent}(\mathit{age}(Y, N_2)) \Leftarrow \mathit{parent}(X, Y), \mathit{age\text{-}diff}(X, Y, N), N < 15.$

From the construction of the inconsistency rules the following theorem is clear. For an update U, the updated database D_U is the database that is obtained by updating database D with U.

Theorem Let D be a consistent database and U an update. Then D_U is consistent iff for each inconsistency rule $\mathit{inconsistent}(A) \Leftarrow II$ and for each substitution σ with $U = A\sigma$, $II\sigma$ is false in D_U.

Proof Let D be a consistent database and U an update. Suppose D_U is consistent, i.e., D_U satisfies the negation of each inconsistency indicator II, where all variables in II are implicitly quantified existentially.
Now let $\mathit{inconsistent}(A) \Leftarrow II$ be an inconsistency rule (with only the variables in II that do not occur in A implicitly quantified existentially) and let σ be a substitution with $U = A\sigma$; and suppose $II\sigma$ were true in D_U. Then II, with all variables implicitly quantified existentially, would be true in D_U. Contradiction.
Conversely, suppose that $II\sigma$ is false in D_U for each inconsistency rule $\mathit{inconsistent}(A) \Leftarrow II$ and for each substitution σ with $A = U\sigma$. And assume that D_U were inconsistent, i.e., D_U satisfies some inconsistency indicator II with all variables in II implicitly quantified existentially. Because D does not satisfy II, there must be an inconsistency rule $\mathit{inconsistent}(A) \Leftarrow II$ and a substitution σ such that $A = U\sigma$ and $II\sigma$ is true in D_U with all variables in $II\sigma$ implicitly quantified existentially. Contradiction.

The integrity checking method based on inconsistency rules, described above, avoids redundancy of the first type: no irrelevant or ineffective derived updates are computed. As we have seen in section 1.1 both the method based on induced updates and the method based on potential updates suffer from this redundancy. However, the method based on inconsistency rules, as presented in [3], contains redundancy of the second and third type. In example 2, the update $\mathit{husband}(1, 2)$ requires the evaluation of $\mathit{parent}(2, Y), \mathit{age\text{-}diff}(X, Y, N), N < 15$ which will be redundant if there are no *father*-facts of the form $father(1, Y)$ (redundancy of the second type) and which will lead to a search for a change of the *mother*-relation as well as the *father*-relation, while the *father*-relation did not change by the update (redundancy of the third type). For this reason we refine our inconsistency rules to revised inconsistency rules in the next section.

2.2 Revised inconsistency rules

In order to illustrate the advantages of revised inconsistency rules informally, consider example 2. Suppose the update to this database is $husband(1, 2)$ and let

$$parent(X, Y), age\text{-}diff(X, Y, N), N < 15$$

be an inconsistency indicator. Following the construction of the inconsistency rules in the previous section, the derived inconsistency rule with respect to the relation *husband* has the following form:

$$inconsistent(husband(Z, X)) \Leftarrow parent(X, Y), age\text{-}diff(X, Y, N), N < 15.$$

This means that whenever the inconsistency rule is applied to the update $husband(1, 2)$, the evaluation in the updated database of the instantiated inconsistency indicator

$$parent(2, Y), age\text{-}diff(2, Y, N), N < 15$$

is necessary. In fact, what we really want to know is if there exists a *parent* in the new database state, which was not present in the previous database state, for which the age difference to the parent's children is less than 15. Note that the update $husband(1, 2)$ only changes the database through the second *parent*-rule. In other words, only new mothers can contribute to the change of the *parent*-relation. But when evaluating the instantiated indicator, the subgoal $parent(2, Y)$ will try to find all parents of this form; so, the *mother*- as well as the $father$-part of the *parent*-rule is searched. But it is known from the update that the $father$-relation has not changed. The idea is to incorporate this knowledge into the (revised) inconsistency rule. In order to do so, the relation *parent* is unfolded until the update of concern is met. The literal $parent(X, Y)$ in the inconsistency rule is replaced by an expression which gives a precise description of the change in *parent*. In general, if $husband(Z, X)$ is an update for some binding of Z and X, the *mother*-rule states that $mother(X, Y)$ is a new instance if there exist fathers of the format $father(Z, Y)$ in the database. So, instances of $father(Z, Y)$ will give new instances of $mother(X, Y)$ and consequently new instances of $parent(X, Y)$. So, only an instance of $father(Z, Y)$ determines a new instance of *parent*. Therefore, in our example in the inconsistency rule with respect to *husband*, $parent(X, Y)$ can be replaced by $father(Z, Y)$. The revised inconsistency rule is:

$$inconsistent(husband(Z, X)) \Leftarrow father(Z, Y), age\text{-}diff(X, Y, N), N < 15.$$

Note that with this revised inconsistency rule not only redundancy of the third type is eliminated, but also redundancy of the second type is minimized. With update $husband(1, 2)$ only *father*-facts of the form $father(1, Y)$ and no *father*-facts of the form $father(3, Y)$ can lead to an inconsistency.

In order to give a precise definition of revised inconsistency rules we first have to introduce update expressions and revised inconsistency indicators. In example 2, given above, the update expressions $\Delta_{\mathcal{N}}^{parent(X,Y)}$ of $parent(X, Y)$, with $\mathcal{N}$ being an updatable node, will be: $\Delta_{husband(Z,X)}^{parent(X,Y)} = father(Z, Y)$, $\Delta_{father(Z,Y)}^{parent(X,Y)} = husband(Z, X)$ and $\Delta_{father(X,Y)}^{parent(X,Y)} = \text{true}$.

Below we give the general definition of update expressions.

Definition 6 Let C and D be literals appearing in some potential update AND/OR tree T_L, in which C is ancestor of D. A conjunction of literals is collected from T_L starting with D as the current node and the empty conjunction. The collecting process proceeds as follows.

- Let D' be the parent node of the current node, then collect all child AND-nodes related to the current node, excluding the current node, and add these nodes to the current literal set, resulting in the literal set $S_{D'}$.

Proceed this process with D' as the current node and $S_{D'}$ as the current literal set.
Continue this algorithm until C is reached. By Δ_D^C we denote the conjunction of all collected AND-nodes in S_C in order of derivation, or *true* if $S_C = \emptyset$.

Definition 7 Let II be an inconsistency indicator, let L be a literal in II and let $\mathcal{N}$ be an updatable node from the potential update AND/OR tree T_L. Then we call $\Delta^L_{\mathcal{N}}$ the *update expression* of L by $\mathcal{N}$.

Definition 8 Let II be an inconsistency indicator, let L be a literal in II and let $\mathcal{N}$ be an updatable node from the potential update AND/OR tree T_L. The expression that is derived from II by replacing L by $\Delta^L_{\mathcal{N}}$ is called a *revised inconsistency indicator* with respect to L and $\mathcal{N}$. This revised inconsistency indicator is denoted by $II(L,\mathcal{N})$.

Definition 9 Let II be an inconsistency indicator, let L be a literal in II. We call the expression that is derived from II by leaving out L from the conjunction the *remainder* II_L of II with respect to L.

Remark The revised inconsistency indicator in definition 8 is expressed by:

$$II(L,\mathcal{N}) = \Delta^L_{\mathcal{N}}, II_L.$$

Let II be the inconsistency indicator in example 2: $parent(X,Y)$, $age\text{-}diff(X,Y,N)$, $N < 15$. Let L be the literal $parent(X,Y)$ and let $\mathcal{N}_1$ be the updatable node $husband(Z,X)$ in the potential update AND/OR tree T_L. Then $II(L,\mathcal{N}_1) =$

$$(\Delta^{parent(X,Y)}_{husband(Z,X)}, II_L) = father(Z,Y), age\text{-}diff(X,Y,N), N < 15.$$

Let $\mathcal{N}_2$ be the updatable node $father(Z,Y)$ in the potential update AND/OR tree T_L. Then $II(L,\mathcal{N}_2) =$

$$(\Delta^{parent(X,Y)}_{father(Z,Y)}, II_L) = husband(Z,X), age\text{-}diff(X,Y,N), N < 15.$$

Let $\mathcal{N}_3$ be the updatable node $father(X,Y)$ in T_L. Then

$$II(L,\mathcal{N}_3) = (\Delta^{parent(X,Y)}_{father(X,Y)}, II_L) = age\text{-}diff(X,Y,N), N < 15.$$

Definition 10 Let II be an inconsistency indicator and T_{II} the corresponding inconsistency tree. For each leaf node $\mathcal{N}$ in T_{II}, which is by definition an updatable node of a potential update AND/OR tree T_L for some literal L in II,

$$inconsistent(\mathcal{N}) \Leftarrow II(L,\mathcal{N})$$

is a *revised inconsistency rule.*

The five revised inconsistency rules that are derived from example 2 are:

- $inconsistent(husband(Z,X)) \Leftarrow father(Z,Y), age\text{-}diff(X,Y,N), N < 15$;
- $inconsistent(father(Z,Y)) \Leftarrow husband(Z,X), age\text{-}diff(X,Y,N), N < 15$;
- $inconsistent(father(X,Y)) \Leftarrow age\text{-}diff(X,Y,N), N < 15$;
- $inconsistent(age(X,N1)) \Leftarrow parent(X,Y), age(Y,N2), N1-N2 < 15$;
- $inconsistent(age(Y,N2)) \Leftarrow parent(X,Y), age(X,N1), N1-N2 < 15$.

Note that the definition of Δ guarantees the following property.

Theorem $\Delta_D^B = \Delta_C^B, \Delta_D^C$ for literals B, C and D in some potential update AND/OR tree, where each of the Δ's are defined and ',' represents the operation of conjunction between the two operands.

Proof In the computation of Δ_D^B, following definition 6 we first collect all literals going from D to C and next all literals going from C to B. Hence, $\Delta_D^B = \Delta_C^B,\ \Delta_D^C$.

This property shows that an adjustment of the database schema will lead to a natural adjustment of Δ. For instance, the adjustment of the definition of a predicate which appears in C does not imply a complete recomputation of Δ_D^B, but is constrained to Δ_D^C.

Summarizing: The method based on revised inconsistency rules does not contain redundancy of the first type, because in methods based on inconsistency rules no inconsistency rule exists for updates not relevant to any inconsistency indicator. So, such updates will not lead to any action in our method, as we might have expected.
Our method minimizes the redundancy of the second type. Given the revised inconsistency rule

$$inconsistent(husband(Z,X)) \Leftarrow father(Z,Y), age\text{-}diff(X,Y,N), N < 15$$

and given update $husband(1,2)$ the evaluation of this rule hardly takes time if there are no facts of the form $father(1,Y)$. Of course, it is important to choose an optimal order of the subgoals in the body of the revised inconsistency rules. In the rule above the literal $father(Z,Y)$ should be the first one in the body.
The optimal order of the subgoals can be determined by a query optimizer, but our Fact Integrity Constraint Checking System $FICCS$ can deliver also revised inconsistency rules in which an optimal order of the subgoals is determined.
From the construction of the revised inconsistency rules it is also clear that it eliminates redundancy of the third type: no branches are searched that are unchanged.

3 Conclusion

In section 1 we saw that the method of induced updates suffers from redundancy of the first and third type, while the method of potential updates suffers from redundancy of all three types. In section 2.1 we saw that the method based on inconsistency rules, as presented in [3], does not suffer from redundancy of the first type, but that it does suffer from redundancy of type 2 and 3. Therefore we introduced in section 2.2 the method based on revised inconsistency rules.

- It does not contain the redundancy of the first type.
- It minimizes the redundancy of the second type by choosing an optimal order of the subgoals in the body of the inconsistency rules.
- It eliminates the redundancy of the third type, because only branches with changes are evaluated.

The method based on revised inconsistency rules can be extended in several ways. For instance, we could introduce negation or recursion into our language, or we could allow updates of rules and/or constraints or even replacements in our transactions. These extensions are elaborated in Seljée [4]. In [4] it is also shown that the method of revised inconsistency rules can easily be implemented, particularly in Prolog. The revised inconsistency rules can be generated automatically at compile time. Therefore, they can be optimized before any update of the database is made. The revised inconsistency

rules can be adjusted incrementally; so, they do not have to be generated from scratch each time the set of rules or constraints is updated.
In the appendix of [4] a case study concerning a Deductive Hospital Information System and some test results are given in order to show the practical use and applicability of the proposed method.

References

[1] S. Y. Lee and T. W. Ling, Improving Integrity Constraint Checking for Stratified Deductive Databases, in: Dimitris Karagiannis, ed., *Lecture Notes in Computer Science*, volume 856 (Springer, 1994) 591-600.

[2] J.M. Nicolas, Logic for Improving Integrity Checking in Relational Databases, *Acta Informatica*, Vol 18, no 3 (1982) 227-253.

[3] R.R. Seljée, A new method for integrity constraint checking in deductive databases, *Data & Knowledge Engineering* 15 (1995) 63-102.

[4] R.R. Seljée, *FICCS*; A Fact Integrity Constraint Checking System for the validation of semantic Integrity Constraints after updating consistent deductive databases, Ph.D. thesis, Tilburg University, 1997. URL: http://wwwis.tue.nl/∼seljee/

[5] W. Weber, W. Stucky and J. Karszt, Integrity Checking in Data Base Systems, *Information Systems*, Vol 8, no 2 (1983) 125-136.

Acknowledgement: we are grateful to two anonymous referees for suggesting many improvements of an earlier version of this paper.