Eindhoven University of Technology

Department of Mathematics and Computing Science

Deductive Database Systems
and integrity constraint checking

by

Ron Seljée

95/11

ISSN 0926-4515

Computing Science Report 95/11
Eindhoven, April 1995

Deductive Database Systems and integrity constraint checking

Ron Seljée
Co-operation Centre Tilburg and Eindhoven Universities
PO 90153
5000 LE Tilburg
The Netherlands
E-mail: R.R.seljee@kub.nl or seljee@win.tue.nl
FAX: +31 13 66 28 92

April 3, 1995

Contents

Abstract

Today, relational database systems have been extended in several ways. One way to extend the relational database model is by adding rules. These rules can make a lot of information implicit. In other words, a lot of intensional facts are stored in the database through rules. Consequently, when one updates such a database, a lot of implicit updates may appear by the presence of rules. When also integrity constraints on data in the database are allowed, the checking of these constraints becomes a problem because an update can cause some implicit updates which in turn may violate some of the constraints. The search for inconsistencies is a time-consuming task which must therefore be done primarily at compile time. This paper presents a new method for checking integrity constraints in databases extended with rules. A special kind of database is used, namely a deductive database, which is a logical database that handles deductive rules. The goal of this paper is to find the causes of efficiency defects in existing methods. A classification of redundant evaluations in existing methods is given. On the basis of this analysis an improvement of the method based on inconsistency rules, which I have developed at an earlier stage (see [26]), is proposed. Compared to existing methods it follows a completely different approach. Inconsistency rules are meta-rules which describe for all possible updates the integrity constraint check that has to be done. Further, the check is mimimal from the redundancy types point of view. Several types of redundancy in integrity checking methods are described in this paper. These redundancies have to be avoided to provide that we only have to access a relevant part of the database that is affected by the update. The improved version of the method based on inconsistency rules is compared to other existing methods, i.e., methods based on *induced updates* and methods based on *potential updates*.

keywords: *logic programming, deductive databases, integrity constraints, consistency.*

1 Introduction

The available integrity checking methods in deductive databases are often criticized for their lack of efficiency. One of the goals of this paper is to present a new integrity checking method which eliminates that flow. In the first section a short introduction to integrity checking in deductive databases is given. In Section **2** methods for integrity checking in deductive databases are described. In Section **3** several redundant evaluation types in the inconsistency check are distinguished.

After this section a revised method based on inconsistency rules is proposed and compared to some existing methods, i.e, methods based on *induced updates* and methods based on *potential updates*. The methods are compared by using a lot of examples showing the specific redundancy problems for each existing method. These examples will give the reader a thorough insight in the problems in checking integrity constraints in deductive databases. Section **5** shows the problems occurring in the case of recursion. This section shows that the proposed method handles recursion well. The methods in this paper were implemented and tested. Section **6** shows the tests and the results.

As the results show, the database state, consisting of the facts and the rules of a database, has a great influence on the performance of the check. This paper intends to show that in all cases the proposed method performs better than all other methods. First, a short introduction to integrity checking in databases is given.

1.1 Integrity Checking in Deductive Databases

A lot of today's commercially available database systems are based on the relational database model. In this relational model facts are stored in tables. Tables contain explicit data. A relational database management system (RDBMS) handles the fast retrieval of information from a database. Besides explicit information tables also contain implicit information. In conventional relational systems the derivation of implicit from explicit information is left to the user. There is a growing need to delegate such derivations to the system. Therefore, a relational database is extended with deductive rules. For this reason deductive databases are studied. This implicit information is made explicit by using rules.

In the deductive database community one distinguish two kinds of facts, i.e., the facts that are extensional and the facts that are intensional. The extensional facts, i.e., the base facts, are all facts that are really present in the database. The intensional facts are those that are not present in the database but which can be deduced from the extensional facts using one or more database rules. Because updates of the intensional database (also known as view updates) will cause too many semantical difficulties (see [11] and [21]) only updates in the extensional database are allowed.

Integrity constraints do not make implicit information explicit but dictate the information the database is allowed to store. Under no circumstances is the database allowed to give erroneous or contradictory information, therefore such information must be kept from the database. In order to do this, integrity constraints are stated in order to monitor the consistency of the database. However, in most cases the check of such constraints still is a user's responsibility. In the future this becomes a system's responsibility. Relational database management systems have some automated integrity checks, such as referential integrity checks, domain integrity checks, etc.. But here all kinds of integrity constraints are allowed as long as they are expressible in our language.

Checking the consistency of the database by checking all constraints each time the database is changed is not feasible. Therefore the methods presently available make use of the assumption that before each update a database is consistent with respect to the specified integrity constraints. Having made this assumption it is possible to focus on only those constraints that are affected by a certain change of the database. In the relational case an update can directly influence a constraint. In the deductive case, a new problem caused by rules occurs. With respect to some update, the rules may generate several induced updates, which in turn can cause an inconsistent database state. Another issue is *"how is the integrity of the database restored after an update causing an inconsistent database"*.

This is called the issue of integrity maintenance. Here, in this article, only the issue of integrity checking is important. This means that we are only interested in how to detect in the most efficient manner that the database is inconsistent. Therefore, if an update causes an inconsistent database state, then the update is simply rejected.

1.2 A New Method based on Inconsistency Rules

When analyzing the way updates can lead to violation of integrity constraints, we want to know which updates influence directly or indirectly an integrity constraint. Therefore, we try to find out which and how general updates lead via which rules to which integrity constraints. This will be illustrated by an example. For, more information about this way of integrity checking see [26].

Suppose a database consists of a patient table, a medicine dispensation table and an integrity constraint that says "do not dispensate medicine A to babies". Now there are no facts in the extensional database that express the concept of baby. But suppose there is some rule that states that "babies are patients with an age less than 1 year". So, the insertion of the patient's age of three months influences the integrity constraint, because the rule links the integrity constraint to the patient table, in particularly to the attribute "age" with value less than one year. Note also that the dispensation table is linked directly to the integrity constraint and that the link from a (part of a) table to an integrity constraint can go via a lot of rules.

In the example, two links to the integrity constraint "do not dispensate medicine A to babies" appear; the link from the value of the age of a patient which is less than 1 year in the patient table and the link from the dispensated medicine in the medicine dispensation table. *Inconsistency rules* express these links in order to instantiate an integrity constraint directly after an update that influences that constraint. So, for instance, in our example the inconsistency rule says that if a patient with an age of less than 1 year is inserted we have to check if medicine A is dispensated to this patient. When a patient's age is set to 16, no inconsistency rules that correspond to this update are found. Therefore, no check needs to be made. However, when a patient's age is set to 10 months, then the inconsistency rule is applied which results in an evaluation of the statement that medicine A is dispensated to this patient. And, if an update of the medicine dispensation table "give patient Y medicine A" reaches the database, then "Y is a baby" must be evaluated. If "Y is a baby" is true, then an inconsistent state is reached.

This example illustrates the advantages of the proposed method. Instead of a full evaluation of the integrity constraint only an instantiation is evaluated which can be found in just one step after an update. If an update does not influence any integrity constraint, no evaluation or computation of any kind is needed, so the update can be accepted directly.

The new method is based on a new concept called "*inconsistency rules*". These rules are derived using the rules and the integrity constraints, and are asserted to the deductive database in order to be fired if an appropriate update passes.

2 Integrity Constraint Checking in Deductive Databases

In a deductive database one distinguishes facts, rules and integrity constraints. These facts, rules and constraints are expressed in a logical language, that of first-order logic. A lot of formal work in this area has been done by Nicolas, Lloyd and others (see [17], [18], [19], [22], [23]).

Throughout this paper updates are represented by ground literals. Let U represent an update to a deductive database D. U is called an *insertion* (resp. a *deletion*) if it is a positive (resp. negative) literal. If U is an insertion (resp. deletion) to a deductive database D, then the database resulting from inserting (resp. deleting) U is denoted by D_U. A set of updates is called a *transaction*. These updates are presented to a database at the same time. Let T be a transaction, then the database D, which is the result of all updates in the set, is denoted by D_T. D_U (resp. D_T) represents the database D extended by U (resp. T).

More about definitions and logical concepts in this section can be found in [16]. Now, a short overview of the concepts used is given. From a logical point of view, a relational or deductive database can be looked at in two different ways (see [7]).

First, there is *the model theoretic view*. Here, the facts (resp. all facts and deducible facts) of the relational database (resp. deductive database) are looked at as an interpretation (or model) of the set of logical formulas corresponding to the integrity constraints. Here, the interpretation assigns truth values to these logical formulas by the database state. If they are true, then the database is a *model* for the formulas (i.e., an interpretation in which all formulas are satisfied). A database is called *consistent* with respect to its specified integrity constraints iff it is a model for the logical formulas corresponding to the integrity constraints.

Second, there is *the proof theoretic view* in which the database can be seen as a first order theory from which integrity constraints can be derived. In this case the database is called consistent with respect to the specified integrity constraints. Now, suppose a database transaction, which can be any collection of updates, additions and/or deletions of facts, takes place. Then the consistent database state can change into an inconsistent one. Proof theoretically, the integrity of the database is preserved if all the integrity constraints are still derivable from the first order theory respesenting the new database state.

In this paper the latter appoach is followed.

2.1 Inconsistency Indicator versus Integrity Constraint

An inconsistency indicator is a statement which becomes true if the database becomes inconsistent by a transaction. An inconsistency indicator is the negation of an integrity constraint. For example, the following constraint which is true in the world we are modeling:

- a child must be at least 15 years younger than both its parents

can be reformulated as the inconsistency indicator:

- there exists a child that is less than 15 years younger than one of its parents.

As we have stated earlier, the system only deals with universally quantified rules and integrity constraints. Note that the integrity constraint is universally quantified; for we can reformulate the constraint as:

- for all children it holds that they are at least 15 years younger than both its parents.

So, as a consequence the inconsistency indicator is existentially quantified. Inconsistency is no longer looked upon as a violation of the integrity constraints but as the true occurrence of an inconsistency indicator in the updated database state.

More formally, constraints are of the form $\neg F$, where F is some closed existentially quantified formula expressed in the underlying language of the theory. So, the indicators are closed as well. In this article the theory is built using F instead of $\neg F$. F is called the *inconsistency indicator* with respect to the constraint. The new proposed method is easier to describe with the concept of inconsistency indicator than with the concept of integrity constraint. An inconsistency indicator F indicates whether the database is inconsistent or not. If F is true, the database is inconsistent. Therefore, the database is consistent if there is no specified inconsistency indicator that is true in the database. Inconsistency indicators play a crucial role in the proposed method. Namely, from these indicators inconsistency rules are built which are asserted to the database in order to detect inconsistent states of the database.

2.2 Induced Update Method versus Potential Update Method

In this subsection two existing classes of methods for checking the consistency of databases are described. For an comparison of several of these methods ([1], [2], [4], [6], [8], [10], [12], [15], [17], [20], [24], [25], [26]) we refer to [5], [9], [26].

First, methods based on induced updates have the common feature that from the update all new deducible facts are generated. Next the set of induced updates is seen as a transaction for which the integrity constraints must be satisfied. Note that a lot of redundant induced updates could be generated, i. e., induced updates that do not influence any inconsistency indicator.

Example 2.1 Consider a database with the following rules and facts:

RULES

R1: mother(X,Y) $\longleftarrow$ husband(Z,X),father(Z,Y)
R2: parent(X,Y) $\longleftarrow$ father(X,Y)
R3: parent(X,Y) $\longleftarrow$ mother(X,Y)

FACTS

F1: father(1,10)
F2: father(1,11)
F3: father(1,12)

Note that the deducible facts are parent(1,10), parent(1,11) and parent(1,12). Suppose the update to this database is husband(1,2). The facts mother(2,10), mother(2,11) and mother(2,12) are directly induced by husband(1,2) by using rule *R1* and facts *F1*, *F2.* and *F3.* In turn, mother(2,10), mother(2,11) and mother(2,12) directly induce parent(2,10), parent(2,11) and parent(2,12), respectivily. So, all induced updates with respect to husband(1,2) are husband(1,2) itself, mother(2,10), mother(2,11), mother(2,12), parent(2,10) parent(2,11) and parent(2,12). Suppose an inconsistency indicator expresses the fact that an age difference of less than 15 years between parent and child is prohibited:

II1: $\exists$X $\exists$Y parent(X,Y), age-diff(X,Y,N), N $<$ 15.

Now, given the update husband(1,2), the induced instances of *II1* are derived from all induced updates relevant to *II1*, i.e., parent(2,10), parent(2,11) and parent(2,12). These instances, which are called the induced instances of that inconsistency indicator, are just the simplified instances of *II1* with respect to the corresponding induced updates, i.e., parent(2,10), age-diff(2,10,N), N $<$ 15; parent(2,11), age-diff(2,11,N), N $<$ 15; parent(2,11), age-diff(2,12,N), N $<$ 15 respectivily. □

Secondly, the methods based on potential updates first generate all possible updates, which are called *potential* updates. From the update a forward reasoning process is started. We look for relations that might be updated by the initial update. Potential updates express these possible updates without checking if they represent real updates. So, the evaluation phase is postponed. Next these potential updates are matched with inconsistency indicators. Then the resulting instantiated indicators, which are called potential instances of those inconsistency indicators, are evaluated. This situation is illustrated in Example **2.2**.

Example 2.2 Consider the database with the rules, facts and inconsistency indicator of Example **2.1**. Suppose the update to this database is husband(1,2). The literal mother(2,Y) directly depends on husband(1,2) by using rule *R1*. In turn, parent(2,Y) directly depends on mother(2,Y). So, all potential updates with respect to husband(1,2) are husband(1,2) itself, mother(2,Y) and parent(2,Y). Now, given the update, the potential instances of *II1* are derived from the potential updates relevant to *II1*, i.e., parent(2,Y). This resulting potential instance of *II1* is just the simplified instance of *II1* with respect to parent(2,Y), i.e., parent(2,Y), age-diff(2,Y,N), N $<$ 15. □

Note that each induced update is an instance of some potential update. Therefore, also each instance of an inconsistency indicator is an instance of some potential instance of this indicator. However, it is possible that some potential instance of an inconsistency indicator does not have any corresponding induced instance. This follows from the fact that some potential updates may not have an instance corresponding to an induced update.

Each of these methods has some drawbacks. In the first class a lot of induced updates may be generated even those which are not affecting any constraint. These induced updates are redundant for checking the inconsistency of the database. In the second class of methods the forward reasoning process is continued as long as there are applicable rules which contain a potential update in its body, even if the potential update does not correspond to any induced update. In that case, these potential updates are redundant. The method proposed in this paper is based on a different approach and does not have these drawbacks. The existing methods reason forward from the rules, facts and update in order to find some instantiated constraints that have to be evaluated in the database.

2.3 Method Based on Inconsistency Rules

In this paper the method based on inconsistency rules ([26]) will be improved. The main feature of methods based on inconsistency rules is that the consistency check itself is completely goal driven. The knowledge how an arbitrary update may influence the inconsistency indicators is represented by so called *inconsistency rules* which are asserted to the deductive database. By using 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 updates or potential updates that are not relevant to any inconsistency indicator.

2.3.1 Potential Update Trees

The new method is based on a new concept called "*inconsistency rules*". These rules are constructed using the rules and the inconsistency indicators, and are asserted to the deductive database. The inconsistency rules are derived from *potential update trees*. Each potential update tree is derived from a relevant literal in an inconsistency indicator. A literal relevant for an indicator is a literal that corresponds to a relation which belongs to the domain of the database. Therefore, literals expressing some computation or comparison without accessing the database are not interesting in the construction of potential update trees.

Definition 2.1 A relation is called an *updatable relation* if the database allows an explicit update in that relation. □

Note that an updatable relation can be a relation that is also derived. In the next definition a strict distinction between updatable relations and derived relations is not made yet. Hence, updates in views are allowed.

Definition 2.2 Let L be a relevant literal that occurs 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. If L is updatable then the only child node of L, which is an AND-node, is L itself. If not, then we start with L as the first constructed node. Let $\mathcal{N}$ be a constructed node, then the following construction rules are applied:

1. If $\mathcal{N}$ is updatable then $\mathcal{N}$ does not have any child node,i. e., the construction stops.

2. 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 $B_1\sigma, \ldots, B_1\sigma$ are AND-nodes with respect to rule R of $\mathcal{N}$ iff it is not redundant. If more than one rule is applicable than 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 child node 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.

For each child node the construction rules are applied until none of these rules can be applied anymore. Note that for each literal relevant for some inconsistency indicator a potential update AND/OR tree is created. □

In real world databases one distinguishes derived relations (views) and base relations. A derived relation is described by some predicate which is defined in terms of one or more other predicates. A base relation is not defined by other relations. In most cases, updates in derived relations are in most cases not allowed. Also comparison relations $<$, $>$, $>=$, $<=$, $=$, ... are not updatable. In the remaining part of this paper this distinction between relations will be made. So, from here derived relations are not updatable.

Example 2.3 Consider the database consisting of the rules, facts and inconsistency indicator of Example **2.1**. Now, parent(X,Y), which is a relevant literal in the inconsistency indicator, is the root literal of the potential update tree $T_{parent(X,Y)}$. Note that parent is a derived relation and therefore, by assumption, not updatable. As a consequence, parent(X,Y) is not a child node of itself. The last construction rule of Definition **2.2** is applicable. The left child node of parent(X,Y) consists of mother(X,Y) and the right child node of parent(X,Y) consists of child node father(X,Y). By applying the mother-rule to mother(X,Y) two subnodes corresponding to the literals in the body of the mother-rule result, i. e., husband(Z,X) and father(Z,Y) (See Figure **2.1** for the complete potential update tree for parent(X,Y)). Of course there also is a potential update tree for age-diff. □

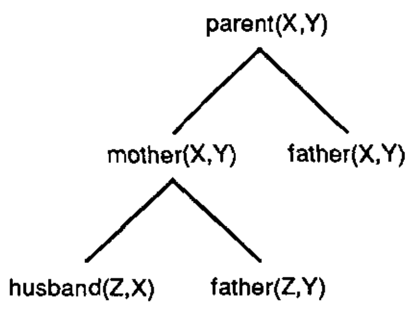


Figure 2.1 *Potential Update Tree for Example* **2.3**.

Note that for each literal relevant for some inconsistency indicator a potential update tree is created.

2.3.2 Inconsistency Trees

Only the updatable relations of nodes in the potential update trees are interesting from the perspective of the method based on inconsistency rules. Relations that are not updatable are therefore not responsible for any change in the consistency of the database. Any success of an inconsistency indicator can only be caused by an updatable node.

Definition 2.3 Let II be an Inconsistency Indicator. A node in a potential update tree of *II* is called an *updatable node* if it corresponds to an updatable relation. □

Here, as we have supposed, a node is updatable if and only if it corresponds to a base relation. Therefore only the leaf nodes in the potential update tree corresponds to updatable nodes and are relevant for deriving the relevant instances of the inconsistency indicators. Figure **2.2** shows how in our example the concepts of potential update tree and inconsistency indicator interact. An update may instantiate a leaf node of some potential update tree. By instantiating the leaf node of a potential update tree the root literal of this potential update tree is instantiated. The instantiated root literal of this potential update tree instantiates the inconsistency indicator. All nodes "between" the root and these leaf nodes are not relevant. Instantiation can be done in just one step. To express this we construct *inconsistency trees*. They are defined using the definition of potential update trees.

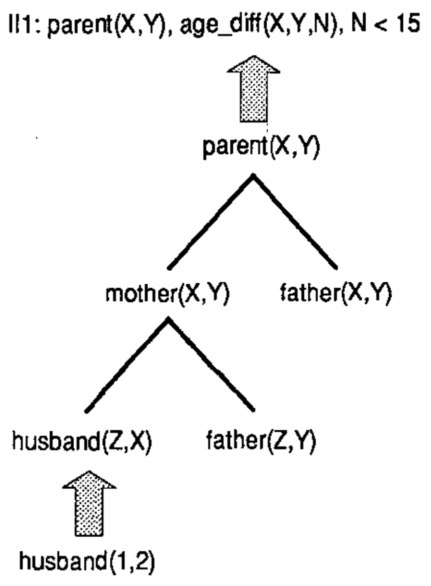


Figure 2.2 *Updatable leaf nodes in the Potential Update Tree for parent leading to II1.*

Definition 2.4 Let II be an inconsistency indicator. The root of an *inconsistency tree* (also called a *one-level inconsistency tree*, see [26]) 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 tree, T_L, for some literal L in II. From $\mathcal{N}$ no other nodes are derived. □

Note that from each potential update tree at least one inconsistency tree can be derived because, by definition, the top level literal of a potential update tree is a literal of an inconsistency indicator. Because L can be a literal which occurs in more than one inconsistency indicator, several inconsistency trees can be derived from a potential update tree of which L is the root literal.

An instantiation of a node implied by some update leads directly to an instantiation of the inconsistency indicator. Although in our method updating derived relations is not forbidden, the one-level inconsistency trees are constructed only for base relations. So, the updatable nodes correspond to base relations.

Example 2.4 Consider the database of Example **2.3**, and the potential update tree $T_{parent(X,Y)}$. For, inconsistency indicator *II1* an inconsistency tree is derived. Because this inconsistency indicator contains two literals, i. e., parent(X,Y) and age_diff(X,Y,N), the inconsistency tree is built from the potential update trees $T_{parent(X,Y)}$ and $T_{age\_diff(X,Y,N)}$. Each updatable node in the potential update tree $T_{parent(X,Y)}$, i. e., each node corresponding to the base relations father or husband, is a node in the inconsistency tree T_{II1}. Suppose we have the following rule which defines the age_diff relation

R1: age_diff(X,Y,N) ⟵ age(X,N1), age(Y,N2), N = N1 - N2,

then the corresponding complete inconsistency tree for *II1* can be found in Figure **2.3**. □

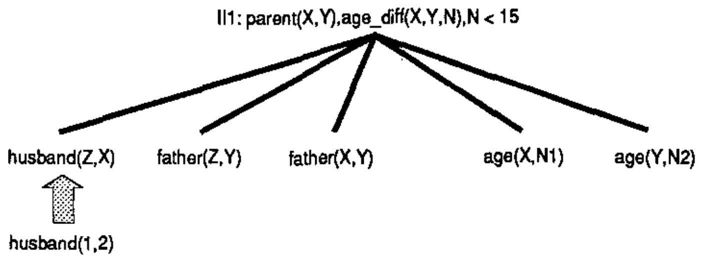


Figure 2.3 *Inconsistency Tree for II1 in Example* **2.4**.

Note that here age is a base relation. In real database applications the relation age as well as the relations father and husband may be linked to a database table. If so, they are derived relations and not updatable.

Example 2.5 For instance, the relations could be derived from a CITIZEN table from a register office database as follows:

R1: age_diff(X,N) ⟵ CITIZEN(X,_,_,_,D,_,_), date_to_age(D,N)
R2: father(X,Y) ⟵ CITIZEN(X,_,_,L,_,_,_,male), member(Y,L), CITIZEN(Y,_,_,_,_,_,_,_)
R3: husband(X,Y) ⟵ CITIZEN(X,_,Y,_,_,_,_,male)

where CITIZEN(_,_,_,_,_,_,_) corresponds to a table of which the first argument is the citizen's identifier, the third argument is the identifier of the one he or she is married with, the fourth argument is a list of identifiers of children of that citizen, the sixth argument is the citizen's date of birth and the last argument is for the citizen's sex. All other arguments are not relevant. Note that date_to_age and member are not updatable relations because they were only meant to compute a new value instead of defining a new relation. □

From now age, husband and father are the only base relations.

2.3.3 Inconsistency Rules

From the inconsistency trees inconsistency rules are constructed, which express the check that has to be performed for all kinds of updates. The actual check is a set of indicators that are instantiated by the update.

Definition 2.5 Let U be an update. Let $\mathcal{N}$ be an updatable node of an inconsistency tree T_{II}, which is unifiable with U. Let σ be a most general unifier of $\mathcal{N}$ and U. Then $II\sigma$ is called *a relevant instance* of II with respect to U. □

The next proposition states that it is sufficient to evaluate (in the updated database) only those instantiations of inconsistency indicators which are derived from the update and the relevant leaf nodes of inconsistency trees. If such an instance is true, then the update is rejected. If not, the updated database is consistent.

Proposition 1 Let U be an update. Suppose D is consistent. Then D_U is consistent iff for every inconsistency indicator II, it holds that each existing relevant instance of II with respect to U is false in D_U.

For a proof we refer to [26].

The next definition shows how the inconsistency rules, which are meta-rules expressing the goals that have to be evaluated after a certain update of the database, are derived from inconsistency trees.

Definition 2.6 Let II be an inconsistency indicator. Now, *inconsistent*$(A) \Leftarrow II$ is called an *inconsistency rule* if A is a leaf node of the inconsistency tree T_{II}. □

All derivable inconsistency rules for each inconsistency indicator are asserted to the database. An update triggers the evaluation of the instantiated indicators. The concept of inconsistency rules can be easily implemented in Prolog.

Example 2.6 Consider the situation as in Example **2.1**. The rules in this example are converted to Prolog as follows.

```
parent(X,Y) :-
  father(X,Y).
parent(X,Y) :-
  mother(X,Y).
mother(X,Y) :-
  husband(Z,X),
  father(Z,Y).
```

The inconsistency indicator which states that parents must be at least 15 years old is in Prolog:

```
  parent(X,Y), age_diff(X,Y,N), N < 15.
```

From the rules and the indicator the following inconsistency rules are derived:

```
inconsistent(father(X,Y)) :-
  parent(X,Y), age_diff(X,Y,N), N < 15.
inconsistent(father(Z,Y)) :-
```

```
  parent(X,Y), age_diff(X,Y,N), N < 15.
inconsistent(husband(Z,X)) :-
  parent(X,Y), age_diff(X,Y,N), N < 15.
```

For instance, an update husband(1,2) will lead to the application of the inconsistency rule with argument husband(Z,X) and the proper instantiation of the variables Z and X, i. e., application of the third inconsistency rule and substitution $\{Z = 1, X = 2\}$. □

The main advantage of these rules is that with some fixed number of deductive rules and inconsistency indicators, the inconsistency rules only have to be generated once and can be used over and over again for updates of facts. It is rather easy to implement a generator for inconsistency rules, which generates from the set of rules and indicators the "optimized" inconsistency rules. For each example previously mentioned the optimized inconsistency rules can be automatically generated in a few seconds from the set of rules and inconsistency indicators.

The first paper on this new method ([26]) shows the advantages of the method based on inconsistency rules when comparing it to methods based on induced updates and methods based on potential updates. However, the proposed method can be improved further. Section **4** shows that this improvement is fundamental. Before discussing the new version of this method redundancies in existing methods are studied and classified.

3 Redundancies in Integrity Checking Methods

This section will show some redundancies in the methods mentioned above. Three types of redundancy are distinguished. Some type may be typical for a method while the other types does not appear in this method. Nevertheless, the types are presented as general kinds of redundancy.

3.1 Redundancy of the First Type

First, the redundancy of the first kind concerns all computations not directly necessary for the consistency check. In other words, computations of which you can say beforehand that they do not influence an inconsistency indicator directly. For instance, the computation of all such induced updates is an example of this redundancy. Also the computation of all such potential updates is an example of this redundancy. Returning to Example **2.1**, we see that any computation of the mother-relation is an example of this kind of redundancy. As we can see in Figure **3.1**, the computation of all mother-facts induced by the update is redundant because there is no inconsistency indicator that directly corresponds to the mother-relation. Note that the redundancy of the first type does not appear in the method based on inconsistency rules.

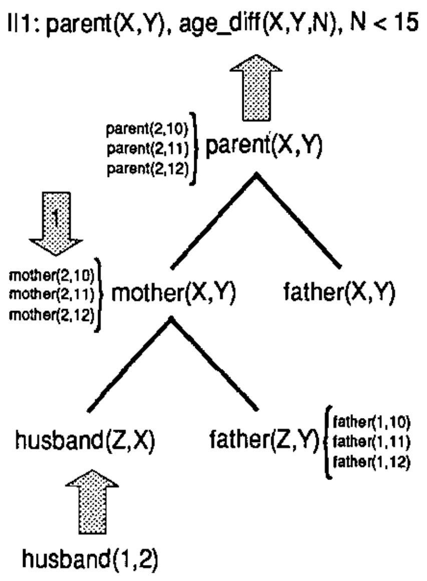


Figure 3.1 *Redundancy of the first type.*

3.2 Redundancy of the Second Type

The redundancy of the second type is the redundant computation and evaluation of a partially instantiated inconsistency indicator that contains a partial relation which is empty. So, no instantiation in this relation is possible beforehand. For instance, although in Example **2.2** the update husband(1,2) can influence the inconsistency indicator *II1*, the extensional database in this case does not have to imply mother-facts and therefore also no new parent-facts. If the father-relation does not contain facts for person 1, then no induced updates with respect to the mother-relation are found. Therefore, there are also no new parent-facts derivable in the database. This means that the evaluation of *II1* is a redundant evaluation because the evaluation of parent(X,Y) in *II1* does not involve parent-facts which were not present before the update. This situation is illustrated in Figure **3.2**.

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

3.3 Redundancy of the Third Type

The third kind of redundancy is the evaluation of inconsistency indicators, implied by some update, that involves a redundant evaluation of parts of the database not affected by that update. For instance, in the method based on potential updates the update

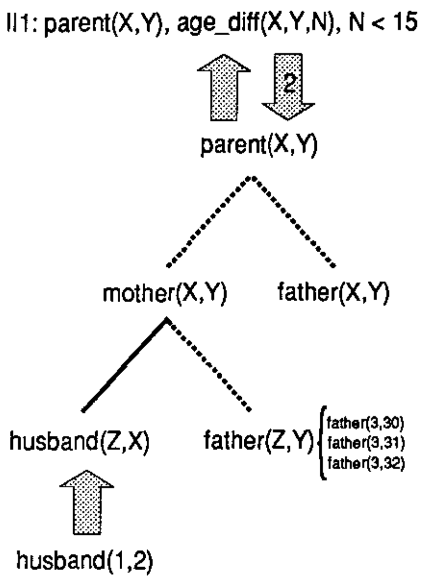


Figure 3.2 *Redundancy of the second type.*

husband(1,2) may cause an implicit update in the parent-relation because the mother-relation has changed. The resulting evaluation of parent(2,Y), age_diff(2,Y,N), N $<$ 15 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 indicator by going into the right branch of our tree is redundant. This situation is shown in Figure **3.3**. This type of redundancy also exists in the first version of my method based on inconsistency trees ([26]). Also note that this kind of redundancy can lead to an enormous overhead in case of trees that are deeply and widely branched. Besides a check of an updated branch of a potential update tree, this could lead to a check of branches which are unchanged. This is represented in Figure **3.4**. The continuous lines in Figure **3.4** 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, even the dotted branches. Note further that in this situation, when we have a combination of redundancy of the second and third kind, the overhead can become extremely large (see the second picture in Figure **3.4**).

There are some other redundancies, which are not studied here. For instance, an update as well as resulting induced updates may be already present in the old database state. In other words other derivation paths may already exist for an induced update. Therefore, any resulting check is redundant. This can be prevented by a check for existing derivation paths (see [3], [13]). Some other redundancies with respect to evaluative predicates and instantiated integrity constraints are described in [14].

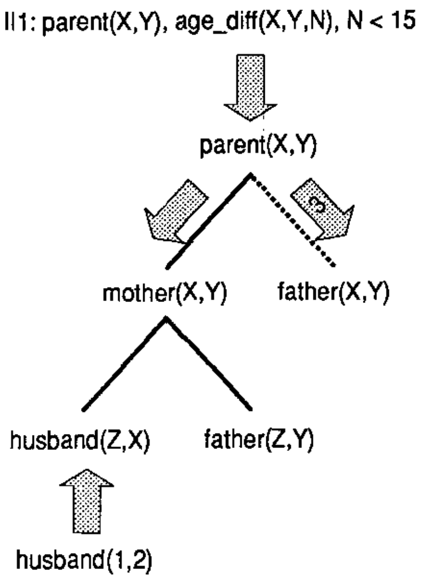


Figure 3.3 *Redundancy of the third type.*

4 Proposed Method Based on Inconsistency Rules

As the previous section shows the existing methods all have to deal with redundancies of one or more types. This is also true for the method based on inconsistency rules although it is already in most cases an improvement of existing methods based on induced updates and potential updates (see [26]). This section presents an improved version of the method based on inconsistency rules that does not contain the redundancy of the third type, while reducing the redundancy of the second type to a minimum. Before presenting the proposed method an example is given to make the improvement clear.

4.1 Improving the Method based on Inconsistency Rules

In the case of a relation which is defined in terms of several other relations, the first version of my proposed method will check a part of the database, which may be unnecessary (in the potential update method this is also a drawback which cannot be solved in a straightforward manner). In order to eliminate this redundancy of the third type we have adjusted the definition of inconsistency rules. We illustrate the improvement by an example.

Consider Example **2.6** and suppose the update to the database is husband(1,2); then the evaluation of the following instantiated inconsistency indicator is necessary:

```
parent(2,Y),age_diff(2,Y,N),N < 15.
```

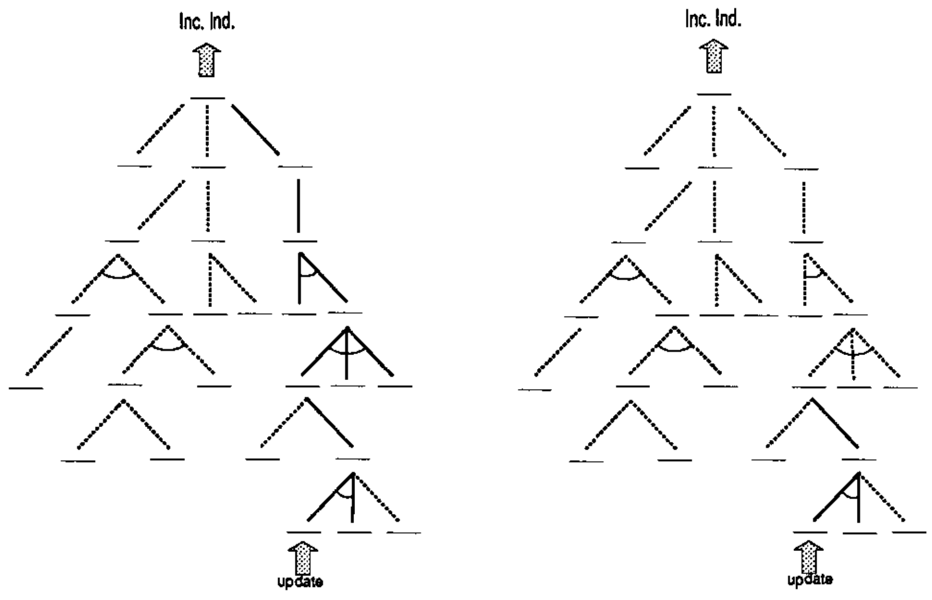


Figure 3.4 *Overview of redundancy of type three in case of a large potential update tree; the update is relevant to an inconsistency indicator.*

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 his/her 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 the knowledge of which part of the relation in the inconsistency rule has changed by the update, into the inconsistency rule. In order to do this 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 give 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)) :-
  father(Z,Y),age_diff(X,Y,N),N < 15.
```

This revision of existing inconsistency rules can be generalized. In the next section AND/OR trees built from the rules are introduced which turn out to be very helpful in optimizing the inconsistency rules.

4.2 AND/OR Potential Update Trees

In order to formulate the revised inconsistency rules, the definitions of potential update trees and inconsistency trees are revised. First, the definition of a potential update tree is revised. The distinction between this revised definition and the previous definition is that now we represent the potential update tree as an AND/OR tree. The construction of AND/OR trees results from dividing goals into subgoals by the application of rules. By each application of a rule we get several related AND-nodes, one for each literal in the body of the rule. The branches to related AND-nodes are joined by arcs. If n rules can be applied to a (sub)goal, then n groups of related AND-nodes originate from that (sub)goal. These groups are called OR-groups. OR-groups are not linked to each other.

The resemblence between the potential update tree and the potential update AND/OR tree is illustrated by an example.

Example 4.1 Consider the database with the rules, facts and inconsistency indicator of Example **2.1**. Now, parent(X,Y), which is a literal in the inconsistency indicator, is the root literal of the potential update tree $T_{parent(X,Y)}$. There exist two OR-branches of parent(X,Y), i.e., a branch which ends in AND-node mother(X,Y) and a branch which ends in AND-node father(X,Y). Now, by applying the mother-rule to mother(X,Y) two related AND-nodes corresponding to the literals in the body of the mother-rule are derived, i.e., husband(Z,X) and father(Z,Y). An arc between the branches to husband(Z,X) and father(Z,Y) expresses the fact that they are related AND-nodes (see Figure **4.1** for the complete potential update AND/OR tree for parent(X,Y)). □

4.3 Revised Inconsistency Trees

From the AND/OR potential update trees the revised inconsistency trees are derived. First, Definition **2.3** is reformulated for potential update AND/OR trees.

Definition 4.1 A node in a potential update AND/OR tree of *II* is called an *updatable node* if the database allows an update in the relation corresponding to that node. □

In the previous definition of inconsistency tree, the root of that tree is an inconsistency indicator. For each inconsistency indicator exactly one tree exists. Now, for each updatable node in the previous inconsistency tree a separate inconsistency tree is constructed. For such trees the root corresponds to an optimized inconsistency indicator. It is optimized because it checks only that part of the database which is influenced by the updatable node. To clarify this situation the following definition and Example **4.2** are helpful.

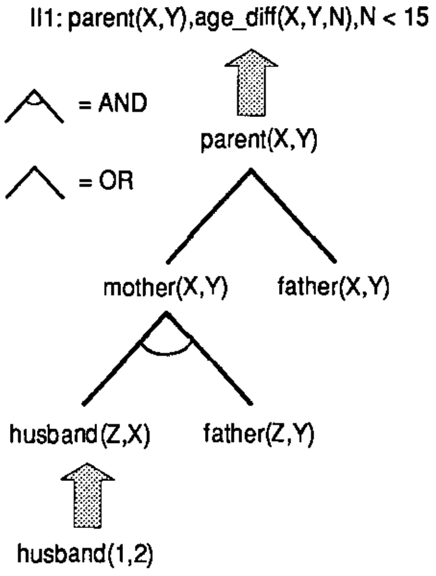


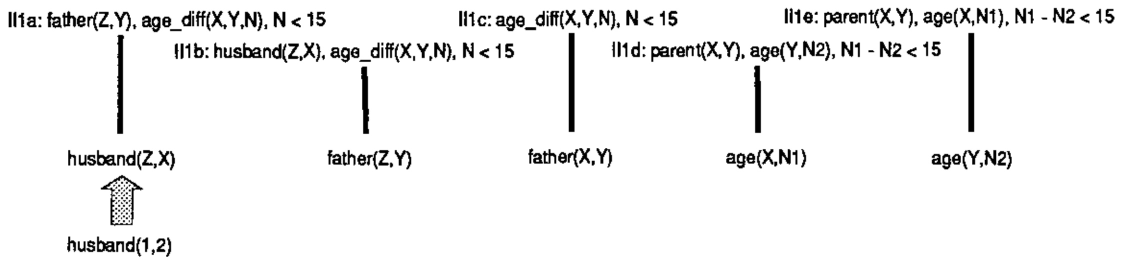
Figure 4.1 *Potential Update AND/OR Tree for Example* **4.1.**

Definition 4.2 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. A ***revised inconsistency tree*** (also called a *one-level inconsistency tree*, see [26]) with subnode $\mathcal{N}$ is constructed as follows. In order to determine the root of the revised inconsistency tree with subnode $\mathcal{N}$ T_L is used. Begin by taking $\mathcal{N}$ as the current node in T_L.

- If the parent node $\mathcal{P}$ of $\mathcal{N}$ is positive, then collect all related AND nodes, if any (not $\mathcal{N}$ itself), and go to the parent node $\mathcal{P}$ of $\mathcal{N}$ in T_L.
- If the parent node $\mathcal{P}$ of $\mathcal{N}$ is negative, then replace all collected AND nodes in an early stage by $\mathcal{P}$, and go to the parent node $\mathcal{P}$ of $\mathcal{N}$ in T_L.

Continue this algorithm until the root of T_L is reached. The root of the revised inconsistency tree consists of the optimized inconsistency indicator,i.e., the inconsistency indicator in which L is replaced by the conjunction of all AND-nodes, which were found by applying this algorithm. Updatable nodes, which belong to the same potential update tree and have the same root, belong to the same inconsistency tree. □

Example 4.2 Consider Example **2.4**. From the inconsistency tree in this example, now five revised inconsistency trees can be derived. The revised *one-level inconsistency trees* with respect to the inconsistency indicator *II1* are presented in Figure **4.2.** □



II1a: father(Z,Y), age_diff(X,Y,N), N < 15
II1b: husband(Z,X), age_diff(X,Y,N), N < 15
II1c: age_diff(X,Y,N), N < 15
II1d: parent(X,Y), age(Y,N2), N1 - N2 < 15
II1e: parent(X,Y), age(X,N1), N1 - N2 < 15
husband(Z,X)
father(Z,Y)
father(X,Y)
age(X,N1)
age(Y,N2)
husband(1,2)

Figure 4.2 *Revised Inconsistency Trees for Example* **2.4**.

5 Recursion in Integrity Checking Methods

The last condition in Definition **2.2** for potential update trees allows the application of recursive rules without getting infinite branches in such trees. Note that deductive databases only contain a finite number of rules with a finite number of arguments. Hence, the potential update tree is finite. The next example shows this finiteness in case of lineair recursion.

Example 5.1 Suppose a(X,Y) is the root literal of some potential update tree $T_{a(X,Y)}$. Let R: a(X,Y) $\longleftarrow$ b(X,Z), a(Z,Y) be the only rule. The left branch of $T_{a(X,Y)}$ consists of b(X,Z) and the right branch of $T_{a(X,Y)}$ consists of a branch with top literal a(Z,Y). Now by applying R to a(Z,Y) only one subnode b(Z,Z1) is derived. A subnode a(Z1,Y) differs only in the first argument from a(Z,Y), but these arguments are variables that do not occur in the root literal. So, a(Z1,Y) is redundant (redundant in the sense that there is no difference in instantiating root literal a(X,Y) if we instantiate either a(Z,Y) or a(Z1,Y); in both cases an update only binds variable Y). In Figure **5.1** the redundant branches of the potential update tree are indicated by the dotted arrows. □

The improvement of the first version of my proposed method (see [26]) with respect to recursion can best be clarified by an example.

Example 5.2 Suppose we have the following parent-facts and the following definition of the ancestor-relation in terms of the parent-relation:

```
parent(1,10).          ancestor(X,Y) :-
parent(1,11).            parent(X,Y).
parent(10,100).        ancestor(X,Y) :-
parent(11,110).          parent(X,Z),
parent(11,111).        ancestor(Z,Y).
parent(2,20).
parent(20,200).
parent(20,201).
```

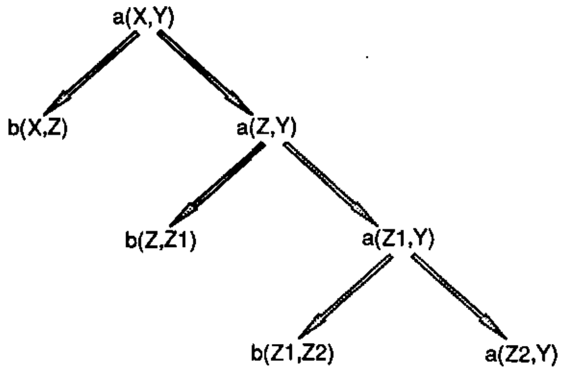


Figure 5.1 *Redundancy in the Potential Update Tree for Example* **5.1**.

The relations are presented as trees (see Figure **5.2**). A person is a parent of someone if there is an arrow from that person to the other. A person is an ancestor of someone if there is a path from that person to the other. Suppose there is an update to the parent-relation,

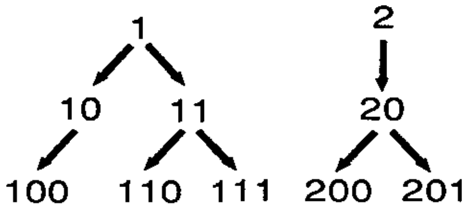


Figure 5.2 *The parent relation for Example* **2.6**.

say,

`update(parent(110,2)).`

What does this mean for the ancestor-relation? The ancestor-relation is updated by the update in the parent-relation too. This is represented in Figure **5.3**. The two parent trees in Figure **5.3** are connected by the update. The update in this tree is depicted as an outlined arrow. The begin node of the outlined arrow, representing the new parent-fact, is marked with X and the end node with Y. The update in the ancestor-relation can be described by the begin and end node of all paths from one node to another that have the outlined arrow in its path. In Figure **5.3** all possible begin nodes are marked with a Z, i.e., the node marked X or an ancestor of that node, and all possible end nodes with a Z1, i.e., the node marked Y or a node that has that node as an ancestor. Because all paths from a node marked with Z to a node marked with Z1 go through the outlined arrow, all new ancestors can be computed for each update in the parent-relation by

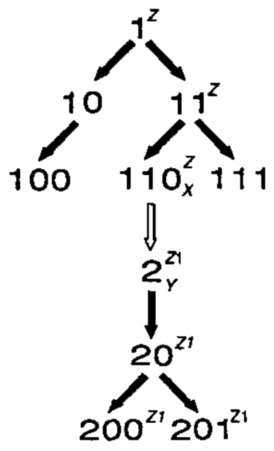


Figure 5.3 *The updated parent relation.*

```
update(ancestor(Z,Z1)) :-
  update(parent(X,Y)),
  (ancestor(Z,X) ; Z = X),
  (ancestor(Y,Z1) ; Z1 = Y).
```

Now, evaluating update(ancestor(Z,Z1)) gives all new ancestors implied by an update in the parent-relation. When the ancestor predicate appears in an inconsistency indicator an update in the parent-relation will affect the ancestor-relation. Suppose that this inconsistency indicator is:

```
ancestor(X,Y), age_diff(X,Y,N), N < 15.
```

□

In the first version of my method (see [26]) the inconsistency rules, which were generated to monitor the state of the database with respect to updates of the parent-relation, were:

```
inconsistent(parent(X,Y)) :-
  ancestor(X,Y), age_diff(X,Y,N), N < 15
inconsistent(parent(X,Z)) :-
  ancestor(X,Y), age_diff(X,Y,N), N < 15
inconsistent(parent(Z,Y)) :-
  ancestor(X,Y), age_diff(X,Y,N), N < 15
inconsistent(parent(Z,Z1)) :-
  ancestor(X,Y), age_diff(X,Y,N), N < 15
```

Note that the last inconsistency rule subsumes all other rules, for this rule corresponds to a full check of the inconsistency rule. The reason for stating the other rules is that this they could lead to an early detection of some inconsistency. Instead of a full check of the inconsistency indicator:

```
  ancestor(X,Y), age_diff(X,Y,N), N < 15
```

we can restrict ourselves to only the new instances of ancestor(X,Y). By using the results above the inconsistency rules can be replaced by the single rule:

```
inconsistent(parent(X,Y)) :-
   (ancestor(Z,X) ; Z = X),
   (ancestor(Y,Z1) ; Z1 = Y),
   age_diff(Z,Z1,N), N < 15.
```

Note that ancestor(Z,Z1) is replaced by (ancestor(Z,X) ; Z = X), (ancestor(Y,Z1) ; Z1 = Y), where X and Y are variables which are instantiated by the update in parent. This method is applicable to any kind of lineair recursive rule. This is a considerable gain in efficiency compared to other existing methods. In case of methods based on potential updates the check of constraints with lineair recursive parts will lead to a full check of inconsistency indicators. In the examples of Das and Williams ([9]) the use of recursive relations is permitted in rules but avoided in the inconsistency indicators itself. Hence, a full check of inconsistency indicators with recursion is avoided.

In the next section I have implemented and tested several examples of Das and Williams. I also implemented and tested an example with recursion in indicators based on the examples of Das and Williams in order to show the advantages of my method compared to others in case of recursion.

6 The Implementation

In general the coupling between a programming language and a database system can be achieved in two ways:

Loose Coupling: The programming language can obtain facts from the database by first copying these facts from the database in the working space memory of the language.

Tight Coupling: The programming language uses the facts in the database directly. This means from the database's point of view that it looks as if the programming language is an ordinary user, i. e., facts are retrieved from and stored to the database directly. Operations performed by the programming language on data of the database correspond directly to operations that are available in the database management system. In other words, the data are completely transparent to the programming language.

This last coupling enables the user to get all the functionalities a DBMS offers.

In order to test the methods presented in this paper, I have implemented these methods in Prolog. The tests were done in main memory of Prolog. So, a real coupling of

Prolog to a DBMS is not used here. The examples in the tests have been taken from Das and Williams ([9]). The examples used here all contain the following rules:

RULES

R1:	mother(X,Y) $\leftarrow$ husband(Z,X),father(Z,Y)
R2:	parent(X,Y) $\leftarrow$ father(X,Y)
R3:	parent(X,Y) $\leftarrow$ mother(X,Y)
R4:	ancestor(X,Y) $\leftarrow$ parent(X,Y)
R5:	ancestor(X,Y) $\leftarrow$ parent(X,Z), ancestor(Z,Y)
R6:	wife(X,Y) $\leftarrow$ husband(Y,X)
R7:	married(X,Y) $\leftarrow$ husband(X,Y)
R8:	married(X,Y) $\leftarrow$ wife(X,Y)
R9:	employed(X) $\leftarrow$ occupation(X,service)
R10:	student(Y) $\leftarrow$ occupation(Y,student)
R11:	dependent(Y,X) $\leftarrow$ parent(X,Y), employed(X), student(Y)
R12:	dependent(Y,X) $\leftarrow$ married(X,Y), employed(X), not employed(Y)
R13:	self(X) $\leftarrow$ married(Y,X), not employed(Y)
R14:	guardian(X,Y) $\leftarrow$ dependent(Y,X)

Each example contains a number of facts and one or more inconsistency indicators. The results of the tests confirm the considerations about different types of redundancy. Another example based on one of Das and Williams was introduced to test the performance of these methods when recursion is introduced in the inconsistency indicators.

6.1 Example A: Potential but no Induced Updates

In the following example a database is described for which no induced update can be generated from the update. However, applying the method based on potential updates, several potential updates are generated. Therefore, in this case, there is no instantiated inconsistency indicator generated in the method based on induced updates, while in the potential update method there are inconsistency indicators that have to be checked.

Suppose the database contains

1085 facts which do not involve constants 1 and 2:

177	father-facts,
229	husband-facts,
620	occupation-facts,
59	sponsor-facts,

and a list of facts which do involve constants 1 and 2:

occupation(1,service),
occupation(2,service).

Suppose the database is consistent with respect to the following inconsistency indicators:

II1: ∃X ∃Y guardian(X,Y), not sponsor(X,Y),

II2: ∃X ∃Y ∃Z sponsor(Z,Y), guardian(X,Y), not parent(Z,Y).

Consider the update: married(1,2).

Note that the relation married is supposed to be updatable. Note also that in this case there is no redundant evaluation of any type for the method based on induced updates. In the other methods, even instantiated inconsistency indicators are generated and evaluated; this is a typical example of redundancy of the second type. By evaluating the instantiated indicators, subtrees are searched for inconsistencies while they are not updated. Hence, here also redundancy of the third type appears. Because redundancy of the third kind does not exist anymore in the proposed method based on revised inconsistency rules, the performance is better than the previous method based on inconsistency rules. As one would expect, Table 1 shows that the method of potential updates does not perform well compared to the other methods and that the proposed method performs as well as the induced update method in this particular case. It is important to note that in this example,

Method	**Time**
Induced	0.055
Potential	0.17
Inconsistency Rules	0.11
Inconsistency Rules (Proposed)	0.055

Table 1 *Timings of several methods for Example A.*

applying the method of induced updates, there will be no evaluation of an inconsistency indicator, while in the case of the proposed method evaluation of instantiated indicators is necessary. However, these two methods perform equally well. In the construction of the inconsistency rules, the body of the inconsistency rule is determined by a bottom-up development of potential update trees. For instance, when a literal guardian(X,Y) appears in an inconsistency indicator, the potential update tree for guardian(X,Y), see Figure **6.1**, is used for deriving the inconsistency rules. When looking at this construction procedurally, the order of the literals in the body of the inconsistency rule is important. First, the literal, for which the potential update tree is used, is put in front of the inconsistency

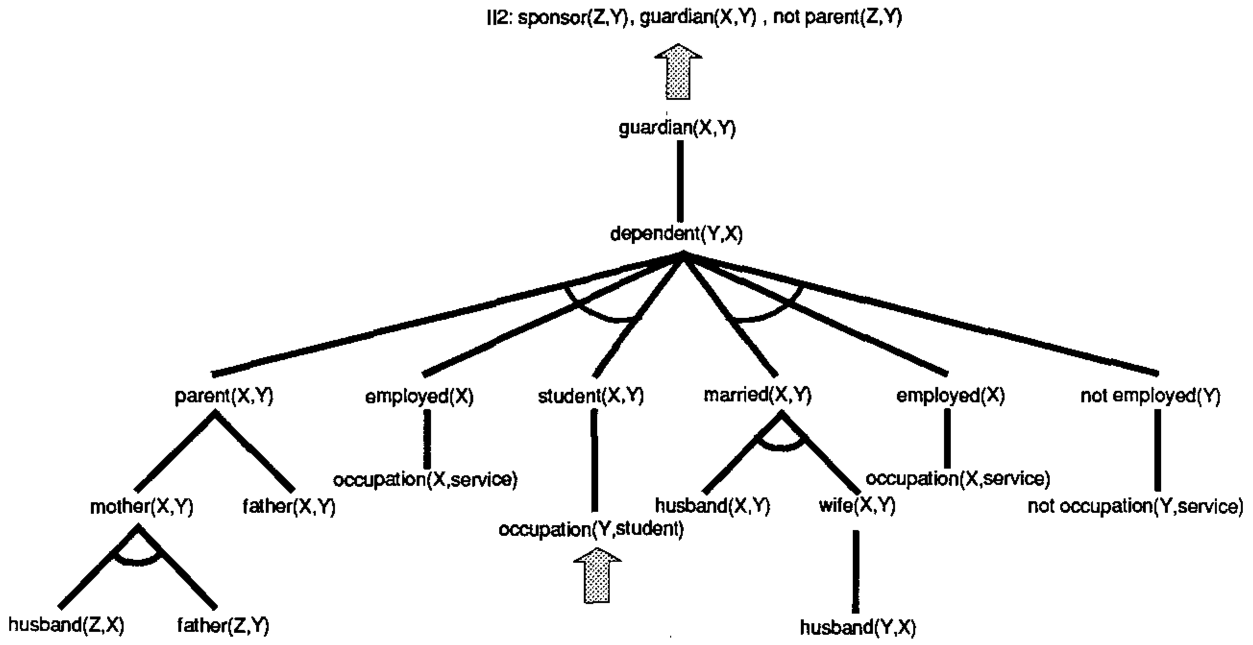


Figure 6.1 *Potential Update Tree for guardian(X,Y).*

indicator. This literal is replaced by a conjunction of literals. So, the resolvent is always at front. In the conjunction itself a literal appears in front of another if it corresponds to a descendent node in the potential update tree. By maintaining the order of appearance, the order of evaluation of the body of the inconsistency rule corresponds to the way the induced updates are computed. Therefore, the evaluation of an inconsistency rule stops just when no more induced updates can be computed.

For instance, suppose we are only interested in the inconsistency rule for *II2* with respect to an update in occupation(Y,student) for some Y. Then guardian(X,Y) in, for example, the indicator $\exists$ X $\exists$ Y $\exists$ Z sponsor(Z,Y), guardian(X,Y), not parent(Z,Y) is put in front of the indicator and replaced by parent(X,Y), employed(X). The variable Y in parent(X,Y), employed(X) is instantiated by the update and gives all new instances of dependent(Y,X) and by applying rule *R11* all new instances of guardian(X,Y). The inconsistency rule with respect to this indicator and update is therefore:

```
inconsistent(occupation(Y,student)) :-
  parent(X,Y), employed(X), sponsor(Z,Y), not parent(Z,Y)
```

Remember that an update occupation(Y,student) implies always an induced update student(Y). Possibly this leads to several induced updates dependent(Y,X) depending of the true evaluation of parent(X,Y), employed(X) in rule *R11*. For each induced update dependent(Y,X) there will always be an induced update guardian(X,Y) by the application of rule *R14*. So, by putting parent(X,Y), employed(X) in front, first the induced updates in the relation guardian are computed. This is shown in Figure **6.2**. In the construction only the continuous lines are important. Nodes that are ancestor nodes of the updated node are not mentioned in the body of the inconsistency rule. Only the and-nodes of these ancestor nodes are collected and are collected in the order of appearance. Note that in the

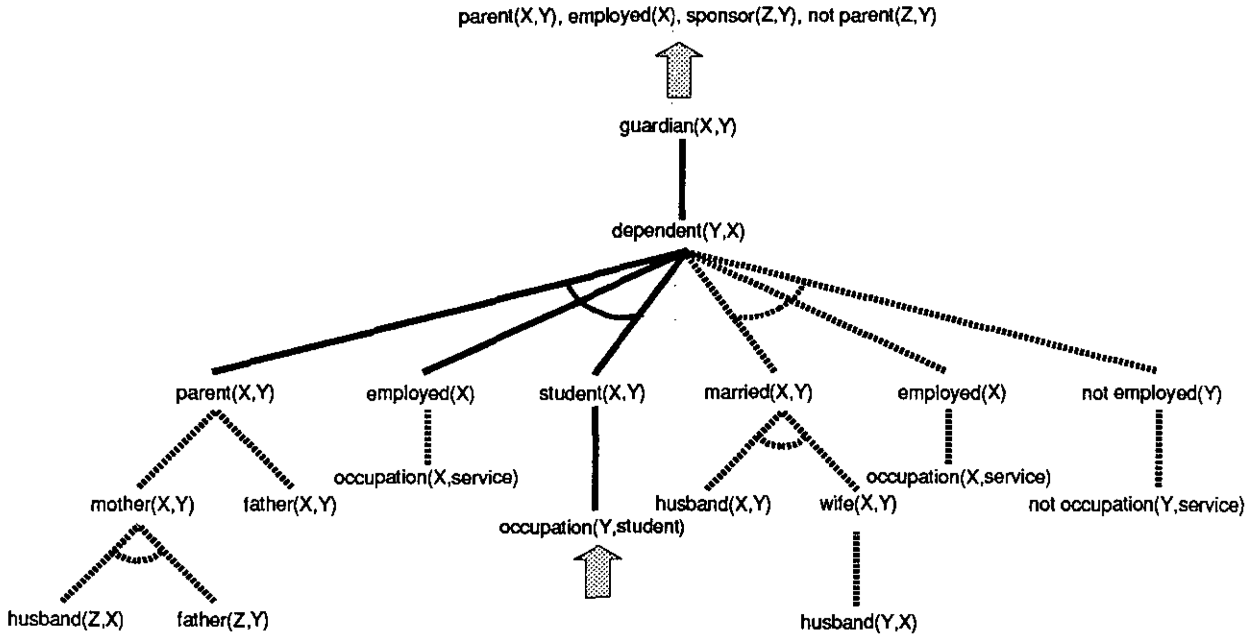


Figure 6.2 *Construction of an inconsistency rule for an update occupation(Y,student).*

check of the proposed method, induced updates that do not require any computation, like for instance induced updates student(Y) and guardian(X,Y) when induced updates occupation(Y,student) and dependent(Y,X) respectivily are already computed, are not involved in the evaluation.

6.2 Example B: Few Potential Updates, many Induced Updates

In this example a database state is created which shows that the redundancy of the first type can be disastrous in the case of methods based on induced updates.

Suppose the database contains

1058 facts which do not involve constants 1,2 and 3:

184	father-facts,
226	husband-facts,
600	occupation-facts,
48	sponsor-facts,

and a list of facts which do involve constants 1,2 and 3:

occupation(2,service),
occupation(3,student),
father(1,3).

Suppose the database is consistent with respect to the following inconsistency indicator:

II1: $\exists$X $\exists$Y $\exists$Z father(X,Z), father(Y,Z), not X = Y.

Consider the update: husband(1,2).

In this example, the update does not influence the inconsistency indicator. So, the update should be accepted immediately. The method proposed here does accept the update immediately, since no inconsistency rule exists for which the update is relevant. But what happens when all induced updates are generated? Because of the list of ten facts under the predicate father with 1 as first argument, rules *R1*, *R3*, *R4*, *R6*, *R7*, *R8* and possibly also *R5*, *R11–R14* produce a considerable number of induced facts. None of them are relevant with respect to the indicator. In this case the method based on induced updates performs poorly because redundancy of the first type has a great influence.

In the case of potential updates the redundancy of the first kind is less influential. Note that the update causes many induced mother-updates, while only one potential mother-update is generated. Each rule produces at most one potential update. That is why the method based on potential updates performs relatively well compared to the method based on induced updates. Table **2** shows the results of the tests for this example. In this case, the full check of the inconsistency indicator, known as the naive method, is more efficient than the method based on induced updates.

6.3 Example C: Update Relevant for Inconsistency Indicator

Consider Example **B** where the inconsistency indicator is replaced by an inconsistency indicator which will be influenced by the update: Suppose the database contains

1058 facts which do not involve constants 1 and 3:

175	father-facts,
228	husband-facts,
620	occupation-facts,
72	sponsor-facts,

Method	Time
Naive	1.86
Induced	3.35
Potential	0.27
Inconsistency Rules	< 0.01
Inconsistency Rules (Proposed)	< 0.01

Table 2 *Timings of several methods for Example B.*

and a list of facts which do involve constants 1 and 3:

```
occupation(1,service),
occupation(3,student),
father(1,3).
```

Suppose the database is consistent with respect to the following inconsistency indicator:

II1: $\exists$X $\exists$Y $\exists$Z guardian(X,Z), guardian(Y,Z), not X = Y.

Consider the update: husband(1,2). The indicator expresses the fact that a person has

Method	Time
Induced	0.60
Potential	0.87
Inconsistency Rules	0.27
Inconsistency Rules (Proposed)	0.11

Table 3 *Timings of several methods for Example C.*

only one guardian. This example is not used in [9]. The results of this test is presented in Table 3. It is intended to show that when updates influence inconsistency indicators, the improvement of the proposed method based on inconsistency rules is significant, because the redundancy of the third type does not exist in this method.

6.4 Example D: Recursiveness in Inconsistency Indicators

Suppose the database contains a list of facts based on the situation in Example 5.2, where the parent-relation is replaced by the father-relation. Suppose the database contains

1085 facts which do not involve constants 1,10,11,100,110,111,2,20,200 and 201:

177	father-facts,
229	husband-facts,
620	occupation-facts,
59	sponsor-facts,

and a list of facts which defines two families,

```
father(1,10),
father(1,11),
father(10,100),
father(11,110),
father(11,111),
father(2,20),
father(20,200),
father(20,201).
```

Suppose the database is consistent with respect to the following inconsistency indicator:

II1: ∃X ∃Y ancestor(X,Y),ancestor(Y,X)

Consider the update: father(110,2).

In this example the proposed method based on inconsistency rules performs better than any other method. Note, that the method based on induced updates is also a good alternative. This has two causes. First, the update is generating mainly new ancestor-facts. which are all relevant to the inconsistency indicator. So, here the redundancy of the first type does not have a great influence. The second cause is that the induced updates in the ancestor-relation have been computed by using the expression

```
update(ancestor(Z,Z1)) :-
  update(parent(X,Y)),
  (ancestor(Z,X);Z = X),
  (ancestor(Y,Z1);Z1 = Y).
```

This is a considerable improvement compared to a more conventional appoach. This is showed by the results in Table 4. Note that in some particular cases methods based on induced updates and potential updates perform very poorly compared to the proposed method. Analysing the database rules, facts and constraints together with the test results, I was able to find the causes of these bad performances. These results were very helpful for the development of the revised method based on inconsistency rules. In all cases, the revised method will perform better than the other methods.

Method	Time
Induced	5.7
Potential	165.1
Inconsistency Rules	41.3
Inconsistency Rules (Proposed)	5.0

Table 4 *Timings of several methods for Example D.*

7 Conclusions

The main goal of this paper was to show that the proposed method is minimal when the types of redundancy are considered. In section **6** some integrity checking methods for deductive databases were applied and tested. As the results show the proposed method is efficient compared to the other existing methods.

Although the response time is rather high, some notes must be made. As the results show, the access time to the database contributes a lot to the overall performance. The efficiency can be improved by optimising techniques applied to rules and inconsistency rules such as indexing, parallel processing, etc.. The tests were done on a PC with a 386-processor; so, the response time for the tests can be reduced using larger and faster computers.

This method is also very suitable for parallel processing because all inconsistency rules are independent of each other. Because the inconsistency rules are known before the transactions are performed, the inconsistency rules can be optimized at compile time. It is even possible to translate the inconsistency rules to SQL-expressions which are executed in the database directly after each transaction. Therefore, I am optimistic about the applicability of this method even when the databases are bigger than the ones which were used in the tests.

The proposed method can be implemented in a straightforward manner in Prolog. A meta-interpreter in order to be able to reason forward is not necessary, because the inconsistency rules are Prolog rules. For a deductive database consisting of a fixed set of rules and constraints the set of inconsistency rules has to be computed once. When this set of rules and constraints is not changed the inconsistency rules can be used for each transaction consisting of facts. By an update in the rule and constraint set only a slight change in inconsistency rule set is needed, namely, the new inconsistency rules affected by the rule resp. constraint update. This is an issue for further research, but seems to be one of the major advantages of the use of inconsistency rules.

Another strong point of the new method is that it is conceptually as clear as in the relational case, i.e., after an update the real checking of the constraints is started.

All these advantages makes this new method very promising for checking the consistency of the fact base in expert database systems.

8 Future Directions

A lot of other work still needs to be done. Several issues for further investigation are:

- allowing a more general set of inconsistency indicators; for instance, indicators with universal quantifiers as well or database functions such as counting,
- allowing more general updates such as rule updates and determining if the rule update violates a database constraint,
- the possibility of indexing the inconsistency rules in order to make the matching of the update with these rules more efficient,
- the time complexity when varying the number of facts, rules and/or inconsistency indicators or the density[2] of the database facts compared to the other methods.
- the growth of the number of generated inconsistency rules and space complexity issues when increasing the number of rules and/or inconsistency indicators. Also the space complexity of the other methods have to be compared.

As far as I can judge at this moment, the results are encouraging with respect to the complexity issues. Technically, extensions to more general updates seem to be no problem at all. It seems that this new method is a big step towards a fast integrity checking module for expert database systems.

9 Acknowledgements

I would like to thank Professor H. de Swart of Tilburg University and Dr. P. de Bra of the university of Eindhoven for their comments on the earlier versions of this paper. Their remarks contributed to the final version of this paper.

[2]the density is called high if there are a lot of deducible facts compared to the existing facts in the database

References

[1] P. Asirelli, P. Inverardi, , and A. Mustaro. Improving integrity constraint checking in deductive databases. In M. Gyssens, J. Paredaens, and D. van Gucht, editors, *Proceedings of the 2nd International Conference on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, pages 72–86, Bruges, Belgium, August-September 1988.

[2] P[atricia] Asirelli, Michèle de Santis, and Maurizio Martelli. Integrity constraints in logic databases. *J. of Logic programming*, 3:221–232, 1985.

[3] Petra Bayer. Update propagation for integrity checking, materialized view maintenance and production rule triggering. Technical Report 92-10, ECRC GMBH, Arabellastr. 17 D-8000 München 81, Germany, 1992.

[4] François Bry, Hendrik Decker, and Rainer Manthey. A uniform approach to constraint satisfaction and constraint satisfiability deductive databases. In J.W. Schmidt, S.Ceri, and M. Missikoff, editors, *Advances in Databases Technology, EDBT '88; Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 488–505, Venice, Italy, 7 Nov. 1987. also: ECRC Technical Report KB-16.

[5] M. Celma, C. García, L. Mota, and H. Decker. Comparing and synthesizing integrity checking methods for deductive databases. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 214–222, Houston, Texas, february 1994. IEEE Computer Society Press.

[6] Matilde Celma, Juan Carlos Casamayor, and Hendrik Decker. Improving integrity checking by compiling diravation paths. In *Proceedings of the 4th Australian Database Conference*, pages 145–160, 1993.

[7] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. 15(2):160–207, June 1990.

[8] S.K. Das and M. Howard Williams. A path finding method for constraint checking in deductive databases. *Data & Knowledge Engineering*, 4:223–244, 1989.

[9] S.K. Das and M.H. Williams. Integrity checking methods in deductive databases: A comparative evaluation. In M.H. Williams, editor, *Proceedings of the Seventh British National Conference on Databases*, pages 85–116. Cambridge University Press, 1989.

[10] H. Decker. Integrity enforcement on deductive databases. In Larry Kerschberg, editor, *Expert Database Systems: Proceedings from the First International Conference*, pages 271–285. Charleston, Sc., 1987.

[11] Hendrik Decker. Drawing updates from derivations. In S. Abiteboul and P.C. Kanellakis, editors, *Proceedings of the third International Conference on Database Theory*, volume 470 of *Lecture Notes in Computer Science*, pages 437–451, Paris, France, December 1990.

[12] Robert Kowalski, Fariba Sadri, and Paul Soper. Integrity checking in deductive databases. In Peter M. Stocker and William Kent, editors, *Proceedings of the Thirteenth Conference on Very Large Data Bases*, pages 61–69, Brighton, England, August 1987.

[13] V. Küchenhoff. On the efficient computation of the difference between consecutive database states. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases; Proceedings of the Second International Conference, DOOD'91*, volume 566 of *Lecture Notes in Computer Science*, pages 478–502, Munich, Germany, December 1991.

[14] Sin Yeung Lee and Tok Wang Ling. Improving integrity constraint checking for stratified deductive databases. In Dimitris Karagiannis, editor, *Lecture Notes in Computer Science*, volume 856, pages 591–600, Athens,Greece, september 1994. Springer Verlag.

[15] Tok-Wang Ling. Integrity constraint checking in deductive databases. *Data & Knowledge Engineering*, 2, 1994.

[16] J.W. Lloyd. *Foundations of Logic Programming; 2nd, Extended Edition.* Springer Verlag, 1987.

[17] J.W. Lloyd, E.A. Sonenberg, and R.W. Topor. Integrity constraint checking in stratified databases. *J. of Logic Programming*, 4:331–343, 1987.

[18] J.W. Lloyd and R.W. Topor. A basis for deductive database systems. *J. of Logic Programming*, 2:93–109, 1985.

[19] J.W. Lloyd and R.W. Topor. A basis for deductive database systems ii. *J. of Logic Programming*, 3(1):55–67, 1986.

[20] Bern Martens and Maurice Bruynooghe. Integrity constraint checking in deductive databases. In Larry Kerschberg, editor, *Expert Database Systems: Proceedings from the First International Work shop*, pages 567–601. Charleston, Sc., 1986.

[21] Amihai Motro. Using integrity constraints to provide intensional answers to relational queries. In Peter M.G. Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 237–246, Amsterdam, The Netherlands, 1989.

[22] J.M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica*, 18(3):227–253, 1982.

[23] J.M. Nicolas and K. Yazdanian. Integrity checking in deductive data bases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 325–346, New York, 1978. Plenum Press NY.

[24] Antoni Olivé. Integrity constraint checking in deductive databases. In Guy M. Lohman, Almicar Sernadas, and Rafael Camps, editors, *Proceedings of the Seventeenth Conference on Very Large Data Bases*, page 513, Barcelona, Spain, September 1991.

[25] Fariba Sadri and Robert Kowalski. A theorem-proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 313–362, Los Altos, 1988. Morgan Kaufmann.

[26] Ron R. Seljée. A new method for integrity constraint checking in deductive databases. *Data & Knowledge Engineering*, pages –, 1995.

93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correct-ness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distribu-ted Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS cir-cuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.

93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Komatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.

94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.
94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\!\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.

94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.
94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma J. Davies R. Gerth S. Graf W. Janssen B. Jonsson S. Katz G. Lowe M. Poel A. Pnueli C. Rump J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.

94/47	R. Bloo F. Kamareddine R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper	Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien	The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem R. Bol J.F. Groote	Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten C. Verhoef	Concrete process algebra, p. 134.
95/04		
95/05	P. Severi	A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen M.G.A. Vehoeven H.M.M. ten Eikelder E.H.L. Aarts	A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn O.S. van Roosmalen	Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo	Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten J.A. Bergstra	Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse R. Verhoeven O. Weber	Math∫pad: A System for On-Line Prepararation of Mathematical Documents, p. 15