A new method for integrity constraint checking in deductive databases

Citation for published version (APA):

Seljée, R. R. (1994). *A new method for integrity constraint checking in deductive databases*. (Computing science notes; Vol. 9413). Technische Universiteit Eindhoven.

Document status and date:

Published: 01/01/1994

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Eindhoven University of Technology
Department of Mathematics and Computing Science

A New Method for Integrity
Constraint Checking in Deductive
Databases

by

R. Seljc

94/13

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

A NEW METHOD FOR INTEGRITY CONSTRAINT CHECKING IN DEDUCTIVE DATABASES

Ron Seljée,*
Co-operation Centre
Tilburg and Eindhoven Universities
the Netherlands
e-mail: seljee@kub.nl

March 14, 1994

Abstract

In the literature, several integrity checking methods for updates in deductive databases are described. All these methods try to instantiate the specified integrity constraints with the update in order to constrain the full check to only a relevant part of the database. Globally, they can be divided in two major classes of methods; methods based on "induced updates" and methods based on "potential updates". In this article a new method will be presented. This method represents also a new class of methods. While in the first two classes one has to generate induced updates and potential updates respectively, even if they are not relevant to any of the constraints, the new method, which will be called the "method based on inconsistency rules", does not have this drawback. Therefore, the proposed method is potentially far more efficient than any other method based on induced updates or potential updates.

1 INTRODUCTION

The relational database model is in several cases not sufficiently adequate to model the world which we want to describe. Therefore, this model is extended in several ways. In this article we consider relational databases extended with rules and integrity constraints. These rules and constraints introduce more knowledge about the world we want to model into the database. Relational databases extended with (backward chaining) rules are also called deductive databases. In this article, when we speak about deductive databases we also consider the specified integrity constraints.

*I would like to thank H. de Swart (KUB) and P. de Bra (TUE) for their comments on the earlier versions of this paper. Their contributions helped a lot in making the final version of this paper.

However, introducing rules and constraints is not straightforward.

In the first place, it is possible that from the rules and the facts in the database some other facts are deducible, which all have to obey the integrity constraints as well. And in some cases, for instance in case of recursive rules, this could lead to a large number of deducible facts. For example, a recursive rule “all ancestors of a person Y are ancestors of a person X if Y is an ancestor of X”, where every parent of a person is an ancestor of that person together with a large number of parent-facts could lead to an enormous amount of ancestor-facts. Note, that this recursiveness is not allowed in today’s relational systems. In the second place, checking the integrity of the database as a whole by checking all constraints each time the database is changed is not feasible. Therefore the presently available methods make use of the assumption that a database is consistent with the specified integrity constraints before each update of the database. With this assumption we can restrict ourselves to only those constraints which are affected by a certain change of the database. In a relational database inserting a fact that John did not pass his final exam for history, does not affect the integrity constraint which states that all classrooms can contain at most thirty people. So, this integrity constraint does not have to be verified because it was satisfied in the previous database state. In the deductive case, we have a new problem caused by rules. With respect to some update, the rules can generate several induced updates, which in their turn can cause an inconsistent database state. For instance, consider the previous example, in which John did not pass his final exam of history, in the deductive case. Suppose it was known that John was in the third grade and all thirty students in the second grade did pass all their final exams. Suppose also that this database contains two rules which state that all students who have passed all their final exams will promote to the next grade, and that all students who have not passed all final exams will be in the same grade as before. Now the same constraint as before—all classrooms can contain at most thirty people—will not be satisfied when we insert a fact that John did not pass his final exam for history, because with the rules applied to the facts and the update this implies that the third grade now consists of John and all the thirty promoted students of the second grade.

Another issue is *“how do we restore the integrity of the database after an update which leads to an inconsistent database”*. We call this the issue of integrity maintenance. Here in this article, we only deal with the issue of integrity checking. This means that we are only interested in how to detect in the most efficient manner that the database is inconsistent. Here, we are not interested in how to restore this inconsistency. If an update causes an inconsistent database state, we say that an update is not allowed and therefore rejected. For instance, in the given example we can refuse to accept that John did not pass his history exam. Note that we could restore the database integrity by splitting the third grade in two groups each group having one separate classroom. So we have two classrooms containing respectively fifteen and sixteen students. (Provided this is not forbidden by other constraints). So, a deductive maintenance system somehow could take actions when an update leads to an inconsistent database state. However, some efficient integrity checking mechanism is essential to a deductive maintenance system.

2 FORMAL DESCRIPTION

In a deductive database one distinguishes facts, rules and integrity constraints. These facts, rules and constraints will now be expressed in a logical language, the first-order logic. A lot of formal work in this area was done by Nicolas, Lloyd and others. (see [NIC82], [NIY78], [LLT85], [LLT86], [LST87]). The symbols used in this language are (1) parentheses, (2) variables and constants, (3) predicate symbols, (4) logical connectives like $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\leftarrow$ (implication), and (5) quantifiers $\forall$ (for all) and $\exists$ (there exists). Throughout the paper we use uppercase letters to represent variables and lowercase letters or words to represent constants and predicate names. More about the definitions and logical concepts in this section can be found in [LLO87]. In our databases terms are supposed to be function-free, i.e., a *term* is a constant or a variable and not a function. We call $p(t_1, t_2, \dots, t_n)$ an **atomic formula**, or an **atom**, if p is a predicate symbol of arity n and $t_1, t_2, \dots, t_n$ are terms. An atomic formula or its negation is called a *literal*. We also call atoms and negated atoms **positive** and **negative literals** respectively. The expressions allowed in this language, the well-formed formulas (wffs), are defined recursively as follows.

1. Any literal is a wff.
2. If w_1 and w_2 are wffs, then $(\neg w_1)$, $(w_1 \vee w_2)$, $(w_1 \wedge w_2)$ and $(w_1 \leftarrow w_2)$ are wffs, and $\forall X[w_1(X)]$ and $\exists X[w_2(X)]$ are wffs if X is free in w_1 and w_2 respectively. (In $\forall X[w_1(X)]$ (resp. $\exists X[w_2(X)]$) we call $w_1(X)$ (resp. $w_2(X)$) the *scope* of $\forall X$ (resp. $\exists X$). A variable X is *bound by* $\forall X$ (resp. $\exists X$) if it occurs in the scope of $\forall X$ (resp. $\exists X$). A variable X in an expression is called *bound* if it is bound by $\forall X$ (resp. $\exists X$) or its occurrence is in $\forall X$ (resp. $\exists X$). A variable is called *free* if it is not bound.)
3. The only wffs are those given by 1), 2) and 3).

A formula is called *closed*, if it contains no free variables. Otherwise, it is called *open*. A formula in which no variables occur is called a *ground* formula.

Facts, rules and constraints are defined as follows.

facts are ground atoms.

rules are clausal form expressions $B \leftarrow A_1 \wedge A_2 \wedge \dots \wedge A_n$, where B is a positive literal and $A_1, A_2, \dots, A_n$ are literals, positive or negative. B and $A_1 \wedge A_2 \wedge \dots \wedge A_n$ are called the *head* and the *body*, respectively, of the rule $B \leftarrow A_1 \wedge A_2 \wedge \dots \wedge A_n$.

Moreover, a variable which occurs in B or in a negative literal of the body of the rule must also occur in a positive literal of the body of the rule. In other words, the rules must be *range-restricted*. Suppose a variable in a negative literal A' of a rule R does not occur in a positive literal of the body of R . The instances of the head of the rule will depend of the database domain as a whole. Application of such rules would require a complete domain search, which is in most cases extremely inefficient.

Constraints are closed first-order formulas. Moreover, we consider only a subclass of these formulas, namely the *universally quantified* formulas. These formulas have the following form:

$$\forall X_1 \cdots \forall X_n [L_1 \wedge \cdots \wedge L_m \rightarrow Q],$$

or equivalently:

$\neg(\exists X_1 \cdots \exists X_n [L_1 \wedge \cdots \wedge L_m \wedge (\neg Q)])$, where $m, n \geq 0$, each L_i is a literal, each variable X_i , $i = 1, 2, \dots, n$, occurs in one or more L_j , $j = 1, 2, \dots, m$, and Q is a literal. We assume that the constraints are also range-restricted. (Note that Q could be a negative literal, which is completely instantiated when the body is instantiated too since the formula is range-restricted. By allowing negative literals in the head of constraints we can express that some facts are not allowed in the database). Now, each constraint has the form $\neg F$, where F is some existentially quantified formula. In this article we build our theory by using F instead of $\neg F$. We call F the *inconsistency indicator* of the constraint. The new proposed method is easier to describe when dealing with the concept of inconsistency indicator rather than 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. The database is consistent if there is no specified inconsistency indicator that is true in the database.

Throughout this paper we assume that updates are represented by ground literals. Let U represents 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 all at once presented to a database. Let T be a transaction, then the database D which is the result of all updates in the set is denoted by D_T . We call D_U (resp. D_T) the database D extended by U (resp. T).

Definition 1 An inconsistency indicator II is *relevant to a set of literals* S iff there exists a literal L in S that is unifiable with a literal of II . In particular we say that II is relevant to L .

A set of literals S is *relevant to an inconsistency indicator* II iff there exists a literal L in S that is relevant to II . In particular we say that L is relevant to II .

Note that an inconsistency indicator is range-restricted iff the corresponding constraint is range-restricted. This follows directly from the definition of the range-restrictedness and from the definition of an inconsistency indicator.

Lemma 1 A database D is consistent with its specified constraints iff all the related inconsistency indicators are false in the database.

Proof: The database is consistent with its specified constraints means that all constraints are true in D . So, the proposition follows by definition, because the inconsistency indicators are negations of the constraints. $\square$

In the remaining part of this paper we only speak about inconsistency indicators instead of integrity constraints when the consistency of a database is concerned. So, the database D is inconsistent if there is some inconsistency indicator which is true in D . A database is consistent if all inconsistency indicators are false in D . The main reason to drop the concept of constraint here is that the proposed method heavily depends upon the concept of inconsistency indicator. Changing of concept at a later stage could lead to an article which is less easy to understand and which could lead to several reformulations of definitions and propositions.

In order to determine whether an updated database is consistent all inconsistency indicators could be checked. But this is in most cases a time consuming task. It turns out that we can restrict ourselves to “simplified instances” of only those indicators that are relevant to an update (see Proposition 1). So, the collection of indicators which have to be checked can often be reduced considerably.

Definition 2 Let $II = \exists X_1 \dots \exists X_n [L_1 \wedge \dots \wedge L_m]$ be an inconsistency indicator relevant to a literal A , where $L_1, L_2, \dots, L_m$ are literals (positive or negative). So, there is a literal L_j which is unifiable with A . Let σ be a most general unifier of L_j and A . We call $II\sigma$ a *simplified instance* of II with respect to A .

Definition 3 Let U be an update and II an inconsistency indicator relevant to U . The simplified instance of II with respect to U is now called an *update instance* of II with respect to U .

Note that from an update and the specified inconsistency indicators a set of update instances can be generated which is possibly empty in the case of an update which is not relevant to any indicator.

First we take a look at the relational case. So, in this situation no rules are involved. Update instances play an important role in checking the consistency of a relational database after an update as we can see in the following proposition.

Proposition 1 Let D be a relational database. Let U be an update. Suppose D is consistent. Then D_U is consistent iff all update instances of inconsistency indicators with respect to U are false in D_U .

PROOF:

$\Rightarrow$ Suppose D_U is consistent; then all inconsistency indicators are false in D_U . (See Lemma 1) So, D_U can be seen as an interpretation in which all inconsistency indicators are false. Suppose there was an update instance I of some indicator II true in the interpretation D_U . Because I is an instantiation implied by U and $U \in D_U = D \cup \{U\}$; II should also be true in the interpretation D_U . Contradiction.

So, there is no update instance true in D_U . (Note that in proving this part of the proposition we do not need the assumption that D is consistent).

$\Leftarrow$ Suppose all update instances with respect to U are false in D_U . In order to prove that D_U is consistent we have to prove that all inconsistency indicators are false in interpretation D_U . First the collection of all inconsistency indicators, say $\mathcal{I}$, is divided in two disjunct sets:

- $\mathcal{I}_U$; the set of all inconsistency indicators relevant to U , and
- $\mathcal{I}_U^C = \mathcal{I}/\mathcal{I}_U$; the set of all inconsistency indicators not relevant to U .

Let $I \in \mathcal{I}_U^C$. Because of the assumption that D is consistent, I is false in D . Because I is not relevant to U only the facts in D are involved in satisfying I in the interpretation D_U . So, because of the assumption that D is consistent, i.e., I is false in D , this implies that I is false in D_U .

Now, let $I \in \mathcal{I}_U$. Let L be a literal of I which is unifiable with U . Let σ_1 be a most general unifier of U and L . We have to prove that I is false in D_U . Suppose I was true in D_U . Because I is false in D , $I\sigma_1$ must be true in D_U . But $I\sigma_1$ is an update instance of I with respect to U . So, by hypothesis, $I\sigma_1$ is false in D_U . Contradiction. $\square$

Proposition 1 is illustrated by figure a. Note that in the relational case we do not have rules yet as in the deductive case. So, we could see a relational database as a deductive one without rules. Note also that here an update leads directly to instantiations of constraints.

Figure a *Overview of integrity checking in the relational case.*

Example 1 Let D be a relational database defined as follows. Let $II = \exists X \exists Y [q(X, Y) \wedge \neg p(X)]$ be the only inconsistency indicator and suppose the fact-base consists of the following facts:

$p(b)$	$q(b, b)$
$p(c)$	$q(b, c)$
$p(e)$	$q(c, a)$

It is easy to check that D is consistent.

Suppose $q(c, b)$ is an update. The update instance that is derived and which has to be false is $\neg p(c)$. Now $\neg p(c)$ is false because $p(c)$ is true in D_U . So the update is allowed.

Now suppose the update is the deletion of $p(b)$. Then the update instance of II is $\exists Y[q(b, Y)]$. This instance is true in D_U . So the deletion of $p(b)$ is not allowed.

Note that an insertion $p(a)$ does not influence the inconsistency indicator and is therefore allowed.

When we had to falsify the whole inconsistency indicator instead of some instance of it we must subsequently instantiate II using each $q(.,.)$ -fact in D_U and falsify each of these instances. In cases of a great number of such facts the falsification could take a lot more time than when only falsifying the update instance.

One can easily generalise proposition 1 for one update to proposition 2 for a set of updates.

Proposition 2 Let D be a consistent relational database. Let T be a transaction. The D_T is consistent iff all update instances of each II with respect to each update in T are false in D_T .

Example 2 Let D be the consistent database which was defined in example 1. Let $T = \{p(a), q(a, b)\}$ be a transaction. The update instance that must be false in D_T is $\neg p(a)$. Note that from $p(a)$ no update instance is derived and that the update $p(a)$ in T assures the falsification of $\neg p(a)$.

So also note that in proposition 2 " D_T is consistent" does not hold when all update instances of each II with respect to update U in T are false in D_U instead of D_T . For instance in this example update $q(a, b)$ is not false in $T_{q(a, b)}$.

In the relational case falsification of inconsistency indicators in databases is straightforward. However, in the deductive case this is not trivial. Deductive databases contain rules. By the generating capabilities of rules an inserted fact can lead to a number of induced updates. These induced updates can in their turn also affect the integrity of the database. So, in deductive databases integrity maintenance is more complicated than in relational databases.

Globally, the existing methods can be divided in two major classes of methods; methods based on *induced updates* and methods based on *potential updates*. In this paper a new method will be presented, which represents a new class. The new method is based on a new concept called "*inconsistency rules*". In the next sections each of these methods will be explained.

3 METHOD BASED ON INDUCED UPDATES

In deductive databases one has to deal with rules also. Now, an update may cause several other implicit changes to the database. These changes are called *induced updates*. Not only the update but also these induced updates may lead to a true instance of an inconsistency indicator.

We describe the concept of induced update more formally:

Definition 4 Let $C \leftarrow A_1 \wedge A_2 \wedge \dots \wedge A_n$ be a deductive rule R . Let U be an update and D_U the updated database of D . Let L be a positive (resp. $\neg L$ a negative) literal which is unifiable with A_i (resp. the complement of A_i) in R , for some i . Let γ be the most general unifier of L and A_i . Let $C' = (C\gamma)\sigma$, where σ is a substitution by which $(A_1 \wedge \dots \wedge A_{i-1} \wedge A_{i+1} \wedge \dots \wedge A_n)\gamma\sigma$ is true in D_U , and C' evaluates to false in D (resp. $\neg C'$ evaluates to true in D). Here, rules are range restricted. So, every variable of C appears also in one or more $A_1, A_2, \dots, A_n$, which means that C' is ground. Then C' (resp. $\neg C'$) is now called *directly induced by L over D_U* . A literal is *induced by L over D_U* iff

1. it is directly induced by L over D_U , or
2. it is directly induced by a literal induced by L over D_U .

Definition 5 Let D be a deductive database and let U be an update. Each literal induced by update U over D_U is called *an induced update induced with respect to update U* (or simply *induced update* if it is clear from the context which update is involved).

Definition 6 Let U be an update. Let L be an induced update and II an inconsistency indicator relevant to L . The simplified instance of II with respect to L is now called *an induced instance of II with respect to U* .

The next example, considered by Das and Williams ([DAW 89]), contains the following set of rules, inconsistency indicators (Note: Das and Williams use constraints) and facts:

RULES

- R1: $\text{mother}(X,Y) \leftarrow \text{father}(Z,Y), \text{husband}(Z,X)$
- R2: $\text{parent}(X,Y) \leftarrow \text{father}(X,Y)$
- R3: $\text{parent}(X,Y) \leftarrow \text{mother}(X,Y)$
- R4: $\text{ancestor}(X,Y) \leftarrow \text{parent}(X,Y)$
- R5: $\text{ancestor}(X,Y) \leftarrow \text{parent}(X,Z), \text{ancestor}(Z,Y)$
- R6: $\text{wife}(X,Y) \leftarrow \text{husband}(Y,X)$
- R7: $\text{married}(X,Y) \leftarrow \text{husband}(X,Y)$
- R8: $\text{married}(X,Y) \leftarrow \text{wife}(X,Y)$
- R9: $\text{employed}(X) \leftarrow \text{occupation}(X, \text{service})$
- R10: $\text{student}(X) \leftarrow \text{occupation}(X, \text{student})$

R11: $\text{dependent}(X,Y) \leftarrow \text{parent}(Y,X), \text{employed}(Y), \text{student}(X)$
 R12: $\text{dependent}(X,Y) \leftarrow \text{married}(Y,X), \text{employed}(Y), \text{not employed}(X)$
 R13: $\text{self}(X) \leftarrow \text{married}(Y,X), \text{not employed}(Y)$
 R14: $\text{guardian}(X,Y) \leftarrow \text{dependent}(Y,X)$

INCONSISTENCY INDICATORS

II1: $\exists X \exists Y \text{ guardian}(X,Y), \text{not sponsor}(X,Y)$
 II2: $\exists X \exists Y \text{ married}(X,Y), \text{student}(X)$
 II3: $\exists X \exists Y \exists Z \text{ occupation}(X,Y), \text{occupation}(X,Z), \text{not } Y = Z$

FACTS

F1: $\text{occupation}(1,\text{service})$
 F2: $\text{occupation}(2,\text{service})$
 F3: $\text{occupation}(3,\text{student})$

 F4: $\text{father}(1,3)$

 F5: $\text{sponsor}(1,3)$

The following facts are deducible from these rules and facts. One can easily verify that the deductive database is consistent with respect to the specified inconsistency indicators II1, II2 and II3.

DEDUCIBLE FACTS

F6: $\text{employed}(1)$	[F1 + R9]
F7: $\text{employed}(2)$	[F2 + R9]
F8: $\text{student}(3)$	[F3 + R10]
F9: $\text{parent}(1,3)$	[F4 + R2]
 F10: $\text{dependent}(3,1)$	 [F6 + F10 + R11]
F11: $\text{ancestor}(1,3)$	[F9 + R4]
 F12: $\text{guardian}(1,3)$	 [F10 + R14]

Suppose the update to this database is $\text{husband}(1,2)$. We can find all induced updates with respect to this update in figure I (see the appendix). For instance, induced update F15: $\text{mother}(2,3)$ is derivable by applying rule R1 to the parent node, which is the update (F13), and the database fact F4: $\text{father}(1,3)$. So, F15: $\text{mother}(2,3)$ is directly induced by $\text{husband}(1,2)$. Now from F15 other induced updates can be derived in one or more steps, i.e., F18: $\text{parent}(2,3)$, F21: $\text{ancestor}(2,3)$, F22: $\text{dependent}(3,2)$ and F23: $\text{guardian}(2,3)$.

Proposition 3 Suppose D is consistent. Then D_U is consistent iff each induced instance of an inconsistency indicator is false in D_U .

Proof: The property follows from proposition 1 by reduction to the relational case. Consider the canonical interpretation of D as a relational database. A canonical interpretation consists of true atoms corresponding to the facts which are in the database or derivable from the database by its rules. A unique canonical interpretation can be determined by restricting the rules to be stratified in the sense of [APT87]. Treat the induced updates as explicit updates to this database. $\square$

To illustrate proposition 3, consider the previous example.

As we can see in figure I an update $\text{husband}(1,2)$ implies several induced updates. However, only few of them are important for the consistency of the database. Namely, only those which influence some inconsistency indicator. For instance $\text{F23:guardian}(2,3)$ is relevant to inconsistency indicator $I/1$. Note that we have to go through the whole branch before $\text{guardian}(2,3)$ is derived. So all of the earlier derived induced updates were needed while none of them are relevant to an inconsistency indicator. All induced updates which are relevant to an inconsistency indicator can be found in the grey boxes of figure I (see the appendix). So, for finding the induced instances of the inconsistency indicators a large number of redundant induced updates is generated. The relevant induced updates are $\text{F17:married}(1,2)$, $\text{F19:married}(2,1)$ resp. $\text{F23:guardian}(2,3)$. The corresponding induced instances of the inconsistency indicators are $\{\text{married}(1,2), \text{student}(1)\}$, $\{\text{married}(2,1), \text{student}(2)\}$ resp. $\{\text{guardian}(2,3), \text{not sponsor}(2,3)\}$.

Figure b Overview of integrity checking in the deductive case; the method based on induced updates.

Figure b visualizes proposition 3. The first step represented by the dotted arrows shows the determination of the induced updates by applying the update to the rules with the aid of the fact base. The second step represented by the continuous arrows shows:

- the determination of the instances of the inconsistency indicators relevant to at least one induced update, and
- the checking of the instances of the inconsistency indicators with the aid of the fact base.

As one can see, following proposition 3 has certain disadvantages: all induced updates are computed even those for which no inconsistency indicator is relevant. (For instance, in our example we generated eight induced updates where only three of them are relevant to an indicator). Bry, Decker and Manthey proposed another approach, which is based on generating (not necessarily ground) potential updates from the rules and the update without calling the fact base in the first place. The next section will describe this method.

4 METHOD BASED ON POTENTIAL UPDATES

When one compares the method based on induced updates with the method based on potential updates the only step in the first method that is postponed is the evaluation phase in which induced updates are determined. By postponing this phase we get potential updates instead of induced updates. Next we select only the potential updates which influence some inconsistency indicator. After this selection the instantiated inconsistency indicators are determined with respect to these potential updates. When we evaluate these instantiated indicators, we are in the postponed evaluation phase too. More formally if in definition 4 one does not apply substitution σ one finds the potential updates. The definition of potential updates is therefore a slight reformulation of definition 4:

Definition 7 Let $B \leftarrow A_1 \wedge A_2 \wedge \dots \wedge A_n$ be a deductive rule R . Suppose L is a positive (resp. $\neg L$ a negative) literal which is unifiable with A_i (resp. the complement of A_i) in R , for some i . Let γ be the most general unifier of L and A_i . Let $B' = (B\gamma)$. B' (resp. $\neg B'$) *directly depends* on L (with respect to R). A literal *depends on* L iff

1. it directly depends on L , or
2. it directly depends on a literal depending on L .

Definition 8 Let D be a deductive database and let U be an update. Each literal which depends on update U with respect to some rule of D is called a *potential update with respect to update U* (or simply *potential update* if it is clear from the context which update is involved).

Here the possible deducible facts are generated. This means that without calling the base facts we reason forward from the update by partially applying the rules in order to

get the potential updates. Note that all induced updates generated by the first method are substitutions of the potential updates generated by the second method.

The analogues of definition 6 and proposition 3 are now stated for potential updates instead of (ground) induced updates.

Definition 9 Let U be an update. Let L be an potential update with respect to U and II an inconsistency indicator relevant to L . The simplified instance of II with respect to L is now called a *potential instance* of II with respect to U .

Every potential instance of an inconsistency indicator with respect to an update U has to be false in the updated database.

Note that each induced update is an instance of some potential update. Therefore each instance of an inconsistency indicator is an instance of some potential instance of this indicator. For we instantiate the inconsistency indicators using relevant induced resp. potential updates. However, it is possible that some potential instance of an inconsistency indicator does not have some corresponding induced instance. This follows from the fact that some potential updates may not have an instance corresponding to an induced update.

With definition 9 the analogue of proposition 3 is:

Proposition 4 Suppose that none of the inconsistency indicators are satisfied in D . Then D_U is consistent iff each potential instance of an inconsistency indicator is false in D_U .

Proof: The property follows from proposition 3. For, firstly, all induced instances of inconsistency indicators are instances of some potential instances of inconsistency indicators. Secondly, all potential instances of indicators which are not related to an induced instance of an indicator are false in D_U because they were already false in the previous database state D . $\square$

To illustrate proposition 4, consider the previous example.

The potential instances of the inconsistency indicators are derived from the relevant potential updates which are present in the grey boxes of figure II (see the appendix). These relevant potential updates are:

PU3: married(1,2),
 PU5: married(2,1),
 PU12: guardian(1,2),
 PU13: guardian(2,Y) resp.
 PU14: guardian(2,1).

The corresponding potential instances of the inconsistency indicators are:

{married(1,2), student(1)},
 {married(2,1), student(2)},
 {guardian(1,2), not sponsor(1,2)},
 {guardian(2,Y), not sponsor(2,Y)} resp.
 {guardian(2,1), not sponsor(2,1)}.

Note that the potential instance $\{\text{guardian}(2,1), \text{not sponsor}(2,1)\}$ is an instantiation of $\{\text{guardian}(2,Y), \text{not sponsor}(2,Y)\}$. So the first one is redundant when we evaluate the latter one. But falsifying the first one can lead to an earlier rejection of the update than when we falsify the latter one. So, $\{\text{guardian}(2,1), \text{not sponsor}(2,1)\}$ may be redundant in a logical sense, it may not be redundant from a procedural point of view.

Note also that following proposition 4 has certain disadvantages: all potential updates are computed even those for which no inconsistency indicator is relevant. For instance, in our example we generated fourteen potential updates where only five of them are relevant to an indicator (the potential updates in the grey boxes) as we can see in figure II (see the appendix). The advantage of the method based on potential updates, when we compare it to the method based on induced updates, is that we do not spoil any evaluation time in order to find new database instances induced by the update.

Figure c *Overview of integrity checking in the deductive case; the method based on potential updates.*

Figure c visualizes proposition 4. The first step represented by the dotted arrows shows the determination of the potential updates by applying the update to the rules without the aid of the fact base. The second step represented by the continuous arrows reproduces:

- the determination of the instances of the inconsistency indicators relevant to at least one potential update, and
- the checking of the instances of the inconsistency indicators with the aid of the fact base.

Although in this example there are generated more potential updates than induced updates, this is not true in general. For instance, suppose we have one rule:

$R: \text{mother}(X,Y) \rightarrow \text{father}(Z,Y), \text{husband}(Z,X)$

and the facts:

$\text{father}(1,10), \text{father}(1,11), \dots, \text{father}(1,19).$

In case of an update $\text{husband}(1,2)$ we now have ten induced updates, i.e.,

$\text{mother}(2,10), \text{mother}(2,11), \dots, \text{mother}(2,19)$

and only one potential update

$\text{mother}(2,Y).$

As we have seen inconsistency indicator falsification in deductive databases according to proposition 4 has a drawback, several redundant potential updates were generated. Now another approach is proposed which is based on generating directly only the relevant potential updates from the rules, inconsistency indicators and the update without calling the fact base. In the next section this method will be described.

5 PROPOSED METHOD BASED ON INCONSISTENCY RULES

Now a new method will be presented, which represents a new class of methods. The method proposed here, goes from the update to the relevant instances of the inconsistency indicators in just one step. So, it does not have the disadvantage of generating induced updates or potential updates which are not relevant to any of the inconsistency indicators. 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 so called *potential update trees*, which are constructed as follows.

Definition 10 Let L be a literal occurring in an inconsistency indicator. Literal L is the root of a *potential update tree*, say T_L . We call L the *root literal* of T_L . The construction of T_L is a top-down construction which proceeds as follows:

- if L is not unifiable with the head of any rule, then the child node of L is L itself.
- Let literal N be a node in T_L . Let $R : H \leftarrow B_1 \wedge \dots \wedge B_m$ be a rule, where H is a positive literal which is unifiable with N and where $B_1, \dots, B_m$ are literals. Let σ be the most general unifier of N and H , then $B_1\sigma, \dots, B_m\sigma$ are child nodes of N .

This algorithm stops if no rule can be applied anymore or if a branch results which is redundant. A branch is redundant:

- if its top node is syntactically the same as some other node in the tree, or

- if its top node only differs in some variables that do not occur in the root literal from some other node in the tree that do not occur in the root literal.

This last condition allows the application of recursive rules without getting infinite branches of a potential update tree. So, because we have only finite rules with a finite amount of arguments the potential update tree is finite.

Example 3 Suppose $a(X, Y)$ is the root literal of some potential update tree $T_{a(X, Y)}$. Let $R : a(X, Y) \leftarrow b(X, Z), a(Z, Y)$ be the only rule. The left branch of $T_{a(X, Y)}$ consists of $b(X, Z)$ and the right branch of $T_{a(X, Y)}$ consist of a branch with top literal $a(Z, Y)$. Now by applying R to $a(Z, Y)$ we only derive one subnode $b(Z, Z')$. A subnode $a(Z', Y)$ differs only in the first argument from $a(Z, Y)$, but these arguments are variables that do not occur in the root literal. So, $a(Z', Y)$ is redundant. (redundant in the sense that there is no difference in instantiating root literal $a(X, Y)$ if we choose either $a(Z, Y)$ or $a(Z', Y)$; in both cases an update only binds variable Y)

We have constructed the potential update trees for the example mentioned above. (see figure III). From each literal in each inconsistency indicator a tree is created. For instance, $\text{guardian}(X, Y)$ which occurs in the first inconsistency indicator is the top of a tree. Its child node is the literal occurring in the body of the rule R14, i.e., $\text{dependent}(Y, X)$. The child nodes of the node $\text{dependent}(Y, X)$ are the literals in the body of rule R11, i.e., $\text{parent}(X, Y)$, $\text{employed}(X)$ and $\text{student}(Y)$ and the literals in the body of rule R12, i.e., $\text{married}(Y, X)$, $\text{employed}(X)$ and $\text{not employed}(Y)$. Note that we have to switch the variables in these literals because the literal in the parent node is also switched compared to the head of the rules. Also note that $\text{employed}(X)$ occurs two times as a child node, so one of them is redundant and can be left off. We can continue this algorithm until none of the rules can be applied anymore. When this has been done for each literal in each inconsistency indicator, we get all the potential update trees. When we have the transaction $\{\text{husband}(1, 2), \text{sponsor}(2, 3)\}$, we unify each husband-literal and each sponsor-literal occurring as nodes in the tree with the updates. (see figure IV). Note that there are no sponsor-literals in the trees, only “not sponsor”-literals. Now unify the update $\text{husband}(1, 2)$ with each unifiable literal in the tree. The resulting substitutions are given to the top node, i.e., a literal of the inconsistency indicator. Note that the instantiated top node literals correspond to all the relevant potential updates generated by the previous method, i.e., PU6: $\text{guardian}(1, 2)$, PU13: $\text{guardian}(2, Y)$, PU14: $\text{guardian}(2, 1)$, PU3: $\text{married}(1, 2)$ and PU5: $\text{married}(2, 1)$.

From the potential update trees we construct the so called *inconsistency trees*. Suppose there is a potential update tree with L as the top node from which an inconsistency tree is derived by replacing L by an inconsistency indicator containing literal L .

Definition 11 Let II be an inconsistency indicator. Then T_{II} is called an *inconsistency tree* of II , if the following conditions are satisfied:

- the root of T_{II} is II ,
- a tree T is a subtree of T_{II} iff T is a potential update tree T_L for some literal L in II .

The inconsistency trees for our example can be found in the APPENDIX (see figure V). Note that from each potential update tree we can construct at least one inconsistency tree because the top level literal of the potential update tree is a literal of an inconsistency indicator by definition. Because L may be a literal which occurs in more than one inconsistency indicator several inconsistency trees can be derived from the potential update tree of which L is the root literal.

What we can see in our example is that essentially only the root and the nodes which are unifiable with the update are relevant for deriving the relevant instances of the inconsistency indicators. All nodes “between” the root and these nodes are not necessary to get the root instantiated. This can be done in just one step. For instance in our example the update $\text{husband}(1,2)$ leads to an instance $\{\text{guardian}(2,Y), \text{not sponsor}(2)\}$ by applying a substitution $\{X = 2, Z = 1\}$ to $\text{husband}(Z,X)$. For this reason we define the one-level inconsistency trees. They are defined using the definition of inconsistency trees.

Definition 12 Let II be an inconsistency indicator. Then the root of a *one-level inconsistency tree* T_{II}^1 is a root of an inconsistency tree T_{II} . N is a child node of the root (i.e., II) of T_{II}^1 if it is a node of T_{II} . From N no other nodes are derived.

Now an instantiation of a node implied by some update leads to an instantiation of the inconsistency indicator directly. In our example the transaction $\{\text{husband}(1,2), \text{sponsor}(2,3)\}$ leads directly to all relevant instances of the inconsistency indicators, i.e., $\{\text{guardian}(1,2), \text{not sponsor}(1,2)\}$, $\{\text{guardian}(2,Y), \text{not sponsor}(2,Y)\}$, $\{\text{guardian}(2,1), \text{not sponsor}(2,1)\}$, $\{\text{married}(1,2), \text{student}(1)\}$ and $\{\text{married}(2,1), \text{student}(2)\}$.

In 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. In case of a base relation this is not true. Updates in derived relations are in most cases not allowed. Although in our example updating derived relations is not forbidden, the one-level inconsistency tree is drawn for only the base relations, i.e., husband, father, occupation, sponsor. (See figure VI on pp.34).

Note that the (one-level) inconsistency trees can be used over and over again as long as the rules and constraints do not change. The next definition shows how the inconsistency rules, which are asserted to the database, are derived from the one-level inconsistency trees.

Definition 13 Let II be an inconsistency indicator.

We call $\text{inconsistent}(A) \leftarrow II$ an *inconsistency rule* if A is a leaf node of the one-level inconsistency tree T_{II}^1 .

Now all derivable inconsistency rules for each inconsistency indicator are asserted to the database. With these rules we do not need the inconsistency indicators to conclude if an update leads to an inconsistent database state.

Definition 14 Let U be an update and let $R: \text{inconsistent}(A) \leftarrow II$ be an inconsistency rule where A is a literal unifiable with U . Let σ be a most general unifier of A 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 inconsistency rules. If it is true then the update is rejected. If it is not, the updated database is consistent. With definition 14 the analogue of proposition 4 is:

Proposition 5 Let U be an update. Suppose D is consistent. Then D_U is consistent iff each relevant instance of an inconsistency indicator is false in D_U .

Proof: First note that the relevant instances of II with respect to U correspond to the potential instances of II with respect to U . For let $II\sigma$ be a relevant instance of II , then there is a inconsistency rule $R : inconsistent(A) \leftarrow II$ where A is unifiable with U with a most general unifier σ and A is a node of the inconsistency tree T_{II} according to definition 13. Then according to definition 11 A is also a node of a potential update tree. Because U is unifiable with A with the most general unifier σ , one gets the potential instance $II\sigma$. In the same manner we can find for each potential instance of II a corresponding relevant instance of II . Secondly note that knowing that there is no difference in the set of relevant instances and potential instances we have proved this proposition by using proposition 4. $\square$

Figure d Overview of integrity checking in the deductive case; the proposed method.

Proposition 5 is illustrated by figure d. Now it is clear that conceptually the inconsistency indicator verification in deductive databases is almost as easy to describe as in relational databases.

The main advantage of these rules is that with some vast number of rules and inconsistency indicators, they only have to be generated once, and can be used over and over again for updates of facts. I have also implemented a generator for inconsistency rules, which

generates from the set of rules and indicators the (“optimized”) inconsistency rules. For each example mentioned in this paper the optimized inconsistency rules can be generated in a mere few seconds from the set of rules and inconsistency indicators.

The concept of inconsistency rules can also be implemented in PROLOG easily. In the next section the inconsistency rules are given for the main example in this paper.

6 IMPLEMENTATION OF INCONSISTENCY RULES

Consider the main example of this paper. We first look how the inconsistency rules are implemented and optimized, before we see in the next section some test results.

The main example contains three inconsistency indicators:

- II1: $\exists X \exists Y$ guardian(X,Y), not sponsor(X,Y)
- II2: $\exists X \exists Y$ married(X,Y), student(X)
- II3: $\exists X \exists Y \exists Z$ occupation(X,Y), occupation(X,Z), not $Y = Z$

For each inconsistency indicator we derive a collection of inconsistency rules. We implemented the inconsistency rules for this example for insertions only. Implementing inconsistency rules for deletions goes in a similar way. For instance, deletion of a sponsor-fact influences the inconsistency indicator guardian(X,Y), not sponsor(X,Y) directly. The inconsistency rule related to this kind of deletions could be

```
inconsistent(del,sponsor(X,Y)) :-
    guardian(X,Y),
    not sponsor(X,Y).
```

assuming that our consistency checking program can handle such rules. The inconsistency rules for insertions can be implemented in PROLOG as follows:

The inconsistency rules derived from II1:

```
inconsistent(father(X,Y)) :-
    guardian(X,Y),
    not sponsor(X,Y).
inconsistent(father(Z,Y)) :-
    guardian(X,Y),
    not sponsor(X,Y).
inconsistent(husband(Z,X)) :-
    guardian(X,Y),
    not sponsor(X,Y).
inconsistent(occupation(X,service)) :-
    guardian(X,Y),
    not sponsor(X,Y).
inconsistent(occupation(Y,student)) :-
    guardian(X,Y),
    not sponsor(X,Y).
```

```

inconsistent(husband(X,Y)) :-
    guardian(X,Y),
    not sponsor(X,Y).
inconsistent(husband(Y,X)) :-
    guardian(X,Y),
    not sponsor(X,Y).

```

The inconsistency rules derived from *II2*:

```

inconsistent(husband(X,Y)) :-
    married(X,Y),
    student(X).
inconsistent(husband(Y,X)) :-
    married(X,Y),
    student(X).
inconsistent(occupation(X,student)) :-
    married(X,Y),
    student(X).

```

And the inconsistency rule derived from *II3*:

```

inconsistent(occupation(X,Y)) :-
    occupation(X,Y),
    occupation(X,Z),
    not Y = Z.

```

Note that in the last case in fact we have generated another inconsistency rule from the second literal in *II3*, namely

```

inconsistent(occupation(X,Z)) :-
    occupation(X,Y),
    occupation(X,Z),
    not Y = Z.

```

But the latter is in fact the same as the first. We get the first inconsistency rule by switching the variables *Y* and *Z* in the second rule. Another issue concerning the derived rule from *II3* is that it is constructed with respect to inconsistency indicator *II3* in which base relations occurs, i.e., *occupation*. This means that an update in the relation “*occupation*” directly influences this inconsistency indicator. So, if we applied the rule for an update *occupation(a,b)*, this would lead to evaluation of *occupation(a,b)*. But this is redundant because we already know that *occupation(a,b)* is asserted to the database, because it was the update itself. So, we can optimize this rule to:

```

inconsistent(occupation(X,Y)) :-
    occupation(X,Z),
    not Y = Z.

```

Also several other optimizations are possible to reduce the time to check the update. First, we can change the order of the subgoals in the body of a clause in order to speed up the evaluation time of an inconsistent-goal. One can put a subgoal in front if it leads to an early failure of the goal. Here, the following considerations were important to decide if a subgoal must be carried out before the other:

- subgoals which are not (partially) instantiated by the update are placed at the end of the clause,
(in the worst case we have an inconsistency rule relevant to the update for which the first subgoal in that rule, say G , has no variables in common with the update literal in the head of the rule. The update leads to a search through the whole relation of G . If there is also a subgoal G' in this clause which is instantiated partially and also has one or more variables in common with G , then subgoal G' must be evaluated before G . For evaluation of G' will instantiate G before G is evaluated, so therefore a full search through the extension of the relation is prevented).
- subgoals involving base predicates are placed before subgoals involving derived predicates,
- subgoals leading to a few facts should be placed before subgoals that lead to a great number of facts,
- negated subgoals will only be evaluated first if they are fully instantiated by the update.

For example, in our inconsistency rules we could change:

```
inconsistent(father(X,Y)) :-
    guardian(X,Y),
    not sponsor(X,Y).
```

in

```
inconsistent(father(X,Y)) :-
    not sponsor(X,Y),
    guardian(X,Y).
```

and

```
inconsistent(husband(X,Y)) :-
    married(X,Y),
    student(X).
```

in

```
inconsistent(husband(X,Y)) :-
    student(X),
    married(X,Y).
```

After the needed optimizations we get the following inconsistency rules which now are ordered by the relations of the update:

INCONSISTENCY RULES

```
inconsistent(father(X,Y)) :-
    not sponsor(X,Y),
    guardian(X,Y).
inconsistent(father(Z,Y)) :-
    guardian(X,Y),
    not sponsor(X,Y).

inconsistent(occupation(X,Y)) :-
    occupation(X,Z),
    not Y = Z.
inconsistent(occupation(X,student)) :-
    married(X,Y).
inconsistent(occupation(X,service)) :-
    guardian(X,Y),
    not sponsor(X,Y).
inconsistent(occupation(Y,student)) :-
    guardian(X,Y),
    not sponsor(X,Y).

inconsistent(husband(Y,X)) :-
    not sponsor(X,Y),
    guardian(X,Y).
inconsistent(husband(X,Y)) :-
    not sponsor(X,Y),
    guardian(X,Y).
inconsistent(husband(X,Y)) :-
    student(X),
    married(X,Y).
inconsistent(husband(Y,X)) :-
    student(X),
    married(X,Y).
inconsistent(husband(Z,X)) :-
    guardian(X,Y),
    not sponsor(X,Y).
```

These inconsistency rules are asserted to the database. The next section will show that the method based on inconsistency rules is potentially far more efficient than all methods based on induced updates or potential updates.

7 COMPARISON WITH EXISTING METHODS

Some tables below show the results of Das and Williams, who implemented and tested several integrity constraint checking methods. (See [DAW 89]). One of these methods is called the *simple method*. This method checks all inconsistency indicators using all possible facts in the database to check the consistency of the updated database. So, this method does not take advantage of the fact that the database was consistent before the update. It is very easy to implement this method in PROLOG, as follows:

IMPLEMENTATION 1 of the SIMPLE method:

```
inconsistent :-
    (i1(X,Y) ; i2(X1,Y1) ; i3(X2,Y2,Z)).

i1(X,Y) :-
    guardian(X,Y),
    not sponsor(X,Y).

i2(X1,Y1) :-
    married(X1,Y1),
    student(X1).

i3(X2,Y2,Z) :-
    occupation(X2,Y2),
    occupation(X2,Z),
    not Z = Y2.
```

Or as short as possible:

IMPLEMENTATION 2 of the SIMPLE method:

```
inconsistent :-
    (guardian(X,Y), not sponsor(X,Y)) ;
    (married(X1,Y1), student(X1)) ;
    (occupation(X2,Y2), occupation(X2,Z), not Z = Y2).
```

The timing results of these different implementations of the simple method do not differ significantly.

In the remaining part of the paper, we have used the first implementation of the simple method. A second matter which could have influenced the test results of Das and Williams is the way they filled their database with facts. In this example they filled a database with 3238 facts. But what was the density of the facts in their databases? In other words, are there many deducible facts (high density) or not (low density). In order to check the influence of this phenomenon, two databases were created. One consisting of 3238 facts involving only ($\sigma =$) 3500 different people and one consisting of 3238 facts

involving ($\sigma =$) 10000 different people. Note that the first database has a higher density than the second. Although it can make a difference which implementation is used, the table shows that within a certain range the results of my implementation of the simple method and that of Das and Williams' are comparable, although they may have used a different PROLOG and a different computer.

Method	Number of Facts		
	1095	2180	3238
Simple (with $\sigma = 3500$)	59.9	241.1	551.7
Simple (with $\sigma = 10000$)	59.0	236.7	526.2

Table 1: My timings of the main example.

Because the timings for the simple method are comparable, we are confident that the results for the proposed method are comparable with the results of the methods tested by Das and Williams, at least with regard to orders of magnitude. The method proposed by Lloyd et al. is based on the method of potential updates. The method proposed by Decker is based on the method of induced updates. Kowalski et al. build a meta-interpreter to reason forward and backward through the rules to search for the instantiated constraint which is evaluated directly. So, we can see this method as a mixture of the methods based on induced resp. potential updates. Das and Williams propose also a mixture of both methods. They generate all positive induced updates and all negative potential updates from the update.

In the next two tables we find resp. the results of Das and Williams and the results of the proposed method based on inconsistency rules

Method	Time
Simple	336.1
Lloyd et al.	10.0
Decker	31.5
Kowalski et al.	133.7
Das and Williams	18.4

Table 2: The timings of Das & Williams of the example from [DAW 89].

The reason why the proposed method is far more efficient than the other presented methods is that in the proposed method all possible violations of updates are represented in the inconsistency rules, which become a part of the deductive database. So, the effort to generate update constraints is already made. This leads to a considerable check time reduction. In the next subsections the performances of the induced, potential and proposed method when using several typical examples is shown.

Method	Time (AT)
Simple	553.4
Proposed Method	1.4

Table 3: The timings of my implementations of the main example using the Simple and the Proposed Method.

7.1 Example A: Potential but no Induced Updates

The next example contains the rule set of the previous example. But instead of the fact base and inconsistency indicators of the main example we have:

a list of 1085 facts not involving constants 1,2:

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

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

INCONSISTENCY INDICATORS

II1: $\exists X \exists Y$ guardian(X,Y), not sponsor(X,Y),
II2: $\exists X \exists Y \exists Z$ sponsor(X,Y), guardian(Z,Y), not parent(X,Y).

update: married(1,2)

In this example no induced update can be generated from the update. So, there is no instantiated inconsistency indicator generated in the method based on induced updates. Therefore the update is accepted. In case of the method based on potential updates, several potential updates are generated. Even instantiated inconsistency indicators are generated and evaluated which is redundant. In the next tabel we can therefore see that the method of potential updates by Lloyd et al. does not perform well here. The second table shows the results of this example for the simple method and for the proposed method.

7.2 Example B: Few Potential Updates, many Induced Updates

The next example contains the rule set of the first example. But instead of the fact base and inconsistency indicators of the first example we have:

Method	Time
Lloyd et al.	3.4
Decker	0.8
Kowalski et al.	0.4
Das and Williams	0.2

Table 4: Timings of Das & williams of example A from [DAW 89].

Method	Time (AT)
Simple	6.6
Proposed Method	0.1

Table 5: My timings of my implementations of the Simple and Proposed method of example A.

a list of 1058 facts not involving constants 1,3:

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

a list of ten facts under the predicate father with 1 as first argument,

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

INCONSISTENCY INDICATOR

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

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, for there exists no inconsistency rule for which the update is relevant. But what happens when we generate all induced updates? 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. Note that this transaction causes many induced mother-updates that are irrelevant with respect to the inconsistency indicators, while only one potential mother-update is generated. In case of the potential updates things are a little bit more under control. Each rule only produces at most one potential update. That is why the method of Lloyd et al. performs well compared to the other three methods which are more or less based on induced updates. But all of the generated potential updates are redundant. The next two tables show the results of the tests for this example. First husband(1,2) was added to the database (see column add), afterwards the husband(1,2)-fact was deleted again in the updated database (see column del).

Method	Operation	
	<i>add</i>	<i>del</i>
Lloyd et al.	0.7	0.7
Decker	62.9	63.0
Kowalski et al.	$\gg 100.0$	$\gg 100.0$
Das and Williams	37.4	0.3

Table 6: Timings of Das & Williams of example B from [DAW 89].

Method	Operation	
	<i>add</i>	<i>del</i>
Simple	3.0	3.0
Proposed Method	0.05	<0.05

Table 7: My timings of my implementation of the Simple and the Proposed Method of example B.

7.3 Example C: A More Complex Database

Now as a final example, we generated a deductive database with a different rule set and more constraints. This example can also be found in [DAW 89]. This database was tested with three different fact bases, so we can see how the results of the different methods evolve when the amount of base facts increases. We have

RULES

- R1: $\text{parent}(X,Y) \leftarrow \text{father}(X,Y)$
R2: $\text{parent}(X,Y) \leftarrow \text{mother}(X,Y)$

R3: $\text{mother}(X,Y) \leftarrow \text{father}(Z,Y), \text{husband}(Z,X)$
 R4: $\text{age}(X,Y) \leftarrow \text{civil_status}(X,Y,P,Q)$
 R5: $\text{sex}(X,Y) \leftarrow \text{civil_status}(X,P,Y,Q)$
 R6: $\text{dependent}(X,Y) \leftarrow \text{parent}(Y,X), \text{occupation}(Y,\text{service}), \text{occupation}(X,\text{student})$
 R7: $\text{occupation}(X,Y) \leftarrow \text{civil_status}(X,P,Q,Y)$

INCONSISTENCY INDICATORS

II1: $\exists X \exists Y_1 \exists Y_2 \exists Z_1 \exists Z_2 \exists T_1 \exists T_2 \text{civil_status}(X,Y_1,Z_1,T_1),$
 $\text{civil_status}(X,Y_2,Z_2,T_2), \text{not } (Y_1 = Y_2, Z_1 = Z_2, T_1 = T_2)$
 II2: $\exists X_1 \exists X_2 \exists Y \text{father}(X_1,Y), \text{father}(X_2,Y), \text{not } X_1 = X_2$
 II3: $\exists X \exists Y_1 \exists Y_2 \text{husband}(X,Y_1), \text{husband}(X,Y_2), \text{not } Y_1 = Y_2$
 II4: $\exists X_1 \exists X_2 \exists Y \text{husband}(X_1,Y), \text{husband}(X_2,Y), \text{not } X_1 = X_2$
 II5: $\exists X \exists Y \exists Z \exists T \text{civil_status}(X,Y,Z,T), \text{not } (X > 0, X < 100000, Y > 0, Y < 125,$
 $Z \in \{\text{male}, \text{female}\}, T \in \{\text{student}, \text{retired}, \text{business}, \text{service}\})$
 II6: $\exists X \exists Y \exists Z \text{civil_status}(X,Y,Z,\text{student}), \text{not } Y < 25$
 II7: $\exists X \exists Y \exists Z \text{civil_status}(X,Y,Z,\text{retired}), \text{not } Y > 60$
 II8: $\exists X \exists Y \text{father}(X,Y), \text{not } (\text{sex}(X,\text{male}), \text{sex}(Y,\text{male}))$
 II9: $\exists X \exists Y \text{husband}(X,Y), \text{not } (\text{sex}(X,\text{male}), \text{sex}(Y,\text{female}))$
 II10: $\exists X \exists Y \exists P \exists Q \text{husband}(X,Y), \text{age}(X,P), \text{age}(Y,Q), (\text{not } P \geq 20, Q \geq 20)$
 II11: $\exists X \exists Y \exists Z \exists T \text{civil_status}(X,Y,Z,T), Y < 20, \text{not } T = \text{student}$
 II12: $\exists X \exists Y \text{dependent}(X,Y), \text{not } \text{tax}(Y,X)$

FACTS

The following facts not involving the constants 1,2,3:

600 (1800 in state 2, 3020 in state 3) $\text{civil_status-facts}$,
 169 (518 in state 2, 860 in state 3) father-facts ,
 220 (669 in state 2, 1120 in state 3) husband-facts ,
 21 (75 in state 2, 119 in state 3) tax-facts .

We have the following transaction:

TRANSACTION

$\text{civil_status}(1,50,\text{male},\text{service}),$
 $\text{civil_status}(2,45,\text{female},\text{business}),$
 $\text{civil_status}(3,19,\text{male},\text{student}),$
 $\text{husband}(1,2),$
 $\text{father}(1,3),$
 $\text{tax}(1,3).$

As we can see in the next two tables the advantage of the proposed method not only increases with the number of constraints but also with the number of facts.

Method	Number of Facts		
	1010	3062	5119
Simple	37.0	311.5	860.6
Lloyd et al.	2.9	11.7	28.3
Decker	16.6	54.0	99.4
Kowalski et al.	14.2	58.6	129.1
Das and Williams	2.5	11.9	29.0

Table 8: Timings of Das & Williams of example C from [DAW 89].

Method	Number of Facts		
	1010	3062	5119
Simple	74.8	657.6	1764.5
Proposed Method	1.2	2.6	5.0

Table 9: My timings of the Simple and the Proposed Method of example C.

8 CONCLUSIONS

As we have seen the proposed method is:

- efficient,
- easy to implement in PROLOG, (even a meta-interpreter is not needed), and
- conceptually as clear as the relational case. (Compare figure a and figure d).

Another advantage of the proposed method is that the inconsistency rules can be used over and over again as long as the set of rules and the set of inconsistency indicators remains unchanged. Also the inconsistency rules can be optimized in compile time.

Also an advantage of the proposed method is the reusability of the constructed inconsistency rules in case of a rule update or an inconsistency indicator update, which was not described in this paper. Only the inconsistency rules generated from these updates are needed to update the set of inconsistency rules. So all other inconsistency rules remain unchanged.

All these advantages makes this new method very promissing, although yet a lot of work has to be done.

9 FUTURE DIRECTIONS

Several issues for further investigation are:

- the handling of recursive predicates in inconsistency indicators,
- allowing a more general set of inconsistency indicators, for instance indicators with universal quantors,
- allowing more general updates like rule updates,
- the growth of the number of generated inconsistency rules when increasing the number of rules and/or inconsistency indicators, i.e., space complexity. Also the space complexity of the other methods have to be compared with these findings.
- the timing complexity when varying the number of facts, rules and/or inconsistency indicators or the density of the database facts compared to the other methods.

REFERENCES

- [BDM 88] Bry, François, Hendrik Decker and Rainer Manthey,
"A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases", in [SCM88], pp. 488-505,
also: ECRC Technical Report KB-16, 7 Nov. 1987.
- [DAW 89] Das, S.K. and M.H. Williams
"Integrity Checking Methods in Deductive Databases: A Comparative Evaluation",
in [WIL 89], pp. 85-116.
- [DEC 87] Decker, H.
"Integrity Enforcement on Deductive Databases",
in [KER 87], pp. 271-285.
- [DWI 89] Das, K. and M. Howard Williams
"A path finding method for constraint checking in deductive databases",
in Data & Knowledge Engineering 4, pp. 223-244,
1989.
- [LLT 85] Lloyd, J.W. and R.W. Topor
"A Basis for Deductive Database Systems",
J. Logic Programming 2, pp. 93-109,
1985.
- [LLT 86] Lloyd, J.W. and R.W. Topor
"A Basis for Deductive Database Systems II",
J. Logic Programming 3 (1), pp. 55-67,
1986.
- [LST 87] Lloyd, J.W., E.A. Sonenberg and R.W. Topor
"Integrity Constraint Checking in Stratified Databases",
J. Logic Programming 4, pp. 331-343,
1987.
- [LLO 87] Lloyd, J.W.
"Foundations of Logic Programming",
2nd, Extended Edition,
Springer Verlag, 1987.
- [NIC 82] Nicolas, J.M.
"Logic for Improving Integrity Checking in Relational Databases".
Acta Informatica 18 (3), pp. 227-253,
1982.
- [NIY 78] Nicolas, J.M. and K. Yazdanian
"Integrity Checking in Deductive Data Bases",

in "Logic and Databases", [GAM78], pp. 325-346,
1978.

- [SAD 88] Sadri, Fariba, and Robert Kowalski
"A Theorem-Proving Approach to Database Integrity",
in [MIN 88], Chapter 9.

PROCEEDINGS

- [GAM 78] Gallaire, H. and J. Minker
"Logic and Databases",
Plenum Press,
New York 1978.
- [KER 87] Kerschberg, Larry (Ed.)
"Expert Database Systems: Proceedings from the First International Conference",
Charleston, Sc., 1987.
- [MIN 88] Minker, J. (Ed.)
"Foundations of Deductive Databases and Logic Programming",
Morgan Kaufmann,
Los Altos, 1988.
- [SCM 88] Schmidt, J.W., S. Ceri and M. Missikoff (Eds.)
"Advances in Databases Technology, EDBT '88",
International Conference on Extending Database Technology, Proceedings,
Venice, Italy, 14-18 March 1988,
Lecture Notes in Computer Science, 303.
- [WIL 89] Williams, M.H. (Ed.)
"Proceedings of the Seventh British National Conference on Databases",
Cambridge University Press,
1989.

10 APPENDIX

Figure I : the induced updates

Figure II : the potential updates

Figure III : the potential update tree

Figure IV : instantiation of the potential update tree

Figure V : the inconsistency trees

Figure VI : the one level inconsistency trees

In this series appeared:

91/01 D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02 R.P. Nederpelt
H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03 J.P. Katoen
L.A.M. Schoenmakers	Parallel Programs for the Recognition of P -invariant Segments, p. 16.
91/04 E. v.d. Sluis
A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05 D. de Reus	An Implementation Model for GOOD, p. 18.
91/06 K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07 E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08 H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09 W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10 R.C.Backhouse
P.J. de Bruin
P. Hoogendijk
G. Malcolm
E. Voermans
J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11 R.C. Backhouse
P.J. de Bruin
G.Malcolm
E.Voermans
J. van der Woude	Relational Catamorphism, p. 31.
91/12 E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13 F. Rietman	A note on Extensionality, p. 21.
91/14 P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15 A.T.M. Aerts
K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16 A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17	A.T.M. Acrts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic process creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.

91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.

92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottom-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for F ω , p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoelen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Bacten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moclrand	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, 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 Distributed 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 circuits, 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 taxonomy 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. Kornatzky 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 Markov Decision Processes 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. Klops D. Kratsch H. Müller	Dominoes, p. 14.