INTEGRITY CHECKING IN DEDUCTIVE DATABASES

Robert Kowalski Fariba Sadri Paul Soper

Department of Computing
Imperial College of Science and Technology
180 Queen's Gate
London SW7 2BZ

Abstract

We describe the theory and implementation of a general theorem-proving technique for checking integrity of deductive databases recently proposed by Sadri and Kowalski. The method uses an extension of the SLDNF proof procedure and achieves the effect of the simplification algorithms of Nicolas, Lloyd, Topor et al, and Decker by reasoning forwards from the update and thus focusing on the relevant parts of the database and the relevant constraints.

Formalisation of the procedure using logic as meta-language forms the basis of our implementation in Prolog. It is further shown that in the absence of implicit deletions a transformation of the database clauses and constraints allows the method to be implemented with efficiencies comparable to implementations of Prolog.

(1) Introduction

This paper continues the development of the *Consistency Method* for checking integrity in deductive databases recently proposed by Sadri and Kowalski [11]. The method is designed to exploit the assumption that the database satisfies its constraints before the transaction, and thus any violation afterwards must involve at least one of the updates in the transaction. It does this by using a proof procedure that allows reasoning forwards from the updates, which has the effect of focusing attention only on the parts of the database and the constraints that are affected by the updates.

This approach contrasts with Prolog-based implementations of deductive databases which rely on the purely backward reasoning strategy of SLDNF [6]. Backward reasoning is well-suited for evaluating queries to a fixed database but not for checking the integrity of a changing database. The simple approach of evaluating all the integrity constraints as queries after each transaction can suffer from unacceptable inefficiencies, because it fails to exploit the assumption that the database satisfies its constraints prior to the transaction, and may redundantly recheck constraints which are unaffected by the transaction.

Methods for avoiding this redundancy have been proposed by Lloyd, Topor et al (called L/T from now on) [7, 8, 13] and Decker [3]. Their simplification algorithms extend Nicolas' algorithm for relational databases [10] and consist primarily of two stages. The first stage derives from the transaction a simplified set of constraints, which is possibly smaller and more highly instantiated than the original set, and whose satisfaction ensures the validity of the updated database. The second stage evaluates the simplified constraints using the SLDNF proof procedure. The Consistency Method achieves the effect of such simplification algorithms while remaining within a uniform theorem-proving framework.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

The proof procedure adopted for this method extends SLDNF by

- o allowing forward reasoning as well as backward reasoning,
- o incorporating additional inference rules for reasoning about implicit deletions caused by changes to the database, and
- o incorporating a generalised resolution step, which is needed for reasoning forwards from negation as failure.

The procedure is identical to SLDNF whenever the top clause is a denial. When the top clause comes from the transaction, the proof procedure can approximate the simplification algorithms of L/T and Decker, by employing different literal selection and search strategies. These strategies are discussed in detail in [11].

Our implementation in Prolog is based on a meta-level formalisation of the new proof procedure in Horn clause logic augmented by negation as failure. The formalisation functions as a meta-interpreter and is consequently less efficient than an implementation that runs directly in Prolog. We show that in special cases a simple transformation of the database and the integrity constraints allows us to dispose of the meta-interpreter and to run the integrity checking method directly in Prolog.

The paper is organised as follows. The Consistency Method is described informally in Section 2. Section 3 presents a formalisation of the method, which forms the basis of our implementation described in Section 4. Finally, in Section 5 we propose a more efficient implementation for a special case.

(2) The Consistency Method

The Consistency Method is a method for checking integrity in range-restricted deductive databases.

A *deductive database* D is a finite set of deductive rules of the form

$$A \text{ <-- } L_1 \text{ and } \ldots \text{ and } L_n, \qquad n \geq 0,$$

where the *conclusion* A is an atomic formula (atom), each *condition* L_i is a literal, i.e. an atom (a positive condition) or a negated atom (a negative condition), and all variables are assumed to be universally quantified over the whole of the rule. When n=0 the deductive rule is also called a *fact*. When n > 0 the rule is said to be *non-atomic*.

D is *range-restricted* if for each rule c every variable of c occurs in a positive condition of c. This is a standard restriction to ensure that negative conditions can be fully instantiated before evaluation. This restriction corresponds exactly to Decker's "range-restriction" [2] and Lloyd and Topor's "allowed" [9] conditions.

Integrity constraints are closed first-order formulae that D is required to satisfy. The set of constraints I specified for D is assumed to be consistent. The Consistency Method deals directly with constraints that have the form of a *denial*

$$\text{<-- } L_1 \text{ and } \ldots \text{ and } L_n$$

where the L_i are literals and variables are assumed to be universally quantified over the whole formula. More general constraints (including those with conclusions having existentially quantified variables) can be transformed into this form as described in [11]. From now on in this paper, without loss of generality, we assume that integrity constraints are in the form of denials.

The Consistency Method appeals to the *consistency view of constraint satisfaction*, according to which D satisfies I if and only if the set Comp(D)$\cup$I is consistent. $\cup$ denotes the set union operation. Comp(D), the completion of the database, is essentially D together with the only-if version of the rules in D and an appropriate equality theory [1,6]. The relationship between this view of constraint satisfaction and the more usual view which insists that the constraints be theorems of Comp(D) is discussed in [11].

By taking the integrity constraints in I as top clauses the

SLDNF proof procedure can be used to check if Comp(D)∪I is consistent. This, however, fails to exploit the assumption that D satisfies its constraints prior to the transaction. The Consistency Method takes advantage of this assumption by taking each update as a top clause in a proof procedure which extends SLDNF.

Suppose a transaction T is performed on D and I to give an updated database DT and an updated set of constraints IT. The Consistency Method takes DT∪IT as input set and each update in T as a candidate top clause. If an update is the insertion of a rule or constraint, the update stands as top clause as it is. If it is the deletion of a fact A the top clause associated with it is NOT(A), provided that A is not implicit (i.e. derivable) in DT. If A is implicit in DT the update does not alter the logical content of the database. (We assume that an update of deleting a fact only deletes the explicit occurrence of that fact). Updates that are deletions of constraints cannot cause inconsistency and are not considered as top clauses. Finally, if an update is the deletion of a non-atomic rule then, following Decker, we determine which instances of the conclusion of the rule are deleted as a result of deleting the rule. The negations of these deleted atoms are then candidate top clauses.

In SLDNF the top clause can only be a denial. All subsequent formulae in a derivation are either denials or the empty clause. To reason forwards from updates, the proof procedure underlying the Consistency Method allows as top clause any deductive rule, denial or negated atom. Subsequent formulae in the derivation may be any of these, the empty clause, or a formula of the form

(*) NOT(A) <-- L_1 and ... and L_n, $n \geq 1$,

where A is an atom and the L_i are literals. This last type of formula can be obtained by an application of the inference rules for implicit deletions, which are described in the next section. For convenience, in the sequel, we use the term *clause* to refer to any deductive rule, denial, negated atom or formula of the form (*).

A *derivation* according to the new proof procedure is a sequence of clauses

$$C_0, C_1, \ldots, C_n$$

such that C_0 is a top clause and C_{i+1} is obtained from C_i as follows. If the selected literal of C_i is a condition of C_i then C_{i+1} is obtained exactly as in SLDNF. If the selected literal is a conclusion then C_{i+1} is either the resolvent of C_i and an input clause (using the standard or a generalised resolution step), or it is a clause derived by an application of an inference rule for implicit deletions. The examples below help illustrate the proof procedure further. A formal definition is given in the next section.

A detailed description of the Consistency Method is given in [11] where it is proved correct in general and complete when the database contains no negative conditions and the transaction contains no deletions. Correctness of the method means that if we obtain a refutation with an update as top clause then the updated database violates its constraints. In cases where the method is complete we can conclude that Comp(DT)∪IT is consistent if all top clauses associated with the updates in T lead to finitely failed search spaces.

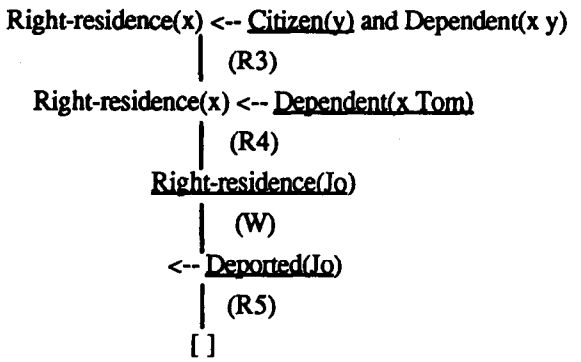
Example 1: Suppose we have the following database, constraint and transaction. Here "Dependent(x y)" means x is a dependent of y. In this paper all constant and predicate symbols start in the upper case, and all variables and function symbols are in the lower case.

D: Right-residence(x) <--
Registered-alien(x) and
NOT(Criminal-record(x)) (R1)
Right-residence(x) <-- Citizen(x) (R2)
Citizen(Tom) (R3)
Dependent(Jo Tom) (R4)
Deported(Jo) (R5)
Deported(Jack) (R6)

I: <-- Right-residence(x) and Deported(x) (W)

T: Add the rule
Right-residence(x) <--
Citizen(y) and Dependent(x y)

D satisfies I. To check if the updated database DT satisfies the constraint we use the update as top clause, with DT∪I as input set. We obtain the following search space, where the selected literals are underlined and [ ] denotes the empty clause:



The search space consists of one refutation illustrating that the updated database violates the integrity constraint.

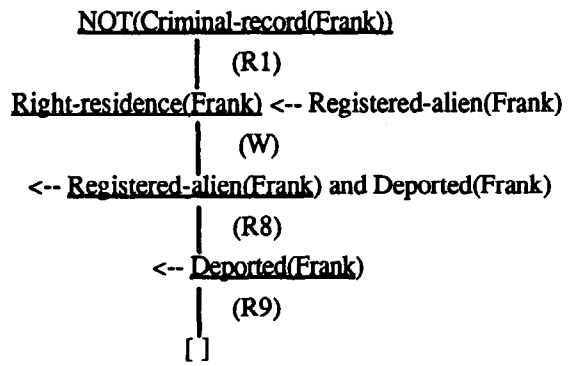
Example 2:

D: As for *Example 1* plus the following facts:
Criminal-record(Frank) (R7)
Registered-alien(Frank) (R8)
Deported(Frank) (R9)

I: Same as *Example 1*

T: Delete Criminal-record(Frank)

D satisfies I. By taking NOT(Criminal-record(Frank)) as top clause we show that DT violates I. The first step is a generalised resolution step, which allows negated atoms to be resolved upon.



Example 3:

D: Same as *Example 2* plus the following facts:
Employed(Alan) (R10)
Registered-alien(Alan) (R11)

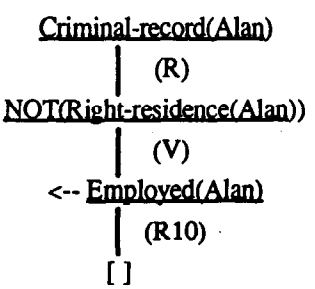
I: <-- Employed(x) and
NOT(Right-residence(x)) (V)

T: Add Criminal-record(Alan)

D satisfies I. To check if DT still satisfies the constraint, we use the update as top clause, as before, but now there is no input clause with which it can resolve. However, intuitively, the addition of the fact "Criminal-record(Alan)" implicitly deletes the fact "Right-residence(Alan)" , which was provable in the database before the update. Thus "NOT(Right-residence(Alan))" is provable in the updated database as a consequence of the update. Since "Employed(Alan)" is also provable in this database, the constraint is violated. We need to reason as follows:

(R) *because* in DT
Criminal-record(Alan) holds and we have
Right-residence(x) <-- Registered-alien(x) and
NOT(Criminal-record(x))
and we have no other way of proving
Right-residence(Alan) and
Right-residence(Alan) was provable in D
then Right-residence(Alan) is deleted.
Thus NOT(Right-residence(Alan)) is provable in DT.

Assuming that we have an inference rule that allows us to reason in this way, we obtain the following search space illustrating the violation of the constraint in the updated database.



There is another symmetric way in which implicit deletion

can occur: when the deletion of a fact implicitly deletes another. For example the deletion of the fact A in the database {A, B<--A} implicitly deletes B. In the next section we show how inference rules for implicit deletions can be generalised and formalised.

(3) Formalisation of the Proof Procedure

The proof procedure is defined using logic as meta-language where the object language is that of the database and the integrity constraints.

To formalise the proof procedure we define a relation Refute(s c) which means that there is a refutation with s as input set and c as top clause. If c is a denial and s a set of deductive rules then Refute(s c) is equivalent to the usual SLDNF provability.

The resolution (standard and generalised) and negation by failure inference rules are formalised by R2 and R3, respectively, together with the base case R1:

(R1) Refute(s [ ])

(R2) Refute(s c) <-- Select(l c) and
In(d s) and
Resolve(d c l r) and
Refute(s r)

(R3) Refute(s c) <-- Select(<-not(p) c) and
NOT(Refute(s <-p)) and
Remove(c <-not(p) r) and
Refute(s r)

Select(l c) means literal l is selected from clause c via a safe literal selection strategy. A literal selection strategy is *safe* if it does not select negative conditions unless they contain no variables. In "Select" and "Remove" the term "<-not(p)" denotes an occurrence of the literal "NOT(p)" in the conditions of a clause. Later we use the notation "l<-" to denote an occurrence of a literal l in the conclusion of a clause.
Remove(c l r) means r is the clause c from which an occurrence of literal l is removed.
Resolve(d c l r) means r is the resolvent of clauses d and c on literal l.
In(d s) means d is a clause in s.
"<-" and "not" are names for "<--" and "NOT".

In (R2) and (R3) an implicit, Prolog-like, unification has been assumed. An alternative approach would be to treat unification explicitly as in the definition of provability given in [5].

Let DT and IT name the updated database and the updated set of integrity constraints obtained after performing the transaction T on database D and constraints I. Then rules (R4) and (R5) cater for the cases of implicit deletions resulting from additions and deletions, respectively. R4 generalises rule R of *Example 3* and R5 deals with the symmetric case.

(R4) Refute(DT∪IT x<-c) <--
Select(x<- x<-c) and
In(a<-b DT) and
On(not(x) b) and
Deleted(DT D a) and
Refute(DT∪IT not(a)<-c)

(R5) Refute(DT∪IT not(x)<-c) <--
Select(not(x)<- not(x)<-c) and
In(a<-b DT) and
On(x b) and
Deleted(DT D a) and
Refute(DT∪IT not(a)<-c)

On(l b) means literal l occurs in the conjunction of literals b. In (R4) and (R5) the variable c stands for a (possibly empty) conjunction of literals.
Deleted(DT D a) means fact a is provable in database D but not in the updated database DT, i.e.

(R6) Deleted(DT D a) <-- Refute(D <-a) and
NOT(Refute(DT <-a))

We now have all the rules we need for defining the relation "Refute" (apart from rules for the subsidiary relations "In", "On", etc.). Let "transact(al dl)" represent the transaction T

which consists of a set of additions al and a set of deletions dl. The following rules, which should be considered as part of the database management system rather than the proof procedure, automatically generate all the updates in T as candidate top clauses.

(I1) IC-Violated(DT∪IT transact(al dl)) <--
On(a al) and
Refute(DT∪IT a)

(I2) IC-Violated(DT∪IT transact(al dl)) <--
On(a<-b dl) and
Deleted(DT D a) and
Refute(DT∪IT not(a))

Note that (I2) also deals with the case where "a<-b" is a fact, i.e. when b is empty. Sadri and Kowalski's formalisation does not cater for updates that modify the integrity constraints. We have changed their formalisation slightly to deal with such updates. Newly added constraints are generated as top clauses by (I1), and deleted constraints are correctly ignored by (I2).

To check if DT satisfies constraints IT, we have to determine whether "IC-Violated(DT∪IT transact(al dl))" is a logical consequence of our formalisation. If we can prove that it is a theorem then the constraints are violated in DT.

(4) Implementation

The Consistency Method for integrity checking has been implemented in Sigma-Prolog running under Unix on the Sun III (see [12] for a detailed account).The interesting features of this implementation are:

- o the representation of the object level clauses,
- o the implementation of reasoning with the two databases D and DT which is required for rule R6,
- o the use of indexing information about input clauses to guide selection of candidate clauses for resolution, thus providing a reasonably efficient search control.

We discuss each of these features, in turn.

The theorem-prover consists essentially of the clauses for "Refute" and the necessary subsidiary definitions. These are represented directly using Sigma-Prolog notation, that is as lists of the form $(M_1...M_n)$, $n \geq 1$, where each M_i has the form (predicate|arguments) and M_1 is the conclusion of the clause.

Object level clauses are represented by terms in the meta-language. They are represented as lists of the form $(L_1...L_n)$, $n \geq 1$, where each L_i has the form (side sign predicate|arguments). "side" is either "Conc" indicating that the literal is the conclusion of the clause or it is "Cond" indicating that the literal is in the conditions. "sign" is either "Pos" for atoms or "Neg" for negated atoms.

The integrity checking method requires reasoning with the initial database D and the updated input set DT∪IT. These sets are represented by clauses of the form

((Member set clause-name clause))

signifying that "clause" belongs to the set named "set" and is given the name "clause-name". These clauses are maintained automatically by preprocessing the initial database and the transaction T. The reason for introducing "clause-name" will become clear shortly when we discuss selection of candidate clauses for resolution. In the implementation "set" is either "OLD", "NEW" or a variable. Clauses identified by the term "OLD" belong to D-DT, where "-" denotes set difference. Those identified by "NEW" belong to (DT-D)∪IT, and those identified by a variable belong to both databases D and DT. Thus to access clauses in DT∪IT we use the identifier "NEW", and to access those in D we use "OLD".

With this convention the top-level Sigma-Prolog goal for integrity checking, for example, is

?((IC-Violated NEW (transact al dl)))

One potential source of inefficiency in the proof procedure is the search involved in rule (R2) for an input clause that

can be resolved with a clause in the derivation on its selected literal. In SLDNF all selected literals are condition literals and therefore SLDNF only needs to search the input set to find conclusions which unify with the selected literal. In the new proof procedure, however, selected literals can come from the conclusion as well as from the conditions of clauses. This requires a larger search over the totality of literals in the input set, looking for unifying conditions as well as conclusions. To reduce this search, for each predicate occurrence "predicate" in each clause in the input sets we include an assertion of the form

(Possible-Resolve input-set clause-name i (side sign predicate))

in the meta-level database. This assertion means that the input clause identified by "input-set clause-name" can potentially resolve (apart from unification of arguments) on its i-th literal with a clause whose selected literal is represented by (side sign predicate|arguments). These assertions are generated automatically, given the two databases and the constraints. There are as many "Possible-Resolve" assertions as there are literal occurrences in the databases and the constraints. Rule (R2) is rewritten to use this set of assertions:

```
((Refute input-set clause)
    (Select (side sign predicate|arguments)
            clause)
    (Possible-Resolve input-set clause-name i
            (side sign predicate))
    (Member input-set clause-name input-clause)
    (Resolve-i input-clause i clause
            (side sign predicate|arguments)
            resolvent)
    (Refute input-set resolvent))
```

Note that "Resolve-i", as compared to "Resolve" in (R2), has an extra argument i which allows fast retrieval of the unifying literal in the input clause. Rules (R4) and (R5) can also be rewritten to exploit the relation "Possible-Resolve".

(5) An Alternative Implementation

Without the inference rules for implicit deletions the proof procedure relies entirely on negation as failure and input resolution. In this form it might be viewed as a fairly minor extension of Prolog's SLDNF proof procedure, and one which, intuitively, should be implementable with comparable efficiency. In this section we show how in this case the need for a meta-interpreter, which is the major overhead of our system, can be avoided by transforming the input clauses.

The "Possible-Resolve" assertions described earlier record for each input clause the literals by which that clause can be entered (i.e. any literal of the clause). Instead of allowing arbitrary literals of input clauses to be resolved upon, we can restrict resolution to the conclusion literal only, if we replace each input clause by as many copies as there are literal occurrences in the clause, with each literal as the conclusion of one of the copies. We add an extra argument to each literal in these copies to indicate whether it comes from the conclusion or the conditions of the original clause c . A "+" argument signifies that the literal is on the same side of "<--" as in c; a "-" argument signifies that it has changed side. This transformation allows us to use Prolog's backwards search to reason forwards as well as backwards.

Example: Input clause

M(x y) <-- N(x y) and NOT(K(y))

is replaced by

M(+ x y) <-- N(+ x y) and NOT(K(+ y))

N(- x y) <-- M(- x y) and NOT(K(+ y))

NOT(K(- y)) <-- M(- x y) and N(+ x y).

We must also transform all top clauses into denials. For example, if an update is the addition of a rule P(x)<--Q(x) then the transformed top clause is "<--P(- x) and Q(+ x)".

If an update is the deletion of an atom P(A) then the transformed top clause is <--NOT(P(- A)).

Example:

D: R(x y) <-- P(x) and Q(y) (R1)
Q(A) (R2)
Q(B) (R3)
T(A) (R4)

I: <-- NOT(T(y)) and R(x y) (I1)

T: Add P(A)

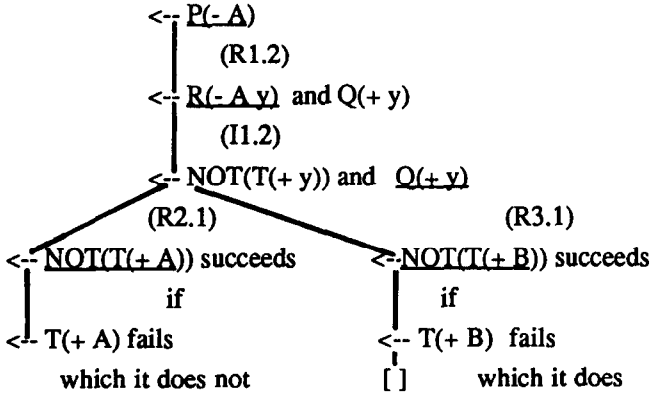
This is transformed to:

D': R(+ x y) <-- P(+ x) and Q(+ y) (R1.1)
P(- x) <-- R(- x y) and Q(+ y) (R1.2)
Q(- y) <-- P(+ x) and R(- x y) (R1.3)
Q(+ A) (R2.1)
Q(+ B) (R3.1)
T(+ A) (R4.1)

I': NOT(T(- y)) <-- R(+ x y) (I1.1)
R(- x y) <-- NOT (T(+ y)) (I1.2)

T': Add P(+ A)

Using <--P(- A) as top clause with Prolog's backward reasoning strategy and left to right literal selection strategy (with a safety condition on the literal selection), we obtain the following search space.



Note that a condition of the form NOT(P(+ args)) can be eliminated only if it is solved by the negation as failure rule, whereas one of the form NOT(P(- args)) can be eliminated only through resolution with an input clause.

Using the same literal selection strategy but reasoning forwards from T with DT∪IT as input set we obtain a search space that has a similar structure to the one above. In general, backward reasoning from the transformed version of the updates using the transformed version of the input set simulates forward reasoning from the original updates using the original input set.

This discussion shows that in the absence of implicit deletions our integrity checking method can be implemented directly in Prolog. If we accept Prolog's literal selection and search strategies we have no need for a special interpreter and can expect efficiencies comparable to implementations of Prolog. We are currently investigating transformations that would also allow us to dispose of the meta-interpreter in the general case where implicit deletions are possible.

(6) Conclusion

We have described a method for checking integrity in deductive databases which achieves the effect of the simplification algorithms of L/T and Decker and which is based on a unified theorem-proving framework. The method uses a proof procedure that is an extension of SLDNF and which allows forward as well as backward reasoning.

We have also described an implementation of this method which shows its feasibility but which also exposes potential inefficiencies. A major overhead is that our system requires a meta-interpreter. We have seen, however, that in the absence of implicit deletions this interpreter can be disposed of. Further work is needed to investigate the possibility of implementing the method directly in Prolog for the general case.

The use of a linear resolution proof procedure as the basis for the Consistency Method makes it easy to compare the method with other simplification algorithms, but is not essential in our approach. This linear proof procedure has

the well-known inefficiencies of SL-resolution which are avoided in the connection graph procedure [4], for example. It would be interesting to investigate how a connection graph proof procedure might be extended to form the basis of the integrity checking method.

In this paper we have discussed only the basic ingredients of the method and it can be improved in many ways. Because it is imbedded within a general-purpose proof procedure, it can, for example, exploit well-researched domain independent heuristics for literal selection and search [5] to enhance efficiency.

We believe that the Consistency Method proof procedure can be useful for a wider range of applications in addition to integrity checking. Forward reasoning is essential for knowledge assimilation as a whole, only one component of which is integrity checking. On adding new data to a knowledge base (kb) we may not only want to check the integrity of the updated kb, but we may want to determine to what extent the new data interacts deductively with the rest of the kb, if it is totally independent of the kb, redundantly implied by the kb, or implies part of the kb.

The ability to mix forward and backward reasoning is also useful for many expert system applications, which often require a flexible mixture of different modes of reasoning.

Acknowledgements

We would like to thank Hendrik Decker, Kave Eshghi, Jean-Marie Nicolas and Rodney Topor for many helpful discussions. We are grateful to Kave Eshghi for his assistance in implementing the theorem-prover.

This work was supported by the Science and Engineering Research Council.

References

[1] Clark, K. L. "Negation as failure", in Gallaire, H. and Minker, J. (eds): "Logic and Data Bases", Plenum, 1978, 293-322.

[2] Decker, H. "The Range Form or How to Avoid Floundering", Internal Report KB-26, May 1987, ECRC, Munich.

[3] Decker, H. "Integrity Enforcement on Deductive Databases", Proc. EDS 86, Charleston, South Carolina, USA, 1986.

[4] Kowalski, R. A. "A Proof Procedure Using Connection Graphs", JACM vol 22, number 4, 1974, 572-595.

[5] Kowalski, R. A. "Logic for Problem Solving", Elsevier North Holland, 1979.

[6] Lloyd, J. W. "Foundations of Logic Programming", Springer Verlag, Symbolic Computation Series, 1984.

[7] Lloyd, J. W., Sonenberg, E. A. and Topor, R. W. "Integrity Constraint Checking In Stratified Databases", Technical Report 86/5, Department of Computer Science, University of Melbourne, 1986.

[8] Lloyd, J. W. and Topor, R. W. "A Basis for Deductive Database Systems", J. Logic Programming, vol 2, number 2, 1985, 93-109.

[9] Lloyd, J. W. and Topor, R. W. "A Basis for Deductive Database Systems II", J. Logic Programming, vol 3, number 1, 1986, 55-67.

[10] Nicolas, J. M. "Logic for Improving Integrity Checking in Relational Data Bases", Acta Informatica, vol 18, number 3, 1982, 227-253.

[11] Sadri, F. and Kowalski R. "An Application of General Purpose Theorem-Proving to Database Integrity", to appear in Minker, J. (ed): "Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming", Morgan Kaufmann, Los Altos, Ca, 1987.

[12] Soper, P. J. R. "Integrity Checking In Deductive Databases", M.Sc. Thesis, September 1986, Department of Computing, Imperial College, University of London.

[13] Topor, R. W., Keddis, T. and Wright, D. W. "Deductive Database Tools", Technical Report 84/7, Revised August 23, 1985, Department of Computer Science, University of Melbourne.