A THEOREM-PROVING APPROACH TO DATABASE INTEGRITY

Fariba Sadri

A thesis submitted to the University of London for the degree of Doctor of Philosophy

Department of Computing
Imperial College of Science and Technology

June 1988

ABSTRACT

This thesis describes the theory, application and implementation of a new method, called the Consistency method, for checking integrity constraints in deductive databases. It also describes a new proof procedure that has been developed for this application.

The Consistency method is suitable for general range-restricted deductive databases where the constraints can be arbitrary formulae of first order predicate logic, and the transactions consist of one or more updates. Each update is an addition, deletion or modification of a database fact or non-atomic rule, or an addition or deletion of an integrity constraint.

The Consistency method is based on general purpose theorem-proving techniques. The new proof procedure is an extension of SLDNF, which is the underlying proof procedure of Prolog. This new proof procedure allows forward as well as backward reasoning, and incorporates the negation as failure rule, as well as additional inference rules for reasoning about implicit deletions.

Backward reasoning is particularly suited to query evaluation, where the database is static but the queries change. Forward reasoning, on the other hand, is particularly suited to knowledge assimilation, one component of which is integrity checking, where the database changes, and it is necessary to investigate the consequences of these changes.

In the Consistency method we reason forward from the updates, and thus focus on the effects of the updates and ignore what remains unchanged. This can increase the efficiency of the integrity checking significantly by avoiding redundant evaluation of those constraints that are satisfied in the database prior to the updates and which are not affected by the updates.

The Consistency method and the new proof procedure have been implemented in Prolog.

The method approximates the algorithms of Decker, Lloyd, Topor, et al, Martens and Bruynooghe, and Bry et al, which are the major existing algorithms for checking integrity of deductive databases.

ACKNOWLEDGEMENTS

I am indebted to my supervisor, Bob Kowalski, without whose patience, guidance and encouragement this thesis would not have been possible.

I am also grateful to Hendrik Decker, Kave Eshghi, Jean Marie Nicolas, Rodney Topor and Paul Soper for many helpful discussions. Kave Eshghi, in particular, has given me invaluable advice particularly on the implementation issues. Thanks are also due to John Lloyd for his thorough reading of the thesis, and for his helpful suggestions for improving it.

I acknowledge with gratitude my two week stay at ECRC in Munich in the summer of 1985, during which time the ideas presented in this thesis began to take shape.

Thanks are also due to Cheryl Anderson and Dania Kowalska for typing part of the thesis. I am very grateful to Cheryl Anderson for teaching me to use the Macintosh, on which this thesis was produced.

This work was supported by the Science and Engineering Research Council, and in part by an ORS award, and by an award from the Department of Computing at Imperial College, for all of which I am very grateful.

CONTENTS

CHAPTER 1

INTRODUCTION

This thesis describes the theory, application and implementation of a new method, based on theorem-proving techniques, for checking integrity constraints in deductive databases. It also describes a new proof procedure that has been developed for this application. The proof procedure allows forward as well as backward reasoning, and can be thought of as an extension of Prolog. Although the proof procedure has been developed with integrity checking in mind, it, in fact, is suitable for knowledge assimilation in general.

Our integrity checking method, called the Consistency method, for reasons that will become clear in the next chapter, is designed for range-restricted deductive databases (defined formally in the next chapter), integrity constraints that are expressed as formulae of first order predicate logic, and transactions that consist of one or more updates. Each update may be an addition, deletion or modification of a fact or non-atomic rule, or an addition or deletion of an integrity constraint.

Deductive databases are extensions of relational databases. Relational databases can be thought of as consisting only of facts. Deductive databases may contain general rules as well as facts. Thus deductive databases have more expressive power, and allow a better compaction of information. A large part of the most recent British Nationality Act, for example, has been formalised as a deductive database and implemented in Prolog (Sergot, Sadri, et al [1986]). It would be very difficult, and quite unnatural to represent complex information such as legislation as a relational database.

Integrity constraints are conditions that the database is expected to satisfy as it changes through time. They are intended to prevent erroneous or undesirable information entering the database. If a transaction violates the constraints, the integrity of the database can be recovered in a number of different ways. One or more updates in the transaction can be rejected, or alternatively, the database or even the constraints can be modified. The recovery may be undertaken autonomously by the database management system, or through interaction with the user.

The following are some examples of integrity constraints expressed in first order logic. Throughout the thesis we use the following notational convention. Predicate and constant symbols start in the upper case, and variable and function symbols start in the lower case. "$\leftarrow$" is the implication sign, and can be replaced by "if".

Example 1.1:

(1) A much quoted constraint is that "human beings are either male or female", or more formally:

$$\forall x\ [\text{Male}(x) \text{ or } \text{Female}(x) \leftarrow \text{Human}(x)].$$

(2) Functional dependencies are particularly common types of

constraints in relational databases. Here is an example: "Each student has exactly one tutor", or more formally:

$\forall x\ \exists y$ [Tutor(y x) $\leftarrow$ Student(x)]

$\forall x\ \forall y\ \forall z$ [y=z $\leftarrow$ Student(x) and Tutor(y x) and Tutor(z x)]

The first constraint states that each student has a tutor, and the second states that each student has no more than one tutor.

(3) Type constraints specify the correct argument types. In relational database terminology these constraints specify the domains of attribute values. For example:

$\forall x\ \forall y$ [Human(x) $\leftarrow$ Salary(x y)]

$\forall x\ \forall y$ [Real-num(y) $\leftarrow$ Salary(x y)]

$\forall x$ [Animal(x) or Plant(x) $\leftarrow$ Animate(x)]

(4) Many constraints can be naturally expressed as denials, which, roughly speaking, are rules with no conclusions (denials are defined formally in the next chapter). Consider, for example, the following constraint:

"no secretary receives a salary higher than a manager".

This can be formalised as:

$\forall x\ \forall y\ \forall u\ \forall v$ [$\leftarrow$Secretary(x) and Manager(y) and
Salary(x u) and Salary(y v) and u>v]

The above formula is equivalent to

$\forall x\ \forall y\ \forall u\ \forall v$ [NOT (Secretary(x) and Manager(y) and
Salary(x u) and Salary(y v) and u>v)].

In general, a formula of the form

$\forall x$ [$\leftarrow$P(x) and Q(x)]

is equivalent to

$\forall x$ [NOT (P(x) and Q(x))],

and can be read as "it is not the case that P and Q are true together for any x". □

All the examples have so far described static constraints, that is constraints specified on a single (usually the current) state of the database. It is also possible to specify dynamic constraints, that is constraints that impose conditions on more than one state of the database. For example, a temporal database that contains historical information can have the following constraint: "salaries do not decrease", or more formally:

$\forall x\ \forall y\ \forall z\ \forall t\ \forall t'$ [$\leftarrow$Sal-at-time(x y t) and Sal-at-time(x z t') and t<t' and z<y],

where "Sal-at-time(x y t)" expresses that x has salary y at time t. A number of formalisms have been proposed for representing temporal information in deductive databases. Sadri [1987] discusses and compares three of the major formalisms.

Alternatively, if D0 and D name the database before and after a transaction, respectively, then the above dynamic constraint can be formalised as follows:

$\forall x\ \forall y\ \forall z$ [$\leftarrow$Demo(D0 salary(x y)) and Demo(D salary(x z)) and z<y],

where "Demo(d p)" expresses that property p is provable in database state d. "salary" is a metalevel function symbol naming the object level predicate symbol "Salary". The constraint can be expressed more generally by replacing the constants D0 and D by variables, d0 and d, say, respectively, and adding an extra condition "Result(d0 t d)" to express that database d results from d0 by transaction t.

Our integrity checking method allows any formula of first order logic as integrity constraint. Thus it can deal with any kind of constraint which is expressible as a first order formula. Note that the last formula, above, is a first order formula. It is, in fact, a formula of amalgamated logic incorporating object and metalevels (Bowen and Kowalski [1982]). We have more to say about this in later chapters.

Since relational databases contain facts only, general rules have to be expressed as integrity constraints. When the constraints are violated they can be used to identify data whose addition to the database would recover integrity. The absence of general rules in the database, although limiting the expressive power, makes the checking of integrity relatively easy in relational databases.

Deductive databases, on the other hand, allow general rules, as well as facts. Nicolas and Gallaire [1978] propose some guidelines for deciding what information should be described as database rules and what should be described as integrity constraints. Intuitively, there is a difference between database rules and integrity constraints. Database rules are statements about the world that is modelled by the database, whereas integrity constraints are statements about the database. Consider the information that "all vicars are men", or more formally:

$$\forall x\ [\text{Man}(x) \leftarrow \text{Vicar}(x)].$$

If this is a database rule, then it allows us to infer that an individual x is a man, given that he is a vicar. On the other hand, if it is an integrity constraint, then it states that if for some individual x, it is possible to prove that x is a vicar, then it is also possible to prove that x is a man.

Alternatively, as a constraint the formula can be interpreted as stating that integrity is violated if for some individual x it is possible to prove that x is a vicar, but it is not possible to prove that x is a man.

It can be argued that since database rules and integrity constraints have different intended meanings, then they should also be distinguished syntactically. Reiter [1988], for example, proposes the use of modal formulae for constraints, and Eshghi and Kowalski [1988] propose the use of metalevel formulae that express provability explicitly. Similar approaches have also been proposed by Noel [1988] and Small [1988]. We have our own way of distinguishing between constraints and database rules, as we will describe in later chapters.

The presence of general rules in deductive databases makes the checking of integrity more difficult than in relational databases, because in deductive databases it is necessary to determine how the updates interact with the existing database, and what new information can be deduced from them.

A simple way of checking integrity constraints is to use a backward reasoning system such as Prolog. The constraints can be set as queries to be checked in the database. This approach, however, can be very inefficient, because it may result in redundant computations, rechecking constraints which are satisfied prior to the transaction and which are not affected by the transaction.

Our method is particularly designed to avoid this inefficiency, and to exploit the assumption that the constraints are satisfied prior to the transaction. We take advantage of this assumption by reasoning forward from the updates. Intuitively, if the database satisfies its constraints before the transaction, then any violation of the constraints after the transaction must be due to the updates. Forward reasoning from the updates has the effect of focusing attention only on parts of the database and the constraints that are affected by the transaction, and ignoring what remains unchanged.

In general, backward (or goal oriented) reasoning, such as the reasoning allowed by Prolog, is most suited to query evaluation in a fixed database. Forward reasoning, on the other hand, is better suited to knowledge assimilation, where the database evolves through time, and the updates have to be assimilated into the database. In this case, it is important to determine how the new knowledge interacts with the old database and its integrity constraints. Integrity checking is only one aspect of knowledge assimilation. It determines if the new knowledge is inconsistent with the old database, or if it violates the constraints. In addition to this, the new knowledge can be related to the old database in a number of different ways (Kowalski [1979]). The new knowledge might, for example, imply part of the old database, in which case a revision of the database might improve the efficiency of query evaluation and save space as well. Alternatively, the new knowledge might be implied by the old database, in which case the new knowledge might simply just be ignored, or it might strengthen confidence in the old knowledge. Finally, the new knowledge might be independent of the old database, that is it might be consistent with the database, but not implied by it, nor might the new knowledge imply any part of the database. In this case the new knowledge might just simply be accepted, or some explanation might be sought for it.

In this thesis we only address the integrity checking component of knowledge assimilation. However, the proof procedure that we have developed for this application, is suitable for knowledge assimilation, in general, because it allows focusing on the updates and exploring the consequent changes. This new proof procedure is an extension of SLDNF (Lloyd [1987]), which is the underlying proof procedure of Prolog. Our proof procedure extends SLDNF by

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

This proof procedure and the integrity checking method have been implemented in Prolog. They have also been proved correct in general, and complete in certain special cases.

There are a number of other integrity checking algorithms, often known as simplification algorithms, that are designed to exploit the assumption of the satisfaction of the constraints in the old database. Bernstein, Blaustein and Clarke [1980], for example, propose such a method for checking certain types of aggregate constraints in relational databases. Nicolas [1982], in a very influential paper, proposes a logic-based simplification method for checking general first order constraints in relational databases. Most of the existing integrity checking methods for deductive databases are extensions of Nicolas' method, amongst these are the algorithms of Lloyd, Sonenberg and Topor [1986], Decker [1986], Martens and Bruynooghe [1987], and Bry, Decker and Manthey [1987].

Our method is also an extension of Nicolas' algorithm. Moreover, it is also an approximation of these latter algorithms. We can, in fact, simulate these algorithms by employing different literal selection and search strategies in our proof procedure.

There are other ways of ensuring the integrity of the database, without actually checking the constraints. Probably the most widely used approach is to "compile" the integrity constraints into the database rules. This is an approach that is used intuitively by almost all programmers, who do not have access to integrity checking facilities. Consider the following simple and informal database and integrity constraint:

database: customer gets goods ← customer orders goods
constraint: ←customer gets goods and
customer in debt

The constraint states that it should not be allowed for a customer who is in debt to receive goods.

Intuitively, a simple way of achieving roughly the same effect, without the integrity constraint, is to rewrite the database rule as follows:

customer gets goods ← customer orders goods and
customer is not in debt,

and do away with integrity checking. The transformed rule ensures that no customer who is in debt gets any goods. Other examples of this kind of "compilation" of constraints in database rules can be found in logic-based planning systems, for example in the situation calculus of McCarthy and Hayes [1969].

This approach to database integrity has been investigated by Asirelli, De Santis and Martelli [1985] for a restricted class of databases and constraints. It is worth exploring this approach further, if only because it is the approach that many programmers adopt quite intuitively, when they need integrity checking, but do not want to employ a separate integrity checking method. We have some on-going research in this area, but this is not in the scope of the thesis. The thesis is concerned with the first approach to database integrity, that is where the constraints are specified explicitly and are checked after transactions.

The thesis is organised as follows. In Chapter 2 we formally define what we mean by deductive databases, integrity constraints and constraint satisfaction. In Chapters 3 and 4 we introduce our integrity checking method by considering two simplified cases. In Chapter 5 we formalise the method in the general case. In Chapter 6 we describe the implementation of the method in Prolog, and propose an alternative and more efficient implementation for a special case. In Chapter 7 we compare our method with other integrity checking algorithms, and in Chapter 8 we discuss the correctness and completeness of our method. Chapter 9 concludes the thesis by a summary and a discussion of further work.

Some of the material presented in this thesis has appeared in Sadri and Kowalski [1988], Kowalski, Sadri and Soper [1987] and in Soper's

M.Sc. thesis (Soper [1986]).

CHAPTER 2

DEFINITIONS

In this chapter we formally define deductive databases, integrity constraints and constraint satisfaction.

2.1 Deductive Databases

A deductive database is a finite set of deductive rules, which are closed formulae of the form

$A \leftarrow L_1$ and . . . and L_n, $\quad n \geq 0$,

where A is an atom, the L_i are literals (i.e. atoms or negated atoms), and all the variables are assumed to be universally quantified in front of the formula in which they occur. A is called the conclusion of the rule and the L_i the conditions. If a condition is an atom then it is a positive condition of the deductive rule. If a condition is a negated atom then it is a negative condition. When n=0 the deductive rule is also called a fact. When n>0 the deductive rule is said to be non-atomic. If all the conditions of a rule are positive then the rule is also called a definite clause. A definite database is a finite set of definite clauses.

It is possible to transform more general formulae of the form "$A \leftarrow W$" into a set of deductive rules. Here, A is an atom, W is an arbitrary first order formula, and all the variables in A and all the free variables in W are assumed to be universally quantified in front of the formula. Such transformations are described by Lloyd and Topor [1984].

For our integrity checking method we assume that the database before and after any updates is range-restricted (r-r). A database is r-r if and only if all the rules in it are r-r. A deductive rule is r-r if and only if any variable that occurs in it has an occurrence in a positive condition of the rule. The motivation for this restriction, which is to avoid "floundering", is discussed in Chapter 4.

Example 2.1:

The following rules are r-r:

P(x y) ← Q(x y) and NOT R(y)

M(x y z) ← P(x x) and Q(y z)

N(x y) ← Q(x z) and R(y) and NOT S(z).

The following rules are not r-r:

P(x y) ← Q(x x) and NOT R(y)

M(x y z) ← P(x x) and Q(y y)

N(x y) ← Q(x y) and R(y) and NOT S(z). □

The r-r restriction corresponds exactly to Decker's "range-restriction" (Decker [1987]), and to the "allowed" condition of Lloyd and Topor [1986] and Topor and Sonenberg [1988].

2.2 Integrity Constraints

Our integrity checking method deals directly with constraints of the form

$\leftarrow L_1$ and ... and L_n, $n > 0$,

where the L_i are literals and all variables are assumed to be universally quantified in front of the constraint in which they occur. We call formulae of this kind denials, or sometimes goals. If a literal in a denial is an atom then it is a positive condition of the denial, and if a literal is a negated atom then it is a negative condition. Denials that have positive conditions only are called negation-free denials (goals), or n-f denials (goals), for short.

Constraints must also be range-restricted, that is any variable that occurs in a negative condition of a denial representing a constraint must also have an occurrence in a positive condition of the denial.

It is also possible to deal with constraints that are in a more general form than denials. Given an arbitrary closed first order formula W as a constraint, we can replace it by a new constraint "$\leftarrow$A", and add a rule

A $\leftarrow$NOT W

to the database, where A is a nullary predicate symbol that does not occur elsewhere in the database or the constraints. The rule "A$\leftarrow$NOT W" can then be transformed to a set of deductive rules as described by Lloyd and Topor [1984]. Because the resulting deductive rules must be range-restricted, this imposes a corresponding range-restriction on the form of the integrity constraint W. The restricted quantification condition on the constraints proposed by Bry et al [1987] is sufficient to ensure that the transformed rules are range-restricted. A closed first order formula F satisfies this condition if and only if every subformula of F is either of the form

$$\exists x_1 \ldots \exists x_n \, [A_1 \text{ and } \ldots \text{ and } A_m \text{ and } Q], \qquad m \geq 1,$$

or of the form

$$\forall x_1 \ldots \forall x_n \, [A_1 \text{ and } \ldots \text{ and } A_m \rightarrow Q], \qquad m \geq 1,$$

where the A_j are atoms, every variable x_i occurs in at least one A_j, and Q is a formula. Some or all of the x_i may be free in Q.

Example 2.2:

Consider the following integrity constraint on a database D:

"Each employee has a supervisor who is a manager".

This can be represented by the first order formula:

$$\forall x \, [\text{Employee}(x) \rightarrow \exists y \, [\text{Supervisor}(y\ x) \text{ and } \text{Manager}(y)]].$$

We replace this by a new constraint "$\leftarrow$A", assuming that the nullary predicate A does not occur in D or in any other integrity constraint on D. We also add the rule

$$A \leftarrow \text{NOT } \forall x[\text{Employee}(x) \rightarrow \exists y[\text{Supervisor}(y\ x) \text{ and } \text{Manager}(y)]]$$

to the database. This rule is then transformed, as described in Lloyd and Topor [1984] into the following deductive rules:

$$A \leftarrow \text{Employee}(x) \text{ and NOT AUX}(x)$$

$$\text{AUX}(x) \leftarrow \text{Supervisor}(y\ x) \text{ and } \text{Manager}(y),$$

where "AUX" is a new predicate symbol not occurring anywhere else in the database or the constraints. □

If a constraint W is in non-Horn clausal form, that is in the form

$$B_1 \text{ or } \ldots \text{ or } B_m \leftarrow A_1 \text{ and } \ldots \text{ and } A_n, \qquad n \geq 0, m \geq 0,$$

where the A_i and the B_i are atoms, then W can simply be rewritten directly as the denial

$$\leftarrow A_1 \text{ and } \ldots \text{ and } A_n \text{ and NOT } B_1 \text{ and } \ldots \text{ and NOT } B_m.$$

We show the correctness of these rewritings in Chapter 8.

Throughout this thesis we restrict our attention to sets of integrity constraints which are mutually consistent. Thus we do not allow, for example, both

$$A\leftarrow \quad \text{and} \quad \leftarrow A$$

to belong to the same set of integrity constraints. Bry, Decker and Manthey [1987], and Bry and Manthey [1986] present an algorithm for checking the consistency (and finite satisfiability) of a set of integrity constraints. (A set of formulae is finitely satisfiable if and only if it has a finite model.) Their algorithm resembles the tableaux method (Smullyan [1968]), and is based on the principle of constructively interpreting the inductive definition of formula semantics. The algorithm is complete for unsatisfiability (as well as for finite satisfiability). Thus if the algorithm terminates successfully, then finite satisfiability, and consequently the consistency, of the formulae is shown. On the other hand, if the algorithm fails, then unsatisfiability, and therefore inconsistency, is shown. In cases where all models of the formulae are infinite the algorithm will not terminate. The algorithm has been implemented by the authors in Prolog.

In the remainder of the thesis, without loss of generality, we assume that constraints are of the form of denials, unless otherwise stated.

2.3 Constraint Satisfaction

The most commonly used definition of constraint satisfaction is that a database D satisfies its constraints I, where I is a set of closed formulae, if and only if the completion of D is consistent, and every formula in I is a logical consequence of the completion of D. We call this the theoremhood view of constraint satisfaction.

The completion of a database D, denoted Comp(D), consists essentially of D together with the "only-if" versions of the rules in D and an appropriate equality theory. For a more precise definition of completion see Clark [1978] or Lloyd [1987]. Here it is sufficient for us to give the following definition.

Comp(D) consists of the completed definitions (to be described below) of all the relations that occur in D, together with an appropriate equality theory. The equality theory states certain properties of the constants and the function symbols in the language, for example

$c \neq d$,

for all pairs c, d of distinct constants,

$f(x_1 \ldots x_n) \neq c$,

for each constant c and function f and any variables $x_1, \ldots, x_n$, and

$f(x_1 \ldots x_n) \neq g(y_1 \ldots y_m)$,

for all pairs f and g of distinct functions and any variables $x_1, \ldots, x_n$,

$y_1,\ldots, y_m$. "$\neq$" denotes not equal.

The completed definitions of database relations are defined as follows. Let P be a relation and

(1) $\quad P(t_1 \ldots t_n) \leftarrow L_1$ and ... and L_m

be a deductive rule defining P in database D . The t_i are terms, and the L_i are literals. This definition is equivalent to the following database rule:

(2) $\quad P(x_1 \ldots x_n) \leftarrow x_1=t_1$ and ... and $x_n=t_n$
and L_1 and ... and L_m,

where the x_i are variables that do not appear in (1). Recall that all the variables occurring in a deductive rule are assumed to be universally quantified in front of the rule. Thus (2) is equivalent to

(3) $\quad P(x_1 \ldots x_n) \leftarrow \exists y_1 \ldots \exists y_r$
$[x_1=t_1$ and ... and $x_n=t_n$
and L_1 and ... and $L_m]$,

where $y_1,\ldots, y_r$ are the variables that occur in (1). Now let

$P(x_1 \ldots x_n) \leftarrow E_1$

:

$P(x_1 \ldots x_n) \leftarrow E_k$

be the transformed versions, as in (3), of all the rules in D that define the relation P. Then the completed definition of P is the formula

$\forall x_1 \ldots \forall x_n [P(x_1 \ldots x_n) \leftrightarrow E_1$ or ... or $E_k]$.

We assume that the completion of D contains a denial of the form

$\leftarrow Q(x_1 \ldots x_n)$

for every n-ary predicate Q in the underlying language, such that Q is not defined in D (i.e. does not occur in the conclusion of any deductive rule in D).

The definition given above for completion is general and covers all deductive rules. It can, however, be simplified in certain circumstances. One useful simplification is as follows. Consider definition (1), above, of the relation P. If in (1), for some q, $1 \leq q \leq n$, t_q is a variable, x, say, then (2) can be simplified to

$$P(x_1 \ldots x_n) \leftarrow x_1=t_1 \text{ and} \ldots \text{and } x_{q-1}=t_{q-1} \text{ and } x_{q+1}=t_{q+1} \text{ and} \ldots \text{and } x_n=t_n \text{ and } L,$$

and (3) can be simplified to

$$P(x_1 \ldots x_n) \leftarrow \exists y_1 \ldots \exists y_s \; [x_1=t_1 \text{ and} \ldots \text{and } x_{q-1}=t_{q-1} \text{ and } x_{q+1}=t_{q+1} \text{ and} \ldots \text{and } x_n=t_n \text{ and } L],$$

where L is the conjunction "L_1 and ... and L_m" in which the variable x is systematically substituted by x_q, and $y_1, \ldots, y_s$ are all the variables in (1) apart from x.

Example 2.3:

Suppose relation Q is defined as follows:

Q(B C)

Q(A B).

Then the completed definition of Q is

$$\forall x \, \forall y \, [Q(x\ y) \leftrightarrow [(x=B \text{ and } y=C) \text{ or } (x=A \text{ and } y=B)]].$$

The equality theory will include the following inequalities:

$B \neq C$

$A \neq B$

$A \neq C.$ □

Example 2.4:

Suppose relation P is defined as follows:

$P(x\ A)$

$P(x\ y) \leftarrow Q(x\ y)$ and NOT $R(y)$

$P(B\ y) \leftarrow S(y)$ and $T(z\ y)$.

Then the completed definition of P is the following formula:

$$\forall x\ \forall y[P(x\ y) \leftrightarrow [(y=A) \text{ or } (Q(x\ y) \text{ and NOT } R(y)) \text{ or } \exists z(x=B \text{ and } S(y) \text{ and } T(z\ y))]].$$ □

Example 2.5:

Suppose relation T is defined as follows:

$T(f(x)\ A) \leftarrow N(x)$

$T(x\ g(y)) \leftarrow M(y\ x)$ and NOT $N(h(x))$.

Then the complete definition of T is:

$$\forall x_1\ \forall x_2\ [T(x_1\ x_2) \leftrightarrow [\exists x\ (x_1=f(x) \text{ and } x_2=A \text{ and } N(x)) \text{ or } \exists y(x_2=g(y) \text{ and } M(y\ x_1) \text{ and NOT } N(h(x_1)))]].$$ □

In practice, for convenience, only the database is represented explicitly and reasoning with its completion is implemented through the negation as finite failure rule (Clark [1978]), which is described in detail in Chapter 4.

In our method we also appeal to the completion, but we use a definition of constraint satisfaction which is slightly different from the theoremhood view. According to our definition a database D satisfies integrity constraints I if and only if the completion of D is consistent with I. We call this the consistency view of constraint satisfaction. Our method also uses negation as failure to implement reasoning with the completion of the database.

The two definitions of constraint satisfaction are equivalent if the completion of the database is consistent, and for any closed formula W in the language of the database and the constraints, either W or its negation is a logical consequence of the completion of the database. The completion of such a database is said to be complete. The two definitions can give different results when the database includes recursive definitions.

Example 2.6:

Let database D consist of the following rule:

$P \leftarrow P.$

Consider the constraint

P.

This constraint is not a theorem of Comp(D), but it is consistent with Comp(D). □

Notice that if the completion of the database is consistent then any theorem of the completion is consistent with the completion. Thus if such a database satisfies its constraints according to the theoremhood view then it also satisfies them according to the consistency view. We discuss the relationship between the two definitions further in Proposition 4.1, in Subsection 4.1.1, where we show that, in the context of a particular implementation of the two views, the difference between them is greatly reduced.

A sufficient condition for the consistency of Comp(D) is that D be stratified (Apt, Blair and Walker [1988]). A deductive database is stratified if there is a mapping M from its set of predicate symbols to natural numbers (the non-negated integers) such that for every database rule R of the form "$P(t_1 \ldots t_n) \leftarrow$Conditions", where the t_i are terms and "Conditions" is a conjunction of literals

$M(Q) \leq M(P)$ if Q is a predicate of a positive condition of R, and

$M(S) < M(P)$ if S is a predicate of a negative condition of R.

Thus the stratification condition allows recursion but in a limited form. It excludes rules such as "P $\leftarrow$ NOT P". If this rule is the only definition of P in a database D, then the completed definition of P would be

$P \leftrightarrow$ NOT P,

which logically implies

P and NOT P.

Thus Comp(D) would be inconsistent.

We have chosen the consistency view not because we believe it is always superior to the theoremhood view, but because we believe it is preferable to the theoremhood view in certain situations. For example, the consistency view is needed for implementing abduction (see, for example, Eshghi and Kowalski [1988], Cox and Pietrzykowski [1986], or Poole [1987]). In a system which incorporates abduction, during the deductive process, propositions may be assumed provided that their addition to the database is consistent with the database and the integrity constraints.

Abduction has been applied to such areas as default reasoning (Eshghi and Kowalski [1988] and Poole [1987]), planning (Eshghi [1988]), natural language understanding (Kakas [1987] and Charniak and McDermott [1985]), and diagnosis (Cox and Pietrzykowski [1986] and Goebel et al [1986]).

The consistency view is also appropriate in a deductive system which incorporates query-the-user facilities, such as the expert system shell APES (Hammond and Sergot [1984]). In such systems it is important to check that the information volunteered by the user is consistent with certain pre-specified integrity constraints and the rest of the database.

CHAPTER 3

THE CONSISTENCY METHOD: SIMPLIFIED CASE 1

In this chapter we describe the Consistency method in a simplified case where the database is definite, the integrity constraints are negation-free denials and the transactions consist of additions only. In this case the Consistency method proof procedure is a form of input resolution which allows any definite clause or negation-free denial as top clause and employs an unrestricted literal selection strategy. This proof procedure is not a special case of SL nor of SLD.

SL (Kowalski and Kuehner [1971]) is a linear proof procedure for non-Horn clauses which allows any clause as top clause but which imposes a last-in-first-out literal selection strategy. (Recall that a non-Horn clause is a formula of the form

$$B_1 \text{ or } \dots \text{ or } B_m \leftarrow A_1 \text{ and } \dots \text{ and } A_n, \qquad n \geq 0, m \geq 0,$$

where the A_i and the B_i are atoms.)

SLD (Apt and van Emden [1982]) (also called Lush resolution in Hill [1974]) is an input proof procedure for definite clauses which allows an unrestricted literal selection strategy (as liberal as ours) but which allows only negation-free denials as top clauses. SLD is the underlying proof procedure of Prolog without negation as failure. The difference between input and linear proof procedures, and the relationship between our proof procedure and SL and SLD will become more clear as this chapter progresses.

Our proof procedure for this simplified case can be most easily understood if it is viewed as an extension of SLD which retains SLD's unrestricted literal selection, but which allows definite clauses as well as negation-free denials as top clauses.

This chapter is divided into three sections. In 3.1 we describe our proof procedure in this simplified case by first describing SLD and then extending it. In 3.2 we illustrate the application of the Consistency method through examples, and in 3.3 we compare our simplified proof procedure with SL.

3.1 The Proof Procedure

3.1.1 The SLD Proof Procedure

SLD is an input proof procedure for definite clauses. A derivation in SLD consists of a (possibly infinite) chain of resolvents as in the figure below:

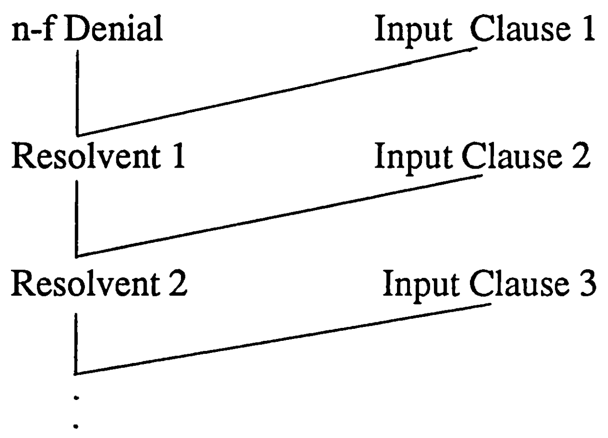


Figure 3.1: The form of an SLD derivation

The input clauses are the rules in the database. A selection function, also called a computation rule in the sequel, selects an atom to be resolved upon next in each resolvent in the chain. Each (i+1)th resolvent is obtained by the resolution of the (i)th resolvent on its selected atom with an input clause. This is the distinguishing characteristic of every input proof procedure. We describe the SLD proof procedure more formally in the rest of this subsection. Lloyd [1987] presents an alternative approach.

A computation rule is a function from derivations to atoms. It selects an atom from the last formula in the derivation.

Let S be a set of definite clauses, G a negation-free denial and R a computation rule. An SLD derivation of $S \cup \{G\}$, where "$\cup$" denotes set union, via R is a (possibly infinite) sequence G_0, G_1, G_2, ..., such that $G=G_0$ and for each i, $i \geq 0$, G_{i+1} is obtained from G_i as follows. Suppose R selects from G_i an atom occurrence A. Then G_{i+1} is the resolvent on A of G_i and some clause in S which resolves with G_i on A. S is called the input set, and G is called the top clause. Each G_i, for i>0, is either a negation-free denial or the empty clause.

A refutation of $S \cup \{G\}$ via R is a derivation of $S \cup \{G\}$ via R which ends at the empty clause.

A negation-free goal G succeeds from S if and only if for some computation rule R there is an SLD refutation of $S \cup \{G\}$ via R.

A finitely failed derivation of $S \cup \{G_0\}$ is a derivation $G_0, G_1, \ldots, G_n$, $n \geq 0$, such that G_n is not the empty clause, and G_n does not resolve on its selected atom with any clause in S.

An SLD search space for $S \cup \{G_0\}$ via R is the set of all SLD derivations of $S \cup \{G_0\}$ via R such that any finite derivation in the set is either a refutation or a finitely failed derivation.

Notice that because of the definition of computation rule, for any two derivations of the form

$$G_0, G_1, \ldots, G_k, G_{k+1}, \ldots$$
$$G_0, G_1, \ldots, G_k, G'_{k+1}, \ldots$$

in a search space, the same literal is selected from G_k. This is less restrictive than having to select the same atom from any two identical goals occurring on different derivations in the search space.

A finitely failed search space is a search space that consists entirely of finitely failed derivations.

The SLD proof procedure is correct because the only inference rule it uses is resolution, and resolution has been proved correct by Robinson [1965]. SLD has been proved complete by Clark [1979] and Hill [1974]. The proofs are also presented in Lloyd [1987]. The correctness of SLD means that if for some computation rule R there is an SLD refutation of $S \cup \{G\}$ via R, then $S \cup \{G\}$ is inconsistent. The completeness of SLD means that if $S \cup \{G\}$ is inconsistent, then for all SLD computation rules R there is an SLD refutation of $S \cup \{G\}$ via R.

By correctness of SLD, if a negation-free goal "$\leftarrow W$" succeeds from S then "$\exists x_1 \ldots \exists x_n W$" is a logical consequence of S, where the x_i are all the variables that occur in W.

Example 3.1:

Suppose goal G is "$\leftarrow P(A)$" and input set S consists of the following rules:

(1) $P(x) \leftarrow Q(x)$ and $R(x\ y)$

(2) $P(x) \leftarrow S(x\ y)$ and $T(y)$

(3) $Q(x) \leftarrow T(x)$

(4) $T(A)$ (5) $R(A\ B)$ (6) $S(A\ B)$

Then the following represents an SLD search space for $S \cup \{G\}$. Throughout the thesis "[ ]" denotes the empty clause. Whenever more than one literal is candidate for selection, the selected literal is underlined. The numbers on the arcs denote input clauses that are used for resolution.

$$\begin{tikzcd}
 & \leftarrow P(A) \arrow[dl, dash, "(1)"'] \arrow[dr, dash, "(2)"] & \\
\leftarrow \underline{Q(A)} \text{ and } R(A\ y) \arrow[d, dash, "(3)"'] & & \leftarrow \underline{S(A\ y)} \text{ and } T(y) \arrow[d, dash, "(6)"] \\
\leftarrow \underline{T(A)} \text{ and } R(A\ y) \arrow[d, dash, "(4)"'] & & \leftarrow T(B) \\
\leftarrow R(A\ y) \arrow[d, dash, "(5)"'] & & \\
[\,] & &
\end{tikzcd}$$

Figure 3.2: An SLD search space for example 3.1

The search space consists of two derivations, a refutation on the left and a finitely failed derivation on the right. The refutation proves that "P(A)" is a logical consequence of S. □

3.1.2 Our Proof Procedure

As mentioned earlier, to exploit the assumption that the database satisfies its constraints prior to the transaction, the Consistency method reasons forward from the updates. Thus the underlying proof procedure must allow as top clauses deductive rules (facts or non-atomic rules) corresponding to updates that insert them into the database, and denials corresponding to updates that add new integrity constraints. In the more general case described in Chapter 4, the proof procedure must also allow negated facts as top clauses. This is needed to deal with deletions from the database, as will be seen later. When the input set is definite and the top clause is a negation-free denial, our proof procedure is identical to SLD. In all other cases our proof procedure is an extension of SLD.

Below, we describe the simplified version of our proof procedure needed for the simplified case discussed in this chapter. This proof procedure extends SLD by allowing as top clauses definite clauses as well as negation-free denials.

As in SLD, a <u>computation rule</u> in our proof procedure is a function from derivations to atoms, such that it selects an atom from the last formula in the derivation.

Let input set S consist of a set D of definite clauses and a set I of negation-free denials. Let C be an element of D or I, and let R be a computation rule. A <u>derivation</u> for S via R with <u>top clause</u> C in our proof procedure is a (possibly infinite) sequence $C_0, C_1, C_2, ...,$ such that $C=C_0$, and for all i, $i \geq 0$, C_{i+1} is the resolvent of C_i, on the atom occurrence selected by R, and a definite clause or a denial in S. Each C_i is thus a definite clause, a negation-free denial, or the empty clause.

As in SLD, a refutation in our proof procedure is a derivation that ends at the empty clause, and a negation-free goal G succeeds from S if and only if for some computation rule R there is a refutation via R with G as top clause and S as input set.

A finitely failed derivation via R is a derivation $C_0, C_1, ..., C_n$, via R such that C_n is not the empty clause, and C_n does not resolve on its selected atom with any definite clause or denial in the input set.

A search space for S via R with top clause C_0 is the set of all derivations for S via R with C_0 as top clause, such that any finite derivation in the set is either a refutation or a finitely failed derivation.

A finitely failed search space is a search space that consists entirely of finitely failed derivations.

In Chapter 8 we prove that our proof procedure is correct and complete for integrity checking in the special case discussed here, in the following sense.

Let T name a transaction. Thus T consists of one or more updates, where each update is an addition of a definite clause, or an integrity constraint in the form of a negation-free denial. Suppose D and DT name the database before and after the transaction, respectively, and let I and IT name the set of constraints before and after the transaction, respectively. We use this naming convention throughout the thesis.

The correctness of our method is defined as follows. If for some computation rule there is a refutation of DT$\cup$IT with an update in T as top clause, then Comp(DT)$\cup$IT is logically inconsistent, and thus the transaction violates the constraints. Conversely, completeness is defined as follows. If Comp(D)$\cup$I is consistent but Comp(DT)$\cup$IT is not, then for all computation rules R and for some update C in T there is a refutation of DT$\cup$IT via R with C as top clause. As a corollary to completeness, if for all updates C in T there is a finitely failed search space for DT$\cup$IT with C as top clause, via some computation rule, then Comp(DT)$\cup$IT is consistent, and therefore the transaction satisfies the constraints.

In our use of the terms "correctness" and "completeness" we have taken a theorem proving point of view. From the integrity checking point of view it is also important to consider the notion of "soundness", which is the combination of correctness and the above corollary to completeness.

In general, when a refutation is obtained it can be examined to identify the clauses that contribute to the proof of inconsistency, and which are therefore candidates for revision to restore integrity.

3.2 Examples

In this section we illustrate our method through a series of examples. The method is described more generally in Chapter 5.

3.2.1 Adding Facts

Example 3.2:

In this example the transaction consists of a single insertion into the database, and the database is relational, that is it consists entirely of facts that contain no variables. "Rank(x y)" expresses that x has rank y, and "Proj(x y)" expresses that x works on project y.

D:

(1)	Rank(John Lect)	(5)	Proj(John LAW)
(2)	Rank(Tom Prof)	(6)	Proj(Tom MMI)
(3)	Rank(Mary Lect)	(7)	Proj(Mary PARLOG)
(4)	Rank(Peter Reader)	(8)	Proj(Peter MMI)
		(9)	Proj(Jo MMI)

I:

(IC) $\leftarrow$ Rank(x Lect) and Proj(x MMI)

The constraint states that no lecturer works on project MMI (man-machine interface).

T:

Insert Rank(Jo Lect).

Let us assume (correctly) that D satisfies its constraint. To determine whether the updated database, i.e. DT=D∪{Rank(Jo Lect)}, still satisfies the constraint we apply our proof procedure with DT∪I as input set and the update as top clause. We obtain the following search space.

$$\begin{tikzcd}
\text{Rank(Jo Lect)} \arrow[d, dash, "\text{(IC)}"] \\
\leftarrow\text{Proj(Jo MMI)} \arrow[d, dash, "\text{(9)}"] \\
[\,]
\end{tikzcd}$$

<u>**Figure 3.3:** A search space for example 3.2 with the update as top clause</u>

The search space consists of a single refutation illustrating that the transaction violates the integrity constraint.

The advantage of selecting the updates as top clauses is that it limits attention to the relevant parts of the database and the relevant instantiations of the integrity constraints.

If the completion of the database is consistent, then any inconsistency must involve an integrity constraint. The completion of a relational database is always consistent, since a relational database is stratified. Thus in example 3.2, instead of the update we can choose the constraint as top clause, while still using the updated database as input set. This results in the following search space, given the literal selection rule indicated by underlining. Because the top clause is a negation-free denial the same search space is obtained by the SLD proof procedure as well.

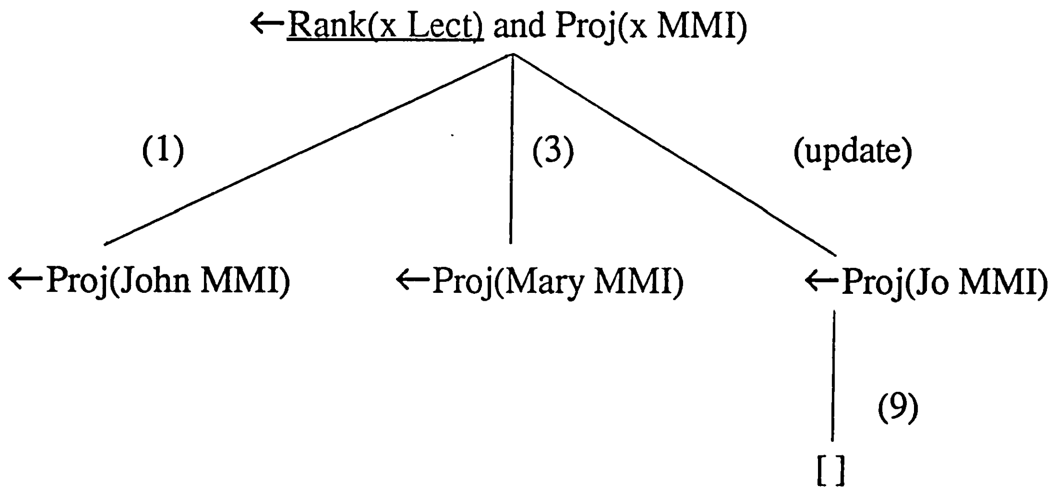


Figure 3.4: A search space for example 3.2 with the constraint as top clause

This search space consists of two finitely failed derivations and a refutation. As the search space in figure 3.3, figure 3.4 also demonstrates that the updated database violates the integrity constraint (because of the correctness of our proof procedure, as shown in Chapter 8, or alternatively because of the correctness of SLD). However, it is larger than the previous search space, because it does not take advantage of the assumption that the constraint is satisfied before the transaction. The alternative literal selection strategy that chooses the second literal of the constraint would also result in a search space as large as figure 3.4, with two finitely failed derivations and one refutation. $\square$

Note that we have defined constraint satisfaction in terms of the completion of the database. But we have described the correctness and completeness results of the SLD proof procedure in terms of the database, itself, and not its completion. This apparent incongruity is resolved by the following proposition.

Proposition 3.1:

Let D be a set of definite clauses, and I a set of negation-free denials. Then $D \cup I$ is logically inconsistent if and only if $\text{Comp}(D) \cup I$ is logically inconsistent.

Proof:

(1) Suppose $D \cup I$ is logically inconsistent. D is a logical consequence of Comp(D) (Lloyd [1987]). Therefore $\text{Comp}(D) \cup I$ is also logically inconsistent.

(2) Suppose $\text{Comp}(D) \cup I$ is inconsistent. Let I be the set of constraints $\{\leftarrow C_1, \ldots, \leftarrow C_n\}$, where each C_i is a conjunction of atoms. $\text{Comp}(D) \cup I$ is inconsistent. So Comp(D) is inconsistent with

$$\text{NOT}\, \exists x^*_1\, C_1 \text{ and .... and } \text{NOT}\, \exists x^*_n\, C_n,$$

where each x^*_i is a vector of all the variables that occur in C_i. So

$$\exists x^*_1\, C_1 \text{ or .... or } \exists x^*_n\, C_n$$

is a logical consequence of Comp(D).

Now any positive sentence that is a consequence of Comp(D) is also a consequence of D (Theorem 15 of Shepherdson [1988]). (A sentence is positive if and only if it is built up using only the connectives "and" and "or", and the quantifiers "$\forall$" and "$\exists$".) So

$\exists x^*_1 C_1$ or .... or $\exists x^*_n C_n$

is a logical consequence of D. Therefore $D \cup I$ is inconsistent. $\square$

3.2.2 Adding Non-Atomic Database Rules

The insertion of non-atomic rules is treated exactly as the addition of facts, as shown in the following example.

Example 3.3:

(The relations, below, are intended to have their intuitive meaning.)

D:

(1) Eligible(x SERC-grant) $\leftarrow$ Student(x) and
Citizen(x UK)

(2) Eligible(x Brit-Council-award) $\leftarrow$
Student(x) and
Citizen(x y) and
Dependent-territory(y)

(3) Dependent-territory(Falkland-Islands)

(4) Student(Mary)

(5) Citizen(Mary Falkland-Islands)

(6) Student(Tom)

(7) Citizen(Tom UK)

I:

(IC) $\leftarrow$ Eligible(x SERC-grant) and
        Eligible(x Brit-Council-award)

The constraint states that no one is eligible for both an SERC grant and a British Council award.

T:

Insert Citizen(x UK) $\leftarrow$ Citizen(x Falkland-Islands).

As before we use the update as top clause and the updated database and the constraint as input set. The following refutation shows that the update violates the constraint.

$$\begin{tikzcd}
\text{Citizen(x UK)} \leftarrow \underline{\text{Citizen(x Falkland-Islands)}} \arrow[d, dash, "(5)"] \\
\text{Citizen(Mary UK)} \arrow[d, dash, "(1)"] \\
\text{Eligible(Mary SERC-grant)} \leftarrow \underline{\text{Student(Mary)}} \arrow[d, dash, "(4)"] \\
\text{Eligible(Mary SERC-grant)} \arrow[d, dash, "(IC)"] \\
\leftarrow \text{Eligible(Mary Brit-Council-award)} \arrow[d, dash, "(2)"] \\
\begin{array}{l} \leftarrow \underline{\text{Student(Mary)}} \text{ and Citizen(Mary y) and} \\ \quad\quad \text{Dependent-territory(y)} \end{array} \arrow[d, dash, "(4)"] \\
\leftarrow \underline{\text{Citizen(Mary y)}} \text{ and Dependent-territory(y)} \arrow[d, dash, "(5)"] \\
\leftarrow \text{Dependent-territory(Falkland-Islands)} \arrow[d, dash, "(3)"] \\
[\,]
\end{tikzcd}$$

Figure 3.5: A refutation for example 3.3 with the update as top clause

The complete search space contains two other derivations that fail finitely. □

3.2.3 Adding Integrity Constraints

This case is similar to the cases of adding facts and non-atomic rules. Thus if a negation-free denial (IC) is to be added to the set of integrity constraints then (IC) is selected as top clause for the proof procedure.

3.2.4 Transactions With Multiple Updates

In general, when a single transaction consists of several updates, each update is a candidate top clause. The input set, as usual, consists of the updated database and the updated set of integrity constraints. If an update leads to a refutation then the transaction violates the integrity constraints. (The proof of inconsistency can then be analysed to determine which of the updates contribute to the inconsistency, and to identify candidate database clauses or constraints for revision to restore integrity.) If all the updates lead to finitely failed search spaces, and our method is complete for the given case, then the transaction satisfies the constraints.

Example 3.4:

In this example

"On-sand(x)"	means x is on a sandwich course,
"Grad(x)"	means x is a graduate,
"Intro(x)"	means x is an introductory course,
"Prac(x)"	means x is a practical course,
"Takes(x y)"	means x takes course y,
"Adv(x)"	means x is an advanced course,
"Sponsored-by(x y)"	means x is sponsored by y, and
"Supervised-by(x y)"	means x is supervised by y.

D:

(1) Grad(Alice)

(2) Grad(Tom)

(3) Grad(Dick)

(4) On-sand(John)

(5) On-sand(Mary)

(6) Sponsored-by(John BT)

(7) Intro(C1)	(10) Prac(C1)	(13) Adv(C5)
(8) Intro(C2)	(11) Prac(C2)	(14) Adv(C6)
(9) Intro(C3)	(12) Prac(C5)	

(15) Takes(x y) ← Prac(y) and On-sand(x) and Sponsored-by(x BT)

I:

(IC1) ← Intro(x) and Adv(x)

(IC2) ← Sponsored-by(x BT) and Supervised-by(x Prof-Smith)

T:

Insert {(IC3) ←Grad(x) and Takes(x y) and Intro(y)

(16) Adv(C7)

(17) Supervised-by(x Prof-Smith) ← Grad(x) and Takes(x C6)}

D satisfies I. The transaction consists of three updates, two insertions into the database and one insertion of a new integrity constraint. To check if DT satisfies IT, each of the three updates must be considered as top clause. In each case the input set consists of DT∪IT, i.e. D∪{(16), (17)}∪I∪{(IC3)}. The three search spaces are shown below. All three fail finitely. Our method is complete for this example, as will be shown in Chapter 8. Therefore we can conclude that the updated database satisfies its constraints.

```
←Grad(x) and Takes(x y) and Intro(y)
  |
  | (15)
  |
←Grad(x) and Prac(y) and On-sand(x) and
  Sponsored-by(x BT) and Intro (y)
  |
  | (6)
  |
←Grad(John) and Prac(y) and On-sand(John) and Intro(y)
```

Figure 3.6: A search space for example 3.4 with an update as top clause

$$\begin{tikzcd} \text{Adv(C7)} \arrow[d, dash, "\text{(IC1)}"] \\ \leftarrow\text{Intro(C7)} \end{tikzcd}$$

<u>**Figure 3.7:** A seach space for example 3.4 with an update as top clause</u>

$$\begin{tikzcd} \underline{\text{Supervised-by(x Prof-Smith)}} \leftarrow \text{Grad(x) and Takes(x C6)} \arrow[d, dash, "\text{(IC2)}"] \\ \leftarrow\text{Grad(x) and Takes(x C6) and } \underline{\text{Sponsored-by(x BT)}} \arrow[d, dash, "\text{(6)}"] \\ \leftarrow\underline{\text{Grad(John)}}\text{ and Takes(John C6)} \end{tikzcd}$$

<u>**Figure 3.8:** A search space for example 3.4 with an update as top clause</u> □

Note that we are using the term "forward reasoning" rather loosely in this thesis. Strictly speaking, forward (or bottom-up) reasoning uses facts and non-atomic rules to derive new facts. For example, given a fact

A	and a rule
B ←A,	we derive the fact
B	by forward reasoning.

Backward (or top-down) reasoning uses denials and rules to derive new denials. For example, given a denial

←A and B	and a rule
B←C and D,	we derive the denial
←A and C and D	by backward reasoning.

Finally, middle-out reasoning uses non-atomic rules to derive new rules. For example, given two rules

A←B	and
B←C,	we derive the rule
A←C	by middle-out reasoning.

A discussion of these three forms of reasoning can be found in Kowalski [1979].

We use the expression "forward reasoning from the updates" to emphasize the use of the updates as top clauses. The actual reasoning, strictly speaking, can be any combination of forward, backward or middle-out according to the above criteria. Nevertheless, even then the Consistency method reasons forward in a more general sense of deriving consequences from asserted information.

3.3 Comparison With SL

SL (Kowalski and Kuehner [1971]) is a linear proof procedure for non-Horn clauses. It allows any non-Horn clause as top clause, and insists on a last-in-first-out literal selection strategy. The name SL stands for Linear resolution with Selection function.

A derivation in any linear proof procedure is a (possibly infinite) sequence of clauses S_0, S_1, S_2, ..., such that S_0 is the top clause, and for all i, $i \geq 0$, S_{i+1} is obtained by the resolution of S_i on its selected literal

either (a) with an input clause,

or (b) with an S_j, for some j, $j<i$. The resolution in (b) is called ancestor resolution.

In SL, as well as (a) and (b), above, two other operations, called factoring and truncation, can be used to obtain the (i+1)th clause in the derivation. These operations do not, however, play a part in the cases we are considering in this chapter, and we therefore ignore them.

SL is more general than our proof procedure, because it caters for non-Horn clauses. It, however, has one major disadvantage compared to our proof procedure, and that is its insistence on a last-in-first-out literal selection strategy. Suppose S_{i+1} is a clause in an SL derivation, obtained by the resolution of S_i, on its selected literal, with an input clause B_i. Then the selection function must choose from S_{i+1} a literal which is contributed by B_i in preference to any contributed by S_i. Furthermore, in all subsequent steps no literal contributed by S_i can be selected until all those contributed by B_i are resolved away.

This inflexibility in literal selection affects the efficiency of SL. In general, in the cases that we are considering in this chapter, we can simulate SL by our proof procedure. Therefore our procedure can be at least as efficient as SL. However, there are cases where our procedure allows much greater efficiency than SL. Example 3.4 illustrates such a case. The SL search space corresponding to figure 3.6 with (IC3) as top clause is shown below. To save space we have used "," to denote the logical connective "and".

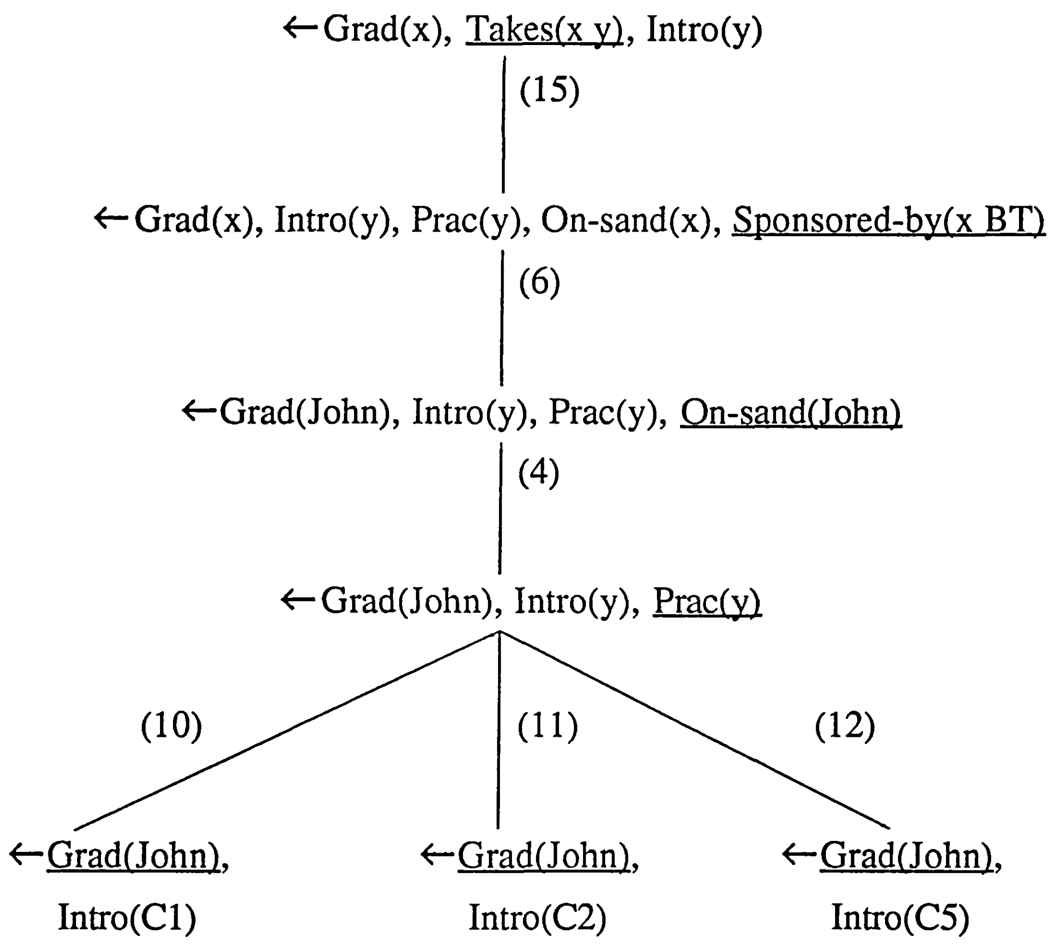

←Grad(x), Takes(x y), Intro(y)
(15)
←Grad(x), Intro(y), Prac(y), On-sand(x), Sponsored-by(x BT)
(6)
←Grad(John), Intro(y), Prac(y), On-sand(John)
(4)
←Grad(John), Intro(y), Prac(y)
(10)
(11)
(12)
←Grad(John), Intro(C1)
←Grad(John), Intro(C2)
←Grad(John), Intro(C5)

Figure 3.9: An SL search space for example 3.4 with an update as top clause

This search space is bigger than the corresponding one in figure 3.6, because in figure 3.9 the literal "Grad(John)" cannot be selected until all the literals contributed by input clause (15) in the first resolution step are resolved away.

CHAPTER 4

THE CONSISTENCY METHOD: SIMPLIFIED CASE 2

In this chapter we extend our method to deal with deductive databases, integrity constraints that are in the form of denials, with or without negative conditions, and transactions that add, delete or modify database rules, or add or delete constraints. We do not, however, cater for implicit deletions here, although we discuss them briefly in Subsection 4.2.5. In the next chapter, we extend our method to deal with implicit deletions as well.

In general, a knowledge assimilation system contains at least two stages. Suppose a user requests a change to the logical contents of the database. Stage 1 determines what transactions consisting of explicit (physical) insertions and deletions would satiafy the user's request. There could be several such transactions. Once one of them has been chosen, stage 2 then determines if the database that results from performing the transaction satisfies the integrity constraints.

For example, suppose we have a database

D: $A \leftarrow B$
$B \leftarrow C$
$E \leftarrow F$
$G \leftarrow F$
F

and a user requests the logical addition of A and the logical deletion of

E. The user's request can be interpreted as requesting that the database be updated so that A becomes a consequence of the completion of the database, but E stops being a consequence of the completion. In general, several alternative transactions will achieve the desired effect. The choice can be made autonomously by the database management system or by interaction with the user.

Two possible transactions of physical updates that can be determined by stage 1 are, for example,

{insert A, delete F} or

{insert C, delete $E \leftarrow F$}.

Given one of these, stage 2 attempts to determine if the new database satisfies the integrity constraints.

This thesis only deals with this second stage, and does not address the first stage at all. The term "implicit deletion" is not concerned with the first stage. It is concerned with deletions that occur implicitly as consequences of physical insertions and deletions. For example, G is "implicitly deleted" in the above database as a consequence of the first transaction. We describe "implicit deletions" in more detail and exemplify them further in Subsection 4.2.5.

In this thesis, by an "update" we mean a physical insertion or a physical deletion determined by stage 1, and by a "transaction" we mean a set of such updates.

To deal with the simplified case of this chapter we extend the proof procedure described earlier in 3.1.2 to cater for negative conditions and explicit deletions. This extended proof procedure can probably be most easily understood as an extension of the SLDNF proof procedure (Lloyd [1987]), which, in turn, is an extension of SLD. SLDNF is the underlying proof procedure of Prolog, and extends SLD by incorporating the negation as failure rule. Similar to SLD, SLDNF only allows denials as top clauses.

In general, when Comp(DT) is consistent, to check if Comp(DT)$\cup$IT is consistent, the integrity constraints in IT can be used as top clauses with the SLDNF proof procedure. Suppose the denial (IC) is a constraint in IT. If there is an SLDNF refutation with (IC) as top clause and DT as input set, then (IC) is violated in the database. But if there is an SLDNF finitely failed derivation with (IC) as top clause, and SLDNF is complete for the given case, then (IC) is satisfied. (SLDNF refutation and derivation are described formally shortly.) However, as argued earlier, using the constraints as top clauses can be very inefficient, because it fails to exploit the assumption that the database satisfies its constraints prior to the transaction. To reason forward from the updates, and thus to avoid this inefficiency, our proof procedure extends SLDNF by allowing as top clauses any deductive rules (facts or non-atomic rules), as well as denials. In addition, a top clause can be a negated fact representing an explicit deletion, as we shall see shortly.

This chapter is divided into two sections. In 4.1 we first describe SLDNF, and then extend it to describe our proof procedure. In 4.2 we illustrate the application of our method through a series of examples.

4.1 The Proof Procedure

4.1.1 The SLDNF Proof Procedure

A computation rule for SLDNF is a function from derivations to literals such that it selects a literal from the last formula in the derivation. A computation rule is safe if and only if it does not select negative conditions unless they are ground, i.e. contain no variables.

Let input set S be a set of deductive rules, G a denial and R a safe computation rule. An SLDNF derivation of $S \cup \{G\}$ via R is a (possibly infinite) sequence G_0, G_1, G_2, ..., such that G_0=G, and for all i, i $\geq$0, G_{i+1} is obtained from G_i by one of (a) or (b) as follows:

(a) Let G_i be "$\leftarrow L_1$ and ...and L_n", and suppose R selects a positive condition L_k from G_i. Then G_{i+1} is the resolvent on L_k of G_i and some input clause in S.

Here, by "resolvent" we mean the obvious generalisation of the standard notion of resolvent: Let C be a deductive rule

$A \leftarrow L'_1$ and ... and L'_m

in input set S, such that A and L_k unify with most general unifier (mgu) ϕ. Then by "resolvent" of G_i and C on L_k we mean the formula

$\leftarrow(L_1$ and ... and L_{k-1} and L_{k+1} and ... and L_n and

L'_1 and ... and $L'_m)\phi$.

(Note that this is a generalisation of resolution because the L_i and the

L'_j can be positive (as in ordinary resolution) or negative.)

(b) Let G_i be "$\leftarrow L_1$ and ...and L_n", and suppose R selects from G_i a literal L_k, which is a negated atom "NOT A". An attempt is made to construct a finitely failed SLDNF search space for $S \cup \{\leftarrow A\}$ via some safe computation rule. If the attempt succeeds then G_{i+1} is G_i with the selected literal L_k removed. If the attempt fails finitely then there is no G_{i+1}. This step is the negation as finite failure step in SLDNF.

As in SLD, G is called the <u>top clause</u>. Each G_i, for i>0, is either a denial or the empty clause.

Similarly to the definition of SLD, an <u>SLDNF refutation</u> is an SLDNF derivation that ends at the empty clause, and a goal G <u>succeeds</u> if and only if there is a refutation with G as top clause.

A <u>finitely failed SLDNF derivation</u> is an SLDNF derivation G_0, G_1,..., G_n, $n \geq 0$, such that G_n is not the empty clause, and it is not possible to construct a derivation G_0, G_1,..., G_n, G_{n+1}. That is the selected literal of G_n is either a negative condition "NOT A" and "$\leftarrow$A" succeeds, or the selected literal is a positive condition and G_n has no resolvent on this literal with any of the deductive rules in the input set.

<u>SLDNF search spaces</u> and <u>finitely failed SLDNF search spaces</u> are defined exactly as their SLD counterparts.

The SLDNF proof procedure has been proved correct for any set of deductive rules and goals (Clark [1978]), and complete for certain restricted cases (Clark [1978], Jaffar, Lassez and Lloyd [1983], Barbuti and Martelli [1986], Kunen [1987 and 1988], Cavedon and Lloyd [1987], and Shepherdson [1988]). Correctness of SLDNF means that if for some safe computation rule R, there is an SLDNF refutation of $S \cup \{G\}$ via R, then $Comp(S) \cup \{G\}$ is inconsistent. Completeness, in those cases where it applies, means that if $Comp(S) \cup \{G\}$ is inconsistent, then for all safe computation rules R there is an SLDNF refutation of $S \cup \{G\}$ via R. (This is an oversimplification of the completeness result for SLDNF. We will give the exact result in Chapter 8.)

By correctness of SLDNF, if a goal "$\leftarrow W$" succeeds from input set S then "$\exists x_1 ... \exists x_n W$" is a logical consequence of Comp(S), where $x_1, ..., x_n$ are all the variables occurring in W.

Example 4.1:

Suppose we have the following database, transaction and integrity constraint:

D:

(1) $P(x) \leftarrow R(x)$

(2) $Q(B) \leftarrow S(B\ x)$ and NOT $T(x)$

(3) R(A)

(4) R(B)

(5) Q(A)

(6) S(B A)

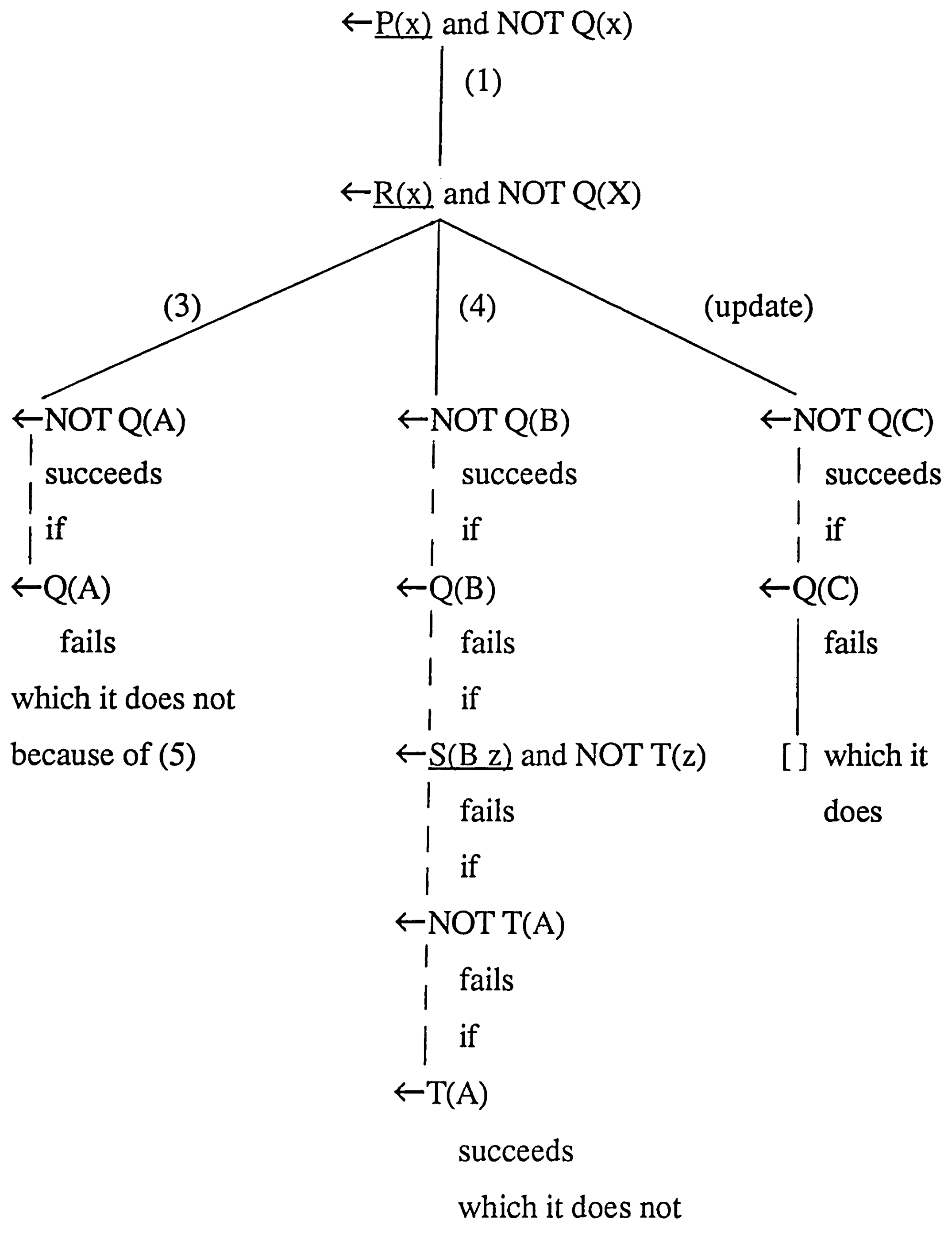
I:

(IC) $\leftarrow$P(x) and NOT Q(x)

T:

Insert R(C).

Then the following is an SLDNF search space for D$T\cup$I. Dotted vertical lines denote subsidiary computations for the negation as failure steps.



<u>**Figure 4.1:** An SLDNF search space for example 4.1 with the constraint as top clause</u>

The search space consists of three derivations, two of which fail finitely. The third which is a refutation shows that Comp(DT)$\cup$I is logically inconsistent, and that therefore the transaction violates the integrity constraint.

Notice that the update contributes only to the rightmost derivation of the search space. The other two derivations only use clauses from $D \cup I$, that is the old database and the constraint. Thus these two derivations are also derivations for $D \cup I$. In effect, they redundantly investigate instances of the constraint that are not affected by the transaction. It is to avoid such redundancies that the Consistency method uses the updates as top clauses, thus concentrating only on what is affected by the transaction, and ignoring what remains unchanged.

It is instructive to compare the search space in figure 4.1 with the search space for proving that (IC) is a theorem of the completion of DT. To prove the theoremhood of (IC) we have to negate it and use it as top clause for an attempted SLDNF refutaton. The negation of (IC), however, is

$\exists x$ [P(x) and NOT Q(x)],

which is not in an appropriate form for SLDNF. We can overcome this problem by defining a new relation "Constraint-satisfied" as follows:

(1) Constraint-satisfied $\leftarrow$ NOT $\exists x$ [P(x) and NOT Q(x)],

and then using the goal

$\leftarrow$Constraint-satisfied

as top clause.

The definition of "Constraint-satisfied" can be transformed into deductive rules, using transformation steps described in Lloyd and Topor [1984], yielding the following rules that would be considered as part of the database, both before and after the transaction:

(2) Constraint-satisfied $\leftarrow$ NOT AUX

(3) AUX $\leftarrow$ P(x) and NOT Q(x),

where "AUX" is a new nullary relation.

Now using "$\leftarrow$Constraint-satisfied" as top clause we obtain the following SLDNF search space.

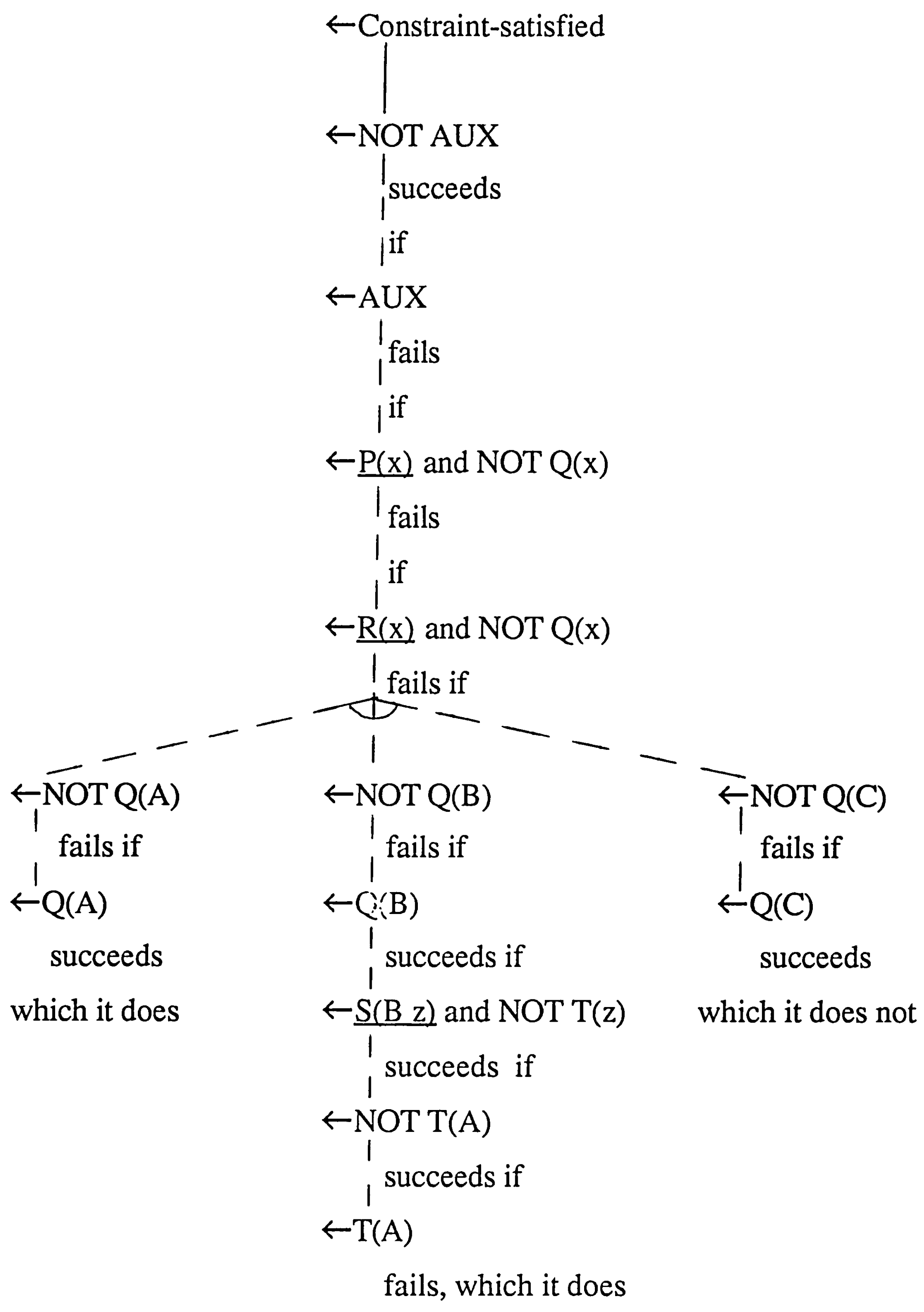


Figure 4.2: An SLDNF search space for example 4.1 attempting to prove the theoremhood of the constraint

The search space fails finitely. Therefore, by Theorem 18.6 of Lloyd [1987], the negation of (IC) is a theorem of Comp(DT). So (IC) is not a theorem of Comp(DT), since Comp(DT) is consistent, because DT is stratified.

Notice that except for the first three steps and the success/failure labels figure 4.2 is identical to figure 4.1. Thus although the two views of integrity satisfaction are conceptually different, operationally, the work involved in checking consistency, with the constraints as top clauses, is virtually the same as the work involved in attempting to prove theoremhood of the constraints. $\square$

In general we can prove the following:

Proposition 4.1:

Let S be a deductive database, and let "$\leftarrow$C" be an integrity constraint on S, where C is a conjunction of literals. Then:

(i) there exists an SLDNF refutation of $S \cup \{\leftarrow C\}$, which shows that the constraint is inconsistent with Comp(S), if and only if the attempt to show by SLDNF that the constraint "$\leftarrow$C" is a theorem fails finitely, and

(ii) there exists an SLDNF proof of the integrity constraint "$\leftarrow$C" as a theorem if and only if the attempt to show that the constraint is inconsistent fails finitely by SLDNF.

(This Proposition is part of problem 9, on page 138 of Lloyd [1987].)

Proof:

The proof is a direct generalisation of the preceding example.

(i) The "only if" half:
Suppose there is an SLDNF refutation of $S \cup \{\leftarrow C\}$. To attempt to prove the theoremhood of "$\leftarrow C$", as in example 4.1, we introduce a new relation, "Theorem", say, with the following definition:

$$\text{Theorem} \leftarrow \text{NOT } \exists x^* \, C,$$

where x^* is a vector of all the variables that occur in C. The definition of "Theorem" can be transformed into the following deductive rules, as in example 4.1:

(1) $\text{Theorem} \leftarrow \text{NOT AUX}$

(2) $\text{AUX} \leftarrow C$,

where "AUX" is a new nullary relation.

Now using the goal "$\leftarrow$Theorem" as top clause, we obtain the following SLDNF search space. We show an incomplete search space because we do not need to consider the details of the success of "$\leftarrow C$".

$$\begin{tikzcd}[row sep=large]
\leftarrow \text{Theorem} \arrow[d, dash] \\
\leftarrow \text{NOT AUX} \arrow[d, dash, "\text{succeeds if}"] \\
\leftarrow \text{AUX} \arrow[d, dash, "\text{fails if}"] \\
\leftarrow C
\end{tikzcd}$$

fails

which it does not,

because there is an SLDNF refutation of $S \cup \{\leftarrow C\}$.

Figure 4.3: A finitely failed SLDNF search space

Thus the attempt to show the theoremhood of the constraint by SLDNF fails finitely.

The "if" half of (i) follows by a similar argument.

(ii) The "if" half:
Suppose there is an SLDNF finitely failed search space for $S \cup \{\leftarrow C\}$. Then the SLDNF search space with the goal "$\leftarrow$Theorem" as top clause is as follows. We show an incomplete search space because we do not need to consider the details of the failure of "$\leftarrow$C".

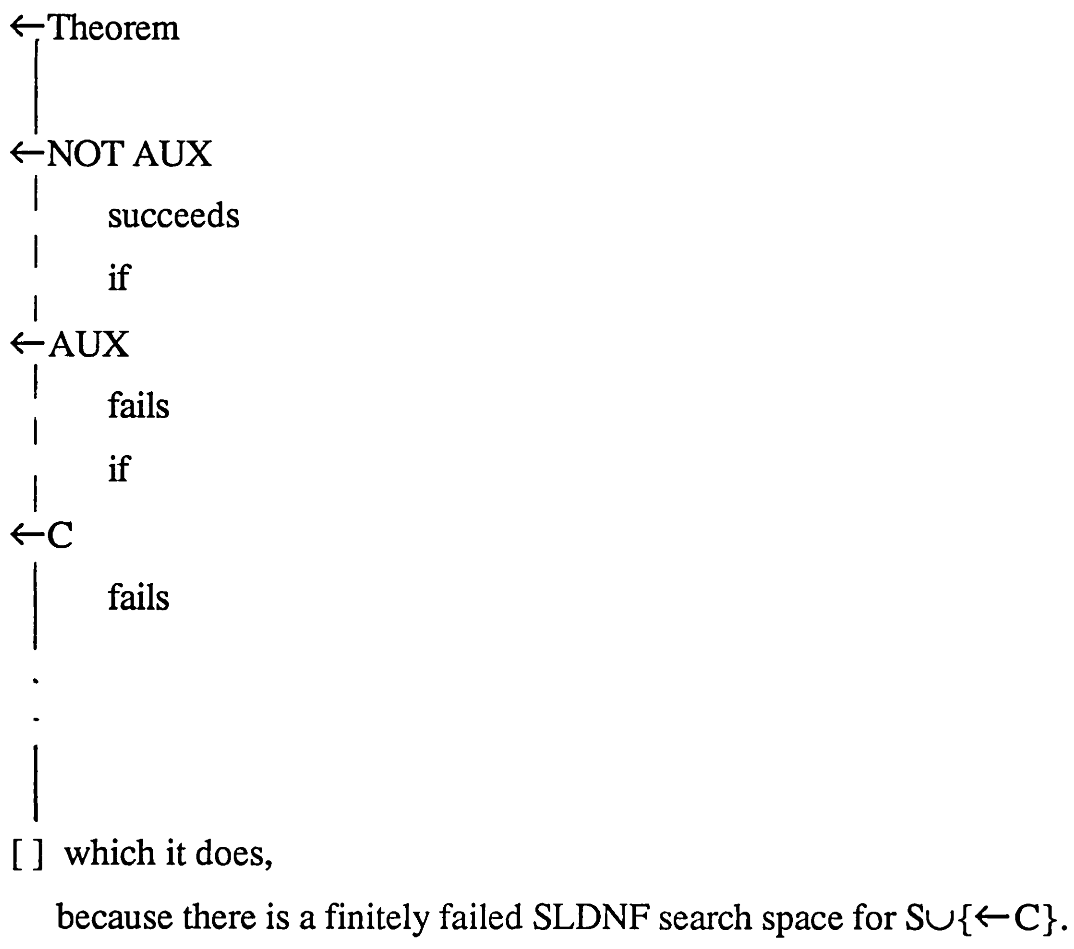

←Theorem
←NOT AUX
succeeds
if
←AUX
fails
if
←C
fails
[ ] which it does,
because there is a finitely failed SLDNF search space for S∪{←C}.

Figure 4.4: An SLDNF search space

Thus there is an SLDNF proof of the theoremhood of the constraint. (Figure 4.4 shows that "Theorem" is a theorem of Comp(S∪{(1),(2)}). Then by Lemmas 3 and 1 of Lloyd and Topor [1984], (IC) is a theorem of Comp(S).)

The "only if" half of (ii) follows by a similar argument. □

Proposition 4.2:

If Comp(S) is consistent and there is an SLDNF finitely failed search space for $S \cup \{\leftarrow C\}$, then $\text{Comp}(S) \cup \{\leftarrow C\}$ is consistent, and therefore the constraint "$\leftarrow C$" is satisfied in S according to the consistency view.

Proof:

The proof is trivial. By Clark's correctness result of negation as failure (Clark [1978], also Theorem 15.4 of Lloyd [1987]), "$\leftarrow C$" is a theorem of Comp(S). The Proposition follows, since any theorem of a consistent theory is itself consistent with the theory. □

As we shall see shortly, our proof procedure is identical to SLDNF when the top clause is a denial.

4.1.2 Our Proof Procedure

To deal with the general case considered in this chapter the Consistency method proof procedure that was described in 3.1.2 needs to be extended in three ways. It needs to incorporate the negation as failure step for solving negative subgoals, it needs to allow reasoning forward from updates that are deletions, and finally it needs to incorporate additional inference rules for implicit deletions (to be described later).

To motivate our proof procedure, we first describe our scheme for associating top clauses with updates, particularly with updates that are deletions.

We assume that the transaction consists of a set, al, and a set, dl, of deductive rules (and a set, ml, to be described later, specifying modifications to existing rules) such that the rules in al are to be explicitly added to the database, and such that all rules occurring explicitly in the database that are variants of a rule in dl are to be explicitly deleted from the database. (A clause C is a <u>variant</u> of a clause C' if C is identical to C' up to a renaming of variables.)

A fact F is <u>explicit</u> in a database D if F has an explicit occurrence in D. F is <u>implicit</u> in D if it is a logical consequence of Comp(D'), where D'=D-{F}.

An update of deleting a fact which is both explicit and implicit deletes the explicit occurrence only.

We assume further that al does not contain any variants of rules in dl. Thus, for example, a transaction cannot include the addition of a rule "P(x) ←Q(x y)" and the deletion of a rule "P(z) ←Q(z x)".

If an update is a deletion of a fact A, which is not implicit in the updated database, then we use the negated fact "NOT A" as top clause, and thus in effect reason forward from the fact that A is not provable in the updated database (or from the fact that the negation of A is a logical consequence of the completion of the updated database). "NOT A" is the <u>top clause associated with the update</u>.

If an update is a deletion of a non-atomic deductive rule, then we first determine what instances of the conclusion of this rule are deleted as a result of the deletion of the rule, and then select as top clauses the negation of these deleted instances. These negated facts are the <u>top clauses associated with the update</u>.

If an update is a deletion of an integrity constraint, then it cannot possibly cause an inconsistency, and therefore we do not need a top clause associated with it.

We treat updates that are additions exactly as described in the last chapter. That is if an update is the addition of a deductive rule or an integrity constraint R, then we use R as top clause as it stands. R is the <u>top clause associated with the update</u>. Of course, if R is a constraint and it is not in the form of a denial, then it first has to be transformed into the required form as described in Chapter 2.

Finally, updates that modify existing rules are treated as a combination of additions and deletions.

Given this scheme for associating top clauses with updates, the proof procedure has to allow as top clause any deductive rule, denial or negated fact. In the rest of this thesis, for convenience, we use the term "clause" loosely to refer to any of these types of formulae.

The resolution and negation as failure steps cater for reasoning forward from deductive rules and denials. To deal with reasoning forward from negated facts we incorporate in the proof procedure an "extended" resolution step that allows the resolution of a negated fact

$\underline{\text{NOT A'}}$

and a rule

$B \leftarrow \underline{\text{NOT A}}$ and C

on the underlined literals if the atoms A and A' unify. ϕ is the most general unifier (mgu) of "NOT A" and "NOT A' " if and only if it is the mgu of A and A'. The resolvent is then

$(B \leftarrow C)\phi$.

This step is an "extended" resolution step only in the sense that it is not incorporated in the SLDNF proof procedure. The step, however, is, in fact, a standard resolution step, as shown by Proposition 8.2 in Chapter 8.

Our proof procedure, without inference rules for implicit deletions, is described as follows.

A computation rule, or literal selection strategy, in our proof procedure is a function from derivations to literals such that it selects a literal from the last clause in the derivation. Safe computation rules are defined as in SLDNF.

Let input set S be a set of deductive rules and denials. Let C_0 be a clause in S or a ground negated fact. C_0 is allowed to be a ground negated fact "NOT A" if $S \cup \{\text{"} \leftarrow A\text{"}\}$ has a finitely failed SLDNF search space. (This is formalised by rule (C2) in Chapter 5.) Let R be a safe computation rule. A derivation for S via R with top clause C_0 is a (possibly infinite) sequence C_0, C_1, C_2,..., such that for all i, $i \geq 0$, C_{i+1} is obtained from C_i by one of (a) or (b) as follows:

(a) Suppose R selects from C_i a literal L which is not a negative condition of C_i. Then C_{i+1} is the resolvent on L of C_i and some input clause in S. We allow both the standard and the extended resolution steps.

(b) Suppose R selects from C_i a negative condition "NOT A". An attempt is made to construct a finitely failed search space with "$\leftarrow$A" as top clause and S as input set. If the attempt succeeds then C_{i+1} is C_i with the selected literal "NOT A" removed. If the attempt fails finitely then there is no C_{i+1}. Either SLDNF or our proof procedure can be used for this subsidiary computation. The two are identical whenever the top clause is a denial. (In such a case all clauses C_i in the derivation are also denials.)

Each C_i, for i>0, is thus a deductive rule or a denial, or the empty clause.

Notes:

(1) Steps (a) and (b) above are extensions of steps (a) and (b), respectively, in the description of SLDNF derivation.

(2) When a condition "NOT A" is selected the only operation that can be performed is the negation as failure step. The resolution step is not applicable here, because the input set does not contain any clauses with

negated conclusions. (Note that a top clause which is a negated fact is not considered to be part of the input set.) On the other hand, when a conclusion "NOT A" is selected the only operation that can be performed is (extended) resolution with an input clause.

In this simplified version of the proof procedure, only the top clause can have such a negated conclusion. Therefore the extended resolution step can only be applied to obtain the second clause in a derivation from the top clause. (In the general case, in the presence of implicit deletions, the extended resolution step can be applied at later stages of a derivation.)

(3) If a clause consists entirely of non-ground negative conditions, then none of its literals can be selected by a safe computation rule. This situation is called floundering (Clark [1978]), and applies to SLDNF, as well as to our proof procedure. The range-restriction condition on the database and the goals prevents floundering (Lloyd and Topor [1986]).

The concepts of refutation, success of a goal, search space and finitely failed search space for our proof procedure are defined similarly to those for SLDNF. Finitely failed derivations are defined as obvious extensions of such derivations in SLDNF. Thus a derivation C_0, C_1,..., C_n is a finitely failed one in our proof procedure if and only if C_n is not the empty clause, and it is not possible to construct a derivation C_0, C_1,..., C_n, C_{n+1} by an application of step (a) or (b) of our proof procedure.

In Chapter 8 we prove that this proof procedure is correct in general and complete in certain special cases. Correctness means that if for some safe computation rule R and for some clause C associated with an update there is a refutation of DT$\cup$IT via R with C as top clause then the transaction violates the constraints. Completeness means that if the transaction violates the constraints then for every safe computation rule R there is a refutation of DT$\cup$IT via R with a top clause associated with one of the updates in the transaction. Thus, in those cases where the proof procedure is complete, if for every update C in T there is a finitely failed search space for DT$\cup$IT with C as top clause via some safe computation rule, then Comp(DT)$\cup$IT is consistent, and therefore the transaction satisfies the constraints.

4.2 Examples

4.2.1 Updates That Are Additions

Example 4.2:

Consider example 4.1, and assume (correctly) that D satisfies I prior to the transaction. To check if DT satisfies I we use the update as top clause with our proof procedure and with DT$\cup$I as input set. We obtain the following search space showing that the transaction violates the integrity constraint.

$$\begin{tikzcd}
R(C) \arrow[d, dash, "(1)"] \\
P(C) \arrow[d, dash, "(IC)"] \\
\leftarrow \text{NOT } Q(C) \arrow[d, dash, "\text{succeeds if}"] \\
\leftarrow Q(C) \arrow[d, dash, "\text{fails}"] \\
[\ ] \quad \text{which it does}
\end{tikzcd}$$

Figure 4.5: A search space for example 4.2 (and example 4.1) with the update as top clause

Compare this search space with the one in figure 4.1. Both show a violation of the constraint. But the one in figure 4.1 is much bigger because it, in effect, irrelevantly explores two instantiations of the integrity constraint, in addition to the one explored in figure 4.5. The selection of the update as top clause avoids the inefficiency illustrated in figure 4.1. □

The next example also concerns a case of adding a fact to a database. This example illustrates why in our method integrity constraints have to be rewritten as denials.

Example 4.3:

In this example "Acc(x y)" expresses that x has access to machine y.

D:

(1) Rank (John Lect)

(2) Rank(Mary Prof)

(3) Proj(John P1)

(4) Proj(Mary P2)

(5) Acc(x VAX) $\leftarrow$ Proj(x P1)

I:

(IC) Rank(x Lect) $\leftarrow$ Proj(x P1)

T:

Insert Proj(Tom P1).

The constraint states that anyone who works on project P1 must have rank lecturer.

Assume (correctly) that D satisfies the constraint. The integrity constraint is not in the form required for our integrity checking method. If we use the constraint as it stands, and select the update as top clause we obtain the following search space.

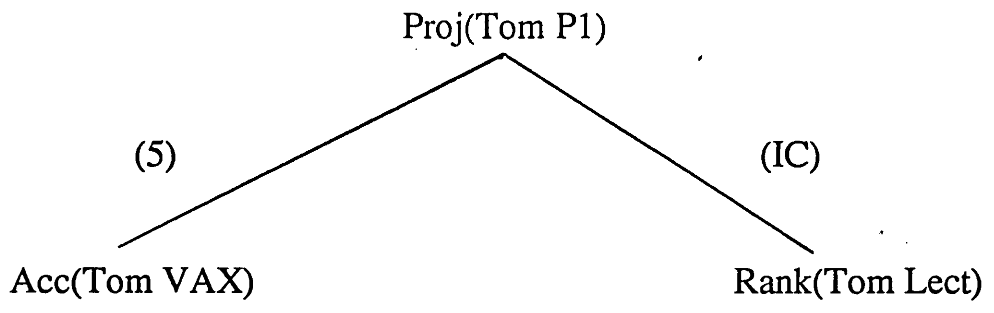


Figure 4.6: A search space for example 4.3 with the update as top clause

This search space fails to demonstrate any inconsistency. The completion of the updated database, however, is inconsistent with the integrity constraint because together they logically imply both

"Rank(Tom Lect)" and "NOT Rank(Tom Lect)".

To simulate reasoning with the completion of the database we rewrite the constaint in the form

$\leftarrow$Proj(x P1) and NOT Rank(x Lect)

before applying our proof procedure.

The following search space, still with the update as top clause, but using the rewritten form of (IC), demonstrates that the updated database violates the integrity constraint.

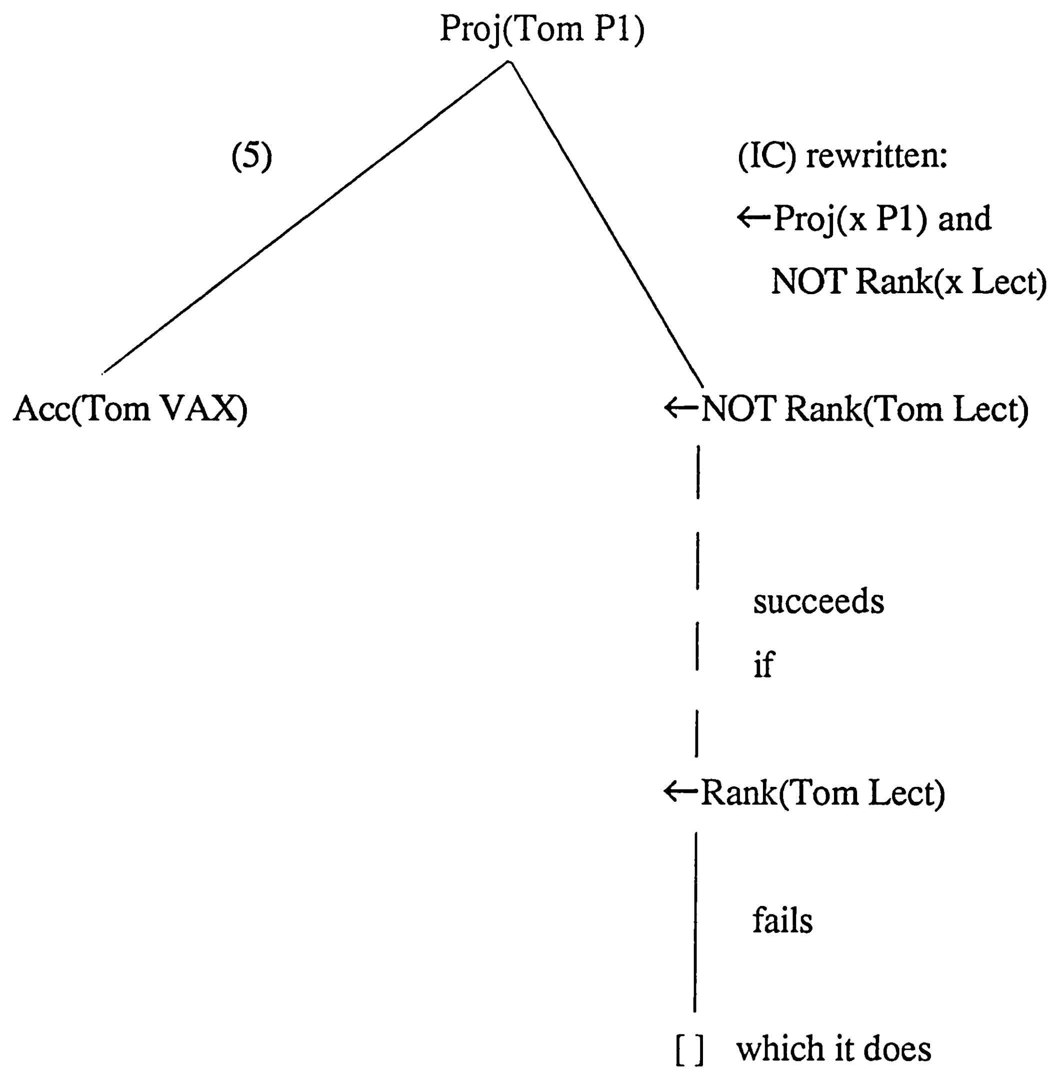


Figure 4.7: A search space for example 4.3 with the update as top clause, using the rewritten form of the integrity constraint

The search space consists of two derivations. The one on the left, where the update is resolved with a deductive rule in the database, is a finitely failed derivation. The one on the right, where the update is resolved with the rewritten form of (IC), is a refutation and shows that the constraint is violated. □

The insertion of non-atomic deductive rules and of integrity constraints is treated exactly as the addition of facts.

Example 4.3 illustrates how we treat the integrity constraints differently from the rules in the database. There are other approaches to treating integrity constraints. Reiter [1988], for example, uses the modal language of Levesque [1981] to distinguish between database rules and constraints. In his formalisation the constraint in example 4.3 is expressed by the modal formula

K Rank(x Lect) ← K Proj(x P1),

where K is a modal operator standing for "knows". Thus the constraint states that for all x, if it is known that x works on project P1, then it is also known that x has rank lecturer. This constraint is satisfied in DT if and only if it is true in DT in the modal language of Levesque.

Eshghi and Kowalski [1988] propose a metalogical variant of both Reiter's and our approaches. Under their metalogical interpretation the above constraint becomes

Demo(d Rank(x Lect)) ← Demo(d Proj(x P1)),

which states that for all database states d, if "Proj(x P1)" is provable from d, for some x, then "Rank(x Lect)" is also provable from d. (For simplicity we have not distinguished between sentences and their names, in the above formula.)

Eshghi and Kowalski [1988] show that, under certain assumptions, their formalisation of the constraints is equivalent to our treatment of them as denials.

Noel [1988] and Small [1988] have also proposed similar interpretations of integrity constraints.

4.2.2 Updates That Are Deletions

To illustrate our handling of deletions, we first consider a simple example of deleting a fact from a relational database.

Example 4.4:

D:

(1) Rank(John Lect)

(2) Rank(Mary Lect)

(3) Rank(Tom Prof)

(4) Proj(John P1)

(5) Proj(Mary P1)

(6) Proj(Tom P2)

I:

(IC) Rank(x Lect) $\leftarrow$ Proj(x P1)

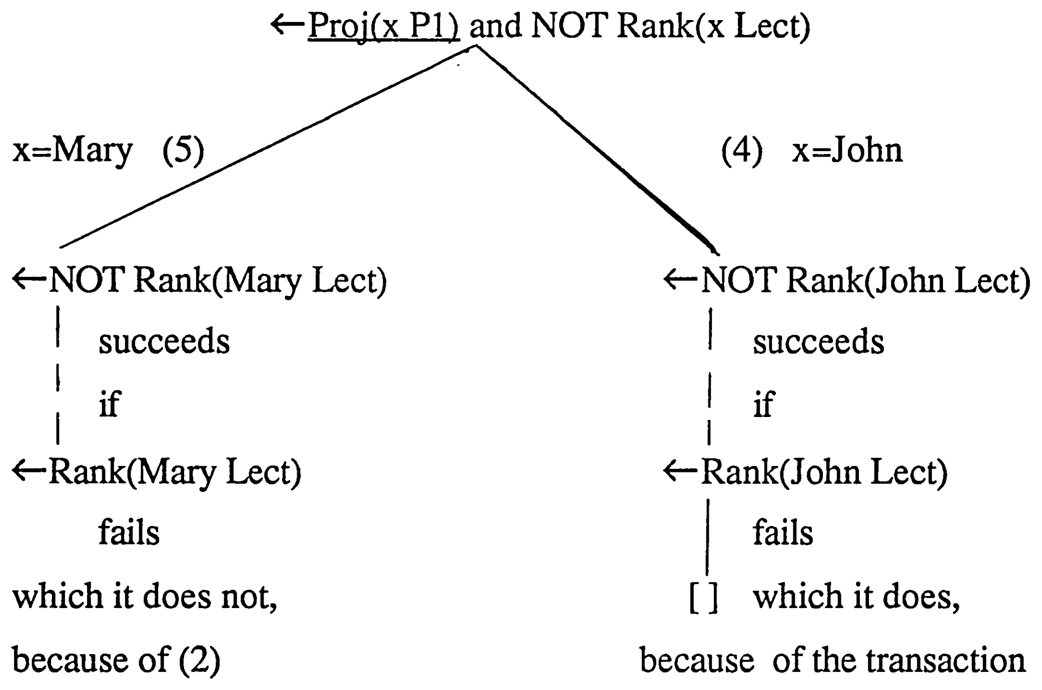
which is rewritten as

$\leftarrow$ Proj(x P1) and NOT Rank(x Lect).

T:

Delete Rank(John Lect).

Before applying our method to this example, we show, below, the search space with the integrity constraint as top clause and the updated database, that is D-{Rank(John Lect)}, as input set, where "-" denotes set difference. This search space shows clearly that the integrity constraint is violated in the updated database because the proof of "Rank(John Lect)" fails. The search space is obtained both by our proof procedure and by SLDNF.



<u>**Figure 4.8:** A search space for example 4.4 with the constraint as top clause</u>

In our approach, to achieve the same effect by reasoning forward from the update, we use as top clause the negated fact

"NOT Rank(John Lect)",

which represents the update. The search space below, with this negated fact as top clause, shows that the update violates the integrity constraint. Note that the updated database, and therefore the input set, do not explicitly contain the negated fact "NOT Rank(John Lect)".

$$\begin{tikzcd}
\text{NOT Rank(John Lect)} \arrow[d, dash, "\text{(IC) rewritten}"] \\
\leftarrow\text{Proj(John P1)} \arrow[d, dash, "(4)"] \\
[\ ]
\end{tikzcd}$$

Figure 4.9: A search space for example 4.4 with the update as top clause

The first step in the search space is an extended resolution step between the negated fact

NOT Rank(John Lect)

and the integrity constraint

←Proj(x P1) and NOT Rank(x Lect)

on the underlined literals.

Note that in this example the database is relational and therefore the fact that is to be deleted by the transaction can only be explicit in the database. In general, however, in a deductive database, facts can be implicit as well as explicit. In such a database, given a transaction including an update of deleting a fact A we need to ensure that A is not provable in the updated database (or, more precisely, that "NOT A" is a theorem of the completion of the updated database), before reasoning forward from "NOT A". (We can use the SLDNF proof procedure to check if A is provable in the updated database.) If A is still provable in the updated database, then the update, which only deletes the explicit occurrence of A, does not alter the logical content of the database, and there is, therefore, no need to consider the update as a top clause for integrity checking. Moreover, in this case, it would be incorrect to reason forward from "NOT A", as the following example illustrates:

```
D:  P←NOT Q
    Q←R
    R
    Q
```

Suppose the update is the deletion of the explicit occurrence of Q. Choosing "NOT Q" as top clause in this case would allow us to derive P which is incorrect, because P is not a consequence of the completion of the updated database.

Compare the search spaces in figures 4.8 and 4.9. Intuitively, 4.9 corresponds to the right branch of 4.8, showing a violation of integrity because "Proj(John P1)" is provable, but "Rank(John Lect)" is not. The left branch of 4.8 is redundant for integrity checking, because it considers an instantiation of the constraint which is not affected by the transaction. □

Because of the possible presence of negative conditions in the deductive rules, deletion of facts can implicitly add new facts to the database. Using the extended resolution rule to reason forward from negated conclusions is also crucial for dealing with such cases, as illustrated in the following example.

Example 4.5:

In this example "Teaches(x y)" means x teaches course y.

D:

(1) Teaches(John Databases)

(2) Rank(John Lect)

(3) Rank(Mary Lect)

(4) Proj(John P1)

(5) Proj(Mary P1)

(6) Academic-visitor(x) ←
Teaches(x Databases) and
NOT Rank(x Lect)

I:

(IC) ← Proj(x P1) and Academic-visitor(x)

T:

Delete Rank(John Lect)

Here the update leads to an inconsistency. This is because the deletion of a fact results in the addition of a fact, which violates the integrity constraint, as shown in the following search space.

$$\begin{tikzcd}
\text{NOT Rank(John Lect)} \arrow[d, dash, "(6)"] \\
\text{Academic-visitor(John)} \leftarrow \underline{\text{Teaches(John Databases)}} \arrow[d, dash, "(1)"] \\
\text{Academic-visitor(John)} \arrow[d, dash, "(IC)"] \\
\leftarrow \text{Proj(John P1)} \arrow[d, dash, "(4)"] \\
[\,]
\end{tikzcd}$$

Figure 4.10: A search space for example 4.5 with the update as top clause □

4.2.3 Transactions With Multiple Updates

The principles are as described in 3.2.4. Thus each update in the transaction is a candidate top clause. The input set in each case consists of the updated database and the updated set of integrity constraints. A refutation implies violation of constraints. On the other hand, if all the updates lead to finitely failed search spaces, and our method is complete for the given case, then the transaction satisfies the constraints.

We believe that our method is as complete as SLDNF. We will discuss this point later in Chapter 8. SLDNF has been proved complete for several restricted classes of databases (Clark [1978], Jaffar, Lassez and Lloyd [1983], Barbuti and Martelli [1986], Kunen [1987] and [1988], Cavedon and Lloyd [1987] and Shepherdson [1988]), which include hierarchical databases such as those before and after the update in the following example. Roughly speaking, a database is hierarchical if it contains no recursion. More precisely, a deductive database D is hierarchical if and only if there is a mapping M from the predicate symbols that occur in D to the natural numbers, such that for every rule "Head←Conditions" in D

M(P) > M(Q) if P is the predicate symbol that occurs in the Head, and Q is a predicate symbol that occurs in the Conditions.

Thus a hierarchical database is also stratified, but not vice versa. Hierarchical databases are less general than stratified ones because hierarchical databases allow no recursion at all. An alternative definition of hierarchical databases can be found in Clark [1978].

Example 4.6:

D:

(1) Employed(Tom)

(2) Self-employed(Tom)

(3) Lecturer(Dick)

(4) Lecturer(Harry)

(5) Lecturer(Bill)

(6) Eligible-for-state-pension(Dick)

(7) Eligible-for-state-pension(Harry)

(8) Eligible-for-state-pension(Bill)

I:

(IC) Eligible-for-state-pension(x) ← Lecturer(x)

which is rewritten as

←Lecturer(x) and NOT Eligible-for-state-pension(x)

T:

{Insert (9) Lecturer(Tom)

(10) Eligible-for-state-pension(x) ←Employed(x) and NOT Self-employed(x)

Delete Self-employed(Tom)}

Assume (correctly) that D satisfies the constraint. The transaction consists of three updates. To check if DT satisfies the constraint, each of the three updates must be considered as top clause. In each case the input set consists of the constraint and the updated database, that is $(D\cup\{(9), (10)\})-\{(2)\}$.

It is not difficult to see, intuitively, that the updated database satisfies the constraint. The transaction could violate the constraint only if it added a new lecturer who was not eligible for state pension, or if it deleted the eligibility of some continuing lecturer. The second case does not arise because the transaction adds more ways of concluding eligibility rather than deleting the existing ways. The first case does not arise because, although the transaction adds Tom as a lecturer, it also implicitly adds Tom's eligibility for state pension.

Below, we show the three search spaces that result from taking each of the updates in the transaction as top clause. All three search spaces fail finitely. Thus, assuming that our method is complete in this example, we can conclude that the transaction satisfies the constraint. (We have not, however, proved our method complete for this case.)

In the search spaces we have abbreviated the predicate symbol "Eligible-for-state-pension" to "E-f-s-p". Each of the three search spaces consists of a single derivation. The second search space consists only of the top clause, and the third consists of two clauses.

Lecturer(Tom)

(IC) rewritten

←NOT E-f-s-p(Tom)

succeeds

if

←E-f-s-p(Tom)

fails

if

←Employed(Tom) and NOT Self-employed(Tom)

fails

if

←NOT Self-employed(Tom)

fails

if

←Self-employed(Tom)

succeeds

which it does not.

Figure 4.11: A search space for example 4.6 with an update as top clause

$\underline{\text{E-f-s-p(x)}} \leftarrow \text{Employed(x) and}$
$\qquad\qquad\text{NOT Self-employed(x)}$

Figure 4.12: A search space for example 4.6 with an update as top clause

$$\begin{tikzcd}
\text{NOT Self-employed(Tom)} \arrow[d, dash, "(10)"] \\
\underline{\text{E-f-s-p(Tom)}} \leftarrow \text{Employed(Tom)}
\end{tikzcd}$$

Figure 4.13: A search space for example 4.6 with an update as top clause □

4.2.4 Updates That Modify Database Rules

Suppose an update requests the modification of a database rule (fact or non-atomic rule) R to R'. Then the database management system must check to see if a variant of R exists in the database. If it does not, then the management system can either inform the user, or simply ignore the update. If a variant exists, then the update can be treated as two updates, the deletion of the existing variant of R, and the addition of R'. These two updates are then treated as explained already.

Modifying integrity constraints involves more work, because of the transformations that are performed on constraints to convert them into denials and deductive rules. To modify a constraint W to W' we need to trace all the deductive rules and the denial which resulted from the transformations performed on W. This can be done by keeping a record of such information. The thesis will not, however, address this issue. We assume that "modify" updates only request modifications to the database rules.

4.2.5 Updates That Require Additional Inference Rules

In deductive databases the deletion of explicitly present facts can cause the deletion of other implicit facts. Consider the following very simple propositional database, for example:

D: A

$B \leftarrow A.$

Fact B is provable in this database. However, if fact A were deleted B would no longer be provable. The deletion of A would implicitly delete B. To check integrity of deductive databases it is necessary to detect such implicit deletions. This requires the addition of a new inference rule, as illustrated in the following example.

Example 4.7:

In this example "Sup(x y)" expresses that x supports project y, and "Alloc(x y)" expresses that project x is allocated machine y.

D:

Sup(SERC P1)

Sup(BP P2)

Sup(MOD P3)

Alloc(x VAX) ← Sup(SERC x)

Alloc(x IBM) ← Sup(BP x)

Alloc(x SUN) ← Sup(MOD x)

I:

(IC) Alloc(P1 VAX) or Alloc(P1 SUN)

which states that project P1 is allocated either the VAX or the SUN.

(IC) is rewritten as

←NOT Alloc(P1 VAX) and NOT Alloc(P1 SUN).

T:

Delete Sup(SERC P1).

The fact "Sup(SERC P1)" is only explicit in D and thus it is not provable in the updated database.

Intuitively speaking, the deletion of the fact "Sup(SERC P1)" also "deletes" the previously derivable fact "Alloc(P1 VAX)". More formally, "←Alloc(P1 VAX)" is a logical consequence of the completion of the updated database D-{Sup(SERC P1)}. Since "←Alloc(P1 SUN)" is also a logical consequence of this completion, the update violates the integrity constraint.

One way to deduce that "Alloc(P1 VAX)" is deleted from the updated database is to reason as follows:

(R1)

because in DT "NOT Sup(SERC P1)" holds

and we have "Alloc(x VAX)$\leftarrow$Sup(SERC x)"

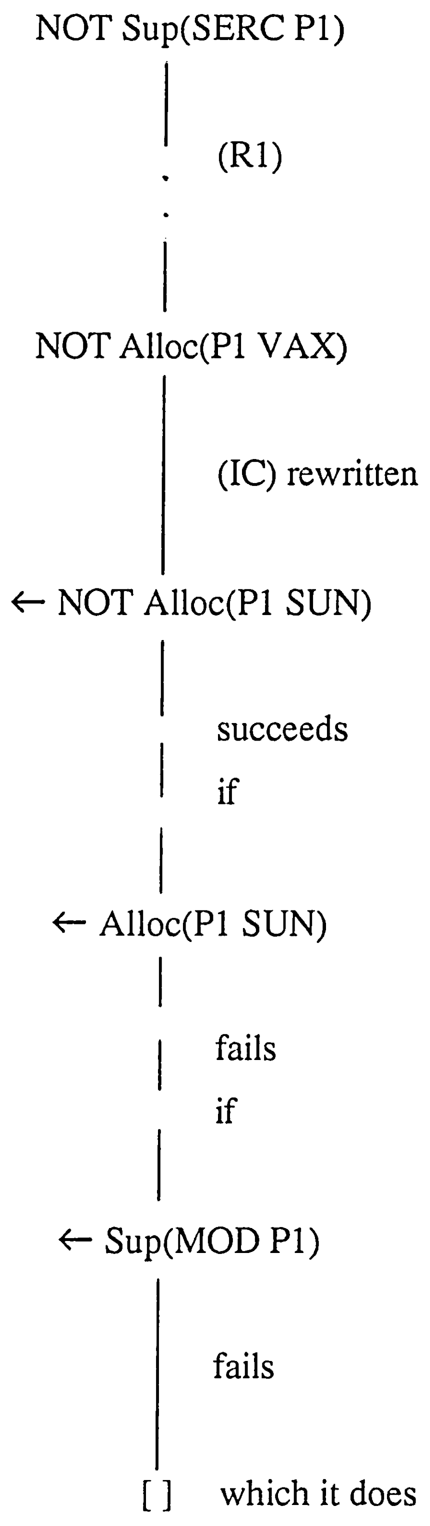
and we have no other way of showing "Alloc(P1 VAX)"

and "Alloc(P1 VAX)" was provable in D

then "Alloc(P1 VAX)" is deleted.

Thus "NOT Alloc(P1 VAX)" holds in DT.

Such a rule, in part, allows us to reason with the completion of the updated database without having to represent the completion of the database explicitly. It can also be thought of as a rule that allows us to reason forward from negated conclusions. We will formalise and generalise the reasoning in (R1), in the next chapter. Assuming for now that we have such a formalisation, the following incomplete refutation shows that the updated database violates the integrity constraint. The refutation is incomplete because we have ignored the details of (R1).



<u>**Figure 4.14:** A refutation for example 4.7 with the update as top clause</u> □

Because of the presence of negative conditions in the deductive rules an addition can also cause implicit deletions. We need another inference rule to cater for such a case, as illustrated in the following example.

Example 4.8:

D:

(1) Overseas-student(x) ← Student(x) and NOT Resident(x UK)

(2) Student(Jim)

(3) Eligible(Jim Brit-Council-award)

I:

(IC) Overseas-student(x) ← Eligible(x Brit-Council-award)

which is rewritten in the form

←Eligible(x Brit-Council-award) and NOT Overseas-student(x)

T:

Insert Resident(Jim UK).

Intuitively speaking, the insertion of the fact "Resident(Jim UK)" "deletes" the previously derivable fact "Overseas-student(Jim)", and thus violates the integrity constraint. To deduce this implicit deletion we need to reason that

(R2)

because in DT "Resident(Jim UK)" holds

    and we have "Overseas-student(x)$\leftarrow$Student(x) and
                                                                    NOT Resident(x UK)"

    and we have no other way of showing "Overseas-student(Jim)"

    and "Overseas-student(Jim)" was provable in D

then "Overseas-student(Jim)" is deleted.

Thus "NOT Overseas-student(Jim)" holds in DT.

In the next chapter we will show how this reasoning can be formalised in general. Assuming that we have the required formalisation, the following incomplete refutation shows that the update violates the integrity constraint. The refutation is incomplete because the details of (R2) are ignored.

$$\begin{tikzcd}
\text{Resident(Jim UK)} \arrow[d, dash, "\text{(R2)}"] \\
\text{NOT Overseas-student(Jim)} \arrow[d, dash, "\text{(IC) rewritten}"] \\
\leftarrow \text{Eligible(Jim Brit-Council-award)} \arrow[d, dash, "\text{(3)}"] \\
[\,]
\end{tikzcd}$$

Figure 4.15: A refutation for example 4.8 with the update as top clause □

This discussion completes the case of the updates that require additional inference rules.

CHAPTER 5

FORMALISATION OF THE CONSISTENCY METHOD IN THE GENERAL CASE

In this chapter we describe the general case of our method. The general proof procedure extends the simplified one described in 4.1.2, by incorporating additional inference rules to deal with implicit deletions. (Note that the simplified procedure in 4.1.2 already caters for implicit additions due to other additions and deletions.)

We present the general method by a logical formalisation of it, using logic as metalanguage. Although our formalisation is intended as an abstract and general description, it is actually runnable in Prolog, and has in fact been implemented in Prolog without much modification, as described in the next chapter.

This chapter is in two sections. In 5.1 we formalise the simplified proof procedure that was described in 4.1.2. In 5.2 we extend this to include the necessary rules for implicit deletions.

5.1 Formalisation Of The Simplified Proof Procedure Described In 4.1.2

The proof procedure without inference rules for implicit deletions consists of two rules of inference, negation as failure and resolution (standard and extended). We formalise these in this section.

In general, we formalise our proof procedure by defining a relation "Inconsistent". "Inconsistent(s c)" holds if and only if there is a refutation with top clause named by c and input set named by s, by means of our proof procedure.

Note that in order to define our proof procedure, it is necessary to name object level sentences and other expressions by metalevel terms. This can be accomplished in a number of ways, and we do not concern ourselves with the details in this thesis.

Suppose "Demo" is the SLDNF provability relation, that is "Demo(d g)" is true when it can be shown that there is an SLDNF refutation of $d'\cup\{\leftarrow g'\}$, where d names a set d' of deductive rules and g names a conjunction g' of literals, and all the variables in g' are assumed to be existentially quantified over the whole of g'. Then the following relationship holds between the relations "Demo" and "Inconsistent":

Demo(d g) $\leftrightarrow$ Inconsistent(d <-g).

This relationship is a consequence of the fact that our proof procedure is identical to SLDNF when the top clause is a denial. The symbol "<-" is a metalevel function symbol representing the object level implication symbol "←".

In the rest of this thesis we ignore the distinction made above between object level expressions and their names, where context makes the intended meaning clear.

The base case for "Inconsistent" is defined by:

(I1) Inconsistent(s [ ])

(As before, "[ ]" denotes the empty clause.)

Rule (I2) formalises the standard and extended resolution rules, and (I3) formalises the negation as failure rule.

```
(I2)  Inconsistent(s c) ←
                     Select-literal (l c a) and
                     In(e s) and
                     Resolvent(e c l a r) and
                     Inconsistent(s r)
```

"Select-literal(l c a)" means literal l is selected from the a-side (Condition or Conclusion) of clause c. This relation must describe a safe computation rule. "In(e s)" means clause e is in input set s, and "Resolvent(e c l a r)" means r is the resolvent of clauses e and c on the literal l occurring on the a-side of c. As will become clear in Section 5.2, where we discuss inference rules for implicit deletions, "Resolvent" must be defined in such a way that it deals appropriately with clauses c that have negated conclusions and zero or more conditions. The a-side parameter is necessary to ensure that the negation as failure rule is only applied to negative conditions and not to negated conclusions.

```
(I3)  Inconsistent(s c) ←
                    Select-literal (not(p) c Condition) and
                    NOT Inconsistent(s  <–p) and
                    Remove-literal (c not(p) Condition c') and
                    Inconsistent(s c')
```

"Remove-literal(c l a c')" expresses that c' is clause c with literal l removed from its a-side. "not" is a metalevel prefix function symbol naming the negation symbol "NOT".

(I1), (I2) and (I3), together with the subsidiary definitions needed for them, formalise the proof procedure as described in 4.1.2. We can vary the literal selection strategy (or computation rule) by using different definitions for the relation "Select-literal".

Recall that computation rules are functions from derivations to literals. For the sake of simplicity, however, we have ignored this in the relation "Select-literal", and ultimately in the definition of "Inconsistent". As the relations stand at the moment, "Select-literal" selects a literal from any clause, without any information about the derivation in which the clause appears. It is possible to modify these relations to conform to our definition of computation rules. This can be done by changing the relations "Inconsistent" and "Select-literal" so that their second parameters denote the entire derivation ending at clause c.

Suppose "Inconsistent*" and "Select-literal*" are the modified relations corresponding to "Inconsistent" and "Select-literal".

"Inconsistent*(s d)" expresses that derivation d can be extended to form a refutation, with input set s, according to our proof procedure. "Select-literal*(l d a)" expresses that literal l is selected from the a-side of the last clause in derivation d. Suppose further that the term "d.r" denotes the derivation which consists of the derivation d extended by clause r. Rules (I1) and (I2) can now be modified as follows:

(I1)' Inconsistent*(s d.[ ])

(I2)' Inconsistent*(s d.c) ←

Select-literal*(l d.c a) and
In(e s) and
Resolvent(e c l a r) and
Inconsistent*(s d.c.r)

(I3) can also be similarly modified. (For a Prolog implementation it would be better to represent derivations backwards in a list. Thus a derivation C_0, C_1, C_2, for example, would be represented as (C_2 C_1 C_0).) As can be seen these modifications are quite trivial, and in the remainder of this thesis, for the sake of simplicity of notation, we shall ignore them and concentrate on the original rules (I1)-(I3).

To consider automatically all top clauses associated with the updates in the transaction we use rules (C1)-(C3), below. (C1) allows us to consider as top clause each of the deductive rules and the integrity constraints that are added by the transaction. (C2) allows us to consider as top clause the negation of each fact which is either explicitly deleted by the transaction, or which is a deleted ground instance of the conclusion of a non-atomic rule which is explicitly deleted. (C3) caters for updates that modify database rules. "transact(al dl ml)" is a term that represents the transaction T which consists of a set of additions al, a set of deletions dl, and a set of modifications ml. Each element of ml is of the form (r r') which is interpreted as an update that modifies a database rule r to r'.

```
(C1) IC-Violated(DT∪IT  transact(al dl ml)) ←
                            In(c al) and
                            Inconsistent(DT∪IT  c)

(C2) IC-Violated(DT∪IT  transact(al dl ml)) ←
                            In(f<-b dl) and
                            In(f<-b D) and
                            Demo(D f) and
                            NOT Demo(DT f) and
                            Inconsistent(DT∪IT not(f))

(C3) IC-Violated(DT∪IT  transact(al dl ml)) ←
                            In((r r')  ml) and
                            In (r  D) and
                            IC-Violated(DT∪IT  transact((r') (r) ()))
```

(C3) simply ignores an update that requests modifications to a database rule no variant of which is present in D. Another metalevel clause can be added to (C1)-(C3) to alert the user in such cases, if desired.

In (C2) the variable b stands for a (possibly empty) conjunction of literals. Thus (C2) caters for both deletion of facts and deletion of non-atomic rules. It correctly ignores the integrity constraints that are deleted by the transaction. It also ignores updates that request the deletion of rules no variants of which are present in D. In the symmetric case of additions, we can add an extra condition
"NOT In(c D)" to (C1) to avoid reasoning forward from clauses that are to be added, but which are already present in the database.

We can replace the "Demo" and "NOT Demo" conditions in (C2) by their equivalents "Inconsistent(D <-f)" and
"NOT Inconsistent(DT <-f)", repectively.

Notice that (C2) and (C3) do not treat unification explicitly. The required unification steps would automatically be performed if the rules were executed by a Prolog-like system. Alternatively, unification can be defined explicitly. In this case , in (C2), for example, an extra argument would be added to "Demo" to denote the appropriate instantiation, and an extra condition would be added to compute the resulting instantiated formula. Thus (C2) modified to incorporate explicit unification and substitution can be as follows:

```
IC-Violated(DT∪IT  transact(al dl ml)) ←
        In(f<-b  dl) and
        In(g<-e  D) and
        Variant(f<-b  g<-e) and
        SDemo(D g sub) and
        Apply-substitution(g sub g') and
        NOT Demo(DT g') and
        Inconsistent(DT∪IT   not(g')),
```

where
g and f name object level atoms, and e and b name object level (possibly empty) conjunctions of atoms,
Variant(c c') means the clause named by c is a variant of the clause named by c',
SDemo(d g s) means the fact named by g is provable (by SLDNF) from the database named by d with substitution named by s, and
Apply-substitution(g s g') means g' names the fact that results from applying the substitution named by s to the fact named by g.

Further discussion of the explicit treatment of unification can be found in Kowalski [1979].

(C1)-(C3) are not considered as part of the proof procedure, but are part of the database management system. To check the satisfaction of integrity constraints IT in database DT obtained from I and D, repectively, by a transaction consisting of a set of additions, al, a set of deletions, dl, and a set of modifications, ml, we evaluate the query

```
        ←IC-Violated(DT∪IT   transact(al dl ml)).
```

If the query succeeds then the constraints are violated. If it fails finitely, and our method is complete for the given case, then the constraints are satisfied.

Given our present scheme we can easily handle conditional updates as well. Such an update may request a transaction T of additions, deletions or modifications of rules in D, provided that D satisfies certain conditions, F, say. In this case we have to check the conditions F in D, and if they succeed we deal with transaction T as explained already. We will not, however, pursue the case of conditional updates any further in this thesis.

To complete the definition of the Consistency method proof procedure we have to augment it with inference rules for implicit deletions.

5.2 Formalisation Of The Rules For Implicit Deletions

There are at least two different ways of formalising the inference rules that we need for reasoning about implicit deletions. The first way, to be discussed in 5.2.1, is to formalise the inference rules as metalevel rules which are included in the input set. The second way, to be discussed in 5.2.2, is to formalise the inference rules as part of the proof procedure. The first approach is probably easier to understand. But the second approach is better for reasons we will explain later.

5.2.1 The First Approach: The Metarule Version

Let "Deleted(DT D f)" express that fact f is deleted in DT, in the sense that f is a logical consequence of Comp(D), but not of Comp(DT). This can be formalised by a metarule:

(MR) Deleted(DT D f) $\leftarrow$ Demo(D f) and NOT Demo(DT f).

We could write this rule more generally using variables for the first two arguments of "Deleted", and adding a condition "Result(d t dt)" to express that dt is the database that results from d by means of a transaction t. This would be necessary if we were to embed our integrity checking method into a more general knowledge assimilation system which processes a stream of transactions. In the context of this thesis, however, such generality makes the notation more cumbersome without providing any benefits.

The use of (MR) as it stands, would give rise to a blind and very inefficient search to find out what facts are deleted from the database as a result of the transaction. We can improve efficiency by adding to (MR) extra conditions to reduce the search. (MR1) and (MR2), below, both result from adding extra conditions to (MR). (MR1) is a general rule corresponding to (R1) in example 4.7. (MR2) corresponds to (R2) in example 4.8. The improved efficiency is due to the fact that, in effect, the extra conditions make the rules focus on the effects of the updates. This point will become more clear shortly.

The formulae (MR1) and (MR2) are not well-formed, because in both formulae p is intended as a metalevel variable ranging over object level facts, although it occupies the place of a metalevel atom. This problem is discussed further in note (3), below, where a solution is proposed.

(MR1) Deleted(DT D f) ←
NOT p and
In(f<-b DT) and
On(p b) and
Demo(D f) and
NOT Demo(DT f)

"On(p b)" means literal p occurs in b which is a conjunction of literals.

(MR2) Deleted(DT D f) ←
p and
In(f<-b DT) and
On(not(p) b) and
Demo(D f) and
NOT Demo(DT f)

The relationship between "Deleted" and negation as failure can be described by the rule:

(MR3) NOT f ← Deleted(DT D f)

Note that DT, the updated database, is explicit in the condition of (MR3), but implicit in the conclusion. This is to conform with the simplified syntax of negation as failure. Essentially, (MR1)-(MR3) together formalise a correct but partial definition of negation by failure as unprovability. Their purpose is to allow us to determine what facts are deleted (i.e. have become unprovable) as a result of the updates. Since the metarules give a partial definition of negation as failure, for the sake of efficiency, they should only be used forward. Otherwise, if used backward, they would only duplicate the effect of normal negation as failure rule. This restriction, and the use of updates as top clauses, ensure that (MR1) and (MR2) are "entered" only through their first conditions. In fact, these metarules are always resolved on their first conditions, with an update or with a clause which is a consequence of an update. Thus these rules, in practice, compute the implicit deletions resulting from the transaction.

Notes:

(1) As in (C2) in Section 5.1, metarules (MR1) and (MR2) do not treat unification explicitly. Again a Prolog-like system would automatically perform the required unification steps. Alternatively we could add extra conditions to these metarules to compute the required mgu's and the resulting instantiated formulae.

(2) The "Demo" and "NOT Demo" conditions in the metarules can be solved either by running metalevel definitions of "Demo" and "NOT Demo", or by using reflection as in FOL (Weyhrauch [1980]) or in amalgamation logic (Bowen and Kowalski [1982]). To solve "Demo(D f)" by reflection we show that the goal named f can be solved by SLDNF (or our proof procedure) in the database named D. To solve "NOT Demo(DT f)" by reflection we show that the goal named f

fails finitely by SLDNF (or our proof procedure) in the database named DT. The reflection rule approach can be formalised by adding two extra rules to the definition of the relation "Inconsistent" as follows.

```
(14) Inconsistent(s c) ←
              Select-literal(demo(d g)  c  Condition) and
              Inconsistent(d  <-g) and
              Remove-literal(c  demo(d g)  Condition  c') and
              Inconsistent(s c')

(15) Inconsistent(s c) ←
              Select-literal(not(demo(d g))  c  Condition) and
              NOT Inconsistent(d <-g) and
              Remove-literal(c  not(demo(d g))  Condition  c') and
              Inconsistent(s c')
```

"demo" is a function symbol naming the metalevel relation "Demo".

(3) The symbols p in (MR1) and (MR2) and f in (MR3) are variables, and are supposed to range over object level facts. These variables, however, do not occur everywhere as arguments of metalevel relations or functions in these rules. Thus our metarules (MR1), (MR2) and (MR3) are not strictly well-formed (although, in practice, they work when using a Prolog-like execution mechanism).This problem can be avoided by replacing the occurrences of p in the first conditions of (MR1) and (MR2) and the occurrence of f in the conclusion of (MR3) by well-formed atoms "Demo(DT p)" and "Demo(DT f)", respectively. This, however, introduces other complications, because it requires additional reflection rules and

modifications to the extended resolution step.

(4) Note that (MR1)-(MR3), together with the standard and the extended resolution rules cater for the propagation of the effects of initial additions and deletions through chains of database rules.

In this version of our method metarules (MR1), (MR2) and (MR3) are part of the input set. The proof procedure is defined by rules (I1)-(I5) and the subsidiary definitions required by these. The database management system would include rules (C1), (C2) and (C3). All occurrences of $DT \cup IT$ in (C1)-(C3) would have to be replaced by $DT \cup IT \cup \{(MR1), (MR2), (MR3)\}$ to include the metarules in the input set.

Notice that the proof procedure needs to to allow in derivations, formulae of the form

$$NOT\ A \leftarrow L_1 \text{ and } \ldots \text{ and } L_n, \qquad n \geq 1,$$

where A is an atom and the L_i are literals. Such formulae can be obtained in derivations by the resolution of a clause in the derivation with metarule (MR1) or (MR2), and the resolution of the resulting resolvent with (MR3).

The inclusion of such formulae in derivations is catered for in the definition of the proof procedure given in 5.1. We extend the term "clause" to include formulae of the above form.

The inclusion of the metarules in the input set is not entirely satisfactory from a methodological point of view. The metarules would have to be distinguished from the other input clauses. As well as requiring special control restrictions to ensure that they are used forward only, the metarules must be protected against modification by user updates. These problems and the naming problem, described in note (3) above, can be avoided by formalising (MR1)-(MR3) as inference rules in the definition of the proof procedure, as described in the remainder of this chapter.

5.2.2 The Second Approach: The Inference Rule Version

In this approach we formalise the rules for implicit deletions as part of the proof procedure simply by adding extra definitions for the relation "Inconsistent". Inference rules (I6) and (I7), below, correspond to (MR1) and (MR2), repectively, and incorporate (MR3) as well.

```
(I6)  Inconsistent(DT∪IT  not(p)<-c) ←
                        Select-literal(not(p)  not(p)<-c  Conclusion) and
                        In(f<-b  DT) and
                        On(p b) and
                        Demo(D f) and
                        NOT Demo(DT f) and
                        Inconsistent(DT∪IT  not(f)<-c)
```

```
(I7)  Inconsistent(DT∪IT  p<-c) ←
                    Select-literal(p  p<-c  Conclusion) and
                    In(f<-b  DT) and
                    On(not(p) b) and
                    Demo(D f) and
                    NOT Demo(DT f) and
                    Inconsistent(DT∪IT  not(f)<-c)
```

As in the case of (C2) in 5.1, we can replace the "Demo" and "NOT Demo" conditions in these rules by appropriate "Inconsistent" and "NOT Inconsistent" conditions, and in fact have done so in the implementation. As before, we have not treated unification explicitly in these rules.

In this scheme the input set consists only of the updated database and the updated set of integrity constraints. The proof procedure is defined by rules (I1)-(I3) and (I6)-(I7), and the subsidiary definitions required by them. The database management system includes rules (C1), (C2) and (C3).

The inference rule approach has a number of advantages over the metarule approach described in the previous subsection. The inference rules are part of the proof procedure and not the input set. They, therefore, cannot be modified by user updates. For the sake of efficiency the metarules should be used forward only, unlike the other deductive rules in the input set that can be used backward as well as forward. This is an undesirable and ad hoc restriction, which is avoided in the inference rule approach. Furthermore, the inference rules are well-formed, whereas the metarules are not.

The metarule and the inference rule approaches have both been implemented in Prolog.

CHAPTER 6

IMPLEMENTATION

Both versions of the Consistency method, that is the metarule and the inference rule versions, have been implemented by Soper [1986], as an M.Sc. project, in Sigma-Prolog on the SUN III. A summary of the implementation has been reported in Kowalski, Sadri and Soper [1987].

In effect, our proof procedure is built as a meta-interpreter on top of Prolog. The implementation is very close to the formalisation presented in the last chapter, and consists essentially of the clauses for "Inconsistent" and "IC-Violated" and the necessary subsidiary definitions. In this chapter we first discuss the part of the implementation that is common to both versions, and then consider the extensions that are necessary to implement each approach. Finally we propose an alternative and more efficient implementation for a special case.

6.1 Parts Common To Both Versions

As explained in the previous chapter, common to both versions of the Consistency method are rules (I1)-(I3) (and the subsidiary definitions required by them), and rules (C1)-(C3) (with minor modifications for the metarule version as explained in 5.2.1) for generating candidate top clauses for the proof procedure. In this section we concentrate on the implementation of these rules.

The main features of this implementation are:

(1) the representation of the input clauses, that is the databases and the integrity constraints,

(2) the implementation of reasoning with the two databases D and DT required for rules (C2) and (C3) (and also for (MR1)-(MR2), and (I6)-(I7)), and

(3) 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 one of these, in turn.

(1):

Input clauses are represented as terms in the metalanguage. They are represented as lists of "literals", where each "literal" is a list of the form

(side sign predicate | arguments).

"|" is Prolog's list construction operator. "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. Thus all the literals in an integrity constraint have "Cond" for "side". "sign" is either "+" for atoms or "-" for negated atoms.

Example 6.1:

(1) The clause

$P(x\ y) \leftarrow Q(x\ y)$ and NOT $R(x)$

is represented by the term

((Conc + P x y) (Cond + Q x y) (Cond - R x)).

(2) The clause

NOT P(A B)

is represented as

((Conc - P A B)). □

For the sake of notational simplicity, we have not distinguished between sentences and their names in the above example. As mentioned earlier, we will continue this practice where context makes the distinction clear. Strictly speaking, however, object level predicate symbols and variables should be represented at the metalevel by function symbols and constants, respectively (Bowen and Kowalski [1982]).

(2):

The Consistency method requires reasoning with two databases D and DT and with the updated input set $DT \cup IT$ (as well as $DT \cup IT \cup \{(MR1), (MR2), (MR3)\}$ in the metarule version; we will postpone discussing this latter case until the next section). It is in fact sufficient to represent only two sets to correspond to D and $DT \cup IT$, without distinguishing the set DT on its own. This is because, in our formalisation, DT occurs on its own only in conditions of the form

(i) NOT Demo(DT f),

(ii) In(f<-b DT), and

(iii) Deleted(DT D f),

and in conclusions of the form

(iv) Deleted(DT D f),

where f is a fact.

The replacement of DT by DT$\cup$IT in all cases (i)-(iv) makes no logical difference, as shown below.

Case (i): In SLDNF, and in our proof procedure, to prove a fact f from DT, the denial "$\leftarrow$f " is used as top clause. In this case all the subsequent clauses in the search space will also be denials or the emtpy clause. Thus no clause in any derivation can possibly resolve against another denial. Thus it makes no difference if the input set is augmented by the set of denials IT; these denials will never be used.

Case (ii): The replacement of DT in the "In" conditions makes no difference, either. This is because "In(f<-b DT$\cup$IT)" is true if and only if the first parameter is a deductive rule which is a member of DT$\cup$IT, and this, in turn, is true if and only if the first parameter is a deductive rule in DT, since IT consists of denials only.

Cases (iii) and (iv): If a fact f is deleted in DT, it is also deleted in DT$\cup$IT, and vice versa. So the replacement of the first argument of "Deleted" by DT$\cup$IT makes no difference.

The sets D and DT$\cup$IT 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". The reason for introducing "clause-name" will become clear shortly when we discuss selection of candidate clauses for resolution. "set" is either "OLD", or "NEW", or a variable. Clauses identified by "OLD" belong to D-DT. Clauses identified by "NEW" belong to (DT-D)$\cup$IT, and those identified by a variable belong to both databases D and DT. Thus, using Prolog's unification, we use the identifier "OLD" to access clauses in D, and the identifier "NEW" to access clauses in DT$\cup$IT. This scheme avoids the duplication of those clauses that are common to both D and DT. With this convention the top-level Sigma-Prolog goal, in the inference rule version, for integrity checking, for example, is

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

(In the metarule version the set "NEW" in this goal would have to be replaced by a set label denoting DT$\cup$IT$\cup$\{(MR1), (MR2), (MR3)\}. We will discuss this in the next section.)

The following simple example should make the set labelling scheme more clear.

Example 6.2:

Suppose D consists of the following clauses:

(CL1) P

(CL2) Q.

Let T be a transaction that deletes (CL1) and adds

(CL3) R.

Thus DT consists of:

(CL2) Q

(CL3) R.

Then clause (CL1) has label "OLD",

clause (CL2) has a variable label, and

clause (CL3) has label "NEW".

The databases are represented by the following "Member" assertions:

((Member OLD CL1 ((Conc + P))))

((Member x CL2 ((Conc + Q))))

((Member NEW CL3 ((Conc + R)))). □

The set labelling is done automatically by another Prolog program that preprocesses the initial input set and the transaction.

(3):

One potential source of inefficiency in the proof procedure is the search involved in (I2) 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 Consistency method proof procedure, however, selected literals can come from the conclusion as well as from the condition of clauses. This requires a larger search over all literal occurrences in the input set, looking for unifying conditions as well as conclusions. The implementation uses a form of indexing, similar to Kowalski's connection graphs (Kowalski [1975]), to reduce this search as follows.

For each predicate occurrence "predicate" in each clause in the input set we include an assertion of the form

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

in the metalevel 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). (The "Possible-resolve" assertions can be refined to take the unification of the arguments into account as well.) These assertions are generated automatically by another Prolog program, 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.

Without the "Possible-resolve" assertions, to find a candidate clause for resolution, we have to access each input clause, in turn, and traverse its literals in the search for a match. The "Possible-resolve" assertions provide a more direct access to matching literals. Although there can be more "Possible-resolve" assertions than "Member" assertions, the use of the former has the advantage of doing away with the list processing involved in traversing the literals of clauses.

Example 6.3:

Suppose the following clause is in set "NEW" and is given name CL1:

P(x) ← Q(x y) and NOT R(x).

Then there is a "Member" assertion:

```
((Member NEW CL1
        ((Conc + P x y) (Cond + Q x y)
                        (Cond - R x)))).
```

The corresponding "Possible-resolve" assertions for this clause are:

```
        ((Possible-resolve NEW CL1  1 (Cond + P)))
        ((Possible-resolve NEW CL1  2 (Conc + Q)))
        ((Possible-resolve NEW CL1  3 (Conc - R))).   □
```

Rule (I2) in 5.1 can be rewritten as follows, to use the "Possible-resolve" assertions:

```
(PI2)
((Inconsistent input-set clause)
        (Select-literal (side sign predicate | arguments) clause side)
        (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)
                side resolvent)
        (Inconsistent input-set resolvent))
```

For simplicity, throughout this chapter, we ignore the Sigma-Prolog convention of starting variables with a "_". We keep our earlier convention of starting variables and function symbols in the lower case, and constant and predicate symbols in the upper case.

"Resolvent-i", as compared to "Resolvent" in (I2), has an extra argument i which allows fast retrieval of the unifying literal in the input clause. Note that with our representation of input clauses the "side" parameters in "Select-literal" and in "Resolvent-i" are now redundant and can be removed.

Inference rules (I6) and (I7) can also be rewritten to exploit the "Possible-resolve" assertions.

A simple implementation for "Select-literal" is a strategy that selects literals in left to right order, delaying the non-ground negative conditions until they become ground, or more simply delaying all the negative conditions until after the positive ones have been selected.

Rule (I3) for the negation as failure step is implemented using Prolog's built-in predicate "NOT" for negation as failure, as follows:

```
(PI3)   ((Inconsistent s c)
              (Select-literal (Cond - predicate | arguments) c)
              (NOT Inconsistent s ((Cond + predicate | arguments)))
              (Remove-literal c (Cond - predicate | arguments) c')
              (Inconsistent s c'))
```

Note that in the "Select-literal" and "Remove-literal" conditions of (PI3) we have left out the "Condition" parameters (which were present in (I3)). These parameters are now redundant as explained above.

The relation "Remove-literal" is defined simply by the following rules:

```
((Remove-literal (x|y) x y))
((Remove-literal (x|y) z (x|y1))
          (Remove-literal y z y1)).
```

The implementation of rules (C1)-(C3) is straightforward and does not require any significant extensions to what has already been described. The "Demo" and "NOT Demo" conditions in these rules are replaced by "Inconsistent" and "NOT Inconsistent" conditions, as was explained in Section 5.1.

We can add another rule to the definition of the relation "Inconsistent" to cater for selected literals that are system, that is built-in Prolog, predicates, as follows:

```
(PSYS)   ((Inconsistent s c)
                    (Select-literal (Cond + | atom) c)
                    (SYS atom)
                    atom).
```

"SYS" is a built-in Prolog primitive such that

```
        (SYS (predicate | arguments))
```

succeeds if "predicate" is a Prolog built-in predicate.

This concludes the discussion of that part of the implementation that is common to both versions of the Consistency method. We next consider each of the two approaches to the formalisation of the rules for implicit deletions.

6.2 The First Approach: The Metarule Version

To implement the metarule version, in addition to what has already been described in Section 6.1, we need to implement metarules (MR1), (MR2) and (MR3) as part of the input set, and to augment the proof procedure with reflection rules of inference (I4) and (I5). The representation of the metarules is straightforward (although somewhat messy!). (MR1), for example, is represented by the following Sigma-Prolog term:

```
((Conc + Deleted NEW OLD (Conc + | atom1))
          (Cond - | atom)
          (Cond + Member NEW clause-name
                 ((Conc + | atom1) | x))
          (Cond + On (Cond + | atom) x)
          (Cond + Demo OLD (Conc + | atom1))
          (Cond - Demo NEW (Conc + | atom1)))
```

The details of the above term are not important. Note only that the relation "Member" implements the relation "In", which occurs in the second condition of (MR1). Note also that whenever we have used the set DT in (MR1) we use the set NEW, which represents DT$\cup$IT, in the above term. We justified this in detail in Section 6.1.

The metarules must be included as part of the updated input set. We give them a set label "META", and replace all occurrences of "NEW" in the implementation of (C1)-(C3) by a new set label "NEWnMETA" which represents DT$\cup$IT$\cup\{$(MR1), (MR2), (MR3)$\}$. We need the following two rules to describe the set "NEWnMETA":

```
((Member NEWnMETA x y) (Member NEW x y))
((Member NEWnMETA x y) (Member META x y)).
```

The top level Sigma-Prolog goal for integrity checking is

?((IC-Violated NEWnMETA (transact al dl ml))).

The relation "On" is a Prolog built-in predicate and is dealt with by rule (PSYS). To deal with the "Demo" and "NOT Demo" conditions of the metarules we need reflection rules (I4) and (I5), which are implemented in the same way as (I1)-(I3).

As mentioned in the last chapter for the sake of efficiency, the metarules should be used forward only. To implement this restriction we simply avoid generating any "Possible-resolve" assertions for the conclusions of the metarules. In fact, as explained in Subsection 5.2.1, the forward restriction and the choice of top clauses associated with the updates have the effect that metarules (MR1) and (MR2) are only "entered" through their first conditions, and it is therefore sufficient to generate "Possible-resolve" assertions only for these conditions, as well as for the condition of (MR3).

There are many ways of preventing the deletion of metarules by user updates. One way is for the database management system to preprocess the updates and reject those that delete the metarules. Another way is to maintain the "META" set label of the metarules after every transaction, and to modify rule (C2) to simply ignore those updates that attempt to delete the metarules. Such a modification can take the following form:

(C2)'

IC-Violated(DT$\cup$IT$\cup${(MR1), (MR2), (MR3)} transact(al dl ml)) $\leftarrow$

In(f<-b dl) and

NOT EQ(f deleted(x y z)) and

NOT EQ(f not(a)) and

Demo(D f) and

NOT Demo(DT f) and

Inconsistent(DT$\cup$IT$\cup${(MR1), (MR2), (MR3)} not(f)),

where "deleted" is a function symbol naming the predicate symbol "Deleted", and "EQ" is a built-in predicate in Prolog expressing identity.

6.3 The Second Approach: The Inference Rule Version

To implement the inference rule version, in addition to what was described in Section 6.1, we need to implement inference rules (I6) and (I7) for implicit deletions. This, however, is straightforward and does not require any new features from what has already been described in 6.1 and 6.2. This version does not require the reflection rules (I4) and (I5). Furthermore, since we do not have the metarules in the database, it is not necessary to restrict the "Possible-resolve" assertions.

6.4 An Alternative Implementation

In this section we propose an alternative and more efficient implementation for the special case where the database is definite, the updates are additions only, and the constraints are denials with or without negative conditions. In such a case there are no implicit deletions. Thus there is no need for inference rules for implicit deletions. Moreover, there is no need for the extended resolution step.

The major overhead of the implementation as described earlier in this chapter is the need for a meta-interpreter. Without the inference rules for implicit deletions, however, 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 that allows forward as well as backward reasoning. Thus one expects that, in this special case, it should be possible to implement the Consistency method proof procedure with efficiencies comparable to Prolog. This can, in fact, be done by a transformation of the input clauses that would make a meta-interpreter unnecessary. The transformation would, in effect, allow us to use Prolog's backward reasoning strategy to reason forward as well as backward.

In Prolog, because it is solely a backward chaining system, the input set needs only be searched for a conclusion that unifies with the selected literal. However, to allow both forward and backward reasoning the input set has to be searched for matching conditions as well. Earlier, in 6.1, we showed how this bigger space can be searched reasonably efficiently by an indexing method, implemented by the "Possible-resolve" assertions. A more effective alternative is to retain Prolog's restriction of only resolving on the conclusion literals of input clauses, and to replace each database clause by as many copies as there are literal occurrences in the clause, with each literal as the conclusion of one of the copies. An extra argument can be added to each literal in these copies to indicate whether it comes from the conclusion or the conditions of the original clause. A "+" argument signifies that the literal is on the same side of "←" as in the original clause; a "-" argument signifies that it has changed side.

Example 6.4:

Database clause

M(x y) ←N(x y) and K(y)

is replaced by

M(+ x y) ← N(+ x y) and K(+ y)

N(- x y) ← M(- x y) and K(+ y)

K(- y) ← M(- x y) and N(+ x y). □

The transformation of the integrity constraints is slightly different, since there is no need to generate copies of the constraints for all the literals that occur in them, but only for the positive literals, as illustrated in the following example.

Example 6.5:

The constraint

$\leftarrow$P(x) and NOT Q(x)

is replaced by the single rule

P(- x) $\leftarrow$ NOT Q(+ x). $\square$

We call each formula resulting from the transformation of a clause C, a <u>backward rule</u> associated with C.

We must also transform all top clauses into denials. Consider an update of adding a rule "P(x)$\leftarrow$Q(x)". This update adds the following two transformed rules:

P(+ x) $\leftarrow$ Q(+ x)

Q(- x) $\leftarrow$ P(- x).

The transformed top clause associated with this update, however, is the denial

$\leftarrow$P(- x) and Q(+ x).

An update of adding a constraint "$\leftarrow$P(x) and Q(x)" adds the following two transformed rules to the input set:

P(- x) $\leftarrow$ Q(+ x)

Q(- x) $\leftarrow$ P(+ x).

The transformed top clause associated with it, however, is the denial

$\leftarrow$P(+ x) and Q(+ x).

Let C' be the transformed top clause associated with a clause C. We call C' the <u>backward denial</u> version of C. Thus "$\leftarrow$P(- A)", for example, is the backward denial version of "P(A)".

The following example illustrates how these transformations allow us to simulate forward reasoning by backward reasoning.

Example 6.6:

D:

(R1) R(x y) ← P(x) and Q(y)

(R2) Q(A)

(R3) Q(B)

(R4) T(A)

I:

(IC) ← NOT T(y) and R(x y)

T: Insert P(A).

D, (IC) and T are transformed to:

D':

(R1.1) R(+ x y) ← P(+ x) and Q(+ y)

(R1.2) P(- x) ← R(- x y) and Q(+ y)

(R1.3) Q(- y) ← P(+ x) and R(- x y)

(R2.1) Q(+ A)

(R3.1) Q(+ B)

(R4.1) T(+ A)

I':

(IC.1) $R(- x\ y) \leftarrow$ NOT $T(+ y)$

T': Insert $P(+ A)$.

Using "$\leftarrow P(- A)$" 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.

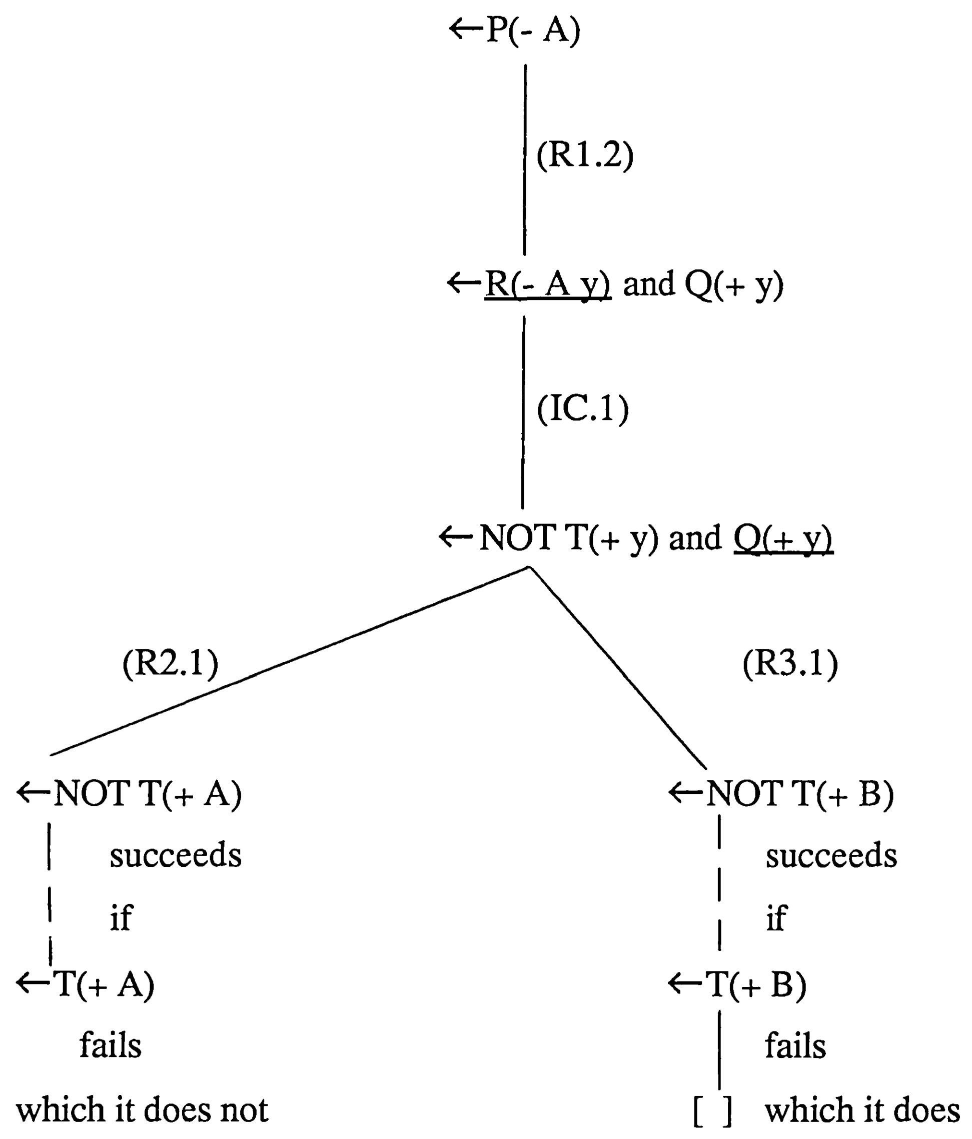


Figure 6.1: A Prolog seach space (via a safe computation rule) for the transformed version of example 6.6

Using an analogous literal selection strategy but reasoning forward from T with DT∪IT as input set we obtain a search space with a similar structure, as is shown below.

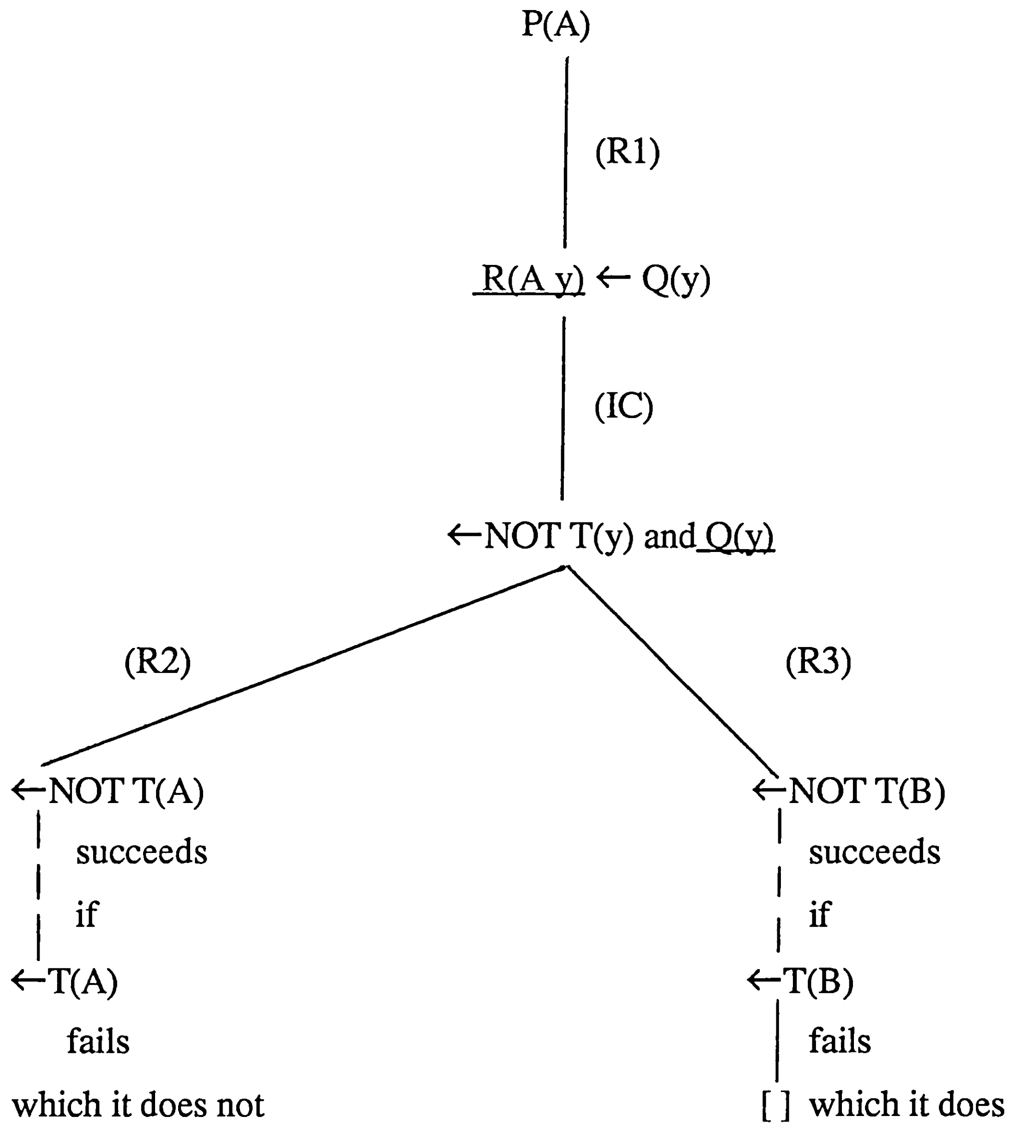


Figure 6.2: A search space for example 6.6 by means of our proof procedure □

In the above example we insisted on a safe computation rule. However, not all implementations of Prolog have safe computation rules. With a minor modification of our transformation scheme we can use the common Prolog left to right computation rule without a safety restriction. The modification necessary is simply to put the negative conditions of the constraints after all the positive ones in the conditions of the constraints after all the positive ones in the transformed versions. This ensures that when the negative conditions are selected they are ground. In the rest of this section , when we refer to Prolog, we assume that either it has a safe computation rule, or that the constraints are transformed as just explained.

In general, for the case we are considering in this section, 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. That is, given a search space in our proof procedure for the original database, constraints and update, we can construct an isomorphic SLDNF search space for the transformed database, constraints and update, and vice versa. We prove this in the following theorem.

Theorem 6.1:

Let S be a set of definite clauses and denials. Let S' be the set of all backward rules associated with the clauses in S. Let C_1 be a clause in S, and let C_1' be the backward denial version of C_1.

There is a derivation $C_1, C_2, \ldots, C_n$ for S by means of our proof procedure, without rules for implicit deletions and without the extended resolution step, with top clause C_1, if and only if there is an SLDNF derivation $C_1', C_2', \ldots, C_n'$ of $S' \cup \{C_1'\}$, such that each C_i' is the backward denial version of C_i.

Proof:

We will only prove the " only if" half of the theorem. The "if" half can be proved by a similar argument.

The "only if" half:
Let C_1,..., C_n be a derivation for S via a safe computation rule R by means of our proof procedure. We show that there is a safe computation rule R' such that there is the required derivation C_1',...,C_n' via R'. This proof is by induction on n:

The base case:
n=1: C_1' is known to be the backward denial version of C_1.

The inductive case:
Suppose C_{n-1}' is the backward denial version of C_{n-1}. C_{n-1} is of the form

$$(a)\leftarrow l_1,\ldots, l_n, \qquad n\geq 0,$$

such that a is an atom and the l_i are literals, and a may be absent. We use "," to denote the connective "and".

C_{n-1}' is of the form

$$\leftarrow(a[-]), l_1[+],\ldots, l_n[+], \qquad n\geq 0,$$

where "a[-]" denotes the atom a with a "-" added as its first argument, and "l_i[+]" denotes the literal l_i with a "+" added as its first argument.

There are three cases:

(1) R selects a positive condition l_j from C_{n-1}. Then C_n, if it exists, is obtained by the resolution on l_j of C_{n-1} and some input clause

$$r: \qquad f \leftarrow b_1, ..., b_m, \qquad m \geq 0,$$

with mgu ϕ. Thus C_n is

$$[(a) \leftarrow l_1, ..., l_{j-1}, l_{j+1}, ..., l_n, b_1, ..., b_m]\phi.$$

In this case let R' select from C_{n-1}' the positive condition $l_j[+]$, and obtain C_n' by the resolution on $l_j[+]$ of C_{n-1}' and the backward rule associated with r which is of the form

$$f[+] \leftarrow b_1[+], ..., b_m[+],$$

where "f[+]" and the "$b_i[+]$" are f and the b_i, respectively, with a "+" added as their first argument. The mgu of this resolution step is ϕ, and C_n' is

$$[\leftarrow (a[-]), l_1[+], ..., l_{j-1}[+], l_{j+1}[+], ..., l_n[+],$$
$$b_1[+], ..., b_m[+]\,]\phi,$$

which is the backward denial version of C_n.

(2) R selects the conclusion a of C_{n-1} (if a is present). Then C_n, if it exists, is obtained by the resolution on a of C_{n-1} and some condition k_i of an input clause

$$s: \quad (h) \leftarrow k_1, ..., k_p$$

with mgu σ. (If s is a constraint then h is absent, and the k_j are literals. If s is a database rule then h is present and the k_j are atoms.) Thus C_n is

$$[\,(h) \leftarrow l_1, ..., l_n, k_1, ..., k_{i-1}, k_{i+1}, ..., k_p\,]\sigma.$$

Let R' select the condition a[-] of C_{n-1}', and obtain C_n' by the resolution on a[-] of C_{n-1}' and the backward rule associated with s which is of the form

$k_i[-] \leftarrow (h[-]), k_1[+], ..., k_{i-1}[+], k_{i+1}[+], ..., k_p[+]$.

C_n' is then

$[\leftarrow (h[-]), l_1[+], ..., l_n[+], k_1[+], k_{i-1}[+], k_{i+1}[+], ..., k_p[+]]\sigma$,

which is the backward denial version of C_n.

(3) R selects a negative condition l_k of C_{n-1}. Suppose this condition is "NOT $P(t^*)$", where t^* is a vector of ground terms. Then C_n, if it exists, is C_{n-1} with this condition removed.

Let R' select the negative condition $l_k[+]$ from C_{n-1}'. $l_k[+]$ is the literal "NOT $P(+\ t^*)$".

To prove that C_n' is the backward denial version of C_n, it is sufficient to prove that if there is a finitely failed search space for $S \cup \{\leftarrow P(t^*)\}$ by means of our proof procedure with "$\leftarrow P(t^*)$" as top clause, then there is an SLDNF finitely failed search space for $S' \cup \{\leftarrow P(+\ t^*)\}$. We show this below:

Let F be a finitely failed search space for $S \cup \{\leftarrow P(t^*)\}$ by means of our proof procedure with "$\leftarrow P(t^*)$" as top clause. Our proof procedure is identical to SLDNF when the top clause is a denial. So F is an SLDNF finitely failed search space for $S \cup \{\leftarrow P(t^*)\}$. Since F is an SLDNF search space, only the definite clauses in S could have contributed to it. So F is an SLDNF finitely failed search space for $D \cup \{\leftarrow P(t^*)\}$, where D is the set of all definite clauses in S. Now let F' be F with a "+" added as first argument to every relation that occurs in F. F' is an

SLDNF finitely failed search space for D"$\cup$\{$\leftarrow$P(+ t^*)\}, where D" is the set of all definite clauses in D with a "+" added as first argument of every literal that occurs in them. Thus D" is a subset of S'. Now F' is also an SLDNF finitely failed search space for S'$\cup$\{$\leftarrow$P(+ t^*)\}. This is because a positive condition of a denial which has a "+" as first argument can only be resolved with a rule in D", and the resolution can only introduce more positive conditions with a "+" as their first arguments. □

Prolog is a special case of SLDNF. Thus any Prolog search space is also an SLDNF search space. Therefore by Theorem 6.1, given a Prolog search space for the transformed version of the update and the transformed version of the input set, there is an analogous search space in our proof procedure with the original update and input set. This shows the correctness of the Prolog simulation of our integrity checking method for definite databases, and in the absence of (explicit and implicit) deletions.

This simulation, however, is not complete, in the sense that there might be a search space in our proof procedure which cannot be simulated by Prolog. This is because of Prolog's left to right literal selection rule, as illustrated in the following simple example.

Example 6.7:

D: R(x) ← R(x)

I:

(IC) ←P(x) and R(x) and Q(x)

T: Insert P(A).

The following is a finitely failed search space in our proof procedure for this example.

$$\begin{tikzcd} P(A) \arrow[d, dash] \\ \leftarrow R(A) \text{ and } \underline{Q(A)} \end{tikzcd}$$

Figure 6.3: A search space for example 6.7 by means of our proof procedure

The analogous finitely failed SLDNF search space is as follows.

$$\begin{tikzcd} \leftarrow P(-A) \arrow[d, dash, "\text{transformed version of (IC)}"] \\ \leftarrow R(+A) \text{ and } \underline{Q(+A)} \end{tikzcd}$$

Figure 6.4: An SLDNF finitely failed search space for the transformed version of example 6.7

The Prolog search space is infinite, however:

$$\begin{tikzcd}
\leftarrow P(-\ A) \arrow[d, dash] \\
\leftarrow \underline{R(+\ A)} \text{ and } Q(+\ A) \arrow[d, dash] \\
\leftarrow \underline{R(+\ A)} \text{ and } Q(+\ A) \arrow[d, dash] \\
\vdots
\end{tikzcd}$$

Figure 6.5: A Prolog search space for the transformed version of example 6.7 □

CHAPTER 7

RELATED WORK

There is a substantial body of literature concerning integrity constraints in databases. Some of this work focuses on relational databases only, but in recent years a number of new papers have emerged addressing the problem of constraints in deductive databases, and even more generally in clausal databases, that is in databases that contain non-Horn clauses with disjunctive conclusions.

In relational databases the dominant approach has been to devise special purpose procedures to deal with specific kinds of integrity constraints and updates (see Ullman [1983], for example). One paper, however, proposes a general method for checking integrity of relational databases. This is the influential paper of Nicolas [1982]. He chooses a logical view of the database and of constraint satisfaction, and allows any closed domain independent formula of first order predicate logic as integrity constraint. Moreover, he proposes a simplification method which exploits the assumption that the constraints are satisfied in the database prior to the transaction. Given a transaction T, and a set I of integrity constraints, Nicolas uses the updates in T to construct from I a simplified set of constraints I', according to certain syntactic criteria, such that to check the satisfaction of I it is sufficient to check the satisfaction of I'. The formulae in I' are typically simpler and further instantiated than those in I, and therefore it involves less work to check the satisfaction of I' than I.

Most of the general integrity checking algorithms for deductive databases are extensions of Nicolas' algorithm. The earliest such algorithms are Lloyd, Topor, et al's (Lloyd, et al [1986], Lloyd and Topor [1985], Topor, et al [1985]), and Decker's (Decker [1986]). Our Consistency method is also an extension of Nicolas' algorithm. Other descendants of Nicolas' work, proposed after the Consistency method are the algorithms of Martens and Bruynooghe [1987], and Bry, Decker and Manthey [1987].

The features required for our method, such as rules for deriving implicit deletions, can be incorporated in proof procedures which are not input procedures. The choice of an input proof procedure, however, facilitates the comparison of our method with other algorithms for integrity checking. It is possible to obtain different algorithms from our method by adopting different strategies for literal selection and for searching the resulting search space. Two such strategies allow us to approximate these other four simplification algorithms for deductive databases.

There are other methods for integrity checking in deductive and non-Horn clausal databases that are not descendants of Nicolas' algorithm. Reiter [1988], for example, proposes a modal approach to describing and checking integrity constraints. In an earlier paper, Reiter [1981] presents an algorithm for checking type constraints in non-Horn clausal databases. Finally, Asirelli, et al [1985] also propose an integrity checking algorithm for deductive databases. Henschen, McCune and Naqvi [1984] present an interesting method for preprocessing update schemas and integrity constraints to generate tests that are carried out when actual updates are requested. Their method caters for relational databases only. However, as it combines theorem-proving techniques and a simplification method for integrity checking, it merits a review in this chapter.

We describe these latter methods in more detail in Section 7.3. Before that we compare our method with the algorithms of Lloyd, Topor, et al and Decker in Section 7.1, and with the algorithms of Martens and Bruynooghe and Bry, et al in Section 7.2.

Note:

Lloyd and Topor [1985] first described a simplification algorithm for checking integrity constraints in definite databases. They and their collaborators (Topor, Keddis and Wright [1985], and Lloyd, Sonenberg and Topor [1986]) then extended this algorithm to deductive databases. For simplicity, in the sequel, we call this more general algorithm the LT algorithm. The LT algorithm and all its related results also appear in Lloyd [1987].

7.1 Comparison Of The Consistency Method With Decker's And The LT Algorithms

Both the LT and Decker algorithms cater for first order formulae of predicate logic as integrity constraints. To avoid floundering they impose restrictions identical to ours on variable occurrences in database rules and integrity constraints. Both algorithms are based on the theoremhood view of constraint satisfaction. Another common feature of the two algorithms is that they each consist of three, possibly interleaved stages. The first stage reasons forward from the updates to compute the facts that are (potentially, in the LT case) added or deleted as a result of the transaction. The second stage uses these facts to simplify the constraints. Finally, the third stage evaluates the simplified constraints. Lloyd, Topor, et al and Decker use special purpose procedures for the first two stages, and SLDNF or a similar proof procedure for the third.

There are no completeness or correctness results available for Decker's algorithm. Lloyd, Sonenberg and Topor [1986], however, have proved the LT algorithm sound for stratified databases . On the whole, their soundness result is more general than our correctness and completeness results which we prove in Chapter 8. We give their result in more detail after the description of their algorithm in Subsection 7.1.2.

7.1.1 Decker's Algorithm

Decker caters for function free range-restricted deductive databases, and function free range-restricted first order formulae as integrity constraints. The function free restriction does not appear to be necessary for correctness (although there are no correctness results available for Decker's algorithm). The restriction seems to be imposed for simplicity. If function symbols are present then the database might need to include an equality theory. Also the presence of function symbols in conjunction with recursion could cause infinite computations. The Consistency method would also suffer similar disadvantages in the presence of function symbols. However, the method is correct in general, even with function symbols, as will be proved in the next chapter.

Decker's transactions consist of additions and deletions of facts and non-atomic rules. His algorithm has been implemented in Prolog.

We illustrate his algorithm and compare it with ours and the LT algorithm by the following example, chosen to illustrate the differences between the three approaches.

Example 7.1:

D:

(1) K(x) $\leftarrow$ P(x) and L(x)
(2) R(x y) $\leftarrow$ P(x) and Q(y)
(3) M(A) $\leftarrow$ NOT V(A)
(4) H(x) $\leftarrow$ K(x)
(5) N(x) $\leftarrow$ T(x)
(6) E(x) $\leftarrow$ G(x)
(7) S(x) $\leftarrow$ E(x)
(8) Q(B)
(9) Q(C)
(10) J(A)
(11) T(A)

I:

(IC1) $W(x) \leftarrow S(x)$

(IC2) $\leftarrow H(x)$ and $J(x)$

```
T:    { insert  P(A)
        insert  V(A)
        insert  G(A)
        delete  T(A) }
```

In general, to check if DT satisfies the constraints, Decker considers each update in T in turn. For each update he incrementally computes the facts added and deleted as a result of the update, simplifies the constraints according to syntactic criteria similar to Nicolas, and evaluates the simplified constraints. As soon as a violation of integrity is detected the algorithm terminates. According to Decker [1986], if the processing of the updates finishes and no violation is detected then the transaction does not violate the constraints (Decker does not, however, give a proof of this). If the database contains no recursion before and after the transaction, then the algorithm will always terminate. The added and deleted facts are computed in a specific order. To facilitate their computation, Decker keeps track of the dependencies in the database, in a way similar to our use of the "Possible-resolve" assertions described in Chapter 6. For each database rule

Head←Conds,

Decker maintains a set of facts of the form

Occurs-negative(l Head←Conds)

for all negative conditions l of the rule, and a set of facts of the form

Occurs-positive(k Head←Conds)

for all the positive conditions k.

Furthermore, he precompiles the integrity constraints into "update constraints" which are then used at the constraint simplification stage of the algorithm. (IC1), for example, is precompiled into the following:

insert S(x) only-if W(x)

delete W(x) only-if NOT S(x).

The effect of using the precompiled constraints in Decker's algorithm can be simulated by rewriting the constraints as denials, in the way we do in our method, and using resolution, as will be shown shortly.

Decker gives an English description of his algorithm which is lengthy and difficult to summarise here. We, therefore, do not give a precise characterisation of his algorithm in the general case, and only describe it in the context of example 7.1.

Assuming that the updates are considered in the order they are written, Decker first considers the insertion of "P(A)". This does not match any of the constraints. Decker, then, considers each of the facts added by the insertion of "P(A)", in turn. These facts are "R(A B)" and "R(A C)", and neither of them match any of the constraints, and no facts are deleted as a result of their insertion, or as a result of the insertion of "P(A)". So Decker moves to the next update, the insertion of "V(A)". This does not match the constraints. It results in the (implicit) deletion of "M(A)", because of rule (3), but this deletion does not match the constraints, either. The third update, the addition of "G(A)", does not match the constraints. It results in the addition of "E(A)", which, in turn, results in the addition of "S(A)". Each is considered in turn. The fact "S(A)" matches the condition of the first constraint. The simplified constraint "W(A)" is then evaluated by attempting to prove that it is a theorem of the completion of the updated database. The attempt fails finitely, showing a violation of the constraint. The fourth update is not considered at all. (The fact that "W(A)" is not a theorem follows from the correctness of SLDNF (Clark [1978]) and the consistency of Comp(DT): the SLDNF search space for $DT \cup \{\leftarrow W(A)\}$ fails finitely. Therefore by correctness of SLDNF "NOT W(A)" is a theorem of Comp(DT), where "NOT" is interpreted as classical negation. DT is stratified, and therefore Comp(DT) is consistent. So "W(A)" cannot be a theorem of Comp(DT).)

We can simulate Decker's algorithm, in general, by using our proof procedure, with the updates as top clauses, and employing the literal selection strategy that always selects a condition of a clause in preference to the conclusion, if there is one. Depth first search of the resulting search spaces then corresponds to Decker's interleaving of the three stages of his algorithm. Adopting these strategies in example 7.1, we obtain the following three search spaces, each with one of the updates as top clauses.

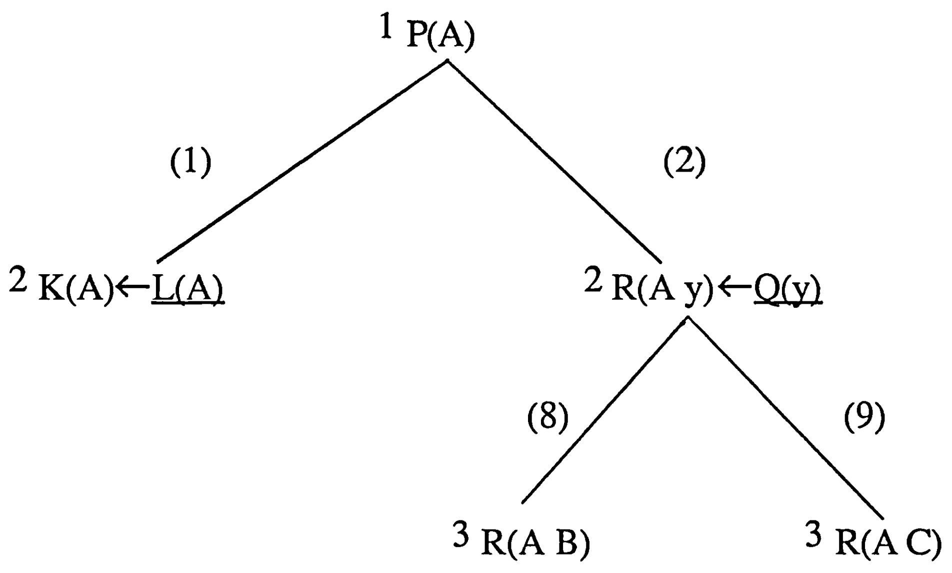


Figure 7.1: A search space for example 7.1 simulating Decker's algorithm

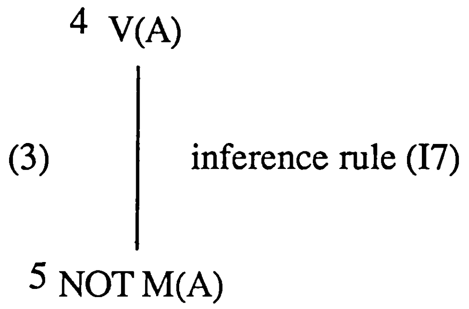


Figure 7.2: A search space for example 7.1 simulating Decker's algorithm

```
6 G(A)
     | (6)
7 E(A)
     | (7)
8 S(A)
     | (IC1) rewritten:
     | ←S(x) and NOT W(x)
9 ←NOT W(A)
     | succeeds
     | if
10 ←W(A)
     | fails
     |
11 [ ] which it does
```

__Figure 7.3: A search space for example 7.1 simulating Decker's algorithm__

The strategy that searches these search spaces in the order indicated by the numbers at the clauses simulates Decker's algorithm almost exactly. Where two clauses have the same number the order in which they are investigated does not matter.

The computations involved in figure 7.1 correspond to Decker's evaluation of the facts added as a result of the first update. The computations in figure 7.2 correspond to Decker's evaluation of the fact deleted as a result of the second update, and the computations in figure 7.3 correspond to his evaluation of the facts added as a result of the third update, and the simplification and evaluation of (IC1).

The main difference between Decker's algorithm and our approximation of it, in general, is that he checks if a fact derivable as a consequence of an update is provable in the database prior to the transaction, and reasons forward from the fact only if it is not. We reason forward from all facts that are consequences of the updates. In the symmetric case of facts not derivable from the updated database, however, we, similar to Decker, ensure that the facts are indeed provable before but not after the transaction, before we treat them as deletions.

We can simulate Decker's algorithm exactly by modifying inference rules (I2) and (I3), and the database management rule (C1), so that we reason forward from facts derivable from the updated database only if they are not derivable from the old database. (C1), for example, can be replaced by the following two rules:

```
IC-Violated(DT∪IT  transact(al dl ml)) ←
                    In(c al) and
                    Fact(c) and
                    NOT Demo(D c) and
                    Inconsistent(DT∪IT  c)

IC-Violated(DT∪IT  transact(al dl ml)) ←
                    In(c al) and
                    NOT Fact(c) and
                    Inconsistent(DT∪IT  c),
```

where "fact(c)" expresses that c is a fact. (I2) and (I3) can be modified in a similar way. This "redundancy check" can sometimes result in better efficiency, and sometimes in worse, depending on the example.

7.1.2 The LT Algorithm

The LT Algorithm is similar to Decker's. The main difference is that to find appropriate instantiations for the integrity constraints, Lloyd, Topor, et al only process the non-atomic rules in the database, and not the facts. Avoiding access to database facts in the first stage of the algorithm can have advantages in organising storage (Lloyd and Topor [1985]) in order to minimise access to secondary storage. As far as overall efficiency is concerned, however, it can be advantageous or disadvantageous depending on individual cases.

Another, less important difference between the LT algorithm and Decker's, is that the former does not do either of the redundancy checks mentioned at the end of the last subsection. Similar to Decker's, the LT algorithm caters for updates that add or delete deductive rules.

The LT algorithm has been implemented in Prolog by students of the University of Melbourne.

Below, we present a brief description of the LT algorithm in the general case , and illustrate its application to example 7.1. Our description of the algorithm is close to that given in Lloyd, Sonenberg and Topor [1986] and in Lloyd [1987]. The only difference is that they describe the first two stages of their algorithm for closed first order formulae as constraints, and databases that consist of rules of the form

$A \leftarrow W$,

where W is a first order formula. We present a slightly simplified description of their algorithm for deductive databases, and constraints written as non-Horn clauses.

Let D and DT be databases and T a transaction whose application to D produces DT. Suppose that the application of the deletions in T produces the intermediate database D". Thus D" is a subset of both D and DT.

The first stage of the LT algorithm is the computation of four sets, $pos_{D'',D}$, $neg_{D'',D}$, $pos_{D'',DT}$ and $neg_{D'',DT}$. Informally, the second and the third sets are the sets of facts that are potentially added to the database as a result of the transaction, and the first and the fourth are the sets of facts that are potentially deleted.

These sets are computed as follows. Let E and E' be two databases such that E is a subset of E'. Then the sets $pos_{E,E'}$ and $neg_{E,E'}$ are defined as follows: (for convenience, we drop the subscripts E,E' in the following.)

$pos = \cup_{n \geq 0}\ pos^n$

$neg = \cup_{n \geq 0}\ neg^n$, where

$pos^0 = \{a:\ a \leftarrow conds\ \text{ is in } E'-E\}$

$neg^0 = \{\ \}$

$pos^{n+1}=$

{ $a\phi$: $a\leftarrow$ conds is in E, and l is a positive condition in conds, and l' is in pos^n, and ϕ is the mgu of l and l'} $\cup$

{ $a\phi$: $a\leftarrow$conds is in E, and l is a negative condition in conds, and l' is in neg^n, and ϕ is the mgu of l and l'}

$neg^{n+1}=$

{ $a\phi$: $a\leftarrow$conds is in E, and l is a positive condition in conds, and l' is in neg^n, and ϕ is the mgu of l and l'} $\cup$

{ $a\phi$: $a\leftarrow$ conds is in E, and l is a negative condition in conds, and l' is in pos^n, and ϕ is the mgu of l and l'}

Thus in example 7.1:

$pos_{D'',D}$ = {T(A), N(A)}

$neg_{D'',D}$ = {}

$pos_{D'',DT}$={P(A), V(A), G(A), E(A), S(A), K(A), R(A y), H(A)}

$neg_{D'',DT}$ ={M(A)}.

If the database contains no recursion, then the computation of the pos and neg sets will terminate. In general, however, the computation may be infinite. Lloyd and Topor [1985] and Lloyd, Sonenberg and Topor [1986] propose certain stopping criteria that can sometimes ensure the termination of this computation.

The next stage of the algorithm is the instantiation of the integrity constraints. Suppose the clause

B_1 or ... or $B_m \leftarrow A_1$ and ... and A_n, $\qquad m,n \geq 0$, is an integrity constraint. Then it is sufficient to evaluate this constraint for all instantiations ϕ such that

> there is an atom l in $pos_{D'',DT}$ and ϕ is the mgu of l and some A_i, or
>
> there is an atom l in $neg_{D'',DT}$ and ϕ is the mgu of l and some B_i, or
>
> there is an atom l in $pos_{D'',D}$ and ϕ is the mgu of l and some B_i, or
>
> there is an atom l in $neg_{D'',D}$ and ϕ is the mgu of l and some A_i.

Thus in example 7.1 the following instances of the constraints have to be evaluated:

(IC1)* W(A)←S(A)

(IC2)* ←H(A) and J(A).

The last stage of the algorithm consists of the evaluation of the constraints by SLDNF. We will just show the evaluation of (IC1)*. It will illustrate a feature of the LT algorithm which we will discuss later.

To evalute (IC1)* we introduce a new relation "Constraint-satisfied", with the definition

Constraint-satisfied ← [NOT S(A) or W(A)],

which can be converted into two deductive rules

Constraint-satisfied ← NOT S(A)

Constraint-satisfied ← W(A).

Now we use the SLDNF proof procedure with "←Constraint-satisfied" as top clause. We obtain the following search space, which shows that the constraint is not satisfied. (The fact that the finitely failed search space shows the violation of the constraint follows by a similar argument to that presented in the paragraph immediately following Figure 4.2 in Chapter 4, where we argued that Figure 4.2 showed that the constraint was not a theorem of the completion of the updated database.)

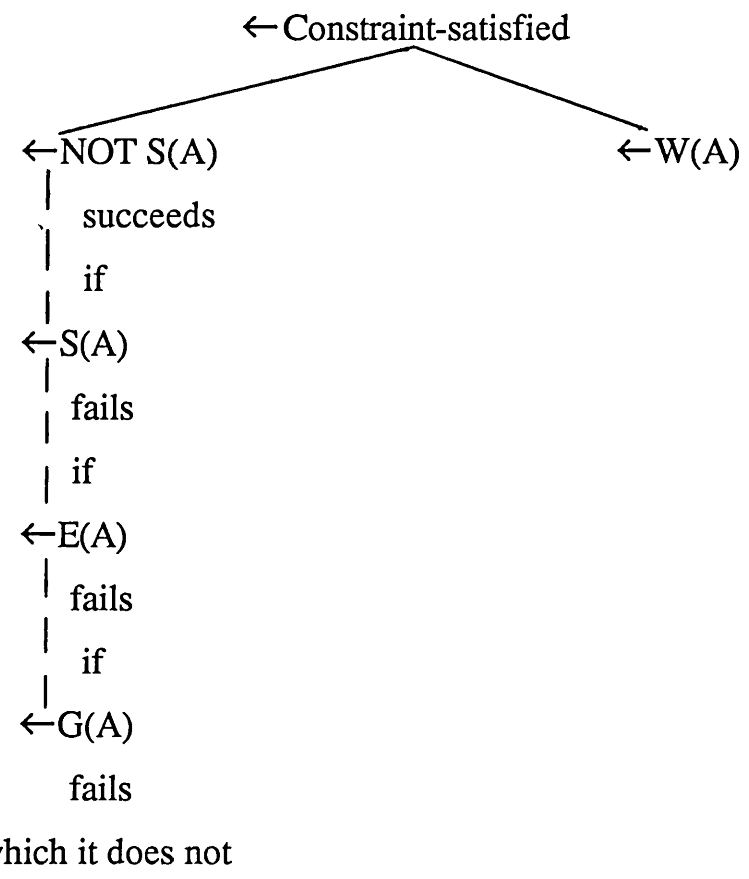
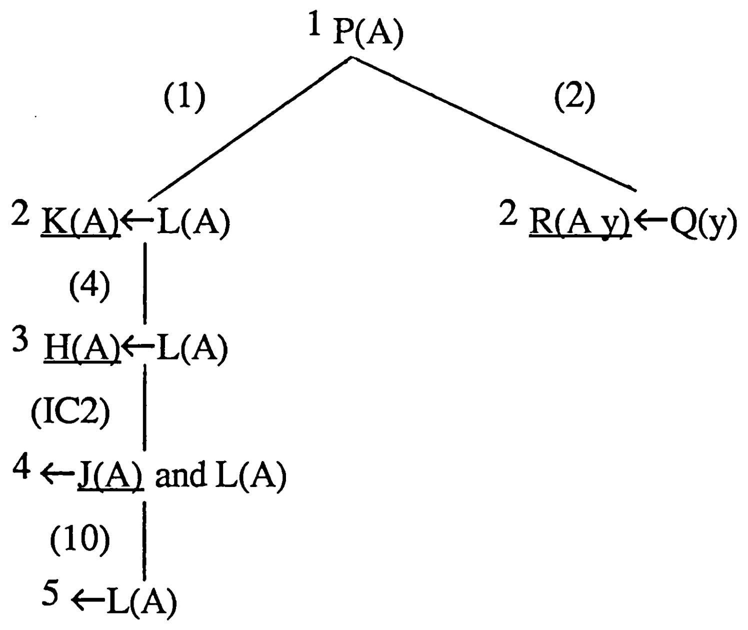


Figure 7.4: An SLDNF search space for example 7.1 showing that (IC1) is not a theorem of the completion of the updated database

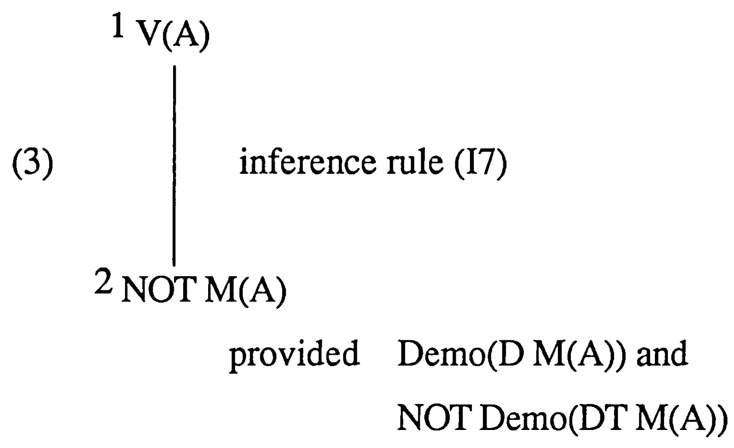
One of the inefficiencies of the LT algorithm is that some of the information obtained during the computation of the pos and neg sets may be thrown away and have to be recomputed when evaluating the instantiated constraints. This redundancy is avoided in our simulation of their algorithm.

The LT algorithm has been proved correct for stratified deductive databases (Lloyd, Sonenberg and Topor [1986]). They define the correctness of their method roughly as follows. Let (IC) be a constraint satisfied in database D, and let DT be the updated database. Let (IC') be the set of all the instantiated constraints resulting from (IC) after the processing of the transaction. Then DT satisfies (IC) if and only if DT satisfies (IC'). They further conclude (easily by the consistency of the completions of stratified databases, and SLDNF results) that: (i) if there is an SLDNF proof of the theoremhood of all the constraints in (IC'), then DT satisfies (IC), and (ii) if SLDNF fails finitely to prove the theoremhood of some constraint in (IC'), then DT violates (IC).

We can approximate the LT algorithm by using our proof procedure with a literal selection strategy that selects conclusions in preference to conditions. Adopting this strategy we obtain the following four search spaces.



<u>**Figure 7.5:** A search space for example 7.1 simulating the LT algorithm</u>



<u>**Figure 7.6:** A search space for example 7.1 simulating the LT algorithm</u>

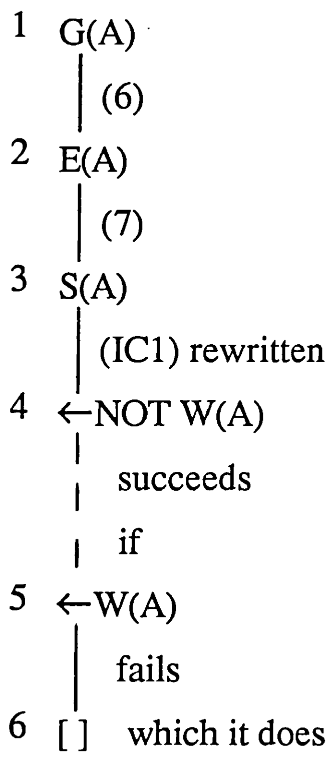


Figure 7.7: A search space for example 7.1 simulating the LT algorithm

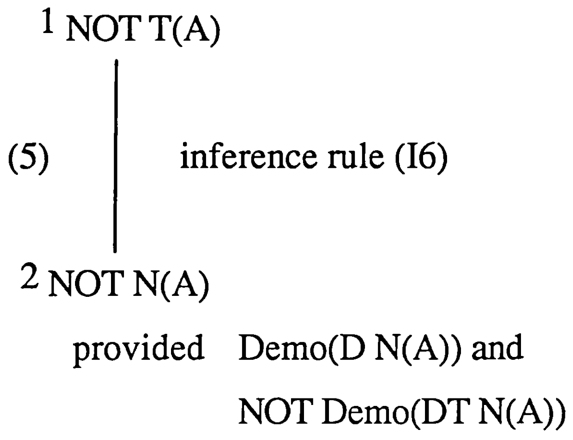


Figure 7.8: A search space for example 7.1 simulating the LT algorithm

The search startegy indicated by the numbers at the clauses allows us to approximate the LT algorithm. As before, where two clauses have the same number the order in which they are investigated does not matter.

The database rules considered and the mgu's computed at clauses numbered 1, 2 and 3 in figures 7.5 and 7.7 correspond exactly to those needed for constructing the sets $pos_{D'',DT}$ and $neg_{D'',D}$. The computations involved in figures 7.6 and 7.8 correspond exactly to those needed for constructing the sets $pos_{D'',D}$ and $neg_{D'',DT}$. The computations involved in the rest of the derivations in figures 7.5 and 7.7 correspond to the instantiation and evaluation of the integrity constraints in the LT algorithm. Notice that in figure 7.7, the work involved from the clause numbered 5 down to the clause numbered 6 corresponds to the right branch of figure 7.4. The left branch of 7.4 duplicates some of the work done in the construction of the set $pos_{D'',DT}$ in the LT algorithm. This inefficiency is avoided in our approach and Decker's, and also in our approximation of the LT algorithm.

This is probably the main difference between the LT algorithm and our simulation of it. Another difference, in general, is the redundancy check that we, like Decker, perform for deleted facts. To avoid this check, our rules for implicit deletions would become quite complex. Consider, for example, rules (I6) and (I7) for implicit deletions described in Chapter 5. The "Demo(D f)" condition in those rules serves two purposes. It checks if "f" is provable in D. Its evaluation also results in instantiating "f", so that the next "NOT Demo" condition becomes ground. If the "Demo" condition were to be removed then some other means of instantiating "f" should be added, and it is this that makes the modification complicated.

Notice that the "Demo" and "NOT Demo" conditions in the derivations in figures 7.6 and 7.8 are not strictly part of the object level derivation. They are metalevel conditions coming from applications of inference rules (I6) and (I7). To simulate the LT algorithm the selection of these two conditions has been deferred. As an alternative, these metalevel conditions can be amalgamated into the object level, and can be activated by the object level selection rule. This effect can be obtained by replacing (I7), for example, by the following:

```
Inconsistent(DT∪IT  p<-c) ←
        Select-literal(p  p<-c  Conclusion) and
        In(f<-b  DT) and
        On(not(p) b) and
        Inconsistent(DT∪IT  not(f)<-(c  and  demo(D  f)  and
                                              not(demo(DT f)))).
```

Here "demo" is a prefix function symbol naming the relation "Demo", and "*and*" is an infix function symbol naming the connective "and". An extra level of reflection rules is needed to execute the "demo" and "not demo" conditions when they are selected. Similar modifications can be made to inference rule (I6).

Our simulation of the algorithms of Decker and Lloyd, Topor, et al provides a good basis for comparing them with one another. In example 7.1, figures 7.1-7.3 simulate Decker's algorithm, and figures 7.5-7.8 simulate the LT algorithm. Figures 7.3 and 7.7, where the update "G(A)" is top clause, are identical. Now consider figures 7.1 and 7.5, where "P(A)" is top clause. The left derivation of 7.1 involves less work than the left derivation of 7.5. The right derivation of 7.1, however, involves more work than the right derivation of 7.5. Figures 7.2 and 7.6 both have "V(A)" as top clause. The work involved in 7.2, simulating Decker, is greater than the work involved in 7.5, simulating LT, because Decker, in effect evaluates the "Demo" and "NOT Demo" conditions, but the LT algorithm does not. Finally, Decker does not need to consider the last update, that is the deletion of "T(A)", as top clause, but the LT algorithm reasons forward from this update as well, as illustrated in figure 7.8. In general, Decker's algorithm is more efficient than the LT algorithm in some cases, and vice versa in others. We believe that it is an advantage of our method that it is not confined to any built-in strategies, and that we can dynamically employ suitable selection and search strategies to obtain the best performance.

We have chosen to embed our method for checking integrity within an input resolution proof procedure in order to facilitate comparison with the other integrity checking algorithms. As a consequence, our proof procedure inherits the inefficiencies of input proof procedures, which are documented in Kowalski [1975], for example. One such inefficiency is illustrated by the following example.

Example 7.2:

D:

(1) $N(x) \leftarrow P(x)$

(2) $R(x\ y) \leftarrow P(x)$ and $Q(y)$

(3) $Q(B)$

(4) $M(A)$

I:

(IC) $\leftarrow N(x)$ and $R(x\ y)$ and $M(y)$

T:

Insert P(A).

Using our method, with the literal selection strategy that simulates the LT algorithm, we can obtain the following search space showing that the updated database satisfies the constraints. (Our method is complete for this example, as will be shown in the next chapter.) We use "," to denote the connective "and".

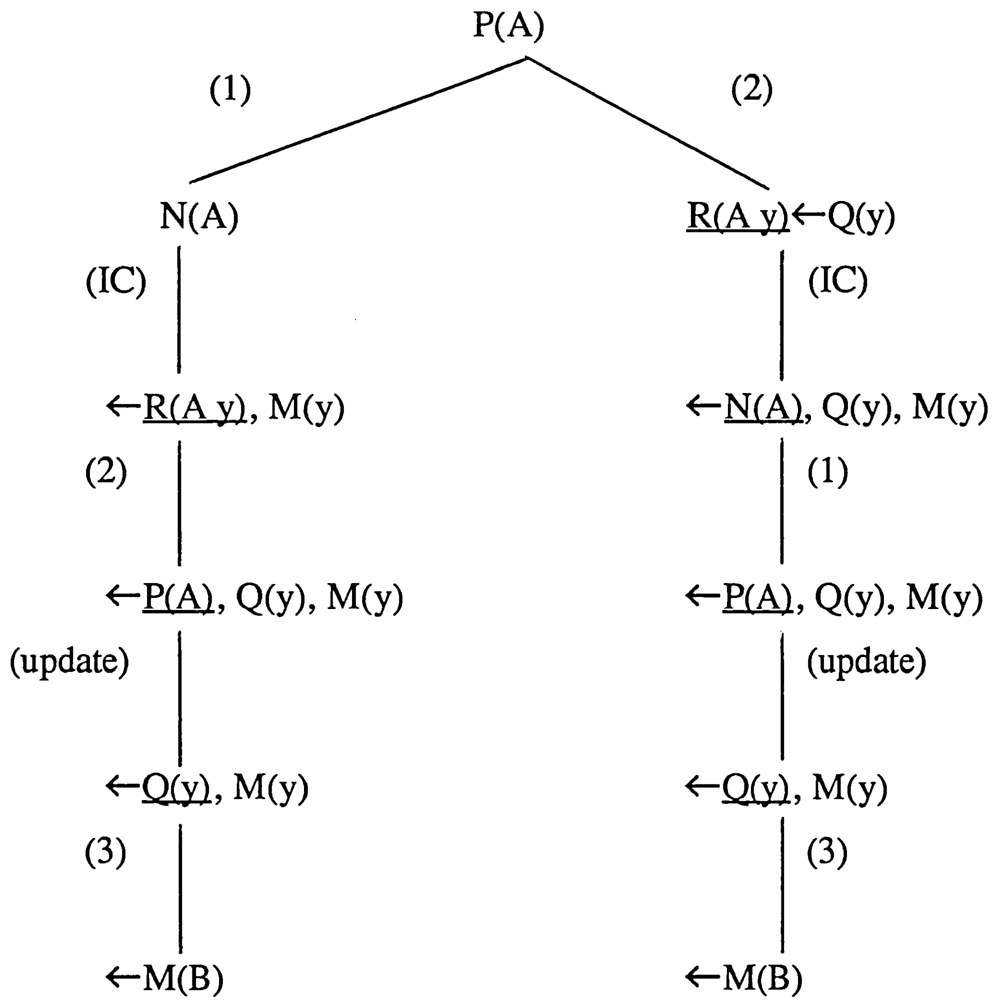


Figure 7.9: A Search Space For Example 7.2 Showing An Inefficiency Of Our Proof Procedure

Notice the duplication of work on the last three clauses of the two derivations. This inefficiency is avoided in the LT algorithm, because of their use of sets, and their method of simplifying constraints only by instantiation. We can also avoid this and other inefficiencies by employing better theorem-proving techniques, such as the connection graph proof procedure (Kowalski [1975]), in our method. □

7.2 Comparison Of The Consistency Method With The Algorithms Of Martens And Bruynooghe, And Bry et al

We have already discussed the relationship between the Consistency method and the algorithms of Decker and Lloyd, Topor, et al. Thus to compare our method with the algorithms of Martens and Bruynooghe, and Bry et al, it is sufficient to compare these last two algorithms with the Decker and LT algorithms.

7.2.1 The Algorithm Of Martens And Bruynooghe

Martens and Bruynooghe [1987] cater for first order formulae as constraints, and function free, range-restricted stratified deductive databases. They impose the function free condition to reduce the possibility of infinite computations. Similar to our method, they adopt the consistency view of constraint satisfaction.

Their method combines an extension of Nicolas' simplification algorithm (Nicolas [1982]), and a modification of Ullman's rule/goal graphs and capture rules for query evaluation (Ullman [1985]). The resulting algorithm is almost identical to Decker's. It involves the same three stages of computation, and exactly the same interleaving of the three stages. It also includes redundancy checks for both added and deleted facts, exactly as in Decker's algorithm. In fact, Martens and Bruynooghe's algorithm can be viewed as a different implementation of Decker's algorithm, where Decker's "Occurs-positive" and "Occurs-negative" facts, and update constraints are described graphically in the modified rule/goal graph of Ullman's.

There are no correctness or completeness results available for the algorithm of Martens and Bruynooghe. The algorithm has not been implemented yet.

7.2.2 The Algorithm Of Bry et al

Bry, Decker and Manthey [1987] cater for function free, range-restricted deductive databases. Their constraints are function free, closed first order formulae that satisfy the restricted quantification condition described in Section 2.2 of this thesis. They only consider updates that add or delete a single ground fact. More general updates are treated by Bry [1987].

Their algorithm, in effect, combines features from both the LT and Decker algorithms. Like these two algorithms, the algorithm of Bry, et al consists of three stages. The first stage is identical to LT's: they compute all the facts that are potentially added or deleted as a result of the update. In this stage they only use the update and the non-atomic rules in the database. The second stage is the simplification of the constraints. At this stage they generate "update constraints" which incorporate Decker's redundancy checks for added and deleted facts. The third stage consists of the evaluation of the "update constraints". The algorithm has been implemented in Prolog. The following simple example helps illustrate the relationship between the three algorithms.

Example 7.3:

Suppose D contains the following rule:

$R(x\ y) \leftarrow P(x)$ and $Q(y)$.

Let the following be a constraint on D:

(IC) $S(x\ y) \leftarrow R(x\ y)$,

and suppose the update is the addition of "P(A)".

To check if DT satisfies (IC),
Decker will proceed as follows:

stage 1: All instantiations ϕ are computed such that
"Demo(DT [R(A y)]ϕ)" and "NOT Demo(D [R(A y)] ϕ)" are true.
stage 2: (IC) is simplified to the form [S(A y)]ϕ for each ϕ computed.
stage 3: All simplified constraints are evaluated.

Lloyd, Topor, et al will proceed as follows:

stage 1: "R(A y)" is computed as a potential addition.
stage 2: (IC) is simplified to "S(A y)$\leftarrow$R(A y)".
stage 3: The simplified constraint is evaluated.

Bry, et al will proceed as follows:

stage 1: As Lloyd, Topor et al.

stage 2: The following "update constraint" is generated

S(A y) ← Demo(DT R(A y)) and NOT Demo(D R(A y)).

stage 3: The "update constraint" is evaluated. □

Thus the main differnce between Bry, et al's and Lloyd, et al's algorithms is the redundancy check that the former does but the latter does not. The main differences between Bry, et al's and Decker's algorithms are firstly that the former does not access the database facts in stage 1, but the latter does, and secondly that the former does the redundancy check in stage 3, but the latter does the check in stage 1. Bry, et al's algorithm shares with the LT algorithm the disadvantage of duplicating in stage 3 some of the work already done in stage 1.

We can approximate Bry, et al's algorithm by our proof procedure with the same literal selection and search strategies that approximate the LT algorithm, while employing the modified versions of inference rules (I2) and (I3), and the modified version of the database management rule (C1), to perform redundancy checks for added facts, as was described in Subsection 7.1.1.

Bry, et al [1987] present a sketch of a proof of the soundness of their method in the case where each transaction consists of the addition or the deletion of a single ground fact. In this case soundness of their method is defined as follows. Let I be a set of constraints on a database D. Let u be an update on D, and let I' be the set of the "update constraints" that result from the processing of u. Then I is satisfied in the updated database DT if and only if I is satisfied in D and I' is satisfied in DT.

7.3 Other Integrity Checking Approaches For Deductive Databases

In this section we present a brief review of the major integrity checking methods that are not descendants of Nicolas' method.

7.3.1 Reiter's Modal Approach

Reiter [1988] concentrates on theoretical rather than practical aspects of integrity constraints. He argues that constraints are statements about the database, not about the world that is modelled by the database. To formalise this notion he uses a first order modal language called KFOPCE, which is due to Levesque [1981]. This is a function free language with equality, with a single modal operator K, that stands for "knows". The database is assumed to consist of function free first order formulae. As an example of the use of KFOPCE consider the constraint, "every employee known to the database must have a known social security number". This is formalised as follows:

$$\forall x\ K[Emp(x)] \rightarrow \exists y\ K[Ss(x\ y)].$$

If, on the other hand, it is only required that every known employee must be known to have a social security number, without the actual number necessarily being known, we will have the following constraint:

$$\forall x\ K[Emp(x)] \rightarrow K[\exists y\ Ss(x\ y)].$$

A similar notion of constraints has been presented by Eshghi and Kowalski [1988] who use first order predicate logic and the metalevel provability relation "Demo" instead of the modal operator "K". In their paper, Eshghi and Kowalski discuss the relationship between their treatment of constraints and ours. Noel [1988] and Small [1988] have also proposed similar approaches to formalising constraints.

In Reiter's approach, a database D satisfies an integrity constraint if the constraint is true in D in the KFOPCE language. To check constraints, Reiter appeals to Levesque's query evaluation in KFOPCE.

Reiter's integrity checking method does not incorporate any simplification, that is it does not exploit the assumption that the database satisfies the constraints prior to the update.

7.3.2 Reiter's Type Checking

In an earlier paper (Reiter [1981]) Reiter proposes a method for checking type constraints in clausal typed databases. In this paper, Reiter remains within first order predicate logic. He considers type constraints of the form

$$R(x_1 \ldots x_n) \rightarrow \text{Type}_1(x_1) \text{ and } \ldots \text{ and } \text{Type}_n(x_n)$$

for every relation R in the database. For example:

$$\text{Father}(x\ y) \rightarrow \text{Human}(x) \text{ and } \text{Human}(y) \text{ and } \text{Male}(x).$$

In addition to the database and the constraints, there is a type database that contains information about the types of the constants in the language, and information about relationships between types, for example

$\leftarrow$Male(x) and Female(x).

Given an update, Reiter uses the type database and constraints, and certain syntactic criteria to transform it into a "reduced type normal form", and then decides whether to accept or reject it according to certain guidelines. It is unnecessary to consider the details here. We only present a simple example.

Example 7.4:

(This is a simplified version of an example given in Reiter [1981].)

Consider the update

(x/Human)(x/Male) [Sister(x) $\leftarrow$ Brother(x)],

which states that for all x which is human and male, if x is a brother then x is a sister. Assume the intuitive argument types for "Brother" and "Sister", namely

(IC1) Brother(x) $\rightarrow$ Human(x) and Male(x)

(IC2) Sister(x) $\rightarrow$ Human(x) and Female(x).

Then the update is transformed into the following:

(u1) (x/Human)(x/Male)(x/Female) [Sister(x)$\leftarrow$Brother(x)]

(u2) (x/Human)(x/Male)(NOT x/Female) [$\leftarrow$Brother(x)].

Very roughly, the idea is that the transformed version of the update represents all the different typings that the update implies. (u2) states that for all x which is of type human and male and which is not of type female, x is not a brother.

Now if we assume that no-one can be both male and female, then (u1) has inconsistent typing. Furthermore, the predicate "Sister" does not occur in (u2). In this case Reiter rejects the update.

The intuition behind the rejection of (u2) is not clear. Reiter reasons that in this case the predicate "Sister" is irrelevant to the original update, and he interprets this as an integrity violation. At best, he argues, there is something questionable about the update. □

In general an update u is rejected if and only if it is inconsistent with the type database (TD) and the type constraints (TC), or if there is a literal l in u which does not occur in any of the formulae of the reduced type normal form of u with consistent typing, or if together with TD and TC, u implies a new type relationship which is not inconsistent with, nor a theorem of TD$\cup$TC, and the user rejects the new type relationship.

It might be of interest to explore the relationship between Reiter's method and a resolution based one. For instance, the following is a derivation in our proof procedure for the above example, with the update as top clause. The input set consists of (IC1), (IC2), the update and the denial
(TD1) $\leftarrow$Male(x) and Female(x).

$$\begin{tikzcd}[column sep=small]
\underline{\text{Sister(x)}}\leftarrow\text{Brother(x)} \arrow[d, dash, "\text{one half of (IC2)}"] \\
\underline{\text{Female(x)}}\leftarrow\text{Brother(x)} \arrow[d, dash, "\text{(TD1)}"] \\
\leftarrow\text{Brother(x) and }\underline{\text{Male(x)}} \arrow[d, dash, "\text{one half of (IC1)}"] \\
\leftarrow\text{Brother(x) and Brother(x)}
\end{tikzcd}$$

Figure 7.10: A derivation for example 7.4 in our proof procedure

The derivation shows that the update is subsumed by a consequence of it together with the type database and constraints. Notice that in this derivation we have adopted a literal selection strategy that prefers conclusions and type literals to others. We conjecture that such a strategy allows us to simulate Reiter's type checking algorithm, in general.

7.3.3 The Asirelli et al Approach

Another integrity checking method proposed is the algorithm of Asirelli, De santis and Martelli [1985]. They consider definite databases only, with restricted forms of integrity constraints. They argue (unconvincingly) that integrity constraints are really the "only if" version of the rules in the database. It is not clear from their paper exactly what class of formulae they allow as constraints.

To check integrity they propose the use of the SLD proof procedure. For example, to check if a constraint

$Q(x) \leftarrow P(x)$

is satisfied in D, they suggest that SLD could generate all substitutions ϕ such that $[P(x)]\phi$ is provable in D, and then SLD could be used again to attempt to prove $[Q(x)]\phi$, for all such ϕ. This approach is equivalent to using SLDNF (or our proof procedure with the constraints as top clauses) to prove the theoremhood of the constraints, and does not incorporate any form of simplification.

7.3.4 The Henschen et al Approach

The last approach we will consider in this chapter is that of Henschen, McCune and Naqvi [1984]. They propose a method based on theorem-proving to preprocess update schemas (to be described shortly) and integrity constraints, to generate tests that are carried out at run-time when updates are actually requested. They cater for relational databases and transactions consisting of single updates. However, as well as additions and deletions of single facts, updates can have more complex forms, such as "delete all facts R(x y) for x=A", or "change all facts R(x y z) for x=A to R(x y B)". For example the update "change all facts Employee(Toy x y) to Employee(Toy x 10000)" requests changing the salaries of all employees in the toy department to 10000. The constraints can be any closed formulae of first order predicate logic expressed in clausal form.

Although their approach is a simplification approach that exploits the satisfaction of the constraints in the database prior to the update, it differs from Nicolas' simplification method in a number of ways. In

Henschen, et al's method the simplification and test generation can be done at the time the database is designed, but in Nicolas' the simplification is done when actual updates are requested. Furthermore, Henschen et al, unlike Nicolas, can deal with updates of the complex form exemplified in the last paragraph. Finally, in Henschen et al's method the assumption of the satisfaction of the constraints in the old database is made explicit by an axiom, which is used in the resolution process. Similar to Nicolas, Henschen et al require the constraints to be domain independent.

The basic idea of Henschen et al's method is as follows. An update schema is an update form rather than an actual update. For example, suppose that "S(x y)" means company x supplies item y. Then the schema "delete S(*A* x)" describes the class of updates that delete all the "S" facts for a given company. Here, "*A*" is a dummy constant. An actual update conforming to this schema will provide an actual value in the place of "*A*".

For each update schema and each constraint, Henschen et al formulate a collection of clauses consisting of the assertion of the satisfaction of the constraint in the old database, a description of the new database in terms of the old one and the update schema, and the denial of the satisfaction of the constraint in the new database. These clauses are then processed by a highly selective process of resolution, paramodulation (Chang and Lee [1973]) and elimination of subsumed clauses. If a contradiction is generated then no update of the form of the schema can possibly violate the constraint. If no contradiction is generated then tests are extracted from the final set of resolvents. These tests would then have to be verified when updates of the form of the schema are actually requested.

Example 7.5:

(This example has been taken from Henschen et al [1984].)

Suppose "S(x y)" means company x supplies item y. Let (IC) be the constraint

S(x Bolts) ← S(x Nuts).

Consider the update schema "change S(*A* Nuts) to S(*B* Nuts)", which is intended to change one of the suppliers of nuts to a different supplier. "*A*" and "*B*" are dummy constants. An actual update conforming to this schema will provide actual values in their places.

Suppose "Sold" expresses the "S" relation in the old database, and "Snew" expresses the "S" relation in the new database. Then the set of clauses formed in this example will include the following:

Sold(x Bolts) ←Sold(x Nuts),

which expresses the satisfaction of (IC) in the old database. The set of clauses will also include clauses such as the following which express the new "S" relation in terms of the old one:

Snew(x y) or x=*A* ← Sold(x y)

Snew(x y) or y=Nuts ← Sold(x y)

Sold(x y) or x=*B* ← Snew(x y),

etc.

Finally, the set will include the denial of the satisfaction of the constraint in the new database, expressed as follows:

Snew(*K* Nuts)

$\leftarrow$Snew(*K* Bolts),

where "*K*" is a Skolem constant (Chang and Lee [1973]). The last clause simply asserts that some company supplies nuts, but not bolts.

The resolution process reduces the set of clauses thus formed to a set consisting of four clauses:

(1)	Sold(x Bolts)	$\leftarrow$ Sold(x Nuts)
(2)		$\leftarrow$ Sold(*B* Bolts)
(3)		$\leftarrow$ Sold(*B* Nuts)
(4)	Sold(*A* Nuts)	

(1) is assumed to hold, so it is ignored. For actual companies replacing "*A*" and "*B*", if any of clauses (2)-(4) can be resolved away a contradiction will be found which will indicate the satisfaction of the constraint after the update. So these clauses are the tests generated in this example. Clause (2), for example, yields the test of checking if "*B*" is already known to supply Bolts. If it is then the constraint continues to be satisfied. Clause (4) yields the test of checking whether "*A*" was not known in the old database to supply nuts. If it was not then the update would be ineffective and the constraint continues to be satisfied.□

The preprocessing of update schemas has the obvious attraction of reducing the work needed for integrity checking when actual updates are requested. It would be interesting to see how our method can be extended to incorporate such preprocessing. We conjecture that this can be done through symbolic processing.

CHAPTER 8

CORRECTNESS AND COMPLETENESS OF THE CONSISTENCY METHOD

Recall that, according to our definition of constraint satisfaction, database D satisfies constraints I if and only if Comp(D)$\cup$I is consistent. In this chapter we discuss the correctness and completeness of our integrity checking method (as formalised by the inference rules in Sections 5.1 and 5.2.2) relative to this specification.

Recall also, that our proof procedure is identical to SLDNF whenever the top clause is a denial. Therefore it is as correct and as complete as SLDNF when the integrity constraints are chosen as top clauses. SLDNF has been proved correct in general by Clark [1978]. It has been proved complete in a number of special cases (Clark [1978], Jaffar, Lassez and Lloyd [1983], Barbuti and Martelli [1986], Kunen [1987] and [1988], Cavedon and Lloyd [1987] and Shepherdson [1988]).

Clark [1978], for example, has proved SLDNF complete for hierarchical databases and "allowed" goals. A goal is <u>allowed</u> if and only if every variable that occurs in a negative condition of the goal, also occurs in a positive condition, such that the positive condition generates candidate ground substitutions for the variable. (Note that this is Clark's definition of "allowed" goals, and is slightly different from the "allowed" condition of Lloyd and Topor [1986] and Topor and Sonenberg [1988].)

Jaffar, Lassez and Lloyd [1983] have proved the following result for the completeness of the negation as failure rule:
Let D be a definite database and G a negation-free goal. If G is a logical consequence of Comp(D), then for all "fair" computation rules R, there is a finitely failed SLD search space for D$\cup${G} via R. A computation rule is <u>fair</u> (Lassez and Maher [1984]) if in each infinite derivation every literal in the goal is eventually selected.

Shepherdson [1988] has slightly extended these results to prove SLDNF complete for

(i) definite databases and ground goals, for all fair computation rules, and

(ii) range-restricted hierarchical databases and range-restricted goals, for all computation rules.

A very similar result to (ii), above, was proved earlier by Lloyd and Topor [1986].

Shepherdson [1988] has also proved SLDNF ϕ-complete for definite databases and allowed goals, for all fair computation rules. SLDNF is ϕ-complete for a goal G if whenever for a substitution ϕ, [G]ϕ is logically inconsistent with Comp(D), there is an SLDNF refutation of D$\cup${G} (with answer including ϕ).

Note that when the database is definite and range-restricted, every range-restricted goal is an allowed goal, because:
(i) in a range-restricted goal, by definition of range-restriction, every variable that occurs in a negative condition also occurs in a positive

one, and

(ii) when the database is definite and range-restricted, if a goal is resolved on a positive condition, P, say, then any variable of P which is not grounded in the resolution step will occur in a positive condition in the resolvent.

Finally, by Proposition 4.2, if Comp(DT) is consistent and for all constraints (IC) in IT there is a finitely failed search space with (IC) as top clause, then Comp(DT)$\cup$IT is consistent and therefore database DT satisfies IT.

The discussion so far addresses the correctness and completeness of our proof procedure with denials as top clauses, and thus the correctness and completeness of the Consistency method when the updates are integrity constraints.

In the remainder of this chapter, we concentrate on the more complicated case where the top clauses are not denials. This corresponds to the cases where the updates are additions, deletions or modifications of database rules. Recall that "modification" updates are treated as combinations of additions and deletions. So without loss of generality, in the rest of this chapter, we assume that transaction T consists of a set, al, of additions, and a set, dl, of deletions of deductive rules, unless otherwise stated.

8.1 Correctness

Proposition 8.1: (Correctness of the simplified proof procedure described in Subsection 3.1.2)

Let S be the union of a set DT of definite clauses and a set IT of negation-free denials. Let C_0 be a clause in DT. If there is a refutation by means of our proof procedure with S as input set and C_0 as top clause, then Comp(DT)$\cup$IT is logically inconsistent.

Proof:

The theorem follows from the more general fact that if C_0, C_1,..., C_n is a derivation, then C_n, for all $n \geq 0$, is a logical consequence of Comp(DT)$\cup$IT. This, in turn, follows from the fact that each C_{i+1} is obtained from C_i and an input clause by resolution, and resolution is correct (Robinson [1965]). □

To prove the correctness of our proof procedure in the general case we first need to prove the correctness of the extended resolution step.

Proposition 8.2: (Correctness of the extended resolution step)

Let clauses

C_1: $\underline{\text{NOT } P(t^*)} \leftarrow$Conds1

and C_2: (L)$\leftarrow\underline{\text{NOT } P(r^*)}$ and Conds2

be logical consequences of Comp(DT)$\cup$IT. Conds1 and Conds2 are (possibly empty) conjunctions of literals, L is a literal which may or may not be present, t^* and r^* are vectors of terms, and P is any predicate symbol. Let C be obtained by the extended resolution of C_1 and C_2 on their underlined literals. Then C is a logical consequence of Comp(DT)$\cup$IT.

Proof:

C is the clause

$[\ (L) \leftarrow \text{Conds1 and Conds2}\ ]\phi$

where ϕ is the mgu of $P(t^*)$ and $P(r^*)$.

C_1 and C_2 are equivalent to

C_1': $\leftarrow$Conds1 and $\underline{P(t^*)}$

C_2': $\underline{P(r^*)}$ or (L) $\leftarrow$Conds2.

Now C is the resolvent, by the standard resolution step, of C_1' and C_2' on their underlined literals. Since standard resolution is correct (Robinson [1965]), C is a logical consequence of Comp(DT)$\cup$IT. □

Theorem 8.1: (Correctness of the general proof procedure -the inference rule version- formalised in Sections 5.1 and 5.2.2)

Let S be the union of a set DT of deductive rules and a set IT of denials. Let C_0 be either a clause in S or the negation of a fact that fails finitely from DT by SLDNF. If there is a refutation by means of our proof procedure with C_0 as top clause and S as input set, then Comp(DT)$\cup$IT is inconsistent.

Proof:

We prove more generally, by induction on n, that if C_0, C_1, ..., C_n is a derivation by means of our proof procedure, then C_n is a logical consequence of Comp(DT)$\cup$IT.

The base case:
C_0 is obviously a consequence of Comp(DT)$\cup$IT if C_0 is a clause in S. By the correctness of SLDNF, C_0 is also a consequence of Comp(DT), and therefore of Comp(DT) $\cup$IT, if it is the negation of a fact that fails finitely from DT.

The inductive case:
Suppose C_{n-1} is a logical consequence of Comp(DT)$\cup$IT. Then if C_n exists, it is obtained from C_{n-1} by one of the following rules (the rule names refer to their names in Chapter 5):
(1) standard resolution, (I2),
(2) extended resolution, (I2),
(3) negation by failure, (I3),
(4) inference rules for implicit deletions, (I6) and (I7).

In cases (1)-(3) C_n is clearly a logical consequence of Comp(DT) $\cup$IT, because each of these steps for deriving C_n from C_{n-1} is logically correct: standard resolution is correct (Robinson [1965]), extended resolution is correct (Proposition 8.2), and the negation as failure step is correct (by correctness of SLDNF (Clark [1978])).

It remains to show that if C_n is obtained from C_{n-1} by (I6) or (I7), then C_n is a logical consequence of Comp(DT)$\cup$IT. In both cases C_n has the form

NOT f $\leftarrow$ c

for some fact f, and some conjunction of literals c, such that the condition

NOT Demo(DT f)

holds. But this condition holds if and only if f fails finitely from DT. But then, by the correctness of negation as failure, "NOT f" is a logical consequence of Comp(DT)$\cup$IT and therefore so is "NOT f $\leftarrow$ c". □

In this argument we have assumed that the condition "NOT Demo(DT f)" is evaluated before the clause "NOT f $\leftarrow$c" is derived, as it would be if the conditions in (I6) and (I7) were executed in Prolog order. But the logical content of these inference rules is independent of the choice of safe computation rules for evaluating their conditions. The above proof therefore implies the correctness of the method for any safe computation rule, such as that employed in our simulation of the LT algorithm in example 7.1.

To end this section we prove the correctness of our rewriting of the integrity constraints as described in Section 2.2. We consider two cases:

(a) the simpler rewriting of constraints that are of the form

A_1 or ... or $A_n \leftarrow B_1$ and ... and B_m,

where the A_i and the B_i are atoms, and

(b) the more complicated rewriting of constraints that are more general formulae of first order predicate logic.

For case (a) we prove the following proposition:

Proposition 8.3:

Let (IC) be a constraint on DT of the form

$$A_1 \text{ or... or } A_n \leftarrow B_1 \text{ and ... and } B_m, \qquad m,n \geq 0,$$

where the A_i and the B_i are atoms. Let (IC') be the rewritten version of (IC):

(IC') $\leftarrow B_1$ and ... and B_m and NOT A_1 and ... and NOT A_n.

Then if there is a refutation by means of our proof procedure with DT$\cup$\{(IC')\} as input set, and a clause u associated with an update in T as top clause, then Comp(DT)$\cup$\{(IC)\} is inconsistent.

Proof:

If such a refutation exists, then by correctness of our method (Theorem 8.1) Comp(DT)$\cup$\{(IC')\} is inconsistent. Thus Comp(DT)$\cup$\{(IC)\} is inconsistent. □

To prove a correctness result for case (b) we need two results proved by Lloyd and Topor [1984]. We give a specialised version of these results necessary for our purposes.

Let DT" be a set of deductive rules, and let V be a closed first order formula of predicate logic. Let DT'=DT"$\cup${A$\leftarrow$V}, where A is a nullary predicate symbol which does not occur anywhere in DT" or V. Let DT be DT' in which the rule "A$\leftarrow$V" is transformed into a set of deductive rules, as described in Lloyd and Topor [1984], and exemplified in Section 2.2 of this thesis. Suppose C is a closed formula which contains predicate symbols that occur in DT' only. Then Lloyd and Topor [1984] have proved the following:

(1) If C is a logical consequence of Comp(DT), then C is a logical consequence of Comp(DT'). (Comp(DT') is defined as an extension of the completion of a deductive database. The details need not concern us.)

(2) A is a logical consequence of Comp(DT') if and only if V is a logical consequence of Comp(DT").

We use (1) and (2) to prove the correctness of our rewriting of constraints in case (b):

Proposition 8.4:

Let W be a closed first order formula, and let V=(NOT W). Let DT, DT', DT" and A be as explained above. If there is a refutation by means of our proof procedure with DT$\cup${$\leftarrow$A} as input set and a clause associated with an update in T as top clause, then Comp(DT")$\cup${W} is inconsistent.

Proof:

If such a refutation exists then by correctness of our method (Theorem 8.1) Comp(DT)$\cup$\{$\leftarrow$A\} is inconsistent. So A is a logical consequence of Comp(DT). By (1), above, A is a logical consequence of Comp(DT'). Then by (2), above, "NOT W" is a logical consequence of Comp(DT"). Therefore Comp(DT")$\cup$\{W\} is inconsistent. □

8.2 Completeness

Recall that our method of integrity checking is based on the assumption that D satisfies the constraints prior to the transaction. With this assumption, we believe that our method is as complete as SLDNF. That is, if there is an SLDNF refutation with an integrity constraint as top clause then for all safe computation rules R (or all fair computation rules R, whenever SLDNF requires fairness of computation rules) there is a refutation via R by means of our proof procedure with a clause associated with an update as top clause.

We shall prove the completeness of our method for the special, but non-trivial case, discussed in Chapter 3, where the database is definite, the integrity constraints are negation-free denials, and the transaction consists only of additions. The proof procedure in this case is non-trivial, because as shown earlier it is neither a special case of SL (Kowalski and Kuehner [1971]), nor a special case of SLD (Apt and van Emden [1982]). In Chapter 3 we showed how our proof procedure extends both SL and SLD.

SLD has been proved complete by Clark [1979] and Hill [1974]. Thus if DT is inconsistent with a constraint (IC) then for any computation rule R there exists an SLD refutation of $DT \cup \{(IC)\}$ via R. In this section we prove the completeness of our method for this special case by proving that the method is as complete as SLD.

Since Comp(D) is consistent with the constraints, any inconsistency after the transaction must involve at least one of the updates. Therefore, in any SLD refutation with a constraint as top clause, showing the violation of the constraint in the updated database, one of the input clauses contributing to the refutation must be an update. Thus to prove the completeness of our method relative to SLD it is sufficient to prove the following.

Theorem 8.2:

Let S be a set of definite clauses and negation-free denials, and let (IC) be a negation-free denial in S. Suppose there is an SLD refutation F of $S \cup \{(IC)\}$. Then, for every computation rule R and for every input clause C contributing to the refutation, there is a refutation via R with input set S by means of our proof procedure with C as top clause. (An input clause C contributes to a derivation $C_0, C_1, ..., C_n$ if and only if for some i, $0<i<n$, C_{i+1} is obtained by the resolution of C and C_i).

Proof:

The proof is obtained by the standard techniques of:

(1) first transforming F to a variable-free SLD refutation F' of $S' \cup \{(IC')\}$, where (IC') is a ground instance of (IC), each clause in S' is a ground instance of a clause in S, and a ground instance C' of C contributes to F',

(2) transforming F' to a ground refutation F'' by means of our proof procedure with C' as top clause, and finally

(3) applying the Lifting Lemma (Chang and Lee [1973]) to obtain the desired refutation F* (isomorphic to F'') of $S \cup \{(IC)\}$ with top clause C.

Parts (1) and (3) are well known techniques, often employed in proofs of (relative) completeness of resolution proof procedures. We will discuss these parts briefly first, and then concentrate on part (2), which is specific to Theorem 8.2.

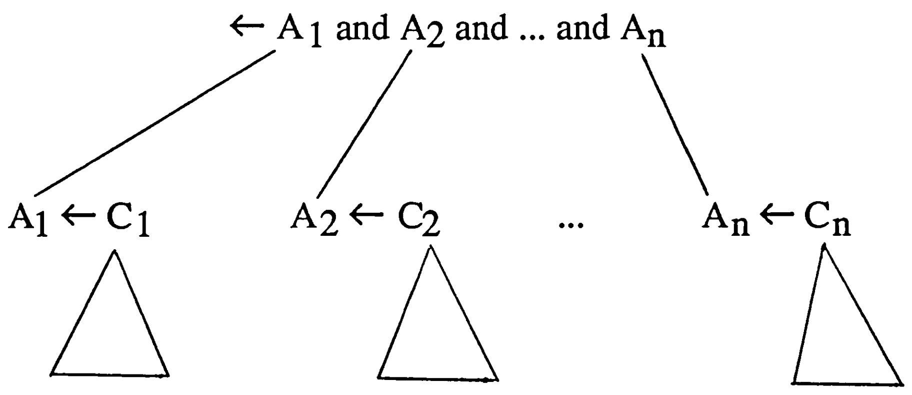
Part (1):
F can be transformed to a variable-free refutation F' as follows. First apply the mgu's generated in F (that is the mgu's of the resolution steps that construct F) to the clauses in F, and to the input clauses that contribute to F. This gives a refutation (possibly containing variables) which uses only propositional resolution (that is resolution steps that do

not involve any instantiation of variables). Now replace all remaining variables systematically by distinct constants. A variable x must be replaced by the same constant symbol wherever it appears in the refutation and in the input clauses that contribute to the refutation.

Part (3):
The Lifting Lemma states that if C_1' and C_2' are instances of C_1 and C_2, respectively, and if E' is a resolvent of C_1' and C_2', then there is a resolvent E of C_1 and C_2, such that E' is an instance of E.

So by this Lemma if F" is a refutation with input set $S' \cup \{(IC')\}$ and top clause C', then, by inductively applying the Lifting Lemma, there is a refutation F* with input set $S \cup \{(IC)\}$ and top clause C.

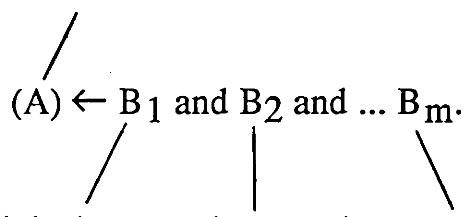
Part (2):
First, transform F' into the form of an and-tree TR: the top node of TR is the denial (IC') with subtrees for every condition A_i in (IC'). These subtrees are joined to the top node by arcs connecting the conditions A_i with the conclusions of the rules in S' with which they resolve in F'. The top of TR then has the form



where the C_i denote (possibly empty) conjunctions of literals. By an induction argument the subtrees with top node "$\leftarrow C_i$" can be constructed similarly. Notice that the and-tree is actually a special form of connection graph (Kowalski [1975]) linking occurrences of clauses in $S' \cup \{(IC')\}$. The same clause can occur in different subtrees. Every occurrence of an atom in TR is linked to only one other occurrence of an atom. Thus every link represents a possible resolution between the clauses.

The original SLD refutation F' is one particular traversal of TR, starting with the top node (IC'). The desired refutation F" is just an alternative traversal of TR starting with an occurrence of C' instead.

F" can be constructed by course of values induction on the number of arcs in TR: Suppose C' occurs at a node N in TR of the form



Each atom in N is linked to exactly one other atom in a node in TR. N may have zero or more conditions, and may or may not have any conclusion. Suppose that computation rule R selects an occurrence of an atom (conclusion or condition) in C and that B is the ground instance of this atom occurrence in C'. Let C" be the resolvent on B of C' with the clause to which B is linked in TR. Replace the two parent nodes in TR by the one node which is the resolvent, letting the resolvent inherit all the unselected links from the parent clauses (exactly as in the connection graph proof procedure (Kowalski [1975])). The resulting tree TR' has exactly one less link than TR. By induction hypothesis for every computation rule R' there exists a refutation F"' of $S' \cup \{(IC')\}$

with top clause C". Choose R' to be the computation rule that selects from a clause C'_n ending a derivation C'_0, ..., C'_n of $S' \cup \{(IC')\}$ the ground instance of the same atom occurrence that R selects from the clause C_n ending a derivation C, C_0, ..., C_n of $S \cup \{(IC)\}$, where C'_0, ..., C'_n is the ground derivation isomorphic to C_0, ..., C_n. The desired refutation F" is then just C' followed by F"'. □

It is possible to extend the proof of Theorem 8.2 to prove completeness of our method relative to SLDNF for a more general case where the database is still definite and all the updates are additions, but where integrity constraints may have negative conditions.

Suppose there is an SLDNF refutation of $DT \cup \{(IC)\}$, for a constraint (IC) in I. Since D satisfies the constraints, one of the updates in the transaction must contribute to this refutation (that is, an update must be an input clause used in the solution of a positive condition of (IC)). If this were not the case then there would be some ground instance "NOT A", say, of a negative condition of (IC) such that "NOT A" is provable by SLDNF in DT, but not in D. Therefore there is a finitely failed SLD search space for $DT \cup \{\leftarrow A\}$. But the search space for $DT \cup \{\leftarrow A\}$ includes the search space for $D \cup \{\leftarrow A\}$ because DT includes D and both consist of definite clauses. Therefore there must be a finitely failed SLD search space for $D \cup \{\leftarrow A\}$, which implies that D violates the constraint (IC) contrary to assumption. Thus to prove our method as complete as SLDNF in this case it is sufficient to prove the following.

Theorem 8.3:

Let S be a set of definite clauses and denials, and let (IC) be a denial in S. Suppose there is an SLDNF refutation F of $S \cup \{(IC)\}$. Then for every safe computation rule R and every input clause C contributing to the refutation there is a refutation via R by means of our proof procedure with C as top clause and S as input set, provided that we employ a fair computation rule in the subproofs of negative conditions.

Proof:

The SLDNF refutation F without its auxiliary negation as failure subproofs has the same structure as an SLD refutation, except for the solution of negative conditions. If for every auxiliary subproof of a ground negative condition "NOT A" in F we add a negative fact "NOT A" to the database, and then rename such negated atoms systematically by means of new and distinct positive ground atoms, we transform F into a form to which Theorem 8.2 applies. Thus we can construct a refutation via any computation rule by means of our proof procedure with C as top clause. In particular, we can construct such a refutation via any computation rule that selects the renamed conditions only when they are ground. If we now undo the renaming and restore the auxiliary negation as failure subproofs, we obtain the desired refutation.

Moreover, we obtain proofs of the negative conditions via any fair computation rule, for the following reason. Suppose there is an SLDNF refutation of $S \cup \{\leftarrow \text{NOT } A\}$, where A is a ground atom. Then there is an SLD finitely failed search space for S$\cup${←A}. Then, by (a specialisation of) the correctness result of SLDNF (Clark [1978]), "←A" is a logical consequence of Comp(S). So, by the Jaffar, Lassez and Lloyd [1983] result described in (b) at the beginning of this chapter, for every fair computation rule there is a finitely failed SLD search space for S$\cup${←A}, and thus there is an SLDNF refutation of S$\cup${←NOT A}. Furthermore, as explained in Chapter 4, our proof procedure is equivalent to SLDNF when the top clause is a denial.

□

As well as ensuring the completeness of the method, the special cases covered in Theorems 8.2 and 8.3 have another major advantage. The proof procedure in these cases can be implemented with efficiencies comparable to that of Prolog implementations, as described in Chapter 6.

We have not yet proved the analogue of Theorem 8.3 for the general case. The main difficulty comes from the need to generalise the and-tree in the proof of Theorem 8.2 to a tree including auxiliary proofs of negation as failure. These auxiliary proofs are not simple and-trees but include entire finitely failed search spaces. This suggests that we may be able to deal with this case by converting finitely failed search spaces into direct proofs of negative conditions using Comp(D) as in Clark's proof of the correctness of SLDNF (Clark [1978]).

The correctness of our proof procedure justifies concluding that if we obtain a refutation with our proof procedure then the updated database violates the constraints. Completeness justifies concluding, as a corollary, that if our proof procedure fails finitely with all the updates as top clauses then integrity is maintained in the updated database.

CHAPTER 9

CONCLUSION

In this thesis we have described the Consistency method for checking integrity of deductive databases, and a new underlying proof procedure. The Consistency method exploits, for efficiency, the assumption that the constraints are satisfied in the database prior to the transaction. It reasons forward from the transaction, and thus concentrates on the effects of the updates, and ignores what remains unchanged. The new proof procedure is an extension of Prolog, and allows forward as well as backward reasoning.

We described the proof procedure and the Consistency method first in two simplified cases and then in the general case. We, then, presented a logical formalisation of the proof procedure and the Consistency method, and described an implementation in Prolog based on this formalisation. We also described an alternative and more efficient implementation for a special case.

The proof procedure was compared with the SL, SLD and SLDNF proof procedures. It was shown to be extensions of the latter two. It was also shown to differ from SL by adopting a more liberal literal selection strategy. Thus although our proof procedure is not as general as SL, it is not a special case of SL either.

The Consistency method was compared with other existing algorithms for integrity checking in deductive databases. It was shown to approximate the algorithms of Decker, Lloyd, Topor, et al, Martens and Bruynooghe and Bry, et al. We discussed in detail the relationship between the Consistency method and the first two algorithms, and that between these algorithms and the latter two.

Finally, we proved our method correct in general, and complete in certain special cases.

This work can be extended in the future in various directions, for example:

(1) Neither we nor any other researchers in deductive database integrity have yet adequately addressed the problem of dealing with constraints involving aggregates. Heath [1988] has done some preliminary, but promising, work on extending our method to deal with aggregate constraints. The subject deserves to be further investigated. Consider, for example, the following constraint:

"The maximum number of students is 100."

This can be formalised by the rule:

$n \leq 100 \leftarrow s=\{x: Student(x)\}$ and $Size(s\ n)$.

(The set construction can be implemented using Prolog's "isall" operator.)

Intuitively, this constraint must be evaluated when and only when a new student is to be recorded in the database. Moreover, we should not have to construct the complete set of all students every time we evaluate the constraint. Forward reasoning, via resolution, from updates that add new students does not achieve either of these requirements.

(2) There is scope for improving the efficiency of our implementation. It would be interesting to see if the alternative implementation presented in Section 6.4 can be extended to cover more general cases. Furthermore, since our approach is based on general theorem-proving techniques, it can benefit from existing approaches for improving the efficiency of the underlying linear proof procedure (Kowalski [1975]).

(3) Eshghi and Kowalski [1988] propose the use of abduction to replace reasoning with negation as failure. For example, a rule

$A \leftarrow \text{NOT } B$

can be rewritten as

$A \leftarrow B'$,

together with a constraint

$\leftarrow B \text{ and } B'$,

where B' is an abducible fact, that is a fact that can be assumed provided its assumption does not cause any inconsistencies. Thus to prove A, fact B' can be assumed provided B is not provable.

Abduction provides an interesting alternative to negation as failure, and for implementing default reasoning in general. The consequences of replacing negation as failure by abduction in our proof procedure are worth investigating, especially since checking abductive assumptions for consistency with constraints is an essential feature of the abductive approach. On the one hand, replacing negation by failure by abduction would influence our integrity checking method. On the other hand, to be efficient, abduction needs an efficient integrity checking method.

(4) Finally, the identification of the largest class of problems for which our method can be proved complete is an outstanding theoretical issue.

REFERENCES

Apt, K. B., Blair, H. and Walker, A. [1988]:
"Towards a Theory of Declarative Knowledge", in "Foundations of Deductive Databases and Logic Programming", Minker J. [Ed.], Morgan Kaufmann, 89-148.

Apt, K. R. and van Emden, M. H. [1982]:
"Contributions to the Theory of Logic Programming", JACM, 29, 3, (July), 841-862.

Asirelli, P., De Santis, M. and Martelli, M. [1985]:
"Integrity Constraints in Logic Databases", J. Logic Programming, volume 2, number 3, 221-232.

Barbuti, R. and Martelli, M. [1986]:
"Completeness of the SLDNF-Resolution for a Class of Logic Programs", Proc. 3rd International Conference on Logic Programming, London, U.K., Springer-Verlag, 600-614.

Bernstein, P. A., Blaustein, B. T. and Clarke, E. M. [1980]:
"Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data", Proc. 6th VLDB, Montreal, Canada, 126-136.

Bowen, K. A. and Kowalski, R. A. [1982]:
"Amalgamating Language and Metalanguage in Logic Programming", in "Logic Programming", Clark, K. L. and Tarnlund, S.-A. [Eds.], Academic Press, 153-172.

Bry, F. [1987]:
"Maintaining Integrity of Deductive Databases", Internal Report KB-45, ECRC, Munich, July.

Bry, F., Decker, H. and Manthey, R. [1987]:
"A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases", Technical Report KB-16, ECRC, Munich, October, 7.

Bry, F. and Manthey, R. [1986]:
"Checking Consistency of Database Constraints: a Logical Basis", Proc. 12th VLDB, Kyoto, Japan, 13-20.

Cavedon, L. and Lloyd, J. W. [1987]:
"A Completeness Theorem for SLDNF-Resolution", Computer Science Department, University of Bristol. To appear in J. Logic Programming.

Chang, C. L. and Lee, R. C. T. [1973]:
"Symbolic Logic and Mechanical Theorem Proving", Academic Press.

Charniak, E. and McDermott, D. [1985]:

"Introduction to Artificial Intelligence", Addison-Wesley Publication Company.

Clark, K. L. [1978]:

"Negation as failure", in "Logic and Data Bases", Gallaire, H. and Minker, J. [Eds.], Plenum Press, 293-322.

Clark, K. L. [1979]:

"Predicate Logic as a Computational Formalism", Research Report 79/59, Department of Computing, Imperial College of Science and Technology, University of London.

Cox, P. T. and Pietrzykowski, T. [1986]:

"Causes for Events: Their Computation and Applications", Proc. CADE-86, Siekmann, J. H. [Ed.], Springer-Verlag Lecture Notes in Computer science, 608-621.

Decker, H. [1986]:

"Integrity Enforcement on Deductive Databases", Proc. EDS 86, Charleston, South Carolina, U.S.A., 271-285.

Decker, H. [1987]:

"The Range Form or How to Avoid Floundering", Internal Report KB-26, ECRC, Munich.

Eshghi, K. [1988]:

"Abductive Planning with Event Calculus", Proc. 5th

International Conference on Logic Programming, Seattle, U.S.A.

Eshghi, K. and Kowalski, R. A. [1988]:
"Abduction Through Deduction", Department of Computing, Imperial College of Science and Technology, University of London, March.

Goebel, R., Furukawa, K. and Poole, P. [1986]:
"Using Definite Clauses and Integrity Constraints as the Basis for a Theory Formation Approach to Diagnostic Reasoning", Proc. 3rd International Conference on Logic Programming, London, U.K., Springer-Verlag, 211-222.

Hammond, P. and Sergot, M. [1984]:
"APES: Augmented Prolog for Expert Systems", Logic Based Systems Ltd, Surrey.

Heath, A. [1988]:
Personal Communication.

Henschen, L. J., McCune, W. W. and Naqvi, S. A. [1984]:
"Compiling Constraint-Checking Programs from First-order Formulas", in "Advances in Database Theory", volume 2, Gallaire, H., Minker, J. and Nicolas, J. M. [Eds.], Plenum Press, 145-169.

Hill, R. [1974]:
"LUSH-Resolution and its Completeness", DCL Memo 78,

Department of Artificial Intelligence, University of Edinburgh.

Jaffar, J. ,Lassez, J.- L. and Lloyd, J. W. [1983]:
"Completeness of the Negation as Failure Rule", IJCAI-83, Karlsruhe, 500-506.

Kakas, A. C. [1987]:
"Knowledge Assimilation", M.Sc. Thesis, Department of Computing, Imperial College of Science and Technology, University of London.

Kowalski, R. A. [1975]:
"A Proof Procedure Using Connection Graphs", JACM, volume 22, number 4, 572-595.

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

Kowalski, R. A. and Kuehner, D. [1971]:
"Linear Resolution with Selection Function", Artificial Intelligence 2, 227-260.

Kowalski, R., Sadri, F. and Soper, P. [1987]:
"Integrity Checking in Deductive Databases", Proc. 13th VLDB, Brighton, England, 61-69.

Kunen, K. [1987]:
"Negation in Logic Programming", J. Logic Programming,

volume 4, number 4, 289-308.

Kunen, K. [1988]:
"Signed Data Dependencies in Logic Programs", to appear in J. Logic Programming.

Lassez, J.- L. and Maher, M. J. [1984]:
"Closures and Fairness in the Semantics of Programming Logic", Theoretical Computer Science 29, 167-184.

Levesque, H. J. [1981]:
"A Formal Treatment of Incomplete Knowledge Bases", Ph.D. thesis, Department of Computer Science, University of Toronto; also available as Technical Report, No. 3, Fairchild Laboratory for Artificial Intelligence Research, Palo Alto, California. A shorter version is available as "The Interaction With Incomplete Knowledge Bases: A Formal Treatment", IJCAI-81, Vancouver, Canada, volume 1, 240-245.

Lloyd, J. W. [1987]:
"Foundations of Logic Programming", Springer Verlag, Symbolic Computation Series. This is the second extended edition of the book that was published in 1984.

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

Lloyd, J. W. and Topor, R. W. [1984]:
"Making Prolog More Expressive", J. Logic Programming, volume 1, number 3, 225-240.

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

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

Martens, B. and Bruynooghe, M. [1987]:
"Integrity Constraints in Deductive Databases Using a Rule/Goal Graph", Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3030 Heverlee, Belgium. Also in Proc. EDS 88, Virginia, U.S.A., 297-310.

McCarthy, J. and Hayes, P. J. [1969]:
"Some Philosophical Problems from the Standpoint of Artificial Intelligence", Machine Intelligence 4, Edinburgh University Press, New York, 463-502.

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

Nicolas, J. M. and Gallaire, H. [1978]:

"Data Base: Theory vs. Interpretation", in "Logic and Data Bases", Gallaire, H. and Minker, J. [Eds.], Plenum Press, New York, 33-54.

Noel, P. [1988]:
"Semantic Constraints in First Order Theories: a Definition and its Applicability", Department of Computer Science, University of Manchester.

Poole, D. L. [1987]:
"A Logical Framework for Default Reasoning", CS-87-59, Department of Computer Science, University of Waterloo.

Reiter, R. [1981]:
"On the Integrity of Typed First Order Data Bases", in "Advances in Database Theory", volume 1, Gallaire, H., Minker, J. and Nicolas, J. M. [Eds.], Plenum Press, 137-157.

Reiter, R. [1988]:
"On Integrity Constraints", Department of Computer Science, University of Toronto, Toronto, Ontario, M5S 1A4, Canada. To appear in "Theoretical Aspects of Reasoning about Knowledge II, Asilomar, Ca., March 6-9.

Robinson, J. A. [1965]:
"A Machine-Oriented Logic Based on the Resolution Principle", JACM, volume 12, number 1, 23-41.

Sadri, F. [1987]:

"Three Recent Approaches to Temporal Reasoning", in "Temporal Logics and Their Applications", Galton, A. [Ed.], Academic Press, 121-168.

Sadri, F. and Kowalski, R. A. [1988]:

"A Theorem-Proving Approach to Database Integrity", in "Foundations of Deductive Databases and Logic Programming", Minker, J. [Ed.], Morgan Kaufmann, 313-362.

Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P. and Cory, H. T. [1986]:

"The British Nationality Act as a Logic Program", CACM, volume 29, number 5, May 1986, 370-386.

Shepherdson, J. C. [1988]:

"Negation in Logic Programming", in "Foundations of Deductive Databases and Logic Programming", Minker, J. [Ed.], Morgan Kaufmann, 19-88.

Small, C. [1988]:

"Guarded Default Databases: an Approach to the Control of Incomplete Information", Ph.D. Thesis, Birkbeck College, University of London.

Smullyan, R. M. [1968]:

"First-Order Logic", Springer Verlag.

Soper, P. J. R. [1986]:

"Integrity Checking In Deductive Databases", M.Sc. Thesis, Department of Computing, Imperial College of Science and Technology, University of London.

Topor, R. W., Keddis, T. and Wright, D. W. [1985] :

"Deductive Database Tools", Technical Report 84/7 (Revised August 23, 1985), Department of Computer Science, University of Melbourne, Parkville, Vic. 3052. Shorter version in Australian Computer Journal, volume 17, number 4, 1985, 163-173.

Topor, R. W. and Sonenberg, E. A. [1988]:

"On Domain Independent Databases", in "Foundations of Deductive Databases and Logic Programming", Minker J. [Ed.], Morgan Kaufmann, 217-240.

Ullman, J. D. [1983]:

"Principles of Database Systems", Pitman, London, second edition.

Ullman, J. D. [1985]:

"Implementation of Logical Query Languages for Databases", ACM Transactions on Database Systems, volume 10, number 3, 289-321.

Weyhrauch, R. [1980]:

"Prolegomena to a Theory of Mechanized Formal Reasoning", Artificial Intelligence 13, 133-170.