Integrity Constraints in Deductive Databases

Subrata Kumar Das

A thesis submitted in fulfillment of the
requirements for the degree of

Doctor of Philosophy

Heriot-Watt University

Department of Computer Science

May 1990

Dedicated to my parents
whose wholehearted wish it is
to have their son's name embellished by 'Dr'.

Contents

Declaration

The work presented in this dissertation was carried out by myself, except where due acknowledgement is made. This dissertation has not been submitted by me for any other degree at this or any other university. However, some material presented in this document has been or will be published as follows:

1. S.K.Das and M.H.Williams, "A path finding method for checking integrity in deductive databases," *Data & Knowledge Engineering*, **Vol.4**, pp. 223-244, Elsevier Science Publishers B.V. (North-Holland), (1989).

2. S.K.Das and M.H.Williams, "Integrity checking methods in deductive databases: A comparative evaluation," *M.H.Williams (ed.): Proceedings of the 7th British National Conference on Databases*, pp. 85-116, Cambridge University Press, (1989).

3. S.K.Das and M.H.Williams, "Extending the integrity maintenance capability in deductive databases," to appear in *Proceedings of the UK ALP-90 Conference*, Bristol, UK, (March 1990).

4. S.K.Das and M.H.Williams, "Negation as possible failure: A rule for inferring negative information from indefinite deductive databases," Submitted for publication.

Acknowledgements

My thanks go first to my supervisor, Prof. M.H.Williams, for his continuous encouragement, able supervision and for directing my immature thinking along the right path throughout my research. I am extremely indebted to him.

Thanks also to the Government of India for awarding me a scholarship for studying abroad. Without this sponsorship I would not have been able to fulfill my ambition for higher studies.

Thanks to all the members of my family at home for patiently accepting my absence amongst themselves and for their inspiration.

The good academic atmosphere in the department, stimulating people in the environment and pleasant entertainment have rekindled my enthusiasm from time to time. I am grateful to everyone in my department - particularly David Ferbrache, Greg Michaelson and Steve Salvini for their helpful comments on my thesis - to my friend Janique for her good company.

Many thanks to all the anonymous referees who have commented either on my published papers or on this thesis. Their comments have improved my thesis substantially.

Finally, I acknowledge and express my gratitude to all those - in the past and at present, either directly or indirectly - who have contributed to the two fields, Deductive Databases and Logic Programming.

Abstract

A deductive database is a logic program that generalises the concept of a relational database. Integrity constraints are properties that the data of a database are required to satisfy and in the context of logic programming, they are expressed as closed formulae. It is desirable to check the integrity of a database at the end of each transaction which changes the database. The simplest approach to checking integrity in a database involves the evaluation of each constraint whenever the database is updated. However, such an approach is too inefficient, especially for large databases, and does not make use of the fact that the database satisfies the constraints prior to the update.

A method, called *the path finding method*, is proposed for checking integrity in definite deductive databases by considering constraints as closed first order formulae. A comparative evaluation is made among previously described methods and the proposed one. *Closed general formulae* is used to express aggregate constraints and Lloyd et al.'s simplification method is generalised to cope with these constraints.

A new definition of constraint satisfiability is introduced in the case of indefinite deductive databases and the path finding method is generalised to check integrity in the presence of static constraints only. To evaluate a constraint in an indefinite deductive database to take full advantage of the query evaluation mechanism underlying the database, a query evaluator is proposed which is based on a definition of semantics, called *negation as possible failure*, for inferring negative information from an indefinite deductive database.

Transitional constraints are expressed using *action relations* and it is shown that transitional constraints can be handled in definite deductive databases in the same way as static constraints if the underlying database is suitably extended. The concept of *implicit update* is introduced and the path finding method is extended to compute facts which are present in action relations. The extended method is capable of checking integrity in definite deductive databases in the presence of transitional constraints.

Combining different generalisations of the path finding method to check integrity in deductive databases in the presence of arbitrary constraints is discussed. An extension of the data manipulation language of SQL is proposed to express a wider range of integrity constraints. This class of constraints can be maintained in a database with the tools provided in this thesis.

Abbreviations

Abbreviation	Full
CWA	Closed World Assumption
DCA	Domain Closure Assumption
DDR	Disjunctive Database Rule
dmf	Disjunctive Minimised Form
EGCWA	Extended Generalised Closed World Assumption
GCWA	Generalised Closed World Assumption
iff	IF and only iF
mgu	Most General Unifier
NAF	Negation As Failure
NAFFNH	Negation As Finite Failure for Non-Horn programs
NF	Normal Form
nH-Prolog	Near Horn-Prolog
PMR	Perfect Model Rule
QBE	Query By Example
QUEL	QUEry Language
resp	RESPectively
SLD	Linear resolution with Selection function for Definite clauses
SLDNF	SLD-resolution augmented by Negation as Failure
SQL	Structured Query Language
Tran	TRANsaction
UNA	Unique Name Assumption
wff	Well-Formed Formula
wrt	With Respect To

List of Figures

List of Tables

List of Notations

Notation	Description	Page
$A \cup B$	Union of two sets A and B	8
$A \cap B$	Intersection of two sets A and B	8
$A - B$	Difference of two sets A and B	8
$<d_1, d_2, ..., d_n>$	n-tuple	8
$D_1 \times D_2 \times \cdots \times D_n$	Cartesian product of the sets $D_1, D_2, ..., D_n$	8
D^n	Cartesian product of the sets D, D, ..., D (n times)	8
$f : A \rightarrow B$	Function f from a set A into a set B	8
$\neg$	Logical negation	9
$\vee$	Logical disjunction	9
$\wedge$	Logical conjunction	9
$\rightarrow$	Logical implication	9
$\leftrightarrow$	Logical equivalence	9
$\forall$	Universal quantifier	9
$\exists$	Existential quantifier	9
$\leftarrow$	Reverse implication	9
$\vdash_I U$	U is true for the interpretation I	13
$\Gamma \models U$	U is a logical consequence of Γ	13
$\Gamma \vdash_T U$, $\Gamma \vdash U$	Wff U is derivable from a set of wffs Γ in a first-order theory T	14
$L(\Gamma)$	First-order language underlying the set of wffs Γ	15
$HU(\Gamma)$	Herband universe of the set of wffs Γ	15
$HB(\Gamma)$	Herband base of the set of wffs Γ	15
$x_1/t_1, ..., x_k/t_k$	A substitution to the variables $x_1, ..., x_k$	16
$E\theta$	Application of the substitution θ to the expression E	16
$\leftarrow L_1 \wedge \cdots \wedge L_n \quad n \geq 1$	A goal or a query	19
$A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n \quad m \geq 1$	A program or a database clause	19
$\square$	An empty clause	20

Contd..

Notation	Description	Page
comp (Π)	The completion of the program Π	22
I_d	Set of constraints in denial form with their heads	48
IC (*No*)	Head of a constraint which is in denial form	48
$L_0 \xrightarrow{R_1} L_1 \xrightarrow{R_2} \cdots \xrightarrow{R_n} L_n$	A path	49
CompL	Complement of the literal *L*	66
Not (*P*)	Normal form negative formula	71
uc (*F*)	Set of update constraints	77
D^*	Set of facts derivable from the database *D*	77
D^T	Set of include-facts due to the transaction *T* on the database *D*	77
D_T	Set of remove-facts due to the transaction *T* on the database *D*	77
mod (*D*)	Modified form of the database *D*	82
EXTRA (*Not* (*P*))	Set of NF-negative formulae	83
gcs (*D*)	Greatest closed set of the database *D*	97
pos (*R* , *A*)	Possible form of the clause *R* wrt *A*	99
Def _true (*D*)	Set of all definitely true facts in the database *D*	100
Def _false (*D*)	Set of all definitely false facts in the database *D*	100
Unknown (*D*)	Set of all possibly true facts in the database *D*	100
D_i^+	Positive database	101
D_i^-	Negative database	102
Def A	Deferred literal	105
def (R , M_p)	Definite form of the clause *R* wrt M_p	105
con (R , M_r)	Contrapositive form of the clause *R* wrt M_r	106
con (G , L_r)	Contrapositive form of the goal *G* wrt L_r	106

Contd..

Notation	Description	Page
$ADD_{D,D'}$	Set of all facts added to the database D due to a transaction	151
$DEL_{D,D'}$	Set of all facts deleted from the database D due to a transaction	151
$UPD_{D,D'}$	Set of all explicit or implicit update facts due to a transaction	152
$<R(T), R(T')>$	$R(T)$ has been updated by $R(T')$	152
$UPD\_ADD_{D,D'}$	Facts added to the database D due to the updates	156
$UPD\_DEL_{D,D'}$	Facts deleted from the database D due to the updates	156
$\dot{ADD}_{D,D'}$	Facts added to the database D not due to the updates	156
$\dot{DEL}_{D,D'}$	Facts deleted from the database D not due to the updates	156
ADD_R	Addition type action relation corresponding to the relation R	156
DEL_R	Deletion type action relation corresponding to the relation R	156
UPD_R	Update type action relation corresponding to the relation R	156

CHAPTER 1

Introduction

1.1. Background

Deductive database systems [34, 35, 45, 59, 64, 76, 102] have first-order logic as their theoretical foundation. This approach has several advantages:

(a) Logic itself has a well-understood semantics.

(b) Logic can be used as a uniform language for expressing facts, rules, queries and integrity constraints.

(c) The well-developed theory of logic can be used to solve different database problems, e.g., null value, indefinite data.

(d) Use of a single rule may replace many explicit facts. It provides an expressive and economic environment for data modelling.

(e) In a natural way, a deductive database generalises the concept of *relational databases* [9, 23, 36, 104].

In a database an *integrity constraint* (or, simply a *constraint*) [20, 33, 51, 104] is a formal representation of a property which the data in the database must satisfy to be consistent with some model of the real world from which it comes. A constraint can specify either some property governing the correct states of the database or the allowable transitions from one database state into another. From the logic point of view, an integrity constraint is defined as a closed formula.

Constraints are classified according to their nature, e.g., static (range, referential, aggregate etc.) or transitional. Some examples of constraints are as follows:

the salary of an employee must be less than 10000 (static & range),

every worker has a manager (static & referential),

the total number of employees in a department may not exceed 100 (static & aggregate),

when updating the salary of an employee, the new value must exceed the old value (transitional).

The major concern behind developing a method for checking integrity in a database is improving efficiency. Efficiency can be improved by making one important assumption that the database satisfies constraints prior to its update. A number of methods have been proposed for checking integrity in deductive database by exploiting this assumption. They include generalised simplification methods by Lloyd et al. [66] , Decker [25] , and Bry et al. [10] , a general theorem-proving technique by Kowalski et al. [91,55] , using the Prolog not-predicate by Ling [60] , consistency proof method and a modified program method by Asirelli et al. [3].

Two major views of constraint satisfiability have been proposed. In the first, the *consistency view*, a database is said to satisfy a set of constraints if every constraint is consistent with the *completed* [16] form of the database. In the second, or the *theoremhood view*, every constraint has to be a theorem of the completed database. The methods of Lloyd et al., Decker, Bry et al. and Ling are based on the theoremhood view; whereas, the methods of Kowalski et al. and Asirelli et al. are based on the consistency view. The former methods are different generalisations of the *simplification method* [79] proposed by Nicolas for checking static non-aggregate kinds of constraints in relational databases.

1.2. Objective of thesis

Integrity constraints are expected to be satisfied after every change to the database. The problem of efficiently checking integrity constraints in a database is becoming increasingly important. Checking integrity constraints in deductive databases is more difficult than for relational databases since a single update in a deductive database may cause a number of implicit additions to and deletions from the database. The following shortcomings have been observed in the methods previously proposed for checking integrity constraints in deductive databases:

(i) There is a dearth of literature discussing the problems of maintaining integrity constraints in deductive databases or comparing the different methods and highlighting their shortcomings from the ideal.

(ii) Most of the methods fail to avoid various overheads, e.g., redundant constraint evaluation, frequent reasoning with two database states (i.e. the database before and after update) during the process of constraint evaluation; efficiency can be increased in some situations by avoiding such overheads.

(iii) The underlying database is always *definite*, i.e. the head of each database clause is an atom and the body is a conjunction of literals. When the database becomes *indefinite* by allowing the head of a database clause to be a disjunction of atoms, the methods are no longer sufficient.

(iv) The methods deal only with static non-aggregate constraints, i.e. constraints which are closed first-order formulae without any aggregation over any attribute of a predicate. The first two examples of constraints given in the previous section fall into this category and can be expressed as closed first-order formulae as follows:

$$\forall x \, \forall y \, \forall z \, (Employee(x, y, z) \rightarrow z < 10000)$$

$$\forall x \, (Worker(x) \rightarrow \exists y \; Manager(y, x))$$

Confining constraints to static non-aggregate types of formulae excludes some important types from consideration. For example, the third constraint from the examples in the previous section

the total number of employees in a department may not exceed 100

cannot be expressed conveniently as a closed first-order formula because of the aggregation involved over the attribute x of *Employee* predicate.

(v) None of the methods can handle transitional constraints.

To try to deal with the problems listed above, the major concerns of the thesis are as follows.

(1) A method, called *the path finding method*, is proposed for checking integrity constraints in definite deductive databases by taking into account some of the problems described in (ii) above.

(2) In response to the first problem, a survey has been carried out to focus on different problems of maintaining integrity constraints in a database. Different approaches are compared as a

basis for assessing the proposed method for checking integrity constraints. This comparison includes the path finding method and is confined to definite deductive databases.

(3) In response to the third problem, the thesis generalises both the simplification and the path finding methods to handle integrity constraint checking in indefinite databases. This is based on a new definition of constraint satisfiability in indefinite databases. This concept of constraint satisfiability is a generalised version of the theoremhood view of constraint satisfiability in the context of definite databases.

Although a constraint verification program based on the above concept of constraint satisfiability can work on any query evaluator for the underlying database, the semantics of the database based on the *negation as possible failure rule* has been considered for inferring negative information from a database. The introduced notion of constraint satisfiability and the negation as possible failure rule are closely related.

(4) In response to the problem cited in (iv), closed formulae which are more general than first-order, called *closed general formulae*, have been considered to represent aggregate constraints. For example, the constraint

the total number of employees in a department may not exceed 100

can be expressed as the closed general formula

$$\forall x \forall v (Dept(v) \wedge Count(\exists u \exists w\ Employee(u, v, w), x) \rightarrow x \leq 100)$$

The path finding method has also been extended to deal with these constraints.

(5) In response to the final problem, *action relations* [80] are considered to represent transitional constraints. The constraint

when updating the salary of an employee, the new value must exceed the old value

can be expressed using action relation *Upd_Employee* corresponding to *Employee* as

$$\forall x \forall y \forall z \forall y' \forall z' (Upd\_Employee(x, y, z, x, y', z') \wedge\ z \neq z' \rightarrow z' > z)$$

To compute tuples of a derived relation, the concept of *implicit updates* is introduced. The path finding method is extended to handle transitional constraints in definite deductive databases.

1.3. Overview of thesis

The remainder of the thesis is organised as follows.

Chapter 2 gives an introduction to the concepts of sets, functions, relations, first-order theory, unification in first-order theory, resolution inference rules, SLDNF-resolution, logic programming etc. The chapter also describes briefly the main features of Prolog.

A discussion on both relational and deductive databases is given in chapter 3 from a logic point of view. The relative merits and demerits of relational and deductive databases are also examined.

Chapter 4 is devoted to different issues of integrity constraints. This chapter includes the definition, classification and different views of satisfiability for integrity constraints.

In chapter 5, the *path finding method* is presented. In this method the verification of the integrity constraints in the updated database is reduced to the process of constructing paths from update literals to the heads of constraints. Correctness of the method has been proved in the context of stratified deductive databases. An explanation of how this method may be realised efficiently in Prolog is given.

Chapter 6 evaluates the merits and demerits of a number of constraint checking methods, including the path finding method, in the context of definite deductive databases as proposed by a number of authors in the last few years. To achieve this, a brief introduction to the methods is given in this chapter and then their performances are compared against each other in the context of different databases. The various formats for representing constraints used by these different methods and the expressiveness of each is also discussed in this chapter.

Chapter 7 presents the rule negation as possible failure for inferring negative information from an indefinite deductive database. This problem has been addressed previously by several workers and their results are given briefly. The implementation of query evaluation based on the introduced semantics for negative information is considered and it is also shown that when the database is definite, this implementation reduces to the mechanism of simple SLDNF. The query evaluator which is required for constraint evaluation is based on this implementation. The Prolog code for the main algorithms from the implementation is given. A similar implementation in the nH-Prolog [69, 93] environment is considered in the following chapter. The null value problem is discussed and guidelines for extending the semantics and implementation to cope with null values

are also given in this chapter.

A generalisation of the path finding method to check integrity constraints in indefinite deductive databases is described in chapter 8. The Prolog implementation of the method and a meta-interpreter for the extended nH-Prolog are discussed. The extended nH-Prolog, which is capable of inferring negative information from the database, is based on the negation as possible failure rule. The query evaluator required for constraint evaluation operates on this extended nH-Prolog. The Prolog code is given for the main algorithms for implementing the extended nH-Prolog as well as that for the method itself.

Chapter 9 considers a class of constraints, called *aggregate constraints*. A set of aggregate predicates (e.g., *count*, *sum*, *max*) is allowed in the definition of constraints and Lloyd et al.'s generalised simplification method is extended to maintain integrity in the presence of aggregate constraints. The implementation of this in Prolog may be achieved using the predicate *setof*.

Chapter 10 extends the path finding method to handle transitional constraints in definite deductive databases. It is argued that transitional constraints can be handled in the same way as static constraints provided that the database is extended using action relations. This idea was originally introduced by Nicolas and Yazdanian [80]. The extension of the path finding method involves the computation of facts in action relations.

The objective of chapter 11 is to propose a data manipulation language for integrity constraints [6]. As a query language, SQL has already proved its popularity. However, the constraint expressing capability in standard SQL [21] is highly limited. An extension is required to express the variety of constraints discussed in the thesis (e.g transitional, aggregate and more general static constraints). An extension to SQL is proposed which increases its capability to express constraints. Some guidelines are given to develop an algorithm for translating constraints expressed in the extended SQL to their equivalent logical formulae. A number of examples of constraints are also given with their syntax in extended SQL and their associated translations in logical formulae.

The final chapter assesses the contribution of this thesis and identifies areas of further work.

The problem of handling the static non-aggregate type of integrity constraint in definite deductive databases has already received much attention by several researchers as compared to more specialized problems such as that of handling integrity constraints in indefinite databases or handling the aggregate and transitional type of integrity constraint in deductive databases. Hence,

there is scope for discussion of the former type of problem and a review of this takes place in chapter 6. On the other hand, such a review cannot take place when dealing with the latter unexplored problems mentioned above. It is author's hope that the proposals put forward in chapters 8, 9 & 10 dealing with, respectively, integrity constraints in indefinite databases, aggregate and transitional type of integrity constraints, will form a basis for future work in the respective areas. A detailed discussion comparing different semantics for negative information in indefinite deductive databases (including the one proposed in chapter 7) considered to be beyond the scope of this thesis.

The description of every relation variables used in this thesis can be found in appendix 1.

CHAPTER 2

From Logic To Logic Programming

This chapter introduces some basic concepts of sets, relations, functions, first-order logic, unification in first-order theory, resolution inference rules, SLDNF-resolution, logic programming, Prolog, etc. The order of introduction of these topics reflects the development of logic programming through different steps from standard first-order logic. Logic programming is introduced here since the results related to logic programming are relevant to deductive databases which are introduced in the next chapter.

2.1. Sets, relations and functions

Definition : A well-defined collection of distinct elements is called a *set* [41,42,98]. The *union* of the sets A and B, denoted by $A \cup B$, is the set of all elements which are members of either A or B (or both). The *intersection* of the sets A and B, denoted by $A \cap B$, is the set of all elements which are members of both A and B. The *difference* of the sets A and B, denoted by $A - B$, is the set of all elements which are members of A but not of B.

Definition : Given a collection of sets D_1, D_2, ..., D_n, the *cartesian product* of these n sets, denoted by $D_1 \times D_2 \times \cdots \times D_n$, is the set of all possible ordered n-tuples $<d_1, d_2, ..., d_n>$ such that $d_i \in D_i$, for $i = 1, 2, ..., n$. When each D_i is equal to D then the product is written as D^n.

Definition : A *relation* R on the sets D_1, D_2, ..., D_n is a subset of the cartesian product $D_1 \times D_2 \times \cdots \times D_n$.

Definition : A *function* from a set A into a set B, denoted by $f : A \rightarrow B$, is a relation on the sets A, B such that each element of A is related to exactly one element of the set B. The set A is called the *domain* of the function and the set B is called the *co-domain* (or *range*).

Example : Let $D_1 = \{a, b\}$ and $D_2 = \{1, 2, 3\}$ be two sets. Then $D_1 \times D_2 = \{<a, 1>, <a, 2>, <a, 3>, <b, 1>, <b, 2>, <b, 3>\}$. A relation on D_1 and D_2 is $\{<a, 1>, <b, 3>\}$.

2.2. First-order logic

Definition : An *alphabet* [63] consists of the following symbols:

1. Parentheses: (,).

2. Sentential connective symbols: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence).

3. Individual variables: $s, t, u, v, w, x, y, z, s_1, t_1, \ldots$.

4. Quantifier symbols: $\forall$ (universal), $\exists$ (existential).

5. N-ary predicate symbols: For each positive integer n, some set (possibly empty) of symbols $P^n, Q^n, R^n, P_1^n, Q_1^n, R_1^n, \ldots$.

6. Constant symbols: Some set (possibly empty) of symbols $a, b, c, \ldots$.

7. N-ary function symbols: For each positive integer n, some set (possibly empty) of symbols $f^n, g^n, h^n, f_1^n, g_1^n, h_1^n, \ldots$.

Constant symbols are often considered as 0-place function symbols to ensure the uniform treatment of function symbols and constant symbols. The symbols "$\neg$", "$\vee$", "$\wedge$", "$\rightarrow$", "$\leftrightarrow$", "$\exists$", "$\forall$" are read as *not*, *or*, *and*, *implies*, *if and only if*, *there exists*, *for all*. To avoid using brackets for the readability of formulae, the precedence hierarchy which has been adopted for the connective symbols and the quantifier symbols is as follows. The symbols "$\neg$", "$\forall$" and "$\exists$" have the highest precedence, then "$\wedge$", followed by "$\vee$" and finally "$\leftarrow$", "$\leftrightarrow$".

Definition : *Terms* are those finite strings of symbols generated by repeated application of the following rules:

(a) A variable is a term.

(b) A constant is a term.

(c) If f is an n-ary function symbol and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.

Definition : The *well-formed formulae* [19,28,74] (*wffs* or simply *formulae*) of a first-order language are those finite strings of symbols generated according to the following rules:

(a) If P is an 0-ary predicate symbol then P is a formula.

(b) If P is an n-ary ($n>0$) predicate symbol and $t_1, ..., t_n$ are terms then $P(t_1, ..., t_n)$ is a formula.

(c) If U and V are formulae then so are $(\neg U)$, $(U \wedge V)$, $(U \vee V)$, $(U \rightarrow V)$ and $(U \leftrightarrow V)$.

(d) If U is a formula and x is a variable then $(\forall x\ U)$ and $(\exists x\ U)$ are formulae.

Often the symbol "$\leftarrow$" (*reverse implication*) is used to represent a formula $U \rightarrow V$ alternatively as $V \leftarrow U$ and in that case the formula $U \rightarrow V$ will be read as *if U then V*. *Atomic formulae* (or more simply *atoms*) are those generated by rules (a) and (b).

Example : $P(x), Q(y, x)$ are atoms and $\forall x(P(x) \rightarrow \exists y\ Q(y, x))$ is a wff.

Definition : The *first-order language* [28] given by an alphabet comprises the set of all formulae constructed from the symbols of the alphabet.

Definition : In $\forall x\ W$, where W is a wff, 'W' is called the *scope* of the quantifier '$\forall x$'. W need not contain the variable x and in that case, $\forall x\ W$ is same thing as W.

Definition : An occurrence of a variable x is said to be *bound* in a wff W if either it is the occurrence of x in the quantifier '$\forall x$' or it lies within the scope of a quantifier '$\forall x$' in W. An occurrence of a variable x in a wff W will be called a *free occurrence* if it is not bound.

Example : The occurrences of x are free in the wff $P(x) \rightarrow \exists y\ Q(y, x)$ whereas y is bound in the same formula.

Definition : A *closed formula* is a formula with no free occurrences of any variable.

Example : The formula $\forall x(P(x) \rightarrow \exists y\ Q(y, x))$ is closed whereas the formula $P(x) \rightarrow \exists y\ Q(y, x)$ is not closed as x is now free in it.

Definition : If U is a wff and t is a term, then t is said to be free for x in U if no free occurrences of x in U lies within the scope of any quantifier $\forall y$, where y is a variable in t.

Definition : A *literal* is an atom or the negation of an atom. A *positive literal* is an atom. A *negative literal* is the negation of an atom. An *expression* is either a term, a literal or a conjunction or disjunction of literals.

Any wff of a first-order language can be written in *prenex normal form* [74] , i.e. with all quantifiers at the front, by applying the usual transformations of the form $\neg(\forall x U) \equiv \exists x (\neg U)$, $\neg(\exists x U) \equiv \forall x (\neg U)$, etc.

Definition : A *clause* is a disjunction of literals all of whose variables are universally quantified over the whole disjunction. A *negative clause* is a disjunction of negative literals only.

A clause will sometime be written as a disjunction of literals without its universal quantifiers and in those cases free variables are assumed to be universally quantified over the whole disjunction.

Example : $\forall x \forall y (\neg P(x, y) \vee Q(x) \vee R(y))$ is a formula in the form of a clause.

Any closed wff in prenex normal form (and hence any closed wff) may be rewritten as a conjunction of clauses, by first applying transformations of the form $U \rightarrow V$ (or $V \leftarrow U$) $\equiv \neg U \vee V$, $U \leftrightarrow V \equiv (\neg U \vee V) \wedge (\neg V \vee U)$, $U \vee (V \wedge W) \equiv (U \vee V) \wedge (U \vee W)$, $\neg(U \wedge V) \equiv \neg U \vee \neg V$, $\neg\neg U \equiv U$, and then each existentially quantified variable is replaced by a function of all the universally quantified variables which precede the existential quantifier in the prefix. The latter is known as *Skolemisation* of the existentially quantified variables.

Definition : A *Horn clause* [70] is a clause with at most one positive literal in the disjunction.

Example : The clause $\forall x \forall y (\neg P(x, y) \vee Q(x))$ is a Horn clause as $Q(x)$ is the only positive literal whereas $\forall x \forall y (\neg P(x, y) \vee Q(x) \vee R(y))$ is not as it contains $Q(x)$ and $R(y)$ both of which are positive literals.

2.2.1. Semantics: Interpretations and models

This sub-section describes the truth-based declarative semantics of first-order languages based on the concepts of interpretation and model.

Definition : An *interpretation* [1, 19, 28, 44, 58, 74] I of a first-order language L consists of the following:

(a) A nonempty set D called the *universe* or *domain* of interpretation I.

(b) For each n-ary predicate symbol in L, the assignment of an n-ary relation in D.

(c) For each n-ary function symbol in L, the assignment of a mapping from D^n to D.

(d) For each constant symbol, the assignment of a fixed element of D.

Given such an interpretation I, variables are thought of as ranging over the domain of the interpretation I, and the sentential connective symbols and quantifiers are given their usual meaning.

Let I be an interpretation with domain D. Let Σ be the set of all sequences of elements of D. For a given sequence $S=(s_1, s_2, ...)\in\Sigma$ and for a term t consider the following *term assignment* of t with respect to I and S as follows:

(a) If t is a variable x_j then its assignment is s_j.

(b) If t is a constant then its assignment is according to I.

(c) If t_1', ..., t_n' are the term assignments of t_1, ..., t_n respectively and f' is the assignment of the n-ary function symbol f, then $f'(t_1', ..., t_n')\in D$ is the term assignment of $f(t_1, ..., t_n)$.

The definition of satisfaction of a formula with respect to a sequence and an interpretation can be inductively defined as follows:

(a) If U is an atomic wff $P(t_1, ..., t_n)$, then the sequence $S=(s_1, s_2, \cdots)$ satisfies U if and only if $P'(t_1', ..., t_n')$ (i.e. the n-tuple $<t_1', ..., t_n'>$ is in the relation P'), where P' is the corresponding n-place relation of the interpretation of P.

(b) S satisfies $\neg U$ if and only if S does not satisfy U.

(c) S satisfies $U\wedge V$ if and only if S satisfies U and S satisfies V.

(d) S satisfies $U\vee V$ if and only if S satisfies U or S satisfies V.

(e) S satisfies $U\rightarrow V$ if and only if either S does not satisfy U or S satisfies V.

(f) S satisfies $U\leftrightarrow V$ if and only if S satisfies both U and V or S satisfies neither U nor V.

(g) S satisfies $\exists x_i(U)$ if and only if there is a sequence S' that differs from S in at most the i th component such that S' satisfies U.

(h) S satisfies $\forall x_i(U)$ if and only if every sequence that differs from S in at most the i th component satisfies U.

Definition : A wff U is *true for the interpretation* (alternatively, U can be given the *truth value* **true**) I (written $\vdash_I U$) if and only if every sequence in Σ satisfies U. U is said to be **false** for I if and only if no sequence of Σ satisfies U.

The truth value of a closed formula does not depend on the sequence of interpretation and in such a situation the satisfaction of the formula is with respect to the interpretation only. If a formula is not closed then it may be neither true nor false for some interpretation. If a formula is closed then for an arbitrary interpretation either the formula or its negation is true with respect to the interpretation.

Definition : Let I be an interpretation of a first-order language L. Then I is said to be a *model* for a closed wff U if U is true with respect to I. I is said to be a *model* for a set Γ of closed wffs of L if and only if every wff in Γ is true with respect to I.

Definition : Let Γ be a set of closed wffs of a first-order language L. Then Γ is

1. *satisfiable* if and only if L has at least one interpretation which is a model for Γ.

2. *valid* if and only if every interpretation of L is a model for Γ.

3. *unsatisfiable* if and only if no interpretation of L is a model for Γ.

Definition : Let U be a closed wff of a first-order language L. A closed wff V is said to be *implied* by U (or, equivalently, U implies V) if and only if for every interpretation I of L, I is a model for U implies that I is a model for V.

Definition : Let Γ be a set of closed wff of L. A closed wff U is said to be a *logical consequence* of Γ (written $\Gamma \models U$) if and only if for every interpretation I of L, I is a model for Γ implies that I is a model for U.

Definition : Two closed wffs U and V are said to be *equivalent* if and only if they imply each other.

2.2.2. Syntax: First-order theory

The syntactic aspect of a first-order language is concerned with proof theory, using axioms and inference rules in order to infer theorems from a given set of wffs.

Definition : A *first-order theory* [1, 19, 28, 44, 58, 74] comprises a set of symbols called the alphabet, the first-order language, the set of *axioms* and the set of *inference rules*. The axioms of a first-order theory are a set of wffs and are divided into two classes: the *logical axioms* and the *proper axioms*. If U, V and W are wffs, then the following are logical axioms of the theory:

(I) $U \rightarrow (V \rightarrow U)$

(II) $(U \rightarrow (V \rightarrow W)) \rightarrow ((U \rightarrow V) \rightarrow (U \rightarrow W))$

(III) $(\neg V \rightarrow \neg U) \rightarrow ((\neg V \rightarrow U) \rightarrow V)$

(IV) $(\forall x\ U) \rightarrow V$ if V is a wff obtained from U by substituting all free occurrences of x by a term t and t is free for x in U.

(V) $\forall x(U \rightarrow V) \rightarrow (U \rightarrow \forall x\ V)$ if U is a wff that contains no free occurrences of x.

Proper axioms vary from theory to theory. A first-order theory in which there are no proper axioms is called a *first-order predicate calculus*. When other wffs are added as axioms to a first-order predicate calculus, the new axioms become proper axioms. The following are rules of inference of any first-order theory:

1. *Modus ponens*: V follows from U if $U \rightarrow V$.

2. *Generalisation*: $\forall x(U)$ follows from U.

Given a set of inference rules, the notion of derivability of a wff from a set of wffs in a first-order theory can be introduced as follows:

Definition : A wff U is *derivable* from a set of wffs Γ in a first-order theory T (written $\Gamma \vdash_T U$) if and only if

1. U is a member of Γ or

2. U is a result of applying a rule of inference to wffs derivable from Γ.

Whenever the theory T is clear from the context, a derivation would be written as $\Gamma \vdash U$.

Definition : If the set Γ is empty then U is a *theorem* of T (written $\vdash_T U$).

Suppose, Γ is a set of wffs and U is a wff. Then from Godel's completeness theorem, $\Gamma \vdash U$ if and only if $\Gamma \models U$. Also, it can be proved that U is a logical consequence of Γ if and only if $\Gamma \cup \{\neg U\}$ is unsatisfiable. For convenience, instead of considering every interpretation in order to show the unsatisfiability of a set of closed wffs, it is sufficient to consider a smaller class of interpretation, called the *Herband interpretation* [70, 63].

Given a set of wffs Γ, it will always be assumed that the underlying first-order language $L(\Gamma)$ is defined by the constants, function symbols and predicate symbols appearing in Γ.

Example : Let Γ be a set of wffs as follows:

$$\forall x \forall y \forall z (G(x, y) \leftarrow F(x, z) \wedge F(z, y))$$
$$F(a, b)$$
$$F(b, c)$$
$$F(b, d)$$

Then the first-order language $L(\Gamma)$ is given by constants a, b, c and d, and binary predicate symbols G and F. There is no function symbol in the language.

Definition : The *Herband universe* $HU(\Gamma)$ of Γ is the set of all ground terms defined recursively as follows:

(a) Every constant symbol of $L(\Gamma)$ is a member of $HU(\Gamma)$. In the case that $L(\Gamma)$ has no constants, $HU(\Gamma)$ contains a specially introduced symbol, say a.

(b) If f is a function symbol of arity n that occurs in $L(\Gamma)$ and $t_1, ..., t_n$ are terms in $HU(\Gamma)$ then $f(t_1, ..., t_n)$ is a member of $HU(\Gamma)$.

(c) No term other than those defined by (a) and (b) above, is a member of $HU(\Gamma)$.

Definition : The *Herband base* $HB(\Gamma)$ of Γ is the set of all ground instances of all atomic wffs which can be formed by using predicate symbols from $L(\Gamma)$ with terms from $HU(\Gamma)$ as arguments.

Example : Let Γ be a set of wffs as follows:

$$\forall x (P(f(x)) \leftarrow P(x))$$

$$P(a)$$

Then $HU(\Gamma)$ is $\{a, f(a), f(f(a)), ...\}$ and $HB(\Gamma)$ is $\{P(a), P(f(a)), P(f(f(a))), ...\}$.

Definition : Let T be a first-order theory and let L be the language of T. A *model* for T is an interpretation for L which is a model for the set of axioms of T. If T has a model then T is said to be *consistent.*

Definition : Given a set of closed wffs Γ, a *Herband model* for Γ is an Herband interpretation for $L(\Gamma)$ which is a model for Γ.

A model of a set of closed wffs Γ will always mean the Herband model of Γ in this thesis.

Definition : A theory T is said to be *complete* if for every closed formula W in the language of the theory, either W or its negation is a logical consequence of T.

2.3. Unification

Unification [47, 73, 81, 88] is a process of determining whether two expressions can be made identical by appropriate substitution for their variables. Various terms related to unification are described in this section.

Definition : A *substitution* θ is a finite set of pairs of variables and terms, denoted by $\{x_1/t_1, ..., x_k/t_k\}$, where x_i's are distinct and each t_i is different from x_i. The term t_i is called a *binding* for x_i. θ is called a *ground substitution* if the t_i are all ground terms. The substitution given by the empty set is called the *empty substitution* (or *identity substitution*).

Let $\theta=\{x_1/t_1, ..., x_k/t_k\}$ be a substitution and E be an expression. The *application* of θ to E, denoted by $E\theta$, is the expression obtained by simultaneously replacing each occurrence of the variable x_i in E by the term t_i. In this case $E\theta$ is called the *instance* of E by θ. If $E\theta$ is ground then $E\theta$ is called a *ground instance* of E. Also, E is referred to as a *generalisation* of $E\theta$.

Example: Let $E = p(x, f(x), y, g(a))$ and $\theta = \{x/b, y/h(x)\}$. Then $E\theta = p(b, f(b), h(x), g(a))$.

Definition : Let $\theta=\{x_1/t_1, ..., x_m/t_m\}$ and $\phi=\{y_1/s_1, ..., y_n/s_n\}$ be two substitution. Then the *composition* $\theta\phi$ of θ and ϕ is the substitution obtained from the set

$$\{x_1/t_1\phi, ..., x_m/t_m\phi, y_1/s_1, ..., y_n/s_n\}$$

by deleting any binding $t_i\phi$ for which $x_i=t_i\phi$ and deleting any binding for which $y_i \in \{x_1, ..., x_m\}$. It can be proved that the composition of substitutions is associative and hence in writing a composition $\theta_1, ..., \theta_n$ of substitutions, parentheses can be omitted.

Example: Let $\theta=\{x/b, y/h(z)\}$ and $\sigma=z/c$. Then $\theta\sigma=\{x/b, y/h(c), z/c\}$.

Definition : A *unifier* of two atoms A and A' is a substitution θ such that $A\theta$ is syntactically identical to $A'\theta$. If the two atoms do not have a unifier then they are not *unifiable*. A unifier θ is called a *most general unifier* (*mgu*) for the two atoms if for each unifier α of A and A' there exists a substitution β such that $\alpha=\theta\beta$.

A mgu is unique up to variable renaming. Because of this property *the* mgu of two expressions would be used quite often.

Example : The mgu of two expressions $p(x, f(a, y))$ and $p(b, z)$ is $\{x/b, z/f(a, y)\}$. A unifier of these two expression is $\{x/b, y/c, z/f(a, c)\}$.

2.4. Resolution theorem proving

Most theorem-proving techniques are based on the inference rule known as the *Robinson resolution principle* which applies to wffs of clausal form. This technique of theorem-proving is termed *resolution theorem-proving* and was developed by J.A.Robinson [89]. Unification is an essential part of this kind of resolution theorem-proving.

2.4.1. Resolution principle

Suppose C and C' are two clauses. If there is a literal L occurring in C and a literal $\neg L'$ occurring in C' such that L and L' have an mgu θ, then it is said that the two clauses C and C' *resolve* and that the new *derived* clause, $R(C, C'; L, L')$, is a resolvent of the two clauses, where $R(C, C'; L, L')$ is obtained from C, C', L, L' and θ by performing the following two steps:

1. The clause $C\theta - \{L\}\theta$ (resp. $C'\theta - \{L'\}\theta$) is obtained from $C\theta$ (resp. $C'\theta$) by removing the occurrence $L\theta$ (resp. $L'\theta$).

2. $R(C, C'; L, L')$ is the disjunction of all the literals occurring in $C\theta - \{L\}\theta$ and $C'\theta - \{L'\}\theta$.

Example : The derived clause obtained from the two clauses $P(x, y) \vee R(y)$ and $Q(z) \vee \neg R(a)$ is $P(x, a) \vee Q(z)$.

If a set of input clauses is unsatisfiable then it is possible to derive by resolution the empty clause from the clauses in the set. To prove a clause from a set of clauses using this process of derivation is known as *resolution refutation* and is described as follows: In order to prove a clause U from a set of clauses Γ (i.e. $\Gamma \vdash U$), negate U and add it to Γ to yield Γ^*. If, by applying resolution on the clausal form of Γ^*, one can derive the empty clause, then Γ and $\neg U$ cannot simultaneously be satisfiable.

Resolution is not efficient when measured in terms of the size of search space for refutation. Various kinds of resolution principles have been proposed [97, 70, 15] to improve the efficiency by reducing the search space. These include *set of support*, *P_1 and N_1 resolution*, *hyperresolution*, *unit resolution*, *unit-resulting resolution*, *input resolution*, *ordered input resolution*, *linear resolution*, *model elimination*, *connection-graph resolution*, *SL-resolution*.

Input resolution requires that one of the parent clauses to each resolution operation must be an input clause, i.e. not a derived clause. Linear resolution is an extension of input resolution in which at least one of the parent clauses to each resolution operation must be either an input clause or an ancestor clause of the parent. Linear resolution with selection function (SL-resolution) is a restricted form of linear resolution. The main restriction is effected by a selection function which choses from each clause a single literal to be resolved upon in that clause. This is also similar to model elimination. *SLD-resolution* [27] (SL-resolution for non-negative Horn clauses) augmented by by the rule *Negation as Failure* [16] is known as *SLDNF-resolution*. SLDNF-resolution mechanism is usually chosen as a procedural counterpart of a correct answer which is declaratively given in terms of the *completion* of the program. Programs and their completions are discussed in the next section.

2.5. Program

A clause can be rewritten as

$$M_1 \vee \cdots \vee M_p \vee \neg N_1 \vee \cdots \vee \neg N_q ,$$

where M_is and N_js are positive literals and variables are implicitly universally quantified over the whole disjunction. Every closed wff may be rewritten in clausal form. The above clause can be written in the form of a *program clause* as

$$M_1' \vee \cdots \vee M_k' \leftarrow N_1 \wedge \cdots \wedge N_q \wedge \neg M_{k+1}' \wedge \cdots \wedge \neg M_p' \qquad k \geq 1$$

or, in the form of a *goal* as

$$\leftarrow N_1 \wedge \cdots \wedge N_q \wedge \neg M_1' \wedge \cdots \wedge \neg M_p'$$

where each M_i' is an M_j, for some j. In general, the definition of a program clause and a goal are as follows:

Definition : A *goal* is a formula of the form

$$\leftarrow L_1 \wedge \cdots \wedge L_n \qquad n \geq 1$$

where $L_1 \wedge \cdots \wedge L_n$ is the *body* of the goal. Each L_j is either an atom or a negated atom. Any variables in L_1, ..., L_n are assumed to be universally quantified over the whole formula.

Definition : A *program clause* is a formula of the form

$$A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n \qquad m \geq 1$$

where $A_1 \vee \cdots \vee A_m$ is the *head* (or *conclusion* or *consequent*) of the program clause and $L_1 \wedge \cdots \wedge L_n$ the *body* (or *condition* or *antecedent*). Each A_i is an atom and each L_j is either an atom (a *positive condition*) or a negated atom (a *negative condition*). Any variables in A_1, ..., A_m, L_1, ..., L_n are assumed to be universally quantified over the whole formula.

For the sake of convenience, in the rest of the thesis, the term 'clause' will refer to a 'program clause'.

The different forms of a program clause for different values of m and n are the following:

1. $m=1$, i.e. the clause has the form

$$A_1 \leftarrow L_1 \wedge \cdots \wedge L_n$$

in which the head is a single atom. This clause is called a *definite clause*.

2. $m>1$, i.e. the clause has the form

$$A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n$$

in which the head is a disjunction of atoms. This clause is called an *indefinite clause*.

3. $m=1$, $n=0$, i.e. the clause has the form

$$A_1 \leftarrow$$

in which the body is empty and the head is a single atom. This clause is called a *unit clause*. Thus, a unit clause is a definite clause with an empty body. A unit clause will be written omitting the symbol "$\leftarrow$".

Example : The clause $P(x,y) \leftarrow Q(x,y) \wedge \neg R(y)$ is a definite clause whereas the clause $P(x,y) \vee R(y) \leftarrow Q(x,y)$ is an indefinite clause.

Definition : A clause (resp. a literal) in which variables do not appear is called a *ground clause* (resp. *ground literal*).

Definition : An *empty clause*, denoted by $\square$, is a clause (not a program clause) with empty disjunction.

Definition : A *general* or *indefinite program* is a finite set of clauses. A *definite program* is a finite set of definite clauses. Thus a definite program is a special form of a general program.

In writing a set of clauses to represent a program, the same variables may be used in two different clauses. It will be understood that if there are any common variables between two different clauses then they are actually different from each other. Variables in a clause are renamed when they are fetched for use.

2.5.1. Definite program completion

Suppose that Π is a definite program and

$$P(t_1, ..., t_n) \leftarrow L_1 \wedge \cdots \wedge L_m$$

is a clause in Π. If the predicate symbol '=' is interpreted as the equality (or identity) relation, and $x_1, ..., x_n$ are variables not appearing elsewhere in the clause, then the above clause is equivalent to the clause

$$P(x_1, ..., x_n) \leftarrow x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge L_1 \wedge \cdots \wedge L_m$$

If $y_1, ..., y_p$ are the variables of the original clause then this can be transformed to

$$P(x_1, ..., x_n) \leftarrow \exists y_1 \cdots \exists y_p (x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge L_1 \wedge \cdots \wedge L_m)$$

The above form is called the *general form of the clause*. Suppose there are exactly k clauses, $k \geq 0$, in Π defining P (a clause C defines P if C has P in its head). Let

$$P(x_1, ..., x_n) \leftarrow E_1$$

.

.

.

$$P(x_1, ..., x_n) \leftarrow E_k$$

be the k clauses in general form. Then the *completed definition* of P is the formula

$$P(x_1, ..., x_n) \leftrightarrow E_1 \vee \cdots \vee E_k$$

Example : Let the predicate symbol P be defined by the following clauses:

$$P(a, z) \leftarrow Q(z, z') \wedge \neg R(z')$$

$$P(b, c)$$

Then the completed definition of P is

$$\forall x \forall y (P(x, y) \leftrightarrow (\exists z \exists z' ((x = a) \wedge (y = z) \wedge Q(z, z') \wedge \neg R(z')) \vee ((x = b) \wedge (y = c))))$$

Some predicate symbols in the program may not appear in the head of any program clause. For each such predicate Q, the *completed definition* of Q is the formula

$$\forall x_1 \cdots \forall x_n \neg Q(x_1, \ldots, x_n)$$

Definition : The *equality theory* (or *identity theory*) for a completed program contains the following axioms:

(a) $c \neq d$, for all pairs c, d of distinct constants (the symbol $\neq$ stands for not equal).

(b) $\forall x_1 \cdots \forall x_n \forall y_1 \cdots \forall y_m (f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m))$, for all pairs f, g of distinct function symbols.

(c) $\forall x_1 \cdots \forall x_n (f(x_1, \ldots, x_n) \neq c)$, for each constant c and function symbol f.

(d) $\forall x (t[x] \neq x)$, for each term $t[x]$ containing x and different from x.

(e) $\forall x_1 \cdots \forall x_n \forall y_1 \cdots \forall y_n ((x_1 \neq y_1) \wedge \cdots \wedge (x_n \neq y_n) \rightarrow f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n))$, for each function symbol f.

(f) $\forall x_1 \cdots \forall x_n \forall y_1 \cdots \forall y_n ((x_1 = y_1) \wedge \cdots \wedge (x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$, for each function symbol f.

(g) $\forall x (x = x)$

(h) $\forall x_1 \cdots \forall x_n \forall y_1 \cdots \forall y_n ((x_1 = y_1) \wedge \cdots \wedge (x_n = y_n) \rightarrow (P(x_1, \ldots, x_n) \rightarrow P(y_1, \ldots, y_n)))$, for each predicate symbol P (including =).

Note that axioms (g) and (h) together imply that = is an equivalence relation.

Definition : The *completion* of Π [16, 56, 63] , denoted by *comp*(Π), is the collection of completed definitions of predicate symbols in P together with the equality theory.

Among the resolution methods mentioned in section 2.4.1, the SLDNF-resolution is of most interest to us as it is the present basis of logic programming. The following subsection is devoted to the SLDNF-resolution and different aspects of it.

2.5.2. SLDNF-resolution in definite programs

This section deals with different aspects of SLDNF-resolution [16, 63] such as SLDNF-derivation, SLDNF-refutation, and SLDNF-tree in definite programs.

Definition : Let G be a goal $\leftarrow A_1 \wedge \cdots \wedge A_m \wedge \cdots \wedge A_k$, and C be $A \leftarrow B_1 \wedge \cdots \wedge B_q$. If θ is the mgu of A_m and A then the goal

$$\leftarrow (A_1 \wedge \cdots \wedge A_{m-1} \wedge B_1 \wedge \cdots \wedge B_q \wedge A_{m+1} \wedge \cdots \wedge A_k)\theta$$

is a *resolvent* of G and C. If $k=1$ and $q=0$ then the resolvent goal is an empty goal.

Definition : A *computation rule* is a function from a set of goals to a set of literals such that the value of the function for a goal is a literal, called the *selected literal*, in that goal. A computation rule is *safe* if negative literals may only be selected if they are ground.

Definition : Let P be a definite program, G be a goal. The definitions of an *SLDNF-refutation of rank k of* $P \cup \{G\}$ and a *finitely failed SLDNF-tree of rank k* for $P \cup \{G\}$, for every non-negative integer k, can be found in [63]. An *SLDNF-refutation* of $P \cup \{G\}$ is an SLDNF-refutation of rank k of $P \cup \{G\}$, for some k. A *finitely failed SLDNF-tree* for $P \cup \{G\}$ is a finitely failed SLDNF-tree of rank k for $P \cup \{G\}$, for some k.

Definition : Let P be a definite program and G a goal. An *SLDNF-derivation* of $P \cup \{G\}$ consists of a sequence $G_0=G, G_1, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of clauses of P or negative literals, and a sequence $\theta_1, \theta_2, \ldots$ of substitutions satisfying the following:

(a) For each i, either

(i) G_i is $\leftarrow L_1 \wedge \cdots \wedge L_k$ and the selected literal L_m is positive. Suppose $C_i: A \leftarrow M_1 \wedge \cdots \wedge M_q$ is the input clause and L_m and A have mgu θ_{i+1}. Then the derived goal G_{i+1} is

$$\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge M_1 \wedge \cdots \wedge M_q \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta_{i+1}$$

(ii) G_i is $\leftarrow L_1 \wedge \cdots \wedge L_k$, the selected literal L_m in G_i is a ground negative literal $\neg A$ and there is a finitely failed SLDNF-tree for $P \cup \{\leftarrow A\}$. In this case, θ_{i+1} is the identity substitution, C_{i+1} is $\neg A$ and G_{i+1} is

$$\leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k$$

(b) If the sequence $G_0, G_1, \ldots$ is finite, then either

(i) the last goal is empty, or

(ii) the last goal is $\leftarrow L_1 \wedge \cdots \wedge L_k$ and the selected literal L_m is positive and there is no program clause whose head unifies with L_m, or

(iii) the last goal is $\leftarrow L_1 \wedge \cdots \wedge L_k$ and the selected literal L_m is a ground negative literal $\neg A$ and there is an SLDNF-refutation of $P \cup \{\leftarrow A\}$.

An SLDNF-derivation is *finite* if it consists of a finite sequence of goals; otherwise, it is *infinite*. An SLDNF-derivation is *successful* if it is finite and the last goal is the empty goal. A successful SLDNF-derivation is indeed an SLDNF-refutation. An SLDNF-derivation is *failed* if it is finite and the last goal is not the empty goal.

Definition : Let P be a definite program, G be a goal. An *SLDNF-tree* for $P \cup \{G\}$ is a tree satisfying the following;

(a) Each node of the tree is a goal.

(b) The root node is G.

(c) Let $\leftarrow L_1 \wedge \cdots \wedge L_k$ $(k \geq 1)$ be a non-leaf node in the tree and suppose that a positive literal L_m is selected. Then this node has a descendant for each input clause $A \leftarrow M_1 \wedge \cdots \wedge M_q$ such that L_m and A are unifiable. The descendant is

$$\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge M_1 \wedge \cdots \wedge M_q \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$$

where θ is an mgu of L_m and A.

(d) Let $\leftarrow L_1, \ldots, L_k$ $(k \geq 1)$ be a non-leaf node in the tree and suppose that the selected literal L_m is a ground negative literal of the form $\neg A$ and there is a finitely failed SLDNF-tree for $P \cup \{\leftarrow A\}$. Then the single descendant of the node is

$$\leftarrow (L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_k).$$

(e) Let $\leftarrow L_1, \ldots, L_k$ $(k \geq 1)$ be a leaf node in the tree and suppose that the literal L_m is selected. Then either

(i) L_m is positive and there is no program clause in P whose head unifies with L_m, or

(ii) L_m is a ground negative literal $\neg A$ and there is an SLDNF-refutation of $P \cup \{\leftarrow A\}$.

(iii) A node which is the empty goal has no descendants.

In an SLDNF-tree, a branch which terminates at an empty goal is called a *success branch*, a branch which does not terminate is called an *infinite branch* and a branch for which terminates at a non-empty goal is called a *failure branch*. An SLDNF-tree for which every branch is a failure branch is indeed a finitely failed SLDNF-tree. Each branch of an SLDNF-tree corresponds to an SLDNF-derivation.

When a positive literal in an SLDNF-resolution is selected, SLD-resolution [54] has been used to derive a new goal. When a ground negative literal is selected, a recursive process is established to apply the *negation as failure* (NAF) rule.

Definition : Let P be a definite program and G a goal. A *computation* of $P \cup \{G\}$ represents an attempt to construct an SLDNF-derivation of $P \cup \{G\}$.

Definition : Let P be a definite program and G a goal. A computation of $P \cup \{G\}$ *flounders* if at some point in the computation a goal is reached which contains only non-ground negative literals.

Definition : Let P be a program and G a goal. A clause $A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n$ in P is *allowed* [63] (or *range-restricted* [35]) if every variable that occurs in the clause occurs in a positive literal of the body $L_1 \wedge \cdots \wedge L_n$. The whole database D is *allowed* if each of its clauses is allowed. A goal G is *allowed* if G is $\leftarrow L_1 \wedge \cdots \wedge L_n$ and every variable that occurs in G occurs in a positive literal of the body $\leftarrow L_1 \wedge \cdots \wedge L_n$.

The restriction of allowedness on a definite program P is introduced in order to prevent floundering of any computation $P \cup \{G\}$, for some goal G. In the context of definite programs, the allowedness restriction corresponds exactly to the notion of *range-restriction* introduced by Decker [25] and Kowalski et al. [55].

Definition : Let P be a definite program and G be a goal $\leftarrow W$. An *answer substitution* for $P \cup \{G\}$ is a substitution for the variables of G. A *computed answer* θ for $D \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \cdots \theta_n$ to the variables of G, where θ_1, ..., θ_n is the sequence of substitutions used in an SLDNF-refutation of $P \cup \{G\}$.

When a definite program P and a goal G are allowed, no computation of $P \cup \{G\}$ flounders and every computed answer for $P \cup \{G\}$ is a ground substitution for variables in G. A computed answer for $P \cup \{G\}$ will also be referred to as an answer substitution for the SLDNF-

refutation of $P \cup \{G\}$.

2.6. Logic programming

Logic programming is concerned with using logic to write programs and with their execution. A logic program [27, 26, 52, 53, 63, 101, 105] consists of a set of sentences which represents knowledge about the problem that the program is intended to solve. First-order logic serves as a tool for representing knowledge and the resolution mechanism within it can be used for executing a logic program. Present logic programming uses a subset of first-order wffs to represent knowledge and SLDNF-resolution mechanism for their execution. Prolog is a language based on these principles. Work is underway to build full first-order [69] and higher-order [75] Prologs. The logic programming language nH-Prolog [69, 93] uses full first-order clauses as sentences and λ-Prolog [78] allows higher-order sentences [43, 7]. Work in the area of developing semantics of logic programming can be found in [38, 39, 46, 50, 57, 84, 83, 85, 37]. Interesting work defining algebra of logic programs can be found in [71, 72].

2.6.1. Prolog

Prolog [8, 17, 29, 95] is a logic programming language based on ordered input resolution (a restriction of model elimination to Horn clauses) with left to right resolution on literals. It also provides a form of negation based on negation as failure. A Prolog program consists of a set of definite clauses. A query to a Prolog program is represented by a goal. It works with the same mechanism as SLDNF with the following two restrictions:

(a) The computation rule in standard Prolog systems always selects the leftmost atom in a goal.

(b) Standard Prolog uses the order of clauses in a program as the fixed order in which clauses are to be tried, i.e. search tree is searched depth-first.

Prolog provides a system predicate called *cut* ('!') to reduce the search space of a computation. Also, there is a set of extra-logical predicates in Prolog such as to perform I/O (*read*, *write* etc.), manipulate programs (*assert*, *retract* etc.) etc.

Example : Consider the following program:

Π: $R1$: $P(x,y) \leftarrow Q(x,y) \wedge \neg R(y)$

$R2$: $P(x,y) \leftarrow S(x) \wedge R(y)$

$F1$: $Q(a,b)$

$F2$: $Q(a,c)$

$F3$: $R(c)$

$F4$: $S(a)$

$F5$: $T(a)$

The SLDNF-tree (Prolog style) for $\Pi \cup \{\leftarrow P(x,y) \wedge T(x)\}$ is given in figure 2.1. The left most derivation is given in table 2.1.

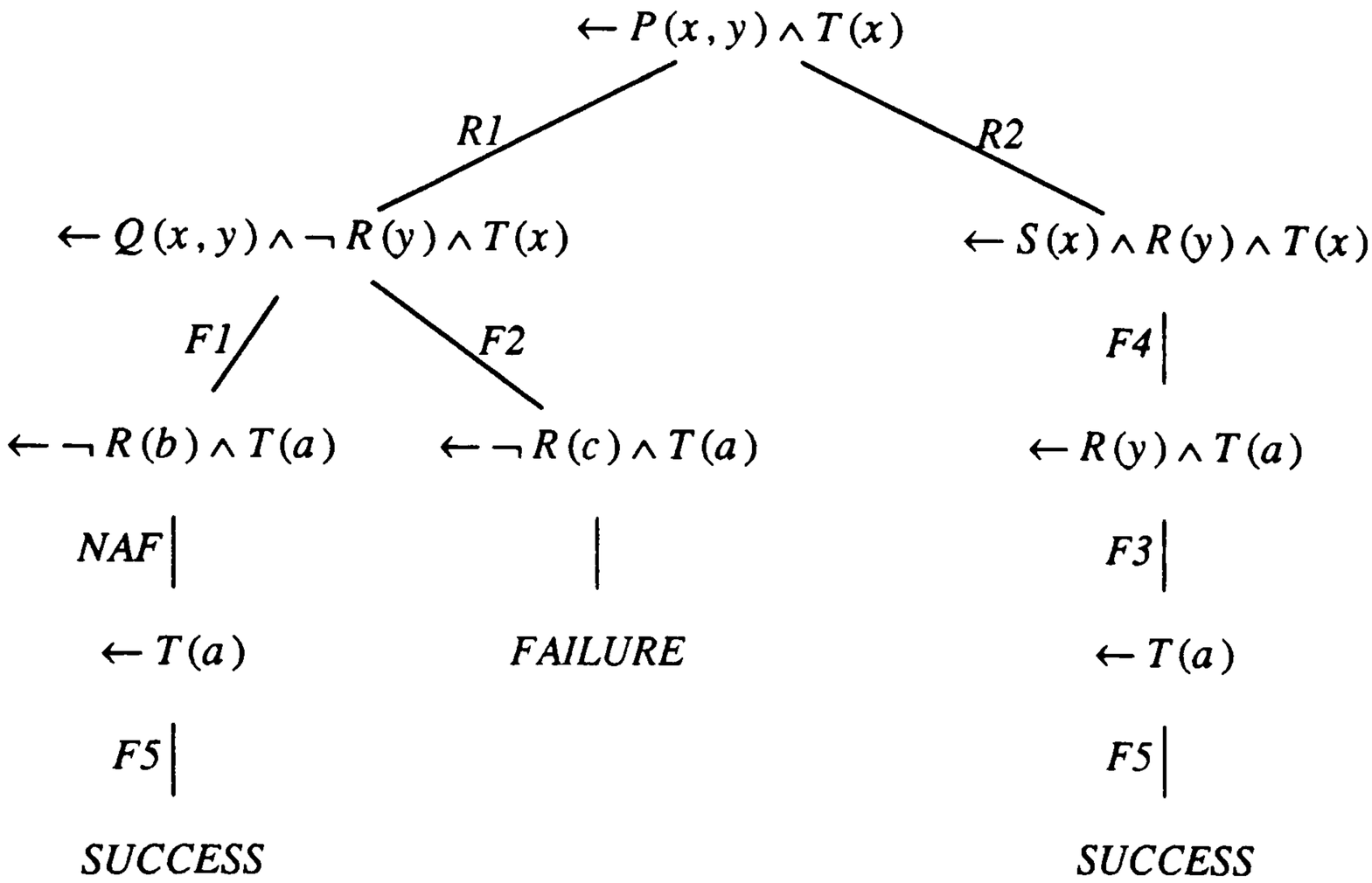

Figure 2.1. The complete SLDNF-tree for $\Pi \cup \{\leftarrow P(x,y) \wedge T(x)\}$.

Goals	Variants	Substitutions
$G_0 = \leftarrow P(x, y) \wedge T(x)$		
$G_1 = \leftarrow Q(x, y) \wedge \neg R(y) \wedge T(x)$	$C_1 = P(x, y) \leftarrow Q(x, y) \wedge \neg R(y)$	$\theta_1 = \{x/x, y/y\}$
$G_2 = \leftarrow \neg R(b) \wedge T(a)$	$C_2 = Q(a, b)$	$\theta_2 = \{x/a, y/b\}$
$G_3 = \leftarrow T(a)$	$C_3 = \neg R(b)$	$\theta_3 = \{\}$
$G_4 = \square$	$C_4 = T(a)$	$\theta_4 = \{\}$

Table 2.1. An SLDNF-derivation for $\Pi \cup \{\leftarrow P(x, y), T(x)\}$.

The SLDNF-tree (Prolog style) for $\Pi \cup \{\leftarrow P(x, y) \wedge ! \wedge T(x)\}$ will contain only the above derivation, i.e. the leftmost branch of figure 2.1.

Throughout this thesis a standard C-Prolog [17] syntax has been used to represent a Prolog program or a query. In C-Prolog, the reverse implication sign '$\leftarrow$' in a clause is denoted by ':–' and '$\leftarrow$' symbol in a goal is denoted by '?–'. The '$\wedge$' in the body of a clause is represented by ','.

CHAPTER 3

Logic and Databases

Both relational and deductive database models are discussed in this chapter and in particular it is shown how each of them can be viewed through logic.

Definition : A *data model* consists of the following:

(a) A mathematical notation for the formal description of data and their relationships, and

(b) A technique for the manipulation of data such as answering queries, checking integrity.

3.1. The relational data model

The set-theoretic relation is used as a mathematical notation for describing data in the relational data model [18]. A relation can be characterised as a two-dimensional table where each row is a *tuple* and each column is an *attribute*. Each entry of the table is an *attribute value*. A *relation scheme* of a relation is the collection of all attribute names for the relation. A *relational database scheme* is the collection of relation schemes used to represent information, and the set of values of the relations is called the *relational database*. Consider the following example of a relational database:

Example 3.1 : Consider the four relations STUDENT, EXAM, SPECIAL COURSE and COURSE given in tables 3.1, 3.2, 3.2 and 3.4 respectively.

STUDENT		
SNo	Group	Result
11111	Sc	Pass
22222	Com	Pass
33333	Sc	Incm
44444	Hum	Pass
55555	Sc	Pass
66666	Com	Fail
77777	Com	Pass
88888	Sc	Fail
99999	Hum	Prom

Table 3.1

EXAM		
SNo	Course	Mark
88888	Phy	55
11111	Chem	65
77777	Acc	62
55555	Math	91
11111	Math	55
44444	Hist	66
55555	Phy	58
77777	BM	72
44444	Geo	46
88888	Chem	45
88888	Math	35
11111	Phy	85
77777	Eco	49
55555	Chem	81
44444	PS	60
22222	BM	65
66666	Acc	38
33333	Math	49
22222	Eco	68
66666	Eco	63
22222	Acc	51
33333	Chem	69
66666	BM	61
99999	Hist	49
99999	PS	65

Table 3.2

SPECIAL COURSE	
SNo	Course
55555	Phy*
11111	Chem
44444	Geo*
11111	Math*
77777	Acc
99999	Hist*
99999	Geo*
77777	Eco*
22222	Acc*

Table 3.3

COURSE	
Course	Group
Phy	Sc
Chem	Sc
Math	Sc
Hist	Hum
Geo	Hum
PS	Hum
Acc	Com
Eco	Com
BM	Com

Table 3.4

* tuples quoted by asterisk are implicit in the database. Without being stored explicitly they can be derived from the database with the help of deductive rules.

In example 3.1, <11111, *Sc*, *Pass*>, <22222, *Com*, *Pass*>, ... are tuples of the STUDENT relation. 11111, 22222, ... are attribute values of the attribute *SNo*. The STUDENT relation scheme contains the attributes *SNo*, *Group* and *Result*. The example database schema contains relations STUDENT, EXAM, SPECIAL COURSE and COURSE. A student can either pass (*Pass*) or fail (*Fail*) or be specially promoted (*Prom*) or have an incomplete result (*Incm*). A student fails if s/he secures less than 40 in any course. A passed student who obtains less than 60 in a particular course will have to go for a special course on that particular course. If a promoted student did not appear or appeared but secured less than 60 for the exam of a particular course then that student will have to go for a special course on that particular course. Anyone else can attend a special course if s/he wishes.

A number of *data manipulation languages* (or *query languages*) have been associated with the relational model. A number of operations, for example, *select*, *join*, *project*, are defined in data manipulation languages for the manipulations of data in relational databases. The effect of these operations can be obtained by expressing *queries* [13]. The query language for the relational model can be divided into two classes, *relational algebra* [18] and *relational calculus*. In relational algebra queries are expressed by applying operators to relations, whereas in the relational calculus approach a query describes a desired set of tuples by specifying a predicate that the tuple must satisfy. Examples of calculus based languages are QUEL [100] and QBE [109]. An example of an algebra based language is SQL [21]. A typical query using the operation *select*, *join* in the context of example 3.1 could be

find all students from the science group who have obtained more than 50 in mathematics.

The above query can be expressed in SQL as

```
SELECT   S.SNo
FROM     STUDENT S, EXAM E
WHERE    S.SNo = E.SNo
  AND    S.Group  = 'Sc'
  AND    E.Course = 'Math'
  AND    E.Mark > 50.
```

The following hypotheses are made in definitions of the relational model for evaluation of queries:

(a) The *closed world assumption* (CWA) states that all information that is not true in the database is assumed to be false (i.e. $\neg R(a_1, ..., a_n)$ is assumed to be true iff the tuple $<a_1, ..., a_n>$ is not explicitly present in the extension of R).

Example : From the CWA it can be said that the tuple <55555, *Chem*> is not in the SPECIAL COURSE relation, i.e. the student identified by the number 55555 is not taking a special course on chemistry.

(b) The *unique name assumption* (UNA) states that constants with different names are different.

Example : From UNA it can be said that the two constants 11111 and 22222 appearing in the relation STUDENT are identifying two different students uniquely.

(c) The *domain closure assumption* (DCA) states that there are no other constants than those in the database.

Example : From DCA it can be said that there is no student with number 12345.

A database can be considered from the viewpoint of logic as

(1) An interpretation of a first order theory, or

(2) A first-order theory.

From the viewpoint of (1), queries and integrity constraints are formulae that are to be evaluated using the semantic definition of truth. From the viewpoint of (2), queries are considered to be theorems that are to be proved and integrity constraints are either considered to be theorem that are to be proved or formulae that are to be consistent with the theory. These two approaches are referred to as the *model-theoretic view* and the *proof-theoretic view* [35] respectively.

3.1.1. Model-theoretic view of relational databases

Let DB be a relational database. Define a first order language L as follows.

(1) L contains an n-place predicate symbol **R** for each n-ary relation R in DB.

(2) L contains a set of constants, one for each element in D, where D is the union of the underlying domains of all attributes that occurs in the relation schema.

(3) There are no function symbols in L.

(4) Arithmetic comparison operators ($<$, $=$, $\neq$, $>$, $\leq$, $\geq$) are assigned their usual interpretation, if they are required.

The database DB can be taken as an interpretation of the first-order theory consisting of the above language L and the variables of L as ranging over the domain D in this interpretation. The evaluation of logical formulae of L in this interpretation is based on:

$$\mathbf{R}(a_1, ..., a_n) \text{ is true iff } <a_1, ..., a_n> \in R.$$

The above way of evaluation preserves the properties of CWA, DCA and UNA. The least Herband model of the constructed theory contains exactly the information that one intended to store.

Although a relational database is generally considered as an interpretation of a first-order theory, i.e. from a model-theoretic point of view, it can also be considered from a proof-theoretic point of view.

3.1.2. Proof-theoretic view of relational databases

Let DB be a relational database and L be a first-order language defined as in the previous section. The proof-theoretic point of view of a relational database DB is obtained by constructing a theory T underlying the language L. The proper axioms of the theory T are as follows:

(a) *Assertions*. For each relation R in DB and for each tuple $<a_1, ..., a_n> \in R$, the assertions $\mathbf{R}(a_1, ..., a_n)$.

(b) *The completion axioms*. For each relation R, if $<a_1^1, ..., a_n^1>$, ..., $<a_1^m, ..., a_n^m>$ denote all the n-tuples of R, the completion axiom for R

$$\forall x_1, ..., \forall x_n (\mathbf{R}(x_1, ..., x_n) \rightarrow (x_1 = a_1^1 \wedge \cdots \wedge x_n = a_n^1) \vee \cdots \vee (x_1 = a_1^m \wedge \cdots \wedge x_n = a_n^m).$$

(c) *The unique name axioms*. If $a_1, ..., a_p$ are all constants in DB, then

$$(a_1 \neq a_2), ..., (a_1 \neq a_p), (a_2 \neq a_3), ..., (a_{p-1} \neq a_p).$$

(d) *The domain closure axioms.*

$$\forall x((x = a_1) \vee \cdots \vee (x = a_p)).$$

(e) *Equality axioms*. The following axioms

$$\forall x(x = x),$$

$$\forall x\ \forall y((x = y) \rightarrow (y = x)),$$

$$\forall x\ \forall y\ \forall z((x = y) \wedge (y = z) \rightarrow (x = z)),$$

$$\forall x_1 \cdots \forall x_n(P(x_1, ..., x_n) \wedge (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n) \rightarrow P(y_1, ..., y_n)).$$

It can be shown that for any U in L, $T \vdash U$ iff U is true in DB. Hence the above two approaches are equivalent.

3.2. The deductive database model

The first-order language is used as a mathematical notation for describing data in a deductive database model. It can also be used as a language for expressing views, queries and integrity constraints. Such databases are called deductive databases (also called *logic databases*, see [34]) because they are able to make deductions from known facts and rules at the time of answering queries. Throughout this thesis, deductive database are considered from the proof-theoretic point of view.

Definition : A *(database) clause* is a program clause (i.e. a clause) and a *deductive database* is a program (i.e. a finite set of clauses). In the context of deductive databases, the different forms of the clause

$$A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n \qquad m \geq 1$$

corresponding to different values of m and n are as follows:

(a) $m=1$, $n=0$, i.e. a unit clause. This clause is called a *(definite) fact.*

(b) $m>1$, $n=0$, i.e. the clause has the form

$$A_1 \vee \cdots \vee A_m,$$

in which the body is empty and the head is a disjunction of more than one atoms. This clause is called an *indefinite fact.*

(c) $m=1, n>0$, i.e. a definite clause with non-empty body. This clause is called a *(definite) rule.*

(d) $m>1, n>0$, i.e. the head is a disjunction of atoms and the body is non-empty. This clause is called an *indefinite rule*.

Thus, a *definite* (*database*) *clause* is either a fact or a rule. A *general* or *indefinite* (*deductive*) *database* is a general program (i.e. finite set of clauses). A *definite* (*deductive*) *database* is a definite program (i.e. finite set of facts and rules). As in the case of programs, a definite database is a special form of a general or indefinite database.

Example 3.2 : A definite database form of example 3.1 consists of the following clauses:

$SNo(11111, Sc, Pass)$, ... etc.

$Exam(88888, Phy, 55)$, ... etc.

$SpecialCourse(x, y) \leftarrow Student(x, z, Prom) \wedge Course(y, z) \wedge \neg Exam(x, y, t)$,

$SpecialCourse(x, y) \leftarrow Student(x, z, Prom) \wedge Course(y, z) \wedge Exam(x, y, t) \wedge t<60$,

$SpecialCourse(x, y) \leftarrow Student(x, z, Pass) \wedge Exam(x, y, t) \wedge t<60$,

$SpecialCourse(11111, Chem)$, $SpecialCourse(77777, Acc)$,

$Course(Phy, Sc)$, ... etc.

Definition : A (*database*) *query* is a goal. For example, the query

find all students from the science group who have secured more than 50 in mathematics

can be expressed in the form of a goal as

$$\leftarrow Student(x, Sc, y) \wedge Exam(x, Math, z) \wedge z>50.$$

Definition : A database DB (resp. query G) is called *allowed* (or *range-restricted*) if the program DB (resp. goal G) is allowed.

To avoid floundering in the construction of a derivation of $DB \cup \{G\}$, where DB is a database and G a goal, it has been assumed in the rest of the thesis that both DB and G are *allowed.*

3.2.1. Proof-theoretic view of definite databases

Let DB be a definite database and L be a first-order language defined as follows:

(1) L contains an n-place predicate symbol **R** for each n-ary predicate R appearing in DB.

(2) L contains a set of constants, one for each element in D, where D is the set of all constants appearing in DB.

(3) L contains a set of function symbols, one for each element in F, where F is the set of all function symbols appearing in DB.

(4) Arithmetic functions (+, −, * , /) and arithmetic comparison operators (<, =, ≠, >, ≤, ≥) are assigned their usual interpretation, if they are required.

The proof-theoretic point of view of DB is obtained by constructing a theory T underlying the language L. The proper axioms of the theory T are as follows:

(a) *Completion axioms*. Obtained by completing each predicate symbol of L (as defined in section 2.5.1) according to the clauses of DB.

(b) *Equality and Unique name axioms*. Axioms of the equality theory (defined in section 2.5.1) according to the constant, function and predicate symbols of L.

(c) *Domain closure axiom*. If $a_1, ..., a_p$ are all the elements of D and $f_1, ..., f_q$ are all the function symbols of F, then the domain closure axiom [63] is as follows:

$$\forall x((x=a_1) \vee \cdots \vee (x=a_p) \vee (\exists x_1 \cdots \exists x_m (x=f_1(x_1, ..., x_m))) \vee \cdots \vee (\exists y_1 \cdots \exists y_n (x=f_q(y_1, ..., y_n)))).$$

Considering DB as a definite program, the above proof-theoretic point of view reduces to $comp(DB)$ plus the above domain closure axiom. The domain closure axiom may be avoided by dealing with allowed queries and clauses. It has been shown in [16, 63] that there is an equivalence (sound but not always complete) between the completion axioms and unique name axioms and the *negation as failure* [16] rule. Hence, from the operational point of view a definite database consists of the following:

(i) A set of definite clauses each of whose elements is allowed.

(ii) The negation as failure rule.

All the notions and results introduced in chapter 2 in the context of definite programs and goals can be taken as valid in the context of definite databases and queries respectively. The terms 'definite program' and 'definite database' will be used synonymously in the remaining chapters. It is obvious that a relational database is a particular definite database and hence a particular indefinite database. Unless otherwise stated, function symbols will be allowed in a database clause.

3.2.2. Transactions on deductive databases

Definition : A *transaction* on a deductive database *DB* is a finite sequence of operations (actions), *insertion*, *deletion* or *update* of clauses.

An update of a clause in a deductive database is normally taken as a deletion followed by an addition. It has been assumed so in the rest of the thesis except in chapter 10. In chapter 10, update operations have been performed only on facts, explicitly present in the database, and they have not been assumed as deletion followed by addition. Instead, their own identities have been maintained to compute implicitly updated facts and to check integrity constraints relevant to the updated facts.

As a deductive database is viewed as a set of clauses rather than their models, no operations is allowed on a fact (either definite or indefinite) which is implicit in the database (i.e. not present in the database, but can be derived from the database). Work related to this can be found in [31, 4, 24].

Definition : A transaction is said to have been *committed* successfully if the entire sequence of operations forming the transaction completes successfully. The main reasons for failing to complete a transaction are, for example, the violation of an *integrity constraint* by operations in the transaction, systems failure, infinite computation or user interruptions.

CHAPTER 4

Integrity Constraints in Databases

The purpose of integrity constraints is to ensure that the database remains in a valid state. This chapter is devoted to different issues of integrity constraints. The formal definition through logic and a detailed classification of integrity constraints are given. Different views of constraint satisfiability are discussed and compared in detail. Finally, the notion of the *denial form* of constraints is introduced and it is shown that a general integrity constraint in closed first-order form can be converted to this form.

4.1. Integrity constraints

In a database, (*semantic*) *integrity* is concerned with limiting the possibility of erroneous modifications being performed and hence incorrect information being stored in a database. Modifications to a database can be performed via transactions. One of the reasons for failing to complete a transaction is the violation of an *integrity constraint* by operations in the transaction.

Definition : Part of the semantics of a database is expressed as (*integrity*) *constraints* [20, 104, 106, 30, 99, 33]. Constraints are properties that the data of a database are required to satisfy.

Some typical constraints in the context of the database of example 3.2 in the previous chapter are as follows:

(i) *each value of the domain of attribute Mark in the EXAM relation must be between 0 and 100,*

(ii) *the domain of attribute SNo in the EXAM relation must be a subset of that in the STUDENT relation.*

The former one can be considered as a *range* constraint whereas the latter is an *existential* constraint.

4.1.1. Classification of constraints

As an initial formal classification of all constraints, the whole class can broadly be divided into two, *immediate* and *deferred*. Constraints that are required to be satisfied after the completion of a transaction are known as *deferred constraints*. Constraints that are required to be satisfied after any action are known as *immediate constraints*. To clarify this distinction consider the constraint which expresses that

the sum of debit and credit in a ledger is equal to the budget amount

Suppose a transaction, corresponding to an expense incurred, causes an amount to be subtracted from the credit amount and that an amount is to be added to the debit amount. The consistency of the database is preserved if the above constraint is treated as a deferred constraint. Consistency would be violated if the same constraint was treated as an immediate constraint.

Considering every action on a database as a single transaction, the class of immediate constraints can be verified in the same way as deferred constraints. Hence, in the thesis constraints are considered as deferred and any further references to constraints will be taken as to deferred constraints.

The set of all deferred constraints can be subdivided into the set of *static* constraints and the set of *transitional* (or *dynamic*) constraints. Static constraints deal with information in a single state of the world. The current state is independent of previous or future states. The following are some examples of static constraints.

(a) *Name of a department belongs to the domain {Fin, Admin, Comp}.*

(b) *An employee's salary must be less than 10000.*

(c) *Every worker has a manager.*

(d) *Every child has only one father.*

(e) *If one of the parents of an unemployed person is employed then the person must be dependent on that parent.*

(f) *Average salary of any department must be less than 10000.*

(g) *Total salary of all the employees in a department must be less than the budget of the department.*

(h) *Maximum number of employee in a department is 100.*

The static constraints can be sub-divided into two categories - *aggregate* and *non-aggregate*. Static aggregate constraints involve one or more aggregate operations like *Count*, *Sum*, *Average*, *Maximum*, *Minimum*. For example, the constraint (h) requires the help of the *Count* operation on the *Employee* relation and hence this will be classified as *static aggregate count* constraint. More on aggregate constraints can be found in chapter 9.

Static non-aggregate constraints are classified into the following five major categories:

1. *Not Null* : Can be specified for any attribute and any attempt to introduce null in such an attribute is rejected. In the rest of this thesis, the considered databases are assumed to be satisfy this constraint.

2. *Check* : Given a relation R, this constraint can be specified for any attribute or combination of attributes within R, provided Not Null is also specified for every column involved. Any attempt to introduce a tuple into R that violates the specified constraint, e.g., the attribute value is not in the specified *Range* (example (b)) or does not belong to the specified *Domain* (example (a)), is rejected.

3. *Unique* : Given a relation R, this constraint can be specified for any attribute or combination of attributes within R, provided Not Null is also specified for every attribute involved. Any attempt to introduce a tuple in R having the same value in the specified attribute or combination of attributes as some existing row will be rejected. *Primary Key* is a special case of Unique. The identified attribute or combination of attributes constitute the primary key for the relation R.

4. *Existential* : An existential constraint is specified on an attribute or a combination of attributes in one relation whose values are required to match values of an attribute (the primary key, in the case of *Foreign Key*) in some relation (example (c)). *References* is an alternative way of specifying foreign keys. The imposed integrity constraint is called a *referential constraint* [22, 12].

5. *Multivalued Dependence* : Multivalued dependencies [32] are a generalisation of *functional dependencies* [32, 11]. Given a relation R, attribute Y of R is functionally dependent on attribute X if and only if each X-value in R has associated with it precisely one Y-value in R (example (d)).

A different classification of constraints concerns whether a constraint is applied to an individual tuple of a base relation or to more than one tuple from different relations. The former class of constraints would be considered as *single record* (or *single tuple*) whereas the latter would be denoted as *multiple record* (or *multiple tuple*). This classification has a special implication for the process of constraint checking. When a transaction is encountered, all the single record constraints can be verified by looking at the transaction alone without accessing the database. But for verifying a multiple record constraint, one has to access the database. Examples (a) and (b) fall into the single record category whereas (c)-(h) fall into multiple record category. As an example, the constraint (b) is applied only to an individual tuple of the *Employee* relation. On the other hand, the constraint (c) is applied to two tuples, one from *Worker* and the other from *Manager*. Likewise the two constraints (g) and (h) are applied to all the tuples of the relation *Employee* who are working in a particular department.

Unlike static constraints, transitional constraints deal with the way in which world evolves, i.e. an imposed constraint is related to at least two states of the database. Consider the constraint, *the minimum loan amount is 10000*. This means that if a person negotiates a loan then initially the amount of the loan has to be greater than or equal to 10000. Subsequently, when the person repays the loan amount by installments, the loan amount outstanding may be less than 10000. Hence transitional constraints have to be satisfied only at the time the operation occurs. Here are some examples of transitional constraints considered in the cases of addition, deletion and updates of tuples to the database.

(I) *Initially, the minimum loan amount is 10000.*

(II) *Lay-off of employees whose income is less than 1000 will be prevented.*

(III) *On updating the salary of an employee it should always increase.*

(IV) *The Maximum loan an employee can have is five times his/her salary unless the employee has been given permission for a special loan.*

(V) *A change in grade must be accompanied by a change in salary.*

More on transitional constraints can be found in chapter 10. Transitional constraints can be subdivided into *addition type*, *deletion type*, *Update Type* and a mixture of one or more of these. Each of the above types of transitional constraints are divided into two categories. An addition type transitional constraint involving aggregate operations (e.g., *Count*, *Sum*, *Average*, *Maximum*, *Minimum*) on one or more relations is classified as an *addition type aggregate transitional constraint*, whereas an addition type transitional constraint without involving aggregate on any relation is classified as an *addition type non-aggregate transitional constraint.* Similarly the case applies for deletion and update types of transitional constraints. For example, the constraints

(A) *on update of the salary of an employee, the difference of salaries should be between 10 and 100*

(B) *on insertion of a new employee in a department the new average salary of the department must be less than the old average salary of the department*

will be considered as an update type non-aggregate transitional constraint and an addition type aggregate transitional constraint respectively.

Figure 4.1 illustrates the scheme of classification of constraints. The symbol '...' denotes repetition and a 'Mixed' type of constraint in a particular level of the hierarchy denotes a constraint of a combination of types in that level.

4.1.2. Syntactic definition of constraints

In the previous chapter logic has been considered as the theoretical foundation of database systems. The idea behind this consideration was to use logic as a uniform language for data, programs, queries and constraints. In the context of logic, the definition of a constraint is as follows:

Definition : A constraint is a closed formula.

All the static constraints can be represented by closed first-order formulae. To represent static aggregate constraints conveniently, some aggregate predicates have been considered. This issue is discussed in detail in chapter 9. To represent a transitional constraint, it is necessary to allow some *action relations*. Using action relations and by modifying the database accordingly, it is

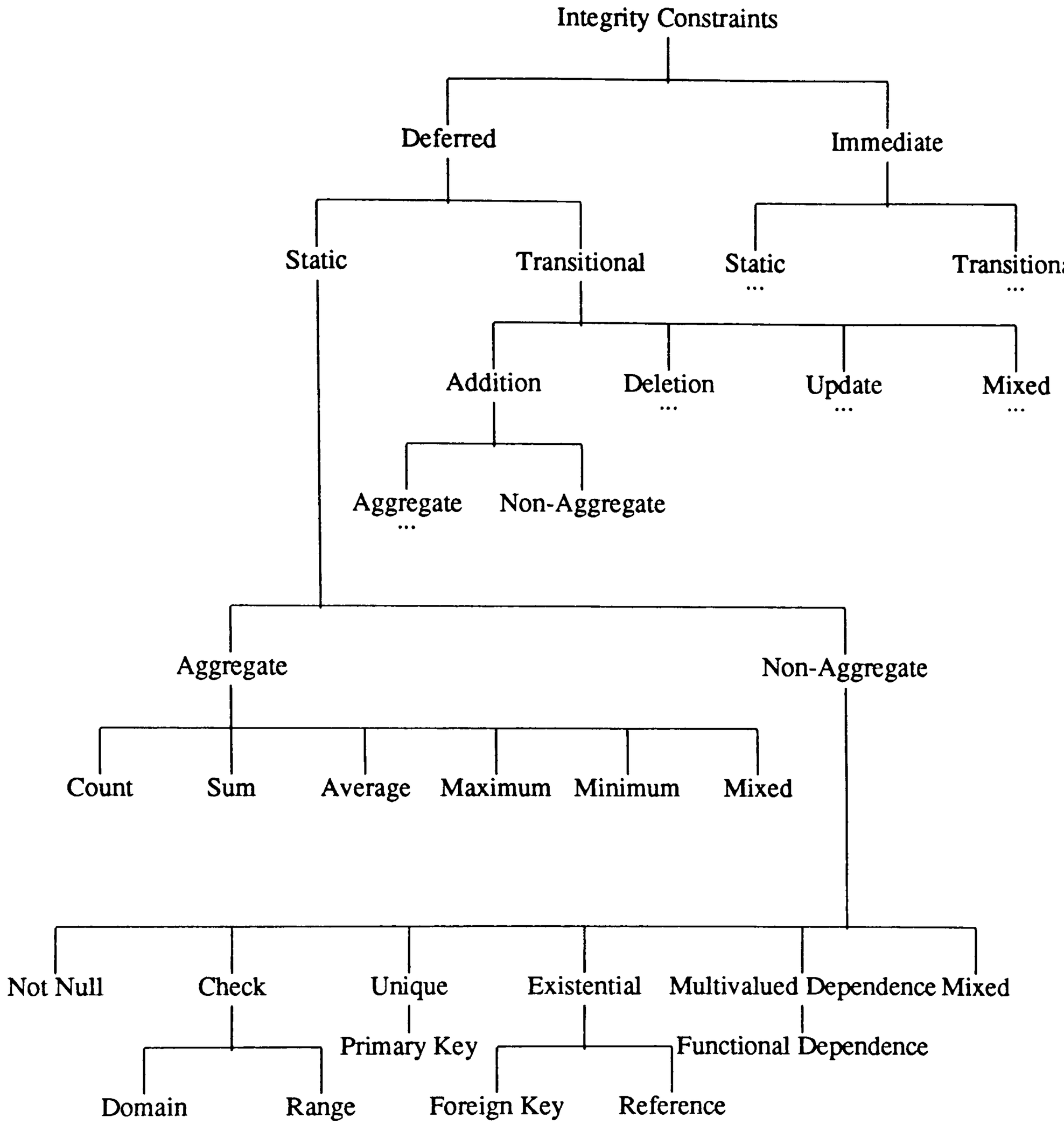

Figure 4.1. Classification of integrity constraints

possible to verify the satisfiability of transitional constraints in the same way as for static constraints. This issue is discussed in chapter 10. Considering constraints as closed first-order formulae, the representation of constraints (a)-(e) in this form can be illustrated as follows:

$$\forall x \forall y \forall z\,(Employee\,(x, y, z) \rightarrow y='Fin' \vee y='Admin' \vee y='Comp')$$

$$\forall x \forall y \forall z\,(Employee\,(x, y, z) \rightarrow z<10000)$$

$$\forall x\,(Worker\,(x) \rightarrow \exists y\ Manager\,(y, x))$$

$$\forall x \forall y\,(Father\,(x, z) \wedge Father\,(y, z) \rightarrow x=y)$$

$$\forall x \forall y (Parent(x, y) \wedge Employed(x) \wedge \neg Employed(y) \rightarrow Dependent(y, x))$$

Any further reference to the term 'constraint' in the rest of this chapter and in chapters 5-8 will refer to a 'non-aggregate static constraint'.

4.1.3. Constraint satisfiability in definite databases

A *constraint* is a closed first-order formula that a database (consider definite for this section) is required to satisfy. The *standard* definition of constraint satisfiability (*the theoremhood view*) in databases is as follows. Let D be a database such that $comp(D)$ is consistent. Then D is said to satisfy W, where W is a constraint, if W is a logical consequence of $comp(D)$; otherwise, D violates W. D is said to satisfy I, where I is a set of constraints, if D satisfies each constraint in I; otherwise, D violates I. This definition is followed in [66, 25, 10, 60].

An alternative definition of constraint satisfiability (*the consistency view*), adopted in [3, 91] is as follows. A database D satisfies constraints I if and only if the set of formulae comprising the completion of D together with the formulae in I, is consistent.

The above two definitions of constraint satisfiability are equivalent if and only if the completion of the database D is complete. If a constraint W is a theorem of the completion of a database D then $comp(D)$ together with W is always consistent. But the converse may not be true. Hence the latter definition of constraint satisfiability deals with a larger class of databases.

The alternative definition of constraint satisfiability may differ from the standard view in its outcome if either of the following two conditions is met :

(a) circular definitions are present in the database, or

(b) a predicate symbol occurs in the constraints but not in the database.

Each of the above points will be clarified with the help of examples.

To illustrate the problem associated with (a), consider the database

D: $P(a) \leftarrow P(a)$

where a is a constant, and the constraint

I: $P(a)$.

The database is recursive and its completion, apart from the equality axiom, is given by

$comp(D)$: $P(x) \leftrightarrow x=a \wedge P(a)$,

where x is a variable. $Comp(D) \cup I$ is consistent as $\{P(a)\}$ is a model of $comp(D) \cup I$. On the other hand $P(a)$ is not a theorem of $comp(D)$.

To clarify point (b), consider the database

D: $P(a)$,

where a is a constant, and a constraint

I: $Q(a)$.

The predicate symbol Q does not occur in the database. The completion of D, apart from the equality axiom, is given by

$comp(D)$: $P(x) \leftrightarrow x=a$

where x is a variable. $Comp(D) \cup I$ has a model $\{P(a), Q(a)\}$ and hence $comp(D) \cup I$ is consistent. But $Q(a)$ cannot be proved as a theorem from $comp(D)$.

To implement reasoning with the completion of the database, the methods in [66, 25, 55] make use of the notion of negation as failure.

Another view of constraint satisfiability, called the *meta* view, has recently been proposed by Kowalski [51]. In this view a constraint is a meta-statement which must be true of the database. None of the proposed methods exploits this view of constraint satisfaction.

4.1.4. Constraints in denial form

Different authors have represented constraints in different forms for the ease of their evaluation. This issue will be discussed in detail in chapter 6. In this section a special format for constraints, called *denial* form, will be introduced which will be needed in the next chapter for the proposed method of constraint evaluation in definite databases.

Definition : A *denial* [91, 55] is a goal, i.e. a formula of the form

$$\leftarrow L_1 \wedge \cdots \wedge L_n, \quad n>0,$$

where the L_i are literals and all variables are assumed to be universally quantified in front of the constraint.

It is possible to convert an arbitrary constraint to its equivalent denial form as follows. Let W be a constraint which is an arbitrary closed first-order formula which has been imposed on a definite database D. Then this constraint can be replaced by a new constraint $\leftarrow A$, where A is a predicate symbol of zero arity that does not occur elsewhere in the database or constraints, provided that a set of rules obtained by transforming

$$A \leftarrow \neg W$$

is added to the database D using the following method of transformation [67] :

(a) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg (U \wedge V) \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg U \wedge W_{i+1} \wedge \cdots \wedge W_n$

and $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg V \wedge W_{i+1} \wedge \cdots \wedge W_n$

(b) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \forall x_1 \cdots \forall x_m U \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg \exists x_1 \cdots \exists x_m \neg U \wedge W_{i+1} \wedge \cdots \wedge W_n$

(c) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg \forall x_1 \cdots \forall x_m U \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \exists x_1 \cdots \exists x_m \neg U \wedge W_{i+1} \wedge \cdots \wedge W_n$

(d) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge (U \leftarrow V) \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge U \wedge W_{i+1} \wedge \cdots \wedge W_n$

and $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg V \wedge W_{i+1} \wedge \cdots \wedge W_n$

(e) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg (U \leftarrow V) \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge V \wedge \neg U \; W_{i+1} \wedge \cdots \wedge W_n$

(f) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge (U \vee V) \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge U \wedge W_{i+1} \wedge \cdots \wedge W_n$

and $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge \cdots \wedge W_n$

(g) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg (U \vee V) \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg U \wedge \neg V \wedge W_{i+1} \wedge \cdots \wedge W_n$

(h) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg \neg U \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge U \wedge W_{i+1} \wedge \cdots \wedge W_n$

(i) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \exists x_1 \cdots \exists x_m U \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge U \wedge W_{i+1} \wedge \cdots \wedge W_n$

(j) Replace $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg \exists x_1 \cdots \exists x_m U \wedge W_{i+1} \wedge \cdots \wedge W_n$

by $A \leftarrow W_1 \wedge \cdots \wedge W_{i-1} \wedge \neg P(y_1, ..., y_k) \wedge W_{i+1} \wedge \cdots \wedge W_n$

and $P(y_1, ..., y_k) \leftarrow \exists x_1 \cdots x_m U$

where $y_1, ..., y_k$ are the free variables in $\exists x_1 \cdots x_m U$ and P is a new predicate symbol not already appearing in the program.

Suppose D' is the resulting database. Then W is a logical consequence of $comp(D)$ if and only if $\leftarrow A$ is a logical consequence of $comp(D')$. This justifies the transformation of a constraint to a denial.

To illustrate the above transformation process, consider the following example:

Example : Suppose the constraint

$$\forall x (Worker(x) \rightarrow \exists y \ Manager(y, x))$$

to be imposed on a database D. The above constraint can be transformed to a denial of the form

$$\leftarrow A$$

by applying successively (e) and (j) and with the addition of the rules

$$A \leftarrow Worker(x) \wedge \neg B(x)$$

$$B(x) \leftarrow Manager(y, x)$$

to the database D, where the two predicates A and B are not used elsewhere in the database D.

Unless otherwise mentioned, in the rest of the thesis, a closed first-order formula representing a constraint will be assumed to be in the form of a denial.

Definition : Let I be a set of constraints in denial form. Then I_d will represent the set of clauses of the form

$$IC(No) \leftarrow B$$

where $\leftarrow$B is a constraint from I and No is a unique identification of the constraint. $IC(No)$ will be called the *head* of the constraint $\leftarrow$B.

CHAPTER 5

The Path Finding Method

In this chapter, the *path finding method* is described for checking integrity in definite databases. A proof of the correctness of this method is given and the code for Prolog implementation of the the main algorithm is proposed. Some possible optimisations are also discussed. Throughout the chapter, the term 'database' will refer to a 'definite database' and the term 'constraint' a 'closed first-order formula in the form of an allowed denial'.

Suppose D is a database and t a transaction whose application to D produces D'. The path finding method of integrity checking in an updated database D' with a set of constraints I is based on finding a path in $D' \cup I_d$ from an update literal to the head of a clause in I_d. If such a path is found then the integrity in the updated database is violated; otherwise, if all possible attempts to find a path fail, the integrity is said to be preserved in the updated database.

5.1. The Path

Definition : Let D be a database. A *path* in D is defined as a chain of literals

$$L_0 \xrightarrow{R_1} L_1 \xrightarrow{R_2} \cdots \xrightarrow{R_n} L_n$$

where L_0 is called the *source* of the path, L_n its *destination*, n its *length* and R_1, ..., R_n are clauses from D *used* to construct the path from L_0 to L_n. If the source L_0 is positive then it is ground. For any two consecutive literals L_i and L_{i+1}, L_{i+1} is called the *successor* of L_i in the path, and is obtained from L_i in one of the following ways:

1. If

(a) L_i is positive, and

(b) L_i unifies with a positive literal L occurring in the body of the clause $R_i: H \leftarrow B$, and

(c) α is an mgu of L_i and L, and

(d) G' is a resolvent of $\leftarrow$B and $L_i \leftarrow$, and

(e) θ is a computed answer for $D \cup \{G'\}$

then L_{i+1} is H$\alpha\theta$.

2. If

(a) L_i is positive, and

(b) the negative literal $\neg L$ occurs in the body of the clause $R_i: H \leftarrow B$ such that L_i unifies with L, and

(c) α is an mgu of L_i and L, and

(d) $\neg H\alpha$ is not an instance of any one of the L_j's, where $0 \leq j \leq i$

then L_{i+1} is $\neg H\alpha$.

3. If

(a) L_i is negative, and

(b) L_i unifies with a negative literal L occurring in the body of the clause $R_i: H \leftarrow B$, and

(c) α is an mgu of L_i and L, and

(d) θ is a computed answer for $D \cup \{G\}$, where G is the goal $\leftarrow$Bα

then L_{i+1} is H$\alpha\theta$.

4. If

(a) L_i is negative and has the form $\neg M$, and

(b) M unifies with a positive literal L occurring in the body of the clause $R_i: H \leftarrow B$, and

(c) α is an mgu of M and L, and

(d) $\neg H\alpha$ is not an instance of any one of the L_j's, where $0 \leq j \leq i$

then L_{i+1} is $\neg H\alpha$.

From this definition, it is possible to construct more than one path starting from the same literal. For simplicity, a path will sometimes be written without showing the clauses used to construct the path. In the path $L_0 \rightarrow L_1 \rightarrow \cdots \rightarrow L_n$, the *distance* of L_p from L_q $(0 \leq q \leq p \leq n)$ is $p-q$ and L_p is said to be *at a distance p-q* from L_q.

Definition : Let D be a database and L a literal such that if L is positive then it is ground. Let S be the set of all paths with L as the source. The *path space* rooted at the literal L is a tree defined as follows :

1. Each node of the tree is a literal.

2. The root node is L.

3. Let N be a node of the tree. Then the set of all the successors of N in the paths of S are the only descendants of N in the tree.

Each branch of the path space corresponds to a path in the database with the root node as the source and vice-versa. A path which ends at the head IC (No) of a constraint will be called a *success path*; otherwise, it will be called a *failure path*. A path space containing at least one branch which corresponds to a success path is referred to as a *success path space*.

5.2. The Method

To check integrity in a database when a new constraint is added, the database is queried directly to make sure that the constraint is a logical consequence of the completed database. If a constraint is deleted then this cannot cause any inconsistency. To check integrity in the updated database D' due to a transaction t applied to D, where a *transaction* is a finite sequence of

additions and deletions of clauses to/from a database, the source is taken as an update literal. An *update literal* of the transaction t applied to a database D may be one of the following :

1. A fact of t which is to be added to D.

2. The negation of a fact of t which is to be deleted from D.

3. If a rule $H \leftarrow B$ in t is to be added to D then for a computed answer θ for $D' \cup \{\leftarrow B\}$, the corresponding instance of the head of the rule $H \leftarrow B$, i.e. $H\theta$, which is implicitly added to D due to the transaction.

4. If a rule $H \leftarrow B$ in t is to be deleted from D then the negation of the head of the rule $H \leftarrow B$, i.e. $\neg H$, whose instances are likely to be deleted from D due to the transaction.

To preserve integrity one must ensure that a success path does not exist in the updated database with the source as an update literal. The different branches of the path space from an update literal can be generated in a number of ways and the backtracking mechanism is one of them.

It has been assumed that a transaction does not add and delete the same clause. Also, a transaction is rejected if a success path is found in the updated database. Modification can be considered as being accomplished by a deletion followed by an addition.

The method is illustrated in the following few examples.

Example 5.1 :

Consider the following database.

Database D 1 :

Rules :

$R1.\ Mother(x, y) \leftarrow Father(z, y) \wedge Husband(z, x)$

$R2.\ Parent(x, y) \leftarrow Father(x, y)$

$R3.\ Parent(x, y) \leftarrow Mother(x, y)$

$R4.\ Ancestor(x, y) \leftarrow Parent(x, y)$

$R5.\ Ancestor(x, y) \leftarrow Parent(x, z) \wedge Ancestor(z, y)$

R 6. $Wife(x, y) \leftarrow Husband(y, x)$

R 7. $Married(x, y) \leftarrow Husband(x, y)$

R 8. $Married(x, y) \leftarrow Wife(x, y)$

R 9. $Employed(x) \leftarrow Occupation(x, Service)$

R 10. $Student(x) \leftarrow Occupation(x, Student)$

R 11. $Dependent(x, y) \leftarrow Parent(y, x) \wedge Employed(y) \wedge Student(x)$

R 12. $Dependent(x, y) \leftarrow Married(y, x) \wedge Employed(y) \wedge \neg Employed(x)$

R 13. $Self(x) \leftarrow Married(y, x) \wedge \neg Employed(y)$

R 14. $Guardian(x, y) \leftarrow Dependent(y, x)$

Facts :

$Occupation(2, Service)$

$Occupation(3, Student)$

$Father(1, 3)$

Integrity Constraints I 1:

IC1. $Guardian(x, y) \rightarrow Sponsor(x, y)$

IC2. $Married(x, y) \rightarrow \neg Student(x)$

$I1_d$:

$IC(1) \leftarrow Guardian(x, y) \wedge \neg Sponsor(x, y)$

$IC(2) \leftarrow Married(x, y) \wedge Student(x)$

Transaction :

insertfact $Husband(1, 2)$

where *Student* and *Service* are constants.

Before the transaction is applied, database D 1 satisfies the constraints I 1. Let D 1' be the updated database after the insertion. The complete path space generated by the insertion is shown in figure 5.1.

At the root of the path space is the update literal $Husband(1, 2)$. If one attempts to unify $Husband(1, 2)$ with a literal on the right hand side of any of the clauses in D 1', the first potential unification is with the literal $Husband(z, x)$ occurring in the body of R 1 for which

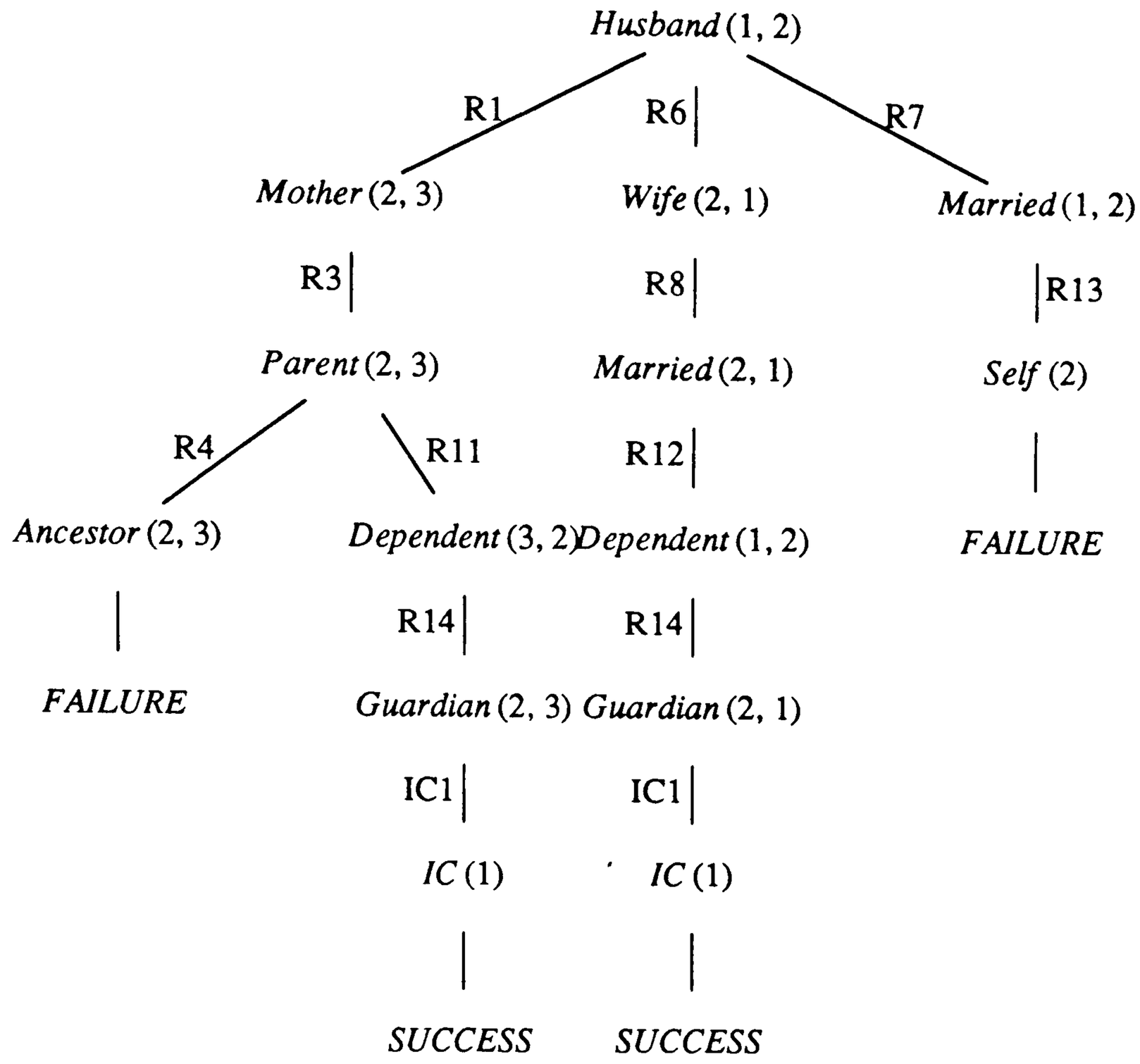

Figure 5.1. The complete path space in the case of example 5.1

$\alpha=\{z/1, x/2\}$ is an mgu. The resolvent of the goal

$$\leftarrow Father(z, y) \wedge Husband(z, x)$$

with

$$Husband(1, 2) \leftarrow$$

is

$$\leftarrow Father(1, y).$$

Now if $\leftarrow Father(1, y)$ is treated as a goal and applied to the database, one will arrive at $\theta=\{y/3\}$ as a computed answer for $D1' \cup I1_d \cup \{\leftarrow Father(1, y)\}$. Thus the next literal of the path is the head of rule $R1$ with the substitution $\{z/1, x/2, y/3\}$ applied, i.e. $Mother(x, y)\alpha\theta$, or $Mother(2, 3)$. In a similar way one can derive the literal $Parent(2, 3)$ as a successor to

Mother (2, 3) in the path with the help of R3. The literal *Parent* (2, 3) unifies with the literal of the body of R4 as well as with a literal of the body of R5 and a literal of the body of *R* 11. If one considers the first case, *Ancestor* (2, 3) will be the next literal of the path. Although it unifies with a literal from the body of R5, there is no explicit or implicit fact under the predicate *Parent* which is present in $D\,1' \cup I\,1_d$ whose second argument is 2. Hence, this path is a failure path. In the second case, when *Parent* (2, 3) unifies with the first literal from the body of R5, 3 is not an ancestor of anyone in $D\,1' \cup I\,1_d$ which will satisfy the unified second literal and hence this path also fails. In the third case, when *Parent* (2, 3) unifies with the literal *Parent* (*y* , *x*) occurring in the body of *R* 11, $\alpha=\{x/3, y/2\}$ would be an mgu of these two literals. The resolvent of the goal

$$\leftarrow Parent(y, x) \wedge Employed(y) \wedge Student(x)$$

and

$$Parent(2, 3) \leftarrow$$

is

$$\leftarrow Employed(2) \wedge Student(3)$$

and there exists a computed answer $\theta=()$ (the identity substitution) for $D\,1' \cup I\,1_d \cup \{\leftarrow Employed(2) \wedge Student(3)\}$. Thus the next literal of the path is *Dependent* (*x* , *y*)$\alpha\theta$, i.e. *Dependent* (3, 2). Using clauses *R* 14 and *IC* 1, one can construct the following success path

$$Husband(1, 2) \xrightarrow{R1} Mother(2, 3) \xrightarrow{R3} Parent(2, 3) \xrightarrow{R11} Dependent(3, 2) \xrightarrow{R14} Guardian(2, 3) \xrightarrow{IC1} IC(1)$$

and hence conclude that the integrity is violated due to the transaction.

Returning to the top-most level, *Husband* (1, 2) may also unify with *Husband* (*y* , *x*) from the body of R6 as well as *Husband* (*x* , *y*) from the body of R7. In the first case one can construct the following success path

$$Husband(1, 2) \xrightarrow{R6} Wife(2, 1) \xrightarrow{R8} married(2, 1) \xrightarrow{R12} Dependent(1, 2) \xrightarrow{R14} Guardian(2, 1) \xrightarrow{IC1} IC(1)$$

In the second case *Married* (1, 2) may unify with

(a) *Married* (*y* , *x*) on the right hand side of rule *R* 12. This reduces the right hand side to

$$\leftarrow \textit{Employed}(1) \wedge \neg \textit{Employed}(2).$$

If this is applied as a goal to the database $D\,1'$ it will fail and hence this rule is ignored.

(b) $\textit{Married}(y, x)$ on the right hand side of rule $R\,13$ and $\alpha=\{y/1, x/2\}$ is an mgu. This reduces the right hand side to

$$\leftarrow \neg \textit{Employed}(1).$$

If this is applied as a goal to $D\,1'$ it will succeed and the computed answer is the identity substitution. Thus the next literal of the path is $\textit{Self}(x)\alpha$ or $\textit{Self}(2)$. This path is a failure path as the literal $\textit{Self}(2)$ or its negation does not unify with any body literal of the clauses of $D\,1' \cup I\,1_d$.

(c) $\textit{Married}(x, y)$ on the right hand side of $IC\,2$. This reduces the right hand side to

$$\leftarrow \textit{Student}(1).$$

If this is applied as a goal to $D\,1'$ it will fail and hence this constraint is ignored.

Example 5.2

Consider the following database.

Database $D\,2$:

Rules :

same as $D\,1$

Facts :

$\textit{Occupation}(1, \textit{Service})$

$\textit{Occupation}(2, \textit{Service})$

$\textit{Husband}(1, 2)$

Integrity Constraints $I\,2$:

same as $D\,1$

Transaction :

deletefact *Occupation* (2, *Service*).

Before the transaction is applied, database *D* 2 satisfies the constraints *I* 2. Let *D* 2' be the updated database after the deletion. Figure 5.2 shows the complete path space arising from the deletion.

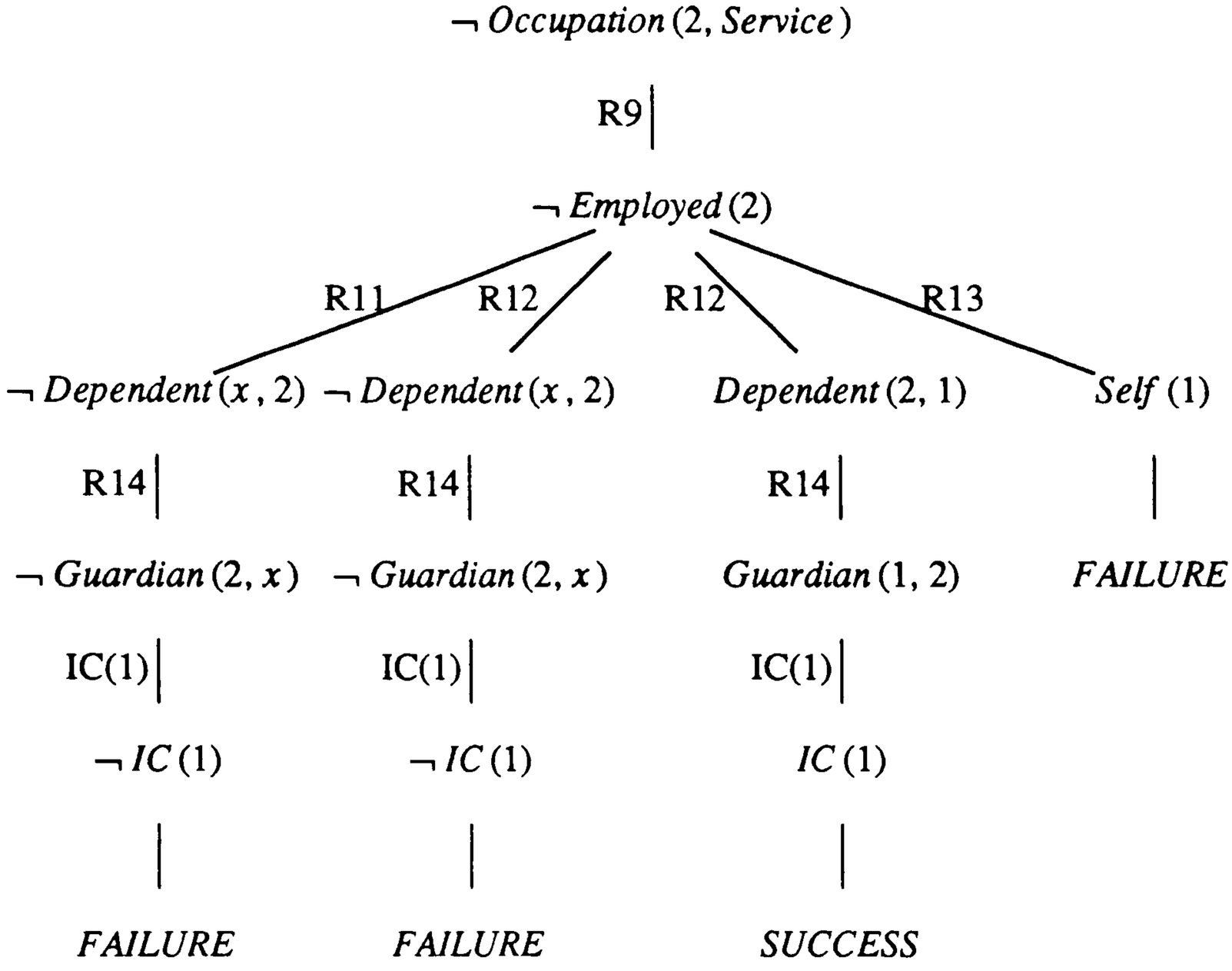

Figure 5.2. The complete path space in the case of example 5.2

As the transaction is the deletion of a fact, the only update literal is the negation of this fact and this is taken as a candidate source literal. Clause R9 shows that an implicit deletion of the fact *Employed* (2) may occur due to this transaction. However, to avoid reasoning with two database states, the facts deleted from the database due to this transaction are not determined yet. Instead, it can be said that the fact *Employed* (2) is likely to be deleted from the database. $\theta=\{x/2\}$ is an mgu of *Occupation* (2, *Service*) and the body literal *Occupation* (*x* , *Service*) of R9. Thus the next literal of the path is the negation of *Employed* (*x*)θ, i.e. ¬ *Employed* (2). The literal *Employed* (2) unifies with the literal *Employed* (*y*) occurring in the body of both *R* 11 and *R* 12 and the negation of *Employed* (2) unifies with a negative literal ¬ *Employed* (*x*) occurring in the body of *R* 12 as well as with ¬ *Employed* (2) in the body of *R* 13. The first and second unification

may cause further implicit deletions from the database and $\alpha=\{y/2\}$ is an mgu for both unifications. The third and fourth unifications may cause implicit additions to the database and an mgu in this case is $\beta=\{x/2\}$. If one considers the first case, the two consecutive literals in this direction of the path will be $\neg\, Dependent(x, 2)$ and $\neg\, Guardian(2, x)$ and it becomes a failure path as the last literal of this path does not unify with any body literal of any of the clauses in $D2' \cup I2_d$. The second case also generates a failure path.

Now, if one considers the third case, the next two consecutive literals of the path will be *Dependent* (2, 1) and *Guardian* (1, 2) and these facts are implicitly added to the database due to the transaction. The clause *IC* 1 unifies with the last literal of the path yielding a success path

$$\neg\, Occupation(2, Service) \xrightarrow{R9} \neg\, Employed(2) \xrightarrow{R12} Dependent(2, 1) \xrightarrow{R14} Guardian(1, 2) \xrightarrow{IC1} IC(1)$$

thereby showing that integrity has been violated. Considering the last case, *Self* (1) would be the successor of $\neg$ *Employed* (2) in another direction. This path eventually fails.

Example 5.3

Consider the following database.

Database *D* 3 :

Rules :

R 1–*R* 10, *R* 12–*R* 14 of *D* 1

Facts :

Occupation (1, *Service*)

Occupation (2, *Service*)

Occupation (3, *Student*)

Husband (1, 2)

Father (1, 3)

Integrity Constraints *I* 3 :

same as *I* 1

Transaction :

insertrule R : $Dependent(x, y) \leftarrow Parent(y, x) \wedge Employed(y) \wedge Student(x)$

Before the transaction is applied, database $D3$ satisfies the constraints $I3$. Let $D3'$ be the updated database. To determine the set of instances of the conclusion of R which are likely to be added to the database due to the addition of the rule R, the following set of computed answers must be derived.

$\{\theta: \theta$ is a computed answer for $D3' \cup \{\leftarrow B\}$, where R is $H \leftarrow B\}$

which is equal to

$$\{\{x/3, y/2\}, \{x/3, y/1\}\}.$$

The corresponding set of instances of the conclusion of R (equivalent to $H\theta$), i.e. the set of update literals, is

$$\{Dependent(3, 2), Dependent(3, 1)\}$$

and the facts in this set are added to the database due to the addition of the rule R to the database. Taking one of the update literals $Dependent(3, 2)$ as a source, the success path

$$Dependent(3, 2) \xrightarrow{R14} Guardian(2, 3) \xrightarrow{IC1} IC(1)$$

in $D3' \cup I3_d$ can be constructed. Thus the integrity in the updated database $D3'$ is violated due to the addition of the rule R to the database.

Example 5.4

Consider the following database.

Database $D4$:

Rules :

same as $D1$

Facts :

Employed (1)

Occupation (1, *Service*)

Occupation (2, *Service*)

Husband (1, 2)

Integrity Constraints $I4$:

same as $I1$

Transaction :

deleterule R : *Employed* $(x) \leftarrow$ *Occupation* $(x,$ *Service* $)$

Before the transaction is applied, database $D4$ satisfies the constraints $I4$. Let $D4'$ be the updated database. The set of ground instances of the conclusion of R which are likely to be deleted from the database due to the deletion of R is not determined. Instead, the negation of the head of R, which represents the set of facts deleted from the database, is taken as a source. From this the following success path can be constructed in the updated database as the fact *Employed* (2) is implicitly deleted from $D4$ due to the deletion of R.

$$\neg\, \textit{Employed}(x) \overset{R12}{\rightarrow} \textit{Dependent}(2, 1) \overset{R14}{\rightarrow} \textit{Guardian}(1, 2) \overset{IC1}{\rightarrow} \textit{IC}(1).$$

Thus the constraints of the database are not satisfied by the deletion of rule R from the database.

5.3. Theorems for Integrity Checking

In this section, the definition of stratified databases [2, 65] is given and the correctness of the path finding method is proved for this class of databases.

Definition : A *level mapping* of a database is a mapping from its set of predicates to the set of non-negative integers. A database is *stratified* if it has a level mapping such that, for every database clause $H \leftarrow B$, the level of the predicate of every positive condition in B is less than or equal to the level of the predicate of H and the level of the predicate of every negated condition in B is less than the level of the predicate of H.

It has been shown in [2] that if a database D is stratified then the completion of D is consistent and hence will have a model. Based on this fact, the database versions of the following two results [65], which are the basis of the method for determining whether a database satisfies or violates a constraint $\leftarrow B$, are given.

Result 5.1 : *Let D be a stratified database, $\leftarrow B$ a constraint. If there exists an SLDNF-refutation of $D \cup \{\leftarrow B\}$, then D violates $\leftarrow B$.*

Result 5.2 : *Let D be a stratified database, $\leftarrow B$ a constraint. If $D \cup \{\leftarrow B\}$ has a finitely failed SLDNF-tree, then D satisfies $\leftarrow B$.*

The proof of the theorem for integrity checking by the path finding method is similar to the one which has been described in [65]. Let D and D' be stratified databases and D' is obtained from D by applying a transaction t to D. Let the transaction t consist of a sequence of deletions followed by a sequence of additions. Suppose that the application of the sequence of deletions to D produces the intermediate database D''. In [65] , the two sets of partially instantiated atoms $Pos_{D'',D'}$ and $Neg_{D'',D'}$ have been calculated inductively by the following formulae (similar formulae for $Pos_{D'',D}$ and $Neg_{D'',D}$).

$$Pos^{0}_{D'',D'} = \{A : A \leftarrow W \in D'/D''\}$$

$$Neg^{0}_{D'',D'} = \{\}$$

$Pos^{n+1}_{D'',D'} = \{A\theta : A \leftarrow W \in D', B$ is a positive condition of $A \leftarrow W$, $C \in Pos^{n}_{D'',D'}$, and θ is an mgu of B and $C\}$

$\cup \{A\theta : A \leftarrow W \in D', B$ is a negative condition of $A \leftarrow W$, $C \in Neg^{n}_{D'',D'}$, and θ is an mgu of B and $C\}$

$Neg^{n+1}_{D'',D'} = \{A\theta : A \leftarrow W \in D', B$ is a positive condition of $A \leftarrow W$, $C \in Neg^{n}_{D'',D'}$, and θ is an mgu of B and $C\}$

$\cup \{A\theta : A \leftarrow W \in D', B$ is a negative condition of $A \leftarrow W$, $C \in Pos^{n}_{D'',D'}$, and θ is an mgu of B and $C\}$

$$Pos_{D'',D'} = \bigcup_{n \geq 0} Pos^{n}_{D'',D'}$$

$$Neg_{D'',D'} = \bigcup_{n \geq 0} Neg^{n}_{D'',D'}$$

The above two sets $Pos_{D'',D'}$ and $Neg_{D'',D'}$ (resp. $Pos_{D'',D}$ and $Neg_{D'',D}$) can be used to capture the differences between the models of comp(D'') and *comp* (D') (resp. comp(D'') and comp(D)). Suppose,

$$Pos_{D,D'} = Pos_{D'',D'} \cup Neg_{D'',D}$$

$$Neg_{D,D'} = Neg_{D'',D'} \cup Pos_{D'',D}$$

Then the above two sets $Pos_{D,D'}$ and $Neg_{D,D'}$ can be used to capture the differences between a model for comp(D) and a model for *comp* (*D'*). Accordingly one can have the following lemma similar to the one described in [65].

Lemma 5.1 : *Let D , D' be stratified databases. Let t be a transaction which when applied to D produces D' and D" as described above. Let* $\leftarrow B$ *be a constraint and* $\Theta = \{\theta: \theta$ *is an mgu of an atom of* $Pos_{D,D'}$ ($Neg_{D,D'}$) *and a positive (complement of a negative) literal occurring in B}. Then D' satisfies* $\leftarrow B$ *if and only if D' satisfies* $\leftarrow B\ \theta$ *for all* $\theta \in \Theta$.

Proof : Result (a) of the simplification theorem given in [65].

Suppose the set $Add_{D,D'}$ (resp. $Del_{D,D'}$) is the set of all positive (resp. complement of negative) literals occurring in all the path spaces generated by the update literals of t. When all the path spaces are generated successfully without going into an infinite loop, one can have the following lemma.

Lemma 5.2 : *Let D be a stratified database, and D' and D" be obtained from D by applying t to D as described above. Then*

(a) *For each* $A \in Add_{D,D'}$ *there exists an* $A' \in Pos_{D,D'}$ *such that A is an instance of A'.*

(b) *If there is a ground instance A of an atom* $A' \in Del_{D,D'}$ *such that comp(D)* $\vdash A$ *but not comp (D')* $\vdash A$ *, then A is an instance of an atom* $A'' \in Neg_{D,D'}$.

(c) *If A is a ground instance of an atom* $A' \in Pos_{D,D'}$ *such that comp (D')* $\vdash A$ *but not comp(D)* $\vdash A$, *then* $A \in Add_{D,D'}$.

(d) *If there is a ground instance A of an atom* $A' \in Neg_{D,D'}$ *such that comp(D)* $\vdash A$ *but not comp (D')* $\vdash A$, *then A is an instance of an atom* $A'' \in Del_{D,D'}$.

Proof : Straightforward from the definition of path.

The above result shows that in the case of the path finding method, the set of ground facts $Add_{D,D'}$ is the portion added to a model of comp(D) and the set of partially instantiated atoms $Del_{D,D'}$ represents the portion deleted from a model of comp(D). The following theorem and its corollary which are based on the above lemma help to reduce the checking of a constraint into none or some of the simplified forms of the constraint.

Theorem 5.1 : *Let D be a stratified database, $\leftarrow B$ be a constraint and D' be obtained from D by applying t to D. Let $\Theta = \{\theta$: θ is an mgu of an atom of $Pos_{D,D'}$ ($Neg_{D,D'}$) and a positive (complement of a negative) literal occurring in $B\}$. Let $\Phi = \{\phi$: ϕ is an mgu of an atom of $Add_{D,D'}$ ($Del_{D,D'}$) and a positive (complement of a negative) literal occurring in $B\}$. Then D' satisfies $\leftarrow B\ \theta$ for all $\theta \in \Theta$ if and only if D' satisfies $\leftarrow B\ \phi$ for all $\phi \in \Phi$.*

Proof : The following is the proof of the 'if' part of the theorem. The proof of the 'only if' part is similar.

Suppose D' satisfies $\leftarrow B\ \theta$ for all $\theta \in \Theta$. If possible, let there exist a $\gamma \in \Phi$ such that D' violates $\leftarrow B\ \gamma$. Suppose L is a positive (negative) literal occurring in B and it (its complement) unifies with an atom A of the set $Add_{D,D'}$ ($Del_{D,D'}$) with the mgu γ. By lemma 5.2(a) (lemma 5.2(b)), there exists an atom L' from $Pos_{D,D'}$ ($Neg_{D,D'}$) such that $L\gamma$ is an instance of L' (negation of L'). Since D' satisfies $\leftarrow B\ \theta$ for all $\theta \in \Theta$, it follows that D' must satisfy $\leftarrow B\ \beta$, where β is an mgu of L' and L. $L\gamma$ is an instance of both L and L' and hence $\leftarrow B\ \gamma$ is an instance of $\leftarrow B\ \beta$. Therefore D' cannot satisfy $\leftarrow B\ \gamma$. This contradicts the initial assumption. Hence D' satisfies ϕ for all $\phi \in \Phi$.

Corollary 5.1: *Let D be a stratified database, $\leftarrow B$ be a constraint and D' be obtained from D by applying t to D. Let $\Phi = \{\phi$: ϕ is an mgu of an atom of $Add_{D,D'}$ ($Del_{D,D'}$) and a positive (complement of a negative) literal occurring in $B\}$. Then D' satisfies $\leftarrow B$ if and only if D' satisfies $\leftarrow B\ \phi$ for all $\phi \in \Phi$.*

Proof : Follows from the above theorem and lemma 5.1.

Theorem 5.2 : *(Theorem for integrity checking by the path finding method) Let D be a stratified database and I be a set of constraints such that D satisfies I. Suppose D' is obtained from D by the application of transaction t to D. Then the following properties hold.*

(a) *If there exists a success path in $D' \cup I_d$ taking the source as one of the update literals, then D' violates I.*

(b) *If there exists no success path in $D' \cup I_d$ from any one of the update literals, then D' satisfies I.*

Proof :

(a) Follows from corollary 5.1 and result 5.2.

(b) Follows from corollary 5.1 and result 5.1.

Corollary 5.2 : *Let D be a hierarchical database [16] and I be a set of constraints such that D satisfies I. Suppose D' is obtained from D with the application of transaction t to D. Then the following properties hold.*

(a) *There exists a success path in $D' \cup I_d$ taking the source as one of the update literals if and only if D' violates I.*

(b) *There exists no success path in $D' \cup I_d$ from any one of the update literals if and only if D' satisfies I.*

Proof : It follows from the above theorem directly as the hierarchical nature of D ensures the termination of any query without going into an infinite loop.

5.4. Prolog Implementation

Facts and rules of a database are stored directly as Prolog facts and rules. A unique number has been assigned to each newly inserted constraint as an identification. A constraint $\leftarrow D$ numbered No is stored directly as a Prolog rule $ic(No) :- D$. A rule or a constraint can be retrieved efficiently by using a reference which uniquely identifies the corresponding rule or constraint.

The path can be computed efficiently by maintaining additional information for each rule and constraint in the database. For each literal L occurring in the body of a rule $H \leftarrow B$, clauses of the form

$$depend(Ref, H, L)$$

$$depend(Ref, not\ H, CompL)$$

have been maintained, where Ref is the reference which uniquely identifies the clause $H \leftarrow B$ and $CompL$, called the complement of L, is defined as follows. When L is positive, $CompL$ is $not\ L$; otherwise, when L is $not\ A$, then $CompL$ is A. For each literal L occurring in a constraint $\leftarrow D$, a clause of the form

$$depend(Ref, ic(No), L).$$

has been maintained, where *ic* (*No*) is the head of the constraint and *Ref* is the reference which uniquely identifies the clause *ic* (*No*)←*D*.

The task of finding a path from a source literal *S* to a destination literal *D* can be achieved in Prolog with the help of the following *meta-interpreter* [94, 95]:

```
path(D,S,_):-
    depend(Ref,D,S),
    clause(D,B,Ref),
    simplify(B,S,SB),
    call(SB).

path(D,S,NegList):-
    depend(_,Via,S),
    Via = (not A),
    not instance(A,NegList),
    path(D,Via,[A|NegList]).

path(D,S,NegList):-
    depend(Ref,Via,S),
    not Via = (not _),
    clause(Via,B,Ref),
    simplify(B,S,SB),
    call(SB),
    path(D,Via,NegList).
```

path (*D* , *S* , *NegList*) means that a path is constructed from the literal *S* to the literal *D* and the set comprising the negation of each literal of the list *NegList* are the only negative literals occurring in this path. *clause* (*H* , *B* , *Ref*) means *H* and *B* are unified respectively with the head and body of a clause which is uniquely identified by the reference *Ref* . The interpretation of *simplify* (*B* , *S* , *SB*) is that if *S* is a positive ground literal then *SB* is obtained by resolving *B* against *S* ; otherwise, *SB* is an instance of *B* obtained by unifying one of *B* 's body literals with *S* . *instance* (*A* , *NegList*) means that A is not an instance of any of the atoms occurring in the list *NegList* . This corresponds to conditions 2(d) and 4(d) in the definition of a path and prevents a possible generation of an infinite path from a negative literal in the presence of recursive rules.

The top level C-Prolog goal for integrity checking (?-*ic_violated*(*Tran*).) can be defined as

```
ic_violated(Tran):-
    path(ic(No),Tran,[]),!.
```

where *Tran* represents an update literal. With the above convention if the database becomes inconsistent due to the transaction *Tran*, then the first inconsistency is reported at a constraint numbered *No*, i.e. this constraint is not a logical consequence of the completion of the updated database.

The efficiency of the meta-interpreter can be improved (at the expense of storage) by storing the two *depend* clauses

$$depend(H, L, B'),$$
$$depend(not\ H, CompL, \_)$$

for each literal L occurring in the body of a clause $H \leftarrow B$, where B' is obtained from B by removing the occurrence of L and $CompL$ is the complement of L. It is obvious that in the new definition of *path*, one can eliminate the calls to the predicates *clause* and *simplify*.

5.5. Optimisations

Some optimisations could be carried out at the time of specifying certain specific kinds of rules or constraints. These optimisations have been illustrated in the context of the alternative implementation. Three cases have been considered. The first two deal with the generation of *depend* clauses of the first type and the third with the generation of the second kind of *depend* clauses in the implementation.

(1) Functional dependencies are a common form of constraint. An example of this kind of constraint is

$$Father(y, x) \wedge Father(z, x) \rightarrow y=z$$

Suppose that a constraint of this kind is present in the database and in the course of a transaction a fact under the predicate *Father* is inserted into the database, either explicitly or implicitly. According to the method and its implementation described in previous sections, it is necessary to evaluate the two simplified instances of the above constraint and they are obtained from the

following two clauses under the predicate *depend*:

$$depend(ic(Id), father(Y, X), (father(Z, X) \rightarrow Y{=}Z))$$
$$depend(ic(Id), father(Z, X), (father(Y, X) \rightarrow Y{=}Z))$$

where Id is a unique identification of the constraint. Evaluation of the simplified instance which arises from the second of the above two is redundant and hence may be excluded. This can be prevented by not generating the second one at the time of specifying the constraint.

(2) The second case deals with rules in which two or more similar instances of a predicate occur in the body. An example of this kind of rule is

$$Result(x, Hons) \leftarrow Exam(x, y, A) \wedge Exam(x, z, A) \wedge y \neq z$$

If a database contains a rule of this form, the alternative implementation of the method described in this section generates the following two clauses under the predicate *depend*:

$$depend(result(X, hons), exam(X, Y, a), (exam(X, Z, a), Y \neq Z))$$
$$depend(result(X, hons), exam(X, Z, a), (exam(X, Y, a), Y \neq Z)).$$

Now, if a transaction inserts into the database either implicitly or explicitly a fact under the predicate *Exam* unifying with $exam(X, Y, a)$, then an instance of $result(X, hons)$ might be added twice implicitly to the database. Consequently, the same path space rooted at the corresponding instance of $result(X, hons)$ would have to be generated twice. This can be avoided by generating only one of the above two clauses under *depend* at the time of compiling the rule.

The above two cases can be formalised as follows. Let D be a database and I be a set of constraints. Suppose there exist literals $L_1, L_2, ..., L_n$ under the same predicate occurring in the body of a clause R in $D \cup I_d$ such that any two of them are instances of each other. Suppose for any two L_i and L_j, R' is obtained from R by interchanging the variables in the same argument position of L_i and L_j. If for all L_i and L_j, R' can be transformed to R with the help of the transformations $M, N \leftrightarrow N, M$, $a{=}b \leftrightarrow b{=}a$ and $a \neq b \leftrightarrow b \neq a$, where M, N are literals and a, b are terms, then generate only one *depend* clause corresponding to one of the L_k's, $1 \leq k \leq n$.

(3) It can be seen from figure 5.2 corresponding to example 5.2 that the first and second branch of the path space are identical to each other. Generating one of them is redundant. This happened because the implicit or explicit deletion of a fact under the predicate *Employed* can delete the

same fact from both rules 11 and 12. Since the method does not compute what facts have been deleted, it will be sufficient if one generates the second kind of *depend* clause corresponding to either rule 11 or rule 12.

The above can be formalised as follows. Suppose there are two depend clauses of the form *depend*(_, *not* H, L) and *depend*(_, *not* H', L') such that L' is an instance of L and H' is an instance of H. If $H'\theta$ is an instance of $H\theta$ and $L'\alpha$ is an instance of L, where θ is an mgu of L and L', and α is an mgu of H and H', then the latter depend clause can be excluded.

CHAPTER 6

A Comparative Evaluation

It is desirable to check the integrity of a database upon every transaction which alters its state. The simplest approach to checking integrity in a database involves the evaluation of each constraint whenever the database is updated. However, such an approach is too costly, especially for large databases, and does not make use of the fact that the database satisfies the constraints prior to the update. Different ways of maintaining integrity in definite databases which have been proposed include generalisations of the simplification method [79] by Lloyd and Topor [66] , Decker [25] , and Bry et al. [10] , a general theorem-proving technique by Sadri and Kowalski [91] , using the Prolog not-predicate by Ling [60] , consistency proof method and a modified program method by Asirelli et al. [3] and the path finding method. The objective of this chapter is to evaluate the merits and demerits of these methods. As the proposed methods are concerned only with static non-aggregate constraints and the underlying database is definite, the comparative evaluation is made in such environments only. Hence, in this chapter, any further reference to the term 'database' will refer to a 'definite database' and any reference to term 'constraint' will refer to a 'static non-aggregate constraint'.

The arrangement of this chapter is as follows. Section 6.1 considers the different forms of constraint representation used by different methods and their expressiveness. Section 6.2 introduces several methods in brief and illustrates each with a single database (database of Example 5.1) to see their behaviour from the same view point. In section 6.3 the methods are compared against each other and different aspects of the comparison are verified by practical tests which have been carried out in a C-Prolog [17] environment running on a High Level Hardware Orion computer.

6.1. Constraint Representation

The format used for expressing constraints varies according to the method studied.
Formats used include closed typed first order formulae, closed first order formulae with restricted quantification, range form, IC-formulae and denials. Each of the expressions is described in turn.

In [66] , Lloyd et al. are concerned with a typed database [87, 66] and the format which they have adopted for representing a constraint is an arbitrary closed typed first order formula. The requirement that the database is typed ensures that important kinds of integrity constraints are maintained automatically. A formula F is said to be *domain-independent* [103] if and only if its truth value does not depend on any domain element other than those occurring in the relations that are explicitly mentioned in F. A formula is *domain-dependent* if and only if it is not domain-independent. A domain-dependent constraint, such as $\forall x/\tau\, p(x)$, where x is a variable of type τ, can be checked by replacing the formula with $\forall x\,(p(x) \leftarrow \tau(x))$ as a constraint. This means that whenever a new constant a is entered into the domain of type τ, a fact $\tau(a)$ is maintained in the database and according to the constraint a fact $p(a)$ will be inserted into the database. Constraints are converted into a type free formula and evaluated in a type free database.
Consider the constraint

$$\forall x\,(Worker(x) \rightarrow \exists y\ Superior(y, x)) \qquad (6.1)$$

The typed form of this constraint might be expressed as follows :

$$\forall x/Name\,(Worker(x) \rightarrow \exists y/Name\ Superior(y, x))$$

where *Name* is a unary type predicate. The corresponding type free form is given by

$$\forall x\,(Name(x) \rightarrow (Worker(x) \rightarrow \exists y\,(Name(y) \wedge Superior(y, x))))$$

In [10] , Bry et al. have defined constraints as function-free, closed first order formulae with restricted quantification which are expressed in normalised form [10]. In other words a quantified formula or sub-formula has one of the forms

$$\exists x_1 \cdots \exists x_n [A_1 \wedge \cdots \wedge A_m \wedge Q]$$
$$\forall x_1 \cdots \forall x_n [\neg A_1 \vee \cdots \vee \neg A_m \vee Q]$$

where A_1, ..., A_m are atoms such that every variable x_i occurs in at least one A_j, and Q is either true or false, or some formula in which some or all x_i are free. In this notation, the constraint (6.1) can be written as

$$\forall x\,[\neg\, Worker\,(x) \vee \exists y\; Employee\,(y\,,\,x)]$$

where A_1 is $Worker\,(x)$ and Q is $\exists y\; Employee\,(y\,,\,x)$. The variable x is free in Q.

Decker [25] has considered a subset of domain-independent formulae, called *range-restricted* formulae, for expressing constraints. A formula F in *disjunctive minimised form* (*dmf*) is *range-restricted* with respect to a variable x if and only if the variable x is existentially quantified in the formula, and there is at least one non-negated x-literal in every conjunction in F with an occurrence of x; or if the formula is universally quantified wrt the variable x, and *dmf*(*Not* F) is range-restricted with respect to x. F is *range-restricted* if and only if it is range restricted with respect to all the quantified variables occurring in F. This class of range-restricted formulae is then represented in their equivalent *range form*. A range form has a range expression, conclusion and remainder. The range form of (6.1) is given by

$$\forall x\,(Worker\,(x) \rightarrow Superior\,(y\,,\,x)),$$

where $Worker\,(x)$ is the range expression for the variable x and $Superior\,(y\,,\,x)$ is the conclusion. The remainder is empty in this case as there is no other literal in the constraint.

The constraint checking method put forward by Kowalski et al. [91, 55] deals with constraints in denial form. Denials must also be range-restricted. The method does not deal with constraints like $\forall x\,(p\,(x))$ since it is not possible to convert this kind of constraint to a denial form which is range-restricted. It is also possible to deal with constraints that are in a more general form than denials as described in section 4.1.4. The path finding method and the method of Kowalski et al. both deal with the same class of constraint.

Ling [60] introduces a special form of closed first order formula, called an *IC-formula*, which uses normal form (NF) negative formulae, to express constraints. An NF negative formula $Not\,(P)$ has the form

$$Not\,(A_1 \wedge \cdots \wedge A_m \wedge Not\,(B_1) \wedge \cdots \wedge Not\,(B_n)),$$

where each A_i is either a positive literal or an evaluable predicate and each Not(B_j) is an NF negative formula. Using this, a constraint formula or IC-formula has been defined by Ling to be a closed formula of the following form :

$$A_1 \wedge \cdots \wedge A_n \wedge Not\,(B_1) \wedge \cdots \wedge Not\,(B_m) \rightarrow C_1 \vee \cdots \vee C_p, \qquad n\,,\,m\,,\,p \geq 0$$

where each C_i has the same form as B_j. The constraint (6.1) is thus already in the form of an IC-formula, where A_1 is *Worker*(x), m is zero, and C_1 is *Superior*(y, x)

Asirelli et al.'s modified program method deals with constraints of the form

$$p(X_1, ..., X_n) \rightarrow \Psi(X_1, ..., X_n)$$

such that the left hand side has only distinct variables. A formula with more complex terms on the left hand side can also be transformed to the above form and has been discussed by the authors in their paper [3]. This form of formula has the problem of representing constraints which involve negative conditions and is therefore more limited than the forms which can be handled by the other methods. The constraint (6.1) can be written in the above form as

$$Worker(x) \rightarrow \psi(x)$$

by adding to the database the following definition of ψ

$$\psi(x) \leftarrow Superior(y, x)$$

6.2. Constraint Checking Methods

In this section a number of methods for checking integrity in databases are described briefly and the operation of each of the methods is expressed with the help of the following example.

Example 6.1

Database D 1 :

Relations :

same as Example 5.1

Rules :

same as Example 5.1

Facts :

F 1. *Occupation*(2, *Service*)

F 2. *Occupation*(3, *Student*)

F 3. *Husband* (1, 2)

F 4. *Father* (1, 3)

F 5. *Sponsor* (2, 3)

F 6. *Sponsor* (2, 1)

Integrity Constraints :

same as Example 5.1

Transaction :

insertfact *Occupation* (1, *Service*).

where 1, 2, 3, *Student* and *Service* are constants.

The integrity checking methods proposed thus far can be classified into the following categories on the basis of their internal mechanisms:

1. Each constraint is evaluated whenever the database is updated (this method will be referred to as *Simple*).

2. The constraint set is simplified using induced updates (the set of facts which are implicitly or explicitly added to or deleted from a database due to a transaction to the database) and only the relevant constraints are evaluated. This idea of simplification of constraints was introduced by Nicolas [79] for checking integrity in relational databases. The methods in [66, 25, 10] and the path finding method are based on this idea of simplification of constraints, but they differ in the way these induced updates are computed and the constraints are evaluated.

3. A generalised proof procedure is used to evaluate the constraints at the end of a transaction. The method described in [55] is based on a proof procedure which generalises the concept of SLDNF-resolution by reasoning forward from an update. The proof procedure of [92] is used to evaluate constraints in the context of deductive databases extended with default rules.

4. Derivation rules are modified to produce a modified database. This technique has been followed in [3, 40]

5. Constraints are evaluated only against the fact base by transforming each constraint into a form which is defined over the base facts only. This idea was first introduced in [14] in the context of relational databases and later in [60] in the context of deductive databases.

6.2.1. Method proposed by Lloyd et al.

The simplification method proposed by Lloyd et al. in [66] is based on a standard view of constraint satisfiability. The correctness of the method was first proved for the class of definite databases in [66] and later for the class of stratified databases [2] in [65]. An important task of the simplification method is to capture the difference between a model for $comp(D')$ and a model for $comp(D)$, where D and D' are databases and D' is obtained from D by the application of a transaction T to D. By using only rules of the database and the transaction, the method computes in stages the two sets of partially instantiated atoms $pos_{D,D'}$ and $neg_{D,D'}$. These two sets of atoms represent respectively the part that is added to the model for $comp(D)$ when passing from D to D' due to a transaction and the part that is deleted. To preserve the consistency of the updated database, constraints are instantiated appropriately with the two sets of atoms $pos_{D,D'}$ and $neg_{D,D'}$, and only affected constraints are evaluated. Instantiated constraints are evaluated using the SLDNF proof procedure.

To apply the algorithm to example 6.1, the following two sets of atoms $pos_{D,D'}$ and $neg_{D,D'}$ are determined.

$pos_{D,D'} = \{ Occupation(1, Service), Employed(1), Dependent(x, 1), Guardian(1, x) \}$

$neg_{D,D'} = \{ Self(x), Dependent(1, y) \}$.

The formulae for deriving the above two sets of atoms have been defined in section 5.3. The following shows the different stages of determining the sets $pos_{D,D'}$ and $neg_{D,D'}$.

$pos^0_{D,D'} = \{Occupation(1, Service)\}$

$neg^0_{D,D'} = \{\}$

$pos^1_{D,D'} = \{Employed(1)\}$

$neg^1_{D,D'} = \{\}$

$pos^2_{D,D'} = \{Dependent(x, 1), Dependent(x, 1)\}$

$neg^{2}_{D',D'}=\{Dependent(1,y), Self(x)\}$

$pos^{3}_{D',D'}=\{Guardian(1,x)\}$

$neg^{3}_{D',D'}=\{Guardian(y,1)\}$

$pos^{4}_{D',D'}=\{\}$

$neg^{4}_{D',D'}=\{\}$

At this point, no more new atoms can be generated. In $pos^{2}_{D',D'}$, the second atom $Dependent(x,1)$ is an instance of the first one and can be excluded while computing the atoms of the next stage. The number of stages for calculating the two sets of atoms is always finite, even in the presence of recursive rules, as at any stage a generated atom is excluded if it is an instance of an atom which is already generated.

Using the two sets of atoms $pos_{D',D'}$ and $neg_{D',D'}$, constraints are instantiated appropriately. The only constraint affected is IC1 and an instance of this that has to be evaluated in the database is the formula

$$Guardian(1,x) \rightarrow Sponsor(1,x).$$

This is evaluated using the SLDNF [16] proof procedure in the updated database. The above instance is negated and chosen as a top clause. figure 6.1 shows the partial search space in which at each step a leftmost literal is selected from a goal under the safe literal selection strategy. In figure 6.2, the tree diagram illustrates the process of deriving the above two sets of atoms by a depth-first approach. All positive literals belong to the set $pos_{D',D'}$ and all negative literals belong to the set $neg_{D',D'}$.

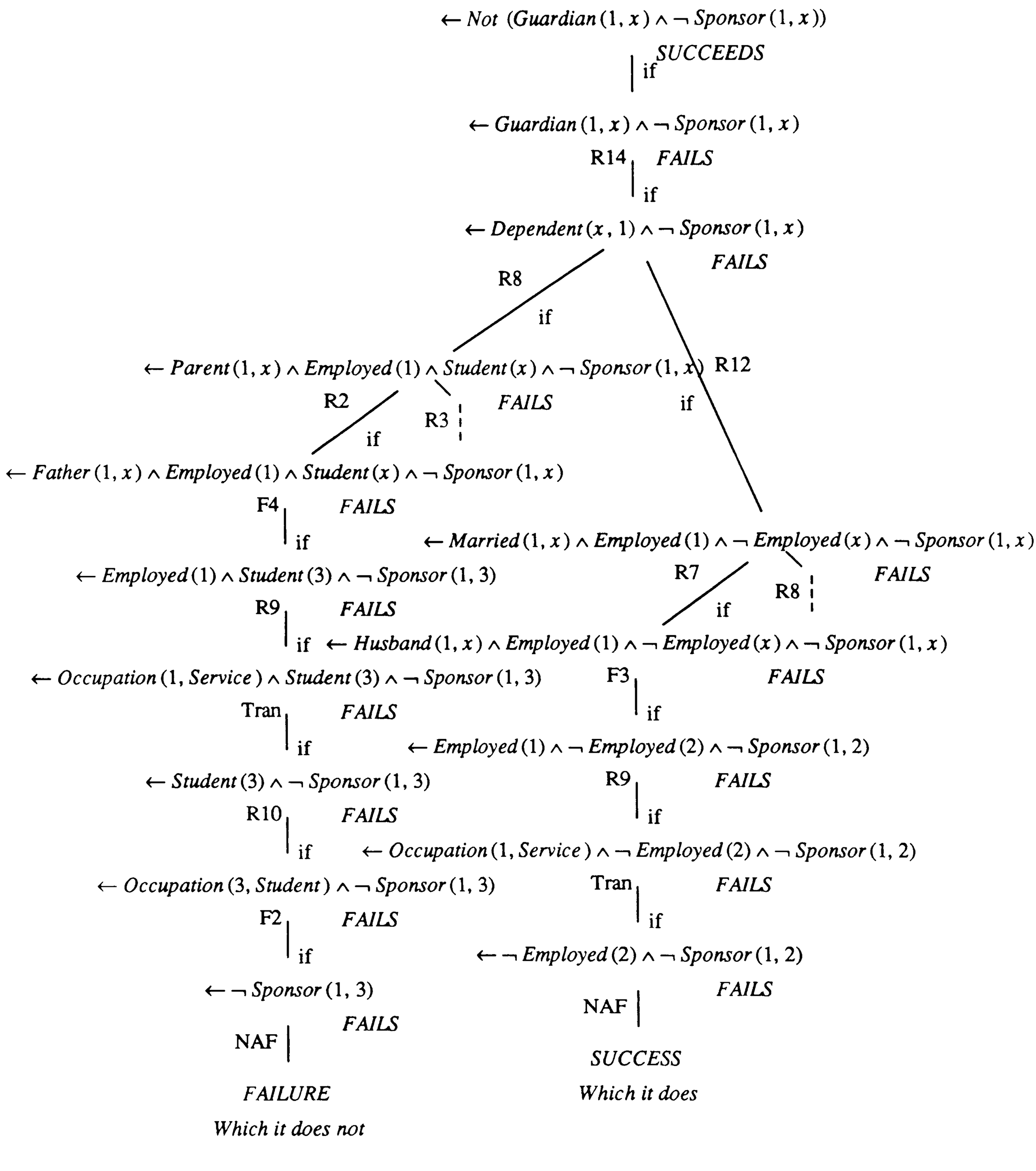

Figure 6.1. Partial search space for evaluating the constraint in Lloyd et al.'s method by SLDNF-proof procedure.

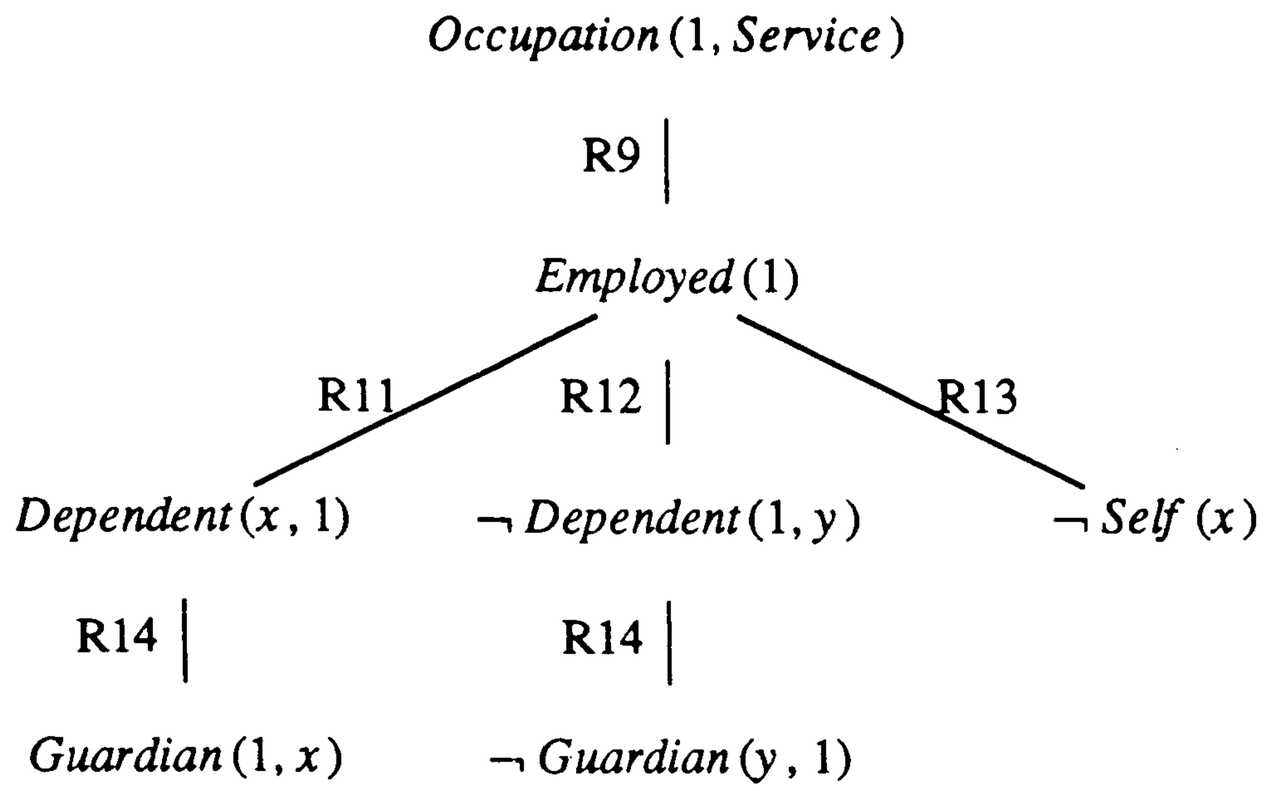

Figure 6.2. Illustration of the process of deriving the two sets of partially instantiated atoms $pos_{D,D'}$ and $neg_{D,D'}$ in Lloyd et al.'s method.

6.2.2. Method proposed by Decker

Like the previous method, the method proposed by Decker in [25] is also based on a standard view of constraint satisfiability. The method is a generalisation of the simplification method put forward by Nicolas [79] for relational databases. Decker's method simplifies the constraints with induced updates and only affected constraints are evaluated. Each constraint F is represented in the database as a set $uc(F)$ of *update constraints*, each of which is either an *insert constraint* or a *delete constraint*. An insert constraint has the form

insert UL only if UC

and a delete constraint has the form

delete UL only if UC,

where *UL* is an *update literal* and UC is an *update condition* which is in range form. When a transaction T is performed on a database D the set of all update literals due to T are divided into two sets of ground atoms, D^T and D_T, defined as follows. Let DT denote the set of clauses obtained after execution of the transaction T on the database D. Then, $D^T = DT^* - D^*$ and $D_T = D^* - DT^*$, where '-' denotes the set difference and D^* denotes the set of facts derivable from D. Each element of D^T has been called an *include-fact* and each element in D_T has been called a *remove-fact*. Intuitively, D^T (D_T) is the set of facts added to (deleted from) the database due to the transaction.

Decker has proposed an algorithm which will evaluate all relevant simplified constraints defined above for each fact in D^T and D_T. The elements of D^T and D_T are computed in stages for each unit transaction of T. The algorithm is called with argument *include*(C) (*remove*(C)) for each element *include*(C) (*remove*(C)) in T. The three-step algorithm essentially does the following. In the first step the relevant insert (delete) constraints simplified by each include (remove) fact of D^C (D_C) are evaluated, where D^C (D_C) is the set of facts added to (deleted from) the database by the clause C itself. In fact, the set of all include (remove) facts D^T (D_T) is the union of all D^C (D_C), where C is in DT (D). If a simplified constraint is falsified then an inconsistency occurs and the algorithm stops. In the second and third stages, the algorithm is called recursively with arguments of the form include(R) or remove(R), where R is an instantiated rule in D which could possibly add to or delete some facts from the database D due to the facts of D^C or D_C.

To apply the algorithm to example 6.1, the following two sets of facts are determined.

The facts added to the database as a result of the transaction are

$D^T = \{$ *Occupation*$(1,$ *Service*$)$, *Employed*(1), *Dependent*$(3, 1)$, *Guardian*$(1, 3)$ $\}$.

The facts deleted from the database as a result of the transaction are

$D_T = \{$ *Self* (2) $\}$.

The following shows the different stages of determining the sets D^T and D_T. D^T (D_T) is the union of all D^{T_i} (D_{T_i}).

Initial Stage :

$D^{T_0} = \{$ *Occupation*$(1,$ *Service*$)$ $\}$

$D_{T_0} = \{\ \}$

No constraint is affected by an atom in D^{T_0} or D_{T_0}.

First Stage :

$D^{T_1} = \{$ *Employed*(1) $\}$

$D_{T_1} = \{$ *Self* (2) $\}$

No constraint is affected by an atom in D^{T_1} or D_{T_1}.

Second Stage :

$D^{T_2} = \{$ *Dependent*$(3, 1)$ $\}$

$D_{T_2} = \{\ \}$

No constraint is affected by an atom in D^{T_2} or D_{T_2}.

Third Stage :

$D^{T_3} = \{\ Guardian(1, 3)\ \}$.

At this stage the fact $Guardian(1, 3)$ in D^{T_3} matches a condition of the first constraint. Therefore, the simplified constraint $Sponsor(1, 3)$ has to be evaluated in the updated database. Clearly this is not evaluable in the updated database and hence the integrity is violated.

The tree diagram in figure 6.3 illustrates the process of deriving the above two sets of atoms and evaluation of constraints. All positive literals belong to D^T and all negative literals belong to the set D_T.

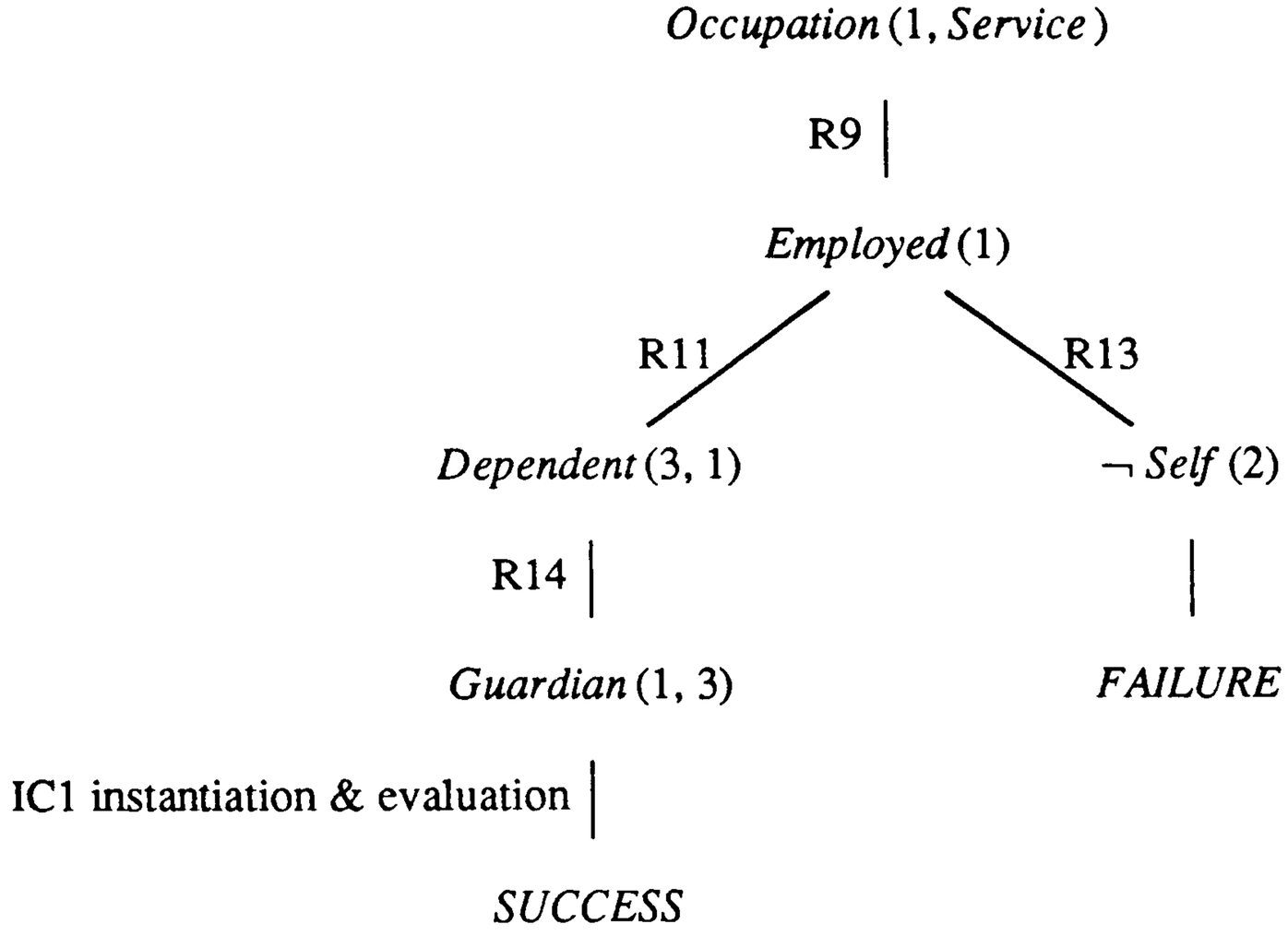

Figure 6.3. Illustration of the process of deriving the two sets of ground atoms D^T and D_T in Decker's method.

6.2.3. Method proposed by Kowalski et al.

The consistency method which was introduced by Sadri and Kowalski in [91] and later formalised in [55] by Kowalski et al., is based on the consistency view of constraint satisfaction. The method is essentially an attempt to construct a *refutation tree* (a refutation tree is a search space in which at least one path ends at the empty clause) taking each of the updates in turn as a

candidate top clause. If all possible attempts fail to do so then integrity is preserved in the updated database. The method uses a proof procedure which is an extension of the SLDNF proof procedure and provides both forward and backward reasoning. In SLDNF the clause can only be a denial and the set of formulae which appear in the derivation are either denials or the empty clause. But to reason forward, the new proof procedure allows as top clause any clause, denial or negated atom and the set of formulae appearing in the derivation may be any of these, the empty clause or a formula of the form

$$Not(A) \leftarrow L_1 \wedge \cdots \wedge L_n \qquad n \geq 1,$$

where A is an atom and the L_i are literals. The method selects literals through a safe literal selection strategy. The method defines some metalevel rules for reasoning about implicit deletions and a generalised resolution step for reasoning forward from negation as failure.

In the formalisation of the proof procedure a relation *Refute* (s, c) has been defined which means that there is a refutation with s as input set and c as top clause. The relation has been defined by five different rules. The first three formalise the process of resolution and negation as failure [16]. The last two cater for the cases of implicit deletions resulting from additions and deletions.

To apply the algorithm to example 6.1, the update is taken as a candidate top clause and a refutation tree is constructed. Part of the search space for this is shown in figure 6.4. According to the order of clauses and constraints of database D 1, only the leftmost branch will be followed. The rightmost branch of the search space shown in the figure implies an implicit deletion of the fact *Self* (2) from the database. In the example, the method selects the leftmost literal through a safe literal selection strategy. For ground negated atoms, the negation as failure rule has been applied. A clause number, labelling an arc, is used for resolution purposes with the selected literal.

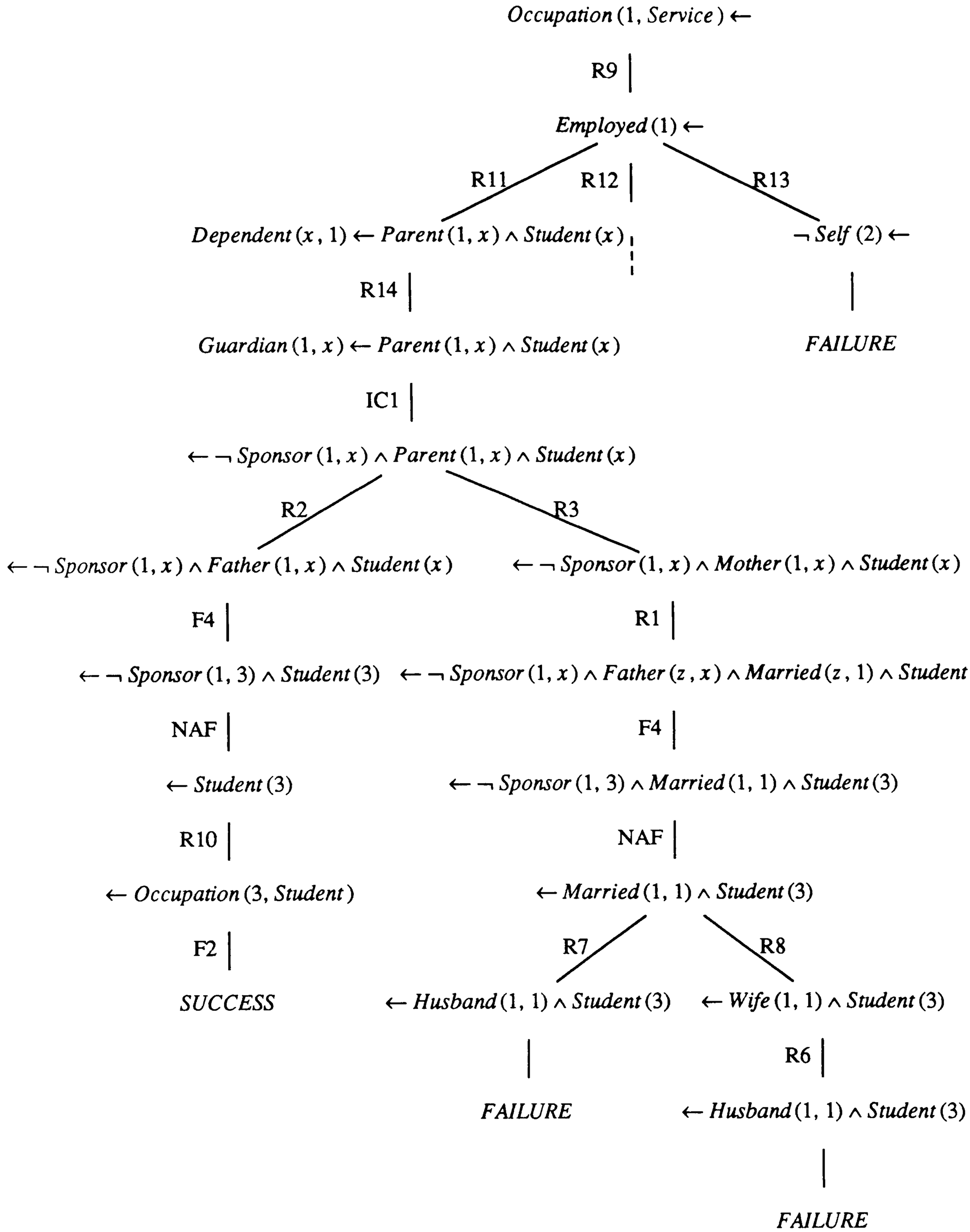

Figure 6.4. Partial search space generated by taking the update as the top clause in Kowalski et al.'s method.

6.2.4. Method proposed by Bry et al.

The integrity checking method proposed in [10] by Bry et al. can be divided into two phases. The first phase, called the *preparatory phase*, determines a set of *potential updates* which represent possible ground updates induced by an update to the database and then generates expressions called *update constraints* from potential updates and constraints. The set of potential updates is determined in the same way as for the method described in section 6.2.1. An update constraint is generated for every potential update which is relevant to a constraint. An update constraint has the form

$$Update\_Constraint(L, (\neg Delta(U, L) \vee New(U, SI))),$$

where $Delta(U, L)$ holds if and only if L is satisfied in $U(D)$ (U denotes a ground single fact update to a database D and $U(D)$ denotes the updated database). $New(U, F)$ denotes the evaluation of the formula F over the updated database $U(D)$. Another phase of the method, called the *evaluation phase*, evaluates all the update constraints generated by the preparatory phase.

Applying the method to example 6.1, in the first phase the set of potential updates is determined in the same way as in section 6.2.1 and consists of the union of the set of all atoms of $pos_{D,D'}$ and the set of all negated atoms obtained by negating each of the atoms of $neg_{D,D'}$. Also, in this phase the set of queries

$$\{ (\neg Delta(Occupation(1, Service), Occupation(1, Service)) \vee$$
$$New(Occupation(1, Service), (\neg Guardian(1, y) \vee Sponsor(1, y)))) \}$$

is obtained from generated update constraints. In the second phase the above set of queries is evaluated and in this case inconsistency will be detected.

6.2.5. Method proposed by Asirelli et al.

Asirelli et al. [3] have proposed two methods. The first, called the *consistency proof method*, is suitable for proving consistency of an existing database but is not very efficient for checking consistency after each transaction. The second method, which will be referred to as the modified program method, handles constraints with respect to a database (or program) D by finding a database, called the *modified database* ($mod(D)$), whose minimal model is exactly the subset of

the minimal model of D which satisfies the constraints. If S is the set of atoms satisfying a set of constraints I, then it can be proved that the minimal model of $mod(D)$ is a subset of S.

To apply the method to example 6.1, the constraint $IC\,2$ has been excluded from the constraint set as the method cannot deal with this constraint. The modified program for the database $D\,1$ and constraints $IC\,1$ of example 6.1 can be constructed as follows. First add to the database $D\,1$ the following definitions:

$$\Psi(u, v) \leftarrow Not\_Unify([u, v], [x, y])$$

$$\Psi(x, y) \leftarrow Sponsor(x, y)$$

where u, v are variables. Then $IC\,1$ can be rewritten as

$$Guardian(x, y) \rightarrow \Psi(x, y).$$

Thus the final database, ie the modified version of the database is given by

$mod(D\,1)$: R1-R13

$$Guardian(x, y) \leftarrow Dependent(y, x) \wedge \Psi(x, y)$$

together with the above definition of Ψ. $Mod(D\,1)$ is an internal form which generates correct answers to queries of $D\,1$, ie $mod(D\,1)$ is equivalent to $D\,1$ constrained by $IC\,1$.

6.2.6. Method proposed by Ling

Given a constraint $Not(P)$ which contains some views (a view is a positive literal which appears as the head of some clause), the method constructs a set of NF negative formulae $EXTRA(Not(P))$ using rules of the database. The two-step algorithm for constructing this set from a given constraint $Not(P)$ operates as follows. The algorithm is first called with the set consisting of the constraint $Not(P)$. The first step of the algorithm replaces a negative formula containing a view by a set of negative formulae with the help of the definitions of the view in the underlying database. The second step of the algorithm invokes the whole algorithm itself recursively with the new set of negative formulae, if there is a view occurring in one of the negative formulae of this set; otherwise, it stops. When a transaction is performed, the method evaluates only the affected constraints from these sets.

The construction of the set of NF negative formulae corresponding to a constraint may enter an infinite loop in the presence of recursive rules. However, in the case of example 6.1 this does not

create a problem and the NF negative formulae which represent constraints in this example are the following:

P : $Not\,(Guardian\,(x\,,\,y\,) \wedge Not\,(Sponsor\,(x\,,\,y\,)))$,

Q : $Not\,(Married\,(x\,,\,y\,) \wedge Student\,(x\,))$.

The following two sets can be constructed with the help of rules.

$EXTRA\,(Not\,(P\,)) =$

$\{\ Not\,(Father\,(x\,,\,y\,) \wedge Occupation\,(x\,,\,Service\,) \wedge Occupation\,(y\,,\,Student\,) \wedge Not\,(Sponsor\,(x\,,\,y\,)))$,

$Not\,(Father\,(z\,,y\,) \wedge Husband\,(z\,,x\,) \wedge Occupation\,(x\,,Service\,) \wedge Occupation\,(y\,,Student\,) \wedge Not\,(Sponsor\,(x\,,y\,)))$,

$Not\,(Husband\,(x\,,\,y\,) \wedge Occupation\,(x\,,\,Service\,) \wedge Not\,(Occupation\,(y\,,\,Service\,) \wedge Not\,(Sponsor\,(x\,,\,y\,))))$,

$Not\,(Husband\,(y\,,\,x\,) \wedge Occupation\,(x\,,\,Service\,) \wedge Not\,(Occupation\,(y\,,\,Service\,) \wedge Not\,(Sponsor\,(x\,,\,y\,))))\ \}$.

$EXTRA\,(Not\,(Q\,)) =$

$\{\ Not\,(Husband\,(x\,,\,y\,) \wedge Occupation\,(x\,,\,Student\,))$, $Not\,(Husband\,(y\,,\,x\,) \wedge Occupation\,(x\,,\,Student\,))\ \}$.

As a result of insertion of the fact $Occupation\,(1,\,Service\,)$, incremental checking is required for the NF negative formulas from the set $EXTRA\,(Not\,(P\,))$. They are unified appropriately by the update literal and simplified. Since the simplified form of the first one is not true in the database, integrity is violated due to the update.

6.2.7. Path finding method

The *path finding method* method has already been described in chapter 5. To apply the path finding method to example 6.1, the update literal $Occupation\,(1,Service\,)$ is taken as a source. Because of the insertion of this fact into the database, with the help of $R9$ one can say that the fact $Employed\,(1)$ is implicitly added to the database and so $Employed\,(1)$ is the next literal of the path. Continuing in this way one can construct the following success path (which ends at the head of a constraint)

$$Occupation\,(1,\,Service\,) \rightarrow Employed\,(1) \rightarrow Dependent\,(3,\,1) \rightarrow Guardian\,(1,\,3) \rightarrow IC\,(1)$$

and the complete path space generated by the insertion is shown in figure 6.5. The leftmost branch corresponds to the success path and this will be followed according to the order of the clauses. The successor literal of $Employed\,(1)$ along the second branch of the path space is $\neg\, Dependent\,(1,\,y\,)$, by rule $R\,12$. This shows that instances of $Dependent\,(1,\,y\,)$ are likely to be

deleted from the database due to the implicit addition of *Employed* (1) to the database.

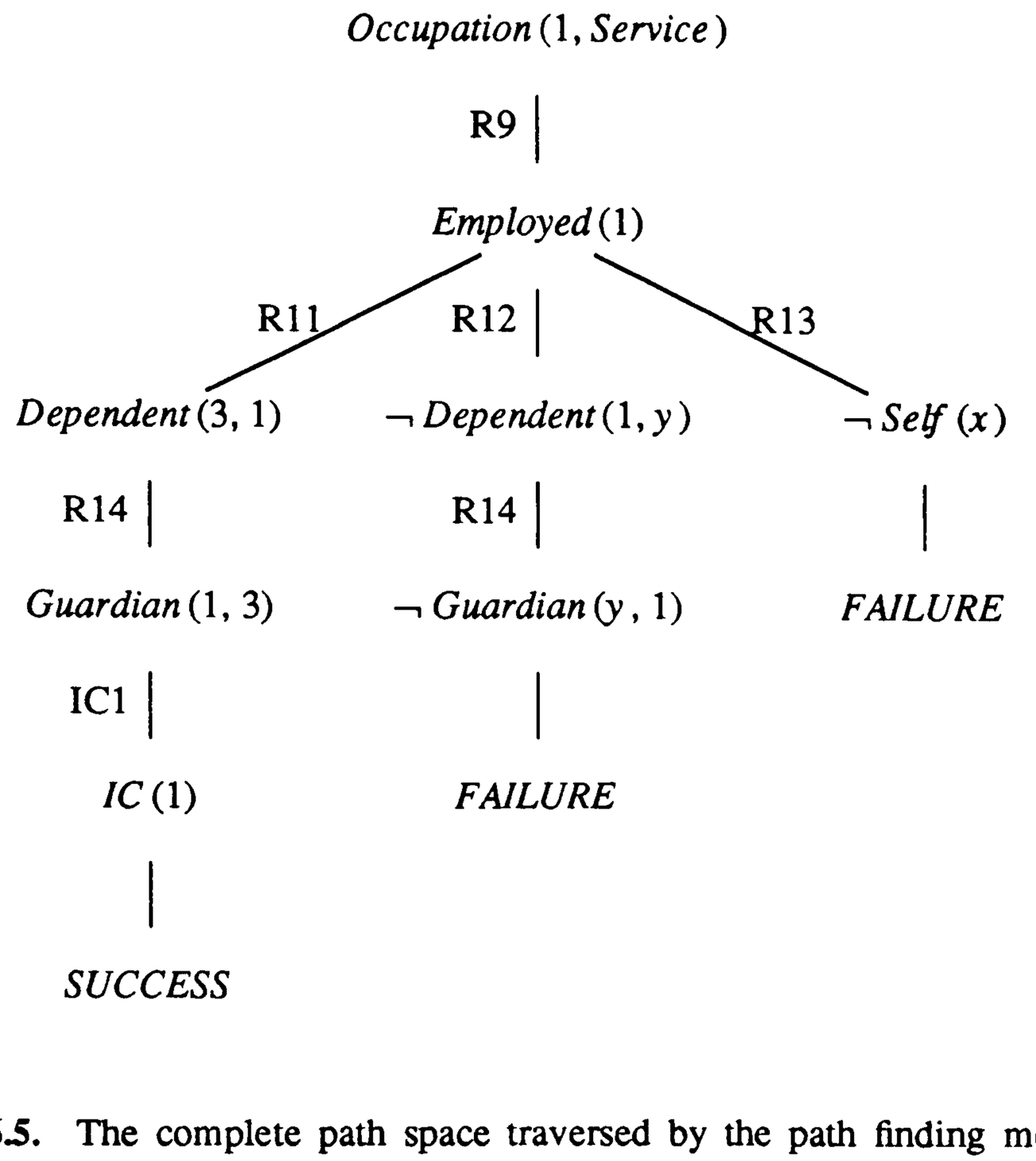

Figure 6.5. The complete path space traversed by the path finding method, taking the update as the source.

6.3. Comparison

In this section, methods are compared from both theoretical and practical points of view. To achieve the latter, the Simple method and the methods proposed in sections 6.2.1, 6.2.2, 6.2.3 and 6.2.7 have been implemented in a simple manner without going into much detail (e.g., if a constraint is violated then the implementation simply reports this fact without showing any detail of the cause of violation). The following paragraph discusses briefly the implementation issue. The other methods have been excluded for various reasons. For example, the reason for excluding Asirelli et al.'s method is its limited constraint expressiveness. Ling's method is excluded because it cannot cope with recursive databases. The reason for excluding Bry et al.'s method is its close similarity with Lloyd et al.'s method with the result that its performance figures will be similar to those of Lloyd et al. in most cases considered here.

In the implementation of each of the methods, facts have been stored directly as Prolog facts. Rules are stored directly as Prolog rules except in the method by Kowalski et al. In

this method rules (and also constraints) have been stored under a ternary predicate *cl* as *cl*(*Ref* , *Type* , *Clause*), where *Ref* is an unique identification of the clause (rule or constraint), *Type* is either a constraint or a rule, and *Clause* is a list of structures where each structure has the form (*Side* , *Sign* , *Atom*) defined as follows. *Side* is either condition or conclusion, *Sign* is either positive or negative and *Atom* is an atom occurring either in the head or the body of the clause identified by *Ref* . Furthermore, a separate clause of the form *depend*(L , $H \leftarrow B$) for Decker's method, *depend*(H , L) for Lloyd et al.'s method, *depend*(*Ref* , H , L) for the path finding method and *depend*(*Ref* , I , S) for Kowalski et al.'s method has been maintained, where L is a literal occurring in the body of the rule $H \leftarrow B$ identified by *Ref* and S is a structure of the form defined above corresponding to the literal occurring in the I -th position of the clause $H \leftarrow$.

Constraints are not distinguished from rules for their representation in both Kowalski et al.'s method and the path finding method. In Decker's method a constraint has been converted to a set of update constraints and each update constraint is stored under the predicate *update_cons*. In Lloyd et al.'s method each constraint has been stored under a separate predicate *ic* . Transactions are stored under the predicate *trans* in an appropriate format to reason with two database states, ie before and after the transaction.

The simplest way of maintaining consistency (ie the Simple method) is by evaluating every constraint for each update; however, this is very inefficient since it ignores the assumption that the database satisfies the constraints prior to the update. All the implemented methods do take advantage of this assumption. To demonstrate the difference which this makes, consider the following example (also, example 6.5 later in this section).

Example 6.2

Database D 2 :

Rules :

as described in database D 1.

Facts :

A list of the following 1095 facts not involving constants 1, 2, 3:

175 facts under the predicate *Father*,

228 facts under the predicate *Husband*,

620 facts under the predicate *Occupation*,

72 facts under the predicate *Sponsor*.

A list of 1085 facts of example 6.3.

A list of 1058 facts of example 6.4.

Occupation (1, *Service*)

Occupation (2, *Service*)

Occupation (3, *Student*)

Father (1, 3)

Sponsor (1, 3)

Integrity Constraints $I2$:

Two constraints of example 6.1.

$Occupation(x, y) \wedge Occupation(x, z) \rightarrow y=z$

Transaction :

insertfact *Husband* (1, 2)

insertfact *Sponsor* (2, 3)

Database $D2$ satisfies $I2$ before the update. Each entry in table 6.1 denotes a method and the time taken by that method to report inconsistency due to update. It is observed that the time taken by the simple SLDNF approach is much greater than that for any of the other methods because it does not take advantage of the assumption that the database $D2$ satisfies $I2$ prior to the update. For example, it evaluates the last constraint in the updated database even though this is redundant as it is not affected by any of the induced updates.

A disadvantage of Kowalski et al.'s method is that the degree of *granularity* [95] required for the meta-interpreter is much finer than that required for the other implemented methods. This is due to the fact that the former models the literal selection strategy. This should result in a loss of efficiency. On the other hand the other implemented methods rely on the strategy built into Prolog. For this reason, in some cases the time taken by Kowalski et al.'s method to report inconsistency is considerably larger than the time taken by any of the other methods for the same transaction. The literal selection strategy in this method can play an important role in the efficiency of the method. The test results here are based on an implementation of the method which selects (under safe selection) a literal from the condition

Method	Time
Simple	336.1
Lloyd et al.	10.0
Decker	31.5(18.9)
Kowalski et al.	133.7(1.1)
Path finding	18.4

Table 6.1. Times taken (in cpu seconds) by different methods
to report inconsistency in the case of example 6.2

part of the top clause, until no such literal remains. Another example of a literal selection strategy is to select a literal from the conclusion part first, if there is a conclusion. In all the tables the additional entry (within parentheses) corresponding to Kowalski et al.'s method is the time taken by the method with this literal selection strategy (excluding rule $R5$ to avoid infinite loop). It has been shown by Kowalski et al. that Decker's method, and the method of Lloyd et al. can be approximated by considering respectively the previous two literal selection strategies. Due to this fact, the test results of Kowalski et al.'s method with the former literal selection strategy behave in the same way as those of Decker's method and test results with latter literal selection strategy behave in the same way as those of Lloyd et al.'s method.

The path finding method, Decker's method and the method of Kowalski et al. can handle those recursive rules for which the underlying SLDNF-resolution terminates. The simplification of constraints in both Lloyd et al.'s method and Bry et al.'s method are independent of the evaluation method chosen. Each of these methods will work in the presence of recursive rules if the underlying query evaluator handles recursion. Lloyd et al.'s method considers a typed database and converts it to a type free database by some transformations. The other methods need the database to be range restricted to evaluate a query correctly.

When a transaction is performed on the database the path finding method derives in stages a set of fully instantiated atoms added to the database and a set of partially instantiated atoms whose instances are likely to be deleted from the database. By contrast Decker's method derives two sets of fully instantiated atoms. The first set is the set of atoms implicitly added to the database and the second set is the set of atoms deleted from the database. The methods of Lloyd et al. and Bry et al. derive two sets of partially instantiated atoms. These represent respectively the set of atoms likely to be added to the database and the set of atoms likely to be

deleted from the database. The approach taken by the path finding method is identical to that of Decker when there is no implicit deletion in the database. When an implicit deletion occurs in a stage, the method follows a similar approach to that of Lloyd et al. It can be observed from figure 6.5 that the rightmost two paths starting from the literal *Employed*(1) are similar to the paths in the same position of figure 6.2. This is because these two paths do not show any implicit addition to the database. It is also observed from figure 6.5 that starting from the root the leftmost branch is similar to the branch in the same position in figure 6.3. This is due to the fact that this path does not show any implicit deletion from the database.

The major difference between the path finding approach and the approach by Decker on the one hand and those of Lloyd et al. and Bry et al. on the other is that, in the first kind of approach instantiation and evaluation of constraints and the derivation of implicitly added or deleted facts is done simultaneously, whereas in the latter approaches instantiation and evaluation of constraints are performed after the derivation of the two sets of partially instantiated atoms.

Decker's method could be inefficient for a complex transaction which requires reasoning with two database states (before and after update) to calculate the two sets of atoms mentioned earlier. For each induced update, ie each fact which is implicitly added or deleted, the method checks whether it is provable in the database before update or not. The method may show better performance than the other methods when constraints are relevant to the induced updates, but some of the induced updates are already provable in the database before update. In all the tables the extra entry within parentheses corresponding to Decker is the time taken by the method when this kind of checking (in the considered examples this is redundant) has been excluded from the implementation of the method.

In both the approaches by Lloyd et al. and Bry et al., two sets of partially instantiated atoms are always derived even if no ground instance of these atoms is true in the updated database and hence may cause redundant partial evaluation of the constraints in the case of Lloyd et al.'s method (Bry et al.'s method discards this possibility by using the *Delta* predicate). Consider the following example.

Example 6.3

Database D 3 :

Rules :

as described in Database D 1.

Facts :

A list of following 1085 facts not involving constants 1, 2:

177 facts under the predicate *Father*,

229 facts under the predicate *Husband*,

620 facts under the predicate *Occupation*,

59 facts under the predicate *Sponsor*.

$Occupation(1, Service)$

$Occupation(2, Service)$

Integrity Constraints I 3:

$Guardian(x, y) \rightarrow Sponsor(x, y)$

$Sponsor(x, y) \wedge Guardian(z, y) \rightarrow parent(x, y)$

Transaction :

insertfact $Husband(1, 2)$.

Database D 3 satisfies the constraints I 3. Now if one wants to insert the fact $Married(1, 2)$ to D 3, these two methods will derive the two sets of atoms

$\{Husband(1, 2), Mother(2, y), Parent(2, y), Ancestor(x, y), Dependent(y, 2), Guardian(2, y), Wife(2, 1), Married(2, 1), Self(1), Married(1, 2), Dependent(2, 1), Guardian(1, 2), Self(2)\}$ and $\{\}$.

Derivation of these two sets of atoms is highly redundant as no ground instance of these atoms under the predicate *Guardian*, which are simplifying the constraints, is true in the updated database and hence will cause redundant evaluations of the instantiated constraints corresponding to the atoms $Guardian(2, y), Guardian(1, 2)$. As no fact is implicitly added to the database corresponding to the insertion of $Married(1, 2)$, in both the approaches by Decker and the path finding method, no constraint would be evaluated. The time taken by different methods to check consistency when the above insertion is made to the database D 3 in the presence of constraints I 3 is given in table 6.2.

Method	Time
Lloyd et al.	3.4
Decker	0.8(0.2)
Kowalski et al.	0.4(3.3)
Path finding	0.2

Table 6.2. Times taken (in cpu seconds) by different methods
to report consistency in the case of example 6.3

In the approach proposed by Lloyd et al. (and also that proposed by Bry et al.), it is sometimes possible to have the evaluation of unsimplified constraints which may cause inefficiency. For example, if one wants to delete (resp. insert) a ground instance of *Occupation* (*y* ,*Service*) from (to) a database D containing rules described in Database D 1, then *Self* (*x*) would be a member of $pos_{D,D'}$ ($neg_{D,D'}$) described in section 4.1. In that case if there is a constraint C relevant to the atom *Self* (*x*) (¬ *Self* (*x*)), no simplification for C is possible before its evaluation. In Decker's method constraints are always simplified before evaluation as the method always calculates two sets of ground atoms implicitly added to or deleted from the database. In the above case of addition, the path finding method will follow Decker's approach and hence C will be simplified accordingly. But, in the above case of deletion, the method will follow the approach of Lloyd et al. and hence will cause the evaluation of unsimplified C.

The path finding method and Decker's method suffer from the drawback that all induced updates are computed, including those for which no constraints are relevant. Kowalski et al.'s method can avoid this by proper literal selection strategy. On the other hand the methods of Lloyd et al. and Bry et al. do not suffer from this drawback as they compute only potential updates that represent possible ground induced updates. Consider the following example.

Example 6.4

Database D 4 :

Rules :

as described in Database D 1.

Facts :

A list of following 1058facts not involving constants 1, 3:

184 facts under the predicate *Father*,

226 facts under the predicate *Husband*,

600 facts under the predicate *Occupation*,

48 facts under the predicate *Sponsor*.

A list of 10 facts under the predicate *Father* with 1 as first argument.

Occupation (2, *Service*)

Occupation (3, *Student*)

Father (1, 3)

Integrity Constraints *I* 4:

$$Father(x, z) \wedge Father(y, z) \rightarrow x = y$$

Transaction :

insertfact *Husband* (1, 2).

Database *D* 4 satisfies the constraints *I* 4. Now if one wants to insert the fact *Husband* (1, 2) to *D* 4, these two methods will derive in stages the following set of atoms

{*Wife* (1, 2), *Married* (1, 2), *Married* (2, 1), *Mother* (2, *C*), *Parent* (2, *C*), *Ancestor* (2, *C*),
Dependent (*C* , 2), *Self* (2), *Guardian* (2, *C*) :
C is a constant and *Father* (1, *C*) is true in the updated database}

and, possibly, some more under the predicate *Ancestor*. It is clear that no constraint in *I* 4 is relevant to the above set of atoms and will cause redundant evaluation. The evaluation time in this case depends mainly on the number of facts present in the database under the predicate *Father* whose first argument is the constant 1. The time taken by the different methods to check consistency when the above insertion is made to the database *D* 4 in the presence of constraints *I* 4 is given in the column corresponding to 'add' of table 6.3.

The entries in the column headed 'del' in table 6.3 are the times taken by different methods to check consistency when the fact *Husband* (1, 2) is deleted from the updated database

Method	Operation	
	add	del
Lloyd et al.	0.7	0.7
Decker	62.9(38.5)	63.0(43.9)
Kowalski et al.	>>100.0(0.5)	>>100.0(31.8)
Path finding	37.4	0.3

Table 6.3. Times taken (in cpu seconds) by different methods
to report consistency in the case of example 6.4

of example 6.4. In this case Decker's method calculates all the facts implicitly deleted from the database due to the deletion of *Husband*(1, 2) and these are precisely those which were implicitly added to $D4$ by the addition of *Husband*(1, 2). This calculation is redundant and the path finding method avoids this by following Lloyd et al.'s approach.

In Lloyd et al.'s approach, an instance of a literal from any one of the two sets of partially instantiated atoms may be evaluated more than once if that particular literal simplifies more than one constraint. This will cause inefficiency. The path finding method also suffers from this drawback but only when a negative literal simplifies a constraint. On the other hand Decker's method does not have this drawback at all as the literal which simplifies the constraint is always true in the database and hence it is removed from the unified constraint.

In the path finding method finding a path from an update to the head of a constraint is similar to the problem of finding a refutation with an update literal as a top clause in Kowalski's method. In the latter method finding a refutation means arriving at an empty head of the denial form of a constraint from an update literal. In the path finding method constructing a success path means reaching the head of a constraint from an update literal. The difference lies in the way in which this is achieved - in the method put forward by Kowalski et al. the computation of positive induced updates can be deferred by a proper literal selection strategy but the computation of negative induced updates is essential, in the path finding method the computation of positive induced updates is essential and the computation of negative induced updates is deferred.

As the database size increases, the times taken by some operations (eg. search, unification) employed with constraint checking increase accordingly and in this kind of situation the time taken by each of the methods to check consistency varies almost directly as the number

of relevant clauses present in the database. Table 6.4 shows the test results obtained by performing the transaction of example 6.2 on the database of example 6.2 varying the number of facts in the following way. Initially the database contains the 1095 facts of example 6.2. After performing the tests another 1085 facts of example 6.3 are added and the tests are performed. Final tests are performed by adding 1058 facts of example 6.4 to the database. No facts in the database involve constants 1, 2, 3. It can be observed that the time taken by each method to perform the transaction increases as the number of facts in the database increases.

Method	No of Facts		
	1095	2180	3238
Lloyd et al.	1.9	2.9	4.0
Decker	4.9(2.5)	15.1(8.6)	31.3(18.7)
Kowalski et al.	31.1(0.7)	74.5(0.8)	133.6(0.9)
Path finding	2.4	8.3	18.3

Table 6.4. Times taken (in cpu seconds) by different methods in the case of example 6.2 varying the number of facts

Finally, from a more practical point of view, a test has been carried out using the following example.

Example 6.5 :

The database *D* 5, the set of constraints *I* 5 and the transaction have been described in appendix 2.

Example 6.5 describes three different states of a database by varying the number of facts but keeping the set of rules the same. Example 6.5 also describes a set of constraints imposed on each of these three states of the database and a set of transactions to be performed on each of these three states of the database. Each of the three states of the database *D* 5 satisfies the set of constraints *I* 5. Table 6.5 contains the results obtained from applying the set of transactions to each of the three different states of the database using the different constraint checking methods.

Method	No of Facts		
	1010	3062	5119
Simple	37.0	311.5	860.6
Lloyd et al.	2.9	11.7	28.3
Decker	16.6(2.5)	54.0(12.3)	99.4(29.8)
Kowalski et al.	14.2(17.0)	58.6(67.8)	129.1(147.9)
Path finding	2.5	11.9	29.0

Table 6.5. Times taken (in cpu seconds) by different methods in the case of example 6.5 varying the number of facts

CHAPTER 7

Negation as Possible Failure

This chapter presents a rule for inferring negative information, referred to as *negation as possible failure*. By this rule, the concept of determining the truth functionality of a ground literal with respect to an indefinite database is based on Clark's idea of the *completed database*. To define the declarative semantics of negation as possible failure, an indefinite database is transformed to a set of its *possible forms* and the semantics is given in terms of the completion of each of these possible forms. A ground atom which can be derived from the completion of each of the possible forms associated with the indefinite database will be taken as true in the database; if its negation can be derived from the completion of each of them, it is taken to be false; otherwise, it is indefinite. The procedural semantics is based on two mutually recursive resolution schemes. These two resolution schemes coincide and reduce to the mechanism of simple SLDNF-resolution when the database is definite. The implementation for query evaluation based on the introduced semantics for negative information is considered. The implementation constructs dynamically the set of definite databases associated with an indefinite database during the execution of a query.

The chapter is organised as follows. The following section provides a brief review of the field of semantics for negative information in indefinite databases. A detailed discussion comparing different semantics, including the one proposed in this chapter, is beyond the scope of the thesis. Section 7.2 introduces the idea of possible forms of an indefinite database. In section 7.3, the declarative semantics for negative information in an indefinite database is given. Section 7.4 provides the two resolution principles for implementing the declarative concept of an answer to a query based on the semantics introduced in section 7.3. The rule negation as possible failure has been introduced in section 7.5. The last section deals with a Prolog implementation of the two resolution principles.

7.1. A brief overview of the field

Semantics for negative information in indefinite databases have already been studied by several authors. Approaches include the Generalised Closed World Assumption (GCWA) by Minker [77] , the Extended Generalised Closed World Assumption (EGCWA) by Yahya and Henschen [107] , the perfect model semantics by Przymusinski [84] , the Disjunctive Database Rule (DDR) by Ross and Topor [90] , the weak completion theory for non-Horn programs by Lobo et al. [68] etc.

In the definition of GCWA, Minker has considered indefinite databases consisting of a set of clauses of the form

$$A_1 \vee \cdots \vee A_m \leftarrow M_1 \wedge \cdots \wedge M_n, \qquad m \geq 1,\ n \geq 0 \qquad (7.1)$$

where A_i's, M_j's are atoms. Under the semantics of the GCWA a ground atom is true in a database D if it is present in all minimal models of D, it is false if it is not present in any minimal model of the database, and it is indefinite otherwise. The EGCWA is an extension of the GCWA in which a disjunction of ground negative literals K may be inferred from a database D if and only if the disjunction K is true in every minimal model for D.

Przymusinski's inference rule is based on perfect models. Suppose M and N are two distinct models of an indefinite database D. Then N is *preferable* to M (or $N \ll M$), if for every ground atom A (under a predicate P) in $N-M$ there is a ground atom B (under a predicate Q) in $M-N$ such that $P < Q$, where $<$ is a *predicate priority relation* between the predicates of D. An important result about such a relation is that D is stratified if and only if $<$ is a partial order. A model M of D is *perfect* if there are no models preferable to M. The *Perfect Model Rule* (PMR) is that a sentence W may be inferred from D if W is true in every perfect model for D.

Ross and Topor propose an inference rule, called the *Disjunctive Database Rule* (DDR), for inferring negative information from disjunctive databases, where a *disjunctive database* is a set of clauses of the form (7.1). Under the syntactic definition of DDR, a ground atom A may be taken to be false in a disjunctive database D if A is in the greatest closed set of D ($gcs(D)$), where the greatest closed set of D is defined as follows. Let S be a subset of $HB(D)$. Then S is a closed set of D if for every element A of S and for every ground instance C of a clause in D such that A is in the head of C, there exists an atom B in the body of C such that B is in S. The *greatest closed set* of D is the union of all closed sets of D. The DDR has been extended to the class of layered databases, where a *layered database* is a pair (D, L) of an

indefinite database D and a level mapping L for D.

Lobo et al. have generalised Clark's completion definition for definite databases to indefinite databases. Such completion has been used as a declarative semantics to study the rule of negation, *Negation As Finite Failure for Non-Horn programs* (NAFFNH). A sound procedure, based on SLGNF-resolution, has been provided for answering queries in indefinite databases. The idea is similar to the one proposed in this chapter but differs from it in a number of respects, both in the definition of declarative semantics and in the resolution procedures.

Several extensions [82, 96, 69, 93] of Prolog-style theorem provers to full first-order logic have been proposed. Each of them can be extended suitably (e.g., capability for inferring negative information without storing negative clauses explicitly) for implementing a query evaluation system for indefinite databases. In the next chapter, nH-Prolog [69, 93] has been extended by the capability for inferring negative information according to the semantics provided in this chapter.

7.2. Possible forms of an indefinite database

Recall that a database clause in an indefinite database has the form

$$A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n \qquad m \geq 1 \qquad (7.2)$$

An alternative form of the above clause which will also be used in this chapter is

$$A_1 \vee \cdots \vee A_m \leftarrow M_1 \wedge \cdots \wedge M_p \wedge \neg N_1 \wedge \cdots \wedge \neg N_q \qquad m \geq 1,\ p, q \geq 0 \qquad (7.3)$$

where A_i's, M_j's and N_k's are atoms.

Unless otherwise mentioned, in the rest of the chapter the term 'database' will always be taken to mean a 'general database', i.e. an 'indefinite database'. Consider the following example of an indefinite database.

Example 7.1 :

Database D 1 :

Rules :

Definite rules :

$$Staff(x) \leftarrow Clerk(x)$$

$$Staff(x) \leftarrow Typist(x)$$

$$Employee(x) \leftarrow Officer(x)$$

Indefinite rule :

$$Typist(x) \vee Clerk(x) \leftarrow Employee(x) \wedge \neg Officer(x)$$

Facts :

Definite fact :

$$Employee(Das)$$

Indefinite fact :

$$Officer(Choux) \vee Clerk(Choux)$$

As for a definite database, negative facts are not explicitly represented in the database. Instead, a special rule is introduced to infer negative facts from the database. This is somewhat similar to the closed world assumption or negation as failure in the case of definite databases.

Definition : Let D be a database and R be a clause in D. Let A be an atom occurring in the head of R. The *possible form* of R with respect to A denoted by $pos(R, A)$ is the clause obtained from R by replacing its head by A.

According to this definition, the possible form of a definite rule R with respect to the only atom in its head is the rule R and the possible form of a definite or indefinite fact with respect to the atom A appearing in it, is the fact A. The set of clauses comprising at least one possible form of each of the clauses of D is called a *possible form* of the database D. For example, database $D1$ has nine possible forms, one of which is the union of the set of all definite clauses of $D1$ with the following two possible forms

$$Clerk(x) \leftarrow Employee(x) \wedge \neg Officer(x)$$

$$Officer(Choux)$$

corresponding to the two indefinite clauses of $D1$.

A query relating to the database $D1$ is

Who are the staff ?

and its representation in the above form is given by

$\leftarrow Staff\,(x)$.

In the context of both relational and definite databases, a substitution for the variables of a query makes the bound body of the query either true or false in the database. If the bound body is true in the database then the corresponding substitution is called an answer substitution. By contrast, an indefinite database may represent several possible worlds and in the context of this kind of database, a substitution for the variables of a query can make the bound body of the query neither true nor false in the database. The substitution corresponding to this possibility is a possible answer substitution. The different possible worlds of an indefinite database arise from different possible forms of the database. Hence, in an arbitrary database an answer to a query is either definite or possible. The substitution $\{x/Das\}$ to the query $\leftarrow Staff\,(x)$ in the context of database $D1$ is a definite answer substitution whereas the substitution $\{x/Choux\}$ is a possible (or indefinite) answer substitution.

Due to the indefiniteness of an answer to a query in an indefinite database, two different query evaluation mechanism are required, i.e. a possible query evaluation mechanism which will return possible answers to a query and a definite query evaluation mechanism which will return definite answers to a query. A possible answer to a query applied to an indefinite database is an answer which is true in at least one of the possible worlds associated with it whereas a definite answer is true in all possible worlds of the database.

7.3. The declarative semantics for negative information

Let D be a database and $HB\,(D)$ its Herband base. Let $\{D_1, ..., D_n\}$ be the set of all possible forms of D such that the completion of each D_i is consistent. Then the Herband base $HB\,(D)$ can be partitioned into three subsets:

$$Def\_true\,(D) = \bigcap_{i=1}^{n} \{A : A \in HB\,(D) \text{ and } comp\,(D_i) \vdash A\}$$

$$Def\_false\,(D) = \bigcap_{i=1}^{n} \{A : A \in HB\,(D) \text{ and } comp\,(D_i) \vdash \neg A\}$$

$$Unknown\,(D) = HB\,(D) - Def\_true\,(D) \cup Def\_false\,(D)$$

where the symbol '–' denotes set difference. The set of all facts in *Def_true*(D) can be taken as *definitely true* in D whereas the set of all facts in *Def_false*(D) are *definitely false* in D. The facts in *Unknown*(D) are *possibly true* in D.

Clearly, when the database is definite the above definition of semantics reduces to Clark's idea of a completed database. When the database is definite and hierarchical the set *Unknown* is empty and the Herband base is divided between the two sets *Def_true* and *Def_false*. The SLDNF-resolution mechanism suffices to determine any element of the set *Def_true* and hence any element of the set *Def_false*. When all three subsets are non-empty, the situation is more complex. In this case the definite resolution mechanism determines the elements of the set *Def_true*. The possible resolution mechanism will be able to determine the set constituting of elements which are either in the set *Unknown* or in the set *Def_true*. An element of the Herband base which cannot even be determined by the possible resolution mechanism can be taken to be false and hence is a member of *Def_false*.

7.4. The procedural semantics for negative information

The procedural semantics is based on two mutually recursive resolution schemes, one for possible resolution and one for definite resolution.

7.4.1. Possible resolution

Definition : A *computation rule* is a function which maps a goal to a literal, called the *selected literal*, in that goal.

Each derivation of a possible resolution constructs a sequence of positive databases and a sequence of negative databases. A possible database corresponds to a possible form and negative databases help to keep track of the ground negative atoms resolved during a derivation. The marking process of clauses helps to construct different possible forms and also to avoid redundant computations. All the clauses in the database are unmarked before the start of a derivation.

Definition : Let D be a database, G a goal, and R a computation rule. Initially, none of the clauses in D is marked as used possibly with respect to atoms occurring in their heads. A *possible derivation* for $D \cup \{G\}$ via R consists of a sequence $D_0^+ = D, D_1^+, D_2^+, \ldots$ of positive

databases, a sequence $D_0^- = \{\}, D_1^-, D_2^-, \ldots$ of negative databases, a sequence $G_0 = G, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of ground negative literals or program clauses from the possible forms of D, and a sequence $\theta_1, \theta_2, \ldots$ of substitutions satis fying the following:

(a) For each i, G_i is $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_p$, the selected literal L_m in G_i is

1) a positive literal and L_m unifies with

I. A definite fact A in D_i^+ with an mgu θ_{i+1}. Then C_{i+1} is A, D_{i+1}^+ and D_{i+1}^- are respectively D_i^+ and D_i^-, and G_{i+1} is $\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$.

II. The head of a definite rule $H \leftarrow B$ in D_i^+ with an mgu θ_{i+1}. Then C_{i+1} is $H \leftarrow B$, D_{i+1}^+ and D_{i+1}^- are respectively D_i^+ and D_i^-, and G_{i+1} is $\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge B \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$.

III. An atom A occurring in an indefinite fact F of D_i^+ with an mgu θ_{i+1} and F has not been used possibly with respect to A. If for every member A' of D_i^-, $D_i^+ \cup \{pos(F, A)\} \cup \{\leftarrow A'\}$ has a finitely failed definite tree (introduced in the next section) then C_{i+1} is $pos(F, A)$, D_{i+1}^+ is $D_i^+ \cup \{A\}$, D_{i+1}^- is D_i^-, and G_{i+1} is $\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$. Finally, F is marked as used possibly with respect to A.

IV. An atom A occurring in the head of an indefinite rule $R: F \leftarrow B$ of D_i^+ with an mgu θ_{i+1} and R has not been used possibly with respect to A. If for every member A' of D_i^-, $D_i^+ \cup \{pos(R, A)\} \cup \{\leftarrow A'\}$ has a finitely failed definite tree then C_{i+1} is $pos(R, A)$, D_{i+1}^+ and D_{i+1}^- are respectively $D_i^+ \cup \{pos(R, A)\}$ and D_i^-, and G_{i+1} is $\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge B \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$. Again, R is marked as used possibly with respect to A.

2) a ground negative literal $\neg A$ and $D_i^+ \cup \{\leftarrow A\}$ has a finitely failed definite tree (introduced in the next section). In this case, θ_{i+1} is an identity substitution, C_i is $\neg A$, D_{i+1}^+ and D_{i+1}^- are respectively D_i^+ and $D_i^- \cup \{A\}$, and G_{i+1} is $\leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p$.

(b) If the sequence $G_0, G_1, \ldots$ of goals is finite, then the last goal G_n of the sequence is either empty or has the form $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_p$, where L_m is selected and

1) L_m is an atom and there is no program clause in D_n^+ for which any of the literals occurring in its head unifies with L_m, or

2) L_m is an atom and there are program clauses $\{R^1, \ldots, R^l\}$ in D_n^+ such that for each j, one of the atoms A occurring in the head H^j of R^j unifies with L_m. But, for each j, there exists a literal A' in D_n^- such that $D_n^+ \cup \{pos(R^j, A)\} \cup \{\leftarrow A'\}$ has a definite refutation (introduced in the next section), or

3) L_m is a ground negative literal of the form $\neg A$ and $D_n^+ \cup \{\leftarrow A\}$ has a definite refutation.

When a goal is refuted by a possible resolution with an answer substitution θ, it can be said that the goal bound with the answer substitution θ is true in the last database of the sequence of positive databases. The sequence of negative databases keep track of the ground negative atoms occurring in goals which have been inferred false by step a.2. Thus, a member of D_i^- is false in D_i^+, for every i. To clarify the definition, consider the following database:

Example 7.2 :

Database $D2$:

$$P(x) \leftarrow Q(x) \wedge \neg R(x) \wedge S(x)$$
$$R(a) \leftarrow S(a)$$
$$Q(a)$$
$$Q(b)$$
$$S(a) \vee S(b)$$

Query $Q2$ (possible resolution) :

$$\leftarrow P(x)$$

As each of the possible forms of $D2$ is hierarchical in nature, their completions are consistent.

Steps a.1.I and a.1.II formalise the SLD-resolution principle for definite clauses and hence in these two cases both positive and negative databases remain fixed. Step a.2 is a negation as failure which is concerned with keeping track of the inferred negative literal by adding it to the

latest negative database.

At the first step of execution of the query $Q2$ in the above example, the selected literal $P(x)$ is resolved with the first clause of $D2$ using ordinary SLD-resolution and producing $\leftarrow Q(x) \wedge \neg R(x) \wedge S(x)$ as the next goal. The next selected literal from this goal, $Q(x)$, unifies with $Q(a)$ yielding $\leftarrow \neg R(a) \wedge S(a)$. To execute step a.2 on the selected negative literal $\neg R(a)$, a refutation of the goal $\leftarrow R(a)$ is attempted in the constructed positive database (which is $D2$ itself). As this is not possible, $\neg R(a)$ is taken as true and the only literal remaining in the goal is $S(a)$. At this stage the positive and negative databases are respectively $D2$ and $\{R(a)\}$. To process the goal $\leftarrow S(a)$, it is possible to resolve it against the first atom $S(a)$ of the indefinite fact $S(a) \vee S(b)$. In this case, the positive database is updated by adding $S(a)$ to it, resulting in a definite refutation of the goal $\leftarrow R(a)$ in the updated positive database, where $R(a)$ has been taken from the latest negative database. The second clause of $D2$ is the main source of this refutation as it implicitly infers $R(a)$ from $S(a)$ which is inconsistent with $\neg R(a)$ which has already been inferred from the database. The resolution of the goal $\leftarrow Q(x) \wedge \neg R(x) \wedge S(x)$ with the fact $Q(b)$ will lead to an answer substitution to the answer substitution $\{x/b\}$ since the inclusion of $S(b)$ in the positive database does not infer implicitly $R(b)$ at any stage.

The process of marking clauses in steps a.1.III and a.1.IV is necessary construct different possible forms and also to avoid redundant answers. In the above example, when $S(a)$ is added to the positive database, it supersedes the indefinite fact $S(a) \vee S(b)$. In any subsequent use of $S(a)$, the definite fact $S(a)$ will be used rather than $S(a) \vee S(b)$.

Referring to the above example, consider the case when the literal $S(a)$ was selected from the goal $\leftarrow \neg R(a) \wedge S(a)$, instead of $\neg R(a)$. In that case, the indefinite fact $S(a) \vee S(b)$ is the obvious candidate for resolving with the goal, causing the addition of the fact $S(a)$ to the positive database. It is quite clear that inferring the only negative literal selected from the resolved goal $\leftarrow \neg R(a)$ is not possible in the updated database due to the presence of the rule $R(a) \leftarrow S(a)$ and the fact $S(a)$.

Definition : A possible derivation is *finite* if it consists of a finite sequence of goals; otherwise, it is *infinite*. A possible derivation is *successful* if it is finite and the last goal is the empty goal. A successful possible derivation is called a *possible refutation*. A possible derivation is *failed* if it is finite and the last goal is not the empty goal.

From the definition of possible derivation, a *possible tree* for $D \cup \{G\}$ via R is defined in the usual way where each branch of the possible tree corresponds to a possible derivation. A branch of a possible tree for which the terminating node is an empty goal is called a *success branch*, a branch which does not terminate is called an *infinite branch* and a branch for which the terminating node is other than an empty goal is called a *failure branch*. A possible tree for which every branch is a failure branch is a *finitely failed possible tree*.

If each possible form of a database D is hierarchical, then the possible resolution in D terminates, provided that, the definite resolution of step a.2 for resolving negative literals terminates.

7.4.2. Definite resolution

The definition of definite derivation can be adapted from the proof procedure SLGNF[68] or nH-Prolog [69, 93] (by extending the capability for inferring negative information). However, in this section, the definition of a definite derivation has been given using a theorem proving approach based on Robinson's resolution principle with a special rule for resolving any ground negative literal under a user defined predicate and occurring in the body of a clause. This has the advantage of simplicity of implementation.

First, the definitions of a clause and a goal have been generalised to allow a different kind of literal in their body.

Definition : An *extended clause* has the form (7.2), where each L_i is either a negated or unnegated atom or a *deferred literal* of the form $Def\ A$, where A is an atom. An *extended goal* has the form

$$\leftarrow L_1 \wedge \cdots \wedge L_n \qquad n \geq 1 \qquad (7.4)$$

where each L_i is either a literal or a deferred literal.

Definition : Let R be a clause of the form (7.2). Then a *definite form* of R with respect to A_p, $1 \leq p \leq m$, denoted by $def\ (R, A_p)$, is the extended clause

$$A_p \leftarrow L_1 \wedge \cdots \wedge L_n \wedge Def\ A_1 \wedge \cdots \wedge Def\ A_{p-1} \wedge Def\ A_{p+1} \wedge \cdots \wedge Def\ A_m .$$

According to the definition, the definite form of a definite clause is the clause itself.

Definition : Let R be a clause of the form (7.3). Then a *contrapositive form* of R with respect to M_r, $1 \leq r \leq p$, denoted by $con(R, M_r)$, is the extended clause

$$M_r \leftarrow M_1 \wedge \cdots \wedge M_{r-1} \wedge M_{r+1} \wedge \cdots \wedge M_p \wedge \neg N_1 \wedge \cdots \wedge \neg N_q \wedge Def\ A_1 \wedge \cdots \wedge Def\ A_m.$$

The above clause is not exactly in traditional contrapositive form as the conclusion of the clause remains M_r and not its negation.

According to the definition, contrapositive forms of facts as well as rules containing only negative literals in their bodies are not defined.

Definition : Let G be a goal of the form (7.4). Then a *contrapositive form* of G with respect to a positive (resp. deferred) literal L_r, $1 \leq r \leq n$, denoted by $con(G, L_r)$, is the extended clause

$$A \leftarrow L_1 \wedge \cdots \wedge L_{r-1} \wedge L_{r+1} \wedge \cdots \wedge L_n$$

where L_r is A (resp. $Def\ A$).

Definition : An *extended computation rule* is a function which maps an extended goal to either a literal or a deferred literal, called the *selected literal*, in that goal.

Definition : Let D be a database, G an extended goal, and R an extended computation rule. Without any loss of generality it can be assumed that the goal G has the form $\leftarrow M$, where M is an atom. A general goal can always be transformed to this form by using a transformation mechanism similar to that described in [67]. A *definite derivation* for $D \cup \{G\}$ via R consists of a sequence $G_0=G, G_1, G_2, \ldots$ of extended goals, a sequence $C_1, C_2, \ldots$ of variants (either ground negative literals or deferred literals or program clauses from the definite forms of D), and a sequence $\theta_1, \theta_2, \ldots$ of substitutions satisfying the following:

(a) For each i, G_i is $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_p$, the selected literal L_m in G_i is a

1) positive literal and L_m unifies with a

I. a definite fact A in D with an mgu θ_{i+1}. Then C_i is A and G_{i+1} is $\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$.

II. the head of a definite rule H←B in D with an mgu θ_{i+1}. Then C_i is H←B and G_{i+1} is $\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge B \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$.

III. an atom A_p occurring in an indefinite fact $F: A_1 \vee \cdots \vee A_p \vee \cdots \vee A_l$ of D with an mgu θ_{i+1}. Then C_{i+1} is $def(F, A_p)$ and G_{i+1} is
$\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p \wedge$
$Def\ A_1 \wedge \cdots \wedge Def\ A_{p-1} \wedge Def\ A_{p+1} \wedge \cdots \wedge Def\ A_p)\theta_{i+1}$.

IV. an atom A_p occurring in the head of an indefinite rule R:$A_1 \vee \cdots \vee A_p \vee \cdots \vee A_l \leftarrow B$ of D with an mgu θ_{i+1}. Then C_{i+1} is $def(R, A_p)$ and G_{i+1} is $\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge B \wedge L_{m+1} \wedge \cdots \wedge L_p \wedge$
$Def\ A_1 \wedge \cdots \wedge Def\ A_{p-1} \wedge Def\ A_{p+1} \wedge \cdots \wedge Def\ A_l)\theta_{i+1}$.

V. an atom A with an mgu θ, where $Def\ A$ (say, L_l') is occurring in $G_k\theta_1 \cdots \theta_i$ $(0 \le k \le i-1)$ and $G_k\theta_1 \cdots \theta_i$ has the form $\leftarrow L_1' \wedge \cdots \wedge L_q'$. Then C_{i+1} is $con(G_k, Def\ A)$ and G_{i+1} is

$\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge L_1' \wedge \cdots \wedge L_{l-1}' \wedge L_{l+1}' \wedge \cdots \wedge L_q' \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta$.

2) ground negative literal $\neg A$ and $D \cup \{\leftarrow A\}$ has finitely failed possible tree. In this case, θ_{i+1} is the identity substitution, C_i is $\neg A$, and G_{i+1} is $\leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p$.

3) deferred literal $Def\ A$ and

I. there is a rule R:$A_1 \vee \cdots \vee A_l \leftarrow B_1 \wedge \cdots \wedge B_p \wedge \cdots \wedge B_s$ such that A unifies with the atom B_p with an mgu θ_{i+1}. Then C_{i+1} is $con(R, B_p)$ and G_{i+1} is $\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge B_1 \wedge \cdots \wedge B_{p-1} \wedge B_{p+1} \wedge \cdots \wedge B_s \wedge$
$Def\ A_1 \wedge \cdots \wedge Def\ A_l \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$.

II. A unifies with a positive literal L_l' occurring in $G_k\theta_1 \cdots \theta_i$ $(0 \le k \le i-1)$ with an mgu θ and $G_k\theta_1 \cdots \theta_i$ has the form $\leftarrow L_1' \wedge \cdots \wedge L_q'$. Then C_{i+1} is $con(G_k, L_l')$ and G_{i+1} is

$\leftarrow(L_1 \wedge \cdots \wedge L_{m-1} \wedge L_1' \wedge \cdots \wedge L_{l-1}' \wedge L_{l+1}' \wedge \cdots \wedge L_q' \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta$.

(b) If the sequence of goals G_0, G_1, ... is finite, then the last goal G_n of the sequence is either empty or has the form $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_p$, where L_m is selected and

1) L_m is an atom and there is no program clause in D for which any of the literals occurring in the head unifies with L_m, or

2) L_m is a ground negative literal of the form $\neg A$ and there is a possible refutation of $D \cup \{\leftarrow A\}$, or

3) L_m is a deferred literal of the form $Def\ A$ and there is no program clause in D for which any of literals occurring in the body unifies with A.

To clarify the definition, consider the following database:

Example 7.3 :

Database $D3$:

$$P(x) \leftarrow Q(x) \wedge \neg R(x)$$
$$Q(a)$$
$$Q(x) \leftarrow S(x)$$
$$Q(x) \leftarrow T(x)$$
$$S(b) \vee T(b)$$
$$R(a) \vee R(c)$$

Query $Q3$ (definite resolution) :

$$\leftarrow P(x)$$

As each of the possible forms of $D3$ is hierarchical in nature, their completions are consistent.

As in the case of possible resolution, steps a.1.I and a.1.II formalise the SLD-resolution for definite clauses and step a.2 is a rule like negation as failure.

At the first attempt the goal $\leftarrow P(x)$ is resolved with the first clause of $D3$ producing as the next goal $\leftarrow Q(x) \wedge \neg R(x)$. The selected literal $Q(x)$ in this goal unifies with the fact $Q(a)$ leaving only the literal $\neg R(a)$ in the resolvent, i.e. in the next goal. To resolve the goal $\leftarrow R(a)$, it is required to produce a finitely failed possible tree for the goal $\leftarrow R(a)$ in the database $D3$. Due to the occurrence of $R(a)$ in the indefinite fact $R(a) \vee R(c)$, this goal has a possible refutation and the derivation fails.

In the next, the literal from the goal $\leftarrow Q(x) \wedge \neg R(x)$ unifies with the only head of the rule $Q(x) \leftarrow R(x)$. Resolving this goal using step a.1.II, the next goal in this direction is $\leftarrow S(x) \wedge \neg R(x)$. If $S(x)$ is selected from this goal, it unifies with $S(b)$ occurring in the indefinite fact $S(b) \vee T(b)$ to produce the next goal as $\leftarrow \neg R(b) \wedge Def\ T(b)$ using step a.1.III. Note that the literals other than $S(b)$ (in this case it is $T(b)$) in the head of the candidate clause have become deferred literals ($Def\ T(b)$) in the resolved goal. If $\neg R(b)$ is the selected literal, then since the goal $\leftarrow R(b)$ has a finitely failed possible tree in D 3, $Def\ T(b)$ remains the only (deferred) literal in the next goal. Step a.3.I states that, to resolve this goal, it is necessary to find a clause from whose body one of the atoms unifies with $T(b)$. The clause $Q(x) \leftarrow T(x)$ serves this purpose and the next goal can be inferred as $\leftarrow Def\ Q(b)$ using the rule a.3.I. In a similar manner, using the first clause, the next goal can be derived as $\leftarrow \neg R(b) \wedge Def\ P(b)$. The literal $\neg R(b)$ can be resolved in the same way as done previously in this example and the goal is now reduced to $\leftarrow Def\ P(b)$.

So far the only binding of a goal variable is $\{x/b\}$ and hence by applying a.3.II, the goal $\leftarrow Def\ P(b)$ can be resolved against the body of the original goal bound with $\{x/b\}$. Hence, there is a refutation of the original goal with an answer substitution $\{x/b\}$.

Definition : The terms *finite*, *infinite* and *successful definite derivations*, *definite refutation*, *definite tree*, *success*, *failure* and *infinite* branches of a definite tree, and *finitely failed definite tree* are defined in the usual way.

Definition : Let D be a database and G a goal. A *possible computation* (resp. *definite computation*) of $D \cup \{G\}$ is an attempt to construct a possible derivation *(resp. definite derivation) of* $D \cup \{G\}$.

A possible (or definite) computation of $D \cup \{G\}$ *flounders* if at some point in the computation a goal is reached which contains only non-ground negative or deferred literals.

The restriction of allowedness on a deductive database D is introduced in order to prevent floundering of any computation $D \cup \{G\}$, for some goal G. This notion of *allowedness* [63] (or *range-restriction* [25,55]) for definite databases can be extended to apply to indefinite databases too. When a goal G is allowed, no computation of $D \cup \{G\}$ flounders.

Definition : Let D be a database and G be a goal $\leftarrow$W. An *answer substitution* for $D \cup \{G\}$ is a substitution for the variables of G. A *possible computed answer* (resp. *definite computed answer*) θ for $D \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \cdots \theta_n$ to the

variables of G, where $\theta_1 \cdots \theta_n$ is the sequence of substitutions used in a possible-refutation (resp. definite refutation) of $D \cup \{G\}$.

When a goal G is allowed, every possible (or definite) computed answer for $D \cup \{G\}$ is a ground substitution for variables in G. Throughout the remainder of the paper, any further references to the term 'database' will be taken to refer to an allowed database.

A possible computed answer (resp. definite computed answer) for $D \cup \{G\}$ will also be referred to as a *possible answer substitution* (resp. *definite answer substitution*) for the possible-refutation (resp. definite-refutation) of $D \cup \{G\}$.

Unlike possible resolution, definite resolution in a database D may not terminate, even if each possible form of D is hierarchical. Consider the following example:

Example 7.4 :

Database $D4$:

$$P(x) \leftarrow Q(x) \wedge R(x)$$
$$Q(a) \vee R(a)$$

Query $Q4$ (definite resolution) :

$$\leftarrow P(x)$$

As each of the possible forms of $D4$ is hierarchical in nature, their completions are consistent.

Although each possible form of $D4$ is hierarchical, the query goes into an infinite loop. This kind of situation can be avoided by imposing the following condition on the definite resolution scheme. A goal G fails if it contains occurrences of an atom A and a deferred literal of the form $Def\ A$. In such a situation the goal is a tautology and cannot be refuted. Applying this rule in the above example, the goal $\leftarrow R(a) \wedge Def\ R(a)$ of the second step of the derivation fails.

In step 3 of the definite resolution scheme, selection of a non-ground deferred literal sometimes causes an infinite derivation. To clarify this, consider the following example:

Example 7.5 :

Database $D5$:

$$P(x,y) \leftarrow Q(x,y) \wedge R(x,y)$$

$$Q(x,y) \vee R(x,z) \leftarrow S(x,y,z)$$

$$S(a,b,b)$$

Query $Q5$ (definite resolution) :

$$\leftarrow P(x,y)$$

The database $D5$ is allowed. As each of the possible forms of $D5$ is hierarchical in nature, their completions are consistent.

The first step of execution will produce the goal $\leftarrow Q(x,y) \wedge R(x,y)$ and if the literal $Q(x,y)$ is selected from this goal, the following goal will be $\leftarrow S(x,y,z) \wedge Def\ R(x,z) \wedge R(x,y)$. The execution will go into an infinite loop if $Def\ R(x,z)$ is selected from this goal and only deferred literals are selected from the subsequent goals. Instead, if $S(x,y,z)$ is selected and the goal is resolved with $S(a,b,b)$, the next goal will be $\leftarrow Def\ R(a,b) \wedge R(a,b)$. This goal is a tautology and hence the derivation fails.

When the underlying database is allowed, a deferred literal $Def\ A$ (resp. a negative literal $\neg A$) occurring in a goal can be made ground at some stage of a derivation by selecting some positive literals before the selection of $Def\ A$ (resp. $\neg A$). For querying a database correctly, a negative literal is required to be ground before its selection.

Definition : A computation rule is *safe* if negative and deferred literals may only be selected for evaluation if they are ground.

7.4.3. Rule for inferring negative information

Based on the two resolution strategies introduced in the previous two sections, the rule (procedural semantics) for inferring negative information denoted by *negation as possible failure* (NAPF), is defined as follows. The negation of a fact A is taken as true in a database D if $D \cup \{\leftarrow A\}$ has a finitely failed possible tree.

The equivalence between the declarative and procedural concepts of negative information in an arbitrary database (at least when each possible form is hierarchical in nature) follows in the same way as in the case of definite databases [16]. The result in the case of definite

databases establishes the fact that SLDNF-resolution can infer those items of information which are declaratively given in terms of the completion of the database. In the context of indefinite databases, the possible and definite resolution schemes can infer items of information which are given in section 3 in terms of the completion of the possible forms of the database.

To illustrate the two resolution schemes together, consider the following example:

Example 7.6 :

Database $D6$:

Rules :

$R1.\ P(x) \leftarrow Q(x)$

$R2.\ P(x) \leftarrow R(x)$

$R3.\ Q(x) \vee R(x) \leftarrow S(x) \wedge \neg T(x)$

$R4.\ T(x) \leftarrow U(x) \wedge \neg V(x)$

$R5.\ V(b) \leftarrow U(b) \wedge W(b)$

Facts :

$F1.\ S(a)$

$F2.\ S(b)$

$F3.\ S(c)$

$F4.\ U(a) \vee V(d)$

$F5.\ U(b) \vee V(d)$

$F6.\ U(c)$

$F7.\ W(b)$

Query $Q6$ (definite resolution):

$\leftarrow P(x)$

As each of the possible forms of $D6$ is hierarchical in nature, their completions are consistent.

Figure 7.1 shows one possible partial search space for finding definite derivations of $D6 \cup \{\leftarrow P(x)\}$. In the search tree, branches indicated by solid lines represent branches of a definite tree whereas branches indicated by dashed lines represent a possible tree. A dotted line indicates a subgoal generation due to the selection of a negative or a deferred literal from a goal.

Selected clause and sometimes variable bindings are also shown beside the respective branch. NAPF represents an application of the negation as possible failure rule and NAF* represents negation as failure like a rule in the transformed database. For example, in the definite tree generated from the goal $\leftarrow V(b)$ the fact $U(b)$ has been taken as definite in the transformed database although its occurrence is indefinite in D 6. The reason is its use as a possible form of the indefinite fact $U(b) \vee V(d)$.

The definite resolution is an adaptation of linear resolution (including ancestor resolution) to incorporate negation as possible failure (NAPF) rule. Hence, by inheriting the completeness property of linear resolution, the definite resolution becomes complete provided the incorporated NAPF rule is complete. Also, its compatibility with the NAPF rule (i.e., negative facts inferred by the NAPF rule is consistent with the database) follows naturally because the inferred negative facts are consistent with every possible form of the database.

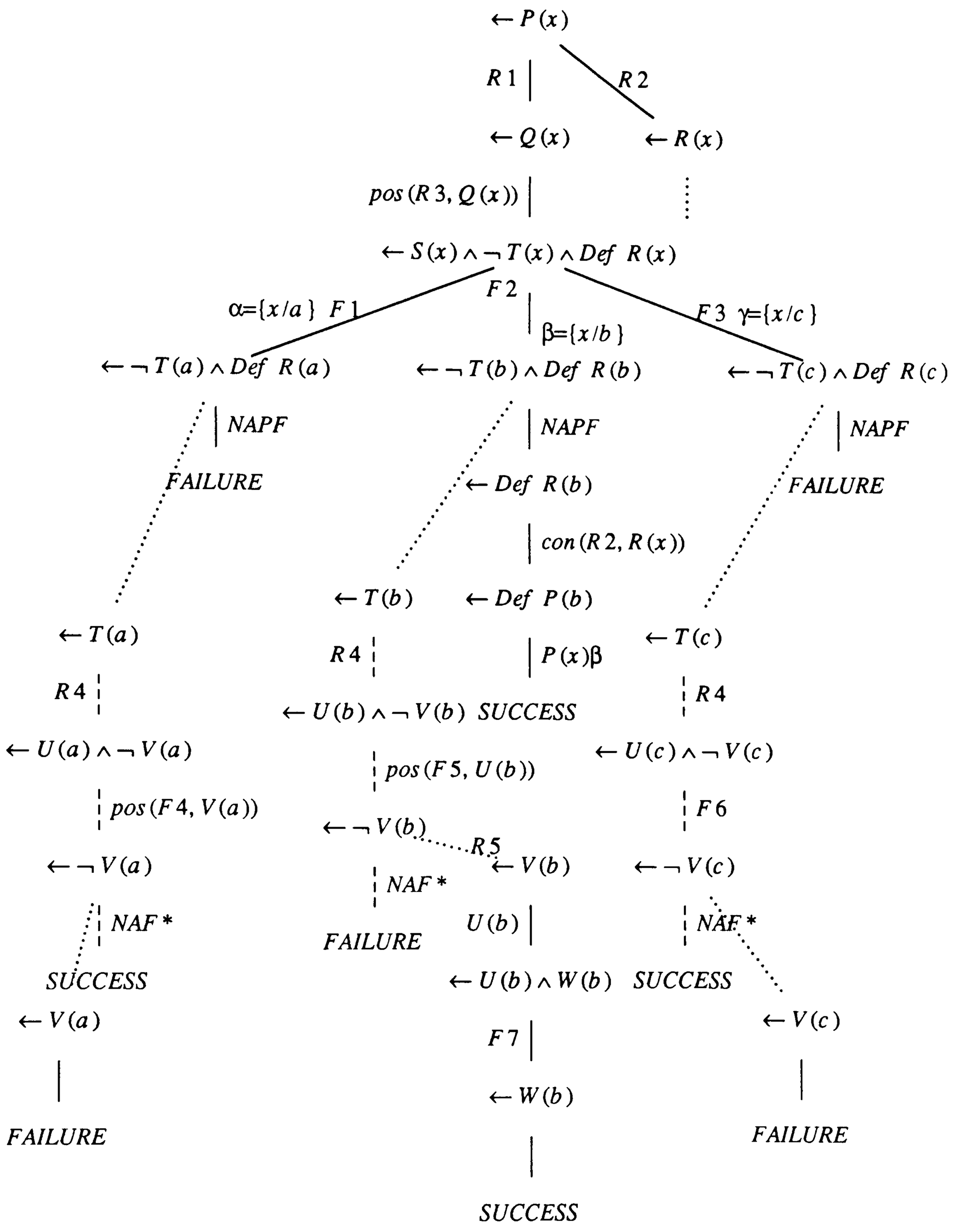

Figure 7.1. A Partial search space for the definite derivation of $D6 \cup \{\leftarrow P(x)\}$.

The definite derivation in the middle of $D6 \cup \{\leftarrow P(x)\}$ is given in table 7.1 and the possible derivation for $D6 \cup \{\leftarrow T(b)\}$ is given in table 7.2.

Goals	Variants	Substitutions
$G_0 = G\ (=M) = \leftarrow P(x)$		
$G_1 = \leftarrow Q(x)$	$C_1 = R1 = P(x) \leftarrow Q(x)$	$\theta_1 = \{\}$
$G_2 = \leftarrow S(x) \wedge \neg T(x) \wedge Def\ R(x)$	$C_2 = def\,(R3, Q(x)) =$ $Q(x) \leftarrow S(x) \wedge \neg T(x)$	$\theta_2 = \{\}$
$G_3 = \leftarrow \neg T(b) \wedge Def\ R(b)$	$C_3 = F2 = S(b)$	$\theta_3 = \{x/b\}(=\beta)$
$G_4 = \leftarrow Def\ R(b)$	$C_4 = \neg T(b)$	$\theta_4 = \{\}$
$G_5 = \leftarrow Def\ P(b)$	$C_5 = con\,(R2, R(x)) =$ $R(x) \leftarrow Def\ P(x)$	$\theta_5 = \{\}$
$G_6 = SUCCESS$	$C_6 = M\beta$	$\theta_6 = \{\}$

Table 7.1. A definite derivation for $D6 \cup \{\leftarrow P(x)\}$.

+ Databases	- Databases	Goals	Variants	Subst.
$D_0^+ = D6$	$D_0^- = \{\}$	$G_0 = \leftarrow T(b)$		
$D_1^+ = D_0^+$	$D_1^- = D_0^-$	$G_1 =$ $\leftarrow U(b) \wedge \neg V(b)$	$C_1 = R4 =$ $T(x) \leftarrow U(x) \wedge \neg V(x)$	$\theta_1 = \{\}$
$D_2^+ = D_1^+ \cup \{U(b)\}$	$D_2^- = D_1^-$	$G_2 = \leftarrow \neg V(b)$	$C_2 = pos\,(F5, U(b))$ $= U(b)$	$\theta_2 = \{\}$

Table 7.2. The possible derivation for $D6 \cup \{\leftarrow T(b)\}$.

7.5. Prolog implementation

In the implementation, clauses are transformed to a set of definite clauses in such a way that the transformed database is directly implementable in Prolog. In addition, in order to simulate the behaviour of definite resolution introduced in section 4 some contrapositives are maintained. In the implementation of definite resolution, only its input resolution [15] version is considered. First the following few notions are introduced.

Let A be an atom. Then $A^{S,Q,P,N,R}$ will represent the atom obtained from A by increasing the arguments of A in its first position by a list $[S, Q, P, N, R]$. If R is:

(a) An indefinite clause of the form (7.3), then an *expanded definite form* of R with respect to the i-th position atom A_i, denoted by $def(R^{S,Q,P,N,R}, A_i)$, $2 \leq i \leq m$, is given by

$$A_i^{+,Q,P,N,R} \leftarrow Control(Ref,i,Q,P,N,R) \wedge M_1^{+,Q,P,N,R} \wedge \cdots \wedge M_p^{+,Q,P,N,R} \wedge$$
$$Neg(N_1,Q,P,N) \wedge \cdots \wedge Neg(N_q,Q,P,N) \wedge$$
$$Deferred\_Def(Ref,i,(A_1 \wedge \cdots \wedge A_{i-1} \wedge A_{i+1} \wedge \cdots \wedge A_m),Q,P,R)$$

whereas the *expanded contrapositive form* of R with respect to M_i, denoted by $con(R^{S,Q,P,N,R}, M_i)$, $1 \leq i \leq p$, is given by

$$M_i^{-,def,P,N,R} \leftarrow M_1^{+,def,P,N,R} \wedge \cdots \wedge M_{i-1}^{+,def,P,N,R} \wedge M_{i+1}^{+,def,P,N,R} \wedge \cdots \wedge M_p^{+,def,P,N,R} \wedge$$
$$Neg(N_1,def,P,N) \wedge \cdots \wedge Neg(N_q,def,P,N) \wedge$$
$$Deferred\_Con(Ref,(A_1 \wedge \cdots \wedge A_m),P,R)$$

(b) A definite clause of form (7.3), given by

$$A \leftarrow M_1 \wedge \cdots \wedge M_p \wedge \neg N_1 \wedge \cdots \wedge \neg N_q$$

then an *expanded definite form* of R, denoted by $R^{S,Q,P,N,R}$, is given by

$$A^{+,Q,P,N,R} \leftarrow M_1^{+,Q,P,N,R} \wedge \cdots \wedge M_p^{+,Q,P,N,R} \wedge Neg(N_1,Q,P,N) \wedge \cdots \wedge Neg(N_q,Q,P,N)$$

In the implementation each definite rule is converted to its expanded definite form. For example, the expanded definite form of the first rule of database D 1 is

$$staff([+,Q,P,N,R],X) :- clerk([+,Q,P,N,R],X).$$

Each indefinite clause is converted to a set of clauses each of which is an expanded definite form with respect to an atom occurring in the head of the clause. For example, the set of expanded definite forms corresponding to the only indefinite rule in the database D 1 contains

$$clerk([+,Q,P,N,R],X) :- control(i\,1,p\,1,Q,P,N,R), employee([+,Q,P,N,R],X),$$
$$neg(officer(X),Q,P,N), deferred\_def(i\,1,p\,1,typist(X),Q,P,R),$$
$$typist([+,Q,P,N,R],X) :- control(i\,1,p\,2,Q,P,N,R), employee([+,Q,P,N,R],X),$$
$$neg(officer(X),Q,P,N), deferred\_def(i\,1,p\,2,clerk(X),Q,P,R).$$

Each rule is converted to a set of clauses each of which is an expanded contrapositive form with respect to an atom occurring in the body of the clause. The set of expanded contrapositive forms

of the only indefinite rule in the database $D1$ contains

$$employee([-,def,P,N,R],X):-neg(officer(X),def,P,N),$$
$$deferred\_con(i1,(clerk(X),typist(X)),P,R).$$

Also, to access all the definite facts in their expanded form a procedure of the form

$$p([+,\_,\_,\_,\_],X_1,...,X_n):-p(X_1,...,X_n).$$

is maintained for each user-defined n-ary relation p.

In the above representation the variable S is called the *side* of the literal and controls the selection of the correct two literals of opposite signs from the goal and the called clause. The variable Q, called the *mode* of the query, carries information about the kind of resolution (definite or possible) which is in progress. A side could be either '+' or '-' and a mode is either 'pos' representing possible resolution or 'def' representing definite resolution. Each of P and N is an open ended structure of the form

$$((...((X,C_1),C_2),...),C_n)$$

where X is variable and each C_i is a structure or an atom and will be described later in this section. This helps to keep information regarding the dynamic construction of a definite form of a database. All the literals which are still to be resolved to obtain a definite answer are accumulated in R through *deferred_con* and *deferred_def.*

The procedure *control* helps to construct a new database by marking some indefinite clauses in the database when a possible derivation is in progress. Under a possible derivation the indefinite clauses which have been used for resolution purposes are marked so that some forms of each can be treated as definite clauses for resolving negative literals in the subsequent part of the execution. Storing a fact *defcl(Ref,Posn)* into P means the definite form with respect to the head at a position *Posn* of the indefinite clause with identification *Ref* has been used. The three procedures *control*, *deferred_con* and *deferred_def* are defined as follows.

```
control(_,_,def,_,_).

control(Ref,Posn,pos,P,_):-
        in_clause(defcl(Ref,Posn),P),!.
```

```
control(Ref,Posn,pos,P,N):-
        cl(Ref,Hs,B),
        head(Posn,Hs,H),
        in_clause(NegL,N),
        link(NegL,H,[]),
        def(NegL,P,N),
        !,fail.

control(Ref,Posn,pos,P,_):-
        append_left(defcl(Ref,Posn),P).

deferred_con(Ref,Hs,P,R):-
        in_clause(defcl(Ref,Posn),P),
        head(Posn,Hs,H),
        add_goals(H,R).

deferred_con(Ref,Hs,P,R):-
        not in_clause(defcl(Ref,_),P),
        add_goals(Hs,R).

deferred_def(_,_,_,pos,_,_).

deferred_def(Ref,Posn,_,def,P,_):-
        in_clause(defcl(Ref,Posn),P),!.

deferred_def(_,_,RestHs,def,_,R):-
        add_goals(RestHs,R).
```

The procedure 'neg' which resolves the negative goal is defined as follows.

```
neg(G,pos,P,N):-
        not def(G,P,N),
        append_left(G,N).

neg(G,def,P,N):-
        not pos(G,P,N).
```

where *head(I,Hs,H)* means the atom in the I-th position of the disjunction of atoms *Hs* is *H*. Also, *append_left* is a procedure which appends its first argument to the open ended structure which is its second argument and is defined as follows.

```
append_left(X,Y):-
        var(Y),!,
        Y=(_,X).
append_left(X,(_,X)):-!.
append_left(X,(Y,_)):-
        append_left(X,Y).
```

The two mutually recursive top-level procedures *def* and *pos* implement the two resolution strategies in section 4. The goal *pos*(G,P,N) (*def*(G,P,N)) is satisfied if there is a possible (definite) refutation of the goal $\leftarrow G$ in the underlying database modified by P and N. They have been defined in the following way.

```
def((G,Gs),P,N):-!,
        def(G,P,N),
        def(Gs,P,N).

def(G,P,N):-
        expand(G,+,def,P,N,R,ExpG),
        call(ExpG),
        evaluate_deferred(R,G,P,N).

pos(Gs,P,N):-
        expand(Gs,+,pos,P,N,_,ExpGs),
        call(ExpGs).
```

where the value of *ExpG* in the procedure *expand(G,S,Q,P,N,R,ExpG)* is $G^{S,Q,P,N,R}$. The procedure *evaluate_deferred* evaluates all the accumulated heads in the open ended structure R. These are the same as either active or deferred heads as in nH-Prolog. The procedure *evaluate_deferred* is not allowed to backtrack as all the heads in R are ground and they do not create any new variable bindings. If all the heads of R are not resolved then the answer of the goal is indefinite with the unresolved heads.

```
evaluate_deferred(R,_,_,_):-
        var(R),!.

evaluate_deferred((R,Rs),G,P,N):-!,
        evaluate_deferred(R,G,P,N),
        evaluate_deferred(Rs,G,P,N).

evaluate_deferred(R,R,_,_):-!.

evaluate_deferred(R,_,_,N):-
        in_clause(R,N),!.

evaluate_deferred(R,G,P,N):-
        expand(R,-,def,P,N,Rev,ExpR),
        call(ExpR),
        evaluate_deferred(Rev,G,P,N).
```

CHAPTER 8

Integrity maintenance in indefinite databases

The problem of integrity maintenance in the context of relational databases has received much attention and more recently attention has been given to definite databases, which has been discussed in detail in chapter 6. The major problem is that the proposed methods for checking integrity are unable to check integrity in indefinite databases. To overcome this problem, this chapter proposes a generalisation of the path finding method for checking integrity in indefinite databases. The generalised method is based on a new definition of constraint satisfiability in indefinite databases. The syntactic property of this definition of constraint satisfiability is by the negation as possible failure rule. The Prolog implementation of the method and a meta interpreter of the extended nH-Prolog are described. The extended nH-Prolog infers negative information from an indefinite database using the idea of negation as possible failure. The query evaluator which is required for constraint evaluation operates on this extended nH-Prolog.

The organisation of the chapter is as follows. The following section describes the new definition of constraint satisfiability in indefinite databases. Section 8.2 presents the generalised path finding method and the following section provides its implementation in the extended nH-Prolog. In this chapter, the class of databases considered is indefinite and the constraints are closed first-order formulae. Unless otherwise mentioned, any usage of the term 'database' will be interpreted as 'indefinite database'.

8.1. Constraint satisfiability in indefinite databases

A possible form of an indefinite database has already been defined in chapter 7. The following declarative definition of constraint satisfiability in indefinite databases is given in terms of the possible forms associated with the database.

Definition : (Declarative definition of constraint satisfiability) A database D *is said to satisfy a constraint* C if C is a logical consequence of the completion of each of the possible forms of D. The database D is said to satisfy a set of constraints I if D satisfies each of the constraints of I; otherwise, D violates I.

Consider the following example of a database to illustrate the definition of constraint satisfiability.

Example 8.1 :

Database D 1 :

Rules :

$$Postgraduate(x) \leftarrow MSc(x)$$

$$Postgraduate(x) \leftarrow PhD(x)$$

$$Student(x) \leftarrow Undergraduate(x)$$

$$MSc(x) \vee PhD(x) \leftarrow Student(x) \wedge \neg Undergraduate(x)$$

Facts :

$$Student(Subrata)$$

$$Supervisor(Howard, Subrata)$$

$$Undergraduate(Choux) \vee MSc(Choux)$$

Integrity constraints I 1 :

$$\forall x\,(Postgraduate(x) \rightarrow \exists y\ Supervisor(y, x))$$

From the introduced definition of constraint satisfiability if one wants to impose the constraint I 1 on the database D 1 the integrity of the database would be violated as *Choux* can be proved to be a postgraduate student in one of the possible forms of D 1 without having a supervisor in that possible form.

For the method introduced in the next section for checking integrity in a database, each constraint is an allowed formula expressed in denial form. If a constraint W is not in denial form then it can be transformed to denial form by the same process as described in section 4.1.4. When the database D is indefinite, the constraint W is a logical consequence of the completion of each of the possible forms of D if and only if the transformed constraint $\leftarrow A$ is a logical

consequence of the completion of each of the possible forms of D_T. This is true because the difference between D_T and D is only a set of definite clauses and they are part of each and every possible form of D_T. This justifies the transformation of a constraint to a denial.

As in the definite case, the syntactic definition of constraint satisfiability, which is equivalent to the declarative definition, is as follows:

Definition : (Syntactic definition of constraint satisfiability) Let D be a database and $\leftarrow B$ a constraint in denial form.

1. If $D \cup \{\leftarrow B\}$ has a finitely failed possible tree, then $\leftarrow B$ is said to satisfy D

2. If $D \cup \{\leftarrow B\}$ has a possible refutation, then $\leftarrow B$ violates D.

8.2. The generalised path finding method

In the case of definite databases the path finding method computes a set of facts implicitly added to the database and a set of partially instantiated atoms whose instances are likely to be deleted from the database due to a transaction. These two sets of atoms are determined through the computation of all possible paths from update literals. If a success path is found then the integrity is said to be violated in the database. In the case of indefinite databases, the method has been generalised by determining the following two sets:

(a) the set of facts implicitly added to the database and possibly true in the updated database, and

(b) the set of atoms whose instances are likely to be deleted from the database.

These two sets are computed through all possible path calculations. The formal definition of a possible path is as follows.

Definition : Let D be a database. A *possible path* in D is defined as a chain of literals

$$L_0 \xrightarrow{R_1} L_1 \xrightarrow{R_2} \cdots \xrightarrow{R_n} L_n$$

where L_0 is called the *source* of the path, L_n its *destination*, n its *length* and R_1, ..., R_n are clauses from the union of the possible forms of D *used* to construct the path from L_0 to L_n. If the source L_0 is positive then it is ground. For any two consecutive literals L_i and L_{i+1}, L_{i+1} is called the

successor of L_i in the path, and is obtained from L_i in one of the following ways :

1. If

   (a) L_i is positive, and

   (b) L_i unifies with a positive literal L occurring in the body of the clause $R : A_1 \vee \cdots \vee A_m \leftarrow B$, and

   (c) α is an mgu of L_i and L, and

   (d) θ is a possible computed answer for $D \cup \{G\}$, where G is the goal $\leftarrow B\alpha$

   then L_{i+1} is the term $A_j\alpha\theta$, $1 \leq j \leq m$, and R_{i+1} is $pos(R, A_j)$.

2. If

   (a) L_i is positive, and

   (b) the negative literal $\neg L$ occurs in the body of the clause $R_i : A_1 \vee \cdots \vee A_m \leftarrow B$ such that L_i unifies with L, and

   (c) α is an mgu of L_i and L, and

   (d) $\neg A_j\alpha$, $1 \leq j \leq m$, is not an instance of any one of the L_k's, where $0 \leq k \leq i$

   then L_{i+1} is the term $\neg A_j\alpha$, $1 \leq j \leq m$, and R_{i+1} is $pos(R, A_j)$.

3. If

   (a) L_i is negative, and

   (b) L_i unifies with a negative literal L occurring in the body of the clause $R_i : A_1 \vee \cdots \vee A_m \leftarrow B$, and

   (c) α is an mgu of L_i and L, and

   (d) θ is a possible computed answer for $D \cup \{G\}$, where G is the goal $\leftarrow B\alpha$

   then L_{i+1} is the term $A_j\alpha\theta$, $1 \leq j \leq m$, and R_{i+1} is $pos(R, A_j)$.

4. If

(a) L_i is negative and has the form $\neg M$, and

(b) M unifies with a positive literal L occurring in the body of the clause $R_i : A_1 \vee \cdots \vee A_m \leftarrow B$, and

(c) θ is an mgu of M and L, and

(d) $\neg A_j\alpha$, $1 \leq j \leq m$, is not an instance of any one of the L_k's, where $0 \leq k \leq i$

then L_{i+1} is the term $\neg A_j\alpha$, $1 \leq j \leq m$, and R_{i+1} is $pos(R, A_j)$.

It is possible to construct more than one possible path starting from the same literal. Furthermore, a positive literal on a possible path is always ground and possibly true in the database. For simplicity, a path will sometimes be written without showing the clauses used to construct the path.

Definition : Let D be a database and L a literal such that if L is positive then it is ground. Let S be the set of all paths with L as the source. The *possible path space* rooted at the literal L is a tree defined as follows :

1. Each node of the tree is a literal.

2. The root node is L.

3. If N is a node of the tree, then the set of all successors of N in the paths of S are the only descendants of N in the tree.

Each branch of the path space corresponds to a path in the database with the root node as the source and vice-versa. A path which ends at the head *IC(No)* of a constraint will be called a *possible success path*; otherwise, it will be called a *failure path*. A path space containing at least one branch which corresponds to a success path is referred to as a *possible success path space*.

When the database D is definite, the definitions possible path, possible path space, possible success path, possible success path space reduces to path, path space, success path, success path space respectively.

As in the case of definite databases, to check integrity in an indefinite database when a new constraint is added, its equivalent denial form is queried directly against the database to make sure that the constraint is a logical consequence of the completion of each of the possible forms of the database. If a constraint is deleted then this cannot cause any inconsistency. To check integrity in the updated database D' as a result of the transaction t applied to D, the source is taken as an update literal. An *update literal* of the transaction t applied to a database D may be one of the following :

1. An atom occurring in a fact of t which is to be added to D.

2. The negation of an atom occurring in a fact of t which is to be deleted from D.

3. If an indefinite rule R:$A_1 \vee \cdots \vee A_m \leftarrow B$ in t is to be added to D then for a possible computed answer θ for $D' \cup \{\leftarrow B\}$, the corresponding instance of an atom occurring in the head of the rule R, i.e. $A_i\theta$, $1 \leq i \leq m$, which is possibly implicitly added to D due to the transaction.

4. If a rule R:$A_1 \vee \cdots \vee A_m \leftarrow B$ in t is to be deleted from D then the negation of an atom occurring in the head of the rule R, i.e. $\neg A_i$, $1 \leq i \leq m$, whose instances are likely to be deleted from D due to the transaction.

To preserve integrity one must ensure that a possible success path space does not exist with source as an update literal in the updated database. The different branches of the possible path space from an update literal can be generated in a number of ways and the backtracking mechanism is one of them.

A transaction is rejected if a possible success path is found in the updated database. Modification can be considered as being accomplished by a deletion followed by an addition.

The method is illustrated by means of the following few examples.

Example 8.2 :

Database $D2$:

Rules :

$R1.\ P(x,y) \leftarrow Q(x,y) \wedge R(x)$

$R2.\ Q(x,y) \leftarrow S(x,y) \wedge \neg M(x)$

$R3.\ R(x) \leftarrow U(x)$

$R4.\ R(x) \leftarrow V(x)$

$R5.\ U(x) \vee V(x) \leftarrow T(x) \wedge \neg W(x)$

$R6.\ M(x) \leftarrow R(x) \wedge N(x)$

Facts :

$N(a)$

$S(a, c)$

$S(b, c)$

Integrity constraints $I2$:

$IC1.\ IC(1) \leftarrow P(x, y) \wedge \neg O(x)$

Transaction :

insertfact $T(a) \vee T(b)$

As each of the possible forms of $D2$ is hierarchical in nature, their completions are consistent. Before the transaction is applied, database $D2$ satisfies the constraint $I2$. Let $D2'$ be the updated database.

As the transaction is the insertion of an indefinite fact $T(a) \vee T(b)$, the set of update literals is $\{T(a), T(b)\}$. Taking $T(a)$ as a source one can have, with the help of $R5$, either of $U(a)$ or $V(a)$ as the next literal of the source. Considering $U(a)$ first, the next literal of the path is $R(a)$. To find a successor of $R(a)$ with the help of $R1$, one has to find a possible computed answer of $D2 \cup \{\leftarrow Q(a, y) \wedge R(a)\}$. The fact $Q(a, c)$ is an instance of $Q(a, y)$ which is true in one of the possible forms of $D2$ provided that $M(a)$ is false in that possible form. $R(a)$ could be taken to be true in that possible form if $M(a)$ was not provable with the help of $R6$. Hence no instance of $Q(a, y) \wedge R(a)$ is true in the constructed database and $R1$ fails to generate a successor for $R(a)$. In the process of backtracking when V(a) is considered, the generated path also becomes failed for the same reason. Backtracking further and considering $T(b)$ as the source, one can construct the possible success path

$$T(b) \xrightarrow{pos(R5, U(x))} U(b) \xrightarrow{R3} R(b) \xrightarrow{R1} P(b, c) \xrightarrow{IC1} IC(1)$$

The two complete possible path spaces generated as a result of the transaction are shown in figure 8.1.

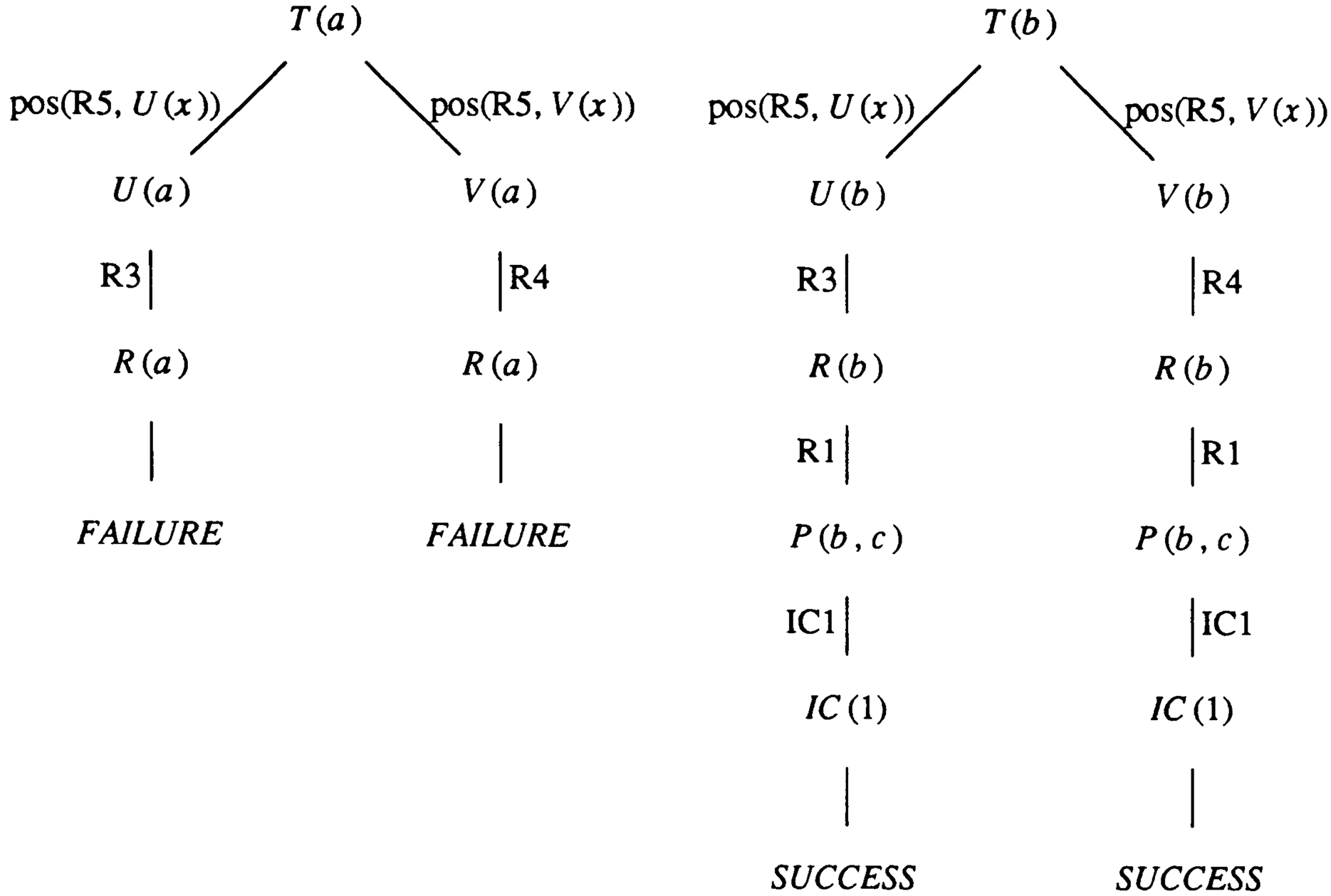

Figure 8.1. The two complete possible path spaces in the case of example 8.2

Example 8.3 :

Database $D3$:

Rules :

Same as database $D2$

Facts :

$T(a)$

$T(b)$

$O(b)$

$S(a, c)$

$S(b, c)$

$W(c) \vee W(a)$

Integrity constraints $I3$:

Same as example 8.2

Transaction :

deletefact $W(c) \vee W(a)$

Before the transaction is applied, database $D3$ satisfies the constraint $I3$. Since the transaction is the deletion of an indefinite fact the set of update literals is the set of all literals obtained by negating each atom occurring in the fact, i.e. $\{\neg W(c), \neg W(a)\}$. Taking $\neg W(c)$ as a source, the rule $R5$ fails to produce the next literal of the path as $T(c)$ is not even possibly true in the database. Taking $\neg W(a)$ as a source and applying rule $R5$, one can say that the next literal of the path is one of $U(a)$ or $V(a)$. Applying $R3$ to $U(a)$ yields $R(a)$ which unifies with a literal occurring in the body of $R1$ and $Q(a, c)$ is possibly true in the database. Hence, the successor of $R(a)$ is $P(a, c)$. Unifying $P(a, c)$ with a literal from the body of the denial form of the constraint, one can obtain $P(a, c) \wedge \neg O(a)$ and this is possibly true in the database. Consequently the integrity is violated in the database due to the transaction. The complete possible path space taking $\neg W(a)$ as a source is shown in figure 8.2.

Lloyd et al.'s simplification method can also be extended for checking integrity in indefinite databases, as in the case of definite databases, by calculating a set of partially instantiated atoms whose instances are likely to be deleted from different possible forms of the database.

8.3. Prolog implementation

This section describes how the integrity checking method described in the previous section and also the two resolution strategies described in chapter 7 may be implemented in Prolog.

A unique number has been assigned to each newly inserted constraint as an identification. A constraint $\leftarrow B$ numbered No is stored directly as a Prolog rule

$ic(No)$:–B .

A rule or a constraint can be retrieved efficiently by using a reference which uniquely identifies the corresponding rule or constraint.

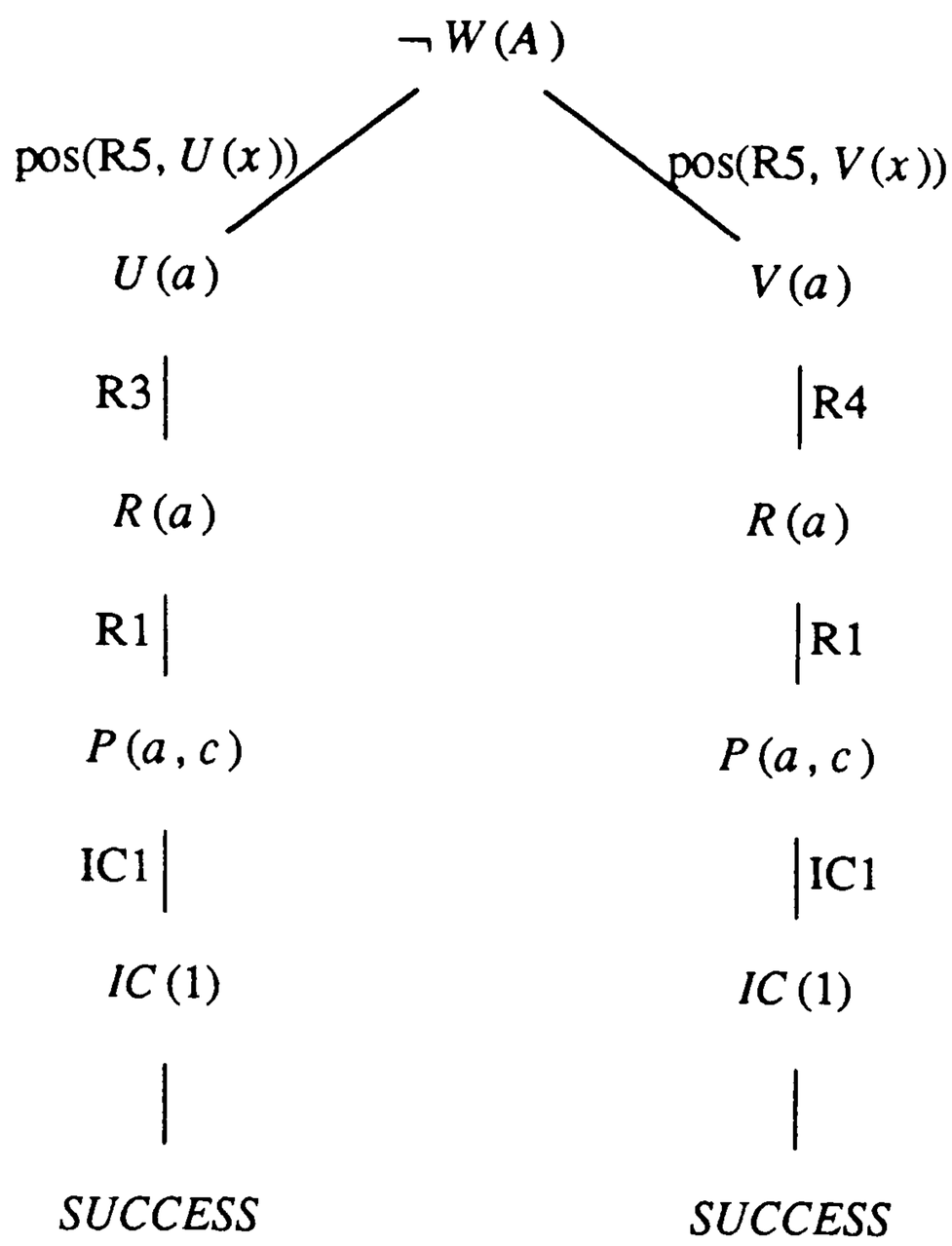

Figure 8.2. The complete possible path space in the case of example 8.3

The top level C-Prolog goal for integrity checking (?-*ic_violated(Tran)*.) can be defined as

```
ic_violated(Tran):-
    path(ic(No),Tran,[],P,N),!.
```

where *Tran* represents an update literal and *path* (*D* , *S* , *NegList* , *P* , *N*) means that a possible path is to be constructed from the literal *S* to the literal *D* and the set comprising the negation of each literal of the list *NegList* are the only negative literals which have occurred so far in this possible path construction. Each of P and N is an open-ended structure as described in the last chapter and achieves the same purpose for keeping information regarding the dynamic construction of a possible form of a database. With this convention if the database becomes inconsistent due to the transaction *Tran*, then the first inconsistency is reported at a constraint numbered *No* .

The possible path can be computed efficiently by maintaining additional information for each rule and constraint in the database. For each literal L occurring in the body of a rule $R: A_1 \vee \cdots \vee A_m \leftarrow B$ and for each i, $1 \leq i \leq m$, clauses of the form

$$depend(Ref, A_i, L)$$

$$depend(Ref, \neg A_i, CompL),$$

have been maintained, where *Ref* is the reference which uniquely identifies the clause R. For each literal L occurring in a constraint $\leftarrow B$, a clause of the form

$$depend(Ref, ic(No), L).$$

has been maintained, where *ic(No)* is the head of the constraint and *Ref* is the reference which uniquely identifies the clause $ic(No) \leftarrow B$.

The task of finding a possible path from a source literal S to a destination literal D can be achieved in Prolog with the help of the following meta-interpreter :

```
path(D,S,_,P,N):-
        depend(Ref,D,S),
        cl(Ref,Hs,B),
        simplify(B,S,SB),
        simplify(Hs,D,_),
        pos(SB,P,N).

path(D,S,NegList,P,N):-
        depend(_,Via,S),
        Via=(not A),
        not instance_list(A,NegList),
        path(D,Via,[A|NegList],P,N).

path(D,S,NegList,P,N):-
        depend(Ref,Via,S),
        not Via=(not _),
        cl(Ref,Hs,B),
        simplify(B,S,SB),
        simplify(Hs,Via,_),
        pos(SB,P,N),
        path(D,Via,NegList,P,N).
```

clause(H,B,Ref) means that H and B are unified respectively with the head and body of a clause which is uniquely identified by the reference *Ref*. The interpretation of *simplify(B,S,SB)* is that *SB*

is an instance of B obtained by unifying one of B's body literals with S. *instance(A,NegList)* means that A is not an instance of any of the atoms occurring in the list *NegList*. This corresponds to conditions 2(d) and 4(d) in the definition of a possible path and prevents a possible generation of an infinite possible path from a negative literal in the presence of recursive rules. The goal $pos(SB, P, N)$ is satisfied if there is a possible refutation of the goal SB in the underlying database modified by P and N. This procedure has already been defined in the implementation section of chapter 7. However, the following is an alternative implementation based on nH-Prolog [69, 93].

In the implementation of a query evaluation system, nH-Prolog has been selected and extended with the capability of deriving negative information from the database according to the semantics of negation as possible failure. In this case the format used for storing a definite clause [93] using the predicate *nH* is

$$nH(Head, Body)$$

whereas an indefinite clause is stored under the same predicate as

$$nH(ChosenHead, AuxiliaryHeads, Body, AncestorList).$$

Beside these a separate clause of the form

$$nH\_index(Functor, Arity, RefList)$$

has been maintained to store all the references to clauses containing an occurrence of a particular user-defined predicate in their heads, where the predicate is identified by *Functor* and *Arity*. In a database the number of clauses defining a predicate could be very high and in such a situation maintaining the above set of clauses would be both inefficient and wasteful of storage. Instead, definite clauses have been stored under the predicate *cl* as

$$cl(StartRef, NextRef, Goal, Body)$$

and an indefinite clause as

$$cl(IndRef, Posn, StartRef, NextRef, Goal, AuxHeads, Body, NewAnc),$$

where *StartRef* is the unique reference (1,2,... etc) to the clause among the set of clauses, each of which contains an occurrence of the predicate in its head defining *Goal*; *NextRef* is the same for the next clause defining P with the same arity and this is represented by a unique reference (zero)

when there is no next clause. For indefinite clauses the two parameters *IndRef* and *Posn* denote respectively a unique indefinite clause reference (i1,i2,... etc) and a unique position (p1,p2,... etc) of *Goal* in the head of the indefinite clause. Under the above convention the database *D* 1 can be represented as follows :

$cl(1, 2, postgraduate(X), msc(X))$

$cl(2, 0, postgraduate(X), phd(X))$

$cl(1, 2, student(X), undergraduate(X))$

$cl(2, 0, student(subrata), true)$

$cl(1, 0, supervisor(howard, subrata), true)$

$cl(i1, p1, 1, 2, msc(X), [dh(phd(X), Y)], (student(X), notundergraduate(X)), Y)$

$cl(i1, p2, 1, 0, phd(X), [dh(msc(X), Y)], (student(X), notundergraduate(X)), Y)$

$cl(i2, p1, 1, 0, undergraduate(choux), [dh(msc(choux), X)], true, X)$

$cl(i2, p2, 2, 0, msc(choux), [dh(undergraduate(choux), X)], true, X)$

The two mutually recursive top-level procedures *def* and *pos* corresponding to the two resolution strategies in section 2 have been defined in the following way.

```
def(Goal,P,N):-
        copy_query(Goal,OldGoal),
        def_init((X,Goal),OldGoal,P,N),
        var(X).

pos(Goal,_,_):-
        reserve(Goal),
        call(Goal).

pos(not Goal,P,N):-
        not def(Goal,P,N),
        append_left(Goal,N).

pos((A,B),P,N):-
        pos(A,P,N),
        pos(B,P,N).
```

```
pos(Goal,P,N):-
        cl(_,_,Goal,Body),
        pos(Body,P,N).

pos(Goal,P,N):-
        generator(Goal,Gen),
        cl(IndRef,Posn,_,_,Gen,_,Body,_),
        not in_clause(defcl(IndRef,Posn),P),
        in_clause(NegL,N),
        link(NegL,H,[]),
        def(NegL,P,N),
        !,fail.

pos(Goal,P,N):-
        cl(IndRef,Posn,_,_,Goal,_,Body,_),
        append_left(defcl(IndRef,Posn),P),
        pos(Body,P,N).
```

where the procedure *copy_query* (*Goal* , *OldGoal*) means *OldGoal* is a copy of the query *Goal* and *generator* (*Goal* , *Gen*) means the generator of the goal *Goal* is *Gen* . Also, *append_left* is a procedure which appends its first argument to its open-ended structure second argument and has already been defined in the implementation section of chapter 7.

The following four procedures *def_init*, *def_solve*, *def_restart* and *def_restart_goal* are respectively the modified forms of *nH_init*, *nH_solve*, *nH_restart* and *nH_restart_goal* in [93].

```
def_init(AnsList,OldGoal,P,N):-
        AnsList = (_,Goal),
        def_solve(Goal,1,[anc(q,[])],_,[],DH,init,P,N),
        ( DH = [_|_]
          -> def_restart(AnsList,OldGoal,DH,P,N);
          true).
```

```
def_solve(Goal,_,_,_,DH,DH,_,_,_):-
        reserve(Goal),
        call(Goal).

def_solve(not Goal,_,_,_,DH,DH,_,P,N):-
        not pos(Goal,P,N).

def_solve((A,B),_,Anc,AH,DH1,DH3,Can,P,N):-
        def_solve(A,1,Anc,AH,DH1,DH2,Can,P,N),
        def_solve(B,1,Anc,AH,DH2,DH3,Can,P,N).


def_solve(Goal,_,_,Goal,DH,DH,yes,_,_).

def_solve(Goal,StartRef,Anc,AH,DH1,DH3,Cancel,P,N):-
        get_cl(IndRef,Posn,StartRef,NextRef,Goal,AuxHeads,Body,NewAnc),
        NewAnc = [anc(Goal,NextRef)|Anc],
        ( (var(IndRef) ; in_clause(defcl(IndRef,Posn),P))
          ->    def_solve(Body,1,NewAnc,AH,DH1,DH3,Cancel,P,N);
          append(AuxHeads,DH1,DH2),
          def_solve(Body,1,NewAnc,AH,DH2,DH3,Cancel,P,N) ).

def_restart(AnsList,OldGoal,DH,P,N):-
        DH = [dh(AH,Anc)|DH1],
        member(anc(ResGoal,StartRef),[anc(not,_)|Anc]),
        def_restart_goal(AnsList,OldGoal,ResGoal,
                         StartRef,Anc,AH,DH1,DH2,Cancel,P,N),
        ( (DH2 = [_|_], Cancel == yes)
          -> def_restart(AnsList,OldGoal,DH2,P,N);
          (DH2 = [], Cancel == yes)
          -> true;
          fail).

def_restart_goal(_,_,not,_,_,AH,DH,DH,yes,_,N):-
        in_clause(AH,N).
```

```
def_restart_goal(_,_,Goal,StartRef,Anc,AH,DH1,DH2,Cancel,P,N):-
        def_solve(Goal,StartRef,Anc,AH,DH1,DH2,Cancel,P,N).

def_restart_goal(AnsList,OldGoal,q,_,Anc,AH,DH1,DH2,Cancel,P,N):-
        copy_query(OldGoal,Query),
        append_left(Query,AnsList),
        def_solve(Query,1,Anc,AH,DH1,DH2,Cancel,P,N).
```

The first seven parameters of the procedure def_solve are defined in the same way as in [93]. The last two parameters P and N are open-ended structures. Each element in P has the form

$$defcl(IndRef, Posn)$$

where the possible form of an indefinite clause identified by *IndRef* with respect to the head at the position *Posn* has been used for a possible resolution and the *def_solve* routine will consider those clauses as definite clauses. Each element in N is an atom representing a negative fact false in the database constructed dynamically.

In the new internal representation of clauses, a clause is retrieved with the help of the following routine *get_cl*

```
% No more clauses
get_cl(_,_,0,_,_,_,_,_):-!,fail.

% Retrieve a definite clause
get_cl(_,_,StartRef,NextRef,Goal,_,Body,_):-
        cl(StartRef,NextRef,Goal,Body).

% Retrieve an indefinite clause
get_cl(IndRef,Posn,StartRef,NextRef,Goal,AuxHeads,Body,NewAnc):-
        cl(IndRef,Posn,StartRef,NextRef,Goal,AuxHeads,Body,NewAnc).

% Retrieve the next clause
get_cl(IndRef,Posn,StartRef,NextRef,Goal,AuxHeads,Body,NewAnc):-
        generator(Goal,Gen),
        (cl(StartRef,Via,Gen,_);cl(_,_,StartRef,Via,Gen,_,_,_)),
        get_cl(IndRef,Posn,Via,NextRef,Goal,AuxHeads,Body,NewAnc).
```

CHAPTER 9

Formalising Aggregate Constraints

The proposed methods for integrity checking in deductive databases deal mainly with the class of static non-aggregate constraints and confining constraints to this form they exclude some important kinds of constraints from consideration. For example, constraints of the form

the total number of employees in a particular department may not exceed 100, or
a student obtaining an average mark greater than or equal to 50, passes the examination, etc.

cannot be expressed conveniently as closed first-order formulae and it is not possible to evaluate them efficiently even if they are expressed so. To take account of this problem extensions of Lloyd et al.'s simplification method as well as of the path finding method are proposed in the context of definite databases which allow a set of *aggregate predicates* (e.g., *Count*, *Sum*, *Maximum*) to be included within formulae representing constraints. The definition of constraint satisfiability has been generalised to include these constraints.

The organisation of this chapter is as follows. The following section introduces the notion of *closed general formulae* to cover a more general class of constraints, called *aggregate constraints*. Section 9.2 generalises the concept of constraint satisfiability in definite databases to include aggregates. The Simplification method by Lloyd et al. and the path finding method have been generalised in section 9.3 to check integrity in definite databases in presence of aggregate constraints. The last section discusses implementation issues in Prolog using the *setof* predicate. Repeated costly global compuations of the *setof* predicate can be replaced by more efficient incremental modifications [49, 5]. In this chapter, the class of databases considered is definite and hence any usage of the term 'database' will be interpreted as 'definite database'.

9.1. General formulae and aggregate constraints

Definition : In extending clauses to include aggregate predicates, the following five have been chosen as an initial set:

Count

Average

Sum

Maximum

Minimum

although this set can clearly be enlarged to include other such functions.

An *aggregate atom* is one of the following forms :

$$Count(W, n)$$

$$Average(x, W, r)$$

$$Sum(x, W, r)$$

$$Maximum(x, W, r)$$

$$Minimum(x, W, r)$$

where W is a conjunction of literals (a goal written without $\leftarrow$) in which some of the variables are free and the remaining variables are assumed to be existentially quantified in front of W. The variable x is one of the bound variables of W, and variables n and r represent the result of the function (as an integer or real number, respectively). The variable x can be replaced by an identifier (e.g., an integer) which will identify uniquely the occurrence of x in W. For the sake of readability, it has been considered as a variable. Thus $Count(W, n)$ is interpreted as counting the number of different answers to the query $\leftarrow W$ which are true in D, and returning this value as n; $Average(x, W, r)$ computes the average of all the values of the variable x obtained from different answers of the query $\leftarrow W$ which are true in D, and return this value as r; and so on. In each case the query $\leftarrow W$ is assumed to be allowed.

An *aggregate literal* is either an aggregate atom or the negation of an aggregate atom. A *general atom* (resp. *general literal*) is either an atom (resp. literal) or an aggregate atom (resp. aggregate literal). In each of the above aggregate atoms the formula W will be called the *key formula.*

Definition : A *general formula* may now be defined as follows :

(1) A general literal is a general formula.

(2) If A and B are general formulae, then so are $\neg$ A, A $\vee$ B, A $\wedge$ B, A$\rightarrow$B and A $\leftrightarrow$ B.

(3) If A is a general formula and x is a variable, then $\forall x$ A and $\exists x$ A are general formulae.

(4) Nothing else is a general formula.

Definition : Given a formula consisting of the single general atom α, then the general atom α is said to *occur positively* in the general formula α. If a general atom α occurs positively (resp. negatively) in a general formula A, then α *occurs positively* (resp. *negatively*) in $\exists x$ A and A $\wedge$ B and A $\vee$ B and B $\rightarrow$ A. If a general atom α occurs positively (resp. negatively) in a general formula A, then α *occurs negatively* (resp. *positively*) in $\neg$ A and A $\rightarrow$ B.

Definition : Let λ be an aggregate literal occurring in a general formula A. Then all the bound variables of the key formula W of λ are said to be *local* to the aggregate literal λ. Any other variables occurring in λ are said to be *global* variables of A.

Definition : A *closed general formula* Γ is a general formula with no free occurrences of any global variables.

Definition : A *first-order constraint* is a constraint, i.e. a closed first-order formula in the form of a denial.

Definition : An *aggregate constraint* is a closed general formula Γ of the form

$$\lambda_1 \wedge \cdots \wedge \lambda_n \rightarrow \alpha_1 \vee \cdots \vee \alpha_m$$

where $\alpha_1 \vee \cdots \vee \alpha_m$ is the *conclusion* of Γ and $\lambda_1 \wedge \cdots \wedge \lambda_n$ is the *condition*. Each α_i is a general atom and each λ_j is a general literal and at least one of α_1, ..., α_m, λ_1, ..., λ_n is an aggregate literal. Global variables occurring in the condition are assumed to be universally quantified at the front of Γ. Any variables in the conclusion which do not occur in the condition are assumed to be existentially quantified at the front of the conclusion. Thus the example of constraints given in the introduction can be expressed using this notation, as follows:

$$Dept(x) \wedge Count(Employee(y, x, z), t) \rightarrow t \leq 100,$$
$$Student(x, y, z) \wedge Average(v, Marks(x, u, v), t) \wedge t \geq 50 \rightarrow Pass(x),$$

where x, y, z, t, u, v are variables. According to the definition, the variables x and t in the first constraint are assumed to be universally quantified in front of the whole constraint, whereas the variables y and z are assumed to be existentially quantified in front of *Employee* (y, x, z).

Definition : A *constraint* is either a first-order constraint or an aggregate constraint.

In following two subsections, the only aggregate predicate considered will be the predicate *Count*. In subsection 9.4 this will be generalised to other aggregate predicates from the initial set.

9.2. Aggregate constraint satisfiability

Let D be a database and Q_D be the set of all queries. Consider the function $\chi_D : Q_D \rightarrow \mathrm{N}$, where N is the set of all non-negative integers, defined in the following way:

$$\text{For} \leftarrow W \in Q_D, \quad \chi_D(\leftarrow W) = card\ \{W\theta : comp(D) \models W\theta\},$$

where *card* X denotes the cardinality of the set X. Clearly the mapping is well defined, i.e. every query $\leftarrow W$ is mapped into a unique integer which is equal to the number of instances of W which can be derived from $comp(D)$. This asserts that the number of different instances of W which are true in D is $\chi_D(\leftarrow W)$. Thus if a constraint contains the aggregate atom *Count* (W, n) and all global variables occurring in W are instantiated, $\chi_D(\leftarrow W)$ represents n. In other words, $\chi_D(\leftarrow W)$ is the number of different computed answers of the SLDNF-derivation of $D \cup \{\leftarrow W\}$, provided that it is capable of returning all answers without going into an infinite loop.

Lemma 9.1 : *Let D be a database and $\leftarrow W$ a member of Q_D. Let $x_1, ..., x_n$ be all the variables of W and D_T be $D \cup \{\forall x_1 \cdots \forall x_n (P(x_1, \ldots, x_n) \leftarrow W)\}$, where P is a predicate symbol not occurring elsewhere in the database D. If $\chi_D : Q_D \rightarrow \mathrm{N}$ and $\chi_{D_T} : Q_{D_T} \rightarrow \mathrm{N}$ are the two mappings defined as above, then $\chi_D(\leftarrow W) = \chi_{D_T}(\leftarrow P(x_1, \ldots, x_n))$.*

Proof : This follows from the fact that $comp(D) \models W\theta$ if and only if $comp(D_T) \models P(x_1, ..., x_n)\theta$. ■

In view of the above lemma, the key formula of an aggregate atom under the predicate *Count* occurring in a constraint will be assumed to be an atom.

Definition : The *first-order equivalent of an aggregate atom* *Count* (A, n) is the first-order atom $Count_A(n, y_1, ..., y_g)$ (or simply $Count(n, y_1, ..., y_g)$), where n is a variable and $y_1, ..., y_g$ are the

arguments of A other than its local variables. The *first-order equivalent form of a general formula* Γ, is obtained from Γ by replacing its aggregate atoms with their first-order equivalent form.

Definition : Let D be a database and $Count(A, n)$ be an aggregate atom. Suppose, $z_1, ..., z_l$ are the local variables of A and $y_1, ..., y_g$ are the other attributes of A (some of which may be constants). One can always assume that A has the form $P(z_1, ..., z_l, y_1, ..., y_g)$, where P is a first-order order predicate. Suppose there are a finite number of constant terms $a_1, ..., a_m$ in the language underlying the database. In the context of database D, the *expansion* of $Count(A, n)$ defining its first-order equivalent, denoted by $Exp_D(Count(A, n))$ (or simply $Exp(Count(A, n))$, when D is clear from the context), is the following set of $m^g + m^l + \frac{m^l(m^l-1)}{2} + \frac{m^l(m^l-1)(m^l-2)}{3.2} + ... + 1$ first-order clauses:

m^g clauses:

$$Count(0, y_1, ..., y_g) \leftarrow y_1=a_1^1 \wedge ..., \wedge y_g=a_g^1 \wedge \neg Count(1, y_1, ..., y_g) \wedge \cdots \wedge \neg Count(m^l, y_1, \ldots, y_g)$$

where each a_i^1 is equal to a_p, for some p.

m^l clauses:

$$Count(1, y_1, ..., y_g) \leftarrow P(a_1^1, ..., a_l^1, y_1, ..., y_g) \wedge \neg Count(2, y_1, ..., y_g) \wedge \cdots \wedge \neg Count(m^l, y_1, ..., y_g)$$

where each a_i^1 is equal to a_p, for some p.

$\frac{m^l(m^l-1)}{2}$ clauses:

$$Count(2, y_1, ..., y_g) \leftarrow P(a_1^1, ..., a_l^1, y_1, ..., y_g) \wedge P(a_1^2, ..., a_l^2, y_1, ..., y_g) \wedge \neg Count(3, y_1, ..., y_g) \wedge \cdots \wedge \neg Count(m^l, y_1, ..., y_g)$$

where each a_i^k, k=1, 2, is equal to a_p, for some p, and

for $k \neq k'$, $1 \leq k, k' \leq 2$, there exists p such that $a_p^k \neq a_p^{k'}$

$\frac{m^l(m^l-1)(m^l-2)}{3.2}$ clauses:

$$Count(3, y_1, ..., y_g) \leftarrow P(a_1^1, ..., a_l^1, y_1, ..., y_g) \wedge \cdots P(a_1^3, ..., a_l^3, y_1, ..., y_g) \wedge \neg Count(4, y_1, ..., y_g) \wedge \cdots \wedge \neg Count(m^l, y_1, ..., y_g)$$

where each a_i^k, k=1, 2, 3 is equal to a_p, for some p, and

for $k \neq k'$, $1 \leq k, k' \leq 3$, there exists an p such that $a_p^k \neq a_p^{k'}$

.

.

.

1 clauses:

$$Count(m^l, y_1, ..., y_g) \leftarrow P(a_1^1, ..., a_l^1, y_1, ..., y_g) \wedge \cdots \wedge P(a_1^{m^l}, ..., a_l^{m^l}, y_1, ..., y_g)$$

where each a_i^k, $k=1, 2, ..., m^l$, is equal to a_p, for some p, and

for $k \neq k'$, $1 \leq k, k' \leq m^l$, there exists a p such that $a_p^k \neq a_p^{k'}$

When the above set of clauses has been added to the database, it would be said that the *database is expanded wrt the aggregate literal* $Count(A, n)$ and the expanded database is denoted as $Exp(D, Count(A, n))$.

Lemma 9.2 : *For all ground substitutions* θ *of the variables in* $\{y_1, ..., y_g\}$, *at most one of* $Count(1, y_1, ..., y_g)\theta$, ..., $Count(m^l, y_1, ..., y_g)\theta$ *can be derived at a time from* $comp(Exp(D, Count(A, n)))$.

Proof : Assume that both $Count(M, y_1, .., y_g)\theta$ and $Count(N, y_1, ..., y_g)\theta$ can be derived from $comp(Exp(D, Count(A, n)))$, for $N \neq M$. Without any loss of generality one can assume $0 \leq M < N \leq m^l$. Since $Count(M, y_1, ..., y_g)\theta$ is derivable from $comp(Exp(D, Count(A, n)))$, from the clauses defining $Count(M, y_1, ..., y_g)$ one can say that none of $Count(M+1, y_1, ..., y_g)\theta$, ..., $Count(N, y_1, ..., y_g)\theta$, ..., $Count(m^l, y_1, ..., y_g)\theta$ is derivable from $comp(Exp(D, Count(A, n)))$. This contradicts the initial assumption. Hence the lemma. ■

Lemma 9.3 : *For all ground substitutions* θ *of the variables in* $\{y_1, ..., y_g\}$, $\chi_D(\leftarrow A\theta) = N$, *if* $Count(N, y_1, ..., y_g)\theta$ *is derivable from* $comp(Exp(D, Count(A, n)))$.

Proof : Suppose that only M different instances of $A\theta$ are derivable from $comp(D)$ (i.e. $\chi_D(\leftarrow A\theta) = M$). The instances of $A\theta$ which are derivable from $comp(D)$, occur in the body of $U\theta$, where U is a clause in $Exp(D, Count(A, n))$. The head of U is $Count(M, y_1, ..., y_g)$ and $Count(M, y_1, ..., y_g)\theta$ is derivable from $comp(Exp(D, Count(A, n)))$. Hence, by Lemma 9.2, $M = N$.

Definition : Let D be a database and Γ an aggregate constraint, and suppose that $Count(A_1, n_1)$, ..., $Count(A_p, n_p)$ are the only aggregate literals occurring in Γ. The *expansion of* D wrt Γ, *denoted by* $Exp(D, \Gamma)$, is the set $D \cup Exp(Count(A_1, n_1)) \cup ... \cup Exp(Count(A_p, n_p))$. Let I be a set of constraints and Γ_1, ..., Γ_p be the only aggregate constraints in I. The *expansion of* D wrt I, *denoted by* $Exp(D, I)$, is the set $D \cup Exp(D, \Gamma_1) \cup ... \cup Exp(D, \Gamma_p)$

Lemma 9.4 : *Let D be a database and I be a set of constraints with occurrences of aggregate literals $Count(A_1, n_1)$, ..., $Count(A_p, n_p)$. If $comp(D)$ is consistent then $comp(Exp(D, I))$ is also consistent.*

Proof : For each integer n, suppose that $Count(n, y_1, ..., y_g)$ corresponding to $Count(P_i, n_i)$ for some $i=1, 2, ..., p$, is replaced by another atom $Countn_i(y_1, ..., y_g)$, where $Countn_i$ does not appear elsewhere in the database. Then the clauses added to D to obtain $comp(Exp(D, I))$ can be taken as satisfying the hierarchical constraint. Because of the hierarchical nature of the clauses added to D and the fact that the added clauses define predicates which are not already defined in D, $comp(D_T)$ is consistent. ■

Suppose Γ is an aggregate constraint and G is its first-order equivalent. In view of Lemma 9.3, D can be said to satisfy Γ if G is a logical consequence of $comp(Exp(D, \Gamma))$; otherwise D violates Γ. D is said to satisfy I, where I is a set of constraints, if D satisfies each constraint in I; otherwise D violates I.

An aggregate constraint Γ may be transformed to its *first-order equivalent denial form* $\leftarrow G_d$ by performing the following steps:

1. Each first-order atom A which occurs in the conclusion and which contains at least one existentially quantified variable of the conclusion, is replaced by $P(x_1, ..., x_n)$, where x_1, ..., x_n are the variables of A also occurring in the condition and the predicate symbol P does not occur elsewhere in the database. At the same time D is expanded by adding the clause $P(x_1, ..., x_n) \leftarrow A$ to it. Let D_T be the transformed database and G_T be the transformed aggregate constraint.

2. Convert G_T to its first-order equivalent form G_f.

3. G_f is transformed to its equivalent denial form $\leftarrow G_d$ by simply adding the negated form of each atom A occurring in the conclusion, as a conjunct of the condition.

When a first-order constraint $\leftarrow W$ in denial form is considered, the standard method of determining whether a database satisfies or violates the constraint is by evaluating the query $\leftarrow W$ in the context of the database. If there is an SLDNF-refutation of $D \cup \{\leftarrow W\}$ via a safe selection then D violates $\leftarrow W$. If $D \cup \{\leftarrow W\}$ has a finitely failed SLDNF-tree, then D satisfies W. When an aggregate constraint Γ is considered, it is transformed to its equivalent first-order denial form $\leftarrow G_d$ in the above way by transforming the database D to $Exp(D_T, \Gamma)$. Then one can have the following syntactic definition of aggregate constraint satisfiability of C.

Lemma 9.5 : *Let D be a database, Γ an aggregate constraint and R a safe selection rule. The terms D_T and $\leftarrow G_d$ are defined as above. Suppose $comp(D)$ is consistent. If there is an SLDNF-refutation of $Exp(D_T, \Gamma) \cup \{\leftarrow G_d\}$ via R then D violates Γ. If $Exp(D_T, \Gamma) \cup \{\leftarrow G_d\}$ has a finitely failed SLDNF-tree via R, then D satisfies Γ.*

Proof : Similar to the case of a first-order constraint. ■

9.3. The extended simplification theorem

Lloyd et al.'s simplification method for checking integrity in definite databases has already been described in chapter 5 and in chapter 6, and the four sets of partially instantiated atoms $Pos_{D'', D'}$, $Neg_{D'', D'}$, $Pos_{D'', D}$ and $Neg_{D'', D}$ have been defined there. The generalised simplification theorem to handle the case for an aggregate constraint is as follows.

Theorem 9.1: *Let D be a database and Γ an aggregate constraint such that D satisfies Γ. Suppose that t is a transaction consisting of a sequence of deletions followed by a sequence of insertions such that when the sequence of deletions is applied to D, it produces the intermediate database D'', and when the sequence of insertions is applied to this, it produces the database D'. Suppose that the completion of each of D and D' are consistent. If $x_1, \ldots, x_m$ are the global variables of Γ, the following three sets can be defined:*

$\Theta = \{\theta$: θ *is the restriction to* $x_1, \ldots, x_m$ *of an mgu of the atom* A, *where* $Count(A, n)$ *is an aggregate atom occurring in* Γ, *and an atom in* $Pos_{D'', D'} \cup Neg_{D'', D} \cup Neg_{D'', D'} \cup Pos_{D'', D}\}$,

$\Phi = \{\phi$: ϕ *is the restriction to* $x_1, \ldots, x_m$ *of an mgu of a first-order atom occurring negatively in* Γ *(i.e. in the condition of* Γ*) and an atom in* $Pos_{D'', D'} \cup Neg_{D'', D}\}$, *and*

$\Psi = \{\psi$: ψ *is the restriction to* $x_1, \ldots, x_m$ *of an mgu of a first-order atom occurring positively in* Γ

(i.e. in the conclusion of Γ*) and an atom in* $Neg_{D'',D'} \cup Pos_{D'',D}$*}.*

Then D' *satisfies* Γ *if and only if* D' *satisfies* $\Gamma\mu$*, for all* $\mu \in \Theta \cup \Phi \cup \Psi$.

Proof : Let E denote $Exp(D, \Gamma)$. If $comp(D)$ is consistent, then by Lemma 9.4, $comp(E)$ is also consistent. Suppose that the transaction t produces E' when it is applied to E and produces the intermediate database E''. Recall that $Count(n, y_1, ..., y_g)$ is the first-order equivalent of an aggregate atom $Count(A, n)$, where $y_1, ..., y_g$ are the attributes other than the bound variables of A. Suppose G is the first-order equivalent of Γ. Consider the following two sets:

Ω_1={ω : ω is the restriction to $y_1, ..., y_g$ of an mgu of an atom $Count(n, y_1, ..., y_g)$ occurring in the condition of G and an atom in $Pos_{E'',E'} \cup Neg_{E'',E}$}, and

Ω_2={ω : ω is the restriction to $y_1, ..., y_g$ of an mgu of an atom $Count(n, y_1, ..., y_g)$ occurring in the conclusion of G and an atom in $Neg_{E'',E'} \cup Pos_{E'',E}$}.

Suppose $\theta \in \Theta$ and $A' \in Pos_{D'',D'} \cup Neg_{D'',D}$ such that A' unifies with A, where A has the form $P(z_1, ..., z_l, y_1, ..., y_g)$. Each y_j is equal to x_k, for some k, $1 \le k \le m$. Whatever the bindings to the variables z_1, ..., z_l corresponding to θ, $Count(1, y_1, ..., y_g)\theta$, ..., $Count(m^l, y_1, ..., y_g)\theta$ are members of $Pos_{E'',E'} \cup Neg_{E'',E}$, and $Count(0, y_1, ..., y_g)\theta$, ..., $Count(m^l - 1, y_1, ..., y_g)\theta$ are members of $Neg_{E'',E'} \cup Pos_{E'',E}$.

Again, suppose $\omega \in \Omega_1$. This implies that $P(a_1^i, ..., a_l^i, y_1, ..., y_g)\omega$ is a member of $Pos_{E'',E'} \cup Neg_{E'',E}$, i.e. $P(a_1^i, ..., a_l^i, y_1, ..., y_g)\omega$ is a member of $Pos_{D'',D'} \cup Neg_{D'',D}$.

The above two paragraphs prove that, in the case of the addition of a fact, the satisfiability of $\Gamma\theta$ in D' is equivalent to the satisfiability of $G\omega$ in E. Similarly, in the case of deletion. Hence,

E' satisfies $G\omega$, for all $\omega \in \Omega_1 \cup \Omega_2 \cup \Phi \cup \Psi$ $\Leftrightarrow$ E' satisfies $G\mu$, for all $\mu \in \Theta \cup \Phi \cup \Psi$

Again,

D' satisfies Γ

$\Leftrightarrow$ E' satisfies G (by definition)

$\Leftrightarrow$ E' satisfies $G\omega$, for all $\omega \in \Omega_1 \cup \Omega_2 \cup \Phi \cup \Psi$ (by simplification theorem

$\Leftrightarrow$ E' satisfies $G\mu$, for all $\mu \in \Theta \cup \Phi \cup \Psi$ for first-order constraints)

$\Leftrightarrow$ D' satisfies $G\theta$, for all $\theta \in \Theta \cup \Phi \cup \Psi$. ■

The path finding method for definite databases can also be extended for checking integrity in definite databases in the presence of aggregate constraints as follows. Suppose, at the time of calculating a path P, a fact A' (or a negative literal $\neg A'$) occurring in it, unifies with the key formula A of an aggregate atom $Count(A, n)$ occurring in a constraint Γ. Let θ be an mgu of A and A'. Then, $\Gamma\theta'$ is evaluated in the updated database, where θ' is an unifier obtained from θ by excluding any bindings to the local variables of $Count(A, n)$. For efficiency purposes, all θ's are stored to avoid redundant evaluation of constraints.

9.4. Generalisation to other aggregate predicates

If an aggregate atom involving the predicate *Sum* occurs in a constraint, then the lemmas in the previous section as well as the above simplification theorem still hold since an occurrence of an aggregate atom of the form $Sum(z, A, r)$ in a constraint can be replaced by $Sum(r, y_1, ..., y_g)$ by adding the following clauses to the database (assuming that A has the form $P(z_1, ..., z_l, y_1, ..., y_g)$ and without any loss of generality z has been taken as z_1) :

1 clause :

$Sum(x, y_1, ..., y_g) \leftarrow Sum(\_, x, y_1, ..., y_g)$

$\frac{m^l(m^l-1) \cdots (m^l-k+1)}{k\ (k-1) \cdots 2\ 1}$ clauses, for each $k=1, ..., m^l$:

$Sum(k, x, y_1, ..., y_g) \leftarrow P(a_1^1, ..., a_l^1, y_1, ..., y_g) \wedge \cdots \wedge P(a_1^k, ..., a_l^k, y_1, ..., y_g) \wedge$

$x = a_1^1 + \cdots + a_1^k \wedge$

$\neg Sum(k+1, \_, y_1, ..., y_g) \wedge \cdots \wedge \neg Sum(m^l, \_, y_1, ..., y_g)$

where each a_i^p, $p=1, ..., k$, is equal to a_q, for some q, and

for $q \neq q'$, $1 \leq q, q' \leq k$, there exists an i such that $a_i^q \neq a_i^{q'}$

For an occurrence of the aggregate predicate *Maximum*, the set of clauses added is the following:

1 clause :

$Maximum(x, y_1, ..., y_g) \leftarrow Maximum(\_, x, y_1, ..., y_g)$

For each $k=1, ..., m^l$, $k \times \frac{m^l(m^l-1) \cdots (m^l-k+1)}{k\ (k-1) \cdots 2\ 1}$ clauses :

$Maximum(k, x, y_1, ..., y_g) \leftarrow P(a_1^1, ..., a_l^1, y_1, ..., y_g) \wedge \cdots \wedge P(a_1^k, ..., a_l^k, y_1, ..., y_g) \wedge$

$$a_i^j > a_1^1 \wedge \cdots \wedge a_i^j > a_i^{j-1} \wedge a_i^j > a_i^{j+1} \wedge \cdots \wedge a_i^j > a_i^k \wedge$$

$$\neg Maximum(k+1, \_, y_1, ..., y_g) \wedge \cdots \wedge \neg Maximum(m^l, \_, y_1, ..., y_g)$$

where each a_i^p, $p=1, ..., k$, is equal to a_q, for some q, and

for $q \neq q'$, $1 \leq q, q' \leq k$, there exists an i such that $a_i^q \neq a_i^{q'}$

The case of an aggregate atom involving the predicate *Minimum* is similar to the above case for *Maximum*. The aggregate predicate *Average* can be defined as an extension of the predicate *Sum*.

9.5. Prolog implementation

Let D be a database and Γ an aggregate constraint. Lemma 9.5 and theorem 9.1 states that the simplified first-order equivalent denial form of Γ would have to be evaluated in the updated database E' to preserve consistency, where E' is obtained by applying the transaction to $E (=Exp(D, \Gamma)$, assuming that the existentially quantified variables have already been resolved). However, in the Prolog implementation, the effect of the resolution in E' of a selected literal $Count(x, y_1, ..., y_g)$ from a goal which is the first-order equivalent of an aggregate literal $Count(P(z_1, ..., z_l, y_1, ..., y_g), n)$, can be achieved by evaluating the Prolog goal $?-setof([Z_1, ..., Z_l], p(Z_1, ..., Z_l, Y_1, ..., Y_g), N)$ in D'. Hence, one does not need to expand the underlying database D to E. In Prolog, the aggregate predicate $Count(A, N)$ is represented by $count(p([Z_1, ..., Z_l], [Y_1, ..., Y_g]), N)$, and defined as

```
count(P, N):-
    P=..[_|[Z|_]],
    setof(Z, P, ZL),
    length(ZL, N).
```

where the procedure *length* (ZL, N) returns the length of the list ZL into N. To get the correct intended meaning of an aggregate constraint Γ, it is necessary to follow a generalised computation rule, called *generalised safe*, which selects

1. a first-order negative literal, only when it is ground, and

2. an aggregate positive literal, only when its key formula is ground wrt to its free variables, and

3. an aggregate negative literal, only when it is ground wrt its global variables.

In Prolog's leftmost literal selection strategy, to get the effect of a generalised safe computation rule an aggregate literal λ (resp. negative literal L) can be placed after all the positive literals containing all the occurrences of global variables of λ (resp. variables of L). The other aggregate predicates can be implemented efficiently using a similar mechanism.

CHAPTER 10

Handling Transitional Constraints

This chapter deals with transitional (or dynamic) constraints in definite databases. Static constraints are concerned with a particular state of a database; by contrast, an imposed transitional constraint relates different states of a database. Integrity can be verified for transitional constraints in the same way as for static constraints, if the underlying language of the database is extended with some *action relations* (or *action predicates*) to represent transitional constraints. The expressible class of constraints using this extended language covers static as well as transitional constraints. Using this concept of action relation, a formalism has been developed by Nicolas and Yazdanian [80] in the context of relational databases to deal with transitional constraints.

No formalism or method has been proposed so far for deductive databases to deal with such constraints. In a deductive database an addition or deletion of a fact may cause a number of implicit additions or deletions of facts while an update may cause a number of *implicit updates* as well as implicit additions and deletions. An update in a deductive database is normally taken as a deletion followed by an addition for the purposes of integrity checking described so far but this approach will not be able to detect any implicit updates. Capturing all implicit updates as well as all implicit additions and deletions (not through updates) due to a transaction in deductive databases is an important issue for constraint checking in the presence of transitional constraints. This issue has been resolved in this chapter for the class of definite databases. Unless otherwise specified, in the rest of this chapter a constraint can either be static or transitional, and the term 'database' will be taken as definite database.

The organisation of this chapter as follows. The initial section defines formally the facts explicitly and implicitly added to a database due to a transaction. Section 10.2 introduces the concept of implicit updates with several examples. Section 10.4 gives the idea of different kinds of action relations. The last two sections present the generalised path finding method and

its implementation respectively.

10.1. Basic Concepts

Definition : Let D be a database, M be a ground literal and A be a ground atom. Then M is said to be *required to prove* A *by a rule* $R : H \leftarrow B$ in D (*and a literal* L occurring in B) if there exists a substitution θ such that the following conditions hold:

(a) $R\theta$ is ground, and

(b) A is $H\theta$, and

(c) $B\theta$ is true in D and M is $L\theta$.

The definition can be extended to its transitive closure by expanding the body of each rule in D. The variables common to both L and H are called *contributory variables* of L to the rule R. Consider the following example which clarifies this definition:

Example 10.1

Database D 1:

$$P(x, y) \leftarrow Q(x, y, z) \wedge \neg R(y, z)$$
$$Q(a, b, d)$$
$$Q(a, c, d)$$
$$R(c, d)$$

In the above example, $Q(a, b, d)$ (and also $\neg R(b, d)$) is required to prove $P(a, b)$ by the first rule and the literal $Q(x, y, z)$ (resp. $\neg R(y, z)$). The variables x and y are the contributory variables.

The concept of facts implicitly or explicitly added to or deleted from a database can formally be defined as follows.

Definition : Let D be a database and t a transaction whose application to D produces the updated database D'.

(a) If an operation of t is an addition of a fact A to D and A is not true in D, then A is said to be a *fact explicitly added* to D due to the transaction t.

(b) If an operation of t is a deletion of a fact A from D and A is present in D, then A is said to be a *fact explicitly deleted* from D due to the transaction t if A is not true in D'.

(c) A fact A is said to be *implicitly added* to D due to the transaction t if the following conditions hold:

1. A is not provable in D but provable in D', and
2. A is a not explicitly added to D.

(d) A fact A is said to be *implicitly deleted* from D due to the transaction t if the following conditions hold:

1. A is provable in D but not provable in D', and
2. A is a not explicitly deleted from D.

(e) The set of all facts which are either explicitly or implicitly added to D are said to be added to D due to the transaction t and will be denoted as $ADD_{D,D'}$.

(f) The set of all facts which are either explicitly or implicitly deleted from D are said to be deleted from D due to the transaction t and will be denoted as $DEL_{D,D'}$.

10.2. Implicit update

Definition : Let D be a database and t a transaction whose application to D produces the database D'.

(a) A fact A has been *updated explicitly* by another fact A' in D due to the transaction t if t contains an operation which is an update of A by A'.

(b) A fact F has been *updated positive-implicitly* by another fact F' in D due to the transaction t if the following conditions are met:

1. F' is a fact which is added to D and F is a fact deleted from D due to the transaction, and

2. a fact A has been updated either explicitly or implicitly by A', and

3. A (resp. A') is required to prove F (resp. F') by a rule $R: H \leftarrow B$ (resp. $R': H' \leftarrow B'$) in D (resp. D') and a literal L (resp. L') occurring in B (resp. B'), and

4. F' unifies with $H\{x_{i_1}/a_{i_1}, ..., x_{i_l}/a_{i_l}\}$, where $i_1, ..., i_l$ are the only attribute positions of A such that the attribute value at each position i_j of A and A' are equal and also each variable x_{i_j} is a contributory variable of the literal L to the rule R.

(c) A fact F has been *updated negative-implicitly* by another fact F' in D due to the transaction t if the following conditions are met:

1. F' is a fact which is added to D and F is a fact deleted from D due to the transaction, and

2. a fact A' has been updated either explicitly or implicitly by A, and

3. $\neg A$ (resp. $\neg A'$) is required to prove F (resp. F') by a rule $R: H \leftarrow B$ (resp. $R': H' \leftarrow B'$) in D (resp. D') and a literal L (resp. L') occurring in B (resp. B'), and

4. F unifies with $H'\{x_{i_1}/a_{i_1}, ..., x_{i_l}/a_{i_l}\}$, where $i_1, ..., i_l$ are the only attribute positions of A' such that the attribute value at each position i_j of A' and A are equal and also each variable x_{i_j} is a contributory variable of the literal L' to the rule R'.

(d) The set of all positive-implicit or negative-implicit updates due to a transaction is called the set of *implicit updates*.

(e) The set of all implicit or explicit updates due to a transaction is called the set of *updates* and is denoted by $UPD_{D,D'}$. Each element of $UPD_{D,D'}$ is a binary tuple of the form $<R(T), R(T')>$ and is read as '$R(T)$ has been updated by $R(T')$'. Each of T and T' is a string of terms separated from each other by commas, i.e an abbreviated form of the attributes of R.

The concept of implicit updates can be illustrated by the following series of examples :

Example 10.2

Database D 2:

$$Employee(x, y, z) \leftarrow EmpDept(x, y) \wedge EmpSal(x, z)$$

$$EmpDept(a, 50)$$

$$EmpSal(a, 10000)$$

Transaction:

update $EmpSal(a, 10000)$ by $EmpSal(a, 12000)$

where 50, 10000 are constants. Before the transaction is applied, the fact $Employee(a, 50, 10000)$ is true in D 2 but $Employee(a, 50, 12000)$ is not. The fact $EmpSal(a, 10000)$ (resp. $EmpSal(a, 12000)$) is required to prove $Employee(a, 50, 10000)$ (resp. $Employee(a, 50, 12000)$) in D 2 (resp. in the updated D 2) by the only rule of D 2 and the binding of the contributory variable x is $\{x/a\}$. The value at the attribute position x is a in both $EmpSal(a, 10000)$ and $EmpSal(a, 12000)$. Since $Employee(a, 50, 12000)$ unifies with $Employee(x, y, z)\{x/a\}$, it can be said that $Employee(a, 50, 10000)$ has been updated implicitly by $Employee(a, 50, 12000)$ due to the explicit update of $EmpSal(a, 10000)$ by $EmpSal(a, 12000)$.

Example 10.3

Database D 3:

$$P(x, y) \leftarrow Q(x, y) \wedge R(x, y)$$

$$P(x_1, y_1) \leftarrow Q(x_1, y_1) \wedge S(y_1)$$

$$P(x_2, y_2) \leftarrow T(x_2, y_2) \wedge Q(z_2, x_2)$$

$$Q(a, b)$$

$$R(a, b)$$

$$S(c)$$

$$T(c, d)$$

Transaction:

update $Q(a, b)$ by $Q(a, c)$

Before the transaction applied to the database, $P(a, b)$ is provable from the database but neither $P(a, c)$ nor $P(c, d)$. Also, $Q(a, b)$ is required to prove $P(a, b)$ in $D3$ by the first rule and $Q(a, c)$ is required to prove $P(a, c)$ in the updated database by the second rule. In the first case, the binding to the contributory variable x is $\{x/a\}$, and the arguments of the update candidates $Q(a, b)$ and $Q(a, c)$ matches only at the position of the contributory variable x of the literal $Q(x, y)$. Since $P(a, c)$ unifies $P(x, y)\{x/a\}$, $P(a, b)$ has been implicitly updated by $P(a, c)$. Although $Q(a, c)$ is required to prove $P(c, d)$ by the third rule in the updated database, $P(a, b)$ has not been updated implicitly by $P(c, d)$ as $P(c, d)$ fails to unify with $P(x, y)\{x/a\}$ (condition (b).4 in section 10.2).

The above example is a case of positive implicit update. The following example demonstrates a case of negative implicit update.

Example 10.4

Database $D4$:

$P(x, y) \leftarrow Q(x, y) \wedge \neg R(x, y)$

$Q(a, b)$

$Q(a, c)$

$R(a, b)$

Transaction:

update $R(a, b)$ by $R(a, c)$

Before the update is applied to the database, $P(a, c)$ is provable from the database but $P(a, b)$ is not. In the updated database, $P(a, b)$ is provable but $P(a, c)$ is not. In this situation, $P(a, c)$ has been implicitly updated by $P(a, b)$ due to the explicit update of $R(a, b)$ by $R(a, c)$.

In certain circumstances two different explicit updates may cause the same implicit update. Consider the following example:

Example 10.5

Database $D5$:

$P(x, y) \leftarrow Q(x) \wedge R(y)$

$Q(a)$

$R(b)$

Transaction:

update $Q(a)$ by $Q(a')$ and

update $R(b)$ by $R(b')$

Each of the updates in the transaction causes an implicit update of $Q(a, b)$ by $Q(a', b')$. Note that in each of the updates there is no contributory variable positions which are carrying the same values before and after the transaction.

The cases when an implicit update of a fact is by itself, have been discarded by imposing conditions (b).1 and (c).1 in section 10.2. Consider the following example:

Example 10.6

Database $D5$:

$P(x, y) \leftarrow Q(x, y, z) \wedge R(y, z)$

$Q(a, b, d)$

$Q(a, b, e)$

$Q(a, c, d)$

$Q(f, g, d)$

$Q(a, c, e)$

$R(b, d)$

Transaction:

1. update $R(b, d)$ by $R(b, e)$
2. update $R(b, d)$ by $R(g, d)$
3. update $R(b, d)$ by $R(c, d)$
4. update $R(b, d)$ by $R(b, e)$

The first explicit update does not cause any implicit update as $P(a, b)$ is not deleted from $D5$ due to the transaction. In the second case the implicit update is $P(a, b)$ by $P(f, g)$ and in each

of the last two updates, $P(a, b)$ has been implicitly updated by $P(a, c)$.

Definition : Let D be a database and t a transaction whose application to D produces the updated database D'. Consider the following four sets:

$$UPD\_ADD_{D,D'} = \{ R(T') : <R(T), R(T')> \in UPD_{D,D'} \}$$
$$UPD\_DEL_{D,D'} = \{ R(T) : <R(T), R(T')> \in UPD_{D,D'} \}$$
$$ADD'_{D,D'} = ADD_{D,D'} - UPD\_ADD_{D,D'}$$
$$DEL'_{D,D'} = DEL_{D,D'} - UPD\_DEL_{D,D'}$$

or, in other words

$$ADD_{D,D'} = UPD\_ADD_{D,D'} \cup ADD'_{D,D'}$$
$$DEL_{D,D'} = UPD\_DEL_{D,D'} \cup DEL'_{D,D'}$$

Each element of $UPD\_ADD_{D,D'}$ (resp. $UPD\_DEL_{D,D'}$) is called a *fact added* (resp.*deleted*) *due to update*. Also, each element of $ADD'_{D,D'}$ (resp. $DEL'_{D,D'}$) is called a *fact added* (resp.*deleted*) *not due to update*. The facts in $ADD'_{D,D'}$ (resp. $DEL'_{D,D'}$) are the set of all facts under all addition (resp. deletion) type action relations defined below, for the current transaction.

10.3. Action relations

The three different types of action relations, *addition type*, *deletion type* and *update type*, are explained in this section, and examples of constraints with occurrences of one or more of these types of relations are given.

Definition : Let D be a database and t a transaction whose application to D produces the updated database D'. Corresponding to each n-ary relation R in D add another three relation names, ADD_R (n-ary), DEL_R (n-ary) and UPD_R (2n-ary), into the language of the database D. The relation name ADD_R is called an *addition type action relation* corresponding to R, DEL_R is called a *deletion type action relation*, and UPD_R is called an *update type action relation* corresponding to R.

When a transaction t is performed on D to produce the updated database D', for constraint verification purposes the updated database is assumed to have been extended with the instances of the action relation given in the three sets $UPD_{D,D'}$, $ADD'_{D,D'}$ and $DEL'_{D,D'}$.

10.3.1. Addition type

A fact A under an action relation ADD_R is taken as true if A is added to the database not due to update, i.e. A is true if it is present in the set $ADD_{D,D'}$. A transitional constraint of the form

Initially, the loan amount is at least 1000

can be expressed in a first-order formula, using an addition type action relation *ADD_Loan* corresponding to the relation *Loan*, as

$$\forall x \forall y (ADD\_Loan(x, y) \rightarrow y \geq 1000)$$

10.3.2. Deletion type

A fact A under an action relation DEL_R is taken as true if A is deleted from the database not due to update, i.e. A is true if it is present in the set $DEL_{D,D'}$. Consider the transitional constraint

Lay-off of employees whose annual salary is less than 10000 is not permitted

Using deletion type action relation *DEL_Employee* corresponding to the relation *Employee* it can be represented as

$$\forall x \forall y \forall z (DEL\_Employee(x, y, z) \rightarrow z \geq 10000)$$

10.3.3. Update type

A fact $UPD\_R(T,T')$ under an action relation UPD_R is taken as true if $R(T)$ is updated by $R(T')$, i.e. $UPD\_R(T,T')$ is true if $<R(T), R(T')>$ is present in the set $UPD_{D,D'}$. For example, the transitional constraint

On updating, the salary of an employee should always increase

can be expressed, using an update type action relation *UPD_Employee* corresponding to the relation *Employee*, as a first-order formula

$$\forall x \forall y \forall y' \forall z \forall z' (UPD\_Employee(x, y, z, x, y', z') \wedge z \neq z' \rightarrow z' > z)$$

If the above constraint is imposed on database $D2$ of example 10.2, the integrity of the database will not be violated due to the transaction as the salary of the employee increases from 10000 to 12000 due the update. A transitional constraint which uses an update type action relation UPD_R is the variation of some arguments (say V) while others (say I) may stay invariant. If any other arguments of R other than those in V and I are left then they are irrelevant to the constraints. Based on the stated fact one can have the following relations: $I \subset X$, $V \subset X$, $I \cap V = \phi$, $I \neq \phi$, $V \neq \phi$, where X is the set of all arguments of R. In the relation $Employee(x, y, z)$ of the above example transitional constraint, the sets I and V are respectively $\{x\}$ and $\{z\}$.

10.3.4. Mixed type

There are transitional constraints which combine different action relations or impose a relationship between different actions. For example, the constraint

A change in grade must be accompanied by a change in income

is a relationship between two action relations *UPD_Income* and *UPD_Grade* corresponding to the relations *Income* and *Grade* respectively. This constraint can be represented in the extended language as

$$\forall x \forall y \forall y' \forall z \forall z' (UPD\_Grade(x, z, x, z') \wedge z \neq z' \rightarrow UPD\_Income(x, y, x, y') \wedge y \neq y')$$

10.4. The generalised path finding method

Let D be a database, I a set of constraints and t a transaction whose application to D produces the updated database D'. The generalised path finding method for checking integrity in the updated database D' in the presence of I is now divided into the following two steps:

(a) Apply the path finding method by taking the set of all constraints of I without any occurrences of action relations. If the integrity is violated then stop; otherwise continue.

(b) Compute the three sets $UPD_{D,D'}$, $\dot{ADD}_{D,D'}$ and $\dot{DEL}_{D,D'}$. The rest of the constraints (with occurrences of at least one action relation of any type) are instantiated appropriately with these three sets and then evaluated in the updated database.

10.5. Implementation

The calculation of the set $ADD_{D,D'}$ involves the calculation of a subset of the set of all facts added to the database D due to the transaction, and the calculation of the set $DEL_{D,D'}$ involves the calculation of a subset of the set of all facts deleted from the database D due to the transaction. By the time that step 2 of the algorithm is executed, the complete path space will have been generated from the update literals through the path finding method in step 1. Let PS be the set of all paths in such a case, and each of which has been collected during the application of the path finding method.

The set $UPD_{D,D'}$ can be computed, efficiently and without reasoning much with the original and updated databases, by using the literals on the paths of PS. Starting from explicit updates and travelling along different paths, pairs of literals representing possible implicit updates along with the rules used in the paths are computed. The irrelevant pairs can be discarded by querying the instantiated bodies of the rules used in the paths. This can be explained in short by considering the following simplified situation. Suppose that a fact A has been updated explicitly by A' and the next literal along a path with source A' is another fact M. A possible member of $UPD_{D,D'}$ is $UPD\_R(T,T')$, where A' is required to prove $R(T')$ and $R(T')$ satisfies other conditions. Hence, $R(T)$ can be obtained from the original database by evaluating some instantiated bodies of rules whose heads are under the predicate R. Once all partially instantiated pairs of literals representing the set $UPD_{D,D'}$ are computed, the relevant pairs of ground atoms can be obtained by querying in both D and D'. The two sets $UPD\_ADD_{D,D'}$ and $UPD\_DEL_{D,D'}$ can be computed using the set $UPD_{D,D'}$.

The set of all facts added to the database due to the transaction and relevant to the constraints can be obtained simply by collecting every positive literal occurring in every path of PS, and unifying it with an addition type action relation which occurs in a constraint but not in $UPD\_ADD_{D,D'}$.

The effect of the set of all facts deleted from the database due to the transaction and relevant to the constraints can be obtained by collecting every negative literal occurring in every path of PS, and unifying it with a deletion type action relation occurring in a constraint.

Before evaluating a constraint with an occurrence of at least one action relation, each of its action relations is unified appropriately and simultaneously by considering all facts of the three sets $UPD_{D,D'}$, $ADD_{D,D'}$ and $DEL_{D,D'}$ which are true in the updated database. In the

instantiated constraint, an occurrence of a positive (resp. negative) literal under an action relation is replaced by *true* (resp. *false*). The transformed constraint is evaluated in the updated database D'.

CHAPTER 11

Integrity Constraint Manipulation Language

The function of SQL is to support the definition, manipulation and control of data in relational databases. Imposing integrity constraints on a database is one of the ways of controlling data of the database. Not all types of constraints, discussed in this thesis, can be specified by the data manipulation language of standard SQL [21]. Extension is required to increase the capability of SQL in expressing such constraints. This chapter proposes an extension of the data manipulation language for integrity constraints in SQL.

The organisation of this chapter is as follows. The following section describes the BNF-syntax for specifying constraints in standard SQL. Section 11.2 describes the extended BNF-syntax summary. Section 11.3 presents an outline of the algorithm for converting constraints, expressed in the extended SQL-syntax, to logical formulae (either first-order or closed general). Different types of constraints are considered in the last section and their corresponding representation in extended SQL-syntax and in logical formulae are presented.

11.1. Constraints in SQL

Certain integrity constraints can be specified in standard SQL at the time of relation creation with the help of the CREATE TABLE command which has the following syntax:

CREATE TABLE <base-table> (<base-table-element-commalist>)

<base-table-element> ::= <column-def> | <table-constraint-def>

<column-def> ::=

<column> <data-type>

[DEFAULT <literal> | USER | NULL]

[<column-constraint-def-list>]

<column-constraint-def> ::=

NOT NULL [UNIQUE | PRIMARY KEY]

| CHECK (<search-condition>)

| REFERENCES <base-table> [(<column-commalist>)]

<table-constraint-def> ::=

{ UNIQUE | PRIMARY KEY } (<column-commalist>)

| CHECK (<search-condition>)

| FOREIGN KEY (<column-commalist>) REFERENCES <base-table> [(<column-commalist>)]

The undefined terms, e.g., <base-table>, <column>, have been elaborated in appendix 3.

Not all constraints discussed in this thesis can be expressed with the above syntax of SQL. Only null, check, unique, primary key, foreign key and reference constraints can be expressed with the above defined syntax. Extensions of SQL are required to express aggregate constraints, transitional constraints and more general forms of static constraints.

11.2. An SQL Grammar for Integrity Constraint Manipulation

An integrity constraint is inserted into and deleted from into a system using the following syntax:

<constraint-exp> ::=

IMPOSE CONSTRAINT <constraint-id> AS <constraint-spec>

| RELEASE CONSTRAINT <constraint-id>

A constraint can be specified as an equality or containment relationship between the results of two queries [104] (pp. 349-355). Since a query in SQL represents a table (i.e. a set) of tuples, a constraint has been expressed by a set inclusion relationship between the results of two SQL queries.

<constraint-spec> ::=

<query-exp> IS IN <query-exp>

| <query-exp> IS EMPTY

See appendix 3 for the syntax of <query-exp>. Intuitively, in the above constraint specification, the second alternative covers the constraints which are already in denial form. For the sake of users convenience, the first alternative has been specified to cover constraints which are more general forms than denials. Such a constraint can always be converted to its equivalent denial form by using transformations in section 4.1.4.

The above syntax of constraint specification can express static, transitional and aggregate constraints. The effect of the actions relations for representing transitional constraints, discussed in chapter 10, has been obtained by appending ON INSERTION, ON UPDATE etc. after the name of relevant relations. The effect of the second order predicates *Average*, *Sum*, *Count*, *Maximum* and *Minimum*, discussed in chapter 9, has been obtained respectively by using the functions AVG, SUM, COUNT, MAX and MIN of SQL.

11.3. Translating constraints to logical formulae

The translation of a constraint, expressed in the language described in the previous section, to its equivalent closed first-order (or closed general) logical formula can be performed mechanically. Guidelines for developing an algorithm to achieve this are given below:

(a) If <constraint-spec> has the form <query-exp> IS IN <query-exp>, then it is translated to *Condition* $\rightarrow$ *Conclusion*; otherwise, if <constraint-spec> has the form <query-exp> IS EMPTY, then it is translated to *Condition* $\rightarrow$, where *Condition* and *Conclusion* are either closed first-order or closed general formulae. The symbol '$\rightarrow$' denotes logical implication. The variables corresponding to the selected columns are common to both *Condition* and *Conclusion*, and they are assumed to be universally quantified in front of the whole formula *Condition* $\rightarrow$ *Conclusion*. Free variables in *Condition* (resp. *Conclusion*) occurring in first-order literals are assumed to be existentially quantified in front of *Condition* (resp. *Conclusion*).

(b) Simple queries of the form

SELECT <selection>
FROM <table-ref-commalist>
WHERE <search-condition>

can be translated directly to a conjunction of atoms (general atoms, in the case when one or more of the functions COUNT, SUM, AVG, MAX, MIN have been used in <selection>). New variables should be introduced if arithmetic expressions have been used in <selection>.

(c) The variables other than those which are common between a query and a subquery occurring within a query through <in-predicate> or <existence-test> are quantified existentially in front of the subquery. The whole logical formula corresponding to a subquery is negated in the case of a negative subquery.

(d) A new view is created in the database corresponding to each occurrence of a <group-by-clause>, where the column references in the view correspond to the column references occurring after GROUP BY in the <group-by-clause>. The predicate corresponding to the newly created view is placed along with the aggregate predicate in the logical formula form of the constraint. In examples 6-8, the view *Department* has been created because of the occurrence of GROUP BY EMP_DEPT in the query.

(e) In the case of a transitional constraint, the occurrence of the from R ON INSERT (resp. R ON DELETE, R ON UPDATE) in a <from-clause> will correspond to the action relation Ins_R (resp. Del_R, Upd_R) occurring in the translated constraint (example 10-14). For a column reference X, OLD X and NEW X are two distinct variables.

A closed general formula obtained by translating a constraint with the help of the above algorithm falls within the category of closed general formulae that have been handled both by the extended path finding method and the extended simplification method.

11.4. Examples

Examples of static non-aggregate(example 11.1), static aggregate (example 11.2) and transitional (example 11.3) constraints are considered in this section. Further examples of each of the above types can be found in appendix 4. The representation of each constraint in both SQL and closed first-order (closed general, in the case of aggregate constraints) formulae have been considered.

Example 11.1:

Type: *Static & Range*

Constraint:

An employee's salary must be less than 10000

Closed formula:

$$\forall x \forall y \forall z \, (Employee\,(x\,,\,y\,,\,z\,) \rightarrow z < 10000)$$

SQL Syntax:

```
IMPOSE CONSTRAINT SR AS
    (
        SELECT *
        FROM EMPLOYEE
        WHERE EMP_SAL >= 10000
    ) IS EMPTY
```

Example 11.2:

Type: *Aggregate on average*

Constraint:

Average salary of any department must be less than 10000

Closed general formula:

$$\forall x \forall y \, (Department\,(x\,) \wedge Average\,(v\,,\,\exists u \exists v \;\; Employee\,(u\,,\,x\,,\,v\,),\,y\,) \rightarrow y < 10000)$$

SQL Syntax:

```
IMPOSE CONSTRAINT AA AS
    (
        SELECT    DISTINCT EMP_DEPT
        FROM      EMPLOYEE
        GROUP     BY EMP_DEPT
        HAVING    AVG(EMP_SAL) >= 10000
    ) IS EMPTY
```

Example 11.3:

Type: *Transition on insertion*

Constraint:

Initially, the loan amount is at least 1000

Closed formula:

$$\forall x \forall y \, (Ins\_Loan\,(x\,,\,y\,) \rightarrow y \geq 1000)$$

SQL Syntax:

```
IMPOSE CONSTRAINT TI AS
    (
        SELECT    EMP_ID
        FROM      LOAN ON INSERT
        WHERE     LOAN_AMOUNT < 1000
    ) IS EMPTY
```

CHAPTER 12

Conclusion

This concluding chapter summarises the concepts and results described in the previous chapters. Deficiencies of some of the results and their possible improvements are also discussed in this chapter. Different directions are considered for continuing research in this field.

Both model and proof theoretic views of relational databases have been presented. Proof-theoretic views of definite databases have also been presented. A proof-theoretic view of indefinite databases remains to be studied.

Definitions of constraints and different views of constraint satisfiability have been given. A detailed classification of constraints has been proposed and examples have been provided for each of these classes. A detailed classification of constraints gives a better opportunity to study their behaviour within databases.

A method, called the path finding method, has been proposed for checking integrity in definite databases. The path finding method reasons forward from an update until it reaches the head of a constraint. The method derives in stages a set of fully instantiated atoms to be added to the database and a set of partially instantiated atoms likely to be deleted from the database. The approach is identical to that of Decker when there is no implicit deletion in the database. When an implicit deletion occurs in a stage, instead of deriving a set of atoms deleted from the database, the method follows a similar approach to that of Lloyd et al. and computes a set of partially instantiated atoms likely to be deleted from the database. The main reason for this is to avoid reasoning with two database states, i.e. before and after update. This contrasts with Decker's approach which reasons with two database states corresponding to times before and after update to exclude all induced facts which were provable in the database before update.

A comparative evaluation has been performed of several methods proposed for checking integrity in definite databases. To summarise the performance of the path finding

method against the methods of Lloyd et al. and Decker, the following four different cases have to be considered.

1. None of the constraints is simplified by an atom of *Pos*, where *Pos* is the set of partially instantiated atoms computed by the method of Lloyd et al. which represents the set of facts added to the database due to a transaction.

2. Some constraints can be simplified by the atoms of *Pos* but no instance of any one of the atoms of *Pos* is true in the updated database.

3. None of the constraints is simplified by an atom of *Neg*, where *Neg* is the set of partially instantiated atoms computed by the method of Lloyd et al. which represents the set of facts deleted from the database due to a transaction.

4. Some constraints can be simplified by the atoms of *Neg* but no instance of the complement of any one of the atoms of *Neg* is true in the updated database.

In the first case, the method of Lloyd et al. will have an advantage over both Decker's method and the path finding method as the latter methods will redundantly construct the set of facts which are implicitly added to the database. In the second case, the method of Lloyd et al. may suffer from a redundant constraint evaluation and the performance of both Decker's method and the path finding method will depend on the number of implicitly added facts. In the third case, both the method of Lloyd et al. and the path finding method will have an advantage over Decker's method as the latter will redundantly construct the set of facts which are implicitly deleted from the database. In the last case, both the method of Lloyd et al. and the path finding method may suffer from a redundant constraint evaluation and the performance of Decker's method in this case will depend on the computation of the number of implicitly deleted facts, performing more efficiently than the former two if the number of implicitly deleted facts is small, but less efficiently than the other two if it is not.

In the path finding method finding a path from an update literal to the head of a constraint is similar to the problem of finding a refutation with an update literal as a top clause in Kowalski's method. In the latter method finding a refutation means arriving at an empty head of the denial form of a constraint from an update literal. In the path finding method constructing a success path means reaching the head of a constraint from an update literal. The difference lies in the way in which this is achieved - in the method put forward by Kowalski et al. the computation of positive induced updates can be deferred by a proper literal selection strategy but the

computation of negative induced updates is essential, in the path finding method the computation of positive induced updates is essential and the computation of negative induced updates is deferred. Also, the literal selection strategy in Kowalski et al.'s method can play an important role in the efficiency of the method. Earlier selection of a literal whose instances are not true in the database may reduce the computation time. The method of Kowalski et al. models the literal selection strategy as well as the inference mechanism for forward reasoning. This may result in a loss of efficiency. On the other hand the path finding method relies mainly on the backward reasoning mechanism built into Prolog.

A new rule, referred to as *negation as possible failure*, has been introduced for inferring negative information from indefinite databases. To define the declarative semantics of negation as possible failure, an indefinite database has been transformed to a set of possible forms and the semantics is given in terms of the completion of each of these possible forms. The procedural semantics is based on two mutually recursive resolution schemes, the definite resolution scheme and the possible resolution scheme. The way in which the negation as possible failure rule infers negative facts demonstrates its conciseness, efficiency, consistency and inclusiveness [90]. Let R denote the above rule for inferring negative facts from the database and $R(D)$ be the set of negative facts that can be inferred from D by the application of R. Then R is

(a) *Concise*, i.e. the set of facts $R(D)$ is relatively large compared to D. In a typical database the number of facts which are true is much less than the whole Herband base associated with the database. In the case of the database $D3$ in chapter 7, there are 15 facts in the Herband base and the number of facts in $Def\_true(D)$, $Def\_false(D)$ and $Unknown(D)$ are respectively 3, 5 and 7.

(b) *Efficient*, i.e. the decision procedure of $R(D)$ should be relatively efficient. It has been shown in the implementation of the procedural semantics of R in section 5 that the truth value of a ground fact can be determined efficiently.

(c) *Consistent*, i.e. $D \cup R(D)$ should be consistent. This follows from the fact that a ground atom is taken as false if it is not true in any of the possible forms of D.

(d) *Inclusive*, i.e. R should interpret disjunction inclusively rather than exclusively. This is true because to obtain all possible forms of a database it is necessary to consider all possible combinations of possible forms of each of its indefinite clauses. For example, the possible forms of the database $\{P, P \vee Q\}$ are $\{P\}$ and $\{P, Q\}$. Hence

there is a world of the database where both P and Q are true.

The *null value* [48, 61, 62, 86, 108] problem can also be solved by the introduced semantics as follows. A null value can be regarded as a Skolem constant [35] representing a missing data item. For example, the fact P(ω) in a database D, where ω is a null value, can be represented by a formula of the form $\exists x P(x)$. If the domain of the database D consists only of $\{a_1, ..., a_n\}$, then the fact P(ω) can be represented by the formula $P(a_1) \vee \cdots \vee P(a_n)$. Thus a null value can be treated as an indefinite fact.

It has been shown that a query evaluation system for indefinite databases based on the introduced semantics can be implemented efficiently in Prolog using the definite resolution scheme. An extension of this implementation is required to represent null values and to process queries relating null values.

The path finding method has been generalised to check integrity in indefinite databases. The generalisation is based on a new definition of constraint satisfiability which, in turn, is based on the concept of negation as possible failure. This definition of constraint satisfiability in indefinite databases is a direct generalisation of the theoremhood view of constraint satisfiability in definite databases.

Analogous to the generalised theoremhood view, a generalised consistency view of constraint satisfiability in the case of indefinite databases can also be defined as follows. A database D is said to satisfy a set of constraints I if the set of constraints I together with the completion of the union of all possible form of D is consistent.

The definition of constraint satisfiability in indefinite databases which has been introduced here may seem restricted in the sense that each constraint has to be a theorem of the completion of each of the possible forms of the database. An alternative view, which is more relaxed, can be defined as follows. A set of constraints I is said to be satisfied by a database D, if there exists at least one possible form D_i of D such that each constraint in I is a theorem of $comp(D_i)$. However, this view has a drawback. Following this view, one can no longer use the imposed constraints for deriving new facts. The following example will demonstrate this problem.

Consider a database D which contains a single indefinite fact $Undergraduate(Choux) \vee MSc(Choux)$ and an imposed constraint on D in denial form is $\leftarrow MSc(Choux)$. The constraint states that *Choux* cannot be an *MSc* student. Following the

alternative relaxed view, the constraint does not violate D as the constraint is satisfied by a possible form of D, i.e. $\{Undergraduate(Choux)\}$. By using the constraint, the definite fact $Undergraduate(Choux)$ can be derived from the database; which means that fact $Undergraduate(Choux)$ is no longer indefinite, as it is in D.

Closed general formulae have been used to express aggregate constraints, i.e. constraints involving operations like count, average, maximum etc. The path finding method has been generalised to check integrity in databases in the presence of aggregate constraints. The proposed class of closed general formulae can express constraints which are not conveniently expressible by closed first-order formulae.

In the presence of transitional constraints in a database, the underlying language of the database has been extended with some action relations. Integrity checking in definite databases in the presence of transitional constraints is handled in the same way as in the presence of static constraints. The path finding method has been extended to compute facts which are in the action relations and evaluate constraints with occurrences of action relations. Transitional constraints have been defined by relating only two states of a database. Generalisation is required to deal with constraints relating more than two states.

The class of databases and constraints handled by the path finding method and its different possible generalisations are shown in table 12.1. The generalisations 1, 4 and 5 have been carried out respectively in chapters 8, 9 and 10. Regarding generalisation 2, an appropriate extension seems possible to maintain integrity in an environment, where both the database is indefinite and the imposed constraints are beyond first-order, by composing the two generalisations of the path finding method proposed in chapters 8 and 9 respectively. The generalised path finding method for handling aggregate constraints and the extended path finding method for handling transitional constraints can be combined together (i.e. generalisation 6) as the actions relations are like any other first-order relations in a database. Hence the generalised path finding method proposed for handling aggregate constraints will also be able to handle transitional constraints if an extra step is added to the generalised method for computing facts which are in the action relations and also for evaluating constraints with occurrences of action relations. To generalise the path finding method in indefinite databases and in the presence of transitional as well as aggregate constraints (i.e. generalisations 3 and 7), it is necessary to generalise the definition of implicit update in the context of indefinite databases.

Method	Databases		Constraints		
	Definite	Indefinite	Static	Aggregate	Transitional
Path finding	yes	no	yes	no	no
Generalisation 1	yes	yes	yes	no	no
Generalisation 2	yes	yes	yes	yes	no
Generalisation 3	yes	yes	yes	no	yes
Generalisation 4	yes	no	yes	yes	no
Generalisation 5	yes	no	yes	no	yes
Generalisation 6	yes	no	yes	yes	yes
Generalisation 7	yes	yes	yes	yes	yes

Table 12.1. The class of databases and constraints handled by the path finding method and its different possible generalisations

A typical implementation of a constraint verification program in a database system exploits the query evaluation system of the database. The constraint verification program takes the form of a meta-interpreter which calculates implicitly added or deleted facts, simplifies the constraints etc. In chapters dealing with constraints in definite databases implemented in the Prolog system, different Prolog meta-interpreters have been developed to serve the purpose of constraint verification programs. In chapter 8, meta-interpreter (extended nH-Prolog) has been developed for evaluating queries in indefinite databases and constraint verification program work on this meta-interpreter. In a database system, implemented in a conventional type of machine, each constraint is instantiated and evaluated, in turn. Here this has been done with the help of the backtracking mechanism, built into a Prolog system. When a set of constraints is imposed on a database system, their properties are independent from each other and hence, the evaluation of an instantiated set of constraints can be made simultaneously. In such cases the efficiency would be considerably higher.

In a parallel machine containing a number of processing elements, this concept of simultaneous evaluation of constraints can be described by considering the following example of hierarchical definite database (rules only) and a set of static constraints imposed on it.

Example 12.1

Database rules:

$$S(x) \leftarrow P(x) \wedge \neg Q(x)$$

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

$$U(x, y) \leftarrow V(x, y) \wedge \neg W(x)$$

Constraints in denial form:

$$IC(1) \leftarrow S(x) \wedge R(x)$$

$$IC(2) \leftarrow U(x, y) \wedge \neg S(y)$$

A *network*, given in figure 12.1, has to be established with the help of the rules and constraints. The network has a *source node* (the node in the bottom layer), a *sink node* (the node in the top layer), seven *predicate nodes* (nodes in the last but bottom layer), two *constraint nodes* (nodes in the next to top layer) and three *rule nodes*(the nodes in the third and the fourth layers). There are as many predicate (resp. constraint and rule) nodes as the number of relations (resp. constraints and rules) in the database. The source node distributes the set of facts, either (implicitly or explicitly) added to and deleted from the database due to the transaction, to the appropriate predicate nodes. The predicate node supplies a literal (positive, if it is an added fact, and negative, if it is a deleted fact) to the appropriate rule or constraint nodes. For example, the set of all added facts under the predicate V is sent to the rule node representing the rule $U(x, y) \leftarrow V(x, y) \wedge \neg W(x)$, as this may implicitly add some facts under the predicate U and they, in turn, may violate constraints with the help of the constraint identified by 2. The set of all deleted facts under the predicate V is not sent to the same rule node, although they may delete implicitly some facts under the predicate U and eventually will not cause any constraint violation. The constraint violation is reported when the control of execution reaches the sink node.

In the actual implementation each node of the network would be considered as a processing element. Each rule (resp. constraint) node would be capable of evaluating an instantiated body of a rule (resp. constraint). The produced literals (ground, in the case of implicit additions) are streamed to the appropriate processing elements.

The penultimate chapter proposes an extension of SQL to increase its constraint expression capability. Some guidelines have been given to develop an algorithm for translating

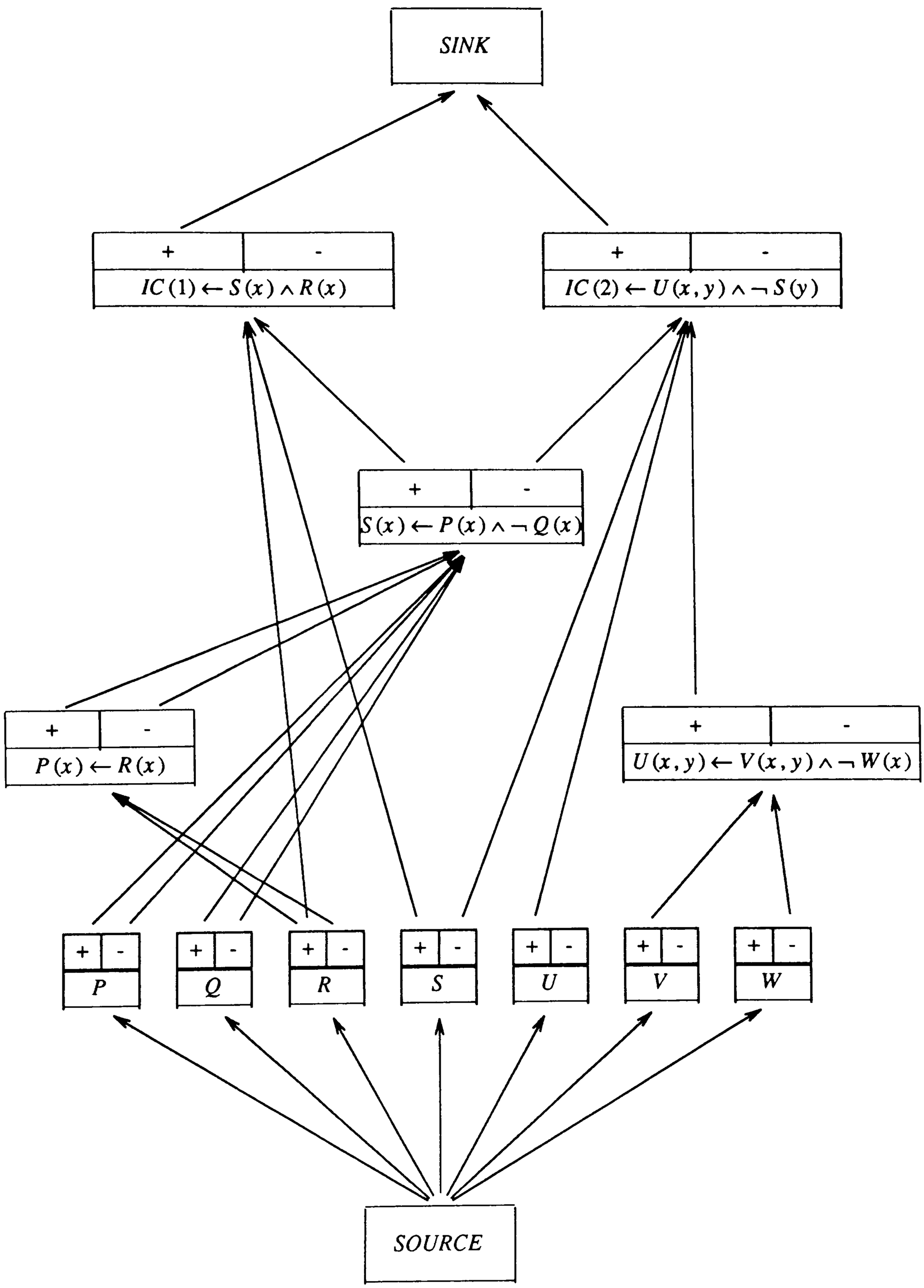

Figure 12.1. The network for constraint evaluation in the case of example 12.1.

constraints expressed in the extended SQL to their equivalent logical formulae. Irrespective of the type of database and its query evaluator, the constraints expressed in the intermediate logical formulae form will ease the simplification process of constraints and some optimisations.

A substantial theory of integrity constraints has been summarised and developed in this thesis. Some interesting unsolved problems are listed below for continuing research in this field.

(a) A comparative evaluation of the two views (i.e., whether the set of constraints is satisfied in at least one possible form of the database or in every possible form) of constraint satisfiability in indefinite databases.

(b) A syntax for a more general class of formulae to express constraints like

the maximum of the average salary of the departments must be between 10000 and 15000

no more than two tests may have an average mark of less than 50

For example, one possible representation of the above constraints in a further general class of formulae might be

$$\forall u\,(Maximum\,(t\,,\,\exists t\,\exists y\,(Department\,(y\,)\,\wedge$$
$$Average\,(z\,,\,\exists x\,\exists z\;\;Employee\,(x\,,\,y\,,\,z\,),\,t\,)),\,u\,)\rightarrow u\geq 10000\,\wedge\,u\leq 15000)$$

$$\forall x\,\forall u\,\forall v\,(Student\,(x\,)\,\wedge\,Count\,(\exists y\;\;(ExamType\,(y\,)\,\wedge$$
$$Average\,(t\,,\,\exists t\;\;ExamMarks\,(x\,,\,y\,,\,z\,,\,t\,),\,u\,),\,u<50),\,v\,)\rightarrow v\leq 2)$$

The path finding method should also be generalised accordingly.

(c) An efficient method for computing the three sets $UPD_{D,D'}$, $\dot{ADD}_{D,D'}$ and $\dot{DEL}_{D,D'}$ mentioned in section 10.4.

(d) A formal algorithm for translating constraints, expressed in the language described in section 11.2, to their equivalent closed first-order (or closed general) logical formulae.

(e) A generalisation of the path finding method in indefinite databases and in the presence of transitional as well as aggregate constraints.

(f) The soundness and completeness result of the negation as possible failure rule with respect to the declarative semantics given in section 7.3.

APPENDIX 1

Relation Schemas

The appendix describes the detail of all the relation variables used throughout the thesis. For each unary relation of the form **P**(x) will mean, 'x has the property **P**', e.g., **Student**(x) means 'x is a student'. Other relations are as follows:

Age(Person, Age)

Ancestor(Ancestor, Descendent)

Award(Student, Paper Awarded On)

Budget(Department, Budget Amount)

CivilStatus(Person, Age, Sex, Occupation)

Dependent(Dependent, Provider)

EmpDept(Employee, Department)

EmpSal(Employee, Salary)

Employee(Employee, Department, Salary)

ExamMarks(Student, Exam Type, Paper, Marks)

Father(Father, Child)

Grade(Employee, Grade)

Guardian(Guardian, Subordinate)

Husband(Husband, Wife)

Income(Employee, Income)

Loan(Employee, Amount)

Manager(Manager, Subordinate)

Marks(Student, Paper, Marks Obtained)

Married(Spouse, Spouse)

Mother(Mother, Child)

Occupation(Person, Occupation)

Parent(Parent, Child)

Sex(Person, Sex)

Sponsor(Sponsorer, Beneficiary)

Tax(Tax Payer, Tax Paid For)

Wife(Wife, Husband)

APPENDIX 2

Example Database

The following is a summary of the contents of the three states of the database, integrity constraints and transaction used in example 6.5.

Database $D5$:

Rules :

$$Age(x, y) \leftarrow CivilStatus(x, y, p, q)$$

$$Sex(x, y) \leftarrow CivilStatus(x, p, y, q)$$

$$Occupation(x, y) \leftarrow CivilStatus(x, p, q, y)$$

$$Mother(x, y) \leftarrow Father(z, y) \wedge Husband(z, x)$$

$$Parent(x, y) \leftarrow Father(x, y)$$

$$Parent(x, y) \leftarrow Mother(x, y)$$

$$Dependent(x, y) \leftarrow Parent(y, x) \wedge Occupation(y, Service) \wedge Occupation(x, Student)$$

Facts :

The following facts not involving 1, 2, 3 in the first attribute domain of *CivilStatus* :

600 (1800 in state2, 3020 in state3) facts under the predicate *CivilStatus* ,

169 (518 in state2, 860 in state3) facts under the predicate *Father* ,

220 (669 in state2, 1120 in state3) facts under the predicate *Husband* ,

21 (75 in state2, 119 in state3) facts under the predicate *Tax* .

Integrity Constraints :

$$CivilStatus(x, y_1, z_1, t_1) \wedge CivilStatus(x, y_2, z_2, t_2) \rightarrow y_1{=}y_2 \wedge z_1{=}z_2 \wedge t_1{=}t_2$$

$$Father(x_1, y) \wedge Father(x_2, y) \rightarrow x_1{=}x_2$$

$$Husband(x, y_1) \wedge Husband(x, y_2) \rightarrow y_1 = y_2$$

$$Husband(x_1, y) \wedge Husband(x_2, y) \rightarrow x_1 = x_2$$

$$CivilStatus(x, y, z, t) \rightarrow x > 0 \wedge x < 100000 \wedge y > 0 \wedge y < 125 \wedge z \in \{Male, Female\} \wedge$$
$$t \in \{Student, Retired, Business, Service\}$$

$$CivilStatus(x, y, z, Student) \rightarrow y < 25$$

$$CivilStatus(x, y, z, Retired) \rightarrow y > 60$$

$$Father(x, y) \rightarrow Sex(x, Male) \wedge Sex(y, Male)$$

$$Husband(x, y) \rightarrow Sex(x, Male) \wedge Sex(y, Female)$$

$$Husband(x, y) \wedge Age(x, p) \wedge Age(y, q) \rightarrow p \geq 20 \wedge q \geq 20$$

$$CivilStatus(x, y, z, t) \wedge y < 20 \rightarrow t = Student$$

$$Dependent(x, y) \rightarrow Tax(y, x)$$

Transaction :

insertfact *CivilStatus* (1, 50, *Male* , *Service*),

insertfact *CivilStatus* (2, 45, *Female* , *Business*),

insertfact *CivilStatus* (3, 19, *Male* , *Student*),

insertfact *Husband* (1, 2),

insertfact *Father* (1, 3),

insertfact *Tax* (1, 3).

where *Male* , *Female* , *Student* , *Retired* , *Business* , *Service* are constants.

APPENDIX 3

BNF Syntax

<query-exp> ::=

<query-term>

| <query-exp> UNION [ALL] <query-term>

<query-term> ::=

<query-spec>

| (<query-exp>)

<query-spec> ::=

SELECT [ALL | DISTINCT] <selection> <table-exp>

<selection> ::=

<scalar-exp-commalist> | *

<table-exp> ::=

<from-clause>

[<where-clause>]

[<group-by-clause>]

[<having-clause>]

<from-clause> ::=

FROM <table-ref-commalist>

<scalar-exp-commalist> ::=

<scalar-exp> [, <scalar-exp-commalist>]

<table-ref-commalist> ::=

<table-ref> [, <table-ref-commalist>]

<table-ref> ::=

<table> [<range-variable>] [ON INSERT | ON DELETE | ON UPDATE <column-ref>]

<where-clause> ::=

WHERE <search-condition>

<group-by-clause> ::=

GROUP BY <column-ref-commalist>

<column-ref-commalist> ::=

<column-ref> [, <column-ref-commalist>]

<having-clause> ::=

HAVING <search-condition>

<search-condition> ::=

<boolean-term>

| <search-condition> OR <boolean-term>

<boolean-term> ::=

<boolean-factor>

| <boolean-term> AND <boolean-factor>

<boolean-factor> ::=

[NOT] <boolean-primary>

<boolean-primary> ::=

<predicate>

| (<search-condition>)

<predicate> ::=

<comparison-predicate>

| <between-predicate>

| <like-predicate>

| <test-for-null>

| <in-predicate>

| <all-or-any-predicate>

| <existence-test>

<comparison-predicate> ::=

<scalar-exp> <comparison> { <scalar-exp> | <subquery> }

<comparison> ::=

= | <> | < | > | <= | >=

<between-predicate> ::=

<scalar-exp> [NOT] BETWEEN <scalar-exp> AND <scalar-exp>

<like-predicate> ::=

<column-ref> [NOT] LIKE <atom> [ESCAPE <atom>]

<test-for-null> ::=

<column-ref> IS [NOT] NULL

<in-predicate> ::=

<scalar-exp> [NOT] IN { <subquery> | <atom> [, <atom-commalist>] }

<atom-commalist> ::=

<atom> [, <atom-commalist>]

<all-or-any-predicate> ::=

<scalar-exp> <comparison> [ALL | ANY | SOME] <subquery>

<existence-test> ::=

EXISTS <subquery>

<subquery> ::=

(SELECT [ALL | DISTINCT] <selection> <table-exp>)

<scalar-exp> ::= <term> | <scalar-exp> { + | - } <term>

<term> ::= <factor> | <term> { * | / } <factor>

<factor> ::= [+ | -] <primary>

<primary> ::= <atom> | <column-ref> | <function-ref> | <scalar-exp>

<atom> ::= <parameter-ref> | <literal> | USER

<parameter-ref> ::= <parameter> [[INDICATOR] <parameter>]

<function-ref> ::= COUNT(*) | <distinct-function-ref> | <all-function-ref>

<distinct-function-ref> ::= { AVG | MAX | MIN | SUM | COUNT } (DISTINCT <column-ref>)

<all-function-ref> ::= { AVG | MAX | MIN | SUM | COUNT } ([ALL] <scalar-exp>)

<table> ::= <base-table> | <view>

<base-table> ::= [<user> .] <identifier>

<view> ::= [<user> .] <identifier>

<user> ::= <authorization-identifier>

<column-ref> ::= [OLD | NEW] [<column-qualifier> .] <column>

<column-qualifier> ::= <table> | <range-variable>

APPENDIX 4

Examples of Integrity Constraints

Example 11.4:

Type: *Static & Domain*

Constraint:

Name of a department belongs to the domain {Fin,Admin,Comp}

Closed formula:

$$\forall x \forall y \forall z (Employee(x, y, z) \rightarrow y='Fin' \vee y='Admin' \vee y='Comp')$$

SQL Syntax:

```
IMPOSE CONSTRAINT SD AS
    (
        SELECT    *
        FROM      EMPLOYEE
        WHERE     EMP_DEPT NOT IN ('Fin','Admin','Comp')
    ) IS EMPTY
```

Example 11.5:

Type: *Static & Existential or Inter-relational or Referential*

Constraint:

Every worker has a manager

Closed formula:

$$\forall x\,(Worker(x, y) \rightarrow \exists z\ Manager(z, x))$$

SQL Syntax:

```
IMPOSE CONSTRAINT SE AS
        (
            SELECT    WORKER_ID
            FROM      WORKER
    ) IS IN
    (
            SELECT    WORKER_ID
            FROM      MANAGER
    )
```

Example 11.6:

Type: *Static & Functional dependency or Check*

Constraint:

A child cannot have more than one father

Closed formula:

$$\forall x\,\forall y\,(Father(x, z) \wedge Father(y, z) \rightarrow x = y)$$

SQL Syntax:

```
IMPOSE CONSTRAINT SF AS
    (
            SELECT    F1.FATHER_ID, F2.FATHER_ID
            FROM      FATHER F1, FATHER F2
            WHERE     F1.CHILD_ID = F2.CHILD_ID
            AND NOT F1.FATHER_ID = F2.FATHER_ID
    ) IS EMPTY
```

Example 11.7:

Type: *Static & Check*

Constraint:

If one of the parents of an unemployed person is employed then the person must be dependent on that parent

Closed formula:

$$\forall x \forall y \, (Parent(x, y) \wedge Employed(x) \wedge \neg Employed(y) \rightarrow Dependent(y, x))$$

SQL Syntax:

```
IMPOSE CONSTRAINT SC AS
        (
                SELECT    P.CHILD_ID, P.PARENT_ID
                FROM      PARENT P, EMPLOYED E
                WHERE     P.PARENT_ID = E.EMP_ID
                AND P.CHILD_ID       NOT IN
                   ( SELECT    EMP_ID
                     FROM      EMPLOYED
                   )
    ) IS IN
    (       SELECT    DEPN_ID, PROV_ID
            FROM      DEPENDENT
    )
```

Example 11.8:

Type: *Aggregate on sum*

Constraint:

Total salary of all the employees in a department must be less than the budget of the department

Closed general formula:

$$\forall x \, \forall y \, (Department(x) \wedge Budget(x, y) \wedge Sum(v, \exists u \, \exists v \; Employee(u, x, v), z) \rightarrow z < y)$$

SQL Syntax:

```
IMPOSE CONSTRAINT ASU AS
        (
            SELECT    EMP_DEPT, EMP_SAL, BUD_AMOUNT
            FROM      EMPLOYEE, BUDGET
            WHERE     EMP_DEPT = BUD_DEPT
            GROUP     BY BUD_DEPT
            HAVING    SUM(EMP_SAL) >= BUD_AMOUNT
    ) IS EMPTY
```

Example 11.9:

Type: *Aggregate on count*

Constraint:

Maximum number of employees in a department is 100

Closed general formula:

$$\forall x \, (Department(v) \wedge Count(\exists u \, \exists w \; Employee(u, v, w), x) \rightarrow x < 100)$$

SQL Syntax:

```
IMPOSE CONSTRAINT AC AS
        (
            SELECT    DISTINCT EMP_DEPT
            FROM      EMPLOYEE
            GROUP     BY EMP_DEPT
            HAVING    COUNT(EMP_ID) > 100
    ) IS EMPTY
```

Example 11.10:

Type: *Transition on deletion*

Constraint:

Lay-off of employees whose income is less than 10000 will not be permitted

Closed formula:

$$\forall x \forall y \forall z (Del\_Employee(x, y, z) \rightarrow z \geq 10000)$$

SQL Syntax:

```
IMPOSE CONSTRAINT TD AS
        (
                SELECT    EMP_ID
                FROM      EMPLOYEE ON DELETE
                WHERE     EMP_SAL < 10000
        ) IS EMPTY
```

Example 11.11:

Type: *Transition on update*

Constraint:

On updating, salary of an employee should always increase

Closed formula:

$$\forall x \forall y \forall z \forall y' \forall z' (Upd\_Employee(x, y, z, x, y', z') \wedge z \neq z' \rightarrow z' > z)$$

SQL Syntax:

```
IMPOSE CONSTRAINT TU AS
        (
                SELECT    EMP_ID
                FROM      EMPLOYEE ON UPDATE EMP_SAL
                WHERE OLD SALARY >= NEW SALARY
        ) IS EMPTY
```

Example 11.12:

Type: *Transition on insertion*

Constraint:

Maximum loan an employee can be given is five times his salary or the employee has been given a special loan permission

Closed formula:

$$\forall x \forall y \, (Ins\_Loan(x, y) \wedge Employee(x, u, v) \wedge \neg SpecialLoan(x) \wedge w = 5*v \rightarrow y \leq w)$$

SQL Syntax:

```
IMPOSE CONSTRAINT TI2 AS
        (
                SELECT    LOAN_AMOUNT, EMP_SAL
                FROM      LOAN L ON INSERT, EMPLOYEE E
                WHERE     L.EMP_ID = E.EMP_ID
                AND E.EMP_ID NOT IN
                   (
                      SELECT    EMP_ID
                      FROM      SPECIAL_LOAN
                   )
                AND LOAN_AMOUNT > 5*E.SALARY
) IS EMPTY
```

Example 11.13:

Type: *Multiple actions on transition*

Constraint:

A change in grade must be accompanied by a change in salary

Closed formula:

$$\forall x \forall y \forall y' \forall z \forall z' (Upd\_Grade(x, y, x, 'y') \wedge y \neq y' \rightarrow Upd\_Employee(x, u, v, x, u', v') \wedge v \neq v'$$

SQL Syntax:

```
IMPOSE CONSTRAINT TM AS
    (
        SELECT    EMP_ID
        FROM      GRADE ON UPDATE EMP_GRADE
        WHERE     OLD EMP_GRADE <> NEW EMP_GRADE
    ) IS IN
    (
        SELECT    EMP_ID
        FROM      EMPLOYEE ON UPDATE EMP_SAL
        WHERE     OLD EMP_SAL ≠ NEW EMP_SAL
    )
```

References

1. P.B.Andrews, *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, Inc., (1986).

2. K.R.Apt, H.A.Blair, and A.Walker, "Towards a theory of declarative knowledge," *J.Minker (ed.): Foundation of Deductive Databases and Logic Programming*, pp. 89-148, Morgan Kaufman Publishers, Inc, (1988).

3. P.Asirelli, M.D.Santis, and M.Martelli, "Integrity constraint in logic databases," *Journal of Logic Programming*, **Vol. 3**, pp. 221-232, (1985).

4. F.Bancilhon and N.Spyratos, "Update semantics of relational views," *ACM Transactions on Database Systems*, **Vol. 6, No.4**, pp. 557-575, (December 1981).

5. P.A.Bernstein, B.T.Blaustein, and E.M.Clarke, "Fast maintenance of semantic integrity assertions using redundant aggregate data," *Proceedings of the 6th International Conference on Very Large Data Bases*, pp. 126-136, (1980).

6. E.Bertino and D.Musto, "Correctness of semantic integrity checking database management systems," *Acta Informatica*, **Vol. 26**, pp. 25-57, (1988).

7. G.S.Boolos and R.C.Jeffrey, *Computability and Logic*, Cambridge University Press, (1988).

8. I.Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley Publishing Company, (1986).

9. M.L.Brodie and F.Manola, "Database management: A survey," *I.Mylopoulos and M.L.Brodie (eds.): Readings in Artificial Intelligence and Databases*, pp. 10-34, Morgan Kaufmann Publishers, Inc., (1989).

10. F.Bry, H.Decker, and R.Manthey, "A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases," *Proceedings of Extending Database Technology*, pp. 488-505, Venice, (1988).

11. C.R.Carlson, A.K.Arora, and M.M.Carlson, "The application of functional dependency theory to relational databases," *The Computer Journal*, **Vol. 25, No.1**, pp. 68-73, (1982).

12. M.A.Casanova, L.Tucherman, and A.L.Furtado, "Enforcing inclusion dependencies and referential integrity," *Proceedings of the 14th International Conference on Very Large Data Bases*, pp. 38-49, (1988).

13. A.K.Chandra, "Theory of database queries," *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 1-9, Austin, Texas, (March 1988).

14. C.L.Chang, "DEDUCE 2: Further investigations of deduction in relational databases," *H.Gallaire and J.Minker (eds.): Logic and Databases, New York*, New York, (1978).

15. C.L.Chang and R.C.T.Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, (1973).

16. K.L.Clark, "Negation as failure," *H.Gallaire and J.Minker (eds.): Logic and Databases*, pp. 293-322, Plenum Press, New York, (1978).

17. W.F.Clocksin and C.S.Mellish, *Programming in Prolog*, Springer International Student Edition, (1984).

18. E.F.Codd, "A relational model for large shared data banks," *Communications of the ACM*, **Vol. 13, No.6**, pp. 377-387, (1970).

19. I.M.Copi, *Symbolic Logic*, Macmillan Publishing Company Ltd, (1979).

20. C.J.Date, *An Introduction to Database Systems, Vol2*, Addison Wesley, (1985).

21. C.J.Date, *A Guide to SQL Standard*, Addison-Wesley Publishing Company, (1989).

22. C.J.Date, "Referential integrity," *Proceedings of the 7th International Conference on Very Large Data Bases*, pp. 2-12, (1981).

23. C.J.Date, *An Introduction to Database Systems, Vol1, 4th edition*, Addison Wesley, (1986).

24. U.Dayal and P.A.Bernstein, "On the updatability of relational views," *Proceedings of the 4th VLDN Conference*, West Berlin, (September, 1978).

25. H.Decker, "Integrity enforcements on deductive databases," *L.Kerschberg (ed.): Proceedings of the First International Conference on Expert Database Systems*, pp. 271-285, Charleston, South Carolina, (April 1986).

26. E.W.Elcock, *Absys: The first logic programming language - A retrospective and a commentary*, Department of Computer Science, The University of Western Ontario, Canada, Technical Report No. 210, (July 1988).

27. K.R.Apt and M.H.Van Emden, "Contributions to the theory of logic programming," *Journal of the Association for Computing Machinery*, **Vol. 29**, pp. 841-862, (July 1982).

28. H.B.Enderton, *A Mathematical Introduction to Logic*, Academic Press, Inc., (1972).

29. D.Maier and D.S.Warren, *Computing with Logic - Logic Programming with Prolog*, Benjamin/Cummings Publishing Company, Inc., (1988).

30. K.P.Eswaran and D.D.Chamberlin, "Functional specification of a subsystem for database integrity," *Proceedings of the First International Conference on Very Large Data Bases*, (1975).

31. R.Fagin, G.M.Kuper, J.D.Ullman, and M.Y.Vardi, "Updating logical databases," *Advances in Computing Research*, **Vol. 3**, pp. 1-18, JAI Press Inc., (1986).

32. R.Fagin and M.Y.Vardi, *The theory of data dependencies - a survey*, IBM Research Laboratory, San Jose, California, Research Report RJ 4321 (47149), (1984).

33. E.B.Fernandez, R.C.Summers, and C.Wood, *Database Security and Integrity*, Addison Wesley, (1981).

34. H.Gallaire, "Logic databases vs deductive databases," *Proceedings of Logic Programming Workshop*, pp. 608-622, Algarve, Portugal, (1983).

35. H.Gallaire, J.Minker, and J.-M.Nicolas, "Logic and databases - a deductive approach," *ACM Computing Surveys*, **Vol. 16, No.2**, pp. 153-185, (1984).

36. G.Gardarin and P.Valduriez, *Relational Databases and Knowledge Bases*, Addison Wesley, (1989).

37. A.Van Gelder, K.Ross, and J.S.Schlipf, "Unfounded sets and well-founded semantics for general logic programs," *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 221-230, Austin, Texas, (March 1988).

38. M.Gelfond and V.Lifschitz, "The stable model semantics for logic programming," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 1070-1080, Seattle, USA, (August 1988).

39. M.Gelfond and H.Przymusinska, "Negation as failure: Careful closure procedure," *Artificial Intelligence*, **Vol. 30**, pp. 273-287, North-Holland Publishing Company, (1986).

40. R.Goebel, K.Furukawa, and D.Poole, "Using definite clauses and integrity constraints as the basis for a theory formation approach to diagnostic reasoning," *E.Shapiro (ed.): Proceedings of the 3rd International Conference on Logic Programming*, pp. 211-222, Springer-Verlag, London, U.K, (July 1986).

41. P.R.Halmos, *Naive Set Theory*, Springer-Verlag New York Inc., (1974).

42. I.N.Herstein, *Topics in algebra*, John Wiley & Sons, (1964).

43. D.Hilbert and W.Ackermann, *Principles of Mathematical Logic*, Chelsea Publishing Company, New York, (1950).

44. W.Hodges, *Logic - An Introduction to Elementary Logic*, Penguin Books, (1988).

45. G.Hulin, A.Pirotte, D.Roelants, and M.Vauclair, "Logic and databases," *A.Thayse (ed.): From Modal Logic to Deductive Databases*, pp. 279-300, John Wiley & Sons, (1989).

46. J.Jaffar, J-L.Lassez, and M.J.Maher, "Some issues and trends in the semantics of logic programming," *E.Shapiro (ed.): Proceedings of the 3rd International Conference on Logic Programming*, pp. 223-241, Springer-Verlag, London, U.K, (July 1986).

47. K.Knight, "Unification: A multidisciplinary survey," *ACM Computing Surveys*, **Vol. 21, No.1**, pp. 93-124, (1989).

48. R.Kocharekar, "Nulls in relational databases: Revisited," *SIGMOD Record*, **Vol. 18, No.1**, pp. 68-73, (JMarch 1989).

49. S.Koenig and R.Paige, "A transformational framework for the automatic control of derived data," *Proceedings of the 7th International Conference on Very Large Data Bases*, pp. 306-318, (1981).

50. P.G.Kolaitis and C.H.Papadimitriou, "Why not negation by fixpoint?" *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 231-239, Austin, Texas, (March 1988).

51. R.A.Kowalski, "Logic programming with integrity constraints," *Workshop on Logic Programming, Imperial College, London*, (1989).

52. R.Kowalski, "Algorithm = Logic + Control," *Communications of the ACM*, **Vol. 22, No.7**, pp. 424-435, (July 1979).

53. R.A.Kowalski, *Logic for Problem Solving*, North-Holland Publisher Co., (1979).

54. R.Kowalski and D.Kuehner, "Linear resolution with selection function," *Artificial Intelligence*, **Vol. 2**, pp. 227-260, (1971).

55. R.A.Kowalski, F.Sadri, and P.Soper, "Integrity checking in deductive databases," *Proceedings of the 13th VLDB Conference*, pp. 61-69, Brighton, (1987).

56. K.Kunen, "Some remarks on the completed database," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 978-992, Seattle, USA, (August 1988).

57. J.-L.Lassez, V.L.Nguyen, and E.A.Sonenberg, "Fixed point theorems and semantics: A folk tale," *Information Processing Letters*, **Vol. 14, No.3**, pp. 112-116, (May 1982).

58. E.J.Lemmon, *Beginning Logic*, Thomas Nelson and Sons Ltd, Great Britain, (1977).

59. Y.Y.Leung and D.L.Lee, "Logic approaches for deductive databases," *IEEE Expert*, (Winter 1988).

60. T.-W.Ling, "Integrity constraint checking in deductive databases using the Prolog not-predicate," *Data & Knowledge Engineering*, **Vol. 2**, pp. 145-168, (1987).

61. W.Lipski, "On semantic issues connected with incomplete information databases," *ACM Transactions on Database Systems*, **Vol. 4, No.3**, pp. 262-296, (September 1979).

62. W.Lipski, "On databases with incomplete information," *Journal of the Association for Computing Machinery*, **Vol. 28, No.1**, pp. 41-70, (January 1981).

63. J.W.Lloyd, *Foundations of Logic Programming, 2nd, Extended Edition*, Springer Verlag, Symbolic Computation Series, (1984).

64. J.W.Lloyd, "An introduction to deductive database systems," *The Australian Computer Journal*, **Vol. 15, No.2**, pp. 52-57, (May 1983).

65. J.W.Lloyd, E.A.Sonenberg, and R.W.Topor, *Integrity constraint checking in stratified databases*, Department of Computer Science, University of Melbourne, Technical Report 86/5, (1986).

66. J.W.Lloyd and R.W.Topor, "A basis for deductive database systems," *Journal of Logic Programming*, **Vol. 2, No.2**, pp. 93-109, (1985).

67. J.W.Lloyd and R.W.Topor, "Making Prolog more expressive," *Journal of Logic Programming*, **Vol. 1, No.3**, pp. 225-240, (1984).

68. A.Lobo, J.Minker, and A.Rajasekhar, "Weak completion theory for non-Horn logic programs," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 828-842, Seattle, USA, (August 1988).

69. D.W.Loveland, "Near-Horn Prolog," *E.Wada (ed.): Proceedings of the 4th International Conference on Logic Programming*, pp. 456-469, Springer-Verlag, Tokyo, Japan, (July 1987).

70. D.W.Loveland, *Automated Theorem Proving*, North-Holland Publishing Company, (1978).

71. M.J.Maher, "Equivalence of logic programs," *E.Shapiro (ed.): Proceedings of the 3rd International Conference on Logic Programming*, pp. 410-424, Springer-Verlag, London, U.K, (July 1986).

72. P.Mancarella and D.Pedreschi, "An algebra of logic programs," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 1006-1023, Seattle, USA, (August 1988).

73. A.Martelli and U.Montanari, "An efficient unification algorithm," *ACM Transactions on Programming Languages and Systems*, **Vol. 4, No.2**, pp. 258-282, (April 1982).

74. E.Mendelson, *Introduction to Mathematical Logic*, Wadsworth & Brooks/Cole Advanced Books and Software, California, (1987).

75. D.A.Miller and G.Nadathur, "Higher-order logic programming," *E.Shapiro (ed.): Proceedings of the 3rd International Conference on Logic Programming*, pp. 448-462, Springer-Verlag, London, U.K, (July 1986).

76. J.Minker, "Perspectives in deductive databases," *Journal of Logic Programming*, **Vol. 5**, pp. 33-60, (1988).

77. J.Minker, "On indefinite databases and closed world assumption," *Readings for non-Monotonic Reasoning*, (1988).

78. G.Nadathur and D.A.Miller, "An overview of Lambda-Prolog," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 810-827, Seattle, USA, (August 1988).

79. J.-M.Nicolas, "Logic for improving integrity checking in relational databases," *Acta Informatica*, **Vol. 18**, pp. 227-253, (1982).

80. J.M.Nicolas and K.Yazdanian, "Integrity checking in deductive databases," *H.Gallaire and J.Minker (eds.): Logic and Databases*, pp. 325-344, Plenum Press, New York, (1978).

81. M.S.Paterson, "Linear unification," *Journal of Computer and Systems Sciences*, **Vol. 16**, pp. 158-167, (1978).

82. D.A.Plaisted, "Non-Horn clause logic programming without contrapositives," *Journal of Automated Reasoning*, **Vol. 4**, pp. 287-325, (1988).

83. H.Przymusinska and T.C.Przymusinska, "Weakly perfect model semantics for logic programs," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 1106-1120, Seattle, USA, (August 1988).

84. T.C.Przymusinski, "On the declarative semantics of deductive databases and logic programs," *J.Minker (ed.): Foundation of Deductive Databases and Logic Programming*, pp. 193-216, Morgan Kaufman Publishers, Inc, (1988).

85. S.Raatz and J.Gallier, "A relational semantics for logic programming," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 1024-1035, Seattle, USA, (August 1988).

86. R.Reiter, "A sound and sometimes complete query evaluation algorithm for relational database with null values," *Journal of the Association for Computing Machinery*, **Vol. 33, No.2**, pp. 349-370, (April 1986).

87. R.Reiter, "On the integrity of typed first order databases," *H.Gallaire,J.Minker and J.-M.Nicolas (eds.): Logic and Databases*, **Vol. 1**, pp. 137-157, Plenum Press, New York, London, (1981).

88. J.A.Robinson, "Computational logic: The unification computation," *B.Meltzer and D.Michie (eds.): Machine Intelligence (6)*, pp. 63-72, (1971).

89. J.A.Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the Association for Computing Machinery*, **Vol. 12**, pp. 23-41, (1965).

90. K.A.Ross and R.W.Topor, "Inferring negative information from disjunctive databases," *Journal of Automated Reasoning*, **Vol. 4**, pp. 397-424, (1988).

91. F.Sadri and R.A.Kowalski, "An application of general purpose theorem-proving to database integrity," *J.Minker (ed.): Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming*, (1987).

92. C.Small, "Guarded default databases: A prototype implementation," *Prolog and Databases: Implementations and new directions*, pp. 121-134, Ellis Horwood Limited, England, (1988).

93. B.T.Smith and D.W.Loveland, "A simple near-Horn Prolog interpreter," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 794-809, Seattle, USA, (August 1988).

94. L.Sterling and A.Lakhotia, "Composing Prolog meta-interpreters," *R.A.Kowalski and K.A.Bowen (eds.): Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 386-403, Seattle, USA, (August 1988).

95. L.Sterling and E.Shapiro, *The Art of Prolog*, pp. 303-329, The MIT Press, Cambridge, Massachusetts, (1986).

96. M.E.Stickel, "A Prolog technology theorem prover: implementation by an extended Prolog compiler," *Proceedings of the 8th International Conference in Automated Deduction*, pp. 573-587, Oxford, England, (July 1986).

97. M.E.Stickel, "Resolution theorem proving," *Annual Review: Computer Science*, **Vol. 3**, pp. 285-316, (1988).

98. R.R.Stoll, *Set Theory and Logic*, W.H.Freeman and Company, (1963).

99. M.Stonebraker, "Implementation of integrity constraints and views by query modification," *J.Mylopoulos and M.L.Brodie (eds.): Readings in Artificial Intelligence and Databases*, pp. 533-546, Morgan Kaufmann Publishers, Inc., (1989).

100. M.Stonebraker, E.Wong, P.Kreps, and G.Held, "The design and implementation of INGRES," *ACM Transactions on Database Systems*, **Vol. 1, No.3**, pp. 189-222, (1976).

101. S-A.Tarnlund, "Logic Programming - from logic point of view," *Proceedings of the 1986 Symposium on Logic Programming*, pp. 96-103, IEEE Computer Society Press, Salt Lake City, Utah, USA, (September 1986).

102. R.W.Topor, T.Keddis, and D.W.Wright, *Deductive database tools*, Department of Computer Science, University of Melbourne, Technical Report 84/7.

103. R.W.Topor and E.A.Sonenberg, "On domain independent databases," *J.Minker (ed.): Foundation of Deductive Databases and Logic Programming*, pp. 217-240, Morgan Kaufman Publishers, Inc, (1988).

104. J.D.Ullman, *Principles of Database Systems, 2nd edition*, Computer Science Press International, Inc, Maryland, USA, (1984).

105. M.H.Van Emden and R.A.Kowalski, "The semantics of predicate logic as a programming language," *Journal of the Association for Computing Machinery*, **Vol. 23, No.4**, pp. 733-742, (October 1976).

106. M.H.Williams, J.C.Neves, and S.O.Anderson, "Security and integrity in logic databases using Query-by-Example," *Proceedings of Logic Programming Workshop*, pp. 304-340, Algarve, Portugal, (1983).

107. Yahya and Henschen, "Deduction in non-Horn databases," *Journal of Automated Reasoning*, **Vol. 1**, pp. 141-160, (1985).

108. C.Zaniolo, "Database relations with null values," *Journal of Computer and System Sciences*, **Vol. 28**, pp. 142-166, Academic Press. Inc., (1984).

109. M.M.Zloof, "Query-by-Example: a database language," *IBM Systems Journal*, **Vol. 16, No.4**, pp. 324-343, (1975).