On Negative Information in Deductive Databases
an invited paper

Marek A. Suchenek *and* Rajshekhar Sunderraman

Journal of Database Administration 1 (1990), pp 28–41

Abstract

This paper provides an overview of deductive data bases, with an emphasis on problems related to negative information. It embeds this subject into a wider context of logic programming, exposing certain peculiarities pertinent to treatment of negation in these fields.

Contents

1 Introduction

In this paper, we discuss some problems related to representing and handling negative information in deductive databases, a species which has been recognized by many as a cutting edge of research in database theory and applications. The subject, albeit over 10 years old, has not lost anything from its appeal and momentum. To the contrary, it appears to gain continuously even more attention from database community, attracting by its intellectually challenging nature the most talented and successful scholars. It may be embedded in the context of another, nowadays fashionable, phenomenon known under the name of non-monotonic logic.

The deductive model of a database emerged from its predecessor relational model, by incorporating certain elements and methods of automated theorem proving. A relational database is an information storage and retrieval system in which entries are organized in a finite collection of relations with the standard operations on these relations as: insertion, deletion, selection, projection, product, join, union, intersection, and difference. Every relation is represented by a finite set of tuples, whose elements are in this relation and are usually called *facts*. Database users may access the information stored in the database by means of *queries* expressed in a suitable language. Database management system plays the role of an interface between its user and the physical database, and is responsible for accepting user's queries and operations as well as returning answers to the user. Moreover, it secures that the integrity constraints pertinent to the database are not violated. This last task involves certain elements of reasoning, because the integrity constraints are usually expressed in the form of rules. Relational database, or rather its management system, does not go much beyond passive verification of the legality of the performed operations against the coded integrity constraints. Another feature of relational databases that requires some form of reasoning is the concept *view*. Views, may be interpreted as definitions of relations not explicitly represented in a physical database. They are usually limited to certain non-recursive rules and are intended to information hiding rather than to equip a database with deductive abilities.

A seemingly natural observation that an active use of rules present in a database may result in the derivation of new facts, not explicitly represented it, leads to the concept of *deductive database*. One may say that this was a passiveness of relational database in handling integrity constraints which

gave rise to deductive databases. It has taken, however, a decade between [GR68] and [Rei78b] to materialize this concept in a form of more or less explicit definition. Concurrent and mostly independent development in logic programming contributed significantly to the faster maturing of deductive databases.

Although deductive database in its current comprehension does not imply or restrict its possible implementations, conceptually it may be viewed as a relational database equipped with additional, "intelligent" layers, capable of complex reasoning from the content of the relational component. In fact, this point of view seems to prevail. One should note, however, that since contemporary relational databases are not purely passive in executing their integrity constraints, the distinction between relational and deductive databases is rather fuzzy, and one certainly can find a variety of cases which may be classified "relational" as well as "deductive".

The deductive model turned out to be more powerful (at least as far as the *expressive* power is concerned) than the relational one, and as a matter of fact, relational databases may, to a large extent, be understood as a very special case of deductive ones. However, its different form of representation caused that the natural one-to-one correspondence between relational databases and the respective subclass of deductive ones is not an identity, although very strongly resembles it. Therefore, to make a relational database deductive, a suitable translation process is necessary. It turns out that this process is largely responsible for the demand of non-standard treatment of negation in processing queries to deductive databases.

If one translates the content of a purely relational database into its deductive counterpart, the problem with the interpretation of negative information manifests its presence. Certainly, while translating tuples onto predicates or more precisely, onto *ground literals*, only positive information of the relational part is explicitly processed, despite the fact, that the relational part *may* encode negative information too. The first systematic treatment of this phenomenon by means of *closed world assumption* may be found in [Rei78b], and since then, a mature age of deductive databases seems to begin.

The closed world assumption may be recognized as a benchmark, which makes a difference between relational and deductive databases, since in the relational model the closed world assumption is somehow hardwired, and is therefore superfluous. Deductive databases, contrary to relational ones, do need the closed world assumption to allow negative conclusions from purely

positive information: all simple positive facts that cannot be derived from the database are assumed to be false under *cwa*, or in other words, their negations are asserted. This simple and seemingly unquestionable rule creates certain problems, however, if mechanically applied to deductive databases which are not a result of translating the content of a relational database. This is why a proper treatment of negation is so crucial in this context.

In the present comprehension of the term, deductive databases consist of the following syntactic elements:

- extensional part, which corresponds to the relations, usually in the form of a set of so called *ground literals*,
- intensional part, which corresponds to view definitions, integrity constraints, and other dependencies, usually in the form of a set of so called *clauses*,
- resolution theorem prover, which derives logical consequences of the extensional and intensional content of the database
- meta-rules for negative inferences, e.g. the closed world assumption.

This paper focuses on the last element, discussing how it affects the entire database and its information content, and what semantic meaning may be consistently associated with rules of this kind. Because the area of deductive databases is not the only one concerned with proper treatment of negation, we also will briefly discuss how this problem is approached in logic programming.

2 Preliminaries

Deductive databases have a form of collection of clauses expressible in a language of first-order logic. In this section, we provide a concise introduction to first-order clause logic, and outline the resolution refutation procedure for automated deduction.

2.1 Syntax of First-Order Logic

Logic is a branch of mathematics which investigates infallible rules of reasoning. *Propositional logic* is perhaps the simplest form of logic which deals

with propositional statements, as "It is sunny today" or "It is not raining today", and reasoning within this context. Propositional symbols are used to represent statements and may incorporate logical connectives $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), and $\neg$ (not), to combine simple statements into more complex ones. E.g. if P denotes "It is sunny today", and Q denotes "It is not raining today", then $P \rightarrow Q$ denotes "If it is sunny today then it is not raining today".

One can always conclude Q from the premises:

$$\text{P and P} \rightarrow \text{Q}$$

based on the inference rule of propositional logic, called *modus ponens*. This rule, together with the propositional axioms (see Appendix) serves as a building block of proofs, i.e. finite sequences of propositions, each of them being either an axiom or a premise, or following from two preceding ones as a result of application of this rule. The last proposition in such a sequence is the one which is being proved. For example, if Q is a premise (in general, one can have as many premises as one wishes) which is for some reason known or asserted true, then the following sequence

1. Q (premise)
2. $\text{Q} \rightarrow (\text{P} \rightarrow \text{Q})$ (axiom)
3. $\text{P} \rightarrow \text{Q}$ (result of rule modus ponens applied to 1. and 2.)

is a proof of $\text{P} \rightarrow \text{Q}$. In a case like this, we say that "Q" proves "$\text{P} \rightarrow \text{Q}$", or equivalently, that "$\text{P} \rightarrow \text{Q}$" is provable from "Q" (see Appendix for example of a proof of $\neg\,\text{Q} \rightarrow \neg\,\text{P}$ from premise $\text{P} \rightarrow \text{Q}$).

Despite its elegant simplicity, propositional logic is not particularly suitable to proper treatment of statements of the form: "All men are mortal", "Socrates is a man", nor to reasoning within this context. E.g. it is not possible to conclude that "Socrates is mortal" from these statements using only propositional axioms and modus ponens. It is the *first-order logic*, which is a more powerful form of logic capable of formally expressing such statements and reasoning with them. In addition to logical connectives, first-order logic allows quantifiers $\forall$ (for all) and $\exists$ (there exists) ranging over individual variables appearing in statements. For example,

1. $(\forall x)(Man(x) \rightarrow Mortal(x))$ may denote the statement "All men are mortal";

2. $Man(Socrates)$ may denote the statement "Socrates is a man".

Within first-order logic one can conclude $Mortal(Socrates)$, which denotes "Socrates is mortal", from the above sentences 1 and 2.

Since first-order logic provides a language for deductive databases, we will take a closer look at its syntax and semantics. The alphabet for first-order language consists of predicate symbols, constant symbols, variable symbols, function symbols, quantifiers, logical connectives, and other usual punctuation symbols. There are two essentially different kinds of expressions one can build out of the symbols of this alphabet: *terms* and *formulas*. Terms provide a repertoire of names for objects of the *universe of discourse*, like "Socrates", "x", "7", "f(x,g(7))", etc. More formally, a term is a constant symbol, or a variable symbol, or a function symbol followed by a list of arguments which are terms themselves. In the above examples "Socrates" and "7" are constants, "x" is a variable, and "f(x,g(7))" is a complex term, where f and g are functions. Formulas are used to express attributes and relationships pertinent to these objects. The statement attributing human race to Socrates, "Socrates is a man", is represented as Man(Socrates) using a unary predicate Man and the constant Socrates in. The statement relating Tom and Mary, "Tom likes Mary", is represented as Likes(Tom,Mary) using a binary predicate Likes and the constants Tom and Mary. One can also use functions to represent statements. For example, the statement "Tom respects Mary's father" may be represented as Respects(Tom,father(Mary)). A mathematical statement which states that x is less than f(x,g(7)) can be written as "< x (f(x,g(7)))", or in more familiar infix form, as "x < f(x,g(7))".

Simple statements of the above form are called *atomic formulas*. More formally, an atomic formula is of the form $P(t_1, \ldots, t_k)$, where P is a predicate symbol and $t_1, \ldots, t_k$ are terms. Quantifiers and logical connectives allow, in an obvious way, to compose simpler formulas into more complex ones. Atomic formulas and negated atomic formulas are referred to as *literals, positive literals* refer to atomic formulas, *negative literals* refer to negated atomic formulas.

Formulas expressible in first-order language may be fairly complicated, for example

$$(\forall x)(\exists y)(\forall z)(z \in y \equiv (\exists v)(v \in x \wedge z \in v)).$$

It should not be a surprise that handling complex formulas like this in a deductive database may be very costly and time consuming. Fortunately, although literals do not have enough expressive power, such complex creatures like the one above are hardly encountered in database applications. It has been widely accepted, that the class of *clauses*, which is a proper subclass of all first-order formulas, is satisfactorily sufficient for this purpose.

The notion of clause is a key syntactic concept in deductive databases and logic programming. Clauses allow to express statements more complex than literals, while maintaining relative simplicity so desirable by prospective users of deductive databases. Using them, one can represent dependencies and constraints of various kinds.

Let us take a look at two examples of statements one may wish to represent in a deductive database: "if X is the parent of Y then X is either the father of Y or the mother of Y", and "if X is a component of Y and Y is a component of Z then X is a sub-component of ". The first statement can be articulated, using the predicates father, Mother and parent as

$$parent(X, Y) \rightarrow father(X, Y) \vee Mother(X, Y)$$

while the second one by using the predicates Component and Sub-component as

$$Component(X, Y) \wedge Component(Y, Z) \rightarrow Sub - component(X, Z).$$

In the context of deductive databases the above formulas are written in a reversed direction, i.e. as

$$father(X, Y) \vee Mother(X, Y) \leftarrow parent(X, Y)$$
$$Sub - component(X, Z) \leftarrow Component(X, Y) \wedge Component(Y, Z)$$

With a help of logical rules, one can transform these formulas into the disjunctive normal form (disjunction of literals):

$$father(X, Y) \vee Mother(X, Y) \vee \neg parent(X, Y),$$
$$Sub - component(X, Z) \vee \neg Component(X, Y) \vee \neg Component(Y, Z).$$

The first formula can be interpreted as "either X is the mother of Y or is the father of Y or is not a parent of Y" and the second formula can be interpreted as "either X is a sub-component of Z or X is not a component of Y or Y is not a component of ". The formulas in the disjunctive normal form are referred to as clauses.

Formally, a clause is a formula of the form

$$(\forall X_1) \ldots (\forall X_n)(P_1 \vee \ldots \vee P_m \vee \neg Q_1 \vee \ldots \vee \neg Q_k)$$

where the X_is are the variable symbols occurring among the positive literals Ps and Qs. The same clause may be written in its equivalent form,

$$(\forall X_1) \ldots (\forall X_n)(P_1 \vee \ldots \vee P_m \leftarrow Q_1 \wedge \ldots \wedge Q_k)$$

For simplicity, they are stripped of the quantifiers when appearing in the context of a deductive database (or a logic program), so the first one takes the form

$$P_1 \vee \ldots \vee P_m \vee \neg Q_1 \vee \ldots \vee \neg Q_k$$

and the second one takes the form

$$P_1 \vee \ldots \vee P_m \leftarrow Q_1 \wedge \ldots \wedge Q_k$$

A clause is said to be a *definite clause* if $m = 1$ and an *indefinite clause* if $m > 1$. The left hand side of the clause is referred to as the *head* and the right hand side as the *body* of the clause. If the body of a clause is empty, the $\leftarrow$ symbol is dropped for notational convenience. A *Horn clause* is either a definite clause or a formula of the form

$$\neg(Q_1 \wedge \ldots \wedge Q_n)$$

which is customarily written as

$$\leftarrow Q_1, \ldots, Q_n$$

where each Q_i is a positive literal. Queries in deductive databases are represented using Horn clauses of this form.

A *ground clause* is a clause in which there are no occurrences of variables. A *positive clause* is one in which all the literals are positive. The table in Figure 1 summarizes the various kinds of clauses. An *empty clause* consists of

Clause Type	Example
Definite Clause	$Sub - component(X, Z) \leftarrow component(X, Y) \wedge component(Y, Z)$
Indefinite Clause	$father(X, Y) \vee Mother(X, Y) \leftarrow parent(X, Y)$
Positive Clause	$supplies(X, P1) \vee supplies(X, P2)$
Ground Clause	$father(Gary, Tom) \leftarrow Son(Tom, Gary)$
Empty Clause	$\square$

Figure 1: Different Kinds of Clauses

no literals and is denoted by $\square$. The empty clause represents a contradiction.

Although clauses constitute a small subclass of all first-order formulas, their expressive power is surprisingly strong. For example, every sentence of the form

$$\forall X_1 \cdots \forall X_n \phi(X_1, \ldots, X_n)$$

where $\phi(X_1, \ldots, X_n)$ is an arbitrarily complex quantifier free formula with variables $X_1, \ldots, X_n$ is logically equivalent to a finite set of clauses. Also, existential statements may be approximated using auxiliary constants or function symbols. These facts explain, to some extent why the language of clauses has been widely accepted for deductive databases, logic programming, and other domains of artificial intelligence.

2.2 Semantics of First-Order Logic

Semantics of first-order language provides an interpretation of the symbols used to construct sentences, so that these sentences become meaningful. It is rather exceptional that a clause itself implies its own meaning ($P \leftarrow P$ is such an exception, a tautologically true clause), because the logical value of the clause may depend on how one interprets the symbols which occur within it. An *interpretation* assigns meanings to variables, constants, functions, and predicate symbols. Formally, an interpretation for a set of clauses in a first-order language consists of a non-empty set D, called the domain (or world of discourse), an assignment of each constant symbol to an element in the domain, an assignment of each function symbol to a function on the domain,

and an assignment of each predicate symbol to a relation on the domain. The role of the domain is to specify the set of individual objects which may have been named by terms of the language, or in other words, to specify the set of all possible values these terms may assume. In case the language does not contain the equality symbol =, interpretation constitutes and idealized, not necessarily finite, relational database.

Example 2.1 Consider a first-order language with the relation symbol wife, and constant symbols Marek, Michele, Raj, Radhika, and Gopi. We can interpret the relation symbol wife in the usual way, i.e., wife(a,b) means "a is the wife of b". Consider the domain, D = { Marek Suchenek, Michele Vincent, Rajshekhar Sunderraman, Radhika Venkataraman, Rajaraman Sunderraman }, which consists of five individuals. If we associate the constants Marek, Michele, Raj, Radhika, and Gopi with the individuals Marek Suchenek, Michele Vincent, Rajshekhar Sunderraman, Radhika Venkataraman, and Rajaraman Sunderraman respectively and associate the predicate symbol wife with the relation { < Michele Vincent, Marek Suchenek >, < Radhika Venkataraman, Rajshekhar Sunderraman > } then the following sentences are true with respect to this interpretation.

1. $wife(Michele, Marek)$
2. $(\exists X) wife(Radhika, X)$

However the sentence $(\exists X) wife(X, Gopi)$ is false under this interpretation. * □

An interpretation $\mathcal{I}$ for a set of sentences S is said to be a *model* for S if each sentence in S is true in $\mathcal{I}$. In general, a set of sentences may have many essentially different models, for example, sizes of their relations may differ. If one treats interpretations as abstract relational databases, different sizes of relations correspond to different information content (the more tuples the greater the information content). It turns out that interpretations with possibly minimal information content, the so called *minimal models*, are the ones which are mostly useful in context of database applications. More precisely, $\mathcal{I}$ is called a *minimal model* if any deletion of tuples from relations of $\mathcal{I}$ infallibly results in an interpretation $\mathcal{J}$ which is not a model of S. A minimal model is the one which has possibly minimal

relations while satisfying all the sentences of S. E.g. the set of sentences $S = \{Male(X) \vee Female(X), Male(Marek), Female(Michele), Cat(Fay)\}$ has two minimal models: one with interprets the predicate symbol Male as a unary relation $\{< Marek >\}$ and the predicate symbol Female as a unary relation $\{< Michele >, < Fay >\}$, and another which interprets predicate symbol Male as a unary relation $\{< Marek >, < Fay >\}$, and symbol Female as a unary relation $\{< Michele >\}$. Models having larger relations, as e.g. one which interprets symbol Male as $\{< Marek >, < Fay >\}$, and symbol Female as $\{< Marek >, < Fay >\}$, are not minimal. Interpretation of symbols Male and Female as $\{< Marek >\}$ and $\{< Michele >\}$, respectively, is not a model of S (does not satisfy the clause Male(X) $\vee$ Female(X) which forces every X to be Male or Female).

Semantics restricted to minimal models, the so called *minimal model semantics*, is one of the central issues in deductive databases. It provides an adequate meaning for negative clauses. Its applications, however, go much beyond deductive databases: its variants have been widely accepted as the semantics in logic programming and in certain non-monotonic logic as well (under the name of predicate circumscription in the latter).

It is somewhat surprising, nevertheless known from the early thirties, that models for consistent sets of clauses may be built out of terms of the language in which these clauses are expressed. This fact, discovered by the German mathematician Herbrand, remained as some kind of curiosity till the seventies, when its application in the area of logics of programs and logic programming. This kind of models are particularly useful in deductive databases, because they allows to restrict semantical considerations of such databases to purely syntactic objects.

These useful objects are called *Herbrand interpretations*. Their domains consist of names of individual objects which may be expressed in the language in question. More precisely, the *Herbrand universe* (domain) of a set of clauses is the of all ground terms that can be formed out of the constant and function symbols that appear in the clauses. For example the Herbrand universe of the clauses Likes (Tom, Mary) and Likes (X, father (Y)) $\leftarrow$ Likes (X, Y) is:

{Tom, Mary, father(Tom), father(Mary), father(father(Tom)),
father(father(Mary)), ... }

The *Herbrand base* of a set of clauses is the set of all possible ground atomic formulas that can be constructed from the predicate symbols and ground terms from the Herbrand universe. For example the Herbrand base for the set of clauses { Likes(Tom,Mary), Married(Tom,Mary) } is:

{ Likes(Tom,Tom), Likes(Tom,Mary), Likes(Mary,Mary), Likes(Mary,Tom),
Married(Tom,Tom), Married(Tom,Mary), Married(Mary,Mary),
Married(Mary,Tom) }

A *Herbrand interpretation* for a set of sentences is an interpretation whose domain is the Herbrand universe and in which constants and functions are assigned to "themselves". Since, for Herbrand interpretations, the assignments of constants and functions is fixed, a Herbrand interpretation is identified by a subset of those elements of the Herbrand base, which are true under this particular interpretation.

A *Herbrand model* for a set of sentences is a Herbrand interpretation in which all the sentences are true.

As an example, consider the set of clauses

Path(a,b)
Path(b,c)
Path(X,Y) ← Path(X,Z), Path(Z,Y)

The Herbrand base for this example is

{ Path(a,a), Path(a,b), Path(a,c), Path(b,a), Path(b,b), Path(b,c),
Path(c,a), Path(c,b), Path(c,c) }

The Herbrand interpretation { Path(a,a), Path(b,b) } is not a model for the set of clauses, whereas the Herbrand interpretation { Path(a,b), Path(b,c), Path(a,c) } is a (minimal) model for this set of clauses.

2.3 Resolution Refutation Procedure

Given a set of definite clauses P and query clause F, one can try to mechanically determine if P logically implies F. Perhaps the most standard solution to this problem is provided by the so called *resolution refutation*. The resolution refutation procedure concludes that F is logically implied by P if it derives a contradiction represented by the empty formula and denoted by □.

Its variants are widely used in contemporary theorem proves, logic programs, and - as one can guess - in deductive databases. The basic step in the resolution refutation procedure is to select a negative literal from query F and unify it with the positive literal of a clause from set P. The unification process is a generalization of the parameter binding process during execution of procedure calls in programming languages like Pascal. The selected literal is then replaced by the body of the program clause and then the substitution involved in the unification process is applied to F. For example consider the clause

$$P(X, Y) \leftarrow Q(X, Z), R(Z, Y)$$

and the query

$$\leftarrow P(a, W), Q(U, V)$$

The literal $P(a, W)$ in the query unifies with the positive literal of the clause $P(X, Y)$ with the substitution a for X and W for Y. On resolving we get the following modified query

$$\leftarrow Q(a, Z), R(Z, W), Q(U, V)$$

This process is continued till the empty formula is derived, in which case the negation of the query clause is a consequence of the program. We shall see examples of this process in a later section.

3 Deductive Databases

Deductive databases may be seen as generalizations of relational databases where, in addition to facts, general rules are allowed to be a part of the database. Let us consider the familiar suppliers and parts database in which the relation S represents information about suppliers, P represents information about parts, and SP represents information about which supplier supplies which part, and in what quantity. A possible instance of this database is

S

S1	NYC
S2	NYC
S3	DC

P

P1	Nut
P2	Bolt
P3	Rivet

SP

S1	P1	100
S1	P2	400
S2	P1	400
S3	P3	250

This relational database can be made into a deductive database by introducing the following rules:

$$\text{NYC\_S(X)} \leftarrow \text{S(X,'NYC')}$$
$$\text{P1\_S(X)} \leftarrow \text{SP(X,'P1',Y)}$$

One can query the deductive database based on the relations S, P, SP (base relations) and NYC_S, P1_S (virtual relations). As the above example suggests, deductive database strictly subsume relational databases and hence are more expressive.

Formally, a deductive database is a system of the form

$$< \Sigma, \Re >$$

where Σ is a finite set of clauses and $\Re$ is a finite set of meta-rules for negative (or, in more general case, non-positive) inferences. Σ, as it has been mentioned in Section 1, is comprised of ground literals (extensional part of a database) and other clauses (intensional part). If Σ contains exclusively definite clauses then $< \Sigma, \Re >$ is called *definite*. Otherwise, it is called *indefinite*. $\Re$ describes how negative conclusions may be derived from Σ. Having in $\Re$ suitable rules for negative inferences, one does not have to represent explicitly the negative information in Σ (although integrity constraints may take a form of purely negative clauses; cf. [Jac88]). In such a case, Σ may be restricted to non-negative clauses.

3.1 Queries and Answers

Let us discuss briefly the notion of a query and its answer. Consider the following database:

supplier(S1,NYC)
supplier(S2,NYC)
supplier(S3,DC)
supplies(S1,P1)
supplies(S1,P2)
supplies(S2,P2)
supplies(S3,P1)

The query, find the supplier number and cities for suppliers who supply part P1, can be written as

$$\{< X, Y > | supplier(X, Y) \wedge supplies(X,' P1')\}$$

which can be interpreted as: find all pairs of values X and Y such that $supplier(X, Y)$ and $supplies(X, P1)$ can be derived from the database. As another example, the query: find the parts that are supplied by suppliers located in NYC, can be written as

$$\{< X > | (\exists Y)(supplier(Y,' NYC') \wedge supplies(Y, X))\}$$

which can be interpreted as: find values for variable X such that there exists a supplier Y who is located in NYC and supplies part X. The answer to the first query can be easily verified to be $< S1, NYC >$ and $< S3, DC >$ and the answer to the second query can be easily verified to be $< P1 >$ and $< P2 >$. In obtaining the answer to these queries, we have made an implicit assumption that if a fact is not present in the database, we assume it to be false. We will make this concept more clear in later sections.

Let us formalize the above notions of queries and their answers. Consider a deductive database $< \Sigma, \Re >$, where Σ is a collection of clauses and $\Re$ is a finite set of meta-rules for negative inferences. A *query* is a specification of the form

$$Q = \{< X_1, \ldots, X_n > | (\exists Y_1) \cdots (\exists Y_m) W(X_1, \ldots, X_n, Y_1, \ldots, Y_m)\}$$

and can be interpreted as:

> Find values for the variables X_is such that there exists values for Y_is which satisfy the formula W.

This query is equivalent to the following definite clause:

$$ANSWER(X_1, \ldots, X_n) \leftarrow W(X_1, \ldots, X_n, Y_1, \ldots, Y_m).$$

Let us denote the set of all negative facts asserted by the meta-rules of $\Re$ by $NEG(\Sigma)$. Let $\overline{c_i}$ denote an n-tuple of constant symbols and $\overline{Y}$ denote $Y_1, \ldots, Y_m$. Then $\{\overline{c_1}, \ldots, \overline{c_n}\}$ is an answer to the query Q with respect to the database $< \Sigma, \Re >$ if

$$\Sigma \cup NEG(\Sigma) \vdash ((\exists \overline{Y}) W(\overline{c_1}, \overline{Y}) \vee \cdots \vee (\exists \overline{Y}) W(\overline{c_n}, \overline{Y})).$$

An answer A to a query Q is said to be *minimal* if no proper subset of A is an answer to Q. We use the notation $\overline{c_1} + \cdots + \overline{c_n}$ instead of the set notation where + indicates disjunction. An answer $\overline{c_1} + \cdots + \overline{c_n}$ is said to be *definite* if $n = 1$, otherwise it is *indefinite*.

3.2 Definite Deductive Databases

As it has been indicated earlier, a definite deductive database consists of definite clauses. To be more precise, in a definite deductive database all answers to queries are definite, which makes the process of their calculation simple. In this section, we describe two approaches to evaluating queries in a definite deductive database: a top-down approach using the resolution refutation procedure, and a bottom-up approach using the relational algebra.

Let us consider the Bill of Materials example. We can represent the sub-component relationship using the following two definite clauses:

$$sub - component(X, Y) \leftarrow component(X, Y)$$
$$sub - component(X, Y) \leftarrow component(X, Z), sub - component(Z, Y)$$

where component(X,Y) means "Y is a component of X" and sub-component(X,Y) means "Y is a sub-component of X". Let us assume the following facts to be true for the component relationship:

$$component(P_1, P_2)$$
$$component(P_1, P_3)$$
$$component(P_1, P_4)$$
$$component(P_2, P_5)$$
$$component(P_2, P_6)$$
$$component(P_4, P_7)$$
$$component(P_4, P_8)$$
$$component(P_7, P_9)$$
$$component(P_7, P_{10})$$

Now, let us consider the query: Find all the subcomponents of part P_4, which can be represented as the definite clause

$$Answer(X) \leftarrow sub - component(P_4, X)$$

or the query clause

$$\leftarrow sub - component(P_4, X)$$

Top-down Approach: Using the resolution refutation procedure on the query clause we obtain the following derivations:

1. Query clause: $\leftarrow sub - component(P_4, X)$
   Program clause: $sub - component(X_1, Y_1) \leftarrow component(X_1, Y_1)$
   Unification substitution: P_4 for X_1 and X for Y_1
   Modified query clause: $\leftarrow component(P_4, X)$
   Program clause: $component(P_4, P_7) \leftarrow$
   Unification substitution: P_7 for X
   Modified query clause: $\square$
   Answer to query: $X = P_7$

2. Query clause: $\leftarrow sub - component(P_4, X)$
   Program clause: $sub - component(X_1, Y_1) \leftarrow component(X_1, Y_1)$
   Unification substitution: P_4 for X_1 and X for Y_1
   Modified query clause: $\leftarrow component(P_4, X)$
   Program clause: $component(P_4, P_8) \leftarrow$
   Unification substitution: P_8 for X
   Modified query clause: $\square$
   Answer to query: $X = P_8$

3. Query clause: $\leftarrow sub - component(P_4, X)$
   Program clause: $sub-component(X_1, Y_1) \leftarrow component(X_1, Z_1), sub-component(Z_1, Y_1)$
   Unification substitution: P_4 for X_1 and X for Y_1
   Modified query clause: $\leftarrow component(P_4, Z_1), sub - component(Z_1, X)$
   Program clause: $component(P_4, P_7) \leftarrow$
   Unification substitution: P_7 for Z_1
   Modified query clause: $\leftarrow sub - component(P_7, X)$
   Program clause: $sub - component(X_1, Y_1) \leftarrow component(X_1, Y_1)$
   Unification substitution: P_7 for X_1 and X for Y_1
   Modified query clause: $\leftarrow component(P_7, X)$
   Program clause: $component(P_7, P_9) \leftarrow$
   Unification substitution: P_9 for X
   Modified query clause: $\square$
   Answer to query: $X = P_9$

4. Query clause: $\leftarrow sub - component(P_4, X)$
   Program clause: $sub-component(X_1, Y_1) \leftarrow component(X_1, Z_1), sub-component(Z_1, Y_1)$
   Unification substitution: P_4 for X_1 and X for Y_1

Modified query clause: $\leftarrow component(P_4, Z_1), sub-component(Z_1, X)$
Program clause: $component(P_4, P_7) \leftarrow$
Unification substitution: P_7 for Z_1
Modified query clause: $\leftarrow sub-component(P_7, X)$
Program clause: $sub-component(X_1, Y_1) \leftarrow component(X_1, Y_1)$
Unification substitution: P_7 for X_1 and X for Y_1
Modified query clause: $\leftarrow component(P_7, X)$
Program clause: $component(P_7, P_{10}) \leftarrow$
Unification substitution: P_{10} for X
Modified query clause: $\square$
Answer to query: $X = P_{10}$

So the answers to the query are: P_4, P_7, P_9, and P_{10}.
Bottom-up Approach: The facts in the database can be represented by the relation

component

P_1	P_2
P_1	P_3
P_1	P_4
P_2	P_5
P_2	P_6
P_4	P_7
P_4	P_8
P_7	P_9
P_7	P_{10}

and the sub-component relation can be computed by solving the following relational algebraic equation:

$$sub-component = component \cup \pi_{1,4}(\sigma_{2=3}(component \times sub-component))$$

The answer to the query can be obtained by evaluating the following relational algebraic expression:

$$answer = \pi_2(\sigma_{1=P_4}(subcomponent))$$

Doing so, we obtain the following answer:

answer
P_7
P_8
P_9
P_{10}

The bottom up approach yields all answers to queries at the same time, whereas the top down approach obtains one answer at a time. As a result, in the presence of a large number of facts and rules, the top down approach is very inefficient. On the other hand, bottom up approach is not (yet) applicable in a general case of deductive databases. There has been some work in combining these two approaches to make use of the advantages of both (see [Ull88] for a detailed discussion).

3.3 Indefinite Deductive Databases

If one wants to allow facts of the form $Lives(Smith, NYC) \vee Lives(Smith, DC)$, which states that Smith lives in NYC or in DC but there is not enough information to ascertain where, and rules of the form

$$\text{father(X,Y)} \vee \text{mother(X,Y)} \leftarrow \text{parent(X,Y)}$$

which states that if X is a parent of Y then X is either the father of Y or the mother of Y, then what one needs is an *indefinite deductive database*, also known under the name of *database with incomplete information*.

Let us consider the following indefinite deductive database:

$sibling(Gary, Chris) \leftarrow$
$sibling(Chris, Gary) \leftarrow$
$sibling(Pat, Mark) \leftarrow$
$sibling(Mark, Pat) \leftarrow$
$sibling(Liz, Pat) \leftarrow$
$sibling(Pat, Liz) \leftarrow$
$sibling(Mark, Craig) \vee sibling(Mark, Don) \leftarrow$
$sibling(Craig, Mark) \vee sibling(Don, Mark) \leftarrow$
$male\_ancestor(Mark, Tom) \leftarrow$
$male\_ancestor(Sam, James) \leftarrow$
$female\_ancestor(Pat, Tom) \leftarrow$
$male\_ancestor(Chris, Tom) \vee female\_ancestor(Chris, Tom) \leftarrow$
$ancestor(X, Y) \leftarrow male\_ancestor(X, Y)$
$ancestor(X, Y) \leftarrow female\_ancestor(X, Y)$

and the query: Find all the siblings of the ancestors of Tom. Based on the semantics of the indefinite deductive database, the answer to this query is

$$\{< Pat >, < Liz >, < Gary >, < Craig > + < Don >\}$$

The notion of answer to a query in such a database is different. It has been introduced and discussed in [Lip77, Rei78a, Rei84]. The semantics of indefinite deductive databases and a top-down approach to answer queries, called SLI resolution procedure, has been defined in [MR89]. However, efficient and computationally feasible procedures to compute answers to queries in indefinite deductive databases is still a topic for future research. Some work has already been reported. Henschen and Park [HP88] compute yes/no answers to queries posed over an indefinite deductive database, Grant and Minker [GM86] provide an algorithm to check if a candidate answer to a query over an indefinite deductive database is in fact an answer and develop an algorithm to find all minimal answers to a query, and Liu and Sunderraman [LS90] generalize the relational model to represent indefinite information and provide an extended relational algebra to compute answers to queries.

4 Closed World Assumptions

As it has been mentioned in the Introduction, the closed world assumption constitutes a landmark, from which the area of deductive databases begins.

Therefore the proper understanding of this concept is a necessary prerequisite for anyone seriously interested in the theory or applications of deductive databases. In this section, we provide the reader with an insight into the nature of the closed world assumption without going into unnecessary technicalities.

It was perhaps a need for efficient representation of information in a database that resulted in the emergence of the closed world assumption. For the purpose of keeping this representation as compact as possible a choice of appropriate language is desired, and indeed there is a quite natural criterion of selection one usually adopts in such circumstances: entries with high information content, as being the most likely (or desired) candidates for storing in a database, should have possibly the simplest syntactic form. In case of predicate languages, customarily used for symbolic representation of information, this simplest form is materialized by atomic (and hence *positive*) statements. This natural convention, if successfully applied, has certain important consequences regarding the form of objectively true sentences expressible in the selected language. Since the amount of information in a statement is inversely proportional to its probability, or frequency of occurrence, it is a *negative* form of information that prevails. This is an idiosyncrasy of the very particular choice of the representation language, and would not necessarily hold if another selection criterion was utilized. It causes that a hypothetical database containing a complete description of the real world in question consists of a few positive statements in a deluge of negative information. Once the database designer realizes this fact, the solution to the problem of efficient representation becomes clear: only the positive statements are physically represented in the database, while all the negative ones are implicitly assumed *by default*. This makes the essence of the closed world assumption in the sense of Reiter.

Formally, the closed world assumption may be defined as follows:

> If an atomic sentence ϕ is not entailed by the information contained in a database then assert $\neg\phi$.

Example 4.1 Suppose that the world of discourse is a collection of suppliers of raw materials and pending orders of such materials. A sample language suitable for representing the state of affairs in our world of discourse contains one binary predicate symbol SUPPLIES and one 5-ary predicate symbol ORDER, 7 constant symbols $S_1, S_2, S_3, P_1, P_2, P_3, P_4$, and a collection of numeric

constant symbols. $SUPPLIES(S, P)$ means "Supplier S supplies part P" and $ORDER(N, P, S, Q, D)$ means "There is an order with order number N for part P with supplier S for quantity Q with a delivery date of D". Let us consider the following content of a database:

$$\begin{array}{l} SUPPLIES(S_1, P_1) \\ SUPPLIES(S_1, P_2) \\ SUPPLIES(S_2, P_2) \\ SUPPLIES(S_2, P_3) \\ SUPPLIES(S_3, P_3) \\ ORDER(1, P_2, S_2, 100, 1/30/90) \\ ORDER(2, P_2, S_2, 150, 6/30/90) \\ ORDER(3, P_3, S_2, 750, 3/15/90) \\ ORDER(4, P_3, S_2, 750, 9/15/90) \\ ORDER(5, P_3, S_2, 500, 12/15/90) \end{array}$$

The statement "There is a supplier who supplies both P_1 and P_2" (namely S_1) is a logical consequence of the database. The statement "P_1 has not been ordered" is not a logical consequence of the database, however, it follows from the database under the closed world assumption. This coincides with the natural interpretation of the content of the database. To store all the negative information implied by the closed world assumption one has to add negative statements of the form

$$\neg SUPPLIES(X, Y)$$

and a prohibitively large number of statements of the form

$$\neg ORDER(X, Y, Z, U, V).$$

Besides the waste of database memory, the usefulness of statements like

$$\neg ORDER(7348, R_3, S_1, 57942, 1/29/90)$$

is at the least doubtful, since its information content is next to zero.

If our database is supposed to admit information about the current status and timely realization of orders then a new unary predicate symbol $DELAY$ may be necessary, where $DELAY(X)$ means "Order with order number X is delayed". If we assume that $DELAY(2)$ is an entry in the database then under the closed world assumption we can infer $\neg DELAY(1)$,

$\neg DELAY(3)$, etc., without the necessity of representing these facts in the database. In the case where most of the orders are subject to a delay the statement $\neg DELAY(1)$, if true, is more informative than $DELAY(1)$, and therefore a different extension of the database is more appropriate: instead of $DELAY$, another unary predicate symbol $ON-TIME$ may be used. Although $ON-TIME(1)$ has, formally, the same meaning as $\neg DELAY(1)$, in the case where the order with order number 1 is the only order not subject to a delay (which means that the probability of delay is, say, around 80%), under the closed world assumption one statement $ON-TIME(1)$ substitutes for 4 statements $DELAY(2), \ldots, DELAY(5)$. *
□

The above example clearly shows that for the purpose of effective minimization of the size of symbolic representation, the choice of representation language has to depend on the probability distribution in the space of representable events.

Reiter's closed world assumption is, in the case of a reasonable choice of representation language, equivalent to asserting *by default* facts which are most likely to happen. This natural scheme, however, is known to lead to a contradiction when applied to indefinite databases. For example, applying the closed world assumption to a database consisting of one entry

$$SUPPLIES(S_2, P_1) \vee SUPPLIES(S_3, P_1)$$

one can infer both $\neg SUPPLIES(S_2, P_1)$ (since $SUPPLIES(S_2, P_1)$ is not contained in the database) and $\neg SUPPLIES(S_3, P_1)$ (same reason as before), or by de Morgan's law

$$\neg(SUPPLIES(S_2, P_1) \vee SUPPLIES(S_3, P_1)),$$

a conclusion that clearly contradicts the actual content of the database. Therefore, the proper treatment of *default* negative information requires a more subtle technique for indefinite databases.

One of the best proven methods of avoiding inconsistency in a formal system is to provide it with a precise meaning, so that the formal deductions allowed in this system preserve all possible meanings for the premises used in such deductions. This concept is known in linguistics (and in logic) under the name of semantics. In particular, any first-order language, including the

one of the example in this section, has its first-order semantics. Finding an appropriate semantics for the closed world assumption is our first step towards its consistent generalization over indefinite databases.

Let us take a closer look at the semantic effects of applying the closed world assumption to a definite database, DB of the form $\{P(C_1, \ldots, C_k), P(D_1, \ldots, D_k), \ldots\}$. What the closed world assumption says is that the only tuples which belong to the relation P are those explicitly specified in DB, i.e. $(C_1, \ldots, C_k)$, $(D_1, \ldots, D_k), \ldots$, etc. Of course, DB itself logically entails that these tuples are in P, but it does not entail, in case of absence of negative information, that no other tuples (namely, those not specified in DB, e.g. $(A_1, \ldots, A_k)$, are not in P. Thus the closed world assumption + DB describes (unambiguously) P as the least relation containing all the tuples listed in DB. From this observation we conclude that the closed world assumption restricts the meaning of the database in question to collection of relations that are minimal (in the sense of set-theoretic inclusion $\subseteq$). This minimization is indeed the semantics of the closed world assumption which will allow for its generalization over indefinite databases. The following definition of minimal entailment provides the intentional semantics of this generalization.

> A clause ϕ is *minimally entailed* by a database if and only if ϕ is true in all minimal models of DB

Example 4.2 The database

$$SUPPLIES(S_1, P_1)$$
$$SUPPLIES(S_2, P_1)$$
$$SUPPLIES(S_2, P_2)$$
$$SUPPLIES(S_2, P_2)$$
$$SUPPLIES(S_3, P_3)$$

has 2^7 models (i.e. 2^7 different S_is and P_js, i,j = 1,2,3, satisfy all statements of this database). For example the set

$$\text{ALL} = \{< S_i, P_j > | i, j = 1, 2, 3\}$$

of all combinations of S_is and P_js is a model of this database. Exactly one of them, namely

$$\{< S_1, P_1 >, < S_2, P_1 >, < S_2, P_2 >, < S_2, P_3 >, < S_3, P_3 >\}$$

is the minimal model of the database. The clause $\neg SUPPLIES(S_1, P_3)$ follows from this database under the closed world assumption. This clause is true in the minimal model of this database, i.e. it is minimally entailed by this database. However, this clause is not true in certain (non-minimal) models of this database, e.g. it is not true in the model ALL mentioned above. Therefore $\neg SUPPLIES(S_1, P_3)$ is not entailed logically by this database. * □

Example 4.3 The database

$$SUPPLIES(S_1, P_1)$$
$$SUPPLIES(S_2, P_2)$$
$$SUPPLIES(S_3, P_3)$$
$$SUPPLIES(S_2, P_1) \vee SUPPLIES(S_2, P_3)$$

has 3×2^7 models, e.g. the set ALL of the previous example is a model of this database. Two of them, namely

$$\{< S_1, P_1 >, < S_2, P_1 >, < S_2, P_2 >, < S_3, P_3 >\}$$

and

$$\{< S_1, P_1 >, < S_2, P_2 >, < S_2, P_3 >, < S_3, P_3 >\}$$

are minimal models of this database. The clauses $\neg SUPPLIES(S_2, P_1)$ and $\neg SUPPLIES(S_2, P_3)$ follow from this database under the closed world assumption. Neither of them, however, is minimally entailed by this database: $\neg SUPPLIES(S_2, P_1)$ is false in the first minimal model (because $SUPPLIES(S_2, P_1)$ is true in this model), and $\neg SUPPLIES(S_2, P_3)$ is false in the second minimal model of this database. * □

The minimal entailment *makes* the semantic equivalent of the closed world assumption. It does not seem, however, at least as it is, particularly useful substitute for the closed world assumption. One can expect that the combinatorial explosion of models practically precludes an efficient method of enumeration of all the minimal models of a database, making the problem: "Is ϕ true in all minimal models of the database" rather hard for direct verification. As in the case of logical entailments, finding appropriate syntactic

characterizations of this concept by means of a proof procedure may bring us closer to a computationally feasible solution of this problem.

The question of a pure syntactic characterization of minimal entailment remained open for almost a decade. Although a generalized closed world assumption GCWA proposed by Minker provided a partial answer to this question, and has been known to avoid the inconsistency paradox of the closed world assumption, it did not completely solve the problem of characterization for disjunctive or quantified queries. It has been demonstrated (Shepherdson) that for purely relational language (no function symbols and no equality symbol) and for any literal A in this language, the following property holds:

> A is true in all discriminant models of deductive database D iff GCWA + D proves A

but for disjunctive statements the above characterization is no longer true. Finally, this problem received a surprisingly simple answer in terms of certain modification cwa_S of the closed world assumption, introduced by Suchenek in [Suc87]. Since, as has been observed before, representing only positive information in a database has been recognized as an efficient means for minimization of the physical size of the database, one can suspect that any positive statement not derivable from the database was not intended as its implicit consequence. Of course, it was a matter of proof (see [Suc89]), nevertheless it seems intuitively acceptable that the following version cwa_S of the closed world assumption provides a complete syntactic characterization of the minimal entailment:

> If adding clause ϕ to a database DB does not enlarge the set of positive logical consequences of DB then assert ϕ.

One may show by not so difficult inspection that for quantifier-free clause ϕ and definite database DB, cwa_S+DB proves ϕ if and only if $cwa+DB$ proves ϕ. For more complex clauses and for indefinite databases, this equivalence is no longer true. In particular, cwa_S, as having semantic counterpart (namely, the minimal entailment), may never lead to a contradiction. This is an important feature that Reiter's cwa does not possess. It also has been shown in [Suc89] that Minker's GCWA may be equivalently expressed by restricting the scope of the clause ϕ in the definition of cwa_S to atomic and negated atomic sentences.

Example 4.4 Let us recall the example of a database DB

$$SUPPLIES(S_2, P_1) \vee SUPPLIES(S_3, P_1)$$

We demonstrated that reiter's cwa is inconsistent, namely the database under the cwa proves both $\neg SUPPLIES(S_2, P_1)$ and $\neg SUPPLIES(S_2, P_1)$. Let us see why cwa_S does not lead to a contradiction when applied to this database. Is $\neg SUPPLIES(S_2, P_1)$ implied by DB under cwa_S? To answer this question, we have to consider positive logical consequences of $DB' = \neg SUPPLIES(S_2, P_1) + DB$. We have DB' logically implies

$$\neg SUPPLIES(S_2, P_1) \wedge (SUPPLIES(S_2, P_1) \vee SUPPLIES(S_3, P_1)),$$

that is to say, DB' implies $SUPPLIES(S_3, P_1)$. This is a positive clause which is not derivable from DB itself. Therefore, by the definition of cwa_S, $DB + CWA_S$ does not assert $\neg SUPPLIES(S_2, P_1)$. Similarly, one can verify that $cwa_S + DB$ does not assert $\neg SUPPLIES(S_3, P_1)$ either. The inconsistency caused by Reiter's cwa has disappeared. * □

The closed world assumption in any of its forms discussed above (cwa, GCWA, cwa_S) minimizes evenly all relation symbols of the database's language, which is not particularly desirable if the database in question consists of conceptual layers built one upon another. If, for example, deductive database is a result of translation of certain relational database with incorporated *views*, then the image of the physical level and images of these views may not necessarily need a uniform treatment. To the contrary, one can expect that relations which belong to the images of more primitive views should be minimized before the others. This observation brings us to the problem of *prioritized* minimization.

There are several possible approaches to incorporate priorities into minimal model semantics. In some logic programs, as we will see in section 5, such priorities are implicitly introduced by certain *preference* relation, defined in class of minimal models, induced by a form of logic program in question. In more general artificial intelligence context, *prioritized circumscription* tackles this problem from position of power, or in other words, second order logic.

Here, we demonstrate how cwa_S may be successfully used to allow for prioritized minimization. We start from a modified example from Clark's paper [Cla78].

Example 4.5 Let D be a deductive database with the extension:

$$\begin{array}{ll} D_E = \{ & \text{student(J.Brown)} \leftarrow \\ & \text{student(D.Smith)} \leftarrow \\ & \text{takes(J.Brown,C101)} \leftarrow \\ & \text{takes(D.Smith,C101)} \leftarrow \\ & \text{takes(D.Smith,C301)} \leftarrow \\ & \text{math\_course(C101)} \leftarrow \\ & \text{math\_course(C301)} \leftarrow \} \end{array}$$

(which may be interpreted as an image of the physical level of a relational database) and the intension:

$$\begin{array}{c} D_I = \{non\_math\_major(X) \vee takes(X,Y) \leftarrow \\ student(X) \wedge math\_course(Y)\} \end{array}$$

(which may be interpreted as an image of a view of this relational database), i.e.

$$D = D_E \cup D_I.$$

D_E lists students, courses and enrollments. D_I says that a student who does not take all listed maths courses is a non-math major. This database has two classes of minimal models: one, in which non_math_major(J.Brown) is true and takes(J.Brown,C301) is false, and the other, in which non_math_major(J.Brown) is false and takes(J.Brown,C301) is true. It seems clear that only the first group of models is adequate to intuitive comprehension of the content of the database D. In particular, one can expect that D implies non_math_major(J.Brown). However, D does not minimally entail non_math_major(J.Brown) since this sentence is false in the second class of minimal models of D. Therefore, $cwa_S + D$ does not prove non_math_major(J.Brown). * □

If a deductive database in question is indeed a result of translating a physical level of relational database with several views implemented on it then the proper cure of the above situation is quite simple: first, the image of the physical level should be minimized, and only after that images of views should undergo subsequent minimization. To achieve this effect using cwa_S one has to apply it several times to the appropriately layered database, starting from the core, corresponding to the physical level, and proceeding from inner to

outer layers. In the above example this proper mode of prioritized minimization is to first minimize simultaneously predicates student, takes and math_course, and after that to minimize predicate non_math_major. This kind of minimization is achieved by a double application of cwa_S:

$$cwa_S(D_I \cup cwa_S(D_E)).$$

Let us verify that this proves non_math_major(J.Brown). First, let us show that $cwa_S(D_E)$ proves $\neg takes(J.Brown, C301)$. Indeed, all positive ground clauses provable from $D_E \cup \{\neg takes(J.Brown, C301)\}$ (within the language of D_E) are already provable from D_E. Since in this model, universal quantification $\forall X \phi(X)$ is equivalent to

$$\phi(J.Brown) \wedge \phi(D.Smith) \wedge \phi(C101) \wedge \phi(C301),$$

from this we infer that *all* positive clauses (not only ground ones) provable from

$$D_E \cup \{\neg takes(J.Brown, C301)\}$$

are already provable from D_E. Now, $D_I \cup cwa_S(D_E)$ proves non_math_major(J.Brown), because

$$D_I \cup D_E \cup \{\neg takes(J.Brown, C301)\}$$

does. Therefore, $cwa_S(D_I \cup cwa_S(D_E))$ proves the same statement. It may be noted that the second application of cwa_S, which ensures minimization of predicate non_math_major, was actually not necessary for proof of non_math_major(J.Brown). It is necessary, however, to prove $\neg non\_math\_major(J.Brown)$.

In a more complex scenario, appropriate precaution should be taken while minimizing layers of deductive databases. For example, if D is an image of a relational database of the form

xxx
xxx

say, D_0 is an image of the physical level, and D_1, D_2, D_3 are images of $view_1, view_2, view_3$ respectively, then appropriate prioritized minimization of D is achieved by

$$cwa_S(D_3 \cup (cwa_S(D_1 \cup cwa_S(D_0)) \cup cwa_S(D_2 \cup cwa_S(D_0)))).$$

Constructions of the above form, although intuitively obvious, are currently under investigation and require further research.

Not surprisingly, cwa_S is computationally more complex than the cwa, and in the general case of large indefinite databases there is little hope that the question "Does ϕ enlarge the set of positive logical consequences of a database" will ever be answered (After all, reasoning under cwa leads to undecidable problems). What is more surprising is that the GCWA, although seemingly much simpler than cwa_S (only atomic and negated atomic clauses ϕ are considered in GCWA), is essentially of the same order of complexity as cwa_S. This seems to be a common disadvantage of many reasoning systems: even PROLOG resolution refutation procedure may never stop if the order of the disjunct in a definition clause is inappropriate. It does not preclude, however, any practical use of PROLOG, and the same may be said about cwa_S. As a matter of fact, there is some promising progress toward the utilization of resolution principle on the one hand (cf. [MR89]), and model-theoretic forcing (cf. [Suc89]) on the other, for the purpose of providing cwa_S and GCWA with reasonably efficient operational approximations. These topics, however, remain out of the scope of the present paper.

5 Deductive Databases vs Logic Programming

The areas of deductive databases and logic programming are so similar that the relationship between them require a special mention. These two phenomena grew from different grounds, however, and there used to be separate research activities pertinent to each of them. Recently, there has been a substantial interplay between these two, as the inspirations, ideas, and results from one area infiltrate the other. This migration of methodology caused certain confusion among the general audience, which has been amplified by a lack of clear widely accepted taxonomy. In particular, the rather extensive use of the term "logic" in the context of deductive databases, as in "logic databases", contributed to the not so uncommon misuse of terminology. So, it is actually the subtle *difference* between them which requires an explanation. To see this problem from a proper perspective, however, let us start from the similarities.

Similarly to deductive databases, logic programs are composed of a finite number of clauses (actually deductive databases borrowed this form from

logic programming). Originally, they were required to consist exclusively of *definite* clauses [vEK76]., however, recently the *general* logic programs with *indefinite* clauses have also been considered useful [MR89]. Therefore, as far as the expressive power is concerned, there is no essential difference between logic programs and deductive databases, although a typical logic program will probably contain much fewer ground clauses (facts) than a typical deductive database. Moreover, indefinite clauses allowed in deductive databases have a different form, namely

$$A_1, \vee \cdots A_n \leftarrow B_1 \wedge \cdots B_m$$

than indefinite clauses allowed in logic programs:

$$A_1 \leftarrow B_1 \wedge \cdots B_m \wedge \neg A_2 \wedge \cdots \wedge \neg A_n$$

Similarly as in deductive databases, in logic programs negation is weaker than in first-order logic: for every positive literal, if ¬P is provable from a consistent program in the classical sense then ¬P will also be derived using any of existing sound *negation as finite failure* procedures, but not necessarily vice versa. Moreover, in the case of definite databases and logic programs, the intended meaning of negation is, as we will see, the same for both.

The last similarity brings us to one of the essential differences between logic programs and deductive databases: in the general indefinite case, their semantics do not coincide. Semantics for logic programs is operational, defined in terms of resolution with negation treated as finite failure to prove, while semantics for deductive databases is denotational, based on the notion of minimal model. As a result, negation in indefinite deductive databases gets a different meaning than in indefinite logic programs. Moreover, this difference is by no means small; semantics of indefinite logic programs *can* distinguish between logically equivalent clauses, as opposed to deductive databases, and of course to first-order logic. This is a result of different rules for inferring negative conclusions in deductive databases and logic programming.

As opposed to deductive databases, where various versions of the closed world assumption are used, in logic programs the following rule of [Cla78] is applied to derive negative ground literal ¬L:

> If the execution of program P on goal L finitely fails (i.e. it is evident that L *cannot* be achieved) then assert ¬L.

This negation as failure rule is, to all appearances, very similar to Reiter's version of cwa, where ¬L is asserted if L cannot be proved. It should be mentioned that "provability" in case of negation as finite failure means, somewhat recursively, provability using possibly negation as finite failure, while in case of cwa, it is pure first-order provability. Let us consider the following.

Example 5.1 The clause

$$Father(Pat, Gary) \leftarrow Parent(Pat, Gary) \wedge \neg Mother(Pat, Gary)$$

and the clause

$$Mother(Pat, Gary) \leftarrow Parent(Pat, Gary) \wedge \neg Father(Pat, Gary)$$

are *indefinite program clauses*, logically equivalent to the following disjunctive clause:

$$Father(Pat, Gary) \vee Mother(Pat, Gary) \leftarrow Parent(Pat, Gary)$$

and therefore equivalent to each other. The programs

$$\text{P} = \{ \quad \begin{array}{l} Father(Pat, Gary) \leftarrow Parent(Pat, Gary), \neg Mother(Pat, Gary) \\ Parent(Pat, Gary) \leftarrow \end{array} \quad \}$$

and

$$\text{Q} = \{ \quad \begin{array}{l} Mother(Pat, Gary) \leftarrow Parent(Pat, Gary), \neg Father(Pat, Gary) \\ Parent(Pat, Gary) \leftarrow \end{array} \quad \}$$

are perfectly legal indefinite logic programs. Although P and Q are logically equivalent (in the sense that one is derivable from the other within first-order logic), they do not return the same answer to the query

$$\leftarrow Father(Pat, Gary)$$

In particular, program P returns the answer "yes" and program Q returns the answer "no". In case of program Q, this query cannot be matched, or more precisely unified, with a head of any clause of P, therefore attempt of proving $Father(Pat, Gary)$ fails finitely (namely in one step). Negation as finite

failure rule yields $\neg Father(Pat, Gary)$. In case of program P, this query trivially unifies with the head of the first clause of P, which subsequently causes the sub-queries $Parent(Pat, Gary)$ and $\neg Mother(Pat, Gary)$ to be issued. Both of these sub-queries are answered "yes', the first one after trivial unification and the second one after finite failure of proving $Mother(Pat, Gary)$.

$* \ \square$

The nature of this paper does not allow us to discuss the technicalities necessary for full explanation of the above phenomenon. To give an idea, however, why this is so, consider the set of minimal models for P and Q. Both programs are equivalent to the following sentence

$$[(Father(Pat, Gary) \vee Mother(Pat, Gary) \leftarrow Parent(Pat, Gary))] \wedge Parent(Pat, Gary)$$

or, in other words, to

$$Father(Pat, Gary) \vee Mother(Pat, Gary).$$

Because equivalent sentences have the same models, in particular the same minimal models, the minimal models for P coincide with minimal models for $Father(Pat, Gary) \vee Mother(Pat, Gary)$, and of course coincide with the minimal models for Q. The entire class of minimal models for $Father(Pat, Gary) \vee Mother(Pat, Gary)$ may be split onto two disjoint classes: models in which $Father(Pat, Gary) \wedge \neg Mother(Pat, Gary)$ is true, and models in which $\neg Father(Pat, Gary) \wedge Mother(Pat, Gary)$ is true. It turns out that the first class makes the semantics of the program P, while the second one makes the semantics of the program Q. For the logically equivalent deductive database

$$\begin{array}{lcl} D & = & \{Father(Pat, Gary) \vee Mother(Pat, Gary) \leftarrow Parent(Pat, Gary) \\ & & Parent(Pat, Gary)\} \end{array}$$

the entire union of the two classes constitutes a valid semantics. This means that *a form* of a logic program encodes certain relation of preference, which favors some minimal models (e.g. those satisfying $Father(Pat, Gary) \wedge \neg Mother(Pat, Gary)$ in case of program P) over others (in this case, those satisfying $Mother(Pat, Gary) \wedge \neg Father(Pat, Gary)$). It may seem counterintuitive, or unacceptable, but it is not, as the following example borrowed from [Prz87] shows.

Example 5.2 Let us assume that Einstein is a physicist, Iacocca is a businessman and that a typical businessman avoids using mathematics unless he happens to be a good mathematician. This information can be represented as the following clauses

$$\begin{array}{rcl} avoids\_math(X) & \leftarrow & businessman(X), \neg good\_mathematician(X) \\ businessman(Iacocca) & & \\ physicist(Einstein) & & \end{array}$$

This logic program has two minimal Herbrand models. In both, Iacocca is the only businessman and Einstein is the only physicist. In one of the models, Iacocca avoids mathematics and in the other Iacocca is a good mathematician. The intended meaning of this program is not captured by both the models. Since we have no information on whether Iacocca is a good mathematician, we are inclined to infer that he avoids using mathematics. Therefore, only the first minimal model in which Iacocca avoids mathematics seems to correspond to the intended meaning of the logic program.

* □

More extensive discussion of *preferred* model semantics and variations of the above may be found in [Prz87].

Since every set of definite clauses, and therefore every definite logic program has exactly one minimal Herbrand model, the least Herbrand model, the relation of *preference* becomes irrelevant in such a case (its domain has one element to choose from). Therefore, definite logic programs and definite deductive databases are equivalent from semantic point of view. In particular, negation has exactly the same meaning in both of them.

If someone hopes that to express the meaning of a deductive database it suffices to combine all logic programs logically equivalent to the database then he or she is wrong. The following example shows that logic programs may hardly be accepted as operational implementations of indefinite deductive databases.

Example 5.3 Let us consider the deductive database

$$\begin{array}{rl} D = \{ & male(X) \vee female(X) \leftarrow human(X), \\ & human(Raj) \\ & human(Radhika)\} \end{array}$$

There are two logic programs equivalent to D:

$$P = \{ \quad male(X) \leftarrow human(X) \wedge \neg female(X), \\ human(Raj) \\ human(Radhika)\}$$

$$Q = \{ \quad female(X) \leftarrow human(X) \wedge \neg male(X), \\ human(Raj) \\ human(Radhika)\}$$

Neither of these two programs allow the clause

$$male(Raj) \wedge female(Radhika)$$

to be true. What is worse is that every computation of P and Q proves

$$male(Raj) \equiv male(Radhika)$$

and the program $R = P \cup Q$ will never stop on query containing $male$ or $female$. Of course, in certain minimal model of D, both $male(Raj)$ and $female(Radhika)$ hold.

* □

There are, of course, other differences between deductive databases and logic programs, but since negation constitutes the main subject of this paper, we will mention them only briefly.

- In deductive databases, queries with free variables, e.g. $P(X, Y)$ are answered by a set of tuples of ground terms, e.g.

  $$\{(t_1, s_1), \ldots, (t_n, s_n)\}$$

  satisfying the query. In logic programs goals are universally quantified (implicitly), therefore the only outcome of computation may be *true* or *false*. In this respect queries directed to deductive databases are strictly more expressive than goals directed to logic programs.

- In deductive databases, similar perhaps to all other kinds of databases, a separate class of issues and techniques are associated with updates, which do *not* belong to the typical domain of logic programs. Consequently, such activities, pertinent to databases, as integrity protection, are not present in logic programs.

It is not very hard to guess that all the attributes distinguishing logic programs from deductive databases pertain also to PROLOG. Moreover, because of deterministic selection of subgoals and modified unification algorithm in PROLOG, deductive databases are relatively more remote from PROLOG programs than from logic programs. A more detailed analysis of relevant differences involving excessive amount of technicalities is well beyond the scope of this paper. It should be mentioned, however, that this language has been successfully used in implementations of various (not necessarily deductive) databases [Li84].

6 Perspectives

6.1 History

In this section we will briefly describe some of the important landmarks in the development of the fields of deductive databases and logic programming, with certain emphasis on achievements contributing to the proper treatment and understanding of negation.

In 1965, Robinson [Rob65] introduced the *resolution principle*, a uniform method for performing automated deduction. This principle is used by automated deduction systems, the so called theorem provers. Using the resolution principle, Green and Raphael [GR68] designed the system QA3.5 which showed the viability of implementing deductive databases in a uniform manner. Then, in 1977, the workshop on Logic and Databases, [GM78], produced a number of important papers that established deductive databases as a legitimate field on its own. Clark's paper on negation as failure and completed database, [Cla78], introduced the meta-rule, *negation as finite failure*, to solve the problem of inferring negative information from a deductive database, in a logic programming context. In this paper, deductive databases were treated as a special case of logic programs with a large number of facts and only few rules. This conjecture has not been widely accepted: as we have seen there are essential differences between deductive databases and logic programs (cf. [Aptar], section 8.3, deductive databases). Consequently negation as failure and its denotational equivalent: program completion have finally found their place in logic programming. Reiter's paper on the *closed world assumption*, [Rei78b], discussed the notion of inferring negative infor-

mation from deductive databases with clear motivations from the relational model of a database. He also introduced the notion of a query and answers to queries in deductive databases there. The closed world assumption may be understood as the database counterpart of Clark's negation as failure rule.

In the logic programming front, Kowalski proposed the use of logic as a programming language in [Kow74]. The paper by van Emden and Kowalski, [vEK76], introduced the notion of unique minimal model of a set of definite clauses and used a fix-point operator to characterize the meaning of definite logic programs. In 1982, Minker introduced the *generalized closed world assumption*, [Min82], which extends the closed world assumption over indefinite deductive databases. Apt and van Emden, [AvE82], characterized linear resolution with selection function in definite logic programs with negation as finite failure (SLDNF) in terms of the fix-point operator. Jaffer, Lassez and Lloyd proved the completeness of negation as finite failure in [JLL83]. Reiter, [Rei84], described formal theories of deductive databases. Lloyd and Topor use Prolog as a query evaluator in [LT84]. Shepherdson, in [She88], provides a proof of the relationships between CWA, GCWA, and minimal semantics. A problem of syntactic characterization of minimal semantics, implicit target of all above mentioned constructs (closed world assumptions, finite failure) has been recently solved in [Suc89]. An excellent source of exhaustive introduction to logic programming is [Aptar]. The book by Lloyd [Llo87] is another source of the developments in the field of logic programming.

Although most of the work in deductive databases and logic programming has been done over the last two decades, the notion of closed world assumption has been used implicitly in automated reasoning systems well before deductive databases. In particular, FORTRAN compilers (late fifties) actually used it (without mentioning its name, of course) while solving a system of equivalences defined by EQUIVALENCE statements.

6.2 Experimental Deductive Database Systems

At present there are no commercial deductive database systems. However, there are a number of experimental projects that may result in commercial systems in the near future. We shall very briefly describe three such projects.

NAIL! (Not Another Implementation of Logic!) is an experimental deductive database system under development at Stanford University. The system restricts the clauses or rules to be definite, however negative literals may

appear in the body of the rule with certain restrictions. The user may ask queries of the deductive database using the query clause. More information can be obtained from [MUG86], and [Ull85].

LDL (Logic Data Language) is another deductive database system and language under development at MCC, in Austin, Texas. Like NAIL!, LDL also restricts the clauses or rules to be definite while allowing negative literals in the body of the rules under certain restrictions, however, LDL is a more comprehensive system which includes a database management system as well as a logic processor. Some references for LDL are [NT88], [Zan88], and [COK+87].

POSTGRES is a successor to the INGRES project under development at the University of California at Berkeley. Its objective is to extend INGRES in a natural way to be capable of reasoning with definite rules. Its query language is an extension of QUEL called POSTQUEL. Additional information can be found in [SR86a], [SR86b] and [Wen88].

The two volume set of books on the principles of database and knowledge-base systems by Ullman [Ull88] contain detailed discussion on the various techniques used in most of these systems.

7 Summary

References

[Aptar] Krzysztof R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North Holland, To appear.

[AvE82] Krzysztof R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *JACM*, 29:841–862, 1982.

[Cla78] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.

[COK+87] D. Chimenti, T. O'Hare, R. Krishnamurthy, S.A. Naqvi, S. Tsur, C. West, and C. Zaniolo. An overview of the LDL system. In C. Zaniolo, editor, *Data Engineering, 10:4*, pages 52–62. 1987.

[GM78] H. Gallaire and J. Minker. *Logic and Data Bases*, volume 0. Plenum Press, New York, 1978.

[GM86] John Grant and Jack Minker. Answering queries in indefinite databases and the null value problem. In P. Kanellakis, editor, *Advances in Computing Research, Volume 3*, pages 247–267. JAI Press, Greenwich, CT, 1986.

[GR68] Cordell C. Green and Bertram Raphael. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 23rd National Conference ACM, Washington*, 1968.

[HP88] Lawrence Henschen and Hyuing-Sik Park. Compiling the GCWA in indefinite deductive databases. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 395–438. Morgan Kaufmann, Los Altos, CA, 1988.

[Jac88] Peter Jackson. On game-theoretic interactions with first-order knowledge bases. In Philippe Smets, Abe Mamdani, Didier Dubois, and Henri Prade, editors, *Non-Standard Logics for Automated Reasoning*, pages 27–54. Academic Press, London, 1988.

[JLL83] J. Jaffer, J. Lassez, and J.W. Lloyd. Completeness of the negation as failure rule. In *Proceedings Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, 8-12 Aug. 1983*, pages 500–506, 1983.

[Kow74] Robert A. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP World Congress, Amsterdam*, pages 569–574, 1974.

[Li84] Deyi Li. *A prolog database system.* John Wiley and Sons Inc., New York, 1984.

[Lip77] Witold Lipski, Jr. On semantic issues connected with incomplete information data bases. In *Proceedings of 3rd International Conference on Very Large Data Bases*, volume 2, pages 71–83, Tokyo, October 1977.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Artificial Intelligence. Springer Verlag, New York, second extended edition, 1987.

[LS90] K.C. Liu and R. Sunderraman. Indefinite and maybe information in relational databases. *ACM Transactions on Database Systems*, 15(1):1–39, 1990.

[LT84] J.W. Lloyd and R.W. Topor. Making PROLOG more expressive. *Journal of Logic Programming*, 1(3):225–240, October 1984.

[Min82] Jack Minker. On indefinite databases and closed world assumption. In *Proceedings of 6-th Conference on Automated Deduction*, Lecture Notes in Computer Science 138, pages 292–308, Berlin, New York, 1982. Springer Verlag.

[MR89] Jack Minker and Arcot Rajasekar. A fixpoint semantics for non-Horn logic programs. *Journal of Logic Programming*, 1989.

[MUG86] K. Morris, Jeffrey D. Ullman, and A. Van Gelder. Design overview of the nail! system. In *Proceedings of the Third International Conference on Logic Programming*, pages 554–568, 1986.

[NT88] Shamim A. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge Bases*. Computer Science Press, Rockville, MD, 1988.

[Prz87] Teodor C. Przymusiński. Non-monotonic reasoning vs logic programming: A new perspective. In Y. Wilks and D. Partridge, editors, *Handbook of Formal Foundations of Artificial Intelligence*. Oxford University Press, 1987.

[Rei78a] Raymond Reiter. Deductive question-answering on relational databases. In H. Gallaire and J. Minker, editors, *Logic and Databases, V0*, pages 149–178. Plenum Press, New York, 1978.

[Rei78b] Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.

[Rei84] Raymond Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modeling*, pages 191–238. Springer-Verlag, Berlin and New York, 1984.

[Rob65] Alan J. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–44, January 1965.

[She88] John C. Shepherdson. Negation in logic programming. In *[?]*, pages 19–88. 1988.

[SR86a] Michael Stonebreaker and Lawrence Rowe. The design of postgres. In *Proceedings, ACM SIGMOD International Conference on Management of Data*, pages 340–355, 1986.

[SR86b] Michael Stonebreaker and Lawrence Rowe. The postgres papers. UCB/ERL M86/85, Dept. of EECS, University of California at Berkeley, 1986.

[Suc87] Marek. A. Suchenek. Forcing versus closed world assumption. In Zbigniew W. Raś and Maria Zemankova, editors, *Methodologies for Intelligent Systems*, pages 453–460. North-Holland, 1987.

[Suc89] Marek A. Suchenek. A syntactic characterization of minimal entailment. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming, North American Conference 1989*, pages 81–91, Cambridge, MA, October 16–20 1989. MIT Press.

[Ull85] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, September 1985.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, Rockville, Maryland, 1988.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *JACM*, 23:733–742, 1976.

[Wen88] S. Wensel. The postgres reference manual. UCB/ERL M88/20, Dept. of EECS, University of California at Berkeley, 1988.

[Zan88] Carlo Zaniolo. Design and implementation of a logic based language for data intensive applications. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1666–1687, 1988.