J. Komorowski Z.W. Raś (Eds.)

Methodologies for Intelligent Systems

7th International Symposium, ISMIS '93
Trondheim, Norway, June 15-18, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Table of Contents

Towards Intelligent Databases

François Bry

ECRC, Arabellastraße 17, 81925 München 81, Germany
Francois.Bry@ecrc.de

Abstract. This article is a presentation of the objectives and techniques of deductive databases. The deductive approach to databases aims at extending with *intensional* definitions other database paradigms that describe applications *extensionally.* We first show how *constructive* specifications can be expressed with *deduction rules,* and how *normative* conditions can be defined using *integrity constraints.* We outline the principles of bottom-up and top-down query answering procedures and present the techniques used for integrity checking. We then argue that it is often desirable to manage with a database system not only database applications, but also specifications of system components. We present such *meta-level* specifications and discuss their advantages over conventional approaches.

1 Introduction

Deductive Databases have been studied since more than a decade. Theoretical issues have been investigated (see e.g. [28, 29, 30, 31, 65, 21, 8, 48, 64, 17, 18, 44, 45] for an overview), and experimental deductive database management systems have been and are still implemented (e.g. [54, 9, 23, 26, 32, 34, 51, 56, 66, 33, 68, 40, 52]). Industrial products are currently developed from research prototypes (e.g. [69]). This article is informal presentation of the notions and objectives of deductive databases. Instead of emphasizing technical aspects (that are explained in a number of articles and tutorials, e.g. [28, 29, 30, 31, 65, 21, 8, 17, 18]), we prefer to insist on the goals of the deductive approach to databases.

A first part of the presentation is devoted to recall how two complementary notions are used in deductive databases for *declaratively* specifying an application. On the one hand, *deduction rules* are used for *constructive* definitions. On the other hand, *normative* specifications are expressed through *integrity constraints.* We informally describe how deduction rules are evaluated for answering queries (see e.g. [17, 18, 1, 2, 4, 7, 55, 57, 60, 61, 67, 13]), and how integrity constraints are checked when the database is updated (see e.g. [17, 18, 15, 25, 39, 43, 47, 49, 53, 19]).

In a second part of the presentation, we argue that it is often desirable to manage with the database system, not only an application, but also specifications of components of the database sytem itself, the description of an application, or various kinds of interpretations of this application. We informally introduce a few such *meta-level* specifications, that rely on meta-programming [58, 62, 63, 59]. Finally, we briefly mention further applications of meta-level specifications towards enhanced database management systems.

2 An Introduction to Deductive Databases

A main trend in database research is the enhancement of data modeling facilities. Deductive database techniques aim at extending conventional, nondeductive databases, in which data are *extensionally* specified, with *intensional* definitions in form of *deduction rules* and *integrity constraints.*

Database management systems historically developed from file managers, in which applications are specified in terms of records and structured according to storage and retrieval criteria. Two data models were proposed at the end of the sixties/beginning of the seventies for improving the descriptions of applications: the hierarchical and the network data models. Like a file, a hierarchical or network database consists of records. However, in contrast to files, records are structured in trees and pointers express relationships between records. Both the hierarchical and the network data models have a major drawback: The pointers these data model rely upon make the design and the querying of databases rather difficult. Database users must be aware of rather complex networks even for posing simple queries.

The relational data model, defined by Codd [24] at the end of the seventies, overcomes this difficulty in an elegant manner: no pointers are used and the conceptual links between records (called tuples) are expressed through regular data. A relational database consists in a set of relations. Relations are set of tuples. The semantical relationship between tuples are expressed through the *values* they contain. Thus, for example, the presence of a same character string (say, a name) in a tuple of a "salary" relation and in a tuple of an "address" relation links salaries, addresses, and employee's names. Because they are *value-based,* relational databases can be interpreted in mathematics as logical theories consisting of formulas or, alternatively as logical models consisting of relations. Relational databases can be seen as more *declarative* than hierarchical or network databases since less knowledge of their internal structure is necessary for querying them. Indeed the knowledge of the relation's names, the so-called database schema, and, possibly, of some values occurring in tuples, suffices for posing queries.

2.1 Deduction Rules

Deductive databases can be seen as an extension of the relational model. In a relational database, the data are specified *extensionally.* That is, the tuples of a relational database are explicitly defined. Deductive databases in contrast, also give rise to specifying data *intensionally* by means of general properties, expressed using *deduction rules.* Consider for example the time-table of the Lufthansa airline. The Lufthansa direct flights from Munich to Paris can be specified by the following "flight" relation:

Monday	0725	0900	LH4356
Tuesday	0725	0900	LH4356
Wednesday	0725	0900	LH4356
Tursday	0725	0900	LH4356
Friday	0725	0900	LH4356
Saturday	0725	0900	LH4356

Monday	1110	1245	LH4384
Tuesday	1110	1245	LH4384
Wednesday	1110	1245	LH4384
Thursday	1110	1245	LH4384
Friday	1110	1245	LH4384
...	...	...	...

The first attribute (column) of this relation indicates the day of the flight, the second and third are the departure and arrival times, respectively, and the last attribute is the flight number. These eleven flights could be specified by the following two deduction rules that somehow "factorize" the data common to several tuples:

flight(D, 0725, 0900, lh4356) $\leftarrow$ day(D), not D = sunday.
flight(D, 1110, 1245, lh4384) $\leftarrow$ day(D), not D = saturday, not D = sunday.

As usual, character strings beginning with an upper case letter (e.g. D) are used for denoting (logical) variables. The membership of a tuple (called "fact" in deductive databases) "t" in a relation "r" is expressed by the term "r(t)". We assume that "day" denotes the relation containing the seven days of the week (monday, tuesday, etc.). Lower case letters are used for distinguishing these constant values from variables. The expression "day(D)" can be thus evaluated to the facts "day(monday)", "day(tuesday)", etc. The meaning of the first rule is that the facts "flight(monday, 0725, 0900, lh4356)", "flight(tuesday, 0725, 0900, lh4356)", ..., "flight(saturday, 0725, 0900, lh4356)" are derivable, i.e. are true facts in the database. In more technical terms,the variable D is (implicitly) universally quantified. The first deduction rule is thus a shorthand notation for the following formula:

$$\forall D \quad [\ (\text{day}(D) \wedge D \neq \text{sunday}) \quad \Rightarrow \quad \text{flight}(D, 0725, 0900, \text{lh}4356)\ ]$$

This simple example illustrates two important advantages of deductive databases compared with relational ones: (1) they require less storage, and (2) they give rise to more natural specifications. The possible size reduction is sometimes dramatic: An analysis of the time table of the Munich public transportation shows for example a reduction factor of about 200! Database applications whose data cannot be specified according to general principles do not benefit as much of deductive techniques. Most databases nevertheless contain some data that were implied from general laws (e.g. business rules, legislation, scientific laws, etc.) and therefore can benefit from deductive database techniques.

One could object that no deductive techniques are needed for achieving the factorization described above. This is true. There are indeed, for this example, two alternative ways to avoid the undesirable duplication of data using relational data structures. The first approach consists in splitting the original relation in two distinct relations, the first one giving the day and the flight number (which obviously is a key), the second relation giving the times and the flight numbers. A join then permits ones to reconstruct the original relation at query time. The second approach consists in using codes like in the following table for expressing on which days a flight is available.

Xe7	0725	0900	LH4356

Xe67	1110	1245	LH4384
...	...	...	...

In this relation, X stands for every day of the week, 6 for Saturdays, 7 for Sundays, Xe7 for every day except on Sundays, and Xe67 for every week days.[1]

We argue that both approaches have severe drawbacks. The first approach (the split of the original relation in two distinct smaller relation) examplifies an often criticized (although necessary) practice in relational database design: For reasons of storage (size) and coherency of the data (when updates are performed), the natural description of an application usually needs to be modified. The two rules given above as opposed achieve the same effect without compromising the natural character of the specifications. The second approach (the encoding of the days in the tuples) is very close to a specification by means of deduction rules. The difference however is that the encoding is a notation "unknown" to the database management system, while deduction rules are "understood" by a deductive database system for what they are. Such an encoding is specific to a given application and must be interpreted in the application programs, that is *outside* the database system. Deduction rules in contrast give rise to interpreting intensional knowledge *within* the database system.

Deduction rules can also be used in lieu of *relational views*. Views are in relational databases means for expressing predefined queries. One could for example define connecting flights using a view: A connecting flight form A to B is defined from a flight from A to C and a flight from C to B such that some conditions on the departure and arrival times in C, and on the location of the airport C are satisfied. A recursive definition give rise to specifying connections involving an indefinite number of flights. Such a definition is quite naturally expressed by the following deduction rule:

```
connection(D, T1, T2, [Nb]) ← flight(D, T1, T2, Nb).
connection(D, T1, T2, [Nb | L]) ← flight(D, T1, T3, Nb),
                                  connection(D, T3, T2, L),
                                  compatible(Nb, L).
```

The first rule specifies a connection consisting of one single flight. The list of flight involved in this connection ([Nb]) thus contains only one flight number. The second rule "links" a flight to a connection and extends its list of flight numbers. The predicate "compatible" is assumed to express whether times and airports are compatible in a connection. It might be specified intensionally by means of deduction rules, or extensionally by a relation. Recursive specification are important in practice for specifying several natural properties that apply on an indefinite number of object. Another example is the definition of a "bill of material": the price of a complex object is obtained by summing up the prices of its parts, whose prices are in turn similarly defined. Like for flight connections, it is desirable to have a specification at our disposal which is not limited to a given number of components (e.g. flights or parts). It has often been observed that recursive specifications are hardly avoidable in real life applications.

Deduction rules thus are very similar to relational views. Since the first relational database systems were not capable of handling recursive views, deduction rules are

[1] This representation is taken from the time table booklet published by Lufthansa.

often seen as the extension of relational views to recursion. In our opinion, deduction rules are more than extended views. Views are not handled like regular data, i.e. tuples and relations, in a relational database management systems, while deduction rules should be seen as first class citizen in a deductive database system. This means that all the facilities that are provided by the system for storing, retrieving, updating, and querying extensional specifications (i.e. facts) should also be applicable to intensionally defined data (i.e. data defined by deduction rules) and to the intensional specifications themselves. The full realization of this objective is still the subject of active research.

2.2 Remarks on the Language of Deduction Rules

The deduction rules specifying connecting flights (cf. previous section) contain complex, nested terms, namely lists. It is often believed that nested terms and term constructors should be prohibited in deductive databases. We think that nested terms are needed (as in the above example). Moreover, the known techniques are (almost) sufficient to acommodate them like flat, so-called first-normal form facts. It is probably the concept of *Datalog,* i.e. the language of rules with flat terms and no negation, which has widespread the idea that deductive databases should only specify first-normal form tuples.

In deductive databases, the same form of negation is needed as in relational databases. This negation has been formalized in various manner and under different names (negation as failure, non-monotonic negation, negation according to the closed-world assumption, etc.). Common to these formalizations is the basic notion that an expression can be considered as false if it cannot be proved. This interpretation of negation is a rather intuitive form of reasoning. This is this way of thinking that leads us to conclude, for example, that there are no direct flights from Munich to Trondheim if we do not find any in the time table. Although there is a general agreement on the semantics of this form of negation for relational databases, it is not always clear how to formalize it in deductive databases. Rules like the following ones are difficult to interpret, indeed:

$$\begin{aligned} a &\leftarrow \text{not } b. \\ b &\leftarrow \text{not } a. \end{aligned}$$

"a" should be derivable only if "b" is not derivable, and "b" should be nonderivable only "a" is also nonderivable. Various more or less complex, more or less intuitive proposals have been made for giving convincing interpretations to such examples (and to more sophisticated ones) as well as for defining query answering procedures according to (some of) these interpretations. The problem is not yet completely solved and is still investigated. There is however a general agreement on the semantics of negation in so-called *stratified* deductive databases (or logic programs). The basic idea of stratification is to partition hierarchically the definitions of predicates, such that no predicate definitions refers to the negation of a predicate defined in a higher strata. Since one might have to deal with incompletely, or even incorrectly specified databases – for example for debugging at design time –, it is desirable to have a semantics (and the corresponding answering procedures) at our disposal which does not impose any syntactical restrictions such as stratification.

There is however a syntactical restriction which is desirable, that of *range restriction.* Range restriction basically requires that any variable occurring in a negated expression in a query or in the body (i.e. the right hand side) of a rule also occurs in a unnegated, positive expression. Thus, "p(X), not q(X)" is range-restricted, but "p(X), not q(X, Y)" is not because the variable Y has no (positive) range. Since, due to the interpretation of negation, negative expressions are absent from the database, range restriction is needed for ensuring that the variables occurring in a query or in a rule body can be assigned values from subexpressions occurring in this query or rule.

2.3 Integrity Constraints

Deduction rules give rise to generating new facts from a database, i.e. deduction rules express *constructive specifications.* In contrast to deduction rules, *integrity constraints* are used for expressing non-constructive, *normative* specifications. Such specifications are needed for ensuring that some properties remain satisfied when data are updated. The following integrity constraint for example states that no flights are allowed to land after 23:00:

$$\forall\ D\ T1\ T2\ Nb\quad [\,\text{flight}(D, T1, T2, Nb) \Rightarrow T2 \geq 2300\,]$$

Any attempt to specify a flight landing after 23:00 would lead to a violation of this integrity constraint. This violation would be reported to the database user who could then either modify the update, or, if it appears to be no more valid, the integrity constraint instead. An integrity constraint can thus be viewed as a yes/no query which is evaluated when the database is updated. Integrity constraints are needed not only for specifying negative properties, as in the previous example, but also for stating disjunctive or existential conditions, like in the following examples stating that at least one of two flights must be recorded (i.e. specified) in the database, and that there exists at least one day on which there is a flight, respectively:

$$\text{flight}(\text{saturday}, 0700, 0745, \text{lh0345}) \vee \text{flight}(\text{saturday}, 0735, 0810, \text{lh0346})$$
$$\exists\ D\quad [\,\text{day}(D) \wedge \text{flight}(D, 0700, 0745, \text{lh0345})\,]$$

Although marketed database management systems can only maintain very limited types of integrity constraints (if at all!), normative specifications are important in all kinds of database applications. Integrity constraints are expressed and maintained through application programs in current databases, that is *outside* the scope of the database system. This is undesirable because this makes the specification and the maintenance of integrity constraints a (generally complex) programming task. In deductive databases, this is part of the database design, for which tools should be available [16]. Integrity constraints are not declaratively specified but are expressed by means of imperative programs. Moreover these programs usually combine the *specifications* of the normative conditions and their efficient *evaluation.* In deductive databases in contrast, one only has to specify integrity constraints. Their efficient evaluation is left to the database management system (cf. Section 4 below). This is not only more convenient for the database designer. This also ensures that integrity constraints are efficiently checked. This is hardly the case when application programs are modified for acommodating the modifications of integrity constraints that are unavoidable in any real life applications.

Range restriction is needed for integrity constraints like for deduction rules. A universal quantification $\forall$ X F[X] is range restricted if the expression F[X] is of the form R[X] $\Rightarrow$ G[X] and if X appears positively in R (cf. [10] for a precise definition). Thus $\forall$ X [p(X) $\Rightarrow$ q(X)] is range restricted, while $\forall$ X [($\neg$ p(X)) $\Rightarrow$ q(X)] is not. An existential constraint $\exists$ X F[X] is range restricted if F[X] is of the form R[X] $\wedge$ G[X] and if X appears positively in R[X] [10]. Range restriction ensures that only updates affecting expressions occurring in a constraint (directly or indirectly through deduction rules) might violate this constraint. This is an essential condition for an efficient integrity checking (cf. Section 4). It is worth noting that range restriction is a very natural requirement: in natural languages, it is almost impossible to express properties that are not range restricted. Moreover, formulas that are not range restricted have "semantically equivalent" counterparts that are range restricted.

2.4 Constraints as Rules

Deduction rules can be used for expressing integrity constraints in two different ways. The first one consists in expressing quantifiers by means of rules, the second approach, in rewriting the integrity constraint as special rules. The following deduction rule express a range-restricted universal quantification:

forall(X, R => F) $\leftarrow$ not (R, not F).

Consider for example the following universal formula: $\forall$ X p(X) $\Rightarrow$ q(X). It would be expressed as "forall(X, p(X) => q(X))" using the formalism defined by the above given rule. This expression evaluates to true if and only if it is impossible to satisfy the conjunctive query "p(X), not q(X)", i.e. to find a value X in the relation "p" which is not also in the relation "q". The deduction rule given above thus specifies a constructive evaluation of range restricted universally quantified expressions [12]. Existential quantifications are even easier to express in the formalism of deduction rule:

exists(X, F) $\leftarrow$ F.

Instead of relying on the above given rules for quantifiers, one can also directly rewrite the integrity constraints as rules. An integrity constraint C is expressed as a rule, called *denial*, corresponding to "false $\leftarrow$ not C". The examples of integrity constraints given above lead thus to the following denials:

false $\leftarrow$ flight(D, T1, T2, Nb), T2 > 2200.
false $\leftarrow$ not flight(saturday, 0700, 0745, lh0345),
          not flight(saturday, 0735, 0810, lh0346).
false $\leftarrow$ not (day(D), flight(D, 0700, 0745, lh0345))

The two approaches are in fact the two sides of a same coin. The second representation is obtained from the first by partial evaluation (or partial deduction) [42, 35, 36, 37, 41] of the rules specifying quantifiers in the integrity constraints.

3 Query Answering

Queries are usually answered against the constructive specifications contained in the database, i.e. against the facts and deduction rules. Standard query answering methods do not make use of integrity constraints. Two complementary techniques can be applied in standard query answering: bottom-up (or forward) or top-down (or backward) reasoning. Bottom-up reasoning procedures basically consist in repeating the following as long as new facts are obtained: the bodies of all rules are evaluated against the explicitly stored facts, and the corresponding facts specified by the heads (i.e. the left hand side) of the rules are added to the database (in a special area). Consider for example the following database which can be interpreted as follows. "f(X, Y)" means that "X" is the father of "Y"; the odd (even, resp.) numbers are in a father-child relationship, and this relationship has circles on letters ("a" and "b" as well as "c" and "d" are "fathers" of each other); "g(X, Y)" means that "X" and "Y" belong to the same generation.

```
g(X, Y) ← f(FX, X), g(FX, FY), f(FY, Y).   f(1, 3)   f(2, 4)   f(a, b)
g(1, 2)                                    f(3, 5)   f(4, 6)   f(b, a)
g(a, c)                                              f(6, 8)   f(c, d)
                                                               f(d, c)
```

The facts "g(1, 2)" and "g(a, c)" give rise to deriving "g(3, 4)", "g(b, d)", and "g(5, 6)" using the deduction rule. Bottom-up reasoning on this database leads to generating these facts in stages:

Stage 1:	g(3, 4)	g(b, d)
Stage 2:	g(3, 4)	g(b, d)
	g(5, 6)	g(a, c)
Stage 3:	g(3, 4)	g(b, d)
	g(5, 6)	g(a, c)
		g(b, d)

The next round derives the same facts are those proven at stage 3. For restricting the repeated derivation of already proven facts, one can require that at least one of the facts produced at the previous stage is used in a proof. This refined procedure is called in the database community, the *semi-naïve* method, while the straightforward, redundant method is called *naïve*. The naïve and semi-naïve methods terminate as soon as no new facts are derived. It is not possible to completely avoid a repeated generation of some facts, for a same fact can have several distinct proofs. Using bottom-up reasoning for answering a query basically consists in generating all derivable facts from the database, and then in evaluating the query against the resulting, extended set of facts. There are methods for restricting to some extent and in some cases this "blind" generation. However, it is an inherent feature of bottom-up reasoning not to make use of the posed query in trying to answer it: bottom-up reasoning is not "goal directed". It is worth emphasizing that the naïve and semi-naïve methods compute *sets* at each stages and that set-oriented techniques from relational system can be applied for computing these sets. An efficient processing of quantifiers, negation, and disjunctions that are frequent in integrity constraints and deduction rules requires to refine over the traditional techniques of relational algebra [10].

Top-down reasoning procedures overcome this drawback by reasoning backward from the posed query. Consider once again the father–generation database given above and the query "g(3, X)" asking for all Xs that are in the same generation as 3. The only solution is 4, since 1 and 2 are fathers of 3 and 4, respectively and belong themselves to a same generation. Reasoning backwards from the query "g(3, X)" consists in selecting a rule whose head unifies with the query. In our case, there is only one candidate rule. The unification of its head with the query binds the variables in its body resulting in the following conjunctive subquery: f(FX, 3), g(FX, FY), f(FY, Y). The first conjunct "f(FX, 3)" has one single solution which binds the variable FX to 1. The next conjunct (or subquery) to evaluate is "g(1, FY)". It can be answered either against the facts, or using once again the deduction rule in which case the same process is repeated.

Top-down reasoning can be formalized in terms of bottom-up reasoning by relying on the formalism of deduction rules as follows [13]:

fact(X) ← query(X), rule(X ← Y), answer(Y).
query(Y) ← query(X), rule(X ← Y).
query(Y1) ← query((Y1, Y2)).
query(Y2) ← query((Y1, Y2), answer(Y1).
answer(X) ← query(X), fact(X).
answer((Y1, Y2)) ← query((Y1, Y2)), answer(Y1), answer(Y1).

Assume that these rules are evaluated bottom-up and that the predicate "fact" ("rule", respectively) range over the facts (deduction rules, resp.) stored in the database. The first rule select a deduction rule the head of which unifies with a query, and, if an "answer" (a predicate defined by other rules) is found, generates a fact. In the formalism of deduction rules used here, unification does not have to be redefined: it is already provided by this formalism. The second rule generates a (generally conjunctive) query by unifying a query with the head of a rule. The third and fourth rules split conjunctive queries; the last two rules derive conjunctive answers by conjuncting already generated answers. The query "g(3, X)" is answered as follows by processing the deduction rules given above with the semi-naïve method:

Stage 1: query((f(ZX, 3), g(ZX, ZY), f(ZY, Y)))
Stage 2: query(f(ZX, 3))
...
Stage 6: answer(g(3, 1)) query((f(ZX, 1), g(ZX, ZY), f(ZY, Y)))
Stage 7: query((g(1, ZY), f(ZY, Y)))
...
Stage 11: answer(g(3, 4))

The above mentioned, rule-based specification of top-down reasoning is interesting for several reasons. Firstly, since it is expressed in terms of bottom-up reasoning, it is easily amenable to set-oriented computations. This is important for the sake of efficiency in databases. Secondly, the above specified top-down procedure is *complete,* more precisely *exhaustive:* If there are finitely many answers, it computes all of them and terminates; if there are infinitely many answers (in presence of function symbols), each single answer in computed in finite time. The top-down reasoning procedure generally used in logic programming, SLD resolution, in contrast might loop in presence of recursive deduction rules. Termination (or exhaustivity) is important in databases, for database users as opposed to programmers cannot be made responsible of termination of the queries they pose to a database. Finally, the specification given above provides with a simple formalization of the Alexander or Magic Set rewriting methods [1, 2, 57, 7, 55, 4, 21, 8]: these rewritings are obtainable from the rule-based specification given above by partial evaluation (or partial deduction) [42, 35, 36, 37, 41]. These points are discussed in more detail in [13]. [61] also shows, from a different angle, that the Alexander and Magic rewritings in fact implement top-down reasoning by means of deduction rules that are evaluated bottom-up.

We would like to conclude this section on deductive database query answering methods with some remarks. Firstly, although "goal directedness" in general is important for efficiency, there are cases where the overhead resulting from generating and managing subgoals does not pay off. In theses cases, that still remain to be fully characterized, a bottom-up reasoning with the semi-naïve method is more efficient than a top-down procedure like the above specified one or Magic Set. Secondly, it is in some cases preferable to compute all derivable facts beforehand instead of generating the needed one for each query at query time. In these cases as well, bottom-up reasoning with the semi-naïve method is preferable. Finally, there has been proposals to use integrity constraints either for speeding up or for enhancing query answering (cf. e.g. [22, 46, 50, 14]). These approaches are very promising. They often give rise to more informative answers than conventional query answering methods.

4 Integrity Checking

Integrity constraints can be seen as yes/no queries (cf. Section 2.3). They can therefore be evaluated like regular queries. This is however often inefficient. Integrity constraints indeed are to be checked only after updates, and updates usually do not affect the whole of a database but only a limited part of it. For the sake of efficiency, it is desirable to check only those integrity constraints that might be affected by an update. Various integrity checking methods have been proposed that all rely on similar principles. Let us illustrates the techniques common to these so-called "integrity checking" methods on an example. Consider an integrity constraint requiring that all employees working for the sales department speak English:

$$\forall X \; [\; (\text{empl}(X) \wedge \text{works-for}(X, \text{sales-dept}) \;) \Rightarrow \text{speaks}(X, \text{english}) \;]$$

Any update to the facts and deduction rules that have no effect on the predicates occurring in this constraint cannot violate it. It is worth noting that this only holds if integrity constraints are range restricted. The insertion of any fact, say "p(a)" might violate a non range restricted constraint such as C: $\forall$ X q(X). If "a" did not occur in the database before the insertion of "p(a)", C indeed does not hold after the change. Whether an update might affect the definition of a relation can be specified using deduction rules as follows:

potential-update(H, Sign) ← rule(H ← B), potential-update(B, Sign).
potential-update((C1, C2), Sign) ← potential-update(C1, Sign).

```
potential-update((C1, C2), Sign) ← potential-update(C2, Sign).
potential-update(not F, Opp-Sign) ← potential-update(F, Sign),
                                    opposite(Sign, Opp-Sign).
potential-update(F, +) ← insert(F).
potential-update(F, -) ← remove(F).
```

Let us comment this specification starting from the last two rules. The insertion (removal, resp.) of a fact F induces a "potential-update" on F with positive (negative, resp.) polarity. Negation changes the polarity of a potential update: For example, if "p(a)" is a potential removal, the negative information "not p(a)" is potentially inserted. The second and third rules specify that potential updates of conjuncts induce potential updates of conjunctions with same polarity. The first rule propagate potential insertions through deduction rules. Thus, if "p(a)" is inserted, the conjunction "(pa), q(a))" is a potential insertion. In presence of a rule "r(X) ← p(X), q(X)." "r(a)" is in turn a potential insertion.

All integrity checking method rely on analyses of possible (or actual) consequences of updates similar to the computation of potential updates which is specified above by means of deduction rules. This is quite intuitive when integrity constraints are expressed as denials. Integrity checking then indeed reduces to verifying whether "false" will become derivable after an update. Denials that cannot give rise to proving "false" can be filtered out by rules like the above mentioned ones, for "false" is derivable after an update only if "potential-update(false, +)" holds.

The analyses performed by the various integrity checking methods in some cases consider, in other cases ignore the values of the attributes. They sometimes perform bottom-up, sometimes top-down reasoning on the deduction rules, or on rules used for specifying the integrity constraints (cf. e.g. [43, 25, 39, 15, 17, 18] and [19] for an overview). Some methods, e.g. [53, 25, 15], simplify the integrity constraints with respect to updates. Such simplifications can be formalized as a partial evaluation (or partial deduction) [42, 35, 36, 37, 41] of deduction rules similar to those specified above.

In the rule-based specification of potential updates which is given above, we assume that the updates are specified as sets specified by the relations "insert" and "remove". It is worth noting that these relations can be defined intensionally by deduction rules as well as extensionally by means of facts. One could for example specify an update by the following rule:

```
insert( speaks(X, english) ) ← nationality(X, british).
```

This rule is rather similar to the deduction rule "speaks(X, english) ← nationality(X, british)". The difference is that it forces the explicit storage of facts in the database, while the deduction rule for "speaks" does not. Integrity constraints can as well be defined on the predicates "insert" and "remove". The following integrity constraint for example forbids to fire of employees who work for the sales department:

$$\forall X \; [\, \text{works-for}(X, \text{sales-dept}) \Rightarrow \neg\, \text{remove}(\, \text{emp}(X)\,)\,]$$

5 Deduction Rules for Specifying System Components

In the previous sections, we have outlined how query answering and integrity checking procedures can be specified by means of deduction rules. The technique,which was used is known as *meta-programming,* for the variables in these deduction rules do not range, as in ordinary rules, over application data but instead over expressions (i.e. integrity constraints or rules) that describe the application data. We have pointed out that rewriting methods used for answering queries and evaluating integrity constraints can be seen as resulting from the partial evaluation (or partial deduction) of rule-based specifications. In this section, we first argue that it is beneficial to specify and implement some components of a database management system in this way. Then, we suggest further applications of this approach.

A first advantage of specifying components of a database management system using deduction rules and partial evaluation is the uniformity of the approach. Instead of implementing several rewriting methods for, say, recursive query processing (e.g. [1, 57, 2, 7, 4]), for simplifying integrity constraints (e.g. [43, 25, 15]), for query optimization (e.g. [46, 22, 10, 11]), etc. one could generate them automatically from the rule-based specifications using techniques as proposed in [58, 62, 42, 27, 35, 36, 37].

System components declaratively specified using deduction rules would probably be easier to prove correct and to maintain than conventional programs. Moreover, the very maintenance and updating tools provided by the database management system (e.g. integrity checking) could be applied to maintaining those system components that are specified in terms of deduction rules.

Specifying system components using deduction rules would in addition contribute to enhance the extensibility of the system. It is indeed easier to extend a set of deduction rules with additional rules for novel functionalities (e.g. additional query optimization strategies) than to extend a conventional program.

To which extent this approach is applicable in designing database management system is not yet known. The approach we suggest has however already been applied, more or less consciously, in many system prototypes that have been developed during the last years, e.g. [41, 68]. From discussions we had with designers of various database system prototypes (e.g. [54, 3, 23, 26, 33, 34, 40, 66, 68, 20, 69]) we gained the impression that meta-programming techniques are rather widely applied, although often quite unconsciously, in implementing database systems. The systematic investigation of this techniques for database system design is, we think, a promising direction of research.

Deduction rules can also be used for specifying data models and query languages. This is a widespread practice in logic programming to specify an interface model or language by means of rules. This can be done in deductive databases as well either for specifying a semantic data model (e.g. a entity-relationship model), or for specifying a query language (e.g. a SQL-like language). Rules can also be used for mapping complex objects on lower level data structures. Deductive databases are often criticized for being, like relational databases, value-based, and for not providing with object identities. Identities are "logical pointers" that give rise to naming objects [5, 6]. Extending the paradigm of logic programming and deductive facilities with identities is a promising issue. We think, this is the key issue to solve for bringing closer together both paradigms of deductive and object-oriented databases.

Deduction rules can finally also be used for interpreting the data stored in a database in various manners. Rules can be specified for various forms of reasoning that can be needed for some applications (e.g. hypothetical or probabilistic queries). Non-standard query answering methods (e.g. [62, 22, 46, 14]) often have been specified using meta-programming techniques.

6 Conclusion

This article has introduced and discussed the goals and techniques of deductive databases. We outlined how deductive databases give rise to *declaratively* specifying both, *constructive* and *normative* aspects of an application, using *deduction rules* and *integrity constraints,* respectively.

We informally presented bottom-up and top-down, set-oriented query answering methods, and we introduced to the principles upon which integrity checking methods are based. We have shown that deduction rules are not only useful for specifying database applications, but can also serve to specify and implement components of a database management system.

We finally argued that this approach is of interest for several reasons: It gives rise to a more uniform system design, system components implemented this way are easier to maintain; system extensibility is made easier.

Finally, we suggested further applications of this approach towards enhanced database systems.

References

1. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic Sets and Other Stange Ways to Implement Logic Programs. Proc. 5th ACM SIGMOD-SIGART Symp. on Principles of Database Systems (1986)
2. Bancilhon, F., Ramakrishnan, R.: An Amateur's Introduction to Recursive Query Processing. Proc. ACM SIGMOD Conf. on the Management of Data (1986)
3. Beierle, C.: Knowledge Based PPS Applications in PROTOS-L. Proc. 2nd Logic Programming Summer School (1992)
4. Beeri, C.: Recursive Query Processing. Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1989) (tutorial)
5. Beeri, C.: A Formal Approach to Object-Oriented Databases. Data & Knowledge Engineering 5 (1990) (Invited paper. A preliminary version of this article appeared in the proc. of the 1st Int. Conf. on Deductive and Object-Oriented Databases)
6. Beeri, C.: Some Thoughts on the Evolution of Object-oriented Database Concepts. Proc. GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft (1993)
7. Beeri, C., Ramakrishnan, R.: On the Power of Magic. Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1987)
8. Bidoit, N.: Bases de Données Déductives. Armand Colin (1992) (in French)
9. Bocca, J.: On the Evaluation Strategy of Educe. Proc. ACM SIGMOD Conf. on the Management of Data (1986)
10. Bry, F: Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. Proc. ACM SIGMOD Conf. on the Management of Data (1989)
11. Bry, F.: Logical Rewritings for Improving the Evaluation of Quantified Queries. Proc. Int. Conf. Mathematical Fundamentals of Data Base Systems (1989)

12. Bry, F.: Logic Programming as Constructivism: A Formalization and its Application to Databases. Proc. 8th ACM-SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1989)
13. Bry, F.: Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. Data & Knowledge Engineering 5 (1990) (Invited paper. A preliminary version of this article appeared in the proc. of the 1st Int. Conf. on Deductive and Object-Oriented Databases)
14. Bry, F.: Constrained Query Answering. Proc. Workshop on Non-Standard Queries and Answers (1991)
15. Bry, F., Decker, H., Manthey, R.: A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. Proc. 1st Int. Conf. on Extending Database Technology (1988)
16. Bry, F., Manthey, R.: Checking Consistency of Database Constraints: A Logical Basis. Proc. 12th Int. Conf. on Very Large Databases (1986)
17. Bry, F., Manthey, R.: Deductive Databases – Tutorial Notes. 6th Int. Conf. on Logic Programming (1989)
18. Bry, F., Manthey, R.: Deductive Databases – Tutorial Notes. 1st Int. Logic Programming Summer School (1992)
19. Bry, F., Manthey, R., Martens, B.: Integrity Verification in Knowledge Bases. Proc. 2nd Russian Conf. on Logic Programming (1991) (invited paper)
20. Cacace, F., Ceri, S., Crespi-Reghizzi, S., Tanca, L., Zicari, R.: Integrating Object-Oriented Data Modelling With a Rule-based Programming Paradigm. Proc. ACM SIGMOD Conf. on the Management of Data (1990)
21. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Surveys in Computer Science, Springer-Verlag (1990)
22. Chakravarthy, U.S., Grant, J., Minker, J.: Foundations of Semantic Query Optimization for Deductive Databases. In [48] (1988)
23. Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S., Tsur, S., Zaniolo, C.: The LDL System Prototype. IEEE Trans. on Knowledge and Data Engineering 2(1) (1990) 76–90
24. Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. Comm. ACM 13 (1970) 377–387
25. Decker, H.: Integrity Enforcement on Deductive Databases. Proc. 1st Int. Conf. Expert Database Systems (1986)
26. Freitag, B., Schütz, H., Specht, G.: LOLA – A Logic Language for Deductive Databases and its Implementation. Proc. 2nd Int. Symp. on Database System for Advanced Applications (1991)
27. Gallagher, J.: Transforming Logic Program by Specializing Interpreters. Proc. European Conf. on Artif. Intelligence (1986) 109-122
28. Gallaire, H., Minker, J. (eds): Logic and Databases. Plenum Press (1978)
29. Gallaire, H., Minker, J., Nicolas, J.-M. (eds): Advances in Database Theory. Vol. 1. Plenum Press (1981)
30. Gallaire, H., Minker, J., Nicolas, J.-M. (eds): Advances in Database Theory. Vol. 2. Plenum Press (1984)
31. Gallaire, H., Minker, J., Nicolas, J.-M. (eds): Logic and Databases: A Deductive Approach. ACM Computing Surveys 16:2 (1984)
32. Haas, L. M., Chang, W., Lohman, G. M., McPherson, J., Wilms, P. F., Lpis, G., Lindsay, B., Pirahesh, H., Carey, M., Shekita, E.: Starburst Mid-Flight: As the Dust Clears. IEEE Trans. on Knowledge and Data Engineering (1990) 143–160

33. Jarke, M., Jeusfeld, M., Rose, T.: Software Process Modelling as a Strategy for KBMS Implementation. Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases (1989)
34. Kiernan, G., de Maindreville, C., Simon, E.: Making Deductive Databases a Practical Technology: A Step Forward. Proc. ACM SIGMOD Conf. on the Management of Data (1990)
35. Komorowski, J.: Partial Evaluation – Tutorial Notes. North Amer. Conf. on Logic Programming (1989)
36. Komorowski, J.: Synthesis of Program in the Framework of Partial Deduction. Technical Report TR-81, Computer Science Depart. Åbo Akademi, Finland (1989)
37. Komorowski, J.: Towards Synthesis of Programs in the Framework of Partial Deduction. Proc. Workshop on Automating Software Design. XIth Int. Joint Conf. on Artif. Intelligence (1989)
38. Komorowski, J.: Towards a Programming Methodology Founded on Partial Deduction. Proc. 9th European Conf. on Artif. Intelligence (1990) 404–409
39. Kowalski, R. Sadri, F., Soper, P.: Integrity Checking in Deductive Databases. Proc. 13th Int. Conf. on Very Large Databases (1987)
40. Lefebvre, A., Vieille, L.: On Query Evaluation in the DedGin* System. Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases (1989)
41. Lei, L., Moll, G.-H., Kouloumdjian, J.: A Deductive Database Architecture Based on Partial Evaluation. SIGMOD Records 19(3) (1990) 24–29
42. Lloyd, J., Shepherdson, J. C.: Partial Evaluation in Logic Programming. Jour. of Logic Programming 11 (1991) 217–242
43. Lloyd, J. W., Sonenberg, E. A., Topor, R. W.: Integrity Constraint Checking in Stratified Databases. Jour. of Logic Programming 1(3) (1984)
44. Lloyd, J. W., Topor, R. W.: A Basis for Deductive Database Systems. Jour. of Logic Programming 2(2) (1985)
45. Lloyd, J. W., Topor, R. W.: A Basis for Deductive Database Systems II. Jour. of Logic Programming 3(1) (1986)
46. Lobo, J., Minker, J.: A Metaprogramming Approach to Semantically Optimize Queries in Deductive Databases. Proc. 2nd Int. Conf. Expert Database Systems (1988)
47. Martens, B., Bruynooghe, M.: Integrity Constraint Checking in Deductive Databases Using a Rule/Goal Graph. Proc. 2nd Int. Conf. Expert Database Systems (1988)
48. Minker, J. (ed.): Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann (1988)
49. Moerkotte, Karl, S.: Efficient Consistency Control in Deductive Databases. Proc. 2nd Int. Conf. on Database Theory (1988)
50. Motro, A.: Using Integrity Constraints to Provide Intensional Responses to Database Queries. Proc. 15th Int. Conf. on Very Large Databases (1989)
51. Morris, K., Ullman, J. D., Van Gelder, A.: Design Overview of the NAIL! System. Proc. 3rd Int. Conf. on Logic Programming (1986)
52. Naqvi, S., Tsur, S.: A Logical Language for Data and Knowledge Bases. Computer Science Press (1989)
53. Nicolas, J.-M.: Logic for Improving Integrity Checking in Relational Databases. Acta Informatica 18(3) (1982)
54. Nicolas, J.-M., Yazdanian, K.: Implantation d'un Système Déductif sur une Base de Données Relationnelle. Research Report, ONERA-CERT, Toulouse, France (1982) (in French)
55. Ramakrishnan, R.: Magic Templates: A Spellbinding Approach to Logic Programming. Proc. 5th Int. Conf. and Symp. on Logic Programming (1988)

56. Ramakrishnan, R., Srivastava, D., Sudarshan, S.: CORAL: Control, Relation and Logic. Proc. Int. Conf. on Very Large Databases (1992)
57. Rohmer, J., Lescœur, R., Kerisit, J.-M.: The Alexander Method. A Technique for the Processing of Recursive Axioms in Deductive Databases. New Generation Computing 4(3) (1986)
58. Safra, S., Shapiro, E.: Meta-interpreters for Real. Information Processing 86. North-Holland (1986) 271–278
59. Sakama, C., Itoh, H.: Partial Evaluation of Queries in Deductive Databases. New Generation Computing 6 (1988) 249–258
60. Schmidt, H., Kiessling, W., Günther, H., Bayer, R.: Compiling Exploratory and Goal-Directed Deduction Into Sloopy Delta-Iteration. Proc. Symp. on Logic Programming (1987)
61. Seki, H.: On the Power of Alexander Templates. Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1989)
62. Sterling, L. S., Beer, R. D.: Meta-interpreters for Expert System Construction. Technical Report TR 86-122, Center for Automation and Intelligent System Research, Case Western Reserve Univ. (1986)
63. Takuchi, A., Furukawa, K.: Partial Evaluation of Prolog Programs and its Application to Meta Programming. Information Processing 86. North-Holland (1986) 415–420
64. Tsur, S.: A (Gentle) Introduction to Deductive Databases. Proc. 2nd Int. Logic Programming Summer School (1992)
65. Ullman, J. D.: Principles of Database and Knowledge-Base Systems. Vol. 1 and 2. Computer Science Press. (1988, 1989)
66. Vaghani, J., Ramamohanarao, K., Kemp, D., Somogyi, Z., Stuckey, P.: The Aditi Deductive Database System. Proc. NACLP Workshop on Deductive Database Systems (1990)
67. Vieille, L.: Recursive Query Processing: The Power of Logic. Theoretical Computer Science 69(1) (1989)
68. Vieille, L., Bayer, P., Küchenhoff, Lefebvre, A.: EKS-V1: A Short Overview. Proc. AAAI-90 Workshop on Knowledge Base Management Systems (1990)
69. Vieille, L.: A Deductive and Object-Oriented Database System: Why and How? Proc. ACM SIGMOD Conf. on the Management of Data (1993)