

## DESIGNING DATABASE APPLICATIONS IN LOGIC PROGRAMMING

Marco A. CASANOVA

Centro Científico de Brasília, IBM Brasil, P.O. Box 853, 70.830, Brasília, DF, Brazil

Claudio M.O. MOURA

Departamento de Informática, PUC/RJ

Rua Marquês de S. Vicente, 209, 22.453, Rio de Janeiro, RJ, Brazil

A method for designing database applications in Logic Programming is presented. The method combines conceptual modelling ideas with a technique for defining correct transactions in a declarative style. In particular, it explains how to achieve consistency preservation by defining clauses that act as assertions to block incorrect updates or that act as triggers to propagate updates. An example application is also included to illustrate the method.

### 1. INTRODUCTION

Most database applications implemented in the past addressed business data processing and were developed by superposing programming techniques with database logical design methods. The final implementation ran on top of a database management system whose interfaces follow one of the traditional data models. However, the last decade saw the development of an increasing number of advanced database applications, consisting of a database component and a set of users' transactions, for a large spectrum of environments, such as engineering design and expert systems, as well as sophisticated interactive business data processing. These newer applications demand more sophisticated and flexible data models, more query/data processing power and better design disciplines.

One of the promising approaches to develop advanced database applications, especially those that require a deductive component, advocates the use of Logic Programming technology [9,12,13,15,16,17,18,19,21,22,23]. The core component of such applications would be a deductive database, where clauses represent data in a database state as well as deductive information, integrity constraints, and even triggers [6]. Deductive databases are different from logic programs in that databases generally have a very large number of facts but only a few rules, whereas logic programs generally have more rules than facts. Although the Logic Programming approach certainly simplifies the development of sophisticated database applications, it still raises software design questions not yet fully explained.

We then explore in this paper how to design a database application in Logic Programming and how to define clauses that will play the role of views, assertions, triggers and users' updates. We do so by adopting familiar database design techniques and by expanding the design problem to also include the definition, in a declarative style, of correct transactions. Without such extension, it is hardly possible to analyse assertions and triggers properly. We exemplify the ideas with the development of a simplified software tool to help design relational conceptual schemas.

### 2. DEDUCTIVE DATABASES

A deductive database, also called a logical, or inferential or knowledge database, consists of:

- a) a database language, which is a first-order language whose non-logical symbols are a finite set of constants  $c_1, \dots, c_n$  and a finite set of predicate symbols;
- b) the domain closure axiom  
$$x=c_1 \vee \dots \vee x=c_n$$
saying that the only known individuals are  $c_1, \dots, c_n$
- c) the unique name axioms  
$$c_1 \neq c_2, \dots, c_{n-1} \neq c_n$$
saying that the names of the known individuals are all different;
- d) the usual equality axioms;
- e) a finite set of first-order clauses over the database language (and, hence, without function symbols) representing information about the real world, either explicitly or in deductive form.
- f) a suitable formalization of the closed world assumption (CWA) [23,24] which says, intuitively, that all information there is to know is stored in the database. Alternatively, one may replace the CWA by the open world assumption which says, intuitively, that there is more information to know than that stored in the database.

The CWA may be formalized, in several special cases, by the negation as failure rule, which states that, for any positive literal P, infer  $\neg P$  iff P cannot be proven from the deductive database [3,10]. It may also be formalized by the so-called completion axioms [24], or by the circumscription rule [20].

To each clause in a deductive database, we may assign a role, which we understand as a meta-level concept that helps design deductive databases as well as process queries and updates. Although in Section 4 we will discuss the role clauses may play from the perspective of database design, we provide below several illustrative examples (the reader familiar with the topic may skip directly to Section 3).

First of all, a clause may represent knowledge about a particular state of the real world (the data in the usual sense). We call such clauses database clauses. For example, suppose that the database language has two unary predicate symbols, REGISTERED STUDENT and JUNIOR STUDENT, three binary predicate symbols, TAKES, TEACHES and TEACHER, and a set of constants that we leave implicitly defined. We may use the clause

(1)  $\text{TAKES}(\text{John}, \text{CS110}) \vee \text{TAKES}(\text{John}, \text{CS115})$

to indicate that we know that John takes either CS110 or CS115, and the clause

(2)  $\neg \text{TAKES}(\text{John}, \text{CS202})$

to express that we know that John does not take CS202.

A form of undefined values may also be captured by using Skolem functions and by modifying the domain closure and the unique name axioms. For example, we may use the clause

(3)  $\text{TAKES}(\text{John}, w)$

where  $w$  is a Skolem constant (and, hence, not in the original database language) to indicate that we know that John takes some course, that we presently do not know which one it is, but we provisionally name it  $w$ .

Clauses may capture general knowledge in the form of deductive laws that permit inferring new information from that stored explicitly. For example, the clause

(4)  $\text{TAKES}(s, c) \ \& \ \text{TEACHES}(t, c) \ \rightarrow \ \text{TEACHER}(t, s)$

which reads "if  $s$  takes  $c$  and  $t$  teaches  $c$  then  $t$  is a teacher of  $s$ ", defines the TEACHER relationship between students and professors. We may also define the clause

(5)  $\text{JUNIOR\_STUDENT}(s) \ \rightarrow \ \text{TAKES}(s, \text{CS110})$

to indicate that every junior grader has to take CS110. Thus, we have clauses that explicitly define facts we know about TAKES side by side with clauses that capture deductive knowledge we have about TAKES.

But clauses may also capture general knowledge in the form of integrity or generation laws [10,12,21,24] that influence the behaviour of updates to a deductive database. For example, consider the clause

(6)  $\text{TAKES}(s, c) \ \rightarrow \ \text{REGISTERED\_STUDENT}(s)$

saying that everyone that takes some course must be a registered student. If we interpret this clause as an integrity law, the deductive database system will block all updates that violate the clause. Alternatively, if we interpret it as a generation law, the system will automatically assert "REGISTERED STUDENT( $s$ )" when an update asserts "TAKES( $s, c$ )" and "REGISTERED STUDENT( $s$ )" does not hold. Note that integrity laws express integrity constraints and that generation laws correspond to triggers [6]. Generation laws are a special case of propagation laws, which we

fully discuss in Section 4. From this perspective, we say that a deductive database is consistent if all clauses that play the role of integrity laws are provable from the rest of the clauses of the deductive database.

Finally, clauses may also express queries. Moreover, just like testing consistency of the database, we may formalize query processing in terms of provability in the theory that describe the deductive database. Hence, we cannot forget that the theory contains the domain closure axiom, the unique name axioms, the equality axioms and a formalization of the CWA. We note that, because we can formulate recursive queries in this formalism, deductive databases offer more querying power than relational calculus [1,2].

An update to a deductive database denotes, in the most general sense, a mapping from sets of clauses into sets of clauses. Thus, updating a deductive database involves more than simply updating data in a conventional database. In particular, new information may contradict that already represented in the deductive database, in which case consistency may be restored either by abandoning or by suitably restricting some clause (or clauses) which participated in the derivation of the contradiction [12]. Plan generation procedures [9,27] that, given the desired update, suggest preliminary actions may be helpful in this context, specially to avoid creating inconsistencies. Contrasting approaches to solving the deductive database update problem may be found in [8,14].

To summarize, deductive databases offer the following advantages. Since a single formalism (that of Logic Programming) suffices to describe the information contained in the database, to define queries and updates and to write users' transactions, deductive databases avoid the gap between declarative query languages and procedural host languages commonly found in traditional database systems. Deductive databases also permit using the same mechanical theorem proving techniques for both data retrieval and program execution. They can also deal with incomplete information and negative data by choosing suitable types of clauses to represent the information about the real world.

### 3. A TOOL FOR DESIGNING RELATIONAL CONCEPTUAL SCHEMAS

We specify in this section a database design tool to help define and maintain relational conceptual schemas that will be used to exemplify the ideas of Section 4. We note that our definition of relational conceptual schema will also include predefined updates that a user might invoke to alter the database (to avoid confusion, we use the term operations when referring to the conceptual schema updates stored by the tool). A much more sophisticated tool, supporting a modular conceptual design discipline, was implemented in micro-PROLOG [4] augmented with APES [11]. In fact, the implementation of this tool motivated most of the ideas reported in Section 4.

We begin by defining a relational conceptual schema as a triple  $S = (R, C, O)$ , where  $R$  is a set of

relation schemes,  $C$  is a set of integrity constraints, and  $O$  is a set of operations. A relation scheme is a statement of the form  $R[A_1, \dots, A_n]$ , where  $R$  is the name and  $A_1, \dots, A_n$  is the attribute list of the scheme. An integrity constraint is a statement of the form  $C:W$  where  $C$  is the name of the constraint and  $W$  is the definition of the constraint, a well-formed formula (wff) of the first-order language induced by the relation schemes in  $R$ . An operation is a procedure (written in an appropriate programming language that hosts a relational data manipulation language) that queries and possibly modifies the values of the schemes in  $R$ .

A relational conceptual schema  $S=(R,C,O)$  is correct iff it satisfies the following design requirements:

REQUIREMENT 1: no two schemes have the same name

REQUIREMENT 2: no two constraints have the same name

REQUIREMENT 3: no two operations have the same name

REQUIREMENT 4: for every constraint  $C$  in  $C$ , if  $R$  is a predicate symbol of arity  $n$  that occurs in  $C$ , then  $R$  is the name of a relation scheme of  $R$  with  $n$  attributes

REQUIREMENT 5: for every operation  $O$  in  $O$ , if  $R$  is a relation name queried or updated in  $O$ , then  $R$  is the name of a relation scheme of  $R$ .

REQUIREMENT 6: for every operation  $O$  in  $O$ , for every constraint  $C$  in  $C$ ,  $O$  preserves consistency with respect to  $C$ .

The tool will store the definition of a relational conceptual schema (for simplicity we assume that there is just one) and help the DBA insert and delete relation schemes, constraints and operations in such a way that the conceptual schema is always correct.

To store the description of a relational conceptual schema, we design a data dictionary using the entity-relationship approach. We first identify three entity sets - SCHEMES, CONSTRAINTS and OPERATIONS. The attributes of SCHEME are RNAME, for the name of the relation scheme, and ATTR, for the attribute list of the scheme; those of CONSTRAINT are CNAME, for the constraint name, and DEF, for a description of the wff defining the constraint; finally, those of operation are ONAME, for the operation name, and BODY, for a description of the program defining the operation. We assume that the description of a wff or of a program takes the form of a list of tokens that we can traverse and analyse.

The problem of implementing the users' updates of the tool reduces to transforming single insertions and deletions of schemes, constraints and operations into transactions that map the set of correct relational conceptual schemas into itself. Here, we must face two major problems:

- the direct enforcement of Requirement 6 requires a program verifier
- if an insertion or deletion violates a requirement, we must decide if it must be rejected or if further corrective updates must be made

We do not solve the first problem, but leave it to the DBA the responsibility for enforcing Requirement 6. We may say that, in general, the tool uses the DBA as an oracle whenever it must do some semantic check involving transactions and constraints. However, we introduce a relationship set, ENFORCES, between OPERATION and CONSTRAINT such that  $(O,C)$  is in ENFORCES iff operation  $O$  contains tests explicitly to guarantee that it never violates the constraint  $C$ . ENFORCES then acts as a simple documentation of the semantic relationship between constraints and operations that must be described by the DBA.

To solve the second problem, we introduce a precedence order among the objects of a relational conceptual schema where the relation schemes have the highest precedence, since they define the universe of discourse, then integrity constraints have the second highest precedence, since we consider that any change to a constraint should propagate to the operations, but not vice-versa, and, finally, operations have the lowest precedence. Thus, for example, the deletion of a schema propagates to constraints and operations since schemes have the highest precedence. If we decided otherwise, the deletion of a relation scheme  $R$  might have been blocked by the existence of a constraint or an operation over  $R$ .

#### 4. DESIGNING DATABASE APPLICATIONS IN LOGIC PROGRAMMING

We outline in this section how to design a database application whose major component is a deductive database, using the tool described in Section 3 to exemplify the process.

The overall idea is quite simply to design the database application almost exactly as in the traditional way [26]. That is, one would first use conceptual modelling methods to define the conceptual and external schemas and to specify the user transactions, and then one would progressively transform their description into a deductive database, with the transactions expressed in Logic Programming. Despite the simplicity, this approach is interesting for two major reasons, in addition to the advantages listed at the end of Section 2. First, because Logic Programming acts as the target formalism, we may introduce assertions and triggers to achieve correct transaction design, thus considerably simplifying the problem of guaranteeing consistency of the database. Second, and equally important, since a conceptual modelling method provided the starting point, each clause has a clear and unambiguous role, which greatly contributes to the understanding of the final logic program. In fact, this observation implies that we may reinterpret traditional database design as a method for Logic Programming.

Consider first how to translate a conceptual schema into Logic Programming. The translation

will result in the core of the database language of the deductive database and in a set of clauses, that we called integrity laws in Section 2, describing the integrity constraints of the conceptual schema. We may also introduce a set of dictionary clauses that describe the conceptual schema itself.

Furthermore, we may also introduce deductive laws to express views (or derived relations) identified during the design of the application, since views are nothing more than defined predicate symbols. As already mentioned in Section 2, we may also introduce deductive laws to complete the definition of a database relation implicitly.

A database state will then be described by a set of clauses of the database language, called database clauses.

Using the entity-relationship description given in Section 3, the core of the database language of our running example consists of four binary predicate symbols - scheme, constraint, operation and enforces - standing for the three entity sets and the relationship set identified. As a consequence, a relation scheme  $R[A_1, \dots, A_n]$ , for example, will be described by a ground clause of the form "scheme( $R, (A_1, \dots, A_n)$ )". Furthermore note that  $R, A_1, \dots, A_n$  have the status of constants of the database language.

To help describe the integrity laws of our example, we introduce deductive laws that define the binary predicate symbols constrains and touches. The intended interpretation of "constrains( $C, S$ )" is that constraint  $C$  is over relation scheme  $S$ , and the intended interpretation of "touches( $O, S$ )" is that operation  $O$  queries or modifies scheme  $S$ . Note that, since wffs and programs are described by lists, we can indeed write a set of clauses that define these two views.

The constraints of our running example are the design rules defining what is a correct conceptual schema, plus the constraints associated with the ENFORCES relationship set. As already discussed in Section 3, except for Requirement 6, all requirements can be readily translated into clauses. For example, Requirement 1 becomes

$$(1) \text{ scheme(x,y) \& scheme(x,z) \rightarrow \text{EQ}(y,z)}$$

where "EQ" denotes the usual equality (we maintained the same notation for clauses used in Section 2, although the reverse arrow notation of PROLOG would be more appropriate in this context). Requirement 4 is, in part, expressed by the clause:

$$(2) \text{ constrains(x,y) \rightarrow scheme(y,f(y))}$$

where "f" is a Skolem function introduced to denote the attribute list of the scheme. Likewise, Requirement 5 is expressed in part by the clause

$$(3) \text{ touches(x,y) \rightarrow scheme(y,g(y))}$$

where "g" is another Skolem function.

We are now ready to discuss how to design the transactions of an application. We first address the problem in terms of our running example, and then state some brief general remarks about the problem. We describe only the insertion and deletion of schemes, and the deletion of constraints.

The transaction that performs the insertion of a new relation scheme has two steps. The first step adds a new ground clause  $G$ , describing the relation scheme, to the set of database clauses. The second step invokes the integrity laws about relation schemes (there is just one corresponding to Requirement 1) to check the consistency of the new set of database clauses. If the test fails, then the second step retracts  $G$ .

The transaction that implements the deletion of a relation scheme named  $R$  also consists of two steps. The first step retracts the ground clause  $G$  describing  $R$  from the set of database clauses. The second step then invokes propagation laws to retract all ground clauses that describe constraints over  $R$  or operations that query or update  $R$ .

Finally, we briefly discuss the transaction that deletes constraints. Given the name  $C$  of a constraint, the transaction first retracts the database clause describing  $C$ . Then, it locates the operations of the relational conceptual schema that contain tests that exist solely to guarantee consistency preservation with respect to  $C$  (such tests obviously became unnecessary). This is accomplished with the help of propagation laws that use the information contained in the "enforces" clauses. Finally, it engages in a dialogue with the DBA to modify the operations.

In general, the framework we suggest for designing transactions consists of defining three classes of clauses: transaction processing clauses, integrity laws and propagation laws. The integrity and propagation laws can be defined before writing the transaction processing clauses, but operate as "declarative subroutines" potentially shared among them. This stratification of clauses facilitates defining transactions in a declarative, incremental style quite suitable for Logic Programming.

In this framework, a transaction will then consist of a set of transaction processing clauses that capture its procedural aspects, including the communication with the user and the actual update of the set of database clauses. The transaction processing clauses may invoke integrity and propagation laws to decide if an update to an object in the database must be accepted, rejected or accepted with propagation to other objects.

The integrity laws act as assertions [5] when invoked from transaction processing clauses, using traditional database terminology. They test if an update can be accepted or if it must be rejected by rolling-back the transaction. They capture integrity constraints of the application, as already mentioned, and they are defined independently of the procedural aspects of database updates.

Propagation laws express when an update to an object must be followed by updates to other

objects to restore consistency. They are obtained by a direct analysis of the integrity constraints of the application, helped by additional information about the structure of the data (in our running example, the precedence order). The propagation laws can also be defined independently of the transaction processing clauses. The generation laws of [21], alluded to in Section 2, are then a special case of propagation laws. When compared to traditional database concepts, propagation laws may be used to formalize triggers [6] and to capture the propagation of updates through entity-relationship diagrams [25].

In the simple case where the database clauses are ground positive literals and the updates simply assert and retract such literals, we can reason as follows to define propagation laws. Let  $C(R_1, \dots, R_n)$  be an integrity law over the predicate symbols  $R_1, \dots, R_n$ , that must all be part of the database language. An update affects this law if it asserts or retracts a literal over  $R_i$ , for some  $i=1, \dots, n$ . A propagation law  $p(R_1, \dots, R_n)$  would then be defined to help determine which other updates must be made in one or more  $R_j$ , for  $j=1, \dots, n$ , to restore consistency. In our example, we used a predetermined precedence order on the objects of the database to help generate propagation laws from the integrity laws. However, the situation becomes much more complex when the database clauses are not simple positive literals, since we now have to face the general update problem for deductive databases [13,14,8]. It also complicates matters if  $R_i$  defines a view, since we would in this case be confronted with the general view update problem [7].

## 5. CONCLUSIONS

According to the Logic Programming approach, a database application will consist of a single logic program, that is, a uniform collection of clauses, structured into the following sets: dictionary clauses, database clauses, deductive laws, integrity laws, propagation laws and transaction processing clauses.

Thus, as it has long been argued, Logic Programming offers a uniform representation for the most important database concepts. However, one can also argue, as we did, that the process of obtaining the logic program describing an application does not deviate from the tradition of database design, which can therefore be reinterpreted as a method for Logic Programming.

## REFERENCES

- [1] A.V. Aho and J.D. Ullman, "Universality of Data Retrieval Languages", Proc. of the 6<sup>th</sup> ACM Annual Symp. on Principles of Programming Languages, San Antonio TX (Jan. 1979), 110-120.
- [2] A.K. Chandra and D. Harel, "Horn Clauses and Fixpoint Query Hierarchy", Proc. of the 1<sup>st</sup> ACM Symp. on Principles of Database Systems, Los Angeles CA (Mar. 1982), 158-163.
- [3] K.L. Clark, "Negation as Failure", in Logic and Databases, H. Gallaire and J. Minker (eds.) Plenum Press (1978).
- [4] K.L. Clark and F.G. McCabe. Micro-PROLOG: programming in Logic. Prentice-Hall (1984).
- [5] K.P. Eswaran and D.D. Chamberlin, "Functional Specification of a Subsystem for Data Base Integrity". Proc. 1<sup>st</sup> Int. Conf. on Very Large Data Bases (Sept. 1975).
- [6] K.P. Eswaran, "Specification, Implementation and Interaction of a Trigger Subsystem in an Integrated Data Base System", IBM Research Report RJ1820 (Aug. 1976).
- [7] A.L. Furtado and M.A. Casanova, "Updating Relational Views" in Query Processing in Database Systems, W. Kim, D.S. Reiner and D.S. Batory (eds.), Springer Verlag (1985), 127-142.
- [8] R. Fagin, J.D. Ullman, M.Y. Vardi, "On the Semantics of Updates in Databases", Proc. of the 2<sup>nd</sup> ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, Atlanta GA (Mar. 1983), 352-365.
- [9] A.L. Furtado and C.M.O. Moura, "Expert Helpers to data-based Information Systems", Proc. of the 1<sup>st</sup> Int. Workshop on Expert Database Systems (1984), 298-313.
- [10] H. Gallaire, J. Minker and J-M. Nicola, "Logic and Databases: A Deductive Approach", ACM Computing Surveys 16:2 (June 1984), 153-185.
- [11] P. Hammond and M. Sergot, APES: Augmented PROLOG for expert systems - reference manual. Logic Based Systems Ltd. (1984).
- [12] R. Kowalski, "Logic for Data Description", in Logic and Databases, H. Gallaire and J. Minker (eds.), Plenum Press (1978).
- [13] R. Kowalski, "Logic as a Database Language", Proc. Advanced Seminar on Theoretical Issues on Databases, Cetrano, Italy (1981).
- [14] R. Kowalski and M. Sergot, "A Logic-based Calculus of Events", Tech. Report Dept. of Computing, Imperial College (Nov. 1984).
- [15] S. Kunifuji and H. Yokota, "PROLOG and Relational Databases for the Fifth Generation Computer Systems", Proc. Workshop for Logical Bases for Data Bases, Toulouse, France (Dec. 1982).
- [16] J.W. Lloyd, "An Introduction to Deductive Database Systems", The Australian Computer Journal, 15:2 (May 83).
- [17] J.W. Lloyd and R.W. Topor, "A Basis for Deductive Database Systems I", Tech. Rep. 85/1, Dept. of Computer Science, Univ. of Melbourne, Victoria, Australia.
- [18] J.W. Lloyd and R.W. Topor, "A Basis for Deductive Database Systems II", Tech. Rep. 85/6, Dept. of Computer Science, Univ. of Melbourne, Victoria, Australia.
- [19] D. Maier, "Databases and the Fifth Generation Project: Is PROLOG a Database Language?", 1984 NYU Symp. in New Directions in Database Systems.
- [20] J. McCarthy, "Circumscription - a Form of Non-monotonic Reasoning", Artificial Intelligence 13 (1980), 27-39.
- [21] J-M. Nicolas and K. Yasdanian, "An Outline of DBGEN: a Deductive DBMS", in Information Processing 83, R.E.A. Mason (ed.), North Holland (1983), 711-717.
- [22] D.S. Parker et al., "Logic Programming and Databases", Panel in First Int. Workshop on Expert Database Systems, Kiawah Island SC (Oct. 1984).

- [23] R. Reiter, "On Closed World Databases", in Logic and Databases, H. Gallaire and J. Minker (eds.), Plenum Press (1978).
- [24] R. Reiter, "Towards a Logical Reconstruction of Relational Database Theory" in On Conceptual Modelling, M.L. Brodie, J. Mylopoulos, J.W. Schmidt (eds.), Springer-Verlag (1984).
- [25] P. Scheuermann, G. Schiffer and H. Weber, "Abstraction Capabilities and Invariant Properties Modelling within the Entity-Relationship Approach", Proc. 1<sup>st</sup> Int. Conf. on Entity-Relationship Approach, Los Angeles CA (Dec. 1979).
- [26] T.J. Teorey, J.P. Fry, Design of Database Structures, Prentice Hall, Inc. (1982).
- [27] D.H.D. Warren, "Warplan: A System for Generating Plans", memo 76, Univ. of Edinburgh (1974).