

# TRANSFORMING CONSTRAINTS INTO LOGIC PROGRAMS: A CASE STUDY

Antonio L. Furtado<sup>\*</sup>, Marco A. Casanova, Luiz Tucheran

IBM Scientific Center  
P.O. Box 1830  
20.071, Rio de Janeiro, RJ

This paper describes an experiment and suggests a general approach to the design of logic programs that update deductive databases. The experiment concerns the construction of a software tool that supports a database design discipline based on the concept of module. The construction of the tool is based on a complete formalization of the design discipline, expressed by a set of requirements, which are then mapped into a fact acquisition strategy.

## 1. INTRODUCTION

One of the promising approaches to develop advanced database applications, especially those that require a deductive component, advocates the use of Logic Programming technology [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. 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. Deductive databases are different from logic programs only 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 raises software design questions not yet fully explained.

This paper describes an experiment in the design of logic programs that update deductive databases. The experiment concerns the construction of a software tool that supports a database design discipline based on the concept of module [12, 13, 14]. The tool is built around a dictionary that stores conceptual schemas and it offers advice to the DBA so that the schemas he defines always meet the design requirements of the discipline. The dictionary, together with the interface, behaves as a sophisticated, albeit small, deductive database.

Briefly, the design of the tool comprised four major steps:

Step 1: design of the conceptual schema of the dictionary;

Step 2: complete formalization of the modularization discipline, resulting in a set of design requirements that characterize what is a correct schema in the context of the discipline;

Step 3: mapping of the design requirements into a fact acquisition strategy;

Step 4: final design of the logic program.

As for the fourth step we just remark here that a first prototype of the tool, written in micro-Prolog [15] extended with apes [16], is fully operational and is described in [17, 18]. In particular, it incorporates, in clause form, a description of the design requirements behind the modularization discipline, which facilitates the incremental addition of new expertise about database design.

The first and the second steps follows from the discussion in section 2 about the basic concepts of the modularization discipline. The core of the paper, sections 3 and 4, concentrate on the third step. Section 5 concludes with some remarks suggesting why our investigation contributes to the problem of designing logic programs that update deductive databases.

<sup>\*</sup> On leave from the Pontificia Universidade Catolica do Rio de Janeiro.

## 2. MODULAR DATABASE DESIGN

This section outlines a database modularization discipline, that is, a database design discipline based on the concept of a database module and on the concept of module constructors. The description of the concepts forms a series of definitions, organized in a top-down fashion, that leave off most of the motivation behind the discipline (see [17]). The basic conclusion of this section is that the modularization discipline can be completely characterized by a set of design requirements that define when a modular database schema is correct. Since the requirements form a long and somewhat repetitive list, they are collected in the Appendix.

The motivation behind the modularization discipline stems from the problem of organizing the design and maintenance of complex database schemas, understood here in the broader sense of a description of both the data structures (static part) and the transactions (dynamic part [19]) of an information system (IS) developed around a database.

The discipline permits collecting structures, constraints and operations that are closely related into separate modules [12, 13, 14], that are in turn introduced in a structured fashion that enhances the understandability of the database. Note that modules also include generic constraints, which reflects our position that constraints act as a declarative documentation of additional semantics of the data, whereas operations incorporate this same semantics procedurally. However, one does not replace the other, and both must coexist in the conceptual model of the enterprise, even with a certain redundancy. The discipline also dictates that the relations of a module  $M$  must be updated only by the operations defined in  $M$ , which corresponds to the usual notion of encapsulation [13]. Hence, if the operations of each module  $M$  preserve consistency with respect to the integrity constraints of  $M$ , the discipline introduces an effective way to guarantee logical consistency of the database. Yet, queries remain unrestrained in our discipline, just like in the traditional database design strategies [20, 21].

The central definitions of the modularization discipline now follow.

A *modular database schema* is a set of modules of three types: primitive, subsumption and external. Every module will be described by a triple, as defined below.

A *conceptual module* is either a primitive module or a subsumption module.

A *primitive module*  $M$  is a triple of the form

$$(\text{'primitive'}, (RS, CN, OP, EN), \emptyset)$$

where  $RS$  is a set of relation structures,  $CN$  is a set of integrity constraints over  $RS$ ,  $OP$  is a set of operations over  $RS$ , and  $EN$  is a set of enforcement clauses over  $CN$  and  $OP$ .

Let  $CN$  and  $OP$  be a set of constraints and operations, respectively. An *enforcement clause* over  $CN$  and  $OP$ , all defined in the primitive module, is a statement of the form ' $O$  enforces  $I_1, \dots, I_k$ ' where  $O$  is the name of an operation in  $OP$  and  $I_j$  is the name of a constraint in  $CN$ , for each  $j = 1, \dots, k$ .

The database designer should use a primitive module to introduce new relation structures, integrity constraints and operations that have no connections with the objects of the current database schema. In particular, he must include an enforcement clause ' $O$  enforces  $I$ ' in the definition of a primitive module whenever some change in the definition of  $I$  implies that the definition of  $O$  also has to be changed. This typically occurs when  $O$  contains queries whose only purpose is to check some condition that guarantees that  $O$  will not violate  $I$ . Thus, enforcement clauses play a fundamental role when redesigning modules, but they cannot be easily derived from the definition of constraints and operations.

A *subsumption module* defined over modules  $M_1, \dots, M_n$  is described by a triple of the form

$$(\text{'subsumption'}, (RS, CN, OP, EN, HI), (M_1, \dots, M_n))$$

Where  $RS$  is a set of relation structures,  $CN$  is a set of integrity constraints,  $OP$  is a set of operations,  $EN$  is a set of enforcement clauses over  $CN$  and  $OP$ , and  $HI$  is a set of hiding clauses over the operations of  $M_1, \dots, M_n$  and constraints in  $CN$ .

Let CN and OP be a set of constraints and operations, respectively. A *hiding clause* over OP and CN is a statement of the form 'O may violate I<sub>1</sub>, . . . , I<sub>k</sub>' where O is the name of an operation in OP and I<sub>j</sub> is the name of a constraint in CN, for each j = 1, . . . , k.

A module M defined by subsumption over modules M<sub>1</sub>, . . . , M<sub>n</sub> automatically inherits all the objects of M<sub>1</sub>, . . . , M<sub>n</sub>, except those operations referred to in a hiding clause. Module M subsumes M<sub>1</sub>, . . . , M<sub>n</sub> in the sense that an object of M<sub>1</sub>, . . . , M<sub>n</sub> can now be accessed only as an inherited object of M (if it was indeed inherited by M). As a consequence, the DBA can no longer use M<sub>1</sub>, . . . , M<sub>n</sub> to define new modules after he defines M, and users can no longer invoke operations hidden in M.

Thus, the database designer must use the subsumption constructor when he wants to add, among other objects, a new constraint I that may be violated by an operation O' defined in an existing module M'. In this case, he must define a new module M by subsumption over M', include I in the definition of M and hide O' in M by indicating that O' may violate I. He must also define a new operation O in M that has almost the same functionality of O', except that O enforces I, which is ensured by including queries in O to check whether it may call O' without violating I.

An *external* module over modules M<sub>1</sub>, . . . , M<sub>n</sub> is defined by a triple of the form

$$(\text{'external'}, (RS, CN, OP, VW, SP), (M_1, \dots, M_n))$$

Where RS is a set of relation structures, CN is a set of integrity constraints over RS, OP is a set of operations over RS, VW is a set that contains, for each structure in RS, a view definition mapping over the union of the relation structures of M<sub>1</sub>, . . . , M<sub>n</sub>, and SR is a set that contains a surrogate for each operation in OP.

A *surrogate* for an operation f is an operation that simulates the action of f by updating the structures of the base modules M<sub>1</sub>, . . . , M<sub>n</sub>.

The database designer should define an external module M over M<sub>1</sub>, . . . , M<sub>n</sub> when he wants to introduce views and operations on views (surrogates avoid by fiat the so-called view update problem [22, 23]). Modules extended by M remain available for further use in module definitions.

If M = (t, l, m) is a subsumption or external module, we say that the sets of objects of the modules in the list m are *imported* by M.

Finally, a *correct modular database schema* is a modular database schema that satisfies the set of *design requirements* listed in the Appendix. The design requirements basically guarantee that if a modular database schema is correct then:

1. for each relation structure, there is at least one operation that adds tuples to it;
2. every operation is accessible to the users either directly or through calls from other operations;
3. every operation directly accessible to the users preserves consistency with respect to all constraints.

The last correctness criterion is especially important to understand why modularization is so interesting for the design of large schemas. Without the restrictions imposed on modules, to guarantee (3), the database designer would have to check that EACH operation preserves consistency with respect to ALL constraints. The judicious choice of the modularization discipline alters the problem of checking consistency preservation in the following way:

- the database designer must explicitly check if each operation of a newly defined module M preserves consistency with respect to all the constraints of M;
- the database designer need not check that the operations of M preserve consistency with respect to the constraints of modules previously defined, since this is automatically guaranteed by the modularization discipline;
- the database designer need not check that the operations of previously defined modules preserve consistency with respect to the constraints of M, since this is automatically guaranteed by the modularization discipline, except if M is defined by subsumption over M', when operations of M' may have to be hidden in M.

### 3. AN ANALYSIS OF THE DESIGN REQUIREMENTS

In this section we begin to explain how we mapped the design requirements into statements of a logic program that helps the DBA create and maintain correct modular database schemas. More specifically, we address two problems:

1. how to test for the requirements;
2. when to apply such tests.

The first problem depends on a classification for the requirements while the second depends on the introduction of what we called the definition order for the sets of objects of a schema.

As an example consider the problem of guaranteeing that a modular database schema  $S^i$  is correct with respect to requirement P7, which says that for each operation  $p$  of a primitive module  $m$  of  $S^i$ , for each constraint  $i$  of  $m$ ,  $i$  must be an invariant of  $p$ . Now recall that a primitive module in  $S^i$  is a triple of the form ('primitive', (RS, CN, OP, EN),  $\emptyset$ ) where RS, CN, OP, EN are the sets of relation structures, constraints, operations and enforcements of the module. Then, we can formalize P7 by the following *defining formula*:

$$(1) \forall r \forall c \forall o \forall e ((\text{'primitive'}, (r, c, o, e), \emptyset) \in S^i \Rightarrow \forall i \forall p (i \in c \ \& \ p \in o \Rightarrow p7(i, p)))$$

where the intended interpretation of " $p7(i, p)$ " is that operation  $p$  preserves constraint  $i$ . Hence, P7 *constrains* the set of constraints and operations of each primitive module of  $S^i$  and is a *semantic* requirement because the truth of  $p7(i, p)$  depends on the meaning of  $i$  and  $p$ .

The formula in (1) should not be directly translated into sentences of the logic program because we are not interested in testing if the complete definition of a schema  $S^i$  is correct, but rather in helping the database designer gradually specify a correct modular database schema. Hence, it is reasonable to replace the defining formula of P7 by a test relating the constraints and operations of a primitive module.

The exact format of the test depends on when it should be applied and on the relative importance of the sets of objects P7 constrains. However, P7 is doubly neutral regarding these points since it neither indicates if constraints should be defined before operations or vice-versa, nor indicates if an operation should be changed if it violates a constraint, or vice-versa. To circumvent this problem we postulate that constraints have precedence over operations and define the *test associated with P7* as follows:

$$(2) \forall i (i \in CN \Rightarrow p7'(i, p))$$

where CN is the set of constraints of the module M currently being defined. Such test will then be implemented in the logic program and applied when each operation  $p$  of M is defined. Hence, the operations of a primitive module *govern* the application of the test associated with P7. In general, we shall say that a set of objects governs the application of a test to mean that the occasion to apply the test is exactly when such objects are being defined. Moreover, since constraints take precedence over operations, if  $p$  fails the test, the logic program will force the database designer to change  $p$ , and not the constraints.

Let us now generalize these observations, without going into the details of each requirement. Let us first consider how to transform a formalization of the requirements into tests that the logic program will apply as the database designer specifies a schema.

For each requirement R, we identify:

- a *defining formula* that formalizes R;
- a *test associated with R* that the logic program will incorporate;
- a class of sets of objects of a module, or imported by the module, that R *constrains*;
- a set of objects of a module that *governs* the application of the test associated with R.

The format of the tests depends on a classification of the defining formulas into *syntactical* and *semantic* and, orthogonally, into *types 0, 1 or 2*. Figure 3.1 at the end of this section summarizes some of these concepts.

Syntactical requirements impose restrictions just on the syntax of modules or objects, whereas semantic requirements have to do with the semantics of operations or formulas.

Type 0 requirements impose simple conditions on the definition of modules and their tests will be applied as soon as the definition of a new module begins, depending on the type of the module. They are all formalized as formulas of the form:

$$\forall x \forall y ((t, x, y) \in S) \Rightarrow Q(y)$$

where  $Q(y)$  is the test associated with the requirement.

Type 1 requirements impose restrictions on sets of objects and their tests will be applied when each object of their governing class is defined. Their defining formula can be profitably massaged (for the purposes of the tool) into the form

$$\forall x_1 \dots \forall x_n ((t, (x_1, \dots, x_n), u) \in S) \Rightarrow \forall y (y \in x_{ij} \Rightarrow Q(x_{i1}, \dots, x_{ij-1}, y, x_{ij+1}, \dots, x_{im}))$$

where " $Q(x_{i1}, \dots, x_{ij-1}, y, x_{ij+1}, \dots, x_{im})$ " is the test associated with the requirement. For the module  $m = (t, (x_1, \dots, x_n), u)$ ,  $x_{i1}, \dots, x_{im}$  are the sets of objects constrained and  $x_{ij}$  is the set of objects governing the application of the test.

Type 2 requirements also impose restrictions on sets of objects but their tests can only be applied right after their governing class is completely defined. Their defining formula has the following general form:

$$\forall x_1 \dots \forall x_n ((t, (x_1, \dots, x_n), u) \in S) \Rightarrow Q(x_{i1}, \dots, x_{im}, u))$$

where " $Q(x_1, \dots, x_n, u)$ " is again the test associated with the requirement.

We now address the second problem, viz., how to decide when the logic program will apply the tests associated with the requirements. To settle this question, we introduce the following concept.

### Definition 3.1

Let  $S$  be a modular database schema. The *definition order* for  $S$ , denoted by " $<$ ", is a binary relation over sets of objects of  $S$  defined as follows:

#### *Object definition order*

Let  $M'$  be a module of  $S$  and, when applicable depending on the type of  $M$ , let  $RS, CN, OP, HI, EN, VW$  and  $SR$  be the sets of structures, constraints, operations, hiding statements, enforcement statements, view definitions and surrogates, respectively, defined in  $M$ . Then,

$$\begin{aligned} RS &< VW \\ RS &< CN, VW < CN \\ CN &< HI \\ RS &< OP, CN < OP, HI < OP \\ CN &< EN, HI < EN, OP < EN \\ VW &< SR, OP < SR \end{aligned}$$

#### *module definition order*

Let  $M'$  and  $M$  be modules of  $S$  such that  $M'$  is defined over  $M$  and let  $X, X'$  be sets of objects of  $M$  and  $M'$ , respectively. Then,

$$X < X'$$

The intuitive interpretation of  $X < X'$  is that, to specify the objects in  $X'$ , the database designer has to know the specification of all objects in  $X$ .

By the careful choice of the definition order, we have that:

**Proposition 3.1**

Let  $x_1, \dots, X_n$  be the sets of objects of a module, or imported by the module, that are constrained by a requirement  $R$  of type 1 or 2, and let  $X_i$  be the governing the application of the test associated with  $R$ . The  $X_i$  is greater, in definition order, than  $X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_n$ .

As a consequence of this proposition, all sets of objects needed to apply the test of a type 1 or 2 requirement will have been defined when the test is actually applied, if the sets of objects are specified in definition order.

**Figure 3.1: A Classification for the Requirements**

Let  $M$  be a module and, when applicable according to the type of  $M$ , let  $RS, CN, OP, HI, EN, VW$  and  $SR$  be the sets of structures, constraints, operations, hiding statements, enforcement statements, view definitions and surrogates, respectively, defined in  $M$ . If  $M$  is defined over modules  $M_1, \dots, M_n$ , let  $RS, CN$  and  $OP$  be the vectors  $(RS_1, \dots, RS_n)$ ,  $(CN_1, \dots, CN_n)$  and  $(OP_1, \dots, OP_n)$ , respectively, where  $RS_i, CN_i$  and  $OP_i$  are the sets of structures, constraints and operations of  $M_i, i = 1, \dots, n$ .

REQ.	CLASSIFICATION		GOVERNING SET	CONSTRAINED SETS
M1	SYNTACTICAL	0	—	—
P1	SYNTACTICAL	1	RS	RS
P2	SYNTACTICAL	1	RS	RS
P3	SYNTACTICAL	1	CN	RS, CN
P4	SYNTACTICAL	1	CN	CN
P5	SYNTACTICAL	1	OP	RS, OP
P6	SYNTACTICAL	1	OP	OP
P7	SEMANTIC	1	OP	CN, OP
P8	SYNTACTICAL	2	OP	RS, OP
P9	SYNTACTICAL	1	EN	CN, OP, EN
P10	SEMANTIC	2	EN	CN, OP, EN
S1	SYNTACTICAL	0	—	—
S2	SYNTACTICAL	0	—	—
S3	SYNTACTICAL	1	RS	RS
S4	SYNTACTICAL	1	RS	RS, RS
S5	SYNTACTICAL	1	CN	RS, RS, CN
S6	SYNTACTICAL	1	CN	CN, CN
S7	SYNTACTICAL	1	OP	RS, OP, RS, OP
S8	SYNTACTICAL	1	OP	OP, OP
S9	SEMANTIC	1	OP	CN, OP
S10	SYNTACTICAL	2	OP	RS, OP
S11	SYNTACTICAL	1	EN	CN, OP, EN
S12	SEMANTIC	2	EN	CN, OP, EN
S13	SYNTACTICAL	1	HI	OP, CN, HI
S14	SEMANTIC	2	HI	OP, CN, HI
S15	SYNTACTICAL	2	OP	HI, OP
S16	SYNTACTICAL	2	EN	HI, OP, EN
E1	SYNTACTICAL	0	—	—
E2	SYNTACTICAL	1	RS	RS
E3	SYNTACTICAL	1	RS	RS
E4	SYNTACTICAL	1	CN	RS, CN
E5	SYNTACTICAL	1	CN	CN
E6	SEMANTIC	1	CN	CN, VW, CN
E7	SYNTACTICAL	1	OP	RS, OP
E8	SYNTACTICAL	1	OP	OP
E9	SYNTACTICAL	2	VW	RS, VW
E10	SYNTACTICAL	1	VW	RS, VW
E11	SYNTACTICAL	2	SR	OP, SR

E12	SYNTACTICAL	1	SR	RS, OP, SR
E13	SEMANTIC	1	SR	OP, VW, OP, SR

#### 4. TESTING THE REQUIREMENTS DURING THE DESIGN OF A SCHEMA

In this section we explain how we embedded in the tool a strategy for defining and modifying modular schemas that guarantees correctness with respect to the design requirements. To facilitate this task, we divided the tool into a *design tool*, that helps the database designer create new modules, and a *redesign tool*, that helps the database designer modify the objects of existing modules. For more details, see [24].

##### 4.1 Testing the Requirements during the Design of a Schema

The design tool prompts the database designer to specify the modules and the sets of objects in definition order. To guarantee correctness with respect to a requirement R, the design tool proceeds as follows.

If R is of type 0, the tool simply applies the test associated with R as soon as the designer starts the specification of a new module, depending on the module type.

Suppose now that R is of type 1 or 2. By Proposition 3.1, when the tool prompts the designer to specify the objects of the set X that governs R, all other sets of the module currently being defined (or imported by the module) constrained by R have already been defined. Then, if R is of type 1, the tool applies the test associated with R when the designer specifies each object in X. But if R is of type 2, the tool applies the test associated with R when the designer signals that he specified all objects in X. Because of the way tests were defined, this strategy guarantees that the final modular schema will satisfy R.

However, the semantic type 2 requirements P10, S12 and S14 actually determine when enforcement and hiding statements must be present. Hence, we may interpret them not as requirements, but as definitions for these two types of objects. Moreover, requirement S16 can also be understood as a way of automatically generating enforcement clauses.

Thus, the tool uses the tests associated with the requirements pertaining to structures, constraints or operations as assertions [25] to accept or reject the definition of objects, and the tests of those referring to hiding or enforcement statements as triggers [26] to synthesize clauses. All tests concerning syntactical requirements do not involve the database designer. However, in the present implementation, most of the tests concerning the semantic requirements involve queries to the database designer. As the development of the tool continues, we plan to replace some of these queries by calls to experts in specific problem areas. For example, a limited program verifier could be designed to handle the problem of consistency preservation for certain types of constraints.

Similar remarks also apply to the redesign tool discussed in the next section.

##### 4.2 Testing the Requirements during the Redesign of a Schema

We discuss in this section how the requirements and the definition order influenced the construction of the redesign tool.

The redesign tool will again force the database designer to consider sets of objects in definition order. He will be able to directly change structures, constraints and operations, but changes to enforcement or hiding statements will only be made by the tool itself, as a consequence of other changes. In the present implementation, the designer will also not be able to change the modular structure of a schema. Hence, type 0 requirements will never be violated.

To illustrate the problems raised by requirements of types 1 or 2, first observe that, when processing a change C to an object O, the tool has to decide whether to

- reject it because it violates some requirement;

- accept it because it does not violate any requirement, sometimes provided that further changes be made to restore correctness.

The choice between these two possibilities is intimately connected with the definition order introduced in Section 3. For example, recall that in a primitive module the definition order was: structures, constraints, operations and enforcement statements. Hence, when the database designer inserts a new constraint, the tool must: (1) reject the insertion if the definition of the constraint refers to a structure not defined in the module (and not ask the database designer to insert a new structure); (2) inform the database designer that he has to recheck all operations to see if they preserve the constraint (and not reject the insertion); (3) if an operation is redefined, inform the database designer that he has to recheck enforcement clauses over the operation.

In general, let  $X$  be the set currently under consideration (which must be a set of structures, or constraints or operations, since the designer is not allowed to directly change enforcement and hiding clauses). Correctness with respect to a type 1 requirement  $R$  governed by  $X$  can be established as follows. If the database designer defined some change on an object  $x$  of  $X$  that creates a new definition  $d$  for  $x$ , the correctness of  $d$  against  $R$  can be tested exactly as in Section 4.1. If the database designer decided to delete  $x$ , the change can be immediately applied without violating  $R$ , by definition of type 1 requirements. If the database designer defined no change on  $x$ , the definition of  $x$  must be rechecked against  $R$ , in principle as in Section 4.1 again.

Correctness with respect to requirements of type 2 governed by  $X$  can be established exactly as in Section 4.1, but only after all changes to  $X$  have been applied.

The semantic type 2 requirements P10, S12 and S14, as well as Requirement S16, will again be used to redefine the enforcement and hiding statements.

This concludes our brief remarks about the redesign tool.

## 5. FINAL REMARKS

The previous sections outlined the strategy we followed for writing a logic program that enforces the design requirements of a specific database design discipline. We started with a complete formalization of the discipline and gradually mapped the requirements into tests applied as the design progressed. The mapping depended on choosing a reasonable definition order for the objects of a schema.

However, this strategy can be generalized into a discipline for writing simple transactions that update deductive databases. By simple transactions, we mean here logic programs that insert, delete or replace facts (ground positive literals) in a deductive database and that preserve the integrity constraints of the application.

Briefly, in this logic programming discipline, a transaction will 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 the database must be accepted, rejected or accepted with propagation.

The *integrity laws* act as assertions 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. *Propagation laws* express when an update must be followed by other updates to restore consistency. When compared to traditional database concepts, they may be used to formalize triggers. Both integrity laws and propagation laws follow from the integrity constraints identified during the design of the database and from additional information about the structure of the data (in our case study, the definition order). Both classes can be defined independently of the transaction processing clauses.

## REFERENCES

- [1] H. Gallaire, J. Minker and J-M. Nicolas, "Logic and Databases: A Deductive Approach", ACM Computing Surveys 16:2 (June 1984), 153-185.

- [2] R. Kowalski, "Logic for Data Description", in *Logic and Databases*, H. Gallaire and J. Minker (eds.), Plenum Press (1978).
- [3] R. Kowalski, "Logic as a DB Language", Proc. Advanced Seminar on Theoretical Issues on Databases, Cetrano, Italy (1981).
- [4] S. Kunifuji and H. Yokota, "PROLOG and Relational Databases for Fifth Generation Computer Systems", Proc. Workshop for Logical Bases for Data Bases, Toulouse, France (Dec. 1982).
- [5] J. W. Lloyd, "An Introduction to Deductive Database Systems", *The Australian Computer Journal*, 15:2 (May 83).
- [6] 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.
- [7] 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.
- [8] D. Maier, "Databases and the Fifth Generation Project: Is PROLOG a Database Language?", 1984 NYU Symp. in new directions in Database Systems.
- [9] J. M. Nicolas, K. Yazdanian, "An outline of DBGEN: a A deductive DBMS", in *Information Processing 83*, R. E. A. Mason (ed.), North Holland (1983), 711-717.
- [10] D. S. Parker et al, "Logic Programming and Databases", Panel in First Int'l Workshop on Expert Database Systems, Kiawah Island, SC (Oct. 1984).
- [11] 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).
- [12] D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *Comm. of the ACM* 15:12 (1972).
- [13] B. Liskov, S. Zilles, "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering SE-1* (1975).
- [14] S. N. Zilles, P. Lucas, J. W. Thatcher, "A Look at Algebraic Specifications", Research Rep. RJ3568 IBM Thomas J. Watson Research Center (1982).
- [15] K. L. Clark and F. G. McCabe, *micro-PROLOG: programming in logic*. Prentice-Hall (1984).
- [16] P. Hammond and M. Sergot, *apes: augmented PROLOG for expert systems - reference manual*. Logic Based Systems Ltd. (1984).
- [17] L. Tucheran, A. L. Furtado and M. A. Casanova, "A Tool for Modular Database Design", Proc. of the 11th Int'l. Conf. on Very Large Data Bases, Stockholm, Sweden (1985), 436-447.
- [18] M. A. Casanova, A. L. Furtado and L. Tucheran, "A Software Tool for Modular Database Design", IBM Brasilia Scientific Center, Technical Report CCB033, (1985).
- [19] M. Brodie, "On modelling behavioral semantics of databases", Proc. of the 7th Int'l. Conf. on Very Large Data Bases, Cannes (1981), 32-43.
- [20] T. J. Teorey, J. P. Fry, *Design of Database Structures*, Prentice-Hall, Inc. (1982).
- [21] S. N. Zilles. "Types, Algebras and Modelling", Proc. of the Workshop on Data Abstractions, Databases and Conceptual Modelling, Pingree Park, Colorado (1980).
- [22] U. Dayal, P. A. Bernstein, "On the Correct Translation of Update Operations on Relational Views", *ACM Transactions on Database Systems* 7:3 (1982), 381-416.
- [23] 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.
- [24] A. L. Furtado, M. A. Casanova, L. Tucheran, "A Framework for Design/Redesign Experts", First Int'l. Conf. on Expert Database Systems, Charleston, SC (Nov. 1985), 313-328.
- [25] K. P. Eswaran and D. D. Chamberlin, "Functional Specification of a Subsystem for Data Base Integrity", Proc. 1st Int'l. Conf. on Very Large Data Bases (September 1975).
- [26] K. P. Eswaran, "Specification, Implementation and Interaction of a Trigger Sybsystem in an Integrated Data Base System", IBM Research Report RJ1820 (Aug. 1976).

s to bottom

5  
6  
5  
4  
3  
2  
1

## APPENDIX: List of Requirements

### MODULAR SCHEMAS

**M1.** no two modules of the modular schema have the same name.

### PRIMITIVE MODULES

Let  $M_k$  be a primitive module and  $RS_k$ ,  $CN_k$ ,  $OP_k$ ,  $EN_k$  be the structures, constraints, operations and enforcement statements defined in  $M_k$ .

**P1.** for each  $R$  in  $RS_k$ ,  $R$  is relation structure.

**P2.** no two structures in  $RS_k$  have the same name.

**P3.** for each  $I$  in  $CN_k$ ,  
 1.  $I$  is an integrity constraint  
 2. if  $I$  references a structure  $S$  then  $S$  must be in  $RS_k$ .

**P4.** no two constraints in  $CN_k$  have the same name.

**P5.** for each  $O$  in  $OP_k$ ,  
 1.  $O$  is an operation;  
 2. if  $O$  queries or updates a structure  $S$ , then  $S$  must be in  $RS_k$ ;  
 3.  $O$  cannot call any operation of  $M_k$  or any other module.

**P6.** no two operations in  $OP_k$  have the same name.

**P7.** for each  $O$  in  $OP_k$ , for each  $I$  in  $CN_k$ ,  $I$  must be an invariant of  $O$ .

**P8.** for each  $S$  in  $RS_k$ , there must be  $O$  in  $OP_k$  that performs insertions into  $S$ .

**P9.** for each statement " $O$  enforces  $I_1, \dots, I_m$ " in  $EN_k$ ,  
 1.  $O$  must be in  $OP_k$ ;  
 2.  $I_j$  must be in  $CN_k$ , for each  $j$  in  $[1, m]$ .

**P10.** for each  $O$  in  $OP_k$ , there is a statement of the form " $O$  enforces  $I_1, \dots, I_m$ " in  $EN_k$  iff  $\{I_1, \dots, I_m\}$  is the set of all constraints in  $CN_k$  whose definition  $O$  has to take into account.

Requirements  $P1$ ,  $P2$ ,  $P3$ ,  $P4$ ,  $P5$ ,  $P6$ , and  $P9$  are just syntactical, but they imply that a primitive module is completely isolated in itself. For example, operations can only query and update the structures defined in the module and call no external operations.

Requirement  $P7$  reflects the fundamental preoccupation that the database should always be left in a consistent state. Requirement  $P8$  guarantees that all structures of a primitive module will become active. Requirement  $P10$  defines when the database designer has to introduce an enforcement statement.

### SUBSUMPTION MODULES

Let  $M_k$  be a subsumption module defined over modules  $M_{ki}$ ,  $i = 1, \dots, n$ . Let  $RS_{ki}$ ,  $CN_{ki}$ ,  $OP_{ki}$  be the structures, constraints and operations of  $M_{ki}$ ,  $i = 1, \dots, n$ . Let  $RS_k$ ,  $CN_k$ ,  $OP_k$ ,  $EN_k$ ,  $HI_k$  be the relation structures, integrity constraints, operations, enforcement statements and hiding statements, respectively, defined in  $M_k$ . Let  $RS$  be the union of  $RS_k$ ,  $RS_{k1}, \dots, RS_{kn}$ ,  $CN$  be the union of  $CN_k$ ,  $CN_{k1}, \dots, CN_{kn}$  and  $OP$  be the union of  $OP_k$ ,  $OP_{k1}, \dots, OP_{kn}$ .

**S1.**  $M_{k1}, \dots, M_{kn}$  must be conceptual modules that are active in the prefix  $M_1; \dots; M_{k-1}$  (that is, active at the time  $M_k$  is defined).

**S2.** the prefix  $M_1; \dots; M_{k-1}$  must not contain an external module defined over  $M_{ki}$ , for some  $i$  in  $[1, n]$ .

- S3.** for each  $R$  in  $RS_k$ ,  $R$  is relation structure.
- S4.** no two structures in  $RS_k$  have the same name.
- S5.** for each  $I$  in  $CN_k$ ,
1.  $I$  is an integrity constraint
  2. if  $I$  references a structure  $S$  then  $S$  must be in  $RS$ .
- S6.** no two constraints in  $CN_k$  have the same name.
- S7.** for each  $O$  in  $OP_k$ ,
1.  $O$  is an operation;
  2. if  $O$  queries a structure  $S$ , then  $S$  must be in  $RS$ ;
  3. if  $O$  updates a structure  $S$ , then  $S$  must be in  $RS_k$ ;
  4. if  $O$  calls an operation  $O'$ , then  $O'$  must be in  $OP_{k1}, \dots, OP_{kn}$ .
- S8.** no two operations in  $OP_k$  have the same name.
- S9.** for each  $O$  in  $OP_k$ , for each  $I$  in  $CN_k$ ,  $I$  must be an invariant of  $O$ .
- S10.** for each structure  $S$  in  $RS_k$  there must be an operation  $O$  in  $OP_k$  that performs insertions into  $S$ .
- S11.** for each statement " $O$  enforces  $I_1, \dots, I_m$ " in  $EN_k$ ,
1.  $O$  must be in  $OP_k$ ;
  2.  $I_j$  must be in  $CN_k$ , for each  $j$  in  $[1, m]$ .
- S12.** for each  $O$  in  $OP_k$ , there is a statement of the form " $O$  enforces  $I_1, \dots, I_m$ " in  $EN_k$  iff  $\{I_1, \dots, I_m\}$  is the set of all constraints in  $CN_k$  whose definition  $O$  has to take into account.
- S13.** for each statement " $O$  may violate  $I_1, \dots, I_m$ " in  $HI_k$ ,
1.  $O$  must be in  $OP_{k1}, \dots, OP_{kn}$
  2.  $I_j$  must be in  $CN_k$ , for each  $j$  in  $[1, m]$ .
- S14.** for each  $O$  in  $OP_{k1}, \dots, OP_{kn}$ , there is a statement of the form " $O$  may violate  $I_1, \dots, I_m$ " in  $HI_k$  iff  $\{I_1, \dots, I_m\}$  is the set of all constraints in  $CN_k$  that are not preserved by  $O$ .
- S15.** for each statement " $O$  may violate  $I_1, \dots, I_m$ " in  $HI_k$ , there must be  $O'$  in  $OP_k$  such that  $O'$  calls  $O$ .
- S16.** for each statement of the form " $O$  may violate  $I_1, \dots, I_m$ " in  $HI_k$  for each operation  $O'$  defined in  $M_k$  that calls  $O$ , there must be a statement of the form " $O'$  enforces  $J_1, \dots, J_n$ " in  $EN_k$  such that  $\{I_1, \dots, I_m\} \subseteq \{J_1, \dots, J_n\}$ .

Requirements S1 and S2 forbid the database designer to define a new module  $M_k$  that subsumes a module  $M_{ki}$  if there is a third module  $M''$  that subsumes or extends  $M_{ki}$ . If this requirement were violated, we would have a module  $M_k$  subsuming  $M_{ki}$  and yet we would permit operations of  $M''$  call operations of  $M_{ki}$  that might be hidden in  $M_k$ . Thus, we would not be able to assure that the operations of  $M''$  preserve consistency with respect to  $CN_k$ . Conversely, if  $M''$  subsumes  $M_{ki}$ , we would permit operations of  $M_k$  call operations of  $M_{ki}$  that might be hidden in  $M''$ , thus leading to consistency violations with respect to the constraints of  $M''$ .

Requirements S3, S4, S5, S6, S7, S8, S11 and S13 are purely syntactical.

However, Requirement S7 guarantees that each operation  $O$  in  $OP_k$  preserves consistency with respect to  $CN_k$  since  $O$  updates the structures of  $M_{ki}$  only through calls to operations of  $M_{ki}$ , for each  $i = 1, \dots, n$ . Requirement S9 states that each operation in  $OP_k$  preserves consistency with respect to  $CN_k$ . Hence, these requirements together imply that each operation in  $OP_k$  preserves consistency with respect to  $CN$ .

Requirements S12 and S14 define when the database designer has to introduce enforcement and hiding statements, respectively.

Requirement S14 also guarantees that each operation in  $OP_{ki}$  that is not hidden preserves consistency with respect to  $CN_k$ . By the syntactical requirements on operations and constraints, that isolate modules from each other, each operation in  $OP_{ki}$  preserves consistency with respect to  $CN_{kj}$ , for  $i, j$  in  $[1, n]$  with  $i \neq j$ . Since we may assume by induction that each operation in  $OP_{ki}$  preserves consistency with respect to  $CN_{ki}$ , we may conclude that each operation in  $OP_{ki}$  that is not hidden preserves consistency with respect to  $CN$ .

Requirement S10 guarantees that all structures defined in a subsumption module will become active, while requirement S15 guarantees that operations that are hidden in  $M_k$  do not become inactive.

Requirement S16 forces the database designer to include the appropriate enforcement clause in the following case. Suppose that  $O$  was hidden because it may violate a constraint  $I$ , and suppose that  $O'$  calls  $O$ . Then,  $O'$  necessarily contains tests to guarantee preservation of  $I$ , as otherwise the call to  $O$  might violate  $I$ . Therefore, the database designer must indicate that  $O'$  enforces  $I$  in some enforcement clause.

## EXTERNAL MODULES

Let  $M_k$  be an external module defined over modules  $M_{ki}$ ,  $i = 1, \dots, n$ . Let  $RS_{ki}$ ,  $CN_{ki}$ ,  $OP_{ki}$  be the structures, constraints and operations of  $M_{ki}$ ,  $i = 1, \dots, n$ . Let  $RS_k$ ,  $CN_k$ ,  $OP_k$ ,  $VW_k$  and  $SR_k$  be the relation structures, integrity constraints, operations, view definitions and surrogates, respectively, defined in  $M_k$ .

- E1.  $M_{k1}, \dots, M_{kn}$  must be either external modules or active conceptual modules in the prefix  $M_1; \dots; M_{k-1}$ .
- E2. for each  $R$  in  $RS_k$ ,  $R$  is relation structure.
- E3. no two structures in  $RS_k$  have the same name.
- E4. for each  $I$  in  $CN_k$ ,
  1.  $I$  is an integrity constraint
  2. if  $I$  references a structure  $S$  then  $S$  must be in  $RS_k$ .
- E5. no two constraints in  $CN_k$  have the same name.
- E6. for each  $I$  in  $CN_k$ ,  $I'$  must be a logical consequence of the integrity constraints in  $CN_{k1}, \dots, CN_{kn}$ , where  $I'$  is obtained from  $I$  by replacing each atomic formula of the form  $R(t_1, \dots, t_j)$  by  $Q[t_1/x_1, \dots, t_j/x_j]$ , where  $R[A_1, \dots, A_j]$ :  $O$  is the view definition of  $R$  described in  $VW_k$ , and the list of free variables of  $Q$  is  $x_1, \dots, x_j$ .
- E7. for each  $O$  in  $OP_k$ ,
  1.  $O$  is an operation;
  2. if  $O$  queries or updates a structure  $S$ , then  $S$  must be in  $RS_k$ ;
  3.  $O$  cannot call any operation of  $M_k$  or any other module.
- E8. no two operations in  $OP_k$  have the same name.
- E9. there is a view definition mapping in  $VW_k$  for a relation structure  $R$  iff  $R$  is in  $RS_k$ .
- E10. for each  $Q$  in  $VM_k$ , if  $Q$  references a structure  $S$ , then  $S$  must be in  $RS_{k1}, \dots, RS_{kn}$ .
- E11. there is a surrogate in  $SR_k$  for an operation  $O$  iff  $O$  is in  $OP_k$ .
- E12. for each surrogate  $O$  in  $SR_k$ ,
  1.  $O$  is an operation;
  2. if  $O$  queries a structure  $S$ , then  $S$  must be in  $RS_{k1}, \dots, RS_{kn}$ ;
  3.  $O$  cannot directly update any structure;
  4. if  $O$  calls an operation  $O'$ , then  $O'$  must be in  $OP_{k1}, \dots, OP_{kn}$ .
- E13. for each  $O$  in  $SR_k$ , if  $O$  is the surrogate of operation  $O'$ , then  $O$  must be an exact and acceptable translation of  $O'$  [23].

Requirement E1 forbids the database designer to define a new module  $M_k$  that extends a module  $M_{ki}$  if there is a third module  $M''$  that subsumes  $M_{ki}$ . If this requirement were violated, we would have a module  $M_k$  whose operations might generate calls to operations of  $M_{ki}$  that are hidden in  $M''$ . Thus, we would not be able to assure that calls to operations of  $M_k$  do not violate constraints of  $M''$ .

Requirements E2, E3, E4, E5, E7, E8, E9, E10, E11 and E12 are purely syntactical.

Requirement E6 guarantees that the integrity constraints of  $M_k$  follow from those of  $M_{k1}, \dots, M_{kn}$ , when each view is interpreted as a defined predicate symbol. Thus, no really new local constraints can be defined in a module created by extension.

However, Requirement E12 also guarantees that each surrogate  $O$  preserves consistency with respect to  $CN_{ki}$  since  $O$  updates the structures of  $M_{ki}$  only, through calls to operations of  $M_{ki}$ , for each  $i = 1, \dots, n$ .

Requirement E13 guarantees that a surrogate  $O$  correctly implements an operation  $O'$  in the sense that  $O'$  and  $O$  have the same effect as far as the views are concerned.