A TOOL FOR MODULAR DATABASE DESIGN

Luiz Tucherman*
Antonio L. Furtado**
Marco A. Casanova***

*Latin American Systems Research Institute/IBM Brazil
**Pontifícia Universidade Católica do Rio de Janeiro
***Brasilia Scientific Center / IBM Brazil

ABSTRACT

A database design method, based on the concept of module, is first described. The method incorporates both a strategy for enforcing integrity constraints and a tactic for organizing large sets of database structures, integrity constraints and operations. A software tool that helps the development and maintenance of database schemas designed according to the method is then specified. Finally, a prototype expert system offering a partial implementation of the tool is described.

1. INTRODUCTION

We discuss in this paper a software tool that helps the database administrator specify and maintain database schemas following a modular discipline.

The tool incorporates knowledge about a database design method, first described in [TCF], that provides structured descriptions of the more traditional notions of conceptual and external schemas. Relation schemes, integrity constraints and operations are grouped into modules [Pa,LZ] and introduced in a structured, orderly fashion that enhances the understandability of the database. The method 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 [LZ]. Hence, if the operations of each module M preserve consistency with respect to the integrity constraints of M, the method introduces an effective way to guarantee logical consistency of the database. Yet, queries remain unrestrained in our method, just like in the traditional database design strategies.

Modular database design is not a new idea, but all references known to us [DMW,EKW,LMWW,SFNC,SNF,We] tend to explore the principles, theoretical and otherwise, of the method. We are, by contrast, interested in immediate applications of the idea.

The design of a database schema in our method consists of the successive addition of new modules to a (possibly empty) kernel database schema. But we also recognize that designing a database schema is intrinsically an interactive process. The database designer frequently has to go back and alter the definition of a schema, either because the application evolves, or because his perception of the application changes. This understanding of the method led us to divide the development of the tool into two phases.

In the initial implementation phase, the tool should incorporate a dictionary to store the description of modular database schemas and should provide facilities to add new modules to an existing schema. A first prototype with these characteristics, written in the APES extension of micro-PROLOG [HS], is fully operational. It incorporates several design rules and offers a very user-friendly interface capable of guiding the database administrator through the various stages of the definition of a module.

In the second stage of development, the tool should account for database redesign. That is, it should help the DBA add, delete or modify the definition of objects of a modular database schema. The redesign process is somewhat more complex, since it must necessarily map a syntactically correct schema satisfying all design requirements into another schema with the same property. As a consequence, the process must adequately cope with the problem of the propagation of changes. At the present time, the second stage is fully specified and the prototype is being extended to cover database redesign.

The paper is divided as follows. Section 2 describes the basic concepts of the database design method. Section 3 defines a dictionary to describe modular database schemas. Section 4 specifies the database design tool, with special emphasis on the problem of changing the definition of modules. Section 5 outlines the current prototype.

Due to space limitations, detailed discussions were left to the technical report version of the paper [TFC].

2. MODULAR DATABASE DESIGN

2.1 The Concept of a Module

A *relation scheme* is a statement of the form R[A1,...,An], where R is the *relation name* and A1,...,An are the *attributes* of the scheme. An *integrity constraint* is a statement of the form n:Q, where n is the *name* of the constraint and Q is a well-formed formula over the relation schemes in question. An *operation* is a procedure definition in some appropriate programming language. We will use the notation f(x1,...,xn): s to indicate an operation named f with parameters x1,...,xn and body s.

A *module* is a triple M = (RS,CN,OP) where

1. RS is a set of relation schemes such that no two schemes in RS have the same relation name;
2. CN is a set of integrity constraints over the relation schemes in RS. CN must contain, for each relation scheme R[A1,...,An], a *relation scheme axiom* indicating that the interpretation of R must be a subset of the cartesian product of the interpretations of A1,...,An.
3. OP is a set of operations over the relation schemes in RS.

2.2 Module Constructors

A module may be either *primitive*, if it is defined without any reference to other modules, or *derived*, if it is defined from previously existing modules by one of the two module constructors, *subsumption* and *extension*.

A primitive module M=(RS,CN,OP) is defined by a statement of the form:

```
(1)  module M
         schemes      RS;
         constraints  CN';
         operations   OP;
         enforcements EN;
     endmodule
```

where CN' is CN without the relation scheme axioms (since these integrity constraints are completely fixed by RS, they may be omitted from CN') and EN is a set of *enforcement clauses* of the form 'O *enforces* I' where O is the name of an operation and I is the name of a constraint of M.

The DBA must include an enforcement clause 'O *enforces* I' whenever the definition of operation O takes into account constraint I. That is, whenever some change to the definition of I affects the definition of O. This type of additional information will be important in Section 4 when we consider the problem of redesigning the database schema.

The rest of this section defines the module constructors in detail, whereas Sections 2.3 and 2.4 indicate how they can be profitably used for database design.

Let Mi = (RSi,CNi,OPi), i=1,...,n, be modules.

Consider the subsumption constructor first. Intuitively, if the DBA defines M by subsumption over modules M1,...,Mn, then M may contain new relation schemes, new integrity constraints and new operations, and M always inherits all the relation schemes and integrity constraints of M1,...,Mn. M also inherits all operations of M1,...,Mn, except that M may hide some of these operations if they violate a new constraint. Moreover, M contains all pertinent enforcement clauses just as in the definition of primitive modules. Modules M1,...,Mn then become inaccessible to the users and can no longer participate in the definition of new modules.

The following statement defines a new module M by *subsumption* over M1,..,Mn:

```
(2)  module  M  subsumes M1,...,Mn with
            schemes       RS0;
            constraints   CN0;
            operations    OP0;
            enforcements  EN;
            hidings       HI;
      endmodule
```

where:

1. RS0 is a set of relation schemes such that no relation name in RS0 occurs in M1,...,Mn, and no two schemes in RS0 have the same relation name;
2. CN0 is a set of (named) integrity constraints over RS0,RS1,...,RSn;
3. OP0 is a set of operations over RS0,RS1,...,RSn;
4. EN is a set of *enforcement clauses* of the form 'O *enforces* I' where O is the name of an operation defined in M and I is the name of a constraint also defined in M;
5. HI is a possibly empty set of *hiding clauses* of the form 'O *may violate* I1,...,Ik' where O is the name of an operation of Mi, for some i in [1,n], and Ij is the name of a constraint defined in CN0, for each j in [1,k]. We say that O is *hidden* by M.

More precisely, the statement in (2) defines a module M=(RS,CN,OP) where

1. RS is the union of RS0,...,RSn
2. CN is the union of CN0,...,CNn
3. OP is the union of OP0,OP1',...,OPn' where OPi' is OPi without all operations hidden in M, for i=1,...,n

We now turn to the definition of the extension constructor. Informally, a module M extends modules M1,...,Mn if each relation scheme of M is a *view* over the relation schemes of M1,...,Mn (that is, a relation scheme derived from those of M1,...,Mn) and each constraint of M is a logical consequence of those of M1,...,Mn, when views are treated as defined predicate symbols. M may also introduce operations on views. But, to avoid the so-called view update problem [FC], the definition of M contains, for each view operation p, an implementation of p in terms of the operations of M1,...,Mn. Unlike subsumption, modules M1,...,Mn remain accessible after the definition of M.

A new module M is defined by *extension* over M1,...,Mn through a statement of the form:

```
(3)  module  M  extends M1,...,Mn with
            schemes      RS0;
            constraints  CN0;
            operations   OP0;
         using
            views        VW;
            surrogates   SR;
     endmodule
```

where:

1. the triple (RS0,CN0,OP0) defines a module M in the sense of Section 2.1.
2. VW contains, for each scheme R[A1,...,Ak] in RS0, a *view definition mapping* of the form R(x1,...,xk) : Q, where Q is a well-formed formula with k free variables, ordered x1,...,xk, over RS1,...,RSn.
3. SR contains, for each operation f(y1,...,ym): r in OP0, a *surrogate*, which is an operation of the form f(y1,...,ym): s over RS1,...,RSn;

The statement in (3) then defines a new module M=(RS0,CN0,OP0) and couples M to M1,...,Mn through the pair (VW,SR). A view definition mapping R[A1,...,Ak]: Q in VW indicates that Q defines R in terms of the relation schemes of M1,...,Mn. Hence, a query over R is translated into a query over the relation schemes of M1,...,Mn with the help of Q. Likewise, a surrogate f(y1,...,ym): s in SR describes an implementation of f(y1,...,ym): r in terms of the operations of M1,...,Mn. Thus, a call to procedure f generates an execution of s, not r.

2.3 Design Rules for Modular Database Schemas

A modular database schema consists of a set of modules that must satisfy a series of design rules, which guarantee that if the database is updated only by the operations visible to the users, the state of the database will always remain consistent. More precisely, the set of consistent modular database schemas and their active modules, is recursively defined as follows:

1. the empty set is a consistent modular database schema with an empty set of active modules;
2. Let D be a consistent modular database schema with active modules set A. Let M be a module such that no module in D has the same name as M. Then D' = D U (M) is a consistent modular database schema iff M satisfies one of the following conditions:
   a. if M is a primitive module then M must satisfy requirement 1 (see Figure 2.1 at the end of this section for the complete list of requirements and a brief explanation of their meaning). The active module set of D' is A' = A U (M)
   b. if M is a module obtained by extending M1,...,Mn then M must satisfy requirements 2,3,4,5. The active module set of D' is A' = A U (M)
   c. if M is a module obtained by subsuming M1,...,Mn then:
      1) the relation names of the new relation schemes defined in M must be different from those of the relation schemes in M1,...,Mn.
      2) M must satisfy requirements 6,7,8,9.

      The active module set of D' is A' = A U (M) - (M1,...,Mn).

Let D be a modular database schema with active modules set A. The set C of conceptual modules of D is the subset of A consisting of all primitive modules and all active modules defined by subsumption; the set E of external modules of D is the set of all modules defined by extension in D. An operation p of D is active, conceptual or external iff p is an operation of an active, conceptual or external module of D, respectively.

A user has in principle access to all active modules of a modular database schema. Hence, he sees all relation schemes and integrity constraints defined in all modules, but he can only update the database using the active operations. He can also freely query any relation scheme.

As for the design of modular database schemas, the process we suggest follows closely the formal definition. The DBA gradually adds new modules to an initially empty database schema. He must pay attention to two aspects: how to define a new module and how to satisfy the design requirements (see Section 2.4 for an example).

To conclude this section, we state a theorem to the effect that the choice of the design requirements suffices to guarantee consistency preservation.

THEOREM 2.1 [TCF]: Let D be a modular database schema. Suppose that D satisfies requirements 1 through 9. Then, every active operation of D preserves consistency with respect to the set of all constraints defined in modules of D.

Figure 2.1: List of Requirements

PRIMITIVE MODULES

Requirement 1: each operation defined in a module M must preserve consistency with respect to all integrity constraints defined in M.

This requirement reflects the fundamental preoccupation that the database should always be left in a consistent state [CCF].

MODULES DEFINED BY EXTENSION

Let M be a module defined by extension over modules Mi=(RSi,CNi,OPi), i=1,...,n. Let RS0,CN0,OP0,VW and SR be the new relation schemes, integrity constraints, operations, view definitions and surrogates, respectively, defined in M.

Requirement 2: if f(y1,...,ym): s is the surrogate of f(y1,...,ym): r defined in SR then s is a faithful translation of r [FC].

Requirement 2 guarantees that s correctly implements r in the sense that r and s must have the same effect as far as the views are concerned.

Requirement 3: if f(y1,...,ym): s is a surrogate defined in SR, then s can only modify the values of relation schemes in M1,...,Mn through calls to the operations defined in M1,...,Mn.

Requirement 3 guarantees that each surrogate s preserves consistency with respect to CNi since s updates the schemes of Mi through calls to operations of Mi, for each i=1,...,n.

Requirement 4: for each integrity constraint I in CN0, I' must be a logical consequence of the integrity constraints of M1,...,Mn, where I' is obtained from I by replacing each atomic formula of the form R(t1,...,tk) by Q[t1/x1,...,tk/xk], where R[A1,...,Ak]: Q is the view definition of R described in VW, and the list of free variables of Q is x1,...,xk.

Requirement 4 guarantees that the integrity constraints of M follow from those of M1,...,Mn 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.

Requirement 5: M1,...,Mn must be active modules of D.

Requirement 5 avoids defining view operations using inactive operations, which may violate consistency.

MODULES DEFINED BY SUBSUMPTION

Let M be a module defined by subsumption over modules Mi=(RSi,CNi,OPi), i=1,...,n. Let RS0, CN0, OP0, HI be the new relation schemes, integrity constraints, operations, and hidden operations, respectively, defined in M. Let CN be the union of CN0,...,CNn and OP be the union of OP0,OP1',...,OPn', where OPi' is the set OPi, except for those operations that were hidden by M, for i=1,...,n.

Requirement 6: each operation in OP preserves consistency with respect to the integrity constraints in CN0.

Requirement 7: each operation in OP0 can only modify the values of relation schemes in M1,...,Mn through calls to the operations defined in M1,...,Mn.

Requirements 6 and 7 suffice to guarantee that each operation in OP preserves consistency with respect to CN.

Requirement 8: D must not contain a module defined by extension using Mi, for some i in [1,n].

Requirement 8 forbids the DBA to define a new module M by subsuming a module Mi if there is a third module M' that extends Mi. This requirement is necessary since it avoids the undesirable situation where M subsumes Mi and yet M' offers direct paths to the objects and operations of Mi. In fact, if Requirement 8 is violated, we cannot assure that calls to operations of M' will not violate constraints of M.

Requirement 9: M1,...,Mn must be conceptual modules of D

Requirement 9 does not permit the subsumption of external modules, again to guarantee that all new operations of M, and those of modules defined by subsuming M, preserve consistency.

2.4 An Example

We will illustrate our method by designing a micro database that stores information about products, warehouses and shipments of products to warehouses.

We begin by creating a schema with just one primitive module, PRODUCT, that represents data about products and contains the operations allowed on products. PRODUCT is defined as follows:

```
module PRODUCT
 schemes
  PROD[P#,NAME]
 constraints
  ONE_N: ∀p∀n∀n'(PROD(p,n) & PROD(p,n')
                 => n=n')
 operations
  ADDPROD(p,n):
   if ¬∃n' PROD(p,n') & P#(p) & NAME(n)
      then insert (p,n) into PROD;
  DELPROD(p):
   delete PROD(x,y) where x=p;
 enforcements
  ADDPROD enforces ONE_N;
endmodule
```

The enforcement clause indicates that ADDPROD takes into account the constraint ONE_N.

The modular database schema contains at this point only one module, PRODUCT, which is obviously active. We then add another primitive module, WAREHOUSE, to represent warehouses and the operations on warehouses. We define WAREHOUSE as follows:

```
module WAREHOUSE
 schemes  WAREHSE[W#,LOC]
 constraints
  ONE_C:
   ∀w∀c∀c'(WAREHSE(w,c) & WAREHSE(w,c')
           => c=c')
 operations
  OPEN(w,c):
   if ¬∃c' WAREHSE(w,c') & W#(w) & LOC(c)
      then insert (w,c) into WAREHSE;
  CLOSE(w):
   delete WAREHSE(x,y) where x=w;
 enforcements
  OPEN enforces ONE_C;
endmodule
```

The modular database schema now has two active modules, PRODUCT and WAREHOUSE. We continue the design by defining a new module, SHIPMENT, that introduces a relationship, shipment, between products and warehouses. Note that a shipment (p,w) requires that product p and warehouse w indeed exist. Since the operations DELPROD and CLOSE may violate this constraint, we must define SHIPMENT by subsumption over PRODUCT and WAREHOUSE and redefine DELPROD and CLOSE appropriately:

```
module SHIPMENT
 subsumes PRODUCT, WAREHOUSE with
 schemes  SHIP[P#,W#,QTY]
 constraints
  ONE_Q:
   ∀p∀w∀q∀q'(SHIP(p,w,q) & SHIP(p,w,q')
            => q=q')
  INC_P: ∀p(∃w∃q SHIP(p,w,q)
            => ∃n PROD(p,n))
  INC_W: ∀w(∃p∃q SHIP(p,w,q)
            => ∃c WAREHSE(w,c))
 operations
  ADDSHIP(p,w,q):
   if ∃n PROD(p,n) & ∃c WAREHSE(w,c) &
      ¬∃q' SHIP(p,w,q') & QTY(q)
      then insert (p,w,q) into SHIP;
  CANSHIP(p,w):
   delete SHIP(x,y,z) where (x=p & y=w);
  CLOSE1(w):
   if ¬∃p∃q SHIP(p,w,q) then CLOSE(w);
  DELPROD1(p):
   if ¬∃w∃q SHIP(p,w,q) then DELPROD(p);
 enforcements
  ADDSHIP  enforces ONE_Q, INC_P, INC_W;
  CLOSE1   enforces INC_W;
  DELPROD1 enforces INC_P;
 hiding
  DELPROD may violate INC_P;
  CLOSE   may violate INC_W;
endmodule
```

The modular database schema now has three modules, SHIPMENT, WAREHOUSE and PRODUCT, but only SHIPMENT is active. Note that SHIPMENT contains all relation schemes and constraints of PRODUCT and WAREHOUSE, plus a newly defined relation scheme and three new constraints. The active operations (that is, those available to users) after the definition of SHIPMENT are: ADDSHIP, CANSHIP, CLOSE1 and DELPROD1, defined in SHIPMENT, and ADDPROD and OPEN, inherited from PRODUCT and WAREHOUSE, respectively. Since the operations DELPROD and CLOSE may violate constraints INC_P and INC_W of SHIPMENT, respectively, they are hidden in SHIPMENT. Hence, CLOSE and DELPROD are no longer visible to users.

Finally, we introduce the module DELIVERY by extending SHIPMENT:

```
module DELIVERY extends SHIPMENT with
 schemes  DELVRY[P#,W#];
 constraints  /* (none) */
 operations
  DEL(p,w):
   delete DELVRY(x,y) where (x=p & y=w)
 using
  views
   DELVRY(p,w) : ∃q SHIP(p,w,q)
  surrogates
   DEL(p,w): CANSHIP(p,w)
endmodule
```

The final database schema therefore has two active modules, SHIPMENT and DELIVERY, and two other modules, PRODUCT and WAREHOUSE. Users have access to three base relation schemes (using traditional terminology), PROD[P#,NAME], WAREHSE[W#LOC], and SHIP[P#,W#,QTY], and one view, DELVRY[P#,W#]. The active operations are ADDSHIP, CANSHIP, ADDPROD, DELPROD1, OPEN, CLOSE1 and DEL. A user has access to any of these operations, but note that a call to DEL invokes the procedure associated with DEL in the surrogates clause of DELIVERY. The procedure associated with DEL in the operations clause of DELIVERY just informs the user the meaning of DEL in terms of its effect on the relation scheme DELVRY.

3. A DICTIONARY DEFINITION

We introduce in this section a dictionary that describes the objects - modules, schemes, constraints, and operations - and relationships between these objects induced by a modular database schema. The conceptual schema of the dictionary will be described in terms of an entity-relationship model. Although it is not essential, we will consider that the dictionary contains only the entities and relationships derived from a single modular conceptual schema D. It is also important to observe that the state of the dictionary representing a database schema D is fully determined by the declarative syntax of the modules of D (that introduced in Section 2), and vice-versa.

We will use B(A1,...,AN) to indicate an entity type named B whose list of attributes is A1,...,An; we will in turn use R(E1,...,Em) to describe a relationship type, whose name is R, without attributes, over the entity types named E1,...,Em. Keys will be underlined whenever necessary. The conceptual schema of the dictionary, together with the intended interpretation of the entity and relationship types, is described below:

ENTITY TYPES

is-primitive(name), is-sub(name) and is-external(name)
each module M, either primitive, defined by subsumption or defined by extension, of the modular conceptual schema D, corresponds to an entity of type is-primitive, is-sub or is-external, respectively. The only attribute is the module name.

module(name)
generalization of the three previous sets. The only attribute is the module name.

scheme(name, list, def)
each relation scheme R defined in a module of D corresponds to an entity of this type. The attributes are the name and the attribute list of R, as well as the view definition mapping of R, if R belongs to a module defined by extension, otherwise the value of attribute def is nil.

constraint(name, def)
each integrity constraint I defined in a module of D corresponds to an entity of this type. The attributes are the name and the defining formula of I.

operation(name, def, surrogate)
each operation O defined in a module of D corresponds to an entity of this type. The attributes are the name and the procedure defining O, as well as the surrogate associated with O, if O belongs to a module defined by extension, otherwise the value of surrogate is nil.

RELATIONSHIP TYPES

subsumes(module, module) and extends(module, module)
the pair (M,N) will be in the set of relationships of type subsumes or extends iff M and N represent two modules such that M is defined by subsumption or by extension, respectively, over N.

is-scheme-defined-in(scheme, module)
the pair (S,M) will be in the set of relationships of type is-scheme-defined-in iff S is a name of a scheme defined in M.

is-constraint-defined-in(constraint, module)
(same, when I is constraint defined in M.)

is-operation-defined-in(operation, module)
(same, when O is operation defined in M.)

is-view-over(scheme, scheme)
the pair (V,S) will be in the set of relationships of type is-view-over iff V represents a view whose view definition mapping involves scheme S.

is-constraint-over(constraint, scheme)
the pair (I,S) will be in the set of relationships of type is-constraint-over iff I represents a constraint whose definition involves scheme S.

is-operation-over(operation, scheme)
the pair (O,S) will be in the set of relationships of type is-operation-over iff O represents an operation whose definition or whose surrogate (if O is an operation defined in a module introduced by extension) involves scheme S.

enforces(operation, constraint)
the pair (O,I) will be in the set of relationships of type enforces iff the definition of operation O guarantees that constraint I will be not violated.

may-violate(operation,constraint)
the pair (O,I) will be in the set of relationships of type may-violate iff O represents an operation which has an execution that may violate constraint I.

calls(operation,operation)
the pair (O,O') will be in the set of relationships of type calls iff O represents an operation whose definition or whose surrogate (if O is an operation defined in a module introduced by extension) calls operation O'.

4. REDESIGNING DATABASE SCHEMAS

This section discusses in general terms how the design tool should help the DBA redesign a database schema. Section 4.1 addresses the problem of redesigning the modular structure of a schema, including the insertion and deletion of complete modules. Section 4.2 discusses the problem of redesigning the schemes, constraints, operations and relationships of modules.

4.1 Redesigning the Modular Structure of a Schema

To add a new module M to an existing modular database schema D, the DBA must successively add the schemes, constraints and operations of M, in this order, to the dictionary. The design tool should then guide the DBA in the process, verifying that he does not violate any of the requirements listed at the end of Section 2.3. However, since we do not assume a general program verifier capable of detecting if an operation violates a constraint, or if two operations are equivalent (for a set of variables), requirements 1, 2, 6 cannot be enforced. A general theorem prover would also be needed to enforce requirement 4. Thus, the DBA has to be trusted as far as these requirements go. The tool can, at most, inform the DBA when these requirements must be obeyed. As for requirements 3, 5, 7, 8 and 9, since they depend on the current state of the dictionary and on syntactic conditions, they can in principle be verified without undue effort.

The deletion of a module M is quite simple to account for, since it suffices to delete all objects defined in M and recursively delete all modules M' whose definition depends directly or transitively on M.

Changing the relationships between modules makes sense in only one case which we discuss in the rest of this section. Recall that, by requirement 8, the DBA cannot define a new module M by subsuming a module M' if there is a third module M" that extends M'. Requirement 8 avoids the undesirable situation where M subsumes M' and yet M" offers direct paths to the objects and operations of M'. In fact, if requirement 8 is violated, we cannot assure that calls to operations of M" will not violate constraints of M. On the other hand, requirement 8 is too strong in several situations. For example, suppose that we let M subsume M' as long as M does not hide any operation used to define surrogates of M". Then, the definition of M" remains valid, provided that we consider that M" now extends M, instead of M'. Since this type of change is quite useful, we introduce a new module constructor, strong subsumption.

We say that a module M strongly subsumes M1,...,Mn iff:

1. M subsumes M1,...,Mn exactly as defined in Section 2, except that requirement 8 is replaced by

   Requirement 8': M does not hide any operation p used to define a surrogate of any module M" that extends Mi, for any i=1,...,n.

2. the dictionary is changed so that any module M" that extends Mi is now considered to extend M, for each i=1,...,n.

Thus, strong subsumption is indeed a change of the database schema in the double sense that it introduces a new module M and may change the definition of several other modules.

4.2 Redesigning Objects within Modules

In order to help the DBA insert, delete or modify the definition of objects within modules, the design tool must verify the correctness of object definitions and determine how changes on a group of objects propagate to others. We focus our discussion in this section on the second problem.

We first observe that fixing how changes must propagate is equivalent to determining a policy governing how updates propagate through the entity-relationship diagram of the dictionary. The policy we adopted is expressed as a set of detailed rules, but in general it reflects a precedence relation on objects as follows:

1. relation schemes have the highest precedence, which implies that a relation scheme *S* is:
   a. never affected by changes on other objects, if *S* is defined in a primitive module or a module defined by subsumption;
   b. affected only by changes on the relation schemes *S* is defined on, if *S* is defined in a module introduced by extension;
2. constraints have the second highest precedence, which implies that a constraint I is affected only by changes on:
   a. the relation schemes I is defined on;
   b. the constraints of the extended modules, if I is defined in a module introduced by extension (to satisfy requirement 4);
3. operations have the lowest precedence, which implies that an operation O is affected by changes on:
   a. the schemes O is defined on;
   b. the constraints that O enforces or may violates, or the constraints of the module where O is defined;
   c. the operations O calls.

The redesign process is organized in two *steps*. The design tool begins the first step by asking the DBA to supply the set of changes he wants to apply to the current schema, and then it takes over and helps the DBA detect and fully specify additional changes that must be made to produce a new consistent schema. This step is itself divided into *stages* as exemplified below. During the second step, the design tool applies all changes to the current schema.

In what follows, we adopt the notation 'E1 R E2' to indicate that there is a binary relationship of type R between entities E1 and E2 in the current state of the dictionary.

As an example, referring to the database schema defined in Section 2.4, suppose that the DBA decides to add a new attribute, WEIGHT, to the relation scheme PROD. The design tool then begins stage 1 of step 1 of the redesign process by looking up in the dictionary which schemes may be affected by the change on PROD. Since there are no views defined on PROD, the tool proceeds to stage 2 where it determines which constraints are affected by the change on PROD. Using the following relationships involving PROD (that can be found in the state of the dictionary describing the database schema in question):

ONE_N *is-constraint-over* PROD
INC_P *is-constraint-over* PROD

and using the propagation rules, the design tool informs the DBA that he has to check the definition of the constraints ONE_N and INC_P. Assume that the DBA, when inspecting ONE_N, decides to modify its defining formula to accomodate the new attribute WEIGHT of PROD and also to retain P# as a key of PROD. Also assume that the DBA decides to modify the definition of INC_P just to include a third argument into the occurrence of PROD, corresponding to the new attribute WEIGHT (these are purely syntactical changes that have to be introduced anyway).

Next, the design tool starts stage 3 of step 1. It first determines how the changes defined on schemes and constraints propagate to the operations. Using the following dictionary relationships involving PROD, ONE_N and INC_P:

ADDPROD *is-operation-over* PROD
DELPROD *is-operation-over* PROD
ADDSHIP *is-operation-over* PROD
ADDPROD *enforces* ONE_N
ADDSHIP *enforces* INC_P
DELPROD1 *enforces* INC_P
DELPROD *may-violate* INC_P

and using the propagation rules, the design tool detects that the DBA must check the definition of ADDPROD, DELPROD, ADDSHIP and DELPROD1. However, the information contained in the dictionary is not sufficient to disclose all consequences of the changes specified on constraints. Indeed, since a constraint, ONE_N, of module PRODUCT was modified, the design tool must ask the DBA if its enforcement now depends also on the operation DELPROD. A similar remark applies to the operations

CANSHIP and CLOSE1, when constraint INC_P is considered. Assume that the DBA decides that CANSHIP and CLOSE1 need not be changed.

The tool proceeds with stage 3 by recursively using the calls relationship to detect consequences of possible changes on operations. The only such relationship in the dictionary involving ADDPROD, DELPROD, ADDSHIP or DELPROD1 is:

DELPROD1 calls DELPROD

Thus, the final set of operations that must be inspected is ADDPROD, DELPROD, ADDSHIP and DELPROD1. The tool then prompts the DBA to supply the changes he wants to apply to these operations. Note that DELPROD1 has to be listed after DELPROD, since the former calls the latter.

Assume that, when asked how to modify ADDPROD, the DBA replies that ADDPROD has to be modified to accommodate the new attribute of PROD and to continue to enforce ONE_N. DELPROD and ADDSHIP need be modified only to add the new column to PROD. Finally, assume that the DBA decides that DELPROD1 need not be changed at all (since the change on DELPROD does not affect DELPROD1). This concludes stage 3 and step 1.

Finally, the design tool enters step 2 and asks the DBA if all resulting changes are indeed satisfactory and, if so, creates a new schema accordingly.

5. AN EXPERT HELPER FOR DATABASE DESIGN

In this section we briefly describe a prototype software tool that helps the DBA interactively add new modules to a database schema. The prototype also partially implements the dictionary described in Section 3.

The prototype is an example of an expert helper, a concept introduced in [FM] to designate relatively small intelligent tools to help in the design, usage and maintenance of large conventional systems. The current version of the tool runs on an IBM personal computer and was written using the apes extension of micro-PROLOG [CM]. Thanks to the use of apes, the prototype is highly interactive.

The design of the tool begins by choosing a representation for a schema D suitable for micro-PROLOG. The key idea is to translate the state of the dictionary describing D (see Section 3) into a set of axioms. Each axiom will be a ground atomic formula of the form 'L1 tab L2', where tab is a binary predicate symbol (infix notation is used) and L1 and L2 are lists.

The general format of an axiom representing a relationship is

((type)(type)) tab ((name)(name)(version))

where the list ((type)(type)) expresses the relationship type, indicated by the types of the objects connected, and the list ((name)(name)(version)) expresses the individual relationship, indicated by the names of the objects ((version) denotes the particular version of the database schema).

Of all entities, only those designating modules are represented in the present version of the tool. An axiom standing for a module has the following format:

(mod) tab ((name) (kind) (version))

where (kind) is one of (primitive, subsumption, extension).

In Table 5.1 we present the correspondence between the entries of the dictionary and their axiomatic representation, as implemented by the tool.

Table 5.1 - Axiomatic Representation

Type / Entry	Axiom
is-primitive (M)	(mod) tab (M 'primitive' n)
is-sub (M)	(mod) tab (M 'subsumption' n)
is-external (M)	(mod) tab (M 'external' n)
scheme (S,L,Q)	not implemented
constraint (I,Q)	not implemented
operation (O,P,P')	not implemented
subsumes (M,N)	(mod mod) tab (M N n)
extends (M,N)	(mod mod) tab (M N n)
is-scheme-defined-in (S,M)	(sch mod) tab (S M n)
is-constraint-defined-in (I,M)	(con mod) tab (I M n)
is-operation-defined-in	

(O,M) (ope mod) tab (O M n)
is-view-over
(V,S) (sch sch) tab (V S n)
is-constraint-over
(I,S) (con sch) tab (I S n)
is-operation-over
(O,S) (ope sch) tab (O S n)
enforces
(O,I) (ope con) tab (O I n)
may-violate
(O,I) ((hid ope) con) tab (O I n)
calls
(O,P) (ope ope) tab (O P n)

Note: n is the version number

In the sequel we sketch how the prototype can be used by a DBA to add a module to a database schema. To begin the definition of a module, the DBA types module <name>. From this point on, the prototype prompts the DBA to supply all information needed to define the schemes, constraints and operations of the module. The "program" consists of the predicate 'module' which in turn calls other predicates to create the several module components. A particular module may or may not have schemes, constraints and operations. However:

- if the module M is not primitive, the DBA must list the modules M subsumes or extends;
- if the module M is defined by extension, each scheme S is a view. So, the DBA must define a mapping of S into the schemes of the modules M extends;
- for each constraint or operation O, the DBA must list all schemes O references;
- only operations of non-primitive modules may call other operations; moreover, all operations of modules created by extension are surrogates and must, therefore, include such calls. The DBA must then inform the calls relationship.

So, the presence of certain relationships (indicated by the insertion of the corresponding axiom) is compulsory, and the predicate 'module' will fail if the DBA declares that they do not exist (by typing "end" when the query is posed to him).

The prototype fixes, procedurally, the sequence to be followed by the DBA in creating the various relationships and their compulsory or optional nature. On the other hand, using the apes features unique-answer and valid-answer, the prototype separately defines, in a declarative style, the criteria to decide whether the values supplied by the DBA as answers are acceptable.

We enumerate below, per type of relationship created, the criteria that are presently enforced.

(mod) tab (x y 1)
y $\in$ {primitive, subsumption, extension}

(mod mod) tab (x y 1)
y is an active module, which must neither have been created by extension nor extended if x is being created by subsumption

(sch sch) tab (x y 1)
scheme y is accessible to some module used in the definition of the module in which the view x is being defined

(con sch) tab (x y 1)
scheme y is accessible to the module in which constraint x is being defined

(ope ope) tab (x y 1)
operation y is accessible to some module used in the definition of the module in which operation x is being defined; if the latter is defined by extension, y is related to some scheme underlying its views

(ope sch) tab (x y 1)
scheme y is accessible to the module where operation x is being defined.

(ope con) tab (x y 1)
operation x and constraint y have some scheme in common

((hid ope) con) tab (x y 1)
operation x is called by an operation of which constraint y depends

The prototype poses the relevant questions to the DBA using natural language sentences, and adopts static and dynamic menus to restrict his answers; it also ensures that names are unique throughout the database schema. Additional features of apes (which-template, in-menu, is-template) are used for these purposes.

Returning to Figure 2.1 at the end of Section 2.3 , we may now compare the implemented criteria with the requirements for correct module design. Requirements 1, 2, 4, 6 and 7 are not enforced; they would require detailed descriptions of the components. Requirements 5, 8 and 9 are explicitly enforced by the implemented criteria. Requirement 3, referring to modules created by extension, is enforced by restricting the views and operations declared in the module to the schemes and operations involved in the modules extended.

To conclude, we could certainly do more in terms of checking the consistency of modular designs using the information that is now extracted from the DBA. However, what we already check is sufficient to demonstrate the usefulness of this kind of expert helper.

6. CONCLUSIONS

We described in this paper a software tool to support the modular database design method first introduced in [TCF]. The method itself was enhanced by incorporating the hiding and enforcement clauses, and by polishing some design rules. The software tool is implemented to the point of helping the database administrator add new modules to an existing database schema. The redesign process, although not implemented, was specified in detail. Future plans include transforming the tool into a full-fledged dictionary system incorporating as much knowledge as possible about the design method.

REFERENCES

[CCF] M.A. Casanova, J.M.V. de Castilho and A.L. Furtado. "Properties of Conceptual and External Database Schemas". Proc. of the TC 2 - Working Conference on Formal Description of Programming Concepts II, Garmish-Partenkirchen (1982)

[CM] K.L. Clark and F.G. McCabe. "micro-PROLOG: programming in logic". Prentice-Hall (1984)

[DMW] W. Dosch, G. Mascari, M. Wirsing "On the Algebraic Specification of Databases". Proc. 8th Int'l Conf. on Very Large Data Bases (1982)

[EKW] H. Ehrig, H.-J. Kreowski, H. Weber. "Algebraic Specification Schemes for Data Base Systems". Proc. 4th Int'l Conf. on Very Large Data Bases (1978)

[FC] A.L. Furtado and M.A. Casanova. "Updating Relational Views", in "Query Processing in Database Systems", Springer Verlag (in print).

[FM] A.L. Furtado and C.M.O. Moura. "Expert helpers to data-based information systems". Proc. of the First International Workshop on Expert Database Systems (1984), 298-313

[HS] P. Hammond and M. Sergot. "apes: augmented PROLOG for expert systems - reference manual". Logic Based Systems Ltd. (1984)

[LMWW] P.C. Lockemann, H.C. Mayr, W.H. Weil, W.H. Wohlleber. "Data Abstractions for Data Base Systems". ACM Transactions on Database Systems 4:1 (1979)

[LZ] B. Liskov, S. Zilles. "Specification Techniques for Data Abstractions". IEEE Transactions on Software Engineering SE-1 (1975)

[Pa] D. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules". Comm. of the ACM 15:12 (1972)

[SFNC] U. Schiel, A.L. Furtado, E.J. Neuhold, M.A. Casanova. "Towards Multi-level and Modular Conceptual Schema Specifications". Inform. Systems 9:1 (1984), 43-57

[SNF] C.S dos Santos, E.J. Neuhold, A.L. Furtado. "A Data Type Approach to the Entity-Relationship Model". Int'l. Conf. of the Entity-Relationship Approach to Systems Analysis and Design (1980)

[TCF] L. Tucherman, M.A. Casanova and A.L. Furtado, "A Pragmatic Approach to Modular Database Design", Proc. of the 9th Int'l. Conf. on Very Large Data Bases, Florence, Italy (1983), 219-231

[TFC] L. Tucherman, A.L. Furtado and M.A. Casanova, "An Expert System for Modular Database Design", Technical Report CCR030, Brasilia Scientific Center, IBM Brazil (1985)

[We] H. Weber. "Modularity in Data Base Systems Design". Proc. Joint IBM/Univ. Newcastle upon Tyne Seminar (1979)