

A Software Tool for Modular Database Design

M. A. CASANOVA, A. L. FURTADO and L. TUCHERMAN
Rio Scientific Center, IBM Brazil

A modularization discipline for database schemas is first described. The discipline 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 modularized according to the discipline is then presented. It offers a user-friendly interface that guides the designer through the various stages of the creation of a new module or through the process of changing objects of existing modules. The tool incorporates, in a declarative style, a description of the design and redesign rules behind the modularization discipline, hence facilitating the incremental addition of new expertise about database design.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs—*abstract, data types, modules, packages*; H.2.1 [Database Management]: Logical Design—*schema and subschema*; H.2.3 [Database Management]: Languages—*data description languages*; I.2.1 [Artificial Intelligence]: *Applications and Expert Systems*

General Terms: Design, Experimentation, Theory

Additional Key Words and Phrases: Abstract data types, consistency preservation, encapsulation, integrity constraints, logical database design, module constructors, modular design

1. INTRODUCTION

We describe in this paper a modularization discipline for database schemas and a software tool that supports its application. Our basic motivation stems from the problem of organizing the design and maintenance of database schemas, understood here in the broad sense of a description of both the data structures (static part) and the transactions (dynamic part [7]) of an information system developed around a database.

The design of a database schema in our discipline consists essentially of the successive addition of new modules to a (possibly empty) kernel database schema. But we also recognize that it is intrinsically an iterative process, since the database designer frequently has to go back and alter the definition

Authors' address: M. A. Casanova and L. Tucherman, Rio Scientific Center/IBM Brazil, P.O. Box 4624, 20.071. Rio de Janeiro, RJ, Brazil; A. L. Furtado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rua Marquês de S. Vicente, 225, 22.453, Rio de Janeiro, RJ, Brazil.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM A0362-5915/91/0300-0209 \$01.50

of modules, either because the application evolves or because his or her perception of the application changes.

This understanding of the design process led us to develop a software tool that not only provides facilities to add a new module to an existing schema, but also helps the database designer add, delete, or modify the definition of objects of the schema. A first prototype of the tool, written in micro-PROLOG [10] extended with APES [20], is fully operational. It offers a user-friendly interface that guides the designer through the various stages of the creation of a new module or through the process of changing objects of existing modules. The prototype stores the description of a schema as clauses that the user can freely query using the facilities of micro-PROLOG. It also incorporates, in clause form, a description of the design and redesign rules behind the modularization discipline, which facilitates the incremental addition of new expertise about database design.

The usefulness of tools to help in the design of information systems based on databases is widely recognized. Among other reasons, they may be the only effective way to place formal design methods within the reach of practitioners. On the other hand, expert systems, incorporating the knowledge of human experts in some area, are being applied to solve a large number of problems. As one would expect, the combination of the two concepts, expert tools, is an active research area [6].

The *knowledge to incorporate* in an expert tool such as ours should, above all, reflect a method for designing and redesigning database schemas. If the method can be expressed by rules, preferably in a declarative style, then we can conceive an expert tool that is driven by the rules and that has its expertise progressively increased by the continuous revision and addition of new rules, corresponding to more refined versions of the method.

The design and maintenance of advanced information systems (ISs) demand flexible data models and better design disciplines, usually provided for by a conceptual modelling language [1-3, 5, 19, 22, 24-28, 33, 35]. Such languages facilitate the design of the static part of an IS through the use of semantic data models, seasoned with data types and abstraction principles such as aggregation and generalization [29]. They cope with the design of the dynamic part by incorporating a programming language that supports the types of the data model adopted and incorporates high-level control abstractions. Modularization is a concept orthogonal to those just mentioned. It imposes a design discipline to cope with size and complexity. Thus, strictly speaking, modularization does not buy additional modelling power.

Modularization, in conceptual database design, is by no means a novel idea. Almost all conceptual design languages treat a module [21, 23, 38] as a set of data structures encapsulated by a set of operations (i.e., only the declared operations can manipulate the structures). However, most of these languages permit specifying only special classes of constraints (ADAPLEX is an exception since it permits stating general constraints).

As for the basic ways of defining modules, most conceptual design languages offer some constructor for defining external schemas. In particular, RIGEL [24] and PLAIN [35] include view update translations in their version

of the constructor, as we do. But, apart from this constructor, the proposals differ considerably. Weber [36] structures the use of modules in the tradition of software engineering; ASTRAL [3] uses a block structure policy to limit the scope of data structure declarations; Galileo [1] uses the notion of environment, which is a denotable value, as a modularization mechanism; ADAPLEX [28] adopts a form of Ada packages for this purpose; TAXIS [22] is more revolutionary in that it uses the same abstraction principles to structure both data and procedures, thus resulting in a very powerful (albeit unusual) programming language. Many other structured conceptual design proposals exist in the areas of programming languages and AI. We would like to note that CLEAR [8, 18] contains several proposals for organizing the definition of theories. The modularization discipline described in this paper was first introduced in [31] and the software tool was partially described in [32].

This paper is divided as follows. Section 2 describes the basic concepts of the modularization discipline. It defines the concept of module, introduces the module constructors, and describes a complete example. Section 3 discusses how to test the correctness of a modular schema. Section 4 describes the architecture of the tool that concerns the creation of a modular schema. Section 5 outlines the aspects of the tool related to the redesign of a modular schema. Finally, Section 6 contains conclusions and directions for future research.

2. A MODULARIZATION DISCIPLINE FOR DATABASE DESIGN

This section contains the basic concepts of the modularization discipline for database design we propose. Section 2.1 introduces the concepts through a simple example. Section 2.2 formally defines the concept of a modular database schema and the types of modules that form the basis of the discipline. Finally, Section 2.3 discusses the design requirements that specify the discipline.

2.1 An Initial Example

We will illustrate the basic concepts of our modularization discipline by designing a simple database that stores information about products, warehouses, and shipments of products to warehouses.

A few preliminary remarks about the modularization discipline will be helpful. Briefly, it proposes to collect structures, integrity constraints, and operations that are closely related into separate modules. The constraints of a module act as a declarative documentation of additional semantics of the data, whereas operations incorporate such semantics procedurally. However, one does not replace the other. Both coexist in the conceptual model of the enterprise, even with a certain redundancy. The discipline also dictates that the structures of a module must be updated only by the operations defined in the module, which correspond to the usual notion of encapsulation [21]. Hence, if the operations of each module preserve consistency with respect to the integrity constraints of the module, the discipline introduces an effective

```

module PRODUCT
  schemes
    PROD[P #,NAME]
  constraints
    ONE_N  $\forall p \forall n \forall n' (\text{PROD}(p,n) \ \& \ \text{PROD}(p,n') \Rightarrow n = n')$ 
  operations
    ADDPROD(p,n)
      if  $\neg \exists n' \text{PROD}(p,n')$ 
      then insert (p,n) into PROD
    DELPROD(p):
      delete PROD(x,y) where x = p
  enforcements
    ADDPROD enforces ONE_N
endmodule

```

Fig. 1. Definition of module PRODUCT

way to guarantee logical consistency of the database. Yet, queries remain unrestrained, just as in the traditional design strategies [30, 37].

The design of a modular database schema essentially consists of the successive addition of new modules to a (possibly empty) kernel database schema, following a bottom-up style. There are three types of modules, described in more detail below. Briefly, a *primitive module* is totally isolated from others; a *subsumption module* is built upon other modules and may redefine their operations to preserve new constraints; finally, an *external module* introduces views together with their operations.

To conclude the preliminary remarks, it should also be noted that the modularization discipline is largely independent of the underlying DDL/DML, but it will be constructed here on top of a generic relational DDL/DML.

We begin the design of our simple database by creating a primitive module, PRODUCT, that represents data about products and contains the operations allowed on products. Module PRODUCT is shown in Figure 1.

The statements under the headings “schemes,” “constraints,” and “operations” in Figure 1 capture familiar concepts and, therefore, need no explanation. The statement “ADDPROD enforces ONE_N” indicates that ADDPROD has a test to guarantee that ONE_N is never violated.

In general, to introduce new relation schemes, integrity constraints, and operations that have no connections to the objects of the current database schema, the DBA may choose to define a new primitive module M . The DBA must primarily guarantee that each operation defined in M preserves consistency with respect to the constraints of M .

The DBA must include an enforcement clause of the form “ O enforces I ” in the definition of M 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 the updates in O will not violate I . The DBA must explicitly provide the enforcement clause because, on the one hand, it plays a fundamental role when one considers the problem of redesigning modules, but, on

```

module WAREHOUSE
  schemes
    WAREHSE[W#,LOC]
  constraints
    ONE_C:  $\forall w \forall c \forall c' (WAREHSE(w,c) \& WAREHSE(w,c') \Rightarrow c = c')$ 
  operations
    OPEN(w,c)
      if  $\neg \exists c' WAREHSE(w,c')$ 
      then insert (w,c) into WAREHSE
    CLOSE(w).
      delete WAREHSE(x,y) where  $x = w$ 
  enforcements
    OPEN enforces ONE_C
endmodule

```

Fig. 2. Definition of module WAREHOUSE.

the other hand, it cannot be easily derived from the definition of constraints and operations. Enforcement clauses also provide useful documentation, tying together the declarative semantics of constraints to the procedural semantics of operations.

The modular database schema contains at this point only one module, PRODUCT. We then add another module, WAREHOUSE, to represent warehouses and operations on warehouses. Since such data will be completely dissociated from the data about products, module WAREHOUSE can also be defined as another primitive module, as shown in Figure 2.

The modular database schema now has two modules, PRODUCT and WAREHOUSE. We continue the design of the modular schema by defining a new module, SHIPMENT, that introduces a new relation scheme, SHIP, that represents a relationship between products and warehouses. However, since a shipment (p, w) requires that product p and warehouse w indeed exist, the definition of SHIPMENT must also include the two constraints below:

$$\text{INC_P: } \forall p (\exists w \exists q \text{ SHIP}(p,w,q) \Rightarrow \exists n \text{ PROD}(p,n))$$

$$\text{INC_W: } \forall w (\exists p \exists q \text{ SHIP}(p,w,q) \Rightarrow \exists c \text{ WAREHSE}(w,c))$$

But this creates problems. First, these constraints involve relation schemes defined in two separate modules, PRODUCT and WAREHOUSE, besides the new relation scheme we want to introduce. Second, an execution of the operation DELPROD may violate INC_P and an execution of the operation CLOSE may violate INC_W. Therefore, unlike the definition of primitive modules, the definition of SHIPMENT must allow its constraints to refer to the relation schemes defined in PRODUCT and WAREHOUSE and it must indicate that users cannot have access to DELPROD and CLOSE anymore to avoid violations of INC_P and INC_W. Although not mandatory, the definition of SHIPMENT should also provide replacements for DELPROD and CLOSE.

To solve such problems, we introduce a second type of module, called a *subsumption module*. We then define a subsumption module SHIPMENT over modules PRODUCT and WAREHOUSE, as shown in Figure 3.

```

module SHIPMENT subsumes PRODUCT, WAREHOUSE with
schemes
  SHIP[P #, W #, QTY]
constraints
  ONE_Q  $\forall p \forall w \forall q \forall q' (\text{SHIP}(p, w, q) \ \& \ \text{SHIP}(p, w, q') \Rightarrow q = q')$ 
  INC_P  $\forall p (\exists w \exists q \text{SHIP}(p, w, q) \Rightarrow \exists n \text{PROD}(p, n))$ 
  INC_W  $\forall w (\exists p \exists q \text{SHIP}(p, w, q) \Rightarrow \exists c \text{WAREHSE}(w, c))$ 
operations
  ADDSHIP(p,w,q)
    if  $\exists n \text{PROD}(p, n) \ \& \ \exists c \text{WAREHSE}(w, c) \ \& \ \neg \exists q' \text{SHIP}(p, w, q')$ 
    then insert (p,w,q) into SHIP
  CANSHIP(p,w)
    delete SHIP(x,y,z) where (x = p & y = w)
  CLOSE1(w)
    if  $\neg \exists p \exists q \text{SHIP}(p, w, q)$  then CLOSE(w)
  DELPROD1(p)
    if  $\neg \exists w \exists q \text{SHIP}(p, w, q)$  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

```

Fig. 3. Definition of module SHIPMENT

The statements under the headings “schemes,” “constraints,” “operations,” and “enforcements” in Figure 3 have the same meaning as for primitive modules. The statements under the “hiding” heading indicates that the execution of DELPROD and CLOSE may violate constraints defined in SHIPMENT and, therefore, these operations must be hidden from the users.

The DBA must then use the subsumption constructor to define a new module M over modules M_1, \dots, M_n when he or she wants to

- create new schemes, constraints and operations; exactly as if M were a primitive module;
- hide some of the operations of M_1, \dots, M_n , if they violate a new constraint.

More precisely, suppose that the DBA wants to add, among other objects, a new constraint I that may be violated by an operation O' defined in an existing module M' . The DBA must then define a subsumption module M over M' , include I in the definition of M , and hide O' in M by including the statement “ O' may violate I .” The DBA may also define a new operation O in M that has almost the same functions as O' , except that O enforces I .

After the DBA defines a subsumption module M over M_1, \dots, M_n , he or she can no longer use M_i to define new modules, and users can no longer access module M_i , for $i = 1, \dots, n$. That is, these modules become *inactive*. But users of module M see the relation schemes and integrity constraints of M_i through M , since M *imports* these objects from M_i . Moreover, users can

```

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
    view definition mappings
      DELVRY(p,w)  $\exists$ q SHIP(p,w,q)
    surrogates
      DEL(p,w).
      CANSHIP(p,w)
endmodule

```

Fig 4. Definition of module DELIVERY.

access all operations of M_i through M , except those that are hidden in M . That is, hidden operations also become *inactive*. This restriction is necessary because an execution of a hidden operation might lead to a consistency violation. But active operations may call hidden operations as subroutines, without fear of violating consistency, if all design requirements are met (see Section 2.3).

We also say that a subsumption or primitive module is a *conceptual module*.

The modular database schema now has three modules, SHIPMENT, WAREHOUSE, and PRODUCT, but only SHIPMENT is active since SHIPMENT subsumes WAREHOUSE and PRODUCT. Therefore, users only see the SHIPMENT module and the DBA can only use this module to create new modules. Module SHIPMENT contains all relation schemes and constraints of PRODUCT and WAREHOUSE, plus a newly defined relation scheme and three new constraints. The active operations 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 the constraints INC_P and INC_W of SHIPMENT, respectively, they are hidden in SHIPMENT. Thus, users see all relation schemes, constraints, and operations defined thus far, except CLOSE and DELPROD.

Finally, we want to introduce a view over SHIP, together with operations on the view and their translations to the base relations. This can be accomplished by resorting to a third type of module, called an *external module*. We then define an external module DELIVERY over SHIPMENT as shown in Figure 4.

The statements under the view definition mappings and the surrogates headings indicate how to translate queries and operations on DELVRY into queries and operations on SHIP.

More precisely, the statement “DELVRY(p,w) : \exists q SHIP(p,w,q)” defines the view DELVRY in terms of the relation scheme SHIP introduced in module SHIPMENT. Hence, the underlying database management system can

```

schema SHIPMENT_CONTROL with
  module PRODUCT      ,
  module WAREHOUSE    ,
  module SHIPMENT     ;
  module DELIVERY     ,
endschema

```

Fig. 5. Definition of schema SHIPMENT_CONTROL

translate a query Q on DELVRY into a query on SHIP simply by replacing each expression of the form “DELVRY(t,u)” in Q by “ $\exists q$ SHIP(t,u,q)” (assuming a syntax for queries similar to that of the first-order languages).

However, there is no uniform way to unambiguously translate updates on views into updates on base relations (the so-called *view update problem* [11, 14, 17]). Therefore, we require that the DBA explicitly provide, as part of the definition of DELIVERY, a correct translation for the operation DEL, which is the operation under the surrogates heading. Hence, the operation defined in the “operations” heading just informs the user the meaning of DEL in terms of its effect on the relation scheme DELVRY. A call to DEL will actually invoke the operation defined under the “surrogates” heading.

In general, the DBA should define an external module M over active modules M_1, \dots, M_n when he or she wants to introduce views over the relation schemes of M_1, \dots, M_n and operations on these views. A user of module M sees only the newly defined relation schemes, integrity constraints, and operations. An external module M is always active and may be used to create new external modules, but it cannot be used to create new subsumption modules.

The relations schemes, constraints, and operations of M have no independent existence in the following sense. First, for each relation scheme R defined in M , the DBA must provide a *view definition mapping* defining R in terms of the relation schemes of M_1, \dots, M_n . Second, the DBA must make sure that the constraints defined in M are logical consequences of the constraints of M_1, \dots, M_n , that is, the semantics of M must be wholly derived from that of M_1, \dots, M_n . Finally, for each operation named f and defined in M , the DBA must provide a *surrogate* for f , which must be an exact and acceptable translation [14] of the operation into the operations of M_1, \dots, M_n . A call to f will generate an execution of the surrogate and, hence, the definition of f just describes the operation in terms of the schemes defined in M .

The final modular database schema, called SHIPMENT_CONTROL, has the format shown in Figure 5. It has one active conceptual module, SHIPMENT, one external module, DELIVERY, and two inactive conceptual modules, PRODUCT and WAREHOUSE. Through the module SHIPMENT, users have access to the base relation schemes (using traditional terminology) PROD, WAREHSE, and SHIP, and to the active operations ADDSHIP, CANSHIP, ADDPROD, DELPROD1, OPEN, and CLOSE1. Users of the module DELIVERY see only one view, DELVRY, and one operation, DEL.

A user has, in principle, access to all active conceptual modules and all external modules of a modular database schema. Hence, he or she sees all

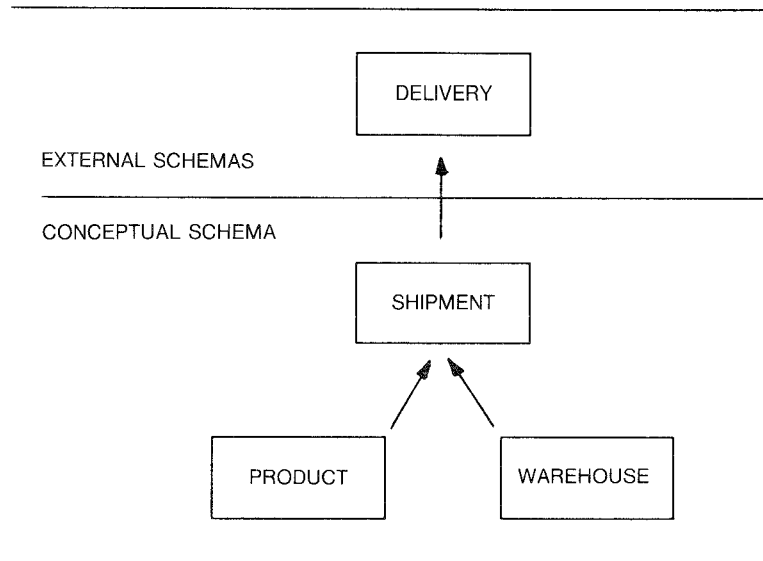


Fig 6. The modular structure of SHIPMENT_CONTROL.

relation schemes and integrity constraints defined in all modules, but he can only update the database using the active operations. He or she can also freely query any relation scheme. The DBA may naturally restrict the privileges of a user by allowing the user to access only certain external schemas. The DBA may define new subsumption modules only over modules that are active and are not external, and may define new external modules only over active modules, either primitive, external, or defined by subsumption.

In terms of the ANSI/X3/SPARC architecture [4], the conceptual schema corresponds to a forest of primitive and subsumption modules. The roots of the forest are the active modules. The external schemas correspond to external modules, which form a partial order stemming from the active conceptual schema modules. Figure 6 shows the modular structure of the SHIPMENT_CONTROL schema just defined.

2.2 Formal Definitions

This section formally defines the concepts of module, module type, and modular database schema and lists some basic syntactical requirements that all modular database schemas must satisfy. It also introduces a language to describe modules.

We first introduce the classes of statements that are the building blocks of modules.

A *relation scheme* is a statement of the form $R[A_1, \dots, A_n]$, where R is the *relation name* and A_1, \dots, A_n are the *attributes* of the scheme. To avoid

naming conflicts when considering different modules, we use, when necessary, $M.R$ to refer to scheme R defined in module M . The same observation also applies to the names of constraints and operations. Attributes play no significant role in our approach and are not mentioned further.

An *integrity constraint* is a statement of the form $C:Q$, where C is the *name* of the constraint and Q is a well-formed formula of the language adopted to write constraints.

An *operation* is a procedure definition of the form “ $f(x_1, \dots, x_n; y_1, \dots, y_m):s$,” where f is the *name* of the operation, x_1, \dots, x_n are the *input parameters*, y_1, \dots, y_m are the *output parameters*, and s is the *body* of the operation, which is a program in the DML adopted.

An *enforcement clause* is a statement of the form “ O enforces I_1, \dots, I_k ,” where O is the name of an operation and I_j is the name of a constraint, for each $j = 1, \dots, k$.

A *hiding clause* is a statement of the form “ O may violate I_1, \dots, I_k ,” where O is the name of an operation and I_j is the name of a constraint, for each $j = 1, \dots, k$.

A *view definition mapping* for a relation scheme $R[A_1, \dots, A_k]$ is a statement of the form $R(x_1, \dots, x_k):Q$, where Q is a well-formed formula with k free variables, ordered x_1, \dots, x_k , of the language used to write view mappings. The well-formed formula “ $\forall x_1 \dots \forall x_k (R(x_1, \dots, x_k) \Leftrightarrow Q)$ ” is called the *view definition axiom* for R .

Finally, given an operation “ $f(x_1, \dots, x_n; y_1, \dots, y_m):r$,” a *surrogate* for f is simply another operation with the same name and parameters as f .

Every module may have relation schemes, integrity constraints, and operations and, depending on its type, may also have statements of the other types. The definitions that follow introduce modules exactly as illustrated before, but without the syntactical sugar used in Section 2.1.

More precisely, a *module* is a tuple of the form $M = (m, t, \mathbf{r}, \mathbf{c}, \mathbf{o}, \mathbf{a}, \mathbf{d})$, where

- m is the *name* of M ;
- t is the *type* of M ;
- \mathbf{r} is a set of relation schemes such that no two schemes in \mathbf{r} have the same relation name;
- \mathbf{c} is a set of integrity constraints such that no two constraints in \mathbf{c} have the same name;
- \mathbf{o} is a set of operations such that no two operations in \mathbf{o} have the same name;
- $\mathbf{a} = [a_1, \dots, a_k]$ is the list of *auxiliary objects* of M ; and
- \mathbf{d} is the (possibly empty) set of modules used to define M .

The values that \mathbf{a} and \mathbf{d} may take depend on the type t of the module and they are defined in what follows.

A *primitive module* is a tuple of the form $M = (m, \mathbf{P}, r, \mathbf{c}, \mathbf{o}, [\mathbf{e}], \emptyset)$ where \mathbf{e} is a set of enforcement clauses.

A *subsumption module* is a tuple of the form $M = (m, \mathbf{S}, r, \mathbf{c}, \mathbf{o}, [\mathbf{e}, \mathbf{h}], \mathbf{d})$ where $\mathbf{d} \neq \emptyset$, \mathbf{e} is a set of enforcement clauses, and \mathbf{h} is a set of hiding clauses. We say that each operation referenced in a hiding clause in \mathbf{h} is *hidden* by M . Let M' be a module in \mathbf{d} . We also say that each relation scheme and each integrity constraint defined or imported by M' is *imported* by M and that each operation defined or imported by M' , but not hidden in M , is *imported* by M . Finally, we say that M was *defined by subsumption* over M' or that M *subsumes* M' .

As for the third and last type, an *external module* is a tuple of the form $M = (m, \mathbf{E}, r, \mathbf{c}, \mathbf{o}, [\mathbf{v}, \mathbf{s}], \mathbf{d})$ where $\mathbf{d} \neq \emptyset$, \mathbf{v} contains, for each scheme $R[A_1, \dots, A_k]$ in r , a view definition mapping, and \mathbf{s} contains a surrogate for each operation in \mathbf{o} . We also say that M was *defined by extension* over M' and that M *extends* M' , for each $M' \in \mathbf{d}$.

A *modular database schema* is a pair of the form $S = (s, [M_1, \dots, M_r])$ where s is the *name* of the schema and (M_1, \dots, M_r) is a list of modules such that no two modules have the same name.

A *conceptual module* of S is either a primitive module or a subsumption module. A module M_i is *active* in S iff either M_i is an external module or M_i is a primitive or subsumption module such that there is no subsumption module M_j , with $1 \leq i < j \leq r$, defined over M_i . An operation O is *active* in S iff O is an operation defined in or imported by an active module of S . Note that, by definition, an active operation is nowhere hidden.

2.3 Requirements for Correct Modular Database Schemas

The definitions in Section 2.2 just introduce the syntax of modules, but they do not guarantee all properties we want from a modular database schema. So, we introduce certain design requirements that imply that

- (1) for each relation scheme, 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; and
- (3) every operation directly accessible to the users preserves consistency with respect to all constraints.

An analysis of the last property is especially important to understand why modularization is so interesting for the design of large schemas. Without the design requirements we introduce in this section, to guarantee (3), the DBA would have to check that EACH operation of each module preserves consistency with respect to ALL constraints of all modules. The judicious choice of the requirements alters the problem of checking consistency preservation in the following way:

- the database designer must explicitly check if each operation defined in a newly introduced module M preserves consistency with respect to all the constraints defined in M ;

REQUIREMENTS FOR A PRIMITIVE MODULE

- P_1 . for each constraint I defined in M , I can only refer to schemes defined in M
- P_2 . for each operation O defined in M , O can query and update only schemes defined in M , as well as call only operations defined in M
- P_3 . for each operation O defined in M , for each constraint I defined in M , I must be an invariant of O .
- P_4 . for each relation scheme S defined in M , there must be an operation O defined in M that performs insertion into S .
- P_5 . for each operation O defined in M , there is an enforcement clause in M of the form " O enforces I_1, \dots, I_m " iff $\{I_1, \dots, I_m\}$ is the set of all constraints of M whose definition O has to take into account

Fig 7. Requirements for a primitive module

- the database designer need not check that the operations defined in M preserve consistency with respect to the constraints of modules previously introduced, since this will be automatically guaranteed by the requirements;
- 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 requirements, except if M is defined by subsumption over M' , when operations of M' may have to be hidden in M .

The list of requirements for primitive modules appears in Figure 7 and their rationale goes as follows. Requirement P_1 is just syntactical and indicates that constraints of a primitive module can only talk about schemes of the module. Requirement P_2 is also syntactical and implies that an operation of a primitive module affects only the schemes defined in the module. Requirement P_3 states that each operation must preserve consistency with respect to all constraints defined in the module [9]. Requirement P_4 guarantees that each scheme S of a primitive module has at least one operation that inserts tuples in S . Requirement P_5 defines when the DBA has to introduce an enforcement statement.

The requirements for subsumption modules appear in Figure 8 and have the following intuition. Let M be a subsumption module defined over modules M_i , $i = 1, \dots, n$.

Requirement S_0 forbids the DBA to define M by subsumption over M_i if there is a third module M' that subsumes or extends M_i . This avoids the undesirable situation in which M subsumes M_i and yet M' offers an indirect path to the operations of M_i . More precisely, if this requirement were violated, we would have a module M subsuming M_i and yet we would permit operations of M' call operations of M_i that might be hidden in M . Thus, we would not be able to assure that the operations of M' preserve consistency with respect to the constraints of M . Conversely, if M' subsumes M_i , we would permit operations of M call operations of M_i that might be hidden in

REQUIREMENTS FOR A SUBSUMPTION MODULE

- S₀. for each module M' used to define M , M' must be an active conceptual module at the time M is created and M' must not have been used to create an external module.
- S₁. for each constraint I defined in M , I can only refer to schemes defined in or imported by M .
- S₂. for each operation O defined in M , O can query schemes and call operations defined or imported by M , but O can update only schemes defined in M .
S₃, S₄ and S₅: identical to P₃, P₄ and P₅, respectively.
- S₆. for each operation O imported by M , there is a hiding clause of M of the form " O may violate I_1, \dots, I_m " iff $\{I_1, \dots, I_m\}$ is the set of all constraints defined in M that are not preserved by O .
- S₇. for each hiding clause " O may violate I_1, \dots, I_m " of M , there must be an operation O' defined in M such that O' calls O .
- S₈. for each hiding clause " O may violate I_1, \dots, I_m " of M , for each operation O' defined in M that calls O , there must be a statement of the form " O' enforces J_1, \dots, J_n " in M such that $\{I_1, \dots, I_m\} \subseteq \{J_1, \dots, J_n\}$.

Fig. 8. Requirements for a subsumption module.

M' , thus leading to consistency violations with respect to the constraints of M' .

Requirement S_1 is similar to P_1 , but it indicates that constraints of a subsumption module can also refer to schemes imported by the module.

Requirement S_2 guarantees that each operation O of M preserves consistency with respect to the constraints of M_i since O updates the schemes of M_i only through calls to operations of M_i , for each $i = 1, \dots, n$. Requirement S_3 states that each operation defined in M preserves consistency with respect to the constraints defined in M . Hence, these requirements together imply that each operation defined in M preserves consistency with respect to the constraints of M, M_1, \dots, M_n .

Requirement S_4 guarantees that each scheme S of a subsumption module has at least one operation that inserts tuples in S .

Requirements S_5 and S_6 define when the DBA has to introduce enforcement and hiding statements, respectively. Requirement S_6 also guarantees that each operation of M_i that is not hidden in M preserves consistency with respect to the constraints of M . By the syntactical requirements on operations and constraints that isolate modules from each other, each operation of M_i preserves consistency with respect to the constraints of M_j , for i, j in $[1, n]$ with $i \neq j$. By an inductive argument, since M_i was defined before M , we may also assume that each operation of M_i preserves consistency with respect to the constraints of M_i , we may conclude that each operation of M_i that is not hidden preserves consistency with respect to the constraints of M, M_1, \dots, M_n .

Requirement S_7 guarantees that operations that are hidden in M do not become inactive.

Requirement S_8 forces the DBA 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

REQUIREMENTS FOR AN EXTERNAL MODULE

- E₀. for each module M' used to define M , M' must be an active conceptual module or an external module at the time M is created.
- E₁. identical to P₁.
- E₂ for each constraint I of M , I' must be a logical consequence of the constraints defined in modules used to create M , 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 " $R[A_1, \dots, A_j] : Q$ " is the view definition of R described in M and the list of free variables of Q is x_1, \dots, x_j .
- E₃. identical to P₂
- E₄. for each surrogate of O of M , O can query schemes or call operations defined in any of the modules used to define M , but O cannot directly update any scheme
- E₅. for each operation O of M , the surrogate of O must be an exact and acceptable translation of O

Fig. 9 Requirements for an external module.

contains tests to guarantee preservation of I , as otherwise the call to O might violate I . Therefore, the DBA must indicate that O' enforces I in some enforcement clause.

Finally, Figure 9 shows the requirements for external modules. Let M be an external module defined over modules M_i , $i = 1, \dots, n$.

Requirement E_0 forbids the DBA to define a module M over M_i if there is a third module M' that subsumes M_i . If this requirement were violated, we would have a module M whose operations might generate calls to operations of M_i that are hidden in M' . Thus, we would not be able to assure that calls to operations of M would not violate constraints of M' .

Requirement E_1 is syntactical and indicates that constraints of an external module can only refer to schemes of the module.

Requirement E_2 guarantees that the integrity constraints of M follow from those of M_1, \dots, M_n , 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.

Requirements E_3 and E_4 are purely syntactical. However, Requirement E_4 also guarantees that each surrogate O preserves consistency with respect to the constraints of M_i since O updates the schemes of M_i only through calls to operations of M_i , for each $i = 1, \dots, n$.

Requirement E_5 guarantees that a surrogate O' for an operation O correctly implements O in the sense that O and O' have the same effect as far as the views are concerned [14].

To conclude, we observe that the truth of certain requirements depends on the semantics fixed for the DDL/DML selected as the starting point. For example, we cannot check if an operation preserves a constraint unless we know the semantics of the languages in which the operation and the constraint were written. However, if the semantics is chosen so that all requirements are satisfied, then we have the following.

THEOREM. *Let S be a modular database schema. Suppose that, for each module M_k defined in S , M_k satisfies all requirements pertaining to its sur-*

Then

- (1) for each relation scheme R defined in a conceptual module of S , there is at least one operation that adds tuples to R ;
- (2) every operation of S is accessible to the users either directly or through calls from other operations;
- (3) every active operation of S preserves consistency with respect to all constraints defined in S .

The proof of this result follows from the discussion about the design requirements.

3. MOVING FROM DESIGN REQUIREMENTS TO A DESIGN TOOL

3.1 An Analysis of the Design Requirements

We begin in this section the description of a software tool that supports the modularization discipline described in Section 2. The tool is built around a dictionary that stores modular schemas, and it offers advice to the DBA so that the modular schemas he or she 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 comprises four major steps: (1) design of the conceptual schema of the dictionary; (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; (3) mapping of the design requirements into a strategy for gradually defining the objects of a correct modular schema; and (4) implementation of the tool.

The first step is a routine task and the second step was already discussed in Section 2. This section concentrates on the third step, leaving a discussion about the fourth step to Sections 4 and 5. Specifically, this section addresses two problems: how to test for the requirements and when to apply such tests. The first problem depends on a classification for the requirements; the second depends on the introduction of what we call 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 is correct with respect to requirement P_3 , which states that for each operation O of a primitive module M of S , for each constraint I of M , I must be an invariant of O . Now recall that a primitive module in S is a tuple of the form $(m, \mathbf{P}, \mathbf{r}, \mathbf{c}, \mathbf{o}, [\mathbf{e}], 0)$ where \mathbf{r} , \mathbf{c} , \mathbf{o} , \mathbf{e} , are the sets of relation schemes, constraints, operations and enforcements of the module. Then, we can formalize P_3 by the following *defining formula*:

$$\forall m \forall \mathbf{r} \forall \mathbf{c} \forall \mathbf{o} \forall \mathbf{e} ((m, \mathbf{P}, \mathbf{r}, \mathbf{c}, \mathbf{o}, [\mathbf{e}], 0) \in S \Rightarrow \forall i \forall f (i \in \mathbf{c} \wedge f \in \mathbf{o} \Rightarrow p_3(i, f))) \quad (1)$$

where the intended interpretation of " $p_3(i, f)$ " is that operation f preserves constraint i . Hence, P_3 *constrains* the set of constraints and operations of each primitive module of S and is a *semantic* requirement because the truth of $p_3(i, f)$ depends on the meaning of i and f .

The formula in (1) should not be directly implemented because the design tool should not simply test if the complete definition of a schema S is correct, but rather it should help the database designer gradually specify a correct modular database schema. Hence, it is reasonable to replace the defining formula of P_3 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 P_3 constrains. However, P_3 is doubly neutral regarding these points since it neither indicates whether constraints should be defined before operations or vice-versa nor indicates whether an operation should be changed if it may violate a constraint, or vice-versa. To circumvent this problem we postulate that constraints *take precedence* over operations and define the *test associated with P_3* as follows

$$\forall i(i \in \mathbf{c} \Rightarrow p'_3(i, f)) \quad (2)$$

where \mathbf{c} is the set of constraints of the primitive module M currently being defined. We also say that the operations of a primitive module *govern* the application of the test.

As far as the tool is concerned, these concepts have the following implications:

- the tool must guarantee that all constraints of a module be defined before all operations;
- when each operation is defined, the tool must test if the operation never violates each constraint; and
- if an operation fails the test, the tool must force the DBA to change the operation, 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 tool applies 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 tool implements;
- a class of sets of objects of a module, or imported by the module, that R *constrains*; and
- 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 (see [15] for a complete classification of the constraints). Syntactical requirements impose restrictions on the syntax of modules or objects; semantic requirements have to do with the semantics of operations or constraints. A type 0 requirement imposes simple conditions on the definition of a module; a type 1 requirement imposes restrictions on a class of sets of objects; a type 2 requirement also imposes restrictions on a class of sets of objects but, as fully

explained in Section 3.2, its test can only be applied when all objects in its governing set are defined.

We now address the second problem, that is, how to decide when the tool applies the tests associated with the requirements. To settle this question, we introduce the concept of a definition order among sets of objects, denoted by “ $<$.” 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 .

More precisely, 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 r , c , o , h , e , v , and s be the sets of schemes, constraints, operations, hiding statements, enforcement statements, view definitions, and surrogates, respectively, defined in M . Then, $r < v < c < h < o < e < s$.

(*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 $X < X'$.

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

PROPOSITION. *Let X_1, \dots, X_n be the sets of objects defined or imported by a module that are constrained by a requirement R of type 1 or 2, and let X_i be the set governing the application of the test associated with R . Then 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 have been defined when the test is actually applied, if the sets of objects are specified in definition order.

To conclude, we observe that the notion of a definition order has an intrinsic importance since it reduces the complexity of correctly defining a modular schema. Indeed, ignoring the problem of building a design tool, the definition order offers some guidance to the database designer during the process of specifying the often closely interwoven sets of objects that compose a modular schema.

The design tool forces the database designer to specify the objects of a modular schema in definition order. But he may bypass the order in certain situations by switching to the redesign tool. Sections 4 and 5 discuss these points in more detail.

3.2 Testing the Requirements during the Design of a Schema

In this section we outline a strategy for defining modular schemas that guarantees correctness with respect to the design requirements.

First of all, the DBA must specify the modules and the sets of objects in definition order. To guarantee correctness with respect to a requirement R , the strategy dictates the following. If R is of type 0, the test associated with R must be applied 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 the proposition at the end of Section 3.1, when the DBA specifies the objects of the set X that governs R , all other sets of the module currently being constrained (or imported by the module) by R have already been defined. Then, if R is of type 1, the test associated with R must be applied when the designer specifies each object x in X . Moreover, if x fails the test, x must be changed (and not the objects in the other sets constrained by R). But if R is of type 2, the test associated with R must be applied when the designer signals that he or she has specified all objects in X . Because of the way tests are defined, this strategy guarantees that the final modular schema satisfies R .

However, the semantic type 2 requirements P_5 , S_5 , and S_6 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. By contrast, the syntactic requirement S_8 should be understood as a way of automatically generating enforcement clauses.

In general, the test associated with the requirements pertaining to schemes, constraints, or operations act as *assertions* [13] and they are used to accept or reject the definition of objects. On the other hand, the tests of the requirements referring to hiding or enforcement statements act as *triggers* [12] to synthesize clauses.

3.3 Testing the Requirements during the Redesign of a Schema

We now sketch a strategy for modifying modular schemas without losing correctness with respect to the design requirements. To simplify the discussion, we assume that the DBA is not allowed to change the modular structure of a schema, which implies that type 0 requirements are never violated.

To illustrate the problems raised by a change f to a scheme, constraint, or operation, f may be:

- rejected, because it violates some requirement;
- unconditionally accepted;
- accepted, provided that further changes be made to restore correctness.

The choice between these possibilities is intimately connected with the definition order introduced in Section 3.1. For example, recall that in a primitive module the definition order was schemes, constraints, operations, and enforcement statements. Hence, when the DBA inserts a new constraint, the tool must (1) reject the insertion if the definition of the constraint refers to a scheme not defined in the module (and not ask the DBA to insert a new scheme); (2) inform the DBA to recheck all operations to see if they preserve the constraint (and not reject the insertion); and (3) if an operation is redefined, inform the DBA to recheck enforcement clauses over the operation.

In general, to redesign a schema, the DBA must again consider sets of objects in definition order. Let X be a set of schemes, constraints, or operations. Correctness with respect to a type 1 requirement R governed by X can be established as follows. If the DBA defines some change on an object x of X that creates a new definition d for x , the correctness of d against R

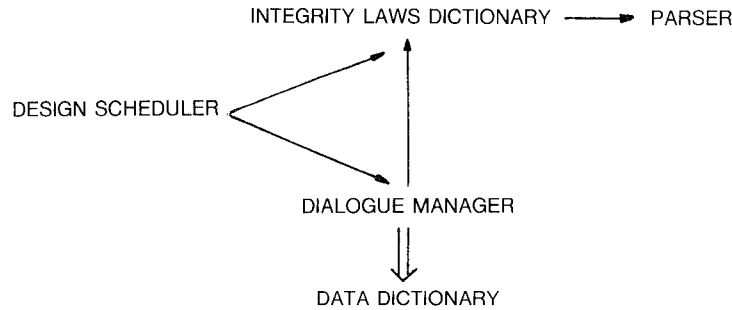


Fig. 10. Major components of the Design Tool.

can be tested exactly as in Section 3.2. If the DBA decides to delete x , the change can be immediately applied without violating R , by the way type 1 requirements are defined. If the DBA defines no change on x , the definition of x must be rechecked against R , in principle as in Section 3.2. Correctness with respect to requirements of type 2 governed by X can be established exactly as in Section 3.2, but only after all changes to X have been applied.

Finally, let X be a set of enforcement and hiding statements. The DBA is allowed to redefine such statements only to reflect changes in constraints and operations, based on the semantic type 2 requirements P_5 , S_5 , and S_6 , as well as the syntactic requirement S_8 .

4. THE DESIGN TOOL

As already mentioned in Section 3.1, the prototype software tool is divided into the *design tool*, described in this section, that helps the DBA create new modules, and the *redesign tool*, described in Section 5, that helps the DBA modify the objects of old modules. Both tools run on IBM personal computers and were implemented as logic programs written in micro-PROLOG augmented with APES.

The design tool has the following major components (see Figure 10): a *parser*, a *data dictionary*, an *integrity laws dictionary*, a *design scheduler*, and a *dialogue manager*. Figure 10 displays the components of the design tool and indicates the flow of control. Simple arrows indicate that one component activates the other and the double arrow denotes output.

A brief description of each component now follows. The parser checks the context-free syntax and constructs parse trees for formulas of the language used to describe constraints and view definition mappings and for operations of the DML adopted to describe operations.

The data dictionary contains a description of the modular schema S (there is only one for simplicity) in the form of a set of *dictionary clauses*.

The integrity laws dictionary consists of a set of clauses that capture the tests associated with the design requirements, as described in Section 3.1. Presently, the laws fully implement only the syntactical requirements and treat the semantic requirements simply by querying the DBA about certain conditions and trusting the answers.

The design scheduler and the dialogue manager use integrity laws corresponding to type 1 requirements as assertions to block the definition of a new scheme, constraint, or operation; those corresponding to some of the type 2 requirements act as assertions to force the DBA to revise a set of schemes, constraints, or operations; and those for Requirements P_5 , S_5 , S_6 , and S_8 act as triggers to create enforcement or hiding clauses. Hence, there are arrows from the design scheduler and the dialogue manager to the integrity laws dictionary.

The integrity laws use the parser in two ways (hence the arrow from the integrity laws dictionary to the parser). First, they use the parser to check the context-free syntax of a formula W describing a view definition or a constraint or the context-free syntax of a program P defining an operation or an operation surrogate. Hence, syntactically wrong formulas or programs are immediately rejected. Second, they traverse the parse trees produced for W (or P), looking for subexpressions of the appropriate form, to check context-sensitive conditions associated with W (or P). For instance, recall the Requirement P_2 mandates, among other things, that operations introduced in a primitive module M can only call operations created in M . Corresponding to P_2 , there is an integrity law that, when invoked with a program P of M , isolates each call statement C within the parse tree of P and then it inspects the dictionary clauses to see if C corresponds to an operation defined in M .

The design scheduler governs the creation of modules and their objects—schemes, constraints, operations, enforcement and hiding clauses, view definitions, and surrogates—in an order compatible with the definition order.

Finally, the dialogue manager controls the communication with the DBA, tests the correctness of a reply, using special internal predicates, and adds the information obtained for the DBA's answers to the data dictionary.

A sample design session, defining the module PRODUCT introduced in Section 2.1, is (the system's messages are shown in boldface):

```

module product
* module type—(type) ?
  answer is primitive
* schemes—((name)((domains))) ?
  answer is (prod (pnum, name))
  answer is enough
* constraints—((name)((definition))) ?
  answer is (one-n ( $\forall p \forall n \forall m$ (prod(p,n) & prod(p,m)  $\Rightarrow$  n = m)))
  answer is enough
* operations—((name)((parameters)) : (body)) ?
  answer is (addprod ((p,n) if  $\neg \exists m$ (prod(p,m)) then insert (p,n) into prod))
  answer is (delprod ((p) delete prod(v1,v2) where v1 = p))
  answer is enough
* ADDPROD enforces ONE-N ? yes
* DELPROD enforces ONE-N ? no
*** module PRODUCT created

```

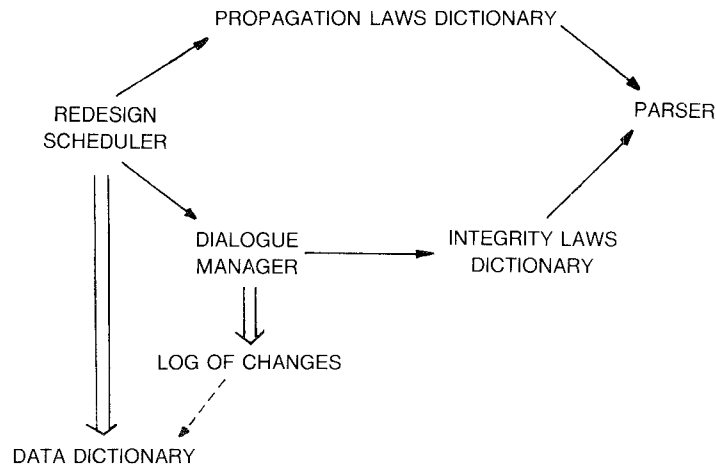


Fig. 11. Major components of the Redesign Tool.

To conclude, we note that, before answering any query generated by the dialogue manager, the DBA can interpose his or her own questions either by using a pull-down menu or by executing one or more PROLOG commands. In any case the system replies to a separate window. One may also use a PLAN-GENERATOR prototype, developed separately [16], to verify or test the sufficiency of preconditions [34] of operations and to check the correctness of the translation of view operations. One may switch to the plan-generator prototype either before using the design prototype or by interrupting a design session. This separate tool is also written in micro-PROLOG.

5. THE REDESIGN TOOL

The Redesign Tool allows the DBA to directly insert, delete, or modify only the relation schemes, constraints, and operations of a module, since the enforcement or hiding statements are a consequence of the definition of the operations and constraints. It has the following major components (see Figure 11): a *parser*, a *data dictionary*, a *log of changes*, an *integrity laws dictionary*, a *propagation laws dictionary*, a *redesign scheduler*, and a *dialogue manager*. Figure 11 displays the components of the redesign tool. Simple arrows indicate that one component activates the other, double arrows denote output, and the dotted arrow expresses the fact that the contents of the log of changes are used to produce the final output, namely the new version of the dictionary.

The parser, the data dictionary, and the integrity laws dictionary are exactly as for the design tool. The dialogue manager is also very similar to that of the design tool.

The log of changes describes the changes already applied to the dictionary.

The propagation laws indicate to the DBA when he must specify an additional change on an object O because some previous change invalidates

the definition of O and when a new object must be inserted in a module because of some previous change. They also automatically synthesize changes in certain cases.

We note that, like the integrity laws, propagation laws invoke the parser. They may also send messages as well as pose questions to the user, which he or she can answer either immediately or after inspecting the current state of the session using a pull-down menu or executing one or more PROLOG commands.

The redesign scheduler governs the execution of the two basic commands, “change,” and “new.”

In the rest of this section, we explain how the redesign scheduler and the dialogue manager work, by outlining the flow of control when the DBA specifies a single change (the tool currently processes one change at a time).

To specify a change, the DBA may enter either:

change (name of object)

to modify or delete a scheme, constraint, or operation, or

new ((class of object)s of (name of module))

to insert one or more schemes, constraints, or operations in the indicated module.

The redesign tool creates an entry in the log of changes, validates the change, and applies it to a copy of the dictionary brought into main storage. Then the tool examines, according to the definition order, the sets of objects belonging either to the module N that contains the object being modified or inserted or to those modules that transitively subsume or extend N .

Let X be either the set of schemes, the set of constraints, or the set of operations of one such module M . For each object $x \in X$, the scheduler checks the existence of previous changes that may affect the correctness of the current definition of x , using the propagation laws. If this is the case, the tool prompts the user to supply an appropriate change (insertion, deletion, modification, or, in some cases, a no-change decision). Here, as in the design phase, the tool validates the new definition of x (or revalidates the current one, if no-change is indicated) by using templates and unique-answer and valid-answer predicates, which again invoke integrity laws.

If X is the set of operations of M , the tool also invokes propagation laws to guarantee that schemes and operations are active, prompting the user to supply new operations if needed. An important case requiring the definition of a new operation occurs when a previously active operation O of M is hidden as a consequence of the insertion or a modification of a constraint I' of a module M' that subsume M . The user must then insert into M' a new operation O' that calls O and enforces I' .

Let X now be the set of hidings or the set of enforcements of M . Propagation laws may in these cases force either the automatic deletion of old objects or the insertion of new objects in X .

Finally, after completing the redesign process, the user may decide to accept the new version of the modular schema or revert to the previous one,

thus abandoning all changes (perhaps because propagation laws forced him to specify unwanted changes). Since the system updates a copy of the dictionary brought into main storage, the above choice amounts to either storing this copy in secondary storage in place of the old version or abandoning it.

As an example of a redesign session, assume that only the module PRODUCT has been created and that the DBA wants to modify the scheme PROD to include an additional domain. The session would go as follows (the system's messages are again shown in boldface):

```
change prod
* scheme PROD
((change)((domains))) ?
  answer is (MOD (pnum,name,weight))
* delete or modify ONE-N in PRODUCT since ONE-N references a scheme PROD
that was modified
* constraint ONE-N
((change)((definition))) ?
  answer is
(MOD (( $\forall p \forall n \forall m \forall w \forall v$ (prod(p,n,w) & prod(p,m,v)  $\Rightarrow$  n = m & w = v)))
* delete or modify ADDPROD in PRODUCT since the definition of ADDPROD
references a scheme PROD that was modified
* is the test in ADDPROD that guarantees that ONE-N is preserved unnecessary or
wrong ? yes
* delete or modify ADDPROD in PRODUCT since ADDPROD contains tests that are
either wrong or unnecessary to guarantee that ONE-N is preserved
* operation ADDPROD
((change)((parameters)): (body))) ?
  answer is
(MOD ((p,n,w) . if  $\neg \exists m \exists v$ (prod(p,m,v)) then insert (p,n,w) into prod))
* delete or modify DELPROD in PRODUCT since the definition of DELPROD
references a scheme PROD that was modified
* should DELPROD contain some test necessary to guarantee that ONE-N is
preserved ? no
* operation DELPROD
((change)((parameters)) : (body))) ?
  answer is
(MOD ((p) : delete prod(v1,v2,v3) where v1 = p))
* does ADDPROD now contain some test to guarantee that ONE-N is preserved ?
yes
*** change and propagation completed
```

Effects on the dictionary are as follows: Entries corresponding to prod, one_n, addprod, and delprod are modified.

6. CONCLUSIONS

We described in this paper a modularization discipline and a software tool to support its application. The discipline contains a detailed set of design requirements that guarantee consistency preservation and the absence of

inactive relation schemes and operations. The software tool not only guides the DBA through the process of creating new modules, but also helps the DBA maintain those already defined.

To construct the design tool, we started with a complete formalization of the discipline, expressed as a set of design requirements, and then gradually mapped the requirements into assertions and triggers. The mapping depended on choosing a reasonable definition order for the objects of a schema. However, note that this strategy can be generalized to a framework for writing assertions and triggers that correspond to a set of integrity constraints of a normal database schema.

PROLOG, especially enhanced with the Apes extension, proved quite appropriate for the development of the software tool since it allowed us to express every component of the tool uniformly as PROLOG clauses, although the orientation differed for each component: The set of design rules was expressed *declaratively*, the schedulers that guided the design/redesign processes were implemented *procedurally* (thanks to the clause ordering discipline of PROLOG), the dialogue managers represented cases of *pattern-matching invocation*, and the final representation of the specification took the form of *factual data*. This experience should carry over to the development of software tools such as ours that basically acquire a specification from an expert software designer, criticize this information based on certain design rules, and map the specification into a lower level language.

Future plans include first expanding the discipline into a complete conceptual modelling language by incorporating concepts such as data abstractions, and then transforming the tool into a full-fledged dictionary system incorporating as much knowledge as possible about the complete language.

The final tool we envision will consist of

- an interpreter for the final conceptual modelling language, that calls
- an expert helper, incorporating knowledge about the language to guide the database administrator, which in turn uses
- a dictionary storing (versions of) several modular database schemas, and finally
- a compiler mapping dictionary entries into DDL/DML commands of the underlying DBMS.

REFERENCES

1. ALBANO, A., CARDELLI, L., OCCHIUTO, M. E., AND ORSINI, R. A modularization mechanism for conceptual modeling. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, 1983), 232-240
2. ALBANO, A., CARDELLI, L., AND ORSINI, R. Galileo: A strongly-typed, interactive conceptual language. *ACM Trans. Database Syst.* 10, 2 (June 1985), 230-260
3. AMBLE, T., BRATBERGSENGEN, K., AND RISNES, O. ASTRAL—A structured and unified approach to data base design and manipulation. In *Data Base Architecture*, G. Bracchi and G. M. Nijssen, Eds., North-Holland, Amsterdam, 1979. 257-274
4. ANSI/X3/SPARC. Study Group on Data Base Management Systems: Interim Report, FDT 7:2, ACM (1975).

5. BORGIDA, A. Features of languages for the development of information systems at the conceptual level. *IEEE Softw.* (Jan. 1985), 63–72.
6. BOUZEGHOUB, M., GARDARIN, G., AND METAIS, E. Database design tools: An expert system approach. In *Proceedings of the 11th International Conference on Very Large Data Bases* (Stockholm, 1985), 436–447.
7. BRODIE, M. On modelling behavioral semantics of databases. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, 1981), 32–43.
8. BURSTALL, R. M., AND GOGUEN, J. A. An informal introduction to specifications using CLEAR. In *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, Eds., Academic Press, New York, 1981, 185–213.
9. CASANOVA, M. A., DE CASTILHO, J. M. V., AND FURTADO, A. L. Properties of conceptual and external database schemas. In *Formal Description of Programming Concepts II*, D. Bjorner, Ed., North Holland, Amsterdam, 1983, 409–430.
10. CLARK, K. L., AND McCABE, F. G. *Micro-PROLOG: Programming in Logic*. Prentice-Hall, Englewood Cliffs, N.J., 1984.
11. DAYAL, U., AND BERNSTEIN, P. A. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.* 7, 3 (1982), 381–416.
12. ESWARAN, K. P. Specification, implementation and interaction of a trigger subsystem in an integrated data base system. IBM Res. Rep. RJ1820, Aug. 1976.
13. ESWARAN, K. P., AND CHAMBERLIN, D. D. Functional specification of a subsystem for data base integrity. In *Proceedings of the 1st International Conference on Very Large Data Bases* (Framingham, Mass. Sept. 1975).
14. FURTADO, A. L., AND CASANOVA, M. A. Updating relational views. In *Query Processing in Database Systems*, W. Kim, D. S. Reiner, and D. S. Batory, Eds., Springer Verlag, New York, 1985, 127–142.
15. FURTADO, A. L., CASANOVA, M. A., AND TUCHERMAN, L. Transforming constraints into logic programs: A case study. In *Proceedings of the TC-2 Working Conference on Knowledge and Data DS-2* (Albufeira, Portugal, Nov. 1986).
16. FURTADO, A. L., AND MOURA, C. M. O. Expert helpers to data-based information systems. In *Proceedings of the First International Workshop on Expert Database Systems* (1984), 298–313.
17. FURTADO, A. L., SEVCIK, K. C., AND SANTOS, C. S. Permitting updates through views of data bases. *Inf. Syst.* 4 (1979), 269–283.
18. GOGUEN, J. A., AND BURSTALL, R. M. Introducing institutions. In *Logics of Programs, LNCS 164*, Springer-Verlag, New York, 1984, 221–256.
19. HAMMER, M., AND BERKOWITZ, B. Dial: A programming language for data intensive applications. In *Proceedings of the 1980 ACM SIGMOD International Conference on the Management of Data* (Santa Monica, Calif., May 1980), ACM, New York, 1980, 75–92.
20. HAMMOND, P., AND SERGOT, M. *Apes: Augmented PROLOG for Expert Systems—Reference Manual*. Logic Based Systems Ltd., 1984.
21. LISKOV, B., AND ZILLES, S. Specification techniques for data abstractions. *IEEE Trans Softw. Eng. SE-1* (1975).
22. MYLOPOULOS, J., BERNSTEIN, P. A., AND WONG, H. K. T. A language facility for designing database-intensive applications. *ACM Trans. Database Syst.* 5, 2 (1980), 185–207.
23. PARNAS, D. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972).
24. ROWE, L. A., AND SCHOENS, K. A. Data abstraction, views and updates in RIGEL. In *Proceedings of the 1979 ACM SIGMOD International Conference on the Management of Data* (Boston, May 1979), ACM, New York, 1979.
25. SCHMIDT, J. W. Some high level language constructs for data of type relation. *ACM Trans. Database Syst.* 2, 3 (Sept. 1977).
26. SHIPMAN, D. W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.* 6, 1 (Mar. 1981), 140–173.
27. SHOPIRO, J. E. Theseus—A programming language for relational databases. *ACM Trans. Database Syst.* 4, 4 (Dec. 1979), 493–517.

28. SMITH, J. M., FOX, S., AND LANCERS, T. *Reference Manual for ADAPLEX* TR CCA-81-02, Computer Corporation of America, May 1981.
29. SMITH, J. M., AND SMITH D. C. P. Database abstractions Aggregation and generalization *ACM Trans. Database Syst.* 2, 2 (1977).
30. TEOREY, T. J., AND FRY, J. P. *Design of Database Structures*, Prentice-Hall, Englewood Cliffs, N J, 1982.
31. TUCHERMAN, L., CASANOVA, M. A., AND FURTADO, A. L. A pragmatic approach to modular database design. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, 1983), 219–231.
32. TUCHERMAN, L., FURTADO, A. L., AND CASANOVA, M. A. A tool for modular database design. In *Proceedings of the 11th International Conference on Very Large Data Bases* (Stockholm, 1985), 436–447.
33. TSUR, S., AND ZANIOLO, C. An implementation of GEM—Supporting a semantic data model on a relational back-end. In *Proceedings of the 1984 ACM SIGMOD International Conference on the Management of Data* (Boston, June 1984) ACM, New York, 1984.
34. VELOSO, P. A. S., AND FURTADO, A. L. Towards simpler and yet complete formal specifications. In *Information Systems: Theoretical and Formal Aspects*, A. Sernadas, J. Bubenko, and A. Olive Eds., North-Holland, Amsterdam, 1985, 175–189.
35. WASSERMAN, A. I. The data management facilities of PLAIN. In *Proceedings of the 1979 ACM SIGMOD International Conference on the Management of Data* (Boston, May 1979). ACM, New York, 1979.
36. WEBER, H. Modularity in data base systems design. In *Proceedings of the Joint IBM/University Newcastle upon Tyne Seminar*, 1979.
37. ZILLES, S. N. Types, algebras and modelling. In *Proceedings of the Workshop on Data Abstractions, Databases and Conceptual Modelling* (Pingree Park, Colo., 1980).
38. ZILLES, S. N., LUCAS, P., AND THATCHER, J. W. A look at algebraic specifications. Res. Rep. RJ3568, IBM Thomas J. Watson Research Center, 1982.

Received December 1985; revised August 1987; accepted August 1988