MODULAR DATABASE DESIGN TOOLS

Luiz Tucherman*
Antonio L. Furtado*1
Marco A. Casanova**

*Latin American Systems Research Institute / IBM Brazil
**Brasilia Scientific Center / IBM Brazil

ABSTRACT

A modularization discipline that helps design complex 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 described. A PROLOG implementation of the tool is completely operational.

Keywords and phrases: logical database design, abstract data types, modular design, module constructors, encapsulation, integrity constraints, consistency preservation, relational model, expert systems, logic programming.

1 On leave of absence from the Pontifical Catholic University of Rio de Janeiro
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 complex database schemas, understood here in the broader sense of a description of both the data structures (static part) and the transactions (dynamic part) of an information system (IS) developed around a database.

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

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 of modules, either because the application evolves, or because his 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 \([CM]\) extended with APES \([HS]\), is fully operational. It offers a very 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.

We close this introduction with brief general remarks about design tools, the design of information systems (ISs) and modularization.

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 [BGM].

The knowledge to incorporate in an expert tool should, above all, reflect a method for designing and redesigning databases. 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 can have 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 ISs demand more flexible data models and better design disciplines, usually provided for by a conceptual modelling language [ABR, ACC, Bo, MB, MW, RS, SPL, Sc, Shi, Sho, TZ, Wa]. 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 [SS]. They cope with the design of the dynamic part by incorporating a programming language that supports the types of the data model adopted and that incorporates high-level control abstractions. Modularization is a concept orthogonal to those just mentioned that imposes a design discipline to cope with size and complexity. Thus, strictly speaking, modularization does not buy additional modelling power.

Modularization, in database conceptual design, is by no means a novel idea [We]. Almost all conceptual design languages treat a module as a set of data structures encapsulated by a set of operations (that is, 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 module constructor the situation goes as follows. Some form of our external constructor is present in most conceptual design languages. In particular, Rigel [RS] and Plain [Wa] include view update translations in their version of the constructor, as we do. But, apart from this constructor, the proposals differ considerably. Weber [We] structures the use of modules in the tradition of software engineering; Astral [ABR] uses a block structure policy to limit the scope of data structure declarations; Galileo [ACO] uses the notion of environment, which is a denotable value, as a modularization mechanism; AdaPlex [SPL] adopts a form of Ada packages for this purpose; Taxis [MBW] 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 recall only that CLEAR [BG, GB] contains several proposals for organizing the definition of theories. The modularization discipline described in this paper was first introduced in [TFC, TFC]. The software tool was partially described in [TFC].

The 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 introduces the design rules for a correct modular database schema and a strategy for change propagation in redesigning the schema. Section 4 describes the architecture of the tool for the creation and redesign of a modular schema. Appendix I summarizes the design rules of the modularization discipline. Appendix II exhibits examples of design and redesign sessions using the prototype.

2. MODULAR DATABASE DESIGN

We describe in this section the basic concepts of our database design method. Section 2.1 defines the concept of module, Section 2.2 introduces the module constructors and Section 2.3 describes a complete example.

2.1 The Concept of a Module

Let \( L \) be a first-order language containing all ordinary symbols (such as equality) to be used in database design.

A relation scheme is a statement of the form \( R[A_1,\ldots,A_n] \), where \( R \) is the relation name and \( A_1,\ldots,A_n \) are the attributes of the scheme. We will treat \( R \) as an n-ary predicate symbol and \( A_1,\ldots,A_n \) as unary predicate symbols and we will assume that none of these symbols belongs to \( L \). Let \( R \) be a set of relation schemes such that no two schemes in \( R \) have the same relation name. A first-order language \( L' \) is induced by \( R \) iff \( L' \) is obtained by adding the symbols \( R, A_1,\ldots,A_n \) to \( L \), for each relation scheme \( R[A_1,\ldots,A_n] \) in \( R \).

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 of \( L' \).

An operation is a procedure definition, that is, a statement of the form \( f(x_1,\ldots,x_n): s \).

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 first-order language \( LM \) induced by \( RS \). \( CN \) must contain, for each relation scheme \( R[A_1,\ldots,A_n] \), a relation scheme axiom of the form

\[ \forall x_1\ldots\forall x_n (R(x_1,\ldots,x_n) \Rightarrow A_1(x_1) \& \ldots \& A_n(x_n)), \]

indicating that the interpretation of \( R \) must be a subset of the cartesian product of the interpretations of \( A_1,\ldots,A_n \).

3. \( OP \) is a set of operations over \( LM \).

The objects of \( M \) are the relation schemes, integrity constraints and operations of \( M \).

Section 2.3 contains examples of module definitions.
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.

Let \( L \) be a fixed first-order language containing all ordinary symbols, and let \( M_i = (RS_i, CN_i, OP_i), i=1, \ldots, n \), be modules.

Consider the subsumption constructor first. Intuitively, if the DBA defines \( M \) by subsumption over modules \( M_1, \ldots, M_n \), 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 \( M_1, \ldots, M_n \). \( M \) also inherits all operations of \( M_1, \ldots, M_n \), 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 \( M_1, \ldots, M_n \) then become inaccessible to users and can no longer participate in the definition of new modules.

The following statement defines a new module \( M \) by subsumption over \( M_1, \ldots, M_n \):

(2) module \( M \) subsumes \( M_1, \ldots, M_n \) with
    schemes \( RS_0 \);
    constraints \( CN_0 \);
    operations \( CP_0 \);
    enforcements \( EN \);
    hidings \( HI \);
endmodule

where:

1. \( RS_0 \) is a set of relation schemes such that no relation name in \( RS_0 \) occurs in \( M_1, \ldots, M_n \), and no two schemes in \( RS_0 \) have the same relation name;

2. \( CN_0 \) is a set of (named) integrity constraints over \( RS_0, RS_1, \ldots, RS_n \);
3. OP0 is a set of operations over RS0, RS1, ..., RSn;

4. BN 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.

Each object of Mi is said to be imported 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 [DE, FC, FS], 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. The well-formed formula '\( \forall x1, ..., xk \cdot R(x1, ..., xk) \leftrightarrow Q \)' is called a view definition axiom.
3. SR contains, for each operation \( f(y_1, \ldots, y_m) : r \) in \( OP_0 \), a surrogate, which is an operation of the form \( f(y_1, \ldots, y_m) : s \) over \( RS_1, \ldots, RS_n \);

The statement in (3) then defines a new module \( M = (RS_0, CN_0, OP_0) \) and couples \( M \) to \( M_1, \ldots, M_n \) through the pair \( (VW, SR) \). A view definition \( \text{mapping } R[A_1, \ldots, A_k] : \varphi \text{ in } VW \) indicates that \( Q \) defines \( R \) in terms of the relation schemes of \( M_1, \ldots, M_n \). Hence, a query over \( R \) is translated into a query over the relation schemes of \( M_1, \ldots, M_n \) with the help of \( Q \). Likewise, a surrogate \( f(y_1, \ldots, y_m) : s \) in \( SR \) describes an implementation of \( f(y_1, \ldots, y_m) : r \) in terms of the operations of \( M_1, \ldots, M_n \). Thus, a call to procedure \( f \) generates an execution of \( s \), not \( r \).

Finally, a modular database design is described by a statement of the form:

```
  schema D with
      M_1; \ldots; M_s
  endschema
```

where \( M_1, \ldots, M_s \) are conceptual or external modules such that no two modules have the same name.

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

In terms of the ANSI/\( \text{X}3/\text{SPARC} \) architecture [ANSI], the conceptual schema corresponds to a forest of primitive and subsumption modules. The roots of the forest are the active modules (underlined in Figure 2.1). The external schemata correspond to external modules, which form a partial order stemming from the active conceptual schema modules.
2.3 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[#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) & NAME(n))
    then insert (p,n) into PROD;
  DELPROD(p):
    delete PROD(x,y) where x=p;
  enforcements
  ADDPROD enforces ONE_N;
endmodule

Thus, PRODUCT is the triple (AS,CN,OP), where:

- RS consists of the relation scheme PROD[#NAME]
• CN contains, in addition to the well-formed formula listed after the **constraint** clause, the following relation scheme axiom:

\[ \forall x \forall y ( \text{PROD} (p, n) \rightarrow p \# (p) \& \text{NAME} (n)) \]

• OP consists of the operations listed after the **operations** clause.

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:

```plaintext
module WAREHOUSE
schemes WAREHOUSE[W#, LOC]
constraints
    ONE_C:
        \[ \forall w \forall c \forall c' (\text{WAREHOUSE} (w, c) \& \text{WAREHOUSE} (w, c')) \rightarrow c = c' \]
operations
    OPEN(w,c):
        if \( \neg \exists c' \text{WAREHOUSE} (w, c') \& \# (w) \& \text{LOC} (c) \)
        then insert (w, c) into WAREHOUSE;
    CLOSE(w):
        delete WAREHOUSE (x, y) where x = w;
    enforcements
        OPEN enforces ONE_C;
endmodule
```

The modular database schema now has two 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:

```plaintext
module SHIPMENT
subsumes PRODUCT, WAREHOUSE with
schemes SHIP[P#, W#, QTY]
constraints
    ONE_C:
        \[ \forall p \forall w \forall q \forall q' (\text{SHIP} (p, w, q) \& \text{SHIP} (p, w, q')) \rightarrow q = q' \]
    INC_P: \[ \forall p \neg \exists j \text{PROD} (p, j) \]
    INC_W: \[ \forall w \neg \exists j \text{WAREHOUSE} (w, j) \]
operations
    ADDSHIP(p,w,q):
        if \( \exists j \text{PROD} (p, j) \& \exists c \text{WAREHOUSE} (w, c) \& \neg \exists q' \text{SHIP} (p, w, q') \& \text{QTY} (q) \)
        then insert (p, w, q) into SHIP;
    CANSHIP(p,w):
        delete SHIP(x, y, z) where \( x = p \& y = w \);
    CLOSE(w):
        if \( \exists j \exists q \text{SHIP} (p, w, q) \) then CLOSE (w);
```
DELPRTD1(p):
  if ¬\exists w \forall q SHIP(p,w,q) then DELPROD(p);
endmodule

The modular database schema now has three modules, SHIPMENT, WAREHOUSE and PRODUCT. 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 DELPROD, 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) : \exists q SHIP(p,w,q)
surrogates
  DEL(p,w) : CANSHIP(p,w)
endmodule

The final database schema therefore has four modules, SHIPMENT, DELIVERY, PRODUCT and WAREHOUSE. Users have access to three base relation schemes (using traditional terminology), PROD[p#,NAME], WAREH[w#LOC], and SHIP[p#,w#,QTY], and one view, DELVRY[p#,w#]. The active operations are ADDSHIP, CANSHIP, ADDPROD, DELPROD, 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. DESIGN REQUIREMENTS, DEFINITION ORDER AND PROPAGATION RULES

We discuss in Section 3.1 the design requirements for a correct modular database schema. Then, we complete the specification of our modular design method, by presenting, in Section 3.2 a definition order to create or change the specifications of objects of a modular schema. Finally, we explain in Section 3.3 the strategy we adopted for the propagation rules which determine how changes on a group of objects propagate to others.
3.1 The Requirements

To present the requirements, we first introduce a few concepts. A scheme is accessible to a module M if it was defined either in M or in a module below M in the subsumption hierarchy. Whenever a scheme is accessible, so are the domains over which the scheme is defined. An operation becomes hidden at a subsumption module if it may violate some constraint of that module. An operation is accessible to a module M if it was defined either in M or in some module M' below M in the subsumption hierarchy, with the proviso that it did not become hidden at any module between M' and M. Accordingly, we may say, except for operations that are explicitly hidden, subsumption leads to the automatic import of schemes and operations in the upward direction.

By contrast, the extension partial order acts as a screen. If M is an extension module, it can only export to the modules above it the schemes (views) and operations defined in M itself.

The set of requirements defines what is a consistent design, ensuring that:

- the operations of a newly defined module M preserve consistency with respect to the constraints of M;

- the operations of M preserve consistency with respect to the constraints of modules previously defined;

- the operations of previously defined modules, without any change, preserve consistency with respect to the constraints of M, except if it is defined by subsumption over M', when operations of M' may be hidden in M;

- schemes and operations are active, in the sense that for every scheme there is at least one operation performing insertions into it, and that every operation is either directly useable or is called by some useable operation.

Requirements (the complete list is presented in Appendix I) are classified either as syntactical or semantic. The syntactical requirements allow integrity constraints and operations defined in a module M to reference any scheme accessible to M. However, only schemes defined in M can be directly updated by the operations of M; to update imported schemes defined, say, at a module M', an operation of M must call as a sub-routine the appropriate operation defined in M'. Domains, of course, can never be updated. If M is an external module, then its views must be defined over schemes accessible to the modules that M extends; in turn, the constraints and operations of M will only reference such views, whereas the surrogates of the operations will reference imported schemes and call imported operations.

Syntactical requirements related to ensuring that schemes and operations be active impose that, for every scheme S (that is not a view) defined in M, there must be an operation also defined in M that performs insertions into S. Also, for every operation O hidden because of a constraint of M, there must be an operation defined in M that calls O. Thus, such requirements guarantee that there are no "useless" objects in the schema.
For \textit{enforces} relationships of a module $M$, it is required that both the operation and the constraint involved be defined in $M$. For \textit{may-violate} relationships, the operation must be accessible to (but not defined in) $M$ and the constraint must be defined in $M$.

At the module level, the syntactical requirements explicitly define subsumption and extension as a hierarchy and a partial order, respectively. They establish that any module $M$ can only subsume active primitive or subsumption modules; also such modules cannot have been extended by some module $M'$, since otherwise it might be possible in $M'$ to achieve updates (by calling accessible operations) in violation of constraints introduced in $M$. External modules, on the other hand, are simply restricted to extend active conceptual modules or other external modules that have been previously defined (to prevent circularity).

The semantic requirements impose that constraints be invariant with respect to operations (i.e., any necessary tests have been effectively included in the critical operations), and all \textit{enforces} and \textit{may-violate} statements have been provided. They also impose that all constraints in an extended module $M$ be logical consequences of constraints of modules below $M$, and that the surrogate of each operation $O$ in an external module correctly corresponds to $O$ (in the precise sense explained in [FC]).

3.2 A Definition Order over the Sets of Objects of Modules

We introduce in this section a \textit{definition order}, denoted by $<$, on the sets of objects of the modules of the schema. The definition order completes the specification of the modular design method since it indicates in which order the DBA must define or modify the objects of a modular schema.

It goes as follows:

case 1: \textbf{object definition order}

Let $M$ be a module and, when applicable depending on the type of $M$, let $R_S, C_N, O_F, H_I, E_N, V_W$ and $S_R$ be constants denoting the sets of schemes, constraints, operations, hiding statements, enforcement statements, view definitions and surrogates, respectively, defined in $M$. Then,

\[
\begin{align*}
R_S &< V_W \\
R_S &< C_N, V_W < C_N \\
C_N &< H_I \\
R_S &< O_F, C_N < O_F, H_I < O_F \\
C_N &< E_N, H_I < E_N, O_F < E_N \\
V_W &< S_R, O_F < S_R
\end{align*}
\]

case 2: \textbf{module definition order}

Let $M'$ be a module defined over $M$ and $X, X'$ be sets of objects of $M$ and $M'$, respectively. Then, $X < X'$.

The intuitive interpretation of $X < X'$ is that, to specify the objects in $X'$, the DBA has to know the specification of all objects in $X$. However, this does not mean that the specification of an object in $X'$ must depend on all objects in $X$. 

The definition order extends to the objects themselves in the natural way: for any two objects \( x \) and \( x' \) such that \( x \in X \) and \( x' \in X' \), \( x \prec x' \) iff \( X \prec X' \).

3.3 The Propagation Rules

The set of propagation rules is used to optimize redesign by limiting attention to those components that may be affected by some change, and therefore might need to be changed in their turn.

The propagation rules are classified into manual (dealing with schemes, constraints and operations) and automatic (for may-violate and enforces relationships). This classification has to do with the fact that, in our method, relationships cannot be changed by themselves, but rather as a consequence of changes to other components. Ideally, a tool implementing the method should be able to determine when a relationship starts to hold or ceases to hold, but this again is a semantic rather than syntactical problem which makes it necessary to resort to the user's judgement.

We begin with the manual rules. Whenever a scheme \( S \) is modified or deleted, views, constraints, operations and surrogates defined over \( S \) may need to be modified or deleted. The insertion of new schemes makes it necessary to introduce operations to perform insertions on them, since otherwise they would become inactive.

The insertion, modification or deletion of constraints may lead to the deletion or modification of an operation \( O \), since \( O \) may become critical to the enforcement of new or modified constraints, or simply because \( O \) may cease to be critical. In the first case, new tests may have to be added to \( O \), or the current tests may have to be modified. In the second case, tests may be dropped because they became unnecessary. Changes on constraints may also lead to the modification or deletion of constraints of external modules.

In a subsumption module \( M \), may-violate relationships may have been signaled (by the user) to exist between operations of modules below \( M \) and new or modified constraints of \( M \). The insertion of these relationships will hide the operations (if they were not already hidden by other constraints), thereby making them inactive unless new operations to call them are inserted. Normally, the new operations should perform tests prior to calling the hidden operations, so as to guarantee the enforcement of the constraints. The need for a new operation to call a hidden operation \( O \) also arises if \( O \) was called by an operation \( O' \) that was deleted or modified and, as a consequence, does not call \( O \) anymore. On the other hand, if an operation \( O \) no longer may violate any constraint, therefore ceasing to be hidden, and an operation \( O' \) existed with the sole purpose of safely calling \( O \), then \( O' \) becomes a candidate for deletion.

Finally, the modification or deletion of operations affects the operations and surrogates that call them, and the deletion of an operation that is the only one to perform insertions into some scheme makes it necessary to create a new operation to assume this task.
The automatic propagation rules establish that an enforces or may-violate relationship holding between an operation O and a constraint C should be deleted if O or C are deleted; if O or C are simply modified, the user is asked whether or not the relationship still holds. If there was no such relationship between O and C and one or both are modified, or still in the case where they have been just created, the user is asked whether the relationship now holds.

4. THE DESIGN TOOL

To support specifications according to the modular design method described in Section 2 and 3, we have built and are further developing a prototype expert system, written in micro-PROLOG [CM] extended with AFES [BS]. To help develop the prototype, we divided the specification of a database application into the design and the redesign phases. This separation should not be viewed as a characteristic of the modularization method, however.

4.1 The Design Phase

For this phase the prototype has:

a) a program to schedule, according to the definition order, the creation of modules and their components;

b) a query-the-user facility (provided by AFES);

c) a parser, for wffs of the first-order predicate calculus and operations of the formal programming language proposed in [CE];

d) the requirements themselves, expressed in a declarative form;

e) a dictionary, produced as output from the dialogue with the user.

Figure 4.1 illustrates the design phase tool, showing its parts and how they are organized. Simple arrows denote that one component activates the other and the double arrow denotes output.

![Figure 4.1: The Design Phase](image)

The requirements are applied firstly to restrict the answers given by the user, and then to prompt him again for missing information. By obeying the requirements, one strives to achieve specifications that are correct by construction. The context-free syntax of logical formulas and operations is
immediately checked by the parser. The context-sensitive syntax (imposing for instance that operations introduced in a module \( M \) can only directly update schemes created in \( M \)) is part of the requirements and is checked over the parse tree produced by the parser. However, the semantic requirements (e.g., the pre-conditions of operations are sufficient to enforce the integrity constraints) are not checked by the prototype, which merely warns the user that this is his responsibility and then trusts his answer. To verify/test the sufficiency of pre-conditions [VF], and to check the correctness of the translation of view operations, a PLAN-GENERATOR prototype developed separately [PM] can be utilized, either before using the design prototype, or by interrupting a design session. This separate tool is also written in micro-PROLOG.

We now explain in more detail how the design phase works. The appendix contains examples.

i) The user enters:

\[
\text{module < name of module being created >}
\]

This invokes the "module" predicate, which activates the query-the-user feature to ask the user the type of the module (primitive, subsumption, external), enters this information in the dictionary (the "tab" predicate) and then sets-up, as sub-goals, the creation of the other classes of components, according to the definition order.

The "module" predicate, together with the predicates it calls to achieve the sub-goals, constitute what we termed the scheduler.

ii) Whenever the user is queried, via the APES "confirm" command, the following features are activated:

- template predicates - user-friendly reformulation of the query
- unique-answer predicates - whenever the query admits one answer only (this is the case of the module type)
- valid-answer predicates - to restrict the answer(s) supplied by the user; predicates formulating the requirements related to the consistency of the design are called by these predicates

If a user's reply is not valid, it is rejected by APES and the user is prompted to reply again. If more than one (valid) answer can be supplied (e.g., a module may have several schemes, constraints and operations) the user must enter the word "enough" after the last answer. Before answering any query posed by the tool, the user can execute one or more PROLOG commands, being able, in particular, to inspect the dictionary, to test a requirement, etc.; windows and a pull-down menu are also provided.

iii) The requirements that impose that schemes and operations be active are called directly from the scheduling program, after the user has supplied all operations for the module being created. If one or more schemes and one or more operations are not active, the user is prompted to supply additional
operations, this step being repeated until the requirements are satisfied.

iv) To verify the requirements on the expressions of view definitions, constraints, operations and operation surrogates the predicates that constitute the parser are called. They yield the parse tree of the given expression, which is traversed to find sub-expressions of the desired class. For instance, we can isolate an insertion statement within the parse tree of an operation and, inside it, the occurrence of a predicate name; to check one of the requirements, the dictionary thus far constructed is inspected to see if the predicate name corresponds to a scheme accessible to the module.

v) The entries (sentences) of the "tab" predicate, which constitute the data dictionary produced as output of the design phase, are created from the dialogue with the user, whence APES records them with the special format - "you told me that <tab entry >". After creating all intended modules, the user may type - "save dialogue < file name >" - whereby "tab" is stored on a diskette. To convert "tab" into an ordinary predicate he may enter - "define tab".

4.2 The Redesign Phase

For this phase the prototype has:

a) another appropriate scheduling program;

b) the query-the-user facility;

c) the parser (same as for the design phase);

d) the requirements (same as for the design phase);

e) the propagation rules, expressed in the same declarative style as the requirements;

f) a change-log, produced from the dialogue with the user as an intermediate output, that marks which components have been changed and the nature of the change;

g) the dictionary resulting from the design phase, which is updated to reflect the changes to the design.

Figure 4.2 displays the redesign phase tool. Simple arrows denote activation, double arrows denote output and the dotted arrow expresses the fact that the contents of the change-log are used to produce the final output, namely the new version of the dictionary.
The scheduling program follows the same definition order. The user indicates the deletion or modification of some scheme, constraint or operation. The insertion of various components of one of these three classes into the same module is also supported. Besides processing the change, the prototype leads the user to examine all other components that may be affected by the change, a propagation process that continues from the affected components that are in their turn changed.

Either the new formulation of an affected component or (in case the user decides not to change it) its current formulation is checked by applying the requirements, exactly as in the design phase. Again, the user is responsible for the semantic requirements, and here too he can resort to the plan-generation tool.

During propagation, questions are asked from and messages are sent to the user. Even the automatic propagation rules do not exclude the need, in some cases, to query the user before changing a relationship; if a constraint is modified the user is asked whether the relationships still hold after the modification - recall that this has also to do with semantic requirements.

As a consequence of the definition order, when each affected component is reached, all possible propagation lines leading to it will have been exploited. Thus, if a scheme and a constraint were modified and both modifications affect an operation, the consequences of both will be available when the moment to consider the operation arrives.

In more detail, the redesign phase works as follows (examples are provided in the appendix):
i) The user enters either:

change < name of component >

if we want to modify or delete a scheme, constraint or operation, or:

new (< class of component > of < name of module >)

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

The invoked predicate - "change" or "new" - create the appropriate entries in the log, with the information obtained by querying the user, particularly the new expressions that he is asked to supply. Then, in both cases, the "propag" predicate is invoked and sets-up as sub-goals to examine, following the definition order, the components of each class, both in the same module and in those above it along the subsumption and extension paths. The predicate "change", "new" and "propag", together with the predicates that the latter calls for the propagation in each class, constitute the scheduler for the redesign phase.

ii) As the scheduler examines each component, it checks the existence of propagation paths leading to it from some component in the change-log. To determine these paths, the predicates formulating the propagation rules (manual or automatic) are invoked. Queries and/or messages are exchanged between the prototype and the user for every propagation path determined. As in the design phase, the user can enter PROLOG commands before answering queries posed by the tool, being able, for example, to inspect the change-log and the dictionary, to test requirements and propagation rules, etc.

iii) If there is at least one propagation path leading to a scheme, constraint or operation the user is queried about how the component should be changed (deletion, modification, insertion or, in some cases, a no-change decision). Here, as in the design phase, templates, unique-answer and valid-answer predicates are again invoked from the valid-answer predicates in order to validate the change (or no-change) indicated by the user.

iv) The propagation rules for keeping schemes and operations active are exploited after the existing operations are examined, and the user is prompted to supply new operations if needed. An important case where a new operation is needed occurs when a previously active operation O of a module M is hidden by a new or modified constraint of a module M' subsuming M; it is then necessary to insert in M' an operation to call O.

v) The parser is not only invoked by the requirements (as mentioned when the design phase was discussed) but also from the propagation rules.

vi) The redesign phase works on the data dictionary. It yields, as an intermediate result, the change-log containing the changes validated with respect to the requirements. Each change registered in the change-log is immediately effected in the dictionary. Modified entries are kept in the same position as before and insertions are made after the last entry of the same
class and module, recalling that the position of entries is vital to keep the dictionary coherent with the definition order.

vii) More precisely, what is updated is a copy of the dictionary brought into main storage, so that if the user is not satisfied with the outcome (perhaps because of changes that he may have been forced to make in the course of propagation) it suffices to abandon this copy, thereby keeping the version residing in main storage.

Both phases of our prototype, as well as the plan-generation prototype (which we are presently trying to integrate within the same research project), run on IBM personal computers. Future plans include further study of semantic requirements and the enhancement of the prototype, both to add new features, some of which have already been investigated (such as redesign at the module level [TFC]), and to make it more useable and efficient.

5. 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 him maintain those already defined.

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
- a compiler mapping dictionary entries into DDL/DML commands of the underlying DBMS.
REFERENCES


ACM SIGMOD Int. Conf. on the Management of Data, Santa Monica, CA (May 1980), 75-92.


APPENDIX I

List of Requirements

PRIMITIVE MODULES

Let M be a primitive module and RS, CN, OP, EN be the schemes, constraints, operations and enforcements statements defined in M.

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].

Requirement 2: for each scheme SN defined in a module M there must be an operation that performs insertion into SN.

This requirement guarantees that all schemes of a primitive module will become active.

MODULES DEFINED BY SUBSUMPTION

Let M be a module defined by subsumption over modules Mi=(RSi,CNi,OPi), i=1,...,n. Let RS0, CNO, OP0, HI be the new relation schemes, integrity constraints, operations, and hidden operations, respectively, defined in M. Let CN be the union of CNO,...,CNo 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 3: each operation in OP preserves consistency with respect to the integrity constraints in CNO.

Requirement 4: each operation in OP can only modify the values of relation schemes in M1,...,Mm through calls to the operations defined in M1,...,Mm.

Requirements 3 and 4 suffice to guarantee that each operation in OP preserves consistency with respect to CN.

Requirement 5: for each scheme SN defined in a module M there must be an operation that performs insertion into SN.

This requirement guarantees that all schemes of a subsumption module will become active.

Requirement 6: for each operation O in HI, there must be an operation O' in OP0 such that O' calls O.

This requirement guarantees that operations that are hidden in Mk do not become inactive.
**Requirement 7:** D must not contain a module defined by extension using \( M_i \), for some \( i \) in \([1, n]\).

Requirement 7 forbids the DBA to define a new module \( M \) by subsuming a module \( M_i \) if there is a third module \( M' \) that extends \( M_i \). This requirement is necessary since it avoids the undesirable situation where \( M \) subsumes \( M_i \) and yet \( M' \) offers direct paths to the objects and operations of \( M_i \). In fact, if Requirement 7 is violated, we cannot assure that calls to operations of \( M' \) will not violate constraints of \( M \).

**Requirement 8:** \( M_1, \ldots, M_n \) must be conceptual modules of \( D \) (that is, active modules of \( D \) not defined by extension).

Requirement 8 does not permit the subsumption of external modules, again to guarantee that all new operations of \( M_i \) and those of modules defined by subsuming \( M_i \), preserve consistency.

**MODULES DEFINED BY EXTENSION**

Let \( M \) be a module defined by extension over modules \( M_1 = (S_1, C_1, O_1) \), \( i = 1, \ldots, n \). Let \( R_0, C_0, O_0, V_0 \) and \( S_R \) be the new relation schemes, integrity constraints, operations, view definitions and surrogates, respectively, defined in \( M \).

**Requirement 9:** if \( f(y_1, \ldots, y_m) : s \) is the surrogate of \( f(y_1, \ldots, y_m) : r \) defined in \( S_R \) then \( s \) is a faithful translation of \( r \) [FC].

Requirement 9 guarantees that \( s \) correctly implements \( r \) in the sense that \( r \) and \( s \) have the same effect as far as the views are concerned.

**Requirement 10:** if \( f(y_1, \ldots, y_m) : s \) is a surrogate defined in \( S_R \), then \( s \) can only modify the values of relation schemes in \( M_1, \ldots, M_n \) through calls to the operations defined in \( M_1, \ldots, M_n \).

Requirement 10 guarantees that each surrogate \( s \) preserves consistency with respect to \( C_1 \) since \( s \) updates the schemes of \( M_i \) through calls to operations of \( M_i \), for each \( i = 1, \ldots, n \).

**Requirement 11:** for each integrity constraint \( I \) in \( C_1 \), \( I' \) must be a logical consequence of the integrity constraints of \( M_1, \ldots, M_n \), where \( I' \) is obtained from \( I \) by replacing each atomic formula of the form \( R(t_1, \ldots, t_k) \) by \( Q[t_1/x_1, \ldots, t_k/x_k] \), where \( R[A_1, \ldots, A_k] \); \( Q \) is the view definition of \( R \) described in \( V_0 \), and the list of free variables of \( Q \) is \( x_1, \ldots, x_k \).

Requirement 11 guarantees that the integrity constraints of \( M \) follow from those of \( M_1, \ldots, M_n \) 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.

**Requirement 12:** \( M_1, \ldots, M_n \) must be active modules of \( D \).

Requirement 12 avoids defining view operations using inactive operations, which may violate consistency.
APPENDIX II

Design and Redesign Examples

A. Design Examples

In this section we show how to use the design tcl to create the modules defined in Section 2.3. After defining the module PRODUCT, we exhibit the state of the dictionary to illustrate its structure. In the creation of the module SHIPMENT, we forced an error by failing to include an operation to call DELPROD, although this operation is hidden in SHIPMENT since it may violate INC-F. The tool then asks that some operation that calls DELPROD be provided.

Example 1

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) : <bdy>)) ?
  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

State of the Dictionary:

tab {primitive (product) ()
tab (scheme (prod product) ((pnum, name)))
tab (constraint (one-n product)
  (\( \forall p \forall n \forall m (prod(p,n) \& prod(p,m) \Rightarrow n = m) \))
tab (operation (addprod product)
  (((p,n) : if \( \neg \exists m (prod(p,m)) \)
    then insert (p,n) into prod))
tab (operation (delprod product)
  ((p) : delete prod(v1,v2) where v1 = p ))
tab (enforces (addprod one-n product) ()


Example 2

module shipment

* module type --- <type> ?
  Answer is subsumption

* subsumes --- <module> ?
  Answer is product
  Answer is warehouse
  Answer is enough

* schemes --- (<name> (<domains>)) ?
  Answer is (ship (pnum, wnum, qty))
  Answer is enough

* constraints --- (<name> (<definition>)) ?
  Answer is (one-g (∀p∀w∀r(∀q(∀s(∀t(∀p(∀w(∀r(∀q(∀s(∀t(ship(p,w,q) & ship(p,w,r) ⇒ q = r)))))))))
  Answer is (inc-g (∀p∀w∀q(ship(p,w,q)) ⇒ ¬∃n(prod(p,n))))))
  Answer is (inc-w (∀w∀p∀q(ship(p,w,q)) ⇒ ¬∃c(warehouse(w,c))))
  Answer is enough

* addprod may-violate one-q ? no
* addprod may-violate inc-p ? no
* addprod may-violate inc-w ? no
* delprod may-violate one-q ? no
* delprod may-violate inc-p ? yes
* delprod may-violate inc-w ? no
* open may-violate one-q ? no
* open may-violate inc-p ? no
* open may-violate inc-w ? no
* close may-violate one-q ? no
* close may-violate inc-p ? no
* close may-violate inc-w ? yes

* operations --- (<name> ((<parameters>) : <body>)) ?
  Answer is (addship ((p,w,q) :
      if ∃n(prod(p,n)) &
      ¬∃c(warehouse(w,c)) &
      ¬∃r(ship(p,w,r))
      then insert (p,w,g) into ship))
  Answer is (canship ((p,w) :
      delete ship(v1,v2,v3)
      where (v1 = p & v2 = w)))
  Answer is (close1 ((w) :
      if ¬∃q(ship(p,w,q))
      then close(w)))
  Answer is enough

* there is no operation calling any of the hidden operations (delprod)

/*
The tool displays again the "operations" template, lists below the three operations already defined, and prompts the user once more with "Answer is "
*/
  Answer is (delprod ((p) :
    if ¬∃w∀q(ship(p,w,q))
    then delprod(p)))
Answer is enough

* addship enforces cne-g ? yes
  * addship enforces inc-f ? yes
  * addship enforces inc-w ? yes
* canship enforces cne-g ? no
  * canship enforces inc-f ? no
* close1 enforces cne-g ? no
  * close1 enforces inc-f ? no
  * close1 enforces inc-w ? yes
* delprod1 enforces cne-g ? no
  * delprod1 enforces inc-f ? yes
  * delprod1 enforces inc-w ? no

*** Module shipment created

Effects on the dictionary: entries describing the module are added.

Example 3

module delivery

* module type --- <type> ?
  Answer is external

* extends --- <module> ?
  Answer is shipment
  Answer is enough

* views --- (<name> (<domains>)
  {(<parameters>):<definition>}) ?
  Answer is (delvry (pnum,wnum))
  {([P,W] : jg(ship(P,W,g)))}
  Answer is enough

* constraints --- (<name> (<definition>)) ?
  Answer is enough

* operations --- (<name> {(<parameters>) : <bcdy>}
  {(<parameters>) : <surrogate>}) ?
  Answer is (del {([F,W] : delete delvry(v1,v2)
    where (v1 = p & v2 = w))
    {([F,W] : canship(P,W)})
  Answer is enough

*** Module delivery created

Effects on the dictionary: entries describing the module are added.

E. Redesign Examples

Examples 4 and 5 assume that only the module PRODUCT has been created, thus illustrating how propagation occurs for a single module. In Example 4, the scheme PROD is modified to include an additional domain whereas, in Example 5, two new schemes are inserted into PRODUCT.
Examples 6 assume that all four modules have been created, and thus illustrate how changing an object in a module may affect objects of modules defined above it. In this example, operation DELPROD is deleted, a change that propagates to the operation DELPROD1 of the subsumption module SHIPMENTS, since DELPROD1 calls DELPROD.

Example 4

\begin{verbatim}
change prod

* scheme prod
  (#change> (#domain>) ) ?
  Answer is (MOD (pnum, name, weight))

* delete or modify one-n in product since one-n references
  a scheme prod that was modified

* 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 (if \( \forall p \forall n \forall w \forall v \left( \text{prod}(p, n, w) \& \text{prod}(p, m, v) \Rightarrow n = m \& w = v \right) \) )

* delete or modify addprod in product since the definition
  of addprod references a scheme prod that was modified

* 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 \( \neg \exists m \forall v \left( \text{prod}(p, m, v) \right) \)
  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

* does delprod now contain some test to guarantee that one-n
  is preserved? no
\end{verbatim}
*** change and propagation completed

Effects on the dictionary: entries corresponding to prod, one-n, addprod and delprod are modified.

**Example 5**

new(schemes of product)

* new schemes of module product
  (<name> (<domains>)) ?
  Answer is (manual (pnum,title))
  Answer is (part (pnum,name))
  Answer is enough

* insert a new operation in product that performs insertions into manual

* insert a new operation in product that performs insertions into part

* new operation of module product
  (<name> ((<parameters>) : <body>)) ?
  Answer is (register (p,r) :
    if p > m9999
      then insert (p,r) into part
      else insert (p,r) into manual)
  Answer is enough

* does register now contain some test to guarantee that one-n is preserved? no

*** insertion and propagation completed

Effects on the dictionary: new entries are added for schemes manual and part and for operation register.

**Example 6**

change delprod

* operation delprod
  (<change> ((<parameters>) : <body>)) ?
  Answer is (DEL {})

* delete or modify delprod in shipment since delprod calls an operation delprod that was deleted

* operation delprod1
  (<change> ((<parameters>) : <body>)) ?
  Answer is (DEL {})

*** change and propagation completed

Effects on the dictionary: the entries corresponding to delprod and delprod1 are deleted as well as the entries
tab may-violate (delprod inc-p shipment {})
tab (enforces (delprod1 inc-p shipment {})
whose deletion is automatic.