A PRAGMATIC APPROACH TO STRUCTURED DATABASE DESIGN

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

* Latin American Systems Research Institute - IBM do Brasil
** Pontifícia Universidade Católica do R.J.
*** Centro Científico de Brasília - IBM do Brasil

ABSTRACT

A database design methodology, based on the concept of module, is proposed as a way of managing the complexity of database descriptions and, at the same time, enforcing integrity constraints. The design of databases is carried out in two levels of abstraction, the specification level, which is independent of any database management system, and the representation level, that refines the first one into an actual implementation of the database.

At the specification level, the definition of a module consists of a high-level description of the structures and operations of the module, as well as the integrity constraints. Two module constructors, extension and subsumption, are used to define new modules from old ones. Extension is similar to the usual view mechanism. Subsumption is a new module constructor that permits adding new structures, operations and constraints to those of old modules, and redefining old operations, which may be required to maintain integrity.

The representation level description of a database is carried out using the SQL/DS system, which indicates that the modular database design proposed can be used in conjunction with present-day systems.

Finally, the concept of module graph is introduced to capture the modular structure of the database.

1. Introduction

Database design has been greatly influenced by the three-level architecture proposed in [ANSI] that suggests dividing the description of a database into the internal schema, the conceptual schema and the external schemas. The internal schema describes the physical organization of the database; the conceptual schema defines the logical organization of the complete database; and the external schemas describe logical subsets of the database relevant to different classes of users. Consequently, database design techniques can be roughly classified as to whether they address physical or logical database design [TF].

Logical database design may be carried out by stepwise refinement starting with the early stages of requirements analysis and culminating in a conceptual schema, based on some adequate data model. Orthogonal to refinements that cross levels of abstraction (and precision), database design methods must also provide for the fact that databases tend to be large, complex objects. One such method, view integration, tries to beat complexity by synthesizing the conceptual schema by gradually combining schemas that represent the knowledge (or requirements) of the various groups of users [TF, CV, NG, WM, YMH].

We explore in this paper an alternative strategy, based on the concept of module [Pa,LZ,ZLT], as a way of managing the complexity of database descriptions. The design methodology we propose has three basic characteristics. First, it is structure in the sense that database objects and operations are designed gradually, level by level. Second, it provides an obvious way of enforcing integrity constraints, through the notion of encapsulation [LZ]. Third, the design of a database is carried out in two levels of abstraction, the specification level which uses a high-level design language and is independent of any DBMS, and the representation level that refines the first one into an actual implementation of the database.

Modular database design is not a new idea, but all references known to us [DMW, EKW, LMWW, SFNC, SNF, We] tend to explore the principles, theoretical and otherwise, of the method. We go further and show that a modular database design strategy is quite feasible using currently existing DBMSs. We substantiate this claim by actually showing how the strategy can be implemented on top of the SQL/DS system [IBM1].

This paper is divided as follows. Section 2 carries our informal discussion on modular database design further. Section 3 defines, at the specification level, the concept of module, the module constructors we use and the concept of module graph. Section 4 indicates how a modular description of a database can be implemented on top of the SQL/DS system. Finally, Section 5 contains conclusions and directions for future research.

2. Modular Database Design

We outline in this section a database design methodology based on the concept of modules. Later sections will discuss in detail the concepts introduced here.

We begin with a brief description of modules. At the specification level, the definition of a module consists of a high-level description of the objects and operations of the module as well as their properties. We consider that the objects of a module are relations described by relation schemas, and that the set of consistent relations is defined via a list of integrity constraints, in the usual way.

Operations are defined as procedures called by value, using a high level programming language, the regular programs of [Ha,CB1,CB2]. (Regular programs are surveyed in Appendix I, which may be skipped on a first reading without loss of continuity). This choice is justified on the grounds that: (i) regular programs have a clean syntax and semantics, without departing too much from currently existing DMLs; (ii) regular programs come equipped with a programming logic that permits investigating correctness problems that arise in module definitions; (iii) our experience [SNFC] indicates that the alternative approach, axiomatic specifications, requires quite complex axioms to express even simple operations.

We stress that operations are an integral part of module definitions in the sense that, although users can freely query the current value of module objects, users can only modify their current value using module operations. This discipline guarantees that no integrity constraint is ever violated, if module operations are designed so that they provably preserve consistency.

The representation level description of a module indicates how to implement the objects and operations contained in the specification level description of the module. We shall adopt here for the representation level the DDL/DML of SQL/DS [IBM2], as mentioned in the Introduction.

This concludes our introduction to the concept of module and we now turn to structured database design.

At the specification level, the structure imposed on the database by the designer is represented by a module graph $G=(V,E,r)$. Briefly, G is a labelled directed acyclic graph whose nodes represent modules and is such that there is an edge from node M to node N iff module N is constructed from module M using one of the module constructor mechanisms; the label assigned to N by the labelling function r indicates which constructor was used. A precise definition of module graphs will be given in the next section.

The module graph is constructed gradually by adding new modules to those already existing. Modules may be added without any connection to previously defined modules. In this case, the module is called primitive. But a new module may also be defined with the help of those already existing using two module constructors, extension and subsumption.

We say that a module M_0 is created by extending modules $M_1,\ldots,M_n$ iff M_0 contains only relations derived from those of $M_1,\ldots,M_n$ (thus, the relations of M_0 are views in the usual sense). Moreover, operations of M_0 are implemented in terms of those of $M_1,\ldots,M_n$. Modules $M_1,\ldots,M_n$ are not altered and remain available for further use in module definitions. Thus, the extension constructor is nothing more than the usual subschema mechanism.

We say that a module M_0 is created by subsuming modules $M_1,\ldots,M_n$ iff M_0 contains new relations and all the relations of $M_1,\ldots,M_n$. Likewise, M_0 contains new operations (which may use old operations of $M_1,\ldots,M_n$ as subroutines) and all operations of $M_1,\ldots,M_n$. However, we also allow M_0 to redefine operations of $M_1,\ldots,M_n$.

The subsumption constructor is necessary because sometimes, when adding new structures and new constraints to the database, it becomes necessary to redefine existing operations so that they also obey the new constraints. The fundamental difference between extension and subsumption lies in that, after $M_1,\ldots,M_n$ are subsumed by M_0, modules $M_1,\ldots,M_n$ are no longer available to construct new modules.

We will impose restrictions on how extension and subsumption can be used so that all primitive modules and those defined by subsumption form a forest F. Modules defined by extension in turn form an acyclic digraph G grafted in the forest F. Thus, F plays the role of a hierarchically structured conceptual schema and G defines a structured set of external schemas, using the ANSI/SPARC terminology.

To summarize, the database design methodology outlined provides structured descriptions of the more traditional notions of conceptual and external schemas. In our specific proposal, relation schemas, as well as integrity constraints, can be introduced in a structured, orderly fashion that enhances the understandability of the database design. But, what is even more important, the strategy of encapsulating relations within a set of operations provides an effective method of enforcing integrity constraints. Yet, queries remain unrestrained as in the traditional approach [Zi].

The next sections will explore these concepts further.

3. The Specification Level

This section first gives a precise definition of the concept of module and then moves to module constructors mechanisms and to the concept of module graph, all at the level of specification.

3.1. The Concept of Module

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

A module is a triple M = (RS,CN,OP) where RS is a set of relation schemas, CN is a set of integrity constraints, and OP is a set of operations.

We now discuss each of these concepts in detail.

Since we adopted the relational model, the data structures of M are relations described by a set RS of relation schemas 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 schema.

For each relation schema $R[A_1,\ldots,A_n]$ in RS, we add to L the symbol R as an n-ary predicate symbol and $A_1,\ldots,A_n$ as unary predicate symbols (we assume that none of these symbols is already in L). The first-order language thus defined is called the language of M and is denoted by LM. We also say that LM was created by adding the relation schemas in RS to L.

The set CN of integrity constraints of M is just a set of wffs of LM. CN necessarily contains, for each relation schema $R[A_1,\ldots,A_n]$, a wff of the form

$\forall x_1 \ldots \forall x_n (R(x_1,\ldots,x_n) \Rightarrow A_1(x_1) \varepsilon \ldots \varepsilon A_n(x_n))$,

called a relation schema axiom, that conveys the idea that the interpretation of R must be a subset of the cartesian product of the interpretations of $A_1,\ldots,A_n$.

Finally, operations are defined by procedure definitions over LM of the form $f(x_1,\ldots,x_m)$:p (see Appendix I).

We require from module definitions that:

requirement 1: each operation in OP must preserve consistency with respect to all wffs in CN (see Appendix I for a precise definition). This requirement reflects the fundamental preoccupation that the database should always be left in a consistent state [CCF].

As a matter of syntactical convenience, we denote M=(RS,CN,OP) as follows:

```
module M
   schemes      RS;
   constraints  CN';
   operations   OP;
endmodule
```

where CN' is CN without the relation schema axioms, since these are completely fixed by the schemes in RS.

We close this section with an example.

EXAMPLE 3.1:

We begin in this example the design of a micro database that will continue throughout the paper. The database stores information about products, warehouses and shipments of products to warehouses. Information about products and warehouses is stored and manipulated via the structures and operations defined in two primitive modules, PRODUCT and WAREHOUSE, defined below:

```
module PRODUCT
 schemes
  PROD[P#,NAME]
 constraints
  ∀p∀n∀n'(PROD(p,n) ε PROD(p,n') ⇒ n=n')
 operations
  ADDPROD(p,n):
   if ¬∃n PROD(p,n) ε P#(p) ε NAME(n)
   then PROD := {(x,y)/PROD(x,y) v (x=p ε y=n)}
  DELPROD(p):
   PROD := {(x,y)/PROD(x,y) ε ¬x=p}
endmodule
```

Then, PRODUCT is the triple P=(RS,CN,OP). The language LP of the module then has the following distinguished symbols: a binary predicate symbol, PROD, and two unary predicate symbols, P# and NAME. In view of the relation schema defined, CN contains, in addition to the wff listed after the constraint clause, the following relation schema axiom:

$\forall p \forall n (PROD(p,n) \Rightarrow P\#(p) \; \varepsilon \; NAME(n))$

The set OP consists of the procedure definitions listed after operations.

Module WAREHOUSE is defined likewise:

```
module WAREHOUSE
  schemes
   WAREHSE[W#,LOC]
  constraints
   ∀w∀c∀c'(WAREHSE(w,c) ε WAREHSE(w,c') ⇒ c=c')
  operations
   OPEN(w,c):
    if ¬∃c' WAREHSE(w,c') ε W#(w) ε LOC(c)
     then WAREHSE :=
          {(x,y)/WAREHSE(x,y) v (x=w ε y=c)}
   CLOSE(w):
     WAREHSE := {(x,y)/WAREHSE(x,y) ε ¬ x=w}
endmodule
```

This concludes the example.

3.2. Module Constructors

Let L be again a fixed first-order language containing all ordinary symbols. Let $M_i = (RS_i, CN_i, OP_i)$, $i=1,\ldots,n$, be modules. Assume that M_i and M_j have no relation names in common.

The extension constructor captures the usual subschema mechanism and may be used to redefine or hide structures as well as operations of old modules. We define a new module M by extension of $M_1,\ldots,M_n$ as follows:

(1) module M extends $M_1,\ldots,M_n$ with

```
        schemes      RS0;
        constraints  CN0;
        operations   OP0;
     using
        views        VW;
        surrogates   SR;
   endmodule
```

This constructor actually has two parts. The triple (RS_0,CN_0,OP_0) defines a new module M_0 in the sense of Section 3.1. We assume that no relation name of M_0 is used in M_i, i=1,...,n.

The pair (VW,SR) then couples M_0 to $M_1,\ldots,M_n$ in the following sense. Let LM be the language obtained by adding all relation schemas of $M_0,M_1,\ldots,M_n$ to L. Let OP be the union of $OP_1,\ldots,OP_n$ (i.e., OP contains all procedures defined in $M_1,\ldots,M_n$). VW contains, for each scheme $R[A_1,\ldots,A_m]$ in RS_0, a view definition, which is a statement of the form $R[A_1,\ldots,A_k]:Q$, where Q is a wff of LM with n free variables ordered $x_1,\ldots,x_k$. We interpret Q as defining R in terms of the relation schemas of $M_1,\ldots,M_n$. SR contains, for each procedure definition $f(y_1,\ldots,y_m):p$ in OP_0, a surrogate, which is again a procedure definition over LM and OP of the form $f(y_1,\ldots,y_m):q$, (that is, q is a regular program over LM that may contain calls to the procedures defined in $M_1,\ldots,M_n$). We understand $f(y_1,\ldots,y_m):q$ as defining $f(y_1,\ldots,y_m):p$. In other words, $f(y_1,\ldots,y_m):p$ is the operation the user believes he is using, but it is $f(y_1,\ldots,y_m):q$ that actually modifies the database. This remark should be kept in mind throughout the rest of the paper.

We require that:

requirement 2: if $f(y_1,\ldots,y_m):q$ is the surrogate of $f(y_1,\ldots,y_m):p$ then q is VW-equivalent to p (see Appendix I for a precise definition);

requirement 3: if $f(y_1,\ldots,y_m):q$ is a surrogate, then q can only modify the values of schemes in $M_1,\ldots,M_n$ through calls to the operations defined in $M_1,\ldots,M_n$;

requirement 4: for each wff P in CN_0, P' must be a logical consequence of $CN_1,\ldots,CN_n$, where P' is obtained from P by replacing each atomic formula of the form $R(z_1,\ldots,z_k)$ by $Q[z_1/x_1,\ldots,z_k/x_k]$, where $R[A_1,\ldots,A_k]:Q$ is a view definition, and the list of free variables of Q is $x_1,\ldots,x_k$.

Requirement 2 guarantees that q correctly implements p. That is, p defines an operation of the module as seen by the user of the module. However, since this operation is on virtual objects (the views), it has to be implemented by operations on the base objects. This implementation is described by q. Requirement 2 can then be interpreted as saying that p and q must have the same effect as seen from the user's point of view. In other words, we avoid the so-called view update problem [DB,SF] by passing it back to the DB designer. Requirement 3 guarantees that each surrogate preserves consistency with respect to $CN_1,\ldots,CN_n$. Requirement 4 guarantees that the integrity constraints of M follows from those of $M_1,\ldots,M_n$ and the view definitions. Thus, no local constraints can really be defined in a module created by extension. Finally, we observe that requirements 2, 3 and 4 guarantee that each operation in OP_0 preserves consistency with respect to CN_0.

Further requirements will be imposed in Section 3.3.

EXAMPLE 3.2:

We define a new module, DELIVERY, by extending the module SHIPMENT of Example 3.3 below as follows:

```
module DELIVERY extends SHIPMENT with
  schemes
    DELVRY[P#,W#];
  constraints
    /* (none) */
```

```
    operations
      DEL(p,w):
        DELVRY := {(x,y)/ DELVRY(x,y) ε
                            ¬(x=p ε y=w)}
  using
    views
      DELVRY[P#,W#] : ∃q SHIP(p,w,q)
    surrogates
      DEL(p,w):
        CANSHIP(p,w)
  endmodule
```

This concludes the example.

We now turn to the subsumption constructor. We begin by observing that it should be used to add new relation schemas and integrity constraints, and to redefine previous operations (which may be required to maintain integrity). We indicate that a new module M is created by subsuming $M_1,\dots,M_n$ as follows:

```
(2)  module M subsumes M1,...,Mn with
          schemes       RS0;
          constraints   CN0;
          operations    OP0;
       using
          replacements RE;
     endmodule
```

We take RS_0 to be a set of relation schemas and assume that no relation name in RS_0 occurs in $M_1,\dots,M_n$.

Let LM again be the language obtained by adding all relation schemas of $M_1,\dots,M_n$ and those in RS_0 to L. Let OP again be the union of $OP_1,\dots,OP_n$. Then, CN_0 is a set of wffs over LM and OP_0 is a set of procedure definitions over LM and OP.

RE is a possibly empty set of clauses of the form

$g(z_1,\dots,z_k)$ is replaced by $f(y_1,\dots,y_k)$: p

where $g(z_1,\dots,z_k)$ is a procedure defined in M_i, for some i in [1,n], and $f(y_1,\dots,y_k)$:p is a procedure definition over LM and OP. We treat $f(y_1,\dots,y_k)$:p as a new procedure definition of M, just as those in OP_0.After the definition of M, operation q cannot be called directly anymore.

The module M defined by the expression in (2) is then the triple (RS,CN,OP) where RS is the union of $RS_0,\dots,RS_n$, CN is the union of $CN_0,\dots,CN_n$ and OP is the union of $OP_0',OP_1',\dots,OP_n'$ where OP_i' is OP_i without all procedure definitions that were redefined in clauses of RE,for i=1,...,n,and OP_0' is the set of all new procedure definitions contained in OP_0 or in clauses of RE.

We require that:

requirement 5: each operation in OP preserves consistency with respect to CN_0;

requirement 6: each operation in OP can only modify the values of schemes in $M_1,\dots,M_n$ through calls to the operations defined in $M_1,\dots,M_n$;

requirement 7: each operation of M_i, for some i, replaced in a clause of RE must not have been used in the surrogates clause of any previously defined module. Requirements 5 and 6 guarantee that each operation in OP preserves consistency with respect to CN. Requirement 7 guarantees that operations redefinitions will not propagate to other modules.

Further requirements will be imposed in Section 3.3.

EXAMPLE 3.3:

We can add a relationship between PRODUCT and WAREHOUSE, called SHIPMENT, as follows:

```
module SHIPMENT subsumes PRODUCT,WAREHOUSE with
  schemes
    SHIP[P#,W#,QTY]
  constraints
    ∀p∀w∀q∀q'(SHIP(p,w,q) ε SHIP(p,w,q') ⇒ q=q')
    ∀p(∃w∃q SHIP(p,w,q) ⇒ ∃n PROD(p,n))
    ∀w(∃p∃q SHIP(p,w,q) ⇒ ∃c WAREHSE(w,c))
  operations
   ADDSHIP(p,w,q):
    if ∃n PROD(p,n) ε ∃c WAREHSE(w,c)
       ∃q' SHIP(p,w,q') ε QTY(q)
       then SHIP:= {(x,y,z)/SHIP(x,y,z) v
                     (x=p ε y=w ε z=q)};
   CANSHIP(p,w):
    SHIP := {(x,y,z)/ SHIP(x,y,z) ε ¬(x=p ε y=w)};
  using
```

replacements

CLOSE is replaced by

CLOSE1(w):

if $\neg\exists p \exists q$ SHIP(p,w,q) then CLOSE(w);

DELPROD is replaced by

DELPROD1(p):

if $\neg\exists w \exists q$ SHIP(p,w,q) then DELPROD(p);

endmodule

This concludes the example.

We close this section by observing that our module constructors are very general mechanisms that subsume the database abstractions - aggregation, generalization and correspondence - of [SS,DMW,SFNC]. This reflects our point of view that these three abstractions are just a sample of the variety of constructors obtained by restricting the mappings that can be used to build new modules. However, such restrictions become interesting when certain properties of module constructors are sought [DMW].

3.3. Module Graphs

As briefly discussed in Section 2, the structure imposed on the database by the designer is represented by a module graph, that is, a labelled directed acyclic graph whose nodes represent modules, whose edges indicate relationships between modules and whose labelling function assigns tags to nodes indicating how the module was created.

To define module graphs and the new requirements, we use the concept of active module. Intuitively, a module M is active in a module graph G iff M is either primitive or defined by subsumption, and in either case M was not subsumed by another module.

We capture both the dynamic aspects of module graphs and the new requirements on module constructors in the following recursive definition of module graphs:

DEFINITION 3.1: The set of module graphs, together with their sets of active modules, is recursively defined as follows:

(1) the empty graph is a module graph with an empty active module set;

(2) Let G=(V,E,r) be a module graph with active module set A. Let M be a primitive module not in V such that no relation name of M occurs in a module in V. Then G'=(V',E',r') is a module graph with active module set A', where:

V' = V ∪ {M};

E' = E;

r'(N) = r(N), if N is in V,

and r'(M) = 'primitive';

A' = A ∪ {M}.

(3) Let G=(V,E,r) be a module graph with active module set A. Let M be a module obtained by extension from $M_1,\dots,M_n$ such that M is not in V and $M_1,\dots,M_n$ are in V, and no relation name of M occurs in a module of V.

Suppose that:

requirement 8: for each i in [1,n], M_i is either defined by extension, or in the active set of G.

Then, G'=(V',E',r') is a module graph with active module set A', where:

V' = V ∪ {M};

E' = E ∪ {(M_i,M)/i=1,...,n};

r'(N)=r(N), if N is in V,

and r'(M)='extension'.

A' = A;

(4) Let G=(V,E,r) be a module graph with active module set A. Let M be a module obtained by subsuming $M_1,\dots,M_n$ such that M, is not in V and $M_1,\dots,M_n$ are in V, and the relation names of M are those of $M_1,\dots,M_n$ plus a new set of relation names not occurring in any module in V.

Suppose that:

requirement 9: for each i in [1,n], M_i is in the active set of G.

Then, G'=(V',E',r') is a module graph with active module set A', where:

V' = V ∪ {M};

E' = E ∪ {(M_i,M)/i=1,...,n};

r'(N)=r(N), if N is in V,

and r'(M) = 'subsumption'.

$$A' = A \cup \{M\} - \{M_1,\dots,M_n\}. \square$$

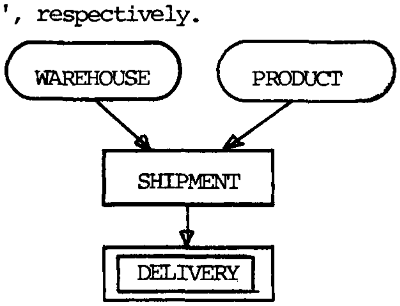
The result of module constructors is captured by G in the sense that there is an arc from N to M iff the definition of M depends on N. Hence, if M is a primitive module, it has no ingoing arcs and if M extends or subsumes $M_1,\dots,M_n$ then there is an arc from M_i to M, for each $i=1,\dots,n$. Since definitions must not be circular, G has to be acyclic, which can easily be proved from the definition.

The concept of a module graph would be complemented by the concept of an <u>operation graph</u> representing the calling relationship between operation specifications. Such graphs would be very similar to the D-graphs of [We]. However, for reasons of brevity we omit its definition.

The following example illustrates the construction of module graphs.

EXAMPLE 3.4:

The module graph corresponding to the modules in Examples 3.1 to 3.3 is shown below. The following notational convention is used to represent the labelling function: an oval, rectangle or double rectangle represents a node labelled 'primitive', 'extension' or 'subsumption', respectively.



This concludes the example. $\square$

The module graph captures the complete modular structure of the database. However, not all modules are <u>visible</u> to users, that is, the user cannot query all modules in G. Likewise, since some operations are redefined whereas others have surrogates, not all operations can be called directly to modify the database, but only those that are <u>active</u>.

DEFINITION 3.2: Let $G=(V,E,r)$ be a module graph with active module set A.

(a) the modules in A form the <u>conceptual schema corresponding to G</u>, and the set of all modules in V defined by extension form the set of <u>external schemas</u> of G. These are the modules that are <u>visible</u> to the users.

(b) The set of <u>active operations</u> of G consists of the set of all operations of active modules of G, plus the set of all surrogates of modules defined by extension in G.

The definition above captures the meaning of a module graph $G=(V,E,r)$ from the user's point-of-view. Another question we may ask is what is the formal semantics of the database described by G. We now briefly discuss this point. We begin by observing that we can associate with G a first-order theory $T=(LT,AT)$ and a set of regular programs RP over LT, where:

(i) LT is the first-order language obtained by adding all relation schemes in modules in V to the base first-order language L;

(ii) AT is the set of all integrity constraints in modules of G, plus the defining axioms for views;

(iii) RP is the set of procedure definitions contained in modules in V.

Now we observe that the semantics of the database described by G is fixed once a universe U for LT (see Appendix I) is fixed. We must assume that each structure of LT in U satisfies all view defining axioms so that views can indeed be considered as defined symbols of T. Thus, each structure in U corresponds to a database state, together with the appropriate values for views and for the ordinary symbols. Given U, the meaning of all operations of modules in V is also fixed by definition (see Appendix I).

The reader is referred to [DMW,EKW] for an alternative formal discussion on modular database specifications.

Finally, we observe that requirements 1, 2 and 5 depend on the universe U that fixes the meaning of the database. However, if U is chosen so that the module graph satisfies these requirements as well as all others, then all active operations indeed preserve consistency.

THEOREM 3.1: Let G=(V,E,r) be a module graph. Let U be the universe that fixes the meaning of the database. Suppose that requirements 1 through 9 were satisfied during the construction of G (for this choice of universe). Then, every active operation of G preserves consistency with respect to the set of all constraints defined in modules of G.

Sketch of Proof

Follows by induction on the number of nodes of G, using requirements 1 through 9 (see [TFC]). □

We can also prove that the set of primitive modules and those defined by subsumption form a hierarchy.

PROPOSITION 3.2: Let G=(V,E,r) be a module graph. Let G'=(V',E') be the subgraph of G spanned by the set of all nodes of G labelled with 'primitive' or 'subsumption'. Then, G' is a forest.

Sketch of Proof

Follows directly from requirement 9 and Definition 3.1. □

This concludes our discussion about structured database design as far as the specification level goes. The next section explains how to represent these ideas in a concrete environment.

4. The Representation Level

This section discusses how to map descriptions of modules from the specification level to the representation level. As already mentioned in previous sections, we adopt SQL/DS as our target system. We begin with a brief description on how SQL/DS facilities can be used to represent module and module constructors. Then, we exemplify the discussion by showing the representation of the SHIPMENT module.

Consider first a primitive module M = (RS,CN,OP). Each relation schema is RS can be defined directly in SQL/DS through the 'CREATE TABLE' command.

EXAMPLE 4.1:

The schema of Example 3.3 would be defined as follows:

```
CREATE  TABLE SHIP
          ( P#  CHAR(10) NOT NULL,
            W#  CHAR(10) NOT NULL,
            QTY INTEGER            )
          IN dbspace-name;          . □
```

Constraints in CN do not generate statements in SQL. Indeed, the role of constraints is limited to a declarative definition of the semantics of the database, which is procedurally implemented through the definition of the operations.

Operations are implemented as PL/I procedures with embedded SQL statements. We suggest using the following skeleton for the procedures (although we do not show it here for reasons of clarity, error routines should also be present in the actual implementation of operations):

```
PROGRAM-NAME:PROC(parameter list)
 delcaration of SQL/DS variables
 verification of conditions that prevent
  violation of integrity constraints
 effect of the operation
  update of the database using DBMS primitives
  or
  call to subsumed operations
 return to the calling program
END
```

Note that there is no COMMIT or ROLLBACK statements in the above skeleton. In fact, we do not define an operation as an SQL/DS work unit, since it sbould be the user's responsability to define which sequences of operations constitute a transaction (or work unit). Hence, the user is responsible for establishing the initial connection with SQL/DS for authorization and for concluding his transaction with COMMIT or ROLLBACK, depending on the success of his transaction. A prologue and epilogue for these purposes could be implemented as PL/I macros. This concludes our brief discussion about primitive modules.

We represent a module defined by extension as follows. The schemes, constraints and operations clauses of the module definition may be ignored. Each view definition in the views clause is represented directly by the 'CREATE VIEW' command of SQL/DS.

EXAMPLE 4.2: The DELVRY view of Example 3.2 would be defined as follows:

```
CREATE VIEW DELVRY AS
        SELECT P#, W#
        FROM SHIP;
```

.□

Operations defined in the surrogates clause are implemented as normal operations, using the ability to call predefined operations from other predefined operations provided by SQL/DS, if it is the case. The fact that each view update is accompanied by an implementation (a surrogate) in terms of the base relations has two advantages. First, the restrictions imposed by SQL/DS on view updates do not affect our discussion. Second, users will still interact with the database as if they were actually updating views.

We now turn to the subsumption operator. A module M defined using the subsumption operator is also straightforward to represent. Each new scheme and each new operation or operation redefinition of M is represented as for primitive modules. The interesting point concerns how to control access to redefined operations, since users cannot call them directly anymore. This restriction is implemented by simulating a CONNECT statement inside a redefined operation so that only the DBA has access to it. (The authorization mechanisms of SQL/DS cannot be used for this purpose because the CONNECT statement is executed by the user's program and remains in effect for the entire execution of the transaction. This solution was, in fact, the first one adopted and did not work out). To simulate a CONNECT, each user must pass as additional parameters his ID and password. When an operation p is redefined, its code is altered to explicitly test if the user ID and password are those of the DBA. If not, then the operation is rejected. Moreover, if the new operation q (replacing the old operation p) will call p, then q must call p with the ID and password of the DBA. This strategy is reflected in the implementation of CLOSE1 and DELPROD1 shown in Example 4.3 below.

This concludes our brief discussion on the representation of modules. We close this section by exhibiting the complete representation of the SHIPMENT module of Example 3.3.

EXAMPLE 4.3: The SHIPMENT module of Example 3.3 is represented as follows:

a) Representation of the schemes:

```
  CREATE  TABLE SHIP
        ( P#  CHAR(10) NOT NULL,
          W#  CHAR(10) NOT NULL,
          QTY INTEGER             )
        IN dbspace-name;
```

b) Representation of the operations:

```
ADDSHIP:PROC(P#,W#,QTY,USERID,PASSWD,RETCODE);
  EXEC SQL BEGIN DECLARE SECTION;
    DCL P#     CHAR(10);
    DCL W#     CHAR(10);
    DCL QTY    FIXED BIN(31);
    DCL COUNT0 FIXED BIN(31);
    DCL COUNT1 FIXED BIN(31);
    DCL COUNT2 FIXED BIN(31);
  EXEC SQL  END  DECLARE SECTION;
    DCL USERID FIXED CHAR(8);
    DCL PASSWD FIXED CHAR(8);
    DCL RETCODE FIXED BIN(31);
/*
  execution of SQL/DS statements
*/
 EXEC SQL SELECT COUNT(*) INTO :COUNT0
          FROM SHIP
          WHERE P# = :P# AND W# = :W#;
 EXEC SQL SELECT COUNT(*) INTO :COUNT1
          FROM PROD
          WHERE P# = :P#;
 EXEC SQL SELECT COUNT(*) INTO :COUNT2
          FROM WAREHSE
          WHERE W# = :W#;
/*
  update of the database, provided that no
  constraint will be violated
  (see Example 3.3 for constraint definition)
*/
 IF COUNT0  = 0   ε
          /* check violation constr. 1 */
    COUNT1 ¬= 0   ε
          /* check violation constr. 2 */
    COUNT2 ¬= 0
          /* check violation constr. 3 */
 THEN
 DO;
   EXEC SQL  INSERT
          INTO SHIP VALUES (:P#,:W#,:QTY);
```

```
    /* SQL/DS return code ignored           */
    RETCODE = 0;  /* indicates normal return */
  END;
  ELSE
    RETCODE = 1;
    /* indicates integrity violation  */
 RETURN;
END;
```

```
CANSHIP:PROC(P#,W#,USERID,PASSWD,RETCODE);
   EXEC SQL BEGIN DECLARE SECTION;
      DCL P#     CHAR(10);
      DCL W#     CHAR(10);
   EXEC SOL  END  DECLARE SECTION:
      DCL USERID FIXED CHAR(8);
      DCL PASSWD FIXED CHAR(8);
      DCL RETCODE FIXED BIN(31);
/*
    operation effect
*/
   EXEC SQL DELETE FROM SHIP
            WHERE P# = :P# AND W# = :W#;
    /* SQL/DS return code ignored           */
   RETCODE = 0; /* indicates normal return */
   RETURN;
END;
```

```
CLOSE1:PROC(W#,USERID,PASSWD,RETCODE);
   DCL CLOSE ENTRY(CHAR(10),FIXED CHAR(8),
         FIXED CHAR(8),FIXED BIN(31)) EXTERNAL;
   EXEC SQL BEGIN DECLARE SECTION;
      DCL W#      CHAR(10);
      DCL COUNT   FIXED BIN(31);
   EXEC SQL  END  DECLARE SECTION;
      DCL USERID FIXED CHAR(8);
      DCL PASSWD FIXED CHAR(8);
      DCL RETCODE FIXED BIN(31);
      DCL DBAID    FIXED CHAR(8);
      DCL DBA_PASSWD    FIXED CHAR(8);
/*
  execution of SQL/DS statements
*/
  EXEC SQL SELECT COUNT(*) INTO :COUNT
           FROM SHIP
           WHERE W# = :W#;
/*
  execution of the CLOSE operation under new
  conditions that reflect the new constraints.
  (CLOSE  is called as if the user were the
  DBA since users should not have access to
  it anymore)
*/
  IF COUNT  = 0  THEN
   DO;
    DBAID  = 'YYYYYYYY';
    DBA_PASSWD = 'XXXXXXXX';
    CALL CLOSE(W#,DBAID,DBA_PASSWD,RETCODE);
   END;
   ELSE
    RETCODE = 1;
    /* indicates integrity violation */
  RETURN;
END;
```

```
DELPROD1:PROC(P#,USERID,PASSWD,RETCODE);
  ( Structured identically to CLOSE1 )
```

This concludes the example and this section.

5. Conclusions and Directions for Future Research

In this paper we outlined a methodology that provides mechanisms both to structure the logical design of databases, using the concept of module, and to enforce consistency preservation, through the encapsulation of database structures within predefined operations. Unlike previous work on modular database design, we covered implementation aspects of the methodology, rather than concentrating on theoretical issues.

Central to the development of the paper was the selection of module constructors that could be easily implemented and yet helped structure the database design. The implementation of such constructors could be carried out further by designing a preprocessor that would automatically do some of the translation from module specifications to SQL/DS statements outlined in Section 4.

Finally, we observe that modular database design acquires another (and considerable) significance in the context of database evolution, since variations of subsumption could also be used to change the database design in response to evolutions in the application.

ACKNOWLEDGEMENTS: The authors are grateful to C.J. Date for helpful suggestions and discussions.

REFERENCES

[ANSI] "Study Group on Data Base Management Systems: Interim Report", FDT 7.2,ACM (1975)

[CB1] M.A.Casanova,P.A.Bernstein. "The Logic of a Relational Database Manipulation Language". Proc. 5th ACM Symp. Principles of Programming Languages (1979)

[CB2] M.A.Casanova, P.A.Bernstein. "A Formal System for Reasoning about Programs Accessing a Relational Data Base". ACM Trans.on Programming Languages 2,3 (1980)

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

[CV] M.A.Casanova, V.M.P. Vidal. "Towards a Sound View Integration Methodology". Proc. 2nd ACM SIGMOD/SIGACT Conf. on Principles of Database Systems (1983).

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

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

[FS] A.L. Furtado, K.C. Sevcik. "Permitting Updates through Views of Data Bases". Information Systems 4 (1979)

[DB] U. Dayal, P.A. Bernstein. "On the Correct Translation of Update Operations on Relational Views". ACM TODS 7.3 (1982)

[Ha] D. Harel. "First-Order Dynamic Logic", Lecture Notes in Computer Science (1979)

[IBM1] IBM Pub. SH24-5014. "SQL/Data System Planning and Administration"

[IBM2] IBM Pub. SH24-5018. "SQL/Data System Application Programming"

[LMWW] P.C. Lockemann, H.C. Mayr, W. H. Weil, W.H. Wohlleber. "Data Abstractions for Data Base Systems".ACM Trans. on Database Systems, 4.1 (1979)

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

[NG] S.B.Navathe, S.G.Gadgil. " A Methodology for View Integration in Logical Database Design",Proc.8th Int'l Conf.on Very Large Data Bases (1982)

[Pa] D. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules". CACM 15.12 (1972)

[SFNC] U. Schiel, A.L. Furtado, E.J. Neuhold, M.A.Casanova. "Towards Multi-level and Modular Conceptual Schema Specifications" Technical Report Institut fur Informatik, Universitat Stuttgart (1982)

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

[SS] J.M. Smith, D.C.P. Smith. "Database Abstractions: Aggregation and Generalization". ACM TODS 2.2 (1977)

[TF] T.J.Teorey, J.P.Fry, "Design of Database Structures", Prentice-Hall, Inc. (1982)

[TFC] L.Tucherman, A.L. Furtado, M.A. Casanova. "A Pragmatic Approach To Structured Database Design", Departamento de Informática Pontifícia Universidade Católica, Rio de Janeiro, Brazil (1983).

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

[WM] G. Wiederhold, R. El-Masri. "A Structural Model for Database Systems", TR STAN-CS-79-722, Stanford Univ. (1979)

[YWH] S.B. Yao, V.E. Waddle, B.C. Housel. "View Modelling and Integration using the Functional Data Model". IEEE Trans. on Software Engineering, SE-8.6 (1982)

[Zi] S.N.Zilles."Types,Algebras and Modelling" Proc.of the Workshop on Data Abstractions Data bases and Conceptual Modelling. Pingree Park, Colorado (1980)

[ZLT] S.N. Zilles, P. Lucas, J.W. Thatcher. "A Look at Algebraic Specifications". Research Rep. RJ 3568 IBM Thomas J.Watson Research Center (1982)

APPENDIX I

Let L be a first-order language with a set of distinguished constants, called <u>scalar program variables</u>, and a set of distinguished <u>n-ary predicate</u> symbols, called <u>n-ary predicate program variables</u>, for each n > 0. (The relation names must be chosen from this special set of predicate symbols). A <u>universe</u> U for L is a set of structures of L satisfying three conditions:

(i) any two structures in U differ only on the values of the scalar or predicate program variables;

(ii) for any I in U, any scalar program variable x and any element e of the domain of I, there is J in U such that I and J differ only on the value of x, which is e in J;

(iii) for any I in U, any n-ary predicate program variable R and any n-ary relation r over the domain of I, there is J in U such that I and J differ only on the value of R, which is r in J.

These conditions guarantee that, for example, if the value of x is changed to e, the resulting structure is in U. That is, the universe is closed under assignment, so to speak. Note that, by (i), all structures in U have the same domain.

The set of <u>regular programs over L</u>, RP[L], is then defined inductively as follows:

<u>syntax</u>

(1) For any scalar program variable x of L and any term t of L, x:=t is in RP[L] and is called an <u>assignment</u>;

(2) for any n-ary predicate program variable R of L and any wff P of L with a list $x_1,...,x_n$ of free variables, $r:=\{(x_1,...,x_n)/P\}$ is in RP[L] and is called a <u>relational assignment</u>;

(3) for any wff P of L, P? is in RP[L] and is called a <u>test</u>;

(4) for any p and q in RP[L], p ∪ q, p ; q and p* are in RP[L] and are called the <u>union</u> of p and q, the <u>composition</u> of p and q and the <u>iteration</u> of p, respectively.

For a given structure I of L and a symbol s of L, let I(s) denote the value of s in I. Likewise, let I(t) be the value of a term t of L in I.

<u>semantics</u>: for a fixed universe U of L, the meaning of programs in RP[L] is given by a function m assigning to each r in RP[L] a binary

relation m(r) in U as follows:

(5) $m(x:=t) = \{(I,J)/ J$ is equal to I, except that $J(x)=I(t)\}$

(6) $m(R:=\{(x_1,\ldots,x_n)/P\}) = \{(I,J)/ J$ is equal to I, except that J(R) is the n-ary relation defined by P in I$\}$

(7) $m(P?) = \{(I,I)/$ P is valid in I$\}$

(8) $m(p \cup q) = m(p) \cup m(q)$ (union of both binary relations)

(9) $m(p;q) = m(p) \circ m(q)$ (composition of both binary relations)

(10) $m(p^*) = (m(p))^*$ (reflexive and transitive closure of m(r))

We proceed by introducing the notion of procedures. Let C be a set of procedure declarations, which are statements of the form $f(x_1,\ldots,x_m)$, where f is a new symbol not in L and $x_1,\ldots,x_m$ are scalar or predicate program variables of L. The set of regular programs over L and C, RP[L,C], is defined as before, with one additional rule:

(11) if $f(x_1,\ldots,x_m)$ is in C, then $f(z_1,\ldots,z_m)$ is in RP[L,C], where z_i is a term of L, if x_i is a scalar program variable of L, or z_i is of the form $\{(y_1,\ldots,y_k)/P\}$, if z_i is a k-ary predicate program variable of L;

Meaning is assigned to programs in RP[L,C] as follows. First, we associate a procedure body p with each procedure declaration $f(x_1,\ldots,x_m)$ in C, where p is a program in RP[L,C]. Then, we define a function m as before, except that

(12) $m(f(z_1,\ldots,z_m)) =$

$$m(x_1:=z_1 \ ; \ \ldots \ ; \ x_m:=z_m \ ; \ p).$$

We may also introduce some familiar constructs by definition as follows:

(13) if P then r else s $= (P?;r) \cup (\neg P?;s)$

(14) if P then r $= (P?;r) \cup \neg P?$

(15) while P do r $= (P?;r)^*;\neg P? \cup \neg P?$

This completes our brief description of regular programs. We refer the reader to [CB1,CB2] for a fuller discussion.

We close this appendix with two concepts. Let W be a set of wffs. We say that program p preserves consistency with respect to W (in a given universe U) iff for any (I,J) in m(p), if I satisfies all wffs in W, then so does J.

Let V be a finite set of view definitions, $R_i[A_{i1},\ldots,A_{im_i}]:P_i$, $i=1,\ldots,n$. Let r be the program

(16) $R_1:=\{\bar{x}_1/P_1\} \ ; \ \ldots \ ; \ R_n := \{\bar{x}_n / P_n\}$

Let p and q be two programs. We say that p is V-equivalent to q iff

(i) if (I,J) is in m(p;r) then there is (I,K) in m(r;q) such that the values of R_i in J and K are the same, for each $i=1,\ldots,n$;

(ii) if (I,K) is in m(r;q) then there is (I,J) in m(p;r) such that the values of R_i in J and K are the same, for each $i=1,\ldots,n$;

Intuitively, program r constructs all views in V from the base relations. Program r;q captures the idea that view update q is applied to the views constructed by r from some initial state I. Program p;r translates the view update by applying some update p to the base relations and then constructing the views using r. If p is V-equivalent to q, then we consider that p is a faithful translation of q. This concludes our brief summary of the definitions concerning regular programs that we need in the body of the paper.