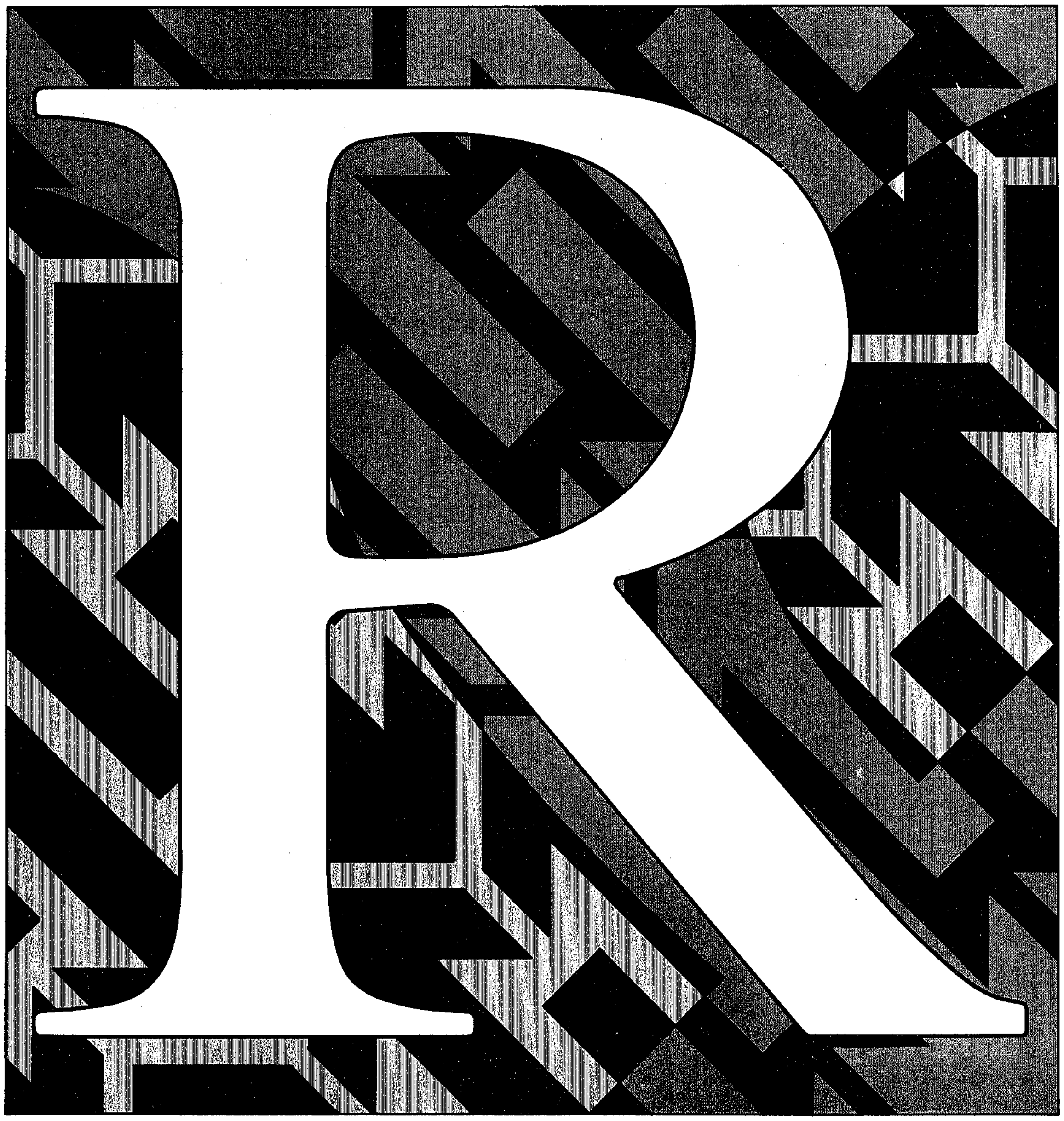
Relational databases

State of the Art Report 14:5



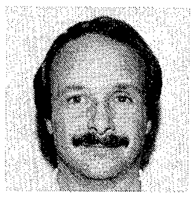
Pergamon Infotech Limited A member of the Pergamon Group Oxford New York Toronto Sydney Beijing Frankfurt

17: Modular database design tools

L Tucherman
A L Furtado
Latin American
Systems Research Institute
IBM Brazil, Rio de Janeiro, Brazil

M A Casanova
Brazilia Scientific Center
IBM Brazil, Brazília, Brazil

A modularisation discipline for the design of complex database schemata is first described. The discipline incorporates strategies for enforcing integrity constraints and for organising large sets of database structures, integrity constraints and operations. A software tool for the development and maintenance of database schemata, modularised according to the discipline, is then described. A PROLOG implementation of the tool is completely operational.



L Tucherman
Luiz Tucherman holds a BSc in Electrical/Civil Engineering from the Federal University of Juiz de Fora and an MSc in Computer Science from the Pontifical Catholic University of Rio de Janeiro, where he is presently enrolled in the PhD program in Computer Science. Mr Tucherman is currently a researcher at the IBM Latin American Systems Research Institute.

A L Furtado
Antonio Furtado has a BSc in Economics from the State University of Rio de Janeiro, an MSc in Business Administration from the Getulio Vargas Foundation, an MSc in Computer Science from the Pontifical Catholic University of Rio de Janeiro and a PhD in Computer Science from the University of Toronto. Dr Furtado is currently a visiting researcher at the IBM Latin American Systems Research Institute and a professor (on leave of absence) at the Pontifical Catholic University in Rio de Janeiro.

M A Casanova
Marco Casanova has a PhD and an MA in Applied Mathematics, both from Harvard University, an MSc in Computer Science from the Pontifical Catholic University of Rio de Janeiro and a BSc in Electronic Engineering from the Military Institute of Engineering. Dr Casanova is a researcher at the IBM Brasília Scientific Center, where he is conducting research into database theory, distributed database management systems and logic programming. He is author of the book 'The concurrency control problem for database systems', published by Springer-Verlag, Inc, co-author of the book 'Principios de sistemas de gerencia de bancos de dados distribuidos', published by Editora Campus, and has published several articles in international scientific journals.

Modular database design tools

Introduction

This paper describes a modularisation discipline, designed by the authors, for database schemata, together with a software tool supporting its application. Their basic motivation stemmed from the problem of organising the design and maintenance of complex database schemata, understood here in the broader sense of a description of both the data structures (static part) and the transactions (dynamic part *(TUC1)*) of an Information System (IS) developed around a database.

The modularisation discipline permits collecting closely related structures, constraints and operations into separate modules *(TUC2–TUC4)*, which are in turn introduced in a structured way, thus enhancing the understanding of the database. Note that modules also include generic constraints, which is a reflection of the view that constraints act as a declarative documentation of additional semantics of the data, whereas operations incorporate these 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 *(TUC3)*. 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 this discipline, just as in the traditional database design strategies *(TUC5, TUC6)*.

The design of a database schema in this discipline consists essentially of the successive addition of new modules to a (possibly empty) kernel database schema. The authors also recognise 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 the authors 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 to add, delete or modify the definition of objects of the schema. A first prototype of the tool, written in micro-PROLOG *(TUC7)* extended with APES *(TUC8)*, 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 modularisation discipline, which facilitates the incremental addition of new expertise about database design.

The usefulness of tools in the design of ISs based on databases is widely recognised. For one thing, 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 areas, 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 *(TUC9)*.

The knowledge to be incorporated into 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, one can conceive an expert tool driven by the rules and having 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 *(TUC10–TUC21)*. 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 generalisation *(TUC22)*. They cope with the design of the dynamic part by incorporating a programming language that supports the types of the data model adopted and incorporates high-level control abstractions. Modularisation is a concept, orthogonal to those just mentioned, which imposes a design discipline to cope with size and complexity. Thus, strictly speaking, modularisation does not buy additional modelling power.

Modularisation, in database conceptual design, is by no means a novel idea *(TUC23)*. 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).

For the module constructors the situation is as follows. Some form of the authors' external constructor is present in most conceptual design languages. In particular, RIGEL *(TUC15)* and PLAIN *(TUC21)* include view update translations in their version of the constructor, as do the authors. But, apart from this constructor, the proposals differ considerably. Weber *(TUC23)* structures the use of modules in the tradition of software engineering; ASTRAL *(TUC10)* uses a block structure policy to limit the scope of data structure declarations; Galileo *(TUC11)* uses the notion of environment, which is a denotable value, as a modularisation mechanism; ADAPLEX *(TUC16)* adopts a form of Ada packages for this purpose; TAXIS *(TUC14)* 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 Artificial Intelligence (AI), for example CLEAR *(TUC24,TUC25)* contains several proposals for organising the definition of theories. The modularisation discipline described in this paper was first published in *(TUC26)* and *(TUC27)*. The software tool is partially described in *(TUC27)*.

Modular database design

The basic concepts of the database design method are described in this section. The concept of a module is first defined, then the module constructors are introduced and, finally, a complete example is described.

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 $R[A_1,...,A_n]$, where R is the *relation name* and $A_1,...,A_n$ are the *attributes* of the scheme. R is treated as an n-ary predicate symbol and $A_1,...,A_n$ as unary predicate symbols and it is assumed 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 if L' is obtained by adding the symbols $R,A_1,...,A_n$ to L, for each relation scheme $R[A_1,...,A_n]$ 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 $f(x_1,...,x_n){:}s$. 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 $R[A_1,...,A_n]$, a *relation scheme axiom* of the form:

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

indicating that the interpretation of R must be a subset of the Cartesian product of the interpretations of $A_1,...,A_n$.

3 OP is a set of operations over LM.

The *objects* of M are the relation schemes, integrity constraints and operations of M.

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 Database Administrator (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$_i$), $i = 1,...,n$, be modules. Let us consider the subsumption constructor first. Intuitively, if the DBA defines M by subsumption over modules $M_1,...,M_n$, 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 $M_1,...,M_n$. M also inherits all operations of $M_1,...,M_n$, 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 $M_1,...,M_n$ 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 $M_1,...,M_n$:

```
(2)  module    M  subsumes M1,...,Mn with
           schemes      RS0;
           constraints  CN0;
           operations   OP0;
           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 $M_1,...,M_n$, 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$_1$,...,RS$_n$.

3 OP$_0$ is a set of operations over RS$_0$,RS$_1$,...,RS$_n$.

4 EN is a set of *enforcement clauses* of the form '*O enforces I*' where O is the name of an operation defined in M and I is the name of a constraint also defined in M.

5 HI is a possibly empty set of *hiding clauses* of the form '*O may violate* $I_1,...,I_k$' where *O* is the name of an operation of M_i, for some *i* in [1,*n*], and I_j is the name of a constraint defined in CN_0, for each *j* in [1,*k*]. We say that *O* is *hidden* by *M*.

Each object of M_i 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 $RS_0,...,RS_n$.

2 CN is the union of $CN_0,...,CN_n$.

3 OP is the union of $OP_0,OP_{1'},...,OP_{n'}$ where $OP_{i'}$ is OP_i 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 $M_1,...,M_n$ if each relation scheme of *M* is a *view* over the relation schemes of $M_1,...,M_n$ (that is a relation scheme derived from those of $M_1,...,M_n$) and each constraint of *M* is a logical consequence of those of $M_1,...,M_n$, when views are treated as defined predicate symbols. *M* may also introduce operations on views. But, to avoid the so-called view update problems *(TUC28–TUC30)*, the definition of *M* contains, for each view operation *p*, an implementation of *p* in terms of the operations of $M_1,...,M_n$. Unlike subsumption, modules $M_1,...,M_n$ remain accessible after the definition of *M*.

A new module *M* is defined by *extension* over $M_1,...,M_n$ 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 (RS_0,CN_0,OP_0) defines a module *M* as previously described.

2 VW contains, for each scheme $R[A_1,...,A_k]$ in RS_0, a *view definition mapping* of the form $R(x_1,...,x_k)$: Q, where Q is a well-formed formula with *k* free variables, ordered $x_1,...,x_k$, over $RS_1,...,RS_n$. The well-formed formula '$\forall x_1...\forall x_k(R(x_1,...,x_k)\Leftrightarrow Q)$' is called a *view definition axiom.*

3 SR contains, for each operation $f(y_1,...,y_m)$:*r* in OP_0, a *surrogate*, which is an operation of the form $f(y_1,...,y_m)$:*s* over $RS_1,...,RS_n$.

The statement in (3) then defines a new module $M = (RS_0,CN_0,OP_0)$ and couples *M* to $M_1,...,M_n$ through the pair (VW,SR). A view definition mapping $R[A_1,...,A_k]$:*Q* in VW indicates that *Q* defines *R* in terms of the relation schemes of $M_1,...,M_n$. Hence, a query over *R* is translated into a query over the relation schemes of $M_1,...,M_n$ with the help of *Q*. Likewise, a surrogate $f(y_1,...,y_m)$:*s* in SR describes an implementation of $f(y_1,...,y_m)$:*r* in terms of $M_1,...,M_n$. 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
       M1;...;Ms
endschema
```

where $M_1,...,M_s$ 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/X3/SPARC architecture *(TUC31)*, the conceptual schema corresponds to a forest of primitive and subsumption modules. The roots of the forest are the active modules (underlined in Figure 1). The external schemata correspond to external modules, which form a partial order stemming from the active conceptual schema modules.

An example

The authors' method will be illustrated by designing a micro database to store information about products, warehouses and shipments to warehouses. Let us create a schema with one primitive module, PRODUCT, representing data about products and the operations allowed on products. It is defined as follows:

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

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

1 RS consists of the relation scheme PROD[P#,NAME].

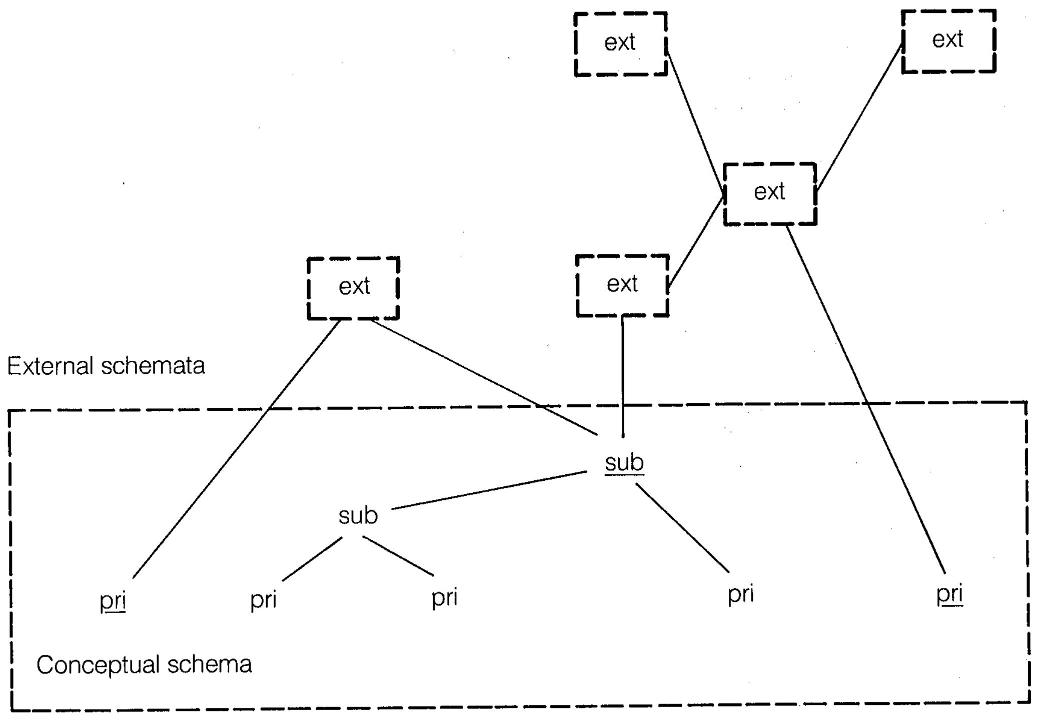


Figure 1: An example of modular architecture

2 CN contains, in addition to the well-formed formula listed after the *constraint* clause, the following relation scheme axiom:

$$\forall p \forall n\ (\mathrm{PROD}(p,n) \Rightarrow \mathrm{P\#}(p)\ \&\ \mathrm{NAME}(n))$$

3 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. Another primitive module, WAREHOUSE, is added to represent warehouses and the operations on warehouses. WAREHOUSE is defined as follows:

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

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

```
module SHIPMENT
  subsumes PRODUCT, WAREHOUSE with
  schemes SHIP[P#,W#,QTY]
  constraints
    ONE__Q:
      ∀p∀w∀q∀q′(SHIP(p,w,q) & SHIP(p,w,q′)
                  ⇒q = q′)
    INC__P:∀p(∃w∃q SHIP(p,w,q)
                ⇒∃n PROD(p,n))
    INC__W:∀w(∃p∃q SHIP(p,w,q)
                ⇒∃c WAREHSE(w,c))
  operations
    ADDSHIP(p,w,q):
      if ∃n PROD(p,n) & ∃c WAREHSE(w,c) &
        ¬∃q′ SHIP(p,w,q′) & QTY(q)
        then insert (p,w,q) into SHIP;
    CANSHIP(p,w):
      delete SHIP(x,y,z) where (x = p & y = w);
    CLOSE1(w):
      if ¬∃p∃q SHIP(p,w,q) then CLOSE(w);
    DELPROD1(p):
      if ¬∃w∃q SHIP(p,w,q) then DELPROD(p);
  enforcements
    ADDSHIP enforces ONE__Q, INC__P, INC__W;
    CLOSE1 enforces INC__W;
```

```
        DELPROD1 enforces INC__P;
      hiding
        DELPROD may violate INC__P;
        CLOSE may violate INC__W;
    endmodule
```

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

Finally, we introduce the module DELIVERY by extending SHIPMENT:

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

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

Design requirements, definition order and propagation rules

First, the design requirements for a correct modular database schema are discussed. Then the specification of the authors' modular design method is completed by presenting a definition order to create or change the specifications of objects of a modular schema. An explanation is then given of the strategy adopted for the propagation rules which determine how changes on one group of objects propagate to others.

The requirements

In order to present the requirements, a few concepts are introduced. 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 violates 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, 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 the schemes (views) and operations defined in M itself to the modules above it.

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

1 The operations of a newly defined module M preserve consistency with respect to the constraints of M.

2 The operations of M preserve consistency with respect to the constraints of modules previously defined.

3 The operations of previously defined modules, without change, preserve consistency of the constraints of M, except when defined by subsumption over M', when operations of M' may be hidden in M.

4 Schemes and operations are active, in the sense that for every scheme there is at least one operation performing insertions and every operation is either directly usable or is called by some usable 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 subroutine 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 *enforces* relationships of a module M, it is required that both the operation and the constraint involved be defined in M. For *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 and also that 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 stipulate that constraints be invariant with respect to operations (that is, any necessary tests should be effectively included in the critical operations) and that all *enforces* and *may-violate* statements will have been provided. They also stipulate 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 should correctly correspond to O (in the precise sense explained in *(TUC29)*).

A definition order over the sets of objects of modules

A *definition order*, denoted by $<$, on the sets of objects of the modules of the schema is introduced in this section. 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:

1 *Case 1 —object definition order:* let M be a module and, when applicable depending on the type of M, let RS,CN,OP,HI,EN,VW and SR be constants denoting the sets of schemes, constraints, operations, hiding statements, enforcement statements, view definitions and surrogates, respectively, defined in M. Then:

RS < VW
RS < CN, VW < CN
CN < HI
RS < OP, CN < OP, HI < OP
CN < EN, HI < EN, OP < EN
VW < SR, OP < SR.

2 *Case 2—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 $x' \in X'$, $x < x'$ if $X < X'$.

The propagation rules

The set of propagation rules is used to optimise 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 the authors' 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.

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 have become 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 signalled (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, 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 any more. On the other hand, if an operation O may no longer violate any constraint, therefore ceasing to be hidden, and an operation O' exists with the sole purpose of safely calling O, 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 available to perform insertions into some scheme, makes it necessary to create a new operation to assume that 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, and also in the case where they have been just created, the user is asked whether the relationship now holds.

The design tool

To support the specifications in the modular design method described in the previous two sections, the authors have built, and are further developing, a prototype expert system, written in micro-PROLOG *(TUC7)* extended with APES *(TUC8)*. To help develop the prototype, the specification of a database application is divided into the design and redesign phases. However, this separation should not be viewed as a characteristic of the modularisation method.

The design phase

For this phase the prototype has:

1 A program to schedule, according to the definition order, the creation of modules and their components.

2 A query-the-user facility (provided by APES).

3 A parser, for well-formed formulae of the first-order predicate calculus and operations of the formal programming language proposed by Casanova and Bernstein *(TUC32)*.

4 The requirements themselves, expressed in a declarative form.

5 A dictionary, produced as output from the dialogue with the user.

Figure 2 illustrates the design phase tool, showing its parts and how they are organised. Single arrows denote that one component activates the other and the double arrow denotes output. The requirements are applied, firstly, to restrict the answers given by the user and, secondly, to reprompt him for missing information. By obeying the requirements, one strives to achieve specifications that are correct by construction. The context-free syntax of logical formulae 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 (for example, the preconditions 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 *(TUC33)* and to check the correctness of the translation of view operations, a PLAN-GENERATOR prototype developed separately *(TUC34)* can be utilised, either before using the design prototype, or by interrupting a design session. This separate tool is also written in micro-PROLOG.

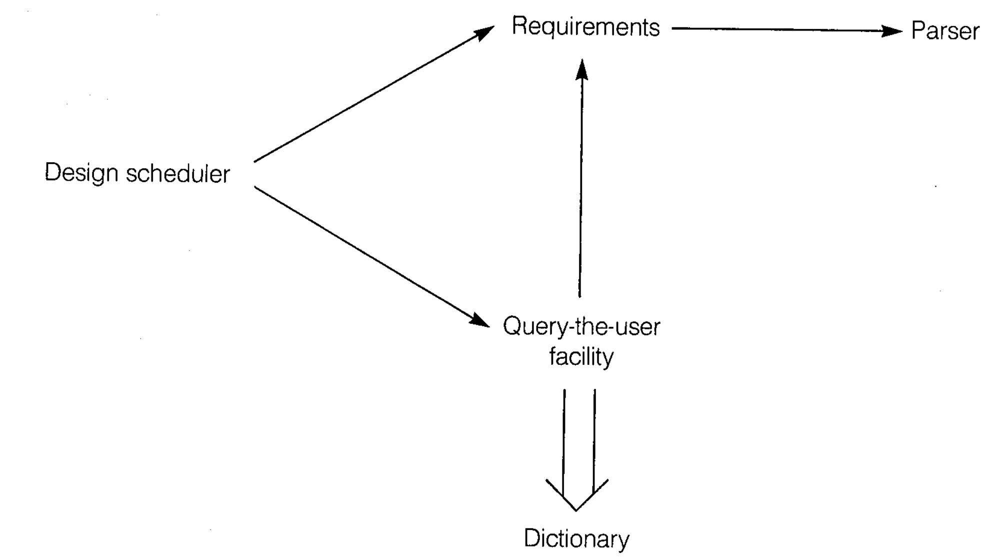


Figure 2: The design phase

The following is a more detailed explanation of how the design phase works. Appendix II contains examples.

1 The user enters:

module < name of module being created >

This invokes the 'module' predicate, which activates the query-the-user feature, which asks the user what type of module he is using (primitive, subsumption, external), enters this information in the dictionary (the 'tab' predicate) and then sets up, as subgoals, the creation of the other classes of component, according to the definition order. The 'module' predicate, together with the predicates it calls to achieve the subgoals, constitutes what is termed the scheduler.

2 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 (a module may have several schemes, constraints and operations) the user must enter '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.

3 The requirements that stipulate 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.

4 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 subexpressions 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.

5 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'.

The redesign phase

For this phase the prototype has:

1 Another appropriate scheduling program.

2 The query-the-user facility.

3 The parser (same as for the design phase).

4 The requirements (same as for the design phase).

5 The propagation rules, expressed in the same declarative style as the requirements.

6 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.

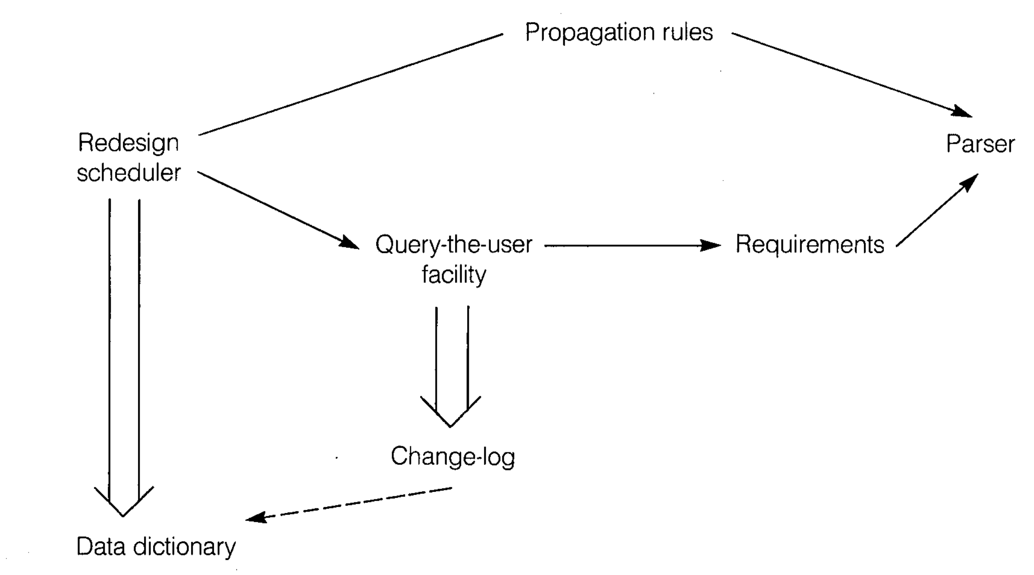


Figure 3: The redesign phase

7 The dictionary resulting from the design phase, which is updated to reflect the changes to the design.

Figure 3 displays the redesign phase tool. Single arrows denote activation, double arrows denote output and the dashed 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, which are in their turn changed.

The new formulation of an affected component, or (in the case where the user decides not to change it) its current formulation, is checked by applying the requirements, exactly as in the design phase. The user is again 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. One should 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 are 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 Appendix II):

1 The user enters either:

change < name of component >

if he wants 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 predicates—'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 it sets up as subgoals 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 predicates 'change', 'new' and 'propag', together with the predicates that the latter calls for the propagation in each class, constitute the scheduler for the re-design phase.

2 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.

3 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.

4 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.

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

6 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.

7 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 the prototype, as well as the plan-generation prototype (which the authors 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 *(TUC27)*), and to make it more usable and efficient.

Conclusions

This paper has described a modularisation 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, secondly, transforming the tool into a fully fledged dictionary system incorporating as much knowledge as possible about the complete language.

The authors envisage a final tool consisting of an interpreter for the final conceptual modelling language; that calls an expert helper, incorporating knowledge about the language to guide the DBA; which in turn uses a dictionary storing (versions of) several module database schemata; and a compiler mapping dictionary entries into DDL/DML commands of the underlying database management systems.

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.

Requirement 1 reflects the fundamental preoccupation that the database should always be left in a consistent state *(TUC35)*.

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

Requirement 2 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 $M_i = (RS_i, CN_i, OP_i)$, $i = 1,...,n$. Let RS_0, CN_0, OP_0 and HI be the new relation schemes, integrity constraints, operations and hidden operations, respectively, defined in M. Let CN be the union of $CN_0,...,CN_n$ and OP be the union of $OP_0, OP_{1'},...,OP_{n'}$, where $OP_{i'}$ is the set OP_i, 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 CN_0.

Requirement 4: each operation in OP_0 can only modify the values of relation schemes in $M_1,...M_n$ through calls to the operations defined in $M_1,...M_n$.

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.

Requirement 5 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 OP_0 such that O' calls O.

Requirement 6 guarantees that operations that are hidden in M_k do not become inactive.

Requirement 7: schema 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 ensure that calls to operations of M'' will not violate constraints of M.

Requirement 8: $M_1,...,M_n$ 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, and those of modules defined by subsuming M, preserve consistency.

Modules defined by extension

Let M be a module defined by extension over modules $M_i = (RS_i, CN_i, OP_i)$, $i = 1,...,n$. Let RS_0, CN_0, OP_0, VW and SR be the new relation schemes, integrity constraints, operations, view definitions and surrogates, respectively, defined in M.

Requirement 9: if $f(y_1,...y_m)$:s is the surrogate of $f(y_1,...,y_m)$:r defined in SR then s is a faithful translation of r *(TUC29)*.

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

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

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

Requirement 11: for each integrity constraint I in CN_0, I' must be a logical consequence of the integrity constraints of $M_1,...,M_n$, where I' is obtained from I by replacing each atomic formula of the form $R(t_1,...,t_k)$ by $Q[t_1/x_1,...,t_k/x_k]$, where $R[A_1,...,A_k]$:Q is the view definition of R described in VW, and the list of free variables of Q is $x_1,...,x_k$.

Requirement 11 guarantees that the integrity constraints of M follow from those of $M_1,...,M_n$ 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: $M_1,...,M_n$ 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

Design examples

This section shows how to use the design tool to create the modules previously defined. After the module PRODUCT has been defined, the state of the dictionary is exhibited to illustrate its structure. In the creation of the module SHIPMENT, an error was forced by failing to include an operation to call DELPROD, although this operation is hidden in SHIPMENT since it may violate INC-P. 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  (∀p∀n∀m(prod(p,n) &
                            prod(p,m) ⇒ n = m)))
      Answer is   enough

* operations --- (<name> ((<parameters>) : <bcdy>)) ?
      Answer is   (addprod  ((p,n) :
                              if ¬∃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)
    (∀p∀n∀m(prod(p,n) & prod(p,m) ⇒ n = m)))
tab (operation (addprod product)
    (((p,n) : if ¬∃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-q  (∀p∀w∀q∀r(ship(p,w,q) &
                          ship(p,w,r) ⇒ q = r)))
      Answer is  (inc-q  (∀p(∃w∃q(ship(p,w,q)) ⇒
                          ∃n(prod(p,n)))))
      Answer is  (inc-w  (∀w(∃p∃q(ship(p,w,q)) ⇒
                          ∃c(warehse(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(warehse(w,c)) &
                               ¬∃r(ship(p,w,r)
                               then insert (p,w,q) into ship))
      Answer is  (canship  ((p,w) :
                            delete ship(v1,v2,v3)
                                  where (v1 = p & v2 = w)))
      Answer is  (close1   ((w) :
                            if ¬∃p∃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  (delprod1  ((p) :
                             if ¬∃w∃q(ship(p,w,q))
                                then delprod(p)))
      Answer is  enough

* addship  enforces one-q ?  yes
* addship  enforces inc-p ?  yes
* addship  enforces inc-w ?  yes
* canship   enforces one-q ?  no
* canship   enforces inc-p  ?  no
* canship   enforces inc-w  ?  no
* close1    enforces one-q ?  no
* close1    enforces inc-p  ?  no
* close1    enforces inc-w  ?  yes
* delprod1  enforces one-q ?  no
* delprod1  enforces inc-p  ?  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) : ∃q(ship(p,w,q)))
    Answer is  enough

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

* operations --- (<name>  ((<parameters>) :  <body>)
                          ((<parameters>) :  <surrogate>)) ?
      Answer is  (del  ((p,w) : delete delvry(v1,v2)
                                where (v1 = p & v2 = w))
                       ((p,w) : canship(p,w)))
      Answer is  enough

*** Module delivery created
```

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

Redesign examples

Examples 4 and 5 assume that only PRODUCT has been created, illustrating how propagation occurs for a single module. In Example 4, PROD includes an additional domain whereas, in Example 5, two new schemes are inserted into PRODUCT. Example 6 assumes that all four modules have been created, and illustrates how changing an object in a module *M* may affect objects of modules defined above *M*. Operation DELPROD is deleted, a change that propagates to the operation DELPROD1 of the subsumption module SHIPMENT, since DELPROD1 calls DELPROD.

Example 4

```
change prod

* scheme prod
* (<change>  (<domains>)) ?
    Answer is  (MOD  (pnum,name,weight))
```

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

* 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   ((~~∀p∀n∀m∀w∀v~~v(prod(p,n,w) & prod(p,m,v)
                             ⇒ n = m & w = v)))

* 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 ¬∃m∃v(prod(p,m,v))
                                     then insert (p,n,w)
                                                 into prod))

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

* should delprod contain some test necessary to guarantee that one-n is preserved ?   no

* operation delprod
  ( < change >  (( < parameters > ) :  < body > )) ?
         Answer is   (MOD   ((p)      : delete prod(v1,v2,v3)
                                        where v1 = p))

* does addprod now contain some test to guarantee that one-n is preserved ?   yes

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

*** 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 delprod1 in shipment since delprod1 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.