

A FRAMEWORK FOR DESIGN/REDESIGN EXPERTS

A. L. Furtado \*, M. A. Casanova \*\*, L. Tucheran \*\*

\* Pontificia Universidade Catolica do R. J.

\*\* IBM Brasil

ABSTRACT

A software tool that helps the specification of data base applications at the conceptual level is first discussed. The tool is based on a modularization method, also briefly summarized. Then, the major features of the tool are analysed and expanded into a framework for the construction of tools with similar goals. The framework consists of an appropriate characterization of the context where such tools act, and suggests that knowledge about the conceptual design method adopted should be expressed declaratively as sets of rules.

1. Introduction

The usefulness of tools to help in the design of information systems based on data bases is widely recognized. Among other reasons, they effectively place formal design methods within the reach of practitioners.

On the other hand, expert systems, incorporating the knowledge of human experts in some area, are being applied to solve a large number of problems. As one would expect, the combination of the two concepts - expert tools - is an active research area [BMG,LM].

The knowledge incorporated to an expert tool should, above all, reflect a method for designing and re-designing data bases. If the method can be expressed by rules, preferably in a declarative style, then we can conceive an expert tool that is driven by the rules and that can have its expertise progressively increased by the continuous revision and addition of new rules, corresponding to more refined versions of the method.

This paper investigates what is a minimal framework for such tools. The framework should be "parametric" in the sense of being able to accomodate different methods, essentially by changing the sets of rules.

To make the discussion more concrete, we first introduce a particular method in Section 2 and then a prototype tool that implements the method and is now fully operational in Section 3. Finally, we characterize the desired framework in Section 4 by generalizing the tool, and we show that the framework is adequate in widely different contexts. Section 5 contains the conclusions and the appendix provides examples of the use of the prototype tool.

2. The Method

2.1 The Modular Architecture

Adopting the relational data model, we consider data base applications involving:

- relational schemes (tables)
- integrity constraints
- operations

The integrity constraints establish which data base states are valid (static constraints) and which transitions between states are valid (transition constraints). A convenient strategy to keep the data base valid, especially with respect to the static constraints, consists of imposing pre-conditions to the operations regarded as critical - such operations would therefore have their effects dependent upon appro-

ropriate tests, included in the body of their respective procedures. To incorporate this strategy, we add to the three components above the following relationship:

- operation enforces constraint

The enforces relationship provides useful documentation tying together the declarative semantics of constraints to the procedural semantics of operations.

The specification of an application is often large and complex, which suggests the use of some divide-and-conquer strategy. We then briefly summarize in what follows a modularization method, fully described in [TFC]. The method uses three types of modules: primitive, subsumption and external.

To introduce new relation schemes, integrity constraints and operations that have no connections with the objects of the current data base schema, the data base administrator (DBA) may choose to define a new primitive module M. The DBA must primarily guarantee that each operation defined in M preserves consistency with respect to the constraints of M. He must include an enforcement relationship "O enforces I" whenever some change in the definition of I implies that the definition of O must also be changed.

If the DBA defines a subsumption module M over modules  $M_1, \dots, M_n$ , then he may list new objects in M exactly as if M were a primitive module. However, M also automatically inherits all the objects of  $M_1, \dots, M_n$ , except that the DBA may hide some of the operations of  $M_1, \dots, M_n$ , if they violate a new constraint. This motivates the introduction of another relationship:

- operation may-violate constraint

Whenever we have that an operation O of module  $M_i$ ,  $1 < i < n$ , may violate constraints  $I_1, \dots, I_k$  of M, operation O cannot be directly called by users anymore (i.e., O becomes hidden). However, O may be <sup>k</sup> called by other operations defined in M, that must have tests adequate to enforce  $I_1, \dots, I_k$ .

The DBA should define an external module M over  $M_1, \dots, M_n$  when he wants to introduce views and operations on views. A view must be introduced with a view definition mapping, and each operation must come with a surrogate, which translates it into operations of the extended modules. Any constraint of M must be a logical consequence of the constraints of the extended modules.

In terms of the ANSI/X3/SPARC architecture [ANSI], the conceptual schema corresponds to a forest of primitive and subsumption modules. The roots of the forest are the active modules (underlined in figure 1). The external schemata correspond to the external modules, which form a partial order stemming from the active conceptual scheme modules.

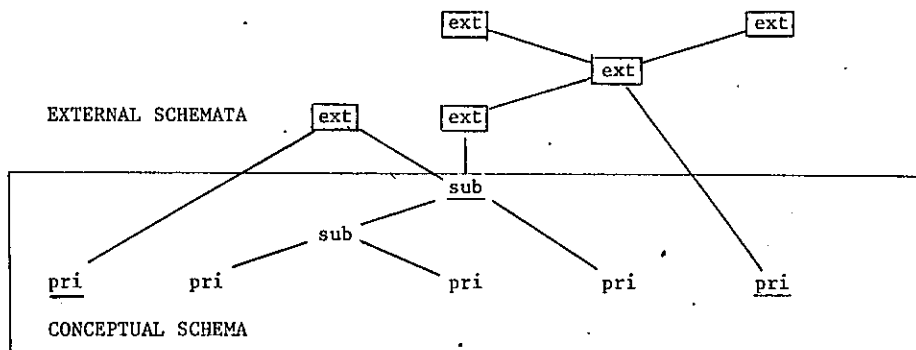


Fig. 1 - Example of Modular Architecture

In addition to the module constructors, the method is characterized by a set of design requirements, a set of propagation rules, discussed in Sections 2.2 and 2.3, respectively, and a natural order to create or change a specification. The order was decided on the principle that, if a concept x needs in any way another concept y for its definition, then it is "natural" to place y before x. At the level of modules, if module M subsumes or extends M', then M' must be defined before M. At the component level, the order is: 1. schemes, 2. constraints, 3. may-violate, 4. operations, 5. enforces. This is because constraints and operations are

defined over schemes, and operations may need pre-conditions to enforce constraints. Finally, recall that may-violate relationships hold in subsumption modules, associating with their constraints operations of the modules subsumed, whereas enforces relationships associate operations and constraints of the same module.

The concept of an ordering for the classes of objects turned out to be very useful for the correct structuring of the design/redesign tool, as well as for the definition of a strategy for testing the design requirements and creating propagation rules.

## 2.2 The Requirements

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

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

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

- the operations of a newly defined module M preserves consistency with respect to the constraints of M;
- the operations of M preserves consistency with respect to the constraints of modules previously defined;
- the operations of previously defined modules, without any change, preserve consistency with respect to the constraints of M, except if is defined by subsumption over M', when operations of M' may be hidden in M;
- schemes and operations are active, in the sense that for every scheme there is at least one operation performing insertions into it, and that every operation is either directly useable or is called by some useable operation.

Requirements are classified either as syntactical or semantic. The syntactical requirements allow integrity constraints and operations defined in a module M to reference any scheme accessible to M. However, only schemes defined in M can be directly updated by the operations of M; to update imported schemes defined, say, at a module M', an operation of M must call as a sub-routine the appropriate operation defined in M'. Domains, of course, can never be updated. If M is an extension 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 the may-violate relationship, the operation must be accessible to (but not defined in) M and the constraint must be defined in M.

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

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

## 3.3 The Propagation Rules

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

which is obeyed both for design and redesign, ensures that propagation of changes can only occur in a forward direction (i.e. if a component C of a module M is changed, the change can only affect components created after C either in M or in modules following M along the subsumption and extension paths).

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

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

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

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

Finally, the modification or deletion of operations affects the operations and surrogates that call them, and the deletion of an operation that is the only one to perform insertions into some scheme makes it necessary to create a new operation to assume this task.

The automatic propagation rules establish that an enforces or may-violate relationship holding between an operation O and a constraint C should be deleted if O or C are deleted; if O or C are simply modified, the user is asked whether or not the relationship still holds. If there was no such relationship between O and C and one or both are modified, or still in the case where they have been just created, the user is asked whether the relationship now holds.

### 3. The Prototype

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

#### 3.1 The Design Phase

For this phase the prototype has:

- a) a program to schedule, according to the natural order, the creation of modules and their components;
- b) a query-the-user facility (provided by APES);
- c) a parser, for wffs of the first-order predicate calculus and operations of the formal programming language proposed in [CB];
- d) the requirements themselves, expressed in a declarative form;
- e) 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 organized. Simple arrows denote that one part activates the other and the double arrow denotes output.

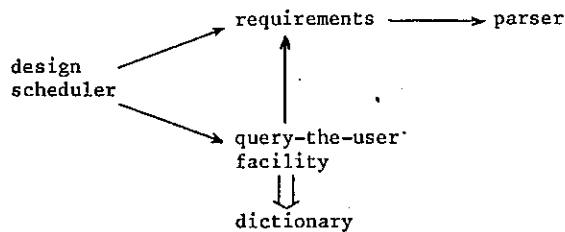


Fig. 2 - The Design Phase

The requirements are applied firstly to restrict the answers given by the user, and then to prompt him again for missing information. By obeying the requirements, one strives to achieve specifications that are correct by construction. The context-free syntax of logical formulas and operations is immediately checked by the parser. The context-sensitive syntax (imposing for instance that operations introduced in a module M can only directly update schemes created in M) is part of the requirements and is checked over the parse tree produced by the parser. However, the semantic requirements (e.g., the 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 [VF], and to check the correctness of the translation of view operations, a PLAN-GENERATOR prototype developed separately [FM] can be utilized, either before using the design prototype, or by interrupting a design session. This separate tool is also written in micro-PROLOG.

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

i) The user enters:

module [name of module being created]

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

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

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

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

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

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

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

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

### 3.2 The Redesign Phase

For this phase the prototype contains:

- a) another appropriate scheduling program;
- b) the query-the-user facility;
- c) the parser (same as for the design phase);
- d) the requirements (same as for the design phase);
- e) the propagation rules, expressed in the same declarative style as the requirements;
- f) a change log, produced from the dialogue with the user as an intermediate output, that marks which components have been changed and the nature of the change;
- g) the dictionary resulting from the design phase, which is updated to reflect the changes to the design.

Figure 3 displays the redesign phase tool. Simple arrows denote activation, double arrows denote output and the dotted arrow expresses the fact that the contents of the change log are used to produce the final output, namely, the new version of the dictionary.

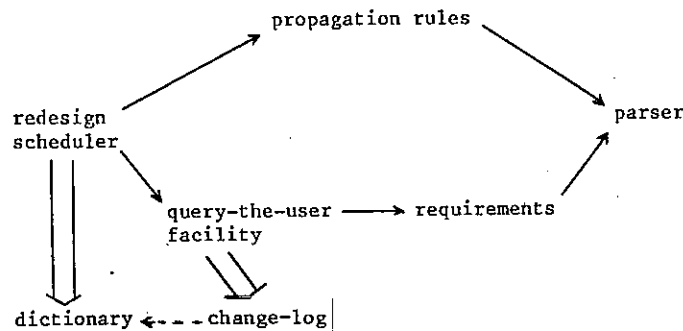


Fig. 3 - The Redesign Phase

The scheduling program follows the same natural order. The user indicates the deletion or modification of some scheme, constraint or operation. The insertion of various components of one of these three classes into the same module is also supported. Besides processing the change, the prototype leads the user to examine all other components that may be affected by the change, a propagation process that continues from the affected components that are in their turn changed.

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

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

As a consequence of the natural order, when each affected component is reached, all possible propagation lines leading to it will have been exploited. Thus, if a scheme and a constraint were modified and both

modifications affect an operation, the consequences of both will be available when the moment to consider the operation arrives.

In more detail, the redesign phase works as follows (examples are provided in the appendix):

i) The user enters either:

change [name of component]

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

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

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

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

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

vi) The redesign phase works on the data dictionary. It yields, as an intermediate result, a log of the changes validated with respect to the requirements. Each change registered in the 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 natural order.

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

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

#### 4. The Framework

We are now in a position to show how a framework for design/redesign expert systems can be characterized through generalizations of the tool described in the previous section.

The current version of the tool works in a context characterized by:

<u>scope</u>	the scope has to do with the specification of data bases, that is, with the definition of sets of objects
<u>level</u>	the specification is done at the conceptual and external schema levels.
<u>data model</u>	the adoption of the relational model determined to what classes the sets of objects belong
<u>language</u>	logical expressions are written in a first-order language and operations in a regular language, both conforming to the relational model
<u>method</u>	the method is formulated, as discussed, in terms of a natural order, a set of requirements and a set of propagation rules

To accomplish its task the tool has a number of features organized as shown in Figures 2 and 3. But the same type of features with the same organization - i.e., the same framework - can work in many different contexts.

First of all, to extend or refine the method one simply adds further requirements and propagation rules. To use a different method, the requirements and propagation rules would be replaced. If the natural order is also revised, the scheduling programs must invoke in a different order the predicates to create the various components.

Changing the language implies expanding or rewriting the parser. Adding new components or changing the choice of components corresponds, respectively, to extending the model or moving to a different one, normally associated with different languages.

By changing the level we may cope with physical design (the internal schema level, in the ANSI/X3/SPARC terminology). Alternatively, one might expand the present method to define new types of modules "below" those of the conceptual schema, where files would be one of the component classes. Schemes would then be mapped downward into files, as they are already mapped upward into views.

The most drastic change refers to scope. Instead of addressing data base design, we might address data base manipulation. Yet the analogy between data bases and data dictionaries ("meta" data bases) is well-understood. If we replace the dictionary by the data base itself as main output of the tool, and the design requirements by ordinary integrity constraints, then the design phase can be compared to loading, i.e., creating the initial state of a data base, simultaneously checking its consistency. Similarly, the redesign phase is converted into data base update, again with consistency checking, and enhanced with the capability to perform other corrective updates (by replacing the propagation rules by generation rules [NY]).

The above discussion, together with a clear separation and loose coupling between the various features of the tool, argue for the generality of the tool. This separation is indeed carried into the internal organization of the features.

The use of PROLOG is not a mandatory characteristic of the framework, but the power and flexibility of the language have certainly proved to be a convenience. Although everything in the tool is uniformly expressed by PROLOG sentences, the orientation differs for each part: requirements and propagation rules are declarative, the two schedulers are procedural, the parser is production rule oriented, the query-the-user features are cases of pattern-matching invocation and the dictionary and the change log are factual, being treated as data.

By combining two or more existing parts and possibly adding new ones, under the PROLOG interpreter, one can perform other useful tasks. Two examples deserve mention:

- If instead of doing the design under the guidance of the tool, we receive an independently prepared design in the form of a dictionary, the requirement sentences can be used together with the parser to check the design.
- A dictionary language, using the parser, can formulate recursive queries such as: "which operations perform deletions on which schemes or views either directly or indirectly (by operation calls)?" Since all items of the dictionary entries are equally arguments of the "tab" predicate, one can ask even the class of a component, for instance to learn that, say, C is a constraint.

## 5. Conclusion

In retrospect, we note that the distribution of the workload between user and expert tool is a critical aspect in the proposed framework. This is acutely felt in the prototype tool here described as an instance of the framework. Although the implemented rules guide to some extent the user's design/redesign efforts, much remains open to him, not only in terms of choosing between valid alternatives (which is desirable) but also of making errors (which is not), especially in connection with semantic requirements.

This seems to be a problem inherent in expert systems, since "expertise" is limited to what can be usefully transferred from human agents to automatic processors. Considerations such as decidability, complexity and efficiency - sometimes absolute and sometimes related to the present state of the art - influence this compromise, which continuing research tries to make more favorable to users.

#### REFERENCES

- [ANSI] "Study Group on Data Base Management Systems: Interim Report" - FDT 7:2, ACM (1975).
- [BGM] M. Bouzeghoub, G. Gardarin, E. Metais - "Database design tools: an expert system approach" - Proc. of the 11<sup>th</sup> Int'l. Conf. on Very Large Data Bases (1985), 436-447.
- [Bo] A. Borgida, "Features of Languages for the development of information systems at the conceptual level" - IEEE Software (Jan. 1985), 63-72.
- [CB] M.A. Casanova, P.A. Bernstein - "A formal system for reasoning about programs accessing a relational database" - ACM Trans. on Programming Languages and Systems 2:3 (1980), 386-414.
- [CM] K.L. Clark, F.G. McCabe - "micro-PROLOG: programming in logic" - Prentice-Hall (1984).
- [FC] A.L. Furtado, M.A. Casanova - "Updating relational views" in Query Processing in Database Systems, W. Kim, D.S. Reiner, D.S. Batory (eds.) - Springer-Verlag (1985), 127-142.
- [FM] A.L. Furtado, C.M.O. Moura - "Expert helpers to data based information systems" - Proc. of the First Int'l. Workshop on Expert Database Systems (1984), 298-313.
- [HS] P. Hammond, M. Sergot - "apes: augmented PROLOG for expert systems - Reference Manual" - Logic Based Systems, Ltd. (1984).
- [LM] P. Lockemann, H.C. Mayr - "Information system design: techniques and software support" - Proc. of the IFIP World Computer Congress '86 - (to appear).
- [NY] J.-M. Nicolas, K. Yazdanian - "An outline of DBGEN: a deductive DBMS" in Information Processing 83, R.E.A. Mason (ed.) - North Holland (1983), 711-717.
- [TFC] L. Tucherman, A.L. Furtado, M.A. Casanova - "A tool for modular database design" - Proc. of the 11<sup>th</sup> Int'l. Conf. on Very Large Data Bases (1985), 436-447.
- [VF] P.A.S. Veloso, A.L. Furtado - "Towards simpler and yet complete formal specifications" in Information Systems: Theoretical and Formal Aspects, A. Sernadas, J. Bubenko, A. Olive (eds.) - North-Holland (1985), 175-189.

#### APPENDIX

##### Design and Redesign Examples

##### A. Design Examples

Examples 1 and 2 show how to use the design tool to create two primitive modules, PRODUCT and WAREHOUSE. Example 1 also illustrates the structure of the dictionary. Example 3 exhibits a subsumption module, SHIPMENT, defined over PRODUCT and WAREHOUSE, and illustrates how the plan-generator (described in [FM]) can help the user find out that operation DELPROD of the module PRODUCT may violate constraint INC-P. It also shows a contrived error, forced by failing to include an operation to call DELPROD, which is necessary since DELPROD is hidden in SHIPMENT because it may violate INC-P (the tool then asks that some operation that calls DELPROD be provided). Finally, Example 4 illustrates the creation of an external module, DELIVERY, defined over SHIPMENT.

The examples reproduce the exact output of the tool.

### Example 1

```
module product

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

* schemes --- <<name> <<domains>> ?
  Answer is (prod (pnum,name))
  Answer is enough

* constraints --- <<name> <<definition>> ?
  Answer is (one-n ( $\forall p \forall n \forall m$ (prod(p,n) &
    prod(p,m) => n = m)))
  Answer is enough

* operations --- <<name> <<parameters>> : <body>> ?
  Answer is (addprod ((p,n) :
    if  $\exists m$ (prod(p,m))
    then insert (p,n) into prod))
  Answer is (delprod ((p) :
    delete prod(v1,v2)
    where v1 = p ))
  Answer is enough

* addprod enforces one-n ? yes
* delprod enforces one-n ? no
*** Module product created
```

### State of the Dictionary:

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

### Example 2

```
module warehouse

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

* schemes --- <<name> <<domains>> ?
  Answer is (warehse (wnum,loc))
  Answer is enough

* constraints --- <<name> <<definition>> ?
  Answer is (one-c ( $\forall w \forall c \forall d$ (warehse(w,c) &
    warehse(w,d) => c = d)))
  Answer is enough

* operations --- <<name> <<parameters>> : <body>> ?
  Answer is (open ((w,c) :
    if  $\exists d$ (warehse(w,d))
    then insert (w,c) into warehse))
  Answer is (close ((w) :
    delete warehse(v1,v2)
    where v1 = w ))
  Answer is enough
```

\* open enforces one-c ? yes

\* close enforces one-c ? no

\*\*\* Module warehouse created

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

### Example 3

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 ( $\forall p \forall w \forall q \forall r$  (ship(p,w,q) &  
ship(p,w,r) => q = r)))  
Answer is (inc-q ( $\forall p$  ( $\exists w \exists q$  (ship(p,w,q) =>  
 $\exists n$  (prod(p,n))))))  
Answer is (inc-w ( $\forall w$  ( $\exists p \exists q$  (ship(p,w,q) =>  
 $\exists c$  (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 ?

A pause! Assume that the user is in doubt about the correct answer to the query above and decides to resort to the plan-generator. He asks the plan-generator to form one or more sequences of operation executions that can lead to a state where INC-P is violated. At such state there would be a SHIP tuple without the corresponding PROD tuple, required by the inclusion dependency expressed by INC-P.

The user would then enter:

state((ship x y z)(not prod x X))

to which the plan-generator replies:

(s0 ; addprod p n ; open w l ; addship p w q ; delprod p)  
(s0 ; open w l ; addprod p n ; addship p w q ; delprod p)

Note that there are two sequences because the pre-requisites allow executions of ADDPROD and OPEN to commute (or to be done in parallel).

Clearly the problem is that DELPROD may be executed after ADDSHIP creates a SHIP tuple, which implies that DELPROD may indeed violate INC-P.

(Continuation of the output of the tool for Example 3)

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

* operations --- (<name> ((parameters)) : <body>)) ?
  Answer is (addship ((p,w,q) :
                if ∃n(prod(p,n)) &
                ∃c(warehouse(w,c)) &
                ¬∃r(ship(p,w,r))
                then insert (p,w,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 4

module delivery

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

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

(Continuation of the output of the tool for Example 4)

```
* views --- (<name> (<domains>
              ((<parameters>):(<definition>))) ?
  Answer is (delvry (pnum,wnum)
            ((p,w) : tq(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.
```

## B: Redesign Examples

Examples 5 and 6 assume that only the module PRODUCT has been created, thus illustrating how propagation occurs for a single module. In Example 5, the scheme PROD is modified to include an additional domain, whereas, in Example 6, two new schemes are inserted into PRODUCT. Examples 7 and 8 assume that all four modules have been created, and thus illustrate how changing an object in a module M may affect objects of modules defined above M. In Example 7, operation DELPROD is deleted, a change that propagates to the operation DELPROD1 of the subsumption module SHIPMENT, since DELPROD1 calls DELPROD. In Example 8, operation CANSHIP is modified, a change that may in principle propagate to the operation DEL, since DEL calls CANSHIP, but which does not occur since the modification on CANSHIP refers to a domain (QTY - quantity) that is not imported by the external module DELIVERY. Hence, this example illustrates a case of a propagation where the user legitimately chooses not to change the affected object.

### Example 5

```
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 (( $\forall p \forall n \forall m \forall w \forall v$ (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
```

(Continuation of the output of the tool for Example 5)

```
* operation addprod
  ((change) ((parameters)) : (body))) ?
  Answer is (MOD ((p,n,w) : if  $\exists m \exists v(\text{prod}(p,m,v))$ 
                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 6

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 7

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.

### Example 8

change canship

\* operation canship

((change) ((parameters)) : (body))) ?  
Answer is (MOD ((p,w) :  
delete ship(v1,v2,v3)  
where v1 = p & v2 = w & v3 < 100))

\* does canship now contain some test to guarantee that one-q  
is preserved ? no

\* does canship now contain some test to guarantee that inc-p  
is preserved ? no

\* does canship now contain some test to guarantee that inc-w  
is preserved ? no

\* delete or modify del in delivery since the surrogate of del  
calls an operation canship that was modified

\* operation del

((change) ((parameters)) :  
(body))((parameters):(surrogate))) ?  
Answer is (SAME () ())

\*\*\* change and propagation completed

Effects on the dictionary: the entry for canship is  
modified, no other change being done.

