VOOD :Un modele de donnees oriente-objet

G. Barbedette, Patrick Richard

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 580

VOOD:
UN MODÈLE DE DONNÉES ORIENTÉ - OBJET

Gilles BARBEDETTE
Philippe RICHARD

Novembre 1986

[McCA 65]

"LISP 1.5 Programmer's manual," J. McCarthy, et al., 2nd ed. Cambridge, Mass.: the MIT press, 1965.

[McLE 85] "Object Managementt and Sharing in Autonomous, Distributed Data/Knowledge Bases," D, McLeod, S. Widjojo, In [DBE 85]

[NEUH 85]

"Objects And Abstract Data Types In Information Systems," E. J. Neuhold In Proc. of the IFIP TC2 Working Conference on Database Semantics, R. Meersman, T. B. Steel (Eds.), Hasselt, Belgium, Jan. 1985, North Holland.

[NIER 85]

"An Object-oriented Environment for OIS applications," O. M. Niertrasz, D. C. Tsichritzis, In Proc of the Eleventh International Conference on Very Large Data Bases, Stockholm, August, 1985.

[PIST 85]

"A Database Language for Sets, Lists, and Tables," IBM Wiss. Zentr. Heidelberg , Technical Report, TR 85.10.004, Oct 1985.

[SCHA 86]

"AN Introduction to Trellis/Owl," C. Schaffert et al. In Proc. of OOPSLA 86 Portland, Oregon, 1986. [SCHE 82]

"Data Structures for an Integrated Database Management and Information Retrieval System," H. J. Scheck, P. Pistor, In Proc of the Eighth International Conference on Very Large Data Bases, 1982.

[STON 76]

"The Design and Implementation of Ingres," M. Stonebraker, et al. , Transactions on Data Base Systems, 2, 3, September 1976.

[STON 85a]

"Inclusion of New Types in Relational Data Base Systems," M. Stonebraker, Electronics Research Laboratories, College of Engineering, U of California, Berkeley, Memorandum No. UCB/ERL M85/ 67.

[STON 85b]

"Extending a Data Base System with Procedures," M. Stonebraker, Electronics Research Laboratories, College of Engineering, U of California, Berkeley, Memorandum No. UCB/ERL M85/ 59.

[TSIC 85]

"Object Species," D. Tsichritzis, In [DBE 85]

[TSUR 84]

"On the Implementation of GEM - supporting a semantic Data Model on a relational Backend," S. Tsur, C. Zaniolo, In Proc. of ACM-SIGMOD Conference on Management of Data, 1984.

[ZANI 85]

[CROFT 85]

"Task Management for an Intelligent Interface," W. B. Croft, In [DBE 85]

[DADA 84]

"Integration of Time Version into a Relational Database System," P. Dadam, V. Lum, H-D. Werner, In Proc of the Tenth International Conference on Very Large Data Bases, Singapore, August, 1984.

[DAYA 85]

"PROBE - A Research Project in Knowledge-oriented Database Systems : Preliminary Analysis," U. Dayal, Et Al., Technical Report, CCA-85-03, July 1985.

[DBE 85]

"A Quaterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering," Special issue on Object-Oriented Systems, Vol. 8, N.4, December, 1985.

[DERR 85]

"Some Aspects of Operations in an Object-Oriented Database," N. Derrett, W. Kent, P. Lyngbaek, In [DBE 85]

[GOLD 83]

"SMALLTALK-80 The Language and its Implementation," A. Goldberg, D. Robson, Addison-Wesleyy, Reading, MA, 1983.

[GOLDM 85]

"ISIS : Interface for a Semantic Information System," K. J. Goldman, S. A. Goldman, P. C. Kanellakis, S. B. Zdonik, In Proc. of ACM-SIGMOD Conference on Management of Data,Austin, Texas, May 28-31, 1985.

[KING 85]

"A Database Design Methodology and Tool for Information System," R. King, D. McLeod, ACM Transactions on Office Information Systems, Vol. 3, No. 1, Jan 1985, pp 2-21

[KRIS 81] "A Survey of the BETA Programming Language," B. B. Kristenssen and al., Norwegian Computing Center Oslo, Norway, 1981.

[LAME 84a]

"Recursive Data Models For Non-Conventional Database Applications," W. Lamersdorf, In IEEE International Conference On Data Engineering, Los-Angeles, April 24-27th, 1984.

[LAME 84b]

"Language Support for Office Modelling," W. Lamesdorf, G. Müller, J. W. Schmidt, In Proc of the Tenth International Conference on Very Large Data Bases, Singapore, August, 1984.

[LYNG 84]

"A personnal Data Manager," P. Lyngbaek, D. McLeod, In Proc of the Tenth International Conference on Very Large Data Bases, Singapore, August, 1984.

[MAIE 85]

"Development of an Object-Oriented DBMS," D. Maier, A. Otis, A. Purdy, In [DBE 85]

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 580

VOOD:
UN MODÈLE DE DONNÉES ORIENTÉ - OBJET

Gilles BARBEDETTE
Philippe RICHARD

Novembre 1986

VOOD : The Verso Object-Oriented Data Model

VOOD : Un modèle de données orienté-objet

Gilles BARBEDETTE et Philippe RICHARD

Institut National de Recherche en Informatique et Automatique
78153 Le Chesnay, cedex
France

ABSTRACT :

This paper is devoted to the management of complex objects. New applications of data base such as office information systems, VLSI CAD, etc, manipulate complex data structures in a very dynamic environment. In this work, we decided to follow an object-oriented approach. The advantages of such an approach are numerous. Among them, is the notion of object which is used at both conceptual and interface level. Futhermore, there is no artificial separation between structural and operational knowledge. In our model, every entity is an *object* which is an instance of a *type*. The type formally defines the structure of its instances and the operations allowed on them. The dynamic structure associated to the type is the *class*. The class is the set of all the instances of a type. We show in this paper how such a model can be implemented on top of an existing DBMS. We also exhibit the advantages of using a non first normal form (N1NF) relational DBMS in order to implement and manipulate complex data structures.

RESUME :

Nous nous intéressons dans ce travail à la gestion de données structurées complexes. En effet, les nouvelles applications des bases de données (CAO, bureautique, etc...) manipulent des données complexes et ce dans un environnement trés dynamique. Nous avons suivi une approche orientée objet. Les avantages d'une telle démarche sont nombreux. Elle permet de supprimer la séparation artificielle entre la connaissance structurelle et opérationnelle. Elle permet de créer des structures plus évolutives. Tout objet est décrit par un type qui contient les opérateurs le manipulant. L'utilisateur n'accéde donc à un objet que via les opérateurs du type et ne peut le manipuler directement. Nous suggérons dans cet article une implantation de notre modèle sur un SGBD existant plutôt que de le programmer dans le cadre d'un langage orienté-objet. Nous montrons l'intérêt d'un SGBD gérant des structures hiérarchiques pour implanter les structures de notre modèle.

1. INTRODUCTION

Standard relational database management systems (RDBMS) only deal whith atomic objects which are stored in flat relations [CODD 71, STON 76]. In the recent years, there has been considerable interest in extending the relational model to allow the manipulation of more complex objects. One approach is to allow user-defined data types for columns in relations [STON 85a, 85b]. This presents the advantage of adding few changes to the relational model and thus to quickly implement prototypes. On the other hand, this provides systems that look a little "patched up". An other interesting approach is that of non first normal form relations [SCHE 82, ABIT 84, VERSO 86]. This approach offers an extension of the relational model to non atomic objects, but still has limitations. It does not solve all problems which are posed by new applications of databases such as CAD, office information systems, software engineering, etc [KING 85, LAME 84b]. These applications need to deal with complex objects whose semantics must be rich [ABIT 86, BAN 86, PIST 85]. Futhermore, the data involved is very evolutive and needs to model not only the structure of real world entities but also their behavior. Using Zdonik terminology [ZDON 85], we call Object Management System (OMS) a tool which provides functions:

_to deal with objects of arbitrary types. The structure and the semantic must be captured by the model. The semantic should be captured by static integrity constraints defined along with the structure and by dynamic constraints that control the evolution of the object.
_to allow modifications at every level of description: individual objects or type description that define them.
_to capture the semantic of very dynamic environments: traces of modifications on objects must be kept (versions management). The OMS must manage the execution of concurrent transactions and must automatically deal with side effects and maintain the integrity of each object.
_to integrate data manipulation and programming languages. Indeed, data bases need better interfaces with application programs and programming languages need to manage their data in a richer way [COPE 84a].

In order to encompass each of these requirements, we decided to incorporate ideas from object-oriented programming and database management systems.
This paper is organized as follows: section II presents our approach and justifies our choices. The third section formally describes the data model, section IV develops the data manipulation language. The fifth section presents the dynamic management of instances and lists open problems that we shall consider in the future. Finally, we expose in section VI how a first prototype of VOOD will be realized on the top of the VERSO DBMS [VERSO 86].

2. *OUR APPROACH*

Our goal is to design a system (named VOOD) managing complex objects. Our model posseses two important features of an object-oriented data model:the concept of entity and of data abstraction [DERR 85].
The main reasons for using this approach are :

_the ability to integrate in the same model complex data, tools and various applications (screen editor, compiler, DBMS, programming language...)
_the absence of artificial separation between structural and operational knowledge. That is, each object is associated to a type that contains both its structural description and all the operations that may be performed on it.

Futhermore as for Smalltalk [GOLD 83], our model provides a clear distinction between object identity and structural equivalence of entities. Indeed, two entities are identical if and only if they are represented via the same object. On the other side two entities may posses the same structure

(identical components) and not be the same object [COPE 84b].
A classical exemple occurs in VLSI design: two NAND gates in a chip may be have the same structure and physically distinct. This kind of distinction can not be provided by classical DBMS.
Our model is caracterized by the following points:

i) Every object is an *instance* of a type. The type may be system-defined or user-defined. Its structure and operations are unambigously defined.
ii) Every entity of the model is an *object*. It can thus be manipulated via the operations that are described in its type definition. The main advantage is that all components of VOOD are integrated in an unique flexible structure.
iii) A *type* is a descriptor (which may be hierarchical and recursive) that specifies the structure of its INSTANCES together with its allowed operations
iv) The dynamic structure associated to a type is the *class* : the class is the set of instances of a given type. In our model, with each type is associated the class of all its instances. For example, let PERS be a type that defines a tuple [NAME : string, FIRST-NAME : string, SEX : string, AGE : integer, ADRESSE : string], the class C.PERS is the current set of objects (in this case tuples) that are persons (i.e tuples instances of PERS) in the database. On C.PERS, one can perform set-operations (since a class is a set) or relational algebra operations if they are already defined for objects having structure set of tuples. In the same way, tuple operations or user defined operations associated to type PERS may be performed on each element of C.PERS.

Rather than implementing our model in an object-oriented environment such as Smalltalk or Bigtalk [COPE 84b], we suggest to use the existing VERSO DBMS developped at INRIA [VERSO 86] and to build our system on top of it. As shown in section VI, VERSO has powerfull features and would allow us to quickly develop a prototype.

3. *THE VOOD DATA MODEL*

3.1. *the notion of type*

Modelling complex data structures requires the notion of type [NEUH85]. Similarly to ENCORE [ZDON 85], a type posseses a set of user-defined operations, can inherit operations from its super-type and is associated to an unique class that contains as elements all its intances.
In VOOD, a type definition is as follows:

```
TYPE          <type_name>;
SUPER_TYPE    <supertype_name>;
{WHERE        <condition>;}
{STRUCTURE    <structure definition>;}
I_METHODS     <set of i_methods>;
END_TYPE;
```

The WHERE clause may be omitted. If not, <condition> is a Data Manipulation Language (DML) expression that represents a restriction on SUPER_TYPE objects (we see examples of such cases in the following section).
The structure of an object is recursively built from atoms and type generators (see section 3.2).

I_METHODS is a set of user-defined instance methods, which are used to manipulate each type instances. Methods are defined by DML constructors (see section IV for a definition of the DML). The class hierarchy is due to the SUPER_TYPE clause. If type T1 has T2 for super-type, then it inherits the structure and all the methods of T2. Each element of the class associated to T1 (C.T1) is associated to an element of C.T2 by an "ISA" relation.
In a first step, we restrict ourselves to a simple inheritance mechanism (with overhiding) similar to that of Smalltalk. The purpose of such a mechanism is to factorize structures and/or methods which are common to multiple classes.An example of such a definition is the case of the type PERS with sub-types MAN and WOMAN. In this case, the where condition is : WHERE sex = male (or female). Every MAN is an object of type PERS that is element of C.PERS and may be viewed as a MAN, i.e an element of C.MAN. Another example is the type PILOT where there is no WHERE clause but extra structural information.

3.2. System-defined types and the class hierarchy

In this section, we present the system defined objects and the class hierarchy.
The first type which is the most general, is the root of the hierarchy. It is described as follows:

```
TYPE          object;
SUPER_TYPE    object;
I_METHODS     { ==, type, elt_of, kill, print, perform, return};
END_TYPE;
```

This type only describes and contains basic methods on objects, i.e. it has no structure declaration. The methods are standard methods defined in an object-oriented language :
(the type of the parameters are given in <>)

- == <object>: tests structural equivalence of two objects.
- type : creates from an object X (defined by a DML clause) a type descriptor which is element of C.type.
- elt_of : gives the name of the class to which belongs an object.
- kill : used to destroy objects .
- print : print an object. It takes as argument the device on which the object is to be printed.
- return : returns the receiver.

Following the terminology of [MAIE 85], every object of VOOD is a *kind* of OBJECT. But no objects are instances of C.OBJECTS. C.OBJECTS is an *abstract* super-class for every class. It provides methods that every object inherits.

System-defined types are atomic types such as:

- integer
- real
- string
- boolean
- text
- picture
- voice
- range
- UND : the object undefined.This object is used as a trap. The result of every message sent to it, is itself, and it is obtained every time a method is not aplicable. Of course the class of UND contains only UND.

For the sake of clearity, atomic classes are represented in figure 3.1 by one class named C.ATOMS. Of course, each of these types defines its own class. The associated methods are the usual ones.
An other important system-defined class is the class of IDENTIFIERS. It is defined by type IDENTIFIER.

```
TYPE          identifier;
SUPER-TYPE    string;
I_METHOD      < <-, print_id>;
END_TYPE;
```

The specific methods of type identifier are the following:
_ <- : This method is used to assign the receiver (an identifier) to the object given in argument. The argument can be either an identifier, in this case the receiver is assigned to the object pointed by the argument, or a block which evaluation results in an object. Section V presents examples of use of "<-".
_ print_id : print the identifier of an object. Examples of use are the creation of an object :
"C.PERS new print_id".
C.PERS is a class of Persons." New" creates a new person and "print_id" prints its internal identifier. If we had used the method "print" of class C.OBJECT, we would have printed the object instead of its ID.
The non-atomic predefined types are mainly type generator as defined in [SCHA 86]. They are used to define methods that will be inherited by user defined types and serves as data structure constructors.

3.2.1. the type COLLECTION which is parametrized by type T

```
TYPE          collection <T>;
SUPER-TYPE    object;
I_METHOD      <do, collect, select>;
END_TYPE;
```

This type defines a class which factorizes commun methods for sets and lists. The semantic of the methods is that of smalltalk. "do" requests to a set or a list to enumerate its elements and "collect" is used to build messages that request to a collection transformation of its elements . "Select" selects the elements of the collection that satisfies a condition given in argument. We see in section IV examples of use of these methods.

3.2.2. the type SET which is parametrized by type T.

```
TYPE          set <T>;
SUPER-TYPE    collection;
STRUCTURE     set (n)of T;
I_METHODS     <S_METHODS>;
END_TYPE;
```

In this declaration n is the upper bound of the cardinality of the set. The set of methods called S_METHODS contains the usual set operations (union, intersection, difference) and the following methods:

_{} : the set constructor.
_IN : tests the membership. Return the boolean objects T (True) or F (False).
_CARD : returns the cardinality of a set.
_COLLAPSE : constructs a set from a set of sets.
_× : THE CARTESIAN PRODUCT. Let A (resp B) be a set of T1 (resp T2). Then the cartesian product A× B is the set of tuples { x ∈ T1, y ∈ T2 [A:x, B:y]}. As will be seen in section IV, the cartesian product of A and B is obtained by sending to A the message " × B", × is the method and B is its argument.

We need the empty set Φ.

3.2.3. the type LIST which is parametrized by type T

```
TYPE          list of <T>;
SUPER_TYPE    collection;
STRUCTURE     list (n1, n2) of T;
I_METHODS     <L_METHODS>;
END_TYPE;
```

n1 and n2 are the lower and upper bounds of the length of the list.
L_METHODS (figure 2) contains usual methods defined on lists in programming languages such as LISP [McCA 65]. we briefly recall them:

_LENGTH : returns the length of the list.
_<> : is the list constructor.
_I : returns the Ith element of the list.
_CAR (resp CDR) : returns the head (resp the tail).
_CONS : Let A of type T, L of type list of T then A CONS:L is a list L' such that "L' CAR" is A and "L' CDR" is L (L' CAR means that the message CAR is sent to L').
_CONC : is similar to collapse for lists of lists.
_× : the product of lists. let A=<x1, ...,xn> and B = <y1, ...,yp> be two lists then A × B is :
<[A:x1,B:y1].[A:x1,B:y2]. ....[A:x1,B:yp]. ....[A:xn,B:y1]. ....[A:xn,B:yp]>

We also need the empty list :NIL.

3.2.4. The type TUPLE

```
TYPE          tuple <T1, T2, ...,Tn>;
SUPER_TYPE    object;
STRUCTURE     BEGIN
              A1 : T1;
              A2 : T2;
                .
                .
              An : Tn;
              END;
I_METHODS     <T_METHODS>;
END_TYPE;
```

We define in T_METHODS the classical methods of renaming, supressing and inserting an attribute in a tuple definition. [] is the tuple constructor and ?Ai is the projection of the tuple on attribute Ai.
We define the method "join on:<cond> with:<object>" to be the tuple join on condition cond as follows:

definition 1 [1] :

Let A=[A1, ...,An] and B=[B1, ...,Bp] be two tuples, let C be a clause in conjonctive normal form and(or(Ci)) where Ci is : (Ai COMP Bj), Ai and Bj being of the same type and COMP a comparison operator defined on this type.
Then "A join on:C with:B" is the tuple [A1...,An, B1, ...,Bp] if C is true and is the object UND if not.

We formally define in the next section the structure of this message. The reader has just to note that A is the receiver and "join on:C with:B" is the selector of the method with its arguments. If C is a tautology, then "join" is a tuple concatenation.

3.2.5. The CASE type

The last system-defined structure is the "case".

```
TYPE          case <T0, T1, T2, ...,Tn>;
SUPER_TYPE    object;
STRUCTURE     CASE dis:<T0> of
              a1 : T1;
              a2 : T2;
                .
                .
              an : Tn;
END_CASE;
I_METHODS     <C_METHODS>;
END_TYPE;
```

The set C_METHOD contains the following methods:

_%DIS : returns a field of the case for a given value of dis.
_CASE : the case constructor : "CASE dis:<object> OF {[val:<OBJECT>]}"

For the case constructor as for the set, list and tuple ones, the objects involved may be existing objects or may result of the evaluation of transmissions (see section IV for the data manipulation). For example if M1, ...,Mn are transmissions such that their evaluation result in objects of the same type, then {M1, ...,Mn} constructs a set of objects.

We define two important sub-types of the preceeding ones. They will be used to insert and manipulate user defined objects in the class hierarchy.

3.2.6. The TYPE type

---

[1] Unlike the relational model the join operation of definition 1 deals with tuples and with SETS of tuples (relations). We shall show later how relational join can be constructed with tuple join and can be extended to lists or hierarchical structures based on tuples.

```
TYPE          type;
SUPER_TYPE    tuple;
STRUCTURE     BEGIN
              type      : string;
              super_type : string;
              structure  : string;
              methods    : set(0, *) of method;
              END;
I_METHODS     <TY_METHOD>
END_TYPE;
```

the set <TY_METHOD> contains the following methods :

_ SELECTORS :give the selectors of the methods defined for the receiver type
_ ALL_SELECTORS : give all selectors of the methods defined for the receiver and its ancestors.
_ IN : has a selector as argument and gives the boolean object "T" if this selector is in the dictionnary of methods of the receiver.
_ ALL_STRUCTURES : give all the structures of the ancestors of the receiver
_ WHERE : give the type of the objects that understand the method given in argument. The search begins with the receiver and scans every ancestor if needed.
_ DEL_METHOD : deletes a previously defined method. The transmission for deleting a method is :" T_X del_method method_sel", where T_X is a type identifier and method_sel is the selector of the method to be destroyed.
_METHOD_CODE : prints the code of a given method.

Every user defined type is an instance of type TYPE, and thus an element of the class C.TYPE. One of the most interesting features of this object oriented approach is that every type is an object of type tuple and the class C.TYPE inherits tuple methods. Any type declaration may be dynamically modified using tuple methods. For example the type name may be changed, the structure may be modified by replacing the old string by a new one. The structure field is a string that represents the syntax tree obtained from the type declaration.

3.2.7. the CLASS type

```
TYPE          class;
SUPER_TYPE    set;
I_METHODS     <CLASS_METHODS>
END_TYPE;
```

Recall that every class of objects is an instance of type class and thus, is an element of the class C.CLASS. A point to be noted is that C.CLASS is a sub_class of C.SET. It thus inherits every set operations.
We see in this declaration that the structure field is optional as the method one but one of them must be present in a type declaration. The instance Methods of class CLASS are the following:
{ new1, new2, sub_classes, super_classes, all_elements, all_sub_classes, all_super_classes };

_ NEW1 : creates a class. New1 takes an instance of C.TYPE as argument.
This method is inherited by every subclass of C.CLASS. When applied to a class "new1" creates an instance of this class.
_ NEW2 ; creates a class and take a DML block (see section 4) as argument.

the type structure of the elements of the class is derivated from the compilation of the block. The new class is filled with the value of the block.

- SUB_CLASSES (resp super_classes) : when applied to class, gives its direct subclasses (resp its super_classes).
- ALL_ELEMENTS :gives all elements of a class.
- ALL_SUB_CLASSES (resp ALL_SUPER_CLASSES) : gives all subclasses of a class (resp superclasses).

We can now describe the Class hierarchy (figures 3.1 and 3.2).

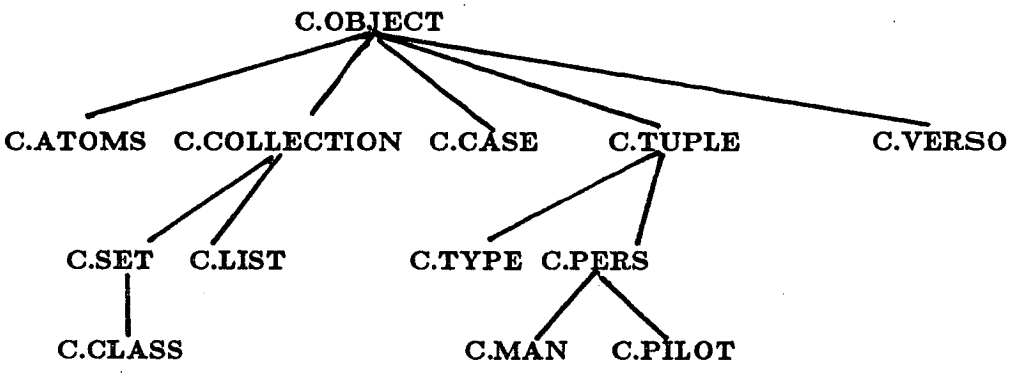


FIGURE 3.1 : THE INHERITANCE TREE

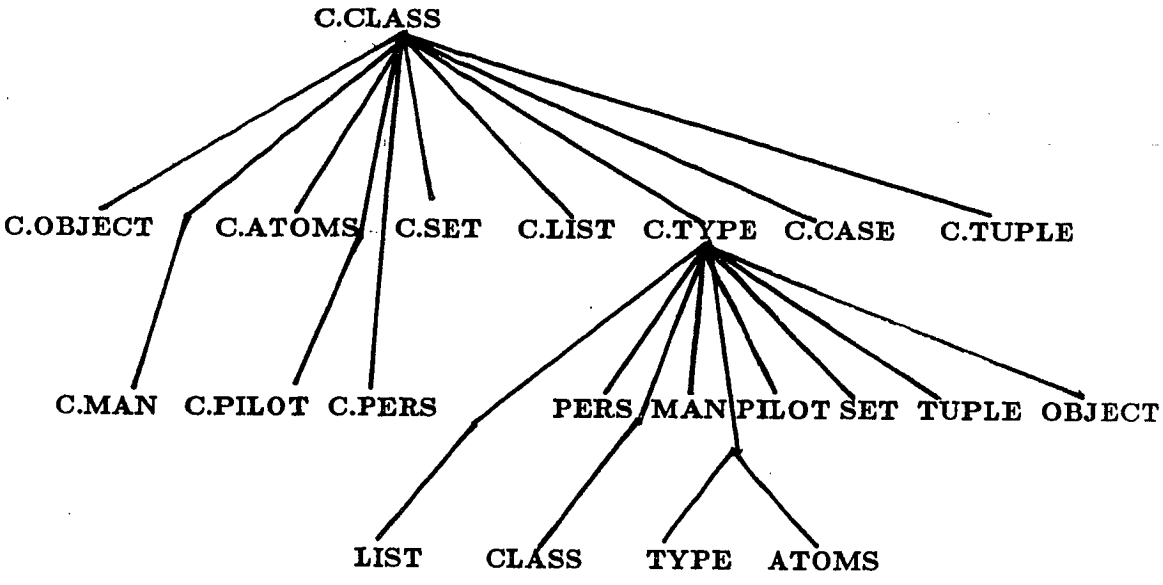


FIGURE 3.2 : THE INSTANCIATION TREE

As can be seen on figure 3.1, C.CLASS is a subclass of C.SET which is a subclass of C.OBJECT. Each element of C.CLASS is a set and an object. On the other hand, C.OBJECT is an element of C.CLASS since it is a class and thus an instance of type "CLASS". Figure 3.2 displays the tree of instances in VOOD.

The VERSO DBMS is viewed as an object on which database operations can be performed The class C.VERSO is thus a singleton.
We recall that our OMS is implemented on top of VERSO (see section V). The type definition of C.VERSO is the following:

```
TYPE          verso;
SUPER_TYPE    object;
I_METHODS     <V_METHODS>;
END_TYPE;
```

V_METHODS contains the VERSO DBMS operations.

We show how new objects are defined in VOOD.

EXAMPLE 3.2.1:
let us first define the PERSON type as follows:

```
TYPE          person;
SUPER_TYPE    tuple;
STRUCTURE     BEGIN
              name       : string;
              first-name : string;
              sex        : (m, f);
              age        : integer;
              address    : string;
              l_children : list(0, 10) of person;
              end;
END_TYPE;
```

The class C.PERSON contains tuples of values of type PERSON (note that this class represents a nested relation). We defined no particular methods for this class but it inherits tuple operations. Note that we allow recursive definitions since the attribute "l_children" is a list of person.
We now define the type MAN to represent persons whose sex is male.

```
TYPE          man;
SUPER_TYPE    person;
WHERE         sex=m;
STRUCTURE     BEGIN
              military_status : string;
              END;
END_TYPE;
```

Every object belonging to C.MAN is related to C.PERSON by an "ISA " relationship. That is, the class C.MAN is used to store additional information on male persons. We could also have defined specific operations on men by adding an "I_METHODS" clause to the type definition.
We say that C.MAN is a *restriction-extension (RE)* subclass of PERSON. Other subclasses are pure *restriction (R)* when there is no "STRUCTURE" clause and pure *extension (E)* when there is no "WHERE" clause. *User-defined* classes which direct parent is a system-defined class are denoted by U. For example the PILOT class is a pure extension subclass (E) of C.person which is a subclass (U) of C.TUPLE.

```
TYPE          pilot;
SUPER_TYPE    person;
STRUCTURE     BEGIN
              flight  : integer;
              company : string;
              END;
END_TYPE;
```

A subclass inherits all of the whole structure of its superclasses. We show in section V how these features are implemented through VERSO relations.

4. THE VOOD DATA MANIPULATION LANGUAGE

For most object oriented models such as SMALLTALK [GOLD 83], object manipulation is done through *messages,* that activate methods. These methods are associated to the receiver or

inherited from its parents . We think that complex object manipulation should ultimately be done through graphical interfaces which will be implemented on top of the system. Adding an interface to the system is conceptually very simple since it is only a new object. We decided to follow a SMALLTALK like approach. We use smalltalk features such as blocks, sequence, cascading, and add ad-hoc features in order to easily query and build complex structures. These features are methods for system- defined classes such as sets, lists, tuples and methods constructors.
We thus keep most of Smalltalk (message constructors etc...). We show in this section how database operations such as nesting, unesting or join can be performed by our message constructors.
Since a method is an object of the system, it must have a type and belong to a class.
We thus define the type METHOD as follows:

```
TYPE          method;
SUPER_TYPE    tuple;
STRUCTURE     BEGIN
              selector : string;
              arg1     : string;
                .      :  .
                .      :  .
                .      :  .
              argn     : string;
              code     : text;
              end;
I_METHODS     <M_METHODS>;
END_TYPE;
```

We see from the structure of the type above, that a method has a selector, a code , and a list of arguments.
The list of arguments "arg1, .., argn" characterizes the type of every argument.
Every method defined on a given type is thus an instance of type METHOD and belongs to C.METHOD.
In our system, querying the data (complex objects) is done through programs. That is a message is sent to the object in order to manipulate it. The main advantage is that there is no artificial separation between application programming and data manipulation.
A transmission has the following syntax:

RECEIVER SELECTOR VAL1. ... VALn

Receiver is an object identifier. This object has to be manipulated via the method which selector is SELECTOR. The list of arguments may contain as values either object-id's belonging to the type of the corresponding arguments or blocks delimited by () (we give examples of blocks in the following).
Let us take as an example the creationof an object. This is done by sending a "NEW" message to its class:

C.PERS new1

If we wish to give an initial value to the new created tuple, we may write:

C.PERS new1 [NAME:Smith,FIRST-NAME:toto,SEXE:m,AGE:28,ADDRESS:Paris]

NEW accepts one argument whose structure depends of the type of the object it creates. In order to associate the new object to a given ID, we type:

#toto <- C.PERS new1 [NAME:Smith,FIRST-NAME:toto,SEXE:m,AGE:28,ADDRESS:Paris]

The symbol <- is a method. It establishes the connection between the ID #toto and the object created by the transmission.
In order to build complex queries, we use the following Smalltalk features :

1) SEQUENCE : M1.M2....Mn
M1, ..., Mn are transmissions and are evaluated in sequence

2) CASCADE : ID SELECTOR1 ARG1 ; SELECTOR2 ARG2 ;. .. ; SELECTORn ARGn
In this case ID is the receiver for each selector of the sequence.For example ,let us consider the following cascade: 1 + 1; + 2; + 3
The result is a stream of objects which are generated from the object 1 by successively applying each selector of the cascade : 2, 3, 4.

3) CONTINUATION : ID SELECTOR1 ARG1 SELECTOR2 ARG2. .. SELECTORn ARGn
ID first receives the message SELECTOR1 ARG1, it then produces a result that receives SELECTOR2 ARG2 and so on.

4) BLOCK : (x1:...:xn | M1...Mp)
Where " M1...Mp " is a sequence of messages and "x1:...xn" are arguments. See [GOLD 83] for a formal definition of block. We show in examples 4.2 and 4.3 how the block structure is used to perform joins or nestings.

We show how the evaluation of a message is done through an example:
2 ×(3 +4)
In this case the argument of the selector × is first evaluated since it is a block, we get 3 +4, i.e. the object 7. Then we have 7 as argument for × which gives the object 14. If we write "2 × 3 + 4", then the result is 10. The first message is evaluated and give the object 6 which becomes the receiver of the next message "+ 4".
We now describe the methods of M_METHODS.

1) SET CONSTRUCTOR: {s1, ..., sn}
let O be an object, and s1, ..., sn be methods that accept O as a receiver:
O {s1, ...sn} = {O s1, ..., O sn}.
This constructor allows us to construct a set of objects with a single message and a set of methods. Let us suppose that we want to generate the set {1,2,3,4} from the object 0. This done with the following transmission :
0 {+1,+2,+3,+4}

2) LIST CONSTRUCTOR: <s1 .... sn>
O <s1. ...sn> = <O s1. .... O sn>.
The list <1.2.3..4> is the result of the transmission " 0 <+1.+2.+3.+4>".

3) TUPLE CONSTRUCTOR: [A1:s1, ..., An:sn]
O [A1:s1, ...An:sn] = [A1:O s1, ..., An:O sn].

4) CONDITION: IF p THEN f ELSE g
let O be an object and p a method that produces the boolean object T or F :
IF O p THEN O f ELSE O g
This constructor is predefined but may be generated as follows :
(self p) value iftrue (self f) value iffalse (self g) value
Where self is defined as for smalltalk, and "iftrue" and "iffalse" are methods of type boolean.

5) FILTER: filter p
p is a message which result is boolean
"O filter /p/" is O if "O p" gives T

6)DEFINED_FOR :
This method adds its receiver to the dictionnary of methods associated to the type given in argument. Let JOIN be a new defined method, then the message " JOIN defined_for T_REL" adds JOIN to the methods of the dictionnary of the type T_REL.

7) FOR_WHICH :
Gives the types for which the receiver has been defined.

8) KILL :
Kill the receiver and removes it from the dictionnaries where it appears.

Since a method is an object, the definition of a new method can be stated via a type definition or via a DML equation. In the latter case, the method is defined exactly as an object is defined through a DML query. For example, we define the relational projection on attribute A as follows:

S_PROJ <- method new [selector:S_PROJ, code: COLLECT ?A]

?A is defined on a tuple and "COLLECT ?A" produces from a set of tuples, a set of A values. We recall that <- associates a value to an object ID. In order to be applied to a relation of type T_REL, we must add this method to the list of methods of T_REL. This is done by sending the message :

S_PROJ defined_for T_REL.

We could have performed both operations in one pass by sending the following :

method new [selector:S_PROJ, code: COLLECT ?A] defined_for T_REL

Every object inherits the methods of its ancestor. When two methods are given the same selector, the system always executes the one which belongs to the type of the object or to its "youngest" ancestor (we recall that we allow only single inheritance). This is one way to mask inheritance of methods.

EXAMPLE 4.1:

We use the class C.PERS of example 1 of section III.Our first query is :

Q1 : Give the list of children of Smith?
SMITH ?L_CHILDREN

Here Smith is an object identifier.

Q2 : Give the name and first-name of people having three children?
C.PERS collect (y|y filter /?L_CHILDREN LENGTH = 3/ ?NAME, FIRST-NAME)

The result is a set of tuples [NAME, FIRST-NAME].The evaluation of this transmission proceeds as follows: First the filter is applied to C.PERS and selects people with 3 children, then the message "?NAME, FIRST-NAME" is applied to the result of the previous transmission.

EXAMPLE 4.2:

Let S = {[A, B]} and T = {[A, C]} be two sets of tuples (we omit the type of each attribute for clarity).
In order to compute a relational join, we first construct a "tuple-set" join from the "tuple-tuple" join of section II :

METHOD new [selector:TS_JOIN ON:WITH:, COND:boolean, REL:T_REL,

code: REL collect (y|self JOIN ON COND WITH y)] defined for T_REL

Let S, T be two relations (set of tuples) and cond be a block which evaluation gives a bolean then the relational join of S and T is:

S collect TS_JOIN ON: cond WITH: T

The query is more complicated than a relational join. This is necessary since in our model, JOIN is predefined on tuples and not on set of tuples. On the other hand it is far more powerful since it can easily be extended to lists of tuples or lists of sets of tuples etc.... We only need to program TS_JOIN one time (i.e creating a new instance of C.METHOD which selector "TS_JOIN") and we just have to define it for each concerned type.

EXAMPLE 4.3:

Let S = {[A:a, B:b]}, we now compute the nesting of S on attribute A, say T= {[A:a, B:{b}]} as follows :

S collect (y|#x <- y ?A. y [A:?A, B:self collect(filter /?A = #x/ ?B)])

In this case we must use a temporary variable #x to group B-values for a given A-value. #x<- y ?A affects to the variable #x the current value of the projection on A.Then we filter every tuple of S on equality on #x in order to construct the set of B-values associated to x.

The unesting of T is :

T collect (y|#x <- y ?A. y ?B collect (z|[A:#x, B:z]))

Collect associates to y, a tuple of T, #x contains its projection on A. Then we construct a set of tuples from the set "y ?B" with an embedded block (recall that B-values are sets). The result has the following structure : {{[A:a, B:b]}}. We thus have to do a set collapse in order to get a pure unnesting of T.

In this section, we have shown the power of extending Smalltalk features to deal with data base requirements. We follow an approach similar to that of [COPE 84a].

5. UPDATES AND DYNAMIC MANAGEMENT OF INSTANCES

We outline in this section the problem of updating objects in VOOD. We show how the dynamic management of instances of a class may be solved using the Verso transaction management system [VERSO 86].

5.1. creation and deletion of objects

We list in the following table the effect of sending a new (or a kill) message to a given user-defined class. We show in what cases the message will be accepted or rejected depending on the type of the class. Recall that U,R,E,RE defines the type of the class (cf section 3).

cat/op	insertion	deletion
U	allowed	allowed
R	forbidden	forbidden
E	allowed+	allowed
RE	forbidden	forbidden

TABLE 5.1

+ : the insertion into C is allowed only if O already belongs to the superclass of C.
* : the update is allowed only if it concerns parts of the object that are described by the type corresponding to C and not by its supertype (i.e parts that are purely extensions).

In table 5.1, we list the operations to be performed on C' subclass of C in order to repercute the side effects of an operation on C.

cat/op	insertion	deletion
R	insertion*	deletion*
E	Φ	deletion*
RE	insertion*	deletion

TABLE 5.2

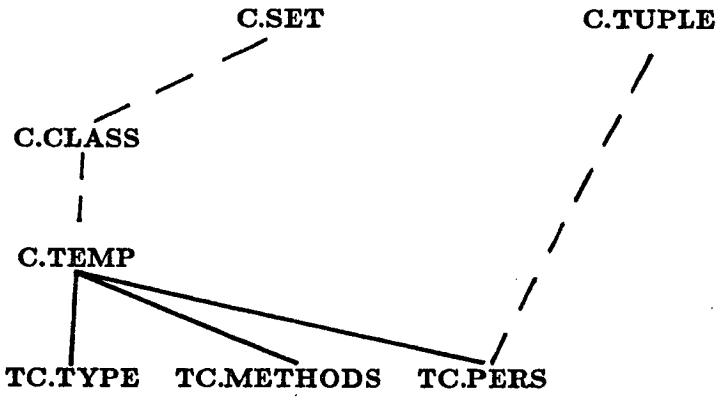
* : The operation must be performed on the subclass only if the where clause was true or becomes true (in cases of updates and insertion).

5.2. dynamic management of instances

The problem is to dynamically permit the creation of instances of a type when an end-user is manipulating objects through the DML and wants to store its new created instances.
We need the possibility to define temporary instances, classes and methods. A session in VOOD always begins with the transmission : SESSION NEW.
The effect of this transmission is the creation of particuliar system defined classes (figure 5.3). These classes are temporary. They are thus prefixed with "TC" and are destroyed when a "SESSION ABORT" or a "SESSION VALIDATE" transmision occur. As can be seen on figure 5.3, an TC.TYPE class is created. This class contains all temporary types defined in the session and inherits the methods of C.TUPLE and posseses a copy of the dictionnary of C.TYPE. The methods of C.TYPE are thus applicable to the instances of TC.TYPE. TC.METHOD corresponds to C.METHODS and is used to manage temporary methods defined on temporary user defined classes. C.TEMP is a *permanent class* which has temporary classes as elements. It is a subclass of C.CLASS. C.TEMP is used to define specific methods on temporary sub-classes. We do not want that temporary sub-classes inherit every method of C.CLASS, since the creation and update of a temporary class differ from that of a permanent class. The specific methods of C.TEMP are the following :

_NEW1 : This method creates a new temporary class with a type declaration as argument.
_NEW2 : creates a new temporary class from a DML block and fill it with the value of the block.

A type can have several associated temporary classes depending of the number of current sessions but it must have one and only one permanent associated class.



: element of
: subclass of

FIGURE 5.3 : THE INHERITANCE TREE

We can now show on an example session the management of temporary instances.

EXAMPLE 5.1 : a session in VOOD (lines contained in brackets are comments)

SESSION NEW
{ a transaction is opened }

#S <- C.PERS COLLECT (?NAME).
{#S is variable that contains a set of names. One can apply on it the methods defined in the class C.OBJECT}

S1 <- C.NAME NEW (C.PERS COLLECT (?NAME)).
{S1 is a new created element of the class C.NAME and is filled with the value of the block}

C.NAME1 <- C.CLASS NEW2 (C.PERS COLLECT (?NAME)).
{C.NAME1 is a new class created by new2 and filled with the value of the block. }

TC.NAME1 <- C.TEMP NEW2 (C.PERS COLLECT (?NAME)).
{TC.NAME1 is a temporary class created by new2 and filled with the value of the block }

JOIN DEFINED_FOR TC.NAME1.
{JOIN is a method already defined for C.PERS that we wish to use in TC.NAME1}

TC.METHODS NEW [SELECTOR:TPRINT,ARG:TNAME1,CODE:***] DEFINED_FOR TC.NAME1.
{we add a specific print method to TC.NAME1.

This method will exist only for the current session}.

SESSION VALIDATE.
{the session is validated, all temporary classes are destroyed
and C.NAME1 is stored in the system. }

The VERSO transaction mechanism [VERSO 86] is in charge of maintaining the consistance of the base V-relations that implement the classes (see next section).

6. IMPLEMENTATION CHOICES

In this section we outline our implementation choices. The prototype will be realized on top of the VERSO DBMS. Verso [VERSO 86] is running on the SM90 mini computer []. Its main feature is its ability to manage non normal form relations (see [ABIT 84] for a formal specification). These relations are described by a COMPACTION FORMAT.

For example let R(NAME,CARS,CHILDREN) be a relation, then a compaction format may be : (NAME (CARS)* (CHILDREN)*)*.
The meaning of such a structure is that the children and cars are grouped for each name in the relation. Futhermore these sets may be empty.
We give a definition of a Verso compaction format:

DEFINITION 6.1 :

If X is a finite string of attributes then (X)* is a flat format. If X is a finite string of attributes and f1,..., fn are formats for n positive, then (X f1 f2 ... fn)* is a format. We require that an attribute does not appear twice in a format.

Since compaction formats describe hierarchical files, they are a good tool to implement our complex data structures. We do not have to deal with a lot of surrogates as if we used a pure relational system [ZDON 85]. Similarly our queries will be translated in VERSO algebra operations that manipulate N1NF relations.
We intend to develop our prototype in Prolog on top of VERSO. In a first step, we are not concerned with performances but wish to quickly have a running system in order to check the validity of our modelling choices. We think that Prolog is well suited for quick prototyping of compilers.
We show how complex objects are implemented via Verso relations (V-relations). In our system, We first consider the predefined structures :

6.1. implementation of sets,lists,tuples and cases

i) Let C.S be a class of objects S which are sets (of objects of type O):
C.S is stored by the following V-relation:

C.S (NB_ELT (S_ID,S_NB_ELT(S_O_ID)*)*.

Where C.S is the name of the class of S (and of the V-relation), NB_ELT is the cardinality of the class C.S, S_ID is the identifier of an object S. S_ID is an object of type IDENTIFIER. We assume that the system produces a new Id each time it is needed. S_NB_ELT is the cardinality of S and S_O_ID is the identifier of an element of S if it is an object already defined in the system. In VOOD an object is stored only once in the V-relation that implements its class and is referred by its ID in the other V-relations. If the type declaration of S explicitely defines the variable instances of the elements of S, then their values are directly stored in C.S instead of

S_O_ID's.

ii) let C.L be a class of lists L of objects of type O:
We have :

C.L (NB_ELT (L_ID,L_NB_ELT (L_O_POS,L_O_ID)*)*.

In this case, we must add the attribute L_O_POS which is the position of an object O in the list L in order to allow duplicate O's in the V-relation (which is a set).

iii) let C.T be a class of tuples T [A1,...,An].
If no nulls are allowed then we have:

C.T (NB_ELT (T_ID,T_A1,...,T_An)*)*

where T_Ai is atomic and has as values the Id of an object of type Ti if there is no class C.Ti, or represents a compaction format if Ti is defined in the type definition of T.
If nulls are allowed then :

C.T (NB_ELT (T_ID (T_A1)*...(T_An)*)*)*

iv) let C.C be a class of cases of type:

```
TYPE          case
SUPER_TYPE    object;
STRUCTURE     CASE dis OF T0
              A1:T1;
                .
                .
                .
              An:Tn;
              END;
END_TYPE;
```

we represent C.C by :

C.C (NB_ELT (C_ID DIS (A1)*...(An)*)*)*.

In this case for every C_ID record only one of the Ai's may not be empty.

EXAMPLE 6.1 :
Let us define the following complex object :

```
TYPE       T;
SUPER_TYPE tuple;
STRUCTURE  BEGIN
           A : BEGIN
               B : set(0,n) of O;
               C : set(0,m) of U;
               END;
           D : string;
           E : set of F : BEGIN
                          G : string;
                          H : string;
                          END;
           END;
END_TYPE;
```

Following the type definition we would produce the following format:

C.T (NB_ELT (T_ID,A_B_NB_ELT (A_B_O)* A_C_NB_ELT,A_C_NB_ELT (A_C_U)* D,E_NB_ELT (E_F_G,E_F_H)*)*

But this format is not allowed in VERSO
We must then store C.T as follows :
C.T (NB_ELT (T_ID,A_B_NB_ELT A_C_NB_ELT,A_C_NB_ELT,D,E_NB_ELT (A_B_O)* (A_C_U)* (E_F_G,E_F_H)*)*)*

6.2. implementation of system defined classes

We now outline the implementation of the system-defined classes.
The first of all is C.OBJECT :

C.OBJECT (NB_ELT (ID (C_RANK,TYPE)*)*)*

NB_ELT is the number of objects in the system, ID is the identifier of an object. Due to the inheritance machanism, an object may correspond to multiple types. For example a pilot is a person that is a tuple. These types are strictly ordered. To each type, corresponds a subclass of the class of its father. We thus construct the following order on CLASSES :

C_RANK(C.OBJECT)=0.

For each system-defined type T :

C_RANK(C.T) = 1.

Then for every class of type T with super type T':

C_RANK(C.T)=C_RANK(C.T')+1.

For example C_RANK(C.CLASS) = C_RANK(C.CTYPE) = 2 and C.RANK(C.PERS) of example III.1 is 2.

We implement C.CLASS as follows:

C.CLASS (NB_ELT (ID, CATEGORY (C_RANK,S_ID)*)*)*

ID is the identifier of a class (recall that a class is an object), CATEGORY takes its values in {R,E,RE,S,U}. S means system-defined classes, U represents user defined classes children of an S class (for example C.PERS which is a subclass of C.TUPLE). R,E,RE corresponds to subclasses of an U class (see section III).
(C_RANK,S_ID)* contains the ranks and id's of every superclass of the current one.

The class C.TYPE is stored as:

(NB_ELT
(ID,TYPE,SUPERTYPE,WHERE,STRUCTURE,(ID_METHOD,CODE(POS,ARG)*)*)*

ID is the object type Id, TYPE is the type name, SUPERTYPE is the supertype name. WHERE contains the syntax tree of the selection clause on the supertype. STRUCTURE contains the syntax tree of the structure declaration.
(SEL_METHOD,CODE(POS,ARG)*)* contains the set of method selectors together with their code and list of arguments.

The last class C.METHOD contains all methods:

C.METHOD (NB_ELT (ID (TYPE_ID SELECTOR (ARG)* (CODE)*)*)*)*

As can be seen we allow a method to be defined on several types.TYPE_ID is of course the identifier of a type, it corresponds to the ID attribute of the V-relation C.TYPE.

6.3. implementation of subclasses

Two solutions were considered :

i) to store in physically distinct structures the objects that belong to sublasses of a given class, i.e to have a V-relation for each class.

ii) to store them in the same V-relation, i.e to have one V-relation for a hierarchy of user-defined classes.For example C.PILOT,C.MAN,C.PERS will be stored in the same

V-relation.

We choose the second solution for the following reasons:

i) There is no duplication of information, it is space preserving.

ii) It is easier to manage because we don't have to repercute updates on copies.

iii) We make better use of the power of Verso (hierarchical structures and null values), and optimize queries that deal with inheritance.

The implementation principle is the following:

let C.T be a V-relation that represents the class C.T :

C.T (NB_ELT (T.ID,X (F1)*,...,(Fn)*)*)*.

Suppose that we define T' to be a sub-type of T obtained by extension,then if T' were of category U, it would have been represented by :

C.T' (NB_ELT(ID,Y (G1)*,...,(Gn)*)*)*.

Since T' is of category E we just extend the V-relation C.T :

C.T (NB_ELT (T.ID,X (F1)*...(Fn)* (Y)* (G1)*...(Gn)*)*)*.

Due to Verso formats,we must store the value Y as (Y)* instead of Y. Adding a new subclass in the system involves no data manipulation because VERSO allows to add dynamically attributes to an existing V-relation without reordering the old ones.So creating a new sub-class does not change the way data is stored.

To illustrate this process let us take again example III.1 :

C.PERS is (NB_ELT(ID,NAME,FIRST-NAME,SEX,CHILDREN,ADRESS)*)*.

We now create the subclass C.PILOT, there is no V-relation C.PILOT, but we have

(NB_ELT(ID,NAME,FIRST-NAME,SEX,CHILDREN,ADRESS (FLIGHT,COMPANY)*)*.

The case of restriction sub-classes involves no more data manipulation. We just perform on the parent class the restriction stored in the type every time a user wants to acces an object via this class.For example any query on a man will be merged by the DML compiler with the clause "|SEX = M" and send to C.PERS instead of C.MAN. Updates will be thus automatically repercuted in every classes. For RE classes, we only have to do both processes.

7. CONCLUSION AND FUTURE WORKS

This paper addressed the problem of managing complex data. Following [STON 85a] a complete type system should allow :

the definition of user-defined types

the definition of new operations for the data types

Three approaches are currently investigated. The first one consists in augmenting the functionalities of the relational model [STON 85a, 85b]. An user may define its own data types, operations and acces methods. For example, a procedure code or a query expressed in data manipulation language may be stored in a column of a relation [STON 85b]. The advantage of this approach is that it uses a well known model and that a prototype can be quickly developped. A drawback is that this approach leads to "patched-up" systems whose complexity may badly increase with the emergence of new applications. The second approach is the extension of the relational model to nested relations. Nested relations [SCHE 82,VERSO 86] can store directly complex data. Futhermore well-defined query languages (algebra [VERSO 86] or Sequel-like [PIST 85]) exits and are implemented. This approach is promizing. It offers a good frame to store and manipulate complex objects. However, it does not provide with good solutions for a lot of problems encountered in new applications of data bases such as Version managing, Triggering, Representation templates or application programming. We thus decided to follow the third approach which consists in investigating Object-oriented programming. Many authors [COPE 84a, DBE 85, NIER 85] used

this approach in the recent years. Some decided to augment an existing Object-Oriented Language (OOL) [COPE 84a, 84b] in order to manage complex data. Others uses an object-oriented approach to define an object management system [DERR 85,NIER 85, ZDON 85]. We decided to emphasize on the notion of type which is not present in an OOL such as Smalltalk [GOLD 83]. Indeed in data base applications, lots of objects share the same structure and performance goals need a compiled approach for complex queries. In our model we insist on Data definition and manipulation but we keep programming features that are a subset of Smalltalk 80. We defined this model with the idea of implementing it on an existing relational data base management system called VERSO [VERSO 86]. The storing and management of data will thus be done by VERSO that offers the power of nested relations. This will free us of manipulating lots of surrogates [ZDON 85] in order to physically represent our data structures. Verso algebra manipulates directly nested relations and will facilitate query execution. An originality of the VOOD data model is that every system defined objects are defined in terms of two data structures : tuple and collection. For example VOOD methods are defined as tuple objects and thus fits well in the model in contrast with Smalltalk. The class of methods is thus a subclass of the class of tuples and inherits of tuple methods. Similarly the class of all classes itself (C.CLASS) is a subclass of the class of collections and thus inherits its methods.
Areas of works that we intend to investigate are :
_ The implementation of a triggering mechanism. In office information systems, forms processing needs a triggering mechanism in order to propagate updates and to implement views constraints. We also would like to use triggers to implement a notion of derivated or calculated field in an object instance. That is, we want to have the ability to compute an instance variable from the values of other instance variables. In our model, objects are passive. They do not have ongoing activity associated with them. One way to implement triggers [DERR 85] is to have "active" objects such as in [BIRT 73], [KRIS 81].

_ One mechanism that is needed in OIS is a temporal support [COPE 84b], [DADA 84]. This has to be done at the instance level (by keeping different versions of a given instance) and at the type level. In VOOD, the latter point would not be too difficult to handle since a type object is simply an instance of the type TYPE.

_ The inheritance in VOOD has to be refined on two points. First, we think it is necessary to manage multiple inheritance. This poses multiple problems from the point of view of performance and semantics. For example, depending on the implementation, searching in the hierarchy may be done by relational joins.

_ For CAD applications, our notion of type is too much rigid. Indeed, in CAD sessions an instance may not be coherent according to its type. Some instance variables may not be present. The coherence can not be checked on the whole instance but only on existing variables. For example, an architectural plan will satisfy the type declaration only at the end of its conception that may take a very long time. In this case, rather than with a type, the system should deal with a "prototype", that describes the final result and allows partial instance values.

_ The notion of representation templates. Office information systems applications (forms processing [COLL 86]) deal with multiple external representations of the same conceptual object. We think that a representation level must be defined that contains a type "Template". An object of type Template describes the way a conceptual object will be viewed by the user. When an object has been "represented" via a template , it has to be displayed via an interface. The way this is done depends of course of the type of the interface (text editor, printer, graphical interface, voice...). We think a third level must be defined that includes "display" objects. These objects contain informations on how a given representation object will be displayed. The process of displaying an object would be done by a transmission such as :
Temp_pers SEES: John AS: Dis_pers.

Where Temp_pers is a template object, John an object of type pers and Dis_pers is a display

object that describes the graphical commands needed to display john. SEES: AS: is the selector of a method that computes the representation of john via the template.

_ The integration in terms of objects of different programming languages. The goal is to break the wall between programming languages and database query languages. If programming languages are well defined in terms of objects of VOOD, then they will cooperate on the same data objects without the need of any translation mechanism. The data structures of these languages will be data objects of the system.

_ The problem of the appartenance of the operations. In our model such as in Smalltalk, an operation belongs to one or more types. Operations that act upon multiple types should not belong to one of them instead of another. An interesting idea is to allow the definition of "free floating' operations [DERR 85].

_ Last but not least, there is a lot of implementation problems to solve. We use data base techniques in order to implement classes and objects. We shall check how a N1NF DBMS such as Verso will behave as a storage support. Verso is based on data filtering and we will see what must be added to the access structures of Verso in order to efficiently process object-oriented queries (mainly in the case of inheritance of data and/or methods)

REFERENCES :

[ABIT 84]

"Non First Normal Relations to Represent Hierarchically Organized Data," S. Abiteboul, N. Bidoit, In Proc. of ACM-SIGMOD Conf. on Principles of Database Systems, Atlanta, 1984, pp 191-200.

[ABIT 86]

"On the Power of Languages For The manipulation of Complex Objects," S.Abiteboul, C. Beeri, Preliminary report.

[AFSA 85]

"An Extensible Object-Oriented Approach to Database for VLSI/CAD," H. Afsarmanesh, D. Knapp, D. McLeod, A. Parker, Computer Science Department, U of Southern California, Los Angeles, TR-85-330.

[AGHA 86]

"A Message Passing Paradigm for Object Management," G. Agha in [DBE 85].

[BANC 86]

"A Calculus for Complex Objects," F. Bancilhon, S. Khoshafian, In Proc. Fifth Annual ACM Symposium on Priciples of Database Systems, Cambridge, Massachusetts, March 1985, pp 53-59.

[BIRT 73]

"Simula Begin," G. M. Birtwistle, O-J. Dahl, B. Myhrhaug, K. Nygaard, Auerbach , Philadelphia, PA, 1973.

[BOBR 80]

"Representing design alternatives," D. G. Bobrow, I. P. Goldstein, In Proc. Artificial Intelligence and Simulation of Behaviour Conference, Amsterdam, July 1980.

[BUNE 82]

"An Implementation Technique for Database Query Languages," P. Buneman, R. E. Frankel, R. Nikhil, In ACM Transactions on Database Systems, vol. 7, No. 2, June 1982, pp 164-186.

[CODD 71]

"Further Normalizations of the Data Base Relational Model," E. F. Codd., Courant Comp. Sc. Symp. 6, Data Base Systems, Prentice Hall, N.I., May, 1971, pp. 65-98.

[COLL 86]

"Les Formulaires Electroniques dans Tigre," C. Collet, RR TIGRE, Feb 1986.

[COPE 84a]

"Making Smalltalk a Database System," G. Copeland, D. Maier, In Proc. of the ACM-SIGMOD International Conference on Management of Data, Boston, June, 1984.

[COPE 84b]

"A Temporal Model for Bigtalk," G. P. Copeland, Servio Logic Corporation, Internal Report No 84, Portland, Oregon

[CROFT 85]

"Task Management for an Intelligent Interface," W. B. Croft, In [DBE 85]

[DADA 84]

"Integration of Time Version into a Relational Database System," P. Dadam, V. Lum, H-D. Werner, In Proc of the Tenth International Conference on Very Large Data Bases, Singapore, August, 1984.

[DAYA 85]

"PROBE - A Research Project in Knowledge-oriented Database Systems : Preliminary Analysis," U. Dayal, Et Al., Technical Report, CCA-85-03, July 1985.

[DBE 85]

"A Quaterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering," Special issue on Object-Oriented Systems, Vol. 8, N.4, December, 1985.

[DERR 85]

"Some Aspects of Operations in an Object-Oriented Database," N. Derrett, W. Kent, P. Lyngbaek, In [DBE 85]

[GOLD 83]

"SMALLTALK-80 The Language and its Implementation," A. Goldberg, D. Robson, Addison-Wesleyy, Reading, MA, 1983.

[GOLDM 85]

"ISIS : Interface for a Semantic Information System," K. J. Goldman, S. A. Goldman, P. C. Kanellakis, S. B. Zdonik, In Proc. of ACM-SIGMOD Conference on Management of Data,Austin, Texas, May 28-31, 1985.

[KING 85]

"A Database Design Methodology and Tool for Information System," R. King, D. McLeod, ACM Transactions on Office Information Systems, Vol. 3, No. 1, Jan 1985, pp 2-21

[KRIS 81] "A Survey of the BETA Programming Language," B. B. Kristenssen and al., Norwegian Computing Center Oslo, Norway, 1981.

[LAME 84a]

"Recursive Data Models For Non-Conventional Database Applications," W. Lamersdorf, In IEEE International Conference On Data Engineering, Los-Angeles, April 24-27th, 1984.

[LAME 84b]

"Language Support for Office Modelling," W. Lamesdorf, G. Müller, J. W. Schmidt, In Proc of the Tenth International Conference on Very Large Data Bases, Singapore, August, 1984.

[LYNG 84]

"A personnal Data Manager," P. Lyngbaek, D. McLeod, In Proc of the Tenth International Conference on Very Large Data Bases, Singapore, August, 1984.

[MAIE 85]

"Development of an Object-Oriented DBMS," D. Maier, A. Otis, A. Purdy, In [DBE 85]

[McCA 65]

"LISP 1.5 Programmer's manual," J. McCarthy, et al., 2nd ed. Cambridge, Mass.: the MIT press, 1965.

[McLE 85] "Object Managementt and Sharing in Autonomous, Distributed Data/Knowledge Bases," D, McLeod, S. Widjojo, In [DBE 85]

[NEUH 85]

"Objects And Abstract Data Types In Information Systems," E. J. Neuhold In Proc. of the IFIP TC2 Working Conference on Database Semantics, R. Meersman, T. B. Steel (Eds.), Hasselt, Belgium, Jan. 1985, North Holland.

[NIER 85]

"An Object-oriented Environment for OIS applications," O. M. Niertrasz, D. C. Tsichritzis, In Proc of the Eleventh International Conference on Very Large Data Bases, Stockholm, August, 1985.

[PIST 85]

"A Database Language for Sets, Lists, and Tables," IBM Wiss. Zentr. Heidelberg , Technical Report, TR 85.10.004, Oct 1985.

[SCHA 86]

"AN Introduction to Trellis/Owl," C. Schaffert et al. In Proc. of OOPSLA 86 Portland, Oregon, 1986. [SCHE 82]

"Data Structures for an Integrated Database Management and Information Retrieval System," H. J. Scheck, P. Pistor, In Proc of the Eighth International Conference on Very Large Data Bases, 1982.

[STON 76]

"The Design and Implementation of Ingres," M. Stonebraker, et al. , Transactions on Data Base Systems, 2, 3, September 1976.

[STON 85a]

"Inclusion of New Types in Relational Data Base Systems," M. Stonebraker, Electronics Research Laboratories, College of Engineering, U of California, Berkeley, Memorandum No. UCB/ERL M85/ 67.

[STON 85b]

"Extending a Data Base System with Procedures," M. Stonebraker, Electronics Research Laboratories, College of Engineering, U of California, Berkeley, Memorandum No. UCB/ERL M85/ 59.

[TSIC 85]

"Object Species," D. Tsichritzis, In [DBE 85]

[TSUR 84]

"On the Implementation of GEM - supporting a semantic Data Model on a relational Backend," S. Tsur, C. Zaniolo, In Proc. of ACM-SIGMOD Conference on Management of Data, 1984.

[ZANI 85]

"The Representation and Deductive Retrieval of Complex Objects," C. Zaniolo, In Proc of the Eleventh International Conference on Very Large Data Bases, Stockholm, August, 1985.

[ZDON 85]

"Object Management Systems for Design Environments," S. Zdonik, In [DBE 85]

[ZDON 84]

"Object Management Systems Concepts," In Proc. of the Second ACM-SIGOA Conference on Office Information Systems, Toronto, Canada, June, 1984.

[VERSO 86]

"VERSO : A Data Base Machine Based on Non 1NF Relations," J. Verso, RR No 523, INRIA, May, 1986.

[WEIS 85]

"An Object-Oriented Protocol for Managing Data," S. P. Weiser, In [DBE 85]

[WOO 85]

"An Object-based Approach to Modelling Office Work," C. C. Woo, F. H. Lochovsky In [DBE 85]

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique