PORTABILITY OF THE PROGRAMS
USING A DATA BASE MANAGEMENT SYSTEM

Michel DEMUYNCK - ELECTRICITE DE FRANCE - DER/IMA.
Centre de Documentation. 1 Avenue du Général de Gaulle 92140
CLAMART.

Patrick MOULIN - ELECTRICITE DE FRANCE - DSFJ/STI/DEMA
21, rue Joseph Bara 92132 ISSY-LES-MOULINEAUX

Serge VINSON - ELECTRICITE DE FRANCE - DSFJ/STI/DEMA
21, rue Joseph Bara 92132 ISSY-LES-MOULINEAUX.

ABSTRACT :

This paper deals with the problem raised by the portability of programs using a data base management system (DBMS).

The study of this problem led us to introduce a number of data access primitives.

Finally we study the translation of these primitives into the data manipulation language for our DBMS.

INTRODUCTION.

Whether one has to execute an application under various data processing system (software and hardware) or to transfer an application from one system to another, data management raises the rather complex problem of portability.

Already quite apparent in file management systems (FMS), the problem becomes even more crucial in DBMS. This paper is concerned with the solution to the latter problem.

We shall first discuss the design and programming of applications. This will suggest a solution to the above-mentioned problem. To put such an approach into practice, we need to define a set of general purpose primitives. In a second section, the primitives will be studied in detail and equal attention will be paid to the syntax and semantic of the primitives.

We shall then proceed to examine the problem raised by the transformation of primitives into statements which can be understood by the system.

Finally, it may prove interesting to analyze the scope of such a method and its possible applications.

1 - PORTABILITY.

1.1. Outline of the operation of a program using a DBMS.

The major functions of a DBMS are either one of the following :

- a) - Definition of the data and file structures.
- b) - Manipulation of data stored in the bases. This may be performed either by using a DBMS embedded in the "host language" (2) or through a command language, or else by the management of transactions between terminals and Data bases.

This paper will deal only with the problems raised by the use of a DBMS, in the "host language" approach. Thus, the pattern of operations of a program using a DBMS will be more closely examined.

To work with a data base in an application, the user has at his disposal a Data Manipulation Language (DML) which he invokes by means of a CALL, Macro-call to the DBMS, or simply by means of the commands (eg, READ, WRITE) already implemented in the language.

In general, this language allows those three kinds of actions :

- a) - Control commands (open, close,...)
- b) - Search commands (locate, reach, obtain, ...)
- c) - Modification commands (add, delete, replace, reorganize ...).

Due to the physical organization of data, and to the operating system environment, manipulation languages vary from one DBMS to another.

While the programs are made less dependent upon data organization by the use of a DBMS, they remain quite dependent upon the DML of the DBMS used.

1.2. The problem raised by portability.

The purpose of our study of portability is to reduce or eliminate the aforementioned dependance.

The issue is transforming a program using a DBMS 1 into a program using a DBMS 2. This raises the following problems :

- a) - The transformation of the data description in terms of DBMS 1's DDL into a description in terms of DBMS 2's DDL.
- b) - The transformation of data managed by DBMS 1 into data manageable by DBMS 2.
- c) - The transformation of the program operating with DML 1 into a program operating with DML 2.

The above must obviously be carried out without altering the semantics of program procedures, and data managed by the DBMS.

Problem (a) is outlined in (3), (4), (5) ; for our purposes, we shall consider that transformation (a) is performed manually by the data manager.

The solution to problem (b) depends on whether a solution exists or not to problem (c).

Finally, assuming, solutions to a) and b), the problem of portability comes down to transforming a program using a DML 1 into an equivalent program using a DML 2.

1.3. Application Design.

The model underlying the ANSI (1) report is a three-level model of Data Base Management systems : an internal schema, an external schema and a conceptual schema.

The major quality of the conceptual schema is that it is a semantically stable point. The external schemas are particular views of the conceptual schema and may evolve as long as the semantics are not altered. The impact of such modifications will be felt at the level of the interfaces between the conceptual schema and the external schemas. The same is true for the modifications of the internal schema which again will affect

only the interface between the internal schema and the conceptual schema.

One may thus measure both the value of the semantically stable point and the importance of the interfaces between the schemas.

As to the portability of programs using a DBMS, the problem results from the differences between the various DBMS, with respect to data description as well as data manipulation.

For the purposes of our problem, it would be interesting to isolate a stable point in relation to the internal data description and the internal program description, in the same manner as in the ANSI report, in which the conceptual schema is independent from the internal schema.

This stable point would thus be a description of data and programs at the conceptual level.

The translation of the data structure description into the internal level is then accomplished by interpreting this description. For programs this process must take into account the physical interpretation of the conceptual structure of the data.

This method can be represented as follows :

1.3.1. Conceptual model

For our purposes, the conceptual model must be able to define both the data structure and the processing of these data.

Moreover, the model must be completely free of physical implementation constraints. The "Entity-Relationship" models proposed in (13) and (14) have the advantage of being easily understandable. But the drawback is not taking into account the description of processing. (15) suggests an approach to represent processing through evolution rules. However, these concepts do not seem currently directly applicable.

As we started to work on the issue of portability, J.R. ABRIAL was developing a tool for rigorous description of data processing applications : the $\mathbb{Z}$ specification language(7). After experimental use demonstrated that it was best suited to our purposes, this tool was selected for the conceptual model. In order to make the remainder of this paper more easily understandable, we shall briefly describe the basic concepts of the language $\mathbb{Z}$.

1.3.1.1. Elements allowing the static description of application.

- Objects and sets of objects.
- Functions between sets of objects.
- Conditions pertaining to objects and functions.

Objects.

There exist several kinds of object denotations :

- . name of object, e.g. : *dupont*, 3
- . function evaluation result, e.g. : *ageofcar* (*c*)
- . object variable, e.g. : *p*
- . "tuple" (i.e., ordered collection) of objects, e.g. : (*march*, 10, 1977)
- . conditional objects.

A distinction is made between concrete objects (e.g. : person *p*) and abstract objects (e.g. : *dupont*, 3 etc.) (10).

Object identifiers begin with a small letter.

Sets of objects.

There are various ways to designate sets of objects, such as :

- . name of set e.g. : *Name*, *Person*, etc...

- . function evaluation result e.g. : *Personnamed (dupont)*
- . set variable
- . cartesian product of sets e.g. : *Day * Month * Year*
- . union, intersection, difference of sets.

Set identifiers begin with a capital letter.

Functions :

- single-valued functions. The identifiers of single-valued functions begin with a small letter.

The value of the function is an object of the target set.

e.g. *Person* *name of person* (1) *Name*

A single-valued function is total if $\forall a. \exists b$ such that $f(a) = b$ (notation : $f^*(I)$)

A single-valued function is partial if it is not total. (notation : $f^*(O)$)

- multi-valued functions. The identifiers of multi-valued functions begin with a capital letter.

The value of the function is a subset of the target set.

e.g. *Name* *Personnamed (n,m)* *Person*

The numbers in parentheses (n and m) are the minimum and maximum numbers of elements in the image of an element of the source set.

Conditions :

The static description of application is complemented by stating the conditions related to the objects and functions. These conditions are expressed by means of mathematical symbols.

e.g. $\forall c \in Car. ageofcar (c) < 10$

1.3.1.2. Processing_description_language.

This language manipulates the preceding elements and is made up of the following types of phrases :

- assignment phrases for objects or sets of objects :

e.g. *name of person (p) ← dupont*
car of person (p) : ∃c

- conditional phrases :

e.g. *if condition then action else action end*

- iterative phrases :

e.g. *for c ∈ Car ... action ... end*
while condition ... end

1.3.2. Internal model

Data are implemented and accessed by using the possibilities offered by the existing DBMS.

The DBMS are either of "hierarchical - network" type, (CODASYL, SOCRATE (9), IMS (8),...) or of relational type (system R, PRTV (11) , SYNTEx (12)).

The choice of an implementation involves two aspects :

- implementing sets of objects,
- implementing relations.

In order to examine what happens at the internal level, to object's sets of conceptual level, the notion of object's representation should be introduced.

This representation may portray the objects in two ways :

Either, the objects are given a unique representation, in other words, a one-to-one correspondence between the conceptual model set of objects and the internal model set of objects.

These sets may be implemented in, for example, the form of root segment in IMS or in the form of entity in SOCRATE.

Or, the objects are given a multiple-potential representation ; in other words, for one conceptual model object, there will be at least one corresponding internal model

object.

These sets may be implemented, for example, as dependent segments or fields in IMS, or, in the case of SOCRATE, as interleaved entities or entity characteristics.

Let us examine the sets of objects of the conceptual model. It has been seen that there are two types of objects : abstract objects which have a value of, for example: 25, toto ... and concrete objects which are compositions of abstract objects, able to represent a physical reality as for example : the person p. The sets of abstract objects will be translated at the internal level by a multiple-potential representation. On the contrary the two possibilities of representation are applicable to the sets of concrete objects.

1.3.3. External model

The external model represents the view of what is real for a given user. This model is made up of a sound and consistent subset of objects, sets and functions of the conceptual model.

1.4. Method of resolution.

The conceptual schema of a data base consists partly of an object description and the relations linking these objects, and partly of the non-algorithmic specification of the processing to be performed on the data.

The processing description uses the objects and relations described in the conceptual data structure. It should be said that, at this stage of design, no other physical contingency occurs, as the problem of the physical implementation of data is dealt with at the level of the internal schema (ref. § 1.3).

From this step, it is necessary to go on to convert this processing formulation into procedures. These procedures are expressed by means of a pseudo-code type of language. (Level Z1 of the Z language).

This phase leads, therefore, to a processing description in algorithmic form, which, as it is important to point out, manipulates the objects and relations of the conceptual structure.

The transition to the following phase will now depend on the options for implementing

data and programs. We then enter into the phase : internal schema description :

- . description of internal data structure,
- . conversion of the pseudo-code into programming language.

This transition to the internal model is the most common one. It corresponds to the well-known approach : systems analysis-programming. All the same, one may wonder if it would be possible to modify this approach, and rather attempt to convert the programs, leaving the data in their conceptual form. The programs would then be clearly independent of the internal data structure. One solution consists in converting the pseudo-code phrases used to manipulate the data, into primitives.

The primitives act as an interface between the concepts contained in the conceptual model and the internal zones of the procedure.

The other procedure phrases may then be directly translated into program phrases by the use of a standard programming language. These phrases will use only strictly portable verbs or variables : as the definition of primitives is equally standard, the program obtained will be portable by construction.

The diagram of 1.3 would then become :

In order to analyze the process in greater detail, let us take the example of a management program using a data base. "Data base" meaning a set of files, we could as

well be concerned with conventional files as with a data base managed by a DBMS.

Three types of instructions appear in this program :

- a) - Computing and transaction instructions, e.g. : statement of algebraic expression, assignment, processing of character chains...
- b) - The control structures for the program, e.g. : if -then -else, do while,...
- c) - CALLs to the DBMS or to access methods and, when describing internal data to the program, declarations of structure, e.g. : *CALL DBMS (selection filter)* or *READ FILE*.

In analyzing the three types of instructions, it appears that the statements of type (a) are independent of the Data Management system. The use of a standardized programming language would suffice to make this part portable.

For the code of type (b), only those statements which express the logic associated with computing and processing areas (type a), are portable.

For the other instructions are associated with the use of a data management system, they are not necessarily portable (this is the case for diagnosis tests, access description, etc...).

The code of type (c) is totally dependent on the manipulation language and the physical implementation.

A transformation of procedures written in pseudo-code into programs expressed in a standard programming language, and data access primitives, leads to the non-portable procedure statements being systematically masked by the primitives.

2 - DEFINITION AND USE OF PRIMITIVES.

The methodology selected leads to design programs using primitives which manipulate the elements defined in the conceptual schema.

The procedure will be programmed by using a standard programming language.

For this study, the COBOL ANS (6) language will be used.

The different functions that the primitives must fulfill are :

- creation of objects or links,
- deletion of objects or links,
- modification of links,
- acquisition of objects.

The creation of objects will be defined by the primitive *CREATE* ; the creation and modification of links, by the primitive *LINK* ; the deletion of objects, by the primitive *KILL*; the deletion of links, by the primitive *UNLINK*, and the acquisition of objects, by the primitive *GET*.

Set traversal primitive will be defined as *OPEN*, *NEXT* and *CLOSE* and an object variable primitive as : *DEFINE*.

These primitives essentially manipulate two types of parameters : * COBOL variable :

- CR : return code composed of a diagnostic and/or of an anomaly-processing address.
- Zc : communications area which can be understood by the rest of the procedure and interface with the definition at the conceptual level of data.
- Adress : disconnecting address at the end of set traversal.

* Z variable :

- objects. Note : Z variables are marked \$ object in order to distinguish them from COBOL variables.
- sets.
- relations.

Let us examine the syntax and semantics of the primitives in greater detail.

Note : The Z_0 specifications of the problem relating to the different examples are given in the Appendix.

2.1. Creation of objects or links.

The creation of objects or links refers either to the creation of an object a in a set A or to the creation of a link between two objects a and b , where b is defined as the result of the multi-valued function, $F(a)$.

2.1.1. Creation of an object.

Suppose that a person, p , is to be created in the set *Person*.

The Z_1 notation is :

$p \leftarrow \text{create} (\text{Person})$

The syntax of the corresponding primitive will be :

CREATE (*cr*, $\$p$, *Person*)

Where : *CR* is the return code,
 $\$p$ the Z variable designating the new element of the set,
Person is the set defined in Z_0 .

2.1.2. Creation of a link.

* Suppose that the date of birth of a person is to be created. The phrase which will be found in the Z_1 description is :

dateofbirth (p) $\leftarrow$ (*january, 31, 1977*)

The syntax of the primitive will be :

LINK (*cr*, *dateofbirth* ($\$p$) = Zc)

where : *cr* is the return code,
 $\$p$ the Z variable designating the person,

$\mathbb{Z}c$, the communications area belonging to the host programming language.

Note : In this case, $\mathbb{Z}c$ will contain a tuple composed of :

(january, 31, 1977).

* Suppose that a link is to be created between a person, p , and a car, c .

The corresponding $\mathbb{Z}_1$ expression is :

Carofperson (p) : $\ni c$

The primitive will be written :

LINK (cr , Carofperson ($\$p$) = $\$c$)

Now, assume that the creation of the link between the person p and the car, defined by its number N , was desired.

The following would have been written :

LINK (cr , Carofperson ($\$p$) = carofnumber (N)).

2.2. Deletion of objects or links.

Just as for the creation of objects or links, the following can be considered in the deletion of objects or links :

- either, an object belonging to a set,
- or, a link between two objects, each belonging to a different set.

If an object a of a set A is deleted, all the links stemming from or leading to this object will consequently be broken.

It is therefore appropriate to examine the type of implementation of the resulting sets of objects.

If the set of objects has a unique representation, the resulting object is not destroyed. If the set has a multiple-potential representation, the internal represen-

tation of this object is destroyed. It is evident that the links will be destroyed, if they stem from or lead to this object. This process may therefore start off a chain reaction of deletions of links and object representations.

2.2.1. Deletion of an object.

★ Suppose that a car c in the set, Car , is to be deleted. The corresponding Z notation is :

kill (c)

The syntax of the primitive will be :

KILL ($cr, \#c, Car$)

This primitive will have the effect of deleting the object c as well as all its links with the sets *Person*, *Make*, *Numbers* and *Number*.

★ Suppose that a person p of the set *Person* is to be deleted. The Z notation is :

kill (p)

The corresponding primitive is :

KILL ($cr, \#p, Person$)

This primitive will have the effect of deleting the object p as well as all its links with the sets *First-name*, *Name*, *Sex*, *Year*, *Month*, *Department*, *City*, *Day*, *University*, *Branch* and *Distinction*.

Moreover, if all these sets have a multiple-potential representation, this primitive will thus also delete the representations concerning the objects.

2.2.2. Deletion of a link.

★ Suppose that a person p , pays a rent n at date $d1$. If at date $d2$ this person becomes the owner of the house, he (or she) will no longer pay rent. The corresponding expression in Z will be :

$rent(p, d2) \leftarrow nil$

Which will be translated into the following primitive :

$UNLINK(cr, rent(\$p \$d2) = nil)$

* Suppose that the link between a car c and the person p is to be deleted. The corresponding Z expression will be :

$Carofperson(p) : \cancel{\$c}$

The translation of this expression in primitives will be :

$UNLINK(cr, Carofperson(\$p) = \$c)$

This primitive will have the only effect of deleting the link between *Car* and *Person* if the set *Car* has a unique representation.

The car could have been designated by its number, using the relationship *numberof-car* between the sets *Car* and *Numbers*.

The primitive would have thus become :

$UNLINK(cr, Carofperson(\$p) = carofnumber(N))$

2.3. Modification of links.

Two cases are involved in link modification :

- a) - If a link is to be modified between two objects connected by a single-valued function, the related primitive will be LINK and its use will be identical to that of a link creation. (§ 2.1).

For example, if the age of the car designated by c , moves from 3 years to 4 years, the Z expression of this link modification will be :

$ageofcar(c) \leftarrow 4$

The corresponding primitive will be :

LINK (*cr*, *ageofcar* ($\$c$) = 4)

- b) - Link modification between two objects connected by a multi-valued function does not involve actual modification. What it does is in fact a link deletion followed by a link creation.

Indeed, suppose that a person *p* no longer possesses the car *c* , but now possesses the car *c'* : the *Z* expression for this action will be :

Carofperson (*p*) : $\#c$

Carofperson (*p*) : $\exists c'$

and the natural translation into primitives will yield :

UNLINK (*cr*, *Carofperson* ($\$p$) = $\$c$)

LINK (*cr*, *Carofperson* ($\$p$) = $\$c'$)

2.4. Obtaining objects.

Objects are obtained through the primitive GET. The purpose of this primitive is to fetch certain objects belonging to sets defined in Zo.

Just as for the preceding primitives, several objects can be obtained by a single call of the primitive.

Example : One wishes to know the make of the car designated by *c*. This information can be obtained by simply writing : *makeofcar* (*c*). There is therefore no corresponding *Z*₁ expression.

The syntax of the primitive will be :

GET (*cr*, *makeofcar* ($\$c$), $\$c$)

2.5. Set traversal.

It is quite often necessary in programs, to perform an action not only on an object, but also on a set of objects. This set of objects can be a set in the *Z* sense of the term, or the result of a multi-valued function.

In order to obtain the successive elements of a set, it is necessary to define a set traversal variable. This variable can be used by the previously examined primitives.

- a) - The primitive *OPEN* permits the definition of a common *Z* variable.
- b) - The primitive *NEXT* is used to initiate the evolution of the common *Z* variable, so as to point to a new object of the set.
- c) - The primitive *CLOSE* is useful for indicating the end of traversal, liberating the *Z* variable defined at the time of *OPEN*.

Example : Suppose that the age of each car is to be increased by one year. The corresponding *Z₁* pseudo-code is :

```

c ← openget (Car)
while c ≠ nil
  ageofcar (c) ← ageofcar (c) + 1
c ← get (Car)
end

```

When translated into primitives, the procedure becomes :

```

OPEN (cr, #c, Car)
DEB. NEXT (cr, #c, Fin)
GET (cr, ageofcar (#c), #c)
ADD 1 TO #c
LINK (cr, ageofcar (#c) = #c)
GOTO DEB.
FIN. CLOSE (cr, #c)

```

Note : If the only cars whose ages must be increased are those belonging to the person *p*, then the *OPEN* primitive will be :

```
OPEN (cr, #c, Carofperson (#p))
```

(No other change would occur in the rest of the procedure).

2.6. Variable definition.

Just as in the preceding paragraph a set traversal variable was defined, it may be

useful to define an object variable designating a particular object. This is the case where, for example, reference is made several times in a program to the same object.

This is the role of the primitive *DEFINE* . This primitive allows both a simplified writing and the possibility to gain access (logical, then physical for the internal model).

For example, if several references are to be made (by using *GET* ,*LINK*, *UNLINK*...) to the name of the person *p*, we shall write :

```
DEFINE (cr, $n, nameofperson ($p))
```

Thus, each time that a reference is made to the name of the person *p*, *\$n* will be included in the primitives.

3 - TRANSLATION OF PRIMITIVES

3.1. General principles.

We've already investigated the static aspects of the application in Z_0 , and its dynamic aspects in Z_1 . The next step will be to translate the resulting Z_1 pseudo-code into COBOL programs which operate on the Z_0 conceptual model. These programs are made up of primitives representing the access to data and of COBOL instructions representing data manipulation.

These programs are perfectly independent of the internal model chosen to represent the data ; in other words, independent of the internal structure of the data in the DBMS and independent of the DBMS itself.

The next step will consist in describing an internal model and translating the access primitives in accordance with the internal model.

Describing the internal model involves converting the Z_0 conceptual structure, into an internal schema, dependent on the DBMS selected and on its own specifications as well as on the system analyst's ideas about implementation.

This schema will be described by means of the data description language (DDL) of the DBMS selected. Discussion of this problem is not included in this study.

In order to be translated, the primitives (*GET*, *CREATE*, *LINK*...) are converted into a language which is understandable by the DBMS. Such a translation requires knowledge of the choices made for the internal structure.

Once the primitives have been translated, they will be entirely dependent on this internal structure. A change in DBMS or in structure will call for retranslation of primitives.

We shall now turn our attention to the translation of a primitive.

A primitive consists of a name and a list of parameters. These parameters are :

- the status returned after completion of the operation,
- the elements of the conceptual model invoked by the primitive,
- the communication area.

The name of the primitive indicates the kind of action which has to be performed.

This action, except for the definition of variables and set traversal(s), will always be punctual and its effect will be limited to those objects and links of the conceptual model which are designated in the primitive.

It is thus necessary to evaluate the Z -expression of the primitive.

The result of this evaluation will be the access path to the internal representation of an object, or a link of the conceptual model. The result will be called the *context* of the Z -expression.

This evaluation will be performed by a dedicated automaton . This automaton obviously requires some information ; in particular, it must know how sets and functions of the conceptual model are represented in the internal model.

This information, needed by the translator, is included in some tables called *translation tables*.

These tables are attached to an internal representation of the conceptual model. Any change in the internal structure of the data involves changes in the translation tables. Other informations will as well be necessary to evaluate Z -expressions. These are informations related to Z -variables. For instance, if we want to evaluate the expression

nameofperson ($\$p$) ,

then we need information on the function *nameofperson* which is contained in the translation tables and we have to identify which person is designated by the variable $\$p$. This information will be found in a particular table, associated with the variable. We call it the *context of variable*.

3.2. Description of translation tables.

These tables are made of different kinds of information :

- informations about functions :
 - . range and domain
 - . allowable access paths
 - . translation of the function
 - . type of the function (single - or multivalued)

- informations about sets :
 - . internal representation (field, segment, record,...)
 - . mapping of name
 - . designation of I/Ø buffers
 - . type of representation (unique or multiple)
 - . coding of data.
- informations about buffers :
 - . description of I/Ø areas
 - . description of access paths.

3.3. Variable context.

Z-variables occurring in Z-expressions, written as "\$ variable", are introduced by means of the primitives for variable definition : e.g. CREATE,DEFINE,OPEN.

We associate a variable context with each Z-variable. This context contains :

- a definition of access to the Z-object internal representation
- a pointer to the active access path
- an indicator of the validity of the Z-object internal representation
- an indicator of the existence of the Z-object internal representation.

3.4. Evaluating Z-expressions.

The purpose of evaluating a Z-expression is to find a context for the expression. This context will permit physical access to the Z-object internal representation. To this aim, we must build a block, the functions being taken one by one from the most internal Z-object of the expression to the left most function. This block will be the definition of the access path. The evaluator builds this block by visiting the translation tables for each function.

3.5. Code generation.

Using the action designated by the name of the primitive, the result of the Z-expression evaluation, and the informations contained in translation tables, the translator will generate Cobol and DML code corresponding to the primitive.

CONCLUSION

We have suggested in this paper a system of primitives which allows programs using a Data Base Management System or a File Management System to be portable. Our approach is limited to programs and does not at all deal with the portability of data structures (it is the hypothesis of this paper that this is dealt with manually).

Our system is founded on the conceptual model chosen (used as the fixed point of the application) : the Z specification language, designed by J.R. ABRIAL.

The primitives have been tested on a few programs.

Studies are now under way to examine the problem of translating these primitives into executable code.

This system, which solves the problem of program portability, may as well be of some interest in the field of distributed data bases, at the level of the global view of local data bases.

°
° °

ACKNOWLEDGMENTS

We would like to thank both, J.R.ABRIAL for all the advice given throughout the development of this system and for the interest he always manifested during our many discussions and J.RANDON for his fruitful remarks and for his constant optimism.

REFERENCES.

- (1) ANSI/X3/SPARC - Study group on DBMS. Interim Report (75.02.08).
- (2) CODASYL - Feature Analysis of generalized DBMS - May 1971.
- (3) M. ADIBA, C. DELOBEL, M. LEONARD - A unified approach for modelling data in logical data base design - Proceedings IFIP TC 2 Working conference - Freudenstadt - January 1976.
- (4) M. ADIBA, C. DELOBEL - The problem of the cooperation between different DBMS. Proceedings IFIP TC 2 Working conference - Nice, January 1977.
- (5) H. TARDIEU, H. HECKENROTH, D. NANJI - Etude d'une méthodologie d'analyse et de conception d'une base de données - Mai 1975 - Research contract IRIA N°74180.
- (6) ANSI/X3 - American National Standard Programming Language COBOL (23-1974).
- (7) J.R. ABRIAL - Z specification language manual - Unpublished.
- (8) IMS/V2 - General Information Manual IBM (SH20 - 0912 - 4).
- (9) SOCRATE - Manuel d'utilisation et Manuel d'opération. ERIA ECA Automation.
- (10) J.R. ABRIAL - Data semantics IFIP TC 2. Working Conference - Cargese 1974.
- (11) S.J.P. TODD - The Peterlee Relational test vehicle - A system overview - IBM system journal n° 4 1976.
- (12) R. DEMOLOMBE - Syntex - Data structure models for information systems. Institut d'informatique de Namur 1975.
- (13) P.P. CHEN - The Entity - Relationship Model - Toward a Unified view of Data. ACM TODS Vol. 1 N°1 (March 1976).
- (14) P. MOULIN, J. RANDON, S. SPACCAPIETRA, S. SAVOYSKI, H. TARDIEU, M. TEBOUL - A conceptual model as a data base design tool. Proceedings IFIP TC 2. Working Conference - Freudenstadt. January 1976.
- (15) G. BENCI, F. BODART, H. BOGAERT, A. CABANES - Concepts for the design of a Conceptual Schema - Proceedings IFIP TC 2. Working Conference - Freudenstadt - January 1976.

APPENDIX : CONCEPTUAL SCHEMA OF EXAMPLES.

The examples found throughout this paper refer to the following conceptual model.

1 - THE SETS OF OBJECTS ARE.

Person
 Name
 Firstname
 Car
 Make
 Numbers
 University
 Branch
 Distinction
 Date
 Number
 City
 Department
 Month
 Year
 Sex
 Day

2 - THE LINKS BETWEEN OBJECTS ARE EXPRESSED BY THE RELATIONSHIPS.

Person ss number (1) Sex x Year x Month x Department x City x Nb
 person of ss number (0)
 Person Diploma (0, -) University x Branch x Distinction
 Person x Date rent(0) Number
 Person Carofperson (0, -) Car
 Car ageofcar (1) Number
 Car makeofcar (1) Make
 Car numberofcar (1) Numbers
 carofnumber
 Person dateofbirth (1) Day x Month x Year
 Person nameofperson (1) Name
 Personnamed (0, -)
 Person firstnameofperson (1) Firstname