DIAM II: The Binary Infological Level and Its Database Language - FORAL

Michael E. Senko

Mathematical Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

The notion of dividing complex problems into hierarchic levels of abstraction has a long history. Recently this approach under the name, structured programming, has been used in a pragmatic way to simplify the structure of procedural programs. The notion has also been used in the data base area - notable early papers include those of Madnick (1) and Meltzer (2). In these early papers, level structures (for example, involving logical and physical levels) were also relatively pragmatic creations.

One new aspect of the DIAM I architecture (3) was that it defined a detailed, relatively formal basis for four levels of abstraction and the mappings between these levels. Each level had a small set of primitive generalized building blocks, each with its own small permanent set of parameters. By specifying values for these parameters, the user could define essentially any reasonable data structure and file organization he wished for his stored information.

In DIAM I, the top level, the Entity Set Level was based on a data structure independent form of single level "logical record". Following Meltzer (2) and Mealy (4), DIAM I emphasized a study of the semantic aspects of this level that was designed to be the End-User's data structure independent view of the real world.

The lower levels were designed to provide progressively more detailed definitions of the stored data structure. The second level, the Access Path Level, provided a generalized set of possible access paths for connecting specified pieces of Entity Set Level "logical records". It allowed the user to select a specific set of efficient access paths from this generalized set. The third level, the Encoding Level, utilizing a Basic Encoding Unit (BEU), provided for the selection of a specific bit encoding for the access paths from a generalized set of encodings ranging from generalized list structures with variable length fields to contiguous fixed length fields in contiguous fixed length records. The fourth level, the Physical Device Level, provided a generalized access method model for record storage management.

Given this generalized model for data structures there was still a need for a non-procedural, data structure independent language for accessing information at the Entity Set Level. A major consideration was that it be designed for use by non-programmers. A first approximation to this language was provided by the Representation Independent accessing Language (RIL). This language was supplemented by a procedural language, Representation Dependent Language (RDL), for following access paths at the Access Path Level. Finally, preliminary algorithms were published for selecting optimal search paths for processing of RIL statements and subsequent compilation of the statements into optimal searches in RDL.

Recently, the ANSI SPARC Committee (5) has published additional work in defining level structures. This group gave added stature to an External Schema (or End-User) Level above the four levels formally defined in the DIAM I structure. It also gave clear recognition to the requirements for a non-redundant, data independent Conceptual Level describing the real world enterprise.

Using this new work on level structure, along with the work of Levien and Maron (6), Langefors (7), Ash and Sibley (8), Sundgren (9), and Abrial (10) on binary representations, the DIAM I model has been modified and improved. The new model, DIAM II (11, 12) contains an *End-User Level* which fulfills the requirements desired by the External Schema of ANSI SPARC and an *Infological Level* (ANSI Conceptual Schema) based on a modified form of binary association. This central level with its language interface, FORAL, is a fundamental improvement in semantic integrity over the Entity Set Model of DIAM I and the somewhat similar n-ary relational model.

In this paper, we will focus on the characteristics of the Infological Level and FORAL, the user-oriented language for processing in binary association networks. In this discussion, we will cover the unique employment of "FORAL Context" for limiting the user to transaction statements that are semantically meaningful in terms of his information network. Finally, we will briefly note the new kind and level of challenge that the selection of optimum search paths for FORAL in a DIAM II environment poses for the optimizing compiler area.

THE GENERAL LEVEL STRUCTURE OF DIAM II

In Figure 1, we present the DIAM II "Double Funnel" diagram. The funnels opening toward upward and downward indicate that the system supports a wider variety of data structures at its upper and lower levels than it does at its central Infological Level.

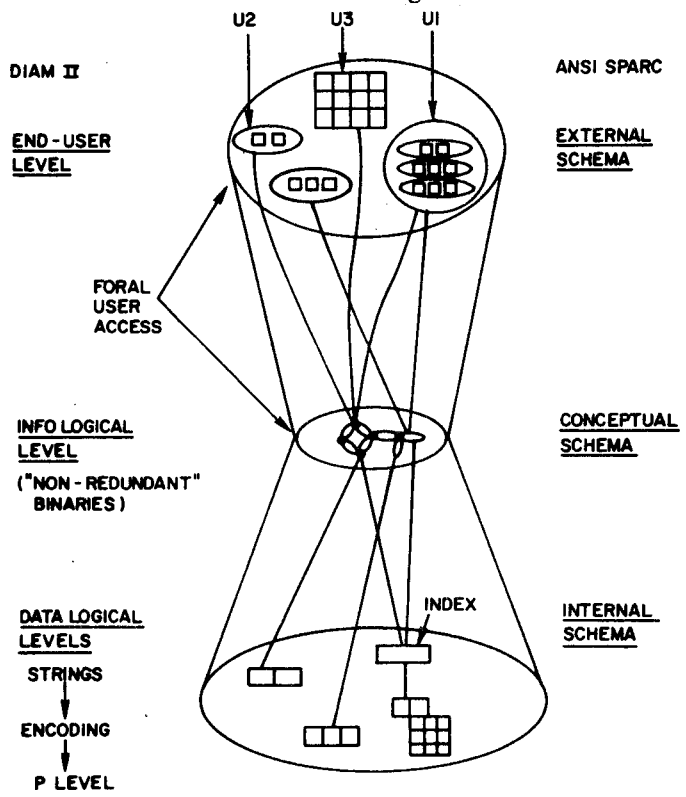


Figure 1 System Overview

The upper End-User Level is designed to support many user and many program (FORTRAN, COBOL, DBTG, IMS, etc.) data structure views. These views are designed to provide user efficiency and compatibility with the data structures of existing programs. Similarly the three lower levels provide a general set of file organization views which can be used to support efficient use of the computer. The Infological Level, in agreement with ANSI SPARC requirements, is designed to be as non-redundant as possible in its representation of the relevant enterprise.

Most details of the Infological Level and its specification have been presented earlier (11,12). In the following section, we will review the essential aspects of the level and present more recent insights. We will then concentrate on a relatively complete presentation of the query language aspects of FORAL. Maintenance aspects will be presented in a later paper.

THE INFOLOGICAL LEVEL

In DIAM II terminology, the real world is composed of Entities which are objects, things or other concepts of interest to the enterprise. Entities are often assigned to sets called Entity Sets on the basis of common properties. These sets have names like PERSONS, DEPARTMENTS, PARTS, etc.

Unambiguous Names for Entities

In our information systems, we seldom operate directly on the entities of the real world. Instead, we process names which stand in place of them.

In DIAM II, our Infological Level starts with application-oriented sets of names for entities. These sets are called Identifiers and the names within the sets are called Identifier Values. For a particular application, the Identifiers might be named EMP-NO, DEPT-NO, PART-NO, etc.

Each Identifier Value within a particular Identifier then will stand for one entity in the real world. For example, an Identfier Value, 012345, of the Identifier, EMP-NO, might stand for a particular employee. And an Identifier Value, 420, from DEPT-NO might stand for a Mathematics Department. Therefore, in the system, by indicating an Identifier and a particular Value we can unambiguously refer to a unique entity in the real world.

Descriptions of Entities

DIAM II *describes* real world entities by storing *representations of facts* about the entities. Such a Fact Representation is an association between names for two entities. For example, a particular Fact Representation might associate DEPT-NO, 420 with EMP-NO, 012345. In Figure 2, we present a specific fact representation. As we can see, in each circle, we have a name for an Identifer and an Identifier Value. The contents of each circle therefore refers to a specific entity in the real world.

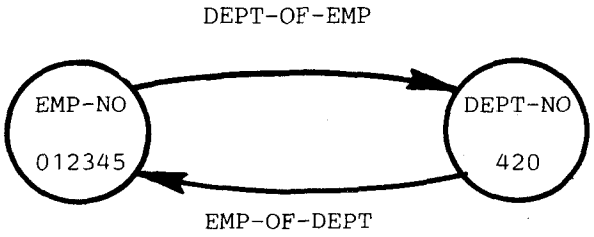


Figure 2

An important aspect of facts is that they can be interpreted using either of two different "Contexts". For example, if we use the Context, EMP-NO 012345, then the fact can be interpreted as giving the value of one of the employee's attributes "DEPT-OF-EMP is 420". On the other hand, if we use the Context, DEPT-NO 420, the value of one of its attributes "EMP-OF-DEPT is 012345".

In using FORAL, the user will indicate his "Context" and then he can have the system access specific attribute values for him by mentioning the appropriate attribute names. For example, when he is in the context, EMP-NO, and gives the attribute name, DEPT-OF-EMP, the system will access the value at the end of the attribute arrow, in this case, 420. Conversely, when he is at DEPT-NO, 420, and gives the attribute name, EMP-OF-DEPT, the system will access the attribute value, 012345.

In the case where a particular entity has many values for a particular attribute (for example, a department has many employees) the system will access all the attribute values.

Semantic Integrity in Data Structures

As we noted, work on binary relations is not new to the data base area. The recent interest is due to the realization that binary networks represent the actual meaningful associations between real world entities much more faithfully than physically oriented record formats. Unlike record or n-tuple structures where some data element associations internal to and between records correspond to facts and others do not, each fact in the real world can be represented by a binary association without creating additional spurious associations. For example, if we have the n-tuple in Figure 3:

Figure 3

The association between MANAGER-NAME and SALARY is of the same form as the association between SECRETARY-NAME and SALARY and it is difficult to determine which association represents a real world fact and which is spurious.

This is not a problem in Figure 4 where no direct association exists between SECRETARY-NAME and SALARY.

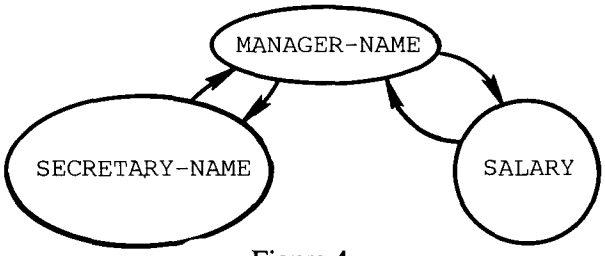


Figure 4

In the above examples, we have used only a single name to identify each entity. In fact, we can consider this name to be the value of an attribute for the entity. For example, a person might have the attribute, EMP-NO is 012345 which is used to identify him. He might also be identified by the attribute, SOCIAL-SECURITY-NO is 345-34-5674. There exist cases where entities are identified by combinations of their attribute values. For example, DATE may be identified by its attributes, DAY-OF-DATE, MONTH-OF-DATE, YEAR-OF-DATE. There is some temptation to assume that this identifier should be represented by an n-tuple, but it seems more semantically useful to consider the identifier to be a combination of binary associations. This becomes more clear when we look at the possible identification of a transaction by the PART-NO, SUPPLIER-NO, and PROJECT-NO corresponding to three of the entities involved in the transaction.

If, as in Figure 5a, we have a QUANTITY associated with the transaction identified by an n-tuple, it is not clear that we cannot perform operations to remove one of the elements of the identifier and obtain the implication that QUANTITY refers to the quantity of the part supplied by the supplier to all projects. In Figure 5b, it is clear that QUANTITY is associated with the transaction and as we shall see, FORAL will normally preserve the semantics of this association.

Figure 5a

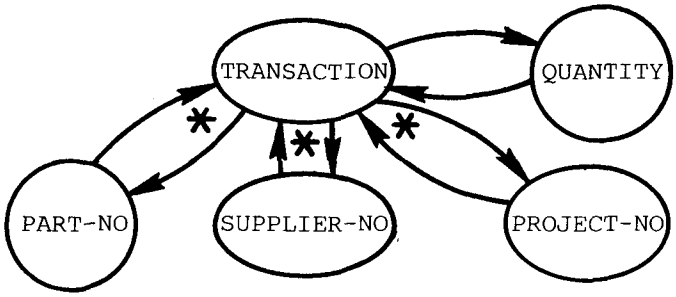


Figure 5b

The Infological Level Catalog

To use FORAL, the user must, of course, know what identifiers and attributes are available in the system. He can get this information from a normal catalog listing the identifiers and their associated attributes, or he can use a DIAM II - FORAL type diagram such as the one shown in Figure 6.

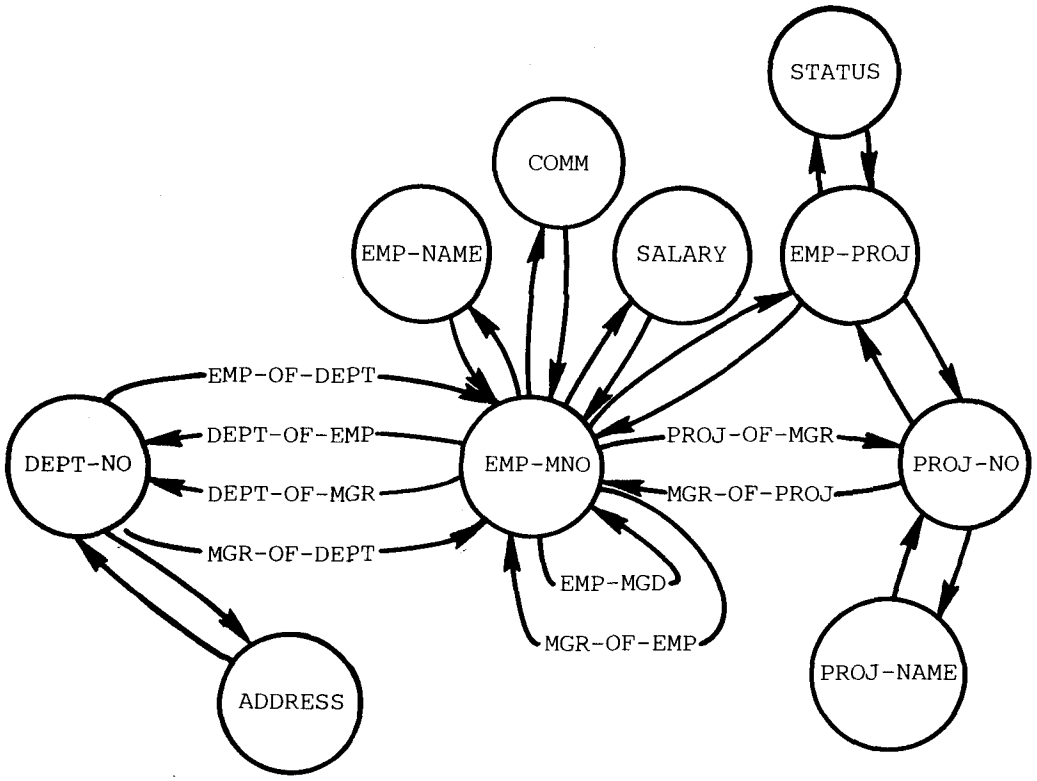


Figure 6 DIAM II - FORAL Type Diagram

In this diagram, the names of Identifiers are shown in the circles. Each pair of arrows then corresponds to a type of fact with a specific meaning. We can see this in the case of departments and employees. There are two fact types involving departments and employees. One fact type associates "employees (members)" and "departments"; the other associates "employees (managers)" and "departments".

Another aid to semantic integrity that binary structures provide is exemplified by the various types of managers portrayed in the DIAM II - FORAL type diagram. In an n-tuple describing an employee, there is the possibility of naming one of the components, MANAGER, without distinguishing which type of manager the designer intends. In the binary structure, managers of departments can only be portrayed by connecting DEPT-NO's and EMP-NO's, project managers by connecting PROJ-NO's and EMP-NO's, etc. This property requires the designer and the user to be more explicit about the meaning of data entries.

There are other problems with the naming of components of n-tuples and with redundancy of fact representations which are also resolved by using binary structures, but we will not go into detail here.

Semantic Integrity in Languages - FORAL Context

Even if we have a structure that represents the semantics of the real world faithfully, we can still have languages that do not make use of the semantic network. In particular, most mathematical notations treat the binary relations as separate tables and the user must, therefore, explicitly specify operations that connect one binary relation to another. In some cases, the connections may be meaningful in the real world and in others they may not be. Using our FORAL type diagram as an example structure, it would be possible to connect the DEPT-OF-EMP association with the the PROJ-NAME-OF-PROJ-NO association by specifying the DEPT-NO = PROJ-NO. This connection has little, if any, meaning.

FORAL with its Contextual interpretation of statements uses the semantic network to a greater degree - especiallly in the implicit creation of connections through the Identifier presently in Context. In particular, each attribute mentioned at any point in a FORAL statement must be directly related to the the Identifier in Context. For this reason, connections based on two different identifiers such as those in the example above cannot occur. In addition to avoiding non-meaningful connections this limitation gives us some other useful properties.

In general, each attribute in the DIAM II system must have one name that is unique across the system, but we can use short NICnames (Names In Context) for attributes when we use FORAL. The reason that this is possible is, as we mentioned above, that at any point in the interpretation of a FORAL statement, there is only one Identifier in Context, and the user can only ask for attributes that are directly related to that Identifier. This means that the attribute NICnames he uses need to be unique across the attributes of the Identifier in Context.

Corresponding to common usage in English, we can for one of the attributes between two identifiers use the name of the Identifier being pointed to. If our present context is DEPT-NO, then we can use EMP-NO as the NICname of the attribute which accesses the members of the department. (All the attribute arrows in the diagram that do not have names attached to them, therefore, can be accessed by using the name of the Identifier that they point to.)

In many cases we can use even shorter NICnames. For example, in the context of PROJ-NO, we can simply use NAME to access PROJ-NAME. Similarly, when we are in the context of EMP-NO, we can use NAME to access EMP-NAME. This capability provides a new convenient form of variable naming which can be added to the naming techniques whose scope is larger than a language statement.

FORAL QUERY LANGUAGE

It seems possible to characterize a large fraction of the transactions in a data base system as the input or output of information on a single class of entities (for example, a report on DEPARTMENTS). This information includes the identifiers for the individual entities in the class and frequently includes values for direct or indirect attributes of the entities.

> (By direct attributes, we mean those directly connected to the entity. For example, EMP-OF-DEPT. By indirect attributes, we mean those indirectly connected. For example, SALARY-OF-EMP of the department or even more indirectly connected, NAME-OF-PROJ of a project managed by an employee of the department. All of these are, in the final analysis, attributes of the department.)

FORAL has been designed to ease and encourage semantically correct specification of these types of transactions.

General Statement Format for FORAL Output

To obtain output using FORAL, we type in a name for the output followed by an arrow " <= " pointing to the output name followed by a specification of the output. That is:

output-name <= output-specification

Listing the Identifier Values for a Class of Entities

To begin an output specification, we enter the name of the Identifier for the desired class of entities. (We will call this the Main Identifier.) This sets the initial Context at the Main Identifier and calls for ordered output of the Identifier's Values. That is:

OUTPUT <= ADDRESS

results in the output of an ordered list of the ADDRESS's under the name OUTPUT. This would be equivalent to the English statement "List as OUTPUT, all the addresses in the system."

Listing the Direct Attributes of an Identifier Context

The English statement "List as OUTPUT, each address and the employees and the departments at each address." asks for a class of entities and two of their direct attributes. To obtain the values of any direct attributes, we simply enter their names within a pair of following parentheses. That is:

OUTPUT <= ADDRESS (EMP-NO, DEPT-NO).

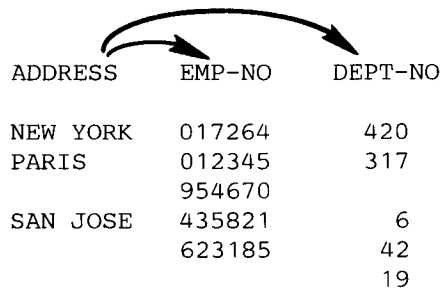
If the output is in the form of an internal file, then the file will be named OUTPUT, will have three fields, and will be ordered starting with ADDRESS as the major key. If the output is a printed report, then the first column will contain ADDRESS, the second, EMP-NO, and the third, DEPT-NO (Figure 7).

Note that the EMP-NOs and the DEPT-NOs are not directly related.

Functions of Direct Attributes

The user can also write functions of attributes in the output colums. For example, the English statement, "List as OUTPUT for employees, their salary plus commmission, their salary and their commission." can be rendered as:

OUTPUT <= EMP-NO (SALARY + COMM, SALARY, COMM).



ADDRESS	EMP-NO	DEPT-NO
NEW YORK	017264	420
PARIS	012345	317
	954670	
SAN JOSE	435821	6
	623185	42
		19

Figure 7

In this case, the first column will be EMP-NO, and the second will be the value of the function, SALARY + COMM. The third column will be SALARY, etc.

If we wish to give a column a different name then we precede the definition of the contents by the new name followed by an arrow pointing left. For example, "List as OUTPUT for employees, their total compensation which is given by salary plus commission.":

OUTPUT <= EMP-NO (TOTAL-COMP <= SALARY + COMM).

This will give two columns, the first will be headed EMP-NO, and the second, TOTAL-COMP. In the first column will be the employee's number and in the second will be his total compensation (which is also a direct attribute of employee, but it is one that can be calculated from the values of other attributes).

For attributes that may have multiple values, there are functions that will access these values and present a count of the number of values, or if the value is numeric will find the maximum value or the sum of the values, etc. For example, "List as OUTPUT for departments, the count of their employees." In FORAL, there will be a reserved word for each useful function. The user can output the value of the function in a column by placing the reserved word in the parentheses followed (in nested parentheses) by the name of the attribute that the function is to be applied to. For example:

OUTPUT <= DEPT-NO (*COUNT* (EMP-NO)).

This will give the output presented in Figure 8.

DEPT-NO	COUNT (EMP-NO)
123	24
420	43
561	134

Figure 8

Listing Indirect Attributes

In many cases, the user will want to print out "indirect" attributes; that is, attributes not directly connected to the initial context. For example, "List as OUTPUT for departments, the salaries of their employees." To accomplish this, he can move the context along a particular attribute arrow by naming it preceded by the reserved word *FOR*. For example, to get the salaries of employees of departments (which are indirect attributes of departments) we write:

OUTPUT <= DEPT-NO *FOR* EMP-NO (SALARY).

This will give the printout in Figure 9.

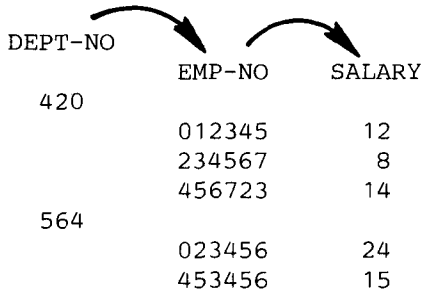


Figure 9

The stepping down of one line after a *FOR* gives us a better indication of the semantics of the printout. In this case, it indicates that SALARY is a direct attribute of EMP-NO and only an indirect attribute of DEPT-NO.

The *FOR* chain may be extended to any length to obtain indirect attributes. In addition, direct attributes of Identifiers in Context along the way may be printed out. For example:

OUTPUT <= ADDRESS (EMP-NO) *FOR* DEPT-NO (MGR-OF-DEPT)

FOR EMP-NO (SALARY) *FOR* PROJ-OF-MGR (NAME).

Statements are ended by a period, they may be continued on additional lines by indenting more than one space. This allows the user to organize long statements for greater readability.

It should be noted that each different path in the structure has a different meaning. For example, this path gives the names of projects of employees of departments located at each address. To obtain the names of projects managed by employees at each address, we would write:

OUTPUT <= ADDRESS *FOR* EMP-NO *FOR* PROJ-OF-MGR (NAME).

This second statement would give the same project names as the first only if all employees were located at the same addresses as their departments.

Multiple Branches to Indirect Attributes

In some cases, the user may wish to list the indirect attributes down two different *FOR* chains from a particular context. For example, "List as OUTPUT for addresses, each department at the address and for each department, the salary of its manager and the salaries of its employees. This requires two *FOR* chains originating at DEPT-NO. In FORAL, we can return a context to a previous location and go out a different branch by simply giving the attribute name that originally lead to the context preceded by a *FOR*. For our example:

OUTPUT <= ADDRESS *FOR* DEPT-NO *FOR* MGR-OF-DEPT (SALARY)

FOR DEPT-NO *FOR* EMP-NO (SALARY).

A possible printout appears in Figure 10.

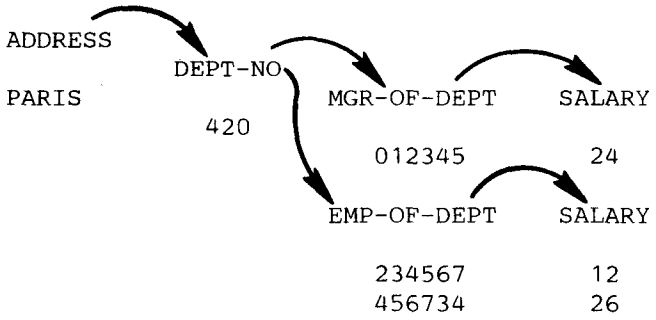


Figure 10

This same capability could have been achieved by utilizing additional sets of parentheses, but the present method seems more readable. However, it does bring up an interesting side issue. Is there an ambiguity if the *FOR* chain doubles back through a previous context? For example, "List as OUTPUT for employees, the departments that they manage and the salaries of the employees in the departments they manage.

We are, in this case, talking about two types of subgroups from the same Identifier. One is employees who are managers of departments and the second is employees who are employees in those departments. We should give each subgroup a different name. FORAL can resolve this issue in one of two functionally equivalent but syntactically different ways. In one method we can give the second group a different name by means of a *CALLED* phrase when we first encounter it in the statement. After that we can, of course, refer to it by its *CALLED* name.

OUTPUT <= EMP-NO *FOR* DEPT-OF-MGR

FOR EMP-NO *CALLED* EMP-IN-DEPT (SALARY).

We could also introduce a special form of qualification by constructing the second group name from the first by the use of a special character, say underline "_". For example: OUTPUT <= EMP-NO *FOR* DEPT-OF-MGR *FOR* EMP-NO_1 (SALARY).

This has the advantage of being more succinct and the qualification "_1" could be used on continuing attributes along this chain, but the first method is more mnemonic. In either case, this delayed naming technique seems preferable to the predicate calculus technique which requires the declaration of the variable names at the beginning of the statement. In the FORAL case, the user only needs to worry about the possibility of two or more subgroups when he encounters a return to an earlier context. Depending on whether he is returning to a old context or encountering a new subgroup, he can take care of the problem at that time. He does not need to go back to the beginning and reframe his query strategy. In fact, FORAL can analyze the statement and give him a warning that he might have a problem. This analysis might be quite difficult in other notations.

Relating Indirect Attributes to a Context

There are cases where the user may wish to obtain counts, sums, or perhaps, the maximum value of a set of indirect attribute values. For example, "List as OUTPUT for departments, the sum of the salaries of their employees." This requires us to relate the indirect attribute values to the context of department. It is possible to devise a syntax that uses *FOR* phrases to accomplish this task, but there is a more English-like syntax which utilizes *OF* phrases. In particular, the first element of the phrase will be the name of the attribute arrow leading to the values we wish to count, sum, etc. (in this case, SALARY). The last element of the phrase will be the name that originally lead to the desired context (DEPT-NO). between, we have the names of the attributes leading from the context to the desired values. For example:

OUTPUT <= DEPT-NO (*SUM* (SALARY *OF* EMP-NO *OF* DEPT-NO)).

Other Reserved Words

In the cases above, the user wished to have counts, sums, etc. over the attributes of values within an Identifier and the mechanism provided above is completely adequate. However in some cases, the user may wish to sum, count, etc. over all values in an Identifier. For example, "List as OUTPUT, the count of all employees in the enterprise." To specify such operations, we will use a special Identifier Name, *GROUP*. When it is used as the first element in an output specification, it acts exactly as a Identifier with one element whose attribute names are all the other Identifier Names in the system. Naming another Identifier in this context is therefore equivalent to providing access to each of the Identifier's values. Our example is then:

OUTPUT <= *GROUP* (*COUNT* (EMP-NO)).

To obtain the sum of all the salaries of employees in the enterprise, we write:

OUTPUT <= *GROUP* (*SUM* (SALARY *OF* EMP-NO *OF GROUP*)).

We can also have other reserved words to make output more convenient or flexible. For example, if we place *ALL* in the output parentheses following a particular context, the system will output the values of "ALL" the direct attributes of that context. If we add "*EXCEPT* attribute-name" the system will output all direct attributes "EXCEPT" those named.

Finally, we can place the reserved word, *NO* in the output parentheses to suppress the printing of the values in context. For example:

OUTPUT <= ADDRESS *FOR* DEPT-NO (*NO*, EMP-NO).

This will suppress the printing of DEPT-NO. It should be noticed, however, that this is a semantically dangerous tactic because the printout without the DEPT-NO's will appear to contain *employees of addresses* rather than *employees of departments of addresses.* as in this case, the user of FORAL must do additional work.

This concludes the elements of the output specification, we now go on the qualifications on the output elements.

QUALIFICATION IN FORAL

In many cases, the user will want the system to limit the set of Identifier or Attribute Values that are to be included in its processing. For example:

"List as OUTPUT only those department numbers over 130."

He can state his limitation in a parenthesized *WHERE* clause immediately following the Identifier or Attribute Name that he wants to limit. For example:

OUTPUT <= DEPT-NO (*WHERE* DEPT-NO *GT* 130).

We will call the Identifer or Attribute whose values are to be limited the *Target* of the WHERE clause, and the individual values the Target Values. In our example, the Target is DEPT-NO and when the system is cycling through the target values, it will test each to see if it is greater than 130. If the value meets the condition, then it will be included in the processing. If it does not, then the value will be discarded and no processing will be done on any attribute branches that extend from it in the FORAL statement. For example, if we have:

OUTPUT <= DEPT-NO (*WHERE* DEPT-NO *GT* 130) (ADDRESS)

FOR EMP-NO (COMM).

For those department numbers greater than 130, the system will print their values, it will then go on to print the value of their attribute ADDRESS and finally, it will print the employees in the department and their commissions. For those departments that do not qualify, the system will immediately discard their department number value ignore their attribute branches and go onto the next department value.

As a second example:

OUTPUT <= ADDRESS *FOR* DEPT-NO (*WHERE* DEPT-NO *GT* 130)(EMP-NO)

FOR ADDRESS *FOR* EMP-NO (COMM).

In this case, DEPT-NO and its associated EMP-NO's will only be printed for departments whose number is greater than 130. However, although *FOR* ADDRESS *FOR* EMP-NO (COMM) appears in the FORAL statement after the department number, it extends from the context of address rather than from the later context of department. Therefore, the values of this branch will be processed independent of the outcome of the limitation on departments.

The WHERE Clause and Its Subcontext

We have seen above how to limit a target value by placing a condition directly on it. We may also require that its direct or indirect attributes meet conditions for it to qualify.

Conditions on Direct Attributes

For example, "List as OUTPUT, employees whose department number is 420."

OUTPUT <= EMP-NO (*WHERE* DEPT-NO *EQ* 420).

Employee number is being limited on the basis of a condition on a direct attribute (DEPT-NO *EQ* 420).

Conditions on Indirect Attributes

For example, "List as OUTPUT, employee numbers whose departments are located in NEW YORK."

OUTPUT <= EMP-NO (*WHERE FOR* DEPT-NO ADDRESS *EQ* "NEW YORK").

Employee number is being limited on the basis of an indirect attribute, address of department of employee.

To provide the capability for stating these conditions and more complex ones each WHERE clause has its own subcontext that behaves exactly like the main context in the FORAL statement. That is:

(1) all attribute names used while the context is in force must be related to the context.

(2) the context may be moved by a *FOR* phrase

(3) indirect attributes may be related to the context by *OF* phrases

In the following sections, we will start with the simplest forms of conditions and build our way, by example, up to the more general forms.

Conditional Expressions in a WHERE Clause

The general form of the parenthesized WHERE clause is:

(*WHERE* conditional-expression)

The most basic form of conditional expression in FORAL is a *simple condition*. It takes the general form:

```
{ identifier-name }
{   or            }    relation    value
{ attribute-name  }
```

where the relations include "*EQ*" (EQual to), "*NE*" (Not Equal to), "*GT*" (Greater Than), "*GE*" (Greater than or Equal to), "*LT*" (Less Than), and "*LE*" (Less than or Equal to).

For example:

ADDRESS-OF-EMP *EQ* "PARIS"

EMP-NAME *EQ* "JOHN"

SALARY-OF-EMP *LT* 30

SALARY-OF-EMP *GT* 10

When a literal (that is, an identifier or attribute value) containing alphabetic characters appears on the right hand side of a relation it should be enclosed in parentheses.

Conditions on Attributes with Many Values

Frequently, the user will want to limit a target by a condition on one of its attributes that has many values. (For example, limiting department numbers by a condition on employee number, DEPT-NO (*WHERE* EMP-NO *EQ* 012345)). In this case, the department number being tested qualifies for further processing if *at least one* of the EMP-NO attribute values makes the condition true.

If the user wishes to specify that some number of employees greater than one is required to qualify the department, then he can use the following form to count the employees in the department and then test the count:

OUTPUT <= DEPT-NO (*WHERE COUNT* (EMP-NO) *GT* 5).

This particular example specifies that the department must have more than five employees to qualify. We could, of course, place conditions on the EMP-NO's also. For example, if we wished to print out department names and numbers for those departments with more than 5 employees earning over 12, we would write:

OUTPUT <= DEPT-NO (*WHERE COUNT* (EMP-NO (*WHERE* SALARY *GT* 12)) *GT* 5).

Combination of Conditions by the Use of *AND*'s and *OR*'s

As a first step toward more complex expressions, we can combine any of the conditions on the values in context or on their direct attribute values into a *conditional expression* by means of *AND*'s, *OR*'s, and parentheses.

In FORAL to avoid confusion on precedence, we will require the user to place parentheses around his expressions in such a way that "*AND*'s" and "*OR*'s" are not mixed. The conditions inside the deepest set of parentheses are evaluated first and the evaluation moves outward step by step. With regard to form, we will use the following example,

```
OUTPUT <= EMP-NO (WHERE EMP-NO GT 200000
                  AND ADDRESS EQ  "PARIS"
                  AND NAME EQ  "JANE").
```

In analogy to natural language punctuation, commas may be used to replace "*AND*'s" or "*OR*'s". They will be interpreted as follows:

(1) If an "*OR*" appears within the expression at a particular level, all commas will be interpreted as "*OR*'s" ("A, B, C, *OR* D" gives "A *OR* B *OR* C *OR* D").

(2) Otherwise, the commas are interpreted as "*AND*'s".

For example, we could have written the above FORAL statement as:

```
OUTPUT <= EMP-NO (WHERE  EMP-NO GT  200000,
                  ADDRESS EQ  "PARIS"
                  AND NAME EQ  "JANE").
```

Combinations on the Right-hand Side of a Simple Condition

We may combine right-hand-sides of simple conditions by means of *AND*'s and *OR*'s *with the restriction* that the resulting combination will be placed in *parentheses*. For example:

Relations and Member Names:

```
EMP-NO (GT 500 AND  LT 100000)
```

Member Names:

```
EMP-NO EQ (012345, 017264 OR 954760)
```

Relations Between Two Attributes of the Same Context

We can also place an attribute name *on the right-hand-side* of a relation. This allows us to qualify the Identifier in context on the basis of a comparison between two of its attributes. For example, if we wished to print out the employees whose commission is greater than their salary, we would write:

```
OUTPUT <= EMP-NO (WHERE  COMM GT  SALARY).
```

Movement of Context - The FOR Phrase

The Context is moved by exactly the same procedure we used in the output specification. That is, preceding the attribute or identifier name by a *FOR*. Generally it makes good sense to state the direct conditions on the current Context before moving the Context to a new attribute. That is,

OUTPUT <= EMP-NO (*WHERE* ADDRESS *EQ* "PARIS",

NAME *EQ* "JANE"

AND FOR MGR SALARY *GT* 20).

The *OF* Phrase

In analogy to the "*OF* phrase" in the output specification, we can also use "*OF* phrases" on the right-hand-side of a relational condition to bring indirect attributes into context. This allows us to test on the on the basis of a relation between two indirect attributes. That is:

Natural Language

"List the employees whose salary is greater than the salary of their manager"

FORAL

OUTPUT <= EMP-NO (*WHERE* SALARY *GT* (SALARY *OF* MGR *OF* EMP-NO)).

GROUP's on the Right Hand Side

On the *left-hand-side of a relational condition*, the attribute names are always directly or indirectly connected by *FOR* phrases to the Target. However, on the right-hand-side of a relational condition are the quantities which may be derived from any valid information in the system. We, therefore, have additional possibilities to consider.

Our main problem will occur with regard to the two possible interpretations of the names that appear. If we are at the Context, DEPT-NO, we may wish to have "COUNT (EMP-NO)" on the right-hand-side. An example would be the statement:

OUTPUT <= DEPT-NO (*WHERE* EMP-QUOTA *LT* *COUNT* (EMP-NO)).

(a) One interpretation is that EMP-NO is a name for a direct attribute. That is:

"List as OUTPUT, all those departments whose employee quota is less than the count of their employees."

In this case, only the attribute values of the identifier value in context should be considered in the test.

(b) A second interpretation is that EMP-NO is the name of an identifier and that the expression should be evaluated on the basis of *all the values of the identifier*. For example:

"List as OUTPUT, the departments whose employee quota is less than the total number of employees in the enterprise."

In this case, we will follow the name by the reserved word, *GROUP*, to distinguish the two interpretations. For example:

COUNT (EMP-NO *GROUP*)

Overlaying the I/O Specification with a WHERE Qualification

In the above discussion, each WHERE clause and its contents applied only to its target. For example, given the statement:

OUTPUT <= DEPT-NO (*WHERE FOR* EMP-NO SALARY *GT* 30)

FOR EMP-NO (NAME, SALARY).

FORAL would test each department to see if it had an employee whose SALARY was greater than 30. If the department had even one employee with a salary over 30, then FORAL would print out the department number and and the name and salary for *every* employee in the department.

However, the user will frequently want to print out the department number and the name and salary for *only those employees* whose salary was greater than 30. He can accomplish this using parenthesized *WHERE* clauses by:

OUTPUT <= DEPT-NO (*WHERE FOR* EMP-NO SALARY *GT* 30)

FOR EMP-NO (*WHERE* SALARY *GT* 30) (NAME, SALARY).

A more convenient method is, however, to use an unparenthesized FORAL *WHERE* clause that will overlay the i/o specification with the qualification. In this form the *WHERE* clause is placed *after* the complete i/o specification and has no parentheses around it.

In this case, the complete *WHERE* clause applies to the Main Identifier, and those *portions* of the "*WHERE* clause *FOR* chains" that match the "i/o specification *FOR* chains" will also apply as *WHERE* clauses to the corresponding levels in the i/o specification. The user can, therefore, obtain the result he desires by:

OUTPUT <= DEPT-NO *FOR* EMP-NO (NAME, SALARY)

WHERE FOR EMP-NO SALARY *GT* 30.

In fact the overlay form is the one that should be used in most transactions. It prints out information only on those members that meet the conditions in the trailing condtional expression.

We will use the the overlay form for even the simplest transactions.

OUTPUT <= EMP-NO *WHERE* SALARY *GT* 30.

The overlay form has some parallel to the query statement format used in the TDMS for treating hierarchic records. (see Blier (13)). In fact, FORAL may be used to access hierarchic record structures. This can be seen if one recognizes that segment types at various levels can be correlated with identifiers and that the attributes of the identifiers are called fields in traditional terminology. However, the fact that the FORAL user can attach parenthesized *WHERE* clauses anywhere in the FORAL statement makes the language more powerful. It does not require the additional TDMS scope modifiers HAS and HAS EVERY.

Hardgrave (14) has pointed out some other problematic, but unusual, queries in his paper on Boolean operations on tree structures. It appears that certain of these unusual queries will require an operator in FORAL that can widen the number of branches that will be considered before a context is qualified or disqualified. We will not go into these operations in the present paper.

FORAL and ARRAYS

Existing procedural languages deal with arrays primarily by storing them in some regular fashion and calculating the addresses associated with particular subscripts. In this case, the subscripts are a very special kind of property of the elements of the array. In FORAL, we can treat the subscripts as just additional attributes of the elements.

Although present conceptions of FORAL would probably not be competitors in a performance sense with the existing procedural languages, it is intriguing to see how one might write programs that would run on both the usual dense and the less usual sparse representations of arrays.

In Figure 11, we present a binary version of a candidate for representation as an array. The set of points (entities) in the main array, A, have the coordinates, X, Y, and Z. These are the attributes whose values serve to identify the entities of type A. These entities have two other attributes, one, T, stands for Temperature, and the second, B, is really another array that is nested inside A. It is a two dimensional array with coordinate attributes, U and V, and other attributes, Q and P.

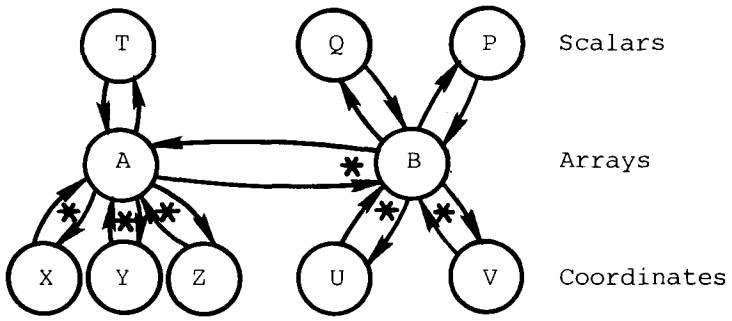


Figure 11

We can tell that B is embedded in A because A is one of the components of its identifier (as indicated by asterisks in the figure). In the FORAL language, it is possible to obtain a new array with Q and P added together by the statement:

ARRAY <= A (X, Y, Z, T) *FOR* B (U, V, Q+P).

This statement is valid regardless of whether the array is represented in storage by the usual FORTRAN arrangement, or is so sparse that only the valid elements are stored with their coordinates (sparse representation). Not all array operations are this simple in FORAL, in fact most would probably be easier in an arithmetic language, but it is interesting to see such operations specified in terms of attributes rather than coordinate nesting (content addressing rather than location addressing). For example, to deal with a particular column V = 7, we simply write a qualification limiting V to 7. That is:

ARRAY <= A (X,Y,Z,T) *FOR* B (*WHERE* V = 7)(U, V, Q+P).

SUMMARY ON FORAL QUERY

In the above, we have presented the major FORAL query functions, generally in a form designed for a non-programmer user. In a later paper, we will separately publish the maintenance language concepts for Basic FORAL.

There seem to be several new aspects to FORAL:

(1) *A user oriented language for binary associations*

There is a long history of the use of binary associations dating from the work of Levien and Maron (6) through the works of Langefors (7), Ash and Sibley (8), Sundgren (9), up to Abrial (10). Generally, the languages used by these authors are closely related to existing mathematical notations or to existing procedural languages. FORAL seems to have more of the characteristics of a natural language and a less procedural form. This reduction of notational clutter may make it an easier vehicle for uninitiated users to formulate difficult queries. In addition, as noticed by Levien and Maron, the binary notation allows us to avoid the complications of multifile (multirelation or multitable) query. In FORAL, there is just one form for getting to a neighboring attribute instead of two (a) a within record (table, relation) form and (b) a more complex "inter-record" form. In this way, the binary form saves many instances of extra work. For example, it is frequently not necessary to distinguish between many-to-one and one-to-many associations since these do not give rise to "within" or "inter-record" forms.

(2) *The notion of a Context in the interpretation of a statement*

Most languages use contexts defined in terms of blocks or subroutines. For the query language user, these contexts may present too wide an array of "seemingly" valid statements (which may not make sense with regard to the real world he is discussing). From this relatively infinite array, he must select one or more to accomplish his transaction. By narrowing the focus of attention to the immediate context and providing a simple set of primitives for relating it to other contexts (WHERE, OF, FOR), FORAL seems to channel the user into making semantic sense and thereby assisting him in formulating his query. It also allows him to use names in a more concise natural language manner. From the standpoint of FORAL statement validation, the use of a Context will allow the FORAL interpreter to, only consider names related to the current Context to be valid.

(3) *the* CALLED *phrase*

This phrase may have appeared elsewhere in the literature. In any case, it seems to be a useful way of giving names to groups or subgroups because it allows the name to be assigned when the definition of the group occurs during the user's transaction rather that at some remote location.

(4) *selection of optimal search strategy*

In compilation of FORAL for the DIAM II system, the compiler is faced with the selection and creation of an optimal search strategy given an arbitrary stored data structure at the time of compilation. This is a new kind and more difficult level of optimization than that faced in existing compilers where the search stategy is given to the compiler the user when he writes his procedural program. It is, however, a less difficult level than that faced by the work of Low at Stanford and Rochester and Schwartz on SETL at NYU. In this latter case, the compiler must also select an optimal data structure for an environment where data structures may dynamically change in size and form. In data base systems, the form of the stored data structure is selected independent of any single transaction and remains stable over a relatively long period of time. The selection of an optimal search strategy is the key to data independence with efficiency in data base systems and, therefore, the solution of this subproblem has major economic importance. Some preliminary work on this problem has been done in the DIAM I environment by Ghosh, Astrahan, and Senko (16).

ACKNOWLEDGEMENTS

Several acknowledgements are given in the formal references. In addition the works of Fehder (15), Hardgrave (14), and the GIS and TDMS projects have been very helpful. Fehder's work on RIL provides an excellent framework to investigating the construction of an information system language. Hardgrave's work also presents a wide-ranging investigation of the requirements of a query language. Conversations with G. M. Nijssen, H-J. Schneider, and H. Schmutz have also been very helpful. Nijssen and Schneider have indicated many important requirements for a user oriented, data independent language and Schmutz indicated the usefulness of a concept like the GROUP function.

REFERENCES

(1) S. E. Madnick, J. W. Alsop: A modular approach to file system design. SJCC, 1969, 1-13.

(2) H. S. Meltzer: Data base concepts and architecture for data base systems, IBM Report to SHARE Information Systems Research Project (August 20, 1969).

(3) M. E. Senko, E. B. Altman, M. M. Astrahan, and P. L. Fehder: Data structures and accessing in data base systems. IBM Systems Journal 1973, 12, 30-93.

(4) G. H. Mealy: Another look at data, FJCC 1967, pp 525-534.

(5) C. W. Bachman: Trends in data base management-1975. Proceedings of the 1975 National Computer Conference, pp 569-576 (AFIPS Press, Montvale, New Jersey, 1975).

SPARC Interim Report, American National Standards Institute Document No 7514ts01 (ANSI, Washington, D. C., February 8, 1975).

(6) R. E. Levien and M. E. Maron: A computer system for inference execution and data retrieval. Commun. ACM 1967, 10, 715-721.

(7) B. Langefors: Information systems. Information Processing 74, 937-945 (North Holland, Amsterdam, 1974).

(8) W. L. Ash and E. H. Sibley: TRAMP: an interpretive associative processor with deductive capabilities. Proc. 1968 ACM National Conference 1968, 144-156.

(9) B. Sundgren: An Infological Approach to Data Bases. (Urval nr 7) (National Central Bureau of Statistics, Stockholm, Sweden, 1973).

(10) J. R. Abrial: Data semantics, To appear in IFIP-TC-2 Working Conference on "Data Base Management Systems", Cargese, Corsica, North Holland (1974).

(11) M. E. Senko: The DDL in the context of a multilevel structured description: DIAM II with FORAL. Data Base Description, B.C.M. Douque and G.M.Nijssen (eds.) pp 239-248 (North Holland, Amsterdam, 1975).

(12) M. E. Senko: Specification of stored data structures and desired output results in DIAM II with FORAL, Proceedings of the Conference on Very Large Data Bases (ACM, New York, 1975).

(13) R. E. Blier: Treating hierarchical data structures in the SDC time-shared data management system (TDMS), Proceedings of the ACM National Meeting 1967 pp 41-49.

(14) W. T. Hardgrave: Theoretical aspects of Boolean operations on tree structures and implications for generalized data management, Computation Center Report TSN-26, University of Texas at Austin (1972).

(15) P. L. Fehder: The representation-independent language part 1: introduction and the subsetting operation. RJ 1121, IBM Research San Jose, California (1972).

P. L. Fehder: The representation-independent language part 2: derivation and insert, update, and delete operations. RJ 1251 IBM Research San Jose, California (1973).

(16) S. P. Ghosh and M. M. Astrahan: "A translator optimizer for obtaining answers to entity set queries from an arbitrary access path network," Proceedings of IFIP Congress, Stockholm, Sweden, August 1974 (North Holland, Amsterdam).

M. M. Astrahan and S. P. Ghosh: "A search path selection algorithm for the Data Independent Access Model (DIAM)," Proceedings of the ACM SIGFIDET Workshop, Ann Arbor, Michigan, 1974 (ACM, New York).

S. P. Ghosh and M. E. Senko: "String path search procedures for data base systems," IBM J. of Research and Development 18, 408-422 (1974).