AN IMPLEMENTATION OF FORAL USING IDMS

by

Daniel Goldschmidt and Allen Reiter

Technical Report #115

December 1977
ABSTRACT

FORAL is a non-procedural data specification language based on binary associations. IDMS is a commercial database management system based on a "network" model and belonging to the CODASYL DBTG group of languages. We describe an implementation of a subset of the FORAL query language over existing IDMS data bases. The mappings from FORAL to IDMS data types cause some problems and require trading off language power and user convenience against efficiency of execution; this tradeoff is left to the database administrator. The data structures and algorithms used in query translation and execution are also described.

Key words: Binary associations, database, data management, network model, query language, searching

Computing Reviews categories: 4.33; 3.74.
1. INTRODUCTION

We describe an implementation of a "binary associations" query language (FORAL) over a database management system which belongs to the "network" family (IDMS).

One of the strengths of FORAL is its use of context for concreteness and for semantic integrity checking. Its world is composed of entities and facts. FORAL does not distinguish between 1-n, n-1 and m-n relations, nor do reflexive relations merit any special treatment.

IDMS on the other hand has a record/field orientation in its data representations, and requires special treatment of n-m and reflexive relations. This necessitates special treatment in setting up file schema and also results in some restrictions on the end-user of FORAL.

The reason why one would choose to implement FORAL using such a seemingly unsuitable vehicle as IDMS is simple: expediency and availability. In his original papers describing FORAL (references [1],[2], and [3]), Senko describes a data management system DIAM suitable for implementation. However, there are no known implementations of DIAM, and to do such an implementation was outside the scope of the project. At the same time, it seemed to us that the many interesting features of FORAL merited attempting a direct implementation over a commonly used DBMS.

Another reason for using IDMS for the implementation was the existence of several data bases which would be candidates for using the language for on-line queries. These data bases are being maintained via conventional COBOL-plus-IDMS programs; had we chosen a different
vehicle for FORAL and still wanted to use it to search these data bases, two different copies of the same data base (with all of the ensuing problems) would have to be maintained. (The alternative - conversion of the entire system - was not even contemplated.)

Sections 2 and 3 of the paper summarize the salient characteristics of FORAL and IDMS. Section 4 deals with the problem of mapping FORAL structures onto IDMS, and the resulting language restrictions. Some technical details pertinent to our construction of the FORAL query executor are described in Section 5, and illustrated by example in Section 6.

2. AN OVERVIEW OF THE FORAL LANGUAGE

FORAL is a language for user interaction with computer stored data bases. As that, it has been designed to be easily understood by non-programmer users. User transactions are written in terms of real world things and their attributes rather than in terms of fields, records, files, etc.

FORAL is less procedural than most computer languages, the user specifying the information he needs, and not how to get it. Special emphasis has been placed in the construction of a set of English-like rules for ease of use and understanding.

To state FORAL queries, a user refers to a base structure similar to the DIAM ([2]) type diagram, although he may be working over a different kind of data base system. This diagram is a graphical representation of the information (corresponding to the DIAM II Information level) present in the data base.
This diagram is constructed using two types of building blocks:

1) Entity sets, which are sets of entities from the real world.
2) Fact sets, which provide a way for describing entities.

Since most entities cannot be stored in a computing system, "storable entities" are used to stand for them. For example, we cannot store an "employee entity" in a computing system, but we can use a storable entity such as "234" to stand for him. In this case, "234" is an Identifier Value representing one specific employee. We could also have a department represented by "234", so that the value "234" would be a member of two entity sets, say EMP-NO for employees and DEPT-NO for departments.

Figure 1
In Figure 1 we have four entity sets. The entities of two of the sets, EMP-NO and DEPT-NO can be stored in a computer and used to identify themselves. They can also be used to identify the entities of other sets, in this case employees and departments.

A Fact Set connects two entity sets, and indicates that the members of one of the entity sets describe in some way members of the other entity set. Fact Sets go in both directions.

For example, the Fact Set

```
EMP-NO -> DEPT-of-EMP -> DEPT-NO
EMP-of-DEPT
```

relates the Entity Sets EMP-NO and DEPT-NO. The names above and below the line are called attribute names.

A basic fact is represented by the association of identifier values for two entities.

For example,

```
EMP-NO
| 350 |
```

```
DEPT-of-EMP
| EMP-of-DEPT |
```

If our context is centered on "EMP-NO 350", then we can get a particular attribute value of this employee by following the association line named "DEPT-of-EMP" to the value "234" to the other end. This tells us that the entity EMP-NO "350" has an attribute named "DEPT-of-EMP" with the attribute value "234".

The converse is also true. If our context is at "DEPT-NO 234", we can get a value of the departments attribute called EMP-of-DEPT.
by following the association line. This tells us the employee number of one of the employees of this department. Note that in this representation there is no difference between one to many, many to many, or many to one relationships.

Context was mentioned earlier. It plays a very important role in FORAL, making the statement of queries simpler and more English-like, and assuring a greater semantic integrity. For example, you can ask for the attribute \texttt{DEPT-of-EMP} only if the context is at \texttt{EMP-NO}.

If only one Fact Set connects two Entity Sets, context enables us to use the target entity set name instead of the attribute name, thus making the sentence more readable in many cases.

A FORAL query specifies a tree over the network defined by the \texttt{DIAM} diagram (the "World"). The tree hierarchy is similar to the hierarchy imposed by the context, only connected nodes can be present. The FORAL query language is very concise, permitting to state in one sentence the equivalent of a complete computer program.

The query tree is specified using a very small set of operators. We will explain their use by stating queries over the database shown by the \texttt{DIAM} diagram of Figure 2.

Some of the arrows emanating from the oval have "\$" on them. These indicate the preferred identifier for the Entity Set in the oval. For example, the preferred identifier for \texttt{EMPLOYEES} is \texttt{EMP-NO}. When \texttt{EMPLOYEES} is addressed, the corresponding value of \texttt{EMP-NO} will be printed out.
2.1 Initial Context

At first, we state the "initial entity set in context":

```
EMPLOYEES
```

...is this the whole query, a list of the EMP-NO for all the EMPLOYEES in the data base will be printed.

Printing Direct Attributes of the Context

a) EMPLOYEES print EMP-NAME, SALARY.

The output will be a three column list, with the EMP-NO, EMP-NAME and SALARY of each EMPLOYEE.

b) DEPARTMENT print EMPLOYEES.
This will produce a two column output, giving for each DEPARTMENT its DEPT-NO and the EMP-NO of each one of its EMPLOYEES

<table>
<thead>
<tr>
<th>DEPT-NO</th>
<th>EMP-NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>341</td>
</tr>
<tr>
<td></td>
<td>820</td>
</tr>
<tr>
<td>250</td>
<td>410</td>
</tr>
<tr>
<td></td>
<td>509</td>
</tr>
</tbody>
</table>

2.2 **Printing Indirect Attributes of the Context**

Indirect attributes of an Entity Set are given by entity sets not immediately adjacent to the oval representing it. To specify them, the "of" phrase is used. The whole path is defined, by stating the names of the attributes from the desired attribute to the context, linked by the work "of".

For example, the names of the employees of a department are an indirect attribute of the department. In FORAL we write:

a) `DEPARTMENT print EMP-NAME of EMPLOYEE of DEPARTMENT`

We will get two columns as in the previous query, but instead of the employee numbers will appear now the EMP-NAME's.

b) `EMPLOYEES print`

`SALARY, CITY of ADDRESS of DEP-MGR of EMPLOYEES, EMP-NAME`

We will get four column output, containing the employee id., salary, city of the department they manage, and their name.
2.3 Moving the Context

The context is moved from one entity set to another (which is connected to it by a fact-set) by means of the "for" phrase. The new entity set is then the entity set in context.

a) DEPARTMENT for EMPLOYEE
   print EMP-NAME,SALARY.

This will produce a four column output, listing all the DEPARTMENTS (actually DEPT-NO) and for each one, all its EMP-NO, together with the EMP-NAME and SALARY each EMPLOYEE.

The context can be moved further by subsequent uses of the "for" phrase.

2.4 Qualifying Entity Set Members

The user can specify which values of an entity set or attribute are to be selected for processing by means of the "where qualifies".

a) EMPLOYEES (where SALARY gt 30) print EMP-NAME

The output will consist of two columns, listing the EMP-NO and EMP-NAME of those EMPLOYEES which salary is greater than 30.

The "where" clause has its own context, which can be moved in the usual way by means of the "for" phrase. It can contain any number of test relations, connected by "and", "or". These relations can involve constants, direct attributes, indirect attributes (by means of "of" phrases) and also functions on the attributes.

b) For each employee whose department is in Atlanta, print his name and the number and street of his department.
C) Print all the employees with department number greater than 200 and located in New York:

```
EMPLOYEES (where for DEPARTMENT, DEPT-NO gt 200 and
            CITY of ADDRESS of DEPARTMENT eq "New-York")
```

2.5 Multigraph queries

When an entity set name which was in context at a higher level appears again as the target of a "for" phrase, the context jumps back to the previous context (thus effectively returning to the previous point in the tree). If instead, we want to keep going "down" in the tree (the members of the new entity set are subordinate to our current context), we follow the entity set name with a "called" phrase.

a) EMPLOYEES
   for DEPARTMENT
      (where DEPT-NO gt 250)
      print ADDRESS
   for EMPLOYEES
   for PROJECT print TERM-DATE

b) EMPLOYEES
   for DEPARTMENT
      (where DEPT-NO gt 250)
      print ADDRESS
   for EMPLOYEES called EIMP_WITH_DEPT_NO_GT_250
   for PROJECT print TERM-DATE

```
Looking at the resulting trees it is easy to understand the difference between both queries.

FORAL has many other interesting features which were not mentioned here. These include all the FORAL Input Language, designed to maintain the data base; the file specification part of the output language, which specifies from which file the information is to be extracted (if it is not the central data base) and whether the results should be outputed or stored in a temporary file for further processing; the "footnote" option, which enables the user to state parts of his query at a later time; the "all" word, which specifies that all the direct attributes of the entity in context are to be printed (instead of listing each one of the attributes); and various functions such as arithmetic operators, count, etc. (Most of these features were omitted from our implementation.)

A very nice design using FORAL is FORAL LP (LP standing for light pen). It makes full use of the DIAM diagram and of the capabilities of interactive display terminals. The DIAM diagram or selected parts of it are displayed on the screen. The user states his query using a light pen, first selecting the first entity set, the WHERE clause, the attributes he wants to extract, and then he moves the context to the next entity set. All these operations are simply done by touching the nodes with the entity set names, the lines (associations joining them) and reserved words listed on the side of the screen.

A complete description of FORAL LP is given in reference [3].
3. AN OVERVIEW OF IDMS

A complete description of IDMS can be found in reference [4]; we give here a brief overview sufficient for our purposes.

**Schema** The IDMS "world" is described in a data base schema. The schema is built using two types of blocks: records and sets. There can be any number of record types, each one composed by a number of fields. Every occurrence of a record type contains information in its fields.

The structure of the data base is given by its sets. A set always has a record type as its owner, and one or more record types as its members. A record can be part of many sets, thus permitting the specification of a network.

If a record type is the owner of a set, every one of its occurrences is the start of a chain which links all the member records which are subordinated to this owner record occurrence.

```
DEPT
   employ-of-dept
      EMPLOYEE
```

The set employ-of-dept has as its owner the record DEPT, and as its member, the record EMPLOYEE. A particular instance of this set could be:

```
DEPT
  EMP 121
  EMP 48
  EMP 51
  EMP 125
  EMP 278
```
There are some restrictions on the schema structure. For example, a record type cannot be the owner and a member of the same set; a member record occurrence can appear at most once in one set (i.e. the employee-125 record cannot be part of the set employ-of-dept for department-121 and for another department).

If such constructs are required in a data base, a "dummy" record-type is used, and the set is split into two sets: The first from the owner to the dummy record, the second from the member record to the dummy record.

For example, if in a manufacturing data base we want to represent the relationship between assemblies and the parts of which they are composed, we would like to have

```
PART
\rightarrow composed-of
```

Instead we have to define

```
PART
\leftarrow composed-of
\rightarrow part-of
```

To find which parts compose a given part, we run through the composed-of set and for every DUMMY record we reach, we ask for its owner in the part-of set.
Retrieval Functions

IDMS supplies a number of data base functions which permit accessing records in the data base, either following sets of sequentially by appearance in storage. Retrieval can also be on the basis of a supplied key.

The functions relevant for the sequel are:

1. OBTAIN NEXT RECORD (recordname) SET (setname) to retrieve the member records of a given set occurrence.
2. OBTAIN OWNER SET (setname) to retrieve the owner record of the current occurrence of the named set.
3. OBTAIN CALC RECORD (recordname) to retrieve the record using its key.
4. OBTAIN NEXT RECORD (recordname) AREA (areaname) to retrieve the records of a given type in the sequence of appearance in the data base.

These functions are invoked from programs written in a host language. Our task is to write a general-purpose FORAL query processor which generates the requisite sequence of calls to IDMS functions.

4. A MAPPING OF FORAL STRUCTURES ONTO IDMS STRUCTURES

There are a number of substantial differences between the data base structures used by FORAL and those used by IDMS.

First, FORAL has only entities and relations; IDMS has records, fields and sets. While relations may roughly correspond to sets, we have to decide whether entities are to be mapped onto records or onto fields; or what; this decision has important implications on usage and performance.
Second, as mentioned earlier FORAL allows many-to-many and reflexive relations and their internal handling is of no interest to the user. In the diagram used by the end user, such relations appear identical to the one-to-many and many-to-one relations. In IDMS however the representation of such relations necessarily involves adding new record types and new sets; without some naming conventions or additional information in the file schema, the user would have to provide specifications not part of the FORAL language.

Since FORAL relations are represented by IDMS sets, and since sets link records, not fields, it is clear that if we map entities onto fields we will lose some of the power of FORAL, since we will not be able to represent relations between these entities and others. Actually, the situation is somewhat better than that: we can make use of the implicit connection between a record type and the fields which it contains to represent a FORAL relation between the entity corresponding to a field and the entity corresponding to a record. (Such relations are sometimes called properties in other systems.) We can similarly relate two fields in the same record type to one another. Nevertheless, a FORAL entity mapped onto an IDMS field in one record type cannot be explicitly related to an entity mapped onto a different record type or onto a field in a different record type.

On the other hand, we could choose to map FORAL entities only onto records; this would give us the full power of FORAL since every relation can not be represented by an IDMS set (or two sets, for many-to-many and reflexive relations). Each record would contain one field (to store the value), but the field name would have no semantic significance. This
solution, while ideal in theory, is not feasible for a number of reasons. First, the storage overhead (in the form of pointers for representing all of the necessary connections) would be formidable. Second, even the most trivial retrieval operation would take an inordinate amount of time, for to collect what we normally would think of as a record would now require collecting its fields from all over the storage medium. Finally, we set out to build a query language which would also work with existing file schemata for some real databases; not unnaturally, these contained a number of different fields per record type, and we wished to allow a user access to these; in fact, we did not want a user to be concerned with the specifics of our mapping.

We compromised by allowing either mapping: an entity can correspond either to a record type or to a field type. The database administrator (when setting up the file schema initially) can trade off retrieval power versus efficiency. Once a schema is set up, the user is not concerned with this mapping, except for one important restriction: we do not allow the FORAL context to be left at an entity corresponding to a field.

Assuming that we start with an existing IDMS data base, we have no difficulty executing a FORAL query over it (assuming that every record type indeed can be interpreted as an entity). If, however, we start with a FORAL world view and want to define an IDMS schema for it, we need to know how to map many-to-many and reflexive relations. Specifically, we need some naming convention or external definitions so that the "dummy" record types which are necessarily introduced in IDMS are transparent to the user.

We chose the latter possibility, i.e. in addition to the standard
IDMS schema we utilize a stored schema which translates a FORAL relation and its inverse into an IDMS set or an IDMS dummy record and two connecting sets. Thus, the FORAL diagram

![FORAL Diagram](image)

is represented in IDMS thus:

![IDMS Diagram](image)

and an additional file schema tells the system that the FORAL "All projects of an employee" in IDMS is executed as "Run through the PROJECT_OF_EMPLOYEE set and for each E_P record found its owner in the EMPLOYEE_OF_PROJECT set".

Similarly, the FORAL diagram

![FORAL Diagram](image)

has the IDMS equivalent

![IDMS Diagram](image)

Note that in this example, without naming conventions or some explicit externally-provided information, the system would have no way of knowing
which IDMS set corresponds to the FORAL "MANAGER_OF" relation and which to its inverse.

Not allowing the FORAL context to rest on a "field" entity does not seem to be much of a restriction in normal usage: the user would not want to remain on such a "dead-end" entity in any case. Entities represented by fields are related automatically to the entity represented by the record; although this relation does not have an explicit IDMS name, we use the field name to stand for the relation (in effect, we name the relation, not the entity). An important consequence of this is that we now allow what are in effect two different FORAL entities to have the same name as in the following FORAL diagram:

![FORAL Diagram 1](image1)

(The ADDRESSes are fields in the corresponding record types.) From the user's point of view, the appropriate FORAL diagram is actually

![FORAL Diagram 2](image2)

where the entity implied by the lower oval (represented by the value of ADDRESS) is unnamed and not explicitly accessible.

IDMS has one other "implicit" way of representing a logical relation between two record types: if record type A is stored using a key
("LOCATION MODE IS CALC"), and record type B has a field whose value
domain ranges over the keys of A, then an occurrence of B defines one or
more specific occurrences of A. Such n-m relations are part of the
normal semantics of an IDMS data base. Note, however, that this
relation is one-way: given an occurrence of A, there is no way to find
the corresponding B's, short of scanning all of them.

We decided to allow such relations to be explicit in FORAL schema.
Such relations can only be traversed in one direction, however, since we
did not want a user inadvertently to execute very time-consuming searches.

5. OUR IMPLEMENTATION

The general query processor schematic is shown in Figure 3. The
program can be divided into two well defined phases: Translation and
execution.

The translator nucleus is a standard top down parser, which calls
a scanner to obtain the next symbol and (when making a reduction) calls
semantic routines which build the execution tree. (A reader not familiar
with compiler-writing is referred e.g. to the excellent book by Gries [5].)
The FORAL schema is used in this phase in order to complete information
supplied in the query and to perform semantic correctness checking.

The execution phase consists of calls to IDMS to retrieve relevant
records: it is driven entirely by the "execution tree" produced by
Phase I. Once a record is brought into core, its effect on the query status
is evaluated and all of its relevant information extracted. When appro-
priate, results are displayed to the user by the PRINT routine.
Figure 3: General Program Layout.
We now describe the phases in more detail.

**Translation.** Our translator is driven by a reduced BNF description of FORAL (see Appendix A). A BNF description of the complete FORAL Output Language appears in reference [5]. Not all of the FORAL features are implemented, but enough to provide the nucleus of a working system and to test FORAL feasibility in a real environment.

The translator's input is the FORAL query, and its output is a tree representation of it, containing further information which is taken from the FORAL schema.

The **Execution Tree**: A glance at the trees in Section 2.6 will give the reader insight into the kind of data structures which are required for the internal representation of the query.

We build a tree in which each node (a "record descriptor" node) corresponds to a record type to be visited along a certain access path when the query is executing. Each time a "for" clause occurs, we have a branch emanating from the "context" node. The information present in each node is: the record name, the set name which leads to it, its role in the set (owner or member), a second name if the "called" option was used (see 2.6), and pointers to two auxiliary data structures: a "print chain", listing the fields in this record to be displayed, and (if a "where" clause was present) a "where tree" used to test whether the retrieved record occurrence meets the qualification criteria specified.

The FORAL schema is used to augment the information present in the input query when the execution tree is being built: when going from one node to another the query only lists the target entity name or relation name; the schema provides the missing record or set name, the preferred
identifier, and the set direction. The semantic correctness of the query in terms of consistency with the schema is also tested.

The print chain contains a link for every field listed in the print specification as well as for the preferred identifier. If the field is a direct attribute (i.e. an IDMS field in the current record) the link points to a field descriptor. If, however, the field is an indirect attribute (if "of" phrases are present, i.e. fields of related records are implied) the link starts a chain of record descriptor nodes (structured as above) which indicates the records and sets to be traversed in getting to the desired field value.

The where tree consists of a "decision table" which describes the logical structure of the query (AND - OR connectives and parentheses) and a chain of relations to be tested. In the simplest case, an entry in this chain consists of a direct attribute, relational operator, and constant value; the corresponding field in the record occurrence is tested against the constant. If an indirect attribute is specified, a chain of record descriptor nodes leads to the field occurrence to be tested. If however an embedded "for" phrase occurs in the "where" clause (leading to an iterated search), the entry contains a link to another "where" tree, a subtree of the present one.

The decision table is used for optimal query execution: while the predicate is being evaluated the facts known so far are recorded in a table, and parts of the query which no longer affect the value of the predicate are skipped. In this way unnecessary comparisons are avoided. (Note, that when an indirect attribute value is involved, this can save a great deal of time in accessing other records.) The data structure and associated algorithms for this are described in [7].
Query Execution: The second phase consists of "executing" the execution tree. At the heart of this is a depth-first recursive tree traversal routine is shown schematically in Figure 4. For each node reached in the execution tree, the corresponding IDMS record is retrieved. If the record qualifies for further processing, its "print chain" is displayed and its subtree in the execution tree is traversed. In both cases, the next record in the current set is retrieved, and the process repeated until the set is exhausted; the process then continues with its father, etc.

```plaintext
Print-Verify (node)
    do while (not-end-of-record)
        Validate (node)
        if OK then print (node)
            Print-Verify (son)
        end
        Print-Verify (brother)
    end Print-Verify
```

Figure 4: Outline of the recursive tree traversal routine.

The records retrieval is performed using standard IDMS routines, selected accordingly to the corresponding execution tree node information. For the root node, if the corresponding IDMS record has CALC location and keys are provided in the query (in the where clause), direct (CALC) retrieval is performed; otherwise, the file is searched sequentially, the first time using OBTAIN FIRST RECORD (recordname) in AREA, and afterwards OBTAIN NEXT RECORD (recorname) in AREA. For the other nodes, a loop of OBTAIN NEXT RECORD (recordname) SET (setname) is used, or OBTAIN OTHER SET (setname), depending on the sets direction, as stated in the execution-tree node.
The "Validate-node" routine in Figure 4 works similarly. It receives as its parameters a current context (in the form of a given record occurrence) and a "where tree" which refers to attributes of this record or to related record occurrences; it must evaluate the predicate expression represented by the "where tree" and return a value TRUE or FALSE. If an entry in this tree refers to a direct attribute, it is evaluated directly; if it involves an indirect attribute, all possible related records are tried until a TRUE value is received or until the possibilities are exhausted (in which case the value is FALSE); if the entry refers to an embedded "where tree", the routine calls itself recursively to evaluate the entry for all possibilities. As soon as one evaluation of TRUE is obtained, traversal is terminated, and evaluation continues with the next entry. This action is converse to that of the PRINT-VERIFY routine which visits every qualifying node. (See also reference [8] where a similar mechanism is described.)

6. A QUERY EXAMPLE

Consider the following query: "For each department located in New York, print the names and numbers of the employees with salary greater than 1000 who earned over 500 in commission." In FORAL, this is expressed as follows:

**DEPARTMENT (where CITY of ADDRESS of DEPARTMENT EQ 'NEW YORK')**

for **EMPLOYEE (where SALARY GT 1000 **

and COMM GT 500)

print NAME, COMM

The execution-tree for this query is shown in Figure 5.
Figure 5.
The first node contains only the record name, DEPARTMENT. All the DEPARTMENT records will be retrieved sequentially. For each one, the where clause has to be evaluated. It consists of only one comparison, but one of the arguments is an indirect attribute. The owner in the set ADDR-OF-DEPT is retrieved, and the CITY field is compared to 'NEW YORK'. If they don't match, this DEPARTMENT record doesn't qualify and the next one is retrieved.

If the DEPARTMENT qualifies for further processing, its DEPTNO field (preferred identifier) is printed and the traversal continues with the son node.

At this node, the record name is EMPLOYEE, the setname is EMP-OF-DEPT and the set direction is MEMBER, so we retrieve the employees of this department using OBTAIN NEXT EMPLOYEE RECORD WITHIN EMP-OF-DEPT SET. For each retrieved record, we test its where-clause. The SALARY field is compared with 1000 and if it is greater, then COMM is compared with 500. The Boolean expression is evaluated by means of the related decision table. If the record qualifies, its EMPNO, NAME and COMM fields are printed.

In any case, we then proceed with the next record (we again issue OBTAIN NEXT EMPLOYEE RECORD WITHIN EMP-OF-DEPT SET). When the end of the set is reached, we go back to the first node, and retrieve the next DEPARTMENT record.
7. SUMMARY

We have described one particular implementation of FORAL. The major interest in the implementation lies in the fact that the data structures and data manipulation language belonged to an entirely different class, with a different underlying data model. We briefly want to summarize our experience in this respect, for it is relevant to implementations of other high-level query languages as well.

1. The major problem is in mapping the basic data types from the one system onto the other. No totally satisfactory solution should in general be expected; the sacrifices are on the one hand language power reduction and restrictions on the user and on the other hand efficiency (or even feasibility) of execution. By requiring an extra level of data description (in the form of a FORAL schema) we have in effect passed the decision on the tradeoffs to the data base administrator when designing a new data base; this we believe is the proper approach. This also enabled us to use FORAL with existing data bases.

2. No particular difficulties with performance were encountered. Whatever inherent shortcomings there may be in IDMS, these were in no way exaggerated by adding a FORAL "front end" to the system. Conversely, there is no a priori reason to believe that any other general-purpose DBMS or even DIAM would in general perform better as a base for FORAL.

3. The data manipulation language of a network systems such as IDMS is well-suited to executing the tree-traversal operations inherent in hierarchical query structures.
APPENDIX

A BNF Specification of our FORAL Subset

Underlined tokens are terminals and are not further defined.

<Query> ::= <entity set print clause> [<for attrib. print clause>]
<entity set print clause> ::= <initial context indicator> [<print clause>]
<initial context indicator> ::= <initial entity set specif> [ (<where qualifier>) ]
<initial entity set specif> ::= <entity set name> | system
<print clause> ::= PRINT <field spec> {, <field spec>, }^n
<field spec> ::= <field name> | <renamer> | ALL
<renamer> ::= CALLED <name for output>
<for attrib. print clause> ::= <removed context indicator> [<print clause>]
<removed context indicator> ::= <context remover> [ (<where qualifier>) [ <removed context indicator> ] ]
<context remover> ::= FOR <context specif>
<context specif> ::= <attribute name> [<called phrase>]
<called phrase> ::= CALLED <new name>
<where qualifier> ::= WHERE <condition>
<condition> ::= [ <condition on the attrib> | <logical term> ]
<logical term> ::= AND | OR
<condition on the attrib> ::= [ <removed context indicator> , ] <relation>
<relation> ::= <field spec> <incomplete relation>
<incomplete relation>::=<relation symbol>{<field spec>|<element spec>}
<relation symbol>::=EQ|NE|LT|LE|GT|GE
<logical term>::=(<condition>)
<element spec>::=<constant>[,<element spec>]

Semantics of some of the tokens are as follows:

fieldname: a field of the record in context
inverse attribute name: relation leaving from the subcontext at
the right, can be recordname or a set name
name in context: the context of the print specification
constant: a number or a string enclosed in quotes
CALLED<new name>: indicates the context goes down the tree
and not back to the previous occurrence of this entity
as the entity in context,