PROGRAM ANALYSIS FOR CONVERSION FROM A NAVIGATION TO A SPECIFICATION DATABASE INTERFACE

Barbara Demo

Istituto di Scienze dell'Informazione, Università di Torino
C.so M. d'Azeglio, n. 42 - 10125 Torino (Italy)

ABSTRACT

The conversion of database application programs is investigated when migration is required from a system with navigation (CODASYL-like) db interface to a system with specification db interface but the database semantics is not changed. We propose an analysis technique of the source program which heavily relies on program control flow. When the program semantics in analyzed from the point of view of data usage, the db statements appearing in the program are associated with one or more semantic record access patterns. A technique is given for analyzing these multiple associations and combining the access patterns into db queries.
Decompilable programs are those which have reducible flow graphs.

1. INTRODUCTION

The conversion of database application programs is investigated when migration is required from a dbms with navigation (CODASYL-like) interface to a dbms with specification interface /Tsic 82/. We will assume here that the referred database semantics is unchanged. Our conversion problem is thus different from the one considered in /Su 81/ which assumes the data semantics is changed but the database interface is not changed.
The environment where our approach to db program conversion is relevant is the project SCOOP, (System for COOPeration) which is being carried out jointly by the University of Paris VI and the University of Turin. The project investigates how a cooperating database system can be built among already existing heterogeneous databases located at different sites while saving the investment in application programs. Figure 1 sketches the SCOOP approach to a heterogeneous distributed database system. Cooperation is achieved through the integration of local db schemata into one global schema which describes the resulting distributed database. First local schemata, expressed in possibly different data models (dm_i), are all converted into a single data model called conceptual dm and then integrated into the global conceptual schema.

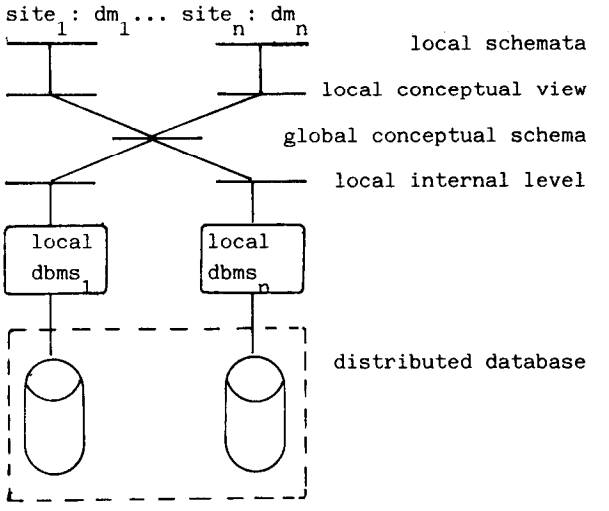


figure 1. Skeleton SCOOP architecture

The users at each site continue to use their previous database programs as well as their local database interface for new programs. The

distributed database management system provides conversion modules for translating both data structure and db programs from the local level to the global level and viceversa. The conversion are only once performed. The data structure translations are performed on the local schema descriptions. The programs are converted by precompile-time systems.
In SCOOP the Entity-Relationship (ER) model /Chen 76/ has been chosen as conceptual data model because of its reasonable power to express the semantics of data while being flexible enough to make the required translations easier. As conceptual data manipulation language (dml), the distributed environment requires a specification language which is able to express a complex set level operation in a single statement /Tsic 82/. This requirement has been argued in /Spac 80/as follows. A global dml statement can request data from different nodes in the network. The minimization of network data traffic requires that each data request results in a set of data which is to be transferred from node to node, rather than many transfers each consisting of a single datum. Should a record at a time navigational interface be used at global level, the dml statements in db programs must be grouped together to form set-at-a-time operators before the request in sent to the network. Control structures of the host language make such grouping a non-trivial task.
The fundamentals of the SCOOP ER DML are described in /Pare 83/ while an overall description of the SCOOP project may be found in /Spac 81, Spac 83/.

While the SCOOP project involves a whole set of conversion problems, one for each dm_i to the conceptual dm, we chose to concentrate first on the conversion of CODASYL COBOL programs into the COBOL embedded SCOOP-ER dml. Pratical motivation for this choice is because there are a large number of CODASYL-like DBMS installations. Theoretically is because the conversion of db programs is still an open area for research, we felt that it is best to focus on a specific host language which is in our case CODASYL COBOL language.
General introductions to the problem of db program conversion can be found in /Tayl 79, Su 81/. The first paper reviews existing works and proposes a framework for research on the subject and the latter gives several motivations for programs conversion in different environments and develops a systematic method for translating programs when the data semantics is changed without changing the db interface.

Current approaches to db program conversion distinguish two main activities (figure 2): the analysis of the source program and the synthesis of the target program.
The major point of the analysis activity is to find the program semantics for its interaction with the db. In other words, the analysis process determines the way in which db data are used by the source program. The program semantics is often described by means of a reference model /Su 81, Shne 82, Katz 82/.
As a concrete example, let us refer to the analysis phase described in /Nati 78, Su 81/. Here, some classes of semantic accesses, each corresponding to one typical use of data, are identified. Some code templates, in the different dmls, are then specified which perform the data access for each class.
Given a source program, in the COBOL CODASYL language for example, the way it uses data is recognized by matching CODASYL dml templates against the program statements. Program statements are considered in the sequence in which they appear in the source code except that, PERFORM statements are substituted by the statements of the corresponding procedure body /Moor 80/ (This approach is valid only for some situations and does not address the general case of data usage patterns). Sequences of semantic accesses identify the db query graphs which are the program semantics description in the Su's model.

In /Katz 82/ a similar approach is applied to the analysis of a db program for converting it from a navigation to a specification dml.

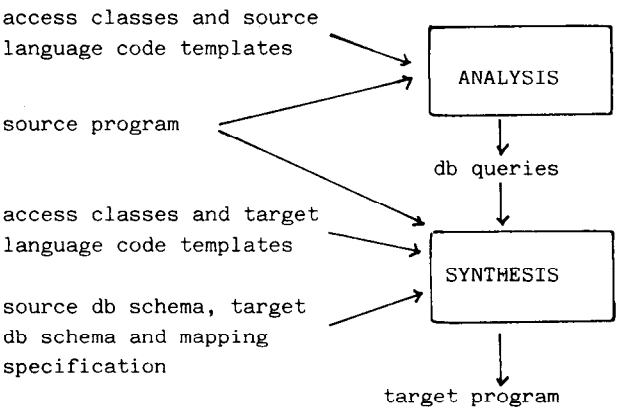


figure 2- Activities in db program conversion

In the synthesis activity (called embedding in /Katz 82/), the program semantics description resulting from the analysis phase is adapted to take into account either the data semantics changes causing the conversion /Lam 79/ or the db interface changes /Katz 82/. That is, in the synthesis phase, db data manipulations are first mapped into the target operators. Finally, the target operators are interfaced with the original program to produce the target program.

The present paper focuses on the analysis activity and particularly for converting a COBOL CODASYL db application program. Our primary contribution is to emphasize the role of the program flow structure in analysing the source program. The three major characteristics of our approach are as follows:
Semantic access to data are defined using a classification of CODASYL FIND statements and dependencies between FIND statements rather than over CODASYL FIND statements themselves. In /Nati 78, Katz 82/, the analysis is based on the assumption that different types of data accesses are expressed through some "standard" code sequences. Code sequences which do not match any of the standard ones, cannot be analyzed /Katz 82, Su 81/ in such a method. Our approach is more general because first we introduce a classification of both CODASYL FIND statements and dependencies between them caused by currency, db status indicators and db variables. The data access types, are then defined based on this classification. Our approach requires some extra processing but it does provide more generality and allows conversion of a larger class of programs.

Each db statement can participate in more than one semantic access. A program, describes all possible ways in which data might be used within a program run. Their actual use depends on the evaluation of control structure conditions at the execution time since the statements of a program can participate in different execution order. Given a FIND statement in the source program, our analysis process for finding out the semantic accesses takes into account each code sequence in which the FIND participates. A FIND statement may thus be associated to more than one semantic data access type depending on the use of data in which the FIND partecipates. Sequences of consecutive semantic accesses, called db queries, are defined by techniques which analyze these multiple associations in the program semantics characterization.

The third contribution of the paper is a formal characterization of the class of decompilable programs defining the set of programs which can be handled. Since this definition can be difficult when the program control structure is concerned, sometime authors describe it through examples /Moor 80/. In our case the characterization directly comes from the techniques used for analysing the program control flow. They are well known in compiler optimization area and apply to programs having reducible flow graphs /Aho 79/.

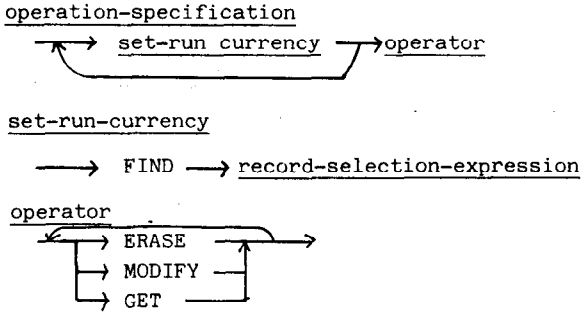
The outline of this paper is as follow:
In section 2 the CODASYL Data Manipulation Language is analyzed. Its FIND operators are distinguished as enumerative selection and single selection operators.
Section 3 and 4 concern the analysis process of the application programs. The former describes the dependencies between two statements created by the use of currencies and db status indicators. The semantic access types are defined as a qualification of statement dependencies. Db queries generation is shown in section 4.
Conclusive remarks and considerations on further research are given in section 5.

2. ANALYSIS OF CODASYL DML

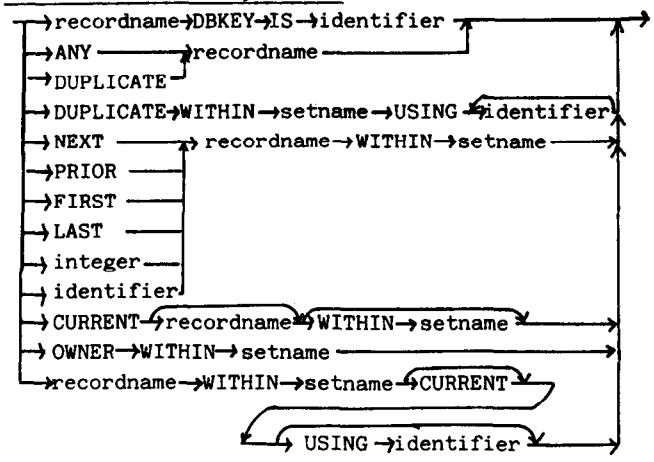
In this section, we analyse the CODASYL Data Manipulation Language for the purpose of classifying CODASYL FIND operators.

2.1 The CODASYL DML

The CODASYL DML is record at-a-time navigation language /TSIC 82/. Its basic operations are specified as follows /CCJD 76/ (an underlined word in small letters denotes a non terminal symbol whose syntax diagram is given, a non underlined word in small letters denotes a non terminal symbol whose syntax diagram is not given, a word in capitals denotes a terminal symbol):



record-selection-expression



A FIND operator in which any record selection expression is specified, transforms the selected record into the current record of its record type, of all sets in which it is participating and of the present run. The CODASYL operations of erasing, modifying, reading and even finding, except some FIND types, are based on these currency indicators. Should a record be deleted, modified or read, it must be first selected within the database through one or more FIND statements. After that, the operation is performed on it. Hereafter FIND statements are called record selection statements. From these remarks, it can be concluded that the way in which a program operates on or accesses a database is to be deduced from FIND operators.

Our analysis on CODASYL DML is mostly an analysis of FIND statements and an analysis of which record selection basic type can be distinguished. The analysis of COBOL CODASYL application programs in section 3 focuses on the types of FIND statement used, how FIND statements depend on one another because of currencies and db status indicators and which sequences of FIND statements are possible in any execution of the analyzed program.

2.2. Record selection

Our approach is based on the following observation. Within the CODASYL model a record selection can be either (1) a selection by enumeration or (2) a single selection.
Selection by enumeration and single selection are called basic record selection types.

Selection by enumeration

We call enumeration the selection of all records within a set, one by one. A data model provides enumeration capability if it allow (1) sets of records to be identified and (2) the selection of each single element of this set to be expressed.
In the CODASYL model, the enumeration capability is provided by FIND statements combined with one of the record selection expressions shown in figure 3. These statements are called selection by enumeration statements.

a. DUPLICATE record1
b. DUPLICATE WITHIN COD-set2 USING identifier2
c. NEXT / PRIOR record3 WITHIN COD-set3

figure 3. Selection by enumeration expressions.

We explain in the following how the expressions in figure 3 provide the CODASYL model with the enumeration capability.

First, the specification of any of the expressions in figure 3 identifies one set of records. The following informal descriptions of the identified record set types are derived from the General Rules of Record Selection Expressions in /CCJD 76/. We have called the record set type identified by the selection expression in figure 3.a (respectively 3.b or 3.c) Record-Set$_a$ (respectively Record-Set$_b$ or Record-Set$_c$).

a. Record-Set$_a$ = {occurrences of record1 having calculation key values equal to the calculation key in the current record of the run unit and greater database key}

b. Record-Set$_b$ = {occurrences of record type being the current of COD-set2 and member of it, having in field referenced by identifier2 contents equal to those in the current record of COD-set2 except the current record itself}

c. Record-Set$_c$ = if record3 is specified {occurrences of record3 being members of COD-set3 and following or preceding the current record of COD-set3 according to the set ordering criteria for that set type}

if record3 is not specified
{ occurrences of all record types within COD-set3 and following or preceding the current record of COD-set3 according to the set ordering criteria for that set type }

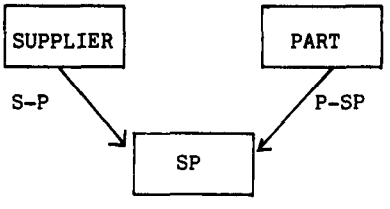
In our interpretation, the current record parametrizes a Record-Set but does non belong to the Record-Set itself.
In the following we will distinguish a set of records from the CODASYL concept of SET referring to them respectively as Record-Set and CODASYL-set.

At the same time as they identify sets, expressions in figure 3 perform the selection of records within the sets. Given the specification of an expression in figure 3, the selection of records within the corresponding set is done by iterating the execution of the same specification. The identified record sets are actually sequences of records that is ordered sets, whose ordering is either user (DBA) defined or system defined. At each iteration the next record in the set is picked up.
Within database application programs, the enumeration is generally expressed through a cycle on the DML enumeration statement. Very often the exit from such a cycle is subject to a test on the db status indicators of END OF SET or RECORD NOT FOUND, provided in CODASYL systems and meaning that all records within the Record-Set must be selected.
As an example, we refer to the sample case used throughout the paper. It is a skeleton code described in /Nati 78/ which produces a list of SNAME, SNO and STATUS of Suppliers and the PNO and PNAME of Parts they supply for all Suppliers based in Chicago. The database structure and sequence of code considered are shown in figure 4. The enumeration of all SUPPLIER records present in the db for printing out those having 'CHICAGO' value in the field CITY is done with a cycle on the statement 1. The exit from the cycle is controlled by statement 2 which is a test of RECORD NOT FOUND condition.

The code sequence which follows accesses only two occurrences of the set occurrences of record type EMPLOYEE associated by the CODASYL-set named WORK to record DEPT with DEPT-NAME='TOY' . The two records are in this case selected through the physical iteration of the two enumeration statements 3 and 5 within the program.

```
1.      MOVE 'TOY' TO DEPT-NAME
2.      FIND DEPT DBKEY IS DEPT-NAME
3.      FIND NEXT EMPLOYEE WITHIN WORK
4.           (print NAME of EMPLOYEE)
5.      FIND NEXT EMPLOYEE WITHIN WORK
6.           (print NAME of EMPLOYEE)
7.      EXIT.
```

Although such cases are possible, they are quite rare and will not be considered in this paper. In the following sections we concentrate on Record-Set enumerations through cycles on the enumeration statements.




```
1.      A.  FIND  NEXT  SUPPLIER  WITHIN  SYST-S
2,3         IF    RECORD NOT FOUND GO TO C
4.          GET   SUPPLIER
5,6         IF    CITY ≠'CHICAGO' GO TO  A
7.                (print  NAME,...IN SUPPLIER)
8.          FIND  FIRST  SP  WITHIN  S-P
9,10    B.  IF    END OF SET      GO TO   A
11.         FIND  OWNER      WITHIN P-SP
12.         GET   PART
13.               (print   PART-NAME,...)
14.         FIND  NEXT   SP  WITHIN  S-P
15.         GO TO B
16.     C.  EXIT
```

figure 4 - Sample Data Schema and Code sequence

Single Selection

Single selection is defined as an access to a record which always selects the same record regardless of the number of times it is iterated.
All CODASYL record selection expressions except those in figure 3 provide single selection. They are then called single selection expressions and FIND statements combined with them are called single selection statements.

A CODASYL COBOL application program expresses manipulations of records selected either through a single FIND statement or through combinations of FIND statements and of the basic record selection types they express.

3. ANALYSIS OF APPLICATION PROGRAM FLOW GRAPHS

A database application program specifies a computation where operations on a database are present. COBOL programs using a CODASYL DML interface to the db are assumed.

Within a db application program, CODASYL FIND statements (and the basic record selection types they express) are combined together to form several possible statement paths. Correctly identifying these statements paths requires that the program be analyzed in its proper order of execution. To make that easier we work on the flow graph transcription of the program.

A program flow graph is a directed graph having one entry node and whose nodes represent statements of the program and arcs represent the relation between a statement and the statement that can follow it within an execution /Aho 79/. The flow graph for the sample code in figure 4 is shown in figure 5.

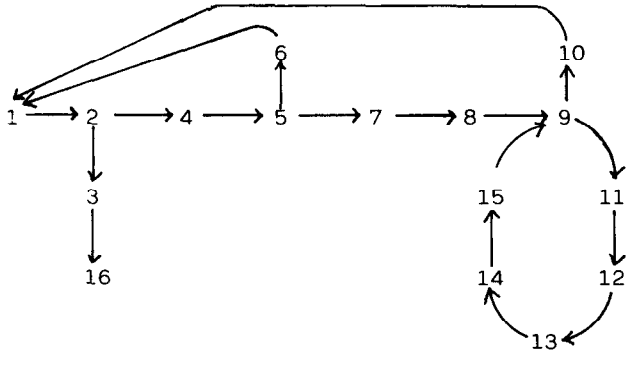


figure 5 - Sample Code Flow Graph

Node 1 in figure 5 is the flow graph entry node. In what follows we will not distinguish between a node of the flow graph and the statement it represents.

The analysis of the program flow graph is divided into the three steps:

1. Analysis of CODASYL currency and db status indicators use. It identifies the dependencies induced by currency and db status indicators among DML statements. The analysis gives what is called the "currency flow graph" in /Katz 82/. We prefer to present the results of the analysis in a tabular form.

2. Qualification of the dependencies between FIND statements. Two basic dependencies are defined between FIND statements: the navigation and the First In Record Set (FIRST) dependency. These basic dependencies are fundamental in determining the program interactions with the db.

3. Identification of db queries. A db query is a sequence of qualified dependencies. Forming db queries we have to consider both 1) the requirement of program conversion by which as much as possible of a procedural program should be replaced by a non procedural operator and 2) the capacity of the target interface to express various record selections of the source program which depend on the execution dynamics.

This section is concerned with the first two steps, where each interstatement dependency is considered by itself. The next section deals with the third step where these interstatement dependencies are considered for grouping together.

3.1 Analysis of CODASYL currency and db status indicator use.

The first step in the analysis of the program flow graph is the detection of dependencies between statements which are engendered by the use of either the CODASYL currency indicators or of the db status indicator.
The program flow graph shows the statements ordered as they are executed. The currency and db status information orders db statements detecting how they might depend one on the other. Record selection templates are matched against this second order and for each dependency we recognize which semantic access type corresponds to it.

The technique for detecting db indicator dependencies is the Global Data Flow Analysis /Aho 79/. Roughly speaking, it consists in examining the entire program to detect for every information used at every point, at which other points that information could have been defined.
In our specific case we are interested in pointing out statements which assign a value to any currency or db status indicator X and statements which have that indicator X as an operand.

The analyzed statements are then all the CODASYL DML statements which set or use currency, db status indicators and db identifiers as described in /CCJD 78/ plus the following COBOL statements
. IF on a db status value
. assignment or use (MOVE, READ, COMPUTE, WRITE,...) of db identifiers.

The analysis of the nth statement identifies

a) the set of statements which assign values to currency or db status indicators or db identifiers used by this statement. This set is indicated as P_n, (preceding of n).
By definition, a set P_n can contain any CODASYL statement and assignment to db identifiers through COBOL statements (like MOVE, READ, COMPUTE,...). No IF statement can belong to P_n.

b) the set of statements which use currency or db status indicators or db identifiers assigned by this statement. This set is indicated as D_n (depending on n).
A set D_n can contain any CODASYL statement plus COBOL IF on a db status value and statements (like WRITE, MOVE, IF, COMPUTE,...) using a db identifier.

In /Katz 82/, the Global Data Flow Analysis technique has been first used to identify how currency definitions propagate through a CODASYL application program. That approach still refers to code sequences for identifying the semantic access patterns.
The reader can refer to /Aho 79/ for a description of the general analysis and to /Katz 82/ for its particular use in identifying the currency flow. Though we extend this analysis to include db status indicator and db variables the process is unchanged and so is not described here.

The Analysis of currency and db status indicators flow on our sample code concerns the statements whose identification number appears in the left column of the Flow Dependency Table in figure 6.

statement/node	P_{node}	D_{node}
1	1	1,2,4,8
2	1	empty
4	1	5,7
5	4	empty
7	4	empty
8	1	9,11,14
9	8,14	empty
11	8,14	12
12	11	13
13	12	empty
14	8,14	9,11,14

figure 6 - FLOW DEPENDENCY table
for the Sample Program

Once sets P_n have been computed for every statement to be analyzed, sets D_n are also completely defined. Although the D_n are redundant, they are provided here for easy reference in the subsequent discussion.

3.2 Qualification of statement dependencies.
Given two distinct statements n and m, a statement dependency exists from n to m if $n \in P_m$ (and, viceversa, $m \in D_n$). Two types of dependencies between two FIND statements are distinguished: the navigation dependency and the FIRST (First In Record SeT) dependency.

Navigation Dependency
This is a dependency from the statement n to m, where $n \in P_m$ and

n is a FIND statement, either single selection or enumeration selection which selects an occurence of record1,
m is a FIND statement which selects a record2 member (respectively, owner) of any CODASYL set in which record1 selected by n partecipates as owner (respectively, member).

Record1 and record2 are usually different record types. Only the 1978 CODASYL DDL version allows the same record type to partecipate in a CODASYL set both as owner and as member record which is

called recursive set /DDLJ 78/.
A Navigation dependency from n to m is indicated as N (n,m).

In our sample case (figure 4) the dependencies from statement 1 to 8, 8 to 11, 14 to 11 are navigation dependencies. Then we have the following: N(1,8), N(8,11) and N(14,11).

FIRST Dependency

The Record set identified by an enumeration statement n is parametrized by the current record selected through a preceding FIND statement m. In other words, the set P_n of an enumeration statement n contains at least one other FIND. Only the record selection expression in figure 3.c has a default current record if no FIND belongs to P_n.

The dependency from the statement n to m, where $n \in P_m$ is called FIRST (First In Record SeT) dependency if:

n is a single selection FIND statement which selects an occurence of record1

m is an enumerative selection FIND statement which refers to a Record-Set containing occurences of record1.

We indicate such a dependency as F (n,m) and n is called FIRST statement.
For the code in figure 5, there is one FIRST dependency, which is F(8,14). The code sequence in figure 7 appeard in /Katz 82/ as an example that could not be decompiled. Statement 2 of the sequence, is a FIND statement which selects an occurence of record EMP. Statement $6 \in D_2$, is enumerative and refers to a Record-Set where occurrences of record EMP will be concerned. Hence the dependency between statement 2 and 6 is a FIRST dependency F(2,6).

```
1.   FIND ANY DEPT
     ...
2.   FIND EMP WITHIN CURRENT WORKS-IN USING
                                BIRTHYR
3.   GET EMP
4.        (print NAME,...in EMP)
5.   ACCEPT SALARY
6. A.FIND DUPLICATE WITHIN WORKS-IN USING
                                SALARY
7.   IF RECORD NOT FOUND GO TO B
     ...
8.   GO TO A
9. B.EXIT
```

figure 7.Code example

Enumeration Identification

An enumeration statement is associated with its enumeration graph which identifies the scope of the enumeration within the program flow graph. Its definition is given in the following.
We consider application programs where enumerations are performed by looping on the enumeration statement. There is a special class of program flow graphs called reducible flow graphs, in which the detection and analysis of loops is particularly facilitated. Several definitions of "reducible flow graph" have been proposed: one of them and references to the existing literature can be found in /Aho 79/. Here we only mention the properties of reducible flow graphs which are most important with regard to our analysis.

Property 1. A cycle is a set of strongly connected nodes, that is, there is a path in the flow graph from each node to every other node of the cycle which is wholly within the cycle. In a reducible flow graph every cycle has a unique entry node, such that all paths from outside the cycle to any node inside it go through that entry node. Cycles with this property are called loops.

Property 2. In a reducible flow graph two loops are either disjoint (except possibly for their entry nodes) or one is a subset of the other. This means that a nested structure can be placed on reducible flow graph loops.

Every loop $L = \{ n_1,\ldots,n_K \}$ in a program flow graph F identifies a subgraph which contains all nodes in L and the edges from F that connect two nodes both in L.
A node of a loop subgraph is called exit node from the subgraph if in the flow graph F n is connected to one or more nodes not in the subgraph. Several exit nodes can appear in a loop subgraph.
Techniques for identifying loops are presented in /Aho 79/. By using them the enumeration statements of the source program flow graph are associated each with possibly several loops and consequently with several subgraphs of F.
Given an enumeration statement m we call enumeration graph of m the subgraph EGm of F which is associated to a loop containing m and not containing any other statement on which m depends and such that every different subgraph of F associated to a loop containg m and not containing other statement on which m depends is subgraph of EGm too.

Intuitively an enumeration graph EGm contains all the statements which equally operate on each element of the set enumerated by the statement m whose execution is iterated through the enumeration loop.

The enumeration statements of our sample program in figure 4 are statements 1 and 14. Both are used for an enumeration by loop. The corresponding enumeration graphs are shown in figure 7.a where enumeration nodes appear encircled.

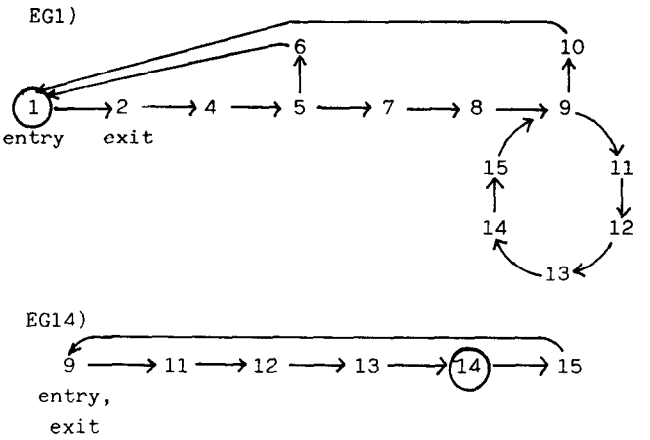


figure 7.a The Program Enumeration Graphs for the Sample program in fig.4.

4. IDENTIFICATION OF DB QUERIES

A db query is a sequence of qualified statement (navigation or first) dependencies. It will be mapped into a query in the target program.
For a given source program, several db queries are generated by a two steps analysis procedure which first analyses the program code within each enumeration subgraph and then considers db statements not belonging to any enumeration. The general strategy of the db queries generation process is informally described in the following with motivations and examples.

The goal of decompilation is to replace as much as possible of a procedural program by specification operators /Spac 81, Katz 82/. In our framework, this goal means that the identified record selections must be combined together into the longest possible sequences (i.e. db queries).

Many reasons make attaining the goal least than a complete success. The major restrictions come from the specification interface itself whether it fits, more or less satisfactorily the record selection capabilities expressed in the source program through the navigation interface and the host language control structures.

4.1 Db queries from enumeration graphs

The process of db queries identification first considers the enumeration subgraphs and analyses the nested enumerations for defining sets of statement dependencies to be all solved in a single db query.
Two enumerations having graphs EGn and EGm respectively are said to be nested if EGm is a subgraph of EGn. The enumeration having EGm is said to be inside the one having graph EGn.

As a general strategy, two nested enumerations are solved within the same db query. Exceptions to this general strategy are the following:

(i) if an enumeration inside another enumeration does not select every element of its corresponding Record-Set then the two enumerations are associated with two different db queries. To identify these cases the cardinality of the inside enumeration must be determined.

(ii) if any program instruction can be executed between two enumerations the general strategy is applied or two different queries are associated to each enumeration depending on the target specification interface.

Last, if an enumeration statement appears in two (or more) FIRST dependencies, the corresponding enumeration is called a multiple activation enumeration. In such a case at least three different db queries are generated, one for each FIRST statement and one corresponding to the enumeration itself.

Enumeration cardinality

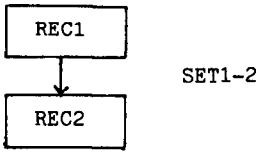
Consider the data schema and sequence code shown in figure 8 (a). For any occurence of REC1, at most N occurrences of REC2 connected by SET1-2 to current REC1 are selected.
In this example we have two enumerations. The first one concerns the occurrences of REC1, statements 12 and 1 are respectively the enumeration statement and the FIRST statement. The second enumeration concerns the occurrences of REC2, statement 5 is the enumeration statement. The corresponding enumeration subgraphs EG12 and EG5 are shown in figures 8(b) and 8(c). In EG12, the node 2 is the only exit node. Since 2 contains a condition on END OF SET we say that EG12 enumerates all the occurrences of REC1. While in EG5 one of the two exit nodes 6 and 9 corresponds to a statement which is different from a condition on END OF SET. Hence we say that the enumeration is a non complete one.

In the case of figure 8 two different db queries are associated to each enumeration keeping them independent one from the other.

EG5 : q_1= N(1,5) F (0,5)

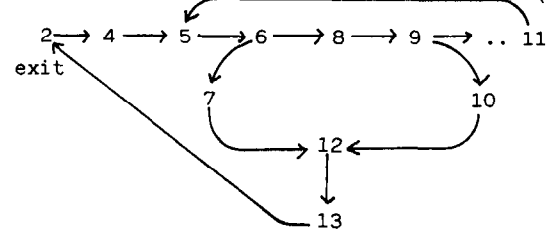
EG12 : q_2= F(1,12)

In figure 8 (a) the enumeration statement 5 has no corresponding FIRST statement. In the query q_1 this results with the F(0,5) dependency qualification.
The notation F(0,n) is used in db queries for any statement n either being an enumeration statement with no corresponding FIRST statement or being any statement depending on no other statement, i.e. with $P_n=\emptyset$.



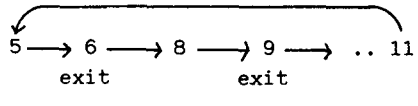

```
1.      FIND FIRST REC1 WITHIN SYST-REC1
2,3 A.  IF END OF SET GO TO D
4.      MOVE Ø TO COUNT                          ( a )
5.  B.  FIND NEXT REC2 WITHIN SET1-2
6,7     IF END OF SET GO TO C.
8.      ADD 1 TO COUNT
9,10    IF COUNT > N THEN GO TO C.
        ...
11.     GO TO B.
12. C.  FIND NEXT REC1 WITHIN SYST-REC1
13.     GO TO A.
14. D.  ...
```

EG12)



(b)

EG5)



(c)

figure 8. Enumeration analysis example

Procedural Break

Program instructions which may be executed between two enumerations one inside the other are called procedural break /Katz 82/. In figure 4, EG14 appears inside EG1. Statement 7, which is a node of subgraph EG1 only, is a procedural break between EG1 and EG14.
The two enumerations can be combined if the specification interface provides a feature to (1) group, for each element of the outer enumeration, the corresponding set of elements of the inner enumeration and (2) a primitive to deliver the grouped elements, one by one.
Otherwise the two enumerations must be solved in two different db queries.

Multiple enumeration activation statements

Consider the following code sequence which is a modification of the code skeleton shown in figure 7.

```
1.      FIND ANY DEPT
2.      IF SEARCH-BY-BIRTH = 'Y'
3.         ACCEPT BIRTHYR
4.         FIND  EMP  WITHIN  WORKS-IN  USING
                 BIRTHYR
5,6        IF RECORD NOT FOUND GO TO B
7.      ELSE ACCEPT ENAME
8.         FIND EMP WITHIN WORKS-IN USING ENAME
9,10       IF RECORD NOT FOUND GO TO B.
11.    A.GET EMP
12.      (print   EMPCODE,...)
13.      FIND DUPLICATE WITHIN-IN USING SALARY
14,15    IF END OF SET GO TO B.
16.      GO TO A.
17.    B....
```

figure 9.Multiple Enumeration activation example

In the above code the following dependencies are identified:
N(1,4), N(1,8), F(4,13), F(8,13).
The enumeration statement 13 appears in two different FIRST dependencies. The db queries generated are shown in figure 10.

q_1= F(4,13), F(8,13)

q_2= N(1,4)

q_3= N(1,8)

figure 10. Db queries after Enumeration Analysis

4.2 IF Branches Analysis

The analysis of IF branches, is the analysis of the db statements partecipating in many dependencies.

In general, two record selection statements both depending on a third statement and appearing in different branches of an IF statement are not solved in the same db query except the above enumeration solving rules.

Different branches here means that in some run one of the two statements may be executed while the other may not. A db query, i.e. a sequence of statement dependencies, is then finished when finding such a case. Two different queries are associated with the IF branches themselves. They hold different till the first common point unless other IF branches or enumerations are found. A fourth query begins for record selection statements which follow the common point.

As an example, in figure 9 code sequence the statements 4 and 8 appear in different branches of the IF statement 2. Being 4 and 8 FIRST statements, they have already been analyzed and solved in queries q_2 and q_3 in figure 10. Then a query q_4 is generated for the sequence of code preceding the IF and is kept independent from q_2 and q_3 which solve the IF branches.

$q_4 = F(0,1)$

If branches appearing in an enumeration graph are analyzed in the same way and generate db queries beside the one which solves the enumeration itself.

The set of the identified db queries is an output of the program flow graph analysis. Through the synthesis activity each db query is mapped into a specification and embedded within the application program to produce the target version. This activity is not covered in the present paper.

5. CONCLUSION AND FUTURE WORK

In this paper we presented an approach to the analysis of a database application program for the purpose of converting its db interface from a navigation to a specification one.

The analysis algorithm is based on the dependencies between statements induced by the currency and db status indicators. Statements dependencies are distinguished whether they express navigation on the db schema from one record type to another or enumeration of occurrences within the same record type.

The global semantics of the program is found out basically through the qualified inter-statement dependencies rather than by matching standard code sequences against the application code. This approach proves more flexible to different possible code sequences than previous techniques /Katz 82/.

The approach uses techniques for program flow graph analysis developed for compiler code optimization /Aho 79/. We have seen how these techniques apply to programs with a reducible flow graph which occur very frequently in practice.

Besides reducibility we require, for a program can be analyzed, that any assumed ordering of records is explicitly declared in the db schema description. In other words, programs assuming a system dependent physical order in their record management are not properly handled. Their conversion into the target interface produces a target program which is not equivalent to the source program.

Some topics which have not been analyzed are worth studying. Among them, we are currently analyzing the COBOL selection restrictions.

Within a CODASYL COBOL db application program records in which the user is interested are selected and transferred to a User Working Area (UWA) through CODASYL statements. After that, records can possibly be distinguished if further manipulated or discarded depending on whether they match some conditions or they do not. The distinction is made through COBOL IF statements evaluating predicates on UWA record fields. We refer to these cases as COBOL selection restrictions.

As an example, the sample code sequence in figure 4 selects (statement 1) and transfers to UWA (statement 4) the occurrences of record type SUPPLIER. The IF statement number 5 distinguished which occurrences are to be manipulated or discarded depending on whether the field CITY in record SUPPLIER is equal to CHICAGO or not. In case the condition is not satisfied, the record is not further considered and a different occurrence of SUPPLIER replaces it.

When a specification interface is used the COBOL selection restrictions ought to be expressed directly in the qualification term of the specification rather than as COBOL statements again. Hence the transfer load of data to UWA results optimized.

Current research also aims to complete the definition of the SCOOP-ER DML. The choice of

first working on program analysis has been found to provide fruitful suggestions for the DML definition also. An example has been given in section 4.1 of the present paper.

Acknowledgements

The author is grateful to S.Spaccapietra and the SCOOP group: this research has been mostly developed while she was visiting the University of Paris VI.

The comments of S. Navathe and of S. Kundu on earlier versions of this paper and the discussions with S. Su and H. Lam were particularly helpful.

REFERENCES

/Aho 79/ Aho, A.V. and Ullman, J.D., Principles of Compiler Design, Addison Wesley, Reading, Mass., 1979.

/CCJD 76/ CODASYL COBOL Journal of Development, 1976

/DDLJ 78/ CODASYL Data Description Language Committee, DDL Journal of Development, 1978

/Katz 82/ Katz, R.H. and Wong, E., Decompiling CODASYL DML into relational queries, ACM Trans. on Database Systems, Vol.7, No.1, March 82.

/Lam 79/ Lam, H., A Generalized System for application program conversion to account for database semantic changes. Design and prototype implementation, PHD Thesis of the University of Florida, 1979.

/Moor 80/ Moore, L.K., The Decompilation of COBOL-DML programs for the purpose of program conversion, M.S. Thesis, University of Florida, 1980.

/Moor 82/ Moore Dorsey, L. and Su, S.Y.W., The Decompilation of COBOL-DML programs for the purpose of program conversion, Proc. COMPSAC Conf., Chicago, Oct., 1982.

/Nati 78/ Nations, J., Su, S.Y.W., Some DML instruction sequences for applications program analysis and conversion, Proc. SIGMOD Conf., 1978.

/Pare 83/ Parent, C., Spaccapietra, S., An entity-relationship algebra, Report MASI n°17, Institut de Programmation, Univ. Paris VI, March 83.

/Spac 80/ Spaccapietra, S., Heterogeneous Data Base distribution, in Distributed Data Bases, Draffan and Poole eds, Cambridge Univ. Press., 1980.

/Spac 81/ Spaccapietra S., et all., An approach to effective heterogeneous databases cooperation, in Distributed Data Sharing System, North-Holland, 1982.

/Spac 83/ Spaccapietra, S., Demo, B., Parent, C., SCOOP: A System for integrating existing heterogeneous distributed data bases and application programs, Proc. of the INFOCOM Conf., San Diego, April 1983.

/Shne 82/ Shneiderman, B., Thomas, G., An Architecture for Automatic Relational Database System Conversion, ACM Trans. on Database Systems, Vol.7, No. 2, June 82.

/Su 81/ Su, S.Y.W., Lam, H., Lo, D.H., Transformation of Data Traversals and Operations in Application Program to Account for Semantic Changes of Databases, ACM Trans. on Database Systems, Vol. 6, No. 2, June 1981.

/Tayl 79/ Taylor, R.W. et all., Database Program Conversion: a Framework for Research, Proc. VLDB Conf., Rio de Janeiro, 1979.

/Tsic 82/ Tsichritzis, D.C., Lochovsky, F.H., Data Models, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

CR Categories and Subject Descriptors: H.2.3: Database Management, Languages, Data manipulation languages, H.2.5: Database Management, Heterogeneous Databases, Program Translation

Additional Key Words and Sentences: Global dataflow analysis, specification and navigation data manipulation languages, CODASYL

This research was partly supported by the Progetto Finalizzato Informatica (PFI) of the Italian C.N.R. and by the French Government.