Decompiling CODASYL DML into Relational Queries

R. H. KATZ and E. WONG
University of California at Berkeley

A "decompilation" algorithm is developed to transform a program written with the procedural operations of CODASYL DML into one which interacts with a relational system via a nonprocedural query specification. An Access Path Model is introduced to interpret the semantic accesses performed by the program. Data flow analysis is used to determine how FIND operations implement semantic accesses. A sequence of these is mapped into a relational query and embedded into the original program. The class of programs for which the algorithm succeeds is characterized.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*data manipulation languages (DML)*; *query languages*; *CODACYL*; H.2.5 [**Database Management**]: Heterogeneous Databases—*program translation*

General Terms: Algorithms

Additional Key Words and Phrases: decompilation, semantic data models

1. INTRODUCTION

Program conversion is concerned with the changes that must be made within an application program whenever there is a change in the representation of the database which it accesses. These changes can occur because a database is either converted from one database system to another or restructured within a single database system. This paper focuses on the changes in applications programs that are caused by converting between database systems that support a different level of procedurality in the data sublanguage. In particular, we present an algorithm for mapping from the procedural operations of CODASYL DML [3] into the nonprocedural relational calculus [4].

Program translation has been of interest to researchers for several years. Housel [6] proposes a system wherein programs are converted by "decompiling" a source program into an abstract specification. The specification can be reexpressed in terms of the target data model and tailored to the target database. It can then be recompiled into the host and manipulation languages of the target system. However, it is difficult to formulate ways to specify the reexpression of the program in terms of the target data. The paper suggests that a translation analyst be responsible for this.

This research was supported by the U.S. Army Research Office Grant DAAG29-76-G-0245, the U.S. Air Force Office of Scientific Research Grant 78-3596, the Honeywell Corporation, and an I.B.M. Fellowship for R. H. Katz.

Authors' present addresses: R. H. Katz, Computer Sciences Department, University of Wisconsin-Madison, 1210 Dayton Street, Madison, WI 53706; E. Wong, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

Another approach is being investigated at the University of Florida [9, 12-15]. A semantic data model to characterize the way in which data are used by an application program is proposed. Language templates [10] are matched against the program to recognize instruction sequences which correspond to particular types of semantic accesses. The matching associates with each instruction sequence an access path graph that is a model-dependent representation of a semantic access pattern. A query graph is constructed from a sequence of these graphs. The query graph can then be decomposed into access path graphs for the target system. The program is recompiled by finding the instruction sequences in the target host and manipulation languages that correspond to the access patterns of the graphs. Unfortunately, the authors do not articulate actual algorithms for realizing the conversion process outlined above. We agree that a semantics-based approach is the most appropriate, and it is the approach taken in this paper. Our contribution is to develop the actual algorithms for mapping sequences of procedural operations into a nonprocedural query specification, and to integrate the conversion process with information gleaned from the database's physical design specification. A more complete review of the literature can be found in [16].

In the rest of this paper we develop the algorithms necessary to decompile CODASYL DML into relational queries. First we analyze the CODASYL DML program to characterize its FIND operations in terms of the records they potentially access. This characterization is parameterized by currency information, from which a "currency flow graph" is formed. Some of the parameters are ambiguously defined. This is rectified by applying transformations to the graph to obtain a "reduced" currency flow graph. An access path expression is derived from the information in the graph, from which relational queries are generated. Finally, these are embedded into the program, replacing the original DML statements and their associated control statements. In the final section we characterize the programs for which our decompilation is successful.

2. QUERY SPECIFICATION

In this section, we briefly introduce the way queries are specified in CODASYL DML and the relational calculus. Figure 1 contains the realizations of an example database in the CODASYL and relational models. In the former, arrows represent sets and point from owner to member. In the latter, the attributes WORKS-IN and ASSIGN in EMP are foreign keys of DEPT and JOB, respectively. Similarly, ENO and JID in QUAL are foreign keys of EMP and JOB.

In the CODASYL model, a query is specified as a sequence of FIND statements that identify the "current record of the database." The current record can be deleted (ERASE), updated (MODIFY), or retrieved (GET). The program communicates with the database system via a user work area (UWA), which contains a record structure for each record type in the database schema. The current record can be transferred into the UWA by executing a GET. A new record is entered into the database by storing values for its data items into the UWA and then executing a STORE operation. Values of data items in the UWA or in current records can be used as parameters in subsequent FIND statements. The current state of the database consists of all currency indicators (for each record type and set type) and the values of data items in the UWA. DML statements are embedded within the program as a syntactic extension of a host language.

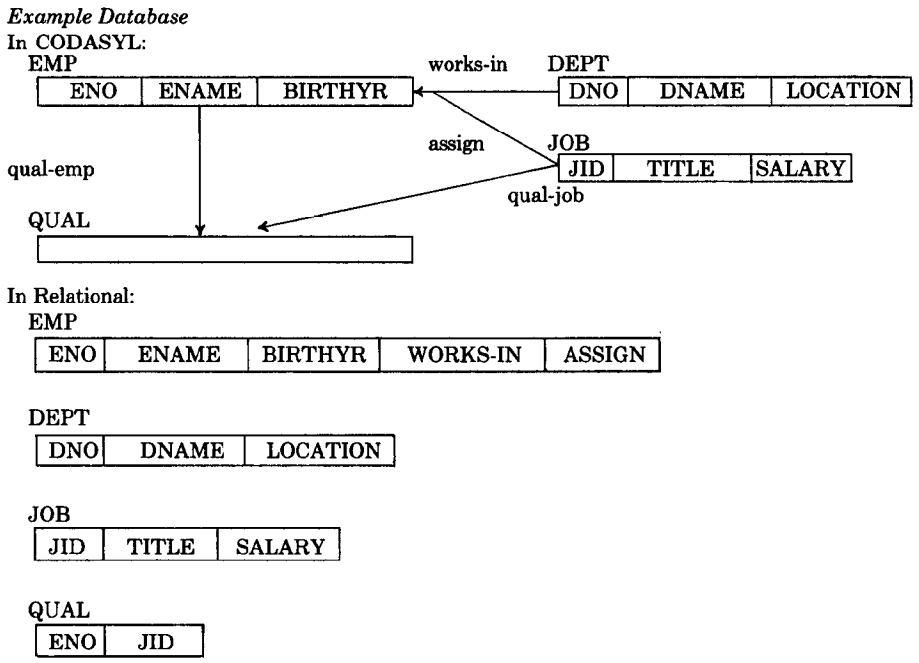


Fig. 1. Schemata for the example database.

In the relational model, the data accessed or updated are specified in terms of a qualification over a Cartesian product of relations. The qualified tuples are projected to obtain the desired attributes. For a concrete syntax, we use the QUEL sublanguage but restrict it to be aggregate free [11]. For example, the query to find the department that employee Fred works in can be written as

```
RANGE OF E IS EMP
RANGE OF D IS DEPT
RETRIEVE (D. DNAME) WHERE E. ENAME = 'Fred'
                    AND   E. WORKS-IN = D. DNO
```

E and D are called tuple variables and are shorthand names for specific relations. The list of attributes within parentheses is called the "target list" and specifies the attributes to retrieve. The qualification following WHERE is called the "where clause." The Cartesian product EMP × DEPT is restricted to those tuples for which the qualification is true. The qualified tuples are projected onto the attribute DNAME.

We define a tuple-at-a-time, cursor-based relational interface as follows. A *query cursor* is associated with each RETRIEVE statement. When OPENed, the query's results are computed. When SELECTed, the next tuple for processing is made ready for transfer into the user work area. When FETCHed, the data are actually transferred. Finally, the cursor must be CLOSEd before it can be reopened and the query reexecuted. A tuple id (TID) is associated with each tuple to encode its location within secondary storage. It is stored in the UWA as a side effect of the SELECT operation. The interface is similar to the one defined

for System-R [2], and although the data to be accessed are specified nonprocedurally, the program still accesses a tuple at a time for processing.

3. DECOMPILATION

Decompilation is the inverse of compilation; it is the process of grouping a sequence of procedural record-at-a-time statements, which represent a plan to process a query into a nonprocedural specification. We are interested in decompiling CODASYL DML programs into relational calculus programs with a tuple-at-a-time interface.

A complete analysis of the applications portion of the program is expensive. Our approach requires information about the host program's control structure, its embedded sublanguage statements, and its interaction with the database system via the user work area.

Decompilation proceeds in two phases. The first is analysis. During this phase we seek to group together a sequence of FIND statements which reference the same logically definable set of records, and to compose these sets whenever possible. In order to do this, we shall make use of an access path model [8]. The result of the first phase is the mapping of a DML program into an access path expression. The second phase is embedding, in which the access expression is mapped into a relational query and interfaced with the original program.

3.1 An Access Path Model

The access path model introduced in [8] consists of sets of objects and functions between them. Any CODASYL schema can be transformed uniquely into an access path schema by identifying

sets: value sets (i.e., field types) and record types;
functions: data items and owner-coupled sets.

A data item is a function mapping a record type into a value set (e.g., ENAME maps employee records into character strings), while an owner-coupled set is a function mapping the member record type into the owner record type (e.g., WORKS-IN maps employee records into department records).

For example, in the CODASYL schema of Figure 1, works-in, assign, and salary are functions

dname: dept → string
works-in: emp → dept
assign: emp → job
salary: job → integer

where → indicates mapping from the domain to the range. An access path schema can be represented in a natural way as a graph in which nodes correspond to sets and arcs represent functions.

Let f be a function in the access path schema, e an element in its domain, and r an element in its range. Then the (singleton) set $A_e = f(e)$ and the set $B_r = f^{-1}(r) = \{e \mid f(e) = r\}$ are each called *atomic access units*. Each represents the set of objects derived from the application of f or f^{-1} to a single parameter.

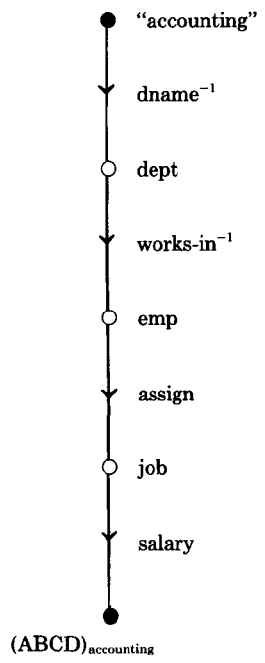


Fig. 2. Example access path expression graph.

If A_x and B_y are two atomic access units such that $\{B_y \mid y \in A_x\}$ is well defined (i.e., B_y is parameterized by objects in A_x), then A_x and B_y can be *composed* $(AB)_x = \{B_y \mid y \in A_x\}$, and a composed sequence of compatible access units will be called an *access path.*

The following are atomic access units:

$$A_w = \text{dname}^{-1}(w) = \{\text{dept} : \text{dname(dept)} = w\}$$

$$B_x = \text{works-in}^{-1}(x) = \{\text{emp} : \text{works-in(emp)} = x\}$$

$$C_y = \text{assign}(y) = \{\text{job} : \text{job} = \text{assign}(y)\}$$

$$D_z = \text{salary}(z) = \{\text{salary(job)} : \text{job} = z\}$$

Composing these, we have the access path expression

$$(ABCD)_w = \text{salary(assign(works-in}^{-1}(\text{dname}^{-1}(w))))$$

and $(ABCD)_{\text{accounting}}$ is the set of all salaries of the jobs to which the employees of the accounting department are assigned.

An atomic access unit is simply a set defined by a mapping between sets. An access path is a composition of those mappings. Alternatively, an access path is a path in the schema graph. For example, the access path for the sample query is shown in Figure 2.

3.2 Analysis

3.2.1 *Characterization of DML Operations.* For each CODASYL FIND statement we shall define a cursor and an object set. A cursor is a pair (index, ordering

specification). An object set is a set of records of the same type that are derived from expressions over atomic access units. If the records of an object set are ordered according to the specification in the cursor, then the index provides an offset into the sequence that locates the record made current by the FIND statement. Roughly speaking, the cursor picks off the purely procedural part of the statement, while the object set is a set of records likely to be repeatedly enumerated. FIND statements with the same object sets are candidates for generating an atomic access unit. The atomic access units so formed will then be composed to form paths wherever possible.

We define the object set and the cursor for the different formats of CODASYL FIND statement as follows. The type 1 find statement enumerates the instances of a record type which optionally match a specified UWA value for a data item or group of data items:

{FIRST, NEXT} record-name [USING data item$_1$, . . .]

Object Set = {record-name ∈ RECORD-NAME ∩ DATA ITEM$_1^{-1}$(current UWA variable) . . . }

Cursor =
FIRST ⇒ ⟨1, record type ordering specification⟩
NEXT ⇒ ⟨c_{index} + 1, record type ordering specification⟩

Record-name and data item$_1$, . . . are filled in from the find statement itself. For example, "FIND FIRST EMP USING BIRTHYR" requests that the current record become the employee record with a value for birthyr that matches the UWA. The object set definition is {emp ∈ EMP | BIRTHYR^{-1}(emp.birthyr)}, that is, the set of employee objects in the access mapping BIRTHYR^{-1}, parameterized by emp.birthyr. If the USING clause appears in the statement, then the term RECORD-NAME can be dropped. The cursor definition is ⟨1, ordering specification⟩, where the specification is extracted from the description of the CODASYL schema, for example, "ASCENDING ON ENAME." The actual object accessed is the employee, born in the specified year, who appears first among the employees listed alphabetically.

The type 2 find statement enumerates the record instances of a record type with a specified key data item value. It is also possible to access a record with the same key value as the one previously accessed:

{ANY, DUPLICATE} record-name USING key-name

Object set =
ANY ⇒ {record-name ∈ KEY DATA ITEM^{-1}(current UWA variable)}
DUPLICATE ⇒ {record-name ∈ KEY DATA ITEM^{-1}(KEY DATA ITEM (current record-name RECORD))}

Cursor =
ANY ⇒ ⟨1, duplicate key ordering specification⟩
DUPLICATE ⇒ ⟨c_{index} + 1, duplicate key ordering specification⟩

A request to find a record with a duplicate key value results in a definition with a parameter for the key value from the current record of the record type. This kind of information cannot be determined directly from the statement, but rather from how other statements interact with it in terms of the program's control flow.

In particular, we need to know the last statement executed which could have accessed a record of the specified type. The duplicate key ordering specification for the cursor definition is found in the schema definition.

The type 3 find statement will access a record within a set with the same selected data item values as the current record of that set:

DUPLICATE WITHIN set-name USING {data item$_1$, $\cdots$}

Object set = {member record name $\in$ SET NAME^{-1}(current owner record) $\cap$ DATA ITEM$_1^{-1}$(DATA ITEM$_1$(current member record)) $\cdots$}

Cursor =
⟨c_{index} + 1, set type ordering specification⟩

The two parameters for the type 3 object set definition are for the current owner record of the set and the data item values from the current member record of the set. The set type ordering specification is extracted from the schema definition. The object set describes all member records of a set instance, identified by the current owner record, that have the specified data item values that match those of the current member record.

The type 4 find statement is used to iterate over the members of a particular set instance:

{NEXT, PRIOR, FIRST, LAST, integer, variable} record-name WITHIN set-name

Object set = {record-name $\in$ SET NAME^{-1}(current owner record)}

Cursor =
NEXT $\Rightarrow$ ⟨c_{index} + 1, set ordering specification⟩
PRIOR $\Rightarrow$ ⟨c_{index} − 1, set ordering specification⟩
FIRST $\Rightarrow$ ⟨1, set ordering specification⟩
LAST $\Rightarrow$ ⟨max index, set ordering specification⟩
integer $\Rightarrow$ ⟨c_{index} + integer, set ordering specification⟩
variable $\Rightarrow$ ⟨c_{index} + variable, set ordering specification⟩

The object set definition describes all the records in the current set instance. The ordering specifications are found in the appropriate places in the schema definition.

Type 5 find statements are not supported by decompilation. In this format, a record is accessed by providing the system with a database key, which is an encoding of the record's physical address. A database key is only guaranteed to reference the same record across a single execution of the program. This makes this type of statement unsuitable for the preexecution analysis of the next section.

The type 6 find accesses the owner record of the current instance of a specified set:

OWNER WITHIN set-name

Object set = {owner record name $\in$ SET NAME(current member record)}

Cursor = ⟨1, —⟩

The object set contains the single owner record of the current member of the set instance. No ordering specification is needed for the cursor.

The type 7 find locates a record, within the current instance of a specified set,

which has data item values that match those of the UWA:

record-name WITHIN set-name CURRENT [USING data item$_1$, $\cdots$]

Object set = {record-name $\in$ SET NAME^{-1}(current owner record) $\cap$ DATA ITEM$_1^{-1}$(current UWA variable) $\cdots$}

Cursor = ⟨1, set type ordering specification⟩

The parameters are the current owner record and the current values of UWA variables specified in the optional USING clause. The first record in the set ordering that simultaneously matches all the USING clause items is the one found.

It is obvious that the object sets are nearly always parameterized by "currencies." To test whether two object sets are the same, we have to compose the corresponding currencies. To do that, we need a careful analysis of the execution dynamics of the program.

3.2.2 *Flow of Currency.* A FIND operation causes a new record to be made the current of the database and its record type and set type(s). Information about a current record, used in subsequent find operations, is represented as parameters in object set definitions. In addition, data manipulation operations (MODIFY, GET, ERASE) operate on a current record.

Currency information allows us to determine that a pair of operations enumerate the same set of objects. These operations implement an atomic access unit by enumerating the objects of the associated object set. Global data flow analysis [1, 5] is a technique used to determine the flow of information through a program. It is a process of preexecution analysis which collects information about how "quantities" in a program are modified, preserved, and used. The quantities we are interested in involve the use and definition of currency information within the host language and data sublanguage statements of the program.

A program is partitioned into blocks of statements for which there is no way to enter except at the first statement, and once entered, every statement must be executed sequentially. Information is collected for each block and then propagated to other blocks using the control flow of the program, represented by a block's successors and predecessors.

We are interested in "reaching definitions." A *currency* is a currency indicator (one for the database, and for each record type and set type) or a UWA variable. A currency *definition* is a statement that can modify a currency, for example, a FIND operation or an assignment statement to a variable in the UWA. A *locally exposed definition* is the last definition of a particular currency within a block. The set of all such definitions for a block x is called GEN(x). A definition of a currency c is killed by x if it reaches the top of block x and x contains a definition of c. The set of killed definitions is KILL(x).

The propagation of currency definitions through the program are described by the following two equations:

$$\text{For each block } x, \quad \text{IN}(x) = \bigcup_{y \in \text{PRED}(x)} \text{OUT}(y)$$

$$\text{For each block } x, \quad \text{OUT}(x) = [\text{IN}(x) - \text{KILL}(x)] \cup \text{GEN}(x)$$

where IN(x), OUT(x) are the sets of definitions that reach the first and last

```
(1)    MOVE 'ACCOUNTANT' TO TITLE IN JOB.
(2)    FIND FIRST JOB USING TITLE.
(3)    FIND FIRST QUAL WITHIN QUAL-JOB.
  L.     IF END-OF-SET GO TO EXIT.
(4)      FIND OWNER WITHIN QUAL-EMP.
(5)      GET EMP.
           .
           .
           .
(6)      FIND NEXT QUAL WITHIN QUAL-JOB.
         GO TO L.
  EXIT.
```

Fig. 3. Qualified accountants.

statements within block x, respectively. Several methods have been developed to find solutions to these equations [1, 5].

Each FIND statement can create a definition for each of the possible currencies associated with the found record. A GET operation will define a currency for each data item transferred into the UWA. Assignments to the UWA create a definition for those variables.

We are also interested in the use of currency information. A currency is *used* if it is referenced by a DML operation. For example, a type 1 find uses the current definition of data item_1. These usages are represented by underlined parameters within the object set definitions. A *locally exposed use* of a currency c is a use of c within x which is not preceded by a definition of c within x. The set of these uses is USES(x). By considering both IN(x), that is, the definitions that reach a block, and USES(x) together, we can establish which definitions in the program can potentially be used by a specific operation.

These concepts can be illustrated by an example. Consider the "pseudo-COBOL" program which accesses the names of qualified accountants, shown in Figure 3.

The object set definitions for each FIND statement are

(2) {job $\in$ TITLE^{-1}(JOB.TITLE)}
(3) {qual $\in$ QUAL-JOB^{-1}(current JOB)}
(4) {emp $\in$ QUAL-EMP(current QUAL)}
(6) {qual $\in$ QUAL-JOB^{-1}(current JOB)}

The program is partitioned into four blocks. The sets of currency information after solving the data flow equations are shown in Figure 4. The format $\langle x, y \rangle$ means that currency x is defined at statement y. A summary of definition and use information for each statement is

(1) defines current TITLE
(2) defines current JOB; uses current TITLE
(3) defines current QUAL; uses current JOB
(4) defines current EMP; uses current QUAL
(5) defines current ENO, ENAME, BIRTHYR; uses current EMP
(6) defines current QUAL; uses current JOB

We define the *currency flow graph* to be $G = (V, E)$, where V represents the statements of the program and E is a set of directed edges. An edge is directed from node v_1 to v_2 if the statement v_1 defines a currency used by the statement v_2. The flow graph for the sample program is given in Figure 5.

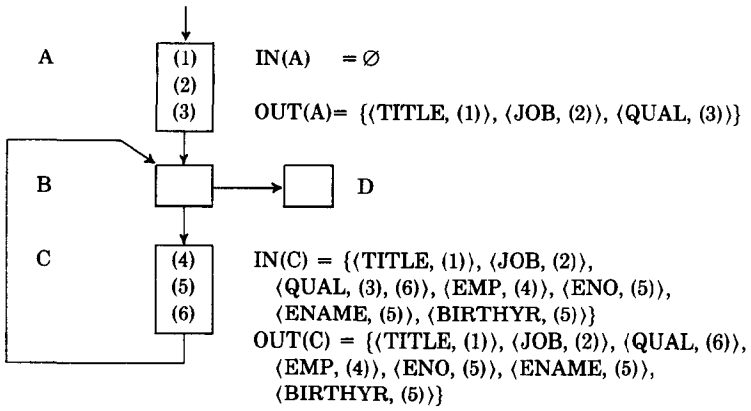


Fig. 4. Currency flow.

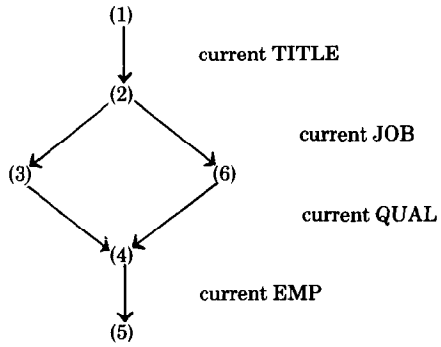


Fig. 5. Currency flow graph.

3.2.3 *Formation of Access Path Expression Graph.* The currency flow information of Section 3.2.2 identifies the object set parameters left unfilled. A difficulty is introduced whenever a parameter is ambiguously defined. For example, in the sample program, statements (3) and (6) provide the same currency information, that is, the current QUAL record, used in statement (4). An ambiguously defined parameter will cause decompilation to fail—there is no unambiguous way to form an access path description for the query.

Often, this ambiguity is caused by the way in which access units are generated in DML. A pair of DML statements is sometimes needed to specify the enumeration of the records within a CODASYL set, that is, to specify an F^{-1} access unit. Consequently, each statement of the pair provides currency information for nested statements, which results in an ambiguity. The statement pairs must be identified, and the currency flow graph modified, to eliminate the ambiguity. Three generic classes of statement pairs can be found. Associated with each class is a transformation to apply to the currency flow graph to group the statements of the pair into a single node. If the transformation cannot be applied, then the statements do not generate an access unit which can be analyzed by our approach.

Standard code sequences for inverse and functional access units are shown in Figures 6 and 7, respectively. Each access unit is parameterized by state information, such as record and data item currencies, which must be furnished by

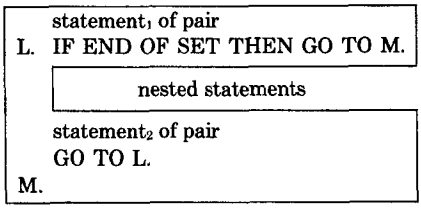


Fig. 6. Inverse mapping atomic access unit.

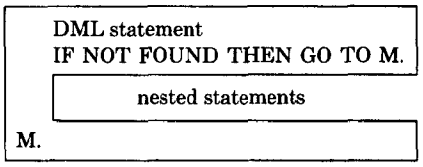


Fig. 7. Functional mapping atomic access unit.

enclosing units. The contours drawn around the code sequences are henceforth called *C-diagrams.*

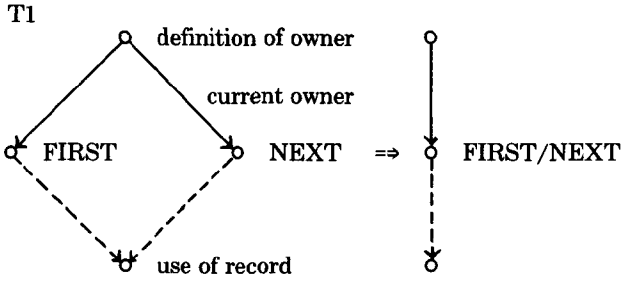
The three classes of statement pairs are

(1) *FIND FIRST/NEXT*

This class consists of the following three pairs of operations:

FIND {FIRST, NEXT} record-name WITHIN set-name
FIND {FIRST, NEXT} record-name WITHIN realm-name
FIND {FIRST, NEXT} record-name [USING data item$_1$, . . .]

The statements generate an atomic access unit, and their nodes in the currency graph can be combined, if the following conditions hold: (1) each statement uses the same definition of an owner record currency (FIND WITHIN SET) or UWA data item currency (FIND USING); (2) definitions from both statements reach all subsequent uses of this record currency. In other words, in terms of the flow of currency information through the program, the statements are completely identical, and they enumerate the same object set. Schematically, the transformation can be represented as: (the left-hand side is a subgraph of the original flow graph—it is replaced by the right-hand side)



("current owner" edge is replaced by "current data item$_1$ value" for the last statement pair)

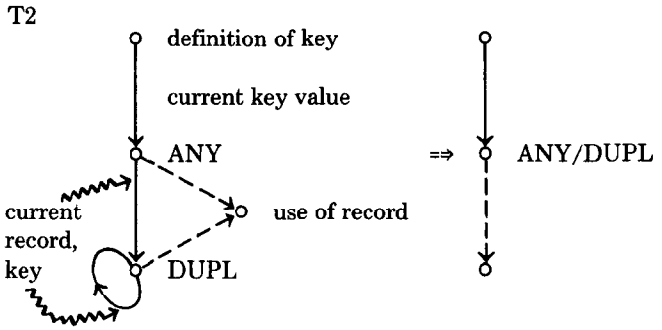
A dashed edge represents a subsequent use of a currency defined by the pair.

(2) *FIND ANY/DUPLICATE*

This class consists of the following find operation pair:

FIND {ANY, DUPLICATE} record-name USING key-name

The statements generate an access unit if (1) the FIND ANY statement defines the record currency used by the FIND DUPLICATE statement (note that the latter defines and uses its own record currency, as can be seen by the feedback in the currency flow graph), (2) the key name is the same for both statements and the key data item definition that reaches the second statement is defined by the first, and (3) definitions from both statements reach all subsequent uses of the record currency. The associated transformation is

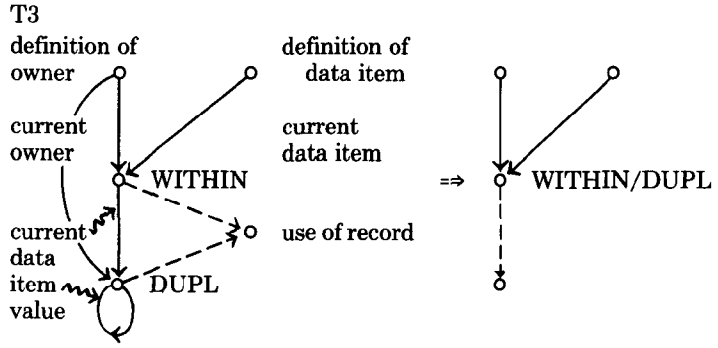


(3) *FIND WITHIN/DUPLICATE*

This class consists of the single operation pair consisting of the following two statements:

FIND record-name WITHIN CURRENT set-name USING data item$_1$, . . .
FIND DUPLICATE WITHIN set-name USING data item$_1$, . . .

These statements can be combined if (1) they use the same definition of owner record, (2) the data items specified are identical, and (3) definitions from both statements reach all subsequent uses of the set's member record type. The transformation is



The data items which appear in the USING clause must be the same for both statements.

The currency flow graph is examined for ambiguous definitions which have been caused by one of the statement pairs. This is determined by grouping together inedges of the same node which represent the same currency definition. Two nodes are candidates for combination if both provide identical currency

information to subsequent nodes. In addition, the statements must conform to the format of a C-diagram. Inedges that represent ambiguous definitions are grouped together and placed into a partition named by the nodes of which they are outedges. These edges are represented by the dashed lines in the transformations. If all the dashed edges do not appear in the same partition, that is, do not originate from the same pair of nodes, then the nodes cannot be combined and decompilation cannot proceed. The algorithm is given in Figure 8.

In the sample program, the partition ⟨(3), (6)⟩ contains the edges (3) → (4) and (6) → (4) labeled "current QUAL." All other partitions are empty. Transformation T1 can be applied to nodes (3) and (6) to obtain the *reduced* flow graph of Figure 9. Further, (3) and (6) are in the appropriate form for an inverse access unit.

Associated with each (composite) node of the flow graph is a parameterized object set definition defined by expressions over atomic access units. The access expression graph is constructed from the nodes and edges of the currency flow graph. The nodes are labeled with the object defined by the object set definition of the corresponding node of the currency flow graph. The edges are labeled with the function or inverse in the object set definition whose parameter is defined at the tail of the edge. The graphical representation of the query is shown in Figure 10.

3.3. Embedding

Once an access expression has been derived, it must be mapped into a relational query and interfaced with the host language program. If program instructions, including those to retrieve data, are potentially executed between nested C-diagrams, then the associated graph must be partitioned into subgraphs. Each of these are mapped into a relational query. Algorithms are presented for determining where to "break" the access path graphs, how to map them into a relational calculus query, and how to embed a query into the original program.

3.3.1 *Procedural Break Analysis.* Program instructions between accessing operations may induce a procedural break. Consider the following code skeleton.

```
    FIND FIRST EMP WITHIN WORKS-IN.
A. IF  END-OF-SET GO TO B.
       — application code1 —
       FIND FIRST QUAL WITHIN QUAL-EMP.
   C.  IF END-OF-SET GO TO D.
          — application code2 —
          FIND NEXT QUAL WITHIN QUAL-EMP.
          GO TO C.
   D.  — application code3 —
       FIND NEXT EMP WITHIN WORKS-IN.
       GO TO A.
B.
```

$Code_1$ and $code_3$ are executed once for each employee, while $code_2$ is executed once for each employee by each qualified job. If the access units are composed, there is no way to combine their C-diagrams so that the code is executed the correct number of times. If $code_1$ and $code_3$ are not empty, then the expression must be partitioned.

As an example, consider the program that finds the departments to which accountants born after 1950 are assigned (see Figure 11).

```
Algorithm ApplyTransformations
STEP 1: identify ambiguous parameters
  FOR EACH node DO
    IF inedges e1, e2 define the same currency THEN
      add e1, e2 to partition
          ⟨tail (e1), tail (e2)⟩
STEP 2: combine nodes
  FOR EACH partition ⟨node1, node2⟩ DO
    IF partition is not empty THEN
      IF class 1 THEN apply T1
      ELSE IF class 2 THEN apply T2
      ELSE IF class 3 THEN apply T3
      ELSE halt decompilation—failure
```

Fig. 8. Apply transformations.

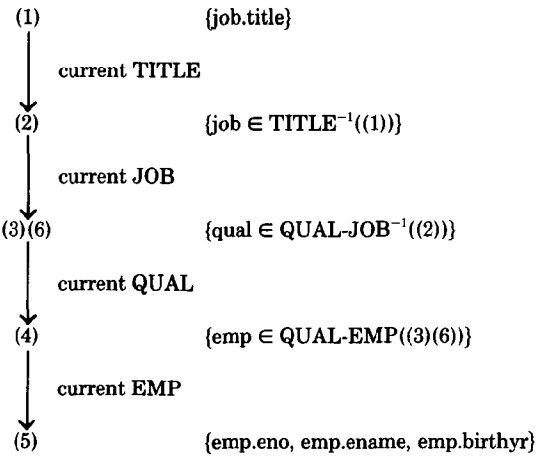


Fig. 9. Reduced flow graph.

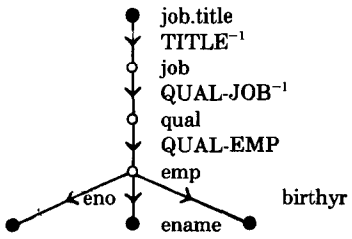


Fig. 10. Access path expression graph.

By following the steps of analysis given in the previous section, the access path expression graph of Figure 12 is derived. However, the expression cannot be mapped directly into a relational query, because the program accesses emp data values before the entire access expression has been computed.

Two composed atomic access units are *adjacent* if the C-diagram associated with the inner access unit is nested within the C-diagram of the outer unit. Further, the only code that can intervene between these is associated with other C-diagrams. No applications statements may appear. To map an access expression graph into a single relational query, every access unit must be adjacent to its

```
MOVE 'ACCOUNTANT' TO TITLE IN JOB.
(1)     FIND FIRST JOB USING TITLE.
   L.     IF NOT-FOUND GO TO EXIT.
    (2)   FIND FIRST EMP WITHIN ASSIGN.
       M. IF END-OF-SET GO TO O.
            GET EMP.
            IF EMP.BIRTHYR ≤ 1950 GO TO N.
            ⟨other code that accesses emp in UWA⟩
          (3)   FIND OWNER WITHIN WORKS-IN.
                GET DEPT.
                 .
                 .
                 .
       N.   FIND NEXT EMP WITHIN ASSIGN.
            GO TO M.
   O.     FIND NEXT JOB USING TITLE.
          GO TO L.
EXIT.
```

Fig. 11. "Young" accountants.

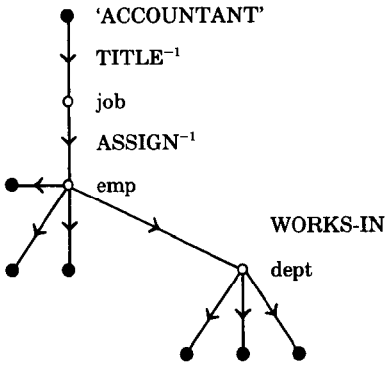


Fig. 12. Access path expression graph for 10.

successor in the graph. If not, then the graph must be partitioned along the node that both edges share in common. A procedural break is caused.

In the sample program, C-diagram (1) is associated with $TITLE^{-1}$, C-diagram (2) is associated with $ASSIGN^{-1}$, and C-diagram (3) is associated with WORKS-IN. $TITLE^{-1}$ and $ASSIGN^{-1}$ are adjacent. However, $ASSIGN^{-1}$ and WORKS-IN are not because of the program statements between their C-diagrams. The graph is partitioned along EMP (see Figure 13). The segment on the left of the partitioning edge is called the predecessor segment, because it is executed first. The right segment is the successor.

3.3.2 *CODASYL-Relational Schema Correspondence.* In [18], we have introduced reversible schema translation procedures between the CODASYL and relational data models. For our purposes here, only the CODASYL to relational direction is of interest.

For a CODASYL schema, a record type will be called *self-identified* if it is the owner of some set type or does not appear as the member of any set. For each

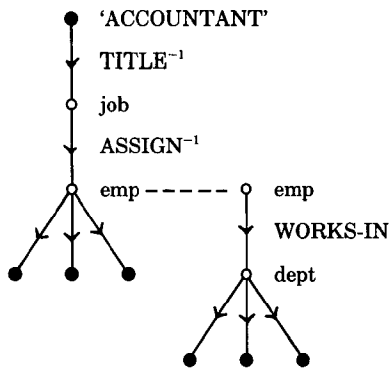


Fig. 13. Partitioned access path expression.

such record type specify (or introduce) an *identifier* which is a unique data item for each record occurrence. The schema translation proceeds as follows:

(a) Introduce a relation for each record type.
(b) The attributes of a relation consist of all data items of the corresponding record type $R(r)$ and the identifiers of the owners of every set type of which R is a member.

If the relation has been derived from a self-identified record type, then its primary key is the attribute derived from the identifier. Otherwise, the relation is uniquely identified by the attributes inherited from its owners. If the semantics indicate that these do not uniquely identify each record, then a unique identifier must be introduced. The example of Figure 1 can be obtained by applying this procedure. For example, the domains WORKS-IN and ASSIGN in EMP are defined over the key domains of DEPT (DNO) and JOB (JID), respectively. Similarly, the ENO and JID domains of QUAL are defined over the keys of EMP (ENO) and JOB (JID), respectively.

3.3.3 *Formation of Relational Queries.* A program may contain several queries, each of which is mapped into a possibly partitioned access path expression. The next step in decompilation is to derive a relational calculus query from each expression.

A *linkage term* is a clause of the relational qualification, that is, the WHERE clause, that represents how interobject relationships are implemented within the relational schema. For example, the association WORKS-IN is represented by a foreign key attribute WORKS-IN of the EMP relation. The mapping is represented by including the linkage term E.WORKS-IN = D.DNO. All edges of the access path expression, except those that represent value set functions, will be mapped into a linkage term.

The mapping between edges and linkage terms is

(1) *data item mapping*: If D is a data item that maps a record type r into a value set v, then the access unit D^{-1} (value) is mapped into the linkage term "range variable$_r$.D = value." For example, if J is the range variable for JOB, then TITLE^{-1}('ACCOUNTANT') is mapped into J.TITLE = 'ACCOUNTANT'.
(2) *set mapping*: If S is a set with owner r_1 and member r_2, then an access unit (in either direction) is mapped into the linkage term "range variable$_{r_1}$.ID =

range variable$_{r_2}$.DERIVED ATTRIBUTE." For example, if E is the range variable of EMP, then ASSIGN(E) or ASSIGN^{-1}(J) is mapped into E.ASSIGN = J.JNO. Similarly, if Q is the range variable of QUAL, then QUAL-EMP(Q) or QUAL-EMP^{-1} (E) is mapped into Q.ENO = E.ENO.

The algorithm for mapping an expression graph into a relational query is

ALGORITHM CreateRelationalQuery

For each partitioned segment, perform the following mappings:

(1) Associate a range variable with each nonvalue vertex of the graph.
(2) Value vertices with a value set mapping inedge appear in the target list as relational attributes derived from the label of the inedge.
(3) Value set inverse edges or edges which are derived from value set mappings become linkage terms in the qualification, depending upon the type of interrelationship represented.
(4) For a partitioned edge, append the following clause to the qualification of the successor query: "range variable$_{\text{LEAF}}$.TID = leaf record.TID", that is, use the TID to reaccess the previous tuple. (Note: TID is a special internal identifier used to uniquely identify each tuple within the database.)

Examples

Access Path Expression of Figure 10:

```
RANGE OF J IS JOB
RANGE OF Q IS QUAL
RANGE OF E IS EMP
RETRIEVE (E.ENO, E.ENAME, E.BIRTHYR)
WHERE J.TITLE = job.title
AND     Q.JID = J.JID
AND     Q.ENO = E.ENO
```

Access Path Expression of Figure 13:

```
RANGE OF J IS JOB
RANGE OF E IS EMP
RETRIEVE (E.TID, E.ENO, E.ENAME, E.BIRTHYR)
WHERE J.TITLE = job.title
AND     E.ASSIGN = J.JID

RANGE OF E IS EMP
RANGE OF D IS DEPT
RETRIEVE (D.DNO, D.NAME, D.LOCATION)
WHERE E.TID = emp.tid
AND     E.WORKS-IN = D.DNO
```

3.3.4 *Query Embedding.* The final step of decompilation is to interface the relational query with the original host program. An important objective is to determine which DML statements and their control structures, for example, tests for NOT FOUND or END OF SET, have been rendered superfluous by decompilation. The approach is based on collapsing together the access operations of the program, as represented by its C-diagrams, by starting with the innermost nested operations and working outward.

We present four program transformations for the cases (1) $f_1 \circ f_2$, (2) $f_1^{-1} \circ f_2^{-1}$, (3) $f_1 \circ f_2^{-1}$, and (4) $f_1^{-1} \circ f_2$. The first case involves the composition of two functional mappings. An inverse function that is not implemented by a DML statement pair is considered to be a function for the purposes of embedding. Statements which

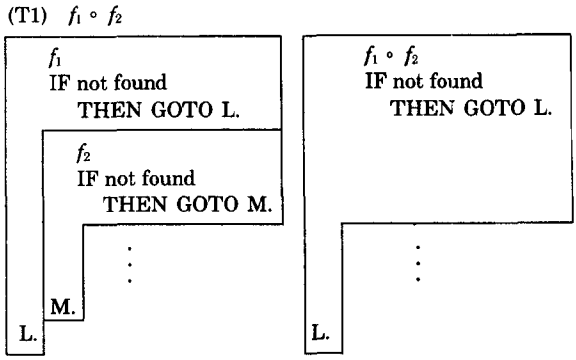


Fig. 14. Transformation T1.

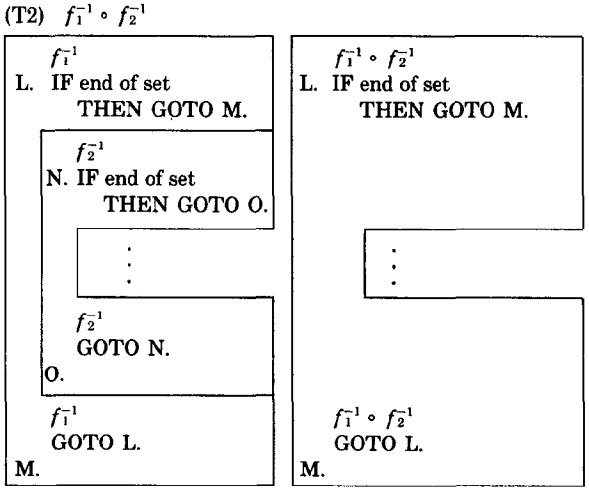


Fig. 15. Transformation T2.

appear on the left of the transformation, but not on the right, are deleted from the program. The transformation is given in Figure 14. The composed mapping can be treated as a functional access in subsequent transformations.

The second case deals with the composition of two inverse mappings. The transformation is shown in Figure 15. The resulting flowchart can be treated as an inverse mapping for subsequent transformations.

The third case handles the composition of a function and an inverse function and is given in Figure 16. The result can subsequently be treated as an inverse mapping.

The final case is the inverse of the above: composition of an inverse mapping and a functional mapping. The transform is shown in Figure 17.

Once the access units have been combined as much as possible, open and close statements must be embedded (see Figure 18). OPEN is embedded just before the first access of the expression and CLOSE is embedded just upon exit. Data accesses become SELECT statements, and GET statements are replaced by FETCHes. In the latter, all GETs for data collected over the access expression must be clustered together so they can be replaced by a single FETCH statement.

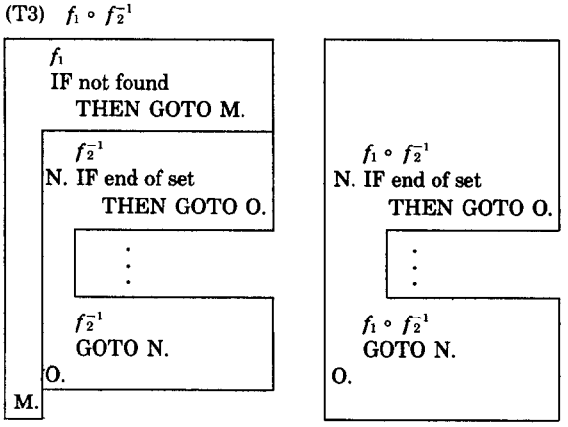


Fig. 16. Transformation T3.

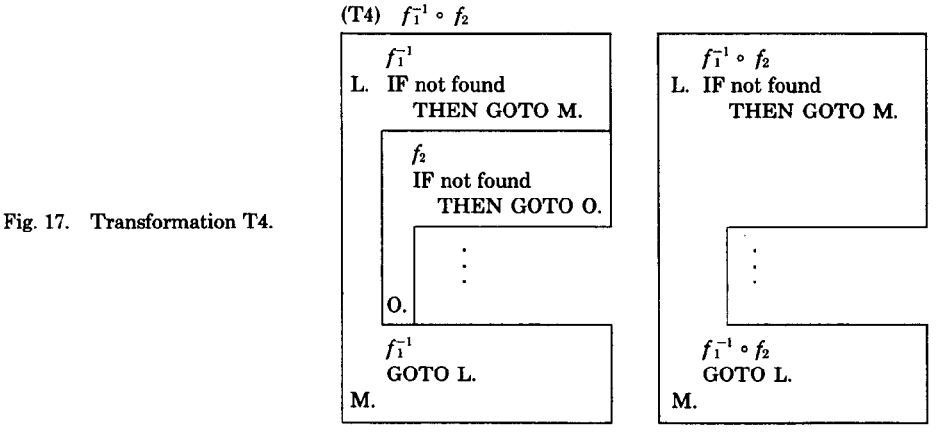


Fig. 17. Transformation T4.

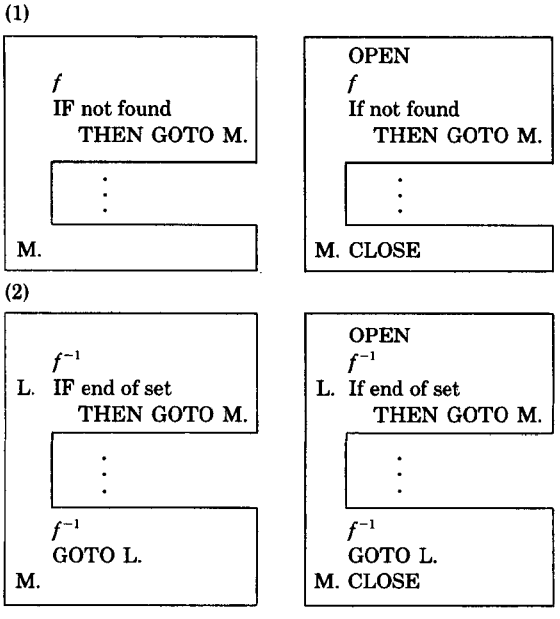


Fig. 18. Open/close embeddings.

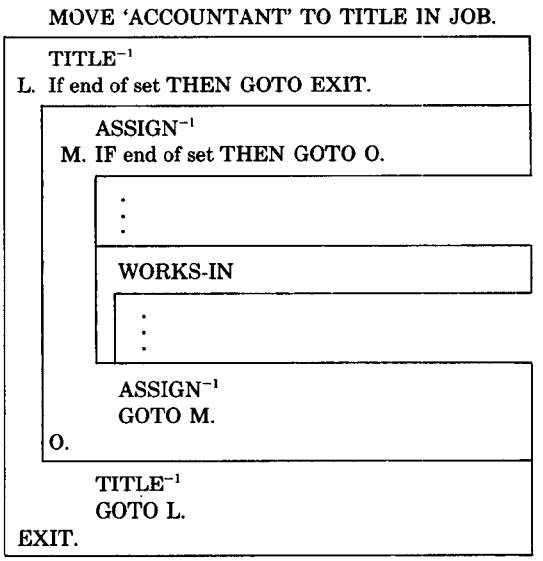


Fig. 19. Example *C*-diagrams.

The algorithm for embedding proceeds in three steps:

ALGORITHM Embed

Step 1. Create flowchart
Step 2. FOR EACH access path expression graph DO
FOR EACH mapping pair DO apply transform
apply OPEN/CLOSE embedding transform
Step 3. Embed the interface statements into the host program

We will illustrate the concepts of embedding with the program of Section 3.3.1. The program is initially represented as in Figure 19. DML find statements have been replaced by the name of their associated access mappings. For the predecessor segment's query, transformation 2 is applied to $\text{TITLE}^{-1} \circ \text{ASSIGN}^{-1}$, and OPEN and CLOSE are embedded (see Figure 20). For the successor segment's query, no transformation need be applied. OPEN and CLOSE statements are embedded (see Figure 21). The embedded program becomes

```
LET C1 BE ⟨predecessor query⟩
LET C2 BE ⟨successor query⟩
MOVE 'ACCOUNTANT' TO TITLE IN JOB.
L.
    OPEN C1.
    SELECT C1.
M. IF end of set GO TO EXIT.
      FETCH C1.
      IF EMP.BIRTHYR ≤ 1950 GO TO N.
      — other code —
         OPEN C2.
         SELECT C2.
```

```
MOVE 'ACCOUNTANT' TO TITLE IN JOB.
   OPEN
   TITLE^-1 ∘ ASSIGN^-1
L. IF end of set THEN GOTO EXIT.
      .
      .
      .
      WORKS-IN
         .
         .
         .
      TITLE^-1 ∘ ASSIGN^-1
      GOTO L.
EXIT. CLOSE
```

Fig. 20. Compose predecessor query.

Fig. 21. Final flow chart.

```
   OPEN
   TITLE^-1 ∘ ASSIGN^-1
L. IF end of set THEN GOTO EXIT.
      .
      .
      .
      OPEN
      WORKS-IN
         .
         .
         .
      CLOSE
      TITLE^-1 ∘ ASSIGN^-1
      GOTO L.
EXIT. CLOSE
```

```
         FETCH C2.
           .
           .
           .
         CLOSE C2.
N.    SELECT C1.
      GO TO L.
EXIT. CLOSE C1.
```

4. THE CLASS OF DECOMPILABLE PROGRAMS

Not every CODASYL DML program can be decomposed into one with a relational access specification. Intuitively, if the data accessed by the program can be generated by a collection of parameterized nonprocedural relational queries, then the program is decompilable.

Several properties of a program are necessary for its decompilability. First, orderings among objects must be logical rather than physical. For example, it is possible to insert a new member record into a set instance after all previously inserted members. This is tantamount to sorting the members by time of insertion. Decompilation will successfully preserve the semantics of the program only if

such orderings are supported logically, for example, by specifying a set order of ascending insertion date, with date included as a data item in each record. Physical orderings are not guaranteed to be maintained should the underlying database be converted to a different data model. In practice, this causes difficulties because existing programs often take advantage of physical orderings. In these cases, human intervention may be required to direct the translation.

Second, the decompilation algorithm is based on the assumption that data access is requested via standard code sequences. These have been selected because of their semantic relationship to access path operations. Other combinations are possible, but they cannot be decompiled by our algorithm. For example, consider the pair

(1) FIND EMP WITHIN CURRENT WORKS-IN USING BIRTHYR.
(2) FIND DUPLICATE WITHIN WORKS-IN USING SALARY.

The above pair requests all employees in the current department who make the same salary as the first employee born in a specified year. The object set associated with (1) consists of all employees within the current department born in the specified year. Associated with (2) is an object set that consists of all employees within the current department who make the same salary as the previously accessed employee. Because the object sets are not the same, the statements cannot be combined.

Last, a canonical form exists for a decompilable program. The program can be represented as a sequence of composite C-diagrams. Each composite C-diagram consists of nested C-diagrams of adjacent access units. Each composite diagram can be mapped into a single relational query. If the program cannot be placed into this form, either because of misused statement pairs or poorly structured access, then it is not decompilable.

We have characterized the class of programs that can be decompiled by the algorithm of this paper. Not every "decompilable" DML program is successfully handled. Ideally, the goal of decompilation is to replace as much of a procedural program by nonprocedural operations as possible. This can be achieved if the relational interface supports some, if not most, of the enumeration capabilities of the DML. The formulation of the appropriate interface is still an open question. A possible candidate is the link and selector language of [17]. It combines aspects of DML enumeration with a nonprocedural query specification.

Even for portions of a program that cannot be represented nonprocedurally, it may still be possible to use nonprocedural operations to severely limit the set of records over which the record-at-a-time operations are applied. In short, the goal is to do as much nonprocedurally as possible. This objective is not completely obtained with our algorithm.

Our algorithm succeeds when the currency flow graph can be reduced, because the conditions for applying the graph transformations are met. A failure is caused by a malformed atomic access unit, as indicated by nodes of the flow graph that should be combined but cannot. A malformed access unit can often be supported procedurally as outlined above. However, we have not investigated methods for finding the covering nonprocedural operations. Note that procedural enumeration operations will cause breaks analogous to those encountered during embedding. Thus a query which contains several procedural access units may be broken into several subqueries.

5. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we have presented an algorithm to map a program written in CODASYL DML into one which accesses its data via a relational interface. An access path model, used for physical database design, is also used to identify semantic accesses within the program.

The class of programs decompilable by the algorithm was characterized. More work is needed to extend the techniques of this paper to a larger class of decompilable programs. For example, more general templates for atomic access units can be found, as well as more general graph transformations. Also, algorithms for compiling relational programs into DML are a fruitful area for research. Some work in this direction is reported in [7].

REFERENCES

1. AHO, A.V., AND ULLMAN, J.D. *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.
2. BLASGEN, M.W., ET AL. "System R: An architectural update," IBM Res. Rep. RJ2581(33481), IBM Research Laboratory, San Jose, Calif., July 1979.
3. CODASYL COBOL COMMITTEE. J. Dev., 1978.
4. CODD, E.F. Further normalization of the data base relational model. In Courant Computer Science Symposia 6, *Data Base Systems*, Prentice-Hall, New York, 1971.
5. HECHT, M.S. *Flow Analysis of Computer Programs*, North-Holland, New York, 1977.
6. HOUSEL, B.C. A unified approach to program and data conversion. In Proc. Int. Conf. Very Large Data Bases (Tokyo, Japan, Oct. 1977), pp. 327–335.
7. KATZ, R.H. Database design and translation for multiple data models. Ph.D. dissertation, Univ. California at Berkeley, June 1980.
8. KATZ, R.H., AND WONG, E. An access path model for physical database design. In Proc. ACM SIGMOD Conf. (Santa Monica, Calif., May 1980), pp. 22–29.
9. NATIONS, J., AND SU, S.Y.W. Some DML instruction sequences for application program analysis and conversion. In Proc. ACM SIGMOD Conf. (Austin, Tex., May–June 1978), pp. 120–131.
10. SCHINDLER, S.J. Templates for structured DML Programs. Working Paper ST 2.2, Data Translation Project, Graduate School of Business Administration, Univ. Michigan, Ann Arbor, Feb. 1977.
11. STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of INGRES. *ACM Trans. Database Systs.* 1, 3 (Sept. 1976), 189–222.
12. SU, S.Y.W. Applications program conversion due to data base changes. In Proc. Conf. Very Large Data Bases, 1976.
13. SU, S.Y.W., AND LIU, B.J. A methodology of application program analysis and conversion based on database semantics. In Proc. ACM SIGMOD Conf. (Toronto, Canada, Aug. 1977), pp. 75–87.
14. SU, S.Y.W., AND REYNOLDS, M.J. Conversion of high-level sublanguage queries to account for database changes. In *Proc. 1978 AFIPS Nat. Computer Conf.* AFIPS Press, Arlington, Va.
15. SU, S.Y.W., ET AL. Application program conversion due to semantic changes. CIS Rep. 7879-2, Dep. Computer and Information Sciences, March 1978.
16. TAYLOR, R.W., ET AL. Database program conversion: A framework for research. In Proc 5th Int. Conf. on Very Large Data Bases, 1979.
17. TSICHRITZIS, D.C. LSL: A link and selector language. In Proc. ACM SIGMOD Conf., May 1976, pp. 123–133.
18. WONG, E., AND KATZ, R.H. Logical design and schema conversion for relational and DBTG databases. In Proc. Int. Conf. Entity-Relationship Approach to Systems Analysis and Design, UCLA, Dec. 1979.

Received June 1980; revised November 1980; accepted February 1981