ANALYSIS OF THE CONTEXT DEPENDENCY OF CODASYL FIND-STATEMENTS WITH APPLICATION TO DATABASE PROGRAM CONVERSION

G Barbara Demo
Dipartimento di Informatica Universita di Torino
42 c M D'Azeglio, 10125 Torino, ITALY

and

Sukhamay Kundu
Computer Science Department
Louisiana State University, Baton Rouge, LA 70803, USA

ABSTRACT

The CODASYL database (db) statements in an application program can have one or more different semantics associated with it, depending on the path through which the execution reaches that statement This makes the CODASYL operations in a db program context dependent The previous works on the conversion of db programs from the CODASYL record at a time interface to the set at a time interface of the relational model consider only a limited class of programs where each db statement in the program is assumed to have a unique semantic interpretation In this paper, we first define a framework for analyzing the multiple semantics of the CODASYL operations and their context dependencies We then show how to convert a CODASYL db program which contains statements having ambiguous, multiply-defined semantics In principle, the method described here allows us to convert all CODASYL db programs

Index terms Abstract and Execution contexts, CODASYL operations, Enumeration loop, Program conversion, Relational interface, Semantic predecessor set

CR-categories H 2 5, H 2 3

1 INTRODUCTION

The database program conversion problem has many facets and can be addressed from a number of different viewpoints [6-13 15] The need to convert a db program may arise due to a variety of reasons (1) migration from one dbms to another, or more generally, migration to a heterogeneous dbms environment, and (2) changes in the db semantics, etc In this paper, we consider the situation (1) and, in particular, the conversion of db-programs from the CODASYL record at a time interface to the set at a time relational interface This type of conversion is specially important for a distributed database environment because the set at a time interface minimizes the number of interactions between the application program and the dbms, and accordingly, reduces the communication overhead

One of the major problems in converting a CODASYL program is that the semantics of a CODASYL db operation often depends on the program context in which the operation is used For example, the operation "FIND DUPLICATE WITHIN <set-name> " may find records of different types depending on which member record type of the set <set-name> has been accessed most recently This context dependency often gives rise to multiple possible interpretations for many of the CODASYL db statements in a program It has been noted in [9] that the multiple interpretations cause ambiguity in the "parameters" of the relational queries for the target program, and causes the algorithm in [9] to fail in such cases It is worth pointing that the ambiguity of the CODASYL operations is in its very nature and is not to be regarded as exceptions [3 4] The objective of this paper is to (1) present a framework for analyzing the multiple semantics of the CODASYL dml in a precise and systematic manner, and (2) show how it can be used in the conversion of the CODASYL db programs The conversion algorithm given here can be considered a refinement of the algorithm in [9] In principle, our algorithm can convert all CODASYL db programs, even when there are context dependent db operations If the program contains no ambiguous statements, then our algorithm reduces to that in [9]

The paper is organized as follows In section 2, we introduce the notion of abstract context of a CODASYL operation Then we define the notion of program context for a CODASYL statement and its one or more execution contexts The concept of coherent execution contexts is introduced to classify the ambiguous interpretations into groups which are "similar" in some sense Section 3 shows how the above concepts are used in the db program conversion A Program Conversion System (PCS) based on these ideas is described in [7]

2 ANALYSIS OF CODASYL OPERATIONS

We restrict our attention here only to the CODASYL

FIND operations The FIND operations in themselves have sufficient diversity to illustrate the various ambiguity issues and our conversion techniques We define the notion of the abstract context of a FIND operation based on its semantics as defined in [3] Each FIND operation is then considered within the context of a db program and we define the notion of its execution contexts

2 1 The CODASYL FIND Operations

For each FIND operation we identify its

(1) abstract context,

(2) the result record type, and

(3) the semantic predecessor set

The *abstract context* of a FIND operation m is a set of parameters or values that a CODASYL database system requires for an execution of m There are two types of such values (a) the values which come from the system area, namely, the currency indicators of the run unit, the records, and the sets, and (2) the values which come from the user working area (UWA), namely, the values of one or more db identifiers We write ac(m) for the abstract context of m We define the result of a FIND operation by specifying the type and the particular occurrence of the record identified

The *semantic predecessor set* sps(m) of a FIND operation m is defined to be a minimal set of db-operations which together define the value of ac(m) The sps(m) may include the CODASYL and also the host language statements The term "predecessor" here indicates that the operations in the sps(m) must precede m along each execution path in the program leading to m in order for them to completely define the value of ac(m) prior to the execution of m The concept of sps is closely related to the notion of "basic instruction sequences" in [10] The sps and the abstract context for the most commonly used FIND operations are shown below

1 m FIND ANY rec-m

Here, "rec-m" denotes the record name referred to in the operation m We assume here that the calculation-key of rec-m is the attribute(s) attr-m

Abstract context

a) System (none)

b) UWA attr-m of rec-m

Result

a) Record type rec-m

b) Occurrence the record whose attr-m value equals the attr-m value in UWA and has the lowest db-key (computed from the attr-m value) among all such records

Semantic predecessor set

A MOVE-statement setting the value of attr-m

2 m FIND DUPLICATE rec-m

The most common use of this operation is in the situation where the current record of the run unit is of the same type as rec-m However, this is not required to be the case according to the CODASYL rules

Abstract context

a) System db-key of the current record of the run unit and the value of its attr-m

b) UWA (none)

Result

a) Record type rec-m

b) Occurrence the record whose calculation-key attribute value is equal to that of the run unit and has the smallest db-key larger than that of the run unit (Here, the calculation-key attribute of rec-m must also be an attribute of the run-unit record type)

Semantic predecessor set

A statement setting the current record of the run unit, which may be different from rec-m

3 m FIND DUPLICATE WITHIN set-m USING attr-m

In this case, a member record of the set-m is identified which has the same type as that of the current member record in set-m Note that set-m can have several member record types (here as well as in the categories (4)-(6)) In particular, the record type selected by m is not determined by the operation m itself (cf the categories (1) and (2) above), but by the abstract context of m Here, attr-m is an arbitrary attribute(s) of rec-m

Abstract context

a) System Value(s) of attr-m in the current member record of set-m and the current of set-m

b) UWA (none)

Result

a) Record type same as that of the current member record of set-m

b) Occurrence the record which follows the current member record of set-m according to the set ordering criteria in set-m and which has the same attr-m value(s) and type as those of the current member of set-m

Semantic predecessor set

A statement setting the current of set-m, which can be any of the member record types of set-m

4 m FIND NEXT rec-m WITHIN set-m

This statement is generally used to scan the member records of type rec-m belonging to a particular set of type set-m, irrespective of their attribute values (cf category (3)) (The analysis of the other variations of m where NEXT is replaced by PRIOR, LAST, etc are similar and are omitted here)

Abstract context

a) System current of set-m

b) UWA (none)

Result

a) Record type rec-m

b) Occurrence the record which follows the

current member record of set-m relative to the set ordering criteria and belong to the current set-m The occurrence is the first member record of type rec-m in that set if the current of set-m is of type owner record of set-m

Semantic predecessor set
A statement setting the current of set-m, which may be any of the member record types or the owner record of set-m

5 m FIND OWNER WITHIN set-m

Abstract context
a) System current of set-m
b) UWA (none)

Result
a) Record type owner of set-m
b) Occurrence the record which is the owner of current set-m

Semantic predecessor set
A statement setting the current of set-m, which may be any of the member record types of set-m

6 m FIND rec-m WITHIN set-m CURRENT USING attr-m

Here, attr-m must be an attribute of rec-m, but otherwise arbitrary

Abstract context
a) System current of set-m
b) UWA value of attr-m in rec-m

Result
a) Record type rec-m
b) Occurrence the member record in the current set-m which is of type rec-m and is the first record of that type relative to the set ordering criteria in set-m, and which has its attr-m value(s) equal to that in UWA

Semantic predecessor set
A statement setting the current of set-m, which may be any of the member record types in set-m, and a statement setting the attr-m in UWA rec-m (It is possible that the two statements are one and the same, for example, m itself can constitute an sps of m as in the case of a loop containing m)

For the FIND operations not shown here, the abstract context and the sps's are defined in an analogous manner The same principles also apply to the other db operations like STORE, etc For simplicity, we limit our discussion here only to the FIND operations(1)-(6) Note that all sps's except for the category (6) have a single element

2 2 The FIND Statements in A DB Program

The *program context* pc(m) of a FIND statement m consists of those statements in the program which set the values of one or more components of ac(m) In other words, pc(m) consists of the statements that can potentially influence the semantics of an execution of m For each component of ac(m) and for any execution path in the program leading to m only one statement which is nearest to m and sets the value of that component is included in pc(m) Thus, there can be many statements in pc(m) which defines the value of the same component of ac(m) if there are many execution paths to m Note that the role of sps(m) is simply determining the types of statements that may belong to pc(m) The determination of pc(m) from the analysis of the source program is easily done by using the global dataflow analysis techniques which is well-known in the theory of compilation and code optimization [1]

The sps of m = "FIND ANY rec-m" (category (1)) contains one statement operating on rec-m The sps(m) for statements in the categories (2)-(5) contains one statement operating on "the current of the run unit", which in principle can be any record of the db schema The sps(m) for statements in the categories (3)-(6) contains one statement operating on "the current of the set-m" which can be either the owner of set-m or one of the member record types of set-m This statement in the sps for the categories (2)-(6) which may operate on different record types is called an *ambiguous definition statement* We refer to the sps's of the categories (2)-(6) as ambiguous For categories (2)-(5), the (unique) statement in the sps is the ambiguous definition statement For category (6), the ambiguous definition statement is the one which sets the current member of set-m In this case, the ambiguity exists both for the multiple and for the single member type sets simply due to the fact that the owner record type of a set is different from the member record types We note that except for the category (3), the result record type of a FIND operation is always determined unambiguously One may view the ambiguity of a statement m as the differences in the record types in ac(m), any differences in the result record type of m is simply a consequence of the former

An *execution context* ec(m) of a FIND statement m is an instantiation of the ac(m) To be more precise, it is a minimal subset of the statements in pc(m) which belong to a single execution path leading to m and which completely defines an instance of the ac(m) One can alternatively view an ec(m) to be an instantiation of the sps(m) The ec(m)'s form a decomposition of pc(m), which may not be disjoint Fig 1(c) shows the program context and the execution contexts for each FIND statement in the program in Fig 1(b) Here, the currency of the D-C set which is used in statement 9 is determined by each of the statements 4 and 7 The statement 9, which is contained in a loop, belongs to its own program context

Two execution contexts $ec_1(m)$ and $ec_2(m)$ of a statement m are said to be *coherent* if the instantiations of the ambiguous definition statement in sps(m) operate on the same record type in both the $ec_i(m)$'s The executions of m associated with two coherent ec(m)'s not only give the same result record type, but the associated instantiations of the ac(m) also have the record types They do not, of course, necessarily result in the same record occurrence since the ec(m)'s may define different values of attr-m, currency of set-m, etc The global dataflow analysis in [1] can easily detect, but not resolve, the incoherencies at the precompile time In Fig 1(c), the statements 4 and 9 operating on the record type COURSE and statement 7 operating on the DEPARTMENT record type implies that $ec_1(9) = \{4\}$ and $ec_3(9) = \{9\}$ are the only coherent ec's for m = 9 The coherency of the ec's is an equivalence relation for any given m

3 THE PROGRAM CONVERSION

We now describe the program conversion algorithm using the notion of execution context We use the ec's to classify the ambiguities in the CODASYL db operations in the program and we essentially define one operation of the relational db interface for each distinct ec

3 1 The Data Schema Mapping

Any program conversion method tacitly assumes an underlying mapping of the source data model to the target data model In the present case the CODASYL network model is mapped to the relational model [16] We map a record type in the CODASYL schema into a relation type which contains, in addition to the original fields of the record type, one additional attribute for every CODASYL set in which it is a member The "value" of such an attribute is simply the tuple identifier (TID) of the owner record of the CODASYL set See Fig 1(d) We refer to these TID's as the set-TID's

3 2 The Relational Interface

We assume that the target interface to the relational data model is a QUEL-like language which contains both declarative primitives for query definition and operational primitives for manipulation of query value sets [9, 14] There are three types of operational primitives The OPEN operation evaluates and defines the *value set* of a query If a query is reOPENed, its previous value set is destroyed and a new value set is computed The OPEN primitive provides a set oriented interace to the program with the relational database A lower level, record oriented interface to the query value set is provided by the notion of a "cursor" [2, 5, 9] The SELECT operation moves the cursor to the next data item in the value set, and the FETCH operation transfers the corresponding data to the UWA A query value set is thus assumed to be ordered in one way or other, which may be arbitrary or as per the user definition via the CODASYL data definition The successive SELECT and FETCH operations transfer the data from the value set according to this order We use the term query and cursor synonymously when there is no confusion likely We refer to the position of a record in the value set via its tuple identifier, "pos TID" In this paper, we use a slightly extended form of the QUEL in [9], where we allow the use of "pos TID" in the formulation of where-clause of a query For example, "pos TID > p" may be used to restrict the selection to a subset of the records which follow the position p in the ordering The main difference between the FIND operation in CODASYL and the SELECT operation in QUEL is that the former involves a database activity while the latter does not The FETCH operation in CODASYL and the GET operation in QUEL are on the other hand equivalent In what follows, we focus on the data retrieval operations only, and do not consider the data update operations

Two remarks about the query value set are in order First, the value set of a query is in general a multi-set, meaning that the same item may occur more than once and perhaps in non-consecutive positions in the ordering Second, the query value set is always homogeneous, i e consists of records of only one type It is usually a subset of the union of one or more CODASYL set occurrences of some type The ordering of a query value set is determined, in general, by that of the navigation paths to the result records The navigation paths are considered ordered by the dictionary ordering For example the ordering of a set of paths of the form from record type A to B to C" is done as follows If $p_1 = (a_1\ b_1\ c_1)$ and $p_2 = (a_2\ b_2\ c_2)$ are two such paths, then $p_1 < p_2$ if $a_1 < a_2$, or $a_1 = a_2$ and $b_1 < b_2$, or $a_1 = a_2$, $b_1 = b_2$, and $c_1 < c_2$ Here, each of the navigation steps "A to B" and "B to C" can be from an owner to a member record type or vice-versa Note that a different query using a different navigation path to the same record type C may generate a different ordering of the result records in its value set The ordering of a query value set requires that the definition of the relational operators (join, projection, selection, etc) be augmented appropriately to reflect the proper ordering of the result relation from those of the arguments This is easily done This augmentation, however, destroys the commutative properties of the cartesian product and the join operation, but that is not of any serious concern because the general optimization techniques for relational queries can still be carried out for the cursors

3 3 The Program Conversion Algorithm

As in [9], we have two basic steps in our conversion algorithm (1) the derivation of the cursor definitions, and (2) the embedding of cursor operations among the host language statements in the target program The steps (1) and (2) are performed in that order

3 3 1 Cursor definitions

Our main concern here is the proper definition of a cursor in the presence of ambiguities in a FIND statement We define one or more cursors for each FIND statement in the source program It is sometimes possible to "cover" several FIND statements within a single cursor and thereby minimize the number of cursors [8, 9] However, we ignore such optimization issues here due to the space limitations Table 1 summarizes the cursor definitions corresponding to the categories (1)-(6) in Section 2 Here, each cursor parameter is written as (par-name par-type), where par-type is either "TID", indicating that the value of this parameter is a TID, or it is "attr-m" indicating that the value is one or more attribute values The actual values of these parameters are specified in the cursor OPEN operation, which are given in the next section When there is no confusion likely, we shall write only the par-name part In Table 1 and in what follows, rel-m is the name of the relation corresponding to the type of record selected by the statement m

The category (3) in Table 1 gives only a "skeletal" definition of the cursor because the relation name in the range-clause may be dependent on the execution context If there are k mutually incoherent ec(m)'s, due to different record types in the execution contexts, then we define k corresponding cursors Cm1, Cm2, , Cmk for the statement m Each Cmi, $1 \le i \le k$, has the range of X defined over a different relation rel-mi corresponding to different member record types in the set-m, but they are identical otherwise In order to select the appropriate Cmi for the OPEN operation at the run time, we introduce a variable Vm associated with m A statement assigning a value 'Cmi', $1 \le i \le k$, to the variable Vm is inserted following each occurrence of the ambiguous definition statement in the pc(m)

For the category (4), we have two different cursor definitions, where unlike the category (3), the number of parameters is different for the two cases The cursor Cm_a corresponds to the case where the record type in the ec(m) consists only of a owner record of set-m, i e , no member of set-m has been accessed yet The cursor Cm_b corresponds to the other case where the ec(m) also includes a member record of set-m which may be different from rec-m As before, a corresponding vari-

able Vm is introduced in this case as well

For the categories (1)-(2) and (5)-(6), we do not need different cursors definitions even when incoherent execution contexts exist This is partly due to the fact that the result record type is not context dependent The actual parameters might be, however, different for the OPEN cursor statements from the different ec(m)'s (Note that the value set of each cursor in Table 1 is indeed homogeneous)

3 3 2 Embedding of Cursor Operations

The second step of the program conversion algorithm is the embedding of the OPEN, SELECT, and FETCH operations among the host language statements in the target program In general, each FIND statement is replaced by a pair of OPEN and SELECT statement, and a GET statement is replaced by a FETCH statement

3 3 2 1 OPEN cursor operations

Table 2 gives the actual parameters of the cursor OPEN operations for each of the categories (1)-(6) Here, m1, m2, , mk refer to the ambiguous definition statements in the pc(m) In each case the case statement results in OPENing exactly one of the cursors defined for m For the category (4), the cursors Cm_a and Cm_b are those defined in Table 1 Note that the first parameter value, the owner record TID, in the various OPEN s of the cursor Cm_b are obtained from different rel-mi's corresponding to the member record types of the set-m

A similar situation arises in both the cases (5)-(6) because the owner of set-m may be reached from different statements m1, m2, , mk in the pc(m), operating on the different member record types of set-m In category (6), two different classes of actual parameters are possible though the cursor definition is unique (cf category (4)) The case Vm = 'Cm1' refers to an ec(m) where the record type is the owner of set-m, the other cases correspond to the ec(m)'s where the record type is a member of set-m The categories (2)-(6) in Table 2 have a case structure for the OPEN operation because these are the cases where the ec's might be incoherent

3 3 2 2 Embedding of OPEN-SELECT operations

In general, we replace a FIND statement by an OPEN statement as defined in Table 2 and this is immediately followed by a SELECT statement of the form shown below, which selects the same cursor Cmi that has been most recently opened for the statement m If sps(m) is unambiguous, then the case statement reduces to a simple "SELECT Cm"

```
case Vm of
   'Cm1'  SELECT Cm1,
   'Cm2'  SELECT Cm2,

   'Cmk'  SELECT Cmk,
end,
```

If the statement m belongs to a loop, then each execution of m in the target program will now cause an OPEN on one of the Cmi's This is clearly quite inefficient from the view point of the program execution speed (though in no way worse than the corresponding situation in CODASYL) There is one case where the number of times the OPEN operation is executed for the entire loop can be reduced to only one, namely, that m has an enumeration loop associated with it A FIND statement m is an *enumeration statement* if it is one of the following forms (cf categories (2)-(4)) "FIND NEXT ", "FIND PRIOR ", or "FIND DUPLICATE " [6] One of the execution contexts for such an m is always the statement {m} itself The *enumeration loop* EL(m) of m is defined to be the smallest non-empty loop if any, which contains m but not containing any statement of sps(m) other than m An EL(m) is basically a refined form of the C-diagram of the first type in [9] It is clear that the second and all subsequent executions of m in the loop EL(m) have the same ec(m) namely, {m} This suggest that if we "unfold" the first iteration of EL(m) then the OPEN operation for the remaining loop can be placed just before the loop entry node (Also, the OPEN operation can be restricted to the specific Cmi resulting from the ec(m) = {m}) However, since the unfolded copy of m can still have incoherent ec's, if the original m had incoherent ec's, the unfolding does not eliminate the problem If there were no incoherent ec's for the original statement m, then the unfolding is not needed and the OPEN operation can be placed before the loop entry This is essentially what is done in [9] In our method, we can put the OPEN operation before the loop entry, without having to unfold the first iteration, in all cases Because the variable Vm takes care of all incoherencies of m, if any The execution of the entire loop will now involve exactly one execution of the OPEN operation for m Note that the use of "pos P1 < pos X TID" in the where-clause in categories (2)-(4) in Table 1 is motivated by the above possibility of placing the OPEN statement outside the loop EL(m) Replacing that by "1 + P1 = pos X TID" would not have sufficed even otherwise because of multiple member types in a set, it would suffice for single member sets and the cursor value set would have at most one record in that case

3 3 2 3 Conversion of Other CODASYL Operators

The methods described in the previous sections can be extended to other CODASYL operations as well As an example, consider the GET statement A GET operation m refers to the current record of the run unit, and its sps(m) consists of a single statement which sets the current record of the run unit, i e , a FIND statement The pc(m) equals the set of all FIND statements m' in the program to which m is applicable We define an ec(m) to be of the form $\{m'\} \cup ec_i(m')$, where $ec_i(m')$ is an execution context for m' It is necessary to define ec(m) in this way because the particular record occurrence involved in m depends both on m' and its ec(m')'s We introduce a variable Wm to identify the particular ec(m) involved at the run time and to apply the FETCH operation to the appropriate cursor among all the cursors for the various m' To that end, we make the assignment Wm = Vm' immediately following the assignment to Vm' (see 3 2 2 1) If k incoherent semantic paths lead to a GET statement, then the GET statement is replaced by the case statement below

```
case Wm of
   'Cm1'  FETCH Cm1,
   'Cm2   FETCH Cm2,

   'Cmk'  FETCH Cmk,
end,
```

Statements such as IF RECORD NOT FOUND or IF END OF SET are replaced by a statement of the form IF END OF CURSOR Vm or IF END OF CURSOR cursor-name

Example The code in Fig 1(b) is one of the cases where the algorithms in [9 10] fail because of ambiguity

in statement 9 Fig 1(e) shows the target program obtained by our conversion method The lines in the target program are numbered here conveniently to show the correspondence with the lines in the source program The statements 4 and 7 give rise to the cursors C1 and C2 while the statement 9 gives rise to two different cursors C3 and C4 because of its incoherent execution contexts {4} and {7} The OPEN statement for m = 9 have been moved outside the enumeration loop EL(9) = {9, 10, 11, 12}

A possible solution for reducing most of the case statements for cursor operations to simple statements is described in [8] where the target interface is assumed to support the notion of CODASYL currency by updating the record TID (and any set-TID's contained in that record) whenever a SELECT operation is performed This currency simulation is not difficult to implement on the top of a relational system and it simplifies the target program considerably, which itself might be a sufficient reason to provide such support In this paper, which basically concerns the principles for attacking the conversion problem, we have presented the results assuming that the target database system is a pure relational system

4 CONCLUSION

We have introduced here the notions of abstract context, semantic predecessor set, and the execution context for analyzing the context dependency of CODASYL db operations We have used them to obtain a more refined technique for decompiling CODASYL programs that takes into account the dependency of CODASYL operations In principle, our technique can convert an arbitrary CODASYL program to one with the relational interface If the source db program contains no ambiguous db statements, then our algorithm when combined with reduces to the algorithm in [9]

The concepts of abstract context, etc can also be applied in designing tools for supporting the development of semantically correct db programs Context dependency analysis is essential in detecting improper use of db operations such as missing components of an abstract context We point out that a framework similar to the one described here can be used for analyzing the context dependency of the db operations in the hierarchical data model and also for analysis of file I/O operations of a general purpose languages

REFERENCES

[1] A Aho and J D Ullman, Principles of compiler design, Addison-Wesley Publ Co , Reading, Mass , 1979

[2] M W Blasgen, et al , System R an architectural update, IBM Research Rep RJ 2581(33481), IBM Research Laboratory, San Jose', Calif , July 1979

[3] CODASYL COBOL Journal of Development, 1976

[4] CODASYL Data Description Language Committee Report, Pergamon Press, 1978

[5] C Date, An introduction to database systems (3rd ed), Addison-Wesley Publ Co , Reading, Mass , 1981

[6] B Demo, Program analysis for conversion from a navigation to a specification database interface, *Proc IX International Conf on VLDB*, Florence, Oct 1983

[7] B Demo and S Kundu, A basic system for decompiling CODASYL DML into a relational database interface, *Proc International Computer Symposium ICS '84*, Taiwan, Dec 1984

[8] B Demo and S Kundu, A new approach to decompiling CODASYL DML into specification interface, *Tech Rep* , Dip Informatica, Univ of Torino, Italy, March 1984

[9] R H Katz and E Wong, Decompiling CODASYL DML into relational queries, *ACM Trans on Database Systems*, 7(1982), pp 1-23

[10] J Nations and S Y W Su, Some DML instruction sequences for application program analysis and conversion, *Proc SIGMOD Conf* , 1978

[11] B Shneiderman and G Thomas, An architecture for automatic relational database system conversion, *ACM Trans on Database Systems*, (1982), pp 235-257

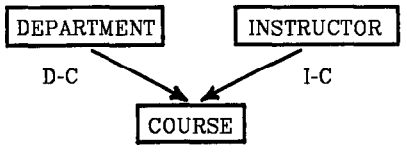
[12] S Spaccapietra et al , An approach to effective heterogeneous database cooperation, in *Distributed Data System* 7(1982), pp 1-23

[13] S Spaccapietra, B Demo, C Parent, SCOOP A System for integrating existing heterogeneous distributed data bases and application programs, *Proc of the INFOCOM Conf* San Diego, April 1983

[14] M Stonebraker, et al , The design and implementation of INGRES, *ACM Trans on Database Systems*, 1(1976), pp 189-222

[15] S Y W Su, H Lam and D H Lo, Transformation of data traversals and operations in application programs to account for semantic changes of database, *ACM Trans on Database Systems*, 6(1981), pp 255-294

[16] D C Tsichritzis and F Lochovsky, Data Models, Prentice Hall, 1982



(a) A CODASYL schema

```
 1                (read OPT)
 2        CASE    OPT OF
 3        'C'     (set COURSE key)
 4                FIND ANY COURSE
 5                IF RECORD NOT FOUND GO TO E
 6        'D'     (set DEPARTMENT key)
 7                FIND ANY DEPARTMENT
 8                IF RECORD NOT FOUND GO TO E
          END
 9    A   FIND    NEXT COURSE WITHIN D-C
10                IF RECORD NOT FOUND GO TO E
11        GET     COURSE
12                (print COURSE)
          GO TO   A
13    E   EXIT
```

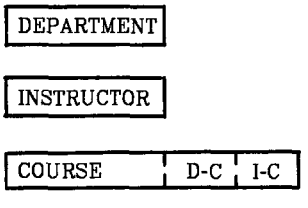
(b) A CODASYL-like code sequence It prints either (for OPT = 'C') the COURSEs which follow a given COURSE in the database ordering and are in the same DEPARTMENT or (for OPT = 'D') all the COURSEs under a given DEPARTMENT

m = 4 pc(4) = {3} = ec(4)

m = 7 pc(7) = {6} = ec(7)

m = 9 pc(9) = {4, 7, 9}
$ec_1(9) = \{4\}$, $ec_2(9) = \{7\}$, $ec_3(9) = \{9\}$
$ec_1(9)$ is coherent with $ec_3(9)$
$ec_1(9)$ is incoherent with $ec_2(9)$

(c) The program contexts and the execution contexts for the statements 4, 7, and 9 in (b)



(d) The relational schema corresponding to the CODASYL schema in (a)

Figure 1 (Contd)

```
cursor C1(P1)        {retrieve COURSE with TID = P1}
range of             X is COURSE
retrieve (X) where   (P1 = X TID)

cursor C2(P1)        {retrieve DEPARTMENT with TID = P1}
range of             Y is DEPARTMENT
retrieve (Y) where   (P1 = Y TID)

cursor C3(P1, P2)    {retrieve COURSEs following P1 and
                     under the DEPARTMENT with TID = P2}
range of             Z is COURSE
retrieve (Z) where   (P1 < pos Z TID) and
                     (P2 = pos Z D-C)
ordered by D-C

cursor C4(P1)        {retrieve COURSEs under the
                     DEPARTMENT with TID = P1}
range of             W is COURSE
retrieve (W) where   (W D-C = P1)
```

```
 1                (read OPT)
 2        CASE    OPT OF
 3        'C'     (set COURSE key)
                  OPEN C1(COURSE key)
 4                SELECT C1
                  (set variable V9 to 'C3 )
                  (set variable W11 to 'C3')
 5                IF END OF CURSOR C1 GO TO E
 6        'D'     (set DEPARTMENT key)
                  OPEN C2(DEPARTENT key)
 7                SELECT C2
                  (set variable V9 to 'C4')
                  (set variable W11 to 'C4')
 8                IF END OF CURSOR C2 GO TO E
          END
          CASE    V9 OF
          'C3'    OPEN C3(COURSE TID, COURSE D-C-TID)
          'C4'    OPEN C4(DEPARTMENT TID)
          END
 9    A   CASE    V9 OF
          'C3     SELECT C3
          'C4'    SELECT C4
          END
10                IF END OF CURSOR V9 GO TO E
11        CASE    W11 OF
          'C3     FETCH C3
          'C4'    FETCH C4
          END
12                (print COURSE)
          GO TO   A
13    E   EXIT
```

(e) The target program

Figure 1 (Contd)

TABLE 1 CURSOR DEFINITIONS

(1)	cursor	Cm(A1 attr-m)
	range of	X is rel-m
	retrieve(X) where	(A1 = X attr-m)
	ordered by	X TID
(2)	cursor	Cm(P1 TID, A2 attr-m)
	range of	X is rel-m
	retrieve(X) where	(pos P1 < pos X TID) and (A2 = X attr-m)
	ordered by	X TID
(3)	cursor	Cm(P1 TID, P2 TID, A3 attr-m)
	range of	X is
	retrieve(X) where	(pos P1 < pos X TID) and (P2 = X set-m) and (A3 = X attr-m)
	ordered by	set-m ordering criteria
(4a)	cursor	Cm_a(P1 TID)
	range of	X is rel-m
	retrieve(X) where	(P1 = X set-m)
	ordered by	set-m ordering criteria
(4b)	cursor	Cm_b(P1 TID, P2 TID)
	range of	X is rel-m
	retrieve(X) where	(pos P1 < pos X TID) and (P2 = X set-m)
	ordered by	set-m ordering criteria
(5)	cursor	Cm(P1 TID)
	range of	X is rel-m
	retrieve(X) where	(P1 = X TID)
(6)	cursor	Cm(P1 TID, A2 attr-m)
	range of	X is rel-m
	retrieve(X) where	(P1 = X set-m) and (A2 = X attr-m)
	ordered by	set-m ordering criteria

TABLE 2 OPEN CURSOR OPERATIONS

```
(1)  OPEN Cm(rel-m attr-m)

(2)  case Vm of
       'Cm1'  OPEN Cm(rel-m1 TID, rel-m1 attr-m),
       'Cm2'  OPEN Cm(rel-m2 TID, rel-m2 attr-m),

       'Cmk'  OPEN Cm(rel-mk TID, relname-mk attr-m),
       end,

(3)  case Vm of
       'Cm1'  OPEN Cm1(rel-m1 TID, rel-m1 set-m,
                  rel-m1 attr-m),
       'Cm2'  OPEN Cm2(rel-m2 TID, rel-m2 set-m,
                  rel-m2 attr-m),

       'Cmk'  OPEN Cmk(rel-mk TID, rel-m2 set-m,
                  rel-m2 attr-m),
       end,

(4)  case Vm of
       'Cm1'  OPEN Cm_a(rel-m1 TID),
       'Cm2'  OPEN Cm_b(rel-m2 set-m, rel-m2 TID),

       'Cmk'  OPEN Cm_b(rel-mk set-m, rel-mk TID),

(5)  case Vm of
       'Cm1'  OPEN Cm(rel-m1 set-m),
       'Cm2'  OPEN Cm(rel-m2 set-m),

       'Cmk'  OPEN Cm(rel-mk set-m),
       end,

(6)  case Vm of
       'Cm1'  OPEN Cm(rel-m1 TID, rel-m1 attr-m),
       'Cm2'  OPEN Cm(rel-m2 set-m, rel-m2 attr-m),

       'Cmk'  OPEN Cm(rel-mk set-m  rel-mk attr-m),
       end,
```