# 8

# Searching for semantics in COBOL legacy applications

*Martin Andersson*
*Swiss Bank Corporation*
*Hochstrasse 16, 4002 Basel, Switzerland*
*Tel/Fax: +41-61-288 3279/3198*
*E-mail: andersson.martin@ch.swissbank.com*

**Abstract**

In spite of advances in database technology, huge amounts of data around the world are still managed in files and accessed by COBOL application programs. It is a major challenge to migrate this data and these applications to modern database technology.

This work provides a basis for overcoming the mismatch between the data model of COBOL and modern object-oriented or semantic data models. We propose an algorithm to determine references and inclusion dependencies in a COBOL data structure. This information makes it possible to generate a schema in any data model, be it a semantic data model like the entity-relationship model, an object-oriented model like ODMG-93 or the relational model.

## 1 INTRODUCTION

### Statement of problem

In spite of advances in database technology, huge amounts of data around the world are still managed in files and accessed by COBOL application programs. It is a major challenge to migrate this data and these applications to modern database technology. This task is particularly important given the current rapid progress in telecommunications technology. The demands on the information retrieval systems in companies and organizations will increase as a consequence of the necessity of making information available to larger communities via the internet and intranets.

Since many COBOL systems can be characterized as *legacy systems* (Brodie and Stonebraker 1995), i.e. badly documented, brittle and monolithic, they are difficult to modify and thus expensive to maintain. Reverse engineering systems that help programmers understand the underlying data models of

Data Mining and Reverse Engineering S. Spaccapietra & F. Maryanski (Eds.)
© 1998 IFIP. Published by Chapman & Hall

legacy systems are thus of practical and economical as well as theoretical importance.

This paper addresses the problem of transforming implicitly represented information into explicit form. We concentrate on two of the most problematic features of COBOL data structures, namely (1) Implicitly modeled references between record types, and (2) Internally unspecified data fields.

## Main contributions

The techniques presented in this paper are novel, since an analysis of the source code is used to identify referential integrity constraints within the data structures. This technique is considerably more reliable than previous work in this domain, see section 7, that is limited to investigations of the data definitions rather than the source code as a whole.

Some important aspects of our approach are:

- We search the application source code for information on references between aggregates that in value-based systems, such as COBOL programs, are represented implicitly.
- The structure of unspecified record fields is determined by a comparison with the structure of other variables used to access the unspecified field.
- We propose a strategy for determining value correspondences between record fields that can be used to determine inclusion dependencies.

This work provides a basis for overcoming the mismatch between the data model of COBOL and modern object-oriented or semantic data models. The information resulting from the five analysis steps can be used in a straightforward way to generate a schema in any data model, be it a semantic data model like the entity-relationship model, an object-oriented model like ODMG-93 or the relational model.

An important application of this work is migration of old applications to modern technology. Many organizations today are dependent on information systems implemented in old-fashioned technology that cannot respond to the requirements of today. Typically, a large organization stores its mission-critical data in a very large IMS database and manipulates the data through transactions consisting of COBOL application programs. Such a system does not allow a modular or component-based approach and is therefore difficult and expensive to maintain. The successful reengineering of such systems must in all cases start with eliciting the business objects and business processes that are hidden in the application programs.

Another important application area can be found in the field of database interoperation, since a collaboration between databases is only possible if they are understood. This is made possible by reverse engineering methods. It is

generally agreed that a common representation should be used to hide the heterogeneity of the different systems. Object-oriented data models are well suited for this purpose through their concept of encapsulation.

## Overview of the method

The algorithm in this paper consists of five steps as follows. An overview is given in figure 1. Each step will be detailed in the remainder of the paper. An example is given in the appendix.

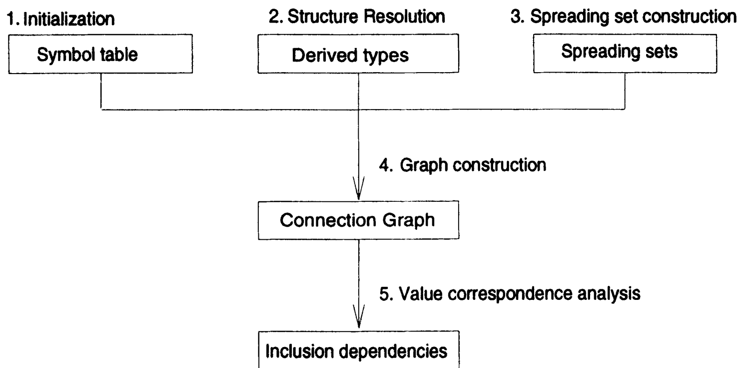



**Figure 1** Overview of the algorithm

1. **Initialization:** In the initialization phase, a symbol table describing the items in the data division of the COBOL program is created as a first-cut version of the data structures. This is done using existing compiler and parsing technology and will not be described in detail (Aho, Sethi and Ullman 1986).
2. **Structure resolution:** In the structure resolution phase, the structure of record fields that are unspecified in the data division is derived from the structure of other variables with a known structure and that are used to access the field. The detailed structure of the record types is represented by a *derived type* as described in section 2.
3. **Construction of spreading sets:** In the next phase, we use data-flow analysis to build *spreading sets* consisting of all variables through which a particular value has flown or will eventually flow. The spreading sets give information on how record fields are related. This phase is described in section 3.
4. **Graph construction:** In the graph construction phase, the information acquired in the previous steps is represented in a graph as described in section 4. The graph represents persistent data items and the relationships

between them. This is a way of *explicitly* expressing the relationships between the record types of a COBOL application.

5. **Value correspondence analysis:** In the value correspondence analysis, inclusion dependencies are derived from source code and data instances. This will be detailed in section 5.

The information resulting from the five analysis steps can be used in a straightforward way to generate a schema in any data model, be it a semantic data model like the entity-relationship model, an object-oriented model like ODMG-93 or the relational model. This paper describes how application semantics are determined from analyzing an existing system. Due to space limitations, the process of schema creation is not covered.

A general assumption of this work is that we focus on *data-centered* information systems. A typical example is administrative systems including large amounts of data and application programs, both batch and interactive. The user interface typically consists of alpha-numerical forms managed by a program that also verifies the correctness of the entered data. The output of the system is through forms or reports. The data access is thus predefined to a large extent, i.e. queries to the database are compiled and interactively written ad-hoc queries are rare. This means that it is possible to analyse statically the great majority of the data manipulation statements.

## A quick look at COBOL data definitions

The *data division* of a COBOL program defines a set of *data items* that can be elementary or structured. An *elementary* data item has a scalar type (*numeric*, *alphanumeric* or *alphabetic*) and a size. A *structured* data item has a set of *components*. The components of the i:th structured data item is a compound $(c_{i_1}, \ldots, c_{i_j}, \ldots, c_{i_n})$ of elementary or structured data items.

A *structure* is a hierarchy of components, where the root node is a structured data item (or record type) and the leaves are elementary data items. Data items defined in the *file section* of a COBOL program are associated with a file where their values can be stored. Values of data items defined in the *working storage section* are volatile. Data items in a record structure can be arrays.

It is possible to rename a set of consecutive persistent data items with the RENAMES statement. This can be done over aggregate boundaries giving overlapping structures. The statement REDEFINES is analogous to the RENAMES statement but applies to non-persistent data.

To update sequential files, *transaction files* are used. These are temporary files where operations are written that are later run in batch jobs.

## 2 STRUCTURE RESOLUTION (SR)

A serious problem is that in COBOL, variables are often declared as flat alphanumeric fields with an unspecified internal structure. This is possible since the type checking is limited to the predefined scalar data types. A structured record will be represented as a contiguous block of storage at runtime. If the record does not involve items with a special internal format to optimize storage space and numeric computations, then this contiguous block may be considered simply as a long character string with the elementary names serving as names for various substrings within the area. Thus, the true structure of the persistent record type cannot be determined by examination of the record declaration, but must instead be derived from the variables used to access the record type.

We define a *matching operation* as an operation suggesting that the values of its operands are the same. In COBOL, the matching operations are those that define variables, i.e. give them a value, and comparisons for equality and non-equality. A match also occurs in the parameter passing when sub-programs are invoked.

Figures 2.a and b show the definitions of the three record types PERSON, PERSON-INTERNAL AND OFFICE-DATA. Figure 2.c shows a set of matching operations on the record types.

### Derived types

It is possible to draw conclusions about the structure of one data item based on the structure of other data items occurring as operands in the same matching operation. To gain a complete view of the data, the different variable structures are combined into a *derived type* that includes all different interpretations of a specific data item.

A derived type is a hierarchical structure with derived elementary data items as leaves and derived structured items as nodes*. Derived data items are computed by first creating a set containing the start and end positions of all leaves. The structured items are then added which means that the structure is built bottom-up.

Figure 2.d shows a schematic representation of the record types defined in 2.a and b. The space between two vertical bars represents the extension of a component. Placing the schematic representations of a set of record types on top of each other makes it possible to visualize the spatial correspondences between the components of the record types. The schematic representation of the derived type is shown in figure 2.e. Intuitively, the derived type is constructed by pushing the smaller components downwards. The smallest components will

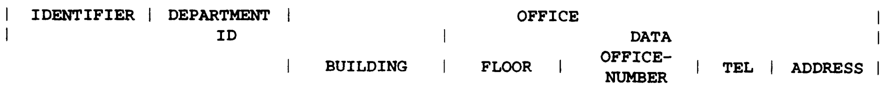
*A derived type is a structure where items on one level may share items on the level below, it is thus not a tree.

a)
```
FILE-SECTION

01 PERSON.
      05 IDENTIFIER PIC X(26).
      05 DEPARTMENT PIC X(10).
      05 OFFICE PIC X(160).
      .
      .
```

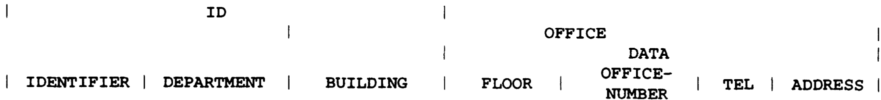
b)
```
WORKING-STORAGE-SECTION

01 PERSON-INTERNAL.
    05 ID PIC X(39).
    05 DATA PIC X(157).

01 OFFICE-DATA.
   05 BUILDING PIC X(3).
   05 FLOOR PIC X.
   05 OFFICE-NUMBER PIC X(2).
   05 TEL PIC X(4).
   05 ADDRESS PIC X(150).
```

c)
```
PROCEDURE-DIVISION
MOVE PERSON TO PERSON-INTERNAL.
MOVE OFFICE TO OFFICE-DATA.
```

d)




e)




f)

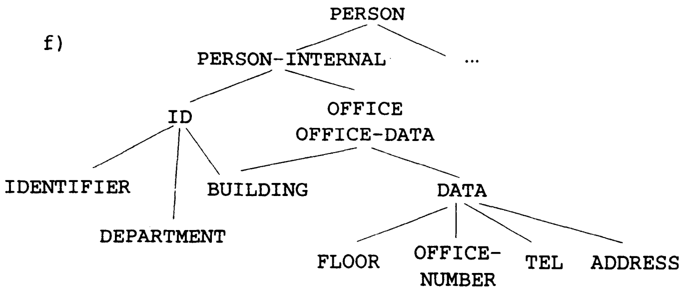



**Figure 2** Structure resolution

thus sink to the bottom and become leaves while the larger ones rest at the top and become nodes as shown in figure 2.f. Note that the generated structure is not a tree. It is quite common that multiple structure are defined on a set of leaf data items. The desired structure must be determined by a user, the structure resolution process only shows how data is structured within the legacy code.

Arrays can be used as operands in matching operations. In SR, an array is treated as a structured data item with as many components as it has array items and with all components having the same size and the same subcomponents. The array case can thus be reduced to the case of scalar data types. For arrays with a length that depends on the value of an integer vari-

able, we set the size as if the array had its maximal number of items and apply the same technique as for fixed-size arrays.

The RENAMES and REDEFINES statements in COBOL are used to declare multiple names for one single data area. From the point of view of SR, this is equivalent to defining a name that is given a structure defined elsewhere instead of in the declaration of the name. Thus, these statements do not cause any particular difficulties.

## 3 CONSTRUCTION OF SPREADING SETS

This section describes the construction of spreading sets which represent the links between variables in a program. A matching operation involving record fields indicates that the fields are related. A matching of two record fields can be done indirectly through a series of matching operations involving other variables, persistent or transient, and we must thus follow the trace from one field to another through a chain of matching operations. The matching operations group variables into equivalence classes called *spreading sets.* The identification of spreading sets is the key to establishing the relationships between the records.

However, record fields are matched differently depending on the flow of execution in the program. This means that all possible definitions and uses of a record field must be taken into consideration. This can be done using *data-flow analysis*, which is a well-known technique originally used for compiler optimization, (see e.g. (Aho et al. 1986)). The presence of a matching operation indicates that the programmer considered that the record fields *could* share a value. Every path in the program flow graph containing a matching operation thus represents a possible relationship.

We start by defining a number of useful concepts. A *point* in a sequence of statements is before or after any statement. A *use* of a variable $v$ is any occurrence of $v$ as an operand. A *definition* of $v$ is a statement where $v$ is assigned a value. A *redefinition* of a variable $v$, given a definition $d_i$ of $v$, is a definition $d_{i+1}$ of $v$ such that there is no other definition $d_j$ of $v$ between $d_i$ and $d_{i+1}$. A definition originates in a *source* that can be another variable, a persistent data item read from a file through a read operation or a value from an external interface, e.g. a user interface.

### Definition-use orders

In order to establish the flow of values in the program, we first need to determine, given the assignment of a value to a variable $x$, which other variables $y_i$ are assigned the value of $x$ before $x$ is redefined with another value. For this purpose, we define the notion of a *definition-use order* (du-order), which represents the flow of values in the program. Examples of du-orders are shown in figure 4.

To compute the du-orders, we need to decide where the variables are *defined* and where they are *used*. For this purpose a *flow-graph*, representing the flow of control in the program, is constructed using adaptations of standard data-flow analysis techniques (Aho et al. 1986).

To build the flow-graph, the program is divided into *basic blocks* with the property of having exactly one entry point and one exit point. Investigating whole blocks of statements rather than single statements makes the analysis considerably faster. The basic blocks are the nodes of the flow-graph, while the edges represent a possible flow of control from one block to another. An example of a flow-graph is shown in figure 3.

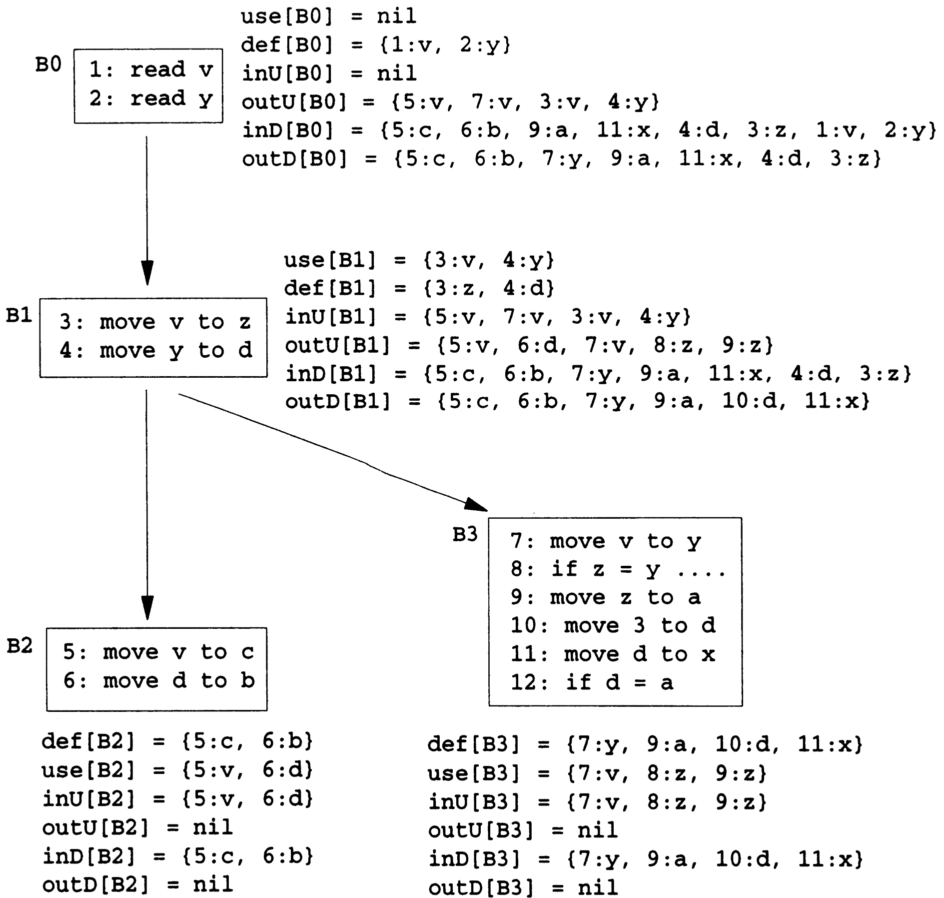



**Figure 3** Use-definition sets

For each definition of a variable *v* in the program, we are interested in the variables to which the value is propagated before *v* is redefined. To find out which these variables are, we establish six *use-definition sets* for each basic

block $B$ that make it possible to iteratively compute data-flow equations to be used in determining the value flows.

The use-definition sets are *use[B]* and *def[B]*, which contain statements where variables are used and defined within a block $B$. *outU[B]* and *inU[B]* contain statements where variables are used and are used to calculate the du-orders. The sets *outD[B]* and *inD[B]* contain definitions of variables and are used to calculate false redefinitions as will be described below. The use-definition sets are defined as follows:

- *def[B]* is the set of *defining* statements that occur in $B$. Referring to figure 3, *def[B1]* contains the two statements *3:z* and *4:d* since $z$ and $d$ are defined in $B_1$.
- *use[B]* is the set of statements $s_i$ in $B$ where

  1. a data item $x_j$ is used and
  2. there is no statement in $B$ prior to $s_i$ defining $x_j$.

  In figure 3, in block *def[B3]*, $y$ is used in statement 8 but since it is defined in statement 7, it is not included in *use[B3]*.
- *outD[B]* is the set of statements $s_i$ in blocks subsequent to $B$, where a variable $v$ is defined and such that there is no redefinition of $v$ between the end of block $B$ and any of the $s_i$. The set contains the first statement in each branch where a variable is defined seen from the end of $B$. E.g., *outD[B1]* includes the statement *10:d* which is replaced by the statement *4:d* in *outD[B0]* since $d$ is redefined in $B_1$.
- *inD[B]* is the set of statements $s_i$ in block $B$ or in blocks subsequent to $B$, where a variable $v$ is defined and such that there is no definition of $v$ between the point just before block $B$ and any of the $s_i$. The set contains the first statement where a variable is defined seen from the beginning of B. E.g. *outD[B2]* includes only the definitions made in $B_2$ since there are no definitions after $B_2$.
- *outU[B]* is the set of statements $s_i$ in blocks subsequent to $B$, where a data item is used, and such that there is no definition of that data item between the end of block $B$ and any of the $s_i$. The set contains all statements where a variable is used before it is redefined seen from the end of $B$. E.g., the statement *11:d* is not in *outU[B1]* since $d$ is defined *11:d* which is between the end of $B_1$ and *11:d*.
- *inU[B]* is the set of statements $s_i$, in or after $B$, where a variable $v$ is used and such that there is no definition of $v$ between the point just before $B$ and the $s_i$. The set contains all statements where a variable is used before it is redefined seen from the beginning of $B$.

*use[B]* and *def[B]* are constants that are constructed by an examination of

the statements in $B$. *inD[B]* and *outD[B]* are calculated with the following ***data-flow equations***:

$$inD[B] = (outD[B] -_D def[B]) \cup def[B]$$
$$outD[B] = \bigcup inD[S],\ S \text{ is an immediate successor of } B$$

The operator $< op_1 > -_D < op_2 >$ is defined on sets of statements and removes the statements from $< op_1 >$ that define the same variable as a statement in $< op_2 >$.

The first equation states that a definition of a variable at the beginning of block $B$ is either a definition that is present also after $B$ and so that the variable is not defined in $B$, or else a definition of the variable in $B$*. *inU[B]* and *outU[B]* are calculated by the following data-flow equations:

$$inU[B] = (outU[B] -_U def[B]) \cup use[B]$$
$$outU[B] = \bigcup inU[S],\ S\ is\ a\ successor\ of\ B$$

The operator $< op_1 > -_U < op_2 >$ is defined on sets of statements and removes the statements from $< op_1 >$ that use a variable defined in a statement in $< op_2 >$.

The first equation states that given a variable that is used but that has not been defined at the point before $B$, (i.e. it is used in $B$ or a block that is subsequent to $B$), there are two possibilities. Either it is undefined after $B$ and not defined within $B$ or else it is used within $B$ before it is redefined in $B$. The second equation states that the uses with no definition at the end of $B$ are the union of the uses at the beginning of its successors' blocks.

The data-flow equations can be calculated by iterative data-flow analysis that works for arbitrary flow-graphs. Algorithms for calculating the data-flow equations are well-known, see e.g. (Aho et al. 1986) for a discussion.

## Constructing du-orders

The use-definition sets make it possible to derive the flow of values in the program. Given a definition $d$ of the variable $v$ in the block $B$, $v$ is redefined in:

1. if $v$ is redefined in a statement $s$ occuring after $d$ in $B$, then $s$ is the only redefinition of $v$ or else
2. the statements in *outD[B]* where $v$ is defined.

The statements related to $d$ by propagation are:

---

*$\cup$ denotes union of sets.

1. the statements $s_i$ in which a variable $w$ is assigned the value of $v$, such that $s_i$ occurs in $B$ after $d$ but before any redefinition of $v$ and $w$ is not related to $v$ by propagation.
2. the statements in *out[B]* where a variable $w$ is assigned the value of $v$ and $w$ is not related to $v$ by propagation.

Note that we must eliminate redefinitions from variables that occur earlier in the du-order and thus have the same value as $v$. We must also check that the redefinitions are not done from a variable that is in the same spreading set*. This case is treated below.

A ***starting point*** is a definition where the source is not a variable. This can be a READ or an ACCEPT statement, or the assignment of a constant as default value. Du-orders are calculated for all starting points.

Figure 4 shows du-orders of the flow-graph in figure 3. There is one du-order with the starting point in statement 2 and another with the starting point in statement 1. Given the definition of $y$ in statement 2, $use[B_0]$ contains no statement including $y$, but $outU[B_0]$ contains a use of $y$ in statement 4 where $d$ is defined, $outU[B_1]$ contains no use of $y$ which means that it is not further used below $B_1$. Note that $y$ is actually used in statement 8, but only after it has been redefined in statement 7. The value of $y$ is propagated to $d$ which is not redefined within $B_1$. $use[B_1]$ contains no use of $d$ but $outU[B_1]$ contains a use of $d$ in statement 6. $outU[B_2]$ and $outU[B_3]$ are both empty. The du-order with starting point in statement 1 is constructed analogously.

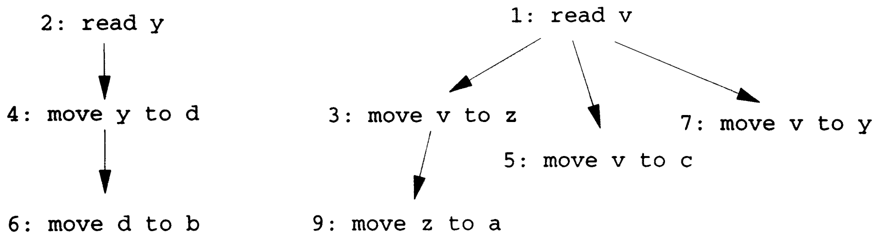



**Figure 4** Du-orders

## Spreading sets

The du-orders represent a vertical flow of values within a program. There is also a horisontal extension of a value represented by statements where variables are compared for equality. Such a statement indicates the ***possibility*** that the variables have the same value, which is all we need.

An ***eq-comparison*** is either a comparison for equality or non-equality. A du-order $ds_1$ is related to another du-order $ds_2$ by the ***sharing*** relation, denoted

*We call this a false redefinition

$\Phi$, if there exists an eq-comparison involving a variable of some statement in $ds_1$ and a variable of some statement in $ds_2$. $\Phi$ partitions the du-orders into equivalence sets $\phi_i$. A ***spreading-set*** is defined as every variable occuring in a statement of an equivalence set $\phi_i$ under $\Phi$.

To find out which du-orders are related, we must determine which variables in statements in the du-orders are matched against each other. Given a definition $d$ of the variable $v$ in the block $B$, the variables that are matched against $v$ are determined as follows:

1. variables in eq-comparisons in $B$ that include $v$ but that occur after $d$.
2. variables in eq-comparisons in outU[B] that include $v$ .

Referring to the example in figure 5, the variables $d$ and $a$ are matched in statement 11. The du-orders thus form an equivalence set $\phi$.

> A ***spreading-set*** is defined as every variable occuring in a statement of an equivalence set $\phi_i$ under $\Phi$.

The spreading set consists of every variable in any statement of $\phi$, i.e. {y, d, b, a, v, z, c}.

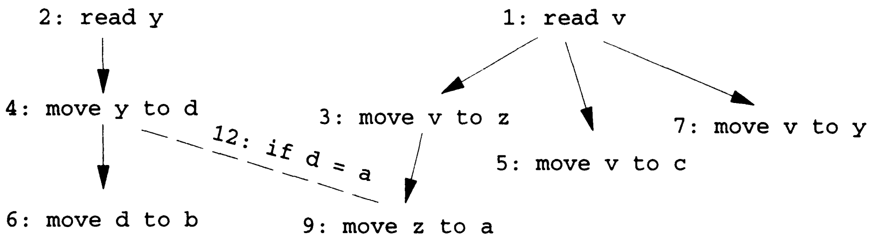



**Figure 5** Du-orders related under $\Phi$

The spreading sets are used to construct the connection graph as described in section 4.

## False redefinitions

When the spreading sets are constructed, it must be checked whether a redefinition is made from a source in the same spreading set. If this is the case the du-order must be extended with the uses up to the next redefinition.

An additional requirement is that the redefinition of a variable $v$ must not be done from a source that has the same value as $v$. This occurs when the redefinition is done from a variable in the same du-order or spreading set. The former case is treated during construction of the spreading sets and has already been discussed. The latter is trickier since we cannot know at the time

of constructing the du-orders which variables will finally end up in the same spreading set.

Consider figure 3 where in statement 7, $y$ was redefined with the value of $v$. After it has been established that the two du-orders in figure 4 are in the same spreading set, we must check if the definition of $y$ reaching statement 7 is a definition within the spreading set. If this is the case, then we know that in statement 7, $y$ is assigned the value it already has. In that case, the redefinition is invalid and we must recalculate the du-order based on the next redefinition of $y$. This can be done as before by identifying the statements where $y$ gives its value to another variable after statement 7 but before $y$ is redefined.

As there is no definition of $y$ after statement 8, we must include all uses of $y$ until the end of block $B_3$ into the du-order of statement 2. The only use of $y$ is in statement 8 where it is compared to $z$ that is already in the spreading set. The du-order therefore does not change.

This process must be repeated until no redefinition is done from a source within the same spreading set. Given a definition $d$ of $x$ in B, the redefinitions of $d$ can be determined as:

1. If there is a definition $d'$ of $x$ in B after $d$ then $d'$ is the only redefinition, or else
2. The definitions of $x$ in *outD[B]*.

For each redefinition $R$, we must check whether in $R$ a variable $y$ is used that has been defined in a statement $s$ that occurs in the same spreading set and such that $y$ has not been defined since it was defined in $s$. To calculate the latest definition of a variable, we can use a standard technique from data-flow analysis called *reaching definitions*. The calculation of reaching definitions is analogous to the calculation of definition-use sets described above, with the difference that reaching definitions are calculated top-down instead of bottom-up. We do not describe the details of this computation, the complete procedure can be found in (Aho et al. 1986). Given that we have found a false redefinition, we recalculate the du-orders and the spreading sets as before until they include no false redefinitions.

## Arrays

A match involving an array is a different case since the array index cannot be determined statically. There are three cases when it is possible to include arrays into the calculation of spreading sets.

1. If it can be shown that all items in the array come from the same source.
2. If it can be concluded that the index value has not changed between a definition and a use of an array item.

3. If it can be concluded that the index value when the item is used is within the interval of index values when the array items were defined.
   It is necessary to eliminate transitive redefinitions using variables that occur higher up in the du-order and thus have the same value as $v$. However, this is not enough since we must also check that the redefinitions are not done from a variable that is in the same spreading set.

## 4 CONNECTION GRAPH CONSTRUCTION

The fourth step in the algorithm is the construction of a connection graph representing the data structure of the COBOL application. The connection graph includes information from the symbol table, the derived types and the spreading sets. Vertices in the connection graph are persistent independent data items, i.e. record types defined in the file section. A vertex is detailed according to the derived type associated with the data item that it represents. An edge is defined between any two variables or data items that occur in the same spreading set. The connection graph is constructed as follows:

1. Each persistent record type $A_i$ defines a vertex $V_i$ in the graph.
2. The structure of $V_i$ is created top-down. Starting from the top, each data item $d$ of $A_i$ that has a derived type $t$, is replaced by $t$ together with all its components. If $d$ has no derived type, it is used in the structure of $V_i$.
3. An edge is defined between any two data items that are members of the same spreading set. These data items are called *anchors*.
4. When a record type includes one single field, an edge can be defined on the whole record type. If this is the case, then the vertex representing the record type is merged with the vertex at the other end of the edge. Two record types connected by an edge are thus merged into one single vertex and a record type that is linked to a field of another record type is merged with that field.

Transaction file records are not merged with the record type it is used to update, even if it is persistent. The reason is that the derived type of the transaction file record and the data file record will not be the same due to the additional meta attributes in the transaction file record that specify the type of operation to be done on the data file record. The transaction file records are excluded from the connection graph even though they are persistent since they do not contribute any new information from the domain of discourse.

## 5 VALUE CORRESPONDENCE ANALYSIS

In order to understand relationships between record types, it is valuable to have information on the correspondences between the values of the anchors.

An important example is inclusion dependencies, which correspond to referential integrity constraints in object-oriented data models.

Referential integrity can be enforced by external key specifications. However, in older systems it is mostly maintained in the application code, i.e. a referencing data item is always assigned the value of an existing record.

An inclusion dependency $A \preceq B$ is established when it can be determined that values inserted into $A$ always come from the set $B$. This can be traced in assignment patterns in the source code by means of spreading sets.

## Source code analysis

It is sometimes possible to determine inclusion dependencies by analyzing patterns of assignments. Recall the definition of spreading-sets in section 3 as the set of variables that share some value. Let $X$ and $Y$ be components of persistent data items and let an *access* of $X$ be a read or a write of $X$.

- If $X$ is accessed in every spreading set where $Y$ is written, then $Y \preceq X$.
- If $X$ is accessed in every spreading set where $Y$ is written and $Y$ is accessed in every spreading set where $X$ is written, then $X = Y$.

Note that in the first point it is enough that the constraint holds where $Y$ is written. $Y$ may occur in a spreading set where $X$ does not occur as long as $Y$ is not written. Using the spreading sets, we can detect cases when there are equal values for two anchors. It is not possible to determine whether values are different, so we cannot detect disjointness. The reason is that we cannot exlude that two different spreading sets could contain the same value. For the same reason, it is impossible to detect a proper subset, since we do not know whether one anchor has values that another has not.

## Data analysis

Information on inclusion dependencies can also be obtained by investigating the data. We determine the correspondences between the values of anchors. The anchors are attributes that are used to relate aggregates and can thus be used to infer referential integrity constraints between aggregates. We therefore focus the data analysis on the anchors. The sets of values of two anchors can be disjoint, have a non-empty intersection, have a subset relationship or be equal.

To calculate these correspondences, the record types are sorted with respect to the investigated attributes. The algorithm that checks for inclusion dependency works on two files simultaneously. One block from each file is read and the values are compared. Given that the files are sorted, it is enough to

traverse each file once. This is done for each pair of anchors. The total cost is:

$$edgeNumb * ((B_{A_i} log B_{A_i} + B_{A_i}) + (B_{A_j} log B_{A_j} + B_{A_j}))$$

where *edgeNumb* is the total number of edges in the connection graph. The cost for one anchor is the sum of the cost of the sort and of the traversal of the file. Our algorithm calculates inclusion dependencies for large connection graphs in reasonable time.

# 6 IDENTIFYING FUNCTIONAL DEPENDENCIES

This section describes how functional dependencies are derived. The data structure represented in the connection graph may include redundancies introduced for efficiency purposes e.g. in order to reduce query response time. The transformation of a data definition for efficiency purposes is called *denormalization.* In order to remove these redundancies, we normalize the denormalized schema based on functional dependencies.

A functional dependency (FD) is a constraint in terms of specific values of instances and it is thus necessary to know the actual values in order to decide if the constraint is satisfied or not. Since the values are not explicitly mentioned in the source code except for constants, a functional dependency can rarely be traced in the source code.

Another source of information is the data. By identifying sources and targets of functional dependencies, we can determine functional dependencies in an analysis of the data. Once the functional dependencies are determined, redundancies can be removed through decomposition with a standard algorithm.

The problem with this approach is that an exhaustive search is unfeasible for real–life–sized schemas. The number of possible combinations of sources and targets is too large. Our approach is to restrict the number of possible sources of functional dependencies to bring down the set of possible combinations. To achieve this, it is tempting to use the information in the connection graph. Attributes on which edges are defined are often keys, i.e. sources of functional dependencies. However, one of the most important reasons to denormalize a database is to avoid costly join operations. A denormalized aggregate is a materialized join and a query on such an aggregate can be limited to a selection of the desired instances. Therefore, in a denormalized database, join-operations are likely to be scarce and the strategy to use eq-comparisons to determine significant attributes may well be a shot in the dark.

Our strategy to find functional dependencies is based on the assumption that a data definition models a set of *object types* in the domain of discourse of the application. We say that a set of functional dependencies $F$ is a *minimal cover* if:

1. For all $(X \rightarrow A) \in F$ is $A$ a single attribute.
2. For no $(X \rightarrow A) \in F$ is the set $F - \{X \rightarrow A\}$ equivalent to $F$.
3. For no $(X \rightarrow A) \in F$ and proper subset $Z$ of $X$ is the set $F - \{X \rightarrow A\} \cup \{Z \rightarrow A\}$ equivalent to $F$.

We define an object type as:

Let $F_i$ be a minimal cover of FDs such that, taken pairwise, there is a bijective mapping between the targets of the FDs. An *object type* is a set consisting of the attributes on which these FDs are defined.

The sources of the FDs are called *identifiers* of the object type, the targets of the functional dependencies are called *properties*.

## Identification of object type identifiers

Access of a single instance of an object type is done through its identifier. This is true regardless of whether the object type identifier is also the identifier of the aggregate representing the object type. Figures 6.a through d show a set of operations used to manipulate the record MACHINE-RESP.

```
MACHINE-RESP
    MACHINE-NAME
    MACHINE-UNIT
    IP-ADDRESS
    RESP-ID
    RESP-UNIT
    RESP-TEL
```

```
MACHINE-NAME  -->  MACHINE-UNIT, IP-ADDRESS,
RESP-ID  --  RESP-UNIT, RESP-TEL
MACHINE-NAME, RESP-ID  --> Ø
```

```
a) delete
   from MACHINE-RESP
   where MACHINE-NAME = :MACHINE
```

```
b) delete
   from MACHINE-RESP
   where RESP-ID = :PERSON-ID
```

```
c) delete
   from MACHINE-RESP
   where RESP-ID = :PERSON-ID
   and MACHINE-NAME = :MACHINE
```

```
d) select machineName
   from MACHINE-RESP
   where RESP-ID = :PERSON-ID
```

**Figure 6** Aggregate in 1NF

These operations are expressed in SQL, but the principle is the same in a COBOL program, the only difference is that in COBOL more code is required to do the same thing. These are basic operations necessary to maintain the data of MACHINE-RESP. This type of operation uses object type identifiers as selection criteria. A typical operation would be to find out which machines a

given person takes care of, like in the query in figure 6.d. The search condition indicates that the identifier of the object type is *Person*. A ***data manipulation operation*** is an operation that is used to maintain the data. Examples of data manipulation operations are:

- *Insertions* of instances in aggregates representing relationships with replicated data, include existence verifications on the related object types. For example, to insert an instance of MACHINE-RESP, the existence of the machine and the manager must first be verified. This can be detected in the code and gives information on the object type identifiers.
- *Deletions* of a single instance is done using the identifier of the instance.
- *Modifications* designate the instance to modify through an object type identifier. Additionally, modifications are often done on single instances.
- *Selections* on object types based on the identifier.

We can define the set of possible sources of a functional dependency. Let $\mathcal{L}_i$ be the set of anchors defined on $A_i$, where $A_i$ is a vertex in the connection graph. Let $\mathcal{C}_i$ be the set of sets of attributes where the attributes in each set are used as identifiers in data manipulation operations and let $\mathcal{K}_i$ be the set of possible keys derived from various sources, e.g. index definitions. The set of possible sources of the functional dependency $\mathcal{S}_i$ is defined as:

$$\mathcal{S}_i = \mathcal{L}_i \cup \mathcal{C}_i \cup \mathcal{K}_i \ .$$

Note that $\mathcal{S}_i$ consists of sets of sets of attributes.

## Computing targets of FDs

Given that the possible sources of FDs can be determined as defined above, we proceed to verify whether these sources actually *are* sources of FD and if so which attributes are their targets. We are interested in elementary FD, and it is therefore enough to check single attributes of an aggregate for being the target of a given source of FD. For every $X \in \mathcal{L}_i \cup \mathcal{C}_i$, $X$ is the source of the functional dependency $X \to Y$ if

1. there is an attribute $Y$ such that $X - Y = X$ and
2. $X \to Y$ satisfies the definition of an FD.

There may be aggregates including only possible sources, e.g. an aggregate that is used to model a many-to-many relationship and that includes only the identifiers of the object types in the relationship. In such an aggregate, only the identifier is a source of FD. For each $X \in \mathcal{K}$, $X \to Y$ holds for every attribute in $A_i$. If there is no such $Y$ in $A_i$, then $X \to \emptyset$ holds.

Let $A_i$ be an aggregate occupying $B_{A_i}$ number of blocks. Let $\mathcal{S}_i$ be the set of possible FD sources in $A_i$. Let $\#\mathcal{S}_i$ be the number of elements in $\mathcal{S}_i$. For each

source $X \in \mathcal{S}_i$, and for each attribute $A \in A_i$, we check that whenever $t_1[X] = t_2[X]$, then also $t_1[A] = t_2[A]$. If this is not the case, $A$ is not functionally determined by $X$.

We assume that the number of blocks $M$ that fit into main memory is smaller than the number of blocks occupied by $A_i$, i.e. $B_{A_i} > M$. Since we are interested in all instances of $A_i$ that agree on $X$, it is advantageous to group the instances based on $X$. $A_i$ is therefore sorted with $X$ as the sort key. The cost of the sort is $B_{A_i} log B_{A_i}$. Given that $A_i$ is sorted according to $X$, the verification of the constraint can be done in one single scan of $A_i$, i.e. in $B_{A_i}$ disk accesses. The total cost of verifying one possible source is thus, $B_{A_i} log B_{A_i} + B_{A_i}$ and the cost of checking all elements of $\mathcal{S}$ is:

$$\#\mathcal{S} * (B_{A_i} log B_{A_i} + B_{A_i})$$

The cost is thus proportional to the number of elements in $\mathcal{S}_i$ and logarithmic to the size of $A_i$. To calculate the FDs of the hundred attribute aggregate, we assume that there is a search condition on 10% of the attributes, i.e. $\#\mathcal{S} = 10$. Further, we assume that $B_{A_i}$ is 30000, about 15 Mbyte. If the time to read a block is 20 msec, the calculation will be finished in under 10 hours. Clearly, this example is based on assumptions, but it still gives a picture of the magnitude of the time needed to compute functional dependencies.

## 7 RELATED WORK

Related work in this domain mainly falls into three categories as follows:

- **Translation from relational to entity-relationship models:**
  A number of approaches exist for translating from the relational to the entity-relationship model, e.g. (Dumpala and Arora 1981), (Navathe and Awong 1987), (Casanova and de Sa 1983), (Markowitz and Makowsky 1990), (Johannesson 1994), (Davis and Arora 1988), (Shoval and Schreiber 1993), (Fonkam and Gray 1992), (Premerlani and Blaha 1993), (Chiang, Barron and Storey 1994).
  These approaches have in common that they assume relational schemas in third normal form and forehand prior knowledge on inclusion dependencies. For older systems, this is an unrealistic assumption since the schemas may well contain redundancies for optimization purposes and since external key specifications are not always supported. Our work is a necessary prerequisite for this type of translation.
- **Information acquisition:**
  A number of recent approaches address the problem of *information acquisition.* For example in (Castellanos 1993), a method is presented for deriving functional and inclusion dependencies from data and for constructing an object-oriented schema. Other methods, focusing on analysing

application source code, are presented in e.g. (Signori, Loffredo, Gregori and Cima 1994), (Petit, Boulicaut, Toumani and Kouloumdjian 1996), (Hainaut, Englebert, Henrard, Hick and Roland 1995). The few existing proposals for extracting an entity relationship schema from the data structures of a COBOL application are mostly limited to searching information in the COBOL data structure (Davis and Arora 1986), (Nilsson 1985).

- **General-purpose tools:**
  In the software engineering community, work has been done on general-purpose tools for program analysis. These include techniques such as e.g. program slicing in (Henrard, Hick, Roland, Englebert and Hainaut 1996) and (Weiser 1984) and variable slicing in (Chen, Tsai, Joiner, Gandamaneni and Sun 1994) and (Joiner, Tsai, Chen, Subramanian, Sun and Gandamaneni 1994). However, these systems do not propose techniques for identifying integrity constraints. A typical example is the method proposed in the REDO project (van Zuylen 1993) for constructing an extended entity relationship schema based on the data definitions of a COBOL application (Sabanis and Stevenson 1992) that does not exploit the powerful source code analysis tools elaborated within other parts of the project.

A more detailed survey of these approaches is given in (Andersson 1995).

## 8 CONCLUSION

We have presented a method to obtain information about the data structure of a COBOL program. We have identified:

- The structure of variables for which no detailed declaration exists in the source code.
- References representing various types of relationships between object types modelled in the data structure.
- Functional dependencies between attributes of the COBOL data structure.

This information can be used to create a schema representing the data structure of the COBOL program in any model, be it a semantic data model or the object-oriented model.

The method has only been partly implemented and it has not yet been possible to test it on a real case. However, a preliminary study has been made on the applicability of the method to a real system used to administrate the students at a university. Here, data was represented with a relational-like data model and with applications written in a proprietary 4GL ressembling COBOL. In the study, it was found that it is possible to find the relationships between aggregates by analysing the program code.

## REFERENCES

Aho, A., Sethi, R. and Ullman, J.: 1986, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, Boston, MA.

Andersson, M.: 1995, Extracting an entity relationship schema from a relational database through reverse engineering, *International Journal of Cooperative Information Systems* **4**(2).

Brodie, M. L. and Stonebraker, M.: 1995, *Migrating Legacy Systems, Gateways, Interfaces & The Incremental Approach*, Morgan Kaufmann publishers, San Francisco.

Casanova, M. A. and de Sa, J. E. A.: 1983, Designing entity-relationship schemas for conventional information systems, *Third Int. ER Conf* .

Castellanos, M.: 1993, A methodology for semantically enriching interoperable databases, *Proceedings of British National Conference on Data Engineering, UK.*

Chen, X. P., Tsai, W. T., Joiner, J. K., Gandamaneni, H. and Sun, J.: 1994, Automatic variable classification for COBOL programs, *Proc. of the Int. Conf. Computer Software and Applications, Taipei, Taiwan.*

Chiang, R., Barron, T. and Storey, V.: 1994, Reverse engineering of relational databases: Extraction of an EER model from a relational database, *Data & Knowledge Engineering* **10**(12).

Davis, H. K. and Arora, A. K.: 1988, Converting a relational database model into an entity-relationship model, *7th International Conference on the Entity Relationship Approach* .

Davis, H. K. and Arora, K. A.: 1986, A methodology for translating a conventional file system into an entity relationship model, *in* P. P. Chen (ed.), *Entity Relationship Approach: the Use of ER Concepts in Knowledge Representation*, IEEE CS Press, North Holland.

Dumpala, S. R. and Arora, S. K.: 1981, Schema translation using the entity relationship approach, *Entity Relationship Approach to Information Modeling and Analysis.*, .

Fonkam, M. M. and Gray, W. A.: 1992, An approach to eliciting the semantics of relational databases, *4th International Conference on Computer Aided Software Engineering (CAiSE)* .

Hainaut, J.-L., Englebert, V., Henrard, J., Hick, J.-M. and Roland, D.: 1995, Requirements for information system reverse engineering support, *Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada.*

Henrard, J., Hick, J.-M., Roland, D., Englebert, V. and Hainaut, J.-L.: 1996, Technique d'analyse de programmes pour la rétro-ingénierie de bases de données, *INFORSID* .

Johannesson, P.: 1994, A method for translating relational schemas into conceptual schemas, *Tenth International Conference on Data Engineering*

Joiner, J. K., Tsai, W. T., Chen, X. P., Subramanian, S., Sun, J. and Gandamaneni, H.: 1994, Data-centered program understanding, *Proceedings of the Int. Conf. on Software Maintenance, Victoria, British Columbia, Canada, September 19-23.*

Markowitz, V. M. and Makowsky, J. A.: 1990, Identifying extended entity-relationship object structures in relational schemas, *IEEE Transactions on Software Engineering* **16**(8).

Navathe, S. and Awong, A.: 1987, Abstracting relational and hierarchical data with a semantic data model, *6th International Conference on the Entity Relationship Approach* .

Nilsson, E. G.: 1985, The translation of a COBOL data structure to an entity-relationship type schema, *4th International Conference on the Entity Relationship Approach* .

Petit, J.-M., Boulicaut, J.-F., Toumani, F. and Kouloumdjian, J.: 1996, Towards the reverse engineering of denormalized relational databases, *12th IEEE Int. Conf. on Data Engineering* .

Premerlani, W. J. and Blaha, M. R.: 1993, An approach for reverse engineering of relational databases, *IEEE Working Conference on Reverse Engineering* .

Sabanis, N. and Stevenson, N.: 1992, Tools and techniques for data remodelling COBOL applications, *Proceedings of the 5th Int. Conf. on Software Engineering and its Applications, Toulouse.*

Shoval, P. and Schreiber, N.: 1993, Database reverse engineering: From the relational to the binary relational model, *Data and Knowledge Engineering* .

Signori, O., Loffredo, M., Gregori, M. and Cima, M.: 1994, Reconstructions of ER schema from database applications: a cognitive approach, *in* P. Loucopoulos (ed.), *Proceedings of the 13th Int. Conf. on the Entity-Relationship Approach, ER'94, Manchester, UK.*

van Zuylen, H. (ed.): 1993, *The REDO Compendium, Reverse Engineering for Software Maintenance*, John Wiley and Sons.

Weiser, M.: 1984, Program slicing, *IEEE Transactions on Software Engineering* **10**(4).

## BIOGRAPHY

Martin Andersson received a Master of Science degree in 1987 from the Linköping Institute of Technology in Sweden. He spent two years at Telelogic AB, Sweden, before joining Asea Brown Boveri AG, Switzerland, in 1989. In 1991, he joined the database laboratory of the Swiss Federal Institute of Technology in Lausanne where he received a Ph.D. in 1996. As of 1997, Martin Andersson holds a position at the Swiss Bank Corporation in Basel.