Automatic High-Quality Reengineering of Database Programs by Abstraction, Transformation and Reimplementation

YOSSI COHEN
Tel Aviv University
and
YISHAI A. FELDMAN
The Interdisciplinary Center, Herzliya

Old-generation database models, such as the indexed-sequential, hierarchical, or network models, provide record-level access to their data, with all application logic residing in the hosting program. In contrast, relational databases can perform complex operations, such as filter, aggregation, and join, on multiple records without an external specification of the record-access logic. Programs written for relational databases attempt to move as much of the application logic as possible into the database, in order to make the most of the optimizations performed internally by the database.

This conceptual gap between the programming styles makes automatic high-quality translation of programs written for the older database models to the relational model difficult. It is not enough to convert just the database-access operations, since this would result in unacceptably inefficient programs. It is necessary to convert parts of the application logic from the procedural style of the hosting program (which is almost always Cobol) to the declarative style of SQL.

This article describes an automatic system, called MIDAS, that performs high-quality reengineering of legacy database programs in this way. MIDAS is based on the paradigm of translation by abstraction, transformation, and reimplementation. The abstract representation is based on the Plan Calculus, with the addition of Query Graphs, introduced in this article in order to abstract the temporal behavior of database access patterns.

The results of MIDAS's translation were found to be superior to those of the naive translation that only converts database-access operations in terms of readability, size of code, speed, and network data traffic. Initial industrial experience with MIDAS also demonstrates the high quality of its translations on large-scale programs.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*; F.3.2 [**Logics and**

This is a revised and expanded version of a paper presented at the Twelfth Automated Software Engineering Conference, Lake Tahoe, Nevada, November, 1997.

Authors' addresses: Y. Cohen, School of Computer Science, Tel Aviv University, 69978 Tel Aviv, Israel; email: uniface@cs.tau.ac.il, Web: www.cs.tau.ac.il/~uniface; Y. A. Feldman, Efi Arazi School of Computer Science, The Interdisciplinary Center, P.O.B. 167, 46150 Herzliya, Israel; email: yishai@idc.ac.il; Web: http://www.idc.ac.il/yishai.

Meanings of Programs]: Semantics of Programming Languages—*program analysis*; H.2.5 [**Database Management**]: Heterogeneous Databases—*program translation*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*automatic analysis of algorithms*; *program transformation*

Additional Key Words and Phrases: Database program reengineering, the plan calculus, query graphs, temporal abstraction

1. INTRODUCTION

Many legacy software applications still in use today are based on old-generation database models, such as indexed-sequential, hierarchical, and network databases. A major goal in reengineering such applications is to upgrade to a modern database technology, which in most cases means relational databases. Typically, the original program is written in some high-level language (almost always in Cobol) with embedded database access commands in the form of subroutine calls or directives to a special preprocessor. The goal of the translation process is to produce a semantically-equivalent program in the same host language but using some form of embedded SQL for the database access. (Other options, such as translating to a different host language or application generator, or changing to object-oriented databases, are discussed in the conclusion, Section 4.2.)

In a survey made in 1994 by IBM's Center for Advanced Studies [Buss et al. 1994], it was found that Fortune 100 companies use an average of 35 millions of lines of code each. This number, they report, increases by 10% every year. Manual reengineering of such applications is an error-prone and costly process, in time and money. Thus automatic translation is desirable.

More recently (2000), the company that implemented this research in a commercial setting (see Section 3.2) tried to estimate the size of the mainframe pre-relational database market. Gartner estimated that there were about 15,000 installations of pre-relational databases. A telemarketing survey among ADABAS and IDMS customers indicated that about 50% would consider reengineering to a relational database within seven years. Based on these numbers, the overall market size was estimated to be around $6.5 billion dollars.

The database-software translation problem consists of three main tasks: schema translation, data migration, and code translation. The first two have received a lot of attention in the literature [Fong 1992; Fong and Bloor 1994; Gillenson 1990; Spooner et al. 1989; Tangorra and Chiarolla 1995; Winans and Davis 1991]. This article deals with the third task. We will assume in this article that the source program is written for a network database, since the network model is the most complex of the older models. The methods that we present can be applied to the other legacy database models as well.

1.1 The Translation Challenge

Any process that translates network database applications to the relational model needs to bridge several gaps that arise from the inherent differences between the two models.

Relationships between database records in network databases are fixed by the database schema. A *link* may be defined between two record types, defining a one-to-many relationship between them. Links are implemented by physical pointers in the respective records. In contrast, a relational database is made of a collection of separate tables, and any two tables can be joined at will, by ad-hoc comparisons of fields in one table with fields in another.

A network database program specifies procedurally how to navigate the database links in order to find the data it needs. In contrast, a relational database program expresses the data it needs declaratively, using operators such as filters, maps, joins, and aggregations. As a result, relational databases have considerably greater expressive power than network databases, and programs that use SQL effectively can be much simpler than their network-database equivalents. However, the lack of pointers means that data-access in a relational database is less efficient, and relational databases compensate using indexes and sophisticated optimization algorithms. These optimizations are most effective on complex queries, which use the expressive power of the relational query language.

In principle, it is possible to translate only the database-access commands of the original program with only minor effects on the rest of the program. This implies that each database operation in the original program is mapped to an equivalent operation in the new database. This one-to-one translation is not easy for a variety of reasons. For example, the network database contains contextual information in its pointers; this information is implicit in the relational database, and needs to be recovered from the original program in order to provide the correct context in the corresponding SQL statements.

The one-to-one translation keeps the application logic in the hosting program. It therefore suffers from the deficiencies of both database models. On the one hand, it loses the inherent efficiency of the pointer-following network database. On the other hand, it does not take advantage of the optimizations possible in the relational database.

In contrast, human programmers do not translate programs in this way. Instead, they analyze the original program to find patterns of database accesses, such as filtering, joins, and aggregative operations. The patterns discovered are then used to “fold” computation from the host language program into the embedded SQL commands. In this way more of the work is performed by the database engine, allowing it to perform effective optimizations as well as reducing the amount of data exchanged between the database and the host program. This can lead to a dramatic reduction in the amount of network traffic in client–server or WEB-based configurations.

1.2 Overview

We have developed an automatic translation system, called MIDAS,[1] that performs the kind of analysis and translation described above. It uses temporal abstraction techniques [Waters 1978, 1979] to discover database access patterns in the host program and translate them to relational-database operations

[1]For MIgrator of Database Application Systems.

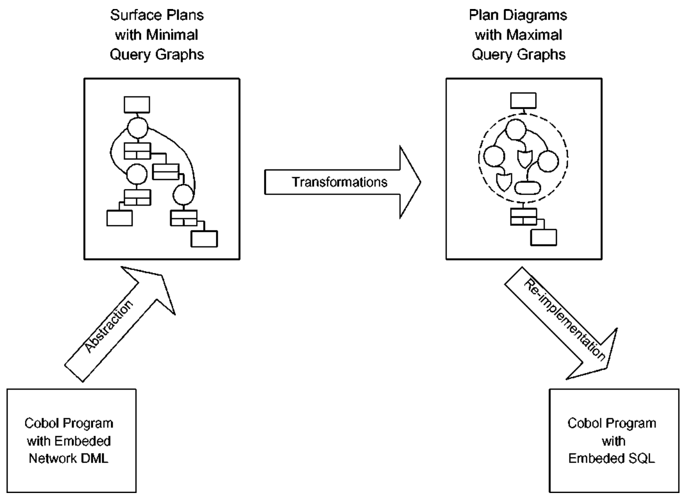
Surface Plans with Minimal Query Graphs
Transformations
Plan Diagrams with Maximal Query Graphs
Abstraction
Re-implementation
Cobol Program with Embeded Network DML
Cobol Program with Embeded SQL

Fig. 1. Translation by abstraction, transformation, and reimplementation.

whenever possible. This process tries to mimic the analysis that would be done by a human programmer manually translating the program.

MIDAS uses the paradigm of translation by abstraction, transformation, and reimplementation (see Figure 1), first described by Waters [1988], and later extended and validated in a large-scale experiment by Feldman and Friedman [1999]. In this scheme, the original program is first analyzed into a more abstract representation, which embodies the semantics of the program in a canonical way, while abstracting away from syntactic coding variations. Transformations are then applied to this abstract representation, in order to get it as close as possible to the semantics of the target language. Finally, the abstract representation of the program is reimplemented in the target language. Typically, this is the simplest part of the process.

In the case of MIDAS, the abstract representation is based on the Plan Calculus [Rich 1981; Rich and Waters 1990], extended with a formalism called *query graphs* for describing database operations. Query graphs are introduced in this article in order to express the temporal abstraction of database access patterns.

During the transformation stage, MIDAS searches the program for database access patterns (also called *idioms*) such as joins, filtering, arithmetic calculations, and aggregations. These idioms are represented as plan diagrams [Rich 1981; Rich and Waters 1990] with embedded query graphs. MIDAS uses a small and fixed set of idiom transformations, which suffice to capture most of the common database-access idioms that appear in legacy programs. Each transformation allows MIDAS to move some of the application logic from the plan diagram, representing the original (Cobol) program, into the query graphs, which will eventually be reimplemented as SQL queries.

The process of abstraction and transformation is independent of the destination relational schema and of the way in which network database objects are mapped to relational database objects. Only the reimplementation stage needs to be aware of the destination schema. MIDAS's reimplementation stage supports a large set of mapping rules (called *schema transformations*) between network database objects (records, sets, fields, etc.) and relational database objects (tables, fields, keys, etc.). The reimplementation stage accepts any destination schema that can be expressed by these schema transformations.

The techniques described in this article are adequate for hierarchical as well as indexed-sequential databases with only minor changes. Most of these changes are expected to influence the parsing process only.

1.3 An Example Program

To illustrate the possible translation approaches, we will use a program written for a typical network database. The example is based on a hypothetical university database containing student information. The program[2] in Figure 2 computes and displays the average grades of the graduate students whose grade average is over 95%, in a given department.

The first part of the program (the paragraph labeled `MAIN`) receives from the user the name of a department, whose students are to be listed. The school to which the department belongs is retrieved by the `FETCH OWNER` command (line 8), for use in the title of the output.

The second part of the program (the paragraph labeled `ALL-DEPT-STUDENTS`) fetches all student records belonging to the given department. Those records not belonging to graduate students are ignored; the rest are processed further. The third paragraph (labeled `SUM-STUDENT-GRADES`) is called repeatedly in order to fetch all the course grades of each graduate student, count them in the variable `GRADES-COUNT`, and sum them into `GRADES-SUM`. (The variables are initialized in the calling paragraph.) Finally, if the average is greater than 95%, the student's details and statistics are displayed.

Only the department record is fetched using an index. The school, the students, and their grades are fetched using the pre-existing database links `DEPT-OF-SCHOOL`, `DEPT-OF-STUDENT`, and `STUDENT-OF-GRADES`. Therefore, the example program is very efficient in the network model. However, it could be made more efficient given a more expressive database model. For example, this program retrieves many records (those representing undergraduates) only to discover immediately that it does not need them. A relational database can filter those records as part of the processing of the query in the server, so that the program never receives them.

1.4 Naive Translation

The result of the naive one-to-one translation is shown in Figures 3–4. The translation maps each database-access command in the network program to a single database-access command in the relational program, adding

[2]Only the procedure division of the program is included in the example. The other divisions are omitted for the sake of brevity.

```
1. MAIN.

2.  DISPLAY "ENTER DEPARTMENT NAME:"
3.  ACCEPT DEPT-NAME
4.  MOVE 0 TO STATUS0
5.  FETCH DEPARTMENT USING DEPT-NAME
6.    AT END MOVE 1 TO STATUS0
7.  IF STATUS0 IS EQUAL TO 0 THEN
8.    FETCH OWNER WITHIN DEPT-OF-SCHOOL
9.    DISPLAY "LIST OF BEST GRADUATE STUDENTS OF " DEPT-NAME
10.            " IN " SCHOOL-NAME
11.    PERFORM ALL-DEPT-STUDENTS
12.  END-IF.
13.  STOP RUN.

14. ALL-DEPT-STUDENTS.

15.  MOVE 0 TO STATUS1
16.  PERFORM UNTIL STATUS1 IS NOT EQUAL TO ZERO
17.    FETCH NEXT STUDENT WITHIN DEPT-OF-STUDENT
18.      AT END MOVE 1 TO STATUS1
19.    IF STATUS1 IS EQUAL TO 0 THEN
20.      IF STUDENT-DEGREE IS EQUAL TO 2 THEN
21.        MOVE 0 TO GRADES-SUM
22.        MOVE 0 TO GRADES-COUNT
23.        PERFORM SUM-STUDENT-GRADES
24.        DIVIDE GRADES-SUM BY GRADES-COUNT GIVING GRADES-AVG
25.        IF GRADES-AVG > 95 THEN
26.          DISPLAY STUDENT-ID STUDENT-FIRST-NAME STUDENT-LAST-NAME
27.                  GRADES-AVG
28.        END-IF
29.      END-IF
30.    END-IF
31.  END-PERFORM.

32. SUM-STUDENT-GRADES.

33.  MOVE 0 TO STATUS2
34.  PERFORM UNTIL STATUS2 IS NOT EQUAL TO ZERO
35.    FETCH NEXT GRADES WITHIN STUDENT-OF-GRADES
36.      AT END MOVE 1 TO STATUS2
37.    IF STATUS2 IS EQUAL TO 0 THEN
38.      ADD GRD-GRADE TO GRADES-SUM
39.      ADD 1 TO GRADES-COUNT
40.    END-IF
41.  END-PERFORM.
```

Fig. 2. The original network-database program.

some overhead code required because of the way SQL is used in hosting programs.

The `FETCH USING` and the `FETCH OWNER` commands, each of which fetches a single record, are translated efficiently to the `EXEC-SQL SELECT` embedded SQL commands that are used for single-record retrieval (lines 17 and 27 in Figure 3). For the `FETCH NEXT` commands, used to loop over all members in a link, the

```
1. DECLARE-CURSORS.

2.  EXEC SQL DECLARE CRS1 CURSOR FOR
3.    SELECT STUDENT-ID, FIRST-NAME, LAST-NAME, ADDRESS, PHONE, AGE,
4.           DEPT-NAME, DEGREE
5.    FROM STUDENT
6.    WHERE DEPT-NAME = :DEPT-NAME
7.  END-EXEC.

8.  EXEC SQL DECLARE CRS2 CURSOR FOR
9.    SELECT STUDENT-ID, COURSE, YEAR, SEMESTER, GRADE
10.    FROM GRADES
11.    WHERE STUDENT-ID = :STUDENT-ID
12.  END-EXEC.

13. MAIN.

14.  DISPLAY "ENTER DEPARTMENT NAME:"
15.  ACCEPT DEPT-NAME
16.  MOVE 0 TO STATUS0
17.  EXEC SQL
18.    SELECT DEPT-NAME,SCHOOL-NAME
19.    INTO :DEPT-NAME,:SCHOOL-NAME
20.    FROM DEPARTMENT
21.    WHERE DEPT-NAME=:DEPT-NAME
22.  END-EXEC
23.  IF SQL-STATUS = SQL-NOT-FOUND THEN
24.     MOVE 1 TO STATUS0
25.  END-IF
26.  IF STATUS0 = 0 THEN
27.    EXEC SQL
28.      SELECT SCHOOL-NAME
29.      INTO :SCHOOL-NAME
30.      FROM SCHOOL
31.      WHERE SCHOOL-NAME=:SCHOOL-NAME
32.    END-EXEC
33.    DISPLAY "LIST OF BEST GRADUATE STUDENTS OF " DEPT-NAME
34.            " IN " SCHOOL-NAME
35.    PERFORM ALL-DEPT-STUDENTS
36.  END-IF.
```

Fig. 3. Naive translation of the example program: definition of cursors and fetching the DEPARTMENT and SCHOOL details (continued in Figure 4).

translated program declares two *cursors*, which are SQL queries for multiple-record retrieval whose results can be accessed by the program record by record (lines 1–12 in Figure 3).

While the database access commands have been replaced, there are no differences in the computations performed by the host program. Although SQL allows joining records from two tables, the student and grades are retrieved using two separate database access commands. Thus, while losing the efficiency of the network model, the program fails to exploit the capabilities of the relational model.

```
ALL-DEPT-STUDENTS.

 MOVE 0 TO STATUS1
 EXEC SQL OPEN CRS1 END-EXEC
 PERFORM UNTIL STATUS1 IS NOT EQUAL TO 0
   EXEC SQL FETCH CRS1
     INTO :STUDENT-ID, :STUDENT-FIRST-NAME, :STUDENT-LAST-NAME,
          :STUDENT-ADDRESS, :STUDENT-PHONE, :STUDENT-AGE,
          :STUDENT-DEPT-NAME, :STUDENT-DEGREE
   END-EXEC.
   IF SQL-STATUS = SQL-NOT-FOUND
     THEN MOVE 1 TO STATUS1.
   IF STATUS1 IS EQUAL TO 0 THEN
     IF STUDENT-DEGREE IS EQUAL TO 2 THEN
       MOVE 0 TO GRADES-SUM
       MOVE 0 TO GRADES-COUNT
       PERFORM SUM-STUDENT-GRADES
       DIVIDE GRADES-SUM INTO GRADES-COUNT GIVING GRADES-AVG
       IF GRADES-AVG > 95 THEN
         DISPLAY STUDENT-ID, STUDENT-FIRST-NAME, STUDENT-LAST-NAME,
                 GRADES-AVG
       END-IF
     END-IF
   END-IF
 END-PERFORM.
 EXEC SQL CLOSE CRS1 END-EXEC.

SUM-STUDENT-GRADES.

 MOVE 0 TO STATUS2
 EXEC SQL OPEN CRS2 END-EXEC.
 PERFORM UNTIL STATUS2 IS NOT EQUAL TO 0
   EXEC SQL FETCH CRS2 INTO :GRD-STUDENT-ID, :GRD-COURSE, :GRD-YEAR,
                            :GRD-SEMESTER, :GRD-GRADE
   END-EXEC.
   IF SQL-STATUS = SQL-NOT-FOUND THEN MOVE 1 TO STATUS2.
   IF STATUS2 IS EQUAL TO 0 THEN
     ADD GRD-GRADE TO GRADES-SUM
     ADD 1 TO GRADES-COUNT
   END-IF
 END-PERFORM.
 EXEC SQL CLOSE CRS2 END-EXEC.
```

Fig. 4. Naive translation of the example program: selecting the best graduate students (continued from Figure 3).

1.5 MIDAS's Translation

The translation produced by MIDAS is shown in Figure 5. It replaces the two database access commands for the DEPARTMENT and SCHOOL tables by a single SQL command that joins the two tables. In addition, it employs a single cursor, which contains almost all the computational content of the ALL-DEPT-STUDENTS and ALL-STUDENTS-GRADES paragraphs. The only part that cannot be performed by the database is the display of the results. This part remains in the host program. In particular, the following operations have been folded into the queries:

```
DECLARE-CURSORS.

 EXEC SQL DECLARE CRS1 CURSOR FOR
   SELECT STUDENT.STUDENT-ID, FIRST-NAME, LAST-NAME, AVG(GRADE)
   FROM STUDENT, GRADES
   WHERE DEGREE = 2
         AND DEPT-NAME = :DEPT-NAME
         AND GRADES.STUDENT-ID = STUDENT.STUDENT-ID
   GROUP BY STUDENT.STUDENT-ID, FIRST-NAME, LAST-NAME
   HAVING AVG(GRADE) > 95
 END-EXEC.

MAIN.

 DISPLAY "ENTER DEPARTMENT NAME:"
 ACCEPT DEPT-NAME
 EXEC SQL
   SELECT DEPT-NAME, SCHOOL.SCHOOL-NAME
   INTO :DEPT-NAME, :SCHOOL-NAME
   FROM DEPARTMENT, SCHOOL
   WHERE DEPT-NAME=:DEPT-NAME  AND
   DEPARTMENT.SCHOOL-NAME=SCHOOL.SCHOOL-NAME
 END-EXEC.
 IF SQL-STATUS NOT = SQL-NOT-FOUND THEN
   DISPLAY "LIST OF BEST GRADUATE STUDENTS OF " DEPT-NAME
           " IN " SCHOOL-NAME
   PERFORM ALL-DEPT-STUDENTS
 END-IF.

ALL-DEPT-STUDENTS.

 EXEC SQL OPEN CRS1 END-EXEC.
 PERFORM UNTIL SQL-STATUS = SQL-NOT-FOUND
   EXEC SQL FETCH CRS1
   INTO :STUDENT-ID, :STUDENT-FIRST-NAME, :STUDENT-LAST-NAME,
        :GRADES-AVG
   END-EXEC.
   DISPLAY STUDENT-ID, STUDENT-FIRST-NAME, STUDENT-LAST-NAME,
           GRADES-AVG
 END-PERFORM
 EXEC SQL CLOSE CRS1 END-EXEC.
```

Fig. 5. Translation by abstraction, transformation, and reimplementation of the example program.

—filtering out nongraduate students;
—joining the department and school tables;
—joining the student and grades tables;
—computing the average grade; and
—filtering out students whose average grade is less than 95%.

In addition, unused variables and fields have been eliminated from the program and queries, and the control-flow and data-flow that manage queries are simplified. MIDAS produces programs that are compact in text and utilize the SQL language in an efficient and natural fashion. The resulting program is therefore much easier to maintain than the output of the one-to-one translation.

In addition, efficient use of SQL significantly reduces the amount of data transferred from the database server to the program.

It should be clear from an inspection of this example that translation by abstraction gives much better results than the one-to-one translation method; in fact, the final program in this case is comparable to what a competent human programmer would produce. As we discuss in Section 3, the performance of this program is much better than that produced by the one-to-one method. This example is a very small; however, MIDAS has successfully converted several real large-scale applications, allowing the owners of those applications to retire their legacy databases (see Section 3.2). An evaluation of MIDAS in terms of various criteria (performance, maintainability, robustness, complexity, etc.) appears in Section 3.

We now turn to describe the internal organization of MIDAS and the transformations it uses.

2. THE MIDAS TOUCH

MIDAS abstracts the program by representing it as a plan diagram [Rich 1981; Rich and Waters 1990], which is a control- and data-flow graph. Each database-access operation is represented as an *atomic query graph*, which denotes the retrieval of all the records of a certain record type in the network database. Section 2.3 provides more details about the abstraction stage.

During the transformation stage, MIDAS searches for a small and fixed set of database access idioms. Like the program, the idioms are represented as plan diagrams with embedded query graphs. An occurrence of an idiom triggers the associated transformation, which replaces the matching sub-graph by another graph that has fewer plan-diagram nodes and larger query graphs. In this way, more and more of the plan diagram is absorbed by query graphs, resulting in higher levels of abstraction. Section 2.4 describes the transformations used by MIDAS.

When MIDAS cannot detect additional occurrences of transformations in the abstracted program, the program is reimplemented in the target language. Parts of query graph that can be expressed in SQL (such as filters, joins, and aggregation) are removed from the host language and implemented by SQL. The other parts of the program are modified as little as possible. Section 2.5 describes the reimplementation stage in detail.

The most interesting part of the translation is the handling of queries. Other database operations (insertions, deletions, and updates) are either single-record operations, or can be viewed as applying a database operation to the result of a query. The query part of the operation will be translated according to the scheme described here, while little can be done to improve the database-modification part of the operation. The rest of this paper therefore concentrates on queries rather than on database modification operations.

2.1 The Plan Calculus

The *plan calculus* [Rich 1981; Rich and Waters 1990] is a language-independent wide-spectrum representation of programs. It is a graph whose nodes represent computations, and has two types of arcs, representing data flow and control flow. The nodes can represent atomic computations at the level of the programming language, such as arithmetic and comparison operations, as well as larger units, corresponding to subroutines. *Overlays* are mappings between two equivalent structures; often, one is a single node and the other is a multinode graph. Such an overlay represents the implementation of an abstract concept by a more detailed plan. Overlays can be used as transformations in both directions. Going from abstractions to implementations can be used in a synthesis tool such as KBEmacs [Rich and Waters 1990]. The other direction can be used for analysis of programs, as was done by Wills [1990] for recognizing high-level abstractions in Lisp, by Feldman and Friedman [1999] for assembly-to-C translation, and also by MIDAS.

Figure 6 shows the plan representation of the inner loop of the example program of Figure 2 (lines 33–41). In this plan, the database is represented as a single object that is passed between the various operations performed on it, and is marked "*DB*" in the figure.[3]

Data-flow arcs are represented in the figure as plain arrows; control-flow arcs by slightly thicker cross-hatched arrows. (Control-flow arcs that parallel data-flow arcs have been removed from the figure for clarity.) The control-flow arc that starts in a small black circle at the top of the figure represents control flow coming from the code just before the loop. It enters the righthand side of the *join specification* box, the lefthand side of which represents control returning from the loop body for another iteration. The three data-flow arcs on the right-hand side represent initial values for the loop variables. From right to left, they are: the database, the initial zero for the `GRADES-SUM` variable, and the initial zero for `GRADES-COUNT`. The join box switches both data and control; it therefore has the same number of incoming data-flow arcs on each side, which is also equal to the number of outgoing data-flow arcs. (In the figure, those have been drawn in the same order in all three places.)

The first operation inside the loop body attempts to fetch the next record from `GRADES`; this operation is modeled as an *I/O specification* that returns the new state of the database and the record just fetched. The database is fed into the *test specification* that checks whether the end of the set has been reached. If not, the control-flow arc on the right-hand side (marked "F") is activated, causing the `GRADE` field to be selected from the record and added to `GRADES-SUM`. Then one is added to `GRADES-COUNT`, and control returns to the start of the loop body through the join box.

If the end of the set has been reached, the control flow on the righthand side of the test causes the loop to be terminated and the division that computes the average to be performed (as indicated by the I/O specification box marked "/"). The average can be used by the code following the loop (not shown here).

As can be seen from this example, the plan calculus goes further in its abstraction than other approaches, such as source-to-source transformations [Loveman 1977], program slicing [Weiser 1984], and program-dependence

[3]The italicized variable names in the figure are not part of the plan; they are external annotations for the benefit of the reader.

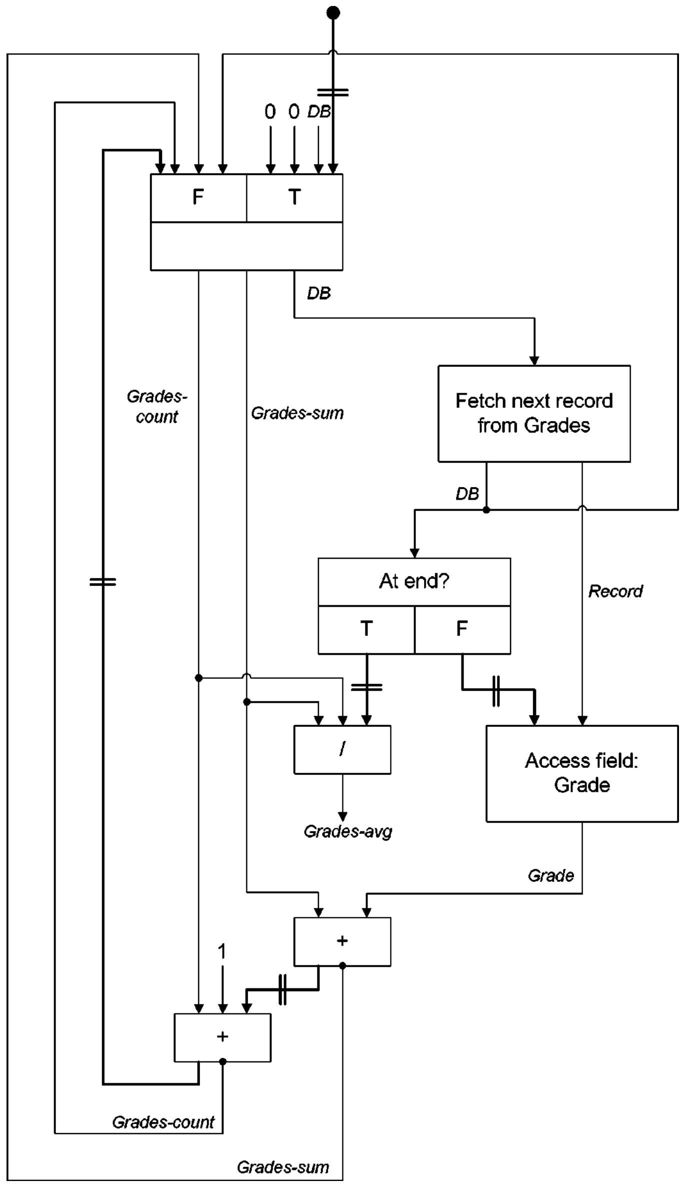

Fig. 6. A plan for the inner loop of the example program.

graphs [Horwitz et al. 1990; Griswold and Notkin 1993], which are more syntax-based. These approaches preserve the syntactic structure of the program as some kind of syntax tree, annotated with data flow and control flow information. In contrast, the plan calculus abstracts away all local variables, replacing them by data-flow arcs. Also, syntactic distinctions between control structures are eliminated. Because it is separated from syntactic conerns, the plan calculus is more amenable to algorithmic analysis. It is also a more canonical representation, and programs that only differ in the syntactic constructs used to achieve data and control flow are represented by the same plan. This could require more work in an interactive setting, where there is a need to continuously change between the syntactic form used by humans and the internal representation used by the application (or, alternatively, to maintain both representations simultaneously). Since MIDAS is completely automatic, this is not a problem for it; translation back into a programming language is done once, during reimplementation (and, indeed, it is necessary to keep syntactic information from the analysis in order to keep the result as close to the source as possible in areas not affected by the translation).

The plan calculus can formally express the equivalence between two plans, typically on different levels of abstraction, by means of *overlays*. (The transformations shown later in the paper are examples of such overlays.) Using overlays, it is possible to represent the program at different levels of abstraction *simultaneously*. This is essential for applying transformations to the representation while keeping the original representation. In particular, temporal abstraction (see below) replaces parts of a loop by higher-level abstractions such as filters or aggregations step by step. Thus, the two representations must be maintained simultaneously during the abstraction process. In MIDAS, this capability is also used in the reimplementation stage; when an abstraction cannot be expressed in SQL, it is returned to a lower-level abstraction.

The more canonical representation of the plan calculus makes it easier to express the patterns that trigger transformations. An algorithmic concept such as filtering may take many differnet syntactic forms, since data flow and control flow can be expressed in many equivalent ways in the source program. The plan calculus abstracts away from these into a single plan, which is easy to express and use in pattern matching.

The initial representation of the source program in MIDAS is a plan, similar to that of Figure 6. During the analysis, parts of this plan are replaced by *query graphs*, which describe higher-level database operations such as those provided by a relational database.

2.2 Query Graphs

Query graphs extend the plan calculus with a representation of database queries. Although the formalism is used in this article to represent network and relational database queries, it is also capable of representing queries for hierarchical and indexed-sequential databases. Query graphs can represent self-contained queries as well as parameterized queries embedded in host programs.

Syntactically, each query graph has a name that uniquely identifies it within the program, and a set of possible outputs.[4] Atomic query graphs have no subcomponents; compound query graphs are composed of simpler query graphs by operations such as filter, join, and accumulation.

[4]The query graph's outputs are referred to as *possible outputs* since an output made available by a query graph need not necessarily be used in the program it is part of.

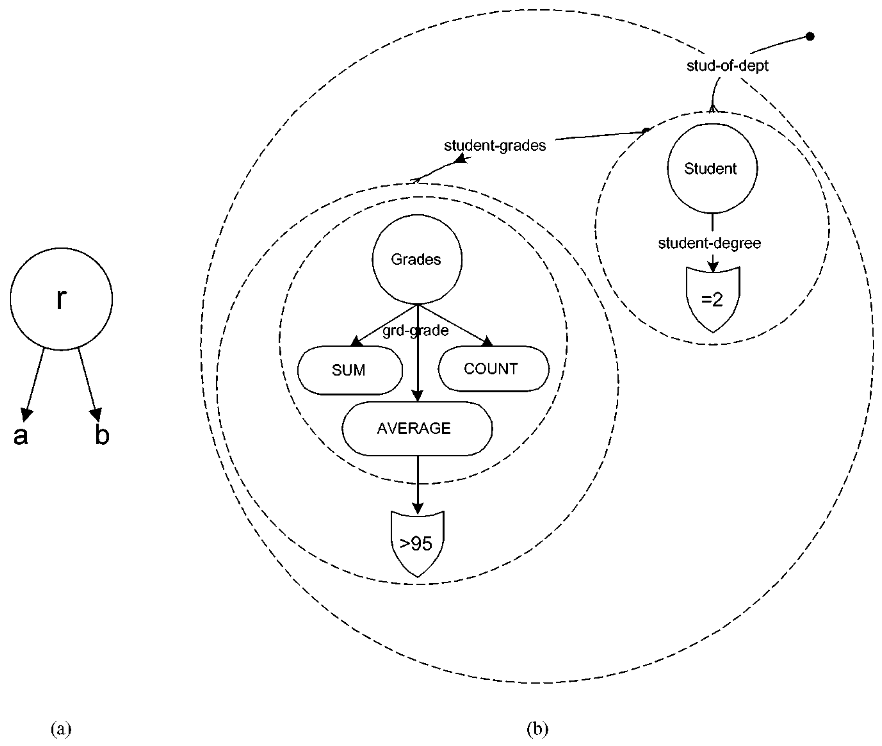

Fig. 7. (a) An atomic query graph. (b) Abstract representation of final program.

The semantics of a query graph is given by a (possibly empty) sequence of records, each of which has a field corresponding to each possible output. An atomic query graph describes the retrieval of all records in the database of a given type, in the order they appear in the database. Compound query graphs describe the results of the appropriate operation on the results of its component query graphs. For example, a filter query graph returns a subset of the records returned by its parameter, and a computed-expression query graph extends each record by an additional field whose value is computed by a given expression from the values of the other fields.

For convenience, we use a graphical notation to describe query graphs. For example, an atomic query graph for a record type with fields *a* and *b* is illustrated in Figure 7(a), and the compound query graph that represents most of the computation of the example program at the end of the abstraction process appears in Figure 7(b). A complete description of compound query graphs appears in Section 2.4, together with the transformations that introduce them into the plan representing the legacy program.

2.3 Program Abstraction

The first stage in the translation task consists of abstracting the given program into a plan diagram. This eliminates much of the syntactic variability inherent in programming languages and the arbitrary decisions made by programmers when a program is represented in textual format. In addition, MIDAS employs additional techniques, such as algebraic simplifications and predicate merging, in order to achieve better canonicalization of the abstracted program.

Reimplementing the result of this step so as to use a relational database will yield the one-to-one translation shown in Figures 3–4. Such reimplementation serves MIDAS as a safety net, in case the transformation stage fails to detect any transformation.

After converting the program into the internal representation, temporal abstraction is applied to the plan in order to convert each operation, originally applied within a loop to a series of separate elements, into a single operation applied to a series. Waters [1978] found that about 95% of the loops in the code of a large numeric-computation package could be replaced by straight-line code using operations such as generate, map, filter, truncate, and accumulate applied to series of values. The concept of temporal abstraction is common in database languages and paradigms. A chain (which is a list of all the members of a single owner record) in a network database and the results of an SQL command both refer to a series of database records as a single object. Hence, the query graph formalism introduced in this article, which is based on temporal abstraction, is a natural representation for database query commands.

Once the program has been converted into a plan with atomic query graphs and loops have been temporally abstracted, MIDAS can go on with its central task, which is to discover high-level database access patterns by applying program transformations.

2.4 Transformations

A transformation expresses the equivalence of two patterns under certain conditions. If the conditions are satisfied when the source pattern is found, it can be spliced out of the plan and replaced by the target pattern. Both patterns are expressed in terms of query graphs embedded in plan diagrams. Each transformation may have constraints on its applicability. Most constraints are concerned with the program's control flow; for example, a constraint may require one database access operation to be in an inner loop with respect to another (as in the downward-join transformation). Constraints may also refer to other attributes of the plan and query graph nodes. For example, the function associated with a plan node may be constrained to belong to a specific set of functions that are expressible in SQL.

Control flow is concisely expressed by *control environments*. A control environment is an equivalence class of plan and query-graph nodes that are executed the same number of times and under the same conditions in all executions of the program. An IF statement splits its control environment into two new control environments, representing the two branches. These can be joined later into the original control environment, in statements that follow the IF. Similarly, a loop creates a new control environment for its body.

The rest of this section presents the complete set of transformations used by MIDAS. Most transformations replace a number of plan and query-graph nodes by a larger query graph. Others are used to establish the conditions for

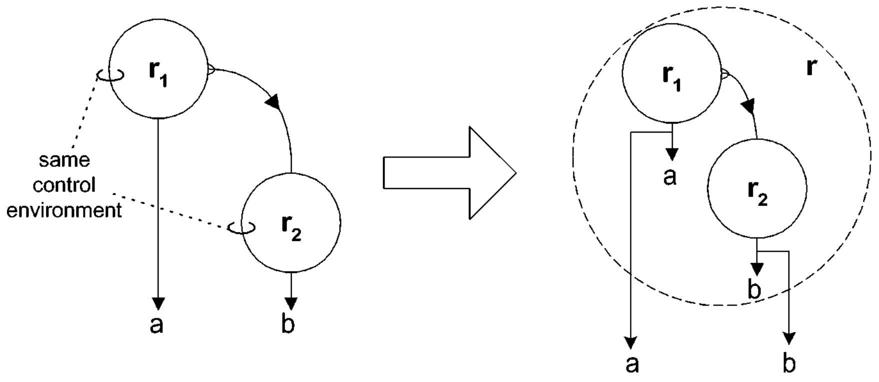

Fig. 8. The upward join transformation.

the application of these transformations, or to remove redundant parts of the plan. The transformations are presented here in the order in which they are applied to the example program of Section 1.3.

2.4.1 *Safe Fetch.* The AT END clause of the FETCH statement can be used to detect whether a network database operation has retrieved any data, or has failed. This clause is activated whenever the operation has failed, and is typically used to set a flag, which is checked by later stages of the program. The *safe fetch* transformation identifies such cases, removes the parts of the program that manage the control flow of the exceptional cases, and replaces them by annotations on the control environments of affected basic blocks and on the query graphs.

There are two variations of the safe-fetch transformation. In the case of an operation that retrieves a single record, the AT END clause is used to check that a record was in fact retrieved. The *safe simple fetch* transformation detects this case. When a database access operation occurs inside a loop, the AT END clause is typically used to set a flag that terminates the loop. This case corresponds to the truncation of a temporal sequence, and is handled by the *safe multiple fetch* transformation.

In the example program (Figure 2), the safe simple-fetch transformation removes the safety check on the retrieval of the department (line 5). The safe multiple-fetch transformation is applied twice, to remove the checks and related control mechanisms from the retrieval of the students and their grades (lines 17 and 35).

2.4.2 *Upward Join.* A link in the network database represents a one-to-many relationship from one record type, called the *owner*, to another, called the *member*. Such a link corresponds to the join of two relational tables. An *upward join* (righthand side of Figure 8) represents the retrieval of an owner of a previously fetched member record. This operation retrieves a single record.

The upward-join operation merges two query graphs. A link l connects the query graph r_1 to another query graph r_2. The member of the link must be an atomic query graph that is part of r_1, and the owner must similarly be an

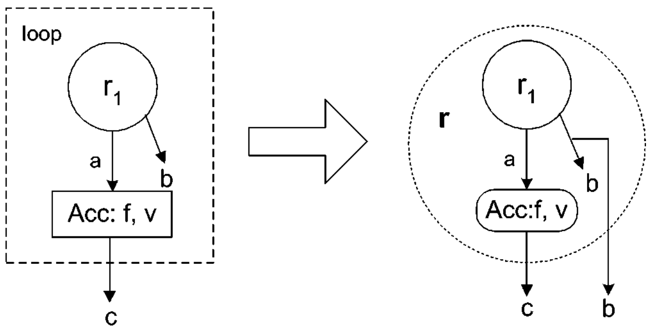

Fig. 9. The accumulation transformation.

atomic query graph contained in r_2. Each database link in a query graph has a *data-flow direction*. In the case of an upward join, the data-flow direction of the link is from the member to the owner. The sequence of records represented by the combined query graph r is the same as that of r_1. However, the set of possible outputs of r is extended to be the union of the outputs of r_1 and the outputs of r_2 (a and b in the example).

An upward-join is graphically represented by a database arc between r_1 and r_2, tagged by the name of the database link. The fork identifies the member. The arrow head pointing to r_2 represents the member-to-owner data-flow direction of l.

The upward join transformation (Figure 8) identifies the case in which the network-database owner of a previously fetched record is retrieved. This transformation merges the two query graphs and the database link between them into a single query graph, which may later be implemented as a join between two tables in the relational database.

In order for the transformation to occur, both query graphs must reside in the same control environment. In addition, there may not be any data-flow between the two query graphs outside of the merged query graph. This constraint is not a correctness requirement; it exists in order to simplify the reimplementation stage by avoiding the detection of database access patterns that are not expressible in SQL.

In the example program, the retrieval of the department (line 5) and the retrieval of the school (line 8) as the owner of the department are in different control environments, since the latter is inside one branch of the IF statement that checks the safety of the former. Therefore the upward-join transformation is inapplicable. However, the safe-fetch transformation merges the two control environments, and then upward-join can merge the two query graphs.

2.4.3 *Accumulation.* The application of a binary function (e.g., addition) to successive elements of a sequence in a loop is temporally abstracted into an *accumulation*, which is an operation that takes a sequence and returns a scalar value. The *accumulation* transformation (Figure 9) recognizes such cases, and replaces that part of the loop that applies the binary operator by an accumulation of the sequence of values represented by the query graph.

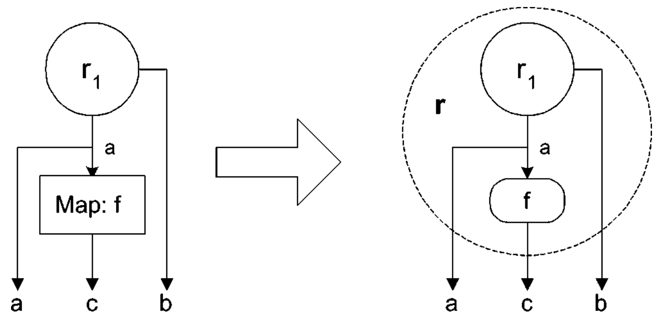

Fig. 10. The computed-expression transformation.

An accumulated-expression query graph (right-hand side of Figure 9) represents aggregative computations such as counting records, summing a field, calculating a field's average, or computing a field's minimum and maximum values. The accumulated-expression operation contributes a new output to the query graph. This new output is the result of applying an aggregative function f from a predefined set of such functions (such as SUM, COUNT, MIN, MAX, and AVG) to the sequence of values represented by an output of the given query graph (in the example, output a of r_1) and a constant initial value v. Like the computed-expression operation, this operation adds a new output to the original query graph; the value of this output in each record in the sequence is the value of the aggregative function. The field whose value is accumulated is removed from the outputs of the query graph.

The accumulation transformation is constrained to occur only if the query graph and the accumulative function reside in the same control environment. The transformation is also constrained to functions that can be reimplemented in the relational model, such as summing a database expression, counting a sequence of records, and detecting the minimal and maximal value of a database expression in a sequence of records. This constraint exists in order to simplify the reimplementation.

The example program contains two cases of accumulation: computation of the sum of the student's grades, and counting those grades. (Counting the elements of a sequence is a degenerate case of accumulation in which the values of the sequence elements are not used.) A later simplification (a special variant of the computed-expression transformation) will recognize that the sum of a sequence of values divided by the number of elements gives the average of the elements, and this will be reimplemented as the SQL aggregative operation `AVG(GRADE)`.

2.4.4 *Computed Expression.* Accumulation adds a new possible output to the result of a query. Relational queries can also add outputs containing the results of non-aggregative operations. The *computed-expression* transformation (Figure 10) transfers such computations from the plan representing the original Cobol program into the query graphs. The new possible outputs can serve as the basis for further transformations, such as accumulation, filtering, and additional computed expressions.

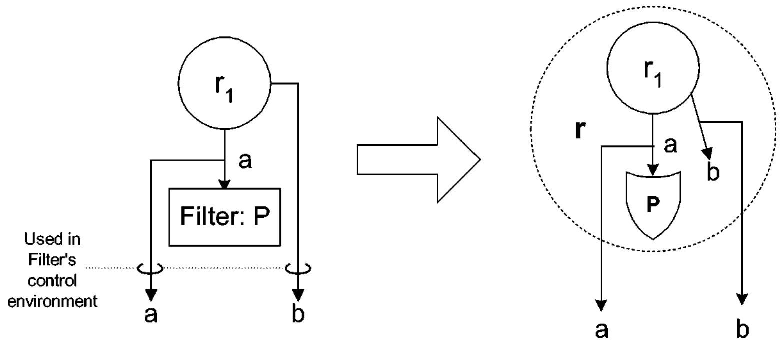

Fig. 11. The filtering transformation.

The computed-expression operation applies a function f on at least one output from a given query graph r_1, (a in the figure) and produces a new output whose value is the sequence of values resulting from the application of the function to each value of the original sequence.

2.4.5 *Filtering.* The network database interface provides only rudimentary filtering capabilities. The only filter possible is to fetch records using equality on key fields. All other filters have to be implemented in the host program by using the `IF` command. The *filtering* transformation (Figure 11) recognizes the case in which only part of the records fetched from a given set are actually processed by subsequent code. In this case, the filtering predicate is added to the query graph, and removed from the plan-calculus representation of the host program. If possible, the reimplementation phase will insert the filter into the SQL statement it generates from the query graph.

The filter operation applies a Boolean predicate P from a predefined set of predicates, on at least one output from a given query graph r_1. The resulting sequence of records of the result is a subset of the sequence of records of r_1, which includes only those records that satisfy the predicate P.

The filtering transformation cannot be applied if there are any operations performed in the control environment of the query graph except for the predicate, since operations that apply to two different sequences of records are not expressible in SQL. The predicate used by the `IF` command must use data-flow from the query graph, since otherwise the predicate cannot be connected to the query graph. In addition, the `IF` command may have either a "then" or an "else" part but not both, and only this control environment may use the query graph's outputs.

In the example program, this transformation is used to replace the filter `IF STUDENT-DEGREE IS EQUAL TO 2` in the original Cobol program by the SQL clause `WHERE STUDENT-DEGREE = 2`. In this case, the filter is applied to a field in the record.

The filtering transformation can also be applied to query graph outputs that are the result of an aggregative operation such as MIN, MAX, SUM, COUNT, and AVG. In the example program, the transformation is used to replace the filter applied to the aggregated value in the `GRADES-AVG` variable in the original Cobol program (`IF GRADES-AVG > 95`) by the SQL clause `HAVING AVG(GRADE) > 95`.

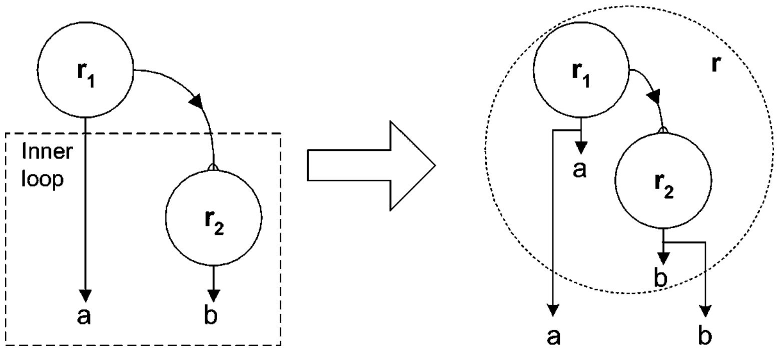

Fig. 12. The Downward join transformation.

2.4.6 *Downward Join.* The *downward-join* transformation (Figure 12) is similar to the upward-join transformation, except that it deals with the case in which the retrieval direction is reversed, that is, multiple member records belonging to a previously retrieved owner record are fetched. The owner query graph and the member query graph must contain atomic query graphs that serve as the owner and the member of some database link, respectively. However, unlike the case of the upward-join transformation, the control environments of the retrieval operations on the owner and member records are different. The member's control environment is constrained to be within the owner's control environment. (One control environment is contained within another control environment if the commands of the inner control environment are executed only if commands from the outer control environment are executed. This relationship is transitive.)

The graphical representation of a downward join is similar to that of upward join, except for the modifications implied from the changes in the roles of the owner and member: r_1 contains the owner and r_2 contains the member, and the data-flow direction of the link is from the owner to the member. The semantics, however, is very different. In this case, each owner may have multiple member records (or none). The sequence of records represented by the result of the downward join is the concatenation of all the member record sequences in r_2 that have an owner record in r_1.

An important restriction on the downward-join transformation is that all outputs of both query graphs are used either by the inner control environment (and other control environments contained within that control environment) or by control environments outside it, but not both. If one of these conditions is met, the two query graphs are merged into a single query graph. In the first case, the new query graph belongs to the control environment of the member query graph; in the second case, the new query graph belongs to the control environment of the owner query graph.

In the example program, the downward-join transformation is used to create a join between the STUDENTS and their GRADES in a single SQL query. The combined query graph is placed in the control environment of the owner query graph, since the student details, grade average, sum, and total number are used by the control environment of the owner query graph (i.e., the one that retrieves the student record).

Like the upward-join transformation, the downward-join transformation is not allowed if there is any data-flow between the two query graphs outside the merged query graph. This is to avoid detection of database access idioms not expressible in SQL, in order to simplify the reimplementation stage.

The new query may be reimplemented later as a relational join or as a subquery, depending on the "safety" annotations on the query graphs and the way in which its outputs are used by the program.

The upward-join and downward-join transformations are the most common cases of the general *cartesian product* transformation, in which the two query graphs are not necessarily related by owner–member links. In this case, any predicate is sufficient to indicate the creation of a cartesian product. These cases are reimplemented as joins or subqueries, and capture the intersection and subtraction database idioms. MIDAS does not currently support cartesian product. Cartesian product transformations are relatively rare in network database programs. However, the general cartesian-product transformation can be implemented according to the same principles, if necessary.

2.4.7 *Elimination of Empty Control Environments.* After the safe-fetch and the accumulation transformations are applied to the `SUM-STUDENT-GRADES` loop, its body is reduced to a single query graph. Similarly, after applying the downward-join transformation, the body of the `SUM-STUDENT-GRADES` loop becomes empty. MIDAS eliminates such control environments from the program. If such a control environment belongs to a loop, the loop command is also removed. If the removed control environment belongs to an `IF` command, and the other control environment of that `IF` command is also empty, the `IF` command is also removed. This simplifies the final reimplemented code.

2.5 Reimplementation

The abstraction stage terminates when the analysis algorithm cannot find further transformations to apply. MIDAS then reimplements the network program as an embedded SQL program. The reimplementation stage receives three more inputs in addition to the abstracted program: the original network schema, the destination relational schema, and a set of schema transformations. Schema transformations map elements in the network schema (record types, links, and fields) to objects in the relational schema (tables, fields, indexes, and referential integrity rules). The reimplementation stage uses these inputs to produce a Cobol program with embedded SQL that is functionally equivalent to the original network program.

2.5.1 *Reimplementing the Program.* The reimplementation stage is relatively straightforward in comparison with the transformation stage. It translates those parts of the final output of the transformation stage that are still expressed as plans back into Cobol. However, several heuristics are used in order to produce a readable and understandable program. For example, information about the structure of the original program is retained, and this structure is preserved as much as possible.

2.5.2 *Reimplementing Query Graphs.* When MIDAS attempts to translate query graphs into SQL, it uses the following pattern:

> SELECT ⟨Names of outputs used outside the query graph⟩
> FROM ⟨Names of all atomic query graphs contained in the query graph⟩
> WHERE ⟨A predicate for each filter and join⟩
> ORDER BY ⟨List of field names deduced from network schema and data-flow direction⟩
> GROUP BY ⟨Field names that manage aggregation, if any⟩

If the query contains filters on aggregative values, the reimplementation format of query graphs will add the following clause:

> HAVING ⟨Predicate for each filter applied to aggregative values⟩

In certain cases, such as when the output value of one query graph is used in a filter of another, MIDAS employs subqueries in the reimplementation.

As mentioned above, query graphs have greater expressive power than SQL. An aggregation of an aggregation and query graphs that have data-flow going from the query graph to the plan diagram and from there back to the same query graph are acceptable query graphs. However, they are not expressible in SQL. The portions that are not expressible in SQL are reimplemented in the host language.

While in principle the transformation and reimplementation stages are separate and need not be aware of each other, it is useful to prevent some transformations whose results are known to be inexpressible in SQL. For some transformations, MIDAS imposes stronger constraints than are necessary for correctness, in order to simplify reimplementation (and also accelerate the transformation stage).

2.5.3 *Schema Awareness.* The reimplementation stage is the only stage in which details of the schema conversion are relevant. Analysis of the original program in terms of query graphs is done on the original database schema. Queries are written for the new schema during reimplementation. Thus, different schema conversions can easily be accommodated. For example, suppose that the relational schema of the example database contains different tables for graduate and undergraduate students, in both of which the STUDENT-DEGREE field has been eliminated. The general technique for dealing with such splits is to generate a union query, whose subqueries add the missing field. However, MIDAS uses algebraic simplification of the union query to eliminates the subquery that refers to the table containing the data of undergraduate students. The added field and the filter will be successively eliminated, resulting in a shorter and simpler query. Similar techniques can be used for other typical modifications.

Unlike network databases, relational databases require their data to be in first normal form (1NF). This means that the original network schema may contain subrecords (also called *group items*), redefinition of data areas, and arrays, none of which may be ported directly to the relational database. MIDAS can easily be extended with new transformations to handle non-1NF data.

In the case of subrecords, MIDAS can flatten the structure and add the group items to the outputs of the query graph. There are two ways to treat redefined data areas: flattening them into the main table or extracting each variation into a separate tables. The translated programs will use the relevant set of relational tables and fields using the logic of the original program (typically by inspecting the content of a tag field). The selection logic may sometimes be reimplemented in SQL, but in some cases will be left in the hosting program.

Arrays represent sequences of data records, and can therefore naturally be represented by query graphs. This would allow MIDAS to use its existing transformations to identify all applicable operations, including filters and accumulation. This can easily be done by translating an access to a record that contains an array by a compound query graph containing one atomic query graph corresponding to the main record, and another query graph for each array, with a downward-join arc from the main record. These downward-join arcs will be tagged as representing an array link, for correct treatment during reimplementation.

The translation of most of the other schema transformations is straightforward.

3. RESULTS

In the worst case, MIDAS reverts to the naive translation. However, since it is able to recognize most of the common database programming patterns, it will generally do much better. Since it folds part of the computation from the host program into embedded database commands (SQL in this case), the resulting program is more efficient, and is also more compact and maintainable. This section examines the performance of MIDAS on several criteria.

3.1 Performance

Many legacy database applications are mission critical; the efficiency of the converted program is therefore of paramount importance. The two main measures of efficiency in this case are execution time and data traffic. The first is important for any system architecture. The second is crucial in distributed environments such as client/server or Internet architectures common to many modern applications.

We compared the performance of the results of the naive one-to-one translation (Figures 3 and 4) and the program produced by MIDAS (Figure 5). Both programs were simulated on the same PC platform. In a series of experiments in which the database was generated randomly with varying parameters, we got the following results:

—the time taken by the program produced by MIDAS was between 81% and 90% shorter, and

—the data traffic was between 90% and 99.9% smaller.

Of course, these results may vary with different implementations and different problems. However, they do indicate that the optimizations performed by MIDAS can be very significant in terms of both time and data traffic. Time efficiency is obtained by moving as much of the computation as possible to the database server, enabling it to perform meaningful optimizations. Part of the reduction in data traffic is due to the same reason. In fact, many queries access a lot of data but only provide very short answers, either because they find a specific piece of data or because they provide an aggregative summary of some sort (possibly even just a Boolean value indicating whether any answer exists to a certain question). In these cases, we can expect the reduction in data traffic to be considerable. Even when the query returns many records, the optimization that drops unused fields may provide a significant reduction of data transferred.

3.2 Industrial Experience with MIDAS

Alexandria, founded in 1999, is a company that develops commercial automatic conversion tools for legacy database applications. These tools convert the database schema, the application data, and the application code. The code conversion part of the tools is based on MIDAS. Although it does not yet contain the full functionality of MIDAS, it does contain several modules that are missing from the research prototype but are necessary for industrial-strength application, such as the treatment of non-1NF schemas (see Section 2.5.3).

One of the first tools developed by Alexandria converts Cobol programs written for IDMS (a network DBMS from Computer Associates) to DB2 (IBM's relational DBMS). This tool has been used so far in two large conversion projects, and has successfully converted more than 1000 programs with over 2.2 million lines of code.

In one of these projects, a trading system of a major brokerage house was converted. The performance of the main program of this system in the original IDMS environment was hand-tuned, reducing its running time from 10 to 2 hours. The converted relational program (without any manual tuning) was slightly faster than the hand-tuned original. (The comparison was done on the same hardware and operating system).

In retrospect, the following characteristics of MIDAS contributed greatly to the project's success:

—The ability to generate compact SQL statements that replace some of the original Cobol code (clients were most impressed by the discovery and construction of joins).
—The flexible support of schema transformations, due to the independence of the analysis and transformation stages from the target schema.
—The generation of cursors or SELECT INTO statements as appropriate, resulting in more compact and maintainable code.

3.3 Utilization of SQL

The one-to-one translation approach uses SQL to simulate the database access operations of the original network database, a task ill-suited for a relational database setting. In contrast, MIDAS is able to use the following powerful features of SQL in order to produce an efficient program.

—*Joins.* Using joins to intersect data that come from two or more tables allows the database server to use indexes to accelerate the computation.
—*Computed Expressions.* Sometimes, instead of field values, a query returns some expression computed from them. In such cases, the database server can perform the computation and returns just the final result, thus reducing the overall data traffic. For example, if a record contains an item's price, amount, and a tax rate, using a computed expression that calculates only the purchase price reduces the data traffic by two thirds, assuming that only the purchase price is actually used by the program.
—*Aggregations.* Many programs compute statistical summaries of the raw data. MIDAS can move these aggregative operations into the SQL query, increasing efficiency and dramatically reducing data traffic.
—*Filters.* Filters (applied to simple or aggregative values) are implemented in SQL as predicates applied to the query results. This too increases efficiency and reduces data traffic.

3.4 Maintainability

The folding of Cobol code into SQL queries results in more compact code, due to the removal of data- and control-flow constructs used to manage the pattern of database-access operations in the original program. MIDAS uses several additional tactics in order to make the resulting code more maintainable, by making it easily readable while preserving the original structure of the program as much as possible. (These tasks mostly operate in the reimplementation stage.)

—*Noncursor SQL Commands.* MIDAS tracks the expected number of records from each query graph (whether it produces a single or multiple records). Therefore, it can avoid using cursors when a query returns a single record, and can use the lightweight `EXEC-SQL SELECT` statement. This reduces the overall amount of resources required by the program, and also makes it shorter. Cursors are quite verbose: a cursor must be declared, opened, used, and finally closed, using four different commands that are spread over several control environments. This can be reduced to a single SQL command when the command is known to produce a single record.
—*Dead-Code and Dead-Variable Elimination.* Network programs must use many temporary variables for their computations. The use of arithmetic and aggregative expressions in the generated SQL queries makes such computations and variables irrelevant and enables MIDAS to discard them from the resulting program, thus producing a more efficient, as well as more compact and readable program.
—*Redundant-Join Elimination.* Network programs sometimes have to use a hierarchy of links in order to access the data they require. In a one-to-one translation, each such link will be replaced by a cursor, a loop, temporary variables, and code to manage loop control-flow and data-flow. However, in

the relational model, it is possible to eliminate some or all of the tables representing inner links if there are foreign keys that connect tables that are nonadjacent in the hierarchy.

3.5 Robustness

If the transformation stage fails to apply any transformations, MIDAS's translation is reduced to the one-to-one translation. Therefore, MIDAS will never fail to produce a working program. Furthermore, many types of syntactic variations in the original program, which might confuse syntax-based translators, disappear when the program is converted to the plan representation [Rich and Waters 1990].

During the transformation stage, MIDAS ignores the original schema, the destination schema structure, and the complexity of the schema transformations. This is a result of the fact that MIDAS represents database access operations by query graphs, which are model independent. This independence is an important contribution to the robustness of the translation process.

3.6 Complexity

The complexity of MIDAS is dominated by the transformation stage. This stage has the following general structure:

```
loop
    for every query graph r in the program do
        for every transformation t in the knowledge base do
            if t matches the code around r then
                apply transformation t to matching code
until no transformation found
```

Therefore, the complexity of the translation process depends on the following factors:

—The number of plan diagram nodes in the original program, n. This number is roughly similar to the number of Cobol commands in the program. Since each transformation reduces the total number of plan diagram nodes by at least one, the number of transformations that can be applied is bounded by n.

—The number, q, of atomic query graphs in the original program. This number is an upper bound on the number of query graphs within the program at any point during the transformation stage. This number can theoretically be as large as n. However, in practice, each database record type is accessed once or twice in a program. Although the number of database accesses in a program is expected to be related to the program size, it is significantly smaller than the number of commands in the program. Since all transformations contain at least one query graph on their left side, the transformation algorithm uses the query graphs as a hook to its matching algorithm and tries to match the transformation around them. Therefore, after at most $O(q)$ attempts the inner loop of the transformation algorithm either finds an applicable transformation or stops.

—The number of transformations, t. This is a small and fixed number. The algorithm checks the patterns of each transformations for any query graph in the program.

—The maximal size e of a control environment in terms of plan diagram nodes. Matching a pattern costs $O(\binom{m}{k})$, where k is size of the pattern and m is the size of the group being searched. The patterns of the transformations used by MIDAS are all composed of two elements, one of which is the query graph around which the pattern is applied. Therefore, in this case $k = 1$, and the pattern-matching cost is $O(m)$. Since the elements of the pattern belong to either one or two control environments, the transformation algorithm tries to detect the patterns only within the relevant control environments. In case there is an inner control environment involved in the pattern, the algorithm uses database flow arcs to reduce the number of control environments searched for the pattern. Therefore, $m = e$, the size of a control environment. This can be theoretically as large as n. Usually, this is not the case, since splits in the control-flow divide the program into several control environments. The size of a control environment is typically independent of the program size.

The overall complexity of the transformation algorithm is $O(nqte)$. In theory, this can be as high as $O(n^3t)$. In practice, the size of control environments (e) is small, as is the number of query graphs in the program (q). Therefore, qte is usually a small number, and the transformation algorithm behavior is expected to be close to linear in most cases.

In addition to the query-graph transformations, it is necessary to convert the source program into a plan, and identify temporal abstractions in it. The former involves standard data-flow and control-flow algorithms. The analysis of the latter is quite similar to the analysis of query-graph transformations given above.

It is difficult to relate the running time of the commercial tool described in Section 3.2 to the complexity of the algorithm, since the tool performs many other actions during conversion (such as collecting information about various characteristics of the program and storing it in a database). An upper bound is the total execution time, which is about 12 seconds for a 1000-line Cobol program. Most of that time is spent in reimplementation; we estimate that the database flow analysis and the transformations take less than 10% of the total conversion time.

3.7 Uniqueness

In many transformation systems the order of activation of the system's transformations may influence the optimality and even the correctness of the final result. However, MIDAS is guaranteed to produce the same result regardless of the order in which transformations are activated.

In order to establish this claim, it is enough to show the Church–Rosser property: any given two *available transformations* (i.e., transformations whose preconditions are fulfilled in the current plan), can be applied in either order, yielding equivalent results.

Since the formal model of query graphs is not in the scope of this article, it is not possible to fully describe the uniqueness proof here, and we leave it to be described in a future article.

3.8 Generalizations

MIDAS translates source programs that use network database DML. However, extending MIDAS to additional legacy database models (the hierarchical and the indexed-sequential) will influence the program's parsing stage only; the transformation stage is not affected. The reimplementation stage also must be changed since it is dependent on both the original schema and the destination schema.

4. DISCUSSION

4.1 Related Work

4.1.1 *Translation by Abstraction and Reimplementation.* The idea of using the plan calculus as a foundation for translating programs from one language to another was introduced by Waters [1988] and extended by Feldman and Friedman [1999]. MIDAS also follows the approach of abstraction, transformation, and reimplementation described in that paper. Representing the program by the plan calculus abstracts it from the specific attributes of the original programming language. It is then possible to reimplement it in the destination language. It is sometimes desirable to abstract the program further. This is done by using transformations, in this case, by detecting the occurrences of idioms and replacing them by high-level constructs. Wills' Recognizer [Wills 1990] is, in some aspects, a demonstration of such capabilities, since it produces a textual representation of the recognized program.

An earlier work that translates Cobol programs into HIBOL (High Business Oriented Language) is Faust's SATCH [Faust 1981]. SATCH has many limitations, such as allowing only a single loop in the translated program. However, it demonstrates several aspects of plans well, especially temporal abstraction. The category of programs SATCH can translate are batch Cobol programs, which loop over an input file, perform some calculation, and produce an output file. Such programs are in many ways like database programs. In addition, the main technique SATCH uses is temporal abstraction. Since query graphs are a specialization of temporal abstraction focused on database access, it is not surprising that most temporal abstraction operations detected by SATCH can be captured by MIDAS and can be expressed as query graphs. Because it does not require that the entire source program be captured by its idioms, MIDAS is not subject to the limitations SATCH has on the class of programs it can analyze.

The above attempts were laboratory experiments on toy programs. A large-scale experiment was first conducted by Feldman and Friedman [1999]. In this work, a large assembly-language system (the Sapiens application generator) was automatically translated into C. The translation tool, Bogart, parsed the assembler programs into plan diagrams. This language-independent representation allowed it to detect function signatures, eliminate the use of registers, differentiate integers from pointers and more. This allowed the reimplementation stage of Bogart to produce an efficient C program.

4.1.2 *Translating Database Applications.* A lot of research has focused on *data reverse engineering*, which is the translation of the database schema and data migration. Davis and Aiken [2000] give a survey and extensive references. Most of this research has ignored the program as a source of information; a notable exception is DB-MAIN [Henrard et al. 1998], a database engineering CASE environment containing reverse engineering tools. These include some source-code analysis functions, which are mostly text-based.

In 1982, Katz and Wong [1982] suggested an approach to the translation of network database queries to the relational model. They focus only on the embedded network DML, basically suggesting a one-to-one translation. They present a database access-path model that focuses on database flow obtainable from the embedded DML. Therefore, the access path model deals with fetching database records, fetching linked records, and using fields from the fetched data. They use pattern-matching techniques in order to detect access paths in the program, called C diagrams. In cases in which the C diagrams are wrapped around each other without any additional commands between them, the two C diagrams are translated into a single cursor. This approach is fragile, and fails to detect the C diagrams if the elements that compose them are delocalized in the program. MIDAS, however, detects such database idioms routinely. The access-path model does not address other database access idioms such as filtering and accumulation.

In 1991, a group at Lockheed performed a migration project of a hierarchical IMS database application to the relational model [Polak et al. 1995]. The approach was again one-to-one translation. Unlike network databases, IMS uses API calls rather than embedded DML. Hence, a major task is to make the database access explicit. The project therefore uses data-flow and control-flow analysis in order to make the IMS DML explicit. The DML is then translated in a one-to-one fashion. Since IMS allows specifying conditions in its API calls, some aspects of filtering are covered. The authors are aware of the existence of additional database idioms such as joins and accumulation. They leave this point to be addressed by future work. The described tool disregards inter-record filtering and filters applied to accumulations.

The authors of that paper recognize that one-to-one translations are unsatisfactory, and describe a component that finds programming idioms in order to provide better translation of them. However, they say:

> At present, the tool understands a small number of frequent patterns and their translation but is easily extended. During application translation, the tool will convert those IMS calls that fall into known patterns. Other calls are flagged for manual inspection and translation. As we manually inspect more and more applications, we find additional translation patterns that are then codified as new patterns and automated.

In contrast, MIDAS demonstrates that a small and fixed set of transformations are sufficient for high-quality translation.

4.2 Conclusion

MIDAS demonstrates that it is possible to perform high-quality conversion of legacy software from one database model to another using a small and fixed set of transformations. These correspond to the native operators available in the target database. It is not necessary to achieve a high-level of program understanding that is based on a large library of idioms [Wills 1990], since the host language remains the same. Furthermore, the process can be completely automatic, since it is always possible to fall back to one-to-one translation if no higher-level abstractions are found.

Our implementation of MIDAS uses Cobol as the host language, since this is the primary language in information-processing applications. However, other procedural languages can be accommodated. Indeed, because the same host language is used in both the source and the target, a relatively low level of understanding of the host program is necessary. Sometimes, it is desirable to change the host language as well. In this case, the use of a language-independent intermediate representation enables the independence of the analysis and reimplementation stages, each of which is concerned with a single language.

A particularly interesting case for database software is migration to an application generator that is integrated with the database. Such conversion requires a paradigm shift from procedural code to a data- and event-driven model. The query graphs constructed by MIDAS from the original program form a sound basis for the data analysis of the program. This has in fact been demonstrated in a precursor of the Alexandria tool mentioned in Section 3.2, which translated RPG programs to Magic, a declarative, codeless, data- and event-driven application generator.

While this article has dealt with the migration from network to relational databases, the techniques are also applicable to other models. The same kind of analysis can be used to discover high-level operations in indexed-sequential and hierarchical database programs, and the transformation and reimplementation phases need not be changed. If the target database model is object-oriented, only the reimplementation stage needs to be changed. Object-oriented query languages (such as OQL) are a superset of SQL, and therefore the same techniques will apply. Furthermore, object-oriented databases are closer to network databases in that they support object identity and pointers. Because of the explicit representation of the relationships between record types in query graphs, it is possible to bypass some problems that are caused by the incompatibility between the network and relational models, and perform a more direct translation to object-oriented databases.

Of course, such use of MIDAS to migrate to object-oriented databases does not use an important feature of such databases, which is the ability to use methods as part of the query. In order to define such methods, a global analysis of the application will be necessary. Such analysis will have to discover repeating patterns of computations around database access operations. We expect query graphs to play a key role in future research and applications of such analyses.

ACKNOWLEDGMENT

We are grateful to Shimon Schocken for helpful comments on earlier drafts of this article.

REFERENCES

Buss, E. et al. 1994. Investigating reverse engineering technologies for the CAS program understanding project. *IBM Systems J. 33*, 3 (Mar.), 477–500.

Davis, K. H. and Aiken, P. H. 2000. Data reverse engineering: A historical survey. In *Proceedings of the 7th Working Conf. Reverse Engineering (WCRE'00)*. 70–78.

Faust, G. 1981. Semiautomatic translation of COBOL into HIBOL. Tech. Rep. 256, MIT Lab. for Computer Science. Master's thesis.

Feldman, Y. A. and Friedman, D. A. 1999. Portability by automatic translation: A large-scale case study. *Artif. Intell. 107*, 1, 1–28.

Fong, J. 1992. Methodology for schema translation from hierarchical or network into relational. *Inf. Softw. Tech. 34*, 3 (Mar.), 159–174.

Fong, J. and Bloor, C. 1994. Data conversion rules from network to relational databases. *Inf. Softw. Tech. 36*, 3 (Mar.), 141–153.

Gillenson, M. L. 1990. Physical design equivalencies in database conversion. *Commun. ACM 33*, 8 (Aug.), 120–131.

Griswold, W. G. and Notkin, D. 1993. Automatic assistance for program restructuring. *ACM Trans. Softw. Eng. Meth. 2*, 3 (July), 228–269.

Henrard, J., Englebert, V., Hick, J.-M., Roland, D., and Hainaut, J.-L. 1998. Program understanding in databases reverse engineering. In *Proceedings of the 9th International Conference Database and Expert Systems Applications (DEXA'98)*. 70–79.

Horwitz, S., Reps, T., and Binkley, D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst. 12*, 1 (Jan.), 26–60.

Katz, R. H. and Wong, E. 1982. Decompiling CODASYL DML into relational queries. *ACM Trans. Datab. Syst. 7*, 1 (Mar.), 1–23.

Loveman, D. B. 1977. Program improvement by source-to-source transformation. *J. ACM 24*, 1 (Jan.), 121–145.

Polak, W., Nelson, L. D., and Bickmore, T. W. 1995. Reengineering IMS databases to relational systems. In *Proceedings of the 7th Annual Software Technology Conference*, (Salt Lake City, Ut.). Published on CD-ROM.

Rich, C. 1981. A formal representation for plans in the Programmer's Apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence* (Vancouver, BC, Canada). 1044–1052. (Reprinted in M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, pages 239–270, Springer-Verlag, New York, NY, 1984, and in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986).

Rich, C. and Waters, R. C. 1990. *The Programmer's Apprentice*. Addison-Wesley, Reading, Mass., and ACM, New York.

Spooner, D. L., Sanderson, D., and Charalambous, G. 1989. A data translation tool for engineering systems. In *Proceedings of the 2nd International Conference on Data and Knowledge Systems for Manufacturing and Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 96–104.

Tangorra, F. and Chiarolla, D. 1995. A methodology for reverse engineering hierarchical databases. *Inf. Softw. Tech. 37*, 4 (Apr.), 225–231.

Waters, R. C. 1978. Automatic analysis of the logical structure of programs. Tech. Rep. 492, MIT Artificial Intelligence Lab. PhD thesis.

Waters, R. C. 1979. A method for analyzing loop programs. *IEEE Trans. Softw. Eng. 5*, 3 (May), 237–247.

Waters, R. C. 1988. Program translation via abstraction and reimplementation. *IEEE Trans. Softw. Eng. 14*, 8 (Aug.), 1207–1228.

Weiser, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10*, 4 (July), 352–357.

Wills, L. M. 1990. Automated program recognition: A feasibility demonstration. *Artif. Intell. 45*, 1–2 (Sept.), 113–172.

WINANS, J. AND DAVIS, K. H. 1991. Software reverse engineering from a currently existing IMS database to an entity-relationship model. In *Entity-Relationship Approach: the Core of Conceptual Modelling, Proceedings of the Ninth International Conference*. Amsterdam, 333–348.

Received June 2002; revised February 2003 and July 2003; accepted September 2003