

Automatic High-Quality Reengineering of Database Programs by Temporal Abstraction

Yossi Cohen and Yishai A. Feldman*

Dept. of Computer Science

Tel Aviv University

69978 Tel Aviv, Israel

E-mail: {uniface, yishai}@math.tau.ac.il

http://www.math.tau.ac.il/~{uniface, yishai}

Abstract

The relational database model is currently the target of choice for the conversion of legacy software that uses older models (such as indexed-sequential, hierarchical, or network models). The relational model makes up for its lower efficiency by greater expressive power, and by optimization of queries, using indexes and other means. However, sophisticated analysis is required in order to take advantage of these features, since converting each database access operation separately does not use the greater expressive power of the target database and does not enable it to perform useful optimizations.

By analyzing the behavior of the host program around the database access operations, it is possible to discover patterns such as filtering, joins, and aggregative operations. It is then possible to remove those operations from the host program and re-implement them in the target database query language. This paper describes an automatic system, called MIDAS, that performs high-quality reengineering of legacy database programs in this way. The results of MIDAS were found to be superior to those of the naive one-to-one translation in terms of readability, size, speed, and network data traffic.

1 Introduction

Many legacy software applications still in use today are based on older database models, such as indexed-sequential, hierarchical, and network databases. A major goal in reengineering such applications is to change to more modern database technology, which in many 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, and the goal 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, and changing to object-oriented databases, are discussed at the end of the paper.)

The database-software translation problem consists of two main tasks: schema and data translation, and software translation. The former has received a lot of attention in the literature [1, 2, 3, 4, 5, 6], although it is the easier of the two in terms of the quantity of code involved and the difficulty of the task itself. This paper concerns itself with the latter task. Also, we will assume in this paper that the source program is written for a network database, since the network model is the most complex of the older models. The same methods can be applied to the other legacy database models.

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, since 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.

However, the one-to-one translation method suffers from the worst of both worlds. It translates a program written for an efficient database model to use a less efficient one. The relational model makes up for its lower efficiency by greater expressive power, and by sophisticated optimization of queries, using indexes and other means. One-to-one translation does not take advantage of the greater expressive power, and does not enable the rela-

* Author's current address: School of Computer & Media Sciences, Interdisciplinary Center, Herzliya, P.O. Box 167, Herzliya 46150, Israel.

tional database to make effective optimizations, since each data record is retrieved by a separate query, and all data processing is done by the host-language program.

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 configurations.

We have developed an automatic translation system, called MIDAS,¹ that performs the kind of analysis and translation described above. It uses temporal abstraction to discover database access patterns in the host program and translate them to relational-database operations whenever possible. The rest of the paper describes this system and our experience with it.

2 Example

The two translation methods will be illustrated by means of an excerpt from a program written for a network database. The example is based on a hypothetical university database that contains student information. The program fragment in Fig. 1 computes and displays the average grades of the best graduate students, defined as those whose grade average is over 95%. (The `FETCH` commands are the database-access interface.)

The first part of the program (the paragraph labeled `ALL-DEPT-STUDENTS`) fetches all the student records within the given department. Those records not belonging to graduate students are ignored; the rest are processed further. The second paragraph (labeled `SUM-STUDENT-GRADES`) is called repeatedly to fetch all the course grades for each graduate student, count them into 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 statistics are displayed.

The one-to-one translation is shown in Fig. 2. This program declares two *cursors* , which are SQL queries whose results can be accessed by the program record-by-record. The first cursor fetches all student records for students in a given department, and the second fetches all grades belonging to a given student. The database access commands

```

1  ALL-DEPT-STUDENTS .
2      MOVE 0 TO STATUS1
3      PERFORM UNTIL STATUS1 IS NOT EQUAL TO ZERO
4          FETCH NEXT STUDENT WITHIN DEPT-OF-STUDENT
5              AT END MOVE 1 TO STATUS1
6          IF STATUS1 IS EQUAL TO 0 THEN
7              IF STUDENT-DEGREE IS EQUAL TO 2 THEN
8                  MOVE 0 TO GRADES-SUM
9                  MOVE 0 TO GRADES-COUNT
10                 PERFORM SUM-STUDENT-GRADES
11                 DIVIDE GRADES-SUM BY GRADES-COUNT
12                     GIVING GRADES-AVG
13                 IF GRADES-AVG > 95 THEN
14                     DISPLAY STUDENT-ID, STUDENT-FIRST-NAME,
15                         STUDENT-LAST-NAME, GRADES-AVG
16                 END-IF
17             END-IF
18         END-IF
19     END-PERFORM .

20 SUM-STUDENT-GRADES .

21     MOVE 0 TO STATUS2
22     PERFORM UNTIL STATUS2 IS NOT EQUAL TO ZERO
23         FETCH NEXT GRADES WITHIN STUDENT-OF-GRADES
24             AT END MOVE 1 TO STATUS2
25         IF STATUS2 IS EQUAL TO 0 THEN
26             ADD GRD-GRADE TO GRADES-SUM
27             ADD 1 TO GRADES-COUNT
28         END-IF
29     END-PERFORM .

```

Figure 1: Excerpt from original network-database program.

have been replaced, but there are no differences in the computation that is performed by the host program.

The translation produced by MIDAS is shown in Fig. 3. It employs a single cursor, which contains almost all of the computational content of the original program fragment. The only part that cannot be performed by the database is the display of the results, and this part remains in the host program. (Had it been possible to perform all the computation in the database, the cursor would have been replaced by an SQL query for immediate execution.) In particular, the following operations have been folded into the query:

- filtering out non-graduate students;
- 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. The folding process proceeds in several stages, which are described in the next

¹For Migrator of Database Application Systems.

```

1 EXEC SQL DECLARE CRS1 CURSOR FOR
2   SELECT STUDENT-ID, FIRST-NAME,
3     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,
10    SEMESTER, GRADE
11  FROM GRADES
12  WHERE STUDENT-ID = :STUDENT-ID
13 END-EXEC.
14 ALL-DEPT-STUDENTS.
15 MOVE 0 TO STATUS1
16 EXEC SQL OPEN CRS1 END-EXEC
17 PERFORM UNTIL STATUS1 IS NOT EQUAL TO 0
18   EXEC SQL FETCH CRS1
19     INTO :STUDENT-ID, :STUDENT-FIRST-NAME,
20         :STUDENT-LAST-NAME, :STUDENT-ADDRESS,
21         :STUDENT-PHONE, :STUDENT-AGE,
22         :STUDENT-DEPT-NAME, :STUDENT-DEGREE
23 END-EXEC.
24 IF SQL-STATUS = SQL-NOT-FOUND
25   THEN MOVE 1 TO STATUS1.
26 IF STATUS1 IS EQUAL TO 0 THEN
27   IF STUDENT-DEGREE IS EQUAL TO 2 THEN
28     MOVE 0 TO GRADES-SUM
29     MOVE 0 TO GRADES-COUNT
30     PERFORM SUM-STUDENT-GRADES
31     DIVIDE GRADES-SUM INTO GRADES-COUNT
32       GIVING GRADES-AVG
33     IF GRADES-AVG > 95 THEN
34       DISPLAY STUDENT-ID, STUDENT-FIRST-NAME,
35         STUDENT-LAST-NAME, GRADES-AVG
36     END-IF
37   END-IF
38 END-IF
39 END-PERFORM.
40 EXEC SQL CLOSE CRS1 END-EXEC.
41 SUM-STUDENT-GRADES.
42 MOVE 0 TO STATUS2
43 EXEC SQL OPEN CRS2 END-EXEC.
44 PERFORM UNTIL STATUS2 IS NOT EQUAL TO 0
45   EXEC SQL FETCH CRS2 INTO :GRD-STUDENT-ID,
46     :GRD-COURSE, :GRD-YEAR, :GRD-SEMESTER,
47     :GRD-GRADE
48 END-EXEC.
49 IF SQL-STATUS = SQL-NOT-FOUND
50   THEN MOVE 1 TO STATUS2.
51 IF STATUS2 IS EQUAL TO 0 THEN
52   ADD GRD-GRADE TO GRADES-SUM
53   ADD 1 TO GRADES-COUNT
54 END-IF
55 END-PERFORM.
56 EXEC SQL CLOSE CRS2 END-EXEC.

```

Figure 2: One-to-one translation of the example.

```

1 EXEC SQL DECLARE CRS1 CURSOR FOR
2   SELECT STUDENT.STUDENT-ID, FIRST-NAME,
3     LAST-NAME, AVG(GRADE)
4   FROM STUDENT, GRADES
5   WHERE DEGREE = 2
6     AND DEPT-NAME = :DEPT-NAME
7     AND GRADES.STUDENT-ID =
8     STUDENT.STUDENT-ID
9   GROUP BY STUDENT.STUDENT-ID, FIRST-NAME,
10    LAST-NAME
11   HAVING AVG(GRADE) > 95
12 END-EXEC.
13 ALL-DEPT-STUDENTS.
14 PERFORM UNTIL SQL-STATUS = SQL-NOT-FOUND
15   EXEC SQL FETCH CRS1
16     INTO :STUDENT-ID, :STUDENT-FIRST-NAME,
17         :STUDENT-LAST-NAME, :GRADES-AVG
18 END-EXEC.
19   DISPLAY STUDENT-ID, STUDENT-FIRST-NAME,
20     STUDENT-LAST-NAME, GRADES-AVG
21 END-PERFORM

```

Figure 3: Example translated by abstraction.

section. 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 is comparable to what a human programmer would produce.

3 The MIDAS Touch

MIDAS is based on the idea of translation by abstraction, transformation, and re-implementation (see Fig. 4), first described by Waters [7], and later extended and validated in a large-scale experiment by Feldman and Friedman [8, 9]. 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 variations of coding. In the case of MIDAS, this representation is based on the Plan Calculus [10, 11], extended with a formalism called *Query Graphs* for describing database operations.

After the abstraction step, the program is further analyzed and transformed, using the abstract representation. In MIDAS, the analysis discovers temporal abstractions in the program and replaces them by explicit representations of the temporal operators. When all temporal abstractions have been found, the program is re-implemented in the target language. Those temporal abstractions that can be expressed in the target database query language (such as filters, joins, and aggregation, in the case of SQL) are removed from the host language and implemented by the

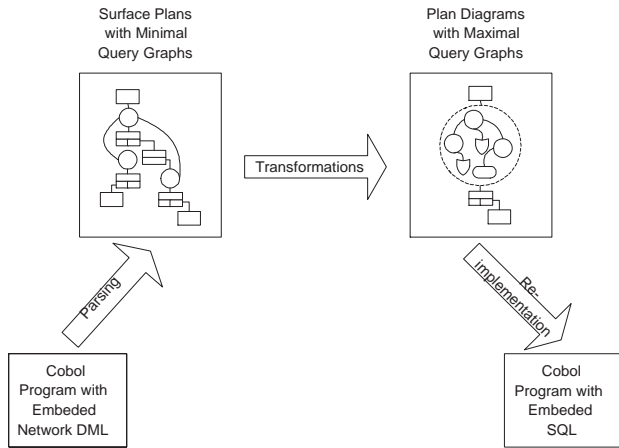


Figure 4: Translation by abstraction, transformation, and re-implementation.

database query language. The other parts of the program are modified as little as possible.

3.1 Query Graphs

Query graphs represent database operations at all levels of detail, from single-record retrievals to complex SQL queries. It is a wide-spectrum formalism, based on Relational Algebra and on Entity-Relationship Diagrams, and can describe the database operations of all models of interest. Figure 5 shows the query graph representing the SQL query in Fig. 3. This query graph is part of the representation of the program at the end of the transformation stage.

Atomic query graphs represent direct queries to the network database, each corresponding to a single `FETCH` statement in the original program. Database arcs may connect these query graphs to other query graphs; such arcs represent the database owner/member links between the records. Query graphs may be composed to form complex query graphs by means of operations such as filtering and aggregation. Space constraints do not allow a full description of this formalism here.

Each query graph represents a database operation that can result in a single record or in multiple records. Network-database operations are of the former kind; as they are merged together during the temporal abstraction process, they may represent more complex queries that fetch multiple records and apply to them operations such as filtering and aggregation.

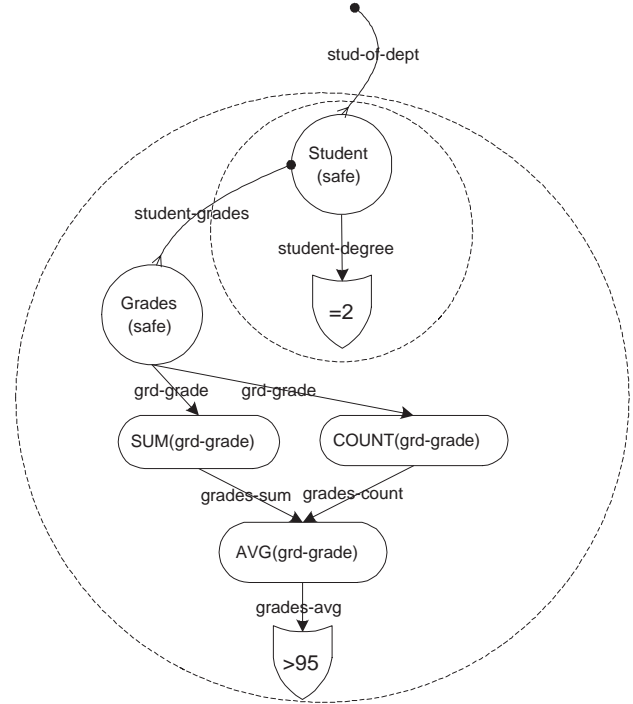


Figure 5: Abstract representation of final program.

3.2 Abstraction

The first step in the translation consists of the abstraction of the given program into an internal representation similar to surface plans [10, 12]. The techniques employed are based on data-flow and control-flow analysis, and are similar to those used by Wills [13] and by Feldman and Friedman [8, 9]. At the end of this step, much of the syntactic variability of the program has been eliminated. Each database access in the original program is represented as a separate query graph. Re-implementing the result of this step to use a relational database will yield the one-to-one translation shown in Fig. 2.

3.3 Transformation

The transformation stage is the most interesting in the translation process. At this stage, MIDAS identifies the patterns surrounding the query graphs in the abstract representation, and replaces them with equivalent but higher-level abstractions. Typically, such a transformation will reduce the parts of the program represented using the Plan Calculus, and will enlarge and combine the parts represented by query graphs. At the end of this stage, the query graphs in the representation will be as large as possible, al-

lowing the re-implementation stage to translate as much of the original program into SQL as possible.

The following discussion concentrates on database queries and ignores other operations, such as add, update, and delete. Database modification operations can be viewed as the application of an operation to the result of a query. The query part is treated and optimized like any other database query; there is little that can be done to improve the modification operation itself.

Following is a representative list of transformations that can be applied in this stage, in roughly chronological order of applicability.

3.3.1 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.

This transformation is used in the example program to remove the two `AT END` clauses and the associated flags and control-flow constructs.

3.3.2 Join Up

The *join up* transformation identifies the cases in which the network-database owner of a previously fetched record is retrieved. This transformation merges the two query graphs and annotates the new link by the type of relationship between the two records. This query graph may later be implemented as a join between two tables in the relational database.

An example for the use of the join-up transformation is the identification of the relationship between a student record and the record of the student's department (implemented in a separate table). This appears in the example program, though not in the fragment excerpted above.

3.3.3 Join Down

The *join down* transformation is similar to join up, except that it deals with the case in which the retrieval direction is reversed, that is, multiple records belonging to the same owner are fetched. (Unlike the case in the join-up transformation, the control environments of the retrieval operations on the owner and member records are different.) In this case as well, the two query graphs are merged, and

may later be re-implemented as a relational join or as a sub-query.

This transformation merges the loop in the `SUM-STUDENT-GRADES` paragraph of the original program into the SQL query in the final version.

Both join up and join down are specializations of *cartesian product*, in which the two query graphs are not necessarily related by owner-member links. These cases are also re-implemented as joins or sub-queries. These transformations also capture the intersection and subtraction idioms.

3.3.4 Accumulation

Temporal abstraction [14] views a sequence of values processed in a loop as an object in its own right. Operations applied to the values in the loop are abstracted into operations on the sequence. In particular, a binary operation applied to a fixed variable with each value in the sequence (such as `ADD GRD-GRADE TO GRADES-SUM`, line 26 of Fig. 1) is abstracted into an accumulation operation, which takes a sequence and produces a single value. The *accumulation* transformation recognizes such cases, and replaces the part of the loop that applies the binary operator by an accumulation operation on the sequence of values represented by the query graph.

The example program contains two cases of accumulation: the computation of the sum of the student's grades, and counting those grades (this is a degenerate case of accumulation in which the values of the sequence elements are not used). A later 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 re-implemented as the SQL aggregative operation `AVG (GRADE)`.

3.3.5 Computed Expressions

Accumulation adds a new field to the result of the query. Relational queries can also add fields containing the results of non-aggregative computations. The *computed expressions* transformation transfers such computations from the plan representing the original Cobol program into the query graphs. Such new fields can serve as the basis for further transformations, such as accumulation or filtering.

3.3.6 Filtering

Another temporal operator is a filter, which takes a sequence and a predicate, and returns the sub-sequence containing those elements that satisfy the predicate. When the *filtering* transformation finds that only elements satisfying a test in the original program are further processed, it adds

the filtering predicate to the query graph and removes it from the representation of the program.

This transformation is applied twice to the example program. It replaces the test `IF STUDENT-DEGREE IS EQUAL TO 2` in the original program (Fig. 1, line 7) by the SQL clause `WHERE DEGREE = 2` (Fig. 3, line 5). It also replaces the test `IF GRADES-AVG > 95` (Fig. 1, line 13) by the SQL clause `HAVING AVG(GRADE) > 95`.²

3.4 Re-implementation

The re-implementation stage is relatively straightforward. It translates those parts of the final output of the transformation stage that are still expressed as plans back into Cobol. It attempts to translate the query graphs into SQL. This is not always possible, because of limitations of the query language; the portions that are not expressible in SQL are also implemented in the host language. Sometimes there is more than one way to translate a query graph into SQL (for example, it may be possible to use a subquery instead of a join). These choices are made during the re-implementation stage, which is the only stage that has detailed knowledge about the target language.

Several heuristics are used in order to produce a readable and understandable program. For example, information is kept about the structure of the original program, and this structure is duplicated as much as possible. Also, cursors are used only when multiple records must be processed by the host program rather than by the database (as in our example, where the records are displayed by the host program).

As part of the re-implementation, unused variables and dead code are eliminated. The network `FETCH` commands bring all fields of the record to memory, even though many of them may be ignored by the program. The SQL queries produced by MIDAS contain only those fields that are actually needed. An example of dead-code elimination is the omission of the `COUNT` and `SUM` aggregations from the final SQL query, since these are only used to compute the average.

Another optimization is redundant-join elimination. When a set of records is accessed through a chain of owner-member links, as when courses are grouped under semesters grouped under students, multiple joins are generated. If in the translated schema, the student-id appears in the courses table, and none of the fields of the semester table are referenced, the semester table is omitted from the join.

²The reason for the syntactic variation in the two SQL clauses is that the second case applies to the average of the grades of a single student, rather than to each grade separately. This is also the reason for the existence of the `GROUP BY` clause.

Table 1: Performance comparison

	Time (sec.)	Data traffic (bytes)
One-to-one translation	131	414,000
MIDAS translation	113	738
Reduction	14%	99.8%

The re-implementation stage is the only stage in which the details of the schema conversion are relevant. The analysis of the original program in terms of query graphs refers only to the original database schema. During re-implementation, the queries are written for the new schema. 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, in our case, algebraic simplification of the union query will eliminate the subquery that refers to the table containing the data of undergraduate students. Then, 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 such as splitting fields between tables, joining previously separate tables, or joining several fields into a single one.³

4 Conclusions

4.1 Results

Table 1 shows a comparison between the one-to-one translation and the program produced by MIDAS.⁴ The results show that MIDAS improves on the one-to-one translation in terms of speed as well as data traffic, which is particularly important in client-server configurations. (Of course, the data traffic results depend on the input data; however, many programs correspond to queries that produce a small number of records from a large database, and in such cases the results can be even better than in our example.) Also, the resulting code is shorter and more readable, and thus enhances maintainability.

The relational database framework is based on operations on tables rather than individual records. This corre-

³A case in point is the combination of `DAY`, `MONTH`, and `YEAR` fields into a `DATE` field.

⁴We did not have access to a network database in order to run the original program.

sponds well to the temporal-abstraction view, which considers sequences of records as atomic units. There is a natural description of all SQL operators in terms of temporal abstractions, and it is therefore sufficient to recognize those abstractions in the original program in order to translate it into SQL, making full use of its power. There is a small number of temporal abstractions (and corresponding SQL operators), and therefore only a small and fixed number of database transformations are required in order to find them.

The translation algorithm itself is quite efficient. The application of each transformation reduces the plan representation of the original program, and thus only a linear number of transformations can be applied. As mentioned above, the number of transformations is small and fixed. The pattern-matching process starts at the query graphs in the original representation, and tries to incorporate parts of the surrounding plan into the query graph. Those parts of the original program that are not related to the database need not be examined at all. For these reasons, we expect the process to scale-up well in practice.

4.2 Other Work

In 1982, Katz and Wong [15] described an algorithm for translating network-database programs into relational queries. Their algorithm is similar to the one-to-one translation described above, except that it is capable of merging an access path consisting of several links into a single join query if no other computation intervenes. Since their algorithm does not recognize other abstractions, such as filters, code implementing such abstractions will prevent their algorithm from using a join.

More recently, a group at Lockheed [16] has implemented a tool to help in reengineering code written for IMS (a hierarchical database) to the relational model. This tool analyzes the source program but requires human intervention in the conversion process. The authors of that paper recognize that one-to-one translation is inadequate, and describe a component that finds programming idioms in order to provide better translation for 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, we claim, as mentioned above, that there is a small and fixed number of such patterns that suffices for

high-quality translation to SQL, and these patterns are easily found using temporal abstraction.

MIDAS relies on several AI tools and techniques. These are:

- *The Plan Calculus*: Originally developed as the internal representation for the Programmer's Apprentice [11], it has been extended with query graphs to form the basis for the abstract representation of the code in MIDAS.
- *Translation by abstraction and re-implementation*: This idea was presented in a theoretical framework by Waters [7]. It has been extended and demonstrated in a large-scale experiment by Feldman and Friedman [8, 9].
- *Program transformations*: Typically performed on textual representations of code, they are more natural and effective when performed on a more abstract representation. This step was added between the abstraction and re-implementation steps in Waters' model, in order to bring the program closer to the conceptual model of the target language.
- *Temporal abstraction*: Waters [14] found that about 90% of the loops in Fortran programs in the IBM Scientific Subroutine Library could be represented very simply by temporal abstraction. In the translation to a relational database, temporal abstraction is the most natural representation tool, since it closely matches the operators of the target language.

4.3 Discussion

MIDAS demonstrates that it is possible to perform high-quality conversion of legacy software from an old database model to a new one using a small and fixed set of transformations. These correspond to the stronger operators available in the new database. However, it is not necessary to achieve a high-level of program understanding that is based on a large library of clichés [13], 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 easily be accommodated. In fact, 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. The 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-driven model. The query graphs constructed by MIDAS from the original program form a sound basis for the data analysis of the program. The identification of events and triggers in the host program is an interesting problem, which we hope to address in future.

While this paper 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 re-implementation phases need not be changed. If the target database model is object-oriented, only the re-implementation 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 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 be a focal point around which to base such an analysis.

References

- [1] J. Fong and C. Bloor, "Data conversion rules from network to relational databases," *Information and Software Technology*, vol. 36, pp. 141–153, Mar. 1994.
- [2] J. S. P. Fong, "Methodology for schema translation from hierarchical or network into relational," *Information and Software Technology*, vol. 34, pp. 159–174, Mar. 1992.
- [3] M. L. Gillenson, "Physical design equivalencies in database conversion," *Comm. ACM*, vol. 33, pp. 120–131, Aug. 1990.
- [4] D. L. Spooner, D. Sanderson, and G. Charalambous, "A data translation tool for engineering systems," in *Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, (Los Alamitos, CA), pp. 96–104, Oct. 1989.
- [5] F. Tangorra and D. Chiarolla, "A methodology for reverse engineering hierarchical databases," *Information and Software Technology*, vol. 37, pp. 225–231, Apr. 1995.
- [6] J. Winans and K. H. Davis, "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), pp. 333–348, Oct. 1991.
- [7] R. C. Waters, "Program translation via abstraction and reimplementation," *IEEE Trans. Software Engineering*, vol. 14, pp. 1207–1228, Aug. 1988.
- [8] Y. A. Feldman and D. A. Friedman, "Portability by automatic translation: A large-scale case study," in *Proc. Tenth Knowledge-Based Software Engineering Conf.*, (Boston), pp. 123–130, Nov. 1995.
- [9] Y. A. Feldman and D. A. Friedman, "Portability by automatic translation: A large-scale case study," *Artificial Intelligence*, to appear.
- [10] C. Rich, "A formal representation for plans in the Programmer's Apprentice," in *Proc. 7th Int. Joint Conf. Artificial Intelligence*, (Vancouver, British Columbia, Canada), pp. 1044–1052, Aug. 1981. 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.
- [11] C. Rich and R. C. Waters, *The Programmer's Apprentice*. Addison-Wesley, Reading, MA, and ACM Press, Baltimore, MD, 1990.
- [12] L. M. Wills, "Automated program recognition by graph parsing," Technical Report 1358, MIT Artificial Intelligence Lab., July 1992. PhD thesis.
- [13] L. M. Wills, "Automated program recognition: A feasibility demonstration," *Artificial Intelligence*, vol. 45, pp. 113–172, Sept. 1990.
- [14] R. C. Waters, "A method for analyzing loop programs," *IEEE Trans. Software Engineering*, vol. 5, pp. 237–247, May 1979.
- [15] R. H. Katz and E. Wong, "Decompiling CODASYL DML into relational queries," *ACM Trans. Database Systems*, vol. 7, pp. 1–23, Mar. 1982.
- [16] W. Polak, L. D. Nelson, and T. W. Bickmore, "Reengineering IMS databases to relational systems," in *Seventh Annual Software Technology Conference*, (Salt Lake City), Apr. 1995. Published on CD-ROM.