The Transition of Application Programs From COBOL to a Fourth Generation Language

Spencer Rugaber Srinivas Doddapaneni

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

It is becoming increasingly desirable to move older application programs from their traditional mainframe execution environments to networked workstations. These management information systems are most often written in COBOL and store their data in files. A networked environment enables the use of a relational database management system and its fourth generation access language, SQL. A conceptual framework is described that comprises a variety of strategies for making such a transition. Decision criteria for selecting among them are then presented. Finally, a variety of experiments intended to explore the strategies are recounted. The experiments include efforts to automate parts of the process.

1 Motivation and Background

The advantages of moving existing COBOL mainframe management information systems to a fourth generation language (4GL) environment based on a relational database management system (RDBMS) are both economic and technical.

- Distributed access. The data and procedures provided by an RDBMS application are available to other programs and people. The programs and people may be geographically distributed.
- Transportability. An RDBMS typically supports the standard database query language, SQL. This enhances vendor and platform independence, thereby increasing flexibility.
- Database features. An RDBMS supports a variety of services that are not likely to be available in a standalone application program. Typical features include security, data integrity, rollback and crash recovery, and locking.
- Increased abstractness. Many application features present in COBOL programs can be described declaratively or avoided altogether when using a 4GL. For example, an RDBMS requires explicit models of the data it contains. Some of this information is available as file descriptions in the COBOL code, but it is often very concrete, and more conceptual aspects of the data are present only implicitly in the procedural code. Moreover, once data have been moved from files into databases, enterprise-wide data integration can be addressed.
- An RDBMS also provides a variety of functions that have to be coded explicitly in a COBOL program. For example, a forms-based, data entry tool can replace much data entry and validation code. Likewise, a report generator can replace code that explicitly formats output reports, and query operations can replace complex procedural code.
- Improved maintainability. Estimates indicate that several hundred billion lines of source code exist in the world and that seventy percent of it is written in COBOL[20].

  The largest cost factor in software development is maintaining existing programs after they are delivered. Up to 75% of life cycle cost may be expended on such maintenance. Moreover, maintenance costs are directly related to the amount of code being maintained[3]. In section 3.4 we show that a 4GL program can be significantly smaller than the corresponding COBOL program. Hence, it should be much less expensive to maintain.

- Simplified use. Data for application programs can be entered and validated on personal computers much more quickly and cheaply than in a batch processing environment on a mainframe. An experiment to support this assertion is described in section 3.5.

1.1 Army Management Information Systems

To investigate these issues, we have undertaken a series of experiments. In particular, we have examined U. S. Army *STAMIS*s. A STAMIS is a **ST**andard **A**rmy **M**anagement **I**nformation **S**ystem. Of the 187 STAMISs listed in the STAMOD database, 118 were written at least partially in COBOL. When all systems and variants are considered, the average size of a STAMIS is 206 thousand lines of code. Each such system is broken up into separate programs. The average number of programs per system is 161. Of the original systems, 78 already access a database of some sort. These databases range from large-scale, third generation systems like DMS1100 to personal computer record managers like DBase. Some systems already access an RDBMS such as Oracle or Informix. In summary, a typical STAMIS is written in COBOL for execution on a mainframe computer as a batch process, accessing files rather than a database.

1.2 A Typical Army Management Information System

For the purposes of this project, we needed to explore the difficulties that arise when actually transiting an application. To do this, we used the Installation Materiel Condition Status Reporting System (IMCSRS)[1]. This STAMIS consists of approximately 10,000 lines of COBOL code, broken into 15 programs. We had become familiar with this system during an earlier project[18] that ultimately lead to the replacement of IMCSRS by a version written in Ada[8, 16].

IMCSRS is smaller than a typical STAMIS, but it performs typical functions. It is responsible for using input transactions to update a master file and then producing a variety of reports describing the status of Army materiel. We used it to examine the hypothesis that most of its functionality can be replaced by RDBMS functions expressed in a 4GL.

2 Conceptual Framework

2.1 Strategies Considered

There are a variety of approaches to moving a COBOL information system into an environment supporting distributed access. This section describes those that we have examined. They are organized roughly from those requiring the least to those requiring the most effort to effect. The strategies are comparable in the sense that each has advantages and disadvantages. For any given situation, the costs and benefits of the various strategies must be compared to select the most effective approach.

2.1.1 As-Is Strategy

The base line against which the other strategies must be measured is the strategy of doing nothing. In this case, there are no real advantages, and the disadvantages are fairly well understood. This strategy may be appropriate if it is known that the application is going to be replaced or phased out or if it is used only infrequently by a single site. In these cases, there is little value in investing in 4GL access.

2.1.2 Direct SQL Access to File Data Strategy

Another strategy leaves both the COBOL program and the data files untouched. This strategy involves the development or acquisition of a tool that provides SQL access to files. New queries can be written directly in SQL without requiring COBOL code alteration. The strategy also supports the incremental replacement of COBOL functionality by SQL queries. This may lead to a hybrid environment, where some programs are written in COBOL and some in SQL and some data reside in files and some in relational database tables.

2.1.3 Direct Porting Strategy

Another strategy is to simply port an existing COBOL system from the mainframe environment onto a networked workstation without adding any new functionality, thereby increasing its accessibility. This is a traditional adaptive maintenance approach. Our experience with this strategy is described in section 3.1. Direct porting may be desirable if execution costs are significantly reduced by using a workstation or if the network access provided by the workstation increases the customer base and timeliness of the application's reports. It may also be applicable as an interim step to some of the other strategies described below.

2.1.4 Transparent Layered Conversion Strategy

Some vendors are already addressing the software transition problem. For example, Liant Software sells a tool called RM/plusDB.[1] Its purpose is to provide a transparent mechanism so that existing COBOL programs can access an RDBMS without having to be altered. RM/plusDB provides an extended run-time environment and a server. The run-time extensions are invoked when a COBOL statement attempts to do I/O to a file. The run-time routines intervene and convert the I/O request to an access to the RDBMS. The server transfers these requests to the RDBMS and passes back any returned data.

This strategy provides some of the sought-after benefits. All of the advantages of DBMSs over files, such as data security and integrity, are available. Also, enhancements are facilitated–new reports can be easily constructed using the 4GL capabilities of the RDBMS. Because the code is not altered, the cost is low. This strategy can also be applied as an interim step to those described below. Disadvantages include the fact that the resultant data organization is naive. It will not contain the conceptual abstractions that would be present if a complete database design had been undertaken. There may also be degraded performance due to the presence of the database.

2.1.5 Code Layering Strategy

The previous strategies have avoided altering the source code of an application system. Sometimes, however, the benefits of direct intervention are warranted. Most database vendors provide a mechanism for directly placing SQL statements into source code. This is accomplished using either a preprocessor or through direct library calls. We did not try any specific experiments with this strategy, but some of the observations related in the next subsection are relevant here.

2.1.6 Code Replacement Strategy

4GL programs are smaller and more maintainable than are programs written in COBOL. These benefits can be a strong inducement to replace parts or all of an application program by one or more 4GL programs. Specifically, 4GLs support the construction of reports and the satisfying of relatively small queries that do not require a great deal of computation. Also, depending on the specific interactions involved, the *view* and *join* capabilities of an RDBMS can replace some complicated computations involving multiple files. In addition, some 4GLs provide declarative data validation mechanisms that can further reduce code size. An experiment to explore this possibility is described in section 3.2.

This strategy is applicable in situations where incremental transition from COBOL to SQL is desirable. This may be warranted when the programmers are receiving on-the-job training and need small examples. The feedback obtained from comparing the existing results with that obtained from SQL can serve to validate the conversion. If the program being converted is convoluted and difficult to understand, incrementally removing segments responsible for relatively self-contained tasks, such as report generation, can reduce the size of the remaining COBOL code to the point where direct comprehension is more feasible.

2.1.7 RDBMS Conversion by Re-engineering Strategy

Program evolution without benefit of a high-level representation of functionality and structure presents risks in terms of quality. The process of reverse engineering existing software yields such a representation that can then be used as a basis for enhancements. The advantages of such an approach are obvious; the disadvantages are, however, difficult to measure. One factor that needs to be understood is that reverse engineering requires a significant commitment of time and effort. Some discussions of mechanisms for partially automating the process are described in the next subsection.

Manual re-engineering is indicated in situations where the existing code will continue to be used extensively for the foreseeable future. Maintenance activities that require modification of existing code (versus simply adding new modules) can also help justify the expense of reverse engineering. Reverse engineering does not have to be applied to an entire system[7]. Even if only a part of a system is being reverse engineered, there is still a need for the program maintainer to understand the context of the component relative to the entire system. Thus, in situations where resources such as accurate documentation or experienced maintenance personal exist, partial reverse engineering may be indicated.

[1] This particular tool is limited to programs accessing index sequential files only.

2.1.8 Automatic Reverse Engineering Strategy

Because of the expense involved in reverse engineering, it is desirable to automate as much as possible the steps involved. Unfortunately, the state of the art is such that few tools exist, and those that do are capable of describing only surface features of an existing system. The strategy of automatic reverse engineering involves extracting features from existing programs and translating them into a standard design representation. We performed an experiment, described in section 3.3, to explore the feasibility of this approach. Specifically, we investigated whether we could automatically extract information from a COBOL program describing the structure of the files that it uses.

2.1.9 From-Scratch Rewrite Strategy

A final strategy needs to be mentioned for reasons of completeness. Under some circumstances, it may be desirable to replace an existing program entirely and to rebuild it from scratch, including new requirements gathering. This situation may arise when the original system needs to be significantly modified and is complex enough that the cost of re-engineering outweighs the costs (and risks) of initial development.

2.2 Summary of Strategies

Figure 1 summarizes the strategies described above.

Strategy
As-Is Strategy
Direct SQL Access to File Data Strategy
Direct Porting Strategy
Transparent Layered Conversion Strategy
Code Layering Strategy
Code Replacement Strategy
RDBMS Conversion by Re-engineering Strategy
Automatic Reverse Engineering Strategy
From-Scratch Rewrite Strategy

Figure 1: Summary of Transition Strategies

2.3 Strategy Selection Criteria

No one strategy is suitable for dealing with all situations. In order to determine which strategy is appropriate in a given situation, the factors that can affect the costs and benefits of applying the strategy should be weighed. The factors serve as decision criteria, and Appendix I provides a sample list.

The various factors and strategies have an internal structure. That is, making one decision may naturally lead to a subsequent question being asked. Some decisions preclude others. To summarize the possibilities, the decisions can be organized into a hierarchical structure called a decision tree. A skeleton decision tree for the various transition strategies is shown in Figure 2. The numbering of the questions indicates their relative placement.

In order to use the decision tree, the criteria must be interpolated. There are two substantial impediments to doing this. The first concerns measurement. While some of the decision criteria are quantifiable ("average cost of an execution of the application program"), many of them are qualitative ("are the political factors ... ?"). In order for there to be a formal decision procedure, metrics for the qualitative factors need to be devised and validated.

The second impediment involves combining the criteria. Even if all criteria were quantitative, it would still not be meaningful to make a decision by combining numbers from several criteria. For example, how does the availability of a staff of maintainers experienced with a COBOL application trade off against a significant increase in system use due to networked availability?

Two approaches to dealing with these impediments are suggested: scenarios and case studies. A scenario is a hypothetical narrative describing a potential transition effort. It can be used to explore difficulties in applying the criteria. Scenarios have been effectively used to solicit system requirements during the early stages of software development[10], and their use here has a similar purpose, to unearth unanticipated costs and benefits. The second suggestion involves the use of case studies of previous transition efforts. In particular, data concerning costs involved, difficulties that arose, and eventual benefits can serve to guide the decision process.

3 Experiments

We have conducted a series of experiments to explore the transition strategies. We have used IMCSRS as our subject system and looked at four of the possible transition paths for it: directly porting it to a networked workstation environment (section 3.1), replacing part of it by a direct SQL query (section 3.2), automatically generating an SQL description of its input file structure (section 3.3), and replacing part of it by the use of an RDBMS report writer tool (section

Should the information system:

1. be left alone(As-Is Strategy)?
2. be moved into a 4GL environment?

2. In moving to a 4GL environment, should the information system:
  2.1. be ported as-is (Direct Porting Strategy)?
  2.2. be migrated to an RDBMS/4GL?

  2.2.A. If migration is desirable, should the new version:
    2.2.A.1. move the data to an RDBMS and leave the code alone to the extent possible (Transparent Layered Conversion Strategy)?
    2.2.A.2. provide the functionality with SQL access to files (Direct SQL Access to File Data Strategy)?
    2.2.A.3. replace selected features of the code with 4GL constructs?

    2.2.A.3. If code is going to be replaced, which combination of features should be replaced:
      2.2.A.3.1. File access replaced by embedded SQL and/or library calls (Code Layering Strategy)?
      2.2.A.3.2. Report formatting replaced by the 4GL report writer?
      2.2.A.3.3. Some reports replaced by SQL or forms-based queries?

      2.2.A.3.1. If file access is to be replaced, should it be accomplished:
        2.2.A.3.1.1. with a layering tool?
        2.2.A.3.1.2. by replacing COBOL I/O with embedded RDBMS access?

  2.2.B. If migration is desirable, should the new version:
    2.2.B.1. be developed from scratch, including requirements gathering (From-Scratch Rewrite Strategy)?
    2.2.B.2. be incrementally migrated from the current version (Code Replacement Strategy)?
    2.2.B.3. be the results of reverse engineering the existing version (RDBMS Conversion by Re-engineering Strategy)?

    2.2.B.3. If reverse engineering is used, should it be done:
      2.2.B.3.1. by hand?
      2.2.B.2.3. using semi-automated tools (Automatic Reverse Engineering Strategy)?

Figure 2: Skeleton Decision Structure

3.4). Section 3.5 reports on a related use of IMCSRS to reduce actual operational costs.

3.1 Direct Code Porting

The first experiment explores the Direct Porting strategy. That is, we took the mainframe COBOL program, IMCSRS, and tried to port it to run on a UNIX workstation. After converting from fixed length records to varying length records and stripping out line numbers and other records included for the benefit of the mainframe source code management system, we tried to compile it.

Unfortunately, IMCSRS is written in an older dialect of COBOL, COBOL 68. The compiler that we used is capable of compiling only the COBOL 74 and COBOL 85 dialects. In order to proceed, we had to convert the COBOL code by hand. This was time consuming but relatively straightforward. The conversion took the form of systematic transformations to the code, many of which are potentially automatable. For example, some COBOL 68 statements (such as "EJECT") are not supported in COBOL 85, and these can simply be deleted; some keywords have been replaced ("EXAMINE" becomes "INSPECT"); the way in which long string literals are handled has been changed; and some COBOL 68 registers are no longer supported. In general, an experienced COBOL programmer, using a screen editor that supports global search and replace operations, can make the transition in a straightforward way. A description of the complete set of transformations is available as an appendix to [6].

After the code was converted, it successfully compiled using the workstation compiler. In order to test the integrity of the conversion, we ran several of the programs and successfully obtained reports.

There are several factors related to the conversion of system data that our experiment did not consider. Issues such as word size, byte order, and character set may cause difficulties in applying the strategy. For example, would word size differences, such as might exist between a 36-bit mainframe and a 32-bit workstation, lead to loss of precision? We also did not consider the related modification of operational procedures. On the mainframe, IMCSRS consists of fifteen programs, some of which produce output that is consumed by others. On a workstation, command scripts would have to be written to support this structure.

Our conclusion from the experiment is that no conceptual difficulty prevents the porting of COBOL programs from the mainframe to the workstation, but a significant number of technical details make the process burdensome. And while running the COBOL program from the workstation does provide a limited form of distributed access, many of the advantages of an RDBMS are not provided.

3.2 Replacement of COBOL by SQL

Our second experiment applied the Code Replacement Strategy to convert one of the IMCSRS programs to SQL. In order to accomplish this, the original COBOL program had to be understood. Traditionally, this would be accomplished informally by trying to understand the program in terms of the target SQL functionality. Although this was the course that we took, our recommendation for subsequent efforts is to make this comprehension process more systematic. In particular, an explicit reverse engineering step is required in which a representation of the program's data processing requirements is constructed. Such a procedure targeted specifically at information systems is described in Batini's book[2]. Another approach, aimed somewhat more at the functional requirements than at data modeling is described in [13].

The converted program is responsible for generating a report on equipment availability. For this experiment, the steps involved in the conversion were manually applied. (The experiment described in section 3.3 explores how to automate part of this process.) The steps involved are the following.

- Create SQL schema definitions for two tables corresponding to the two input File Description statements (FDs) in the program. The schema definitions take the form of SQL *create* commands.
- Create sample data, and use the RDBMS loader to load it into the tables.
- Determine the data accesses and computations needed to produce the output report.
- Determine the format of the output report.
- Construct the SQL statements to retrieve the data and format it for the output report.

We note several things about this process. First, in the case of one of the two input files, the FD did not contain sufficient information to describe the structure of the file data. The specific FD as defined in the FILE SECTION of the program has a single field, 151 characters long. It is in the WORKING STORAGE section, however, where a storage area is defined that indicates that this field can be viewed as consisting of 25 smaller fields. The fact that the FD and storage area are related is not made explicit by COBOL syntax anywhere in the FILE SECTION or WORKING STORAGE section. In the procedural code, however, where a READ statement transfers the data from the input file to WORKING STORAGE, the connection can be made. Therefore, an automated tool will need to analyze the PROCEDURE DIVISION as well as the FILE SECTION.

The experiment was successful in the sense that an SQL report was generated from an RDBMS. Full details of this experiment are given in [6]. In all, a 650 line program was replaced by two *create* commands to define the schema for the input files, two *load* commands to load the data into the RDBMS, four formatting commands to produce page and column headings for the output report, and a *select* command to actually access the data. Note, however, that the appearance of the output report did not attempt to duplicate the appearance of the original report. Section 3.4 describes our use of a report writer to pursue this goal.

3.3 COBOL to CASE to RDBMS

Our most ambitious experiment concerned the automatic construction of SQL *create* commands from COBOL programs. In particular, we used a grammar-based, program analysis tool to extract FD statements from the COBOL source. The statements were then transformed into an Entity Relationship (ER) diagram[4] for the Software Through Pictures (STP) CASE tool. Then the STP schema generation feature was used to generate SQL *create* commands to define tables for holding the data.

A COBOL program is a highly structured description of computations and data. In order to construct a high-level design representation of a program, the design decisions that went into its development and maintenance must be reconstructed. This can be accomplished by a systematic analysis of the program text, simultaneously constructing a description of the application domain and procedures that the program models. A description of this process is given in [13].

Because of the structured nature of programming languages, any analysis must be based on a grammatical description of that language, i. e., a COBOL grammar. We used a grammar-based tool, called **NewYacc**[15], to annotate the grammar with rules. The rules were applied during traversals of the program parse tree in order to extract constructs from the program.

The rules indicate which program features to extract. In the case of IMCSRS, FD statements were extracted. These can be used to automate parts of the several other strategies. For example, in the Transparent Layered Conversion strategy, the FDs have been used to build a description of the tables in the RDBMS that hold the input and output data. Also, in the RDBMS Conversion by Re-engineering strategy, they can help construct a high-level representation of the data manipulation requirements of the program.

STP has an open architecture. This means that it is extensible in a variety of ways. In particular, diagrams are represented textually, and the format of the representation is documented. Normally, an STP user manually selects icons and places them in a diagram on the screen. Using the published file format, however, we were able to automatically construct diagrams based on the information extracted by **NewYacc**[11, 12].

The program visualization provided by this approach is limited in several ways. First, STP provides no mechanism for refining an ER diagram into lower level diagrams. This is not a fundamental limitation. The Batini book[2], for example, describes several ways in which complex diagrams can be abstracted. The second limitation has to do with *relationships*. An ER diagram consists of entities and relationships. In our case, the entities correspond to files, and they can be automatically extracted using the procedure described above. Relationships, however, are more troublesome. Relationships between an input and an output file can be arbitrarily complex, and many relationships are only implicitly apparent in the procedural code.

Once an ER diagram has been built by the automatic procedure, it is editable just as if the diagram had been drawn initially from within STP. Moreover, the diagram supports future enhancements to the information system. For example, if a program enhancement adds new information to an input file, the corresponding ER diagram can be extracted and edited. Then, the CASE tool's template generation mechanism can be used to automatically produce a new version of the FD statement describing the file. In a sense, maintenance has been moved from a code editing activity to a conceptual alteration expressed graphically. A more detailed description of the experiment is provided in [6].

3.4 Replacement of COBOL by a Report Writer

Our most recent experiment involved reengineering a report generation application, which was originally written in COBOL, using Oracle's SQL*ReportWriter. The program we chose was the same one used in the previous experiment. The report it generates is a structured document consisting of groups corresponding to classes of equipment. Each group contains detailed information on each type of equipment in that class. Each page in the report is printed with page headings consisting of title, date, station name, etc. We reverse engineered the program to extract its specification. With the aid of this information and a sample output page, the program computations were extracted.

In section 3.1, we described a direct use of SQL. That experiment produced the same output data as the original report, but the format of the data was not nearly so well organized. We found that the complex structured query needed for duplicating the original report can not be directly specified in SQL. In particular, there is no facility for interleaving the results of the SQL query, which extracts data on the type of equipment in each class, with the results of the query for extracting data on class of equipment.

An alternative is to use a report writer. In this experiment we reimplemented the program using the Oracle RDBMS, SQL, and various Oracle tools such as SQL*ReportWriter, SQL*Menu and SQL*Forms.

As a result of reverse engineering the program, we obtained a description of its input files. We constructed a table corresponding to each file. For a simple collection of files the implementation of each file by a relational table works well. But for a more complex set of files, it will be necessary to do a database design. There is a facility in Oracle's CASE*Dictionary tool to do automatic generation of a default database design that might prove useful. CASE*Dictionary can be populated with a data model for the problem through Oracle's ER graphical editor.

A report is first specified in SQL*ReportWriter through its interactive forms-based interface. Then the report can be customized by providing the specifications for page headings and footings, group headings and footings, field labels, page size, page margins, and the placement of groups and fields relative to each other.

The generated report contains several values computed from the fields of the relational tables. Since these computations are complex, we found it necessary to specify various intermediate views on the tables. This simplified the query specification in SQL*ReportWriter. Two views are used to compute the sums of certain columns. These sums are then used in other views to compute values for the target report. The views are expressed in SQL*Plus, Oracle's extension to SQL.

The COBOL implementation of the program consisted of 650 lines of source code. The implementation of the report in SQL*ReportWriter consisted of 100 lines of SQL for table and view definitions, and 35 lines of SQL*ReportWriter specification. We found that SQL*ReportWriter is also very easy to use. A report specified in it can be altered in minutes and the results immediately observed. We believe the resulting implementation is also easy to modify and maintain.

3.5 Economic Justification

One related "experiment" should be reported on. Our original examination of IMCSRS was part of an exercise to determine the issues involved in translating a COBOL program to Ada[9]. The project involved reverse engineering IMCSRS and then constructing two designs, one using functional decomposition and one using object-oriented design. The designs were compared and the object-oriented design selected for implementation. An Ada implementation was then built and analyzed. As part of the analysis, the system was installed in a site where it replaced use of the COBOL version.

The Ada version runs on a personal computer instead of a mainframe. Data can be validated locally without having to wait for physical transmission to and from the mainframe. Results were so positive that the Ada version has replaced the COBOL version as part of regular procedures at the site of the trial. As other sites learned of the improved performance, they requested and obtained the personal computer version[8, 16]. Subsequently, an economic analysis was performed to predict the saving that would result for using the Ada version over a ten year period[19]. The predicted results for this system alone were for a $3,000,000 saving. And this does not include any savings in maintenance cost due to the increased encapsulation provided by the object-oriented design. The payback period for the re-engineering effort was projected to be less than six months.

In Section 1, we indicated that one advantage of moving an application off of a mainframe is simplified use. With the Ada version, improved availability provided by the personal computers was one of the largest contributing factors to the projected cost savings. The same improvement should obtain when the target language is a 4GL rather that Ada.

4 Recommendations for Future Work

Some of the questions raised by our explorations present interesting research possibilities.

- Relationship detection. Our investigations of the automatic construction of ER diagrams proved successful at describing those entities that correspond to program files. More difficult is the question of detecting relationships in programs. Some work in this direction is described in [5, 14]. We would like to use techniques from compiler data flow analysis to pursue this goal.
- Other types of applications. The experiment described in section 3.4 was limited to a program solely concerned with writing a report. Other types of applications will no doubt be more difficult to deal with. Candidate program types include replacing small programs with direct SQL queries (in a sense, decompiling) and replacing data editing and validation programs with the use of an RDBMS forms-based data entry tool.
- Transformation abstraction. Our reverse engineering activities involve systematically transforming a program to a higher level of abstraction. We have by now collected enough data on this process that we should be able to characterize which kinds of abstractions have proven useful, what their enabling conditions are, to what extent they are automatically detectable, and how they can be combined to support higher order transformations.
- Domain modeling. Our reverse engineering activities were based on an informal description of the application domain. Certain application domains, such as report writing, are well enough understood that an explicit domain model can be constructed. The model can be used in several ways. It can be compared to the model that resulted during the reverse engineering process, and it can be used to support reverse engineering efforts on related programs. From these efforts, we can learn the ways in which domain models can best support reverse engineering activities.
- Knowledge representation. Domain models must be explicitly represented. One approach to this is to use an existing knowledge representation language. Other approaches are described in [17]. We would like to explore how well such languages are capable of supporting reverse engineering efforts such as those described here.

Acknowledgements

The authors gratefully acknowledge the support of the Army Research Laboratory through contract DAKF 11-91-D-0004 and the participation of Richard Clayton, Bret Johnson, and Gary Pardun.

Appendix I - Strategy Selection Decision Criteria

Factors Related to Usage of the Existing System

- Usage profile and availability: How many users does the system currently have? How are they distributed topologically (do they log into the mainframe, do they submit batch jobs, or are run requests handled manually)? How frequently does a given user make use of the application? In what different ways is the application used (what is the ratio of data updates to reports produced)? How frequently does each such use occur? What is the physical process by which the application is used (data entry and validation handled separately; manual or electronic distribution of reports)? How many different sites use the existing system?

- Expected lifetime: What is the expected lifetime of the existing application? Is use growing or shrinking?

- Current execution costs: How much does it currently cost to execute the program in terms of machine and human resources? How does this cost vary across types of uses?

- Ownership and control: Are there political factors that would impede the reduction in information control that comes from distributed access?

- Administration: Are there administrative procedures that would be difficult to provide in a distributed environment? What are the costs in transforming these procedures?

- Interoperation: Do other applications depend directly on the data produced by this application (master file, report files, exception files)? Does this application depend on the products of other applications?

Factors Related to the Structure and Functionality of the Existing System

- Current architecture: How amenable is the current architecture to the client/server model? Is the application primarily batch or interactive?

- Hardware configuration considerations/resource availability: What external resources and interfaces does the application require? How extensively are they used?

- Software configuration considerations: Does the existing system make use of non-portable operating system capabilities? Does the existing system interface to other systems?

- Reports: Does the existing system write reports? If so, how separable is the computational functionality from the report construction functionality? Are there reports that can be replaced by SQL queries? Are there reports that can be replaced by the RDBMS report writer tool?

- Other RDBMS features: Does the current application do significant data validation that can be replaced by the data validation features of an RDBMS? Can the current application make effective use of advanced RDBMS operations like views and joins?

Factors Related to Expected Usage of the Transited System

- Increased use: What is the expected increase in use of the system due to networked availability? What is the expected change in use (e. g., from batch to interactive) promoted by distributed access?

- DBMS functions: Can the application take advantage of DBMS features such as security and integrity?

- Proposed execution costs: What is the expected change in execution cost in terms of machine and human resources?

Factors Related to Expected Evolution of the Transited System

- Technical impediments: Does the existing system make use of a DBMS? Is it relational? Does the existing system make use of an older version of COBOL? Are there portability issues related to data conversion? Can this application be integrated into others?

- Maintenance requirements: How much corrective maintenance activity is there currently on the system? What enhancements to the system are planned? Which enhancements would be facilitated by the use of an SQL interface to the data?
- Support issues: Are there personnel available that have experience with the internals of the existing system? Is there existing documentation for the system? How up-to-date and accurate is it? Is sufficient funding available for a comprehensive reverse engineering effort? Does this include funding to support the training of users in 4GLs? How feasible is incremental conversion?
- Standards: Is the application part of the effort to standardize the use of data item names across applications? How closely does it conform to these standards?

References

[1] *Automated Data Systems Manual, Installation Materiel Condition Status Reporting System (IMCSRS), Functional User's Manual,* Commander FORSCOM, AFLG-RO, Ft. McPherson, Georgia, April 1, 1984.

[2] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe, *Conceptual Database Design — An Entity-Relationship Approach,* Benjamin Cummings, 1992.

[3] Barry W. Boehm, *Software Engineering Economics,* Prentice Hall, 1981.

[4] P. P. Chen, "The Entity-Relationship Model-Toward a Unified View of Data," *ACM Transactions on Database Systems,* pp 9-36, March 1976.

[5] Kathi Hogshead Davis and Adarsh K. Arora, "A Methodology for Translating a Conventional File System into an Entity-Relationship Model," *Proceedings 4th International Conference on the Entity-Relationship Approach,* IEEE, 1985.

[6] Melody Eidbo and et al., "ISA-97 Compliant Architecture Testbed (ICAT) Project Final Report," ASQB-GC-92-006, AIRMICS.

[7] Ian J. Hayes, "Applying Formal Specification to Software Development in Industry," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 2, pp 169-178, February 1985.

[8] Reginald L. Hobbs, John R. Mitchell, Glenn E. Racine, and Richard Wassmath, "Re-engineering Old Production Systems: A Case Study of Systems Redevelopment and Evaluation of Success," *Emerging Information Technologies for Competitive Advantage and Economic Development: Proceedings of the 1992 Information Resources Management Association International Conference,* pp 29-37, Harrisburg, Pennsylvania, May 1992.

[9] Reginald L. Hobbs, Joseph J. Nealon, and Richard Wassmath, "Ada Transition Research Project (Phase I) Final Report," ASQB-GI-91-005, AIRMICS, December 10, 1990.

[10] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering - A Use Case Approach,* Addison-Wesley, 1992.

[11] Bret Johnson, "Reverse Engineering with a CASE Tool," SRC-TR-91-07, Software Research Center, College of Computing, Georgia Institute of Technology, December 1991.

[12] Bret Johnson, Steve Ornburn, and Spencer Rugaber, "A Quick Tools Strategy for Program Analysis and Software Maintenance," *Proceedings of the Conference on Software Maintenance,* pp 206-213, Orlando, Florida, November 1992.

[13] Kit Kamper and Spencer Rugaber, "Reverse Engineering Methodology for Data Processing Applications," GIT-SERC-90/02, Software Engineering Research Center, Georgia Institute of Technology, March 1990.

[14] Erik G. Nilsson, "The Translation of a Cobol Data Structure to an Entity-Relationship Type Conceptual Schema," *Proceedings 4th International Conference on the Entity-Relationship Approach,* IEEE, 1985.

[15] James J. Purtilo and John R. Callahan, "Parse Tree Annotations," *Communications of the ACM,* vol. 32, no. 12, pp 1467-1477, December 1989.

[16] Glenn E. Racine, Reginald L. Hobbs, and Richard Wassmath, "Ada Transition Research Project (A Software Modernization Effort)," *Proceedings of the 10th Annual National Conference on Ada Technology,* pp 192-201, February 1992.

[17] Spencer Rugaber and Richard Clayton, "The Representation Problem in Reverse Engineering," *Proceedings of the First Working Conference on Reverse Engineering,* Baltimore, Maryland, May 21-23, 1993.

[18] A Spencer Rugaber and Kit Kamper, "Design Decision Analysis Research Project," GIT-SERC-90/01, Software Engineering Research Center, Georgia Institute of Technology, January 28, 1990.

[19] Peter G. Sassone, "Economic Justification," ASQB-GM-92-008, AIRMICS, December 1991.

[20] Edward Yourdon, *Structured Walkthroughs,* Yourdon Press, 1989.