A REVERSE ENGINEERING METHODOLOGY FOR DATA PROCESSING APPLICATIONS

Kit Kamper
Spencer Rugaber

School of Information and Computer Science
and
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450

ABSTRACT

Reverse engineering produces a high-level representation of a software system from a low-level one. This paper describes a methodology for reverse engineering that constructs an architectural design for a system from its source code and related documentation. The methodology makes use of several techniques normally used during the forward software development process as well as a new technique called Synchronized Refinement. Synchronized Refinement is a systematic approach to detecting design decisions in source code and relating the detected decisions to the functionality of the system. Examples are given demonstrating the application of the methodology to the reverse engineering of a production software system.

March 9, 1994

A REVERSE ENGINEERING METHODOLOGY FOR DATA PROCESSING APPLICATIONS

Kit Kamper
Spencer Rugaber

School of Information and Computer Science
and
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450

1. INTRODUCTION

Chikofsky and Cross[7] define reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction". Typically, this means constructing an architectural design of a system from its source code. Reverse engineering can be used for a variety of purposes: to reconstruct or otherwise improve documentation; to facilitate software maintenance or conversion activities; or to redesign and re-engineer an existing system.

This paper describes a methodology for reverse engineering and how it was applied to an operational software system. The methodology makes use of approaches and representations typically found in the forward software development process, including a high-level, textual overview and graphical representations of data flows and file structures. Additionally, a new program analysis procedure, called Synchronize Refinement, is used to extract detailed implementation information.

2. PROJECT DESCRIPTION

The reverse engineering project examined a system targeted for redesign, the Installation Materiel Condition Status Reporting System (IMCSRS), a Standard Army Management Information System. The function of IMCSRS is to assimilate and distribute equipment readiness and maintenance status information to U. S. Army installations and divisions. In essence, it is a file creation, maintenance, and reporting system. It is composed of sixteen programs grouped into six execution cycles. Together, these comprise just under 10,000 lines of source code written in COBOL. The system uses standard support software for executive processes, error detection and handling, file spooling, and common library routines. IMCSRS runs on an IBM/370 computer under the OS/MVS operating system. Available documentation consists of the *Program Maintenance Manual*, the *Functional User's Manual*, and the *Computer Operations Manual*.

The reverse engineering effort is part of a larger project that has a goal to investigate the applicability of the Ada programming language for systems like IMCSRS. The high-level representation produced by reverse engineering will serve as the basis of two redesign efforts, one using traditional Functional Decomposition[14], and the other using Object Oriented Design[4]. If feasible, two implementations will be completed and the results compared for performance, maintainability, and development cost.

The reverse engineering was conducted in a research laboratory at the Georgia Institute of Technology. The primary computing vehicle used was a SUN 386i workstation, running the SUN (UNIX-derivative) operating system and the SUNVIEW windowing environment. The standard editing tool (**vi**) and source control system (**sccs**) were used for manipulating the source code. The campus network

provided access to a mainframe COBOL compiler.

A substantial part of the reverse engineering effort involved the production of graphical representations of program structures. For this purpose, the Software Through Pictures CASE tool[1], a product of the Interactive Development Environments Corporation, was used.

3. ISSUES

Part of the charter for the reverse engineering project was to develop a systematic methodology for the reverse engineering of programs similar to IMCSRS. Because of this specific objective, several issues arose.

1. The reverse engineering is targeted at redesign/conversion rather than at other possible goals, such as improving maintainability. What implications does this have on the methodology developed?
2. IMCSRS was the first system to be looked at for conversion. There were no available data indicating the amount of effort required to analyze and describe a system of this size. On the other hand, there was a specific time constraint: the reverse engineering had to be completed in four calendar months. What approach should be taken to the reverse engineering that would guarantee useful information at the end of the time period?
3. IMCSRS is a typical data processing application. What implication does this have on the methodology?
4. The results of the reverse engineering are to be used as a starting point for two parallel redesign projects. What forms should the high-level representations take that would not bias the comparison in favor of one or the other of the design approaches?
5. IMCSRS is relatively small, the code is cleanly structured, and documentation is readily available. How must the methodology be adapted to deal with more difficult systems?

The approach that was chosen was top-down and used a variety of analysis mechanisms and representations. The fact that it was top-down assured us of producing some results, even if we ran out of time before completely understanding all of the low-level implementation details. The approach was also more likely to generate architectural information of use during redesign. If the project goals had stressed improving the program's maintainability, rather than facilitating redesign, then a more bottom-up approach would have been appropriate. This addressed the first two issues.

The fact that the approach taken used a variety of analysis mechanisms and representations addressed issues 3 and 4, raised above. A mixed approach helped to avoid biasing the redesign effort and allowed us to take advantage of particular representation mechanisms suitable for this type of application.

The last issue is addressed in Section 6.2.

4. A METHODOLOGY FOR REVERSE ENGINEERING

4.1. Background

A systematic methodology provides a variety of benefits to software engineers. Most importantly, quality is improved. Because a methodology precisely describes the intermediate and final results of production, comparisons are possible between the products of independent efforts, and planning and scheduling efforts are facilitated. Also, because uniform procedures are used, support tools can be built, which also contribute to productivity and quality improvements. Furthermore, uniformity allows training materials to be developed such as tutorials and courseware. Finally, a methodology supports interim feedback. This can take the form of built-in validation mechanisms to detect inconsistency and

incompleteness.

4.2. Overview

The methodology developed for reverse engineering IMCSRS consists of four phases: a documentation review, an analysis of system input/output structure, an analysis of the structure of the input and output files, and a detailed analysis of the source code using Synchronized Refinement.

The running example used in the following sections is taken from the final report of the IMCSRS project[11]. It begins with the high-level details and gradually focuses on the Generate-Separate-Report-Files program (P09AGU).

4.3. Documentation Review

System documentation has a reputation for being out-of-date and inaccurate. In the absence of accurate documentation, reverse engineers are confronted with a dilemma. They are expected to construct a description of what a system does given only a description of how it does it. Nevertheless, the existing documentation may be the only starting point from which the application can be appreciated.

The first stage of the methodology involves a review of the existing documentation. Its product is a functional description of the system, making no mention of implementation details or programming language. It is instead concerned with the system's functional behavior, the constraints imposed by the computing environment, the input and output files, the generated reports, and the user interface.

The description is top-down. That is, it proceeds from a discussion of the overall system behavior to a discussion of those sub-components that are visible to the user. Components that exist only as the result of a specific implementation strategy are not described. It begins with a short summary of the overall system behavior. In the case of IMCSRS, this takes the form of the following statements.

- The function of the IMCSRS is to assimilate and distribute equipment readiness and maintenance status information at the installation/division level. Management level reports are produced detailing the status, posture and condition of selected items of equipment. Both detailed and summarized reports are produced . . .
- The IMCSRS structure is composed of COBOL programs grouped into six cycles. Information on these cycles can be found in . . .
- Cycle AGU04: Inputs the valid 2406 transaction file created in the AGU03 cycle and produces formatted equipment and materiel readiness reports . . .
- Program P09AGU: Generate Separate Report Files. Creates five separate report files . . .

Figure 1: Textual Description Excerpts

4.4. System Input/Output Behavior

The next stage of the methodology consists of an analysis of the system's input/output behavior expressed in terms of nested data flow diagrams. A data flow diagram consists of three types of objects: nodes indicating system activities or processes, directed arcs representing flows of data among the activities, and data repositories containing persistent data[12].

At the highest level, the system behavior is represented by a Context Diagram. A Context Diagram is a data flow diagram with one activity—the system being analyzed. All external files consumed or produced by the system are represented as repositories. The direction of the arcs indicates whether the file is used for input or output. The Context Diagram for IMCSRS is shown in Figure 2.

Figure 2: IMCSRS Context Diagram

The Context Diagram generated from the system description produced in the previous stage is verified by examining the source code. This serves as an internal check on the consistency of the derived design.

Data flow diagrams may be nested. That is, a process node at one level can be expanded into an entire diagram at a lower level. Arcs that enter the process node at the higher level are represented by arcs with no source at the lower level. Likewise, arcs the leave the process node at the higher level are indicated by arcs with no target at the lower level. For example, Figure 3 describes the major subcomponent structure of IMCSRS. It was defined by expanding the process node in the Context Diagram. The figure shows the primary data of the system: the equipment and materiel information contained in the ECC/LIN master file and the installation description information contained in the UIC master file. These are updated and combined in cycles AGU01-3. Cycles AGU04-6 produce reports as indicated by squares containing the letter 'R'.

The data flow diagrams are drawn using Software Through Pictures, which provides more than just a diagram editor. It also checks the consistency and completeness of the diagrams at different levels. This serves as another validation of the emerging design.

The nesting of the diagrams proceeds until all system input/output behavior has been described. In the case of IMCSRS, two of the cycles had sufficiently complicated structures that data flow diagrams were produced that described their interactions, particularly their use of intermediate files. Cycle AGU04 consists of ten programs primarily responsible for producing reports. The example program, P09AGU, is responsible for separating AGU04's input data into separate files, suitable for generating individual reports.

4.5. Structure of Input and Output Files

The next stage of the methodology consists of an analysis of the structure of the files used in the system. The analysis is expressed in terms of Jackson Data Structure diagrams, a part of the Jackson System Design Methodology[9]. Jackson's methodology has proven to be particularly appropriate for data processing applications such as IMCSRS.

A Jackson diagram describes the organization of a file as a tree-structured collection of boxes. There are three types of boxes indicating three ways in which a file may be decomposed. The type is indicated in the top right hand corner of each box. Unmarked boxes show the breakdown of the file into pieces; a box marked by an asterisk indicates a repeating group; and boxes with small circles denote alternatives.

Figure 3: Data Flow Diagram Showing Execution Cycles

As an example, Figure 4 describes the structure of the R08AGU file that serves as input to P09AGU. The file is composed of a sequence of Report Records. A Report Record may be one of five varieties, depending on the value of one of its fields. Jackson diagrams are constructed by examining the FILE SECTION of the corresponding COBOL program.

Figure 4: P09AGU Input Data Structure

5. SYNCHRONIZED REFINEMENT

It is sometimes necessary to obtain a detailed description of how a specific function is accomplished by a program. This might arise in the case of complex algorithms or where there is some question as to the accuracy or completeness of the documentation. Such a description must relate the specific details of the code to the associated application functions being implemented. That is, the process of analyzing the code must go on in parallel with the process of building up a description of what the code accomplishes.

A procedure, called Synchronized Refinement, has been devised to accomplish this. It consists of the parallel analysis of the source code and synthesis of a functional description. The process is driven by the detection of design decisions in the source code. Each decision is annotated in the functional description. The annotation states how the decision contributes to the function accomplished by the corresponding code. After each decision is detected and annotated, the corresponding part of the source code is replaced by an abbreviated description. In this way, the source code continually grows shorter while the functional description grows more complete.

5.1. Design Decisions

Design decisions are structural decisions made by the original designer or programmer. Typical design decisions include the decomposition of a function into its subfunctions, the handling of special cases, and the use of one data structure to represent another that is not directly provided by the programming language. Other decisions include the encapsulation of a set of procedures into a module and the introduction of a data item to save the result of a computation for later use. A more detailed description of design decisions is given in[10].

There is no strict order for detecting design decisions. Complex programs are typically designed as layers of abstraction, and, as one layer is detected during the reverse engineering process, it opens the door for the detection of other decisions in higher layers. Nevertheless, it is possible to give the following guidelines for the detection process.

- Particularly where older programming languages are used, begin by looking for the occurrence of modern control structures implemented by more primitive constructs. An example of this is the use of nested IF statements and GOTOs to implement a SWITCH statement.

- Likewise, look for the use of primitive data structures to represent unavailable ones, such as the use of specific integer or character constants to encode one of a set of values.
- It may sometimes be necessary to restructure the control flow of the program. This is essentially what a restructuring tool provides. The resulting code should be functionally equivalent to the original.
- Look for special cases; that is, look for similar sections of code that differ only in a small number of ways. Replace these by the parameterized use of a more abstract construct.
- If the language does not support modularization, look for code that should be grouped together and separated from the rest of the program by an abstract interface.
- In contrast, sometimes a programmer intentionally interleaves the accomplishment of two objectives within the same section of code. This is often done for reasons of efficiency. To understand the code, it becomes necessary to segregate the two functions and annotate them separately.
- If a collection of data items are used together to implement an unavailable construct, annotate that fact.
- Make sure that variables are only used for one purpose. Replace dual-use variables by two separate variables.

5.2. The Procedure

Synchronized Refinement consists of two parallel activities: analysis of the program and synthesis of the functional description. The analysis process consists of detecting design decisions and replacing the related code by an abbreviated version. In this way, the program description continues to become smaller and more abstract.

The synthesis process begins with a high-level description of the overall program as obtained from the documentation review, possibly augmented with comments from the source code. This description leads to certain expectations in the reverse engineer's mind. For example, if a sorted report is to be produced, it is expected that part of the code will be responsible for the sort (or preparing data for an external sort), and there is likely to be a section controlling the pagination of the report. Note that the expectations need not be complete nor even entirely accurate at this stage.

A dynamic list of expectations is kept. As the analysis process proceeds, decisions are detected that relate to an expectation. For example, in the case of the pagination expectation, a section of code is found that keeps a counter that resets after reaching the length of a page. The annotation for the detected decision is placed together with the relevant expectation, specifying the type of the decision and the corresponding sections of the program. During the process, certain expectations will be confirmed and others may be refuted. A confirmed expectation may lead to others. Gradually, a hierarchical description of the structure of the program emerges. As the program source code shrinks, the functional description expands.

The process can be summarized by the following.

- Recognize a design decision in the code. Determine its type and the lines that implement it.
- Verify the decision, if necessary, by searching the source code for relevant evidence.
- Annotate the decision in the Functional Description.
- Indicate expectations that have been confirmed. Add new expectation arising from confirmed ones. Delete refuted expectations, if any.
- Abstract the program by replacing the code that implements the decision by a high level description.
- Use the currently outstanding expectations to guide the search for a new decision.
- Iterate these steps until the program has been abstracted to a single statement describing its behavior.

The top-down aspects of this process are similar to the psychological model of code comprehension described by Ruven Brooks[6]. The analysis of program text, as driven by design decisions, is new. An alternative procedure, based on control flow analysis and program slicing, is described in a paper by Hausler et al[8]..

5.3. Example

In the case of the program P09AGU, Synchronized Refinement begins with the description of its purpose as obtained in the documentation review and from a comment† in the beginning of the source code (Figure 5). These form the initial functional description of the program.

```
*REMARKS. THIS PGM CREATES SEPARATE A, B, C, G, E, AND F
*         FILES FROM P07AGU ALSO CREATES 12 CD SUMMARY
*         DISK FILE H09AGU FROM Q CARDS IN 'A' FILE.
```

Figure 5: P09AGU: Comments Obtained From The Source Code

From this description, a list of expectations outlining what might be found in the code is compiled. (See Figure 6.)

1. Open input files
2. Open output files
3. Read input record
4. Select appropriate output file
5. Write output record
6. Error checks
7. Stop
8. Select 'Q' cards from 'A' file

Figure 6: P09AGU: Initial Expectations

The program's PROCEDURE DIVISION is given in Figure 7.

† It will turn out that the comment in the source code is incorrect.

```
0119  PROCEDURE DIVISION.
0120  0010-BEGIN.
0121      MOVE MUGUIRA TO WS-MUGUIRA-FORMAT.
0122      DISPLAY MUG-DISPLAY.
0123      OPEN INPUT IR08AGU OUTPUT OB09AGU OC09AGU OG09AGU
0124          OE09AGU OF09AGU.
0125  0020-READ-DISK.
0126      READ IR08AGU
0127          AT END
0128          CLOSE IR08AGU OB09AGU OC09AGU OG09AGU OE09AGU
0129              OF09AGU
0130          STOP RUN.
0131      IF CARD-CODE = 'B'
0132          WRITE B-RECORD FROM REPORT-FILE
0136              INVALID KEY
0137              DISPLAY 'P09AGU 02 H B09AGU EXCEEDED SYS022'
0138              PERFORM 0030-STOP.
0140      IF CARD-CODE = 'C'
0141          WRITE C-RECORD FROM REPORT-FILE
0145              INVALID KEY
0146              DISPLAY 'P09AGU 03 H C09AGU EXCEEDED SYS041'
0147              PERFORM 0030-STOP.
0149      IF CARD-CODE = 'D'
0150          WRITE D-RECORD FROM REPORT-FILE
0154              INVALID KEY
0155              DISPLAY 'P09AGU 04 H G09AGU EXCEEDED SYS042'
0156              PERFORM 0030-STOP.
0158      IF CARD-CODE = 'E'
0159          WRITE E-RECORD FROM REPORT-FILE
0163              INVALID KEY
0164              DISPLAY 'P09AGU 05 H E09AGU EXCEEDED SYS024'
0165              PERFORM 0030-STOP.
0167      IF CARD-CODE = 'F'
0169          WRITE F-RECORD FROM REPORT-FILE
0174              INVALID KEY
0175              DISPLAY 'P09AGU 06 H F09AGU EXCEEDED SYS025'
0176              PERFORM 0030-STOP.
0177      GO TO 0020-READ-DISK.
0178  0030-STOP.
0179      DISPLAY
0181          'P09AGU 07 I JOB TERMINATED BY PROGRAM: NO OPERATOR INTER
0184          'VENTION REQUIRED'.
0186      CLOSE IR08AGU OB09AGU OC09AGU OG09AGU OE09AGU
0187          OF09AGU.
0189      CALL 'S01ATU'.
0191      STOP RUN.
```

Figure 7: P09AGU PROCEDURE DIVISION

To illustrate Synchronized Refinement, a *representation* decision is located and processed. Representation decisions are of value when an abstraction is not directly available in the programming language. For example, COBOL does not contain an enumeration data type such as can be found in C or Pascal. The programmer of P09AGU was forced to use a character string variable (CARD-CODE)

to represent this concept.

Representation decisions also occur with control structures, particularly where older languages, such as Fortran or COBOL, are concerned. A casual reading of P09AGU reveals a great deal of regularity in the program structure. This fact, coupled with the expectation that the program needs to partition input records, suggests that the IF statements have been used to implement a CASE/SWITCH construct, as might be found in the Pascal or C programming languages. This is confirmed by constructing a flow diagram for the procedural part of the program. (See Figure 8.) From this a 5-way branch based on the variable CARD-CODE can be observed.

Figure 8: P09AGU Program Control Flow

Once this decision has been recognized and confirmed, the program text can be transformed (Figure 9), and the functional description can be updated with the annotated decision (Figure 10).

```
        switch_on CARD_CODE
0131        case B
0132            WRITE B-RECORD FROM REPORT-FILE
0136                INVALID KEY
0137                DISPLAY 'P09AGU 02 H B09AGU EXCEEDED SYS022'
0138                PERFORM 0030-STOP.
0140        case C
0141            WRITE C-RECORD FROM REPORT-FILE
0145                INVALID KEY
0146                DISPLAY 'P09AGU 03 H C09AGU EXCEEDED SYS041'
0147                PERFORM 0030-STOP.
0149        case D
0150            WRITE D-RECORD FROM REPORT-FILE
0154                INVALID KEY
0155                DISPLAY 'P09AGU 04 H G09AGU EXCEEDED SYS042'
0156                PERFORM 0030-STOP.
0158        case E
0159            WRITE E-RECORD FROM REPORT-FILE
0163                INVALID KEY
0164                DISPLAY 'P09AGU 05 H E09AGU EXCEEDED SYS024'
0165                PERFORM 0030-STOP.
0167        case F
0169            WRITE F-RECORD FROM REPORT-FILE
0174                INVALID KEY
0175                DISPLAY 'P09AGU 06 H F09AGU EXCEEDED SYS025'
0176                PERFORM 0030-STOP.
        end switch
```

Figure 9: SWITCH Statement Represented by IF-THEN Statements

1. Open input files
2. Open output files
3. Read input record
4. Select appropriate output file -
   switch statement on CARD-CODE selects appropriate code section;
      Representation decision, lines 131-176
5. Write output record
6. Error checks
7. Stop
8. Select 'Q' cards from 'A' file

Figure 10: Confirmation of Expectation 4

Further observation of the code indicates that all of the output files are handled in a similar fashion. For example, lines 131-138 are similar to line 140-147. Each of these sets of lines are really instances of a more general procedure. They differ by parameters controlling the selection code, the name of the output record, and the text of the message to be displayed on error. This type of design decision is called *generalization*.

Occasionally, the code reveals unexpected functionality. For example, P09AGU contains statements to print messages on startup and termination. These are part of the functional behavior of the program and need to be added to the functional description. Also, no mention of the 'Q' cards or the 'A' file can be found in the source code. This expectation is refuted; the original comment was

incorrect!

Another unanticipated construct not on the original list of expectations is the loop on lines 125 and 177. It is a WHILE loop, implemented with a label and a GOTO statement. This was an oversight in the original expectation list that, in retrospect, should have been obvious. It is also an instance of a representation decision.

Other design decisions can be detected in the source code. These include the *decomposition* of the program into its constituent pieces and the use of several names (B-RECORD, C-RECORD, etc.) for the same value, an instance of an *interleaving* design decision.

When all of the decisions have been detected, annotated and abstracted, the resulting program appears as in Figure 11.

Format startup message
Partition input file into output files based on CARD-CODE field
Display termination message

Figure 11: Source Code Modification - High Level Abstraction

The final list of expectations appears in Figure 12.

1. Open input files - line 123
2. Open output files - lines 123-4
3. Read input record - line 126
4. Select appropriate output file -
   Switch statement on CARD-CODE selects appropriate code section
   All cases handled identically other than where parameterized
   4.1 Input field used to discriminate output file -
      CARD-CODE is a field in the input record
   4.2 File descriptors for output files contain appropriate record names
5. Write output record - occurs in each of the cases of the switch
6. Error checks - occurs in each of the cases of the switch
7. Stop - line 191
8. Write startup message - lines 121-122
9. Write termination message - lines 179-184
10. Loop over input records - terminate on end of file

Figure 12: P09AGU Expectations - Confirmed and Expanded

The program and its annotations can then be combined into a functional description of the program behavior as illustrated in Figure 13.

The program produces a message upon starting.
The program loops over the input file reading records.
   From each record a code field is examined.
   The code field is used to direct the output to the appropriate file.
   An error message is written upon the detection of an output error condition.
A message is written indicating successful termination.

Figure 13: Final Representation - P09AGU

6. CONCLUSIONS

6.1. Summary of the Methodology

A methodology consists of a systematic procedure for accomplishing a task, a well-defined representation for the final results and any intermediate products, and a built-in mechanism for validation. The methodology used on this project involves a top-down construction of a program description. At the top level, this consists of a textual description of the program function expressed in application-domain terminology.

The textual description is followed by the construction of nested data flow diagrams summarizing the input/output behavior of the system and its functional components. Then, each external file is described by a Jackson diagram giving its structure in terms of subcomponents, alternatives, and repeating groups. Where necessary, a detailed analysis of the source code is performed using Synchronized Refinement. This produces a functional description of the code and annotates how the function is accomplished in terms of important design decisions.

Validation is accomplished by using internal consistency checks. The Context Diagram is constructed from the textual description and checked against the source code. Nested data flow diagrams are automatically checked against each other by the CASE tool. Jackson diagrams are constructed from the FILE SECTIONs of the source code and checked against the textual description and the data flows. Finally, detailed code analysis using Synchronized Refinement is used to check that the code accomplishes what it is supposed to.

6.2. Scaling

It should be pointed out that the particular reverse engineering exercise described in this paper may not be typical. Not only is it relatively small (around 10,000 lines), but reasonably accurate documentation exists, and the code is cleanly structured. In situations where larger systems are being analyzed or where the software has degraded, adaptations may be required.

Most important is the question of scaling. If a larger system were involved, more people would be required to perform the analysis. For the same reasons that initial design "must proceed from one mind, or from a very small number of agreeing resonant minds."[5], so too should the overall reverse engineering process be the responsibility of a single person. Subtasks can be delegated, such as the production of a specific diagram, but the evolving understanding of the system function must be comprehended by a single individual.

On the other hand, there is a significant difference between forward and reverse engineering regarding the flow of information. Normally, the principle of *information hiding*[13] restricts the publication and use of implementation decisions during design. This control facilitates subsequent maintenance activities. In the case of reverse engineering, however, it is important that details discovered by one participant be broadcast to others so that the overall comprehension process is facilitated. Instances where implicit dependencies among sections of code hamper the maintenance process ("ripple effects") are notoriously troublesome.

6.3. Tools

It would be desirable if the entire process described above were supported by an integrated set of tools†. This would have several benefits in addition to reducing the total effort and time required to reverse engineer a system. First, the chance of error would be reduced. The chance of error exists in any systematic, labor-intensive process, and reverse engineering is no exception. Second, the production of documentation would be improved. The documentation for the IMCSRS reverse engineering

---

† There has been some interesting work by Benedusi et al. to automate the production of data flow diagrams and Jackson diagrams[2, 3].

project consisted of text formatted both on a MacIntosh personal computer and the SUN workstation, data flow diagrams entered into Software Through Pictures, Jackson diagrams entered into the MacIntosh, and Synchronized Refinement performed using **vi** and **sccs** on the SUN. Of course, if all the representations were available in the same environment, opportunities for internal consistency checks would increase. Finally, and most intriguing, is the possibility of using the integrated representation directly in the later redesign effort. If all of the information obtained were available for browsing and manipulation by the redesigner, that task would be facilitated as well.

REFERENCES

References

1. *Software Through Pictures*, Interactive Development Environments, Corporation. 150 Fourth Street, Suite 210, San Fransisco, California, 94103.
2. P. Antonini, P. Benedusi, G. Cantone, and A. Cimitile, ''Maintenance and Reverse Engineering: Low-Level Design Documents Production And Improvement,'' *Proceedings of the Conference on Software Maintenance - 1987*, pp. 91-100, IEEE Computer Society Press, Austin, Texas, September 21-24, 1987.
3. P. Benedusi, A. Cimitile, and U. De Carlini, ''A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams for Software Maintenance,'' *Proceedings Conference on Software Maintenance-1989*, pp. 180-189, Miami, Florida, October 16-19, 1989.
4. Grady Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings Publishing Co., 1991.
5. Frederick P. Brooks, *The Mythical Man-Month, Essays on Software Engineering*, Addison-Wesley Publishing Company, 1982.
6. Ruven Brooks, ''Towards a Theory of the Comprehension of Computer Programs,'' *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.
7. Elliot J. Chikofsky and James H. Cross II, ''Reverse Engineering and Design Recovery: A Taxonomy,'' *IEEE Software*, vol. 7, no. 1, pp. 13-17, January 1990.
8. Philip A. Hausler, Mark G. Pleszkoch, Richard C. Linger, and Alan R. Hevner, ''Using Function Abstraction to Understand Program Behavior,'' *IEEE Software*, vol. 7, no. 1, pp. 55-63, January 1990.
9. M. A. Jackson, *Principles of Program Design*, Academic Press, 1975.
10. Bret Johnson, Steve Ornburn, and Spencer Rugaber, ''A Quick Tools Strategy for Program Analysis and Software Maintenance,'' SRC-TR-92/02, Software Research Center, Georgia Institute of Technology, June 1992.
11. Kit Kamper and Spencer Rugaber, *Joint Georgia Tech/BNR, Inc. Reverse Engineering Research Project*, Software Engineering Research Center, School of Information and Computer Science, Georgia Institute of Technology, September 18, 1990.
12. James Martin and Carma McClure, *Structured Techniques: The Basis for CASE, Revised Edition*, Prentice Hall, 1988.
13. D. L. Parnas, ''Information Distribution Aspects of Design Methodology,'' *Information Processing 71*, North-Holland, 1972.
14. W. P. Stevens, G. J. Myers, and L. L. Constantine, ''Structured Design,'' *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974.