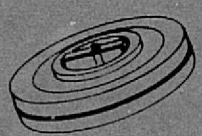
NASA

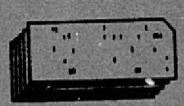
AUTOMATIC DOCUMENTATION SYSTEM EXTENSION TO MULTI-MANUFACTURERS' COMPUTERS AND TO MEASURE, IMPROVE, AND PREDICT SOFTWARE RELIABILITY

FINAL REPORT

NASA CONTRACT NO. NAS 5-20715

OCTOBER 1975





Submitted to

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND

Submitted by

DATA PROCESSING CENTER

TEXAS A&M UNIVERSITY

AUTOMATIC DOCUMENTATION SYSTEM EXTENSION TO MULTI-MANUFACTURERS' COMPUTERS AND TO MEASURE, IMPROVE, AND PREDICT SOFTWARE RELIABILITY

Final Report

NASA Contract No. NAS5-20715

October, 1975

Submitted to

National Aeronautics and Space Administration
Goddard Space Flight Center
Greenbelt, Maryland 20771

Submitted by

Dick B. Simmons

of the

Data Processing Center
Texas Engineering Experiment Station
College of Engineering
Texas A&M University
College Station, Texas 77843

FOREWORD

This report presents the results of the project to extend the DOMONIC system to multi-manufacturers' computers and to measure, improve and predict software reliability. This work was performed by the Data Processing Center of the Texas Engineering Experiment Station of Texas A&M University, College Station, Texas. This work was performed under Contract NAS5-20715 of the National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt, Maryland. The Project Monitor was Mr. E.P. Damon.

The Principal Investigator of the Project was Dr. Dick B. Simmons. The Manager over the development and extension of the DOMONIC system has been Mr. Pete Marchbanks. Major contributors to the development and extension of the DOMONIC system were: Louis Devito, Mike Quick, and Glen Hascall. Students working on this phase of the project have been Chap Chi Wong, Eliseo Pena, and Ollie Polk.

Major contributors to the extension of the system to measure, improve and predict software reliability, have been Dr. Roger Elliott, Dr. Larry Ringer, Dr. William Lively, Dr. Richard Fairley, and Mrs. Jean Zolnowski, who has acted as coordinator for this phase of the project. Dr. Dick B. Simmons has directed both phases of the project.

Phase I of the project is described in Chapters I and II of the final report. The DOMONIC Users Manual which was updated during this phase is included as Appendix A. The DOMONIC Command Reference Manual is included as Appendix B.

Phase II of the project is described in Chapter I and Chapters III through VII. Chapter III was written by Drs. Elliott and Ringer, Chapter IV by Mrs. Jean Zolnowski, Chapters V and VII by Dr. Fairley, and Chapter VI by Dr. Lively.

ABSTRACT

This report describes the work done on the project to extend the DOMONIC system to Multi-Manufacturers' Computers and to Measure, Improve, and Predict Software Reliability. The DOMONIC system has been modified to run on the Univac 1108 and the CDC 6600 as well as the IBM 370 computer system. The DOMONIC monitor system has been implemented to gather data which can be used to optimize the DOMONIC system and to predict the reliability of software developed using DOMONIC. The areas of quality metrics, error characterization, program complexity, program testing, validation and verification are analyzed. A software reliability model for estimating program completion levels and one on which to base system acceptance have been developed. The DAVE system which performs flow analysis and error detection has been converted from the University of Colorado CDC 6400/6600 computer to the IBM 360/370 computer system for use with the DOMONIC system.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

1.0 INTRODUCTION

The Data Processing Center at Texas A&M University is pleased to submit this final report for Contract NAS5-20715 to the National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt, Maryland. The purpose of the project was to extend the capabilities of the automatic system for computer program documentation which was developed by Texas A&M University for NASA. The system has been designed to produce timely up-to-date documentation at relatively low cost. The system has been designed to document any computer language and to run on any hardware while taking advantage of the existing documentation aids. The system is easy to use and places no restrictions on the programmer.

During the initial phase of the development, the system was implemented to run on the IBM 360/370 series computers. Major emphasis during the first phase has been to document programs written in FORTRAN. The extensions covered by this final report were broken into two phases:

Phase 1 - Extend the system to operate on CDC and Univac computers and integrate appropriate additional documentation aids into the system.

Phase 2 - Extend the system to measure, improve, and predict software reliability.

The automatic documentation system developed during previous stages has been extended to monitor and control the development process. The extended system is called the DOMONIC (Documentation, Monitor and Control) system. Accomplishments during Phase 1 and Phase 2 of the current contract

are described in Chapter II. Research reports resulting from Phase 2 are contained in Chapters III through VII. Appendices A and B contain the DOMONIC Users Manual and the Command Reference Manual.

2.0 DOMONIC SYSTEM STATUS

Phase 1 of the current contract was to extend the DOMONIC system to multi-manufacturers' computers and to expand the system capabilities. Included in this Phase was training of NASA personnel in the maintenance of the DOMONIC system, assist NASA in using the DOMONIC system to document the programs at NASA, use the DOMONIC system to document software developed at Texas A&M for NASA, implement the system on the Univac 1108 and the CDC 6600 and add additional documentation aids to the system. During Phase 2 the DOMONIC system was expanded to monitor the development process giving data which could be used on reliability models to predict software reliability. The tasks of this phase were to define categories of errors, design and implement a monitor within DOMONIC, evaluate existing software reliability techniques, and develop a software reliability model.

Phase 1 - Training sessions of NASA personnel were held at both Texas A&M University and Goddard Space Flight Center. Mr. Jack Kohout of NASA/ Goddard Space Flight Center made an on-site visit to Texas A&M University during early March, 1975. Training techniques and DOMONIC system operations were discussed with him. He made numerous suggestions to the next up-date of the DOMONIC Users Guide and Command Reference Manual. During this trip, an updated version of the DOMONIC system which incorporated some suggestions from Mr. Kohout was planned for later installation at Goddard.

An updated version of the DOMONIC system was installed at Goddard during the period of June 6-11, 1975. During that time, additional training

was conducted and suggestions were made by NASA on ways to improve the DOMONIC system.

Once the DOMONIC system became operational, modules from the system were loaded into DOMONIC and the DOMONIC system was used to maintain itself. The security and monitor modules were completely developed using DOMONIC.

The Texas A&M Data Processing Center Billing team has used DOMONIC to assist them in developing and maintaining the billing applications at the Texas A&M Data Processing Center. The Data Processing Center is in the process of expanding the use of DOMONIC.

A major effort has been expended in converting the DOMONIC system from the IBM 360/370 computer to the CDC 6600 and Univac 1108. The initial conversion effort was to convert the system to the CDC 6600 at The University of Texas (UT) at Austin. This presented the most convenient site location due to the time-sharing agreement between UT and Texas A&M University. UT uses the CDC Version 3 COBOL system which was the latest version when the initial language study for DOMONIC was made. The first attempt to convert six routines was very discouraging due to the differences in IBM's standard COBOL and the CDC Version 3 COBOL. However, after considerable thought was given to system conversion, a program was developed which would convert about 90% of the statements that had to be converted. The only remaining effort before a full-fledged conversion effort could be started was the use of the copy facilities to introduce common program segments into the programs from the centralized library. The capability was originally present at UT and, after extensive checking with the UT Systems Group and the CDC representatives in Austin and Houston, it was found that changes in the UT

operating system prevented use of the copy facilities without revision to the UT software. The absence of a copy facility was deemed unacceptable since DOMONIC has over 60,000 COBOL source statements and this figure would more than double without the use of the common descriptions which were to be copied. Maintenance of the system would be more complex if copy facilities were not used because the single modification to a common entry would become a repeated task in all modules using the common item.

A decision was made to look for another CDC system which provided better support to COBOL development. Region IV of the Texas Education Agency in Houston was selected as the more preferable conversion site. They were equipped with Version 4 of the CDC COBOL compiler which was a much closer approximation to ANS COBOL. In addition, COBOL was the primary language on their machine which was evidenced by their proficiency in its use. This was demonstrated when the six programs, which did not run at UT, were successfully converted after a single morning's work. After the visits to Region IV, the conversion program was modified to correct approximately 95% of the items that needed to be corrected.

Even though COBOL was designed to be a machine independent language, there are a number of characteristics which are machine-oriented. One of the major problems in the program conversion was that techniques and procedures used on the IBM machine at Texas A&M University for efficiency caused inefficiencies on the CDC and Univac machines. More specifically, a computational item used on the IBM 360 to conserve space and reduce data conversions was very wasteful of space on the other machines which have different word lengths and use different size characters. Also, routine communications

facilities provided by the IBM return code are not available on the Univac or the CDC machines. These items were converted with about 60% efficiency by the conversion program developed by Texas A&M University. The return code was a simple matter compared to the computational items. To solve the problems of computational items, variable lengths were redefined and movement lengths were changed in going to character variables. The efficiency problems were solved by recompiling everything on the IBM system to be exactly compatible with the other manufacturers' systems.

A number of subroutines on the IBM version of DOMONIC were written in assembly language to improve the efficiency of the system. All of these programs were rewritten in COBOL for transportability with the exception of terminal I/O programs which are machine-dependent.

Initial conversion to the Univac computer was begun by using the University of Houston's Univac 1108. Fortunately, that system had upgraded their standard COBOL considerably since the initial language study was made for DOMONIC. Many problems with the Univac version of COBOL were identified during trips to the University of Houston. After analyzing the problems, the programs which had been converted to CDC were able to run on the University of Houston system. A small sample has also been compiled at the Johnson Space Flight Center in Houston with complete success. The COBOL converter developed as part of this contract can be used to move other COBOL programs from IBM to Univac and CDC computers.

Additional documentation aids were added to the DOMONIC system during this contract period. Programs to provide general cross-references were added for COBOL and assembly language programs. Aids that helped to clean

up and reformat programs were added for FORTRAN, COBOL and PL/1. The DAVE system was installed at Texas A&M University to be used in conjunction with the DOMONIC system. DAVE is a large FORTRAN program designed to perform data flow analysis on FORTRAN programs.

Phase II

In this part of Phase 2, a monitor was developed which can gather information that is useful in system improvement, software reliability studies and resource billing. The monitor sub-system was developed using DOMONIC which was possible because the monitor sub-system could be developed after the initial version of DOMONIC was finished.

The monitor collects data using two record types. Information concerning data is collected with the data record and includes user-supplied and automatically-recorded information. User-supplied information consists of flags to turn "on" and "off" the monitoring of commands and the estimated, revised and actual completion dates of data modules. Automatically recorded information consists of command counts, user that entered the module, last user to update, size of module, number of times a module has been edited as well as the date of entry and date last updated. User information collected automatically consists of logging the User ID, total log on time, CPU time used, number of times the user has logged on and the date of the last log-on.

During Phase 2 an extensive effort was made to develop techniques to categorize errors, and establish appropriate quality metrics which would be useful in software reliability models. Chapter III of this report, which was written by Drs. Elliott and Ringer, describes quality metrics,

reliability models, and error categorization. In that chapter, two reliability models which were developed as part of this contract are described. One is a model on which to base acceptance of a system.

Types of errors made in program development in many cases depend upon program complexity. Jean Zolnowski, in Chapter IV, analyzes program complexity and describes vectors which should be considered in measuring complexity. She wrote a number of SNOBOL programs to automatically analyze existing programs to determine their complexity as described in Chapter IV. This work was done in conjunction with her dissertation.

A brief description of the installation of the DAVE system is included in Chapter V which was written by Dr. Fairley.

Very closely related to the reliability of software are the techniques used for testing, validation and verification of software as described in Chapter VI which was written by Dr. Lively. Chapter VI is concerned mainly with the testing aspect of reliability with emphasis on the elimination of errors or bugs in software. The chapter deals primarily with discussions about testing batch programs. Some of these techniques are applicable to time sharing and real time programs, but additional complexities of non-batch programs create a number of other problems.

Chapter VII, written by Dr. Fairley, is included as a summary of modern software design techniques. Chapter VIII suggests future extensions.

3.0 QUALITY METRICS/RELIABILITY MODELS/ERROR CATEGORIZATION

Other chapters will deal with a number of different software techniques designed to improve the quality of a computer software product. This chapter will provide a comprehensive overview of techniques for measuring that quality.

The principal problem with evaluating software quality is that of evaluating any human activity -- namely, that it consists of many varied characteristics that are difficult to quantify. Although it is difficult to describe software quality in general terms, a software quality problem is easily recognized even though it may come in a variety of forms.

Example: A program for computing bi-weekly paychecks is used every two weeks for nearly three years and performs flawlessly. Perversely, on New Year's Eve in 1972 the program generates paychecks nearly every one of which is incorrect. Further investigation shows that the program had been written in such a way as to handle years with no more than 365 days. Since 1972 is a leap year, the program malfunctions. This was a case of a software quality problem due to a lack of functional capability.

Example: An accounting system has been developed and operates flawlessly over an extended period of time. Eventually, however, due to some organizational and product changes it becomes necessary to modify the manner in which some entries are made and to change approximately 15% of the reports that are generated. After spending several weeks in an attempt to understand the logic and structure of the existing programs, the programmer assigned to the task

of making the modifications gives up in disgust and re-writes the system from scratch. This is an example of a software quality problem due to a lack of modifiability.

Example: A set of programs is written to code and decode data prior to and subsequent to transmission over a tape-to-tape transmission device. A subsequent analysis shows that over 70% of the data being transmitted consists of blanks and that if rudimentary data compression schemes were employed the cost of operators and communication lines could be reduced by nearly 60%, far overshadowing the cost of implementing and utilizing the data compression scheme. This is an example of a software quality problem due to a lack of efficiency.

Clearly, these are only a few of the more obvious kinds of quality problems but there are a host of others. In this chapter an attempt will be made to do two things. First, since software quality problems are related to errors, schemes for classifying software errors will be reviewed and a comprehensive program for collecting error data will be described. In addition, some of the existing error data will be summarized. Next, the characteristics of good quality computer software will be described and a number of metrics for measuring software quality will be proposed and evaluated. The quality characteristic which has received the most attention in the literature is that of software reliability, primarily because it is most amenable to mathematical treatment. Reliability measures and models will be discussed extensively.

3.1 Errors

In the past 20 years, millions of computer programs have been written

in this country. In view of this and the fact that each of these programs went through a development phase in which errors were detected and eliminated, it is remarkable to note how little is known about errors as they exist in software products. For example, there are no definitive statistics on the types of errors which occur, there are no reliable statistics on the distribution of errors over time, and there is little known about the basic sources of errors.

This is due to two factors. First, there has not existed any uniformly accepted classification system for software errors. The little data that has been collected is in many heterogeneous forms and does not yield to comparison. Secondly, there has been little effort to collect error data. Software development personnel have concentrated on the operational problems involved in their work and have spent relatively little time analyzing what they have been doing.

This section consists of three parts. First, several of the attempts which have been made to develop error classification systems will be reviewed. Then a data collection program for error data will be outlined. Finally, some preliminary error data which has been collected will be presented.

3.1.1 A Survey of Error Classification Techniques

Rubey, Wick, and Beathley [1968] have prepared a comparison of PL/I with several alternative languages. Preparatory to doing this they developed a categorization scheme which included the following error categories:

1. Computation and assignment statements
2. Character handling statements

3. Sequence control and decision statements
4. System interaction statements
5. Data file and format description statements
6. Procedure, function and subroutine statements
7. Comments
8. Delimiters
9. Labels
10. Punctuation

This categorization scheme is useful for a broad variety of problem oriented languages but it has two deficiencies. First, it assumes a programming language of the general nature of PL/I and would not be useful for other languages such as assembly language or APL. More importantly, however, the categorization is deficient in that it provides for a relatively limited number of types of errors. It, for example, does not include provisions for indicating data preparation errors, keypunch errors, system failures, etc.

A more recent work in the area has been attempted by Ramamoorthy, Cheung, and Kim [1974]. They addressed the area of reliability of large computer programs and in so doing developed a classification scheme by first attempting to develop a comprehensive list of sources of errors and then combined these into classes wherein most common errors can be described.

Their sources of error include the following:

1. Program specification
2. Faulty algorithm design

3. Overlooking special cases for input data
4. Coding errors (including incorrect schematics, language constructs, logic errors, array/overarrayed and so forth)
5. Structural errors (including incorrect flow of control, unreachable program segments, no exit path from segment and so forth)
6. Loop termination
7. Interface errors

Given these sources of errors, the authors then combined these into five general classes consisting of the following:

1. Interface data
2. Sequencing
3. Data Integrity
4. Semantics and language construct
5. Structure and formation

E.A. Young [1970] did a study on the error-proneness in programming as it was manifested in the use of five programming languages, namely ALGOL, BASIC, COBOL, FORTRAN, and PL/I.

He defined errors in terms of the language employed by the user and in terms of his satisfaction with the results. Young supplied instructions, confidential questionnaires, debugging run log blanks and a final questionnaire for each problem. Errors were classified by making use of the information available from the questionnaires at the end of each programmer's debugging process. The errors were coded in terms of the best description

of their primary cause and not in terms of intermediate or final results. The error coding card for the experiment specifically asked for items such as: general cause of the error, specific cause of the error, the number of possible errors which could be classified like this one, system diagnosis of the error, system action, and the number of diagnostics for this error.

For the purposes of Young's study an "error" was defined as "each omission or commission, usually on the part of a programmer, which results in potential or actual computer actions not desirable and/or not acceptable within the programmer's interpretation of the program specifications." Based on this definition and the results of the experiment, the following are the error classes Young came up with (those categories with a higher proportion of errors occurring are noted and an explanation is given):

- Job identification
- Execution request
- External I/O assignment
- Other command language
- Procedure identification
*- Allocation (those parts of the program which explicitly guide the definition of symbols within a program)
- Label, location marker
*- Assignment, Computation (encompasses most direct data manipulation)
- Comment, pseudo-op, no-op
*- Iteration mechanism (repeated use of a small number of instructions)
- Unconditional branch

*- Conditional branch, executing (change course of execution - though there were relatively few occurrences of errors in these, they tended to occur not only out of proportion to the number of conditional statements but these errors also lasted extraordinarily long)

*- I/O formatting (more syntactic problems with this than anything else)

*- Other I/O (what is to be written or read)

- System subprogram invocation

- Other subprogram invocation

*- Parameter/subscript list (mainly miscorrespondence between real and actual parameters)

- Subprogram termination

- Data

*- Vertical delimiter (serve only to indicate statement groupings such as iteration loop or block termination)

- None

The most extensive error categorization scheme that has been developed is that of Amory and Clapp [1973].

The goal of this project was to "provide as extensive a categorization method as possible and then encourage experimenters to use it as a guideline, subsetting and extending it as appropriate to the goals of the experiment." For the purposes of this study, an error is defined as "a conflict between two or more viewpoints which must be resolved" and a bug is in turn defined "as an error which involves software as one of the conflicting viewpoints."

It is most important that an error classification scheme handle errors which cannot be neatly assigned to a single category. So this method uses a set of concurrent dimensions and an hierarchical organization of classes so

that aspects of an error which occur simultaneously can be organized to reflect this fact. Accordingly, error data is organized in five dimensions:

WHERE - covers context in which error appeared

WHAT - describes manifestations of the error

HOW - identify specific code or data which was incorrect

WHEN - gives development stage at which error occurred

WHY - presents reason(s) for making the error.

Each of these dimensions is presented as a set of categories, sub-categories, and subdivisions, as appropriate to the dimension's orientation and prospective use.

An analysis and investigation of error distributions in system programs was recently reported on by A. Endres [1975]. The fact that he actually collected data and analyzed it is important in itself, although the class of programs he analyzed is not like most application programs and therefore his classification schemes are not useable on a general basis.

Errors analyzed were those discovered during internal tests of components of the operating system DOS/VS. A typical project activity consisted of changing or adding about 50 instructions in an existing module of about 200 instructions. Errors are detected in code which is a mixture of "old" and "new" programming styles. Also, a record of errors was kept only for the formal test period of 5 months which is only a part of the complete test cycle. So a nicely detailed error/change history is not available for the modules.

Also, irregularities or errors in the system were documented via their external manifestation by the testing group. This, in turn, was passed on

to the original development group which analyzed the problem, classified it, and filled in additional background information. So the classification system was developed after the errors had been found and therefore relied heavily on interpretation of errors by those not "committing" them.

Errors found were eventually classified into the following groups: program errors, machine errors, user or operator error, suggestions for improvement, duplicate (of a previously identified program error), and documentation error. These classifications are the result of a specific group of questions asked at a specific time based on a specific type of program. Unfortunately, this program type is a systems program composed of both old and new code and the time span does not encompass total program development. Therefore, while the author's discussions are useful and perhaps some of his techniques are worthwhile, his data is still not that which is needed.

An experiment to collect data on types and frequencies of errors was conducted at Bell Laboratories by Shooman and Bolsky [1975]. This also was an initial attempt such as Endres' [1975] to examine the feasibility of collecting error data and to set up an error collection system which would be tolerable to programmers involved and yet collect useable and worthwhile data. The authors chose a program of about 4000 machine words on which to investigate the number and types of errors occurring in its test and integration phase. So they, too, restricted data collection to a certain time period in program development and do not attempt to get a full error/change history of the program.

The objectives of this study were: to design a useable and useful data

collection form; to get information on how this form affected those who had to make use of it; to compare overall error count results with other work done; and to obtain whatever information possible on resource expenditure during the testing and integration phase.

The article then was a discussion of these above topics and points out the deficiencies in data collection forms and how very flexible these indeed must be. The study itself was a reasonable beginning in an area that has had much discussion but has had minimal studies such as this to provide worthwhile information. Its two obvious deficiencies were a concentration on one specific program type which had no major changes in functional specifications and a resulting lack of a spectrum of error types, and its concentration on a particular phase of the programming effort.

3.1.2 A Prototype Software Development Error Data Collection System

This section describes a prototype system for the collection and use of software error data. Its operation is symbolically described in Figure 1. Essentially it consists of five components.

1. Error data collection forms.
2. Data bank containing error data accumulated from these forms.
3. A computer program for updating the data bank and performing analyses of the error data.
4. A set of reports generated from the error analyses procedure.
5. An inquiry program whereby a manager can inquire into the error data bank and extract facts on an ad hoc basis.

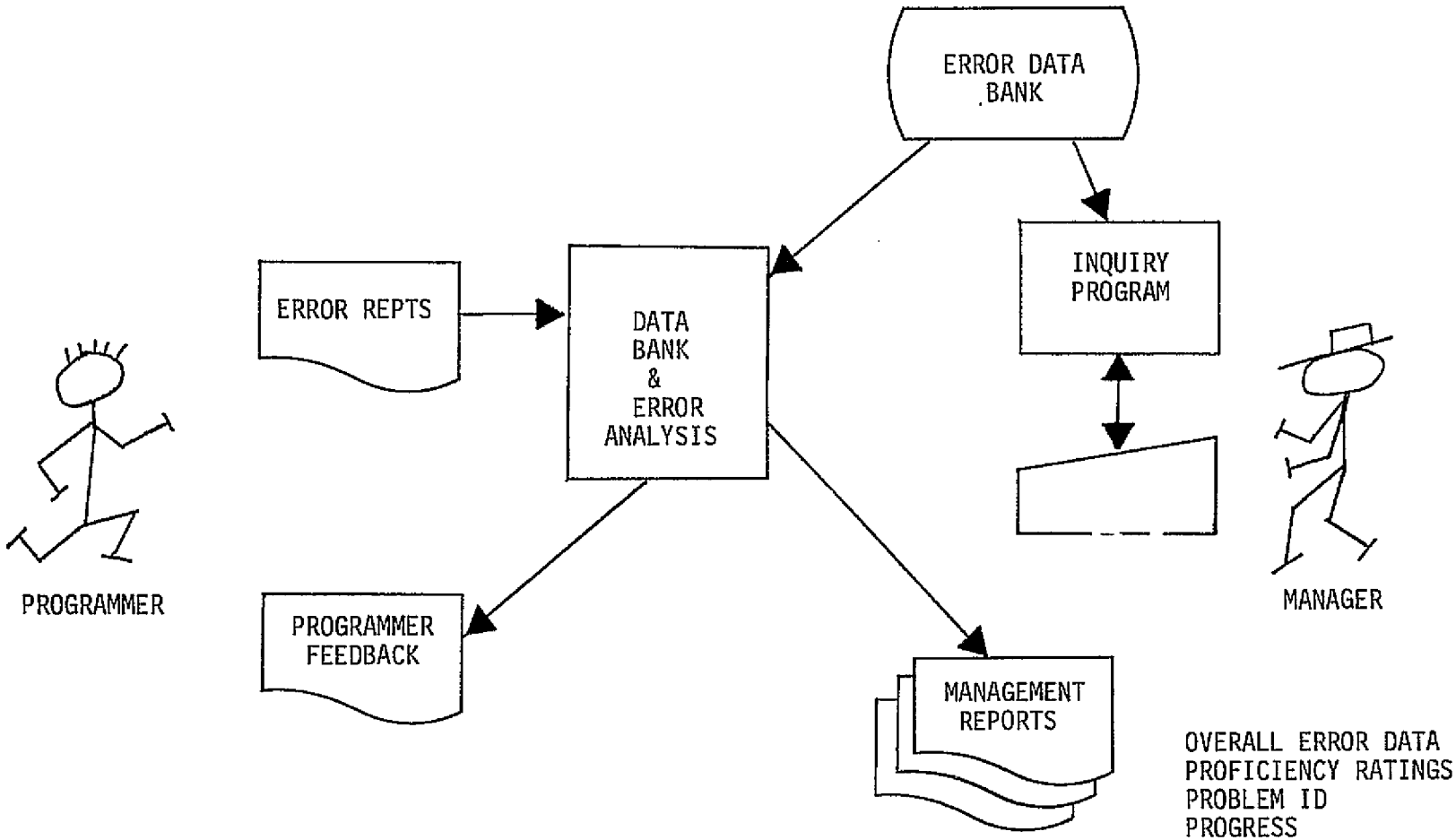


Figure 1. PROTOTYPE ERROR REPORTING SYSTEM

The data collection forms are shown in Figures 2 and 3. The "background sheet" identifies the programming task to be performed and the environment in which it is performed. It contains information describing the type of program, language size, and hardware and very briefly describes the programmer's experience both as a programmer and in the application area. This form is to be used in conjunction with the form shown in Figure 3 entitled Error Reports. This report contains a line on which the result of each run from the start of debugging until program delivery is recorded. The programmer specifies a good deal of information concerning the run including the manifestation of the error, when it occurred, why it occurred, the programming cause of the error and so forth. In addition, some general information consisting of the severity of the errors and how much of the program has been changed since the previous run is also indicated on this form.

The error data bank essentially consists of an indexed file containing all of the run data that is on all of the background sheets and each run from the error report. The data bank is indexed in such a way that it can be conveniently accessed from a terminal to produce a variety of statistical information.

When new data are added to the data bank, a number of reports are automatically produced. One is a program feedback report which is produced for the programmer. This report is essentially a summary of all of the runs made for a particular task and provides him with a convenient record of activities.

The other reports are intended for the programming manager. One of the reports, "Overall Error Data" summarizes the error data recorded for the

BACKGROUND SHEET

1. LANGUAGE

☐ Assembler	☐ PL/I	☐ List Processor
☐ FORTRAN	☐ APL	☐ Data Management
☐ COBOL	☐ Simulation Language	☐ Other

2. TYPE OF PROGRAM

☐ System	☐ Financial/Accounting	☐ Other
☐ Scientific	☐ Data Manipulation	

3. PURPOSE OF PROGRAM

4. TOTAL MODULE SIZE (lines of code)

☐ less than 100	☐ 500-1000	☐ over 2000
☐ 100-500	☐ 1000-2000	

5. HARDWARE NECESSARY

☐ Tape Files	☐ Card Files	☐ Other
☐ Disk Files	☐ Plotter	
☐ Print Files	☐ Terminal	

6. PROGRAMMER EXPERIENCE

... in Application Area	... as Programmer
☐ less than 6 months	☐ less than 6 months
☐ 6 to 18 months	☐ 6 - 18 months
☐ over 18 months	☐ over 18 months

7. COMMENTS

Figure 2. BACKGROUND SHEET

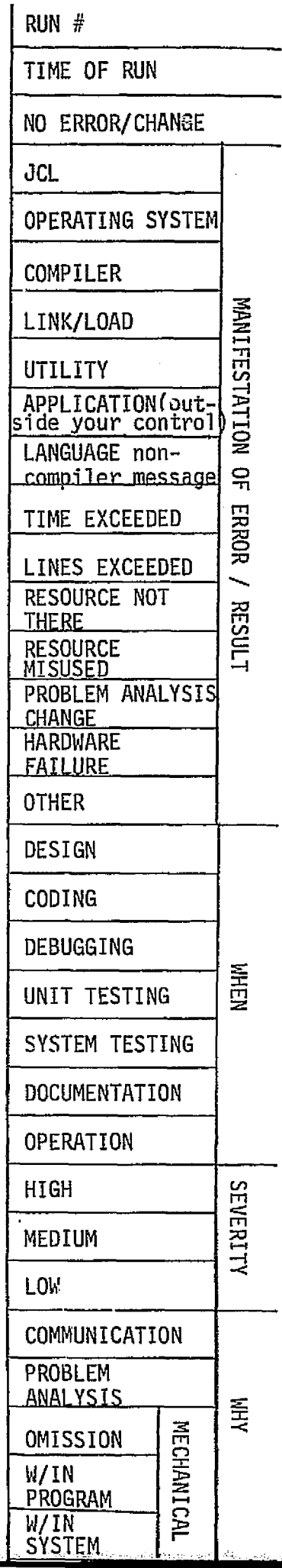


Figure 3.

(Continued next Page)

PROGRAMMING CAUSE
INVALID JCL
ALLOCATION
LABEL (location marked)
ASSIGNMENT COMPUTATION
COMMENT PSEUDO-OP NO-OP
ITERATION-MECHANISM
UNCONDITIONAL BRANCH
CONDITIONAL BRANCH
I/O FORMATTING
OTHER I/O
SYSTEM outside your control SUBPROGRAM INVOCATION
SUBPROGRAM INVOCATION
PARAMETER/ SUBSCRIPT/ LINKAGE LIST
SUBPROGRAM TERMINATION
DATA VALIDITY
OTHER
NONE

APPROXIMATELY HOW MANY LINES AFFECTED + , - ,

INPUTS CHANGED FROM LAST RUN?
NEW SET
MAJORITY CHANGED DUE TO VALIDITY
FEW CHANGED-INVAL-DATA
INCREASE TO INPUT SET
DECREASE THE SET
OTHER

Page ____ of ____

Figure 3. (continued)

reporting period and compares it with the data that are currently in the data bank. The format of this report is described in Figure 4. It shows the number of tasks that are active, the number of runs that have been made in each category from design through operation, and, for each error type, it shows the frequency of occurrence of that error, the percentage of a total number of runs made for that period which exhibited that error, and a comparison of the percentage of all runs which exhibited the error with those recorded in the data base. This report is intended to give the manager a feel for the overall activity that is taking place in his shop and any indication of any changes which may be taking place in the types of errors which are being recorded.

The next report is entitled "Problem Area Identification" and is shown in Figure 5. This report is an exception report which identifies any significant deviations from normal expected conditions. For example, errors due to invalid JCL during system testing would normally have very few occurrences in the data bank. Should a large number of JCL errors be reported for a program which is in system testing, this deviation from the normal would be detected and reported by this program. Similarly, excessive hardware failures, data preparation, or design errors could be brought to the manager's attention in a similar fashion.

The final report is a "Progress Report" which is depicted in Figure 6. This report develops, for each task, a progress report which shows the number of runs and the status of the task as indicated by a summary of the errors of different types. Given an appropriate model for determining

SAMPLE MANAGEMENT REPORT

OVER-ALL ERROR DATA

DATE 12/11/75

TOTAL TASKS REPORTING 16

RUN STATUS REPORT

PHASE	NUMBER	PERCENTAGE
DESIGN	10	2
CODING	32	7
DEBUG	107	24
UNIT TEST	84	19
SYSTEM TEST	16	4
DOCUMENTATION	2	0
OPERATIONS	198	45
TOTAL	439	

RUN:TASK RATIO 27.4

ERROR SUMMARY

ERROR CLASS	NUMBER	% OBSERVED	% EXPECTED
A	63	24	28
B	42	16	14
C	11	4	8
D	107	41	28
E	21	8	12
F	18	7	10
	262		

TASK SUMMARY

TASK	STATUS	RUNS
FDFRACCT	UNIT TEST	34
	SYSTEM TEST	11
FDFRBUDG	DEBUG	11
PHYSINVT	CODING	7

etc.

Figure 4.

SAMPLE MANAGEMENT REPORT

PROBLEM AREA IDENTIFICATION

DATE 12/11/75
TOTAL TASKS 16
TOTAL RUNS REPORTING 439

EXCEPTIONAL ERROR FREQUENCIES - ALL TASKS

ERROR CLASS	MEANING	OBSERVED	EXPECTED
D	JCL	41	28

EXCEPTIONAL ERROR FREQUENCIES - BY TASK

TASK	ERROR CLASS	MEANING	OBSERVED	EXPECTED
FDFRACCT	D	JCL	16	5
	F	DATA PREP	5	1
FDFRBDGT	B	SYNTAX	12	3
	D	JCL	8	2
PHYSINVT	D	JCL	13	3

Figure 5.

Problem Area Identification Report

SAMPLE MANAGEMENT REPORT

TASK	WEEK	RUNS
FDFRACCT	04/16	D D
	04/23	D D D D
	04/30	# # # # # # # # # # # # # # # # # # #
	05/07	# #
	05/14	# # # # U U U U U
	05/21	U U U U U U U U U U U U
	05/28	U U U U U U S S S S S S S S S S S S S
FDFRBDGT	05/07	D D D # # # #
	05/14	# # # # # # # # # # # # # #
	05/21	# U U U
	05/28	U U U U

KEY D = DESIGN, # = DEBUG, U = UNIT TEST, S = SYSTEM TEST,
X = DOCUMENTATION, O - OPERATION

Figure 6.

Progress Report

programming proficiency, this report could also contain a proficiency index for each programmer who is associated with that particular task.

3.1.3 Errors in Software Development - An Empirical Study

Figures 2 and 3 of Section 3.1.2 present an error categorization form which was designed for future use with an automated system (DOMONIC) and more immediate usage in the prototype error data collection system of the previous section. An empirical study (a manual data collection scheme) was therefore undertaken to test the feasibility of using this form. Not only were the programmers' reactions to filling out the form studied but also examined was the effectiveness of this form in providing reasonable categorizations for a programmers' errors or changes in coding. Essentially, then, this initial study was not so much oriented toward collecting a large amount of data as it was concerned with testing and reinforcing an error categorization scheme for the future collection of large amounts of data.

Accordingly, forms were given to a small test group of data processing programmers for use from the start of their program design through the program production phase. The form was found to be relatively easy to use. However, this did not insure that it was well filled out. The programmer was asked to use the form after each run of a program, regardless of whether there were errors/changes or not. If the group studied is any indication, the majority did not follow this rule and exact data was collected only from those who would be conscientious under any circumstances. The major problem is that most people need pressure brought to bear on them and need an incentive to cooperate -- they have to realize results from this extra work of

using a form or cooperation is minimal. This emphasizes the fact that results in the form of reports which would be realized from the scheme of Section 3.1.2 and the DOMONIC system would provide sufficient feedback to ensure programmers' cooperation.

The form itself was quite useable for the programmers and relatively easy for them to maintain. Examples of summaries of results from two COBOL programs developed during the project each in the same size group (100-500 lines) follow:

	Example A	Example B
Number of Runs	24	11
Number of Errors/Changes	38	16
Average Number of Runs Per Day	1.9	1.1
Average Number of Errors Per Run	1.6	1.5
Manifestation of the Error		
1. Compiler	4	
2. Link/Load	1	
3. Problem Analysis	32	16
4. No Error/Change	1	
When Error Occurred		
1. Coding	2	
2. Debugging	4	
3. Unit Testing	31	13
4. System Test	1	3

	Example A	Example B
Severity of Error		
1. High	8	
2. Medium	14	5
3. Low	15	10
Why Did Error/Change Occur		
1. Problem in Functional Analysis		1
2. Omission	6	1
3. Mechanical within Program	32	14
Program Cause of Error		
1. Label	2	
2. Assignment/Computation	11	3
3. Conditional Branching	7	5
4. I/O Formatting	7	3
5. Subprogram Invocation	1	2
6. Data Validity	2	1
7. Iteration Mechanism	3	1
8. Other	3	
Average Number of Lines Affected by an Error/Change	8.5	3.5

Several of the categories were seen to be insufficient for categorizing a specific area. For instance, both examples above indicate that the manifestation of most errors was seen through problem analysis. The programmers' involved felt that this general category, problem analysis, was not at all

specific enough to pinpoint where the error appeared, i.e. it did not sufficiently reflect the environment of the error. However, the format of the form, since it is a checklist, has proved amenable to expansions and revisions derived from information given by participating programmers.

Problems such as these - feedback from the actual use of the data collection form - have provided the information necessary for an evaluation of the run categorization scheme and for confidence in its merits for future full-scale usage.

3.2 Software Quality Attributes

There exists no widely accepted definition for software quality. The emphasis which an individual places on software quality varies a good deal depending upon the situation.

A person using a statistical program for a particular problem may be particularly interested in accuracy.

Someone given the task of modifying an existing program may place primary emphasis on the characteristics of the documentation.

For a real-time control system, quality may be equated with speed.

A software salesman might equate quality with generality.

Depending upon one's point of view, any number of quality attributes might be of primary importance. The list below includes some that have been suggested in the literature.

Logical correctness	Ease of use
Ease of modification	Conciseness
Completeness	Augmentability

Accuracy	Reliability
Structuredness	Human engineering
Modularity	Testability
Device Efficiency	Speed of execution
Legibility	Documentation

3.2.1 Overview of Software Quality Metrics

It is abundantly clear from the foregoing discussion that there exists no agreed-upon definition of software quality. This is, of course, not surprising because qualitative judgments are never straightforward. They are inextricably interwoven with the objective point of view and perspective of the person making the judgment. In those cases where quality indicators are selected and used, there is invariably a good deal of discussion regarding the appropriateness of those indicators.

On the other hand, people do exhibit a remarkable uniformity in making quality assessments in a wide variety of areas.

In the arts, there is a broad base of agreement as to which books are good, which movies are entertaining, which paintings are striking, even though personal preferences vary widely.

A similar situation exists in the evaluation of personal performances. Managers, teachers, housewives, or persons in any field of endeavor exhibit different personal styles in the performance of their tasks. Even giving allowance for personal preferences, there will normally be a broad consensus as to who is a good manager and who is not, who is a good teacher and who is not, and so forth.

Similar situations exist in almost any area. People evaluate shares of common stock, used cars, pieces of property, tools, and so forth in a most informal manner, but they exhibit a great deal of uniformity in making their assessments.

Selection differences arise more from the fact that quality normally has many dimensions, and there are personal preferences for one dimension or another rather than from the lack of uniformity in the quality assessments. For example, an investment has dimensions which include risk, potential return and capital requirements. Knowledgeable investors make different investments because of individual preferences for different combinations of risk, potential return, and capital expenditure rather than because of differences in evaluation of these factors.

In spite of the fact that humans exhibit a remarkable uniformity in quality assessments, the development of a definition of software quality (or of any other kind of quality) is a hopeless task, and the development of the theoretically sound metric to measure software quality is a similarly hopeless task. Both involve a level of model building which is beyond the scope of today's technology.

On the other hand, the real question is not whether one can build a theoretically sound model, but whether one can build a useful model for measuring software quality. There is good reason to believe that a useful model of software quality can indeed be constructed because useful quality models have been developed in many areas.

Most quality models make use of some sort of scoring approach. A list of representative quality factors is assembled, and the object being assessed is evaluated as to the presence or absence of these quality factors. Points or weights may be attached to each factor or a simple summation may be used. This type of procedure is common in personnel assessments, real estate appraisal, used car evaluation, and many other similar quality evaluation problems.

Two assumptions are implicit in the approach. The first is that the factors sampled constitute a representative sample. There are many, many factors which make up quality. Only a few of these factors are selected, but if they are in fact quality factors and if they are selected in an unbiased fashion, one can estimate with reasonable accuracy the proportion of all of the quality factors which exist in the object under consideration.

There exists a substantial body of statistical literature on sampling theory, e.g. Freund [1971].

The second assumption implicit in the use of this approach is that it is going to be a little wrong most of the time (and probably mostly wrong part of the time). Although the use of imperfect measuring devices seems abhorrent to most people, they are used all of the time. Carpenters do not use micrometers because folding rulers are easier to use and sufficiently accurate. Employers of laborers seldom use psychometric testing because they do not need that much information about potential employees.

Also, one has to consider the alternative to imperfect measuring devices - in many cases it is not having any measuring devices at all.

In many situations, one can make a case for using only factors which can be determined precisely in any evaluation. A promotion system in which the only consideration is time and grade is popular among some people for this reason.

Because software quality cannot be precisely measured, it has almost invariably been ignored as an evaluation parameter. Ignoring software quality is a luxury that we can no longer afford. A technique for measuring software quality is badly needed for many reasons. It is needed in order to properly assess programmer productivity. When given a choice between alternate pieces of software one needs a technique to assist in choosing among them. A measure of software quality is needed to provide a level of quality control in an industry in which quality is fast becoming a scandal.

Given that the development of quality measures is a useful undertaking, even though any such measure must of necessity be somewhat imperfect, it is useful to consider the characte istics of a good quality metric.

First, any metric should be consistent with what constitutes a consensus judgment of persons knowledgeable in the field. That is, it should not be counter-intuitive to informal measures.

Second, the measures should be user-independent. The application of the metric should be essentially independent of the person who is applying it. It would be preferable to have all software quality metrics machine-derivable but that does not appear to be practical at this stage in their development.

Third, the measure should be easy to evaluate. It should not, for example, take as long to evaluate a piece of code as it does to write it.

One thousand lines per man-hour would appear to be a reasonable goal for a level of effort.

Fourth, any quality metric should be dimensional, and the various dimensions of the metric should be discernible. Depending upon one's objective, one has a different perspective on quality. Someone purchasing a car may be primarily interested in economy, or in comfort, or in maintainability. Similary, someone procuring a piece of software may be interested in portability, or in modifiability, or in efficiency, or in some other factor. The extent to which each of these factors is present should be discernable.

Finally, the dimensions must be meaningful. One approach to devising dimensional metrics might be to assemble a large number of factors, observe the present or absence of these factors in a large number of programs, and statistically cluster the factors into subsets. While the subsets so derived will be statistically cohesive, they will not normally be meaningful in the usual sense, and their useability is therefore impaired.

A metric should have a meaningful and useful range. Ranges form 0 - 1 or from -1 to +1 or from 0 to 100 are normally used.

The literature contains numerous references to characteristics which will be found in quality software. These characteristics have been lumped into the following six software quality attributes as shown in Figure 7.

Efficiency - measures the effectiveness with which resources including time and storage are used.

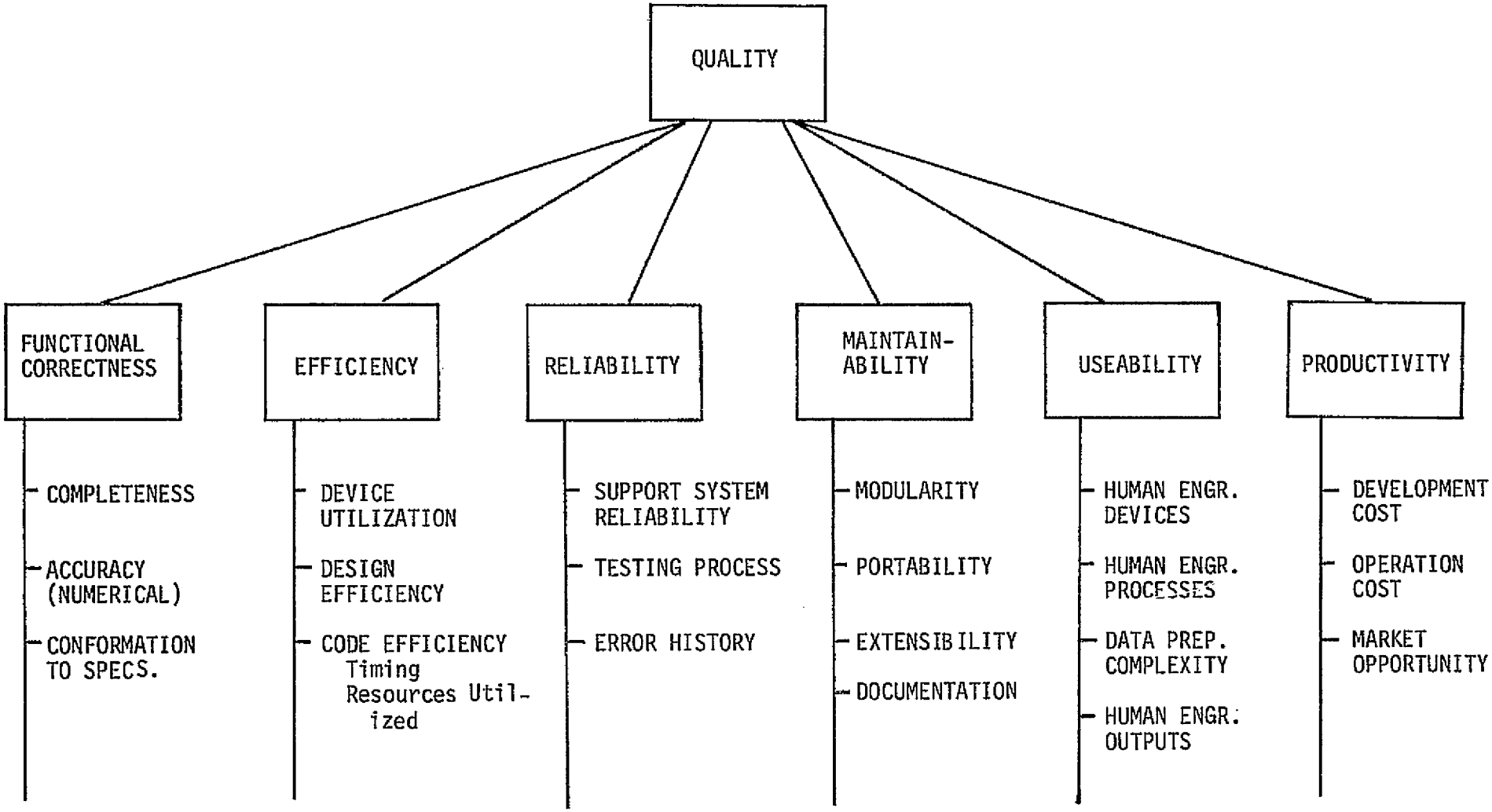


Figure 7.

Quality Attributes and Subattributes

Reliability - measures the likelihood of error-free performance over a given time period.

Modifiability - measures the level of difficulty involved in keeping a program operational over its expected life. It involves documentation portability, extensibility, etc.

Useability - measures the degree to which a user can easily and accurately use the system. It is related to the human engineering of the devices and processes with which a user interacts with the system.

Functional Correctness - measures the degree to which a program's capabilities coincide with the program designer's concept of what those capabilities should be.

Productivity - measures the benefits that the user of the software product can reasonably expect to accrue in relation to the cost of the product and other market opportunities.

The following sections contain discussions of these quality attributes and describe metrics which have been devised for evaluating some of them. These metrics may be used individually or can be combined into a single figure of merit. In developing these metrics, we have drawn heavily on the ideas of Rubey, Hartwick and Dean [1968] and on Boehm et.al. [1973].

It must be emphasized that these metrics are based on weighted samples of factors indicative of each attribute and, as such, are subject to error. Some evidence has been developed, however, that they are consistent with knowledgeable judgments and are reasonably user independent. For these reasons, it is felt that they have some promise of being useful, if not as absolute indicators, as indicators of possible strengths or weaknesses in a

program which bear further examination.

3.2.2 Modifiability

Based only on an examination of a program and its documentation, the quality attribute to which a manager is most likely to give the heaviest weight is modifiability. This term has been chosen rather than the more usual term, maintainability, because maintainability as it is usually defined, relates to the time required to return a system to an operational state (in an unchanged environment) once it has malfunctioned. In a software system, however, the system does not malfunction, rather the environment changes - by changing the functional requirements of the system or by altering the computer components - and it is necessary to modify the system to operate in the new environment, hence, modifiability instead of maintainability. Modifiability has five sub-attributes.

Internal Documentation - is a measure of the readability of the code.

External Documentation - is a measure of the value of the program documentation (external to the code).

Modularity - is a measure of how well the program has been broken into small functionally independent modules.

Portability - is a measure of how easy the program would be to move to another environment.

Extensibility - is a measure of how easy it would be to extend the functional capability of the system.

3.2.2.1 Internal Documentation - is synonymous with good programming practice, but it is often missing. McCracken and Weinberg [1972] have written an excellent guide to writing "readable" FORTRAN programs. Their suggestions are applicable, with some modification, to all procedure-oriented languages.

Comments contribute greatly to the readability of any program. A number of comments approximately equal to the number of operational statements is not excessive. The comments should describe, in some detail, the functions of each module and its relationship to other parts of the program. Comments should also indicate the authorship of each module.

Descriptive variable names and identifiers contribute greatly to the readability of a program.

A number of editing practices can be used to enhance readability. Indentation to define the ranges of loops helps. Sequence numbering the statements and the statement labels helps.

Finally, a programmer can do a great deal to enhance the readability of his code by using simple coding structures. From the point of view of internal documentation, an effort should be made to utilize straightforward code even at the expense of efficiency. Experiments with structured programming indicate that restricting coding structures to a few relatively simple structures and minimizing the use of go-to statements greatly enhances the code readability.

3.2.2.2 External Documentation - (documentation other than the source listing) is also an important constituent of modifiability. Many installations have documentation standards and, where these exist, documentation should conform to this standard.

Program _____ Subject#_____

Attribute Modifiability Sub-Attribute Documentation (Internal)

	0	1	2	3	4
Are comments used extensively? Virtually none-0; 25% of source-2; 50% of source-4		X		X	
Are descriptive variable names used? Almost never-0; Almost always-4; Not applicable-4		X		X	
Is the function of each module described? No-0; Sometimes-1; Always-2; Not applicable-2				X	X
Are inter-relationships among modules clearly specified? No-0; Sometimes-1; Always-2; Not applicable-2				X	X
Are simple coding structures employed? No-0; With few exceptions-1: Generally-2				X	X
Is the source code sequence-numbered? No-0; Yes-1			X	X	X
Are statement labels sequentially numbered? No-0; Yes-1; Not applicable-1			X	X	X
Have indentations been used to improve readability? Usually not-0; Usually-1; Not applicable-1			X	X	X
Does the source code contain the author's name and date of last revision? No-0; Yes-2		X		X	X
How extensively are GO-TO's used? Over 10% of statements-0; Fewer than 10% but more than 5% of statements-2; Less than 5% of statements-4		X		X	

This documentation should contain both a system flowchart and a logic flowchart or some equivalent device (e.g. HIPO's). Logic flowcharts, to be of any utility, must be "higher level" flowcharts which describe the logic of the flow without single statement detail.

Similarly, the documentation should contain a "glossary" of all variables used in the program and a good statement describing the program's capabilities and limitations.

Instructions for preparing input to the program and interpreting output should also be included when appropriate. Sample input and output together with run instructions are a necessary part of the documentation.

Finally, detailed descriptions of all the data sets and instructions for their management should be included in the documentation.

3.2.2.3 Modularity - contributes to modifiability by making it easy to isolate specific functions for maintenance purposes. A program written with a high degree of modularity will have some distinct characteristics.

It will, of course, be segmented into a number of small functionally-defined "chunks." The proper size for these chunks has been the subject of some discussion, but most programmers agree that they should have an upper limit of about 60 statements including comments (one page of computer output).

The types of statements employed in a program are changed significantly as the result of this modularity. Except at the bottom level, most modules consist of a large percentage of module calls rather than other operational statements. The level of nesting of individual statements is significantly increased.

Program _____ Subject#_____

Attribute Modifiability Sub-Attribute Documentation (External)

	0	1	2	3	4
Does a logic flow-chart exist? No flow-chart-0; Autoflow (1:1)-1; Good quality hand-drawn-3; Autoflow chart or equivalent-4; deduct 1 for non-std symbols; deduct 2 for not current			X		
Does a system flow-chart exist? No-0; Yes-2; Not applicable-2		X		X	X
Do instructions for data preparation exist? None-0; Minimal-2; Good-4; Not applicable-4		X		X	
Does a "Glossary" exist? No-0; Yes-1			X	X	X
Do run instructions exist? No-0; Minimal-1; Good-2; Not applicable-2				X	X
Does the documentation include a statement describing the program's capabilities and limitations? No-0; Minimal-1; Good-2; Not applicable-2				X	X
Do descriptions of all the program's files exist? No-0; Some-1; All-2; Not applicable-2				X	X
Are sample inputs and outputs available? No-0; Yes-1; Not applicable-2				X	X
If documentation standard exists, does this documentation confirm to the standard? No-0; In some respects-2; Yes-4; Not applicable-4		X		X	
Do instructions for interpreting output exist? None-0; Minimal-1; Good-2; Not applicable-2				X	X

If modular programming is to be effective, the interfaces between the module must be simple. If one merely breaks the code into pieces and passes all the data from one piece to the next, nothing is really achieved. In general, the fewer data elements passed from one module to another, the better.

3.2.2.4 Portability - is an important subattribute of modifiability only if the program must be moved to a different machine or machine configuration.

The programming language selected is the major factor in portability. Assembler code greatly restricts portability as does the use of any language not generally available on a variety of machines. When higher level languages are used, it is important to indicate non-standard language features, machine dependent features, etc.

The utilization of local subroutines significantly impacts program portability. Similarly, the use of devices such as A-D converters, special terminals, etc. which are not generally available may impact portability.

Although many languages are externally independent of the word size of the machine on which they are implemented, computations that occur in the program may be affected by word size and this aspect must be carefully considered.

Finally, portability often implies changing the core available in which to execute the program. If the program is not overlayed already, or if it is very tightly overlayed a reduction in available core may make movement to a smaller machine virtually impossible.

Program _____ Subject# _____

Attribute Modifiability Sub-Attribute Modularity

	0	1	2	3	4
What is the average number of statements (excluding comments) per module? 0 to 40 - 4 pts; 4 to 80 - 2 pts; over 80-0 pts		X		X	
Randomly sample 10 statements. What is the average "level of nesting" 0 to 1 - 1 pt; 1 to 2 - 2 pts; 2 to 3 - 3 pts; 3 to 4 - 4 pts	X				
Randomly sample 10 modules. What is the average number of data elements passed to and from the module? (An array is a single element) 0 to 5 - 4 pts; 5 to 10 - 2 pts; over 10 - 0 pts.		X		X	
Randomly sample 10 modules. What is the average of the ratios of module calls: total statements? < .05 - 0 pts; between .05 and .20 - 2 pts; over .20 - 4 pts.		X		X	

Program _____ Subject# _____

Attribute Modifiability Sub-Attribute Portability

	0	1	2	3	4
Does the program make use of devices which are not available on a number of machines (e.g. data cell)? Yes-0; No-2; Not applicable-2		X		X	X
Does the program include assembler code? all or most - 0; some - 2; none - 4; Not applicable - 4		X		X	
Is the programming language generally available? Most manufacturers - 4; More than one manufacturer - 2; One manufacturer - 0; Not applicable - 4		X		X	
Is the program designed in such a way that the core requirements could be reduced? Yes, with great difficulty-0; With some difficulty-2; Yes-easily-4; Not applicable-4		X		X	
Are computations independent of word size? No-0; Most-2; Yes-4; Not applicable-4		X		X	
Does the program make use of subroutines which are "local"? Extensively-0; Some-1; No-2; Not applicable-2				X	X
Are special machine-dependent features and non-standard language constructs indicated by comments? Never-0; Sometimes-1; Always-2; Not applicable-2				X	X

3.2.2.5 Extensibility - is an important aspect of modifiability. The most common modification to a program is the addition of some new capability. This may involve adding a capability for handling additional data (for example: processing additional inventory codes in an inventory system) or it may involve adding functions (for example: adding a tax withholding module to a payroll system).

Some extensions can be handled in an almost trivial fashion when room for expansion has been provided in appropriate arrays, tables and data sets. Limiting constants for these items should be in symbolic form, rather than in absolute form.

The primary limitation to extensibility is the total utilization of some resource by the existing program. When any device is used to its absolute capacity, for example a disk, it is impossible to expand the capability of the program without a substantial effort.

Other dangerous situations involve programs which leave insufficient unused memory to allow extension without reorganization or which have response times at or very near the maximum allowable, thereby leaving no room or time for the addition of added functions.

3.2.3 Efficiency

As an attribute, efficiency is difficult to evaluate because it has several subattributes which are to some extent conflicting. For example, one might look for efficiency in core utilization or efficiency in operation speed. But, there is usually a trade-off in which one can increase operation speed by using additional storage.

Program _____ Subject#_____

Attribute Modifiability Sub-Attribute Extensibility

	0	1	2	3	4
Does there exist sufficient unused memory so that additional functions could be added without reorganizing the entire program? No-0; No-but reorganization would be simple-1; Yes-2				X	X
Would additional functional capability degrade processing times for existing functions to an unacceptable level? Yes-0; Perhaps-1; No-2; Not applicable-2				X	X
Are any system components currently operating at their absolute capacity? e.g. is a DASD file currently limited by the capacity of the disk? Yes-0; One-1; More than one-2; Not applicable-2				X	X
Have constants containing system parameters been indicated symbolically rather than explicitly? Seldom-0; Usually-1; Always-2; Not applicable-2				X	X
Has room for expansion been provided in arrays, tables, and data sets? Seldom-0; Usually-1; Always-2; Not applicable-2				X	X

Similarly, efficiency conflicts with several of the other quality attributes. For example, modularity contributes to modifiability but it comes at some cost both in terms of space and time.

For purposes of this discussion, efficiency has four sub-attributes.

Execution Speed relates to the utilization of coding practices which result in fast running codes.

Core Utilization relates to the utilization of coding practices which result in very compact code.

File Utilization relates to the organization and utilization of files and their effect on processing time and trial space.

Overall Processing Organization relates to major design decisions usually made early in the system design effort which impacts program efficiency.

3.2.3.1 Execution speed enhancements which can be affected by coding practices are, for the most part, language dependent. In this discussion, emphasis will be on FORTRAN coding conventions which enhance execution speed, although many of them are equally applicable to other languages.

The most critical statements relative to execution speed are input-output statements. Execution can be greatly speeded up by minimizing the number of READs or WRITEs in a program, minimizing the number of items in I/O lists, or using unformatted files for temporary files.

The next critical aspect is probably subroutine calls. Execution speed can be facilitated by minimizing the number of calls and by minimizing the number of common blocks that are used in the program.

In addition, a number of very simple practices can be followed with good results. A conscious effort should be made to minimize the type conversions required in computation. For example, the execution time of the loop:

```
      DO 1 I=1,1000
    1 X(I)=I
```

can be greatly reduced by re-writing it as:

```
      Y=1.0
      DO 1=I,1000
      X(I)=Y
    1 Y=Y+1.0
```

Similarly, a conscious effort should be made to utilize the data types with the fastest execution speeds (e.g., INTEGER instead of REAL) and to utilize the operatives which will execute faster when a choice is available (e.g., use X+X instead of 2*X). Some reductions in speed can be effected by judicious nesting. For example, the loop (a) below:

(a)

```
      DO 1  I=1,100
      DO 2  J=1,30
      DO 3  K=1,3
            .
            .
            .
    3 CONTINUE
    2 CONTINUE
    1 CONTINUE
```

(b)

```
      DO 1  K=1,3
      DO 2  J=1,30
      DO 3  I=1,100
            .
            .
            .
    3 CONTINUE
    2 CONTINUE
    1 CONTINUE
```

will incur more than 50% more overhead in the looping mechanism than loop (b) which is equivalent for most purposes.

In some compilers, significant reductions in execution speed can be achieved by paying attention to the way expressions are written or the number of times they require evaluation. Subscripts can be evaluated much more quickly if written in one of the "preferred" forms. The loop (a) is much less efficient in terms of execution time than (b).

```
          DO1  I=1,123                Y=(Z+T)/F
(a)       1  X=(Z+T)/F+1     (b)      DO 1  I=1,123
                                      1 X=Y+I
```

There are numerous similar things that can be done. As indicated previously, there are some trade-offs between speed and space and between speed and some of the other quality attributes. Although a programmer may not desire to optimize all of a program, he should seriously consider optimizing those portions of his program that are used repeatedly.

3.2.3.2 Core utilization may or may not be an important quality factor, depending upon the facilities available and the charging algorithm. Unless there is some cost differential based on core used or unless the amount of main storage is a limiting factor, there is little incentive for minimizing its utilization. Indicators of the attention which has been given to minimizing core use include the following.

Dramatic decreases in storage requirements can be achieved by properly segmenting the program so that all of it does not need to reside in core

Program _____ Subject# _____

Attribute Efficiency Sub-Attribute Execution Speed

	0	1	2	3	4
Are the number of I/O statements minimized? e.g. where possible, have multiple READ's been combined? Always-2; Sometimes-1; No-0; Not applicable-2				X	X
Has an attempt been made to minimize the number of items on I/O lists? Always-2; Sometimes-1; No-0; Not applicable-2				X	X
Have frequently executed routines been optimized? Always-4; Sometimes-2; No-0; Not applicable-4		X		X	
Are expressions written in such a way as to minimize the number of data-type conversions? Always-4; Sometimes-2; No-0; Not applicable-4		X		X	
Are the data types with the "fastest" execution speeds used? (e.g. integer instead of real) Always-2; Sometimes-1; No-0; Not applicable-2				X	X
When possible, are loops nested in such a way as to minimize the execution frequencies? Always-4; Sometimes-2; No-0; Not applicable-4		X		X	
Has the number of COMMON Blocks used been minimized? Yes-1; No-0; Not applicable-1			X	X	X
Have the number of execution-time evaluations of expressions been minimized? Always-4; Sometimes-2; No-0; Not applicable-4		X		X	
Are "preferred" subscripts used? Always-2; Sometimes-1; Rarely-0; Not applicable-2				X	X
Has an attempt been made to minimize the number of subroutine CALL's? Yes-4; Some-2; No-0; Not applicable-4		X		X	

simultaneously. In addition, when possible, EQUIVALENCE statements should be used to overlay arrays.

Because format processing is expensive in both time and space, the same format statement should be used for two or more I/O operations whenever possible. Similarly, data type conversions should be consistent with the accuracy of the variables being used.

Finally, external subroutine calls are expensive and should be minimized. On the other hand, many compilers will expand intrinsic functions (e.g. SORT) for execution speed efficiency. As a result, there may be many copies of these subroutines in the object code. The expansion can be inhibited by declaring these functions as EXTERNAL.

3.2.3.3 File Utilization is an area in which one can do a great deal to increase the efficiency of a program.

The organization of files is critical, and care must be used to make the organization appropriate to the utilization. Sequential files that are subsequently sorted should be ordered in such a way as to minimize the number of sorts required.

Data elements on files should be formatted as to make their subsequent use and storage most efficient. Temporary FORTRAN files should be unformatted. Numeric data items should usually be in binary or punched decimal form.

The extent of buffering should be consistent with the utilization of files and criticality of time or space.

Program _____ Subject#_____

Attribute Efficiency Sub-Attribute Core Utilization

	0	1	2	3	4
Have EQUIVALENCE statements been used to overlay arrays when possible? Always-4; Sometimes-2; No-0; Not applicable-4		X		X	
Has the program been segmented in such a way as to minimize core requirements? Yes-2; No-0; Not applicable-2		X		X	X
Are intrinsic functions and subroutines declared as EXTERNAL? Yes-1; No-0; Not applicable-1			X	X	X
Has the number of conversions required among data types been minimized? Always-4; Sometimes-2; Seldom-0; Not applicable-4		X		X	
Has an effort been made to minimize the number of different formats used? (Use the same format for two or more I/O operations where possible?)				X	X
Are the data types and lengths consistent with the accuracy of the variables being operated on? Usually not-0; Usually-1; Always-2; Not applicable-2				X	X

Program _____ Subject# _____

Attribute Efficiency Sub-Attribute Utilization of Files

	0	1	2	3	4
Are data sets physically located on separate devices so as to minimize head positioning? Always-4; Sometimes-2; No-0; Not applicable-4		X		X	
Are sequential files ordered in such a way as to minimize the sorts necessary for their utilization? Always-4; Sometimes-2; No-0; Not applicable-4		X		X	
Are files created by FORTRAN programs as temporary files unformatted? Always-4; Sometimes-2; No-0; Not applicable-4		X		X	
Are data items stored in an appropriately compressed format? Always-2; Sometimes-1; No-0; Not applicable-2				X	X
Are heavily-used randomly-accessed files stored in direct or relative files? Always-2; Sometimes-1; No-0				X	X
Are large buffers provided for heavily used files? Always-2; Sometimes-1; No-0; Not applicable-2				X	X
Are these files which are used sequentially organized sequentially instead of randomly? No-0; Usually-1; Always-2; Not applicable-2				X	X

3.2.3.4 Over-All Processing Organization includes the major design decisions made early in the system development process. Some indicators of efficiency in the over-all processing include the following.

The language chosen should be appropriate to the application, and an optimizing compiler should be used if at all possible.

The program itself should be organized in such a way as to minimize processing, for example by preparing similar reports simultaneously where possible. Where programs have multiple functions, the program should include adequate parameterization so that all aspects of a program not be used if they are not needed.

The over-all organization should include provision for unsuccessful runs. Where appropriate, checkpoint/restart facilities should be used. Long-running programs should be organized in such a way that if a run is to be unsuccessful, this fact will be determined early.

3.2.4 Useability

Useability, as a quality attribute, is a measure of the facility with which the system can be used. The importance of this attribute is highly dependent upon the use to which the software system is to be put and the composition of the user community.

Programs that are used only a single time may be designed in such a way that inputs and outputs are convenient for programming purposes rather than for the user. Input devices are not critical nor are data collection and output distribution procedures.

Program _____ Subject#_____

Attribute Efficiency Sub-Attribute Over-all Processing Organization

	0	1	2	3	4
Has an optimizing compiler been used? Yes-4; No-0; Not applicable-4		X	X	X	
Is the program organized in such a way that if a run is to be unsuccessful, this fact will be determined "early"? No-0; Usually-1; Always-2; Not applicable-2				X	X
Does the system provide for parameterization so that only processing required for desired output is required, and all aspects of program need not be used every time? No-0; In some cases-1; Yes-2; Not applicable-2				X	X
Where possible, have similar reports been produced simultaneously rather than in separate steps? No-0; Usually-1; Always-2; Not applicable-2				X	X
Is the language appropriate to the application? No-0; Appropriate but not optimal-2; Yes-4; Not applicable-4		X		X	

On the other hand, the design of a program that is to be used over an extended period of time by a large user community must devote a great deal of care to making input procedures straightforward and the output easily interpretable.

For ...ms that are heavily used by a relatively small user community, (e.g. a bank teller system) the emphasis is more on convenience and ease of use than on the formatting of the input or output.

Useability has three sub-attributes. The first is device useability which relates to the physical ease with which users can interface with the system. The second is output utility which is essentially a measure of the convenience and usefulness of the system's output. The third sub-attribute is process simplicity. Process simplicity is a measure of the complexity, and thus the likelihood of error involved in utilizing the system.

3.2.4.1 Device Useability. Software systems may use a single device (e.g., a terminal which serves as both an input and output device) or a broad range of devices (e.g., card punches, readers, microfilm, copiers, etc.). The useability of these devices is largely a function of the user's familiarity with the device, the reliability of the device, and the appropriateness of the device to the application.

Devices that are generally available and used for a broad variety of applications are usually more familiar and less likely to be incorrectly used. In addition, common, generally-used devices are more likely to be available. Reliability affects useability in many ways, including the restriction of availability.

The complexity involved in using various system devices varies widely. In many cases, very general purpose devices are used which are extremely flexible but which involve, at the same time, a rather complicated user interface. On the other hand, less general devices are usually more appropriate for a specific application.

One measure of the complexity of a device is the training time required to effectively use it. It is important to separate the training time associated with the program. Training time can be affected by two factors. One is the documentation that is available for providing instructions. The second is the other use to which a particular device is put. For example, a keypunch might be a rather complicated device, but since it is so generally used, it is well understood and poses no complexity problems.

In some cases, devices lack the physical capabilities necessary to properly support the system. This usually occurs in two areas, both of which result from faulty systems analysis. The first area is one in which the device simply lacks the capability to support the system -- usually because it cannot operate fast enough.

The second area is one in which a device has the nominal characteristics desired but lacks sufficient reliability to adequately support the system.

3.2.4.2 Output Utility. Output utility is primarily a result of the systems analysis effort rather than the programming effort, but it greatly affects useability of the program.

One-time programs and programs used by a small number of persons do not have a heavy requirement for output formatting, but most others do.

Program _____ Subject# _____

Attribute Useability Sub-Attribute Device Useability

	0	1	2	3	4
Are the devices used by the system generally available or are they used only for this system? Only this system-0; Generally available-2; Not applicable-2		X		X	X
Are the data rate capacities of the devices consistent with the data rates going through them? No-0; Usually-1; Yes-2; Not applicable-2				X	X
Do the users use these devices for other applications, or is this the only one? Only one-0; Other applications-1; Not applicable-1			X	X	X
Are instructions for using system devices included in the documentation? or otherwise available? No-0; Minimal-1; Yes-2; Not applicable-2				X	X

Indeed, under certain circumstances, output formatting may require as much programming as the remainder of processing. Output should be generated in a form in which it can be used directly without further typing, graphing, extracting, etc. It should be properly labeled (with units) in such a way that the output is interpretable without referring to the documentation. Data elements on specific reports should be ordered in such a way as to make the reports easily useable. In general, exception reporting should be done. This is far better than using the machine to generate all possible information and then manually extracting that which is of interest.

The physical conditions under which reports are produced are important. Output must be timely to be useful. They must be produced in sufficient numbers so as to make them available to all people who need them. Output should be produced on an appropriate medium. Teletype, CRT's, graphics terminals, and COM devices are all appropriate devices under the proper conditions.

3.2.4.3 Process Simplicity. Process simplicity refers to the simplicity of the process through which a user must go in order to utilize a software product. It may be very simple (e.g., select from one of three prepared data cards, run program, read output) or very complicated (e.g., prepare input using a number of complex coding rules, punch cards, translate to paper tape, validate tape, set up plotter, etc.).

One measure of the complexity of the using process is the number of persons involved in producing the output. In general, the more persons involved, the less likely it is that the task can be done correctly.

Program _____ Subject#_____

Attribute Useability Sub-Attribute Output Utility

	0	1	2	3	4
Has the output been produced in a manner so as to make it interpretable without a great deal of training? e.g. have columns been labeled units indicated, headings provided, pages numbered, etc. Seldon-0; Sometimes-1; Always-2; Not applicable-2				X	X
Is the output in final form, or must it be subjected to additional processing before it can be used? e.g. graphed, re-typed, etc. Seldom-0; Sometimes-1; Always-2; Not Applicable-2				X	X
Is the output produced on a sufficiently timely basis to be useable? No-0; Somewhat-1; Yes-2; Not applicable-2				X	X
Is the output medium appropriate for its utilization? No-0; Yes-1; Not Applicable-1			X	X	X
Are the outputs produced in sufficient numbers to be distributed to all persons who need them? No-0; Yes-1; Not Applicable-1			X	X	X
Are the elements in the output reports ordered in the manner which makes the reports most easily used? No-0; Yes-1; Not applicable-1			X	X	X
Are "exceptional conditions" indicated or are only exceptional conditions reported (is exception reporting used?) No-0; Sometimes-1; Always-2; Not applicable-2				X	X

Program _____ Subject# _____

Attribute Useability Sub-Attribute Process Simplicity

	0	1	2	3	4
How many different people are involved in preparing the input? One-4; Two to Four-2; Over four-0; Not applicable-4		X		X	
What percentage of the input is prepared from "scratch" as opposed to being machine generated? 100%-0; 50-89%-1; Less than 50%-2; Not applicable-2				X	X
Characterize the data coding required. Minimal-4; Slightly difficult-2; Extremely difficult-0; N/A-4		X		X	
Does the input process include checks to insure the validity of the input? None-0; Some-1; Always-2; Not applicable-2				X	X
In the event of input errors, does the program print appropriate diagnostic messages? No-0; Some-1; Always-2; Not applicable-2				X	X

In many cases, the input process can be significantly simplified by computer-prepared input with manually generated input for exceptional cases.

Efforts should be made to simplify data coding requirements as much as possible and to include validation checks to insure the validity of input. Crossfooting, checkruns, and hash totals are some common checks.

The useability of a software system can also be enhanced by including diagnostic messages with suggested fixes when erroneous data are encountered.

3.2.5 Software Reliability. While mathematical models for reliability have been in use for some time the application of mathematical models to software reliability appears to be a recent development. The models presented in the literature are generally adaptations of familiar reliability models with the parameters functions of such things as debug time, number of instructions, etc.

The study of software reliability models may serve two purposes. One purpose is to determine which characteristics of the software package, programming practices, etc. have an effect on software reliability. The results of this study would be useful to the project manager in planning software development. A second purpose of software reliability models would be for the prediction of the reliability of a software package. This prediction could serve as a criteria for acceptance of a software package.

3.2.5.1 Definitions of Software Reliability. A generally accepted definition of reliability is the "probability of performing without failure

a specified function under given conditions for a specified period of time" (Gryna, et al. [1960]). Note that this definition is in terms of a probability and requires a definition of successful performance, operating environment and required operating time. In order to extend this definition to software reliability, we must define what we mean by the successful performance of a software package and the conditions under which it is to operate. The concept of a time period for which it is to operate can be thought of in terms of such things as operating time or number of unique input sets.

Dickson, et al. [1972] define software reliability as "the probability that a given software program operates for some time period, without a software error, on the machine for which it was designed given that it is used within design limits." They consider a model which expresses software reliability as a function of debugging time and the error detection rate.

Mulock [1971] defines software reliability as "the probability that we have no system failures attributable to software." Schneidewind [1972] defines software errors or troubles as "any logical or clerical error made by the programmer in creating or coding an algorithm which causes the algorithm to produce an incorrect result when the algorithm is presented with a correct input." With this definition, compilation errors and errors caused by the operating system are excluded. He then defines software reliability to be "the probability that a program will operate without a single occurrence of a specified severity of trouble, or worse, for a specified length of time t, and with a specified input load." Note that this definition is in terms of the severity of the trouble and there may be different reliability models, depending on the severity specified.

W.H. MacWilliams [1973] talks about three levels of definitions for software reliability. His top level definition is very similar to the classic definition of reliability and is not specific to software reliability. He states that, unlike hardware, "the continuing fidelity to an accepted design does not exist as a significant software problem" This is because, in general, software does change with time. He says that software failures are usually a result of changing input sets so that his intermediate-level definition is "the probability that the requirements capability continues to be met during a stated interval and under stated conditions representative of operational use."

One theme runs throughout these definitions. The reliability is restricted to use of the software under specified conditions. This would seem to imply that any statement about the reliability of a particular software package must contain a statement as to the conditions under which it is to operate and the use for which it is intended.

3.2.5.2 Overview of Reliability Models.

3.2.5.2.1 General Reliability Models - Before discussing the models for software reliability presented in the literature a brief discussion of general reliability theory will be helpful. Further discussion may be found in several texts (e.g. Bazovsky [1963], Lloyd and Lipow [1964], Shooman [1968]).

The most common definition of reliability of a device is that it is the probability that the device will operate without failure for time t,

denoted by R(t). If we let F(t) denote the probability that the failure occurs at a time less than or equal to t then

$$R(t) = 1 - F(t) \quad . \tag{1}$$

The function F(t) is known as the cumulative density function and in most cases is a continuous function of t. Thus the reliability can be found by specifying the probability density function, f(t) = F'(t), or the cumulative density function for the time to failure.

Rather than specifying either f(t) or F(T) one can specify the hazard function, or instantaneous failure rate, h(t), which is defined as the limit as $\Delta t \to 0$ of the probability that the device will fail in the interval (t, t + Δt) given that it has survived to t, divided by Δt. Thus, if the random failure time is denoted by T

$$h(t) = \lim_{\Delta t \to 0} \frac{Pr\{t<T\leq t+\Delta t\}}{\Delta t \; Pr\{T>t\}}$$

$$= \lim_{\Delta t \to 0} \frac{R(t+\Delta t)-R(t)}{\Delta t \; R(t)}$$

$$= \frac{1}{R(t)}\left\{\frac{-dR(t)}{dt}\right\} = \frac{f(t)}{R(t)} \quad . \tag{2}$$

Suppose that a failure cannot occur before time γ (usually $\gamma = 0$). Then, integrating both sides of equation (2) from γ to t we have

$$\int_{\gamma}^{t} h(x)dx = \int_{\gamma}^{t} \frac{f(x)dx}{R(x)} = \ln R(\gamma) - \ln R(t)$$

or

$$R(t) = R(\gamma) \exp\{-\int_{\gamma}^{t} h(x)dx\} \quad . \tag{3}$$

Since $R(\gamma) = 1$, we obtain the general reliability equation

$$R(t) = \exp\{-\int_{\gamma}^{t} h(x)dx\} \tag{4}$$

which relates the reliability and the hazard function. Hence, one may either use a model for F(t) and consequently R(t) or a model for h(t).

There are three broad classes of hazard functions commonly used in reliability work. For a decreasing failure rate, h(t) is a decreasing function of t. This is the model for what is frequently referred to as the "infant mortality" stage. During this period the probability of a device failing in the next Δt time units decreases as the item survives longer. For hardware reliability this is the period where poor workmanship and sloppy assembly are causing failures. As time increases without failure there is less chance of one of these causes still being present. For software this type of hazard function might be appropriate at the initial stages of development where, as errors are found and corrected, fewer errors remain and so the chance of failure is reduced.

A constant failure rate results when the hazard function is a constant and doesn't depend on the age t. The model leads to the negative exponential distribution for failure time. The constant failure rate model is commonly used with electronic hardware.

When h(t) is an increasing function of t, that is, we have an increasing failure rate, we are concerned with what is frequently called the "wear out" stage. For this hazard function, the longer the device survives the higher the conditional probability that it will fail in the next Δt time units. The increasing failure rate is used for hardware reliability for failures caused by wear of equipment, etc. For software models it might be appropriate if so few errors remain that the correction of one adds several new ones. It might also be appropriate if the reliability is a function of number of unique data sets (t = number of data sets) and as more data sets are successfully processed the probability of encountering one which will cause failure is increased. Also, reliability may not be a function of time but merely the probability of successful performance for a go, no-go system.

From the above it can be seen that we may build a model for software reliability by considering the probability of failure prior to t or by considering the hazard function. While t is usually a continuous variable called time, it may also be the number of times an algorithm is called, the number of data sets processed, or any of a number of measures.

In addition to a model for reliability one is usually interested in such things as the mean time to failure and variance of the failure time. For discussion of these and how they may be found from F(t) reference is made to any beginning or intermediate mathematical statistics text (e.g. Freund [1971], Hoel [1965], Larsen [1969]).

3.2.5.2.2 Software Reliability Models. One approach to determining a model for software reliability would be to adapt a well-known probability density function to software failures. The assumption that times between recurrence of one particular uncorrected software failure is a random variable with constant mean leads to the exponential distribution for the time to failure and the Poisson distribution for number of failures in a given time period. Mulock [1971] suggests the Gamma distribution to describe the means which characterize the different failures. He also states that if the failures are independent the negative binomial distribution may be used to describe the effect of the failures on the software system.

Mac Williams [1973] presents two models for software reliability, one a function of the input space and the other a function of test results. The input model assumes that an error occurrence depends on the input. Let N be the number of unique points in the input space and p_i be the probability that the i-th input point occurs. If $e_i = 1$ when the i-th input point results in error-free performance and $e_i = 0$ when the i-th input point results in an error, the software system reliability is given by

$$R_1 = \frac{1}{N} \sum_i p_i e_i \quad . \tag{5}$$

Since most sample spaces are extremely large it is impossible to describe the performance for each. However, estimates of software reliability could be found by sampling the input space. It would seem that the divisor N is incorrect since $\sum_i p_i = 1$ by definition and if all points result in error free performance equation (5) gives $R_1 = \frac{1}{N}$ while the correct value should be $R_1 = 1$.

The second model given by MacWilliams [1973] uses the results of M unique test cases to estimate software reliability. For test i let n_i be the number of errors found, $w_i(n_i)$ be a weighting factor for the seriousness of the errors observed, and $E_i(n_i)$ be a decreasing non-negative function such that $E_i(0) = 1$. Then, software reliability is given by

$$R_2 = \frac{1}{M} \sum_i E_i(n_i) w_i(n_i) \quad . \tag{6}$$

If we consider a test involving every point of the input space (M = N) and let

$$E_i(n_i) = \begin{cases} 1 \ , & n_i = 0 \ , \\ 0 \ , & n_i > 0 \ , \end{cases}$$

then, when all points giving error free performance,

$$R_2 = \frac{1}{M} \Sigma w_i(0)$$

and, using the corrected form of R_1,

$$R_1 = 1 \quad .$$

Since the two should agree it would seem that R_2 should be divided by $\Sigma\, w_i(n_i)$ rather than M. This would be in keeping with the usual practice with weighted averages. Here $E_i(n_i)$ is an estimate of the reliability for a given test case and errors and $w_i(n_i)$ is a weight given this reliability. Hence, the modified reliability models become

$$R_1 = \Sigma\, p_i e_i \ , \tag{5'}$$

$$R_2 = \frac{\Sigma E_i(n_i) w_i(n_i)}{\Sigma w_i(n_i)} \ . \tag{6'}$$

Expressions for software reliability obtained by modeling the hazard function have been given by Dickson, et al. [1972], Shooman [1972], [1973], and Jelinski and Moranda [1972]. The first three papers essentially describe the same model. The model presented by Dickson, et al. [1972], [1973] and Shooman [1972], [1973] assumes that the number of errors in the program at the start, E, is a constant and decreases directly as the errors are corrected. Also, the number of machine language instructions, I, is assumed constant. Let τ denote the debugging time since the start of system integration and $\rho(\tau)$ be the error detection rate per instruction as a function of time. The cumulative error function, as a ratio of the number of instructions is given by

$$\varepsilon(\tau) = \int_0^{\tau} \rho(x)\, dx \tag{7}$$

and the residual errors, as a proportion of number of instructions is

$$\varepsilon_r(\tau) = (E/I) - \varepsilon(\tau) \ . \tag{8}$$

From this model for the number of errors in the system the hazard function is developed. It is hypothesized that the failure rate for system operating time, t, is proportional to the number of residual errors, i.e.

$$h(t) = C\, \varepsilon_r(\tau) \tag{9}$$

which is a constant with respect to t. This leads to the familiar exponential reliability function

$$R(t) = \exp\{-C\ \varepsilon_r(\tau)t\}$$

$$= \exp\{-C[(E/I) - \varepsilon(\tau)]t\} \quad ,$$

with the hazard function a function of τ but not t. In order to use the model we need to specify $\varepsilon(\tau)$ (or $\rho(\tau)$) and C.

Dickson, et al. [1972] give two possible models for $\rho(\tau)$. One uses a constant error correction rate up to time τ_0 and zero error correction rate thereafter,

$$\rho(\tau) = \begin{cases} \rho_0 & , 0 \leq \tau \leq \tau_0 \\ 0 & , \tau > \tau_0 \end{cases} \quad . \tag{10}$$

This gives the reliability function

$$R(t) = \exp\{-\left[\frac{E}{I} - \rho_0\tau_0\right] Ct\} \ , \ t > 0 \quad . \tag{11}$$

Since the exponent must be negative and not equal to zero we have the restraint

$$0 \leq \rho_0\tau_0 < \frac{E}{I} \quad . \tag{12}$$

Another model is a triangular model with $\rho(\tau)$ an increasing function in the interval $[0, \tau_1]$ and decreasing in the interval $[\tau_1, \tau_2]$. The error correction rate is expressed by

$$\rho(\tau) = \begin{cases} \dfrac{\rho\tau}{\tau_1} , & 0 \leq \tau \leq \tau_1 , \\ \dfrac{\rho(\tau_2-\tau)}{\tau_2-\tau_1} , & \tau_1 \leq \tau \leq \tau_2 . \end{cases} \tag{13}$$

Shooman [1972] suggests that the constant C in the model given above can be broken into the product of K and r_p where

$$K = \frac{\text{number of catastrophic errors detected}}{\text{total number of errors detected}}$$

and

r_p = number of unique instructions processed per unit time period (processing rate) .

Other models given by Shooman [1972] include one which postulates that the number of errors corrected per time period is proportional to the number of errors present, leading to an exponential model for the error detection rate

$$\rho(\tau) = A \frac{E}{I} \exp [-A\tau] , \tag{14}$$

where A is a constant. He also presents a model in which the error detection rate is modified for varying manpower.

A model with a non-constant hazard function is given by Jelinski amd Moranda [1972]. They argue that the hazard rate is proportional to the number of remaining errors so that

$$h(t) = \phi[N - (i - 1)] \quad , \quad t_{i-1} < t < t_i \quad , \tag{15}$$

where N is the total number of errors originally, ϕ is a properly chosen constant and t_i is the time of the i-th error detection.

Schick and Wolverton [1972] modify the model given by Jelinski and Moranda so that the hazard rate increases as a function of time

$$h(t) = \phi[N - (i - 1)]t, \quad t_{i-1} < t < t_i \quad . \tag{16}$$

They argue that operation is a succession of different trials which gradually closes in on the remaining errors and corresponds to sampling without replacement.

3.2.5.2.3 Summary.

A number of different models for software reliability have been presented. Some models have the simplicity of a constant hazard function which lends them to many statistical procedures. The parameters of the models need to be evaluated in terms of their sensitivity to such things as software size, type, and other variables of software development.

3.2.5.3 A Model for Estimating Program Completion Level

Shooman's model and its subsequent variants are based upon historical data, principally the rate at which errors are discovered, and basically models of how "checked out" a program is. These models make no attempt to differentiate among various types of errors which occur, and they are designed to function at the end of the program development cycle.

On the other hand, the types of errors which occur as a software development project proceeds vary over time. For example, in early checkout

runs, syntax errors abound. When the program is 90% complete, however, they should be rare. In early checkout runs the programmer discovers very few errors resulting from incorrect understanding of the problem, because he is still working on syntax errors. Late in the development cycle there should also be few errors found which result from misunderstanding of the problem, but these errors will and should be found during the midpoint of the program development cycle.

The model discussed in this section is designed to predict completion level during the whole project development cycle, and it does this by making use of the different types of errors which occur.

Basically, the model proposed consists of activities in two phases as shown in Figure 8. In phase 1, empirical data are collected in which the distribution of errors in various categories is documented. The output of this phase is an error characteristic matrix which shows, for some pre-selected set of completion levels, the distribution of errors in various categories.

Completion level is based on runs completed and is not necessarily related to time. If it requires 300 runs to produce an operational program, the completion level is 50% when 150 runs have been made; 67% after 200 runs, and so forth.

In phase 2 empirical data collected in phase 1 are used to make completion level estimates of software development projects which are under development. Basically, this is done by observing the characteristics of errors which have occurred, selecting the completion level which exhibited error characteristics which followed a pattern most similar to that which was observed, and using that completion level as an estimator of the completion

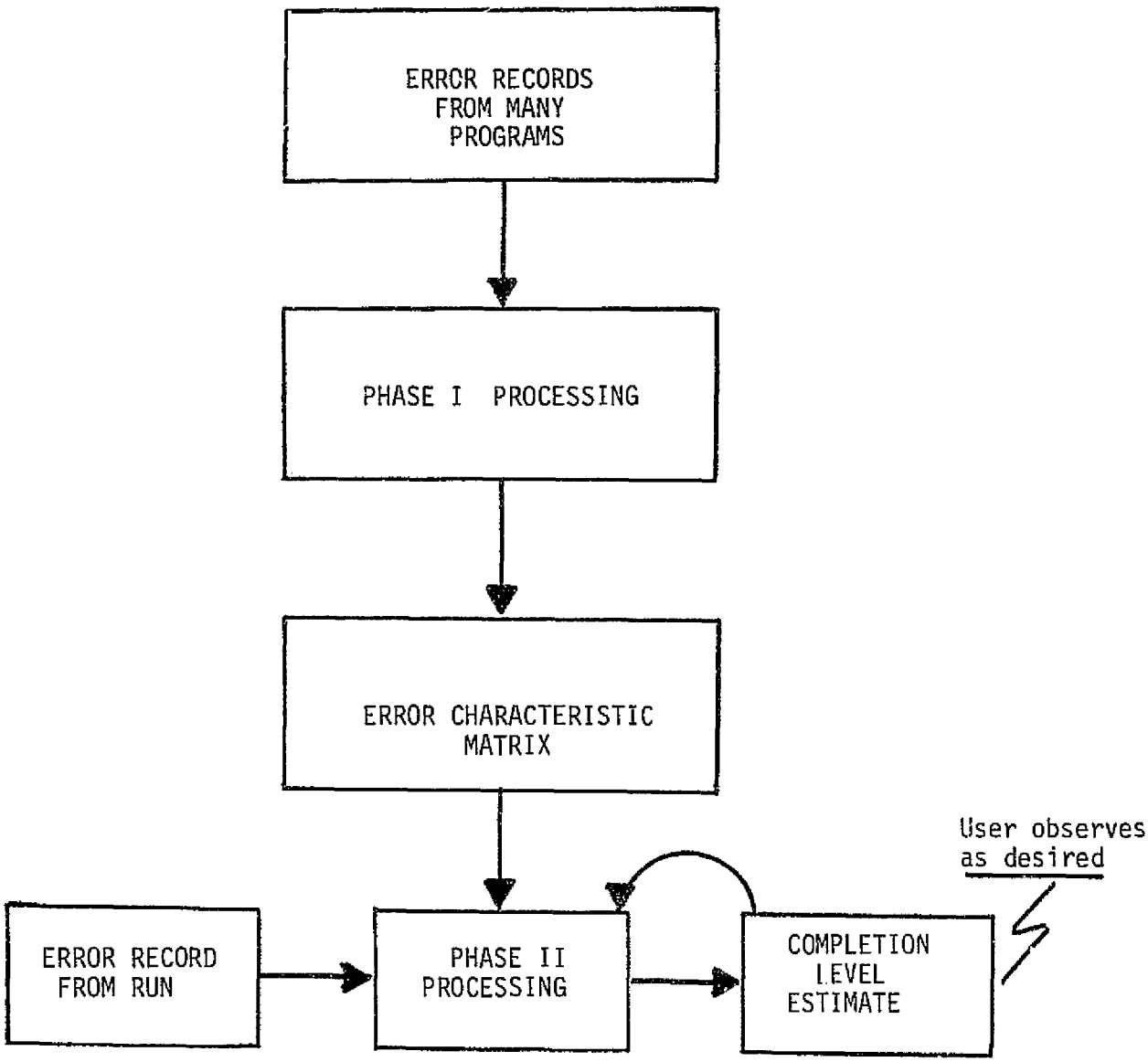


Figure 8.

Block Diagram of Historical Model

level of the program. In practice, this algorithm results in a curve which has the proper general characteristics but which is unduly erratic. The performance of the model can be improved substantially by smoothing these estimates. This has been done in two ways: first, instead of basing the estimate on a single observation (run) a composite error vector based on several (five in our tests) runs is developed. This error vector presents a better representation of the error characteristics of a particular point in the development cycle. Second, instead of making each estimate of completion level independent, successive estimates are used to modify a running estimate of the completion level. A more formal description of the algorithm and an example are presented below.

Phase 1 - Data Collection

The first step in the data collection phase is to decide upon a set of error categories which are to be used. These categories are somewhat dependent upon the environment in which the model is to be used. For example, some types of errors which might be good indicators in a data management context might never appear in a scientific computing environment.

The error categories need not be independent, although the more independence exhibited by the error categories the better. The error categories should be easily identifiable (that is, an error should be easily classified into one or more categories) and, to the maximum extent possible, they should exhibit approximately equal frequencies of occurrence over all completion levels.

The next step is the collection of the set of error matrices E^i. Each row of this matrix corresponds to the record of a single test run -- the first row being the record of the first test run, the second row being the record of the second test run, etc. The k-th element of each row is 1 or 0 depending upon whether or not an error of category k was observed in that run. The E^i th matrix has m^i rows (test runs) and n columns (error categories). A typical error matrix is shown in Figure 9.

Next, define a completion level vector, comp, consisting of p+1 completion levels such that

$$0 = comp_0 < comp_1 < comp_2 <...< comp_p = 1.0$$

The output of the data collection phase is an error characteristic matrix, EC. It has dimensions p rows (completion levels) by n columns (error categories). If e^i_j is the j^{th} row of the i^{th} error matrix and ec_j is the j^{th} row of the error characteristic matrix, then each row of the error characteristic matrix, EC, is computed as follows:

$$ec_k = \frac{\sum e^i_j}{\sum_i m^i}$$

where the sum in the numerator is over all values of i, j such that $comp_{k-1} < \frac{j}{m^i} \leq comp_k$.

Figure 9 is an example of a typical error matrix, and Figure 10 is an example of an error characteristic matrix.

ERROR CATEGORY

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	1	0	0	1	0	0	0	0	0
2	1	0	0	0	1	0	0	1	0	0	0	0
3	0	0	0	1	0	0	1	0	0	0	0	0
4	1	0	0	1	0	0	0	0	0	0	0	0
5	0	1	0	0	0	1	0	0	0	0	0	0
6	0	1	0	0	0	0	1	0	0	0	0	1
7	0	0	0	0	0	1	0	0	0	0	1	0
8	0	1	0	0	0	0	1	0	0	0	1	0
9	0	0	0	0	0	1	0	0	0	0	0	1
10	0	1	0	0	0	1	1	0	0	0	0	1
11	0	0	0	0	0	1	0	0	0	0	1	0
12	0	0	1	0	0	0	1	0	0	0	0	1
13	0	0	1	0	0	0	1	0	0	0	0	1
14	0	0	1	0	0	1	0	0	0	0	0	1
15	0	0	0	0	0	1	0	0	0	0	1	0

Figure 9

Typical Error Matrix

ERROR CATEGORY

COMPLETION LEVEL	1	2	3	4	5	6
.1	.83	.84	.14	.02	.02	.02
.2	.72	.84	.16	.04	.02	.02
.3	.61	.71	.13	.04	.02	.17
.4	.40	.69	.22	.39	.02	.36
.5	.13	.63	.19	.57	.02	.17
.6	.06	.42	.18	.63	.02	.05
.7	.02	.41	.10	.41	.02	.04
.8	.02	.13	.02	.22	.02	.03
.9	.02	.02	.01	.05	.31	.02
1.0	.02	.02	.01	.01	.46	.02

Figure 10

Typical Error Characteristic Matrix

Phase 2 Completion Status Estimates

The inputs to phase 2 are the result of error categorizations from a sequence of test runs and the error characteristic matrix developed in phase 1. The output is a sequence of status estimates, M_j, where M_j is the status estimate after the j-th test run. The algorithm consists of the following seven steps:

1) Set j to 1 and M_0 to 0.

2) Collect data from the j-th test run. Classify the errors, generating an error vector v^j. If j is less than 5, add 1 to j and go to step 2; otherwise go to step 3.

3) Calculate a vector

$$S = \sum_{q=j-4,j} V^q$$

4) Calculate a response vector, r with elements r_1, $r_2 \ldots r_n$ by calculating

$$r = S*EC$$

where "*" indicates that the elements of r are to be calculated by correlating the vector s with the corresponding columns of EC.

5) Determine the i for which r_i is a maximum.

6) Calculate $M_j = M_{j-1} + \alpha \left(comp_i - M_{j-1} \right)$ where α is a pre-selected damping factor (alpha equals .5 was used in our test) and $comp_i$ is completion level i.

7) Add 1 to j, return to step 2.

Experiments with the Model - In an effort to validate the model, a simulation experiment was conducted. This experiment was not sufficiently extensive to do much more than indicate that the model may have some validity and merits further investigation. The model requires a good deal of error data that simply is not available. However, there is no theoretical reason why it could not be collected and made available.

The first experiment that was conducted was simply an effort to validate the mathematics of the model. The question to be tested was, given an error characteristic matrix of sufficient diversity and an error history in which errors occur as indicated in that error characteristic matrix, would the model predict completion levels consistent with the actual completion levels?

An error characteristic matrix was postulated. This error characteristic matrix is shown in Figure 11. Then, random error occurrences were generated from that error characteristic matrix for development projects of different lengths. For example, a development project of length 300 runs was used.

For runs 1 through 30 the first row of the error characteristic matrix was used to generate the error record; for runs 31 through 60 the second row was used, etc. For runs 1 through 31 a category 1 error (language error) was generated with probability .78; a category 2 error (problem analysis) with probability .05; etc.

These runs were then fed back into phase II of the model and used to estimate completion levels. A perfect model would exhibit a straight line passing through the origin with slope=1 when completion level is

		Why-Language Error	Why-Problem Analysis	Why-How of System	Why-Human Error	When-Before Compile	When-Compile	When-Link/Load	When-<.1SEC Exec	When->.1SEC Exec	What-Program I/O	What-Prog Xfer Cont	What-Prog Comp/Man	What-Prog Data Desc	What-Control DO	What-Control Other	What-Data Prep
ERROR	10	78	15	5	32	50	30	11	6	3	10	10	30	30	20	17	2
PROBABILITIES	20	74	10	6	26	40	35	18	4	3	10	30	30	20	15	12	4
AT	30	68	22	8	24	30	40	16	7	7	20	20	25	15	10	7	8
VARIOUS	40	46	36	10	24	22	35	13	10	20	20	30	30	15	5	2	8
RELIABILITY	50	38	38	12	18	8	30	10	30	12	10	35	35	15	3	2	20
LEVELS	60	10	44	14	18	6	20	9	49	18	15	25	20	10	2	2	30
	70	10	31	16	28	4	16	4	47	33	10	20	20	5	2	1	40
	80	5	24	18	38	3	12	2	33	50	5	15	15	5	1	1	60
	90	2	15	20	52	3	3	2	32	60	3	8	7	5	1	1	78
	100	1	9	22	68	1	1	1	17	80	2	3	4	1	0	1	88

Figure 11

Postulated Error Characteristic Matrix

plotted against run number/total runs in project. As indicated in Figure 12, the model performed very much as expected.

As indicated previously, this exercise merely checks the mathematics of the model and does little to answer the primary question on which the model is based, namely, "do programs from a given environment exhibit a uniformity in the distribution of errors and is this uniformity sufficient to enable one to predict completion levels based on these errors?"

3.2.5.4 A Reliability Model on Which to Base Acceptance Testing

NASA, like other State and Federal Government Agencies and many large corporations, acquires most of its computer software from contractors outside their sphere of direct control. Although the contracting agency develops the specifications for a software product, it cannot, because of legal and practical considerations, monitor the development process, so it is faced with the task of making an assessment of the product after its completion.

A statistical tool is needed which will enable a manager to make, on the basis of a relatively small number of tests, a judgment as to the overall reliability of the system and provide some confidence limits along with that assessment. It is clear that in all but the most trivial cases, exhaustive testing is not possible. In addition, because of the cost in time and other resources involved in testing, it is absolutely imperative that statistical models be developed so that management can determine the reliability they can guarantee with a given amount of testing.

Consider the following characterization of the problem. A computer program can be thought of as a processor with a single multi-dimensional

HISTORICAL MODEL

α = 5 300 Runs

10-run Sample
1-10-75

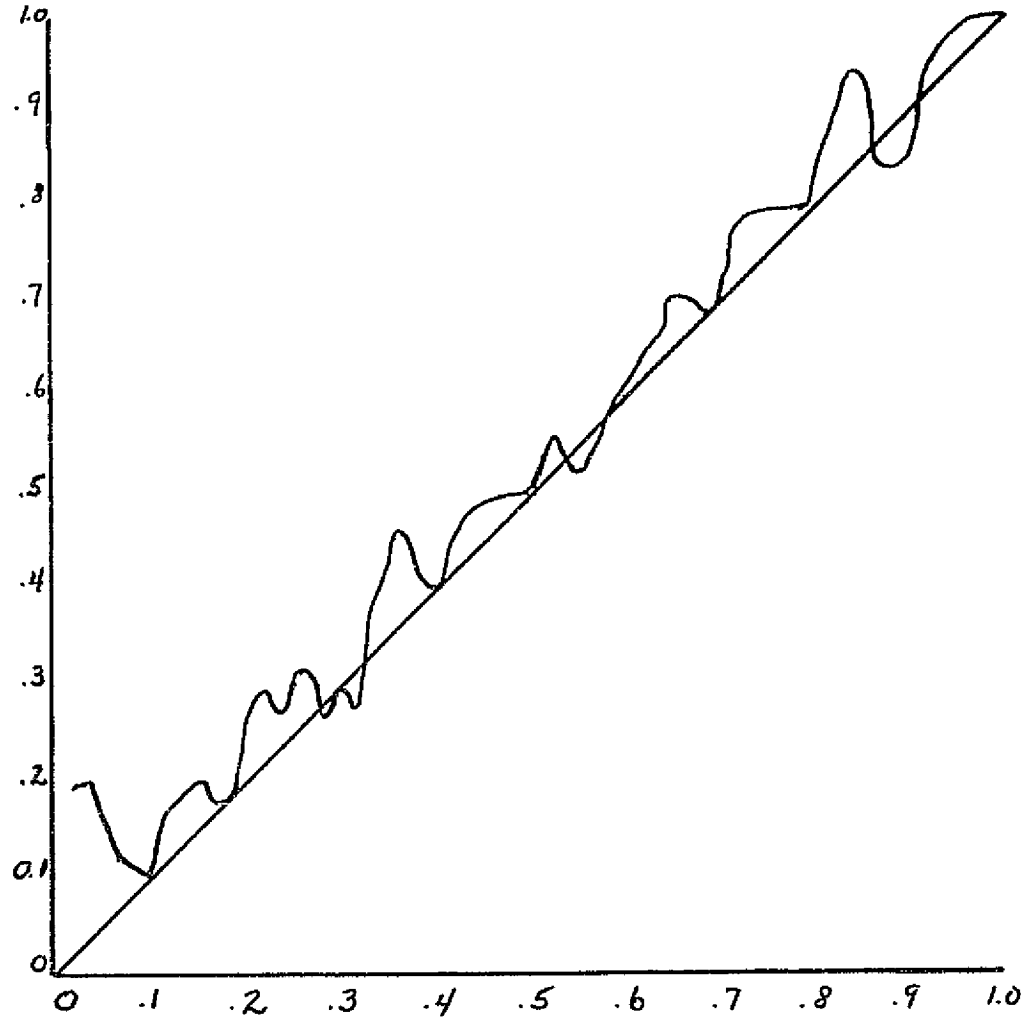


Figure 12.

input and a single multi-dimensional output.

As an example, an inventory program might have the following inputs:

Parameter	Allowable Values
Transaction Type:	RECeipt, DISbursement, MODification
Part Number:	A 5-digit number
Source:	One of 200 different suppliers
Number:	A 3-digit number

Each input then consists of a four-dimensional vector, e.g. (REC, 63921, JONES SUPPLY, 003).

Although there may be a very large number of these different inputs, the number of paths through the program is usually relatively small. This is due to two reasons. First, some parameter-value combinations may not occur so it is not necessary to have as many different paths as there are possible inputs. Secondly, many of the possible inputs normally utilize the same program path. In the above example, all inputs of type REC may utilize one path through the program, while all those of type DIS may use another, etc.

The classical definition of system reliability is the probability that a system will operate successfully under stated conditions over a specified time interval. Let us consider an analogous definition for software reliability. We will define the reliability of a program as the probability

that the program will process K consecutive representative inputs correctly. Using this definition, we will proceed to define a reliability assessment procedure.

This procedure is based on the following assumptions:

(1) It is assumed that an analyst can determine if a program has correctly processed a given input. This may not always be the case. For example, in situations involving long, complex mathematical calculations, parallel manual calculations may not be practically possible. Similarly, in situations involving real-time processing, it may not be possible to "stop" the action sufficiently to determine what the inputs were to a given program.

(2) It is assumed that an analyst can partition the program inputs in such a way that each partition defines a path or group of paths through the program. In the preceding example, the inputs might be effectively partitioned on the basis of "Transaction Type."

(3) It is assumed that an analyst can determine the probability of a random input being selected from any given partition. In the preceding example, it would be necessary to determine the relative numbers of receipts, disbursements, and modifications.

(4) It is assumed that it is possible to randomly select representative inputs from each partition.

C.2

Suppose that the input space is partitioned into Q disjoint partitions. Denote by p_i, i = 1, 2, ..., Q, the probability that an input selected at random will come from partition i, where the p_i are known. Let R_i be the probability that an input randomly selected from input partition i will be successfully processed, that is, R_i is the reliability for partition i. The probability that an input selected at random from the entire input space will be processed successfully is

$$R(1) = \sum_{i=1,Q} \Pr\{\text{input from partition } i\}\Pr\{\text{success given } i\},$$

$$= \sum_{i=1,Q} p_i R_i \qquad (1)$$

which may be estimated by using estimates of the R_i.

In order to estimate the R_i, select r_i random inputs from partition i, i = 1, 2, ..., Q. The number of inputs from each partition should be as large as possible consistent with the time and resource constraints on the testing process. Although it will not usually provide an optimum allocation, selecting test inputs from various partitions in proportions to the p_i will provide an acceptable test pattern. Each input is processed and the number of correctly processed inputs, f_i, for partition i recorded. The R_i are then estimated by

$$\hat{R}_i = f_i/r_i \qquad (2)$$

and the system reliability, R(1), is estimated by

$$\hat{R}(1) = \sum_{i=1,Q} p_i \hat{R}_i = \sum_{i=1,Q} p_i f_i/r_i \quad . \qquad (3)$$

Because the test inputs are selected at random, if the tests were repeated the estimates of R_i would vary. As a consequence, the estimate $\hat{R}(1)$ is no more than an estimate and the probability that it is exactly equal to the true value of R(1) is zero. However, to give a better answer we may appeal to the estimation procedure known as confidence intervals.

Basically, the confidence interval, at say the 90% confidence level, is based on an interval determined using specific procedures such that the probability of obtaining an interval which contains the true value of the quantity of interest is 90%. Procedures for confidence intervals, and specifically for the parameter of a binomial distribution (which the R_i discussed above is) are contained in most books on statistical methods. One which will be illustrated later uses the normal approximation to the binomial distribution. The lower confidence limit for R_i, with level of confidence C_i, is found by use of the binomial distribution and a probability statement of the form

$$\Pr\{R_i > R_{Li}\} = C_i \tag{4}$$

where R_{Li} is a function of r_i, f_i, and C_i, and is a random variable. If repeated samples of size r_i were taken and R_{Li} calculated, using (4), for each sample, a porportion C_i of the R_{Li} would be less than the unknown reliability, R_i .

A conservative lower limit for R(1) with confidence level γ (conservative in that the probability of a correct interval is most likely greater than γ) can be found by using confidence limits for the individual R_i. The argument is as follows:

1. $Pr\{R_i > R_{Li}\} = C_i$ implies $Pr\{p_iR_i > p_iR_{Li}\} = C_i$.

2. $p_iR_i > p_iR_{Li}$, all i, implies $\Sigma p_iR_i > \Sigma p_iR_{Li}$
   (but converse not necessarily so).

3. From 2, the event A, $p_iR_i > p_iR_{LI}$ all i, is a subset of the event B, $\Sigma p_iR_i > \Sigma p_iR_{Li}$.

4. From 3, the probability of B is greater than or equal to the probability of A, or

$$Pr\{\Sigma p_iR_i > \Sigma p_iR_{Li}\} \geq Pr\{p_iR_i > p_iR_{Li}, \text{ all } i\}$$

   or

$$Pr\{R(1) > \Sigma p_iR_{Li}\} \geq \prod_{i=1,Q} C_i = \gamma \ ,$$

   since the results for the different partitions are independent.

Hence, a conservative lower confidence limit for R(1) is

$$R_L(1) = \sum_{i=1,Q} p_iR_{Li} \ , \tag{5}$$

found by using the lower confidence limits for the individual R_i . One choice for the $C_i = C = \gamma^{1/Q}$

This procedure gives the following algorithm for estimating the software reliability.

Step 1: Partition the input space into Q disjoint partitions. Determine the probability of occurrence of an input from each partition.

Step 2: Select r_i representative random inputs from the first partition, r_2 from the second and so forth down through r_Q from

the Q-th partition.

Step 3: Process each input, noticing the number of inputs which are processed correctly and which are processed incorrectly in each partition. Compute f_i, the number of successful tests observed when testing inputs selected from the i-th partition, for each partition $i = 1,2, \ldots,Q$.

Step 4: Select a confidence level, C. Then, for each partition, compute a lower bound on the probability of successfully completing a test of a random representative input selected from the population defined by that partition. The lower bound for the i-th partition can be computed by the following formula.

$$R_{Li} = \frac{\hat{R}_i + Z_c^2 \Big/ (2r_i) - Z_c \sqrt{\dfrac{\hat{R}_i (1-\hat{R}_i)}{r_i} + Z_c^2 \Big/ (4r_i^2)}}{1 + Z_c^2 / r_i}$$

$$= \frac{r_i \left(2f_i + Z_c^2\right) - Z_c \sqrt{r_i \left[4f_i (r_i - f_i) + r_i Z_c^2\right]}}{2r_i \left(r_i + Z_c^2\right)}$$

where r_i is the number of tests performed on inputs from the i-th partition and Z_c is a standard normal deviate determined from the confidence level selected.

Step 5: Compute the lower bound, $R_L(1)$, on the probability that a single random input from the whole input space will be processed correctly. This quantity can be calculated by

$$R_L(1) = \sum_{i=1,Q} p_i \, R_{Li}$$

Step 6: Compute the expected probability that a random input from the whole input space will be processed correctly. This quantity can be calculated by

$$\hat{R}(1) = \sum_{i=1,Q} p_i \, \hat{R}_i$$

Step 7: Compute the lower bound on the probability that K consecutive random inputs from the whole input space will be processed correctly. This quantity, $R_L(k)$, is simply

$$R_L(K) = \left[\hat{R}_L(1)\right]^k$$

Step 8: Compute the expected reliability, $\hat{R}(K)$ by

$$\hat{R}(K) = \left[\hat{R}(1)\right]^k$$

With these calculations completed, we are now in a position to make two statements concerning the reliability of the program. First, the expected reliability of the program is R(K). Second, we can, at a confidence level C^Q guarantee that the reliability of the program is greater than $R_L(K)$.

Example:

The following example demonstrates the method. It is based on the example presented previously in this section.

Step 1: Partition the input space into three subsets by transaction type. All REC transactions will be in one subset, all DIS transactions in a second, etc. It is determined that random transactions

will occur with the following relative frequencies

TYPE	PROBABILITY, P_i
REC	.25
DIS	.44
MOD	.31

Step 2: It is determined that fewer than 300 test cases will be run. Random inputs are selected in the following numbers.

TYPE	NUMBER OF PARTS
REC	72
DIS	120
MOD	80
Total	272

Step 3: These inputs are then tested with the following results:

TRANSACTION TYPE	i	SUCCESSES, f_i	FAILURES	$\hat{R}_i$
REC	1	70	2	.9722
DIS	2	120	0	1.0
MOD	3	79	1	.9875

Step 4: The confidence level attached to the reliability estimate is determined to be $\gamma = .90$. The lower bounds for the probability for successfully processing an input from each partition is then determined. For $C = .90^{1/3} = .9655$, $z_c = 1.82$.

$$\text{Then } R_{L1} = \frac{72(140+1.82^2) - 1.82\sqrt{72[(70)(2)+1.82^2(72)]}}{2(72)(72+1.82^2)}$$

$$= .911$$

$$R_{L2} = .973$$

$$R_{L3} = .929$$

Step 5: Compute the lower bound on the probability that a single input from the whole input space will be processed correctly

$$R_L(1) = .25(.911) + .44(.973) + .31(.929)$$

$$= .944$$

Step 6: Compute the expected probability that a random input from the whole input space will be processed correctly.

$$\hat{R}(1) = .25(.9722) + .44(1.0) + .31(.9875)$$

$$= .243 + .440 + .3061$$

$$= .989$$

Step 7: Assume that we are interested in a reliability period of 10 transactions. Compute the lower bound on the probability of successfully processing 10 consecutive transactions.

$$R_L(10) = (.944)^{10} = .561$$

Step 8: Compute the expected probability of successfully processing 10 consecutive transactions.

$$R(10) = (.989)^{10} = .895$$

Hence, we are in a position to state that the expected reliability of the program is .895 and we can assure, with a confidence level of .90, that the reliability of the program is over .561.

An alternative approach to finding the lower confidence bound for R(1) would be to consider the distribution of

$$\hat{R}(1) = \sum_{i=1,Q} P_i \hat{R}_i \tag{6}$$

Since the R_i are in fact independent bionomial random variables the random variable $\hat{R}(1)$ has mean $\sum_{i=1,Q} P_i R_i$ and variance

$$V(\hat{R}(1)) = \sum_{i=1,Q} p_i^2 \frac{R_i(1-R_i)}{r_i} \tag{7}$$

If, at the same time the individual r_i are large enough so that the $\hat{R}_i$'s are approximately normally distributed the $\hat{R}(1)$ is a linear function of normal variables and is also approximately normally distributed. As a consequence, the lower bound with confidence γ is given by

$$R_L(1) = \hat{R}(1) - Z_\gamma \sqrt{\sum_{i=1,Q} \frac{p_i^2 R_i(1-R_i)}{r_i}} \quad . \tag{8}$$

One difficulty with (8) is the R_i's under the radical since they are unknown. There are two remedies available. One is to replace them by the corresponding estimates $\hat{R}_I$,

$$R_L(1) = \hat{R}(1) - Z_\gamma \sqrt{\sum_{i=1,Q} \frac{p_i^2 \hat{R}_i(1-\hat{R}_i)}{r_i}} \quad . \qquad (9)$$

But this may give difficulty if the $\hat{R}_i$'s are very near or equal to 1. The other remedy is to note that since $0 \leq R_i \leq 1$, the product $R_i(1-R_i)$ has a maximum value .25 and the largest value of $V(\hat{R}(1))$ would occur if $R_i = .05$, all i. Hence

$$V(\hat{R}(1)) \leq .5 \sqrt{\sum_{i=1,Q} \frac{p_i^2}{r_i}} \qquad (10)$$

so that a conservative lower bound would be

$$R_L(1) = \hat{R}(1) - .5Z_\gamma \sqrt{\sum_{i=1,Q} \frac{p_i^2}{r_i}} \quad . \qquad (11)$$

Consider the use of this procedure for finding lower confidence bounds for the example presented previously, with $\gamma = .90$.

Using (9), we obtain

$$R_L(1) = .989 - 1.28\sqrt{V(\hat{R}(1))}$$

where

$$V(\hat{R}(1)) = \frac{.25^2(.9722)\ (.0272)}{72} + \frac{.44^2(1.0)(0.0)}{120} + \frac{.31^2(.9875)(.0125)}{80}$$

$$R_L(1) = .978$$

and

$$R_L(10) = (.978)^{10} = .798 \quad .$$

Using (11), the corresponding limits are given by

$$R_L(1) = .989 - .64\sqrt{\frac{.25^2}{72} + \frac{.44^2}{120} + \frac{.31^2}{80}}$$

$$= .954$$

and

$$R_L(10) = (.954)^{10} = .622 \quad .$$

Of the procedures for determining a lower confidence bound the first, involving the sum of individual confidence bounds is ultra-conservative and is the least preferred of the three. Since, in general, all the individual $\hat{R}_i$'s will be near 1.0 the use of (11) may be too conservative, however it does have simplicity of calculation and recognizes that even if $\hat{R}_i = 1.0$, the true value of $R_i(1-R_i)$ may not be zero. A compromise between using (9) and (11) may be to assume that $R_i \geq R^*$, say, for all i, and replace the .5 in (11) by $\sqrt{R^*(1-R^*)}$.

N	# of input parameters
n_i	# of possible values for the i-th input parameter
Q	# of disjoint partitions of input space
r_i	# of random inputs selected from the i-th partition
p_i	probability that an input to the system will come from the i-th partition of the input space
C_i	confidence level attached to reliability estimates, individual partition
γ	confidence level attached to reliability estimate, entire input
R_{Li}	lower bound on probability of successfully processing a single input from the i-th partition
$R_L(1)$	lower bound on the probability of successfully processing a single input from the input space
$R(1)$	expected probability of successfully processing a single input fron the input space
$R_L(K)$ $R(K)$	lower bound and expected probability, respectively, of processing K consecutive random inputs from the input space.

3.2.6 Functional Correctness

Functional correctness is the degree to which a program's capabilities coincide with the designer's expectation of what those capabilities should be.

The degree to which a program is functionally correct depends, in large measure, on the diligence and skill of the program designers. In theory, the program designer specifies the algorithm to be employed, the data checks to be made, and specifies the capability of a proposed software product in some detail. The programming task is then one of translating these specifications into code.

In practice, a separation between designer and programmer may not exist. And, when it does exist, designers typically give programmers a great deal of latitude in the actual implementation of a program. As a result, many programs suffer from lack of completeness. Some program features typically expected to be included by designers but not specified include the following:

(1) Error messages

(2) Checkpoint/Restart capabilities

(3) Positioning of I/O devices

(4) Validity checks on input variables

(5) Range-testing subscripts before they are used.

Clearly, these are but a few in a long list of features which a designer may expect but may not specify, thereby contributing to a lack of functional correctness. Other similar problems arise due to lack of completeness in describing possible inputs, procedures for handling erroneous data, etc.

The functional correctness problem of accuracy in computation may be due to improper algorithm selection or to programming errors in implementation. Algorithm selection, in most cases, is less likely to be the culprit than are programming errors. Some insidious programming errors which are likely to cause erroneous results and which are difficult to trace might include the following:

(1) Overflows that occur in special situations

(2) Loss of precision due to type conversions

(3) Truncation in intermediate results

(4) Lack of precision in number representation.

Again, these are but a few in a long list of problems which can contribute to a lack of accuracy and a subsequent lack of functional correctness in a software product. Lack of accuracy is a major problem because many algorithms used on modern computers cannot be effectively tested by hand computation.

Functional correctness is measured indirectly in other ways. Proof of correctness techniques basically test functional correctness although they normally do not adequately test for accuracy nor do they test for completeness.

3.2.7 Productivity

A piece of software may be functionally correct, efficiently written, etc. but still not represent a worthwhile effort if it is not productive. A program is non-productive if it is not necessary, is less efficient than some alternate approach to the problem, or represents an investment with less return than some alternate investment.

As examples, a payroll program may be considered to be non-productive even though it possesses other quality attributes if it costs more to use than a manual operation that performs the same function. Or, a digital control system which would speed assembly line operations might be considered non-productive if an equivalent expenditure in marketing would produce a greater profit for the company.

There are three factors which determine whether or not a piece of software is considered productive. They are the cost of the piece of software, its value to the firm, and the availability of alternate market opportunities.

No attempts have been made to develop a productivity metric because the productivity determination should be an accounting decision which can be measured in dollars. In firms with reasonably enlightened managements this is always the case. In other cases, however, justification is on an emotional basis and discussion of metrics is not relevant.

Costs may be broken down into production costs and operation costs. When software is purchased, production costs are easily identified, although costs incurred in conversion and interruption of services must still be calculated. When software is developed in-house, costs are much more difficult to assess. Most programmer teams work on multiple projects and it is difficult to properly allocate time spent. Support facilities that are used are difficult to allocate. In general, a whole range of indirect costs must be attributed to the cost of a software development project.

Operation costs are even more difficult to assess because they typically involve users outside the computer operation. While it may be

relatively easy to cost out machine time, operators, and so forth, the costs of people preparing data and using information are virtually impossible to quantify. In addition, the cost of maintenance may easily exceed development costs and these costs are very difficult to predict.

Value determinations are equally difficult to make. In some cases, depreciated development cost is used as the value of a software system. This is a useful accounting device and may accurately reflect the value of a system. In other cases, however, the cost may grossly underestimate the value of a software system to the firm. Many modern industries are heavily dependent upon information, and to these industries a software system's value is only indirectly related to its costs. Examples of industries of this type include the airlines and the banks, neither of which could exist as we know them today without computer support.

A growing trend toward the marketing of application packages has an impact on value determination. If a piece of software is to be marketed, its potential sales less marketing costs must be considered in the value determination.

Finally, in most firms, the development of a piece of software is a business decision, not unlike the development of a new product or the decision to purchase a piece of property. The decision to embark on a venture of this type must be based on return on investment, and in that sense, software must compete with other market opportunities for priority.

3.3 Summary

This chapter has contained a discussion of software quality attributes and their measurement. Because software quality is directly

related to errors in the development process, this area has also been discussed.

Remarkably little is known about the types of errors and distribution of these errors in the software development process. This is especially unfortunate because this error data are needed to support research in the area of software reliability and other quality attributes, and error data are potentially useful as a measurement tool.

A prototype error data collection/management system has been proposed. This system adds error data to a data base and generates a number of reports of interest to programming personnel and management.

In general, there have been very few attempts to measure software quality, simply because no adequate models of software quality exist. This is also unfortunate because software quality metrics are needed to evaluate productivity, to evaluate alternate software products, and for a variety of other reasons.

Software quality is a multi-dimensional entity. Any one of these dimensions can be critical, depending upon the situation. For purposes of this discussion, software quality has six attributes:

Efficiency,	Useability,	Reliability,
Modifiability,	Functional Correctness,	Productivity.

Of these attributes, only one, reliability, has been extensively researched. There exist a number of statistical models for reliability, most of which are variations of an idea presented by Shooman in which he calculates reliability based on the rate at which errors occur in an operating program. Shooman's model operates at the end of the program development cycle, and makes no use of information regarding the different

types of errors which occur. Two new models, one designed to operate at an earlier part of the program development cycle and one model which is applicable for acceptance testing purposes were described.

Finally, metrics for efficiency, modifiability, and useability based on subjective judgments were presented.

3.4 Bibliography

Bazovsky, Igor. [1963]. Reliability: Theory and Practice. Prentice-Hall, Inc. Englewood Cliffs, New Jersey.

Dickson, J.C , J.L. Hesse, A.C. Kientz, and M.L. Shooman [1972]. "Quantitative Analysis of Software Reliability." Proceedings of 1972 Annual Reliability and Maintainability Symposium. pp. 148-157.

Freund, John E. [1971]. Mathematical Statistics, 2nd Ed. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Gryna, Frank M., Jr., Naomi J. McAfee, C.M. Ryerson, and Stanley Zwerling, Editors [1960]. Reliability Training Text, 2nd Ed. The Institute of Radio Engineers, Inc. New York.

Hoel, Paul G. [1965]. Introduction to Mathematical Statistics, 3rd Ed., John Wiley and Sons, Inc., New York.

Jelinski, Z. and P. Moranda [1972]. "Software Reliability Research" in Statistical Computer Performance Evaluation, Walter Freiberger, Ed. Academic Press. New York. pp. 465-484.

Larsen, Harold J. [1969]. Introduction to Probability Theory and Statistical Inference. John Wiley and Sons, Inc., New York.

Lloyd, David K. and Myron Lipow [1964]. Reliability: Management, Methods, and Mathematics. Prentice-Hall, Inc. Englewood Cliffs, New Jersey.

MacWilliams, W.H. [1973]. "Reliability of Large Real-Time Control Software Systems." 1973 IEEE Symposium on Software Reliability. pp. 1-6.

Mulock, R.B. [1971]. "Program Correctness, Software Reliability, Today's Capabilities. Summary." International Symposium on Fault Tolerant Computing. pp. 137-139.

Schick, G.J. and R.W. Wolverton [1972]. "Assessment of Software Reliability." 11th Annual Meeting, German Operations Research Society. Hamburg, Germany.

Schneidewind, Norman F. [1972]. "An Approach to Software Reliability Prediction and Quality Control." Proceedings Fall Joint Computer Conference. pp. 837-846.

Shooman, Martin L. [1968]. Probabilistic Reliability: An Engineering Approach. McGraw-Hill Book Company. New York

_____ [1972]. "Probabilistic Models for Software Reliability Prediction in Statistical Computer Performance Evaluation, Walter Freiberger, Ed. Academic Press. New York. pp. 485-502.

_____ [1973]. "Operational Testing and Software Reliability Estimation During Program Development." 1973 IEEE Symposium on Software Reliability. pp. 51-57.

Amory, W., Clapp, J.A. A Software Error Classification Methodology, MITRE Corporation, MTR-2648, Vol. VII, June, 1973.

Young, E.A. Error-Proneness in Programming, Ph.D. Thesis, University of North Carolina: Chapel Hill, North Carolina, 1970.

Rubey, R.J., Wick, R.C. and Beathley, L. Comparative Evaluation of PL/1. U.S. AirForce Report AD-669 096, July, 1968.

Ramamoorthy, C.V., Cheung, R.C., and Kim, K.H. Reliability and Integrity of Large Computer Programs, U.S. Government Report AD-779 339, March, 1974.

Endres, A. An Analysis of Errors and Their Causes in System Programs, Proceedings of the International Conference on Reliable Software. April, 1975.

Shooman, M.L., Bolsky, M.I. Types, Distribution, and Test and Correction Times for Programming Errors, Proceedings of the International Conference on Reliable Software, April, 1975.

Boehm, B.W. et al. Characteristics of Software Quality. TRW Software Systems Report TRW-SS-73-09, 1973.

McCracken, Daniel D. and Weinberg, Gerald M. How to Write a Readable FORTRAN Program. DATAMATION, Vol. 18, No. 10, Oct., 1972. pp. 73-77.

Rubey, R. and Hartwick, R.D. Quantitative Measurement of Program Quality, Proc. 23rd National Conference, ACM, 1968. pp. 671-677.

3.5 References

Baker, F.T. Chief Programmer Team Management of Production Programming, IBM System Journal Vol. II. No. 1. (1972), pp. 56-73.

Benson, Jeoffry P. Structured Programming Techniques, Proc. of 1973 IEEE Symposium on Computer Software Reliability, pp. 143-147.

Bequaert, F.C. A System for Automatic Program Generation, Proc. FJCC, 1968, pp. 611-616.

Borgerson, Barry R. Dynamic Confirmation of System Integrity, Proc. of the 1972 Fall Joint Computer Conference, pp. 89-96.

Brown, J.R. and Hoffman, R.H. Evaluating the Effectiveness of Software Verification - Practical Experience with an Automated Tool, AFIPS Fall Joint Computer Conference, Anaheim, California, December 1972.

Dijkstra, E.W. A Constructive Approach to the Problem of Program Correctness, BIT. Vol. 8, No. 3, (1968), pp. 174-186.

Freeman, P. Functional Programming, Testing and Machine Aids, Program Test Methods, (Ed.) W.C. Hetzel, Englewood Cliffs, Prentice-Hall, Inc., 1973, pp. 45-46.

Floyd, R.W. Assigning Meanings to Programs, Mathematical Aspects of Computer Science, (Proceedings of Symposium in Applied Mathematics, 19) (Ed.) J.T. Schwarts, Providence, American Mathematical Society, 1967, pp. 19-32.

Foley, M. and Hoare, C.A.R. Proof of a Recursive Program--Quicksort, Computer Journal, Vol. 14, November 1971, pp. 391-395.

Good, D.I. and Ragland, L.C. Nucleus - A Language of Provable Programs, Program Test Methods, (Ed.) W.C. Hetzel, Englewood Cliffs, Prentice-Hall, Inc., 1973, pp. 93-117.

Graham, R.M., Clancy, G.J., Jr. and DeVaney, D.B. A Software Design and Evaluation System, CACM, 16,2 (Feb. 73), pp. 110-116.

Gruenberger, F. Program Testing and Validation, Datamation, Vol. 14,7 July 1968, pp. 39-47.

Hart, L.D. The User's Guide to Evaluation Products, Datamation, December 15, 1970, pp. 32-35.

Hoare, C.A.R. An Axiomatic Basis for Computer Programming, CACM, Vol. 12, No. 10, (Oct. 1969), pp. 576-583.

Hoare, C.A.R. Proof of a Program, CACM, Vol. 14, No. 1, (Jan. 1971), pp. 39-45.

Hunt, B.R. A Comment on Axiomatic Approaches to Programming, CACM, Vol. 13, No. 7, (July 1970), p. 452.

Kolence, K.W. A Software View of Measurement Tools, Datamation, January 1, 1971, pp. 32-38.

Ireson, W. and Grant, E. (Eds.) Handbook of Industrial Engineering and Management, Prentice-Hall, Englewood Cliffs, N.J., 1955.

Kang, A.N.C. On the Complexity of Proving Functions, Proc. of the 1972 Spring Joint Computer Conference, pp. 493-501.

Karush, A. Quality Assurance, Datamation, October 1968, pp. 61-66.

King, J. A Verifying Compiler, Debugging Techniques in Large Systems, (ed.) Randall Rustin, Prentice-Hall, 1971, pp. 17-39.

King, James C. Proving Programs to be Correct, IEEE Transactions on Computers, Vol. C-20, No. 11 (Nov. 1971), pp. 1331-1336.

Kral, J. One Way of Estimating Frequencies of Jumps in a Program, Communicatons of the ACM, Vol. 11, No. 7, July 1968, pp. 475-480.

Levitt, Karl N. The Application of Program-Flowing Techniques to the Verification of Synchronization Processes, Proc. of the 1972 Fall Joint Computer Conference, pp. 33-47.

Linden, T.A. A Summary of Progress Toward Proving Program Correctness, Proc. of the 1972 Fall Joint Computer Conference, pp. 201-211.

Liguori, F. The Test Language Dilemma, Proc. ACM Nat. Conf. 1971, pp. 388-396.

London, R.L. The Current State of Proving Programs Correct, Proc. 1972 Assoc. Comput. Mach. National Conference.

Manna F. and Waldinger, R.J. Toward Automatic Program Synthesis, CACM Vol. 14, No. 3, 1971, pp. 151-165.

Miller and Maloney. Automatic Mistake Analysis of Digital Computer Programs, CACM, February 1963, pp. 58-63.

Mills, Harlan.Syntax Directed Documentation, Communications of the ACM, Vol, 13, No. 4, p. 216.

Mills, H. Top Down Programming in Large Systems, Debugging Techniques in Large Systems, (Ed.) R. Rustin, Prentice-Hall, 1971, pp. 43-55.

Mills, H.D. Mathematical Foundations for Structured Programming, Federal Systems Division, IBM, Gaithersburg, Md., FSC 72-6012, February 1972.

Nauer, Peter. Proof of Algorithms by General Snapshots, BIT. Vol. 6, No. 4, 1966, pp. 310-316.

Nunamaker, J.F., Jr. A Methodology for the Design and Optimization of Information Processing Systems, Proc. SJCC, AFIPS Press, Montvale, N.J., 1971, pp. 283-294.

Paige, M.R. and Miller E.F. Methodology for Software Validation--A Survey of the Literature, General Research Co. Rm. 1549, March 1972.

Ramamoorthy, C.V. Meeker, R.E. and Turner, J. Design and Construction of an Automated Software Evaluation System, Proc. of the 1973 IEEE Symposium on Computer Software Reliability, pp. 28-37.

Rhodes, J. A Step Beyond Programming, Systems Analysis Techniques, (Eds.) Couger J.D. and Knapps, R.W., John Wiley and Sons, N.Y., 1973, p. 14.

Russell, E.C. and Estrin, G. Measurement Based Automatic Analysis of FORTRAN Programs, AFIPS Spring Joint Computer Conference, 1969.

Rustin, Randall, (Ed.) Debugging Techniques in Large Systems, Prentice-Hall, Englewood Cliffs, N.J., 1971.

Sauder, R.L. General Test Data Generator for COBOL, AFIPS Conference Proceedings SJCC, 1962, pp. 317-323.

Scheff, B. Decision Table Structure as Input Format for Programming Automatic Test Equipment Systems, IEEE Transactions ED-14 April 1965.

Schwarts, J.T. An Overview of Bugs. Debugging Techniques in Large Systems, Rustin, R. (Ed.), Prentice-Hall, 1972.

Stucki, C.G. Automatic Generation of Self-Metric Software, Proc. of the 1973 IEEE Symposium on Software Reliability.

Tucker, A.E. Correlation of Computer Programming Quality with Testing Effort, TM 2219 Systems Development Corporation, Santa Monica, California, January 28, 1965.

Weinberg, Gerald M. The Psychology of Computer Programming, N.Y., Van Nostrand-Reinhold Co., 1971.

Budd, A.E., "A Method for the Evaluation of Software: Executive, Operating or Monitor Systems," Mitre Corporation Report No. MTR-197, Vol. 3. September 1967 (Also issued as EDS-TR-66-113).

Carey, Levi J., "Software Quality Assurance - A State-of-the-Art Report," WESCON, September, 1972.

Brinch, Hansen, P., "Structured Multiprogramming," Comm. of the ACM, July 1972, Vol. 15, #7, pp. 574-578.

Brown, J.R., et al, "Automated Software Quality Assurance," Program Test Methods, Prentice-Hall, 1973, Chapter 15.

Brown, J.R., "Practical Applications of Automated Software Tools, Product Assurance, TRW Systems Group. (WESCON 1972, Session 21, Automating Software Verification).

Gibson, C.G. and Railing, R.L., "Verification Guidelines," TRW Note No. 71-FMT-884, Project Apollo, Task MSC/TRW A-527, 27 August 1971.

Holland, J.R., "Acceptance Testing for Applications Programs," Program Test Methods, Prentice-Hall, 1973, Chapter 20.

Knuth, D.E., "An Empirical Study of FORTRAN Programs," Software Practice and Experience, Vol. 1, pp. 105-133 (1971).

Kosy, D.W., "Annotated Bibliography of Debugging, Testing and Validation Techniques for Computer Programs," RAND Corp., WN-7271-PR, January 1971.

Mangold, E.R., "Software Tools," TRW Systems Group, Redondo Beach, Calif., March 1, 1973.

Mills, H.D., "Mathematical Foundations for Structured Programming," IBM Report, FSC72-6012, February 1972.

Navigation, Institute of, "Software Tools for Certifying Operational Flight Programs," Proc., Nat'l. Space Navigation Meeting, Inst. of Nav., Washington, D.C., March 1967, pp. 164-177.

Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques," WESCON Tech. Paper, August 1970, Vol. 14.

"The Quantitative Measurement of Software Safety and Reliability," TRW Systems Group, One Space Park, Redondo Beach, California, August 24, 1973.

4. PROBLEM OF PROGRAM COMPLEXITY

The goal of this research is the development of a measure, i.e. a yardstick - with which to evaluate a program's complexity. At this stage in software research, there exist few validated tools for program evaluation or for comparisons between software products, except on a gross scale. As Weinwurm (3) points out, there are no generally applicable or empirically validated categorizations for computer programming and further there are no generally accepted, comprehensive, and validated measures of computer program complexity or difficulty. The motivation then for an exploration of complexity lies in the above - to build a valid measuring tool or at the very least, to discover what program characteristics are relevant to the problem of complexity.

The task at hand then becomes essentially a question of where to start. Section 4.1 presents an overview and justification. Here the fact is emphasized that any measurement of an abstract quality, whether it be program complexity or program maintainability, etc. is totally dependent on what is known about the program and its characteristics. Unfortunately, there is a great void of data relevant to software, whether it be with regard to programs' error/change histories, their useability, complexity, or any other program quality. Further, in order to measure an abstract quality, that quality must have a definition in terms of software. This implies that program characteristics on which the abstract quality depends are known. With program maintainability or complexity or similar entities, this is not the case. Therefore, before any measure of this quality,

complexity, can be put to a program, some technique must be developed to ascertain which program characteristics affect complexity, i.e. what program variables are relevant in a measure for complexity.

As Section 4.2 outlines in detail, there are a great number of opinions which have been put forth on the topic of program complexity. None of these have any objective validation to reinforce them but they are the opinions of experts. This is the only information at hand with which to work and any characteristics to be tested as complexity factors must necessarily come from these opinions. When these subjective judgments are used in objective data collection and analyses, they provide a reasonable starting point from which to investigate complexity.

Development of a complexity measure then has been organized into two phases. The implementation phase involved setting up a data collection scheme for analyzing program characteristics deemed relevant to program complexity. The second phase utilizes data collected via this static analysis system from sample COBOL and FORTRAN programs. The data serves as input to statistical analysis techniques such as multivariate analysis, cluster analysis, and factor analysis in order to reduce the data to a set of independent characteristics which effectively measure complexity.

To date, a set of SPITBOL programs have been developed to serve as a vehicle for data collection. Data collection and analysis on FORTRAN and COBOL sample programs is on-going. Section 4.3 outlines this data collection process. Section 4.4 provides detailed descriptions of the types of data being collected. In addition, some preliminary data analyses have begun. The initial analyses on data have been concentrated on statistics

such as means, minimums, maximums, medians, percentages, etc. - attempts to refine the data collected into a set of relevant data points. These preliminary analyses point out various directions in which data analysis could proceed. Section 4.5 contains tables on data that have been analyzed for a set of FORTRAN samples and provides a detailed description of the data categories chosen for a preliminary analysis as well as a detailed discussion of trends seen in this data.

Results so far indicate that the data reflect two ways of viewing a measure - as a set of standard norms against which a program can be evaluated or as a set of factors which can contribute to a complexity score. Preliminary results serve to emphasize that there are a large number of possible complexity factors. Techniques must therefore be developed for investigating the varied combinations of these variables and for structuring these variables into relevant frames of reference so that complexity can be measured via a multi-faceted measure. Section 4.5.2 provides a detailed description of experimental procedures which use statistical techniques and varied frames of reference to develop a measure. In approaching future analyses in this open-ended way, our selection of a measure will not be pre-determined and will be an attempt at objectivity just as our selection of program characteristics for analysis was.

4.1 Overview

The determination of a program's reliability has been approached in a varied number of ways. A number of statistical models, testing methods, programming techniques, and management tools have been developed for the purposes of predicting and improving reliability. Each attempts to attach

a figure of merit to a particular programming effort. Yet, these above techniques are themselves based on subjective hypotheses about software and need a framework of objective data about a particular program in order to function correctly. Therefore, the most important factor in predicting the reliability of a particular programming effort is an understanding of the source of the problem, the program itself. There are a number of program aspects about which various types of data collection can provide information.

An error/change history is a technique for gathering objective data about a program's development. This data can be recorded and utilized by various reliability models to predict the reliability of a program. However, a complete error/change history for a software effort is difficult to obtain with a reasonable amount of accuracy. Few automated tools exist for the recording of such data and manual data collection schemes are on the whole unsatisfactory.

Even with an error/change history, questions such as what program characteristics provide essential information for describing the program - which characteristics affect reliability - what differentiates a particular programming effort from another, etc. are unsolved. These questions are answered by objective data collected on a finished program. While an error/change history gives a picture of a program's stability as it was developed, the data collected on a completed program provides information on the program "as it stands" - i.e. what its relevant measurable characteristics are. In attempting to gather this type of data, there remain the problems of knowing what program characteristics accurately define a program

and what analysis techniques should be used to solve this dilemma.

One method for collecting objective program data for analysis is to measure the finished program in terms of certain of its abstract qualities - e.g. structuredness, useability, maintainability, complexity, etc. In so measuring a program it is possible to get a broad picture of the program via these different aspects and thereby procure information on a wide range of relevant program characteristics. In addition, these program quality measurements can in turn be themselves utilized in a valid figure of merit for reliability - a measurement of software reliability as a function of abstract program qualities.

However, before such program qualities can themselves be utilized in an effective measure or can even themselves be measured, a reasonable definition of each as applicable to software must be decided upon. More importantly, program characteristics which act as independent variables in determining each abstract quality must be found - i.e. what is a valid set of characteristics to use in measuring complexity or maintainability or useability. Indeed, gaining knowledge concerning these characteristics is essentially developing a metric for the abstract quality.

This paper describes a technique for the investigation of one such abstract quality - program complexity. Our method is to approach the problem utilizing program characteristics that various experienced authors have deemed important. The technique then is to use these varying interpretations that have been given to complexity and the various factors ascribed to it in a static analysis of sample programs. The output of this collection scheme can be termed a feature vector of complexity factors

for a program. This vector can in turn be analyzed and refined into a set of independent characteristics which effectively measure complexity.

The following sections provide details on the justification for and background of the problem, the data collection system built, results to date, and plans for future data analysis.

4.2 Survey of Background Information

Multiple definitions for complexity exist. In relevant papers reviewed below, the term "complexity" has been tossed around frequently with a general lack of specific definitions put forth. Complexity has been allied with maintainability, comprehensibility, degree of difficulty, etc. The most flexible policy for this section is to equate complexity with how an author chooses to measure it. This section will put forth the varying interpretations that have been given to complexity, the various factors ascribed to it, and the techniques that have been developed to measure it. Included are any characteristics authors say are relevant to program complexity. Essentially, it is a summary of opinions (most of which have not been validated).

There are three areas where the topic of complexity has been discussed and these will be outlined in this section.

The first area is cost estimation techniques developed for programming projects with emphasis on how a complexity factor is treated in these techniques. The second area presents the various program factors and characteristics used by different authors to define or qualify complexity. The last area for discussion will be a consideration of complexity measures.

No matter which author is read, the sentiments are the same - for most

large, complex software systems there is extremely incomplete knowledge of how to estimate cost, the proper relation of cost to quality, or even what quality is [1]. However, there have been attempts to examine the problem and actual techniques have been developed for cost estimation. Yet, according to Pietrasanta [2], "... the problem of resource estimating or computer program system development is fundamentally qualitative rather than quantitative ..." - "... we don't understand what has to be estimated well enough to make accurate estimates." Presented below are opinions of several authors as to the influence of complexity on cost estimation and examples of costing systems that have attempted to include a factor for complexity.

There exist varying opinions as to what is involved in a complexity factor but most fall into Pietrasanta's description above - there is insufficient understanding of the total problem. Weinwurm [3] maintains that there are no generally applicable or empirically validated categorizations for computer programming and further that there are no generally accepted, comprehensive, and validated measures of computer program complexity or difficulty. Factors such as the number of instructions or subprograms or type of application are components of a measure but they do not by themselves yield consistent and reliable results. He feels that unless experience-data from different computer programming jobs can be normalized to take complexity and difficulty into account, economic comparisons will be misleading.

Pietrasanta [4] discusses a functional estimating procedure - in what is termed a component development phase he wants the following defined: component specification, program and data specification, size, complexity, and so on. Yet he fails to mention what this complexity is and how to

measure it. In another paper [2], he does discuss various aspects of the problem. Below are some of his thoughts:

> "... identify common program sizes of separate components of system since different components vary in complexity and there seems to be a high correlation between complexity and productivity."
>
> "Are large systems nothing more than bigger small systems or are there characteristics of large systems other than size that distinguish them from small systems?"
>
> "... the dominant characteristic of the system spectrum may be system complexity rather than system size."
>
> "... some systems of equal size differ greatly in complexity with a corresponding impact on resource expenditures."
>
> "... returning to the definition of a system as consisting of both elements and interfaces, size relates to elements and complexity relates to interfaces."

Pietrasanta feels that much work still needs to be done in order to quantify system complexity before it can be subjected to an estimating analysis - an examination of the influencing variables and their causal relationships is essential if estimates are to be improved.

IBM [5] has developed some guidelines and a specific technique for cost estimation and they have attempted to include complexity as a factor in cost estimation. They do caution that any method of estimating is no better than the knowledge, experience, and judgment of the estimator and also they state that their proposed technique appears to be more exact than it is. There are eight steps to their estimating procedure, the first of which is the primary concern of this paper - to <u>determine program complexity</u>. The view here is that the complexity of the program depends upon input and output characteristics and the processing functions which take place. Thus, their complexity factor estimation is a 2-step process:

1) weight program I/O characteristics; 2) weight major processing functions. Their weighting points are assigned to such Input/Output characteristics as the card input (single and multiple formats), each tape per input file, each disk per input file, each print per output record format, each tape per output file, card output (single and multiple format) and each disk per output file. This technique also assumes that by a consideration of functions rather than number of program steps, the program's complexity can be more accurately gauged. The functions which are to be applied to the estimation process are: restructure data, condition checking, data retrieval and presentation, calculate, and linkage. If these functions are applicable to a program being estimated, then the estimator, himself, makes a value judgment on whether the function is simple, complex, or very complex and then applies the appropriate weighting. These weightings given to simple, complex, and very complex are not precise and are not intended to preclude the use of judgment. So essentially what IBM has done is to present a guideline for the estimator - these estimates might be far more useful and accurate if there was greater objectivity in the factors chosen to measure complexity and the measure which is applied. Further in IBM's collection of existing material for estimating systems' costs is a list of factors they feel affect programming estimates. Job Difficulty is one of the factors mentioned - its subcategories are the following: complexity of system, number of subprograms, number of data formats, percent clerical instructions, percent transformation/reformatting instructions, percent generation function.

Aron of IBM [5] proffers some ideas on estimation and how to get at

complexity. He ranks sound experience as the most reliable method of estimation - the quantitative methods (such as the method described above) are substantially less reliable and the answers supplied by this type of technique are only approximate representations of system requirements. He mentions the SDS Programming Management Project which attempted to analyze a large amount of historical data to identify factors that affect programming cost the most [6]. The key variables found fall into three groups: uniqueness, development environment, and job type and difficulty. Difficulty is categorized herein as dealing with system interactions due to program and data base elements and the relative variation between different types of programs. Their estimation of difficulty consists of choosing from 3 categories: easy, medium, and hard - this "estimate" should be essentially based on the number of interactions found in the various program classes. However, in a paper which is a planning guide for computer program development, [7] several authors at System Development Corporation do try to be more objective concerning complexity - "... the entry for complexity must be a local standard, such as a scale of one to five, that reflects not only the number of interactions among subfunctions and the number of interfaces with other programs, but also the number and variety of data types input, manipulated, and output; or, the standard could actually be a rough count of these items."

Wolverton of the TRW Systems group has expressed some ideas on the cost of developing large-scale software [8 and 9]. His basic assumption is that costs vary proportionally with the number of instructions. For each identified routine, his estimating procedure combines an estimate of the number

of object instructions, category, relative degree of difficulty and historic data in dollars per instruction from a cost data base to give a trial estimate of total cost. To account for degree of difficulty of a given kind of routine, the designer is supposed to estimate a risk or complexity factor - the most crucial step in the estimating process. The software parameter estimation for a complexity factor is the key problem and Wolverton presents two views of how it should be done. Brandon [9] feels that a single individual should establish a complexity rating scale (A,B,C,D,E,F) and make a "standard" estimate for each job based on: complexity rating of each job, machine used, language used, and estimated number of instructions. Lecht [9], on the other hand, believes that the estimator should interview the member of the technical staff who will do the job and negotiate personal agreement on effort. His estimate would then be based on: similarity with previous modules, person doing the job, the machine used, the language used, and the estimated number of instructions - he does not believe that meaningful performance standards can be set for software. Wolverton's answer then to this complexity factor is simply to ask if the routine is new or old, and if it is easy, medium, or hard - and based on this, to apply sort of complexity rating coefficient. He does however attempt to help the estimator with the above nebulous process for complexity rating - a simplfied version of how complexity might be handled would be to estimate the number of executable instructions, categorize as to type and compare it to others seen to rate the degree of difficulty. His final thoughts in the paper imply, however, that the problem is far from being solved "... what are the crucial parameters that define problem complexity?"

And this appears to be the problem with the above techniques - parameters affecting complexity seem to have been arbitrarily chosen with a minimal amount of objectivity to back up the choice. Even in describing their estimation techniques, the authors warn repeatedly that they are only presenting guidelines. Aside from these estimation processes, complexity has also been mentioned in various articles as the software reliability research has grown. Below are some opinions on what parameters are relevant to the problem of complexity and various techniques espoused by authors to put these in some sort of measure.

Weissman [10] believes that experimental studies should be performed to measure those factors which make programs difficult to understand and maintain. So, his definition of complexity then relates to comprehensibility and maintainability. The author feels that many ideas have been expounded on how to reduce complexity: e.g. documentation standards for programmers, use of high level languages for system implementation, and the idea of structured programming. Yet he feels that articles which extol the benefits of the above fail to give any quantitative evidence that these techniques have in fact improved the quality of programs produced - we "... have passed the point of platitudes and must establish concrete, quantitative evidence of those factors which contribute to program complexity before we can hope to reduce it." A list of factors he feels contribute to the complexity of programs follows below:

I. PROGRAM FLOW

1. Presence or absence of well-placed comments
2. Placement of declarations
3. Paragraphing of program listing
4. Choice of variable names
5. Redeclaration of variable name in inner scope

II. CONTROL FLOW

1. Complexity of control flow graph of program
2. Choice of control constructs
3. Length of program segments
4. Passing procedures
5. Recursion
6. Levels of nesting

III. DATA FLOW

1. Scope of variables
2. Clustering of data references
3. Declaration and use of data structures
4. Locality of operations performed on data structures
5. Use of pointers
6. Arithmetic on pointers

V. INTERACTION BETWEEN CONTROL AND DATA FLOW

1. Flag testing
2. Side effects affecting control flow
3. Changing iteration variable
4. Method of parameter passing

Anderson and Crandon [11] state that complexity depends on much more than size - also, they feel that the undisciplined use of GO TO statements does increase the complexity of a computer program and consequently decreases the reliability. So we have here the degree of program complexity related to a program's reliability. Dicksen et. al. [12] discuss complexity and its relation to program size - a large program is defined as one which is very complex and/or interactive with many instructions whereas a small program is defined as limited in complexity, particularly in the number of branch statements. Rubey et.al. [13] suggest that a language plays a part in the complexity of a programming effort - he feels that programmers tend to write either more complex or more wordy statements in PL/1. The more programmers avail themselves of what PL/1 has to offer, the more complex the whole problem becomes.

At a symposium conducted on the high cost of software [14], several aspects of complexity were mentioned. One of the technical factors responsible for difficulties in obtaining correct software that meets user objectives was seen to be the complexity arising from the mismatch of programming languages to the representational needs of the application domain. Further, the symposium participants felt that the tools for dealing with complexity are the means for abstraction provided by programming languages - in particular, the means for giving structure to programs. Examples would be arrays, list structures, finite set types, and functional data structures. The lack of objective analyses of programs and/or of the programming process was seen to be a major gap in any further progress.

Structured programming has been proffered as a major contributor in reducing program complexity. Mills [15] writes that the purpose of structured programming is to control complexity through theory and discipline - it is seen to be a systematic way of coping with complexity in program design and development. The assumption behind Mills' espousal of structured programming is that he feels the size and complexity of any programming system can be handled by a tree structure of segments where each segment - whether high level or low level in the system hierarchy - is of precisely limited size and complexity.

This concept of structured programming as an aid in reducing program complexity is found in the writings of many authors. Dijkstra [16] wants programs so well structured that the intellectual effort (measured in some loose sense) needed to understand them is proportional to program length - he implies that the "divide and rule" principle will reduce complexity.

IBM's Management Overview [17] has structured programming segmenting code into reasonable amounts of logic that are easily understandable. Donaldson [18] states that structured programming is a technique which has been developed to improve both program complexity and program clarity. He claims that much of a program's complexity arises from the fact that the program contains many jumps to other parts of the program - jumps both forward and backward in the code. Therefore his definition of complexity is eventually equated to flow of control - simplify control paths and reduce complexity.

Ramamoorthy et.al. [19] also state that structured programming is a technique that reduces a program's complexity, and therefore increases its clarity - although they do not substantiate this claim with any proof. Yet, an arbitrary modularization may obscure many interactions (interaction complexity) so that subtle software bugs may be created. Also, unnecessary functional complexity can be introduced by putting too many functions in a module or by failing to abstract a common function shared by different modules. Their opinion is that the complexity of the system will depend on the number of interactions of system components; while at the component level, complexity depends on the number of branches and external references. Reducing a program's complexity can be considered a process of removing obstacles from the program - complicated control paths, obscure structures, uninformative comments, unnecessary jumps, redundant and obsolete code, ambiguous constructs, etc. On the other side of the problem, improving program clarity can be thought of as a process of adding things to the program - meaningful names, informative comments, clear code layout and indentation, more levels of modularization, good documentation, clean interfaces, etc.

The modular programming aspect was investigated by Rhodes [20]. He lists the following as attempts to limit the complexity of a module:

(1) setting a maximum for the number of decision statements

(2) setting a maximum for the possible number of paths through a module

(3) setting a maximum for the number of test data cases required to test modules exhaustively

(4) setting a fixed size of paper to be used to contain the block diagram

(5) setting a maximum development time for the module.

These are attempts to limit complexity rather than explicitly restricting module size. Here it is important to note that Rhodes, too, differentiates between size and complexity - he feels that limiting size does not differentiate between long and simple portions of straight line coding and short and difficult portions of code.

In their discussion of a new concept termed structured design [21], Constantine et.al. stress simplicity of module connections. For FORTRAN or COBOL applications a module can be thought of as a subprogram. They state that the fewer and simpler the connections between modules the better. For the complexity of a system is affected not only by the number of connections but by the degree to which each connection associates two modules, making them interdependent rather than independent. When two or more modules interface with the same area of storage, data region, or device they share a common environment - and each element in this common environment adds to the complexity of the total system. In turn, the complexity

of an interface is a matter of how much information is needed to state or to understand the connection.

An attempt toward a measure of complexity is Goodman's paper on computational complexity [22]. For our present purposes, computational complexity is not applicable but Goodman does discuss a definition for a complexity measure as a scheme for measuring a specific type of complexity. A complexity measure is some function of the amount of a particular resource used by a program as it processes a specific input value - this resource might be time, space, CPU usage, channel activity, etc.

Clapp and Sullivan [1] also have views on the complexity issue. Two of their hypotheses relevant to the topic are the following:

1. Structured programming leads to greater comprehensibility and reliability.
2. Complexity (the inverse of comprehensibility) and the cost of debugging are strongly covariant.

Their view of complexity is in terms of the number of independent paths and Sullivan [23] has written a report on an application of this towards an actual measure of complexity. He presents several measures of computer program complexity, in the sense of comprehensibility or intellectual manageability. He defines the "C2 complexity" at any node of an elementary scheme to be one less than the number of paths from the start node to that node, not counting paths where any node occurs more than x times, where x = 2 unless otherwise stipulated. The complexity of the composite scheme is then defined as the sum of complexities of its elementary subschemes. So, essentially Sullivan counts literally every path through any elementary

scheme. However, he does feel that a measure should be sensitive to the distribution of references to a data object over segments of the control graph, and so he attempts to define a process complexity measure at a node in terms of just those process operations (data references) relevant to it and the paths among those operations. This second measure is at the present time untenable and does not really sufficiently handle data node complexity.

Another measure of complexity has been proposed by Peterson et.al. [24]. They propose to measure the complexity of a flowchart via a pair of integers (N, M) where N is the number of nodes in a largest multi-entry component [a multi-entry component is equivalent to a loop (any sequence of statements capable of being executed repeatedly) where there exist paths, whose nodes are not elements of the loop, from the start node to more than one node in the loop] and M is the number of multi-entry components with N nodes. In this article, a flowchart is equivalent to the flow graph of Aho and Ullman [25], - a 2-dimensional representation of a program that displays the flow of control between basic blocks of a program. A basic block is defined as a group of statements such that no transfer occurs into a group except to the first statement in that group and once the first statement is executed, all statements in the group are executed sequentially. There is at least some justification for an investigation of a program's flow graph as de Balbine [26] and others have stated that the structure of the flow graph alone is sufficient to perform a good restructuring of a program. However, the above measure by itself is not sufficient as an indicator of program complexity as was pointed out in Knuth's study [27]. The majority of FORTRAN programs he analyzed in his survey had no multiple entry loops.

Mills [28] tends to negate these rather simplistic approaches to a complexity measure -

> ... measuring the complexity of programs is no simple task. It is easy to form simple hypotheses about such measures, but it is just as easy to demolish them with counter-examples of common experience. The idea of equating complexity with the difficulty of understanding a program has been generated out of the frustrations of concocting and demolishing more simple-minded, direct ideas, such as counts of branches, data references, etc.

He feels that structured programming is a first broad attempt to deal with the complexity of control logic in programs - yet the control of data reference complexity is as important but as yet there has been minimal work done. The current trends in programming theory - subroutines, multiprogramming, etc. are attempts to factor problems of complexity into smaller units comprehensible by human intelligence. Mills then postulates that the complexity of a program is equivalent to the difficulty of proving the program correct. Since proofs of program correctness are barely in the preliminary stages, this is not a feasible approach.

Within these three areas discussed above, there has been little quantitative or qualitative evidence to validate one opinion or another. The major problem is that there have been far too many subjective opinions offered and just as many simple hypotheses formed on what is a valid complexity measure. Many of the individual factors which have been put forth such as the number of executable instructions or the number of inter-program and intra-program interactions or the number of independent paths are perhaps components of an effective measure but each by itself is not a consistent or reliable variable. We must know what to measure in order to know.

Therefore, our first step is to develop methods which will enable a determination of what programming characteristics are indeed relevant to program complexity.

4.3 Data Collection System

The goal is to build a complexity metric - "a measure of the extent or degree to which a program possesses and exhibits complexity" [29]. The preceding section provided insights from various authors as to what program variables constitute a complexity factor. Several complexity measures were also described based on program characteristics which may be relevant in some circumstances but which lack any sort of objective proof or data to reinforce them. The major fault with the hypotheses offered to date is that they assume the measure - e.g. number of independent paths - and proceed to build their analysis technique around it. As Knuth's study [27] pointed out, too often what we hypothesize to be important is quite the opposite from what actual program data shows to be reality.

With the opinions so diverse and varied as to what is important and what is not with regard to complexity, it seems far more logical to base a measure on the assumption that program complexity is a function of several variables and not just one or two. Yet, which program variables are the correct ones to use in a measure poses a major problem. Since a metric should "correlate well with established notions of software quality" [29], it seems feasible to develop a technique for investigating which program variables are factors in a measure of complexity by utilizing as many established notions as possible. A data collection scheme whereby data based on these pre-established characteristics can be collected from sample programs and analyzed would serve such a purpose. This data - essentially a feature vector of possible factors for a complexity measure of the sample program - can in turn be analyzed via multivariate analysis

or factor analysis methods. These techniques will determine the "fundamental and meaningful dimensions" [30] of the vector's domain (equivalently, a measure) - i.e. which of the variables serve as the minimum number of dimensions required for the description of differences between samples analyzed.

The following paragraphs will outline the implementation of the above ideas for data collection, will discuss the sampling procedure and sample programs collected, and the data analysis techniques to be used.

4.3.1 Data Collection

In order to analyze programs with respect to specific variables, a data collection system had to be set up. As can be seen from Section 4.2, there are a wide variety of program characteristics said to affect complexity and therefore the data collection must be extremely flexible. Information on items such as the type and purpose of the program, the language written in, the environment of the program, its subprogram interactions, the control and data flow of the program, the program size, instruction mix characteristics, etc. must be collected.

In addition to its flexibility, the system had to be fairly inexpensive to use, automated, and based on an analysis for program samples which would be easy to obtain. Manual systems, by their very nature, rely too heavily on cooperation by people not involved in the analysis project and, most importantly, not interested in the project. Evidence to this effect can be seen in some of the error/history data collection schemes to date [31, 32]. Ideally, analysis of a program by the data collection system

should be independent of the author/maintainer of the program.

The ability to sample a wide variety of programming styles within a language as well as the desire to obtain knowledge about frequently used and easily useable languages prompted the data collection system to be set up for the analysis of COBOL and FORTRAN source programs. These two languages are the most heavily used by people at the Data Processing Center and there exists a broad sample of users' programming styles available as will be discussed below.

The SPITBOL language was chosen as the vehicle for system implementation. The data collection involves the manipulation of a lot of data in the forms of lists, counts, tables, and strings. These have to be handled quickly and efficiently. SPITBOL was designed to facilitate these difficulties, i.e. it is specifically a pattern-recognizing and string-manipulating language.

4.3.1.1 Manual Data Collection - Questionnaire

Unfortunately, not all of the necessary information is collectible through an automated program analysis. The author's/maintainer's input is needed for a minimal description of the program as outlined below and for an initial complexity rating of the program. This complexity rating is useful in differentiating between the sample programs collected and provides some input from the author/maintainer as to his/her opinion of the program's complicatedness. Therefore, before describing the SPITBOL programs that constitute the formal data collection system, the additional information collected on the programs is discussed below. The questionnaire

was kept as "objective" as possible and only entails a few minutes of a programmer's time to fill out.

QUESTIONS:

1. Language	
2. Type of Program	(How would the programmer classify this code.) e.g. Data Manipulation, File processing, statistical, computational, etc.
3. Purpose of Program	(Brief description of the function of the program.)
4. Batch or Inter-active	
5. Frequency of Use	(The running schedule for a program affects the style in which it was written.)
6. On a program complexity rating scale of 0 through 10:	(A means of differentiating between programs.)
A. How would the general category of programs named in question 2 above rate on this scale?	(Attempt to categorize the programmer's evaluation of his type.)
B. How would the particular program named in question 2 above rate on this scale?	(How does this program rate with respect to the programmer's other "works"?)
C. Why? i.e. what particular program characteristics affected your rating?	(How does the programmer regard complexity?)
7. Are you the program's author or maintainer?	(Perspective on the person's attitude toward the program.)
8. A. How large is the program?	(A through E relates the program to be analyzed within its total environment.)
B. Approximately how many lines did you write?	
C. Does the program use system utilities?	

D. Is the program part of a larger system? If so, approximately how big is this system? and how much of it are you responsible for/have control over?

E. Are all modules written in the same language?

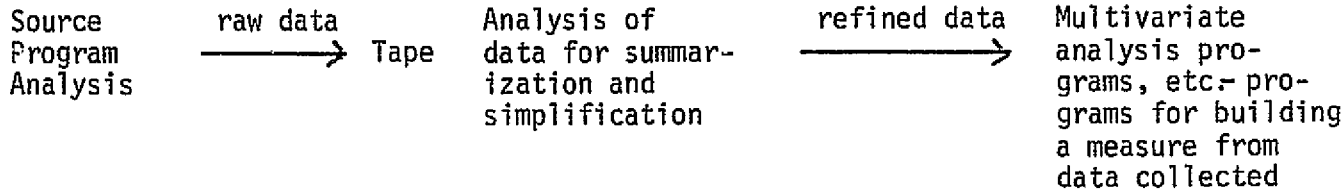
9. Does this program utilize special techniques? e.g. use structured programming? is it modular? top-down design, etc. (These factors should be evidenced in the analysis -- knowing the programmer used special techniques facilitates an analysis of whether these techniques indeed differentiate between this program and others not utilizing them.)

10. Are there hardware devices necessary for the program's execution and how many? e.g. tapes, disks, terminals, plotter, etc. (Device-dependedness)

11. How would you characterize the files used in the program? e.g. stand-alone, shared (under your control?) etc. (How much "outside" interaction is there?)

4.3.1.2 Automated Data Collection - Source Program Scanner

As was mentioned earlier, the automated data collection system consists of a set of analysis programs written in SPITBOL. The programs for the actual data collection are a set of eight SPITBOL programs (SNOINST, SNODATA, SNOCONTR, SNOCONTR2, SNOCINST, SNOCDATA, SNOCCONTR, SNOCCONTR2). These programs analyze source code and result in a set of data points for the program which are in turn output to a tape. The raw data on tape is then available for further analysis by a series of programs which will be discussed in Section 4.3.3. Due to the expense of running large-sized programs in any language, not just SPITBOL, it was decided that the data collection system should be broken up in the following way:



The specific characteristics analyzed by these eight static analysis programs named above are important in themselves and these SPITBOL data collection programs are only a tool for obtaining this information. Accordingly, specific complexity characteristics analyzed via the eight programs will be discussed in their totality in Section 4.4. What will be pointed out below are some of the problems which occurred in building the tools for data collection - i.e. the problems involved in writing the programs and the specific routines that had to be written to obtain some of the more hidden information - e.g. a flow graph. It must also be further emphasized that at a certain point program length starts to retard program efficiency with SPITBOL. In particular, the frequent use of tables, arrays and fairly long pattern strings made it much more efficient and maintainable to build the system out of eight small and fairly simple "pieces," each piece having a specific function.

The eight programs used for data collection each perform a different task as follows:

SNOINST -

Length: $\sim$ 700 source statements (excluding COMMENTS)

Input: FORTRAN program plus its subprograms

Purpose: Analyze a FORTRAN program in order to collect data on instruction mix characteristics, some control flow information, and subroutine nesting data.

SNODATA -

Length: ∿ 410 source statements (excluding COMMENTS)

Input: FORTRAN program plus its subprograms

Purpose: Analyze a FORTRAN program in order to collect data on program variables' locality and reference, some control flow information, levels of DO loop nesting, and other pertinent DO loop information.

SNOCONTR -

Length: ∿ 505 source statements (excluding COMMENTS)

Input: FORTRAN program plus its subprograms

Purpose: Analyze a FORTRAN program and collect structure and control flow data, parameter nesting levels and function reference data. The program's primary purpose is to build a flow graph for a FORTRAN program.

SNOCONTR2 -

Length: ∿ 325 source statements (excluding COMMENTS)

Input: Structure and control flow data from SNOCONTR

Purpose: Analyze program loop structures, spans of branches, and implement the complexity measure of Petersen et.al. [24] described in Sections 4.2 and 4.4

SNOCINST -

Length: ∿ 850 source statements (excluding COMMENTS)

Input: COBOL source program

Purpose: Analyze COBOL program in order to collect instruction mix characteristics and some control flow information.

SNOCDATA -

Length: ∿ 400 source statements (excluding COMMENTS)

Input: COBOL source program

Purpose: Analyze a COBOL program in order to collect data reference and locality information.

SNOCCONTR -

Length: ∿ 505 source statements (excluding COMMENTS)

Input: COBOL source program

Purpose: Analyze a COBOL program in order to collect structure and control flow data. The primary purpose of the program is to build a flow graph for a COBOL program.

SNOCCONTR2 -

Length: ∿ 400 source statements (excluding COMMENTS)

Input: Structure and control flow data from SNOCCONTR

Purpose: Analyze program loop structure (exluding branches caused by PERFORMS), spans of branches, and implement the complexity measure of Petersen et.al. [24] described in Sections 4.2 and 4.4.

4.3.2 Program Samples

To ascertain what variables affect program complexity, sample programs must be analyzed to see which of these complexity characteristics appear often enough to be relevant. Outlined above was a system which does a static analysis of both FORTRAN and COBOL source programs. But a data collection system is useless without a good set of samples to analyze. Therefore, it was necessary to collect FORTRAN and COBOL programs.

Debugged production programs, written by full-time programmers and non-student personnel were desired.

A university environment is ideal for this sort of problem as its computer users run full spectrum from the professional programmer to the scientist who is strictly FORTRAN-oriented and not schooled in new theories such as structured programming. In addition, the DOMONIC system was written utilizing the concepts of structured programming. Therefore, its modules offer a contrast to the orientation of other COBOL application programs available via the Data Processing Center. So there exists a sufficient range of samples to select from and the problem becomes how to collect these sample programs.

With any kind of data collection scheme there has to be some dependence on people. They in turn tend to be far more cooperative when their input activity is minimal but the return on their time investment is maximized. Therefore, the sampling procedure consisted only of two steps:

(1) a questionnaire to be filled out

(2) collecting three programs which the person sampled had rated as simple, medium, and complex.

The sample group was requested not to make size a differentiating factor in the programs they chose as samples. Emphasis was placed for their decision to be based on specific program characteristics they felt affected the complexity of their programs and not on the fact that one program was 1800 statements long, another 900 statements, and another 150 statements. Program size is obviously a factor affecting complexity but often it can be an overwhelming one and obscure other just-as-relevant characteristics.

When asked to help in this project, most people were very enthusiastic about wanting to see results of the analysis - i.e. they were quite curious as to how their programs related to others, what were its important characteristics, etc. This enthusiasm carried over into their selection of sample programs for analysis. There is a broad spectrum of varying degrees of complexity among the programs received so far.

Approximately fifty sample FORTRAN programs are available for analysis at the present moment and they have been collected from people in academic departments, research groups, and the Data Processing Center. These FORTRAN programs range in size from a couple of hundred statements to combined program/subprogram size of over 2500 statements. The FORTRAN programs also vary in type. We have access to text editing programs, a flow charter, data editing programs, insect population models, a light penetration model, statistical-oriented applications, financial and budget analysis programs, an analog-to-digital processor program, and mathematical and scientific-oriented programs.

Approximately twenty-five COBOL programs have been collected and an attempt was made to cover a range of programming styles. The DOMONIC system provides data on programs written in a structured programming style. This system was coded in a top-down design and therefore sample modules range from drivers of major system commands through single-purpose modules. Also, it was possible to collect sample COBOL programs from people who were not the author of a set of programs but who now must maintain these programs. This provides a different viewpoint than that of the author who is also the maintainer of a program.

Hence, the sample collection process is well underway and there are a sufficient number of samples at hand to perform worthwhile analyses.

4.3.3 Data Analysis

The previous two sections outlined the major portion of the data collection system. The purpose of this section is to briefly outline the kinds of data retrieval and analysis programs that are needed to perform basic analyses on the data collected.

There are many varied statistics collected on each program and therefore the possibilities for worthwhile results from even simple analysis routines are quite promising. More elaborate schemes for data analysis will be discussed in Section 4.5.2.

The static analysis routines of the data collection system produces a feature vector - $f(X_1, X_2, \ldots, X_n)$ - of complexity characteristics, X_i, for a program. These feature vectors in turn will be utilized in various statistical analyses to gain insights into the hierarchial classifications among the variables within the vector and between the vectors themselves. That is, the final output from analysis will be a complexity measure = $F_c(X_1,X_2,X_3,X_4,X_5)$ such that

$X_1 = g_1$ (DATA LOCALITY)
$X_2 = g_2$ (INTERACTIONS)
$X_3 = g_3$ (CONTROL FLOW)
$X_4 = g_4$ (STRUCTURE)
$X_5 = g_5$ (INSTRUCTION MIX)

Refinement of the raw data collected and previously put on tape for each sample program consists of programs which will compute total counts, percentages, maximums, minimums, means, etc. For example, each sample has a count taken of its instructions. These counts in turn can generate a large number of varied statistics depending on the emphasis of the analysis, e.g. percent sequential versus percent non-sequential; percent I/O versus percent computation, percent conditional statements; etc. Also, for each DO statement in a FORTRAN program, the nesting levels within the range of the DO have been collected. So all these nesting level counts for a program must be summarized via a maximum or mean statistic.

The refined data for each program must be utilized in at least a minimal analysis to get data points such as means, maximums, and minimums between each of the vectors, i.e. inter-vector measurements. Examples would consist of entities such as mean number of loops within all FORTRAN sample programs analyzed or mean number of breaks in sequential flow for programs with complexity rating of 0 - 4, and in the 5 - 7 rating group, and the 8 - 10 group. There is a great deal of information that can be processed via analyses such as these.

There are also inter-language analyses that can be run. For example, FORTRAN and COBOL both allow interactions in the form of subroutines and also instruction type comparisons in such forms as numbers of conditional type statements vs. unconditional type statements vs. sequential statements. FORTRAN and COBOL programs are reducible to a flow graph which itself eliminates the language barrier and shows only control flow. Therefore, comparisons of various data points between the two languages are possible.

Beyond the above types of analyses there are various factor analysis and multivariate analysis techniques available to aid in differentiating between variables within a set of vectors. Packaged routines will be used for applying these statistical methods to analyze the refined data further.

4.4 Complexity Characteristics

This section will enumerate in detail the complexity characteristics collected by the static analysis routines described previously. It is important to note that this is a description of the raw data which must in turn be refined and used as input into other data manipulation schemes which allow more generalized categories. For example, the counts of various instruction types can be categorized into percent executable, percent non-executable or percent specification, percent subprogram, percent iterative, percent computation, percent conditional, percent unconditional transfer. These types of categories in turn are independent of language which allows flexibility for analyzing the data across languages.

There are obvious differences between languages as to the kinds of detailed program characteristics that can be analyzed via a static analysis system. For example, PL/1 has more language features and therefore has more measureable characteristics than FORTRAN and COBOL. On the whole, the collection scheme centers on characteristics of the program itself, and not on characteristics of the language in which the program is written. Data points measured for FORTRAN or COBOL programs can be categorized under general headings, although the measured characteristics under these headings may be language dependent. It is interesting to note here that the differences in the lists of characteristics under each heading for FORTRAN and COBOL serve to underscore the strengths and weaknesses of each of these languages.

There are a great number of complexity data points collected. All of this data is in some aspect relevant to the problem. No one of these variables in itself constitutes a solution as a measure of complexity. The

discussion in Section 2 concerning measures which have been proposed to date emphasizes this fact. However, analyses of these characteristics in various groupings should provide a combination(s) of program variables that indeed do define program complexity.

When analyzing a program it is most important to examine its total environment. Both the common environment a program "shares" with other programs/subprograms and the environment of the program as an entity unto itself must be examined. Therefore, program characteristics affecting complexity have been categorized into two areas: those that are concerned with a program's interaction with other independent programs/subprograms and those that concern the program itself. The discussion below outlines capabilities of the data collection scheme in terms of the complexity data points analyzed.

4.4.1 Program Interactions

In this category, emphasis has been placed on what program characteristics affect interactions between the program and other programs/subprograms - i.e. the environment of the program within a total system. This can include shared files, file manipulation, subprogram connections and interfaces, shared data, etc. Even though program simplicity can be enhanced by dividing a program into a system of separate pieces each of which is an independent entity, complications can and do arise with the coupling effects of such a division. Sheer numbers of connections between pieces and the various types of interfaces necessary for communication provide an enormous impact on the complexity of a single piece of a system.

Accordingly, the data collected under this heading leans toward subprogram interaction and the varied types of parameter passage. Emphasis has been placed not only on the number of connections a program has with other programs but also on the data involved in this connection - how many parameters are passed, how complicated is the parameter list expression, how "far-reaching" is the value of the variable passed, etc. We are trying to measure the potential impact of its common environment on a program.

The data points collected are as follows:

COBOL

I. Connection Information:

- number of entry points
- number of subroutines called
- number of times each subroutine called
- nesting level of each subroutine call

II. Interface Information:

- number of linkage parameters per entry point
- number of linkage parameters for main program
- number of parameters passed for each subroutine call
- total number of variables in the LINKAGE SECTION
- Total number of COPY variables in the LINKAGE SECTION

FORTRAN

I. Connection Information:

- number of subprograms with multiple returns plus the number of returns for each

- number of subroutines called plus the number of times each called
- number of function statements plus the number of times each referred to
- number of function subprograms plus the number of times each referenced
- number of calls to FORTRAN supplied functions plus the number of times each called
- number of substitutes for external variables called (counted only in subprograms where external variables have been passed)
- number of external variables referenced in a set of arguments passed to a subprogram (counted only in main program)
- number of entry points in each subprogram
- nesting level of each function reference in the main program
- number of LABELED COMMON areas plus the name of each area and the number of variables in each area

II. Interface Information:

- number of variables in blank COMMON
- number of parameters passed to each function reference (plus the number of computations and function references in the argument list)
- nesting levels of parameters

  A. For each parameter in the argument list of a subroutine or subprogram definition the following data is kept:

1. the nesting level(s) of the parameter
2. the total number of function references the parameter is passed through
3. a string of (non-duplicated) function names through which the parameter's value is passed (See Figure 1)

B. For each parameter passed in a function reference's argument list both in the main program and each subprogram the following data is kept:
1. the nesting level(s) of each parameter
2. the total number of function references the parameter's value passes through
3. the string of non-duplicated function names the variable passed through

(Note that in both A and B above, part '3' gives the number of unique function references for a parameter while '2' yields the total number of such references.)

4.4.2 Characteristics of the Program as an Independent Entity

Within the program itself - neglecting its shared environment - there are a number of program characteristics that have been proffered as variables affecting complexity. These can be categorized under three general headings - data reference, structure and control flow, and instruction mix characteristics. Data falling into these headings have been analyzed with regard to the program as it stands. Justification for the enumeration of many of these

FIGURE 1

```
SUBROUTINE    SUB1(X,Y,Z)
   .
   .
   .
CALL SUB2(X,A,B)
   .
   .
   .
AC = SIN(X)
AK = TAN(Y)
CALL SUB2(X,AC,B)
   .
   .
   .
RETURN
END


SUBROUTINE    SUB2(Q,AC,B)
   .
   .
A = B + SIN(Q)
   .
   .
   .
STOP
END
```

PARAMETER INFORMATION:

Argument	Nesting Levels	Number of Function References	Functions Passed Through
X-SUB1	/2/1/2/	5	SUB2/SIN
Y-SUB1	/1/	1	TAN
Z-SUB1	-	-	-
Q-SUB2	/1/	1	SIN
R-SUB2	-	-	-
S-SUB2	-	-	-

SAMPLE FORTRAN PROGRAM WITH

PARAMETER DATA COLLECTED ON ARGUMENTS

variables as complexity characteristics has been outlined previously. Further explanation for some of the variables collected - especially in the area of structure and control flow - will be rendered as is necessary. Section 4.4.2.1 discusses instruction mix characteristics while Section 4.4.2.2 is concerned with data reference variables and Section 4.4.2.3 deals with the measurement of structure and control flow.

4.4.2.1 Instruction Mix Characteristics. The numbers and types of instructions used in a program is an indicator of how the programmer used a language and is an obvious reference point in describing a program. An instruction mix indicates the kinds of control constructs governing the program; how much of the code is actually executable; how understandable the program is via the number of comments, etc. There are also other types of variables related to instruction counts which are analyzed.

Information can be collected on simple assignment-type statements versus computation-oriented assignments. For a language such as FORTRAN which is compute oriented it is important to consider factors such as function references and computations which complicate subscripting as well as various instruction types. There are verbs in COBOL such as the SORT that can be as complicated as a programmer chooses. With these particular verbs, data should be collected on all relevant aspects of the verb's use. COBOL also has the capability for nesting IF statements. This can greatly complicate a program's structure as well as its readability et al.

The size of a program is an important characteristic for measurement, as is repeated by many different authors. Size for a COBOL program can be

measured by the actual number of verbs and also by the number of statements (80 character lines). The number of labels, for both FORTRAN and COBOL, and particularly the number of statements between labels yields a rough picture of how many segments there are to the program - i.e. how many program pieces must be kept track of.

The following is a list of data obtained under the heading of instruction mix:

COBOL

- size of program - number of statements and number of verbs in procedure division
- number of paragraphs
- number of sections
- segment sizes - number of statements in each paragraph - number of verbs in each paragraph
- number of outer nested IFs - the number of IFs that either stand alone or begin a nested sequence
- nesting level of IF - gives depth of nesting (plus breadth) for every nested branch
- SORT verb data - indicates how the SORT is done - via input procedures, output procedures, files, etc.
- ENTER verb data - number of routines to be executed in another language
- SET verb data - number of assignments in the statement
- COMPUTE verb data - number of operations in the statement

- ADD and SUBTRACT verbs data - number of assignments in the statement
- number of instruction types - total number of instructions plus the number of each different instruction type

FORTRAN

- size of program
- number of labels
- segment sizes - number of statements between each label
- number of computations per statement - this count is kept for any statement in which an arithmetic operation can be performed
- number of function references per statement - this count is kept for any statement in which a function reference is possible
- number of instruction types - total number of instructions plus the number of each different instruction type (IF statements are counted individually under the statement following the conditions)

4.4.2.2 Data Reference. Ideas describing what is and is not important to measure for data variables have been tossed around as frequently and as madly as FORTRAN programmers use 6-letter non-mneumonic names in their programs. Data locality, scope of variable reference, structure of the data, etc. are entities that authors say should be investigated and yet it is difficult to find consensus definitions of these terms and worse to locate algorithms with which to measure them. Therefore, data has been collected

on what is reasonably attainable via static analysis, without resorting to complicated traces through various paths to trace data definition points.

Information on counts of different variable types, sheer numbers of variables, numbers of input variables, as well as specific information pertaining to the locality of reference for a variable is available for analysis. One of the most important characteristics of a program variable is that its value must be retained by the programmer for as much of the program as it is referenced. Each variable serves then as something that must be remembered by the programmer/maintainer as the program is being used. The span of a variable is an attempt to measure this characteristic in a superficial sense - i.e., for what percentage of the program is a variable a retention problem. The frequency of reference for a variable within this span is measured by the average distance between references to the variable.

The DATA DIVISION of COBOL specifically defines every program variable. What is of interest, though, is how these different data types are used within a program - i.e. are the group hierarchical variables used more frequently than the elementary level, how often are copy variable names referenced, etc. FORTRAN does not offer this wide variety of data use but relevant counts can be made of numbers of input variables, dimensioned variables, etc.

The following are data reference variables collected for analysis.
Data reference information:

COBOL

- total number of variables used in the Procedure Division
- the number of COPY variables defined in the Data Division

- the number of variables defined in the LINKAGE section
- the number of files READ plus the variables affected by the INTO option of the READ
- for each variable:
    1. the number of references to the variable within the program
    2. the span of the variable - the percentage of the program spanned by the variable from its first to last reference
    3. average distance between references within the span
    4. type of variable - an indicator of whether the variable falls within any of the following classes: COPY, GROUP, ELEMENTARY, LINKAGE, RENAMES, CONDITION, OCCURS, REDEFINES, paragraph name, function name

FORTRAN

- total number of variables
- number of function variables
- number of dimensioned variables
- number of input variables
- for each variable:
    1. the number of references to the variable
    2. span of the variable
    3. average distance between references to the variable within its span
    4. type of variable - an indicator of whether the variable is DIMENSIONED, EXTERNAL, FUNCTION, or INPUT

NOTE: if the variable has been passed as a parameter in an argument list for some function reference, depth of nesting information is collected as described in Section 4.4.2.

4.4.2.3 Structure and Control Flow. In a discussion of structure and control flow, there are the obvious program variables which should be enumerated - e.g. numbers and kinds of control constructs, number of times sequential flow altered by a jump upward or downward, numbers of conditionals and unconditionals and language-defined loops, levels of DO loop nesting and sizes of DO loops, ad infinitum.

These types of variables are collected for analysis as follows.

Basic Structure Data:

COBOL

- number of each conditional type - conditional types are: IF, ON, SEARCH, EOP, INVALID, END, SIZE
- number and type of instructions executed on each condition - conditional types are: IF, ON, SEARCH, EOP, INVALID, END, SIZE
- number of branches in GO DEPENDING
- number of ON conditions in a SORT
- number of when conditions in a SEARCH
- number and type of test for each condition - (SIGN, CLASS, CONDITION, RELATIONAL)
- number of logical conditions evaluated - count the number of connectives plus the number of conditional operators
- number of times the sequential flow altered - number of jumps in

program (indicates whether an upward or downward flow)

- number of alters in an ALTER statement

FORTRAN

- number of unconditional statement types - unconditionals: RETURN, STOP, GO TO
- number of conditional statement types - conditional statement types: COMPUTED GO TO, ASSIGNED GO TO, ARITHMETIC IF, IF..., READ W/ERR=, READ W/END=, CALL WITH MULTIPLE RETURNS
- number of DO loops
- length of each DO loop - number of statements in the loop
- nesting levels of DO loops - counts every nesting sequence within an outer DO
- number of nested DOs within an outer DO
- number of conditions evaluated in an IF condition - counts all connectives and relational operators
- number of function references and computations involved in a conditional expression - to indicate complications involved in the logical expression
- number of times sequential flow altered - number of jumps in the program - indicates whether an upward or downward jump

The above mentioned variables are relevant to program structure and do provide necessary data points for a description of a program. However, no one of these characteristics in itself sufficiently handles the problem of measuring structure. Each adds a dimension to a measure but does not

yield an all-encompassing figure of merit for the structure of a particular program. It is undecided among authors just what kind of a measure is a reasonable one for handling the structure of a program. Below are some opinions on the topic.

Gileadi and Ledgard [33] have written a paper concerning a measure of program structure. They attempt to measure how well-structured a particular flowchart is with respect to the precepts of structured programming. An abstracted flowchart of a program is mapped into a deterministic finite-state sequential machine of the Mealy type and then a measure of software-work is applied to the automaton. Implicit in their work is an assumption that the flowchart doing the least amount of work to compute a given function is in fact the best structured. The authors present a minimal example of their technique and offer no proof for their assumptions. Further, their ideas are impractical for large programs.

De Balbine [26] has written an automated tool for the purpose of rewriting existing programs to make their logic more understandable. He claims that the structure of a program's flow graph alone is sufficient knowledge in order to perform a good restructuring of the program. Meissner [34] also emphasizes the flow graph as a means of providing independent structural information. Peterson et. al. [24] have devised an algorithm to restructure a program directly via its flow graph.

The much-heralded structured programming has been promoted as a cure-all for structuring difficulties. If a program is written according to the precepts of this discipline then the program is supposedly far less complex and eminently understandable (Donaldson et. al. [18]). In examining various

opinions on structured programming, authors point up program characteristics such as: no jumping around, simple control paths, single entry, single exit, minimal use of go to, read strictly from top to bottom, no back-tracking, direct correspondence between the static form of the program and its dynamic flow during execution, program divided into easily understandable units, etc. Also, there exist restructuring algorithms (e.g. reference [24]) which make use of a flow graph in transforming a program to a structured form.

But does this mean that programs cannot be written without using the specific key constructs of structured programming and still exhibit the above-mentioned characteristics of good structure? Is the problem that FORTRAN or COBOL programs lack structure or rather that their structure may be difficult to discern? Meissner [34] contends that the key control statements of FORTRAN really do little to enhance the recognition of good program structure. Therefore, what is needed is an objective way of examining a program, structured or otherwise, to discern what is bad structure from what is good structure. In so doing, the ideas stated above as to what program characteristics affect structure can be utilized.

These characteristics of good structure will be analyzed as they appear in arbitrary FORTRAN and COBOL programs. Included in this analysis is an application of an algorithm used in a mechanical restructuring method [24] which determines how much effort is involved in restructuring a program. This algorithm will be defined further on. This approach will utilize the flow graph of a program both for its measureable characteristics and for its facility in finding other structural data points. This flow graph analysis

is independent of whether the program is written in COBOL or FORTRAN with one exception. This has to do with counting the number of loops in a COBOL program and will be discussed further in the narrative.

As the specific structural data points are enumerated, definitions and explanations will be inserted where applicable.

CONTROL FLOW INFORMATION - FLOW GRAPH DATA

A. Flow Graph

1. Definitions: (The following is from Aho and Ullman [25] - cf. Figure 2 for an application of these definitions)

- Defn: A statement S in a program P is a basic block entry if
  a. S is the first statement in P or
  b. S is labeled by an identifier which appears after GO TO in a GO TO or conditional statement, or
  c. S is a statement immediately following a conditional statement.
- Defn: The basic block belonging to a block entry S consists of S and all statements following S
  a. Up to and including a halt statement or
  b. Up to but not including the next block entry.
- Defn: A flow graph is a labeled directed graph G containing a distinguished node N such that every node in G is accessible from N. Node N is called the begin node.
- Defn: A flow graph of a program is a flow graph in which each node of the graph corresponds to a block of the program. Suppose

```
                read p
                read q
loop            r  ← remainder (p,q)
                t  ← r*r
                if  t = o   go to done
                p  ← q
                q  ← r
                go to loop
done            write q
                halt
```

Figure 2A - Sample Program

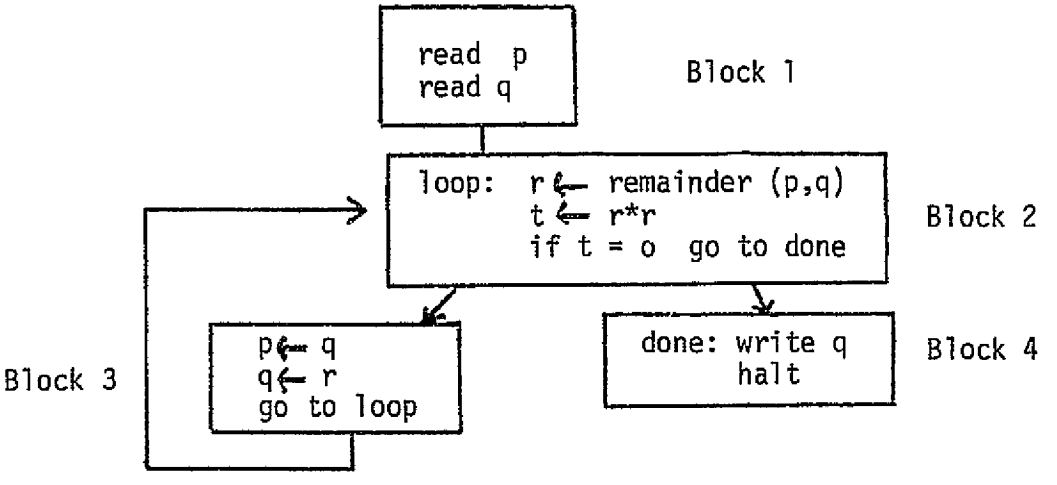


Figure 2. Application of Flow Graph Definitions

that nodes i and j of the flow graph correspond to blocks i and j of the program. Then an edge is drawn from node i to j if

a. the last statement in block i is not a go to or halt statement and block j follows block i in the program, or

b. the last statement in block i is GO TO L or IF...GO TO L and L is the label of the first statement of block j.

2. Data Collected from Flow Graph

- total number of basic blocks in the program
- size of each block - number of verbs in each block
- blocks flowed to <u>from</u> each block
- number of branches <u>to</u> each block

Note: Figures 3A and 3 present an example of this flow graph data.

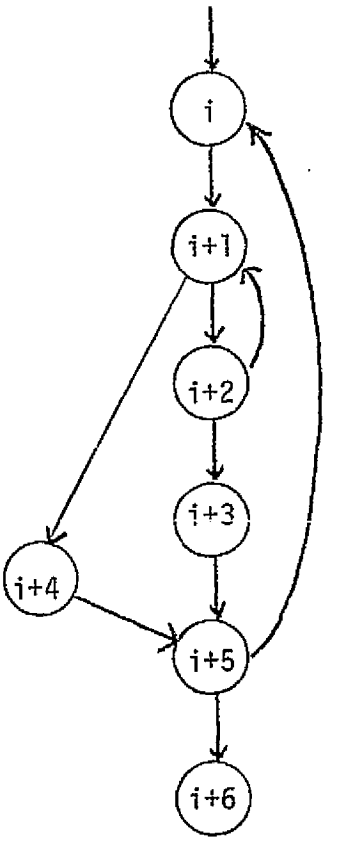
B. Interval Analysis on Flow Graph

What is useful about interval analysis is that it places a hierarchial structure on the program; divides the program into pieces which can be examined independently, and it eliminates some of the bowl-of-spaghetti effect by eliminating sequential and unimportant branches.

1. Definitions:

- <u>Defn</u>: A technique used in data flow analysis for compiler optimization. It is ar algorithm for partitioning a flow graph uniquely into disjoint intervals as follows: If h is a node of a flow graph F, define I(h), the interval with header h, as the set of nodes of F constructed as follows:

```
Block i                    DO 214 J3 = 1,N
                           J = TSTOR (J3)
                           N5 = FMORT (CJ3,2)

Block i + 1                DO 213 J5 = 2,N1
                           IPOINT = FN(J5)
                           IF (IPOINT . GT. I) GO TO 212

Block i + 2        213     CONTINUE

Block i + 3                SAU(J3) = IPOINT
                           GO TO 214

Block i + 4        212     SAU(J3) = ENDIT(M,2)
                           XFACT = IPOINT

Block i + 5        214     CONTINUE

Block i + 6                STOP
                           END
```



Flow Graph for Figure 3A.

Figure 3A. Excerpt from Sample FORTRAN Program with Basic Blocks Denoted.

```
TOTAL NUMBER OF BLOCKS  =  7

    SIZE OF BLOCKS:          Size (i) = 3
                             Size (i+1) = 3
                             Size (i+2) = 1
                             Size (i+3) = 2
                             Size (i+4) = 2
                             Size (i+5) = 1
                             Size (i+6) = 2

          Number Branches to i = 2
          Number Branches to i+1 = 2
          Number Branches to i+2 = 1
          Number Branches to i+3 = 1
          Number Branches to i+4 = 1
          Number Branches to i+5 = 2
          Number Branches to i+6 = 1


     Block i  flows to  i+1
     Block i+1 flows to i+2, i+4
     Block i+2 flows to i+1, i+3
     Block i+3 flows to i+5
     Block i+4 flows to i+5
     Block i+5 flows to i, i+6
     Block i+6 flows to _____
```

Figure 3. Sample Flow Graph Data

a. $h \leftarrow I(h)$

b. If n is a node not yet in I(h), n is not the begin node, and all edges entering n leave nodes in I(h), then add n to I(h).

c. Repeat 2 until no more nodes can be added to I(h)

Note: Figure 4 presents an example of interval analysis applied to a flow graph.

2. Data Collected from Interval Analysis Applied to a Flow Graph

- total number of intervals formed from the flow graph initially
- number of blocks contained in each interval
- span of each interval

  % of instructions in the program contained in the interval
- number of branches inside each interval

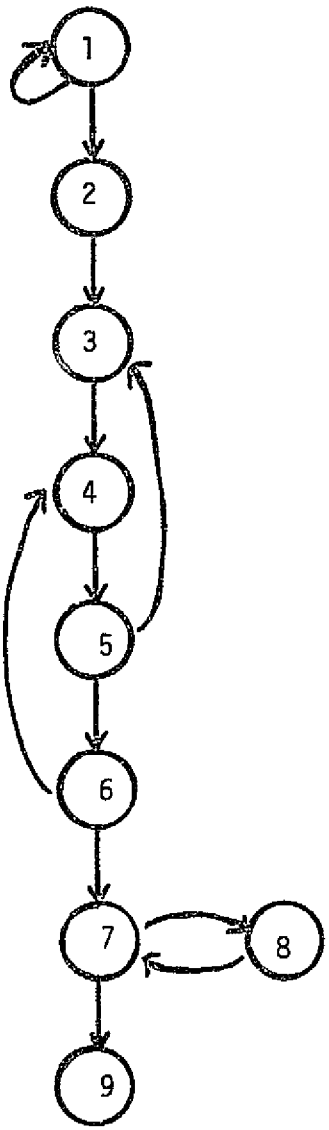
  non-sequential branches from 1 block to another within interval
- interval number to which each branch from an interval flows

  gives some feeling for the "sequentialness" of the <u>inter-connections</u> between intervals

  gives the hierarchy of the structural flow
- sequentialness of the flow

  an interval is formed by adding node branches from one block to another - therefore, if there is a large jump in the program flow it will be indicated by the <u>block numbers</u> making up the intervals. cf. Figure 5

Flow Graph, F:

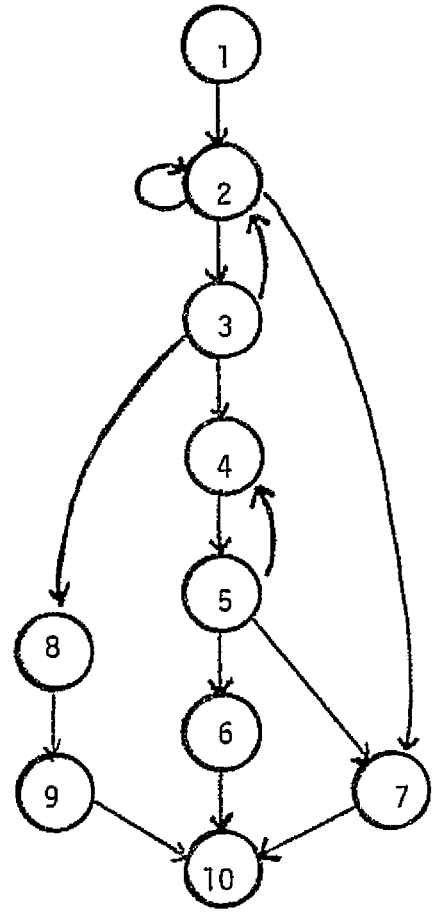


Intervals of F are as follows:

Interval 1 = {node 1, node 2}
Interval 2 = {node 3}
Interval 3 = {node 4, node 5, node 6}
Interval 4 = {node 7, node 8, node 9}

Figure 4. Interval Analysis Applied to a Flow Graph.

Flow Graph, F:



Intervals of F:

I_1 = node 1

*I_2 = node 2, node 3, node 8, node 9

I_3 = node 7

I_4 = node 10

I_5 = node 4, node 5, node 6

* NOTE the nodes composing I_2 - indicates non-sequentialness of the flow.

Figure 5. Interval Analysis Used in Analyzing Non-Sequential Flow

C. Variables Measureable via a Flow Graph

Aside from yielding information about itself, the flow graph can be used in measuring other variables as follows:

- span of each branch - number of statements jumped over by a backward or forward branch
- number of backward branches

vs:

- number of loops - a loop defined as any sequence of blocks that can be executed repeatedly

Note: It must be mentioned here that COBOL verbs - PERFORM...UNTIL, PERFORM...VARY, PERFORM...TIMES - do set up a looping sequence which will be evident in the flow graph. However, these loops are essentially independent - i.e. at the end of a PERFORMed paragraph, bar any go to's from the paragraph, control returns only to the statement after the PERFORM verb and cannot flow elsewhere as a flow graph might indicate. Figure 6 provides an example of this problem.

Two sets of data therefore are available for COBOL programs:

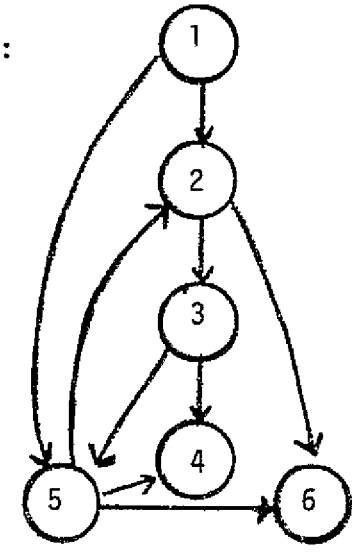
1. Block and interval analysis information with PERFORM branches included
2. Block and interval analysis plus looping information with PERFORM back branches and inner looping eliminated
   i.e. elimination of PERFORM branches except for actual branches to do the PERFORM

Suppose a COBOL program was set up as follows:

Procedure Division Using A_1, A_2 .

```
          PAR-1.
Block 1     PERFORM P1 THRU P1-EXIT UNTIL A EQUAL TO B.

Block 2   MOVE I to J.
          GO TO PAR-2, P2 DEPENDING ON COUNT.

          PAR-2.
Block 3     PERFORM P1 THRU P1-EXIT UNTIL J EQUAL TO K.

Block 4   ADD K TO 1

          GO BACK.

Block 5   P1.
            MOVE B TO C, D.
          P1-EXIT.
            EXIT.

Block 6   P2.
            GO BACK.
```

then a flow diagram for the above program would be:



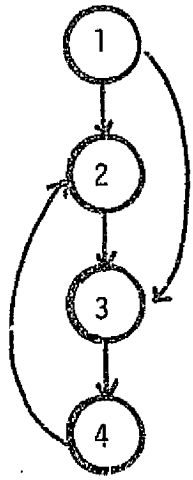
NOTE: The flow graph shows interconnection between Perform loops that really are not there, e.g. the graph shows that 2-3-5 is a loop.

This is impossible since the perform of Block 1 is completed before any perform at Block 3 and therefore the branch to 2 out of 5 does not really exist upon entering Block 3.

Figure 6. Example of Looping Problem with COBOL PERFORM verbs.

- number of closed path loops
- intersections and nestings of loops
- size of each loop
- number of exits out of a loop
- measurement of how well-structured a program is (i.e. how difficult it would be to restructure the program). As was described in Section 4.2, Peterson, et al [24] have developed an algorithm which determines the degree of difficulty involved in restructuring a program via its flow graph into a well-structured program consisting of only sequences, alternative clauses, iterative clauses and multi-level exits. The method essentially consists of counting the number of loops which have multiple-independent entry points - i.e., there exist paths, whose nodes are not elements of the loop, from the start node to more than one node in the loop.

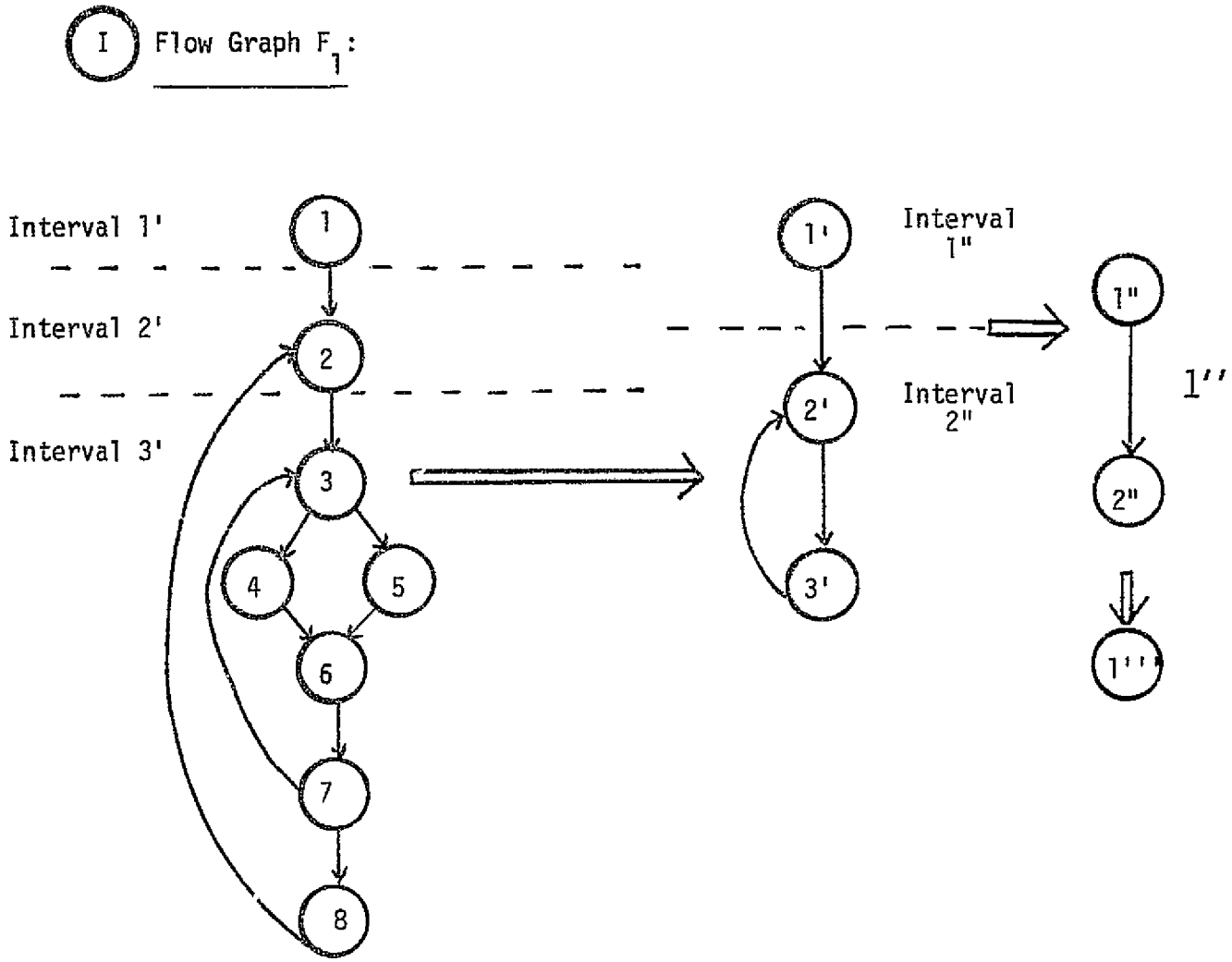
e.g.:



2-3-4 is a loop with 2 entry points, nodes 2 and 3.

Specifically, they define the complexity of a flowchart by a pair of integers (N,M) where N is the number of nodes in a largest multi-entry loop and M is the number of multi-entry loops with N nodes.

This measure is easily attainable via a continued application of the interval reduction algorithm described previously in this section. If a program has no multiple entry loops then continued applications of the interval reduction algorithm will reduce the graph to a single node. If the program does have loops with multiple entries then the interval reduction algorithm reduces its flow graph so that all the flow graph shows are the multiple-entried loops. Figure 7 provides examples of interval analysis applied in this manner.

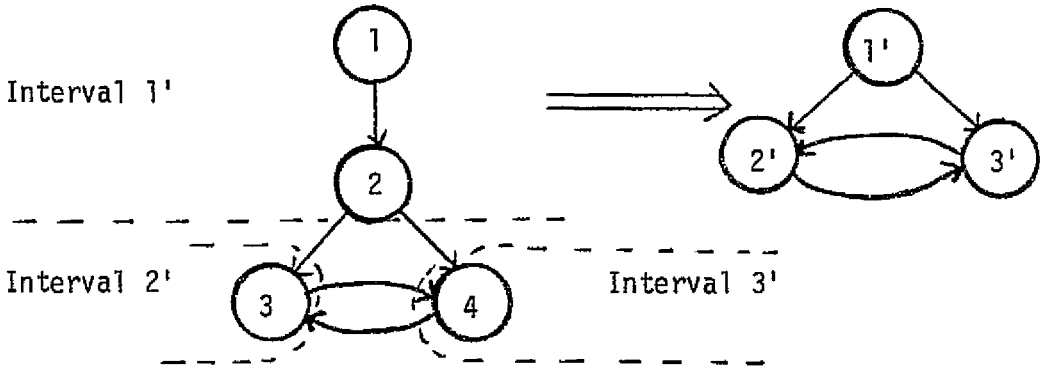


Flowgraph F_1 is reducible and therefore has no multiple-entry loops.

Figure 7.

(Continued next page)

(II) Flow Graph F_2



The intervals cannot be combined any further. The flow graph is reduced to a loop between nodes 2' and 3' with 2 entry points, nodes 2' and 3'.

Figure 7. Examples of Interval Analysis Reduction.

4.5 Preliminary Results/Future Analyses/Summary

At this point, the data collection system has been written and both FORTRAN and COBOL samples have been collected. The emphasis to date has been concerned with the actual collection of data - what program variables should be collected - how the data should be taken - and in general making certain that as much information as possible is procured from this static data collection. It is far better to have the capability of obtaining too much data than too little and it is most important to ensure that this data is in a form whereby it can be utilized easily.

This section discusses some preliminary results from data collected on several FORTRAN sample programs via the SPITBOL programs SNOINST and SNODATA. The data is incomplete as information is not available for all of the FORTRAN samples from the structure program SNOCONTR at this time. Yet, an examination and discussion of these initial results provides some interesting insights and yields results which lead into several important directions for future analysis of the data. These directions are in essence various aspects of what constitutes a measure and provide a frame of reference for a discussion of the term "measure."

4.5.1 Preliminary Results

The analysis was based on data points collected from 14 FORTRAN programs each with a complexity rating as given by their author/maintainer. This is a summary of some results and it is not intended to be all-encompassing of the data so far collected. The data points were selected somewhat at random as an initial starting point for determining what types of statistics and counts are relevant and which do not provide useful information and

therefore need another perspective. Each category will be defined below, the results indicated, and then a discussion of the program data collected under each heading follows. The circles around numbers indicate those counts or averages or percentages which are grossly different from other numbers under a particular heading - note that these circled numbers do not always agree with the rating of the program given it by its author/maintainer.

These circles under particular headings serve to emphasize categories from which more data types should be analyzed - e.g. number of function references discriminates between programs. This implies data on interface connections, types of subprograms, parameter nesting, etc. should be investigated. After a discussion of the data points individually, a summary of all the data will be shown via Figure 8, and trends seen in the data will be discussed more fully.

In a discussion of each of the individual headings the weaknesses in each category will be emphasized - i.e. where statistics appear to be faulty, where more data obviously needs to be analyzed, etc. Also, strengths in these categories and where these seem to be leading will be pointed out. Note that we are trying to get statistics that indicate the "complicatedness" of the program. This can be done mainly in two ways: via a norm to compare programs against or by getting data points which differentiate between programs.

1. Complexity Rating - the score given to the program by its author/maintainer in the questionnaire previously described.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Complexity Rating	1	2	3	5	5	5	6	6.5	7	7.5	8	9	9	9

The first category indicates the "complexity rating" given to the program by its author. No attempt has been made to normalize these. They are as they have been given - to serve as an initial separator between programs. It will be obvious, however, with close scrutiny of the data to follow that some of the program writers thought their programs to be far more complicated than the data indicates. Some tended to confuse the difficulty in solving the actual problem with the difficulty of writing the code for it. Or perhaps there are features to these programs, e.g. structure and control flow, which are not illuminated by the present collected data but which do merit the complexity rating given by the program's author.

2. Number of Subprograms attached to the Main Program

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Subprograms Attached	0	0	0	0	4	0	7	13	4	3	13	5	9	10

3. Percentage of the Program Spanned by Subprograms - how much of the total code is contained in subroutines.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
% Program Spanned by Subprograms	0	0	0	0	42%	0	50%	90%	39%	73%	56%	81%	62%	93%

The number of subroutines attached to the main program appears to differ greatly between the programs - and yet the amount of code performed within these subroutines is on the whole always greater than 50% of the total code, implying that the programmers attempted modularization of a fashion. Obviously, more samples are needed here to see if this phenomenon holds true in general. A better data point might be to look at how modular the code indeed is - are the total instructions mainly in one or two subprograms or are the subprograms relatively small or at least all of approximately the same size? The way in which the FORTRAN program data has been collected greatly facilitates this as data counts have been taken for each subprogram independently of its main program.

4. Number of Logical Operations / Number of IF Statements - What is the average complexity of the logical expressions.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Log Operns. / # IFs	1	1	1	1	1.1	1.2	1.6	1.4	1.2	1	1.1	1	1.1	1.1

This category, dealing with the complexity of logical expressions, appears to differentiate minimally between programs.

5. Number of Function References - the number of subroutines, subprograms, function statements, and FORTRAN functions referenced in the program.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# FN Refs.	7	20	14	5	(66)	17	40	108	28	(74)	(129)	29	(61)	48

Under this heading, there is a variation in the counts between programs but it should be noted that the largest numbers of function references are not necessarily found in what have been classed as the most complex programs. Since this does appear to be a discriminating characteristic, more data in this interface/function reference environment should be included in future analyses.

6. Total Number of Instructions - the upper part of the block indicates number of instructions including COMMENT statements; the lower part of the block is the total instructions without COMMENTS.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Instrucs.	93	200	229	387	378	233	791	758	686	398	989	538	950	958
# Instrucs. (No comments	82	166	167	366	324	185	591	688	321	320	758	440	700	755

Category 6 is strictly an indicator of size. It is interesting to note that the sizes are not in order of increasing complexity - e.g. the author of Program 7 with its 791 instructions gave it a rating of 6 whereas the author of Program 12 with its 538 instructions gave it a rating of 9. This at least indicates that some thought toward program characteristics as opposed to program size went into the samples the programmers chose for analysis. It should be pointed out here the difference COMMENT statements can make in some program sizes - e.g. Program 9 is cut by a factor of 2 in size when COMMENT statements are not counted.

7. Percentage of Total Instruction Count that are comments.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
% Comments	12%	17%	27%	5%	14%	21%	25%	9%	53%	20%	23%	18%	26%	19%

8. Percentage of Total Instruction Count that are unconditional statement types - the upper part of the block indicates the number with COMMENTS included in the instruction count.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
%	6%	1%	2%	4%	5%	1%	7%	4%	3%	5%	3%	3%	7%	5%
Uncondit.	7%	1%	2%	4%	5%	2%	10%	5%	6%	6%	3%	3%	9%	6%

9. Percentage of Total Instruction Counts that are conditional statement types - the upper part of the block indicates the number with COMMENTS included in the instruction count.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
%	16%	4%	4%	13%	11%	5%	21%	12%	5%	6%	8%	9%	18%	9%
Conditional	18%	5%	6%	14%	12%	6%	28%	13%	11%	11%	10%	11%	24%	11%

10. Percentage of Total Instruction Counts that are DO statements - the upper part of the block indicates the number with COMMENTS included in the instruction count.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
%DO	4%	13%	3%	9%	3%	5%	4%	6%	1%	4%	10%	8%	2%	11%
	5%	15%	4%	10%	4%	7%	5%	7%	2%	5%	12%	10%	3%	14%

11. Percentage of Total Instruction Counts that are Sequential statement types - the upper part of the block indicates the number with COMMENTS included as an instruction type.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
%	61%	66%	64%	68%	67%	68%	43%	69%	38%	66%	57%	62%	47%	56%
Sequential	70	79	88	72	79	86	(57)	75	(81)	83	74	76	(64)	69

Categories 7 through 11 are a minimal attempt at instruction type categorization. With few exceptions, the percentages under unconditional, conditional, do, and sequential fall within similar ranges for each of the programs. This suggests that instead of serving as a differentiating factor between programs, these categories could serve as a norm against which programs could be evaluated. Obviously, more samples need to be looked at before any conclusions can be drawn but this initial data indicates how this type of statistic could be used.

12. Average Subprogram Nesting Depth

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Avg. Subprogram Nesting Depth	1.	1.	1.	1.	1.6	1.	2.	2.6	1.9	1.3	1.4	2.1	1.9	2.6

13. Number of Subprogram Nestings - how many subprogram branches emanate from main program.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Subprogram Nestings	3	1	3	2	20	6	17	25	11	6	21	11	27	26

The average depth of subroutine nesting, category 12, does vary between programs but here average is not a good indicator of anything. The median depth or maximum/minimum depths would yield a wider spread between programs. Perhaps, though, this average nesting depth should be looked at in conjunction with category 13, the number of nestings. For example, the average nesting depth of 1.4 of program 11 compares favorably with that of program 12 with its nesting depth of 2 when consideration is given to the fact that program 11 has 21 such nestings whereas program 12 has only 11. So where program 12 has more depth in its function references, program 11 has more breadth, i.e. more nestings to be concerned with.

14. Number of Breaks in Sequential Flow of Program - how many times is the program broken by a jump upward or downward or a do loop.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Breaks	22	34	16	(95)	58	18	(193)	(228)	45	52	(155)	94	(234)	(209)

The number of breaks in sequential program flow obviously differentiates between programs - witness how program 9 and program 10 appear to be totally out of place in this category with programs having even lower complexity ratings than they. This general category leaves the door wide open for exploration into all facets of this program flow problem and emphasizes the need for inclusion of the control flow and structure data from programs SNOCONTR and SNOCONTR2.

15. Number of Instructions / Number of Sequential Breaks - an indicator of how many program statements on the average can be spanned without hitting a "jump" of some sort. (The upper part of the block indicates the average for total instructions including comments - the lower portion of the block is for instructions excluding comments.)

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Inst.	4.2	5.9	14.3	4.1	6.5	12.9	4.1	3.3	15.2	7.7	6.5	5.7	4.1	4.9
# Seq. Breaks	3.2	4.9	10.4	3.9	5.7	10.3	3.1	3.1	7.2	6.2	5.0	4.7	3.0	3.8

Category 15 is an attempt to negate size and question within any program, on the average, how many statements can be spanned before hitting a break in the flow. Notice that while program 1 is a relatively small program of 93 instructions with a not-too-high total number of breaks in sequential flow (when this category looked at independently), it has a very low ratio of program jumps to number of statements, indicated by category 15, making it comparable in this regard to programs with complexity ratings of 10!!

16. Percentage of Breaks in Sequential Flow Downward -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
% Down	82%	24%	37.5%	54%	64%	28%	72%	65%	[illegible]	48%	37%	50%	71%	45%

17. Percentage of Breaks in Sequential Flow Upward -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
% UP	0%	3%	25%	9%	16%	5%	12%	14%	36%	23%	2%	3%	21%	4%

18. Percentage of Breaks in Sequential Flow Due to DO loops -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
% Loop	18%	73%	37.5%	37%	20%	67%	16%	21%	15%	29%	61%	47%	8%	51%

Categories 16, 17, 18 delineate these flow breaks of category 14 into upward, downward, and loop. This data, in its present form, does not really indicate too much of anything. Even within the set of programs with a large number of breaks in their flow (e.g. Program 7 and Program 11) the percentages of each under the three headings vary greatly. A lot of upward flow, according to good programming techniques is considered a bad omen - but perhaps this is best measured in terms of numbers of loops and backward branches and the span of each and not just by a count of how many upward jumps exist in the program.

19. Average size of DO loops -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Avg. Do Size	17.0	14.3	41.8	19.3	19.3	9.2	4.9	11.7	11.6	6.6	13.6	11.1	7.9	4.0

20. Number of DO Loop Nestings - the number of outer DO's in the program.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# DO LOOP Nestings	2	12	4	(24)	9	11	(30)	(31)	6	11	(54)	(25)	19	(79)

21. Average Depth of DO LOOP Nesting -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Avg. Nesting Depth	1.	1.3	.5	1.8	.7	.4	.13	.8	.7	.6	1.7	.9	.05	.5

The average DO size, category 19, is again an instance where average is not a good statistic. This average would tend to de-emphasize programs where essentially the whole program was a DO loop but where there were many smaller DO loops within the program thus decreasing the average size. A maximum/minimum or median statistic or even the numbers of DO loops falling into various size categories might be a better indicator of how complex the DO loop structure of the program really is.

Whereas the number of DO loop nestings seems to be a good differentiator, the average nesting depth for a DO loop appears to be a faulty statistic in some regards. It does not yield a good picture of how complicated the DO expressions of a program are. Here a statistic such as the maximum nesting depth and breadth and the size of the outer DO for this nesting would yield a more informative data point. Other statistics on nesting will at least indicate whether what appears to be true right now, i.e. nesting depth of DO loops (no matter how it is measured) is minimal for most programs - is indeed correct. Also, we

C.3

want statistics that indicate "complicatedness" within a program and perhaps the above mentioned is a better way to get at this.

22. Average Number of References to a Variable -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Avg. # Refs.	5.4	15.6	4.4	11.0	5.1	5.3	3.4	4.0	4.9	6.7	5.7	3.6	3.7	6.4

23. Average Span of Each Variable - on the average, what percentage of the statements in the program are spanned by the variable.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Avg. Span	38.3	66.8	23.9	35.1	30.5	37.7	25.6	32.0	34.4	35.6	33.0	32.0	31.8	42.2

24. Total Number of Variables Referenced in the Program -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Vars.	28	30	107	96	173	100	402	435	191	135	314	368	532	297

25. Percentage of the Variables which are Inputs -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
% Input	14%	20%	10%	24%	6%	8%	2%	6%	23%	5%	4%	7%	7%	1%

Variables within a program are considered under categories 22 through 25. Of these four, only the number of variables differs greatly between the programs. It is interesting to note that the average number of statements spanned by a variable from its first reference in a program to its last reference falls within the same approximate range for most of these 14 programs. It is also of interest to point out that this span of a variable's reference is quite large - "remembering" a variable and its value for 30 statements or more certainly retards retention span.

26. Average Segment Size - average number of statements between labels.

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Avg. Segment Size	3.6	4.1	6.2	3.4	6.4	8.0	3.9	3.7	4.9	5.3	5.3	6.2	3.4	3.8

27. Number of Labels in the Program -

Program Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
# Labels	22	40	26	107	50	22	145	177	61	37	131	66	200	193

Average segment size and number of labels is an attempt to look at pieces of a program - what is the so-called eye-span for the programmer as the program is scanned. This average and the count of labels is in no way indicative of the program's flow. Since a labeled statement tends to be a "referred to" statement within a program, the fact that the data collected

	1 COMPLEXITY RATING	2 SUBR ATTACHED	3 % PROGRAM SPAN BY SUBRS	4 # LOGOPERS / # IFS	5 # FN REFS	6 # INSTRUCS: # W/OUT COMMENTS	7 % COMMENTS	8 % UNCON	9 % CONDIT	10 % DO	11 % SEQUENTIAL	12 AVG. SUBR NESTING DEPTH	13 # SUBR NESTINGS	14 # SEQ BREAKS	15 # INS / # SEQ BREAKS	16 % DOWN	17 % UP	18 % LOOP	19 AVG DO SIZE	20 # NESTINGS	21 AVG NESTING DEPTH	22 AVG # UAR REFS	23 AVG UAR SPAN	24 # UARS	25 % INPUT	26 AVG SEGMENT SIZE	27 # LABELS
PROGRAM 1	1	0	0	1	7	93/82	12%	6%/7%	6%/18%	4%/5%	61%/70%	1.	3	22	4.2/3.2	82%	0%	18%	17.0	2	1.	5.4	38.3	28	14%	3.6	22
PROGRAM 2	2	0	0	1	20	200/166	17%	1%/1%	4%/5%	13%/15%	66%/79%	1.	1	34	5.9/4.9	24%	3%	73%	14.3	12	1.3	15.6	66.8	30	20%	4.1	40
PROGRAM 3	3	0	0	1	14	229/167	27%	2%/2%	4%/6%	3%/4%	64%/88%	1.	3	16	14.3/10.4	37.5%	25%	37.5%	41.8	4	.5	4.4	23.9	107	10%	6.2	26
PROGRAM 4	5	0	0	1	5	387/366	5%	4%/4%	13%/14%	9%/10%	68%/72%	1.	2	95	4.1/3.9	54%	9%	37%	19.3	24	1.8	11.0	35.1	96	24%	3.4	107
PROGRAM 5	5	4	42%	1.1	66	378/324	14%	5%/5%	11%/12%	3%/4%	67%/79%	1.6	20	58	6.5/5.7	64%	16%	20%	19.3	9	.7	5.1	30.5	173	6%	6.4	50
PROGRAM 6	5	0	0	1.2	17	233/185	21%	1%/2%	5%/6%	5%/7%	68%/86%	1.	6	18	12.9/10.3	28%	5%	67%	9.2	11	.4	5.3	37.7	100	8%	8.0	22
PROGRAM 7	6	7	50%	1.6	40	791/591	25%	7%/10%	21%/28%	4%/5%	43%/57%	2.	17	143	4.1/3.1	72%	12%	16%	4.9	30	.13	3.4	25.6	402	2%	3.9	145
PROGRAM 8	6.5	13	90%	1.4	108	758/688	9%	4%/5%	12%/15%	6%/7%	69%/75%	2.6	25	228	3.3/3.1	65%	14%	21%	11.7	31	.8	4.0	32.0	435	6%	3.7	177
PROGRAM 9	7	4	39%	1.2	28	68%/321	53%	3%/6%	5%/11%	1%/2%	38%/81	1.9	11	45	15.2/7.2	49%	36%	15%	11.6	6	.7	4.9	34.4	191	23%	4.9	61
PROGRAM 10	7.5	3	73%	1	74	398/320	20%	5%/6%	6%/7%	4%/5%	66%/83	1.8	6	52	7.7/6.2	48%	23%	29%	6.6	11	.6	6.7	35.6	135	5%	5.3	57
PROGRAM 11	8	13	56%	1.1	129	989/758	23%	3%/3%	8%/10%	10%/12%	57%/74	1.4	21	155	6.5/5.0	37%	27%	61%	13.6	54	1.7	5.7	33.0	314	4%	5.3	131
PROGRAM 12	9	5	81%	1	29	538/440	18%	3%/3%	9%/11%	8%/10%	62%/76	2.1	11	94	5.7/4.7	50%	3%	47%	11.1	25	9	3.6	32.0	368	7%	6.2	66
PROGRAM 13	9	9	62%	1.1	61	950/700	26%	7%/9%	18%/24%	2%/3%	47%/64	1.9	27	234	4.1/3.0	71%	21%	8%	7.9	19	.05	3.7	31.8	532	7%	3.4	200
PROGRAM 14	9	10	93%	1.1	48	958/775	14%	5%/6%	9%/11%	11%/14%	56%/69	2.6	26	209	4.9/3.8	45%	4%	51%	4.0	79	.5	6.4	42.2	297	1%	3.8	193

Figure 8.

for average segment size differentiates between the programs, reinforces the need for some sort of measurement of program pieces. The control flow blocks and interval analysis described previously is a better statistic for this data - these data points at least follow the true control flow of the program yielding program segments more indicative of what the eye-span in following the program has to be.

4.5.2 Future Analysis

Figure 8 contains the data just enumerated in summary form. With even a cursory look at this figure a couple of points are obvious: first, the complexity rating scheme of the questionnaire, while providing an initial means of separating the programs is too dependent on the individual programmer and therefore is not uniform, and second, there is no one variable that by itself appears to differentiate between categories of programs. But it is clear that the analysis techniques employed in Figure 8 for getting counts, averages, percentages, etc. can yield valuable information and can serve as a beginning in predicting trends in data. Some of these data categories indicate they could be a standard norm against which programs can be evaluated, e.g. complexity of logical expressions (category 4) while others are obvious differentiators between programs, e.g. category 14, the number of breaks in sequential flow. This type of analysis also points up where categories appear to be inadequate and yield no conclusive information and where a different perspective on a variable is needed - e.g. average nesting depth of a DO loop.

There are a large number of ways to manipulate the data collected but there are also limits as to how many samples can be collected and analyzed.

A 50-dimensional feature vector cannot be put through an analysis based on only 50 sample programs. Therefore, the types of analyses done in Figure 8 must be used to refine the data collected into a set of relevant statistics which can in turn be used in more complicated schemes. But even at this, there are just too many data points in the program's feature vector. Therefore, some type of selection technique must be chosen to decide exactly what variables should constitute a reduced feature vector. This implies in essence that we are selecting a specific measure with which to evaluate complexity.

Instead of choosing specific variables as definitive complexity factors, a scheme was set up to investigate a wide variety of variables as possible complexity factors. Similarly, a more objective method for selection of a complexity metric would be to refrain from choosing a specific technique as a measurement tool and instead explore multiple techniques that could serve as a means for getting a complexity measure. The data can then be looked at through various frames of references. That is, different sets of variables can be analyzed using varying techniques. By not choosing one specific direction for data analysis, the program can be viewed as a learning problem, as a series of retention barriers, or as simply a group of structural factors or data factors or combinations of each. Also, techniques that allow a random choice of variables to be input and then provide a classification of these arbitrary variables into groups can be utilized.

Statistical techniques such as multivariate analysis, factor analysis, and cluster analysis are eminently suitable for taking different pieces of information and combining these pieces into a single "best" predictor of complexity. It is possible to approach a measure in several ways using these methods - e.g. as an equation for predicting an unknown complexity score of a program from the known set of complexity factors or as a measure of the significance of differences among groups of feature vectors.

There are methods then available for exploring various sets of complexity characteristics. These can be utilized blindly, i.e. with no rhyme or reason as to how the characteristics are put into the feature vector. The interpretation of the results might be difficult in these cases and would necessitate some type of subjective judgment as to what the measure actually was predicting. Therefore, a frame of reference must be given to data used in these techniques. As was seen from the preliminary results, there are quite a few natural groupings in the variables which could be used. Some ideas for different frames of reference for the variables are as follows.

Weinberg [35] states that "psychologists have long observed that the capacity of the brain to deal with several items at one time is limited." He feels that about 30 lines of code, divided into 3 or 4 groups, is about all that can be mastered. Miller [36] points out that there exists a finite span of immediate memory and for a lot of different kinds of test materials this span is about seven items in length. Further, to increase the amount of information that can be dealt with it is necessary to group or organize the input sequence into units or chunks. One can then ask what are the program factors that affect retention - what are programming tools and characteristics

that organize a program into "good chunks"? Therefore, collected program characteristics that affect retention span or affect our "learning" the program can be used as input into a multivariate analysis scheme. Factors such as numbers and sizes of modules, depth of subroutine and parameter nesting, numbers of blocks in a flow graph, numbers and sizes of unique intervals the program can be broken into, etc. are just a few of the possible input variables.

Another frame of reference would be to use Gunning's [37] style and attempt to take the fog out of a program - to eliminate "noisy" unnecessary complexity. Instead of coming up with factors that differentiate, a set of standard norms would be developed - a set of principles which would serve as a yardstick against which other programs would be measured. Variables used in these techniques would be those that initial analysis showed could be normalized to give a number considered "non-harmful" or a reasonable standard to set for the program's comprehensibility.

The list of possibilities for structuring the data is endless and each presents a new orientation which is far more worthwhile than a single complexity measure, $F_c (X_1, X_2, \ldots X_n)$, where each X_i represents some arbitrary program characteristic. The goal is a single complexity measure but one which presents complexity as a multi-faceted problem. Structural complexity versus interface complexity versus retention span complexity, etc. will all be aspects for which measures are at hand. These are perspectives on the problem which can in turn be measured for their individual worth in evaluating a program. In this way, then, we will have attempted to investigate the

problem from all sides and will therefore have far better tools for an evaluation and judgment for any program at hand.

4.5.3 Summary

The previous sections have provided justification for an investigation of program complexity and have described a method which attempts such an investigation. The implementation phase of this data collection method is underway and emphasis is presently on actual data analysis. It is now possible to discuss a program objectively in terms of its collectible characteristics.

If results to date prove correct, even straightforward analyses of the data can be quite useful in assessing a program via its complexity. With data available from COBOL programs as well as FORTRAN programs, inter-language comparisons can be made. Conclusions from analyses are therefore applicable in both the scientific and data processing spheres of influence.

Each phase of the investigation provides a reuseable tool for program analysis not only in terms of complexity but also in other areas of on-going research. The previous section described the thrust for future research in the complexity area and the solution of the immediate goal of a complexity measure. However, since the data collected provides objective information on a program, then a profile of each program in terms of its measureable characteristics is available. This provides a firm basis for understanding the source of the reliability problem, the program itself. For it is impossible to discuss software without objective data to reinforce opinions and this data collection method makes uch data readily available.

4.6 Bibliography

1. Clapp, J.A., Sullivan, J.E. Automated Monitoring of Software Quality. Proceedings of National Computer Conference, 1974, pp. 337-341.

2. Pietrasanta, A.M. Resource Analysis of Computer Program System Development. On the Management of Computer Programming, (Ed.) Weinwurm, Auerback, Philadelphia, 1970.

3. Weinwurm, G.F. On the Economic Analysis of Computer Programming. On the Management of Computer Programming, (Ed.) Weinwurm, Auerbach, Philadelphia, 1970.

4. Pietrasanta, A.M. Functional Estimating of Computer Program System Development. On the Management of Computer Programming, (Ed.) Weinwurm, Auerbach, Philadelphia, 1970.

5. IBM, Custom Contract Services. Estimating Systems Costs, reprinted November 20, 1971.

6. Nelson, E.A. Management Handbook for Estimation of Computer Programming Costs, S.D.C., AD648 760, March 20, 1967.

7. Farr, L., LaBolle, V., Willmorth, N. Planning Guide for Computer Program Development, S.D.C., May 10, 1965.

8. Wolverton, R.W. The Cost of Developing Large Scale Software, 1972 IEEE International Convention Digest, pp. 178-179.

9. Wolverton, R.W. The Cost of Developing Large Scale Software, IEEE Transactions on Computers, Vol. C-23, No. 6, June 1974, pp. 615-636.

10. Weissman, L. Psychological Complexity of Computer Programs: An Experimental Methodology. SIGPLAN Notices, June 1974.

11. Anderson, P.G. and Crandon, L.H. Computer Program Reliability, RCA Engineer 19, 5 (Feb. - Mar. 1974).

12. Dickson, J.C., Hesse, J.L., Kientz, A.C. and Shooman, M.L. Quantitative Analysis of Software Reliability. Proceedings of 1972 Annual Reliability and Maintainability Symposium. IEEE Cat. No. 72CHO 577-7R, pp. 148-157.

13. Rubey, R.J., Wick, R.C. and Beathley, L. Comparative Evaluation of PL/1. U.S. Air Force Report AD-669 096, July 1968.

14. Goldberg, J. (Ed.) Proceedings of a Symposium on the High Cost of Software, September 17-19, 1973.

15. Mills, H.D. Mathematical Foundations for Structured Programming, IBM, February, 1972.

16. Dijkstra, E.W., Dahl, O.S., Hoare, C.A.R. Structured Programming, Academic Press, 1972.

17. IBM, Management Overview, 1973.

18. Donaldson, J.R. Structured Programming, Datamation, December, 1973.

19. Ramamoorthy, C.V., Cheung, R.C., and Kim, K.H. Reliability and Integrity of Large Computer Programs, U.S. Government Report AD-779 339, March 1974.

20 Rhodes, J. Tackle Software with Modular Programming. Computer Decisions, October 1973, pp. 21-25.

21. Constantine, L.L., Stevens, W.P., Myers, G.J. Structured Design, IBM Systems Journal, No. 2, 1974.

22. Goodman, L.I. Complexity Measures for Programming Languages. U.S. Government Report AD-729 011, October, 1971.

23. Sullivan, J.E. Measuring the Complexity of Computer Software. Mitre Corporation, MTR-2648, Vol. V, June 1973.

24. Peterson, W.W., Kasami, T., Tokura, N. On the Capabilities of While, Repeat, and Exit Statements, CACM, Volume 16, November 8, 1973.

25. Aho, A.V., Ullman, J.D. The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling, Prentice-Hall, Inc., 1973.

26. de Balbine, G. The Structuring Engine: A Transitional Tool, Computer, Vol. 8, No. 6, 1975.

27. Knuth, D. E. An Empirical Study of Fortran Programs, U.S. Government Report AD-715-513, February, 1971.

28. Mills, H.D. The Complexity of Programs. Program Test Methods, (Ed.) W.C. Hetzel, Prentice-Hall, Inc., 1973, pp. 225-239.

29. TRW Systems Group, The Quantitative Measurement of Software Safety and Reliability, Redondo Beach, California, August, 1973.

30. Cooley, W.W., Lohmes, P.R. Multivariate Procedures for the Behavioral Sciences, John Wiley & Sons, Inc., 1962.

31. Endres, A. An Analysis of Errors and Their Causes in System Programs, Proceedings of the International Conference on Reliable Software, April, 1975.

32. Shooman, M.L., Bolsky, M.I. Types, Distribution and Test and Correction Times for Programming Errors, Proceedings of the International Conference on Reliable Software, April, 1975.

33. Gileadi, A.N., Ledgard, H.F. On a Proposed Measure of Program Structure, ACM SIGPLAN Notices, Vol. 9, No. 5, 1974.

34. Meissner, L.P. A Method to Expose the Hidden Structure of Fortram Programs, ACM SIGPLAN Notices, Vol. 10, No. 6, 1975.

35. Weinberg, G.M. PL/1 Programming: A Manual of Style, McGraw-Hill Co., 1970.

36. Miller, G.A. The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information, The Psychological Review, Vol. 63, No.2, 1956.

37. Gunning, R. How to Take the Fog Out of Writing, The Dartnell Corp., 1964.

4.7 References

Elspas, B., Green, M.W. and Levitt, J.N. Software Reliability. Computer, January/February, 1971, pp. 21-27.

Mulock, R.B. Software Reliability Engineering. Proceedings of 1972 Annual R&M (Reliability and Maintenance) Conference. IEEE Cat. No. 72 CH0577-7R, pp. 586-593.

de S. Coutinho, J. Quality Assurance of Automated Data Processing Systems, Parts I and II. Journal of Quality Technology, Vol. 4, #2, April, 1972 pp. 93-101. Vol. 4, #3, July, 1972, pp. 145-155.

Ellingson, O.E. Computer Program and Change Control. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73H0741-9CSR, pp. 107-116.

Keezer, E.I. Practical Experiences in Establishing Software Quality Assurance. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73H0741-9CSR, pp. 132-135.

Girard, E. and Rault, J.C. A Programming Technique for Software Reliability. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73H0 741-9CSR, pp. 44-50.

McGeachie, J.S. Reliability of the Dartmouth Time Sharing System. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73H0741-9CSR, pp. -17-123.

Bloom, S., McPheters, J.J. and Tsiang, S.H. Software Quality Control. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73H0 741-9CSR, pp. 107-116.

Elmendorf, W.R. Controlling the Functional Testing of an Operating System. IEEE Transactions on Systems Science and Cybernetics. SSC-5. October, 1969, pp. 284-290.

Taylor, T., LeVarn, M. and O'Connell, J. Performance Monitoring and Fault Diagnostics of Command and Control Systems. RE-19-5-24, January, 1974.

Chandler, A.R. Software Verification and Validation for Command and Control Systems. RE-19-5-23, November, 1973.

Teichroew, D. and Sayani, H. Automation of System Building. Datamation, August 15, 1971, pp. 25-30.

Ogdin, J.L. Designing Reliable Software. Datamation, July, 1972, pp. 73-76.

Ogdin, J.L. Improving Software Reliability. Datamation, January, 1973.

Trauboth, H. Guidelines for Documentation of Scientific Software Systems. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73H0741-9CSR, pp. 124-131.

Sevcik, K.C., Atwood, J.W., Grushcow, M.S., Holt, R.C. Horning, J.J., Tsichritzis, D. Project SUE as a Learning Experience. Proceedings AFIPS 1972 FJCC. Vol. 41, pp. 331-338.

Rose, C.W. LOGOS and the Software Engineer. Proceedings AFIPS 1972 FJCC. Vol. 41, pp. 311-323.

Buzen, J.P., Chen, P.P. and Goldberg, R.P. Virtual Machine Techniques for Improving System Reliability. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0741-9CSR, pp. 12-17.

de S. Coutinho, J. Software Reliability Growth. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0741-9CST, pp. 58-64.

Littlewood, B. and Verrall, J.L. A Bayesian Reliability Growth Model for Computer Software. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CHO 741-9CSR, pp. 70-77.

Jelinski, Z. and Moranda, P.B. Applications of a Probability - Based model to a code reading experiment. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CHO 741-9CSR, pp. 78-81.

Beizer, B. Analytical Techniques for the Statistical Evaluation of Program Running Time. Proceedings AFIPS 1970, FJCC, pp. 519-524.

Flynn, R.J. On the Smallest Number of Program Modules Needed to Duplicate Dynamic Independent Interactions. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH074-9CSR, pp. 65-69.

Ingalls, D.H.H. Jr. Execution Time Analyzer. OFFICIAL GAZETTE of U.S. Patent Office, October 31, 1972, p. 1714.

Stucki, L.G. A Prototype Automatic Program Testing Tool. Proceedings AFIPS 1972 FJCC, Vol. 41, pp. 829-836.

Estrin, G., Hopkins, D., Coggan, B. and Crocker, S.D. Proceedings AFIPS 1967, SJCC, pp. 645-656.

Fragola, J.R. and Spahn, J.R. The Software Error Effects Analysis: A Qualitative Design Tool. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0741-9CSR, pp. 90-93.

Lucena, C.J. A Methodology for Producing Reliable Software Systems. Proceedings of the ACM, 1973, p. 432.

Paige, M.R. On Testing Programs. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0-741-9CSR, pp. 23-37.

Buckley, F.J. Software Testing - A Report from the Field. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0741-9CSR, pp. 102-106.

Fisher, R.A. Automated Testing of Software Systems. Proceedings of the Third Annual Hawaii International Conference on System Sciences. Part 2, 1970, pp. 886-889.

Hanford, K.V. Automatic Generation of Test Cases. IBM Systems Journal 9 9(1970).

Burkhardt, W.H. Generating Test Programs from Syntax. Computing, Vol. 2, No. 1, 1967, pp. 53-73.

Kulp, J.C. Impact of Hardware/Software Tradeoffs in a Command and Control System. RE-19-5-22, December, 1973.

Rowe, L.A., Hopwood, M.D. and Farber, D. J. Software Methods for Achieving Fail-Soft Behavior in the Distributed Computing System. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73 CH0741-9CSR, pp. 58-75.

Grishman, R. Criteria for a Debugging Language. Debugging Techniques in Large Systems. Rustin, R. (ed.) Prentice-Hall, 1971, pp. 58-75.

Itoh, D. and Izutani, T. Fadebug-1, A New Tool for Program Debugging. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0741-9CSR, pp. 38-43.

Balzer, R.M. EXDAMS- Extendable Debugging and Monitoring System. Proceedings AFIPS 1969, SJCC, pp. 567-580.

Grishman, R. The Debugging System AIDS. Proceedings AFIPS 1970, SJCC, pp. 59-64.

Dijkstra, E.W. The Humble Programmer. Communications of the ACM 15. 10(October 1972), pp. 859-866.

Mills, H.D. On the Development of Large Reliable Programs. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0741-9CSR, pp. 155-159.

Weissman, L. and Stacey, G.M. An Interface System for Improving Reliability of Software Systems. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0741-9CSR, pp. 136-142.

Parnas, D.C. Some Conclusions from an Experiment in Software Engineering Techniques. Proceedings AFIPS 1972 FJCC, Vol. 41, pp. 325-329.

Cohen, A. Modular Programs: Defining the Module. Datamation, January, 1972. pp. 34-37.

McIntosh, C.S., Evans, R.H. and Lastra, R. Survey and Analysis of Major Computing Operating Systems. U.S. Government Report, AD-7014138, June, 1970.

Mittwede, W.C. Computer Operating Systems Capabilities: A Source Selection and Analysis Aid. U.S. Government Report, AD-718973, April, 1970.

McIntosh, C.S., Choak, K.P. and Mittwede, W.C. Analysis of Major Computer Operating Systems. U.S. Government Report, AD-715919, February, 1970.

Ziegler, E.W. An Introduction to the UMTA Specification Language. SIGPLAN NOTICES 9, 4(April, 1974), pp. 127-132.

Parnas, D.L. More on Simulation, Languages and Design Methodology for Computer Systems, Proceedings AFIPS 1969 SJCC, pp. 739-743.

London, R.L. Software Reliability Through Proving Programs Correct. Digest of 1971 International Symposium on Fault-Tolerant Computing. IEEE Cat. No. 71C-6-C, pp. 125-129.

King, J.C. Proving Programs to be Correct. Digest of 1971 International Symposium on Fault-Tolerant Computing. IEEE Cat. No. 71C-6-C, pp. 130-133.

Lyons, T. and Bruno, J. An Interactive System for Program Verification. Algorithm Specification. Rustin, R. (ed.) Prentice-Hall, New York, 1972, pp. 117-141.

Weinberg, G.M. and Gressett, G.L. An Experiment in Automatic Vertification of Programs. Communications of the ACM 6. 10(October 1963), pp. 610-613.

Elspas, B., Levitt, K.N. and Waksman, A. A Comparison of Formal Program Validation Techniques. Digest of 1971 International Symposium on Fault-Tolerant Computing IEEE Cat. No. 71C-6-C, pp. 140-145.

Kay, R.H. The Management and Organization of Large Scale Software Development Projects. Proceedings AFIPS 1969 SJCC. pp. 425-432.

Schneidewind, N.F. A Methodology for Software Reliability Prediction and Quality Control. Naval Postgraduate School Report No. NPS55SS72032B, March, 1972.

Anderson, P.G. and Crandon, L.H. Computer Program Reliability. RE-19-5-21, November, 1973.

Boies, S.J. User Behavior on an Interactive Computer System. U.S. Government Report AD-745-836, March 1973.

Clapp, L. Interactive Programming Systems and Languages. U.S. Government Report AD-728-224, October, 1971.

Miller, L.A. Programming by Non-Programmers. U.S. Government Report AD-760043, July 1973.

Smith, M.H.A. Toward Better Computer Programming. U.S. Government Report AD-727-848, December 1971.

Grignetti, M.C. and Miller, D.C. Information Processing Models and Computer Aids for Human Performance, Task 2. Models of Human-Computer Interaction. U.S. Government Report AD-732232, December 1971.

Zinn, K.L. Comparative Study of Languages for Programming Interactive Use of Computers in Instruction. U.S. Government Report AD-692506, October, 1969.

Mathur, F.P. A Brief Description and Comparison of Programming Languages Fortran, Algol, Cobol, PL/1, and Lisp 1.5 from a Critical Standpoint. U.S. Government Report N72-33190, February 1973.

James, T.A., Hall, B.C. and Newbold, P.M. Advanced Software Techniques for Data Management Systems. Vol. 3, Programming Language Characteristics and Comparisons Reference. Nasa Report NASA-CR-115515, July 1972.

DesRoches, J.C. A General Basis for Comparative Evaluation of AED, COBOL, JOVIAL, and PL/1. U.S. Government Report AD-785205, May 1973.

Rubey, R.J., Wick, R.C. and Beathley, L. Comparative Evaluation of PL/1. U.S. Air Force Report AD-669 096, July 1968.

Sakakibara, K., Hatano, A. and Yoda, K. Algol Programming, U.S. Government Report N69-15155, February 1968.

Callender, E.D. and Rhodus, N.W. Pl/1 and a Data Base. U.S. Government Report AD-682305, April 1969.

Berkowitz, R.L. A Comparison of Some Fortran Languages. U.S. Government Report AD-716 738, March, 1971.

Wirth, N. On Certain Basic Concepts of Programming Languages. U.S. Government Report PB-176 766, February, 1968.

Perstein, M.H. Some Techniques for Describing Programming Languages. U.S. Government Report AD-666 370, May, 1968.

Tierran, J.C. Programming Languages for Digital Weapon Systems: Evaluation. U.S. Government Report AD-669 443, July, 1968.

Teichroew, D. and Merten, A. The Impact of Problem Statement Languages on Evaluating and Improving Software Performance. Proceedings AFIPS 1972 FJCC, Vol. 41, pp. 849-857.

Kernighan, B.W. and Plauger, P.J. Programming Style for Programmers and Language Designers. Record of 1973 IEEE Symposium on Computer Software Reliability. IEEE Cat. No. 73CH0741-9CSR, pp. 148-154.

Schwartz, J.T. An Overview of Bugs. Debugging Techniques in Large Systems. Prentice-Hall, 1971.

Knuth, D.E. and R.W. Floyd, "Notes on avoiding go to statements." Information Processing Letters 1, North-Holland, Amsterdam, 1971

Dijkstra, "The Humble Programmer." Comm. ACM 15:10, 1972.

Hoare,C.A.R. "A Note on the for Statement." Bit 12:3, 1972.

Hopkins, M.C. "A case for the go to." SIGPLAN Notices 7:11, 1972.

Leavenworth, B.M. "Programming with(out) the go to." Proceedings ACM 1972; SIGPLAN Notices 7:11, 1972.

Hull, T.E. "Would you believe structured Fortran?" SIGNUM Newsletter 8:4, 1973.

Cooper, D.C. "Some transformations and standard forms of graphs, with applications to computer programs." Machine Intelligence 2, American Elsevier, N.Y., 1968.

Bochmann, G.V. "Multiple exits from a loop without the go to." Comm. ACM 16:7, 1973.

Schmitt, S.A. Measuring Uncertainty: An Elementary Introduction to Bayesian Statistics, Addison-Wesley Co., Mass.

Meisel, W.S. Computer-Oriented Approaches to Pattern Recognition, Academic Press, 1972.

Hilgard, E.R., Bower, G.H. Theories of Learning, Meredith Publishing Co., 1966.

Kernighan, B.W., Plauger, P.J. The Elements of Programming Style, McGraw-Hill, Co., 1974

Plum, T.W-S. Mathematical Overkill and the Structure Theorem, ACM SIGPLAN Notices, Vol. 10, No. 2, 1975.

Fleischer, R.J. Effects of Management Philosophy on Software Production, Mitre Corporation Technical Report, No. 2648, Vol. II.

Meyer, A.R. Ritchie, D.M. The Complexity of Loop Programs, Proc. of ACM National Meeting, 1967.

Goodenough, J.B. et al. The Effect of Software Structure on Software Reliability, Modifiability, Reusability and Efficiency: A Preliminary Analysis, Government Report AD-780-841, December, 1973.

Weinberg, G.M. The Psychology of Improved Programming Performance, Datamation, November, 1972.

Stewart, S.L. Concepts in Quality Software Design, Government Report, COM-74-50697, August, 1974

Lyon, G. Stillman, R.B. A FORTRAN Analyzer, Government Report Com. 74-50998, October, 1974.

Bullen, R.H.Jr. Software First Concepts, Mitre Corp. Technical Report, No. 2648, Vol. III, June, 1973.

Cheng, L.L. Some Case Studies in Structured Programming, Mitre Corporation, Technical Report No. 2648, Vol. VI, June, 1973.

Clapp, J.A., La Padula, L.J. Engineering of Quality Software Systems, Mitre Corp. Technical Report No. 2648, Vol. I, June, 1973.

Henderson, P., Snowdon, R. An Experiment in Structured Programming, BIT, Vol.12. 1972.

Browne, et al. Specifications and Programs for Computer Software Validation, Government Report N74-21831, November, 1973.

Bjorner, D. Flowchart Machines, Bit, Vol. 10, 1970.

Elshoff, J.L. A Case Study of Experiences with Top Down Design and Structured Programming, General Motors Research Laboratories Research Publication GMR-1742.

Stucki, L.G., Svegel, N.P. Software Automated Verification System Study, Government Report AD-784086, January, 1974.

Sites, R.L., Some Thoughts on Proving Clean Termination of Programs, Government Report PB 234102, May, 1974.

5.0 INSTALLATION AND USE OF THE DAVE SYSTEM AT TEXAS A&M UNIVERSITY

DAVE is a large FORTRAN program (approximately 22,000 FORTRAN statements and comments) designed to perform static analysis of FORTRAN programs. DAVE was developed at the University of Colorado by Professors Fosdick and Osterweil during the past two years.* A comprehensive description of the DAVE System is provided in Section 6 of this report. This section summarizes the experiences gained in installing DAVE at Texas A&M University for the eventual purpose of running it through DOMONIC.

DAVE was acquired by Texas A&M in order to gain firsthand experience with a large, automated program testing tool. Installation of DAVE was complicated by the fact that it was developed on a CDC 6400 Computer and was transported to the IBM 360/65 at Texas A&M. A previous attempt to install DAVE on the IBM Computer at Argonne National Laboratory was unsuccessful, largely due to incompatibilities between the CDC and IBM FORTRAN Systems.

A tape copy of DAVE was received by Texas A&M in May, 1975. During the next month, a low level effort was directed to reading the tape and setting up the 30 separate disk files of information that comprise the DAVE System.

In June, the head Colorado programmer on DAVE visited Texas A&M for a week, and substantial progress was made toward installation of DAVE. A short time later, after exchanging numerous phone calls and letters with Colorado, DAVE became operational at Texas A&M. As a result of the experience gained here, DAVE was successfully installed at Argonne National Laboratory.

* This work was sponsored by NSF Grant DCR 74-24546.

Many of the problems encountered during the installation of DAVE were caused by the size and complexity of DAVE and by the resulting communication problems in understanding what was required to install DAVE. For example, 30 files had to be created and named, and certain of the files had to be processed in specific order to initialize the system.

Other problems were installation dependent. In some cases, bugs in the DAVE System that did not affect the operations of DAVE on CDC equipment were exposed on the IBM System. For example, an array that was not properly initialized by DAVE did not affect the CDC operation, because the CDC System automatically initializes core memory to zero. However, the IBM System does not initialize core and the problem became evident at Texas A&M. Also, certain CDC non-ANSI characters were used in writing DAVE and had to be replaced with EBCDIC characters for IBM operation.

Another problem was that the maximum allowable length of a single record on IBM equipment is 32K bytes, which is less than the limit that was being used to write certain records of information on the CDC machine.

During the past two months (mid-July to mid-September), DAVE has been used to analyze several FORTRAN programs. Experience with DAVE indicates that it is a large system and that it is rather expensive to operate. For example, processing a 50-line FORTRAN program requires approximately 300K bytes of memory and 1.5 minutes of execution time. Compilation of the DAVE System, which is required only once per installation, needed 110K bytes of core memory and 25 minutes of execution time.

On the other hand, DAVE is very good at identifying illegal and questionable FORTRAN program constructions. A typical DAVE analysis of

a FORTRAN program is attached to this paper. The cost of analyzing a program using the DAVE System must of course be balanced against the cost of an undetected problem in the program.

One feature of the DAVE System deserves special comment: DAVE assumes that the program being analyzed adheres to the ANSI Standards for FORTRAN programs. However, it does not check for conformance. Thus, a non-ANSI program can result in unpredictable behavior by DAVE. The system would be improved by adding an ANSI checker as a preprocessor to the DAVE System. This would, of course, increase the size and complexity of an already large and complex system.

The following pages provide examples of output from the DAVE System.

```
LINE  STMT  BLOCK        SOURCE

1
    1     1     0          COMMON/BLK/S,R,XMAX,XMIN
    2     2     0          DIMENSION Q(100), R(100,2)
    3     3     1          READ(5,10)I,S
    4     4     0     10   FORMAT(I10,F6.2)
    5     5     1          CALL INIT(R,Q,I)
    6     6     1          WRITE(6,20)(Q(J),J=1,100)
    7     7     0     20   FORMAT(10F8.2)
    8     8     1          INTS=S
    9     9     1          IF(INTS.LE.0)
    9    10     2         $INS=1
   10    11     3          M=MAXMIN(R)*INTS
   11    12     3          STOP
   12    13     0          END
1
   13     1     0          SUBROUTINE INIT(A,VECTOR,I)
   14     2     0          DIMENSION A(100),VECTR(100)
   15     3     1          IF(I.LT.1.OR.I.GT.98)
   15     4     2         $I=1
   16     5     3          DO10 J=1,I
   17     6     4     10   A(J)=J*J-10*(J*J/10)
   18     7     5          IP1=I+1
   19     8     5          IF(I.GT.99)
   19     9     6         $I=99
   20    10     7          DO20 K=IP1,100
   21    11     8     20   A(J)=0
   22    12     9          READ(5,30)(VECTR(J),J=1,100)
   23    13     0     30   FORMAT(10F8.2)
   24    14     9          RETURN
   25    15     0          END
1
   26     1     0          FUNCTION MAXMIN(R)
   27     2     0          DIMENSION R(100)
   28     3     0          COMMON/BLK/RMAX,RMIN,DUMMY(201)
   29     4     1          RMAX=R(1)
   30     5     1          RMIN=R(1)
   31     6     1          DO 10 I = 1,100
   32     7     2          IF(RMAX.LT.R(I))
   32     8     3         $RMAX=R(I)
   33     9     4          IF(RMIN.GT.R(I))
   33    10     5         $RMIN=R(I)
   34    11     6     10   CONTINUE
   35    12     7          IF(RMAX.NE.RMIN)
   35    13     8         $MAXMIN=RMAX-RMIN
   36    14     9          RETURN
   37    15     0          END
```

```
0          GLOBAL MESSAGES FOR PROGRAM UNIT         INIT
0
0
 ************************************************************************
 *****WARNING

0      A VARIABLE IN A PARAMETER LIST IS USED FOR NEITHER INPUT NOR
       OUTPUT.

     NAME OF VARIABLE         VECTOR
 ************************************************************************
0          LOCAL MESSAGES FOR PROGRAM UNIT         INIT
0
0
 ************************************************************************
 *****WARNING

0      THE LOCAL VARIABLE NAMED   VECTR RECEIVES A VALUE IN ITS
       LAST USAGE.
 ************************************************************************
0
 ************************************************************************
 *****ERROR

0      THE VARIABLE NAMED        J BECOMES UNDEFINED (FALLING THROUGH
       STATEMENT NUMBER      6), YET IS ALWAYS USED THEREAFTER TO SUPPLY
       A VALUE.
 ************************************************************************
```

1 CLASSIFICATION OF PARAMETER AND COMMON VARIABLES

0 SUBROUTINE INIT

0. PARAMETERS/ENTRIES

ORDER	NAME	INPUT CLASS	OUTPUT CLASS
1	A	NON	STRICT
2	VECTOR	NON	NON
3	I	STRICT	OUTPUT

```
1
1                CALL GRAPH TABLE ENTRIES
0SUBPROGRAM NAME=    SYSMAIN
 PROCESSED AS ITEM NUMBER              1    IN THE INPUT FILE
 EXTERNAL CALLS=              2
0SUBPROGRAM NAME=       INIT
 PROCESSED AS ITEM NUMBER              2    IN THE INPUT FILE
 EXTERNAL CALLS=              0
 NAMES OF CALLERS
      SYSMAIN
0SUBPROGRAM NAME=     MAXMIN
 PROCESSED AS ITEM NUMBER              3    IN THE INPUT FILE
 EXTERNAL CALLS=              0
 NAMES OF CALLERS
      SYSMAIN
 NEXT LEAF IS FILE ENTRY NUMBER          2
 NEXT LEAF IS FILE ENTRY NUMBER          3
 NEXT LEAF IS FILE ENTRY NUMBER          1
```

```
1
0           GLOBAL MESSAGES FOR PROGRAM UNIT       MAXMIN
0
0
 ************************************************************************
 *****WARNING

0      A FUNCTION NAME IS NOT ALWAYS ASSIGNED A VALUE

     NAME OF FUNCTION          MAXMIN
 ************************************************************************
0           LOCAL MESSAGES FOR PROGRAM UNIT       MAXMIN
```

```
0
1              CLASSIFICATION OF PARAMETER AND COMMON VARIABLES
0                        FUNCTION          MAXMIN
0                          PARAMETERS/ENTRIES
       ORDER              NAME              INPUT CLASS              OUTPUT CLASS
          1             MAXMIN                NON                      OUTPUT
          2                  R                STRICT                   NON
0                        COMMON                    BLK
0                          PARAMETERS/ENTRIES
       ORDER              NAME              INPUT CLASS              OUTPUT CLASS
          1               RMAX                NON                      STRICT
          2               RMIN                NON                      STRICT
          3              DUMMY                NON                      NON
```

```
1
0          GLOBAL MESSAGES FOR PROGRAM UNIT
0
0
 ************************************************************
 *****WARNING          STATEMENT NO.     5   BASIC BLOCK NO.     1

0      CORRESPONDING ARGUMENTS IN THE PARAMETER LISTS ARE OF
       DIFFERENT DIMENSIONALITY.

                        CALLING SUBPROGRAM   CALLED SUBPROGRAM
                                 SYSMAIN              INIT
    ARGUMENT POSITION              1                   1
 NUMBER OF DIMENSIONS              2                   1
    NAME OF ARGUMENT                 R                   A
    KIND OF ARGUMENT          IDENTIFIER
          INPUT CLASS                          NON-INPUT
         OUTPUT CLASS                          STR.OUTPUT
 ************************************************************
0
 ************************************************************
 *****WARNING          STATEMENT NO.     5   BASIC BLOCK NO.     1

0      CORRESPONDING ARGUMENTS IN THE PARAMETER LISTS ARE OF
       DIFFERENT DIMENSIONALITY.

                        CALLING SUBPROGRAM   CALLED SUBPROGRAM
                                 SYSMAIN              INIT
    ARGUMENT POSITION              2                   2
 NUMBER OF DIMENSIONS              1                   0
    NAME OF ARGUMENT                 Q                VECTOR
    KIND OF ARGUMENT          IDENTIFIER
          INPUT CLASS                          NON-INPUT
         OUTPUT CLASS                          NON-OUTPUT
 ************************************************************
0
 ************************************************************
 *****WARNING          STATEMENT NO.    11   BASIC BLOCK NO.     3

0      CORRESPONDING ARGUMENTS IN THE PARAMETER LISTS ARE OF
       DIFFERENT DIMENSIONALITY.

                        CALLING SUBPROGRAM   CALLED SUBPROGRAM
                                 SYSMAIN             MAXMIN
    ARGUMENT POSITION              1                   1
 NUMBER OF DIMENSIONS              2                   1
    NAME OF ARGUMENT                 R                   P
    KIND OF ARGUMENT          IDENTIFIER
          INPUT CLASS                          STR.INPUT
         OUTPUT CLASS                          NON-OUTPUT
 ************************************************************
0
 ************************************************************
```

*****WARNING STATEMENT NO. 11 BASIC BLOCK NO. 3

Ø A POSSIBLE ILLEGAL SIDE EFFECT HAS BEEN DETECTED. AN ARGUMENT PASSED BY THE CALLING SUBPROGRAM (VIA PARAMETER LIST OR COMMON) IS ALTERED BY THE CALLED SUBPROGRAM AS AN INDIRECT CONSEQUENCE OF THE CALL.

	CALLING SUBPROGRAM	CALLED SUBPROGRAM
	SYSMAIN	MAXMIN
COMMON BLOCK NAME	BLK	BLK
NAME OF ARGUMENT	R	
INPUT CLASS	STR.INPUT	
OUTPUT CLASS	NON-OUTPUT	STR.OUTPUT

****

Ø LOCAL MESSAGES FOR PROGRAM UNIT
Ø
Ø

****
*****WARNING

Ø THE LOCAL VARIABLE NAMED I RECEIVES A VALUE IN ITS LAST USAGE

********** **********
Ø

****
*****WARNING

Ø THE LOCAL VARIABLE NAMED INS RECEIVES A VALUE IN ITS LAST USAGE.

****
Ø

****
*****WARNING

Ø THE LOCAL VARIABLE NAMED M RECEIVES A VALUE IN ITS LAST USAGE.

****

1 CLASSIFICATION OF PARAMETER AND COMMON VARIABLES
Ø COMMON BLK
2 PARAMETERS/ENTRIES

ORDER	NAME	INPUT CLASS	OUTPUT CLASS
1	S	NON	STRICT
2	R	NON	STRICT
3	XMAX	NON	NON
4	XMIN	NON	NON

6.0 TESTING, VALIDATION AND VERIFICATION

The objective of testing, validation and verification is to produce quality software. Yet, what is quality software? As the adjective implies, quality software is software of high value. It is software that performs the function it was designed to perform with dependability and reliability. It is software that is well-structured and documented, as well as easily manageable and modifiable.

This report is not concerned so much with the specifics of structure and documentation, et al, as it is with the testing aspect of reliability -- the elimination of errors or bugs in software. An error or bug indicates the software failed to perform its intended function for a particular input. This is not to say that structure and documentation are not important aspects of software quality that facilitate testing and, in fact, several aspects of developing software systems that facilitate testing will be discussed.

The following discussions deal primarily with testing batch programs. These techniques are applicable to some extent to time-sharing and real-time programs, but the additional complexities of these types of programs create a number of difficult problems for testing.

6.1 Terminology

To establish a basis for discussion, several definitions will be presented. Testing implies an attempt to measure how well specifications are met. There are two problems in this: first, to define the acceptance criteria to be used, and second, to specify unambiguously exactly what is

expected of a piece of software. The specification must be adaptable to allow extensions and alterations to the code without making the acceptance criteria ambiguous. Errors or bugs are examples of failures to meet the specifications for a particular piece of software. But differentiation between debugging and testing should be made. Testing has already been defined, debugging starts with known errors and attempts corrections. The two are related since testing discovers the inputs (errors) for the debugging process.

Validation will be defined as assuring the logical correctness of a program in its operating environment. While verification will be concerned with logical correctness independent of the program's environment, the non-logical properties such as resource utilization, execution time, I/O device requirements and functional measures of effectiveness will comprise the area of performance testing [1].

6.2 Goals of Testing

The primary goal of testing, as mentioned earlier, is to aid in producing a piece of quality software. The objective is to remove errors in the software so it meets its specifications. This is certainly a non-trivial problem. Dijsktra has said, "Testing shows the presence not the absence of errors" [2]. Unfortunately, this is all too true. Testing problems are broadly placed into two categories: 1) how do you test software, and 2) how do you know you have effectively tested the software. The types of testing will be discussed later.

Simply stated, the goal of testing is to eliminate errors. For a sophisticated piece of software, this becomes extremely difficult. The

feasibility of testing all variable ranges and branch paths may well require astronomical amounts of time [3]. So, what is the solution? After-the-fact testing, as already stated, is a very difficult task. Some solutions to the problem may be found in software generation practices. Many techniques are now coming into vogue that generate software with fewer errors and also facilitate testing [3a,b,c]. The latter concept is very important -- to design and generate software with testing plans incorporated. The above approaches will tend to make testing a more tractable problem.

Concomitant goals of testing are to minimize effort, cost and time for checking software. Also, it is desirable to have some means for measuring testing effectiveness. The continuous testing of software as it is being generated, plus documentation of this testing, greatly enhances the over-all aspect of testing.

Ideally, the test philosophy is that tests be complete, controlled, reproducible and documented in depth.

6.3 Types of Testing

Testing can be employed in a myriad of ways. The following sections discuss testing from informal tests through automated formal testing.

6.3.1 Informal Versus Formal

One of the basic premises of testing is to test early in the development. Statistics [4] have shown that testing can comprise up to 50% of software development time. One simplistic way of trying to offset this large expenditure of time is to train programmers to perform informal testing as they develop the software. Simple desk checking and the running

of a simple test case on small sections of code as it is developed can be very beneficial.

Formal testing can cover several aspects. Formal test specifications may be created with guidelines as to their executions. Formal proofs of correctness may be attempted. These are discussed in a later section.

Separate test teams may be formed to test the software. For if a separate test team can indeed test the software using only the system documentation as provided by the developers, we have a better piece of software.

6.3.2 Testing Stages

Beyond the stage of informal individual programmer testing of software, testing may be viewed in three stages; integration, acceptance and field. Software systems are usually developed in modules. As modules are completed, they must be integrated into the total system and interfaces must be checked out to assure compatibility. Once all the modules have been integrated into the system and it has been delivered to the customer, the software system must undergo an acceptance test. Here, specifications for the system are checked against the operational software. Certain software systems may have field sites for operation. If this is the case, tests to check confirmation with field specifications are necessary.

6.3.3 Manual Versus Automated

The discussion of testing so far has centered around temporal aspects. But once we have decided when to test, how do we actually perform the testing? Testing runs the gamut from structured manual to automated systems --

which of these techniques is applicable? How can test data be generated, manually or automatically? These topics will be covered in a latter part of the report.

6.4 Integrated Top-Down Testing

Classical software development has been a bottom-up procedure where the lowest level programs are coded first and then tested and integrated into the system. Extraneous driver programs are needed to perform testing and lower levels of integration. Data definitions and interfaces tend to be simultaneously defined by several people and therefore inconsistent. Therefore, during integration definition problems arise. Interfaces and data definitions frequently need to be reworked. Problem isolation is difficult because of the large number of possible sources.

6.4.1 Top-Down Development

The top-down approach is modeled after the approach to system designs and requires that programming proceed from developing the interfaces and data definitions downward to developing and integrating the functional units [5]. Top-down programming is an ordering of system development which allows for continual integration of the system parts as they are developed and provides for interfaces prior to the parts being developed.

In top-down programming, the system is organized into a tree structure of segments. The top segment contains the highest level of control logic and decisions within the program, and either passes control to lower level segments, or identifies lower level segments for in-line inclusion. This process continues for as many levels as required until all functions within

a system are defined in executable code. The top-down approach requires that the data base definition statements be coded and that actual data records be generated before exercising any segment which references them.

Software is produced which is always operable and always available for successive levels of testing that accompany the corresponding levels of implementation.

6.4.2 Testing and Integration

The top-down approach to testing and integration starts with the testing of the highest level system segment once it is coded. Since this segment will normally call or include lower level segments, code must exist for the next lower level segment. This code, called a program stub, may be empty, may output a message for debugging purposes each time it is executed or may provide a minimal subset of the functions required. These stubs are later expanded into full functional segments, which in turn require lower level segments. Integration is, therefore, a continuous activity throughout the development process. During testing, the system executes the segments that are coded and uses the stubs as substitutes for what is not yet coded. Thus, the need for special test data drivers is eliminated. The developing system itself can support testing because all the code that is to be executed before the newly added segments has previously been integrated and tested and can be used to supply test data to the new segments. Therefore, most problems are localized to the newly added code. As the new segments are tested within the developing system, the control architecture and system logic are also tested.

The testing cycle is partitioned in the following manner. Test requirements identify the functions to be tested, specify the number of cases, the ranges and limits of data and describe the hardware and software environment. The test requirements specify the degree to which the product goals -- function, interaction, performance, operability, and useability are evaluated.

The test specification details the test design approach and test structure, and identifies the methodology and procedures for testing.

The design review tests the software specification compliance with system requirements and assesses implementational feasibility. The review also evaluates accuracy, compatibility with other software and hardware and compliance to standards.

6.5 Automated Testing Tools

Automated testing tools can be partitioned broadly into two categories. First, the automatic generation of test data for exercising software and, second, the automatic monitoring of software to obtain characteristics of the software. Obviously, the two interface, for the test data may drive the program for automatic monitoring.

6.5.1 Automatic Test Generation

Three aspects of automatic test generation are the generation of pattern data, variable range data and data for testing possible branch paths. Of course, the last two may be intimately connected.

6.5.1.1 In the IBM OS utilities there exists a program, IEBDG, a data set utility to provide a pattern of test data. Seven patterns are available;

alphanumeric, alphabetic, zoned decimal, packed decimal, binary number, collating sequence and random number. The user supplies as input to IEBDG the pattern type and the appropriate output pattern (random) is generated.

6.5.1.2 The question arises as to how to generate data to test variable ranges. It would be desirable to have languages that allow variable range specification. In turn, the compiler could insert code to check these ranges.

Another approach being investigated is the Monte-Carlo generation of software tests. This is coupled with heuristics to make the test generation process more efficient by increasing the probability that a randomly generated test will exercise a portion of the range of a variable which had not previously been exercised and test the code more thoroughly [6].

6.5.1.3 A third problem is concerned with obtaining an optimal set of test cases which exercise all branches in the source code of the user's software modules. Exercising every path is impractical, but it is desirable and feasible to exercise all logical branches in a module. A determination of these paths allows generation of the test data.

To arrive at this objective, a segment of code may be defined as the smallest set of consecutively executable statements to which control can be transferred during program execution. The first statement will be directly accessible from another segment and the last will be a transfer to a new segment. The segment relationship will be defined as the relationship between two segments of code resulting from the transfer of control of execution from the first segment to the second. The objective now

becomes: find the minimum set of paths which exercise all the segment relationships in any subject module.

A method has been devised and mechanized to automatically generate the optimal path. This path and the relevant branch information in source code form is displayed to the user to aid in generating the required test data for execution [3].

Howden [7] has developed a method for decomposing a program into a finite set of classes of paths in a manner such that an intuitively complete set of test cases would cause the execution of one path in each class. The approach attempts to generate test data for as many of the classes of paths as possible. The method constructs descriptions of input data subsets which cause the classes of paths to be followed. Then the method transforms these descriptions into systems of predicates which it attempts to solve.

Miller [8] has based automatic test case generation on a priori knowledge of two forms of internal information: a representation of the tree of subschema automatically identified from within each program text, and a representation of the iteration structure of each subschema. This partition of a large program allows for efficient and effective automatic test case generation using backtracking techniques.

During backtracking, a number of simplifying, consolidating, and consistency analyses are applied. The result is either (1) early recognition of the impossibility of a particular program flow, or (2) efficient generation of input variable specifications which cause the test case to traverse each portion of the required program flow.

6.5.2 Automated Monitoring Systems

The following sections review five automated software systems for the facilitation of testing and consequently improving software reliability.

Two types of analysis may be recognized - static and dynamic. Static anaylsis results in code being examined without execution, while dynamic analysis studies execution time characteristics. The following systems exemplify both types.

6.5.2.1 PET

The Program Evaluator and Tester (PET) [9] is a program test evaluation tool which automatically generates self-metric software from existing software. Basically, the technique instruments the source code to effectively measure its own behavior. This system was developed at McDonnell Douglas Astronautics Company.

This system has been implemented for FORTRAN and consequently has demonstrated the value of a self-metric approach for higher level languages. This tool basically instruments an application software package by inserting the software equivalent of sensors into the package. Therefore the package is self-measuring. Each time a significant event occurs, the system records it.

Two techniques have been used to implement software sensors: (1) direct code insertion, and (2) invocation of runtime routines. The direct code insertion appears to be faster in most cases but the run-time routine is more flexible in that measurements can be more easily altered at execution time.

As a result of running the instrumented application program, a profile is produced containing part or all of the following measurements:

1. The number and percentage of all potential executable source statements which were executed one or more times.
2. The number and percentage of those program branches taken.
3. The number and percentage of those subroutine calls which were executed.
4. The number of times each subroutine was called, together with a list of those subroutines that were never entered.
5. Relative timing on the subroutine level.
6. Specific data associated with each executable source statement.
   a. Detailed execution counts.
   b. Detailed branch counts on all IF and GOTO statements.
   c. Optional data range values (min/max/first/last) on assignment statements.
   d. Optional min/max ranges on DO-loop control variables.

These summaries and detailed reports can be employed to establish a figure for the degree of testing to which the program has been subjected.

PET consists of 2 major components (Figure 1):

(1) A highly structured preprocessor which instruments the source program in such a way as to make it self-metric, and

(2) a post-processor to generate reports from the execution measurement data produced by the instrumented self-metric software.

Both of these test systems components are written in a high-level

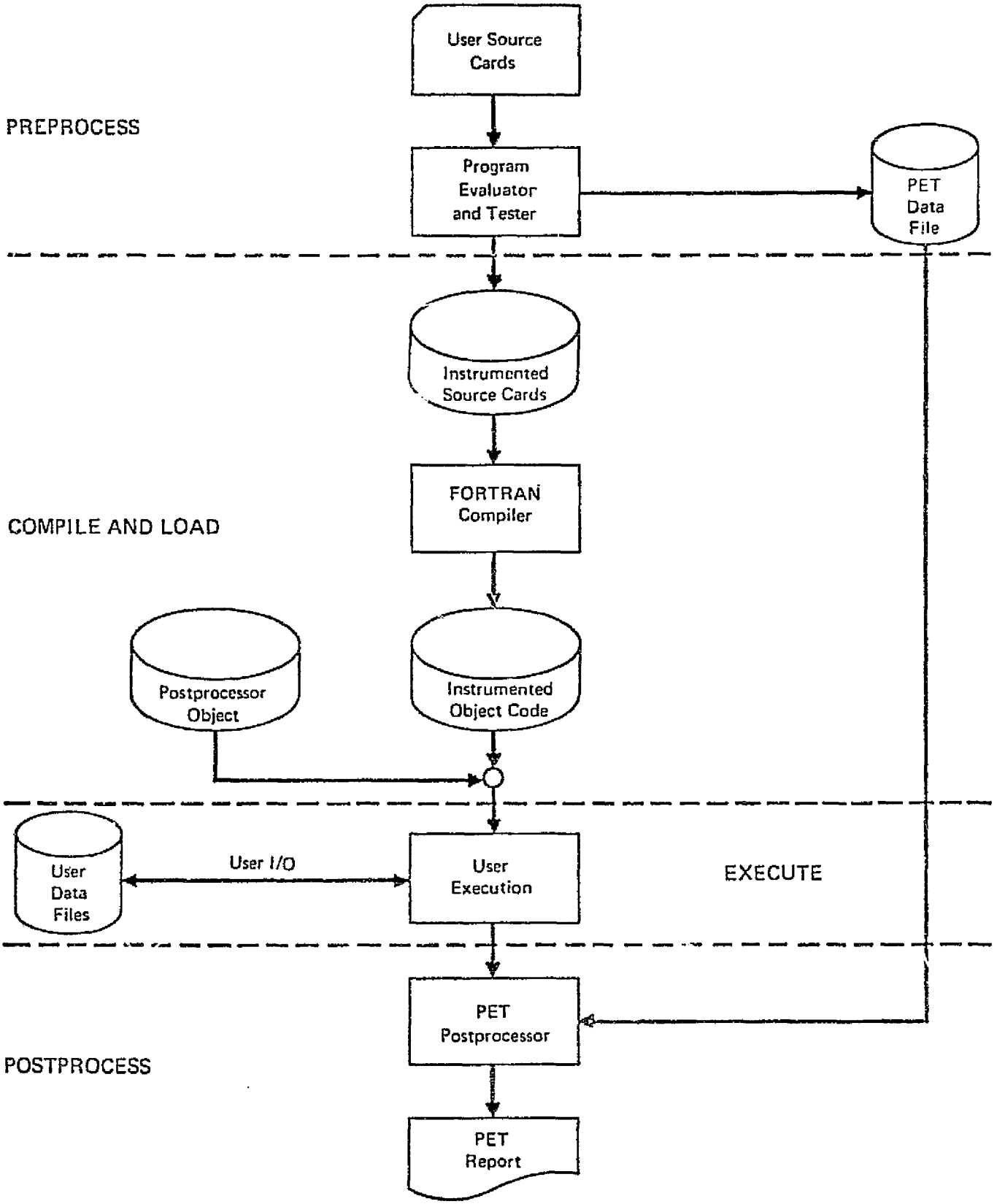


Figure 1. Program Evaluator and Tester (PET) Job Flow

language to create a more easily maintained system. The post-processor component will generally be written in the same high level source language for which the system was designed.

As an example of results obtained from using PET, consider the evaluation of an eigenvalue eigenvector routine which had been in production for over two years. Counts of 'true' and 'false' evaluations for branches revealed a logically impossible flow path. Statistics showed that only 44.5% of the possible executable source statements were actually tested and only 35.1% of the possible branches were tested. Information such as the above tends to indicate the software we design may be very inefficient and wasteful.

Both space and time artifacts are introduced when using self-metric instrumentation. The time artifact ranges between approximately a 1.25 to 2.5 factor for the execution time of a self-metric program, depending on the measurement options specified.

The space artifact also varies widely depending on the types of measurements being performed. Space artifacts are introduced in the following areas:

(1) Additional memory is allocated for counters, timing cells, and data range storage cells.

(2) Additional code is added to make the program self-metric.

6.5.2.2 PACE

The Product Assurance Confidence Evaluator (PACE) System [10] is designed to provide programmers with debugging tools and managers support in determining and controlling computer program quality. Specifically, PACE assists in the planning, production execution and evaluation of computer

program testing. PACE consists of the following four phases:

(1) Test Planning (Preparation of test materials)

- ° Analysis of computer program anatomy to determine what must be tested
- ° Instrumentation of the program to render it measurable as a test item
- ° Development of test data to exercise the desired portions of program anatomy

(2) Test Production (Synthesis of test materials into a test package)

- ° Synthesis of test stimulus data for a test
- ° Selection of test driver and data environment structure for a test
- ° Configuration of test job containing the above materials ready for execution

(3) Test Execution (Operation of the computer program with test data)

- ° Computer execution of the instrumented test item
- ° Measurement and recording of test output

(4) Test Evaluation (Analysis of test results and program performance)

- ° Analysis of execution frequency of program elements
- ° Analysis of comprehensiveness of test execution
- ° Assessment of validation confidence

The object of PACE is not to find errors, but to quantitatively assess how thoroughly and rigorously a program has been tested and to use this information in the improvement of test design in order to prescribe and carry out the conditions for validation.

PACE has been used both for FORTRAN and assembly language programs. The initial implementation of PACE was the FLOW module. FLOW analyzes a FORTRAN program and instruments the code such that subsequent compilation and execution is allowed. FLOW provides for an accumulation of frequencies with which selected elements (e.g. statements, small segments of code, subprograms, etc.) are exercised as the program is being tested. A modification of, and extension to, FLOW for UNIVAC systems has recently been completed. This system is called TDEM (Test Data Effectiveness Measurement).

The TDEM systems consists of three elements: QAMOD - the code analysis instruction program, QAPROC - which monitors execution and provides summary statistics, detailed trace information and an indication of the effectiveness of the test data, and QATRAK - which uses these results and displays internal program transfer variables which can be changed to effect execution of the unexercised code.

QATRAK also displays the statements which compute or input the transfer variables. Figures 2 and 3 illustrate the program and data file interfaces of the TDEM subsystem.

The QAMOD program sequence analyzes each statement of a FORTRAN source program and the following results occur:

(1) The first executable statement of each element (i.e., subroutine or main program) is assigned a pseudo statement number of one. Each subsequent statement is assigned a sequential pseudo number and the statements are displayed with their pseudo numbers. Statements are later referenced from QAPROC and QATRAK by element name and pseudo number.

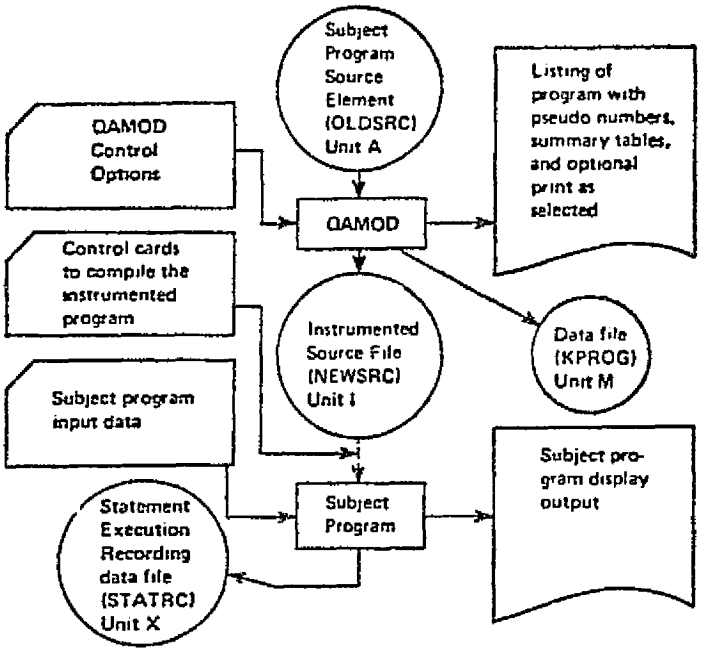


FIGURE 2 .TDEM/QAMOD Interfaces

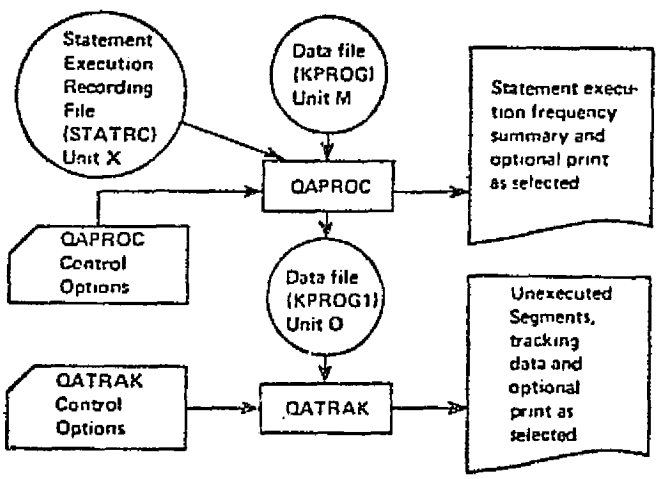


FIGURE 3 .TDEM/QAPROC-QATRAK Interfaces

(2) The code is instrumented by the insertion of traps to an execution monitor subroutine. The function of these traps is the generation of a recording file during execution of the instrumented program. The recording file registers the execution of each statement and the order of execution.

After the analysis, the instrumented source code is output to a file, NEWSRC, for compilation and execution.

A data file (KPROG) is also generated as QAMOD processed the program. This file contains information describing each statement and information relative to program size and structure.

The QAPROC program accesses the statement execution recording file generated by execution of the instrumented subject program and produces an evaluation and summary of the test case executed. The recording file is sequentially accessed and the data are assimilated into an internal table, MAPTAB. At times designated by the input control options, a display is printed which included the following:

(1) A map, delineated by subroutine, indicating the number of executions which have been recorded for each statement.

(2) Statistics indicating the percentage of the total executable statements which were executed.

(3) Statistics indicating the percentage of the total number of subroutines which were executed.

(4) A list of the names of subroutines which were not executed.

After processing the entire recording file, statement usage frequency

information is added to the data from the input KPROG file and this revised information is output on a data file KPROG1. Statistics from recording files (i.e. several executions of the subject program) may be summed and a cumulative summary compiled.

6.5.2.3 ACES

The Automated Code Evaluation System (ACES) [11] is a language processor which examines source statements, performs a lexical and syntactical analysis, and generates a data base containing symbols and their use, a list of statement types and a graphical representation of the program structure. The major advantabe of this system is the transfer of large amounts of programming code into a more usable form (the data base) for use in the validation process. An examination of the data base might show which subroutines and functions are called by a system component and the names of parameters used in each call, allowing an analysis of interfaces.

Analysis

ACES detects two types of program errors - those related to semantics and language constructs and those related to program structure and well-formation. Additionally, program characteristics are collected and stored in the data base, and an automatic monitor insertion feature is included for an execution-time analysis of specified program variables.

Lexical Analysis

A lexical and syntactical analysis forms the basis for the detection of semantic errors and undependable language constructs. Since the programs submitted to ACES are supposedly working programs, they should contain few

syntactical or semantical errors. This allows the statement scanning routines to quickly pick out the pertinent information. For example, assignment statements may be scanned for variable and function names while ignoring all intervening operators.

One of the primary functions of the lexical analysis is the detection of undependable language constructs. Of course, this tends to be very language dependent. Present programming languages are designed primarily for effectiveness and flexibility rather than absolute dependability. As a result, they often contain language constructs which are conducive to execution errors.

CENTRAN, the language ACES was designed for, is no exception to this rule. Certain constructs, such as GOTO like statements, are susceptible to execution-time errors. ACES notes such occurrences and the corresponding statement numbers are stored for an error summary.

The analysis is a simple recognition process, but it does automatically pinpoint sources for error. An extension of this process would be an automatic examination of critical variables involved in statements such as computed GOTO's. Careful examination and consideration of a programming language is necessary to determine error prone conditions such as those described above.

Data Base Generation

A primary feature of ACES is extraction of program characteristics and the construction of a data base which provides a convenient means of retrieving this information.

Therefore, the program investigators do not have to tediously examine program listings for the information to validate the program. The data base consists of four tables: symbol table, symbol use table, statement type table, and an abbreviated connection matrix.

The symbol table contains information regarding all variables, items, functions, macros, and labels used in a program. An entry in this table consists of the symbol name, module number, type, and linkage to the symbol use table. The symbol use table contains a record of the use of a symbol name in a program. An entry consists of an indicator for the type of use (either input or output), the statement number in which the symbol was used, and linkage to other references to the symbol contained in the table. The statement type table is simply a list of codes indicating the statement type of each statement in the program. The logical structure of a program is stored as an abbreviated connection matrix. Thus the data base provides statistical information on symbol names and statements and can answer questions such as:

(1) Does variable V_i appear as an input (output) to any of the following statements: S_1, S_2, ..., S_k?

(2) In what statements does V_i occur?

(3) What are the inputs (outputs) to statements S_k?

(4) Does any variable appear as an output and not as an input?

(5) What are the inputs for conditional branch S_i? Where do they appear as outputs? What are the inputs to these statements?

This information is important in the analysis of program behavior and is particularly useful as an aid to implementing changes in syntax, program

modifications, and changes in programming practices. For example, the effects of changes in a program variable, macro, or label can be easily determined by accessing the list of references to that symbol in the program module and other related modules.

Structural Analysis

The analysis of program structure is essential to the validation process since it allows the detection of structural flows and the examination of critical or interesting flow paths through the program.

Program structure is modelled by ACES as a directed graph in which nodes represent program elements and edges represent lines of program flow. Program elements may be either single statements or groups of statements making up a program segment. The graph is first generated and stored using individual statements as elements and later reduced to reflect the relations between program segments.

An intermediate representation of program structure, consisting of a list of "non-normal transition pairs," is generated from the existing tables to serve as a basis for the following analysis. "Non-normal transition pairs," (i,j) represent permissible transitions from statement i to statement j, excluding "fall-through transitions (j=i+1)." The necessary information for their generation is the use of labels (explicit transfers) and the identification of statement types (implicit transfers e.g. IF statement). Special codes are inserted as the second element of a pair for certain transitions, such as RETURN, END, computed GOTO, etc. The list of pairs is then transformed into a more usable set of linked lists.

From this information and that of the symbol table, it becomes a simple matter to extract the structural characteristics of a program. These include lists of undefined labels, unreferenced labels, unreachable statements, statements with no successors within the program (RETURN, END, true faults), and direct predecessors to all labeled statements. In addition, an enumeration of program loops is provided.

The structural analysis performed by ACES also includes the generation of reaching vectors for a specified set of statements. The reaching vector of a particular statement provides a list of those statements whose execution may lead to the execution of the statement in question. This information can be easily extracted from the linked list representation of the program graph.

Automatic Monitor Insertion

The methods of analysis presented thus far are of a static nature, i.e., the execution of the program to be validated is not involved. As previously mentioned, exhaustive testing of large programs is not feasible. However, the behavior of certain critical variables may be important in validation and a means of making selective observations at run-time would be valuable. This capability is provided to a limited extent in debugging systems through trace and trap routines. The approach taken in ACES is somewhat different in that the program source code is temporarily modified by automatically inserting calls to a monitoring routine. This relieves the investigator of the tedious task of locating all occurrences of a given variable and allows flexibility in the monitor functions performed.

The input to ACES for the monitoring function is simply a list of variables and their corresponding upper and lower bounds. The occurrence of one of these variables as output from a statement causes the ACES system to insert a call to a CENTRAN subroutine. This subroutine, which has been written for use in conjunction with ACES, determines whether the current value of a variable is within the specified bounds. If an out-of-bounds value is detected, the variable name, value, and corresponding statement number are reported to the user. Otherwise, no report is made. This differs from the usual trace procedure of reporting every change of value.

The system also allows the value of a specified array to be checked in toto, i.e. each change in value of an array element is checked against the bounds for the entire array. In addition, provisions are made to monitor implicit changes in program variables. For example, CENTRAN allows the declaration and referencing of bit patterns (items) with a data word. If one item overlaps another item of the same variable, it is possible for the value of one item to be changed by changing the value of the other item. Subtle errors may be caused by such a condition which are very difficult to detect. The detection of implicit changes therefore causes the monitoring subroutine to produce a warning message at run time.

Organization and Operation of ACES

The operation of the ACES package is performed in three phases: information gathering and monitor insertion, information publication, and structural analysis (Figure 4). These phases are further divided into functional elements (modules) each of which contains one or more subroutines. Thus a modular structure is imposed on the system, facilitating modification and extension as well as debugging efforts.

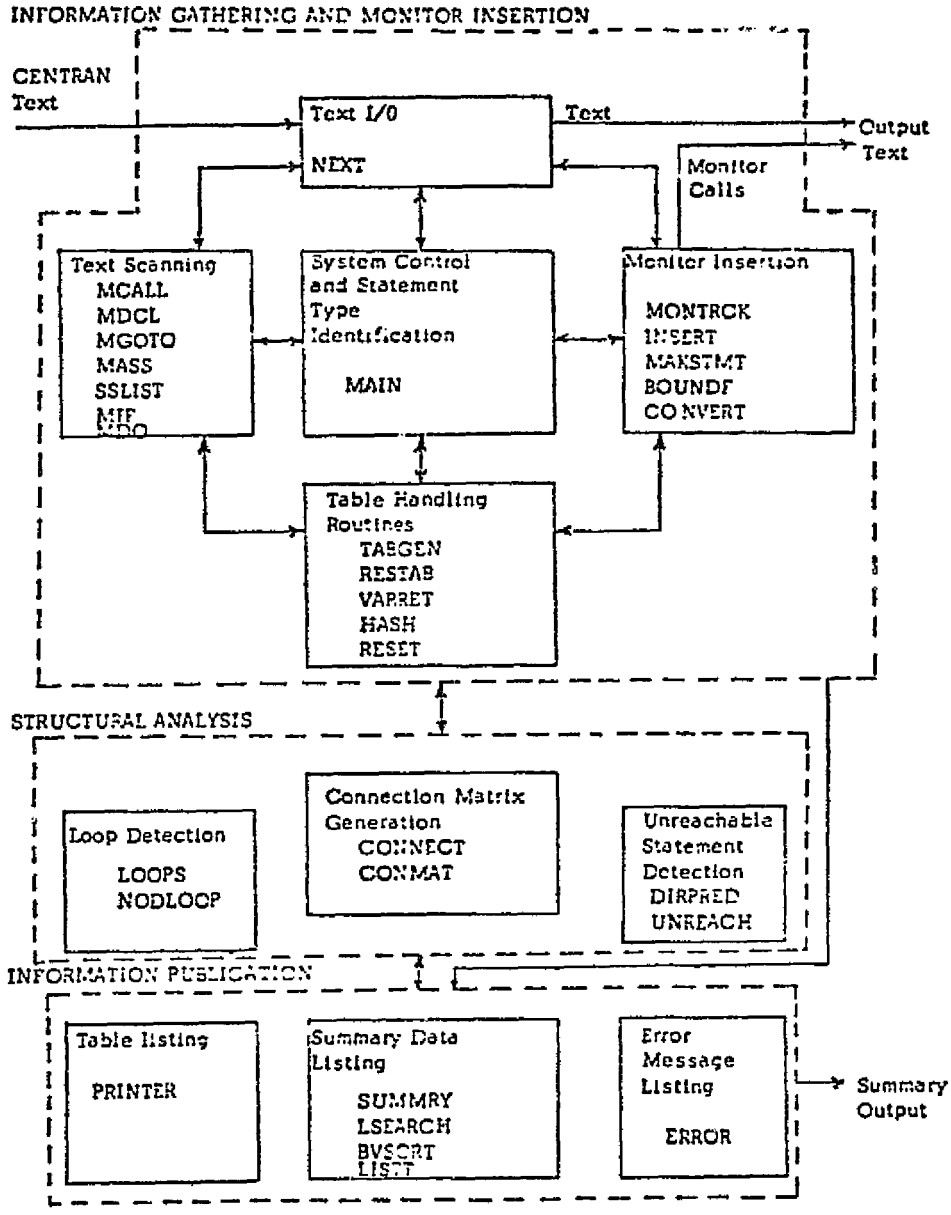


Fig. 4. Block Structure of Automated Software Evaluation System.

Phase I

Phase I consists basically of scanning the CENTRAN text, collecting information to be stored in the data base, and making any necessary modifications to the source code. The functional modules involved are similar to those utilized by standard language processors (compilers).

Phase II

Structural analysis constitutes a second phase of ACES processing. Three functional modules provide for connection matrix generation, unreachable statement detection, and loop detection. These functions are performed in sequential fashion in the manner previously described. A separate module for the detection of a reaching vector may be used in this phase of system processing or as a stand-alone program operating on the stored program graph (connection matrix).

Phase III

A final phase of processing consists of the publication of information generated by the system. The extent of information reported is left to the discretion of the user. A scan of the data base results in a summary listing which includes various statements and variable cross reference listings, warning messages issued by the system, structural characteristics, etc.

The results of ACES analysis are a modified source listing, a printed summary of information generated and a data base to be used for further analysis on a local or global level.

A complete system evaluation is beyond the capability of ACES. However, systems such as ACES can be powerful tools when judiciously used in an overall validation scheme. At the very least, they are quite useful in gleaning

information from large volumes of source code which could not otherwise be obtained. When properly utilized, such automated systems provide the framework for three important validation concepts:

(1) Establishment of a running configuration of the software system through well-formation analysis.

(2) Generation of a structured data base for debugging and for checking the compatibility of program updates and modifications.

(3) Instrumentation of the source code for run time analysis and simulation.

Some of the features of ACES would be most helpful in debugging efforts during program development. For this reason, the incorporation of such features in compilers and run time systems would be valuable and could reduce debugging time while producing more dependable code. However, this does not reduce their effectiveness in evaluating existing software systems.

6.5.2.4 ISM

The ISM System [6.5.2.4] is designed to allow experimentation with a wide variety of information collection, analysis, and display tools. The design methodology is applicable to procedural programming languages, and ALGOL 60 is being used as the vehicle for elaboration of design principles and implementation techniques.

ISM System Design Concepts

The ISM system provides the capability for characterizing programs by both a static syntactic structure and a dynamic run time structure. Static information is obtained by performing a syntactic analysis of the program

text and recording certain structural attributes of the program in a data base. Other attributes of the program can then be inferred from the collected data. For example, recording attributes such as:

- modes and types of identifiers
- statement numbers that each identifier appears in
- statement types
- input/output variables of statements
- basic blocks
- control paths

permits inference of attributes such as:

- graph structure of the program
- classes of execution paths
- unreachable code segments
- input/output variables to subroutines
- calling sequences of subroutines

Dynamic analysis involves recording the program's execution history (or some portion of it) into a data base and gleaning desired information from the data after the program has terminated execution. The execution history is collected by instrumenting the source program with subroutine calls to record program history events in the data base. History collecting subroutine calls are inserted into the source program by a preprocessor prior to compilation of the program, thus rendering the source program "self-metric." The ISM System is designed to collect, analyze, and display both static and dynamic information. However, major emphasis is placed on processing of dynamic information.

Dynamic information can be classified under the headings of:

- Execution Summary Statistics
- Control Flow Information
- Data Flow Information
- Data Sensitivity Information
- Execution Environment Information
- Assertion Verification

Execution summary statistics include ranges of variables, statement execution counts, number of traversals of program segments, and timing estimates. Control flow and data flow information can be combined to provide variable dependency and computation dependency information. As is subsequently described, the execution history can be interpreted in reverse, permitting control flow and data flow tracebacks. Data sensitivity information can be collected to show the effects of input data inaccuracies and finite word length by tracing the numerical significance of the computation. Environmental information includes scope and extent of identifiers, and parameter passing and procedure evaluation environments. Assertions can be local or global. A local assertion is a conjecture about the state of the computation at a particular point in the execution sequence. A global assertion is a conjecture about invariant conditions throughout a given segment (perhaps all) of the execution history. Assertions are verified (or refuted) by comparing the expected behavior to the execution history.

The primary advantages of analyzing and displaying results from a post mortem data base after the program has terminated execution are:

1. The program is executed in the actual usage environment.
2. The functions of data collection and data analysis are separated, thus making the analysis and display tools independent of the programming language being analyzed.
3. Experimentation with a variety of testing aids is facilitated.
4. Global summary information can be collected.
5. Global assertions concerning behavior can be checked.
6. The execution history can be interrupted backward in time, permitting analysis and display of how a given computation was influenced by previous computations.
7. Existing processors (compilers, interpreters, loaders, libraries) can be utilized to collect the data base.

The primary disadvantages of the data base approach are: the inability of the user to directly interact with an executing program, the potentially large size of the data base, and the execution timing distortions caused by the history gathering subroutines. The first difficulty is somewhat alleviated by the ability to re-execute the instrumented program using new input data entered via the user's terminal, thus creating a new execution history. The third problem is a consequence of the design methodology and is shared by all self-metric performance evaluation packages.

The ISM Data Base

Major components of the data base are: an identifier table, a program model, and one or more execution histories. In addition, the original source text, and a textual cross reference table between the program model and source text are maintained.

The various components of the ISM data base are interfaced to permit the association of names with values, and control flow with program text. The entire history can be searched to collect global information and summary statistics. For example, ranges of variables can be obtained, assertions about program behavior can be checked, data and control flow traces can be accomplished, and statement execution counts can be obtained by interrogating the data base. Local information concerning program behavior can be obtained by aligning the history pointer to a particular position in the program model and examining the computational state. Information is recorded into the history to permit interpretation of the program model either forward or backward in execution time. Thus, execution can be reversed in order to determine how a particular computational state was influenced by previous states.

The ISM Preprocessor

The preprocessor builds the symbol table and program model, prepares a compressed version of the source program, and instruments the compressed source code with subroutine calls. The various modules of the preprocessor are generated automatically by a parser generating program (the PARSEC metatranslator). PARSEC accepts a BNF-like notation (PARSEL) as input, and generates a translator to parse and perform semantic actions on programs whose syntax conforms to the grammar specified in PARSEL.

ISM Program Analysis and Display

Analysis and display of program behavior is accomplished by routines that access the data base via the primitive accessing functions, and construct

appropriately formatted information displays. The data base can be accessed in either the batch or the interactive mode, and the displays can be displayed on an interactive terminal or printed as batch output.

Displays are termed semantic models of program execution. Many different types of semantic models are required to display the various attributes of computer programs. A major goal of this study is to determine what information is useful for testing purposes and how to display that information in meaningful formats.

A partial list of useful information that can be displayed includes:

- variable range summaries
- statement execution counts
- branch execution counts
- syntactic structure of various program components
- control flow traces
- data flow traces
- control flow tracebacks
- data flow tracebacks
- data sensitivity analysis
- identifier accessing environments
- parameter passing environments
- procedure evaluation environments
- recursive procedure environments
- timing estimates
- assertion checks

Program execution can be observed in either the control flow domain or the data flow domain. In the control flow domain, control flow and source code displays can be combined to allow the user to observe the source statements as they are executed in either the forward or backward direction. The complete source code of every statement can be displayed, or source code can be suppressed below a given level. In addition, data values can be associated with every variable in every statement, or some variables in some statements, if desired.

In the data flow domain, the sequence of values assigned to variables can be displayed as they evolve forward or backward in execution time. The association of source text and control flow information with particular data values is possible. Thus, the dependence of data values on other data values can be obtained from the data base, along with the source text associated with those values.

Preprocessing and execution of the program being tested results in the creation of a data base that contains the source text, a symbol table, a program, model, and one or more execution histories. The various components of the data base are interfaced to permit the association of names with data values, and source text with control flow. Each step in the execution history can be reconstructed, permitting both forward and backward execution of the program. The entire history can be scanned to collect global information and summary statistics. The data base is constructed using primitive construction functions, and accessed using primitive accession functions, thus isolating the details of internal data base organization from construction and access.

6.5.2.5 DAVE

A study of data flow within a program provides a key to an understanding of program behavior. Such problems as detection of semantic program errors, automated creation of assertions and automated production of program documentation can be attacked. Thus, the data flow analysis system to be described tracks the flow of data from statement to statement, block to block, and subprogram to subprogram. The system is called DAVE (Documentation, Assertion generating, Validation, Error detection) [3].

The flow of control is represented by a directed graph. The nodes in this graph are sequences of statements called basic blocks. A basic block is a maximal sequence of statements having the property that whenever any one of the statements in the basic block is executed, every statement in the basic block is executed.

A variable plays a role in data flow for execution of a statement, a basic block, and a subprogram by assigning an input-output classification to it for each of these structures. In a statement such as:

```
A = B + C
```

the variables B and C are referenced to define a value for A. To identify the role of the variables B and C they are called strict input variables for this statement and A is a strict output variable for this statement. In a statement a variable may be strict input and strict output; this is the case for X in the statement:

```
X = X + Y
```

For completeness, the input role and output role of a variable in a statement should be classified. In the first statement above A is non-input

and strict output (NI,SO) while B and C are non-output and strict input (SI,NO). The classification is extended to basic blocks and subprograms. Thus a basic block or subprogram strict input variable is one which is referenced by the block or subprogram before all definitions of the variable. A strict output is one which is defined for all control paths within the block or subprogram.

Data Flow Anomalies Detected

DAVE recognizes two types of data flow anomalies: event 1 - referring to a variable which has not been assigned a value on a path leading to this reference; event 2 - assigning a value to a variable which is not referenced on a path leading from this assignment. Either of these events is considered symptomatic of an error. Event 1 anomalies violate the principle that a value must flow into a variable before it can flow out, and event 2 anomalies violate the principle that data which flows into a variable should flow out. The viewpoint here is that there is a conservation principle to be applied to the data flow: it should be free of sources and sinks, excepting data boundary points (READ's and WRITE's) and violation of this principle is likely to be symptomatic of errors in the program. Violation of the principle may be traced to such things as: key-punch error, misspelling, statements out of order, failure to initialize, incorrect label, incorrect use of parameters in a subprogram reference, etc.

DAVE issues messages where the presence of data flow anomalies is detected or suspected. These messages are in the form of warnings and errors. Errors consist of those situations which are certain to yield an illegal computation, while warning messages are issued only when the

possibility of an illegal computation is established. Observation of event 1 on all control paths leading to a statement will cause an error message to be issued, while a warning message is issued if the event is present on some, but not all, control paths. A warning is issued even if event 2 is present on all paths, because event 2 does not seem to imply an erroneous computation in the same way event 1 does.

Structure of the Analysis Program

The structure of DAVE is indicated in Figure 5. The subject program, consisting of a main program and all subprograms referenced either directly or indirectly is first preprocessed by a program analysis and instrumentation package. It is assumed that the subject program is a syntactically correct ANSI FORTRAN program, however, as noted below recovery procedures are possible when illegal statements are encountered.

During this pass, the program is divided into program units and these are divided into basic blocks and statements. Statement type determination is also made here. The preprocessed program is then passed to a lexical analysis routine. This routine creates a token list to represent each of the program's source statements. Clearly, knowledge of the statement type makes the job of the token list generator easier.

As the token lists are created, comprehensive data bases of information about the various program units are also created. The data bases are accessed using a data base creation and accessing package, designed to facilitate data base restructuring. Each subprogram data base contains a symbol table, label table, statement table, and table of subprogram wide data. The symbol and label tables contain much the same type of information

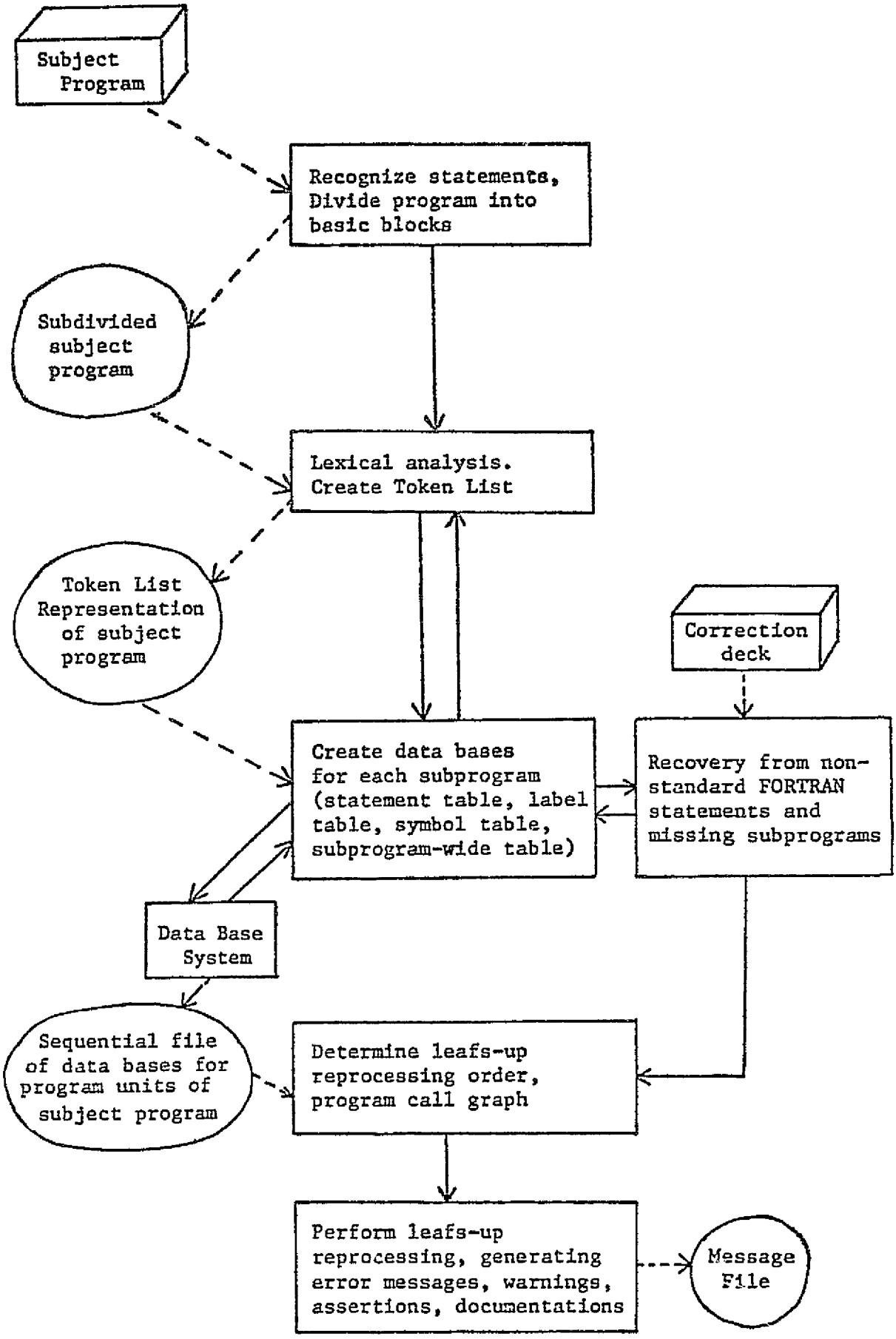


Figure 5. Structure of DAVE.

found in most compiler symbol and label tables, listing symbol and label attributes as well as the locations of all references to the symbols and labels. The primary purpose of the statement table is to hold the input-output classification for every variable referenced or defined in each executable or DATA statement. During this lexical scan phase, it is possible to determine the input-output classes of all variable references and definitions except those in which variables are used as arguments to subprogram invocations. DAVE contains the input-output classifications for ANSI FORTRAN intrinsic functions and basic external functions so the input-output classes of variables used as arguments in these functions are also determined during this phase. This determinable input-output data is stored in the statement table. Blanks are placed in the statement table for the input-output classifications of variables used as actual parameters in subprogram invocations; these blanks will be filled in during a later phase of processing.

The table of subprogram-wide data for a given program unit contains a list of all subprograms referenced by the program unit, as well as representation of non-local variable lists. Ultimately, the non-local variable lists will be used to hold data about the subprogram-wide input-output behavior of these non-local variables. The external reference lists of the various program units will be used to construct the program call graph.

During this phase of processing, statements which are syntactically illegal under the ANSI standard may be encountered. The system is capable of pausing at this point and accepting a correction deck containing replacements for the offending statements. In addition, the system will examine

the external reference lists to determine whether all referenced subprograms have been submitted. If not, DAVE will, at this time, also accept new symbolic decks in order to satisfy such unsatisfied external references.

In the next phase, DAVE builds and examines the program call graph. Using the call graph, leaf subprograms (those with no external references) are identified. For such subprograms, the input-output classifications of all non-local variables are made. Hence, the input-output behavior of all variables used in all invocations of such subprograms can now be filled in, enabling in turn determinations of input-output classifications of non-local variables in other subprograms. Using this scheme, all input-output classifications can eventually be entered for all variables in all statement table entries in all program unit data bases. This leafs-up (inverse invocation order) subprogram reprocessing order is determined in the next phase through analysis of the program call graph.

The final phase of processing is the most interesting. During this phase, the program units are reprocessed in the above-mentioned leafs-up order. Missing input-output information is supplied, and global data flow analysis is performed. It is at this time that events of types 1 and 2 in the data flow are identified, and data flow assertions are made.

Data Flow Analysis Phase of Processing

The final phase of processing begins with the analysis of leaf subprograms. The analysis begins with the construction of a basic block table for the subprogram. This table holds input-output information about all variables referenced in each of the basic blocks. It is constructed from data in the subprogram's statement table.

Once the basic block table has been constructed, the input-output classification of program variables can be determined through the use of an algorithm [14]. The local variables are analyzed first. An error message is generated for all local variables which are found to be strict input f subprogram, since this situation implies a type 1 anomaly is certain. Correspondingly, local variables found to be of type input cause the generation of a warning message. The last usage of all local variables is also determined by means of an output category classification algorithm. If a local variable is used last as an output, an event of type 2 is present and a warning is issued.

The input-output classifications of the non-local variables are then determined. These classifications are printed out, and also stored in the subprogram-wide table of the subprogram under study. Warning messages are also printed for all parameters which are found to be non-input and non-output. Clearly each of these items of data in the subprogram-wide table can be viewed as being an automatically generated assertion about the subprogram. These assertions are useful moreover in producing documentation about the subprogram. This table is then copied into a master data base, so that all invoking program units will be able to easily access the data needed to classify the input-output categories of variables used as arguments in invocations of this subprogram.

The analysis of a non-leaf program unit is more complicated. Such a program unit will, of course, not be analyzed until all subprograms which it calls have been analyzed. At such a time, however, it is possible to fill in all entries which had to be left blank during the creation of the

calling unit's statement table. Hence, such blanks are filled in. Certain FORTRAN semantic errors are also detected as this proceeds.

The system also exposes concealed data flows through subprogram invocations. Concealed data flows result from the use of COMMON variables as inputs (or outputs) to (from) an invoked subprogram. Such situations are easily exposed by examination of the COMMON block variable lists in the subprogram-wide table of the invoked program. Because data flows through such COMMON variables just as surely as through explicitly referenced parameters, the statement table entry of such an invocation statement is augmented by the input-output classifications of such variables. This assures that the results of global input-output category determination within the invoking program unit will be correct for these variables. DAVE can also print out the names and usages of all the variables which are used as input or outputs to a statement but are not explicitly referenced. Such information seems to be useful as a form of automated documentation. It also seems to be useful as a debugging aid in that it may alert a programmer to data flows which are hidden, perhaps forgotten, and hence more prone to error.

The omission of a COMMON block declaration in an invoking program unit presents a tricky problem. If the COMMON block is referenced in the invoked subprogram, then the variables named in the COMMON block may or may not become undefined upon return to the calling program unit. Undefinition will not occur provided that the COMMON block is defined in some program unit currently invoking the program unit which omits the COMMON declaration. In the absence of such a reference by a higher level program unit, errors are

possible. In particular, variables in such a COMMON block which are strict output or output from the invoked subprogram will become undefined - a type 2 event - and a warning is issued. Variables in such a COMMON block which are strict input or input can receive values only through BLOCK DATA subprograms. Hence, a check of the subprogram-wide tables of such subprograms is made. If no data initialization is found, a warning is issued.

If a COMMON block, B, is declared by a high level program unit which invokes a subprogram, S, in which the block is not declared, then the ANSI standard specifies that B must still be regarded as implicitly defined in S provided that some subprogram directly or indirectly invoked by S does declare B. Hence, data referenced by the variables in B may flow freely through routines which do not even make reference to B. As already observed, such data flows are noted and monitored by DAVE. In addition, DAVE is capable of printing out the names and descriptions of all COMMON blocks whose declarations are implicit in a given subprogram. This, too, seems to be useful program documentation. The algorithm for determining which blocks are implicitly defined in which routines involves a preliminary leafs-up pass though the program call graph and then a final root to leafs pass.

Only after all of the above described checking and insertion of input-output data into the statement table has been done, does the system proceed to the creation of the basic block table. As might be expected, the creation of the basic block table entry for a basic block containing subprogram invocations is rather complicated. The algorithm must contend with such problems as non-strict usage of variables, and references to variables not explicitly named.

Once the basic block table is constructed, analysis of the variables, explicit and implicit, proceeds as described in the case of a leaf subprogram.

Subprograms are processed in this way until the main program is reached. Processing of the main program is the same as the processing of any non-leaf, except that COMMON variables must be treated differently. Any COMMON variable which has an input or strict input classification for the main program must be initialized in a BLOCK DATA subprogram, if not, a warning message (class is input) or an error message (class is strict input) is issued. Similarly, if a COMMON variable's last use was as an output from a main program, a warning is issued.

6.5.2.6 Discussion

Exclusive of the DAVE system, the other systems ascertain test-effectiveness by determining whether : (a) each source statement has been executed at least once; (2) each branch path has been executed at least once; and (c) each subroutine call has been executed at least once. With this information the systems seek to reduce the errors associated with the actual structure of the computer program to a minimum.

In addition to measuring test effectiveness, the tools may be useful for debugging and tuning purposes. Branching problems and code receiving high or low usage can be determined by the tools. These features and others such as tracing subroutine calls and reporting of assignment statement limit values are important to understanding and correcting improper program operation.

Tuning involves studying high usage areas of code, then trimming and altering the code to produce more efficient operation. Analyses have shown

that small percentages of code execute large percentages of the time - therefore the need for tuning. Simplification of code can dramatically impact the execution-time requirements it places on the system.

The objective of the test tools is aimed at increasing the reliability of computer programs. Statistical information (4) indicates they only partially fulfill their purpose. Sequencing and control errors can be significantly reduced, but these errors account for only about 20% of the total error types commonly found.

Functional testing (conformation to specifications) is necessary, especially if the software is time-critical. Structural analysis tools do not attempt to test either the timing or data relationships that exist within computer programs. Functional testing is also necessary to determine that all required functions were implemented and that no functions were implemented which were not specified.

The key to effective and efficient use of these test tools to do structural analysis is to base their use on levels of criticality applied to the individual program modules. This aids in the goal of reducing the cost required (CPU time, analyst hours and documentation) for using these test tools properly. In certain cases these costs can easily outweigh the benefits derived.

6.5.3 Debugging Techniques

We have specifically delineated testing and debugging into two separate categories. The former is the precurser of the latter. Yet this is not a hard and fixed definition and therefore, some aspects of debugging will be discussed - primarily, interactive and batch debugging systems [15]. The

philosophy taken is that the presence of an error is known, and some means for locating the error is desired. Also, two additional requirements on debugging aids will be assumed - transparency to the program and ease of identification and removal.

6.5.3.1 Batch Debugging

For batch purposes there are some relatively simple techniques for debugging programs. Hopefully, most syntactic errors will be detected by the language translator. The simple concept of hand checking at a desk can be beneficial, but is frequently not exploited. The programmer should exhibit 'good citizenship' - completely checking each run to locate as many errors as possible. Lazy programmers exhibit 'poor citizenship' by trying to have the computer solve all their problems.

Cross-references produced by language translators also prove to be invaluable for debugging; multiply defined variables, point of definition and points of reference can be studied.

Trace routines may be invoked to follow instruction-by-instruction the program execution. Control flow and variable information are presented in the trace output.

Dumps provide useful debugging aids. They may be complete, selective or snapshot. The dump simply allows an examination of core images.

Execution profiles reflect characteristics such as control flow, variable ranges, instruction execution frequency, plus additional features that are dependent on the system producing the profile. Several of these

systems are discussed in section 6.5.2.

6.5.3.2 Interactive Debugging

Interactive debugging implies time-sharing execution at some type of TTY or CRT terminal. Debugging aids similar to batch systems exist, but interactive debugging extends beyond these through on-line interaction, ease of use and speed of response.

Incremental execution with incremental traces may be invoked interactively. A good text editor will allow rapid and sophisticated modification of program text. With an interactive system, insertion and deletion of code becomes very simple. The ability to stop execution and change variable values is a powerful tool.

Interactive computing tends to encourage 'bad citizenship' because of its very nature. It is certainly easier in this environment to let the computer perform the work for you.

6.6 Certification

Certification can be considered the endpoint of testing, validation and verification [16]. Ultimately a certified program is one which has been widely accepted within the community of experts and users. It carries the connotation of an authoritative endorsement and seems to imply testifying in writing that the program is of a certain standard of quality. To assure this credibility, the process of certification should include examination of:

1) completeness of program documentation
2) performance of the program relative to its documentation
3) comparison of the program with others of the same type in terms

appropriate to the problem

4) adequacy of continuing maintenance and support

A formal guarantee that the certification process has been satisfactorily performed would be expressed by a document issued by an agency or institution recognized by the communities of users and experts.

6.7 Proof of Correctness

Given the formal specifications and the text of a program in a formally defined language, the question can be asked whether the program text is correct with respect to those specifications.

It should be made clear that a proof of correctness is decidedly different from the standard process of testing a program. Testing can and often does prove a program is incorrect, but no reasonable amount of testing can ever prove that a non-trivial program will be correct over all allowable inputs [17,18].

6.7.1 Nature of Correctness Proofs

The idea of proving program correctness was conceived to a large extent in the earliest days of computing. Goldstine and von Neuman noted that a proof of program correctness could be achieved if the programmer could provide or assert a description of the state of the vector of program variables after each step, or only selected steps, of the program. This state information, which can be viewed as a relation among program variables at a given instant, can then be used to verify the program.

This assertion approach has been recently formalized by Floyd [19] and Manna [20]. The goal with regard to proving the implications associated

with state assertions, is for the mechanization of the process through the use of automatic theorem-proving programs.

The procedure is generally known as the method of inductive assertions. The assertion at the output is the specification of the correctness of the program. The assertions at the input define the input conditions for which the program is to produce output satisfying the output assertion.

The proof technique works as follows: somewhere within each loop of a program an assertion is added that adequately characterizes an invariant property of the loop. It is now possible to break the flow chart of a program into tree-like sections such that each section begins and ends with assertions and no section contains a loop. It is desired to show that if the execution of a section begins in a state with the assertion at its head true, when the execution leaves that section, the assertion at the exit must also be true. By taking an assertion at the end of each of these sections and using the semantics of the program statement above it, one can generate an assertion which should have held before that statement if the assertions after it are to be guaranteed true. Working up the tree, all the assertions at the head of the respective sections can be generated. Each section will then preserve truth from its first to its last assertions if the first assertion implies the assertion that was generated at the head of the section. One thus obtains the logical theorems or verification conditions for each section. If these theorems can all be proven and if the program halts, then it will halt with the correct output values.

6.7.2 Manual Proofs

The size of programs which can be proven by hand depends on the level of formality that is used. A more informal approach to proofs has come into vogue. The approach is rigorous, but uses a level of formality like that in mathematics texts. Arguments are based on an intuitive definition of the semantics of the programming language without a complete axiomatization. Using these techniques a number of efficient programs to do sorting, merging and search have been proven correct. The proof of a twenty line sort program may require three pages.

A person proving a program correct by manual techniques must first achieve a very thorough understanding of all details of the program. Manual proofs techniques therefore are clearly limited to programs simple enough to be totally comprehended by the program provers.

6.7.3 Automated Proofs of Correctness

Computer generated proofs at the present have not produced very meaningful results except for very small and simple programs. Effort is now being directed toward computer assistance for proving correctness. This takes the form of systems to generate verification conditions and to do proof checking, formal simplification and editing and semi-automatic or interactive theorem proving. Unfortunately, at this time almost any automation of the proof process forces one into more detailed formalism and reduces the size of the program that can be proven. This is because the logical size of the proof steps that can be taken in a partially automated proof system is still quite small.

6.7.4 Integrating Proofs with Program Design

Classically, proving a program correct has been done after the program is written. An alternative is to integrate the proof with the program design. This direction gives credence to the hope that proofs will eventually help to organize and simplify the program production process. A proof of correctness will greatly increase the amount of formalism that must be dealt with. However, if a proof can be integrated into the design and writing stages, it should eliminate most of the need for debugging and may alleviate the problems of documentation and maintenance.

6.7.5 Discussion

Programs can be said to be correct only with regard to formal specifications or I/O assertions. Nothing says these specifications express what the function of the program actually is. The assertions must be manually produced, requiring in-depth understanding of the program. A real problem lies in the fact that even for simple programs, the theorems that are generated become quite long. After the proof is completed, a number of things could still be wrong. What is actually proven may not be what one thought was being proved. The proof may be incorrect or assumptions about either the execution environment or the problem domain may not be valid.

In light of the above inherent difficulties, one may ask - Is proving correctness worthwhile? The answer must be affirmative to the extent that proving correctness causes an in-depth inspection of the program and consequently may bring to light problems within the program, plus in general a better overall understanding of the program.

6.8 Summary

As we view the prospect of testing a piece of software, hopefully much thought and concern with regard to testing has taken place in the specification, design, coding and documentation of the code. If not, testing tends to become an almost intractable problem, especially for a large software system.

What are some of the features of the code and its development that cause the code to be amenable to testing? The code should be readable (well documented), easy to modify, easy to maintain. The code should be structured in modules. The code should be simple - eliminate coding tricks.

Aspects of the management of code development are very important. Top-down development presents several advantages for its use. Continual awareness of testing and planning for testing as code is developed are important considerations.

The use of automatic monitor systems can check the degree to which code is exercised and produce useful execution profiles. Automatic test case generation is still in the rudimentary stages of development but offers promise for the future.

6.9 Bibliography

1. Hetzel, W.C., "A Defintional Framework," in Program Test Methods, Prentice-Hall, 1973.

2. Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, October 1969.

3. Krause, K.W., et al, "Optimal Software Test Planning Through Automated Network Analysis," Record of 1973 IEEE Symposium on Computer Software Reliability, 1973.

3a. Donaldson, J.R., "Structured Programming," Datamation, December 1973.

3b. Baker, F.T. and Mills, H.O., "Chief Programmer Teams,"Datamation, December 1973.

3c. Miller, E.F. and Lindamood, G.E., "Structured Programming: Top-down Approach," Datamation, December 1973.

4. Boehm, B.W., "Software and its Impact: A Quantitative Assessment," Datamation, May 1973.

5. McHenry, R.C., "Management Concepts for Top-down Structured Programming," IBM Corporation, Gaithersburg, Md.

6. Lundstrom, S.F., Ph.D.Dissertation incomplete.

7. Howden, W.E., "Methodology for the Generation of Program Test Data," IEEE Transactions on Computers, Vol. C-24, No. 5, May 1975.

8. Miller, E.F. and Melton, R.A., "Automated Generation of Testcase Datasets," Proceedings International Conference on Software Reliability, April 1975.

9. Stucki, L.G. "Automatic Generation of Self-Metric Software," 1973 IEEE Symposium on Computer Software Reliability, April 30, 1973.

10. Brown, J.R., et. al. "Automated Software Quality Assurance," Program Test Methods, Prentice-Hall, 1973.

11. Ramamoorthy, C.V., et. al. "Design and Construction of an Automated Software Evaluation System," 1973 IEEE Symposium on Computer Software Reliability, April 30, 1973.

12. Fairley, R.E. "An Experimental Program Testing Facility," Manuscript submitted for publication.

13. Osterweil, L.J. and Fosdick, L.D. "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection," Dept. of Computer Science, University of Colorado Technical Report CU-CS-055-74, September 1974.

14. Osterweil, L.J. and Fosdick, L.D. "Automated I/O Variable Classification as an Aid to the Validation of Fortran Programs," Dept. of Computer Science, University of Colorado Technical Report CU-CS-055-74, #37, January 1974.

15. Aron, J.D., The Programs Development Process, Addison-Wesley, 1974.

16. Keirstead, R.E. and Parker, D.B., "The Feasibility of Formal Certification," in Program Test Methods, Prentice-Hall, 1973.

17. Linden, T.A., "A Summary of Progress Toward Proving Program Correctness," FJCC, 1972.

18. Elspas, B. et. al, An Assessment of Techniques for Proving Program Correctness, Computing Surveys, Vol. 4, No. 2, June 1972.

19. Floyd, R.W., "Assigning Meanings to Programs,"Proceedings Mathematical Aspects of Computer Science, 1967.

20 Manna, Z. "The Correctness of Programs," Journal of Computer and System Sciences, Vol. 3, #2, May 1969.

6.10 References

Paige, M.R. and Miller, E.F., "Methodology for Software Validation - A Survey of the literature," Gen. Research Corp., March 1972.

Reifer, D. and Ettenger, R.L. "Test Tools: Are they a Cure-all?" prepared for Space and Missile Systems Organization Air Force Systems Commands, October 1974.

Rustin, R., Editor, Debugging Techniques in Large Systems, Prentice-Hall, 1971

MODERN SOFTWARE DESIGN TECHNIQUES

7.0 INTRODUCTION

The end product of the software design process is a set of design specifications for a software system that will implement an acceptable solution to the problem at hand. Design specifications are detailed descriptions of the algorithms, data structures, and interfaces necessary to satisfy the functional requirements of a system. Typically, the functional requirements are the starting point of a software development project; they include a description of the problem to be solved, and the constraints that exist for its solution. Design specifications are derived from the functional requirements, and in turn form the basis for implementation, acceptance testing, and delivery of the system; design specifications thus provide the link between functional requirements and an implemented software system that satisfies those requirements.

Creative aspects of the software design process include establishing a conceptual view of the system, developing algorithms and data structures to reflect that conceptualization, identifying system functions, decoupling the functions, decomposing functions to elementary levels, deciding what functions to place in which program modules, establishing interfaces between modules, and establishing interfaces to global data structures. All of this must be accomplished within the framework of meeting operational requirements, satisfying various design constraints, and promoting desirable quality attributes in the system.

The cost, complexity, and failure rates of existing software systems are well known (1). During the past few years, software has increased in

size and complexity to the point that software design and development costs presently exceed computing hardware costs; the expectation is that this trend will continue into the foreseeable future. Software is typically late in delivery, overpriced and unreliable. In addition to the high development costs of new systems, enormous fiscal and social costs are accrued by poorly designed, unreliable software systems that are in the "production" phase of the software life cycle. There is considerable social and economic incentive for the systematic design and implementation of reliable and efficient software systems, developed on time and within cost estimates. This task is the charter of the software engineering discipline.

The goal of systematic software design is development of detailed design specifications for software systems that will meet their operational requirements, satisfy various design constraints, and exhibit desirable quality attributes. Design constraints are imposed by functional requirements, and by the resources available to implement and maintain the system. The functional requirements might, for example, specify COBOL as the implementation language (perhaps in the interest of transportability), even though technical considerations would favor the use of a special purpose language more suitable to the particular application area. Resources required to implement and maintain a system include the hardware, supporting software, personnel (programmers, operators, users), and time. A software system must of course be realistic in terms of resource utilization.

The primary quality attributes of software include design clarity, reliability, efficiency, and modifiability. Design clarity and reliability

are generally the most important attributes of a software system. Efficiency is usually a secondary consideration to reliability; efficiency is of interest only when the system is functioning properly. There are two aspects to design clarity: First is the clarity with which the system design reflects the functional requirements; and second is the degree to which the design specifications are mirrors in the source code implementation of those specifications. Both aspects of design clarity are essential in achieving a well-designed and properly-implemented software system. Design clarity contributes to all of the other quality attributes, including efficiency; the performance of a well-designed and properly-implemented system is by definition more easily measured and tuned than is that of a poorly designed system. Similarly, a well designed system will be easier to understand and debug, hence easier to modify and maintain.

Functional modularity is the key to design clarity. Modularity is achieved by decomposing a system into distinct program modules that communicate through well-defined interfaces. Program modules are named sequences of statements that can be referred to by their collective name. Methods of implementing program modules include closed subroutines, macros, and library members. Methods for establishing the interfaces between modules are discussed later.

Functional modularity is achieved by identifying each module in the system with a specific, well-defined system function. Functional modularity reduces system complexity and enhances design clarity by decoupling the interactions among modules. This decoupling has numerous beneficial effects: interfaces between modules are explicitly specified as part of the design

process; modules can be tested either independently or in integrated fashion; errors and design deficiencies are more easily localized; and modifications can be made with minimal side effects.

The art and craft of software design is comprised of identifying the system functions, deciding what functions to put into which modules, and establishing interfaces between the modules. However, modularization of a software system is a somewhat arbitrary process without a conceptual framework for systematically achieving the goals of the design process. Liskov (2) discusses the fact that improper modularization may introduce additional complexity into a system in one or more of the following ways:

(1) too many related but different functions in a module will tend to obscure the logic with tests to distinguish among the various functions

(2) a common function is not identified soon enough, and as a result, it is distributed among several different modules, obscuring the logic of each

(3) modules may interact on common data in unexpected ways.

In this chapter, several techniques for achieving modular designs are described. Also included are discussions of notational schemes for specifying the design, intramodule design, and the influence of the implementation language on the design. The thesis of this chapter is that software quality must be designed into a software system, and that design techniques and notational schemes are available to facilitate the production of high quality software systems.

The effect of size and scale must be taken into account in any meaningful

discussion of software design. In a small system (one written by one man in less than one month) modular design will result in a superior product, but is perhaps not essential to the success of the project. In larger systems, a methodical approach to modular design is a necessary condition for the success of the project.

7.1 Basic Design Strategies

Two basic strategies for achieving a modular design are the "top-down" and "bottom-up" approaches. Using the top-down approach, attention is first focused on global aspects of the overall system. As the design progresses, the system is decomposed into sub-systems and more consideration is given to specific issues. In the bottom-up approach to software design, the designer first attempts to identify a set of primitive concepts, notions and actions. Higher level concepts are then formulated in terms of the basic ones. The system design is thus facilitated by identification of the "proper" set of primitive ideas. In practice, the design of a software system is seldom (if ever) accomplished in pure top-down or pure bottom-up fashion. However, most authors advocate a predominantly top-down design strategy.

The top-down strategy of decomposing tasks into algorithms and data into data structures has been termed "step-wise refinement," "step-wise program development," and "successive refinement" (3). The basic principles of step-wise refinement include:

(1) decomposing design decisions to elementary levels,

(2) isolating design aspects that are not truly interdependent, and

(3) postponing decisions concerning representation details as long as possible.

Numerous examples of the step-wise program development process can be found in references 3, 4, 5, and 6. Perhaps the best known example for illustrating step-wise program development is the 8 queens problem discussed by Wirth in reference 5, and by Dijkstra in reference 6.

The concept of backtracking is fundamental to top-down design. As design decisions are decomposed to more elementary levels it may become apparent that higher level decisions have led to an inefficient or awkward modularization of lower level functions. Thus, a higher level decision may have to be reconsidered and the system restructured accordingly. In order to minimize backtracking, many designers advocate a mixed strategy which is predominantly top-down, but which permits specification of the lowest level modules first. A pure top-down strategy is most successful when a well-defined environment exists for software development; as for example, in writing a compiler for use with an existing operating system. When the environment is ill defined, as in the development of an operating system for a new machine, the design strategy must of necessity be a mixed strategy or perhaps a predominantly bottom-up strategy.

7.2 Interface Design

The techniques by which interfaces between modules are established provide a point of reference for discussing modular design methodologies. The types of interfaces that exist betwen modules include: control interfaces, data interfaces, and service interfaces. Control interfaces exist in the invocations of, and in the entry and exit points of, the various modules. Data interfaces are established by the parameters used to pass information between modules, and by global data that is referenced in two or more modules.

Service interfaces among modules are manifest in the services that modules perform for one another.

7.2.1 Control Interface Design

Traditional software design techniques concentrate on control interfaces. Systems designed along control interface lines are characterized by the use of flowcharts as design tools, and the program modules are typically implemented as subroutines. This strategy can produce clearly defined control interfaces, but it also tends to produce complex data interfaces.

A control interface design methodology that has yielded impressive results is the strategy of integrated top-down design, coding, and testing [7]. In integrated top-down design, coding, and testing, the design proceeds top-down from the highest level routine whose primary function is to coordinate the sequencing of lower level routines. Lower level routines may be implementations of elementary functions (those which call no other routines), or they may invoke more primitive routines in order to accomplish their function. There is thus a hierarchical structure to a top-down system in which routines can invoke lower level routines but cannot invoke routines on the same or a higher level.

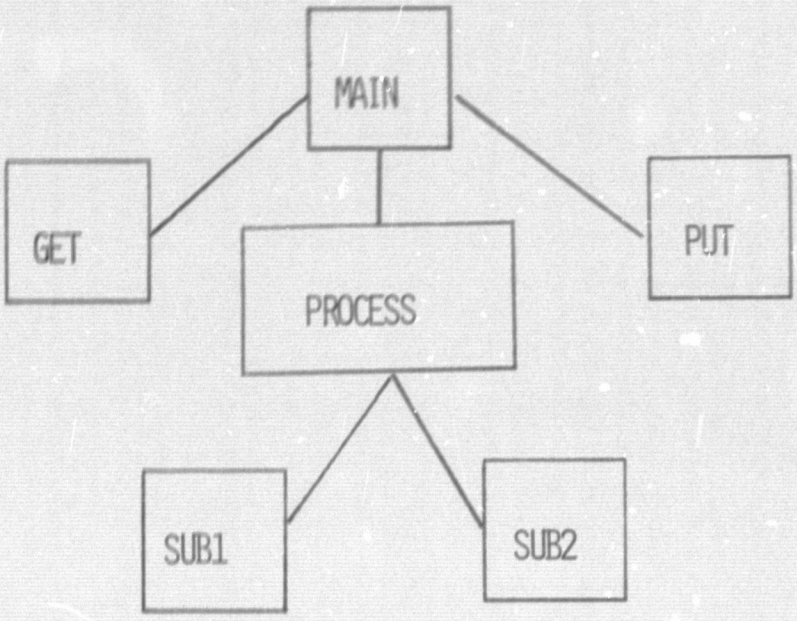
The integration of design, coding, and testing is illustrated by the following example. It is assumed that the design of the system has proceeded to the point illustrated in Figure 1. The purpose of procedure MAIN is to coordinate and sequence the GET, PROCESS, and PUT modules. These three modules can communicate only through MAIN; similarly, SUB1 and SUB2 (which support PROCESS), can communicate only through PROCESS. Some designers would allow MAIN to communicate directly with SUB1 and SUB2 while others

would require that MAIN communicate with SUB1 and SUB2 by going through PROCESS. In some cases a designer might restrict communication of data between modules to the parameter lists, while in other cases global variables might be permitted. A reasonable compromise is to restrict access of common global data to modules on the same level of hierarchy. This approach is particularly attractive when each hierarchical level is identified as a "level of abstraction" in the system design [8].

The coding and testing strategy for the example might be as illustrated in Figure 1.

Stubs are dummy modules that are written to simulate subfunctions in support of higher level functions. As coding and testing progresses, they are expanded into full functional units which may in turn require lower level functions to support them. The stub can provide a number of useful purposes prior to expansion into a functional unit. Stubs can provide output messages, test input parameters, provide simulated output parameters, and simulate timing requirements and resource utilization. In this manner, a simulated system can be operational as the design progresses.

The integrated top-down strategy provides an orderly and systematic framework for system development. Design and coding are integrated because expansion of a stub will typically require creation of new stubs to support it. Test cases are developed systematically and each module is tested in a simulated operating environment. A further advantage of the integrated top-down approach is the reduction of the system integration phase of the project; the interfaces are established, coded, and tested as the design progresses. The primary disadvantages of the top-down approach is that early,



STRATEGY:		
	CODE	MAIN
	STUBS FOR	GET, PROCESS, PUT
	TEST	MAIN
	CODE	GET
	TEST	MAIN, GET
	CODE	PROCESS
	STUBS FOR	SUB1, SUB2
	TEST	MAIN, GET, PROCESS
	CODE	PUT
	TEST	MAIN, GET, PROCESS, PUT
	CODE	SUB1
	TEST	MAIN, GET, PROCESS, PUT, SUB1
	CODE	SUB2
	TEST	MAIN, GET, PROCESS, PUT, SUB1, SUB2

FIGURE 1. TOP-DOWN INTEGRATED CODING AND TESTING

high level design decisions (such as choice of data representations) may have to be reconsidered when the design progresses to the lower level functions. This may require design backtracking, and considerable rewriting of code.

7.2.2 Data Interface Design

Traditional approaches to data coupling between program modules include: parameter lists in the calling module, global variables known in two or more modules, and access to a common data base by several modules. Many transaction driven systems are designed around a large data base, which is the focal point of the design.

Liskov has described a design strategy which emphasizes data interfaces [9]. In her approach, a software system is viewed as a collection of abstract data types and operations on those data types. An abstract data type is defined in an "Operation Cluster," which defines the data type in terms of the operations that can be performed on it. For example, a stack might be defined by the abstract operators: push, pop, return top, erase top, and empty test. The internal details of operation clusters are hidden from the modules that make use of the clusters. Thus, a stack can only be accessed by the defining operators and their parameters. This reinforces the functional modularity of the system.

A programming language called CLU is being developed to support direct implementation of software systems designed following the data interface strategy. In CLU, a cluster definition consists of four parts: (1) a description of the interface which the cluster presents to its users (cluster name, parameters, and list of operations defining the cluster type),

(2) details concerning the internal representation of the data type, (3) the code required to create instances of the data type, and (4) the operator definitions. Operator definitions are similar to ordinary procedure declarations, except that they have meaning only as part of a cluster declaration.

The cluster description defines the template of an abstract data type; instances of that type are created by assigning the template name to program variables. It is therefore possible to define the abstract data type "stack" and to create and manipulate several instances of stacks in the program. The situation is analogous to the treatment of classes in SIMULA [10]. Because the manipulation of abstract data types involves their defining operations, most of the procedure calls (abstract operations) in a program are specific to, and subordinate to, the data types being manipulated in the program. In this manner, the data interfaces in the program are emphasized, and procedure calls are incidental to the manipulation of abstract data. An illuminating example of programming with abstract data types is presented in reference 10.

7.2.3. Service Interfaces

In the service interface design method, a software system is viewed as a set of modules that perform services for one another. Emphasis is placed on identification of a set of service functions that will implement the system task. Three design strategies in the service interface category are: levels of abstraction, nucleus extension, and information hiding. As originally described by Dijkstra [8] levels of abstraction is a bottom-up design technique in which an operating system was designed as a layering

of hierarchical levels, starting with level 0 (processor allocation, real time clock interrupts) and building up to the level of processing independent user programs. Each level of abstraction is composed of a group of related functions, some of which are external (can be invoked by functions on higher levels of abstraction), and some of which are internal to the level. Internal functions can only be invoked by other functions on the same level and are used to perform tasks common to the work being performed on that level of abstraction. Each level of abstraction performs a set of services for the functions on the next higher level of abstraction. Thus, a file manipulation system might be layered as a set of routines to manipulate fields (bit vectors on level 0), a set of routines to manipulate records (sets of fields on level 1), and a set of routines to manipulate files (sets of records on level 2). Each level of abstraction has exclusive use of certain resources (I/O devices, data) that other levels are not permitted to access. Higher level functions can invoke functions on lower levels but lower level functions cannot invoke or in any way make use of higher level functions. The latter restriction is important because the lower level modules are self-sufficient for supporting abstractions up to their level; they can be used without change as the lower level routines in other applications, or in adaptations and modifications to an existing system. In addition, the strict hierarchical ordering of routines permits "intellectual manageability" of a complex software system [6].

A related design technique is the nucleus extension approach described by Hansen Brinch [11]. Using this approach, the basic nucleus of a software system is identified and implemented in such a way as to permit systematic extension of the nucleus to a complete system. In the case of an operating

system, the nucleus might consist of facilities to handle dynamic creation, control, and removal of processes, as well as communication between processes.

Although levels of abstraction and nucleus extension were developed specifically as bottom-up techniques for the design and implementation of operating systems, they are of much broader applicability. For instance, some software designers advocate a combined top-down and levels of abstraction approach to software design [2]. In this case, levels of abstraction provides a conceptual framework for the placement of modules within the top-down hierarchy.

The third service interface approach to be discussed is the "information hiding" technique described by Parnas [12, 13]. Using this technique, each module is chosen and designed to hide as much information as possible from the other modules in the system. This criterion not only provides a basic design strategy, but also provides a standard for elaboration and refinement of a design. Parnas observes that the approach results in designs that are functionally modular, and that have minimal coupling between modules. This in turn provides increased clarity of the design.

An interesting aspect of the information hiding approach is the use of a non-procedural specification language to describe the functional modules. In a well known example, Parnas illustrates the conventional control interface design of a KWIC index production system, and an unconventional design of the same system using the information hiding strategy [12]. The later design is clearly superior to the conventional design in terms of functional modularity. However, the information hiding approach tends to produce systems which require a great deal of switching between modules. Efficiency considerations dictate that the implementation of linkages between modules be accomplished by

techniques other than the traditional procedure call (which would impose a control interface implementation on a service interface design). Parnas suggests that functional modules be written as procedures to reinforce modularity at the source code level, but assembled by in-line compilation of code and by highly specialized linkage mechanisms, thus obscuring the modularity of the object code and improving the efficiency of the implementation.

7.3 Structured Design

A software design methodology called "Structured Design" or "Composite Design" has been described by Stevens, Meyers, and Constantine [14]. In structured design, the goal is to make coding, debugging, and modification easier, faster, and less expensive by reducing the complexity of the system. It is argued that the programmer cost is the largest factor in the cost of producing and maintaining a software system, and that reducing program complexity will increase programmer productivity. Of course, the techniques must be balanced with other constraints such as memory space and execution time required for the resulting programs. However, it is always easier to optimize a functionally correct program of straightforward design than it is to understand and debug an efficient but poorly designed and unreliable program.

The conceptual approach advocated in structured design is to configure the system so that the number and complexity of connections between modules is minimized. This is accomplished by minimizing the degree of coupling between modules and by maximizing the internal cohesion of each module. The strength of the coupling between two modules is influenced by several factors,

including: (1) the complexity of the interface, (2) the type of connection, and (3) the type of communication.

The complexity of an interface is a function of how much information is needed to state or understand the connection. Thus, obvious relationships result in lower coupling than obscure or inferred ones. For example, interfaces established by common control blocks, common data blocks, common overlay regions of memory, common I/O devices, and/or global variable names are more complex (more highly coupled) than interfaces established by parameter lists passed between modules.

The connections between modules may be established by referencing a module as a functional unit by name, which yields lower coupling than a connection which references internal elements of another module. In the latter case, the entire content of the referenced module may have to be taken into account when updating modules that refer to it. Modules that can be used without any knowledge of their internal details produce lower degrees of coupling.

The type of communication between modules includes passing of data, passing elements of control (such as flags, switches, labels, and procedure names), and modification of one module's code by another module. The degree of coupling is lowest for data communication, higher for control communication, and highest for modules that modify other modules.

Internal cohesion of a module is measured in terms of the strength of binding of elements within the module. Binding of elements occurs on a scale of weakest to strongest in the following order:

1) coincidental

2) logical

3) temporal

4) communicational

5) sequential

6) functional

Coincidental binding occurs when the elements within a module have no apparent relationship to one another. This results from "modularizing" an existing program into arbitrary modules, or from creating a module of unrelated instructions that appear several times in one or more modules.

Logical binding implies some relationship among the elements of the module; as for example, in a module that performs all input and output operations, or a module that edits all data. A logically bound module often tends to combine several related functions in a complex and inter-related fashion; resulting in the passing of unnecessary parameters, and in shared and tricky code which is difficult to understand or modify. For example, a module to edit all data might better be decomposed into four modules for editing master records, editing update records, editing addition records, and editing deletion records.

Modules with temporal binding tend to exhibit the same disadvantages as logically bound modules. However, they are higher on the scale of binding because all the elements are executed at one time, and no parameters or logic are required to determine which elements to execute.

The elements of a communicationally bound module are related by reference to the same set of input and/or output data. For example,

C.4

"print and punch the output file" is communicationally bound. Communicational binding is higher on the binding scale than temporal binding because the elements are executed at one time and also refer to the same data.

Sequential binding of elements occurs when the output of one element is in the input for the next element. For example, "read next transaction and update master file" is sequentially bound. Sequential binding is high on the binding scale because the module structure usually bears close resemblance to the problem structure. However, a sequentially bound module can contain several functions or part of a function since the procedural process in a program is often distinct from the function of the program.

Functional binding is the strongest, and hence most desirable, type of binding of elements in a module because all elements are related to the performance of a single function. Examples of function bound modules are "compute square root," "obtain random number," and "write record to output file."

A useful technique for determining whether a module is functionally bound is to write a single sentence describing the purpose of the module, and to perform the following tests on the sentence:

1. If the sentence has to be a compound sentence containing a comma or containing more than one verb, the module is probably performing more than one function. Therefore, it probably has sequential or communicational binding.
2. If a sentence contains words relating to time, (such as "first," "next," "then," "after," etc.), the module probably has sequential or temporal binding.

3. If the predicate of the sentence doesn't contain a single specific object following the verb, the module is probably logically bound; for example, Edit All Data has logical binding. Edit Source Data may have functional binding.
4. Words such as "initialize," "clean up," etc. imply temporal binding.

If the types of sentences described are unavoidable in a complete description of the module, then the module is probably not functionally bound.

The division of sub-functions into separate modules should be continued until each module contains no subset of elements that could be useful alone and until each module is small enough that its entire implementation can be grasped at once. It is suggested that the implementation of a module should require between 5 and 100 executable source statements. Weinberg has suggested that a group of about 30 statements is the upper limit that can be assimilated on first reading of a module [15]. In the initial design, one should subdivide too finely when in doubt because small functions can easily be recombined later, and duplicate functions may not be identified if the subdivision is too coarse.

The hierarchical tree structure depicted in Figure 2 is suggested as a general form that will usually result in the lowest cost implementation. The concepts of "scope of control" and "scope of effect" are useful aids for determining the relative positions of modules in a hierarchical framework. The "scope of control" of a module is that module plus all modules that are subordinate to the module. In Figure 2, the scope of control of module B

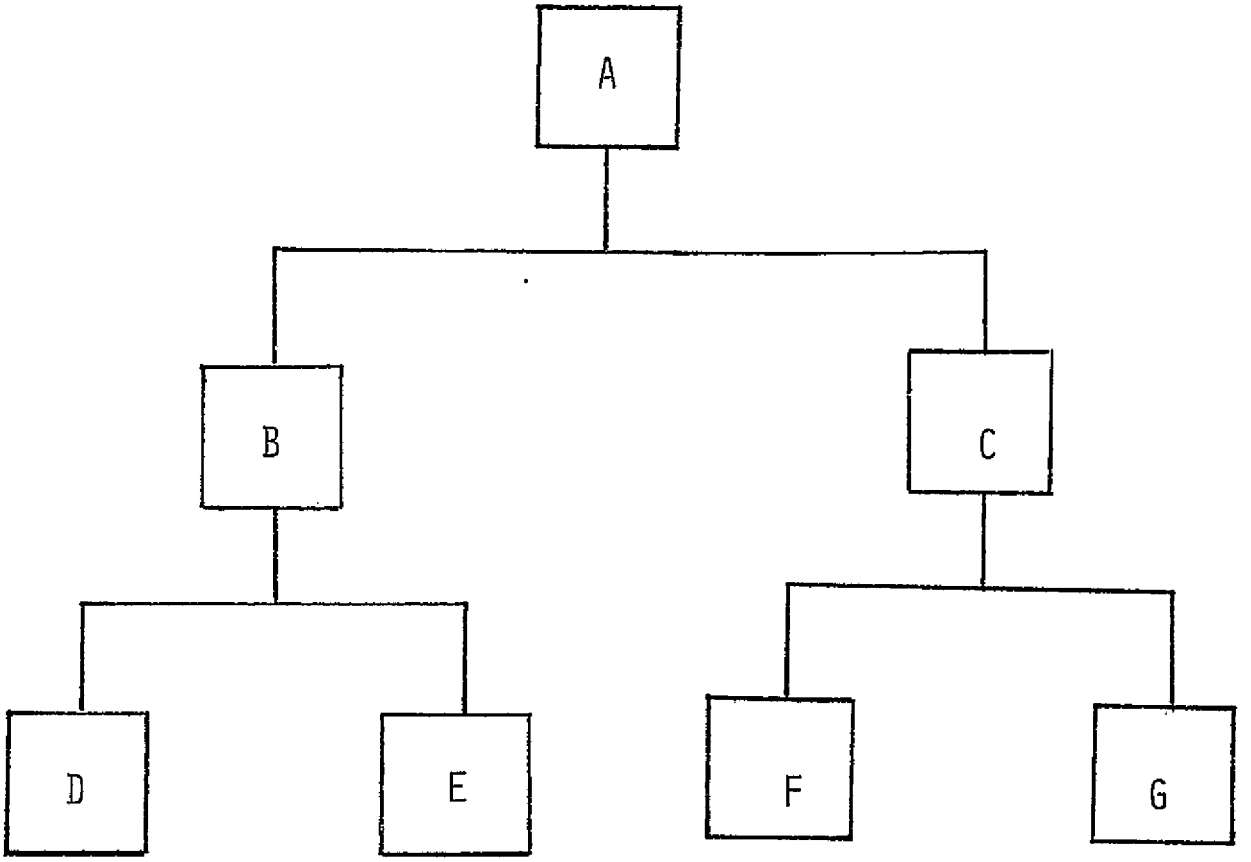


Figure 2. Hierarchical Software Structure

is B, D, and E. The "scope of effect" of a decision is the set of all modules that contain some code whose execution is based on the outcome of that decision. Systems are simpler when the scope of effect of a decision is within the scope of control of the module containing the decision. The following example illustrates the situation.

Referring to Figure 2, if the execution of some code in module A is dependent on the outcome of decision X in module B then either B will have to return a flag to A, or the decision will have to be repeated in A. The former approach results in added coding to implement the flag, and the latter results in duplicating some of B's function (decision X) in Module A. Duplicates of decision X result in difficulties of coordinating changes to both copies if the source code for decision X should be changed. In general, the scope of effect can be brought within the scope of control either by moving the decision element upward in the hierarchical structure, or by taking those modules that are in the scope of effect but not in the scope of control and moving them so that they fall within the scope of control.

7.4 Software Design Notation

In software design, as in mathematics, the notational scheme employed is of fundamental importance. Good notation can clarify the interrelationships and interactions of interest, while poor notation can complicate and interfere with good design practice. At least two levels of design specifications exist: general design specification describing the structure of the system (what functions, what interfaces); and detailed design specifications describing control flow and algorithmic considerations within the various modules. Some notational schemes are appropriate for stating

both general and detailed specifications while some are appropriate for one or the other. This section describes several notational schemes commonly used in software design, including Structure Charts, HIPOS, pseudo code, structured flowcharts, and decision tables.

7.4.1 Structure Charts

Structure Charts are useful during general program design as an aid in determining the functions, parameters and interfaces of the system. A structure chart differs from a flowchart in two ways: a structure chart has no decision boxes; and the sequential ordering of tasks inherent in a flowchart is suppressed in a structure chart. Figure 3 illustrates a flowchart and a structure chart for three modules; A which calls B which calls C. The structure chart emphasizes the connections between modules more clearly than does the flowchart. Thus, for example, it is obvious from the structure chart that module A is responsible for invoking module B. This, in turn, focuses attention on the parameters to be communicated between A and B.

The structure of a hierarchical system is often described using a structure chart as in Figure 4. The chart can be augmented with a module by module description of the input and output parameters. At the higher levels parameters are abstract; they become more concrete at the lower levels of the hierarchy. The lowest level routines deal with physical data objects as input and output parameters.

7.4.2 HIPOS

HIPO Diagrams (Hierarchy plus Input-Process-Output) were developed

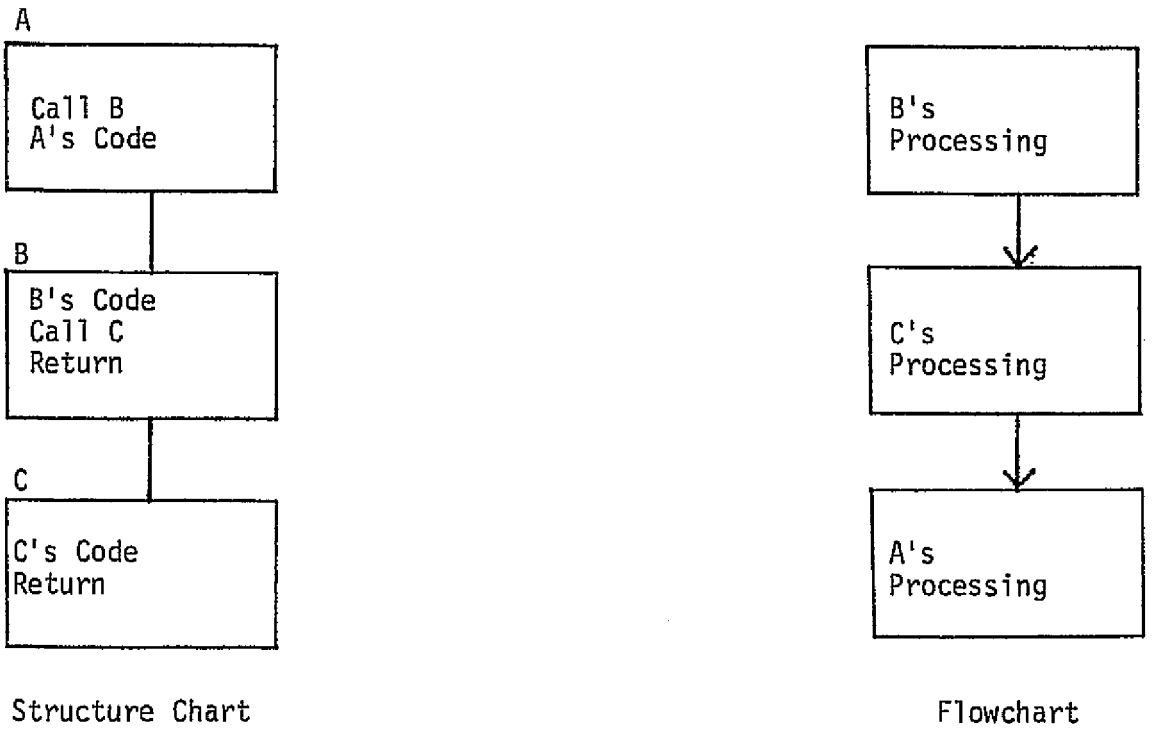


Figure 3. Structure Chart and Flowchart

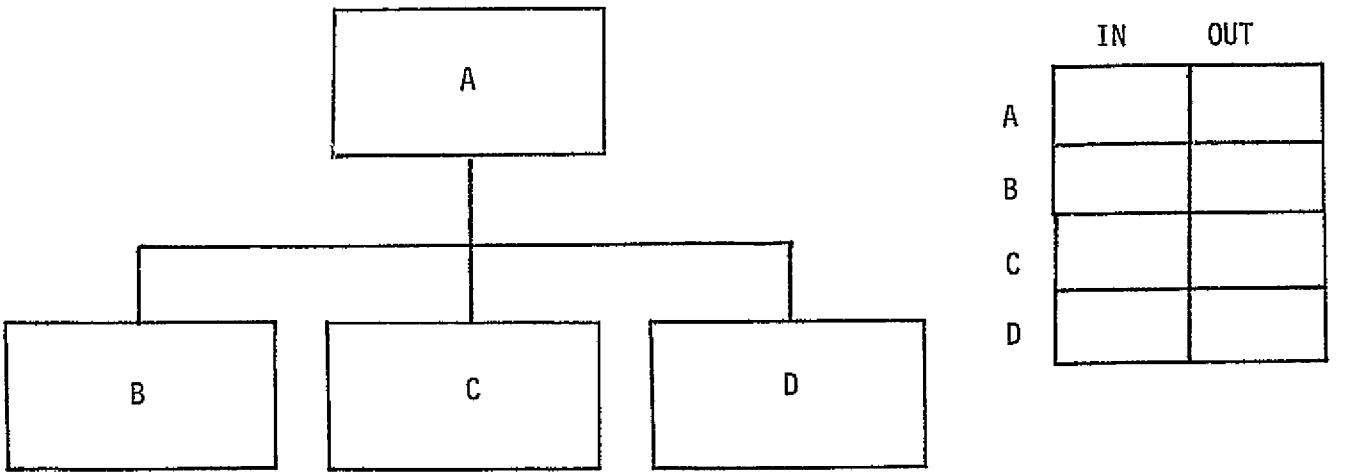


Figure 4. Structure Chart Augmented with Input/Output Table

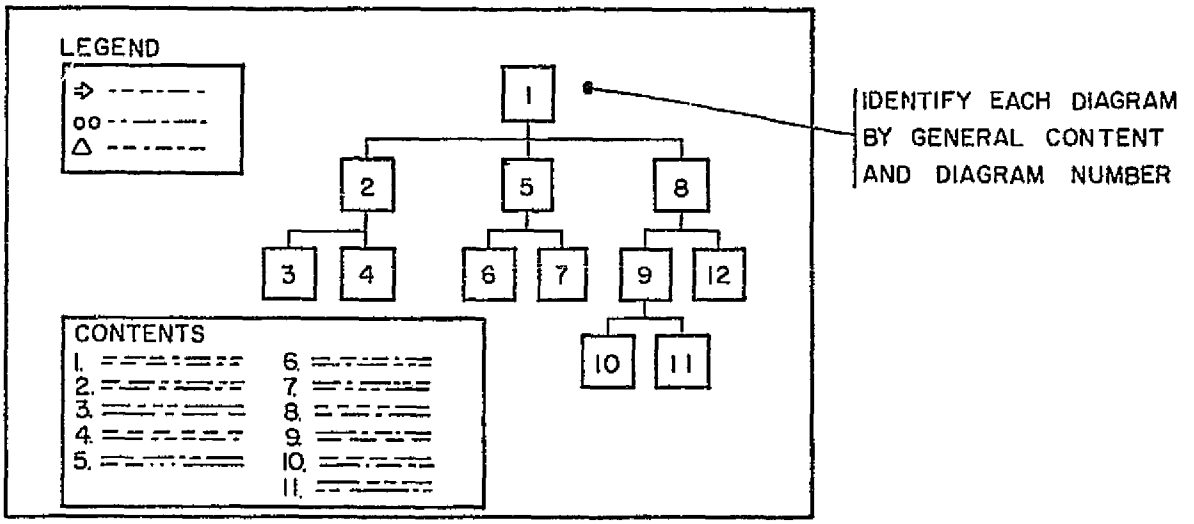
by IBM for use as tools in top-down software development, and as software documentation aids. HIPOS are formalized structure charts; as such they describe function and not internal flow control. A HIPO package comprises a set of diagrams that graphically describe the functional nature of a system proceeding from the general to the detailed level. Typically, a set of HIPO Diagrams consists of a Visual Table of Contents, Overview Diagrams, and Detail Diagrams. The Visual Table of Contents is a directory to the set of diagrams in the package; it consists of a structural overview diagram, a summary of the contents of each of the overview diagrams, and a legend of symbol definitions.

Overview Diagrams describe inputs, a process to support the function being described, and the results of the process. Each overview diagram may point to several subordinate detail diagrams, the exact number being a function of the process described. Typical formats for the Visual Table of Contents, Overview Diagrams, and Detail Diagrams are presented in Figures 5 and 6. HIPOS can be used as design tools and also as documentation aids; design specifications and documentation are thus in the same format, which facilitates comparison of the desired product and the actual product.

7.4.3 Pseudocode

Pseudocode notation can be used in both the general and the detail design phases. The designer describes the design using short concise English language statements that are structured by key words such as IF-THEN-ELSE-DO-WHILE and ENDDO. Key words and indentation describe the flow of control, while the English phrases describe the processing function.

VTOC



The overview and detail diagrams describe function. Each diagram shows:

- A process that supports the function being described.
- Results of the process.
- Necessary inputs.

Stated graphically:

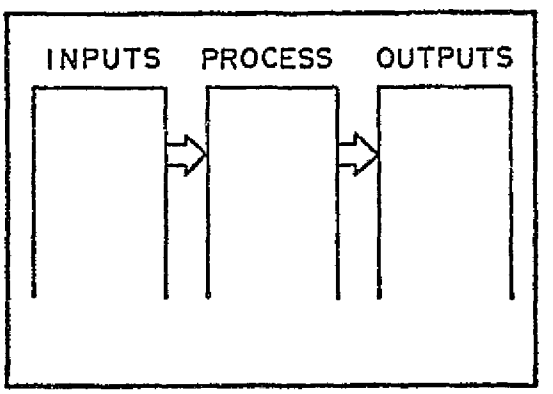
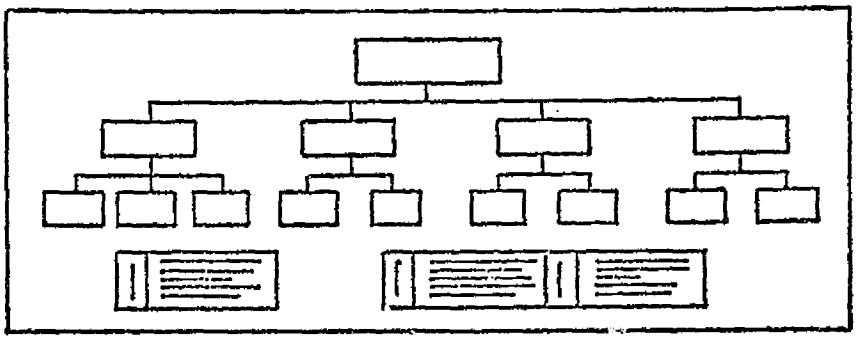
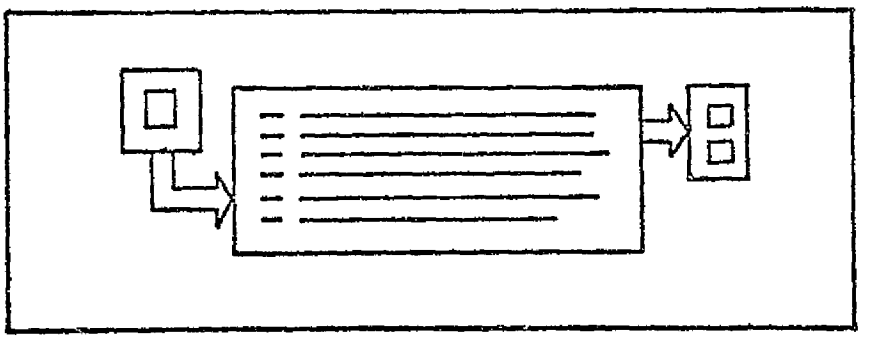


Figure 5. HIPO Diagram Formats

Visual Table of Contents



Overview Diagrams



Detail Diagrams

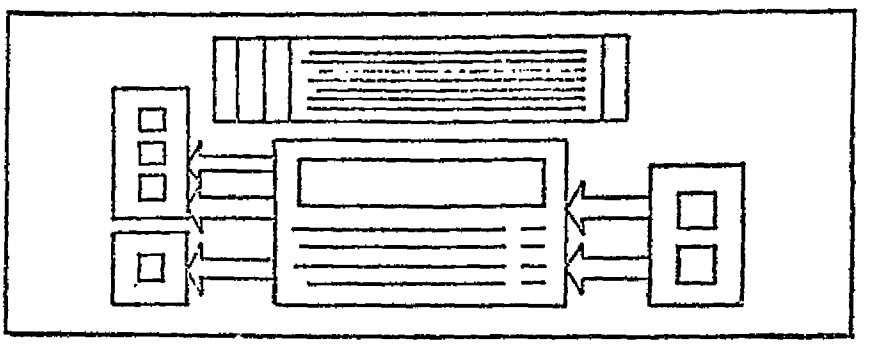


Figure 6. HIPO Diagram Formats

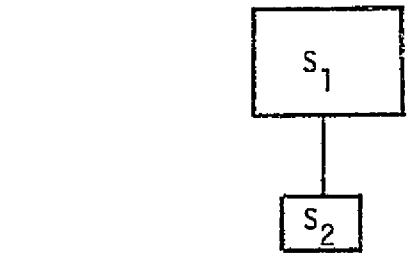
As an example of a pseudocode design specification, assume that a word frequency analysis program is to be described. The program will read a set of textual records and extract each individual word from each record. A table is to be constructed that contains each unique word found in the text, and a count of the number of times each word occurs. When all records have been processed, the contents of the table and other summary information is to be printed. The pseudocode code description of the word frequency analysis program might have the following form:

```
INITIALIZE THE PROGRAM
READ THE FIRST TEXT RECORD
DO WHILE THERE ARE MORE WORDS IN THE TEXT RECORD
    DO WHILE THERE ARE MORE WORDS IN THE TEXT RECORD
        EXTRACT THE NEXT TEXT WORD
        SEARCH THE WORD-TABLE FOR THE EXTRACTED WORD
        IF THE EXTRACTED WORD IS FOUND
            INCREMENT THE WORD'S OCCURRENCE COUNT
        ELSE
            INSERT THE EXTRACTED WORD INTO THE TABLE
        END IF
        INCREMENT THE WORDS-PROCESSED COUNT
    END DO AT THE END OF THE TEXT RECORD
    READ THE NEXT TEXT RECORD
END DO WHEN ALL TEXT RECORDS HAVE BEEN READ
PRINT THE TABLE AND SUMMARY INFORMATION
TERMINATE THE PROGRAM
```

In the top-down design strategy, each English phrase in the pseudocode can be expanded into a more detailed pseudocode description of that phrase. This can be continued until the design reaches the actual coding level. Pseudocode can be used to replace flowcharts, and to reduce the amount of external documentation required to describe the design.

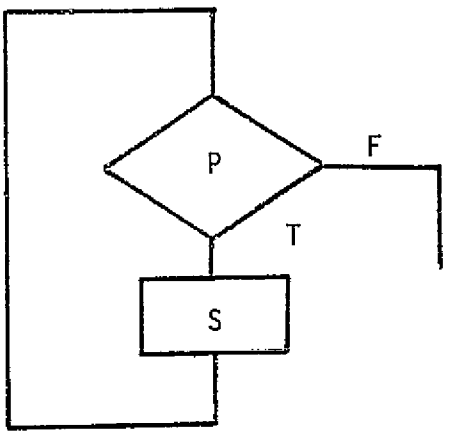
7.4.4 Structured Flowcharts

Flowcharts are the traditional means for specifying and documenting a software system designed along control interface lines. Typically, flowcharts incorporate rectangular boxes for actions, diamond shaped boxes for decisions, directed arcs for specifying inter-connections between boxes, and several other specially shaped symbols [16]. Structured flowcharts differ from traditional flowcharts in that structured flowcharts are restricted to compositions of certain basic forms. This makes the resulting flowchart the graphical equivalent of a textual pseudocode description. A typical set of basic forms and their pseudocode equivalents are illustrated in Figure 7. The basic forms are characterized by single entry into and single exit from the form. Thus forms can be nested within forms to any arbitrary level, and in any arbitrary fashion, so long as the single entry-single exit property is preserved. A composite structured flowchart and its pseudocode equivalent are illustrated in Figure 8. Because structured flowcharts are logically equivalent to pseudocode, they have the same expressive power as pseudocode. In particular, the single entry-single exit property allows hierarchical nesting of structured flow charts to describe a top-down design, starting with general design considerations and proceeding through detailed design. Structured flowcharts tend to place



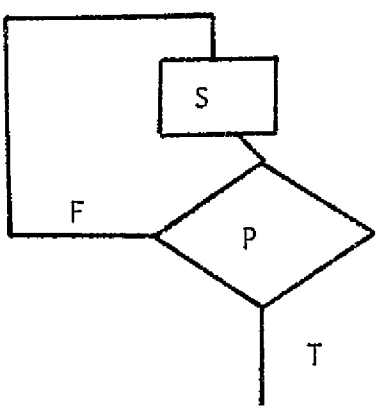
S_1

S_2



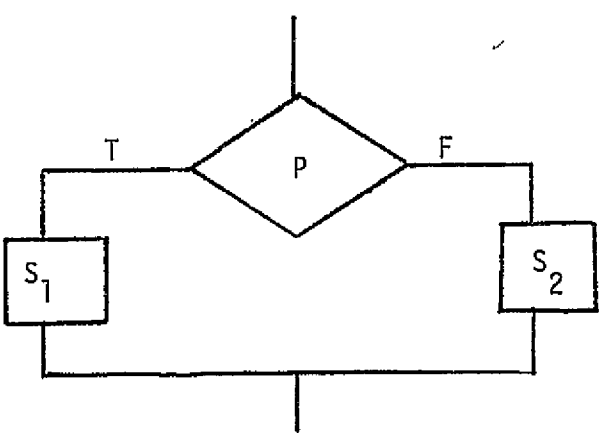
While P DO S

End



REPEAT S

UNTIL P



IF P THEN S_1

ELSE S_2

Figure 7. Equivalent Flowchart and Pseudocode Notations

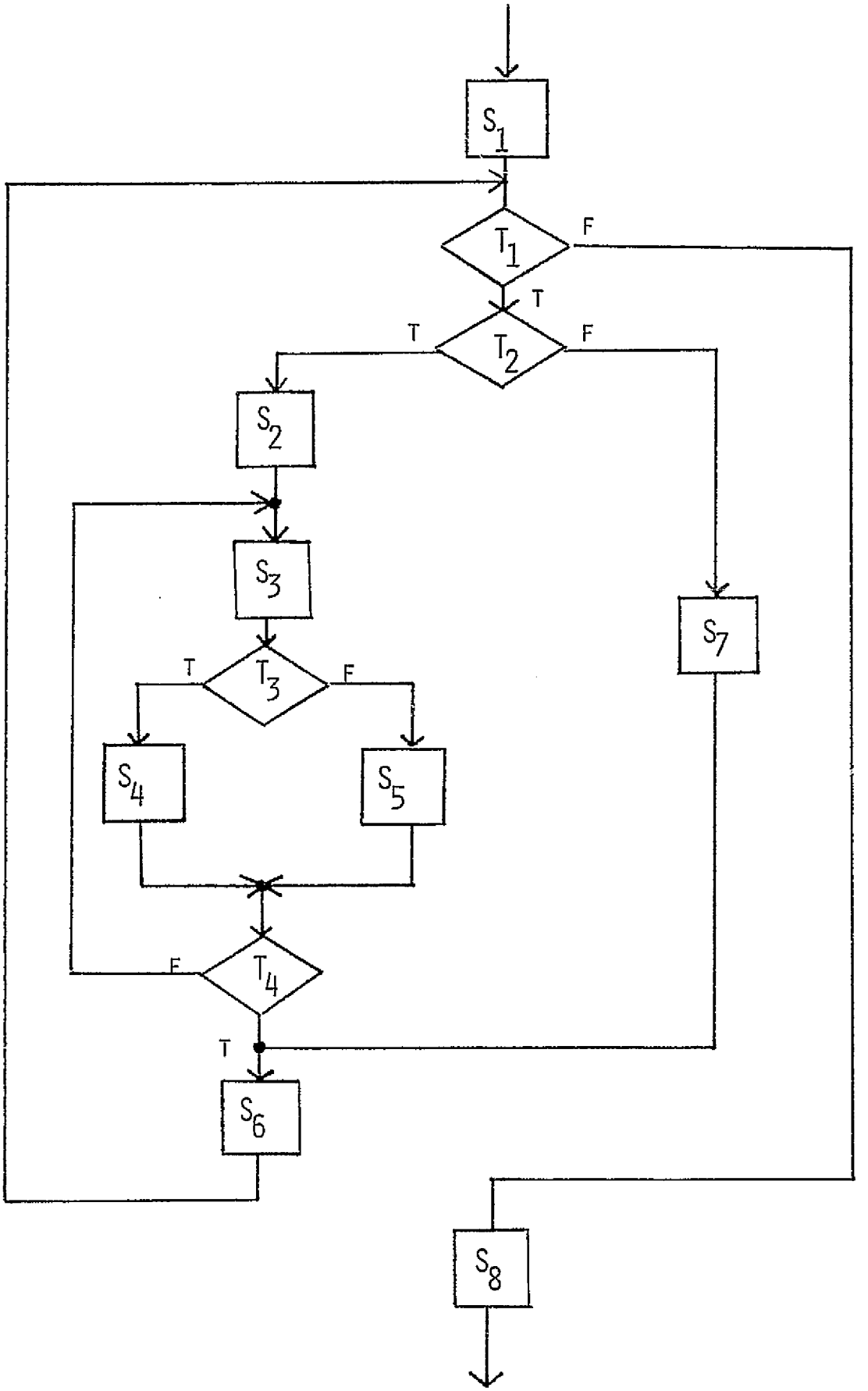


Figure 8a. Structured Flowchart

PSEUDO - CODE

```
S1
WHILE T1 DO
        IF T2 THEN S2
                REPEAT S3
                        IF T3 THEN S4
                                ELSE S5
                        END
                UNTIL T4
            ELSE S7
        END
        S6
END
S8
```

Figure 8b. Pseudocode Equivalent of Figure 8a.

increased emphasis on flow of control mechanisms due to the graphical nature of the visual image. They are thus appropriate when the decision mechanisms and sequencing of control flow are to be emphasized.

7.4.5 Decision Tables

Decision tables, like flowcharts, are useful for describing flow of control mechanisms. Decision tables are particularly useful when the flow of control is dependent on complex combinations of several conditions. In this case, flowcharts tend to become complex and difficult to follow.

A decision table consists of four quadrants called the condition stub, condition entry, action stub, and action entry. The condition stub occupies the upper left quadrant, and contains a list of the conditions to be examined. The condition entry is in the upper right quadrant of the table. The condition stub and condition entry specify the conditions to be tested. The lower left quadrant is called the action stub and contains a statement corresponding to each action that can result from the conditions described in the upper quadrants. The action entry section of the table (lower right quadrant) indicates which of the actions in the action stub are to be accomplished in response to a particular condition. A comprehensive discussion of decision tables is presented in Reference 17.

7.5 Influence of the Implementation Language

The implementation language provides a conceptual framework and a set of basic notions for the design of a software system. Thus, a LISP-like language based on recursive function theory and list structures will encourage the designer to formulate algorithms as recursive expressions operating on lists and binary trees, while FORTRAN will influence the designer in the direction of algorithms operating on arrays. Implementation of a software system is simplified when the data types, data structures, algorithms, and interfaces described in the design specification correspond to notions supported by the implementation language.

The conceptual aspects of a programming language are manifest in the data types, operators, and control structures of the language. Data types include basic and structured data types. The basic data types available in a programming language are typically a subset of the data types supported by the hardware, and may include any or all of: integers, floating point numbers, decimal numbers, characters, logical values, address pointers and bit vectors. Mechanisms for structuring basic data types include arrays, hierarchical structures, character strings, lists, trees, graphs, sequences, and sets. The ease of writing algorithms to transform and manipulate data of a given type (basic or structured) is determined by the operators provided for that data type. FORTRAN and ALGOL 60 provide arithmetic, relational, and logical operators for the basic data types of integer, floating point, and logical. A subroutine capability allows the user to implement abstract operators on basic and structured data. Newer languages, such as PL/1 and APL provide some built-in operators for the structured data types, and a subroutine capability for user definition of abstract operators.

The control structures in most programming languages reflect the basic architecture of sequential machines; in the absence of explicit control constructs execution proceeds from one statement to the next in lexicographic order. Explicit control statements include various forms of conditional statements, iteration mechanisms, and subprogram calls and returns. Conditional statements include the arithmetic and logical IF of FORTRAN, nested IF-THEN-ELSE statements in ALGOL 60, success and failure exits in SNOBOL statements, and the COND expression of LISP. Iteration mechanisms include looping statements (such as the FORTRAN and PL/1 DO statement, and the ALGOL 60 and PASCAL FOR statement) in which initialization, incrementing and exit testing is described in the statement forms, as well as loops constructed using IF statements and GOTOs in which initialization, incrementing, and testing is handled explicitly in the source text.

The current interest in structured programming is motivated by the desire to provide control constructs that preserve the clarity of the design specifications in the source code implementation of the design. Basic premises underlying the use of structured control mechanisms are: 1) each basic construct must preserve the single-entry/single-exit property, and 2) no basic construct permits backward transfers of control in the source text (except the implicit transfers in looping constructs). Single-entry/single-exit control structures can be hierarchically nested within other control structures to any arbitrary depth, and indentation of nesting levels facilitates readability of the source code. When the assumptions of nested single-entry/single exit constructs and no backward transfers are satisfied, the dynamic execution flow through the program can be made

to correspond closely to the static structure of the source text. The source text is thus highly readable and makes a significant contribution to its own documentation.

The use of GOTO statements has been criticized by structured programming advocates because the GOTO provides an unrestricted mechanism for structuring control flow, and it is quite easy to (intentionally or unintentionally) violate the basic premises of structured programming using GO TO's. Elson lists the following positive benefits to programming without GO TO's [18]:

1) The programmer is forced to discipline his thought processes, to formulate his logic according to an appropriate structure.
2) The programmer finds himself looking for similarities rather than differences in sub-portions of this problem. Rather than simply generating conditional branches to many program locations to handle a number of cases, he is encouraged to handle them together perhaps with additional use of variables serving as parameters to differentiate between the cases.
3) The reader or inheritor of a program has a much easier time following program logic if he can read the program sequentially rather than with constant page flipping through the program listing.

Much of the current research in structured programming is concerned with development of control structures that reflect the system design in the source code without unduly restricting the programmer or forcing him into unnatural modes of expression.

In addition to selecting a language appropriate to the application, numerous implementation details will influence the quality of the software

produced. The implementation features of interest may include:

1) Compile time and/or execution time efficiency
2) Memory space utilization
3) Adequacy of error messages
4) Debugging options
5) Adherence to formal standards
6) Stability of the implementation

The stability of an implementation is revealed by answers to a series of questions such as: who maintains the implementation? at what level of support? when was the last update released? how long has the language been installed? how often is it used? what are the experiences of other users?

Documentation should be clear, concise, and well-structured. The documentation should include an introductory users manual, an authoritative reference manual, and explanations of machine dependent implementation features. All documents should be cross-referenced and indexed to permit rapid access to any desired level of detail. The implementation of ambiguous, incomplete, and inconsistent features should be noted, along with extensions to, and sub-setting of the language.

7.6 Summary

This chapter has discussed various aspects of the software design process, including methodical approaches to software design, notational schemes for describing the design, and the influence of the programming language on the design process.

Design approaches discussed included top-down and bottom-up design; successive refinement; integrated top-down design, coding, and testing; programming by action clusters; levels of abstraction; nucleus extension; information hiding; and structured design.

Notational schemes discussed included structure charts, HIPOS, pseudocode, structured flowcharts, and decision tables. Various language concepts and their influence on the software design process were mentioned.

The theme of this chapter has been that high quality software can only be achieved by thoughtful design and implementation, and that design methodologies and notations do exist to facilitate the production of high quality software. All of the techniques described have relative strengths and weaknesses which make them more or less appropriate in particular circumstances. No single technique is clearly superior to all others in all situations.

The tragedy of sloppy system design and poor quality software is not due to the lack of notational schemes and design techniques, but rather is largely due to our failure to apply and experiment with the existing methodologies.

7.7 Bibliography

1. Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973.

2. Liskov, B.H., "A Design Methodology for Reliable Software Systems," FJCC Proceedings, 1972.

3. Wirth, N., "Systematic Programming: An Introduction," Chapter 15, Prentice-Hall, 1973.

4. Wirth, N., "Systematic Programming: An Introduction," Prentice-Hall, 1973.

5. Wirth, N., "Program Development by Stepwise Refinement," CACM, Volume 14, No. 4, April 1971.

6. Dahl, O.J., Dijkstra, E.W. and C.A.R. Hoare, "Structured Programming," Academic Press, 1972.

7. McGowan, C.L. and J.R. Kelly," Top-Down Structured Programming Techniques," Petrocelli/Charter, 1975.

8. Dijkstra, E.W., "The Structure of the 'THE' Multiprogramming System," CACM, Vol. 11, No. 5, 1968.

9. Liskov, B. and S. Zilles, "Programming with Abstract Data Types," ACM SIGPLAN Notices, Vol. 9, No. 4, 1974.

10. Palme, J., "SIMULA as a tool for Extensible Program Products," ACM SIGPLAN Notices, Vol. 9, No. 2, 1974.

11. Brinch, Hansen, P., "The Nucleus of a Multiprogramming System," CACM, Vol. 13, No. 4, 1974.

12. Parnas, D.L., "A Technique for Software Module Specification with Examples," CACM Vol. 15, No. 5, 1972.

13. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM Vol. 15, No. 12, 1972.

14. Stevens, W.P., G.J. Myers, and L. L. Constantine, "Structured Design," IBM Systems Journal, No. 2, 1974.

15. Weinberg, G.M., "The Psychology of Computer Programming," Van Nostrand Reinhold, 1971.

16. Chapin, N. "Flowcharting with ANSI Standard: A Tutorial," ACM Computing Surveys, Vol. 2, No. 2, 1970.

17. Pooch, U.W., "Translation of Decision Tables," ACM Computing Surveys, Vol. 6, No. 2, 1974.

18. Elson, M., "Concepts of Programming Languages," Science Research Associates, Inc., 1973.

8.0 FUTURE EXTENSIONS

The DOMONIC system is operational and is being used in a number of software projects at Texas A&M University. The latest version of DOMONIC was installed at NASA, Greenbelt, in June 1975. The system presently consists of 315 separate modules and requires 290K bytes of memory. The DOMONIC system contains an editor which can be used to edit source programs as well as all forms of documentation. The DOMONIC system has been optimized to interactively edit documentation while requiring the voluminous documentation to be produced off-line. At present, the compilation process can be initiated from an interactive terminal but must run batch. The DOMONIC system should be modified to allow the user to compile and execute programs in time sharing mode. The actual compilation and execution process should be performed under appropriate systems such as TSO on the IBM 360/370. Other system improvements such as improved garbage collection routines and appropriate utilities for transferring documentation units from disk to tape should be developed.

Many aspects of program activity, project status, etc. can be monitored through the monitor points made available within DOMONIC. Effectiveness of a monitor requires more than designing monitoring capabilities into the system. The monitor data must be captured in a form that can be digested and reported in documents that are meaningful to management. Currently, the DOMONIC system is monitoring resource utilization within the DOMONIC system. Some of this data is useful for driving the reliability model developed as part of the DOMONIC project. A monitor should be extended to automatically

record and report information useful in validating software reliability models. Two reliability models (a completion model and an acceptance model) were developed as part of this project and were described in detail in Chapter III. The DOMONIC system should be used to gather extensive data to validate the above models as well as other reliability measurement algorithms described in the literature.

While the DOMONIC system has a number of existing documentation aids, additional documentation aids should be developed for use with the system. A generalized graphics documenting system should be incorporated in DOMONIC. This system should have the ability to draw flowcharts, hierarchical diagrams, overlay maps and HIPO charts.