

AD-A046 588

PATTERN ANALYSIS AND RECOGNITION CORP ROME N Y

F/G 6/4

PATTERN RECOGNITION METHODS FOR DETERMINING SOFTWARE QUALITY.(U)

OCT 77 T L MCGIBBON, H M HERSH, J M MORRIS

F30602-76-C-0214

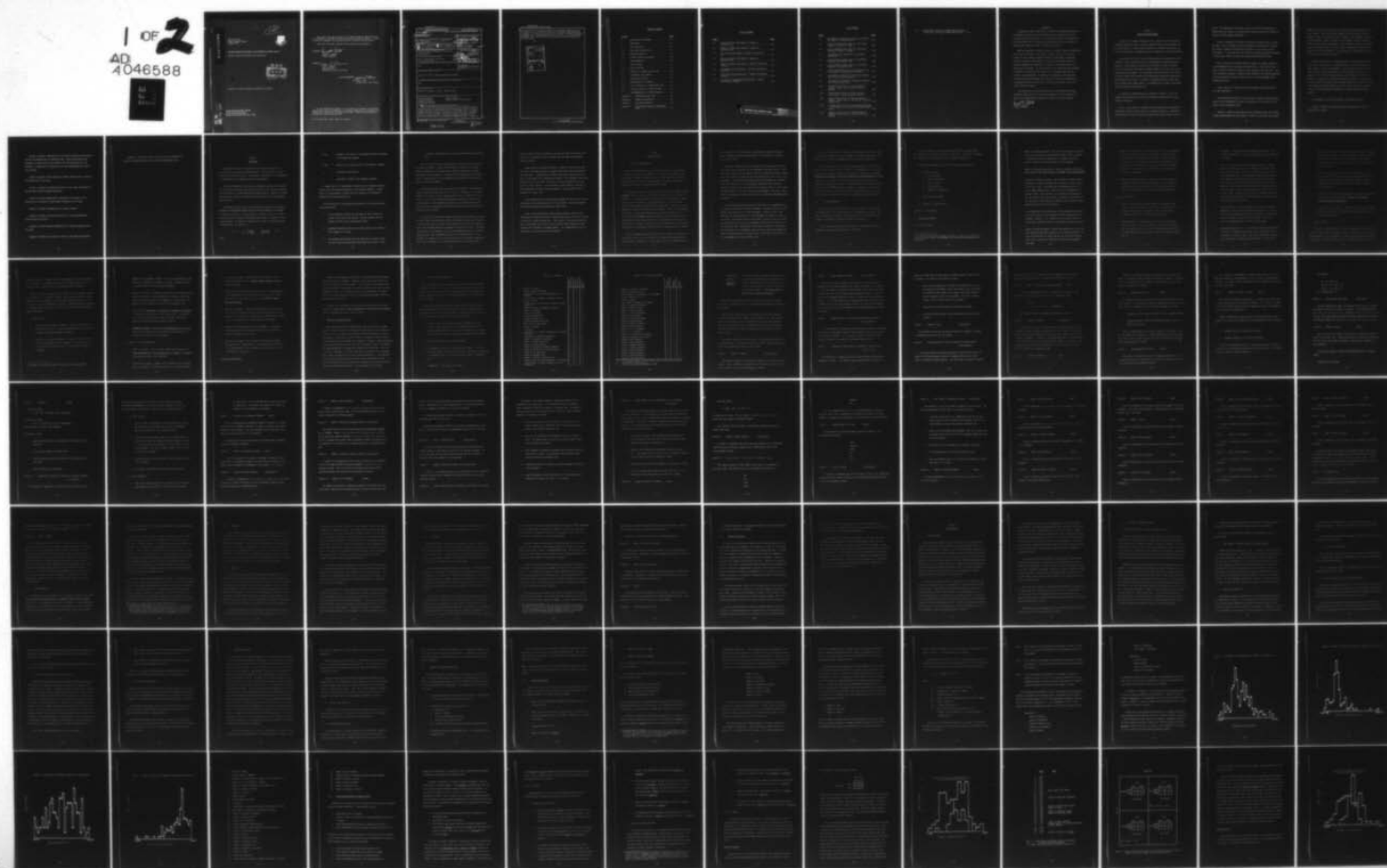
UNCLASSIFIED

PAR-77-13

RADC-TR-77-325

NL

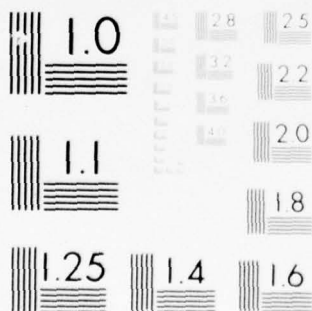
1 OF 2
AD
4046588



OF



046588



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

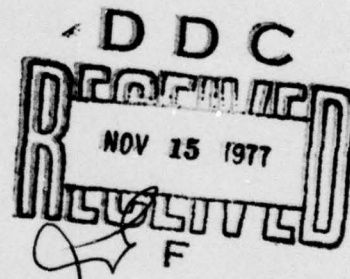
AD No. _____
DDC FILE COPY

AD A046588

RADC-TR-77-325
Final Technical Report
October 1977



PATTERN RECOGNITION METHODS FOR DETERMINING SOFTWARE QUALITY
Pattern Analysis and Recognition Corporation



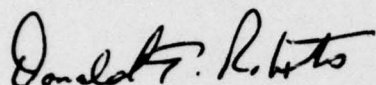
Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

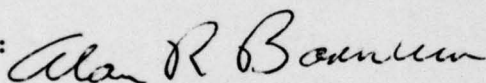
This report has been reviewed and is approved for publication.

APPROVED:



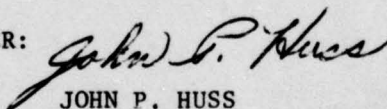
DONALD F. ROBERTS
Project Engineer

APPROVED:



ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER:



JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 <u>RADC-TR-77-325</u>	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 <u>PATTERN RECOGNITION METHODS FOR DETERMINING SOFTWARE QUALITY</u>	5. TYPE OF REPORT & PERIOD COVERED 9 <u>Final Technical Report</u>	6. PERFORMING ORG. REPORT NUMBER 14 <u>PAR-877-13</u>
7. AUTHOR(s) 10 <u>Thomas L. McGibbon, David A. Bennett</u> <u>Harry M. Hersh, Christopher Landauer</u> <u>John M. Morris</u>	8. CONTRACT OR GRANT NUMBER(s) 15 <u>F30602-76-C-0214</u>	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Pattern Analysis and Recognition Corporation 228 Liberty Plaza Rome NY 13440	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 16 <u>62702F</u> <u>5581410</u> 17 <u>14</u>	
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441	12. REPORT DATE 11 <u>Oct 77</u>	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	13. NUMBER OF PAGES <u>12</u> 18 <u>187p.</u>	
15. SECURITY CLASS. (of this report) UNCLASSIFIED	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Donald F. Roberts (ISIS)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Pattern Recognition Quality Software Feature Extraction Computer Program Characteristics Automatic Classifiers Programming Constructs		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The On-Line Pattern Analysis and Recognition System (OLPARS) was demonstrated as a tool for evaluating program characteristics which contribute to program readability, freedom from errors, and development time. Structural features were extracted automatically from a data base of 155 PL/I programs and used as inputs to OLPARS. As expected, program length had a dominant effect on the time required to understand programs; additional features affecting understandability included GOTO and RETURN statements. Significant		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

390101

Jmac

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

contributions of this study were methods for estimating program error rates from archival data, and reliable techniques for estimating understandability of programs. An interesting by-product was the ability to identify individual programmer styles. The principal outcome of the study was the demonstration that OLPARS could provide a facility for evaluating factors contributing to software quality.

ACCESSION for	
NTIS	White Section <input checked="" type="checkbox"/>
DDC	B.H. Section <input type="checkbox"/>
UNANNOUNCED	<input type="checkbox"/>
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist.	SPECIAL
A	

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

TABLE OF CONTENTS

<u>Section</u>	<u>Page</u>
1. Introduction and Summary	1-1
2. Background	2-1
3. Data Collection.	3-1
3.1. Data Base Characteristics.	3-1
3.2. Feature Definition	3-3
3.3. Physical Feature Extraction.	3-13
3.4. Understandbilty	3-37
4. Data Analysis.	4-1
4.1. OLPARS Overview.	4-1
4.2. Analysis of Program Quality Data	4-10
5. Substantive Conclusions.	5-1
5.1. Analysis of Results.	5-1
6. Methodological Conclusions	6-1
6.1. Use of OLPARS as a Classification Tool	6-1
6.2. Discrete Classes or Continuous Data.	6-4
7. Recommendations for Further Research	7-1
Appendix A Understandability Study	A-1
Appendix B Program Documentation	B-1
Appendix C Data Base Generation.	C-1
Appendix D Classification Based on Programming Style	D-1

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
4-1	Development Time after Number of Lexemes is Partialled Out	4-18
4-2	Number of Changes after Number of Lexemes is Partialled Out	4-19
4-3	Mean Rating after Number of Lexemes is Partialled Out.	4-20
4-4	Mean Log Latency after Number of Lexemes is Partialled Out	4-21
4-5	Number of Changes (Partialled) - Optimal Discriminant Direction.	4-29
4-6	Development Time (Partialled) - Optimal Discriminant Direction.	4-35
4-7	Difficulty Rating (Partialled) - Optimal Discriminant Direction.	4-41
4-8	Log Time for Understanding (Partialled) - Optimal Discriminant Direction	4-42

PRECEDING PAGE BLANK-NOT FILMED

LIST OF TABLES

<u>Table</u>		<u>Page</u>
4-1	Discriminant Coefficients (Partialled Features) for Number of Changes Analysis.	4-30
4-2	Cross Validated Fisher Logic for Classifying Programs by Number of Changes	4-31
4-3	Cross Validated Fisher Logic for Classifying Programs by Number of Changes	4-33
4-4	Development Time (Partialled) Discriminant Coefficients.	4-36
4-5	Cross Validated Fisher Logic for Classifying Programs by Development Time.	4-37
4-6	Cross Validated Fisher Logic for Classifying Programs by Development Time.	4-38
4-7	Fisher Discriminant Logic of Ratings and Laten- cies Using All 32 (Partialled) Features	4-40
4-8	Discriminant Coefficients (Partialled Features) for Difficulty Rating Analysis.	4-43
4-9	Discriminant Coefficients (Partialled Features) for Log Latency Analysis.	4-44
4-10	Classification Tables for Understandability Ratings - Fisher Logic on Nine Partialled Features.	4-46
4-11	Classification Tables for Study Latencies - Fisher Logic on Nine Partialled Features. . . .	4-47
4-12	Classification Tables for Understandability Ratings - Fisher Logic on Ten Features Including Program Size.	4-48
4-13	Classification Tables for Understanding Latencies - Fisher Logic on Ten Features Including Program Size.	4-50
4-14	Classification Tables for Understandability Ratings - Fisher Logic on Single Program Size Feature	4-51

4-15

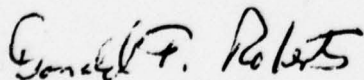
Classification Tables for Understanding Latencies
- Fisher Logic on Single Program Size Features. . 4-52

EVALUATION

Attaining the goals of RADC TPO-5, Software Cost Reduction requires the development of methods for producing and measuring software that is both reliable and easily maintained. This includes development of standards for writing software that is reliable and maintainable as well as methods for measuring the software in order to predict its quality.

RADC has undertaken a number of efforts to develop models for predicting the quality of software. This effort is unique in that it is the first attempt at using classical pattern recognition technology for analyzing software. The goal of the effort is to identify structural features of computer programs that contribute in a negative or positive sense to the quality of the software. If such features can be identified, then automatic classifiers for predicting the quality of the software, based on these features, can be designed. In addition, the analysis of the structural features using pattern recognition techniques, is useful in identifying programming constructs and practices that should be avoided in order to produce quality software.

For the study, the On-Line Pattern Analysis and Recognition System (OLPARS) at RADC was used. The study demonstrated that such systems are useful for performing rapid analysis of software structural features.


DONALD F. Roberts
Project Engineer

SECTION 1

INTRODUCTION AND SUMMARY

The goal of the research described in this report was the development of effective algorithms, based on pattern recognition theory and technology, for the identification of features which contribute to software reliability, and for the classification of programs into such categories as reliable/unreliable.

It was shown that RADC's On-Line Pattern Analysis and Recognition System (OLPARS) provides an effective tool for the analysis of program structural features. It was possible through the use of OLPARS to apply statistical tests to determine the relevance of program features in the data base to the classification of programs. It was possible to replicate some of the results of earlier studies very rapidly, since OLPARS made an extensive repertoire of tools available to the user. A number of interesting further results were also obtained, which added support to recommendations for modern programming practices developed recently by RADC.

In addition to demonstrating the usefulness of OLPARS as a tool for program structure analysis, this study included the development of several research techniques.

First, a large number of program structural features were identified as potentially relevant to the prediction of program reliability. Algorithms for extraction of these features were prepared and used to provide inputs to

OLPARS. The feasibility of automatic feature extraction from programs was demonstrated, and several of the structural features proved to be relevant to factors affecting program reliability.

Second, a technique for automatic derivation of program error rates was developed. Such a technique obtains an estimate of error rates by counting the number of changes introduced into programs over the course of their development. It was shown that this technique can produce automatic estimates of error rates, without reliance upon manually-produced report forms.

Third, a method for obtaining reliable estimates of program readability or understandability was developed and demonstrated. Human subjects were asked to read and respond to questions about programs in the data base. Both their subjective estimates of program understandability and objective measures of the time required and their scores on tests of their knowledge of the programs were used.

Fourth, methods for obtaining program development time from archival data were implemented.

One interesting by-product of the study was a scheme for discriminating among the four programmers who wrote the programs in the data base, on the basis of their programming style.

Finally, a number of substantive results concerning factors which affect program understandability were obtained. Briefly, it was shown that program

length was the most critical factor in determining the understandability of these programs. Other factors which appear to have some effect, when the effect of length is held constant, are: number of GOTO statements, number of RETURN statements, number of operators per assignment statement, number of variables used, number of parameters in call statements, number of externally called subroutines, number of labels, number of assignment statements, and number of complex ELSE clauses. A full discussion of substantive results of this research is included in Section 5.

It should be noted that the data base used for these experiments was quite small, consisting of 155 programs written in the PL/I language for a single system by just four programmers. For these reasons, results of the study cannot be applied uncritically to other software developments. The emphasis of the study has rather been on a demonstration of the value of OLPARS in performing rapid, effective statistical analyses of program data. As more extensive data bases become available, OLPARS will provide a tool for investigating additional languages and other program features to provide a body of reliable information concerning the factors that affect program quality.

The remainder of this report includes the following sections:

Section 2 provides background information concerning the goals and methods of the project.

Section 3 contains a description of the features extracted from the data base and the methods used for extracting them. Many features which were proposed, but which could not be extracted from the given data base, are described. A description of methods used in the understandability study is also provided.

Section 4 provides a brief overview of OLPARS, together with a review of the methods used in the study.

Section 5 includes the substantive results of the study, indicating the features that affected program reliability.

Section 6 contains methodological conclusions of the study, with a description of the manner in which OLPARS contributed to the study.

Section 7 includes recommendations for further research.

Appendix A contains a detailed description of the understandability study reported in Section 3.

Appendix B provides program documentation for software produced during the study.

Appendix C describes the routines by which the data base was generated.

Appendix D contains the results of the study which automatically classified programs according to the author's programming style.

SECTION 2

BACKGROUND

During the recent past, several models have been developed for the prediction and evaluation of software quality. In this section, one such model will be briefly described. In addition, a discussion of the background and motivation for the present project will be included.

Schick and Wolverton [9] evaluated a statistical analysis done by Hatter [3] of software reliability data collected by TRW in 1971, during the development and operation of large operational software systems. Whenever a software deficiency was detected, a software problem report (SPR) was issued against that deficiency.

The method used was a least-squares regression analysis for the establishment of relationships among variables which were thought to contribute to software reliability. In this analysis, a surface of best fit is determined for any set of linear relationships between data. A goodness of fit is then determined. The measure of goodness of fit used is the adjusted index of determination, as defined by

$$r^2 = 1 - \left(1 - \frac{\sigma^2_{\text{est}}}{\sigma^2_{\text{obs}}} \right) \left(\frac{n - 1}{n - T} \right) \quad (1)$$

where:

σ^2_{est} = variance of the values of the dependent variable predicted by the regression equation

σ^2_{obs} = variance of the actual values of the dependent variable

n = the number of data points

T = the number of terms in the regression equation

A large value of r^2 (approaching 1) means that the estimating equation accounts well for observed variations in the dependent variable. A small value of r^2 (approaching 0) implies that the variance of the dependent variable has not been accounted for.

From the results of the regression calculations, the following observations were derived:

1. The only reliable predictor of the number of SPR's charged to a routine is the size of the routine. However, greater than 50% variance in SPR's is left unexplained by size.
2. Programmer experience seems to have little effect on the number of SPR's charged to a routine.
3. All possible relationships among the data were not analyzed (only the most intuitively appealing relationships were tested). Thus,

it cannot be concluded that no useful relationships exist among the data.

This Schick and Wolverton study represents a type of research which can be assisted by OLPARS. Specific factors which are believed to influence program performance can be tested, and a model for prediction of program reliability can be developed. In addition, factors influencing other components of program quality, such as readability, development time, and error rates, can be quickly tested.

The experiments which will be reported here represent a continuation of studies begun during the summer of 1973, near the beginning of a new implementation of OLPARS in the PL/I language under Multics (Contract #F30602-73-C-0351). At that time, structured programming concepts were new and relatively untested; it was therefore proposed that the OLPARS implementation utilize structured programming techniques and evaluate their effectiveness in improving program quality.

The emphasis in the earlier research was on the use of structured coding, in which control structures were limited to SEQUENCE, DOWHILE, and IFTHENELSE. (A preprocessor made it possible to use DOUNTIL and CASE, but the programmers found these extended structures unnecessary and did not use them.) An experimental design was established, in which two programmers would use structured coding, and two other programmers would use non-structured forms. As it happened, the enthusiasm of all of the programmers for structured programming meant that the majority of the programs were written in structured form.

However, several programs, including a few that had been written before the start of the evaluation project, included GOTO and other non-structured control statements.

During the Multics OLPARS development, a number of error report forms were used to document the types of program errors which were encountered by the programmers. Although these provided a good deal of qualitative data, they did not appear to be statistically reliable. The programmers were often working alone, at late hours, and there was little motivation for keeping records of their mistakes. For this reason, a more mechanical approach to the determination of error rates was required. Such an approach will be described in Section 3.1.1.

In the experiments to be reported here, OLPARS was both the tool for the evaluation, and the system which was to be evaluated. The actual programs which OLPARS comprises were analyzed by OLPARS itself.

Later, as more substantial systems become available, together with statistically reliable error report forms and other data, it will be possible to extend the studies reported here. The purpose of this initial study has been to demonstrate that OLPARS could be used for rapid evaluation of the features that contribute to program quality. As a demonstration of such a capability, it has been completely successful.

SECTION 3

DATA COLLECTION

3.1. DATA BASE CHARACTERISTICS

This section describes some of the characteristics of the data base of programs to be analyzed by OLPARS. The analysis was performed on the 260 programs that make up OLPARS itself, since the data base used for this study consisted of the PL/I programs developed for RADC as the Multics OLPARS Operating System (Contract #30602-75-C-0226).

The data base consists of chronological versions of the OLPARS programs. **Details** of the procedure used in creating the data base can be found in Appendix C. The average number of unique copies of each program is approximately 4, with a minimum of 1 and a maximum of 16. The 260 programs represent three years of work (June 1973 through May 1976) by the four programmers of the Multics OLPARS system development. Each of the programmers had roughly the same programming experience (three were directly out of college with some FORTRAN programming experience, and one came from a non-programming-related teaching job), each had a mathematics-related college background (i.e., physics, computer science, and mathematics), and each had no previous PL/I programming experience prior to working on Multics OLPARS.

From a programmer's viewpoint, OLPARS can be viewed as an interactive pattern recognition system requiring certain graphic, file manipulation, and mathematical capabilities. Each program written for OLPARS performed

its own graphic, file manipulation, and mathematical operations when required, although subprograms either existed or were written to simplify each of these tasks.

The Multics OLPARS project was organized in such a way that there was one project leader, with each of the programmers reporting directly to him. The project leader, with the support of pattern recognition analysts, defined each task that would be required, and then assigned each task to one or more programmers. The programmers designed, wrote, and debugged the programs to their own satisfaction. If the programmer experienced any problems in writing his program, help was sought from the project leader or other related company personnel. After the program had been debugged, the project leader tested the programs for further errors, and the errors were corrected by the programmer.

The programming environment was such that each of the programmers was in constant contact with the other programmers. Programs written by one programmer were usually read by other programmers. Because of this cross-checking, the programming style of one programmer was frequently copied by the others. The original design of the project called for the use of structured program code by two of the programmers, while the others were to use non-structured code. Nevertheless, after about two months of programming, the enthusiasm of the programmers using structured code had encouraged the others to begin writing in a similar style. By the end of the project, all the programmers were using structured code.

In spite of this sharing of styles, however, there was sufficient difference among the programmers to permit OLPARS to discriminate among them. The results of this study of programming styles are reported in Appendix D.

The use of this data base to evaluate program quality had several drawbacks. Certainly the structural features of the programs could be evaluated, for they were contained in the programs themselves. Other, more transient, data were not available or were only available as rough approximations to the actual data. For example, information such as error reports and actual development time were not available. Approximations to these variables were derived from the Multics archival information. Although the data base was deficient in these and other ways, it still provided a basis for the evaluation of pattern recognition techniques for determining software quality.

3.2. FEATURE DEFINITION

This section describes the process of feature selection, and includes an itemized list of all features which were considered for extraction. The list was compiled by reviewing those characteristics of programs which were thought to contribute to program reliability/unreliability.

In the following itemized list of features, a justification for the inclusion or exclusion of each feature is presented.

During the course of feature generation and selection, three factors were important in determining the final feature space. They were: pertinence to reliability, ease of measurement, and language independence.*

The generated variables fall into the following classifications:

1. Structural Features
 - a. use of variables
 - b. control flow metrics
 - c. simple counters
 - d. complex counters
 - e. comments or reading aids
2. Non-Structural Features
3. Measures of Program Quality

ITEMIZED LIST OF VARIABLES

1. Structural Features
 - a. Use of Variables

* The features selected are language independent. However, the methods for extracting these features are dependent on the particular programming language used.

- o Number of variables declared. This was easily measured by counting names in the compiler's cross-reference listing. This is thought to influence the size and complexity of a program, and hence a person's inability to deal effectively with the program.
- o Number of unreferenced variables. These are variables declared, but never used; as such, they represent oversights or poor proofreading.
- o Locality of variable reference. This measures the extent to which references to a single variable tend to cluster in a small section of the source code. It was measured by considering the text as a real number between 0 and 1, where each line corresponds to a discrete number. Each reference to a variable was represented as the real number of the line on which it occurred. The variance of all instances of each variable was computed, and the global feature became the mean of all such variances.

It was hoped that this measure would yield substantial information about the influence of any "working set" effect when writing (or reading) a program. If all variables occur in highly local contexts, then it should be easier to understand the function of each.

- o Number of pointer variables. Pointer type variables in PL/I do for data structures what GOTO statements do for control structures. It was conjectured that the indiscriminant use of pointer variables would lead to unstructured (and hence less mentally manageable) data flow.

- o Number of based variables. Based variables in PL/I are a mechanism which types a pointer variable. This typing is dynamic and thus does not allow a consistent static analysis, or enforce uniform use of pointers. For this reason, based variables were thought to be hazardous to the reliability of a program. Note that the measurement here is of the number of declared items, not the number of instances of each item.
- o Number of undeclared variables. This is an attribute which PL/I acknowledges through its declaration semantics. If a variable is referenced but not declared, there exists the possibility of an undetected typographical error accidentally becoming a variable.

b. Control Flow Metrics

- o Nesting. By measuring the extent to which DO statements occur nested in other DO statements, some notion is gained of the complexity of the implemented algorithm. Using the same logic, BEGIN-END nesting and IF nesting were also measured.
- o Maximum nesting. By measuring the extent to which any statement may occur nested in any other statement, the overall complexity of the algorithm is measured. The only statements which were capable of containing others in the context of this project were DO, BEGIN, and IF.

- o Paragraphing. The white space surrounding program source text is an important determiner of the perceived organization and readability of a program.
- o IF balance. It was conjectured that understandability suffered whenever the TRUE clause of an IF statement and the FALSE clause were grossly different in size, and the FALSE clause was only one or two statements long. The reasoning was that, once a person had read through a lengthy compound statement before encountering the ELSE clause, the sense of the original condition would have been forgotten. This out-of-balance construct was a frequent occurrence in the data base. When the single statement in a FALSE clause is an error report, reversal of the sense of the original Boolean condition will allow a reader to keep fewer facts in his mind at once, serving to modularize the code.
- o Complex IF clauses. The IF statement is PL/I's primary decision mechanism. To the extent that statement alternatives are simple, the decision represented by the IF will be simple. A simple statement is one which contains no other statements. This metric is thus a crude approximation of control flow complexity.
- o Complexity measures C2, P2. These metrics (developed by Mitre[11]) are based on techniques which take a control flow graph as input and produce a scalar as output. The control flow graph is reduced by the method to its simplest structural form. The output from

the measures is linearly related to the number of simple statements remaining and the number of control flow paths which do fit into a simple sequence, decision, or iteration model. The essential measurement is the degree to which control flow is structured in the input algorithm.

- o Complexity of IF statements. This metric represents a measure of the complexity of the Boolean expression associated with each IF statement. Since comprehension of decision points is crucial to comprehension of a program, this was felt to reflect general understandability.
- o Complexity of declaration/initialization. Declarational complexity measures data manageability just as control flow complexity measures algorithm manageability.
- o Complexity of I/O statements. Frequent and/or obscure file manipulations can be a source of confusion to a program author (and thus lead to unreliability).

c. Simple Counters

All metrics in this section are counts of the number of occurrences of some "critical" syntactic structures. Most of them are valid for all ALGOL-based languages. The justification for simply counting syntactic structures is twofold: 1) if GOTO statements can be considered harmful, perhaps other

statements can also be harmful, and 2) relationships between certain features may be confirmed. It was expected that at least program length and GOTO statement counts would show some relationship to program reliability.

Among the counters suggested were the numbers of lines, lexemes, assignment statements, I/O statements, external calls, external procedures, formal parameters, actual parameters, semicolons, global variables, types of I/O statements, and the following PL/I primitives: ALLOCATE, BEGIN, CALL, DECLARE, DO, DO WHILE, END, FREE, LABEL, GET, GOTO, IF, ON, PROC, PUT, READ, RETURN, REVERT, STOP, WRITE.

d. Complex Counters

- o Multiple target assignment statements. These are statements of the form $V_1, V_2, V_3 = X$. It was hypothesized that the use of these statements would affect program reliability.
- o Lexemes* in executable statements. The length or size of a statement is a rough measure of its complexity. This metric is a representation of the total size of all executable statements in a program.

* A lexeme is the smallest syntactic construct of the language [14].

- o Operators per assignment statement. This is a rough measure of the arithmetic complexity of a program. It serves to break the notion of complexity into one of its primary components.
 - o Length of I/O lists. The length of an I/O list is a function of the complexity of the I/O being performed by the program. Due to the heavy use of Multics pointers into segments in the data base, this metric is not as accurate as it would be in the general case.
 - o Non-scalar data structures. This metric is a measure of the number of instances of data structures other than scalars. The only such data structures used in the data base are arrays, and for this reason data structure instances were not counted.
 - o Dimensions of arrays. The sum of the dimensionalities of all multi-dimensional variables forms an indicator of data structure complexity and was measured for each program.
- e. Comments or Other Reading Aids
- o Number of comments. This, and other comment-related metrics, are rough approximations to the meaningfulness of comments. The number of comments may predict the readability.
 - o Characters per comment. Comments may be analyzed in any of several ways in an attempt to estimate the quality of a particular comment.

Extracting the length of each comment was an attempt to decide empirically whether terse or lengthy comments tended to occur in reliable programs.

- o Density of non-blank characters within a line. It was hypothesized that large numbers of non-blank characters on a line had a detrimental effect on the ability of a person to analyze a program listing efficiently.
- o Locality of comments. This metric was measured in the same way as the locality of variable reference, and its rationale was similar. The degree of clustering of comments about a single mean was assumed to have some bearing on the comprehensiveness of comments.
- o Density of non-blank characters outside comments. An attempt is made here to measure the proportion of white space utilization across the program text as a whole.
- o Variable name lengths. This feature is an attempt to estimate meaningfulness of variable names. The working hypothesis was that a long name probably is more meaningful than a short one, and thus contributes to readability.

2. Non-Structural Features

There are other variables, not related to program structure, which may affect the quality of programs. Variables of this type would include (but would not be limited to) the following: (1) the amount of testing and verification performed on the program (e.g. by the author, by the coder, by other programmers, etc.), (2) the amount of time spent in designing the program, and (3) the design procedures used (e.g. top down program design, chief programmer team concept, etc.).

However, since information of this type was not kept during the implementation of OLPARS and thus **was not available for this study**, these features were not considered any further for this effort.

3. Measures of Program Quality

Many important aspects of programs fall under the heading of program quality. Certain factors directly affect the reliability of the operation of software, such as the number and severity of errors encountered in running the program. Other aspects are more related to managerial aspects, such as the development time of programs and the number of changes a program encounters. Finally, there are aspects which deal with the software itself. For example, the understandability of program can be measured in several ways. How easy it is for a programmer to read and understand the operation of a program may have a direct influence on the ease of software maintenance. Such important operations as software implementation and software modification can be directly affected by the ease with which a programmer can understand, at the macroscopic and microscopic levels, the functioning of the software.

3.3. PHYSICAL FEATURE EXTRACTION

This section describes the algorithms used in extracting the features described in section 3.2. (The list of features used in this study is contained in Figure 3-1.) Since many of these algorithms are dependent upon features of Multics, they will require modification for use with other systems or languages.

Some of these features are extracted from the PL/I source listing and others are extracted from the PL/I compilation listing. The PL/I source listing contains only PL/I statements which follow Multics PL/I syntax rules. The PL/I compilation listing has the following attributes:

1. The top of the listing contains header information about the listing. This information includes program name, Multics PL/I compiler implementation date, date and time **at** which this program was compiled, and options used when the compiler was interrogated.
2. Each PL/I statement has a statement number.
3. Following the listing is a cross-reference listing. It contains information about all variables used. This information is presented in columnar fashion. Some of the relevant columns of information are:

IDENTIFIER - the name of the variable

Figure 3-1 Features

	Feature Considered	Feature Extracted	Feature Used
o number of comments	X	X	X
o average length of comments	X	X	X
o average density of non-blank characters in comments	X	X	- ^b
o distribution of comments throughout program	X	X	X
o number of lines	X	X	X
o average density of non-blank characters outside comments	X	X	X
o number of multiple assignment statements	X	X	X
o number of variables	X	X	X
o number of semicolons	X	X	X
o maximum nesting level	X	X	X
o maximum BEGIN-END level	X	X	- ^b
o maximum IF-THEN-ELSE nesting	X	X	- ^b
o mean variable name length	X	X	X
o number of lexemes	X	X	- ^b
o IF balance	X	X	X
o distribution of variable occurrences vs. program statements	X	X	X
o complexity of assignment statements	X	X	X
o number of assignment statements	X	X	X
o number of pointer variables	X	X	X
o number of based variables	X	X	X
o number of implicitly declared variables	X	- ^a	-
o number of explicitly declared variables	X	- ^a	-
o number of I/O statements	X	X	X
o number of external calls	X	X	- ^b
o number of external procedures used	X	X	X
o average number of formal parameters in procedure	X	X	- ^b
o average number of actual parameters in call statements	X	X	X

Figure 3-1 Features (Continued)

	Feature Considered	Feature Extracted	Feature Used
o number of complex IF statements	X	X	X
o number of global variables	X	X	X
o average number of each kind of I/O statement	X	X	- ^b
o average length of I/O list	X	X	- ^b
o number of arrays	X	X	X
o total number of dimensions of arrays	X	X	- ^b
o number of ALLOCATE statements	X	X	- ^b
o number of BEGIN statements	X	X	- ^b
o number of CALL statements	X	X	X
o number of DECLARE statements	X	X	- ^b
o number of DO statements	X	X	X
o number of DO WHILE statements	X	X	X
o number of END statements	X	X	- ^b
o number of LABELS statements	X	X	X
o number of GET statements	X	X	- ^b
o number of GO TO statements	X	X	X
o number of IF statements	X	X	X
o number of ON condition declarations	X	X	- ^b
o number of procedure declarations	X	X	- ^b
o number of PUT statements	X	X	X
o number of READ statements	X	X	- ^b
o number of RETURN statements	X	X	X
o number of REVERT statements	X	X	
o number of STOP statements	X	X	- ^b
o number of WRITE statements	X	X	- ^b
o amount of math computation	X	X ^c	X
o amount of user interaction	X	X ^c	X

a- not extracted because it was a required feature of analyzed programs

b- removed due to small variance

c- extracted during understandability study

STORAGE CLASS - e.g. based, constant, automatic, parameter, etc.

DATA TYPE - e.g. integer, floating point, entry, pointer, etc.

ATTRIBUTES & REFERENCES - contains information about the attributes of a variable (e.g. unaligned, external), followed by the statement numbers of indicating where the variable was declared and referenced.

The extraction algorithms will work correctly only on PL/I programs which cause no compilation errors to be produced when compiled by the Multics PL/I compiler.

Three routines (named "parse," "list_extract," and "count_comments") were written to extract the features listed in Figure 3-1: "list_extract" extracts those features from the compiled PL/I listing, "count_comments" extracts those features related to comments from the PL/I source listing, and "parse" extracts the remaining features from the context of the PL/I source listing.

The manner in which each of the features is functionally extracted will be described below. In parentheses next to the feature name will appear the name of the routine which extracted that feature.

Feature 1	Number of comments	(count_comments)
-----------	--------------------	------------------

The number of comments is a count of the number of /* in a PL/I source segment. Thus a comment which extends over several lines, but has only one /* on the first line, will be recorded as one comment.

Feature 2 Average length of comments (count_comments)

The average length of comments is calculated as the total number of characters inside comments divided by the number of comments (i.e., feature 1). The total number of characters inside comments is the sum of the number of blank characters and non-blank characters between /* and */. Blank characters at the beginning and end of a line of a comment are also included in this sum. It is assumed there are 125 characters per line and thus a line of a comment which has its last non-blank character in column 85 and continues on the next line will also be assumed to have 40 blank characters at the end of the line.

Feature 3 Average density of non-blank characters within comments
(count_comments)

The average density of non-blank characters within comments is calculated as the number of non-blank characters inside comments divided by the number of characters in comments. As in feature 2, blanks at the end of comment lines are included in the number of characters inside comments.

Feature 4 Uniformity of distribution of comments vs. statement lines
(count_comments)

This feature is a measure of the way in which comments are distributed throughout a program. It is felt that many comments spread throughout a

program are better than the same number of comments grouped in one place in the program. This metric is calculated as follows:

1. Every time the beginning of a comment is encountered (i.e., a /*), the normalized line number on which the comment occurs is recorded. (The normalized line number is the comment line number divided by the total number of lines in the program). This list of numbers creates a list of every occurrence of a comment.
2. A mean value is then calculated for this list of numbers.
3. The variance from the mean is then calculated and is used for this feature.

Feature 5 Number of lines (list extract)

This feature is a count of the number of lines in a program. A counter is incremented each time a new line appears.

[illegible]

The average density of non-blank characters outside comments equals the number of non-blank characters outside of comments divided by the total number of characters outside comments. The total number of characters outside

comments equals the sum of blank and non-blank characters outside comments.

Blanks before and after lines of text are also included in this sum.

Feature 7 Number of multiple assignment statements (parse)

A counter is incremented each time a statement is recognized in which more than one variable occurs to the left of an assignment symbol.

Syntax:

$$v_1, v_2, v_3, \dots v_n = \text{expression} \quad ;$$

Note that the counter is incremented once for each v_j where $j > 1$.

Feature 8 Number of variables (list_extract)

A count of the number of variables declared is extracted from the cross-reference listing on the Multics compilation of a PL/I program. The section of the cross-reference listing used appears under the heading "NAMES DECLARED BY DECLARE STATEMENT." A counter is incremented each time a new entry appears in the "IDENTIFIER" field of the cross-reference listing. The counter is not incremented if the word "entry" appears in the "DATA TYPE" field, implying that the identifier is a subroutine name.

Feature 9 Number of semicolons (parse)

A counter is incremented once for each semicolon in the source text which occurs outside a literal string and outside of a comment. A semicolon is a line terminator and thus the number of semicolons is a count of the number of statements in a program.

Feature 10 Maximum nesting level (parse)

This integer represents the deepest level of nesting encountered in the source program. It is measured by taking the maximum value (across the entire source text) of an integer "nestlevel," which is evaluated as follows:

1. Increment for each instance of either a DO, IF, or BEGIN statement.
2. Decrement whenever an END corresponding with either a DO, IF, or BEGIN is encountered.

This is a rough measure of the depth of nesting of the program. Note that including the IF statement as a block bracket tends to make a program look "deep." That is, most people do not tend to include the IF statement when estimating depth of nesting.

Feature 11 Maximum BEGIN-END nesting (parse)

This feature is subsumed by feature 10 (maximum nesting level). It decomposes the notion of nesting depth into one of its component parts, i.e. the nesting created with BEGIN-END brackets.

It is measured by incrementing an integer, "beginnestlevel," once for each BEGIN statement, and decrementing it once for each corresponding END statement. The feature value is the maximum value of "beginnestlevel" across an entire source text.

Feature 12 Maximum IF-THEN-ELSE nesting (parse)

This feature is subsumed by feature #10. It reflects the maximum depth of nesting generated by the IF-THEN-ELSE construct. It is similar to feature 11 (maximum BEGIN-END nesting) in that a component of depth of containment of block structures is being measured.

Maximum IF-THEN-ELSE nesting is derived by taking the maximum value across a source text of an integer called "ifthenelselevel," which is derived by:

1. Increment for each IF statement encountered.
2. Decrement whenever an IF statement terminates.

Since an IF statement may contain 2 other statements (each of which may be compound), this value will become greater than one whenever an IF statement contains at least one other IF statement.

The construct:

```
if <b1> then <e1>
else if <b2 > then <e2 >
else if <b3 > then <e3 >
```

has a nesting level of 3.

Feature 13 Mean variable name length (list_extract)

The mean variable name length is calculated as the sum of all variable name lengths divided by the number of variables. The variable names and their lengths are extracted from the "IDENTIFIER" field of the cross-reference listing. Only those variables declared by DECLARE statements are included in this feature; however, all variables in the current data base were declared.

Feature 14 Number of lexemes (parse)

This is a count of the number of primitive PL/I lexemes (or tokens, or atoms) in each source text. Lexemes are groupings of one or more characters from the source which compose identifiers, simple numbers, operators, and special symbols.

This feature provides a semantically meaningful measure of program length.

Comments are not included.

Feature 15 IF balance

(parse)

Given the syntax:

if exp then trueclause else falseclause

This feature is simply

size of (falseclause) - size of (trueclause),

when falseclause is present; zero otherwise

This measure becomes:

1. Large and positive when the falseclause is much larger than the trueclause.
2. Zero when both clauses are of equal size.
3. Large and negative when the trueclause is much larger than the falseclause.
4. Zero when there is no falseclause.

Feature 16 Distribution of variable occurrences vs. program statements

(list_extract)

This feature is a measure of how localized variable references were.

This feature is extracted from information provided under the heading "ATTRIBUTES AND REFERENCES" of the cross-reference listing, and thus no undeclared variables are included. This feature is extracted as follows:

1. For each variable:
 - a. The word "ref" is searched for in the cross-reference listing. The information that follows "ref" is a list of statement numbers indicating where this variable was used.
 - b. For each reference statement number, this statement number is recorded (in actuality a normalized statement number is stored, i.e. the statement number divided by the total number of lines in the program). This list of numbers creates a list of every occurrence of a variable.
 - c. The mean normalized statement number is then calculated for this list of numbers.
 - d. The variance from the mean is then calculated and stored.
2. For all variables:
 - a. The list of numbers generated in 1d. creates a list of numbers which measures the locality of reference for each variable.

The mean value of ld. is calculated and returned as the value for feature 16. This feature thus measures the locality of reference for all variables in one program.

Feature 17 Complexity of assignment statements (parse)

This is a crude measure of assignment statement complexity. It simply counts, for each assignment statement, the number of arithmetic and logical operators in the statement. (All such statements have a complexity of at least one, since the initial "=" is tabulated.)

The measure for the entire source text is the mean number of operators across all assignment statements.

Feature 18 Number of assignment statements (parse)

A count is maintained of the number of assignment statements in each source text. An assignment statement with more than one identifier as its leftmost part constitutes one statement in this context. (cf. feature 7)

Feature 19 Number of pointer variables (list_extract)

A counter is incremented if, for a particular variable, the word "pointer" appears in the "DATA TYPE" field of the cross-reference listing of those variables declared by a DECLARE statement.

feature 20 Number of based variables (list_extract)

A counter is incremented if, for a particular variable, the word "based" appears in the "STORAGE CLASS" field of the cross-reference listing of those variables declared by a DECLARE statement.

Feature 21 Number of implicitly declared variables (list_extract)

This feature may be useful for certain high-level programming languages (e.g., FORTRAN). However, since the Multics "L/I compiler issues a warning for all implicitly declared variables, and since we assume we are extracting from PL/I programs which compile with no warnings or errors, this feature was not applicable to the current set of programs and thus it is currently set to zero.

Feature 22 Number of explicitly declared variables (list_extract)

A counter is incremented each time an entry is found in the "IDENTIFIER" field of the "NAMES DECLARED BY DECLARE STATEMENT" section of the cross-reference listing. Since all Multics PL/I programs must have all their variables declared, this feature will be identical to feature 8.

Feature 23 Number of I/O statements (parse)

An attempt is made here to estimate the amount of I/O activity for each source text. Because of the unorthodox nature of Multics virtual memory and

Multics PL/I I/O, severe constraints were placed on what could be measured easily. Consequently, this index represents only I/O activity expressed in PL/I I/O statements, and Multics I/O to the user terminal.

The measure was derived by summing the number of instances of PL/I I/O statements and Multics "ioa_" calls.

Applying this measure to IBM PL/I, or any other implementation in which PL/I I/O is used, would result in a comprehensive indicator of I/O activity.

Feature 24 Number of external calls (list_extract)

If in the "DATA TYPE" field of the cross-reference listing the word "entry" appears, we know that this variable is an external subroutine. If this is the case, the word "ref" is then searched for, and a counter is incremented for each reference to the subroutine.

Feature 25 Number of external procedures used (list_extract)

A counter is incremented each time a new entry appears in the "IDENTIFIER" field and the word "entry" appears in the "DATA TYPE" field of the cross-reference listing.

Feature 26 Average number of formal parameters in procedures (list_extract)

This feature is the average number of parameters described in the declaration of the subroutine. It is calculated as the total number of formal parameters divided by the number of procedures used. The number of procedures used comes from feature number 25. The total number of formal parameters is extracted as follows:

1. If we have a new entry in the "IDENTIFIER" field and the word "entry" appears in the "DATA TYPE" field of the cross-reference listing, then continue; otherwise stop.
2. Search in the "ATTRIBUTES AND REFERENCES" field for the word "dcl". The number after the word "dcl" is the line number where this subroutine was declared.
3. Start scanning the statement containing the declaration until the word "entry" is found. The information after the word "entry" formally describes the parameter list.
4. Initially increment the counter by one, which assumes at least one formal parameter.
5. Scan the remainder of the string until a ";" is encountered and increment the counter each time a "," is located.

Feature 27 Average number of actual parameters in call statements

(list_extract)

This feature is the average number of parameters described in the first call to a subroutine. It is calculated as the total number of actual parameters used in the first call to subroutines divided by the number of external procedures used. The number of external procedures used comes from feature number 25. The total number of actual parameters used in the first call to subroutines is extracted as follows:

1. If we have a new entry in the "IDENTIFIER" field and the word "entry" appears in the "DATA TYPE" field of the cross-reference listing, then continue; otherwise stop.
2. Search in the "ATTRIBUTES AND REFERENCES" field for the word "ref". The number after the word "ref" is the statement number in which the first call to this subroutine is made.
3. We assume at least one actual parameter, so increment the counter.
4. Scan the statement which contains the first call until a ";" is found and increment the counter each time a "," is found.

Feature 28 Number of complex IF statements (parse)

Given the syntax:

if <expr> then <S1> else <S2> ,

if either the S1 clause or the S2 clause is compound (that is, IF, DO, or BEGIN) then that clause is considered complex.

This feature is the total number of such complex clauses occurring in a single source text.

Feature 29 Number of global variables (list_extract)

A counter is incremented each time a new entry appears in the "IDENTIFIER" field and the word "external" appears in the "STORAGE CLASS" field of the cross-reference listing.

Feature 30 Average number of each kind of I/O statement (parse)

This feature measures the mean number of each kind of I/O statement in the source code. The possible kinds of I/O statements considered are:

get

put

read

write

readlist

ioa_

Due to the extensive use of Multics I/O through pointers in the data base, the value of this feature was not expected to be completely accurate. However, in a general PL/I environment, this measure could be very informative.

Feature 31 Average length of I/O list (parse)

The mean number of primitive elements in each I/O statement. I/O statements examined were:

read

readlist

write

get

put

ioa_

Feature 32 Number of arrays (list_extract)

A counter is incremented each time a new entry is found in the "IDENTIFIER" field and the word "array" is found in the "ATTRIBUTES AND REFERENCES" field of the cross-reference listing.

Feature 33 Total number of dimensions of arrays (list_extract)

This feature is a sum of the number of dimensions for each array. The number of dimensions for each array is calculated as follows:

1. If we have a new entry in the "IDENTIFIER" field and the word "array" appears in the "ATTRIBUTES AND REFERENCES" field of the cross-reference listing, then continue; otherwise stop.
2. Search in the "ATTRIBUTES AND REFERENCES" field for the word "dcl". The number after the word "dcl" is the statement number where this array was declared.
3. We assume at least one dimension, so increment the counter.
4. In the declaration, scan until the variable is found.
5. Scan the statement until a ")" is found and increment the counter each time a "," is found.

Feature 34 Number of ALLOCATE statements (parse)

A count is maintained, for each source text, of the number of PL/I ALLOCATE statements.

Feature 35 Number of BEGIN statements (parse)

A count is maintained, for each source text, of the number of PL/I BEGIN statements.

Feature 36 Number of CALL statements (parse)

A count is maintained, for each source text, of the number of procedures invoked through the PL/I CALL mechanism.

Feature 37 Number of DECLARE statements (parse)

A count is maintained for each source text of the number of "declare" or "dcl" statements.

Feature 38 Number of DO statements (parse)

A count is maintained, for each source text, of the number of PL/I DO statements.

Feature 39 Number of DOWHILE statements (parse)

A count is maintained, for each source text, of the number of PL/I "do" statements containing a WHILE clause.

Feature 40 Number of END statements (parse)

A count is maintained, for each source text, of the number of PL/I END statements. Note that an END statement is always paired with a "start block bracket" such as DO, BEGIN.

Feature 41 Number of labels (parse)

A count is maintained, for each source text, of the number of labelled statements.

Feature 42 Number of GET statements (parse)

A count is maintained, for each source text, of the number of PL/I GET statements.

Feature 43 Number of GOTO statements (parse)

A count is maintained, for each source text, of the number of PL/I GOTO statements.

Feature 44 Number of IF statements (parse)

A count is maintained, for each source text, of the number of PL/I IF statements.

Feature 45 Number of ON condition declarations (parse)

A count is maintained, for each source text, of the number of PL/I ON condition blocks which were declared.

Feature 46 Number of procedure declarations (parse)

A count is maintained, for each source text, of the number of external routines used.

Feature 47 Number of PUT statements (parse)

A count is maintained, for each source text, of the number of PL/I PUT statements.

Feature 48 Number of READ statements (parse)

A count is maintained, for each source text, of the number of PL/I READ statements.

Feature 49 Number of RETURN statements (parse)

A count is maintained, for each source text, of the number of PL/I RETURN statements.

Feature 50 Number of REVERT statements (parse)

A count is maintained, for each source text, of the number of PL/I REVERT statements.

Feature 51 Number of STOP statements (parse)

A count is maintained, for each source text, of the number of PL/I STOP statements.

Feature 52 Number of WRITE statements (parse)

A count is maintained, for each source text, of the number of PL/I WRITE statements.

An executive routine was written to go through the entire data base of PL/I programs and extract the features from the first compilable version of each program, creating a feature file readable by Multics/OLPARS. Besides performing this function, the executive also extracts two variables which are used as estimates of the reliability of programs.

Variable 1 Development time

This is an approximation to the development time of each of the programs. Each copy of the program in each directory represents a different month which

this program was debugged and tested. This variable is a count of the number of different months that this program was debugged or tested.

Variable 2 Number of changes

This variable represents the number of lines of code that have been changed, inserted, or deleted in a program starting from the first compilable version of the program through the final version of the program. Changes could occur when program specifications change or when a programming error is detected and then fixed. Even though both of these types of changes could have been occurring, the OLPARS system being evaluated is a well-defined system (see [8]) and thus most changes were due to programming errors. Lines of comments and leading blanks are ignored in calculating the number of changes. A straight ASCII comparison (using Multics routine "cpa") was made of the initial and final version of a program. The final result of this comparison was stored in a file. This file was then scanned to count the lines of change.

3.4. UNDERSTANDABILITY

One important aspect of program quality is understandability or psychological complexity, the ease with which a programmer (other than the original author) can read a program and understand its operation. Although independent understanding of a program is not a necessary condition for reliable program execution, the amount of cognitive effort required to understand a program

can directly influence the cost of software development, software maintenance, and software modification.

This portion of the study was an attempt to derive two variables which measure the conceptual clarity of the programs in the data base through psychological scaling techniques. In a psychological research paradigm, one of the most direct techniques for generating an evaluative scale (i.e. a feature) is simply to ask the subjects to rate the relevant parameter along a numeric scale: e.g. from 1 to 10 [12]. Over wide ranges of context, experimental demands, and subject populations, rating procedures have been found to be remarkably reliable, both in the ability of individual subjects to replicate their own ratings and in the general agreement of ratings across subjects.* In this study, subjects were asked to rate the understandability of a set of programs on a nine-point scale.

The second feature was more performance-oriented. If the programs in the data base vary in their understandability, then the time it takes a programmer to read and understand a program should be directly related to the program's clarity. Thus reading latency (i.e. reading time) will be an independent measure of program complexity. In addition, as a check on the validity of the rating procedure, the two measures should be at least moderately correlated. After all, the latencies and the ratings are two different procedures for measuring a common construct, program understandability.

* Certainly there are inter-subject differences in the use of any particular rating scale. Some subjects tend to use the upper end of a scale for every judgment while other subjects tend to use the lower end. But even when the mean ratings are very different, the pattern of responses is very similar: typical inter-subject correlations may be well over .90.

3.4.1. Subjects

Twenty employees of Pattern Analysis and Recognition Corporation served as subjects for this study and were paid for their participation. (See Section 3.4.3. for details of the payment procedure.) The subjects were all experienced programmers: the mean (full-time equivalent) programming experience of the subjects was 5.25 years, with a minimum of one year (full-time equivalent) of programming experience. In addition, seven possessed Ph.D.'s; five, M.S.'s; and six, B.S.'s in mathematics, computer science, or related areas. Thus every subject had sufficient academic/work experience in reading and writing programs to evaluate the understandability of programs.

3.4.2. Stimuli

The original set of 260 PL/I programs varied considerably in size. For the analysis of psychological complexity, programs of trivial length (less than 25 lines) as well as excessive length (more than 360 lines) were removed. The remaining 155 programs still covered a wide range of sizes, yet their understandability could be reliably assessed with a minimal amount of inconvenience to the subjects. For each of these programs, the author was asked to generate a question whose answer would require an understanding of the program's operation. These questions (along with a monetary incentive) were used to insure that the subjects carefully read every program encountered.

It was unreasonable to run the subjects for more than one hour per day or for longer than two weeks. As a pilot study showed that only 3 to 4 of

the programs could be read in one hour, it was necessary to give each subject a subset of the programs to read. The programs were divided into four groups of approximately equal size on the basis of program length. For each subject, a set of programs was randomly selected from the four size groups according to the following constraints: 1) Each size group was (approximately) equally represented; 2) Each subject would see 31 programs; 3) Each program would be seen by four subjects; and 4) No two subjects would see the same set of programs. To increase the reliability of the two measures, it was decided that each program would be read by four subjects and the mean values of the ratings and latencies would constitute the raw data.

A loose-leaf notebook was assembled to aid in the study of the programs. The notebook contained verbal descriptions of every subroutine called by the programs and an explanation of the parameters of the calling sequence. The notebook also contained descriptions of all files used by the programs. The subroutine and file descriptions were organized individually to facilitate their use by the programmers.

As an additional aid to the subjects, each program listing had a program description attached to it. This description contained a verbal explanation of the program's function (at a conceptual level) as well as lists of the files and subroutines used and their parameters. It was necessary to include this information as some of the programs already contained similar descriptions as part of their comments. The descriptions gave only information about the global properties of the programs: e.g. the algorithm used. None

of the descriptions gave any clues to the flow of the program or other specific features which might influence the performance of the subjects.

3.4.3. Procedure

One week before the start of the experiment, the subjects attended a PL/I refresher seminar. Features of PL/I and of the Multics operating system were reviewed to insure that everyone was familiar with the language syntax, I/O specifications, etc. Summary sheets of the features were given to the subjects, as were sample programs. The purpose was to minimize any difficulties due to the language or the operating system.

Subjects were run either individually or in pairs. They began by reading two pages of instructions. The instructions stated that they were to read one program at a time, until they understood it sufficiently to translate it into another high-level language. (A copy of the instructions to the subjects is contained in Appendix A.) The subjects were then given the notebooks of subroutine and file descriptions, and were allowed to examine them for a few minutes.

When a subject indicated he was ready, he was given a program listing and corresponding description chosen randomly from his set of 31 programs. Simultaneously, a stopwatch was started. When he indicated that he was finished studying the program, the watch was stopped and the time recorded to the nearest second. If the subject was still studying a program after 25 minutes, he was stopped, and 25 minutes was entered as the latency. Subjects

were told that if they had a question concerning the syntax of PL/I, the watch would be stopped while the question was answered. We were concerned only with differential performance relative to the programs in this study, not in peculiarities or difficulties in the language itself.

After the latency was recorded, the subject was asked to rate the program on a scale from one to nine, where a rating of one implied a trivial program and a rating of nine, an incomprehensible one. The subject was also asked to rate the degree of user interaction in the program and whether the program was mainly numeric or non-numeric in nature.*

Finally, the subject was given a question concerning the operation of the program. If the question was answered correctly, 25¢ was credited to the subject. If the answer was incorrect, 10¢ was subtracted from his account. We were not interested in the answers per se. This procedure was used to maintain motivation in the subjects, for a subject's answer had to be correct in order to produce a monetary gain, and he needed to understand the program in order to answer the question correctly.

After the question was answered, the process was repeated with another randomly selected program. Each subject saw three programs per day for nine days. On the tenth day, he saw four programs. A typical session lasted less

* The subjects were asked to evaluate these two program classification variables since it was felt that there might be an interaction between the type of program and the specific features related to reliability. These classifications reflect global characteristics of the programs and thus could not be automatically extracted at this time.

than one hour, and the ten sessions were spread over three weeks. After the final session, a subject was given the money due him.

The results of this study produced the following information:

Feature 53 Amount of numerical processing

The value was the mean rating across subjects of the relative amount of numeric processing. A rating of 1 indicated a numeric routine and 2 indicated a non-numeric routine.

Feature 53 Amount of user interaction

The mean rating of user I/O across subjects was taken as a global measure of interaction. A rating of 1 indicated no user interaction; 2, user output provided; and 3, the program was interactive.

Variable 3 Latency

The latency was the time duration from the moment a subject received a program listing until he indicated he was finished reading the program. (See section 3.4.4. for a further discussion of this variable.)

Variable 4 Understandability rating

The understandability of a program was defined as the mean rating given by the subjects who read the program.

3.4.4. Variable Generation

In order for the ratings and latencies to be considered measures of the understandability of the programs, it is important to show that the subjects did, in fact, understand the operation of the programs they read. The number of correct answers to the questions was collapsed across the programs to arrive at a measure of the performance level of the subjects. Overall, the questions were answered correctly 94% of the time. There were no systematic patterns to the errors; they appeared randomly distributed over programs and subjects. Moreover, it appeared that a portion of the errors were due to ambiguous questions and subject carelessness (e.g. in locating a particular element of an array). Thus overall it appeared that subjects did understand the programs which they read.

The understandability ratings for each program were collapsed across the four subjects who read it to arrive at a mean understandability rating for that program. Similarly, mean estimates of degree of user interaction and of numeric/non-numeric processing were obtained for each program. These three variables were added to the list of physical features for these programs.

Latency or reaction-time data tend to be highly skewed in a positive direction. A standard procedure for normalizing such time measurements is to transform the data by replacing the latencies with their common logarithms

[7,10]. After they were transformed, the data were collapsed across the subjects to arrive at a mean (log) latency for each program. This variable was also added to the set of physical program features.

It was claimed at the beginning of section 3.4. that both the understandability ratings and the latencies were measuring a common construct which might be called the psychological complexity of the programs. If they both were measuring the same construct in different ways, then the two measures would be related. The correlation between the understandability ratings and the log latencies was 0.80. That is, the two variables were highly related. In fact, even when the effect of the size of the programs was partialled out (i.e. statistically removed), the correlation between the ratings and log latencies was 0.62 (see section 4.3.3.).

SECTION 4

DATA ANALYSIS

4.1. OLPARS OVERVIEW

This section will provide a brief description of the On-Line Pattern Analysis and Recognition System (OLPARS) as implemented under Multics on RADC's HIS 6180 computer. Elements of the pattern recognition problem and a functional overview of the Multics OLPARS Operating System will be included. Emphasis is on the practical use of the system to solve a pattern recognition problem. (For a more detailed study, see "Multics OLPARS Operating System," RADC-TR-76-271.) To test the usefulness of OLPARS in examining program data, a pattern recognition problem was designed to try to classify programmers merely by using the structural features of each programmer's programs. Appendix D presents the details of this procedure.

The pattern recognition problem is described as the recognition of the state of an environment based on L measurements or features extracted from the environment. Thus, the pattern recognition problem is composed of (1) feature extraction, that is, the definition of the measurements, and (2) pattern classification. The objective in selecting features is to provide a set of measurements which yield information which will aid in discriminating between the various environmental states. The pattern classification problem requires that we design the recognition logic, which classifies the state of the environment using the previously defined L features.

The concept of a vector space is fundamental to all of the problems discussed here. The features (measurements) define the basis of the space; an object or an event is represented as a vector in that space. Feature extraction involves defining the representation space, and pattern classification involves defining the partitionment of this space into regions associated with each of the states (or classes) of the environment. In order to solve a pattern classification problem, statistical sample vectors from each state (or class) must be collected and analyzed to yield a satisfactory classification logic.

The pattern analysis problem differs from the pattern classification problem in that the states (or classes) of the environment are a priori unknown to the researcher. The data comprise a set of L-dimensional vectors which must be analyzed to determine the natural or inherent classes contained in the vector data. The detection and identification of a substructure of clusters (sample vectors which cluster together in the vector space) is the solution to this problem.

The vector data structure is represented within OLPARS as a hierarchical tree where each node corresponds to a list of vectors. Partitionment of a list of vectors (node) is represented by branches to lower-order nodes emanating from the node corresponding to the original list, with each subnode being associated with a sub-list.

OLPARS facilities for solving problems of pattern analysis and classification consist of the following types of routines:

1. Data Input, Storage and Output

Data input from cards, tape and other Multics files.

Permanent storage facilities in which OLPARS data may be maintained either for the exclusive access of a given user (exclusive user storage) or for common access by a number of analysts (common user storage). Data trees may be output to either type of storage area, retrieved, and deleted. Lists of the data trees in each area may be obtained by user command. In addition, OLPARS logic and projection vectors may be stored, retrieved and deleted from exclusive user storage, and lists of those data may be obtained separately.

Programs for listing and deleting data trees from current storage are available. In addition, data trees within the current storage area may be modified by adding data classes from other data trees, by combining classes within the data tree, by deleting any data class or data class substructure, or by removing individual data vectors from the data set. A node substructure may be added to the current data tree via the structure analysis module. Any data tree or data class name may be changed, and a display of the current data tree is immediately available. Finally, a new tree may be created from data classes existing in available data trees, or by extracting a percentage of data vectors from an existing data tree. The purpose of this final option is to provide a facility for the creation of randomly assigned design and test sets for the design and independent testing of classification logic.

Listings and data manipulations were very useful in doing preliminary analysis of the programmer classification problem (see Appendix D).

Data trees from current data storage may be permanently stored on magnetic tape.

2. Data Display - Projection Planes and Display Formats

OLPARS provides data display in a 2-space scatter or cluster format and a 1-space histogram format. These displays may be viewed in several projection planes (coordinate, eigenvector, discriminant, and user-supplied). Facilities for user manipulation of these data projection displays include printouts indexing specified points, modifying scale factors, sequencing appropriate data projections, storing projection vectors for later use, changing the data class composition of a display or highlighting specified data classes, and implementing partitions drawn via cursor on the display terminal. For the programmer classification problem, displays were useful in observing within class and between class scatter of the programmer data.

3. Measurement Evaluation

Measurement evaluation computations are also provided which measure the discriminability of features. The measurements are then presented in rank order display format; manipulations available for these displays include printout, rankings for selected classes, class pairs, or measurements, display of the distribution of data along a selected measurement in histogram format,

and selection of a measurement subset for inclusion within a data set of reduced dimensionality. Finally, a program for data set reduction is provided.

For programmer classification, measurement evaluation was primarily used for purposes of data reduction.

4. Data Tree Transformations

Three additional options are available for creation of transformed data sets: normalization, eigenvector transformation, and linguistic transformation of individual measurements.

Since the programmer data had to be normalized prior to analysis, these transformations were very useful.

5. Structure Analysis Partitions and Projections

The creation of subnode structure in a data tree (the structure analysis function) can be implemented via partition of a data projection display or by linguistic statements based on a priori knowledge of the ranges and relationships of data distributions within and between data classes.

An additional data projection display is available for the structure analysis function in the form of a nonlinear mapping algorithm. This algorithm has been equipped with a data set clustering algorithm which allows its

use on large data sets despite the time and space limitations inherent in the maintenance of arrays related exponentially to data set size, which are required by this algorithm.

Since no structure analysis was necessary for the programmer data, options described above were not used.

6. Classification Logic Design and Evaluation

OLPARS logic design facilities provide extensive mathematical/graphical techniques for allowing the user to tailor decision logic design to the structure of the class data. In general, pattern classification is undertaken following a pattern analysis conducted on each of the data classes for which logic is to be designed. The purpose of this analysis is to ensure that each data class is unimodal; that is, the vectors from each class are clustered in one region of the measurement space. Although not always required, the unimodality property is highly desirable in order to ensure an effective logic design. In those cases where the class data are found to be multimodal, the approach dictates that each mode be identified and the sample vectors corresponding to each mode be grouped as a named subclass. Upon completion of the logic design, the decision region in the measurement space corresponding to each subclass can be reidentified with the original multimodal classes.

Basic logic design operations fall into two categories:

- a. Logic capable of completely classifying vectors within the reference group of data classes (complete within-group logic); and
- b. Logic capable of identifying and partitioning completely disjoint data class groups (between group logic).

The full set of logic design capabilities in OLPARS were very useful for purposes of data analysis and logic design of the programmer data.

7. Complete Within-Group Logic

Nearest-mean-vector logic implementation provides capabilities for classification of data utilizing one of three metrics (Euclidean distance, weighted vector distance and Mahalanobis weighted distance). An unknown vector, then, is assigned to the reference class for which the decision metric is minimized.

Fisher pairwise discriminant logic is constructed by computing optimal linear discriminants and thresholds to distinguish between every pair of classes (subclasses) within a designated group.

Closed decision boundary logic creates an L-dimensional closed hyperregion for each class of the selected data set. An unknown vector is assigned to a class if and only if it lies in the hyperregion associated with that class and no other.

8. Between-Group Logic

Data Projections. An obvious drawback to computing pairwise discriminants is the potentially large number of combinations. In most problems of interest, some of the classes (subclasses) are statistically disjoint and quite easily separated from one another. If these disjoint class groups can be identified and logic can be designed to discriminate the groups, then the pairwise discrimination need only be computed for the statistically overlapped classes (subclasses) within the group. The OLPARS user will not ordinarily know a priori how to group the classes (subclasses); therefore, options are provided to project the class (subclass) data onto one- or two-dimensional subspaces and display the results. If the user detects nonoverlapping groups of classes (subclasses), he can draw separating piecewise linear boundaries on the display. These boundaries may be stored within the system as piecewise linear hyperplane boundaries which partition the original L-dimensional measurement space. The user can continue this procedure by selecting one of the class groups and projecting the corresponding data onto a new two-dimensional subspace. If between-class separation is again evident, the user may again partition the original L-space with piecewise linear hyperplanes. If, due to statistical overlap, the classes (subclasses) cannot be completely separated using this procedure, it is recommended that the user complete the logic via within-group discrimination procedures.

Scatter Plot Partitions. The user has the capability to draw multiple piecewise linear convex boundaries. The region external to the drawn bound-

aries may be designated as a reject region, or can be used for data class designation.

Boolean (linguistic) Logic Partitions. OLPARS provides for the implementation of linguistically-defined logic partitions. The user can write any Boolean statement (any statement that can be evaluated true/false) for use as classification logic.

Temporary logic evaluation results are displayed following any logic implementation. Upon completing the logic design, the user can next evaluate the design against any data set and review the results of that evaluation within a confusion matrix format. Logic which provides adequate discrimination may be output to the system printer or stored within exclusive user storage. Inadequate logic may be supplemented, modified, or deleted.

9. Lattice Logic Structure

A capability is provided which permits the analyst to create a lattice-type logic tree structure. This allows, in effect, for two or more logic nodes in an OLPARS logic tree structure to branch together.

10. FORTRAN Subroutine Logic

As an alternative to a simple listing of the discriminants, weighting matrices, etc., which make up the classification logic associated with a given logic tree structure, a FORTRAN subroutine may be created which can

execute the logic. The generated subroutine is in "standard" FORTRAN and may be punched on cards for use at other facilities. A commented listing of the subroutine may also be produced and any data set may be classified with the compiled subroutine.

4.2. ANALYSIS OF PROGRAM QUALITY DATA

This section describes the analysis of the data base of 155 PL/I programs. Since the present study was primarily an investigation of the use of OLPARS to evaluate program quality, the use of OLPARS in performing the types of analysis required will be highlighted. The analysis will be divided into three parts:

1. A preliminary analysis of the structure of the data. This analysis will be used to determine measurement reduction.
2. Classification of program quality based on
 - a. development time
 - b. number of changes
 - c. program understandability rating
 - d. program understanding latenciesfor determining those features affecting these dependent measures.
3. Classification based on programming style: i.e. classification by program author.

The 155 programs used in this analysis represent a majority of the 200 programs that currently exist in the Multics/OLPARS system. These 155 programs represent the programs used in the psychological complexity study as described in Section 3.4.

[Note: Throughout this discussion, when a reference is made to the actual name of an option used in Multics/OLPARS, that option's name will be underlined.]

4.2.1. Preliminary Analysis

At the start of the statistical analysis, it was noticed that certain of the original 54 features were not applicable, since these were not used in the programs examined. The features removed included:

1. Number of implicitly declared variables (all variables must be declared in PL/I)
2. Number of explicitly declared variables (since all variables must be declared in Multics PL/I, this number is identical to the number of variables)
3. Average length of I/O list
4. Number of instances of allocate

5. Number of instances of read

6. Number of instances of write

(The I/O features were not applicable, since Multics PL/I I/O is performed by system subroutines.)

As a result of the psychological complexity study (Section 3.4.), four new features were added:

1. Mean user interaction classification
2. Mean numeric/non-numeric processing
3. Mean log latency of understanding
4. Mean understandability rating

The result was a set of 52 features; 46 were structural, 4 were related to program quality, and 2 were non-structural (user interaction and amount of computation). The next task was to examine the structure of the data.

Initially, an eigenvector analysis was performed.* As an eigenvector analysis requires normalized data, the data were normalized by normxfrm. The eigenvector analysis (eigv\$sal) produced a listing of the eigenvalues and the

* Assuming that the features can be represented in a multidimensional space, the first eigenvector points in the direction of maximum variance; the second eigenvector points in the direction of maximum variance under the constraint that is it orthogonal to the first; etc.

corresponding eigenvectors. From an examination of the eigenvalues, it was found that the first eigenvector accounted for 35.2% of the variance whereas the second vector accounted for only 8.2% of the variance. From this, it can be seen that the first eigenvector accounts for a large part of the variance of the data. Those features which were weighted most heavily on the first eigenvector (i.e. correlated highest with this direction) were:

- Number of lines
- Number of variables
- Number of semicolons
- Number of assignment statements
- Number of instances of do
- Number of instances of end
- Number of lexemes

The hope was that the dependent measures (development time, number of changes, etc.) would be of some importance. However, the first occurrence (absolute weight greater than .2) of any of the dependent measures did not appear until the seventh eigenvector (development time loading = $-.332$). This vector accounted for only 3.4% of the variance.

Thus the features which weighted highest on the first eigenvector, were related to program size. Other findings give similar results (see Section 2). As a result of these findings, it has become programming

practice to recommend making all programs short (less than 100 executable lines). However, this merely moves the problem of complexity one level higher to the system design level without examining the effects of other programming elements on program quality.

It was felt that this finding was of limited interest by itself. What was interesting was that over 60% of the variance was not accounted for by program size. A question remained as to what features influence program quality when the effect of program size is held constant. Had all programs been of the same size, examining these effects would have been straightforward. However, the data were from programs which varied over 10:1 in size. It was possible to eliminate size effects statistically by partialling out size from each of the features. This could have been done by partialling out any one of the following size related features (which are all correlated with values greater than .95):

- o Number of lines
- o Number of semicolons
- o Number of lexemes

Since the number of lexemes gave the highest weighting on the first eigenvector, and since the number of lexemes more nearly reflects the overall size of programs, it was chosen to be partialled out. By partialling out the

effect of number of lexemes, the result will measure each feature as if the number of lexemes in each program were held constant.

Partiallying out the effect of number of lexemes from each feature is done according to the following relation (see [6] for a complete discussion of the partialling procedure in the context of analysis of covariance):

$$G_i = F_i - (r \frac{\sigma_F}{\sigma_P} (P_i - \bar{P}) + \bar{F})$$

where

F_i = Value of original feature for i^{th} sample

\bar{F} = Mean value of F across all samples

σ_F = Standard deviation of F

P_i = Value of feature being partialled out (for i^{th} sample)

\bar{P} = Mean of feature P

σ_P = Standard deviation of P

r = Product-moment correlation between F and P

G = Feature F after influence of P has been eliminated
(partialled out)

The effect on the individual features, in relation to the dependent measures, of partialling out the number of lexemes can be described by the following three cases:

- Case 1: If the feature is uncorrelated with the number of lexemes (correlation = 0.0), then the correlation with the dependent measure will be unchanged.
- Case 2: If the feature is identically correlated with the number of lexemes (correlation = 1.0), then the correlation with the dependent measure will become 0.0.
- Case 3: If the correlation of the feature to the number of lexemes is between 0.0 and 1.0 (0.0 and -1.0), then the correlation with the dependent measure is reduced (increased) according to the degree of correlation between the feature and the number of lexemes.

The partialling was performed on the set of features and the eigenvector analysis was repeated on the resulting data. The results of this analysis were much more encouraging in that no one eigenvector accounted for more than 50% of the variance (eigenvector 1 \approx 15%, eigenvector 2 \approx 10%). Those features which were weighted most highly on the first two eigenvectors were:

eigenvector 1

- number of variables
- number of assignments
- number of external calls
- number of I/O statements
- number of arrays

amount of computation
total number of dimensions

eigenvector 2

number of lines
maximum nesting
number of complex ELSE clauses
number of DO statements

The dependent measures did not load highly on any eigenvectors until the sixth vector, but this vector nevertheless accounts for approximately 5% of the variance and its effect is thus still of interest.

In Figures 4-1 through 4-4, it can be seen that the dependent measures are still continuous variables (as they should be). Figure 4-1 shows a histogram (crdv\$sal) of the development time, 4-2 shows a histogram of the number of changes, 4-3 shows a histogram of the mean rating, and 4-4 shows a histogram of the mean log latency.

Another goal of the preliminary analysis was to eliminate any measurements (features) whose value was constant or typically constant, or which was a linear combination of some other set of features. Examining a listing of the means, standard deviations, ranges, and correlation matrix (all from dataprnt) achieved this goal. As a result of this analysis, the following set of 32 structural and non-structural features remained:

Figure 4-1 Development Time after Number of Lexemes is Partialled Out

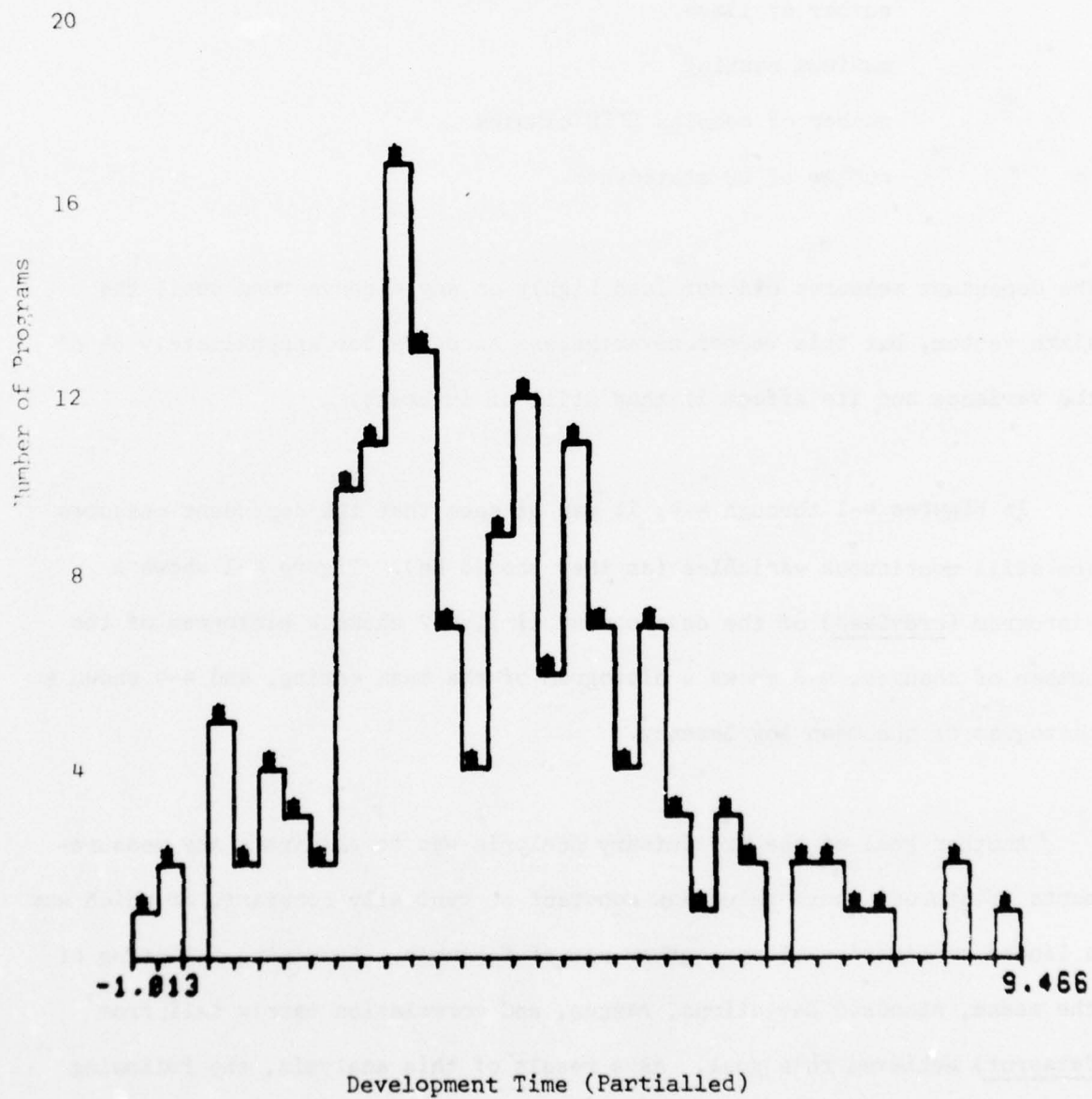


Figure 4-2 Number of Changes after Number of Lexemes is Partialled Out

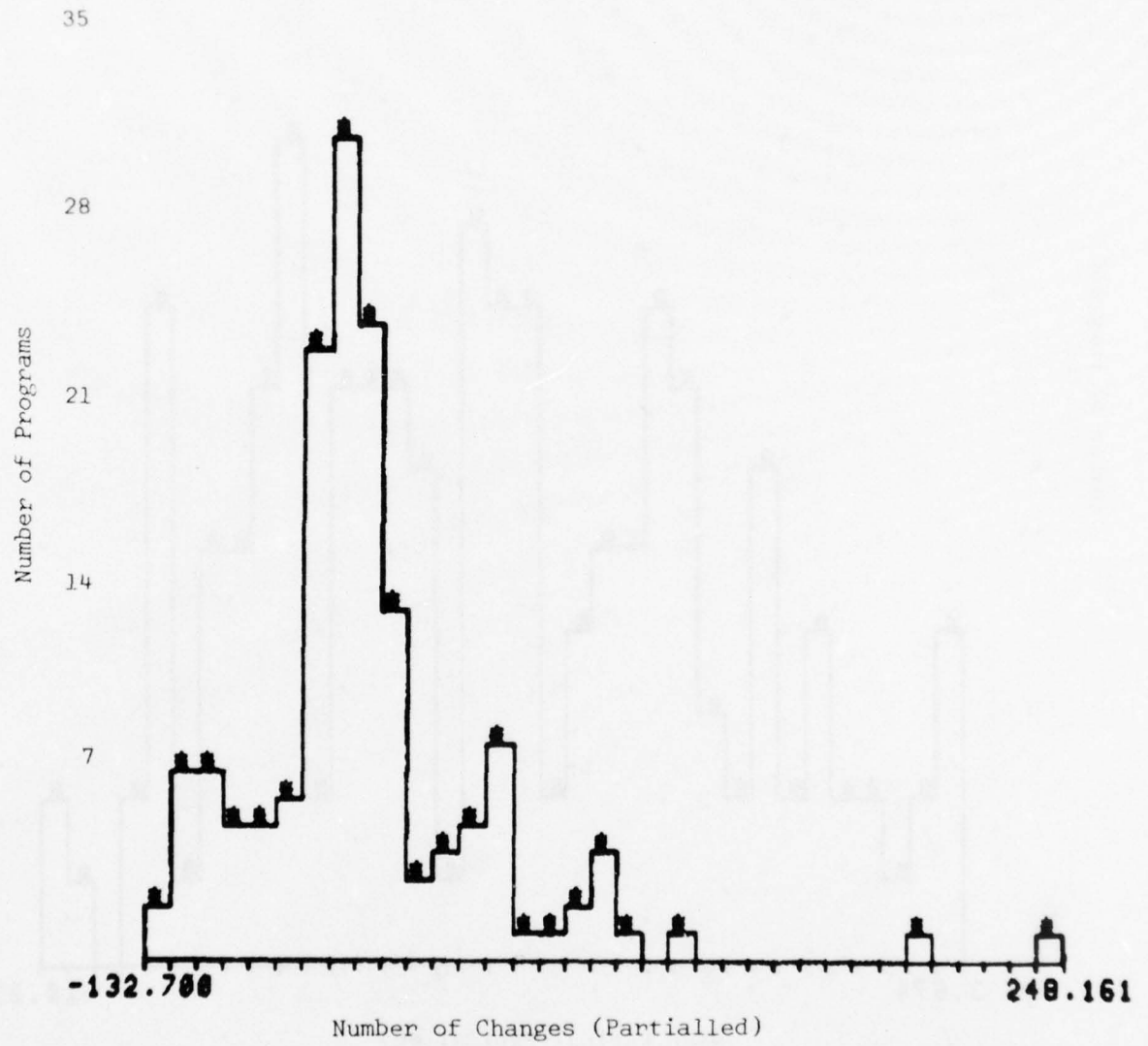


Figure 4-3 Mean Rating after Number of Lexemes is Partialled Out

12

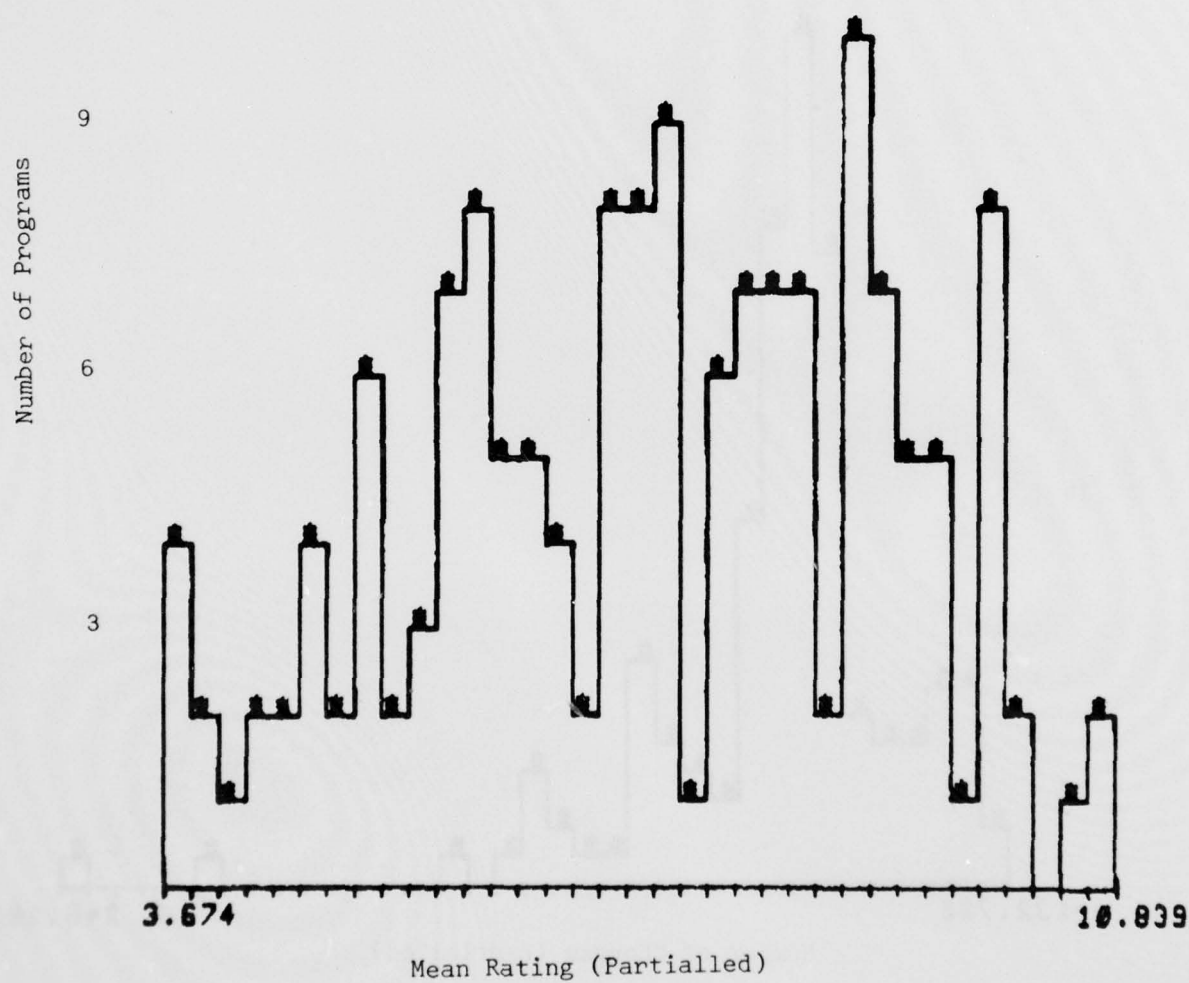
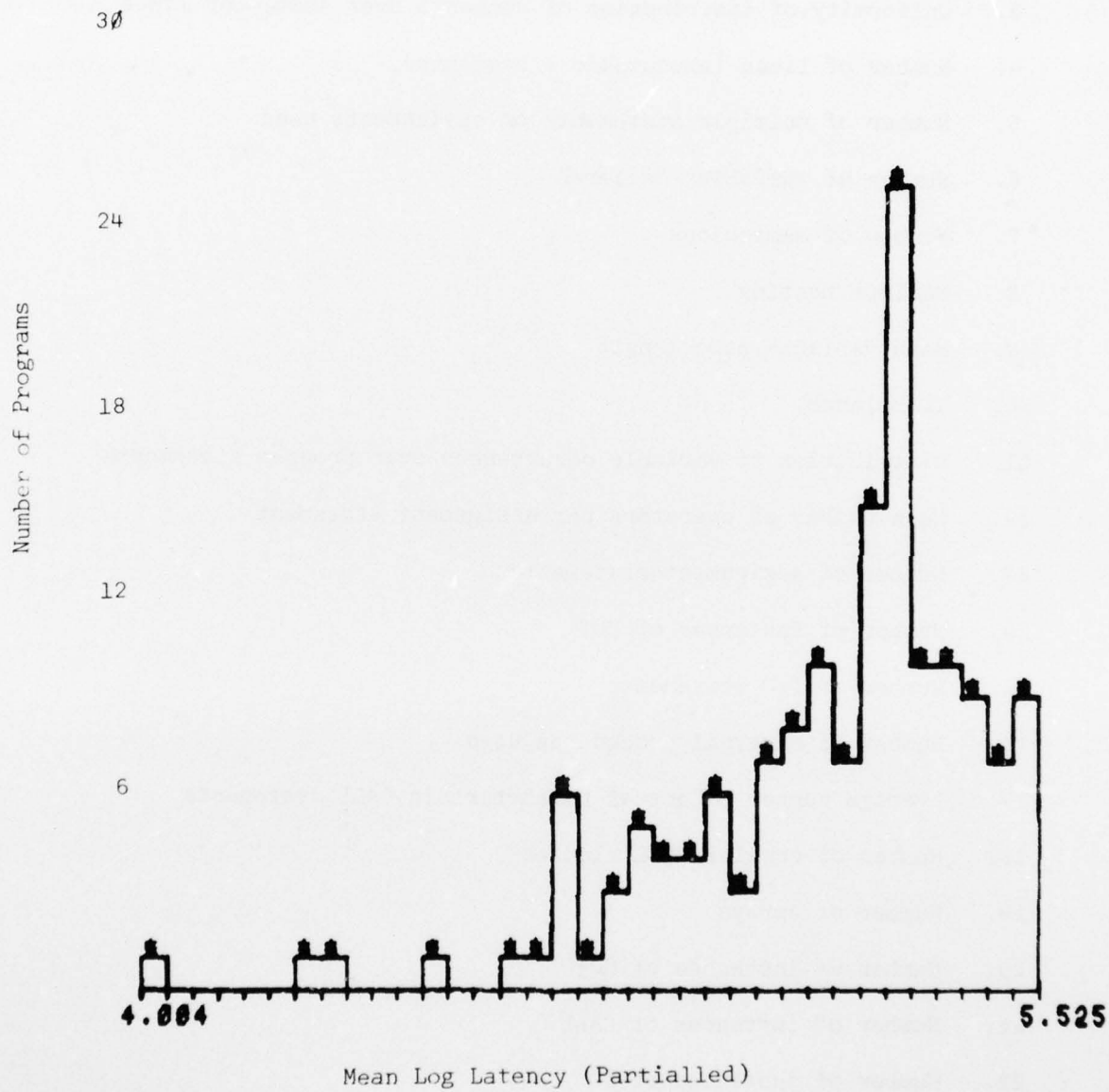


Figure 4-4 Mean Log Latency after Number of Lexemes is Partialled Out



1. Number of comments
2. Average length of comments
3. Uniformity of distribution of comments over statement lines
4. Number of lines (executable + comments)
5. Number of multiple statements or assignments used
6. Number of variables declared
7. Number of semicolons
8. Maximum nesting
9. Mean variable name length
10. IF balance
11. Distribution of variable occurrences over program statements
12. Mean number of operators per assignment statement
13. Number of assignment statements
14. Number of instances of PUT
15. Number of I/O statements
16. Number of external procedures used
17. Average number of actual parameters in CALL statements
18. Number of complex ELSE clauses
19. Number of arrays
20. Number of instances of GOTO
21. Number of instances of CALL
22. Number of instances of DO
23. Number of instances of DO WHILE
24. Number of labels
25. Mean user interaction
26. Mean amount of computation (numeric/non-numeric processing)

27. Number of based variables
28. Average density of non-blank characters outside comments
29. Number of pointer variables
30. Number of global variables
31. Number of instances of IF
32. Number of instances of RETURN

4.2.2. Classification Based on Program Quality

Contained in the feature set are variables which are measures of program quality or program reliability. These measures include:

1. Development time of a program
2. Number of lines of code that have changed throughout the history of a program
3. Mean understandability rating of a program (see Section 3.4.)
4. Mean (logarithmic) time to understand a program (see Section 3.4.)

The purpose of the data analysis described in this section was to establish a set of pattern recognition schemes which would distinguish the extremes of these variables; that is, it would distinguish:

1. Short development time from long development time
2. Small number of changes from large number of changes
3. High understandability from low understandability
4. Short understanding time from long understanding time

Perhaps more importantly, the analysis was used to determine which features influenced the four measures of program quality.

A necessary condition for a pattern recognition paradigm is that the data consist of discrete classes. Using crdv\$sal, histograms were drawn for each of the partialled dependent measures (classification variables). As can be seen in Figures 4-1, 4-2, 4-3, and 4-4, each of the measures was continuous in nature. Thus, in order to use a pattern recognition approach, it was necessary to create classes corresponding to the long and short development time, etc. The procedure for generating the classes from the continuous data was as follows:

1. The vectors were plotted along the feature of interest (e.g., development time).
2. The middle 25 vectors were removed.
3. Those vectors whose values were below the region removed were placed in the reliable class (class name rrrr) and those which were above the removed region were placed in the unreliable class (class name uuuu).

The number of vectors (programs) of interest had thus been reduced from 155 to 130 with 65 in each class. Each vector contained 32 features. The features used in the development time and number of changes problems were extracted from the first compilable version of each program, since one of the questions to be answered was: "Will a program have a short development time?" or "Will a program have a small number of changes?" The features used

in the rating and latency problems were extracted from the final (accepted) version of each program since the ratings and latencies came from subjects who had read these final versions.

4.2.2.1. Procedure

A methodology was established for performing the analysis of the following pattern recognition problems for each of the data sets:

1. Using the partialled data
 - a. A Fisher discriminant (fisher) analysis was performed on the entire data set of 130 vectors, using all 32 features. The Fisher discriminant is a linear combination of features which maximally discriminates among the classes.
 - b. The Fisher direction (i.e. the direction of maximum discrimination) was plotted using gndv\$sal and a listing of the coefficients of this direction (via vec\$save and vec\$list) was produced.
 - c. The number of features was reduced by selecting the nine features whose discriminant weights were the largest. Note that in using this criterion, features were selected according to how well they correlated with the Fisher direction, independent of whether the selected set of features were inter-

related. The reduction was performed using dscrmeas and trnsform.*

- d. The vectors were randomly divided into two data sets (data set 1 and 2) using crrandts. Each set contained 50% of the vectors. In the subsequent analysis, one data set was used to generate the decision logic (fisher) while the other set was used to test the logic (logicevl).
- e. Fisher discriminant logic was designed on data set 1 (fisher) and tested on data set 2 (logicevl).
- f. For purposes of cross-validation, Fisher discriminant logic was designed on data set 2 (fisher) and tested on data set 1 (logicevl).

2. Using the unpartialled data

As was shown in Section 4.2.1., program size had a significant effect on the measures of program quality. The analysis of the partialled data determined which features influenced program quality when program size was held constant. This part of the analysis used a size feature (number of program lines) and the nine features from the partialled analysis to determine the overall program classification.

* The motivation for reducing the number of features was to maintain a significant number of **degrees** of freedom for evaluating the design of the decision logic. Unless the number of features used is much less than the number of samples, the decision logic will be unduly affected by random noise in the values of the features.

- a. The data set was reduced to the 9 features used in lc plus feature 4 (the number of lines) (using dscrmeas and trnsform).
- b. A data set (data set 4) was randomly created which contained 50% of the data vectors in this data set (data set 3) (crrandts).
- c. Fisher discriminant logic was designed on data set 3 (fisher) and tested on data set 4 (logicevl).
- d. For purposes of cross-validation, Fisher discriminant logic was designed on data set 4 (fisher) and tested on data set 3 (logicevl).

4.2.2.2. Results

The analysis described above was used to design classification logic on the four measures of program quality. The results of these analyses are described below. It is important to keep in mind here the convenience and efficiency which OLPARS affords the user. For example, each of the studies described here was completed in 30-45 minutes. This was the total elapsed time from when the operator sat down at the terminal until the last hard-copy graph was produced.

Number of Changes

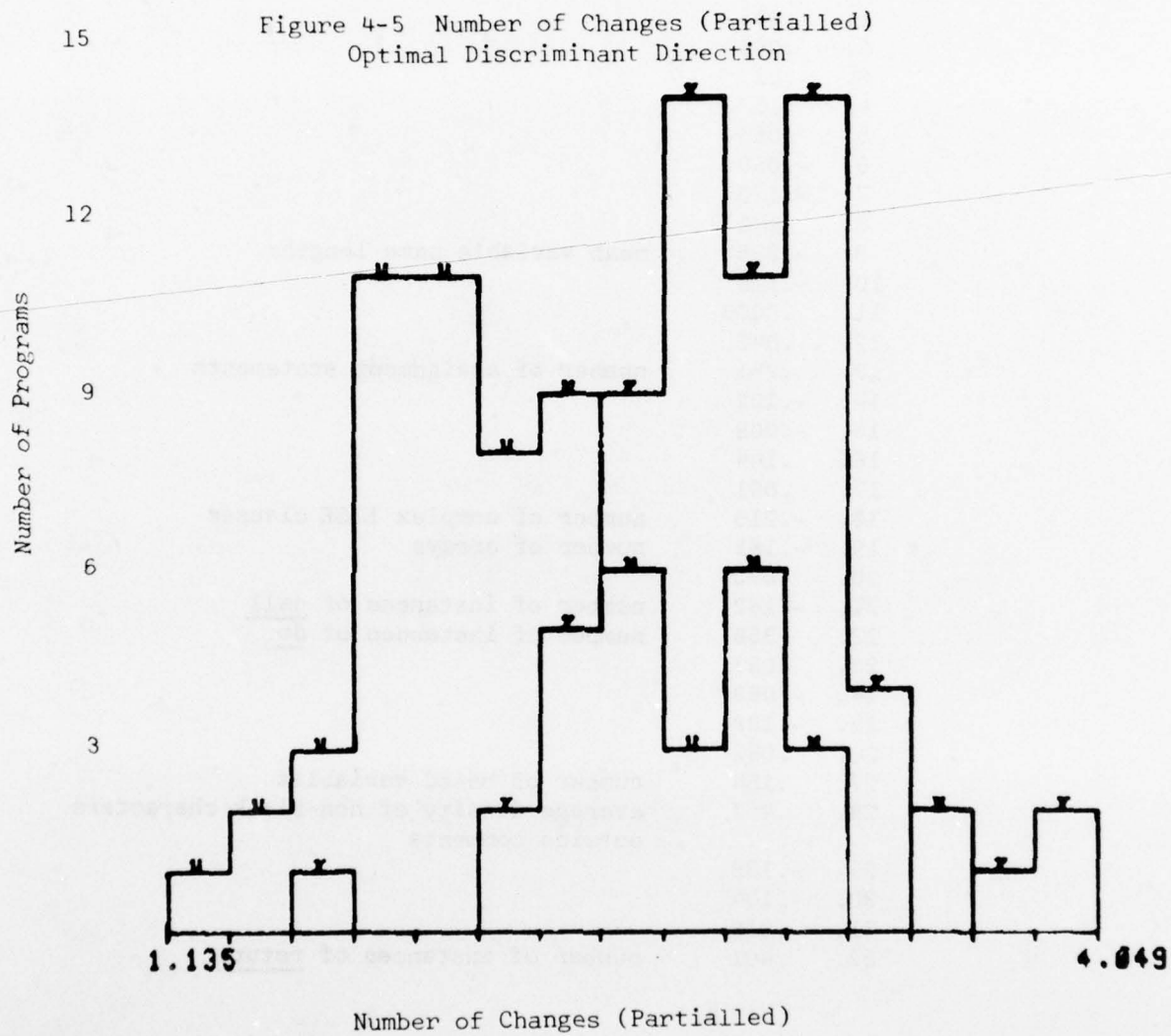
Using the full set of 32 partialled features, the discriminant analysis correctly classified programs containing large or small numbers of changes

76% of the time. The classification table,

		True Class	
		small	large
Classified	small	50	16
	large	15	49

indicates that there was some confusion in the ability to predict the relative number of program changes. Figure 4-5 is a histogram of the two classes projected onto the discriminant direction as produced by gndv\$sal. The extent of the overlap of the two classes is rather clear here. Table 4-1 shows the discriminant weights for this classification. The named features were the ones chosen for the subsequent analysis, since they correlated most highly with the discriminant vector direction.

The data were then randomly divided into two data sets of equal size, and Fisher discriminant logic was designed and tested on these two sets. Table 4-2 shows the resulting design and cross-validation performance using logic derived from the nine partialled features. The upper right and lower left boxes of Figure 4-2 show the results of the cross validations, where decision logic was designed on set one and tested on set two, and vice versa. The significance of the two-by-two classification tables was assessed by a chi-square (χ^2) test to determine if the classification of programs was at a chance level. (If it was, the cell frequencies in each two-by-two table would be about even.) The overall low level of correct test classification (64%), though significantly better than chance in each case, indicates that



	<u>Value</u>	<u>Name</u>
1.	.035	
2.	.056	
3.	-.02	
4.	.157	
5.	.055	
6.	-.060	
7.	-.120	
8.	.141	
9.	-.335	mean variable name length
10.	-.110	
11.	.0003	
12.	.042	
13.	.251	number of assignment statements
14.	-.102	
15.	-.009	
16.	.144	
17.	.091	
18.	-.215	number of complex ELSE clauses
19.	-.195	number of arrays
20.	.073	
21.	-.162	number of instances of <u>call</u>
22.	.358	number of instances of <u>do</u>
23.	.057	
24.	-.063	
25.	-.108	
26.	.082	
27.	.164	number of based variables
28.	.457	average density of non-blank characters outside comments
29.	-.138	
30.	-.104	
31.	.016	
32.	.401	number of instances of <u>return</u>

Table 4-1 Discriminant Coefficients (partialled features)
for Number of Changes Analysis
(The named features represent the reduced feature set.)

DESIGN SET

TEST SET

1

2

		True	
		Small	Large
Classified	Small	25	11
	Large	8	22
71% Correct			

		True	
		Small	Large
Classified	Small	24	9
	Large	8	23
73% Correct			
$(\chi^2 = 14.12, P < .001)$			

		True	
		Small	Large
Classified	Small	18	9
	Large	14	23
64% Correct			

		True	
		Small	Large
Classified	Small	23	14
	Large	10	19
64% Correct			

$(\chi^2 = 6.62, P < .05)$

Table 4-2 Cross Validated Fisher Logic for Classifying Programs by Number of Changes (9 Features, Program Size was Partialled Out)

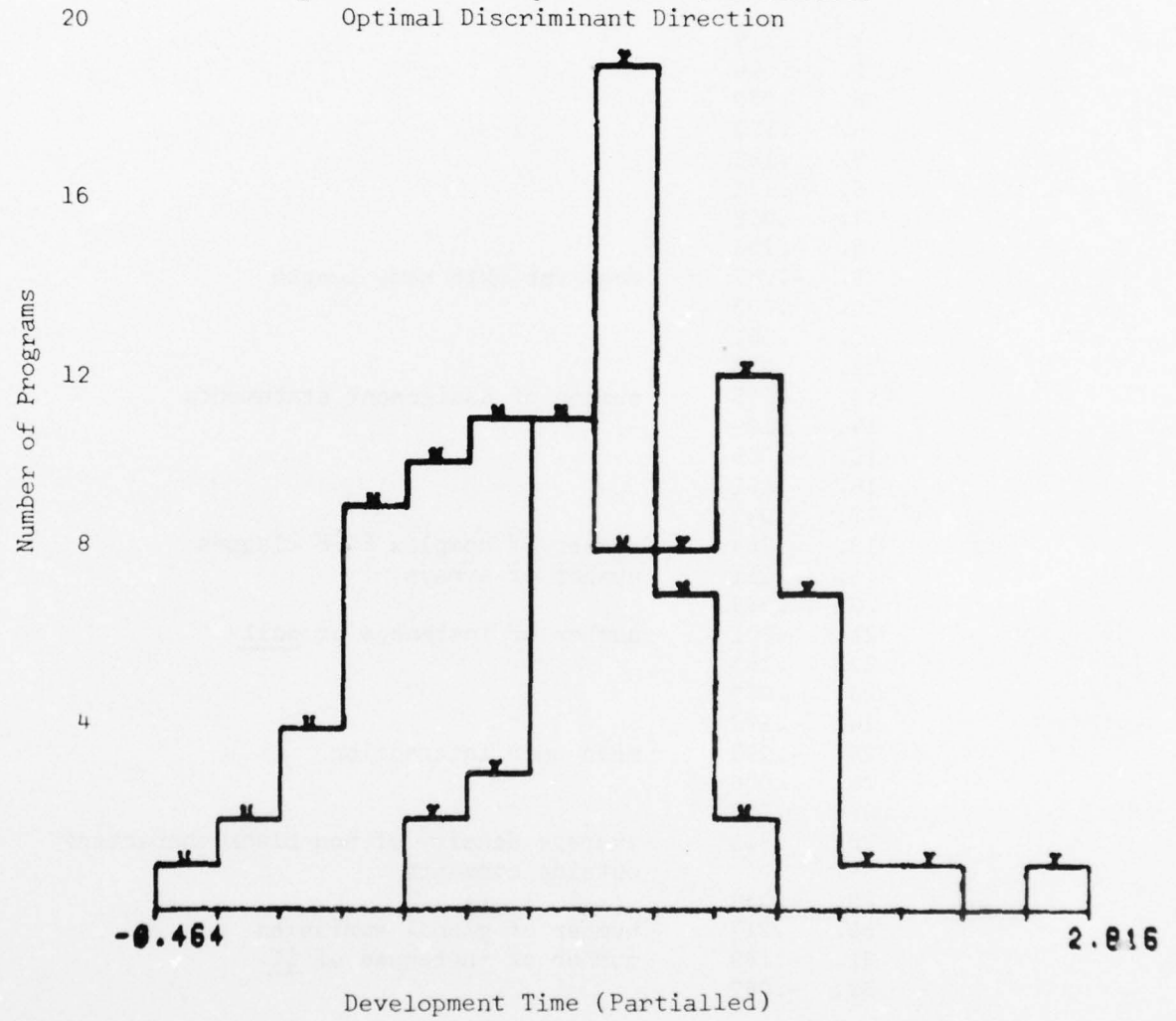
there is considerable overlap in the two classes, even when projected in the optimal direction.

The nine features were then combined with a program size feature (number of lines of code) and discriminant functions were again computed using the unpartialled data. Table 4-3 shows the results of the design and test classifications for the unpartialled number of changes analysis. Although one test classification is not significant, this result appears to be due to the non-uniformity of the randomly chosen design and test sets. In any case, the mean correct classification rate for the two test sets (64%) indicates that the programs are being classified correctly only slightly better than at the chance rate of 50%. Some of the selected features (e.g. number of DO loops, number of based variables) do intuitively relate to a program's complexity. However, the inability to classify programs accurately (according to number of changes) using the ten best features implies that other factors may have a substantial effect on the number of changes a program experiences between being the time of initial compilation and that of final acceptance.

Development Time

Using the full set of 32 partialled features, the discriminant analysis correctly classified programs which had a short or long development time 75% of the time. The two-by-two classification table,

Figure 4-6 Development Time (Partialled)
Optimal Discriminant Direction



AD-A046 588

PATTERN ANALYSIS AND RECOGNITION CORP ROME N Y
PATTERN RECOGNITION METHODS FOR DETERMINING SOFTWARE QUALITY.(U)
OCT 77 T L MCGIBBON, H M HERSH, J M MORRIS F30602-76-C-0214
PAR-77-13 RADC-TR-77-325 NL

F/G 6/4

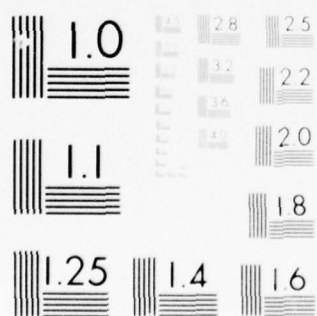
UNCLASSIFIED

2 OF 2

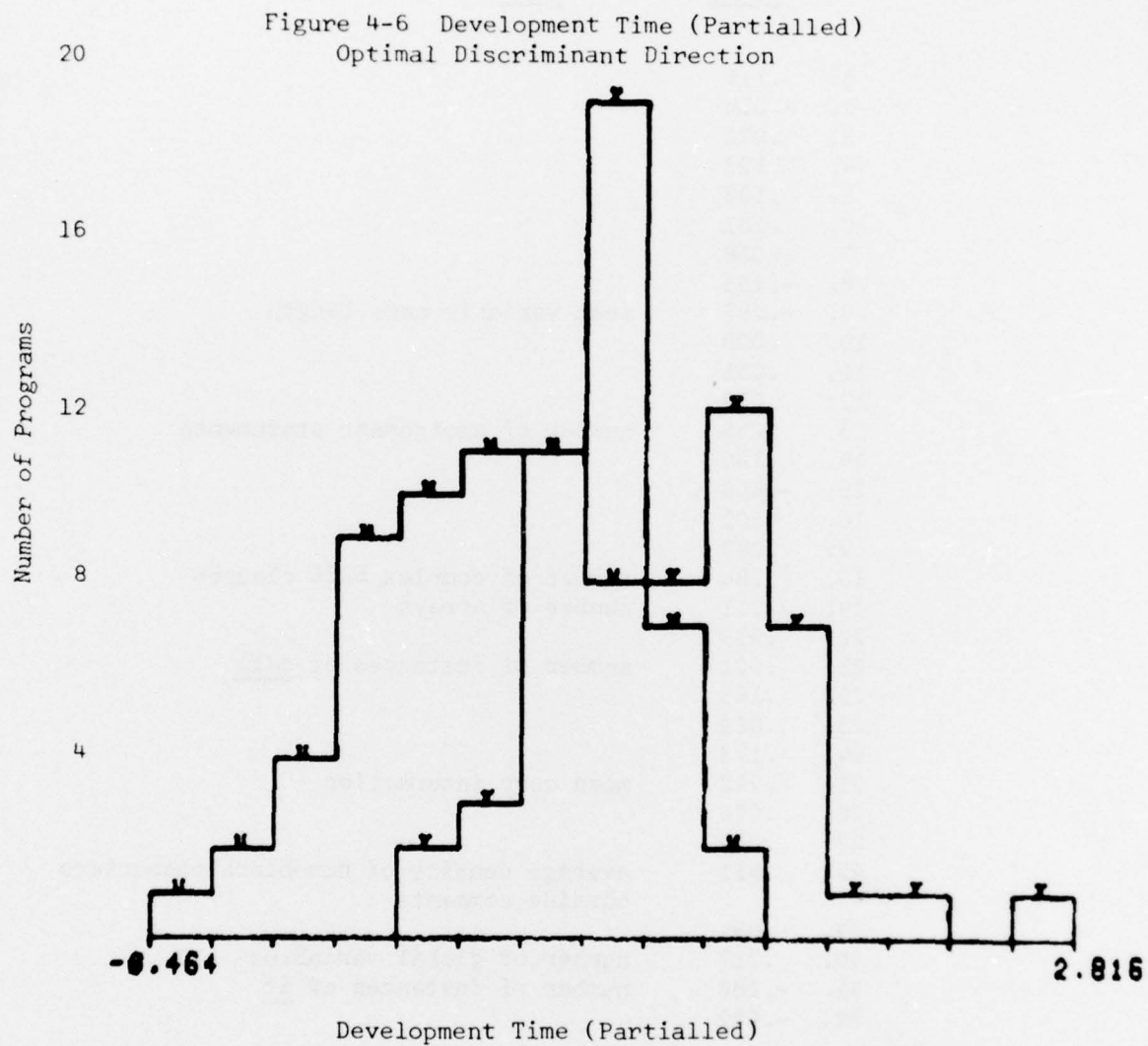
AD
A046588



046588



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



	<u>Value</u>	<u>Name</u>
1.	.118	
2.	-.014	
3.	.020	
4.	.123	
5.	.183	
6.	.131	
7.	-.058	
8.	-.135	
9.	-.247	mean variable name length
10.	.003	
11.	.051	
12.	.073	
13.	.255	number of assignment statements
14.	.109	
15.	-.065	
16.	.101	
17.	.049	
18.	.264	number of complex ELSE clauses
19.	-.211	number of arrays
20.	.338	
21.	.301	number of instances of <u>call</u>
22.	.165	
23.	.069	
24.	-.173	
25.	-.293	mean user interaction
26.	.070	
27.	.052	
28.	.411	average density of non-blank characters outside comments
29.	-.034	
30.	.217	number of global variables
31.	-.188	number of instances of <u>if</u>
32.	-.062	

Table 4-4 Development Time (partialled) Discriminant Coefficients

DESIGN SET

1

2

True

Short Long

Classified
Short
Long

Short	21	12
Long	12	21

63.6% Correct

True

Short Long

Classified
Short
Long

Short	20	13
Long	12	19

60.9% Correct

$(\chi^2 = 3.12, \text{ not significant})$

TEST SET

True

Short Long

Classified
Short
Long

Short	18	10
Long	15	23

62.1% Correct

True

Short Long

Classified
Short
Long

Short	19	8
Long	13	24

67.2% Correct

$(\chi^2 = 5.39, P < .05)$

2

Table 4-5 Cross Validated Fisher Logic for Classifying Programs by Development Time (9 Features, with Program Size Partialled Out)

DESIGN SET

1

2

True

Short Long

Short
Classified
Long

26	10
7	23

74.2% Correct

True

Short Long

Short
Classified
Long

29	11
3	21

78.1% Correct

$(\chi^2 = 24.2, P < .001)$

TEST SET

True

Short Long

Short
Classified
Long

22	10
11	23

68.2% Correct

True

Short Long

Short
Classified
Long

27	6
5	26

82.8% Correct

$(\chi^2 = 8.79, P < .01)$

2

Table 4-6 Cross Validated Fisher Logic for Classifying Programs by Development Time (Ten Features, Including Program Size)

short development period. Note that size of the nine selected features was also selected for classifying the number of changes in a program's development. These features (e.g. number of arrays, number of CALL statements) which reflect the logical complexity of the programs influence both development time and number of changes. Perhaps this is not too surprising, for there is a moderate relationship between these two variables: the produce-moment correlation is 0.26 even with the effects of program size partialled out.

4.2.2.3. Psychological Complexity (Understandability) Measures

Since both the understandability ratings and the latencies were closely related ($r = .62$ after program size was partialled out), the results of the two analyses will be discussed together. As in the previous analyses, discriminant functions were computed using all 32 partialled features. Table 4-7 shows the results of the classifications, and Figures 4-7 and 4-8 represent the histograms of the data projected onto the Fisher directions for the ratings and latencies, respectively.

Tables 4-8 and 4-9 list the discriminant coefficients for the analyses, along with the labels of the selected features. Few of the features selected in the previous analyses were also chosen here. Perhaps this is not too surprising. Development time and number of changes refer to situations in which a programmer is working with his own program. In addition, these two variables reflect the transformations that occur in a program from an initial attempt to a final version. By contrast, the rating and latency variables refer to programmers who were examining programs that someone else wrote.

		True Class	
		Easy	Difficult
Classified	Easy	51	10
	Difficult	12	55

Understandability Ratings

83.1% Correct

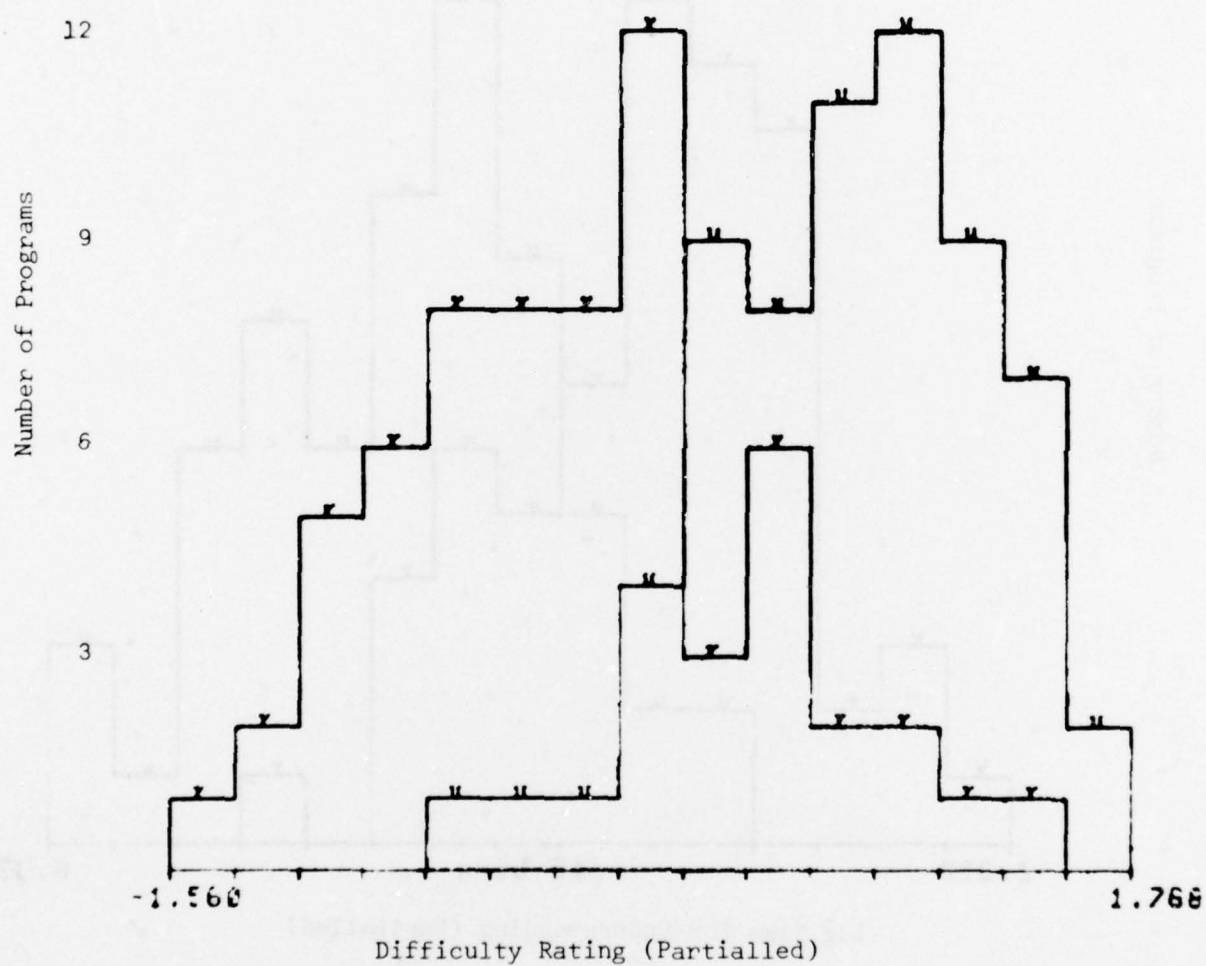
		Short	Long
Classified	Short	50	15
	Long	15	50

Reading Latency

76.9% Correct

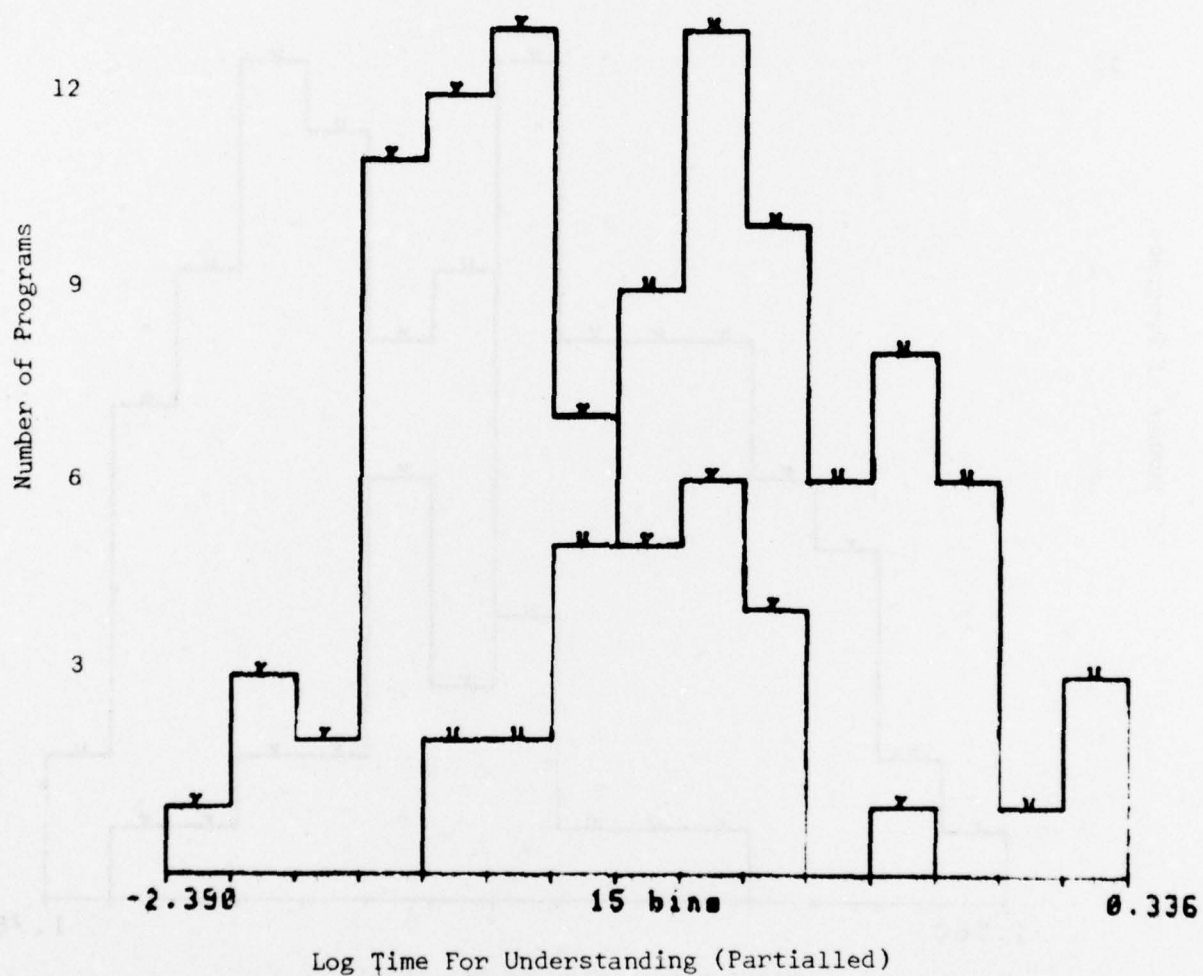
Table 4-7 Fisher Discriminant Logic of Ratings and Latencies
Using All 32 (Partialled) Features

Figure 4-7 Difficulty Rating (Partialled)
Optimal Discriminant Direction



15

Figure 4-8 Log Time For Understanding (Partialled)
Optimal Discriminant Direction



<u>Value</u>	<u>Name</u>
1. .042	
2. -.127	
3. -.032	
4. .080	
5. -.182	
6. -.225	number of variables declared
7. .013	
8. .287	maximum nesting
9. -.132	
10. -.098	
11. .032	
12. .149	
13. .358	number of assignment statements
14. .100	
15. -.239	number of I/O statements
16. -.020	
17. .086	
18. .197	
19. .325	number of arrays
20. .173	
21. .210	number of CALL
22. -.134	
23. .337	number of DO WHILE
24. .170	
25. -.084	
26. -.023	
27. .255	number of based variables
28. .131	
29. .252	number of pointer variables
30. -.049	
31. -.001	
32. .125	

Table 4-8 Discriminant Coefficients (Partialled Features)
for Difficulty Rating Analysis
(The named features represent the reduced feature set.)

	<u>Value</u>	<u>Name</u>
1.	.114	
2.	-.156	
3.	.027	
4.	.100	
5.	-.066	
6.	-.277	number of variables declared
7.	-.018	
8.	.171	
9.	-.142	
10.	.071	
11.	.008	
12.	.319	mean number of operators/assignment statement
13.	.221	number of assignment statements
14.	.058	
15.	.158	
16.	.234	number of external procedures used
17.	.245	average number of actual parameters in call statements
18.	-.208	number of complex ELSE clauses
19.	-.123	
20.	.382	number of GOTO statements
21.	-.150	
22.	.198	
23.	.133	
24.	-.022	number of labels
25.	.034	
26.	-.110	
27.	.160	
28.	-.060	
29.	.103	
30.	.138	
31.	-.060	
32.	.329	number of RETURN statements

Table 4-9 Discriminant Coefficients (Partialled Features) for Log Latency Analysis
(The named features represent the reduced feature set.)

But these programs were also finished products, with many of the problems and errors already eliminated. With the two pairs of variables actually measuring quite different aspects of the programs, it would have been surprising if many of the same features were used in the classification logics.

It was encouraging that many of the features selected in each analysis reflect the findings of other recent studies. For example, the degree of nesting and the use of pointer variables both contributed to increased psychological complexity, as measured by the ratings. In fact, several of the study participants had spontaneously mentioned that programs containing pointer variables were difficult to follow. For the latency analysis, both the number of GOTO statements and the number of RETURN statements were selected. Both these features reflect structured programming goals: eliminate the use of unconditional jumps and provide only a single exit from routines. (A more detailed analysis of the features can be found in section 5.2.).

The programs were randomly split into two data sets independently for each analysis. Fisher discriminant logic was then calculated in a cross-validation design. Table 4-10 shows the classification results for the ratings. The proportion of correct classifications was somewhat better than chance (mean test proportion correct = .65). The proportion of correct latency classifications (.59), as shown in Table 4-11, was slightly lower than in the rating analysis and, in fact, was not significant.

The program size feature was added to the nine selected features in each analysis, and Fisher discriminant functions were again computed. Tables 4-12

DESIGN SET

1

2

True

Easy Difficult

Easy Classified	25	5
Difficult	8	28

80.3% Correct

True

Easy Difficult

Easy Classified	20	12
Difficult	12	20

62.5% Correct

$(\chi^2 = 4.0, P < .05)$

True

Easy Difficult

Easy Classified	24	13
Difficult	9	20

66.7% Correct

True

Easy Difficult

Easy Classified	23	12
Difficult	9	20

67.2% Correct

$(\chi^2 = 8.3, P < .01)$

TEST SET

2

Table 4-10 Classification Tables for Understandability Ratings - Fisher Logic on Nine Partialled Features

DESIGN SET

1

2

True

Short Long

Short
Classified
Long

24

11

9

22

69.7% Correct

True

Short Long

Short
Classified
Long

18

12

14

20

59.4% Correct

$(\chi^2 = 2.5, \text{ Not Significant})$

True

Short Long

Short
Classified
Long

22

16

11

17

59.1% Correct

True

Short Long

Short
Classified
Long

21

10

11

22

67.2% Correct

$(\chi^2 = 3.7, \text{ Not Significant})$

TEST SET

2

Table 4-11 Classification Tables for Study Latencies - Fisher Logic on Nine Partialled Features

DESIGN SET

1

2

True

Easy Difficult

Easy
Classified
Difficult

32	8
1	25

86.4% Correct

True

Easy Difficult

Easy
Classified
Difficult

28	2
4	30

90.6% Correct

$(\chi^2 = 42.5, P < .001)$

True

Easy Difficult

Easy
Classified
Difficult

31	8
2	25

84.8% Correct

$(\chi^2 = 34.2, P < .001)$

True

Easy Difficult

Easy
Classified
Difficult

29	4
3	28

89.1% Correct

TEST SET

2

Table 4-12 Classification Tables for Understandability Ratings - Fisher Logic on Ten Features Including Program Size

and 4-13 show the results of classifying the programs by ratings and latencies using the augmented feature sets. The overall proportions of correct classifications, .88 for the ratings and .82 for the latencies, were highly significant. That is, the addition of the program size feature had improved the ability of the logic to classify programs in terms of their psychological complexity.

The results of adding the size feature raised the question of how well this feature could discriminate by itself. Was program size the only important feature, or was it one of several influential features? In order to assess the importance of program size, this feature was used by itself to classify the programs in both analyses. The results were then compared with the results of classifying the programs using size plus the nine selected features.* Table 4-14 shows the results of using only program size to predict understandability ratings. The results are significant, but the overall level of performance is considerably less than the classification which used both the size feature and the other nine (.76 vs. .88). The inclusion of the selected features increased the performance by 12%, implying that these other features do influence the psychological complexity of programs in addition to the effects of program size.

The latency analysis gave very different results. The classifications, as shown in Table 4-15, are highly significant. The level of performance,

* Statistical procedures have not yet been developed for assessing differential performance as described here. Thus the effects will be evaluated simply in terms of gross differences in performance, i.e. in terms of differences in percentage of programs correctly classified. The question of statistical evaluation will be discussed further in Section 6.

DESIGN SET

1

2

True

Short Long

Short
Classified
Long

29	11
4	26

83.3% Correct

True

Short Long

Short
Classified
Long

27	7
5	25

81.2% Correct

$(\chi^2 = 25.2, P < .001)$

True

Short Long

Short
Classified
Long

31	9
2	24

83.3% Correct

True

Short Long

Short
Classified
Long

32	3
0	29

95.3% Correct

$(\chi^2 = 32.3, P < .001)$

TEST SET

2

Table 4-13 Classification Tables for Understanding Latencies - Fisher Logic on Ten Features Including Program Size

DESIGN SET

1

2

True

Easy Difficult

Easy Classified	26	7
Difficult	7	26

78.79% Correct

True

Easy Difficult

Easy Classified	25	10
Difficult	7	22

73.44% Correct

$(\chi^2 = 14.6, P < .001)$

True

Easy Difficult

Easy Classified	26	7
Difficult	7	26

78.79% Correct

True

Easy Difficult

Easy Classified	25	8
Difficult	7	24

76.56% Correct

$(\chi^2 = 21.9, P < .001)$

TEST SET

2

Table 4-14 Classification Tables for Understandability Ratings - Fisher Logic on Single Program Size Feature

DESIGN SET

1

2

		True	
		Short	Long
Classified	Short	29	7
	Long	4	26

83.33% Correct

		True	
		Short	Long
Classified	Short	27	4
	Long	5	28

85.94% Correct

$(\chi^2 = 33.1, P < .001)$

		True	
		Short	Long
Classified	Short	29	10
	Long	4	23

78.79% Correct

$(\chi^2 = 24.1, P < .001)$

		True	
		Short	Long
Classified	Short	27	5
	Long	5	27

84.38% Correct

Table 4-15 Classification Tables for Understanding Latencies - Fisher Logic on Single Program Size Feature

however, is no different than when the nine features were added (.823 vs. .824). The addition of the nine features had no noticeable effect on the classification performance. This finding is in agreement with the fact that the nine features were classifying the programs (according to understanding latency) at essentially a chance level, as shown in Table 4-11.

SECTION 5

SUBSTANTIVE CONCLUSIONS

5.1. ANALYSIS OF RESULTS

This section will assess the relative merits of each of the program quality analyses as described in Section 4. Where appropriate, a qualitative analysis of the results will be included.

Results of the analysis performed on the latency and rating measures suggest that substantive features do exist which affect program quality. The discussion will begin with an analysis of these results.

The results of the analysis of development time and number of changes are less clear in determining those factors which affect program quality, if development time and number of changes are used as measures of program quality. For each of these program quality measures, correct classifications were made at a level only slightly better than chance (50%). Section 5.1.2. will attempt to assess possible reasons for the inconclusive results.

5.1.1. Understandability Ratings and Latencies

As can be seen from the results presented in Section 4, program size is an important factor affecting the assigned understandability rating and understanding time of a program. Correct classifications increased from 59%

to 82% for latencies and 62% to 88% for ratings when size was included as a factor. In fact, it was shown that size was the only significant factor affecting latency. When size (number of lines) was used as the only feature in the analysis, correct classifications were identical to those when 10 features (including size) were used. Size was the primary, but not the only, factor affecting the rating of a program when size was used as the only feature in the analysis, correct classifications were made at a 76% level vs. 88% when 10 features were used.

When the effects of size are removed from each feature, results are as follows:

5.1.1.1. Latency

In general, after the effects of size are partialled, the selected set of features for distinguishing programs which required a relatively short amount of time to understand, versus those which required a relatively long time to understand, while less than significant, reflects features of programs which exemplify varying degrees of program complexity. Upon examining the sign of the discriminant coefficients of the selected set of 9 features (positive implies a large amount of time, negative implies a short amount of time), the following effects can be seen. Note that since the results were less than significant, no conclusions can be drawn. This analysis is included primarily because the selected features had considerable intuitive appeal from the standpoint of program complexity.

1. As the number of GOTO statements increased, it took longer to comprehend a program. In general, programs in the data base do not have GOTO statements. This result suggests that those programs which do have GOTO statements required much more time to be understood.
2. As the number of RETURN statements increased, the time required to understand a program increased. This suggests that as the number of multiple returns increases, understandability decreases.
3. As the average number of operators per assignment statement increased, it took longer to understand a program. This suggests that the more mathematically complex a program is, the longer it takes to understand it. (Number 8, below, also suggests this).
4. As the number of variables increased, it took a shorter time to understand a program. This would imply that having many variables, each with one role, is better than having few variables, each fulfilling many functions.
5. As the number of parameters in a call statement increased, the length of time required to understand a program increased. This feature reflects the complexity of intersubroutine communication.
6. As the number of externally called subroutines increased, the time required to understand a program increased. This seems reasonable,

since when a subroutine is called, the calling sequence and subroutine description have to be researched.

7. As the number of labels increased, the time required to understand a program decreased. Labels were primarily used to identify entry points in a program. Thus a large number of labels implies a large number of subroutines in one physical program. It can then be argued that a program with a large number of smaller subroutines is easier to understand than a program containing a few larger subroutines. This suggests that modular subroutines, where subroutine is small, are easier to read than subroutines of longer length.
8. As the number of assignment statements increased, the time required to understand a program increased. This feature, in conjunction with the average number of operators per assignment statement, is a measure of a program's mathematical complexity. Thus, as would be expected, as the mathematical complexity of a routine increases, the difficulty of comprehending a program increases. (Remember that this feature measures the number of assignment statements after program size has been partialled out.)
9. As the number of complex ELSE clauses increased, the time required to understand a program decreased. A complex ELSE clause was viewed as an ELSE clause containing more than one statement. When viewed in the light that ELSE clauses were usually used for processing error conditions, which are typically simple constructs, this

feature then measures the modularity (i.e., how small each module is) of the program. Thus programs containing small, self-contained modules were easier to read than programs containing more complex modules.

5.1.1.2. Understandability Ratings

The analysis performed on the understandability ratings, after size was partialled, showed significant results. In this section those 9 features selected as the primary factors affecting the user assigned ratings will be analyzed to determine those concepts which caused a program to be less understandable. Ratings ranged from 1 to 9, where 1 is a trivial program and 9 is an incomprehensible program. The 9 features selected were (in order of decreasing significance):

1. Number of assignment statements - the larger the number of assignment statements, the less understandable the program. This feature is one of several features used as a measure of the mathematical complexity of a program. Thus, as would be expected, the more mathematically complex routines had higher ratings.
2. Number of DO WHILE statements - as the number of DO WHILE statements increased, the program became less understandable. This feature, in some sense, measures the logical complexity of a program. Thus, as would be expected, the more logically complex routines received higher ratings.

3. Number of arrays - as the number of arrays increased, the rating increased. Since arrays were typically used to hold intermediate mathematical values, this feature also signifies that the more mathematically complex routines had higher ratings.
4. Maximum nesting level - as the maximum nesting level increased, the rating increased. As with the number of DO WHILE statements, this feature can be interpreted as a measure of the logical complexity of a routine.
5. Number of based variables - as the number of based variables increased, the rating increased. In the data base, based variables were used (in conjunction with pointer variables) to access and modify values in OLPARS data files. This implies that the more types of data (floating point, integer, ASCII, etc.) being used in files, the higher the understandability rating.
6. Number of pointer variables - as the number of pointer variables increased, the rating increased. Typically, pointers were used to access and modify values in OLPARS data files. The pointer variable allows a program to access particular elements of a file. This suggests that the more complex the filing structure used by a program, the higher the rating.*

* There is also anecdotal evidence to support features 5 and 6. Several of the subjects in the study spontaneously mentioned that programs containing pointer variables were quite difficult. One reason given was that the structure of the data is implicit with pointer variables, while the structure is much more explicit when the data are stored in arrays.

7. Number of I/O statements - as the number of I/O statements increased, the understandability rating decreased. For OLPARS programs, the I/O statements being measured are user interaction I/O statements (as opposed to disk I/O). Typically, when user I/O is required, the user is asked a question (via a call to ioa_) and the response is received (via a call to read_list_). The meaning of this type of I/O is thus reasonably clear to any one reading the program.
8. Number of variables - as the number of variables increased, the rating decreased. This would imply that having many variables, each taking on one function, is more understandable than a few variables taking on many functions.
9. Number of external CALL statements - as the number of CALL statements increased, the rating increased. An external procedure call requires the reader to understand the calling sequence and purpose of the routine being called. This would add to the logical complexity of a program.

5.1.2. Development Time and Number of Changes

This section will assess possible causes for the inconclusive results of the analysis of development time and number of changes. Three possible sources of difficulty can be identified:

1. The characteristics of the particular program data base used.

2. The reliability and validity of the measures of program quality.
3. The quantification of program features.

Any combination of these factors could have influenced the results. In the next three sections, each factor will be addressed separately.*

5.1.2.1. Problems with the Program Data Base

As was stated in section 3.1.2., the characteristics of the data base were such that programs selected to be included represented programs from one company, from one programming project, from one programming language, and from programs of only four programmers. It can reasonably be inferred, then, that the data base may represent too homogeneous a population of programs. That is, programs to be included in a data base for this type of analysis should either include programs from a wide variety of sources, or include programs whose features were varied in a systematic manner. Had other sets of programs been included, results may have been more significant since slight differences in a structural feature would not greatly change any measure of program quality.

* Factors 1 and 3 may have also influenced the suboptimal classification performance with the rating and latency analyses. Although the relation between the two factors and these analyses will not be explicitly discussed, it should be recognized that the discussions apply to these analyses as well.

5.1.2.2. The Validity of the Program Quality Measures

On performing the eigenvector analysis on the unpartialled data (Section 4.1.), it was shown that the first eigenvector accounted for a large portion of the variance in the data. Upon further investigation, it was shown that this first eigenvector related to the size of the programs. Further, the dependent measures (development time, etc.) did not weight highly on any eigenvector until the seventh and eighth. The dependent measures were independent of the size-related features. This suggests that the dependent variables and the independent variables (features) were actually measuring very different things.

5.1.2.2.1. Number of Changes

The number of changes was used to estimate the number of errors that occurred in a program. It was measured as the number of lines of code which changed over the entire development of each program. Possible problems with this measure exist in that no qualification for the cause of each change was made (such data were not available). Thus a program may have had many changes which were due to a redefinition of the program function. In such a case, the program should have been divided into two subgroups, where the first group represents the programs used in the earliest definition of the program and the second group represents the programs used in the second definition of the program. The two groups would then be viewed as two separate programs. However, since such qualification data was not available, it was not possible

to do this. Thus the estimate may not only represent the number of errors, but also stylistic changes (comments, indentations) or modifications of functions.

5.1.2.2.2. Development Time

The development time was used to estimate the amount of effort required to make a program function properly. Two problems exist with this measure:

1. The measure was at best a rough estimate of the development time. It was measured as the number of different months on which a program was tested, without regard to the amount of effort expended in each month on that program (since the data were not available).
2. As with the number of changes, no reasons were available to explain why programs were being debugged in a particular month or how many manhours actually went into the development of the program.

Thus the measure may not have accurately represented the amount of effort expended in making a program function properly.

5.1.2.3. Problems with Structural Features

In defining structural features, the goal was to quantify certain qualities of a program which affected the logical and psychological complexity of a program. For example:

1. In trying to measure how well a program was commented, the following features were selected:

- a. Number of comments
- b. Average length of comments
- c. Average density of non-blank characters within comments
- d. Uniformity of distribution of comments over a program (variance from mean is used)

Distributing comments over a program in many different ways will give almost identical results for each of these features.

2. In trying to measure how mnemonic or useful variable names were, the feature selected was the mean variable name length. Perhaps a better measure would be the mode, maximum, or minimum variable name length. Or perhaps some quantity other than length (e.g., a direct measure of mnemonic value of a name) would have been better.

3. In trying to measure how much computation a program performed, the features selected were:

- a. Number of assignment statements
- b. Mean number of operators per assignment statement

Instead of the mean number of operators, perhaps the maximum number of operators more accurately represents the mathematical complexity of a program.

SECTION 6

METHODOLOGICAL CONCLUSIONS

The research discussed in this report approached the problem of investigating program quality from the perspective of a classification analysis. Although the validity of such an approach will be discussed in Section 6.2., the methodological aspects of the analysis actually performed will first be reviewed.

6.1. USE OF OLPARS AS A CLASSIFICATION TOOL

Perhaps one of the strongest arguments for the use of OLPARS to analyze data sets is that the total analysis time decreases dramatically. Anyone who has tried to run an analytic routine using a standard statistical package such as BMD or SPSS can certainly appreciate the convenience of using a fully integrated system such as OLPARS. The ability to run analyses (e.g., eigenvectors, discriminant functions) by the typing of a single command, while the system automatically maintains the data files, necessarily results in increased throughput (and reduced frustration on the part of the user). For example, all the analyses described in Section 3.4.2.2. were run in a single 3-hour session. But even then, part of the 3 hours was used to create hard copies of the graphic displays. In another session, the total analysis of the stylistic differences among the programmers (Appendix D) required only one hour of connect time. This included the complete development and testing of the hierarchical classification logic.

Another very positive aspect of OLPARS as used in this study was that graphic displays were generated effortlessly. The ease of constructing plots of histograms and scatterplots enabled the analyst to remain close to the data throughout the analysis, thereby allowing the immediate detection of anomalous situations. For example, by displaying the distributions of the data projected onto the Fisher discriminant directions, it was possible directly to assess whether the shapes of the distributions had affected the performance of the classification logic.

Other aspects of OLPARS were not so convenient, either due to the lack of desirable features of the system, or due to the nature of the problem at hand. For example, OLPARS contains a feature which allows users to specify arbitrary data transformations by entering a PL/I program. It would have been desirable to partial out the effects of size using such a user-specified transformation. Unfortunately, the routine that allows the creation of these PL/I programs was not designed for such involved transformations. As a result, a minor restriction prevented the use of the transformation package.*

Another inconvenience was due to the nature of the problem at hand. In a typical pattern recognition problem, an analyst starts with two (or more) sets of sample vectors, one from each a priori class. The goal is to use a combination of the features in the sample vectors to predict the class membership of the samples. The only requirement is that the ratio of sample

* The problem was that the OLPARS routine automatically inserts a semicolon at the end of each line. As a result, multi-line statements were not allowed. Thus, it was impossible to write data definitions for the full mean and standard deviation vectors. By making the inclusion of semicolons at the end of statements a manual operation, this restriction could be easily circumvented.

size to number of features be large. In the present problem, the goal of correct classification was a necessary but not a sufficient condition. It was important to be able to specify the quality of a program as either good or bad, but, perhaps more importantly, it was desirable to determine which features actually contributed to this distinction.

In the type of research discussed in this report, it is desirable to investigate the contribution of individual features. For example, it is possible to assess whether a discriminant function of n features is classifying above a chance level by means of a χ^2 goodness of fit test (or perhaps more appropriately, by means of a multivariate analysis of variance [2]. However, there is no way that one can objectively (i.e., statistically) determine whether the addition of the $n+1$ feature will significantly improve the classification performance.

The lack of an evaluative procedure is not unique to OLPARS. It is unclear whether any such evaluative procedure has been developed or even discussed in the statistical or pattern recognition literature, although procedures are available when the dependent variables are continuous. This situation is discussed further in Section 6.2.

The fact that the data were initially continuous implied that only one class was present. As a pattern classification approach was to be used in analyzing the data, it was necessary to split the data arbitrarily into two classes. OLPARS assumes that the data already reflect discrete classes, and

thus the procedures for splitting a single class are less than adequate. The problem was not that an additional routine could not be incorporated into OLPARS, for the subroutine would be relatively simple. Rather, OLPARS was designed as a system for analyzing class data, and classes are typically unique and specifiable a priori.

6.2. DISCRETE CLASSES OR CONTINUOUS DATA

The analyses performed in this project were all approached from the perspective of a pattern classification paradigm. In certain situations, this was entirely appropriate. For example, the analysis of programmer style clearly fits the classification paradigm. The data represented four unique classes, defined by the particular authors of the programs. There were no confusions as to which program belonged to which class.

The analyses of program quality were somewhat different. Each of the dependent variables was continuous in nature; in fact, each was measured along an interval scale. It is always possible to reduce interval measurements to ordinal or categorical (dichotomous) scales, but considerable information is lost. (This point will be expanded below.) Figures 4-1, 4-2, 4-3, and 4-4 clearly show that the dependent measures are not dichotomous, but continuous. There are no distinct clusters, not even bimodality in the distributions: there is only a gradual transition from simple to difficult, short to long development time, etc. One can always divide the continuum in two, but the division will be arbitrary. Another analyst will split the data at another point. Even the same analyst will probably not be able to replicate

his line of demarcation accurately. (See [1] and [4] for a further discussion of this point.)

Of course, it is always possible to reduce the data to ordinal or even categorical scales (as was done in this study). For example, Table 6-1 lists the performance of several hypothetical systems as measured on different scales. In reducing the scale of measurement from interval to ordinal, the difference between systems A and B becomes comparable to the difference between systems B and C. The fact that system A is considerably better is no longer retained. A similar information reduction occurs when the scale of measurement becomes categorical.

In the present study, the dependent measures were reduced to categorical information because originally it was felt that program quality was categorical (e.g., reliable/unreliable). Later, the categorical nature was retained because the data were compatible with the analytic routines contained in OLPARS. One point that should be made is that the same analysis could have been used if the dependent variables remained continuous. It is known that for a two-class problem, the linear discriminant function is equivalent to the linear multiple regression equation (see [13] and [5]).

But there are additional benefits to be gained by the use of continuous data. For example, it is possible to use analysis of variance techniques in order to assess whether the addition of another feature will result in a significant increase in performance. Also, separate design and test data sets are unnecessary, as there are additional statistical techniques to

Table 6-1 Performance (Percentage) of Several Hypothetical Systems

<u>System</u>	<u>Interval Measurement</u>	<u>Ordinal (Rank) Measurement</u>	<u>Dichotomous Measurement</u>
A	99.9%	1	1
B	79.9	2	1
C	79.2	3	1
D	79.1	4	0
E	64.0	5	0
F	62.7	6	0

determine the amount of decrease in performance as a function of sample size, number of features, and obtained performance level.

The point is that the current OLPARS system did not contain routines to deal with continuous data. Thus in order to use this efficient system, the data need to be transformed to be compatible. It is important to note that the inability to manipulate continuous data is not a shortcoming of OLPARS, as the system was designed for discrete class applications. The next section will discuss extensions/modifications to OLPARS so that continuous data can be processed as efficiently as discrete data.

SECTION 7

RECOMMENDATIONS FOR FURTHER RESEARCH

This report has emphasized the ease with which OLPARS could be used to determine the relationships among selected features of computer software and to assist in classification of that software into categories of interest.

As a pilot study, the experimentation reported here was surprisingly successful. The ability to classify programs according to the style of the programmer who wrote them, for example, was an unexpected outcome. At the same time, the data base was relatively small, and other limitations in the available data would make it unwise to generalize the substantive results of this project.

As more data become available, it will be desirable to conduct further small-scale studies to test carefully defined hypotheses concerning components of software quality. Among the experiments to which OLPARS might be applied are such studies as the following:

1. In the determination of features which contributed to program readability, a set of existing programs was used. Another, more controlled approach would call for the generation of sample programs with designated features to be tested for understandability, and other characteristics of program quality, in an experimental setting.

2. As noted in this study, the length of a program is likely to be inversely correlated with its understandability. This observation has led to the recommendation that program modules be kept short--typically, to a page or two in length. It has not been shown, however, that a reduction in module length will lead to greater system quality, since a larger number of modules, with a much larger number of interfaces, are likely to be required by the total system. (In the extreme case, N modules will require $N(N-1)/2$ interfaces.) For this reason, an evaluation of total system quality is needed, with techniques for estimating and predicting system reliability/unreliability.
3. As of April, 1977, RADC has gathered reports concerning over 25,000 errors from seven software development projects. This newly acquired data base can serve as a source for future studies of factors affecting software quality. Since this data base includes a variety of languages, programmers, and project goals, it should serve to validate or modify the substantive results of the study reported here. Specifically, the program features and classifications used in this study should be applied to RADC's expanded data base to determine factors affecting program quality, using methods which are based on those developed for this study.
4. Research in pattern recognition, as applied to such areas as waveform processing and image identification, has shown that it is essential to understand the process by which specific patterns are generated.

For example, an effective logic for recognition of radar patterns requires a detailed knowledge of radar technology, as it enters into the generation of those patterns. Similarly, it is likely that a much better understanding of the software production process will be required to identify those characteristics which contribute to software quality. For this purpose, controlled studies of the software production process will be needed. During the present study, methods for evaluating the understandability of programs were developed; such techniques ought to be expanded to include tests of the ease with which programmers can use various types of program specifications, program structures, programmer team organizations, documentation standards, and other factors which can contribute to software quality.

The studies reported here have demonstrated that the existing OLPARS implementation is capable of testing features against designated classes, to develop a classification logic for programs. It would be interesting to determine what additional facilities might be added to provide a comprehensive set of tools for research in factors contributing to program reliability.

Other versions of OLPARS have been developed with specialized capabilities for waveform analysis and for image extraction, enhancement, and analysis. A new implementation, the Automatic Feature Extraction System, will provide specialized cartographic capabilities. Similarly, it is possible to describe an OLPARS which provides specialized tools for research in software reliability.

Such a specialized system might include the following facilities:

1. A language-independent feature extractor. This would permit the user to specify syntax of the language under investigation, and would extract selected features from the target language. Output from this operation would be feature vectors for input to other OLPARS routines.
2. Several of the software studies undertaken under RADC sponsorship require techniques for dealing with continuous variables, rather than the disjoint distributions assumed by OLPARS. For example, estimates of software reliability would require the ability to estimate a variable quantity, rather than merely to classify programs into reliable vs. unreliable categories. One typical tool for producing such estimates, which is not now contained in OLPARS, would be a facility for multiple regression analysis. Techniques for regression analysis are well known and could easily be included in a specialized OLPARS implementation.
3. Input features to OLPARS classification logic now must be numerical quantities, measured on a ratio scale. Many of the features of interest in software reliability research, however, are qualitative rather than quantitative in nature. To take a simplified example, some languages (such as ANSI FORTRAN) rely most heavily on a DOUNTIL structure for DO loops, while others (such as PL/I) rely on a DOWHILE structure. It should be possible to include this as a

feature for input to OLPARS; but it is a qualitative (DOUNTIL vs. DOWHILE) feature, rather than a quantitative one.

An earlier version of OLPARS (AMOLPARS), which used the Goodyear Associative Memory, provided the ability to use qualitative features, as well as quantitative features. This ability should be included in a specialized OLPARS facility for software reliability research.

4. PAR has developed a meta-compiler, PARLEZ, which permits the user to define syntax and semantics of a new language, and which provides a compiler for the language as defined. A second meta-compiler, XMETA, is also under development. One or both of these facilities should be provided in a specialized software research laboratory, for the purpose of developing and testing new language structures in an efficient way.
5. General enhancements of the OLPARS design would be desirable in a new facility, including such statistical tools as the following:
 - a. Stepwise multiple regression analysis
 - b. Optimal regression techniques, indicating the best number of features to be selected
 - c. Traditional factor analysis routines

- d. More flexible non-linear mapping and multi-dimensional scaling routines
 - e. Improved procedures for splitting classes
 - f. Analysis of variance routines
 - g. Covariance analysis routines
6. Interactive facilities for the specialized OLPARS might include the ability to display two or more software modules in a variety of positions, to permit the user to make judgments concerning their readability, structural characteristics, depth of nesting, etc.

Since existing OLPARS routines have been found to be valuable in the analysis and classification of programs, and since the extensions described here are relatively minor and well within the state of the art, the development of a specialized OLPARS implementation for the study of software appears to be a feasible task.

REFERENCES

1. Black, M., "Vagueness," Philosophy of Science, 1937, Vol. 4, Pages 427-455.
2. Harris, R.J., A Primer of Multivariate Statistics, New York: Academic Press, Pages 111-113, 1975.
3. Hatter, R.J., "Excerpt from CCIP Study Regarding Analysis of TRW Software Analysis Data," Lulejian and Associates, Inc., Redondo Beach, California, November 1971.
4. Hersh, H.M. and Caramazza, A., "A Fuzzy Set Approach to Modifiers and Vagueness in Natural Language," Journal of Experimental Psychology: General, Vol. 105, No. 3, Pages 254-276, 1976.
5. Kerlinger, F.N. and Pedhazur, E.J., Multiple Regression in Behavioral Research, NY: Holt, Rinehart, and Winston, 1973.
6. McNemar, Q., Psychological Statistics, Fourth Edition, New York: John Wiley and Sons, 1969.
7. Myers, J.L., Fundamentals of Experimental Design, Second Edition, Boston: Allyn and Bacon, 1972.

8. Sammon, J.W. Jr., "Interactive Pattern Analysis and Classification," IEEE Transactions on Computers, Vol. C-19, pp. 594-616, July 1970.
9. Shick, G.J. and Wolverton, R.W., "Achieving Reliability in Large Scale Software Systems," Proceedings of the 1974 Annual Reliability and Maintainability Symposium.
10. Smith, J.E.K., "Data Transformations in Analysis of Variance," Journal of Verbal Learning and Verbal Behavior, 15, Pages 339-346, 1976.
11. Sullivan, T.E., "Measuring the Complexity of Computer Software," RADC-TR-74-325, Vol. 5, 1975.
12. Torgerson, W.S., Theory and Methods of Scaling, New York: John Wiley and Sons, 1958.
13. Van de Geer, J.P., Introduction to Multivariate Analysis for the Social Sciences, San Francisco: Freeman, 1971.
14. Honeywell Information Sciences, "MULTICS PL/I Language Manual," July 1972.

APPENDICES

APPENDIX A

UNDERSTANDABILITY STUDY

A.1. INSTRUCTIONS TO SUBJECTS

The purpose of this study is to evaluate factors which enter into the reliability of computer programs. One possible factor is the ease or difficulty with which a programmer can read and understand the operation of the program.

In this study you are to assume that your task is to translate PL/I programs into another high-level language. You will be given a set of programs, one at a time. You are to study the first program until you understand it sufficiently to be able to rewrite it in another language. (You need not memorize it, only understand its operation.) The length of time you study the program will be timed, but you should take your time to read and understand the program. You will be given scrap paper which you may use to take notes, draw flowcharts, hand-simulate the program, etc.

When you have indicated that you are finished studying the program, you will be asked to rate the program from 1 to 9 on a scale of understandability. A rating of 1 will imply that you understood the program the instant you looked at it: it was trivial. A rating of 9 will imply that the program is incomprehensible: it looks like a random list of statements. Intermediate ratings will indicate that intermediate levels of effort were required in

order to understand the program. That is, the rating should roughly reflect how hard you needed to work in order to understand the program. You should try to use the entire scale from 1 to 9 to assess the understandability of the programs.

After you rate the program, you will be asked to classify the program. You will select one category from the following classification scheme, and record the appropriate number:

	No User I/O	User Output Provided	Interactive I/O
Numeric Routine	1	2	3
Non-numeric Routine	4	5	6

For example, if a program is mainly a numerical routine which interacts with the user, the appropriate classification will be class 3. (It might help you to associate Non-numeric Routines with I/O bound programs, and Numeric Routines with CPU bound programs, but remember that this association is only a guideline.)

Finally, you will be asked a question about the functioning of the program. You will have one minute to answer the question. (You will have access to the program listing at all times.) If you answer the question

correctly, you will be credited with 25¢. If you answer incorrectly, or if you do not answer in time, you will lose 10¢. The difficulty level of the question will be independent of the complexity of the particular program. After the question, you will be given a short break, and then the cycle will repeat.

Each session will last approximately one hour, in which time you should be able to study 4 programs. There will be 8 of these sessions; you will see a total of 31 programs. If you answer every question correctly, you will be paid a total of \$8.00 at the end of the study (as well as being able to charge the 8 hours to the OLPARS Reliability project). But remember that 10¢ will be deducted for every incorrect response, so be sure to study each program carefully until you feel that you understand it. If you fail to correctly answer a total of 5 of the questions, your participation in the study will terminate. However, you will be compensated for your participation.

APPENDIX B

PROGRAM DOCUMENTATION

Program Name: lastextract

Program Type: Command level routine

Program Call: Type in "lastextract"

Input Files: lastextract assumes a data base organization as described in Section 3.1.1.

Output Files: lastextract creates a file named "filedata" in the process directory. This file will contain the feature vectors corresponding to the last version of programs in the PL/I OLPARS/Reliability data base. Upon completion, this file is in a form such that if Multics/OLPARS function "fileinput" is executed, these vectors will be input into Multics/OLPARS.

Function:

lastextract is the executive routine to perform the feature extraction on the last version of all 260 programs in the OLPARS/Reliability data base.

Subroutines "list_extract", "parse", and "count_comments" are called to extract structural features from the programs in the data base.

Features:

<u>Feature Number</u>	<u>Feature Name</u>
53	Development time
54	Number of changes

Detailed Program Description: See listing

Program Name: list_extract

Program Type: Subroutine

Subroutine Call: Call list_extract (listptr, ccount, features)

Input Parameters:

<u>listptr</u>	Pointer to the cross-reference listing of a program
----------------	---

<u>ccount</u>	Number of characters in the cross-reference listing
---------------	---

Output Parameters:

<u>features</u>	An array of features corresponding to the extracted features of this program
-----------------	--

Function:

list_extract extracts the following structural features from the cross-reference listing of a PL/I program:

<u>Feature Number</u>	<u>Feature Name</u>
5	Number of lines
8	Number of variables declared
9	Number of variables declared but not referenced
13	Mean variable name length
16	Distribution of variable occurrences vs. program statements
19	Number of pointer variables
20	Number of based variables
21	Number of implicitly declared variables
22	Number of explicitly declared variables
24	Number of external calls
25	Number of external procedures used
26	Mean number of formal parameters in procedures
27	Mean number of actual parameters in call statements
29	Number of global variables
32	Number of arrays
33	Total number of dimensions of arrays

Detailed Program Description: See listing

Program Name: count_comments

Program Type: Subroutine

Subroutine Call: Call count_comments (p01ptr, cc, num_lines,
features)

Input Parameters:

p01ptr Pointer to the source listing of a PL/I program

cc Number of characters in the source listing

num_lines Number of lines in the source listing

Output Parameters:

features An array of features corresponding to the
extracted features of this program

Function:

count_comments extracts the following structural features (relating to
comments) from the source listing of a PL/I program:

Feature Number

Feature Name

1	Number of comments
2	Average length of comments
3	Average density of non-blank characters within comments
4	Uniformity of distribution of comments vs. statement lines
6	Average density of non-blank characters outside comments

Detailed Program Description: See listing

Program Name: clt

Program Type: Subroutine

Subroutine Call: Call clt (firstname, lastname, feature)

Input Parameters:

<u>firstname</u>	The name of the first version of the file to be compared
------------------	--

<u>lastname</u>	The name of the last version of the file to be compared
-----------------	---

Output Parameters:

<u>feature</u>	The number of lines of code that have changed
----------------	---

Function:

clt compares the first version of a PL/I source program to the last version of the program and returns the number of lines of code that have changed.

Detailed Program Description: See listing

Program Name: extractem

Program Type: Command level routine

Program Call: Type in "extractem"

Input Files: extractem assumes a data base organization as described in Section 3.1.1.

Output Files: extractem creates a file named "filedata" in the process directory. This file will contain the feature vectors corresponding to the first compilable versions of programs in the PL/I OLPARS/Reliability program data base. Upon completion, this file is in a form such that if Multics/OLPARS function "fileinput" is executed, these vectors will be input into Multics/OLPARS.

Function:

extractem is the executive routine to perform the feature extraction on the first compilable version of all 260 programs in the OLPARS/Reliability data base. Subroutines "list_extract," "parse," and "count_comments" are called to extract structural features from the programs in the data base. "extractem" then extracts the following two non-structural "lastextract" then extracts the following two non-structural features:

Feature Number

Feature Name

53

Development time

54

Number of changes

Detailed Program Description: See listing

Program Name: middle

Program Type: Command level routine

Program Call: Type in "middle"

Input Files: "middle" assumes that in file "filedata1" in the process directory is a single node of 155 vectors with 52 dimensions in the MOOS function "fileinput" format.

Output Files: "middle" creates a file called "filedata" in the process directory. This is a file in "fileinput" format containing two nodes with the number of vectors in each node set by the user but each vector has 52 dimensions.

Function:

"middle" is a routine which operates externally to OLPARS with the function of removing N (set by user) vectors from a data set. The N vectors removed are those vectors which comprise the middle N vectors if each vector is ranked according to a selected measurement. Two classes are created, the first class representing the first $(155-(N/2))$ vectors of the ranked list, and the second class representing the final $(155-(N/2))$ vectors. This

routine was used to create the two classes each of low and high development time, number of changes, understanding latency, and understandability rating. Such a routine may not be needed in a future version of OLPARS.

User Interaction:

"How many vectors are to be deleted from the middle?"

N

"Enter 2 new 4 character class names"

rrrr

uuuu

"What feature is to be used?"

n

Detailed Program Description: See listing

APPENDIX C

DATA BASE GENERATION

The OLPARS programs were prepared on RADC's Multics computer facility (HIS 645, later upgraded to HIS 6180). As a monthly procedure, the Multics operator saves all files (programs and data) that exist on the system at that time. In addition, a tabular listing of all saved files is generated. This listing contains the tape numbers of the tapes used, and the names of the directories and segments saved on each of the tapes.

The tapes and listings prepared by Multics provided the historical data required by this project. By comparing earlier and later versions of the programs, an estimate could be made of the number of corrections required for each program. In addition, it was possible to estimate development time for the programs, by determining the time at which the program first appeared and the time at which corrections were no longer required. It was also possible to locate the earliest versions of the programs, and thus to determine those features which were most likely to require correction.

It was first necessary to locate and restore those programs required for this study, by searching the archived listings for all applicable segments. This task was somewhat simplified, since the names of the directories used in the OLPARS development were known, together with the dates at which the directories were active. A list of the tape numbers, directory names, and segment names to be restored was then generated. This list was supplied to the Multics operator, who then restored the required files in the Multics system.

The next task was to organize the restored programs into an easily accessible form (i.e. to create the program data base). A sub-directory (in the form "program_name".all) was then created for each unique program name. There are currently 260 such sub-directories, representing the 260 programs considered for this study.

After the files were restored by the Multics operator, there were many copies and versions of each program. Every copy of each PL/I program was then placed in the sub-directory corresponding to the program's name. Prior to insertion into the sub-directory, a unique two-character extension was added to the program's file name, so that each copy of each program would have a unique name. This extension was based on the tape, directory, and archive from which the program came. The assigned extension name is such that if an ASCII sort is performed on all of the programs using the extension name, a chronological sequence of all copies of the program will be the result.

The new name assigned to the program is of the form "program_name.extension.pl1". The copy of the program, with the new name, was then inserted into its appropriate sub-directory. Each sub-directory was then sorted on the extension to give a chronological history of that program. Identical copies of each program were deleted, with the result that each sub-directory contained every unique copy of that program sorted in chronological order. The first compilable version of each program was then identified, and all other versions of that program were archived to save space.

In summary, then, each sub-directory represents each program to be used in this study. Each sub-directory contains every unique version of that program listed in chronological order by copy date.

APPENDIX D

CLASSIFICATION BASED ON PROGRAMMING STYLE

In Section 4.2.2., program features which influence program quality were determined, where program quality was variously defined as development time, number of changes, psychological complexity rating, and understanding latency. In this appendix, an attempt is made to classify programs by author. It was assumed that each of the four authors wrote programs using individual combinations of the various structural features. If the programs of the four programmers differed systematically in regard to quality of their programs, it might be possible to relate the success of an author's programs to the use (or elimination) of certain specific features. This section presents an analysis, performed using OLPARS, which attempted to distinguish the programs by author.

The four programmers varied in the number of programs each wrote, as can be seen in Figure D-1 (treedraw). Before the classification analysis began, it was important to determine whether any significant differences existed among the four sets of programs in regard to the four measures of program quality. In order to assess the presence of systematic differences, a one-way analysis of variance was computed for each of the measures of program quality. (A multivariate analysis of variance would have been a more appropriate statistic, but since a program to perform this analysis was not readily available, individual univariate analyses were used.) The analysis of variance tables for each of the program quality variables are shown in Table D-1. There were no significant differences between the programs written by the

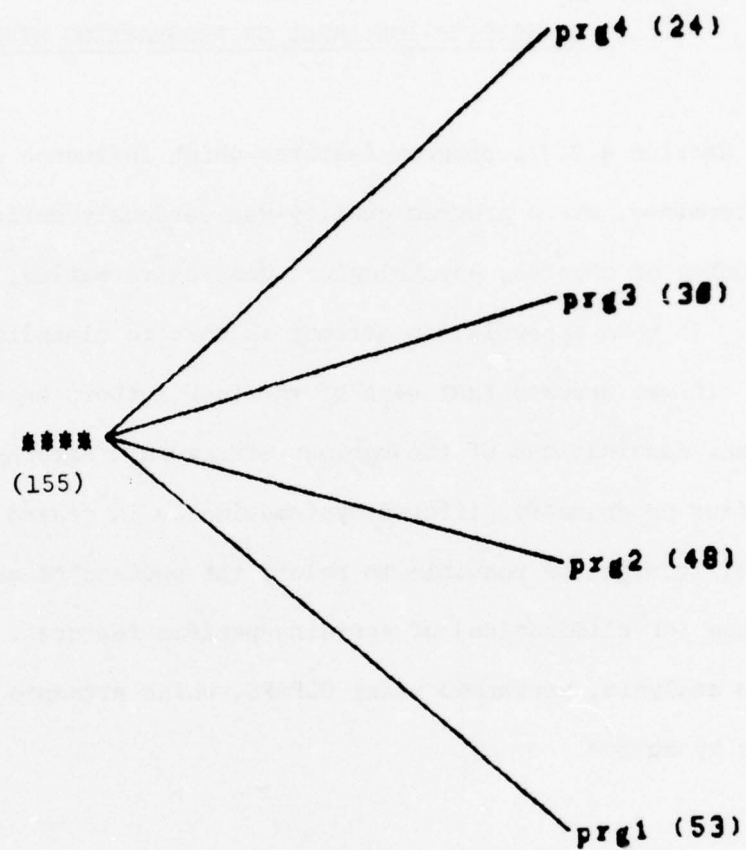


Figure D-1 Data Tree for Programmer Style Analysis

<u>Source of Variation</u>	<u>Degrees of Freedom</u>	<u>Sums of Squares</u>	<u>Mean Squares</u>	<u>F</u>
Development Time	3	22.09	7.365	1.58*
S/Development Time	151	703.10	4.656	-
Number of Changes	3	13619.195	4539.73	1.37*
S/Number of Changes	151	500306.661	3313.29	-
Latency	3	.269	.090	1.58*
S/Latency	151	8.61	.057	-
Ratings	3	20.09	6.697	2.31*
S/Ratings	151	438.64	2.90	-

* $p > .05$

Table D-1 Analysis of Variance Tables for Assessing Differences in Program Quality Among the Programmers

four programmers in terms of any of the measures. From another perspective, there was as much variation in the quality of an individual author's programs as there was across the programs of the four authors. The implication was that the subsequent style analysis would still discover how the programmers varied among themselves, but the variation would be strictly stylistic, and not related to program quality.

D.1. CLASSIFYING USING 32 FEATURES

The first goal was to classify the programs correctly by programmer using all 32 structural features. By examining the discriminant direction ($\text{gndv}\$1d1$) for all 4 programmer classes simultaneously (see Figure D-2), it can be seen that programs written by programmers 1 and 3 are relatively discernable while those of programmers 2 and 4 are not. The first step was thus to separate programs written by programmer 3 from all others. This was accomplished by finding the direction which maximally discriminated programs of programmer 3 from those of programmer 1, 2, and 4 ($\text{ardg}\$1d1$ where group 1 = programmer 3 and group 2 = programmer 1, programmer 2, and programmer 4). After accomplishing this, a threshold was set ($\text{dra}\$bndy$) for separating out programs of programmer 3 (see Figure D-3). On the design set, with this threshold value, correct classifications were made 90% of the time.

The next step was to determine which class was most discriminable of the remaining classes 1, 2, and 4. By examining the discriminant direction ($\text{gndv}\$1d1$) for programs of programmers 1, 2, and 4 simultaneously (see Figure D-4), it can be seen that programs of programmer 2 are discriminable from those of programmers 1 and 4. From this, the next step was to separate

Figure D-2 Discriminant Direction for All Classes Superimposed

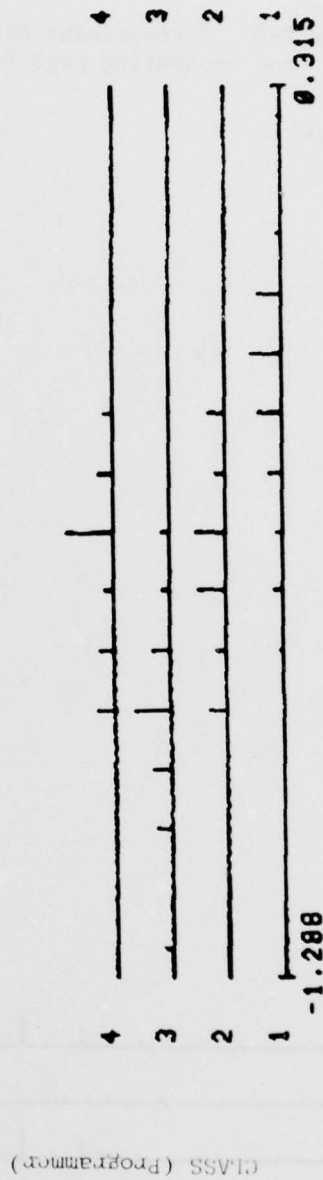


Figure D-3 Discriminant Direction and Threshold Value
for Separating prg3 from (prg1, prg2, prg4)

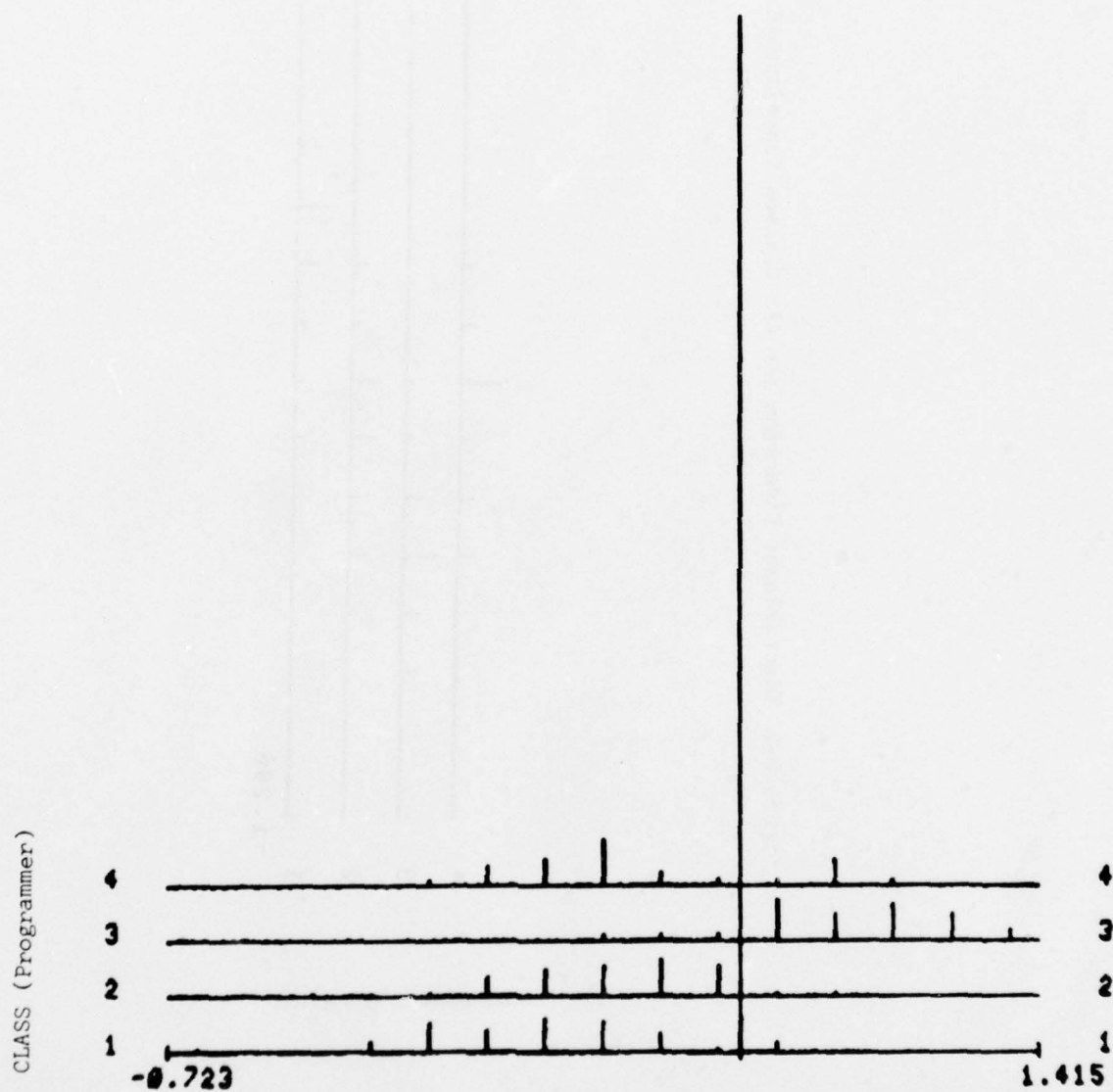
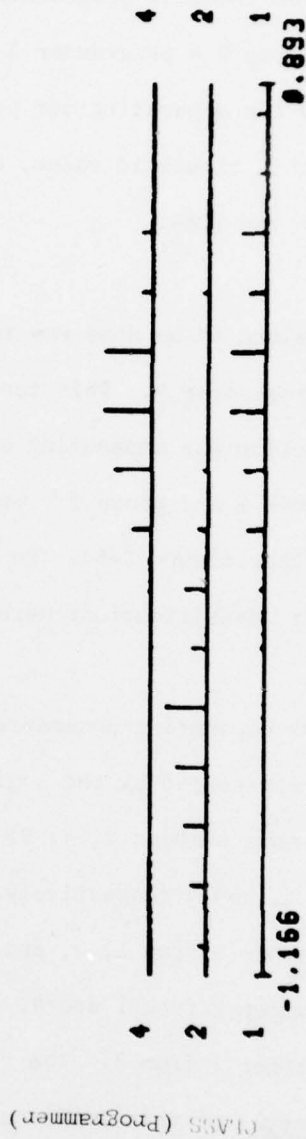


Figure D-4 Discriminant Direction for Programs of Programmers 1, 2, and 4
Superimposed



programmer 2 by finding the direction which maximally discriminates programs of programmer 2 from those of programmers 1 and 4 (ardg\$ld1 where group 1 = programmer 2 and group 2 = programmer 1 and programmer 4). Next, a threshold was set (dra\$bndy) for separating out programmer 2 (see Figure D-5). On the design set, with this threshold value, correct classifications were made, at this level, 93% of the time.

All that remained to be done was to discriminate programs of programmer 1 from those of programmer 4. This task was accomplished by examining the discriminant direction for separating class 1 from class 4 (ardg\$ld1 where group 1 = programmer 1 and group 2 = programmer 4) and setting a threshold value (dra\$bndy) (see Figure D-6). On the design set, with the selected threshold, correct classifications were made, at this level, 95% of the time.

The logic for separating programmers is now complete, and the logic designed can be represented by the logic tree of Figure D-7 (drawn by draw\$log), where node numbers 2, 4, 6, and 7 represents a classification to programmer 3, 2, 1, and 4 respectively. Node 1 represents the logic for separating programmer 3 from 1, 2, and 4. Node 3 represents the logic for separating programmer 2 from 1 and 4. Node 5 represents the logic for separating programmer 1 from 4. The final step was to perform an overall evaluation (logicev1) of this logic on the 32 dimension design set. The confusion matrix for this test is given in Table D-2 and it shows that correct classifications were made 83% of the time.

Figure D-5 Discriminant Direction and Threshold for Separating
Class 2 from Classes 1 and 4

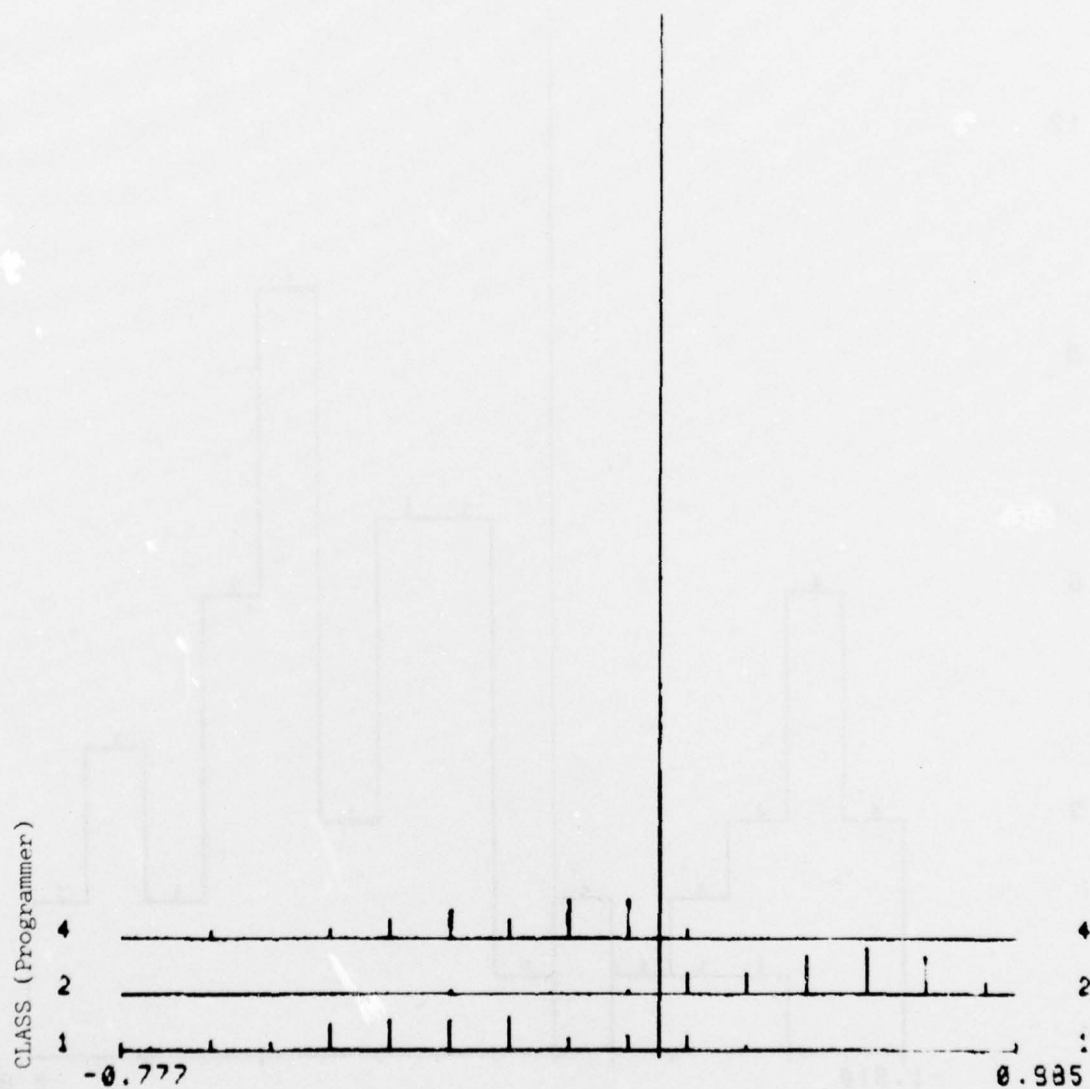
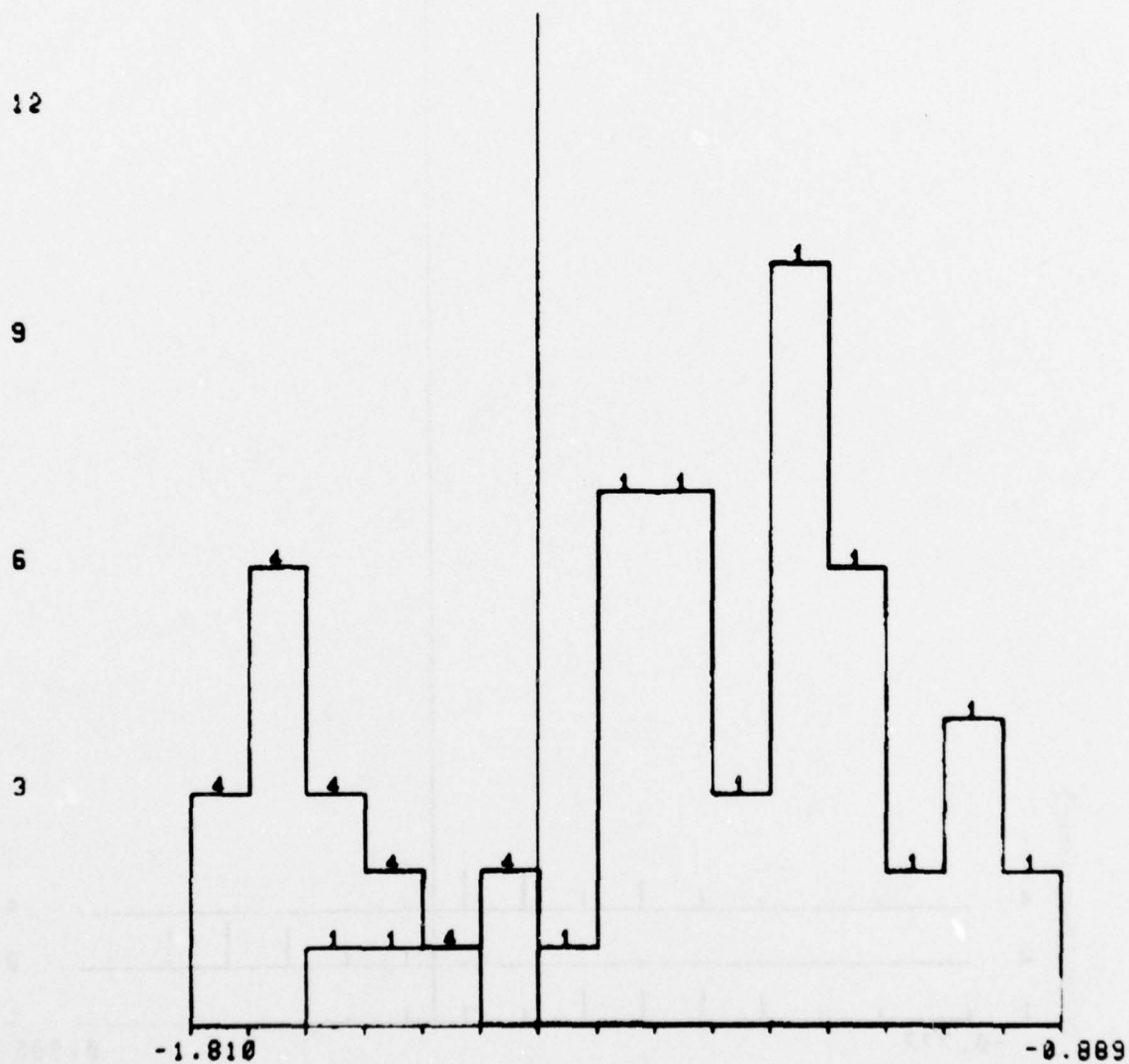


Figure D-6 Discriminant Direction and Threshold for Separating Programs of Programmer 1 from Those of Programmer 4



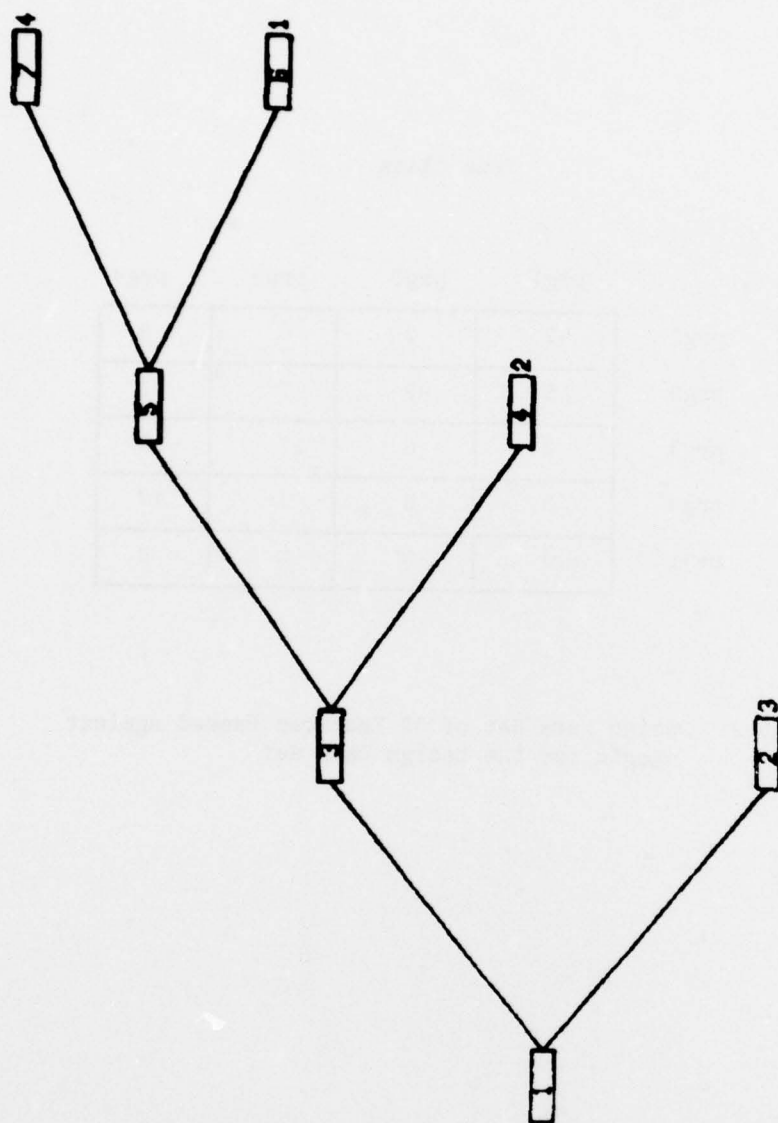


Figure D-7 Logic Tree for Separating Programs of
4 Programmers Using 32 Features

True Class

Classified		prg1	prg2	prg3	prg4
	prg1	42	2	0	0
	prg2	5	42	2	1
	prg3	3	4	27	6
	prg4	3	0	1	17
	rejt	0	0	0	0

Table D-2 Design Data Set of 32 Features Passed against
Logic for the Design Data Set

In the next two sections, the logic as described in this section will be repeated, except that at each level of the logic, the best subset of 10 features will be used for performing the discrimination. This subset of the best 10 features will be derived from the highest (absolute value) 10 coefficients of the direction which performed the discrimination.

The coefficients of the direction which separated programs of programmer 3 from those of programmers 1, 2, and 4 (corresponding to the coefficients of the direction described graphically by Figure D-3, and to the logic designed at logic node 1 in Figure D-7), and the selected 10 features are listed in Table D-3.

The coefficients of the direction which separated programs of programmer 2 from those of programmers 1 and 4 (corresponding to the coefficients of the direction described graphically by Figure D-5, and representing the logic designed at logic node 3 in Figure D-7), and the selected 10 features are listed in Table D-4.

The coefficients of the direction which separated programmers 1 and 4 (corresponding to the coefficients of the direction described graphically by Figure D-6, and representing the logic designed at logic node 5 in Figure D-7), and the selected 10 features are listed in Table D-5.

	<u>Value</u>	<u>Name</u>
1.	.045	
2.	.025	
3.	.132	distribution of comments vs. statement lines
4.	-.368	number of lines
5.	-.141	number of multiple statements of assignments used
6.	.064	
7.	.638	number of semicolons
8.	.002	
9.	.273	mean variable name length
10.	-.079	
11.	-.017	
12.	.082	
13.	-.118	
14.	-.085	
15.	.155	number of I/O statements
16.	.207	number of external procedures used
17.	-.012	
18.	-.088	
19.	-.006	
20.	-.072	
21.	-.235	number of instances of CALL
22.	-.283	number of instances of DO
23.	-.070	
24.	.048	
25.	.072	
26.	-.006	
27.	.076	
28.	-.213	average density of nonblank characters outside comments
29.	.025	
30.	-.099	
31.	.067	
32.	-.083	

Table D-3 Table of Coefficients and Selected Set of 10 Features for Separating Class 3 from Classes 1, 2, and 4
(The named features represent the reduced feature set.)

	<u>Value</u>	<u>Name</u>
1.	.088	
2.	.008	
3.	.066	
4.	-.278	number of lines
5.	.004	
6.	-.063	
7.	-.387	number of semicolons
8.	.032	
9.	.223	mean variable name length
10.	-.205	if-then-else balance
11.	-.082	
12.	.062	
13.	.349	number of assignment statements
14.	.001	
15.	-.243	number of I/O statements
16.	-.183	number of external procedures used
17.	-.065	
18.	.077	
19.	-.077	
20.	.078	
21.	.574	number of instances of CALL
22.	.0002	
23.	-.042	
24.	-.098	
25.	.0001	
26.	-.008	
27.	.142	number of based variables
28.	-.044	
29.	-.075	
30.	-.013	
31.	-.146	
32.	.151	number of instances of RETURN

Table D-4 Coefficients and Selected Set of 10 Features used for Separating Class 2 from Classes 1 and 4
(The named features represent the reduced feature set.)

	<u>Value</u>	<u>Name</u>
1.	-.070	
2.	-.040	
3.	-.091	
4.	.484	number of lines
5.	.068	
6.	-.361	number of variables
7.	-.324	number of semicolons
8.	.170	maximum nesting
9.	-.341	mean variable name length
10.	.014	
11.	-.027	
12.	-.224	mean number of operators per assignment statement
13.	.164	
14.	.099	
15.	-.007	
16.	-.047	
17.	.163	
18.	-.207	number of complex ELSE clauses
19.	-.0004	
20.	.217	number of instances of GOTO
21.	.052	
22.	-.008	
23.	-.021	
24.	-.283	number of labels
25.	-.029	
26.	-.019	
27.	-.001	
28.	-.003	
29.	.036	
30.	.190	number of global variables
31.	.145	
32.	.112	

Table D-5 Coefficients and Selected Set of 10 Features used for Separating Class 1 from Class 4
(The named features represent the reduced feature set.)

D.2. CLASSIFICATION OF PROGRAMMERS USING 10 FEATURES ON ENTIRE DESIGN SET

The next step in the analysis procedure was to rerun the logic described in Section 4.2.3.1., using only the best subset of 10 features at each level of the logic (the selected set of 10 to be used at each logic level is described in Section 4.2.3.1.).

Figure D-8 represents the direction and the selected threshold value which separates programs of programmer 3 from those of programmers 1, 2, and 4 (via ardg\$ld1 where group 1 = programmer 3, group 2 = (programmer 1, programmer 2, programmer 4), and measurement reduction where measurements used are 3, 4, 5, 7, 9, 15, 16, 21, 22, 28). With the selected threshold value, correct classifications were made 85% of the time.

Figure D-9 represents the direction and the selected threshold value which separates programs of programmer 2 from those of programmers 1 and 4 (via ardg\$ld1 where group 1 = programmer 2, group 2 = (programmer 1, programmer 4); and measurement reduction where measurements used are 4, 7, 9, 10, 13, 15, 16, 21, 27, 32). With the selected threshold value, correct classifications at this level of the logic were made 89% of the time.

Figure D-10 represents the direction and the selected threshold value which separates programs of programmer 1 from those of programmer 4 (via ardg\$ld1 where group 1 = programmer 1, group 2 = programmer 4; selected measurements used are 4, 6, 7, 8, 9, 12, 18, 20, 24, 30). With the selected

Figure D-8 - Direction and Threshold for Separating Programs of Programmer 3
from Those of Programmers 1, 2, and 4 (10 Features)

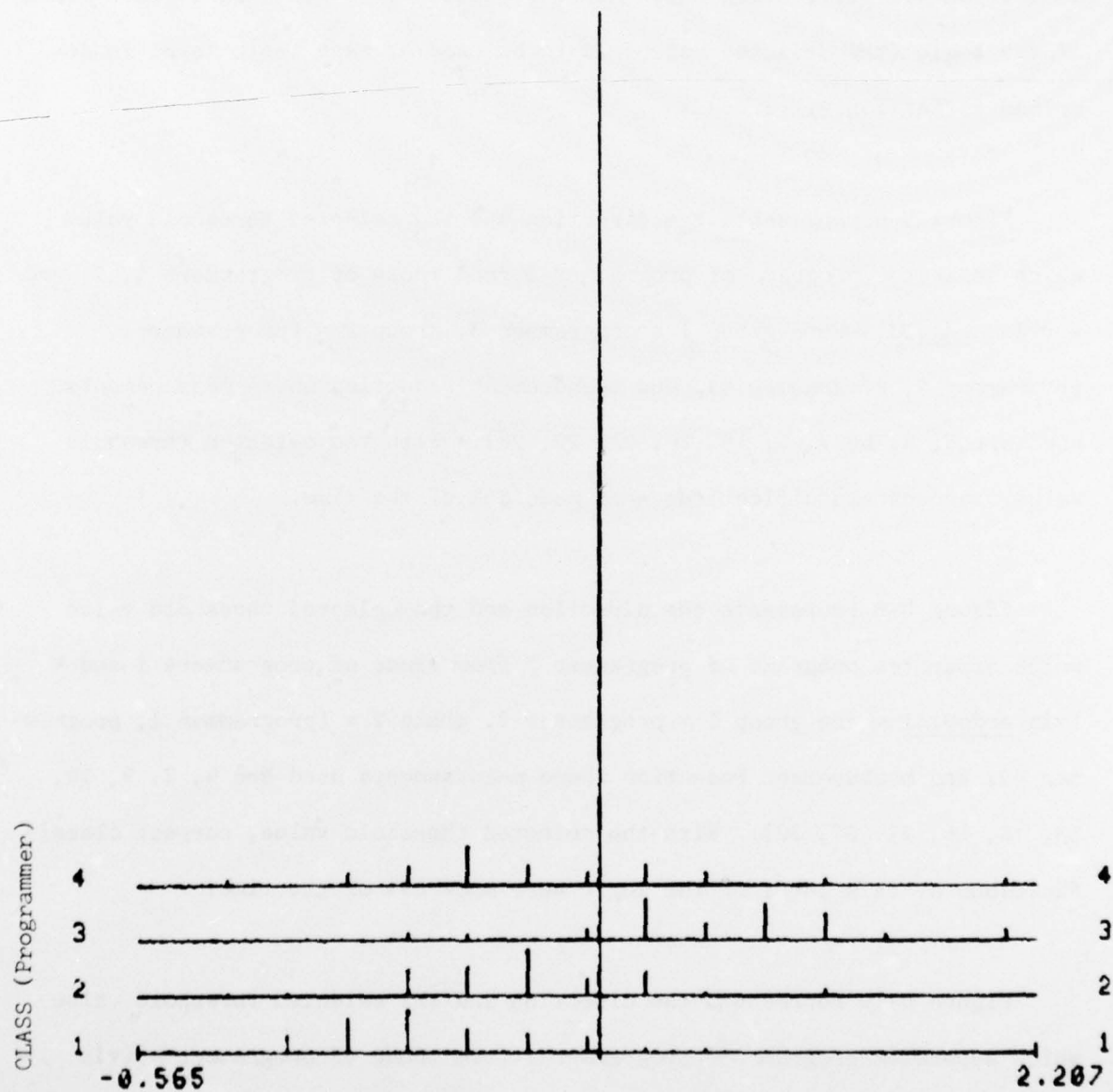


Figure D-9 Direction and Threshold for Separating Programs of Programmer 2
from Those of Programmers 1 and 4 (10 Features)

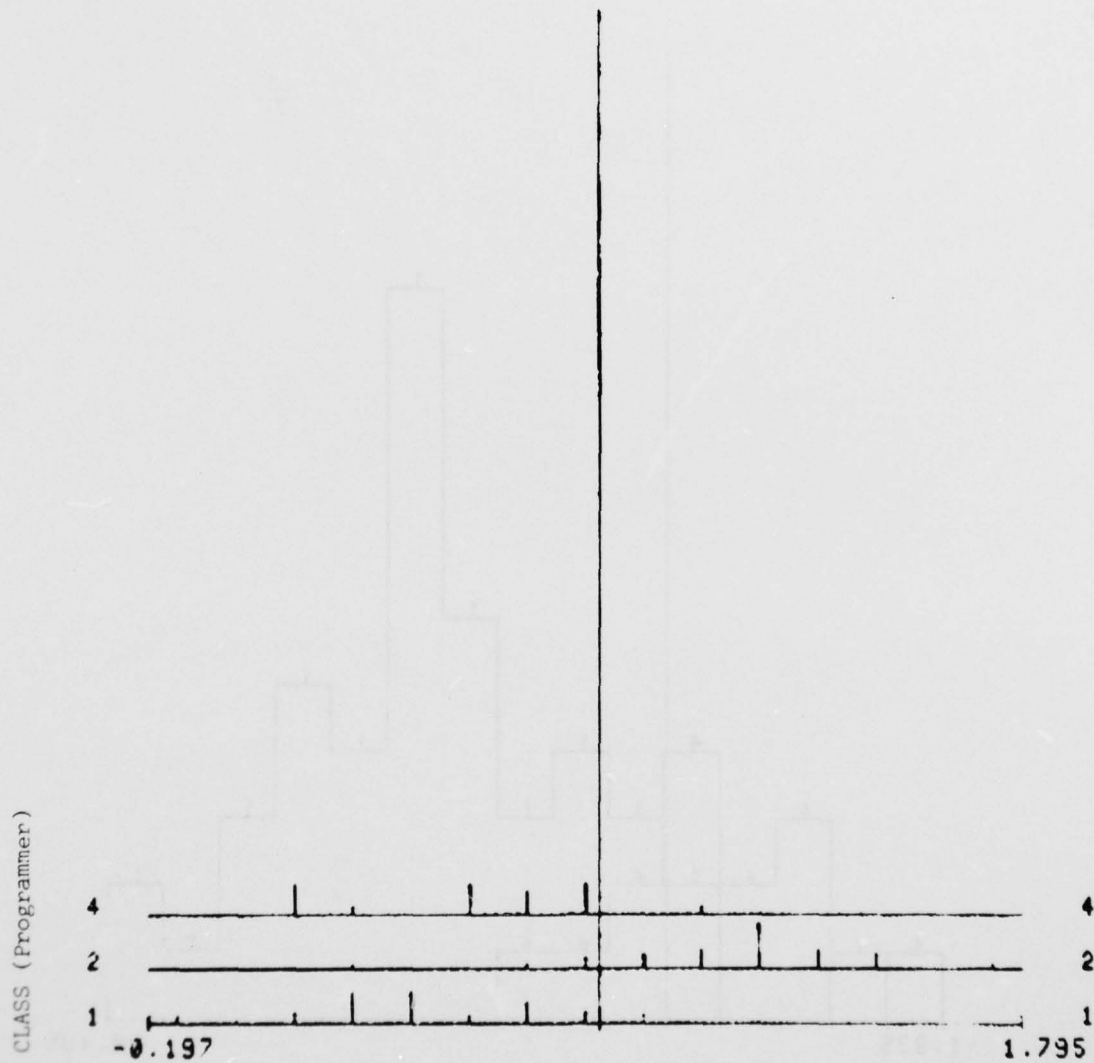
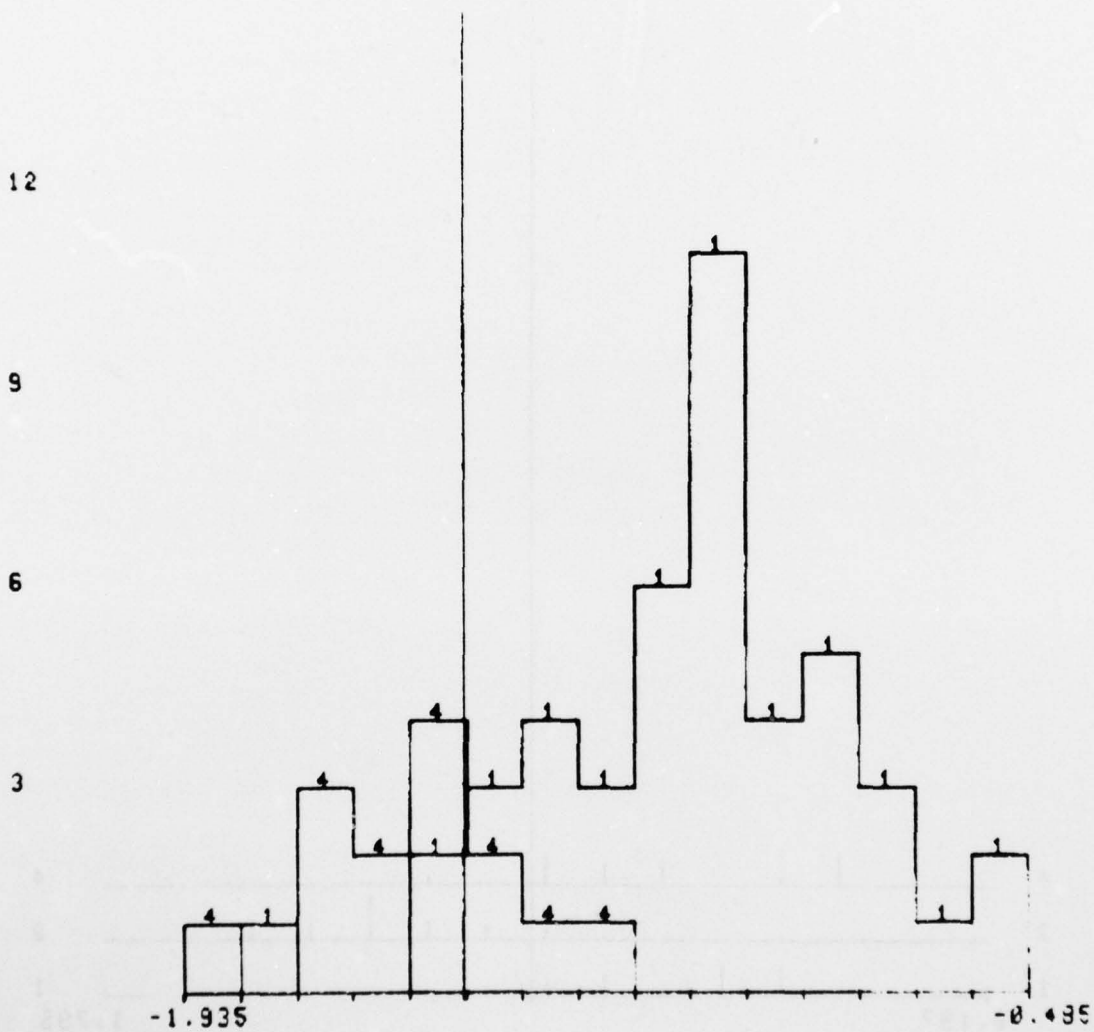


Figure D-10 Direction and Threshold Value for Separating Programs of Programmer 1
from Those of Programmer 4 (10 Features)

15



threshold value, correct classifications at this level of the logic were made 88% of the time.

The logic is complete, and an overall evaluation (logicevl) of this logic was run on the design set. Correct classifications, as shown below, were made 73% of the time.

		True Class			
		prg1	prg2	prg3	prg4
Classified	prg1	42	2	0	4
	prg2	5	33	1	3
	prg3	3	11	28	7
	prg4	3	2	1	10

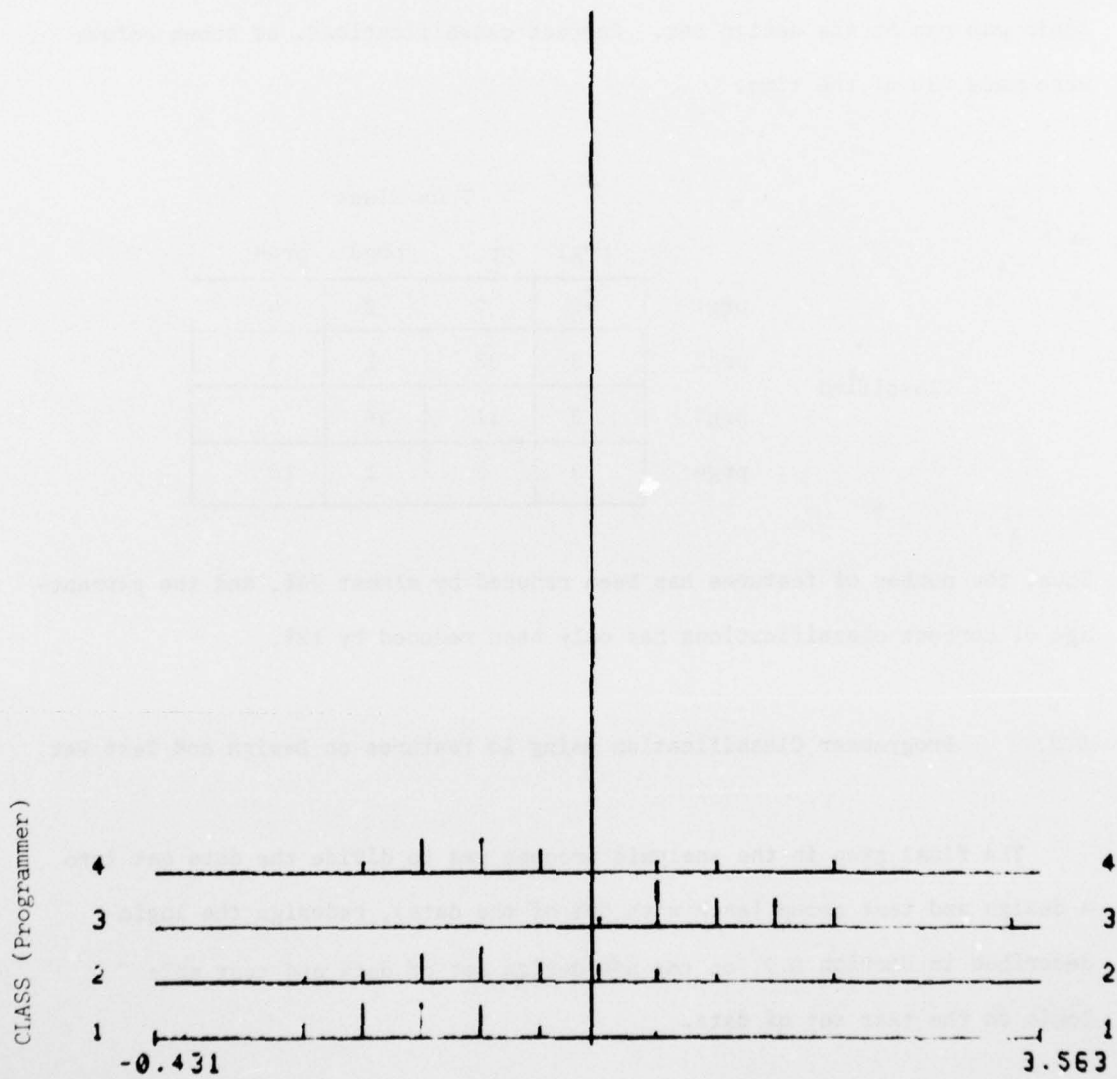
Thus, the number of features has been reduced by almost 70%, and the percentage of correct classifications has only been reduced by 12%.

D.3. Programmer Classification using 10 Features on Design and Test Set

The final step in the analysis process was to divide the data set into a design and test group (each with 50% of the data), redesign the logic described in Section D.2. on the new design set of data and test this logic on the test set of data.

Figure D-11 represents the direction and the selected threshold value which separates programs of programmer 3 from those of programmers 1, 2, and

Figure D-11 Direction and Threshold Value for Separating Programs of Programmer 3 from Those of Programmers 1, 2, and 4 (10 Features, Design Set)



4 (via ardg\$ld1 where group 1 = program 3, group 2 = (programmer 1, programmer 2, programmer 4); and measurement reduction where measurements used are 3, 4, 5, 7, 9, 15, 16, 21, 22, 28). On the design set, with the selected threshold value, correct classifications were made 90% of the time.

Figure D-12 represents the direction and the selected threshold value which separates programs of programmer 2 from those of programmers 1 and 4 (via ardg\$ld1 where group 1 = programmer 2, group 2 = (programmer 1, programmer 4); and measurement reduction where measurements used are 4, 7, 9, 10, 13, 15, 16, 21, 27, 32). On the design set, with the selected threshold value, correct classifications at this level of the logic were made 87% of the time.

Figure D-13 represents the direction and the selected threshold value which separates programs of programmer 1 from those of programmer 4 (via ardg\$ld1 where group 1 = programmer 1, group 2 = programmer 4, selected measurements used are 4, 6, 7, 8, 9, 12, 18, 20, 24, 30). On the design set, with the selected threshold value, correct classifications at this level of the logic were made 83% of the time.

The logic is now complete, and an overall evaluation (logicevl) of this logic was tested using the design data set. Correct classifications, as shown below, were made 74% of the time.

Figure D-12 Direction and Threshold Value for Separating Programs of Programmer 2 from Those of Programmers 1 and 4 (10 Features, Design Set)

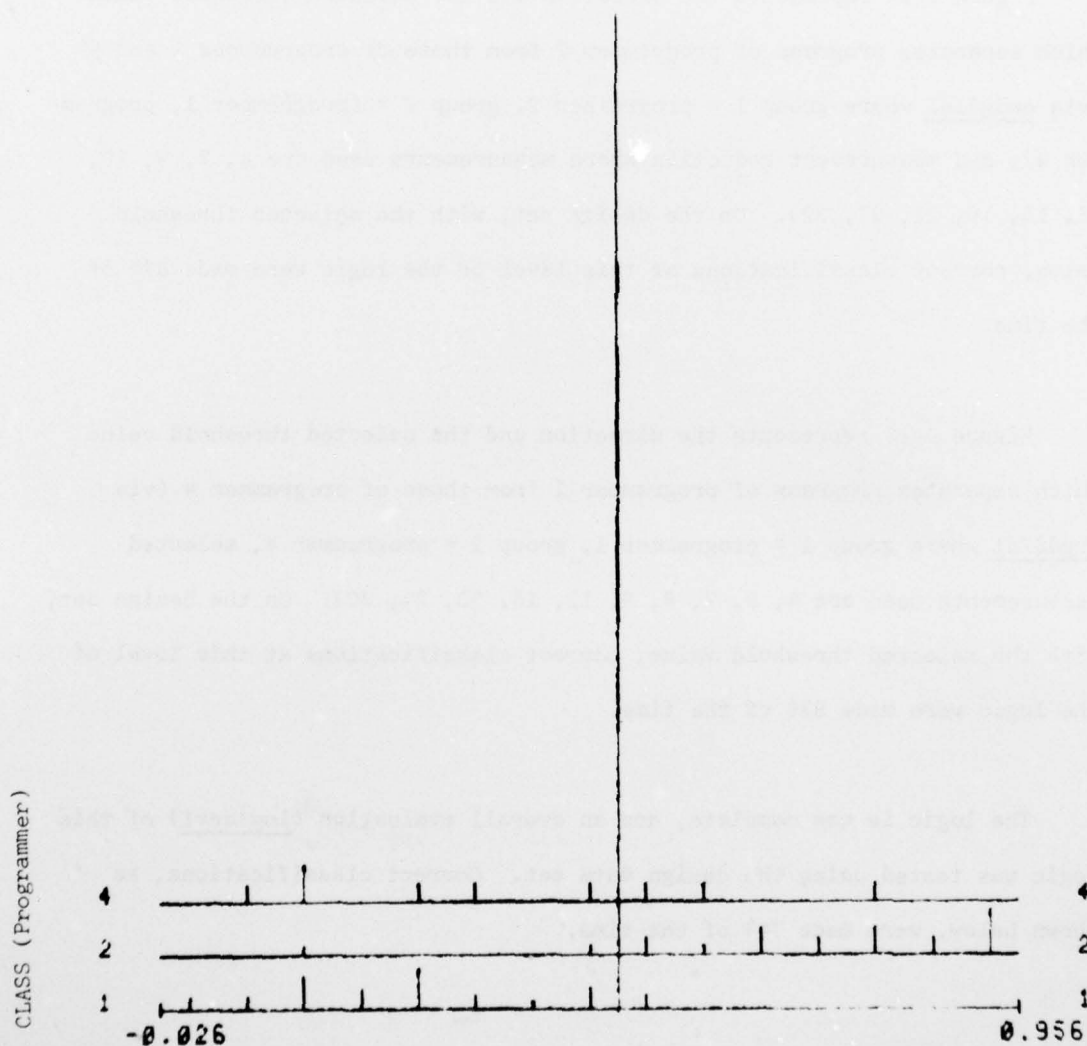
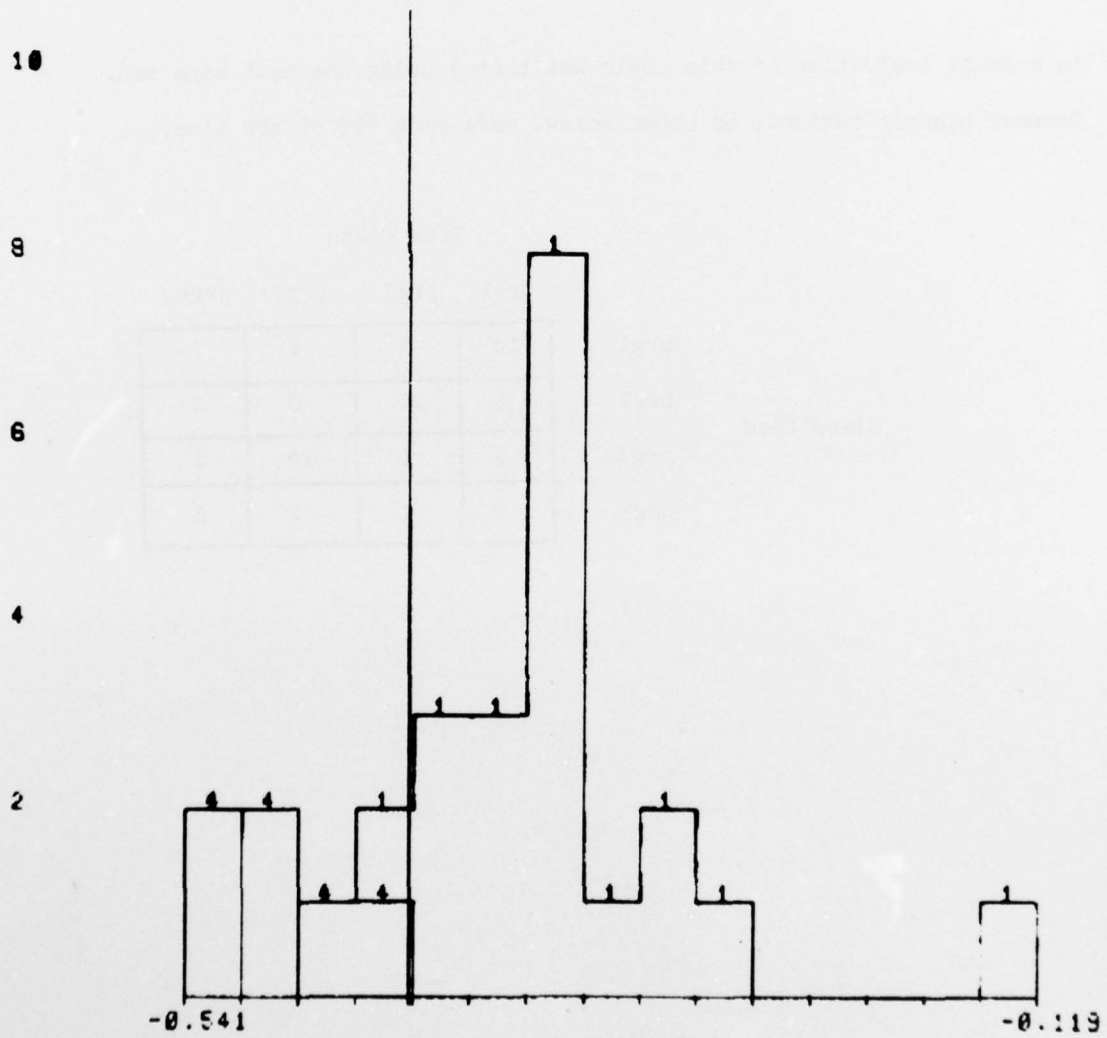


Figure D-13 Direction and Threshold Value for Separating Programs of Programmer 1 from Those of Programmer 4 (10 Features, Design Set)



		True Class			
		prg1	prg2	prg3	prg4
Classified	prg1	19	2	0	0
	prg2	2	18	0	3
	prg3	1	4	15	3
	prg4	5	0	0	6

An overall evaluation of this logic was tested using the test data set.

Correct classifications, as shown below, were made 60% of the time.

		True Class			
		prg1	prg2	prg3	prg4
Classified	prg1	10	1	0	2
	prg2	6	20	3	3
	prg3	3	2	10	1
	prg4	7	1	2	6