AN EMPIRICAL STUDY OF PROGRAMMER ACTIVITIES DURING SOFTWARE DEVELOPMENT AND INTEGRATION

by

KITTUR V. GANESH

B.E. (Mechanical Engg.), Bangalore University, India

M.S. (Industrial Engg.), Kansas State University, Manhattan, KS.

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer and Information Sciences

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

Approved by:

Major Professor

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my advisor Dr. David Gustafson, for his valuable advice, guidance, and much needed encouragement throughout this work.

I also wish to extend my thanks to Dr. William Hankley and Dr. Austin Melton for serving on the graduate committee.

Last but not the least, I thank the faculty and all my friends in the department of Computer and Information Sciences, for their support during my graduate study.

TABLE OF CONTENTS

Contents	Page
Chapter 1: Introduction	1
Chapter 2: Background Of Software Engineering.....	3
Chapter 3: Objective Of Study	8
Chapter 4: Programmer Activities - A Classification	9
Chapter 5: Data Collection and Utility Tools	16
Chapter 6: Results and Discussion	23
Chapter 7: Conclusions	31
References	34
APPENDIX A	35
APPENDIX B	54

CHAPTER 1

Introduction

During the past decade, software costs have risen dramatically, becoming a key expense in many computer based systems, and is now always a cause for concern for the software developer. In a software market that is very competitive, the key to success lies in the efficient utilization of hardware and software tools, including manpower and time.

In order to accomplish a successful software development project, we must understand the scope of work to be done, the resources required, the tasks to be accomplished, the costs to be expended, and most important the schedule to be followed. For this, estimates have to be made and these estimates can be accepted only with some degree of uncertainty, which of course seems natural. Also, since there is no rigid historical data available to be used as a guide, the estimation of time schedules and milestones to be tracked are most often subject to change throughout the software development process, depending upon the progress with which the software development process is heading.

It is very important to assess the progress of a project during the different phases of development and is essential in order to determine if adjustments to project's costs and time schedules are necessary.

The project discussed in this report arrives at the time distribution of programmer activities during the course of the software development and integration phases of the software development cycle, based on an analysis of patterns of changes in source code of successive document versions , each version being the result of the systhesis of prior versions. The project is conducted by monitoring the development cycle of actual software projects.

The programmer activities are classified empirically based on past data obtained through earlier research on software projects in the department of Computer and Information Sciences, at Kansas State University.

The time distribution of programmer activities should enable the project personnel to visualize the amount of time spent on each of the activities, thereby helping them to take appropriate action depending on the importance of the activity within the context of the nature of software being developed.

CHAPTER 2

Background Of Software Engineering

2.1 Definition of Software Engineering

One of the early definitions of Software Engineering was proposed by Fritz Bauer of Technical University, Munich, Germany, and is defined as "the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines " [Bau72]. Many more comprehensive definitions have been proposed, all reinforcing the importance of engineering discipline in software development.

Software Engineering is an outgrowth of hardware and system engineering. It encompasses a set of three key elements - methods, tools, and procedures that enable the manager to control the process of software development and provide the software practitioner with a foundation for building high-quality software in a productive manner.

2.2 The Three Phases Of Software Engineering

The software development process contains three generic phases, which forms the basis of a software engineering methodology that is application dependent. The three phases, definition, development and maintenance, are all encountered in

all software development, regardless of application area, project size, or complexity.

2.2.1 Definition Phase:

The definition phase is the starting point of software development. In this phase, the user's requirements are carefully analyzed, so that the resources required to develop the software are predicted, and cost and schedule estimates are established. After analysis, the user's requirements are transformed into what is known as Software Specification. Alternately, a prototype is built and evaluated by the user in an attempt to solidify requirements. Both, the user and the software developer are jointly involved in the software requirement analysis and definition.

At the end of this phase, products like a software plan, an optional prototype development, quality assurance and software verification plans are obtained.

2.2.2 Development Phase :

This phase, as illustrated in Figure 1, translates a set of requirements into an operational system element that we call "Software". In the development phase, three specific steps always occur in some form :

1. Software Design : Design involves translating the software requirements specification into descriptions of architectural and data design. That is, the system is organized

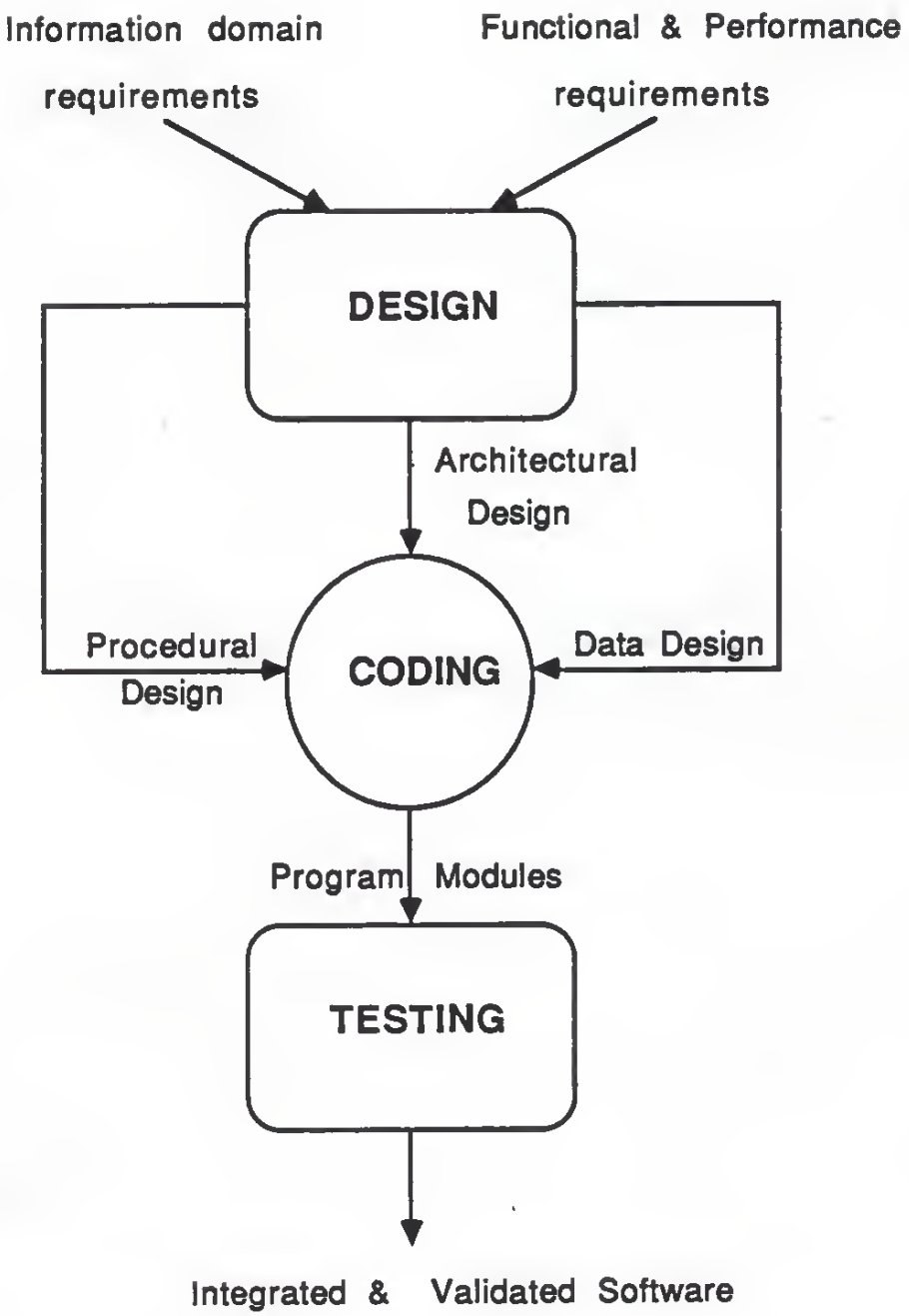

FIGURE 1 : The Development Phase In Software Engineering.

into a modular structure, and through the data flow analysis, the data structures are established. The design phase has two sub phases : the preliminary design and detailed design. The preliminary design phase involves identifying the various modules and defining the software hierarchy. During the detailed design phase, formal module specifications are developed, explaining in detail the function of each module in the software structure.

2. Coding : This phase follows immediately after the design phase. Software Engineering views coding as a consequence of good design. Here, each module is coded using an appropriate programming language. It is ensured that the code is directly tracable to the algoritms produced during the design phase in addition to adhering to the corresponding module specification. This phase actually constitutes only fifteen percent of the total time spent [Gilbert 1983].

3. Software Testing : Software testing is indeed a critical element of software quality assurance and it attempts to validate the correctness of the software package. In fact, it is the final review of the specification, design, and coding stages. The increasing visibility of software as a system element and the costs associated with a software failure are the motivating forces for well planned, thorough testing. The objective of testing is to execute a program with the intent of finding an error. If testing is conducted successfully, it will uncover

errors in the software. It also demonstrates that the software functions appear to be working as per the specification and that performance requirements have been met. There are three categories in testing :

1. Unit testing : attempts to validate the functional performance of individual modules. It is always white box oriented; that is, it uses the control structure of the procedural design to derive test cases.

2. Integration testing : This checks the correctness of the interfaces after the individual modules are integrated into an overall system, in terms of the parameters, global variable effects, initialization of variables etc. This uncovers the interface errors between modules.

3. Validation testing : This is done after the software is assembled as a package, to ensure whether the software requirements are met as per the specifications outlined in the definition phase. The approach used here is one of black box testing, which focuses on functional requirements of the software. It attempts to uncover errors in data structures or external database access, performance errors, interface errors, missing functions, initialization and termination errors etc. Thus, attention in this phase is focussed on the information domain.

2.2.3 Maintenance Phase

After the development phase is completed, a review is made to ensure that all documentation is available for maintenance tasks to follow. This phase begins prior to the release of software and continues throughout its useful life. During this phase, errors are corrected, adaptations are effected, and enhancements are implemented.

CHAPTER 3

OBJECTIVE OF STUDY

The objective of this study is to make an empirical analysis of Programmer Activities (explained in subsequent sections) during the software development and integration phases of actual software projects.

The software project per se is made of a series of document histories (different programs) and each document history in turn made up of a set of versions of one document(program), each version resulting from the synthesis of prior versions.

Taking into consideration that the software development process is viewed as the process of refining a set of document versions throughout the product evolution, and the process of development as being time-dependent, this study attempts to arrive at the time distribution of the various programmer activities during software development. The distribution is based on the patterns of changes in source code of successive document versions of the programs developed during software development.

CHAPTER 4

PROGRAMMER ACTIVITIES - A CLASSIFICATION

The classification of program change patterns, hereafter referred to as the Programmer Activities in this study, was empirically derived in an earlier research by Yu-Hua Hsu [Hsu'88]. A detailed description of these classifications, and the basis for such a classification can be found in the master's thesis [Hsu'88].

The classification of programmer activities during the project development is actually based on the patterns of progress during its development. These patterns of progress were determined by analyzing the program change data - data concerning changes made in the successive versions of a program.

These classifications of programmer activities make it possible for a software manager to monitor the progress of the project during the different phases of its development. Also, since it is easily understandable by both technical and non-technical managers alike, it acts as a tool for making project decisions as to whether the development is progressing in the right direction, or to see if any adjustments in the project's costs and time schedules are necessary. Making proper timely decisions will go a long way in reducing the overall cost expended on the software project.

4.1 Classification of programmer activities

The program change patterns were based on a detailed analysis of data involving :

1. the difference of the counts of each statement type between a pair of non-pretty printed program versions.
2. the difference of the counts of each indentation level between a pair of pretty printed program versions.
3. the total number of statements for each type which have been modified between a pair of pretty printed program versions.
4. the total number of statements for each type which have been modified between a pair of non-pretty printed program versions.

The program itself could be in a pretty printed format or any other free style format. Also, the statement types defined were the following :

FOR statement	ASSIGNMENT statement
WHILE statement	PREPROCESSOR statement
IF statement	COMMENT statement
ELSE statement	BLANKLINE statement
SWITCH statement	RETURN statement
CASE statement	INPUT statement
GOTO statement	OUTPUT statement
BREAK statement	FUNCTION statement
CONTINUE statement	DECLARATION statement
DEFAULT statement	

Ten classifications of programmer activities were defined after a detailed analysis of data concerning program change patterns [Hsu'88] :

1. Debugging
2. Removing Debugging
3. Documentation
4. Removing Documentation
5. Pretty Printing
6. Correction
7. Reconstruction
8. Redistribution
9. Adding Functionality
10. Removing Functionality

4.2 Definition Of Programmer Activities

A brief description of the characteristics for each type of programmer activity follows.

4.2.1 Debugging

This programmer activity is said to have taken place if the program change pattern shows an increase in the counts of OUTPUT statements. These OUTPUT statements are added by the programmer so as to monitor the behavior of the program.

4.2.2 Remove Debugging

This programmer activity is said to have occurred if those statements that were added for the purpose of debugging are removed. This invariably occurs during the final stage of software development. The program change patterns show a decrease in the count of OUTPUT statements between two successive versions of a program.

4.2.3 Documentation

This activity occurs when COMMENT statements are added. This activity is said to have occurred when the program change pattern between successive versions of a program shows an increase in the count of COMMENT statements.

4.2.4 Removing Documentation

This activity occurs when the program change patterns between successive versions of a program shows a decrease in the count of COMMENT statements.

4.2.5 Pretty Printing

The following steps are first executed :

1. find the differences between the two successive versions of a program
2. repeat step 1 except that both versions are pretty printed before finding the differences.

If the results show a change pattern of more statements being different in step 1 than those in step 2 , then pretty

printing activity is said to have taken place. Pretty printing makes the structure of a program explicit.

4.2.6 Correction

This activity is said to have occurred if the program change patterns between a pair of successive versions of a program show an addition or deletion of a few lines of code of various statement types, including control statements, FUNCTION, DECLARATION, and ASSIGNMENT.

4.2.7 Reconstruction

This activity is said to have taken place if the change patterns between two successive versions of a program show a trend of decreasing or increasing number of indentation for each level. This happens when the structure of a program is altered.

4.2.8 Redistribution

This activity is said to have taken place when the change patterns between two successive versions of a program show an indication of removing FUNCTION statements and adding PREPROCESSOR statements, or vice-versa.

4.2.9 Adding Functionality

This activity is said to have occurred when the change patterns between two successive versions of a program show an addition of control statements including FUNCTION, ASSIGNMENT, PREPROCESSOR, and DECLARATION statements. This activity results in a significant increase in the lines of code.

4.2.10 Removing Functionality

This activity has taken place when the change patterns between two successive versions of a program show a deletion of control statements including FUNCTION, ASSIGNMENT, PREPROCESSOR, and DECLARATION statements. This activity results in a significant difference in the lines of code of the two versions.

4.3 Threshold Values For Programmer Activities - based on PROBIT

Threshold values for each of the programmer activities, were obtained using PROBIT, a statistical procedure available under the SAS package. A listing of the various classifications and their corresponding threshold values are shown in TABLE 4.1 .

The classification of the programmer activities and the corresponding thresholds were used in this project. Classifying the programmer activities will make it easier to monitor the progress of the project during the development cycle. The progress of the project could be easily assessed by visual inspection of the classifications.

TABLE 4.1 Classification of Programmer Activities and Their Thresholds.

Type Of Classification	Threshold Values
1. Debugging	increase of more than 5 lines of OUTPUT statements.
2. Removing Debugging	a decrease of more than 1 line of OUTPUT statement.
3. Documentation	an increase of more than 1 line of COMMENT statement.
4. Removing Documentation ..	a decrease of more than 2 line of COMMENT statements.
5. Pretty Printing	N / M greater than 0.1 where N statement types have gone through pretty printing, and M stat. types changed.
6. Correction	less than 10 lines changed, and more than 1 change in FUNCTION,DECLARATION, ASSIGNMENT, or control stmts.
7. Reconstruction	J/I greater than 0.5, where J indentation levels have been changed, and I is the highest indentation level.
8. Redistribution	changes in FUNCTION and PREPROCESSOR are in opposite directions, and > 3 lines of changes in those stmts.
9. Adding Functionality..	addition of more than 4 lines of FUNCTION, DECLARATION, ASSIGNMENT, PREPROCESSOR, or control statements.
10. Removing Functionality..	decrease of more than 2 lines of FUNCTION, DECLARATION, ASSIGNMENT, PREPROCESSOR, or control statements.

CHAPTER 5

Data Collection and Utility Tools

Chapter 4 was devoted to the classification of programmer activities. It was made clear that the classification was based on the program change patterns occurring during software development. This chapter first deals with the background of the data used (different programs and their corresponding versions) for the analysis. This is followed by a description of the various utility software tools used for analyzing the data. Using these tools, the data pertaining to the change patterns during software development, and the corresponding programmer activities (as per the classification and their related threshold values) are extracted into an output file called "class.results", from which statistical measures reflecting the progress of software development are determined.

5.1 Background of Data Used

Basically, the data used are program document histories, each program document history being made up of several versions of one document. All these programs that were used in this research were written in the language "C" and implemented on VAX 11/780 under UNIX. These programs were developed by the undergraduate students of Computer Science enrolled in CMPSC 541, a Software Engineering core course. The students were asked to

apply the concepts of software engineering methodology taught in the course during the development of projects assigned to them. Successive versions of the programs were saved on a daily basis which then served as data for this research.

Two different versions of the same program are selected for analysis at one time. 1321 program pairs were analyzed in this study.

5.2 Utility Programs For Data Collection

The following Utility programs were utilized for collecting data used for extracting the various programmer activities involved during the course of the project development.

1. MAIN.c
2. CHANGES
3. class.new
4. stat1
5. stat2
6. stat3

A description of the above utility programs follows.

5.2.1 MAIN.c (usage : r.out "inputfilename")

MAIN.c is the driver or the main program written in "C" programming language. The source listing of MAIN.c is given in Appendix A. This program takes as its input a data file which

contains the filenames of different versions of programs to be analyzed, including the date on which each of the programs was saved. Also, the data file must be ordered by filename.

The program picks up a pair of different versions of the same program and executes the shell command :

"CHANGES programfile1 programfile2 "

where CHANGES is another utility software explained later. Programfile1 and programfile2 are programs that are selected for analysis. This is repeated until all of the possible program pairs have been executed. When the end of the data file is reached, the program terminates gracefully.

5.2.2 CHANGES (usage : CHANGES programfile1 programfile2)

This program was developed in an earlier research [Hsu'88]. This project is actually an extension of Yuhua's research. A detailed description of the CHANGES program can be found in [Hsu'88]. A brief description of its purpose follows.

CHANGES is actually a C-shell program and is used for collecting data for program change analysis. It utilizes data manipulation tools such as diff (for finding the differences between two files), grep (for matching patterns in a set of files). In addition, it also uses the C-shell command "cb" which is used for beautifying a program into an appropriate indentation format.

This program takes two programs as its input and

generates an output file called "main.results". As an example, refer to the file main.results in Appendix A, obtained by executing the CHANGES program on two program files called "examplefile1" and "examplefile2". The source code for these sample files are shown in Appendix A. The file main.results shown in Appendix A contain the results of the analysis carried out on a pair of different versions of the same program, namely examplefile1 and examplefile2. In the file, it could be seen that it is divided into four blocks separated by double broken lines. The first block of data are the statistics for the first program of the program pair analyzed. It basically is the count of the total number of occurrences of the statement types and that of the levels of indentation for each line of code which ranges from 1 to 6. The second block contains the same data as the first block except that the data pertains to the second program of the program pair. The third block of data is the statistics for the changes found in the program pair, both of these programs pre-processed by the "cb" command. It contains the differences between the two programs analyzed in terms of the total number of occurrences of each of the statement types(defined in chapter 4). The fourth block of data is same as that in the third block, except that the programs are not pre-processed with the "cb" command. The file main.results acts as an input to the utility program "class.new".

5.2.3 class.new (usage : class.new main.results)

This program is basically an Awk program. Awk is a UNIX data manipulation tool and is a pattern matching language and a report generator. This program takes as its input the file generated by CHANGES namely main.results.

Based on the change patterns in the file main.results, and using the threshold values attached to each of the programmer activities (as per the discussion in chapter 4 on classifications), an output file called "class.results" as shown in the Appendix A is generated.

The file class.results thus contains the name of the second program of the program pair analyzed, the date on which the program was saved, and the corresponding programmer activities generated as a result of the analysis. This file class.results is an important output file as it acts as an input file to the utility software stat1, stat2, stat3, all of which determine appropriate statistical measures which will enable the software managers to assess the progress of the programs during the development stage.

5.2.4 extract (usage : extract class.results)

"extract" is a short awk program. The source code is shown in Appendix A. Its purpose is to extract all those data that belong to the months of january, february, march, april and may; and store the data in corresponding files called "jan",

"feb", "mar", "apr", "may" respectively. Basically, data now pertain to the development activities during those individual months. Each of these files are fed as input to the utility software stat1, stat2 and stat3 for obtaining time distribution of the various activities during development for those individual months.

5.2.5 stat1 (usage : stat1 "programfile")

"stat1" is again an Awk program. It takes as its input a file in the format of class.results, and calculates the percentage of the time distribution of each of the programmer activities, based on the total count of the activities generated in the programfile. It yields a measure of the total EFFORT involved in the software development stage. For example, the percentage of time distribution for DEBUGGING activity would be :

$$\frac{\text{total number of occurrences of DEBUGGING activity}}{\text{total count of all the activities generated}} * 100$$

5.2.6 stat2 (usage : stat2 programfile)

This program is also an Awk program, and takes as its input a file in the format of class.results. It calculates the number of times each of the programmer activities was involved in terms of the changes between different versions, based on the total number of program pairs analyzed. The values obtained are converted into a percent form for ease of interpretation. For example, the percent of times the DEBUGGING activity was involved

would be :

$$\frac{\text{total number of occurrences of DEBUGGING activity}}{\text{total count of all the program pairs analyzed}} * 100$$

5.2.7 stat3 (usage : stat3 programfile)

"stat3" is an Awk program that takes as its input any file having the format as class.results. The purpose of this program is to determine a measure of the involvement of each of the programmer activities in any program development, based on unique number of programs analyzed. As an example, for DEBUGGING activity say, the measure would be :

$$\frac{\text{total number of unique programs involving DEBUGGING}}{\text{total count of all the program pairs analyzed}} * 100$$

The total number of unique programs involving an activity is counted in such way that, if there is more than on count for a particular unique program, then the count is incremented only once.

As mentioned earlier, 1321 program pairs were analyzed using all the utility software described in the above sections. The program pairs were distributed in the different periods of software development as follows: 213 in January, 141 in February, 62 in March, 553 in April, and 352 in May. The next chapter discusses the results obtained, including the interpretations of the measures for determining the progress of the software during its development.

CHAPTER 6

RESULTS and DISCUSSION

6.1 Results

As mentioned in the previous chapters, 1321 program pairs were analyzed in this study. The change patterns of programs between successive versions were first obtained. The programmer activities based on the threshold values discussed in chapter 4 were then obtained for three cases. Case 1 was based on the count of total number of activities generated in the output data; Case 2 was based on the total number of program pairs analyzed; and Case 3, was based on the total number of unique programs analyzed. The three cases will be referenced as Case 1, Case 2 and Case 3 respectively in further discussion. Tables 6.1 through 6.3 give the results obtained for all three cases. Notice that there are seven columns in the tables. The first column lists the various programmer activities, and the other columns give the time distribution of these activities for different periods of development.

Appendix B lists the relevant graphs obtained using the results shown in tables 6.1 through 6.3. These graphs give a visual description of the patterns of progress during the software development stage. A complete analysis of the results obtained follows in the next section.

TABLE 6.1 : Time Distribution Of Each Activity/Classification
(In Percent - Based on total Activities)

	entstat	janstat	febstat	marstat	aprstat	maystat
DBG	5.67	5.65	8.59	17.89	5.71	3.13
RDBG	2.45	2.93	2.73	1.05	2.37	2.35
DOC	7.34	7.11	5.08	7.37	5.71	10.84
RDOC	0.99	1.05	1.17	1.05	1.06	0.78
PPTR	43.62	41.63	50.00	42.11	42.20	45.17
RECN	13.12	12.13	10.94	9.47	14.53	12.66
REDN	0.04	0.21	0.0	0.0	0.0	0.0
CORR	13.26	13.39	8.98	10.53	13.47	14.62
AFNC	8.05	6.69	7.42	4.21	9.88	6.66
RFNC	5.46	9.21	5.08	6.32	5.06	3.79

TABLE 6.2 : Time Distribution Of Each Activity/Classification
(In Percent - Based on total Program Runs)

	entstat	janstat	febstat	marstat	aprstat	maystat
DBG	12.11	12.68	15.6	27.42	12.66	6.82
RDBG	5.22	6.57	4.96	1.61	5.24	5.11
DOC	15.67	15.96	9.22	11.29	12.66	23.58
RDOC	2.12	2.35	2.13	1.61	2.35	1.70
PPTR	93.11	93.43	90.78	64.52	93.49	98.30
RECN	28.01	27.23	19.86	14.52	32.19	27.56
REDN	0.08	0.47	0.0	0.0	0.0	0.0
CORR	28.31	30.05	16.31	16.13	29.84	31.82
AFNC	17.18	15.02	13.48	6.45	21.88	14.49
RFNC	11.66	20.66	9.22	9.68	11.21	8.24

TABLE 6.3 : Time Distribution Of Each Activity/Classification
(In Percent - Based on Unique Program Runs)

	entstat	janstat	febstat	marstat	aprstat	maystat
DBG	15.43	18.60	14.08	43.33	16.78	8.49
RDBG	8.27	9.30	9.86	3.33	7.53	8.49
DOC	24.12	23.26	12.68	20.00	20.89	32.08
RDOC	3.65	3.88	4.23	3.33	3.77	2.83
PPTR	96.49	93.80	95.77	83.33	96.58	99.53
RECN	34.08	30.23	23.94	26.67	38.70	33.49
REDN	0.14	0.78	0.0	0.0	0.0	0.0
CORR	34.64	31.01	23.94	30.00	36.99	36.79
AFNC	23.14	19.38	21.13	13.33	29.45	17.45
RFNC	17.11	26.36	12.68	16.67	17.12	12.74

6.2 Discussion of Results Obtained

Figure 1 (Appendix B) shows the time distribution of the various programmer activities considering the output data generated during the entire software development stage, for all the three cases (defined earlier). It leads to some very interesting observations regarding the patterns of progress made during software development. We can observe from the graph that 45% of the total effort was put into the Pretty Printing activity, in case 1. The same trend follows for case 2 and case 3. This strongly suggests that the programs developed were presented taking into account factors like proper indentation, program readability, explicit program structure etc., which are considered the merits of Pretty Printing activity.

Reconstruction and Correction activities are the next two significant activities after Pretty Printing in Case 1. This trend again is true in the other cases also. This means that the structure of the programs developed underwent a lot of changes before reaching their final correct forms. Therefore, it could be said that the programs developed had a reasonable degree of complexity which also correlates with the fact that Correction activity involved about 15% of the total effort. Which also means that the programmers also put in effort in trying to eliminate errors already made in earlier versions.

Also note from the graph that the Redistribution

activity was not at all significant for Case 1. This is true in the other cases too. This clearly indicates that the programmers did develop programs that had a stable structure.

Adding functionality and Removing functionality activities amounted to less than 10% of the total effort, and Removing functionality was slightly less than Adding functionality. This again is true in case 2 and case 3 also. Although not a significant amount of effort was put into these activities during the entire period of development, nevertheless it suggests that function, declaration, preprocessor and control statements were involved in the development of programs again indicating some degree of program complexity.

Overall, Figure 1 (Appendix B) suggests a key point concerning the software development process in that Pretty Printing activity consumed nearly half the total effort involved, making it a very significant activity in the development process. Thus, if one wants to optimize the development completion time, concentration must be towards reducing the Pretty Printing activity or encouraging the use of pretty printers such as "cb". Next, Reconstruction and Correction activities together constituted roughly 25% of the total effort put into the entire development process. All the other activities therefore involved only about 30% of the total effort. The same trend follows in the other two cases. Thus, for optimizing the total development

time, the above activities must be looked into more seriously. Surprisingly, the Debugging activity was not very significant as it only consumed about 6% of the total effort involved. This suggests that either the programs were not very complex in nature or that the programmers had a very good understanding of the software engineering methodology taught to them in the course CMPSC 541.

The next step in this discussion is to analyze the behavior of the programmer activities in different stages of development starting from the month of January to the end of May. Refering to Figure 2 (Appendix B) which shows the development process for the month of January. It can be observed that the Pretty Printing activity is again the most significant activity. It is true again in the other two cases also. It again renders the merits discussed in earlier sections. The effort involved in the Documentation activity was minimal, which is understandable as the process of development is in its initial stages. Naturally, Remove documentation is also insignificant. Redistribution and correction activites did involve about 10 to 15% of the total effort. The next two significant activities are Adding functionality and Removing functionality. This seems to indicate that the process of building the correct program structure and correct logic would have been the case as this is the first period of development. Thus the graph clearly indicates

that the process of development is actually heading in the right direction. It also indicates that progress is being made in trying to build the program structure and logic.

Figure 3(Appendix B) shows the development process for the month of February. As seen from the graph the Debugging activity has almost doubled its value in January. The same trend is present in the other two cases. Also, Pretty Printing remains more or less the same as it was in the month of January. It is still significant. There is not much of a change in Documentation, Removing Documentation, and Redistribution. Adding Functionality is more than Removing Functionality. All of the above suggests that the development is progressing and the programmers are now somewhat correct in their logic and program structure.

Figure 4(Appendix B) shows the development process in the month of March. Owing to the small number of program pairs analyzed in this month (as it had less number of working days because of University's spring break) compared to the other periods of development, it would not be appropriate to compare the results in this month with the other periods of development. Anyway, for completion sake, the results in this period indicate that the Debugging activity was most significant. This might give an indication that the programs developed are becoming more complex in nature than before and a lot of time is spent on

testing the behavior of the programs.

Figure 5 (Appendix B) again shows the development process for the month of April. Next to the Pretty Printing activity, Reconstruction and Correction activities are significant compared to others which remain steady at low values. This suggests that the development process of the programs is at the peak and therefore a high percentage of effort in Reconstruction and Correction activities. This is true for the other two cases also.

Figure 6(Appendix B) shows the development process for the month of May. Pretty Printing activity remains significant and has approximately the same value as it had in April. The effort put into Adding Functionality and Removing Functionality have been reduced in this month compared to April. This trend exists for the other two cases too. Also, the Documentation activity has significantly increased compared to its value in April suggesting more effort into it in May. Considering the fact that May being the last period in the development stage, it comes as no surprise that Documentation activity increased. Since the effort put into Adding functionality and Removing functionality have also decreased significantly, the progress in developing the software has been made in building the proper program structure. Figures 7 through 16 show the progress patterns of individual activities for the entire period of development.

Finally, Figure 17 gives an indication of the process of integration. Nearly 50% of the programs took only a single day for completion. This means that all of these modules were completed in the very first attempt and it was not necessary to later on redo any work in these modules at the time of integration. In addition, the mean and standard deviation for the amount of time (in days) taken by all the programs analyzed turn out to be 17 and 31 days. Since, the mean is significantly low, and the standard deviation is nearly double the mean, it clearly indicates that a fewer number of modules involved rework. Hence, the overall process of integration must have taken place smoothly without too much effort put into the rework activity.

CHAPTER 7

CONCLUSIONS

A time distribution of the various programmer activities during the course of software development and integration phases of the software development cycle, based on an analysis of changes in source code of successive document versions, each version being the result of the synthesis of prior versions. The project was conducted by monitoring the development cycle of actual software projects.

In this study, the time distribution obtained shows that the Pretty Printing actvity was the most significant activity consuming about 45% of the total effort put during the development process. The next two significant activities were Reconstruction and Correction. Surprisingly, Debugging was not very significant, and so were other activities. But, in general all the activities were involved in the development process. Also, it was evident from the results that the set of activities classified proved to be a good representation of the overall development phase of the software development cycle.

The time distribution of the various programmer activities has been shown to indicate the progress of the software during development and integration phases in an easily

understandable visual representation. By using this approach, even non-technical software managers in-charge of the development process should be able to monitor the amount of time spent on each of the programmer activities during development. In addition, it should also be able for them to take appropriate action in readjusting the schedule of activities. The action to be taken depends on the importance of the activity within the context of the nature of software being developed.

Future research could extend this study by defining the various programmer activities at a much finer level. For example, Debugging activity must include only those output statements which are used for the purpose of testing the behavior of the programs and not all output statements as used in this study. Research could also be directed towards quantifying the degree of actual progress made during the course of software development in addition to visual representations.

REFERENCES

[Baker'87] A.L. Baker, James M. Biemann, David A. Gustafson, and Austin C. Melton,Modelling and Measuring the Software Development Process, Procedures of HICCS 1987.

[Bau'72] F. L. Bauer, Software Engineering, Information Processing 71 (Amsterdam : North Holland Publishing Co.,1972), pp. 530.

[Gilbert'87] Phillip Gilbert, Software Design and Development, Science Research Associates Inc., Chicago IL. 1983.

[Gustafson'85] David A. Gustafson, Austin C. Melton, Chyuam Samuel Hseih, An Analysis Of Software Changes During Maintenance and Enhancement, Proceedings of Conference on Software Maintenance, Nov 11-13, 1985 pp. 92-95.

[Lanchbury'86] Mary Lou A. Lanchbury, A Model of Successful Patterns of Progress During the Integration of Software, Masters' Thesis, Kansas State University, 1986.

[Hsu'88] Yu-Hua Hsu, Classifying Program Changes During Software Development, Masters' Thesis, Kansas State University, 1988.

[Pressman'87] Roger S. Pressman, Software Engineering - A Practioners' Approach, Second Edition 1987, McGraw Hill Book Co.

APPENDIX A

SOURCE FILE FOR examplefile1

```
#include <stdio.h>
#include "typedefs"
#define n 21

compare_data_items(user_data,valid_data,filename)
struct di_struct *user_data, *valid_data;
char *filename;

{
  int status;
  char *data_items;
  FILE *data;
  struct di_struct *invalid, *user_temp,
                *valid_temp, *back_user;
  struct di_struct *difference, *dif_temp,
      *temp_data;

  temp_data = user_data;
  printf("0);
  while (temp_data != NULL)
  {
    printf("the user data item is %s 0,
            temp_data->data_item);
    temp_data = temp_data->d_next;
  }
  temp_data = valid_data;
  printf("0);
  while (temp_data != NULL)
  {
    printf("the valid data item is %s 0,
            temp_data->data_item);
    temp_data = temp_data->d_next;
  }
  difference = invalid = dif_temp;
  user_temp = user_data;
  back_user = user_data;
  while (user_temp)
  {
    printf("in while user_temp0);
    valid_temp = valid_data;
    status = 0;
    printf(" status is %d and !status is %d ",
        status,!status);
    if (!(status)) printf(" sets status okay 0);
    else printf(" **** Status not set okay 0);
    while ((valid_temp) && (!(status)))
    {
```

```
    printf(" *****The user data item is %s 0,
      user_temp->data_item);
    printf(" *****The valid data item is %s 0,
            valid_temp->data_item);
    if (!scomp(user_temp->data_item,
     valid_temp->data_item))
    {
      printf("%%%%%%%%%%%%%%compare was
       successful 0);
      status = 1;
    }
    else valid_temp = valid_temp->d_next;
  }
  printf("0##### out of while valid_temp
       and status 0);
  if (!(status))
  {
     printf("0*** The data itmes did not
       match *** 0);
     dif_temp = (struct di_struct *) calloc(1,sizeof
          (struct di_struct));
     dif_temp->d_next = NULL;
     printf(" the user_data->data_item %s ",
       user_data->data_item);
     if (back_user == user_data)
       user_data = user_data->d_next;
     else back_user->d_next = valid_temp->d_next;
     invalid->d_next = dif_temp;
  }
  user_temp = user_temp->d_next;
  back_user = back_user->d_next;
}
if ((difference))
{
  invalid = difference;
  printf(" The following data items are not
     valid to be searched 0);
  while (invalid)
  {
    printf(" in the invalid while loop0);
    printf("%s 0,invalid->data_item);
    invalid = invalid->d_next;
  }
  data = fopen(filename,"w");
  user_temp = user_data;
  while (user_temp)
  {
     fputs(user_temp->data_item,data);
     user_temp = user_temp->d_next;
  }
}
return;
```

```
}
```

** END OF SOURCE FILE **

SOURCE FILE FOR examplefile2

```
#include <stdio.h>
#include "typedefs"
#define n 21

compare_data_items(user_data,valid_data,filename)
struct di_struct *user_data, *valid_data;
char *filename;

{
 int status;
 char *data_items;
 FILE *data;
 struct di_struct *invalid, *user_temp,
     *valid_temp, *back_user;
 struct di_struct *difference, *dif_temp,
        *temp_data;

 temp_data = user_data;
 printf("0);
 while (temp_data != NULL)
 {
  printf("the user data item is %s 0,
      temp_data->data_item);
  temp_data = temp_data->d_next;
 }
 temp_data = valid_data;
 printf("0);
 while (temp_data != NULL)
 {
  printf("the valid data item is %s 0,
      temp_data->data_item);
  temp_data = temp_data->d_next;
 }
 difference = invalid = dif_temp;
 user_temp = user_data;
 back_user = user_data;
 while (user_temp)
 {
  printf("in while user_temp0);
  valid_temp = valid_data;
  status = 0;
  while ((valid_temp) && (!(status)))
  {
   printf(" *****The user data item is %s 0,
     user_temp->data_item);
   printf(" *****The valid data item is %s 0,
    valid_temp->data_item);
```

```
    if (lscomp(valid_temp->data_item,
     user_temp->data_item))
    {
      printf("%%%%%%%%%%%%%%compare was
       successful 0);
      status = 1;
    }
    else valid_temp = valid_temp->d_next;
  }
  printf("0##### out of while valid_temp and
     status 0);
  user_temp = user_temp->d_next;
  back_user = back_user->d_next;
 }
 return;
}
```

** END OF SOURCE FILE **

FILE : main.results

%*% This is start of analysis data
START1
File Name1 : ./Jan21/examplefile1
File Name2 : ./Jan21/examplefile2
Wed Jun 22 07:41:01 CDT 1988

File Name : ./Jan21/examplefile1

for	0
while	6
if	5
else	3
switch	0
case	0
goto	0
break	0
continue	0
assignments	24
preprocessor	3
comments	0
blanklines	3
return	1
input	0
output	18
functions	12
declarations	6
default	0

Weights/Lines = 2.70000
Weights = 232.20000
Lines of code = 86

Levels :

zero	11
one	25
two	29
three	18
four	3
five	0
six	0

Zeroave = 12.791
Oneave = 29.070
Twoave = 33.721
Threeave = 20.930
Fourave = 3.488

```
Fiveave       0.000
Sixave =      0.000
Total average =      273.256
Sum =                 86
Lines of code =        86
Sum/Lines :          1.000
-----------------------
END1
START2
=======================
File Name :   ./Jan21/examplefile2
-----------------------
for                0
while              4
if                 1
else               1
switch             0
case               0
goto               0
break              0
continue           0
assignments        14
preprocessor       3
comments           0
blanklines         3
return             1
input              0
output             9
functions          5
declarations       6
default            0
-----------------------
Weights/Lines =     3.14074
Weights =         169.60000
Lines of code =        54
-----------------------
Levels :
~~~~~~~~~~~~~

zero      11
one       22
two       13
three     6
four      2
five      0
six       0
-----------------------
Zeroave =     20.370
Oneave =      40.741
Twoave =      24.074
Threeave =    11.111
Fourave =     3.704
Fiveave       0.000
```

```
Sixave =      0.000
Total average =      237.037
Sum =                 54
Lines of code =        54
Sum/Lines :          1.000
-------------------------
=========================
END2
START3
File Name :   changes.with.TAB
-------------------------
for                0
while               0
if                 1
else               0
switch              0
case               0
goto               0
break               0
continue            0
assignments          0
preprocessor         0
comments             0
blanklines          0
return             0
input              0
output              0
functions           1
declarations         0
default            0
-------------------------
Weights/Lines =    11.40000
Weights =         11.40000
Lines of code =          1
-------------------------
Levels :
~~~~~~~~~~~~~

zero       0
one        0
two         0
three      1
four       0
five       0
six        0
-------------------------
Zeroave =      0.000
Oneave =       0.000
Twoave =        0.000
Threeave =     100.000
Fourave =      0.000
Fiveave       0.000
Sixave =       0.000
```

```
Total average =      400.000
Sum =                  1
Lines of code =        1
Sum/Lines :          1.000
-------------------------
=========================
END3
START4
File Name :  changes.without.TAB
-------------------------
for                0
while              0
if                 1
else               0
switch             0
case               0
goto               0
break              0
continue           0
assignments        0
preprocessor       0
comments           0
blanklines         0
return             0
input              0
output             0
functions          1
declarations       0
default            0
-------------------------
Weights/Lines =    11.40000
Weights =          11.40000
Lines of code =        1
-------------------------
END4
```

*** END OF main.results ***

FILE : "class.results"

FILE NAME	DATE	CLASSIFICATION
Jan21/examplefile2	Jan 21	REMOVING DEBUGGING
Jan21/examplefile2	Jan 21	RECONSTRUCTION
Jan21/examplefile2	Jan 21	REMOVING FUNCTIONALITY
Jan21/examplefile2	Jan 21	CORRECTION
Jan21/examplefile2	Jan 21	PRETTY PRINTING

***** END OF class.results ******

SOURCE CODE FOR MAIN.c

```
/* This program takes as its input a data file  */
/* having the format as in file class.results,  */
/* containing the filenames of different versions*/
/* of programs to be analyzed. The program picks */
/* up a pair of different versions of the same  */
/* program and executes the command :          */
/* "CHANGES programfile1 programfile2" , where  */
/* CHANGES is another utility program. The whole*/
/* process is repeated until the end of file is */
/* reached.                                    */

#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{

   FILE *fopen(), *fp;
   char string[100];
   char st1[100], su1 [100], st2[100],su2[100],
 cmd[200];

   fp = fopen(*++argv, "r");
   getstring(string, fp);
   strcpy(st1, string);
   copysubstring(su1,string);
   getstring(string, fp);
   strcpy(st2, string);
   copysubstring(su2,string);
   while (string[0] != '*')
   {
    if (strcmp(su1,su2) == 0)
     {
      sprintf(cmd, "sh CHANGES1 %s %s", st1, st2);
      printf ("cmd_str = '%s'0, cmd);
      system(cmd);
      copystring(st1, st2);
      getstring(string, fp);
      copystring(st2,string);
      copysubstring(su2,string);
     }
      else
        {
          copystring(st1, st2);
          copystring(su1, su2);
          getstring(string, fp);
```

```
            copystring(st2,string);
            copysubstring(su2,string);
          }
    }
}

getstring(s, fp)
char s[];
FILE *fp;
{
  int c, n, i;
  n = 0;
  fscanf (fp, "%s %*s %*s %*s", s);
}
 copystring(s, t)
 char *s, *t ;
 {
   while( (*s++ = *t++ ) != ' ' )
 ;
 }
 copysubstring(s1,s2)
 char s1[], s2[];
 {
   int i,count, j;
   if (s2[6] == '/')
   {
   strcpy(s1, &s2[7]);
   }
     else
   {
   strcpy(s1, &s2[8]);
   }
 }
```

* * END OF SOURCE CODE FOR MAIN.c * *

SOURCE CODE FOR "stat1"

```
# This program arrives at the TIME Distribution
# of the various programmer activities based on
# the count of the total number of activities
# generated in the outputdata. This gives an
# indication of the total EFFORT put into the
# software development project.

BEGIN { }
{
 totoutput = totoutput + 1;
 x = $3; y = $4;
 if (x ~ /^ PRETTY/) {++pretty}
 else
 if (x ~ /^ DEBUGGING/ ) {++debug}
 else
  if (x ~ /^ DOCUMENTATION/ ) {++docn}
  else
   if (x ~ /^ RECONSTRUCTION/ ) {++recons}
    else
     if (x ~ /^ REDISTRIBUTION/ ) {++redist}
     else
      if (x ~ /^ CORRECTION/ ) {++correc}
      else
       if (x ~ /^ ADDING/ ) {++addfunc}
       else
        if (x ~ /^ REMOVING/ )
          {
         if (y ~ /^ DOCUMENTATION/ ){++remdoc}
         else
          if (y ~ /^ FUNCTIONALITY/ ){++remfunc}
          else
           if (y ~ /^ DEBUGGING/ ) {++remdebug}
            }
}
END {
  printf " percent of activities of each classification0 >> "STAT";
  printf " (all activities considered)0 >> "STAT";
  printf " ----------0 >> "STAT";
  printf " NO OF OUTPUT LINES  : %6d0,totoutput >> "STAT";
  printf " DEBUGGING  %.2f0,((debug/totoutput)*100) >> "STAT";
  printf " REMOVING DEBUGGING  %.2f0,((remdebug/totoutput)*100) >> "STAT";
  printf " DOCUMENTATION          %.2f0,( (docn/totoutput)*100) >> "STAT";
  printf " REMOVING DOCUMENTATION %.2f0,((remdoc/totoutput)*100) >> "STAT";
  printf " PRETTY PRINTING        %.2f0,((pretty/totoutput)*100) >> "STAT";
  printf " RECONSTRUCTION          %.2f0,((recons/totoutput)*100) >> "STAT";
  printf " REDISTRIBUTION         %.2f0,((redist/totoutput)*100) >> "STAT";
  printf " CORRECTION             %.2f0,((correc/totoutput)*100) >> "STAT";
```

```
printf " ADDING FUNCTIONALITY   %.2f0,((addfunc/totoutput)*100) >>"STAT";
printf " REMOVING FUNCTIONALITY %.2f0,((remfunc/totoutput)*100) >>"STAT";
}
```

** END OF SOURCE CODE FOR stat1 **

SOURCE CODE FOR "stat2"

```
# This program takes as its input any file having the
# format contained in the "class.results" file, and
# its purpose is to determine the number of times each
# of the programmer activities was involved  in terms
# of the changes between different versions, based on
# the total number of program pairs analyzed. The
# ease of interpretation.

BEGIN {l= "zz";x = "aa"; y = "bb" }
{
 if (l != $1)
 ++pgmruns;l = $1;x = $3; y = $4;
 if (x ~ /^ PRETTY/) {++pretty}
 else
  if (x ~ /^ DEBUGGING/ ) {++debug }
  else
   if (x ~ /^ DOCUMENTATION/ ) {++docn}
   else
    if (x ~ /^ RECONSTRUCTION/ ) {++recons}
    else
     if (x ~ /^ REDISTRIBUTION/ ) {++redist}
     else
      if (x ~ /^ CORRECTION/ ) {++correc}
      else
       if (x ~ /^ ADDING/ ) {++addfunc}
       else
       if (x ~ /^ REMOVING/ )
       {
       if (y ~ /^ DOCUMENTATION/ ) {++remdoc}
       else
        if (y ~ /^ FUNCTIONALITY/ ) {++remfunc}
        else
        if (y ~ /^ DEBUGGING/ ) {++remdebug}
       }
}
END {
printf " percent of activities of each classification0 >> "STAT";
printf "         (Total Program Runs considered )0 >> "STAT";
printf " ------------------------------------0 >> "STAT";
printf " TOTAL PROGRAM RUNS  : %6d0,pgmruns >> "STAT";
printf " DEBUGGING       %.2f0,((debug/pgmruns)*100) >> "STAT";
printf " REMOVING DEBUGGING %.2f0,((remdebug/pgmruns)*100) >> "STAT";
printf " DOCUMENTATION    %.2f0,((docn/pgmruns)*100) >> "STAT";
printf " REMOVING DOCUMENTATION %.2f0,((remdoc/pgmruns)*100) >> "STAT";
printf " PRETTY PRINTING  %.2f0,((pretty/pgmruns)*100) >> "STAT";
printf " RECONSTRUCTION  %.2f0,((recons/pgmruns)*100) >> "STAT";
printf " REDISTRIBUTION         %.2f0,((redist/pgmruns)*100) >> "STAT";
```

```
printf " CORRECTION      %.2f0,((correc/pgmruns)*100) >>"STAT";
printf " ADDING FUNCTIONALITY   %.2f0,((addfunc/pgmruns)*100) >>"STAT";
printf " REMOVING FUNCTIONALITY %.2f0,((remfunc/pgmruns)*100) >>"STAT";
    }
```

* * END OF SOURCE CODE FOR stat2 * *

SOURCE CODE FOR "stat3"

```
# This program is an Awk program that takes as its input
# any file having the format as the file "class.results"
# and it determines a measure of the involvement of
# each of the programmer activities in any program
# development, based on the unique number of programs
# analyzed.

BEGIN {x = "aa"; y = "bb" }
{
  l = $2; g =1;
  while (g == 1)
  {
  ++uniq;
  while (l == $2)
  {
   l = $2; x = $4;
   if (x ~ /^ REMOVING/ )
     y = $5;
   if (x ~ /^ PRETTY/) {p = 1}
   else
    if (x ~ /^ DEBUGGING/ ) {deb = 1}
    else
     if (x ~ /^ DOCUMENTATION/ ) {doc = 1}
     else
      if (x ~ /^ RECONSTRUCTION/ ) {recn = 1}
      else
       if (x ~ /^ REDISTRIBUTION/ ) {redn = 1}
       else
        if (x ~ /^ CORRECTION/ ) {corr = 1}
        else
         if (x ~ /^ ADDING/ ) {adfn = 1}
         else
          if (x ~ /^ REMOVING/ )
           {
        if (y ~ /^ DOCUMENTATION/ ) {rdoc = 1}
        else
          if (y ~ /^ FUNCTIONALITY/ ) {refn = 1}
          else
           if (y ~ /^ DEBUGGING/ ) {rdeb = 1}
           }
  g = getline;
  if (g == 1) {
   continue;
            }
   }
   pretty += p; p = 0;
   debug += deb; deb = 0;
```

```
    docn += doc;  doc = 0;
    recons += recn; recn = 0;
    redist += redn; redn = 0;
    correc += corr; corr = 0;
    addfunc += adfn; adfn = 0;
    remdoc += rdoc; rdoc = 0;
    remfunc += refn; refn = 0;
    remdebug += rdeb; rdeb = 0;
    if (g == 1) { 1 = $2; continue }
   }

}
 END {
 printf " percent of activities of each classification0 >>"STAT";
 printf " (Percentage of time a program underwent a 0 >>"STAT";
 printf "        Particular Activity) 0 >>"STAT";
 printf " ------------------------------------0 >>"STAT";
 printf " NO OF (UNIQUE) PROGRAMS RUN : %6d0,uniq >>"STAT";
 printf " DEBUGGING      %.2f0,((debug/uniq)*100) >>"STAT";
 printf " REMOVING DEBUGGING  %.2f0,((remdebug/uniq)*100) >>"STAT";
 printf " DOCUMENTATION      %.2f0,((docn/uniq)*100) >>"STAT";
 printf " REMOVING DOCUMENTATION %.2f0,((remdoc/uniq)*100) >>"STAT";
 printf " PRETTY PRINTING  %.2f0,((pretty/uniq)*100) >>"STAT";
 printf " RECONSTRUCTION   %.2f0,((recons/uniq)*100) >>"STAT";
 printf " REDISTRIBUTION   %.2f0,((redist/uniq)*100) >>"STAT";
 printf " CORRECTION       %.2f0,((correc/uniq)*100) >>"STAT";
 printf " ADDING FUNCTIONALITY %.2f0,((addfunc/uniq)*100) >>"STAT";
 printf " REMOVING FUNCTIONALITY %.2f0,((remfunc/uniq)*100) >>"STAT";
  }
```

* * END OF SOURCE CODE FOR stat3 * *

APPENDIX B

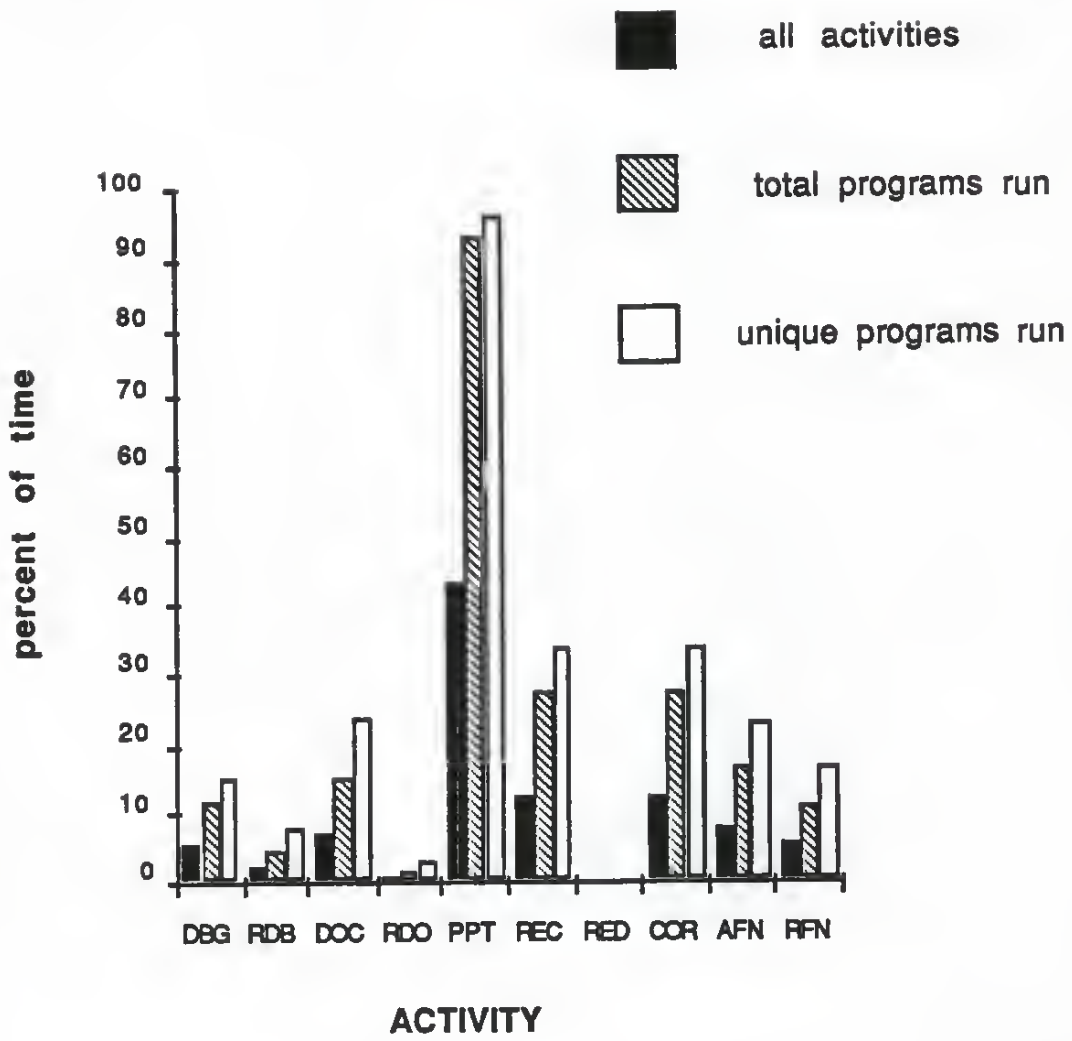

FIGURE 1. GRAPH OF percent of time VS. programmer activities

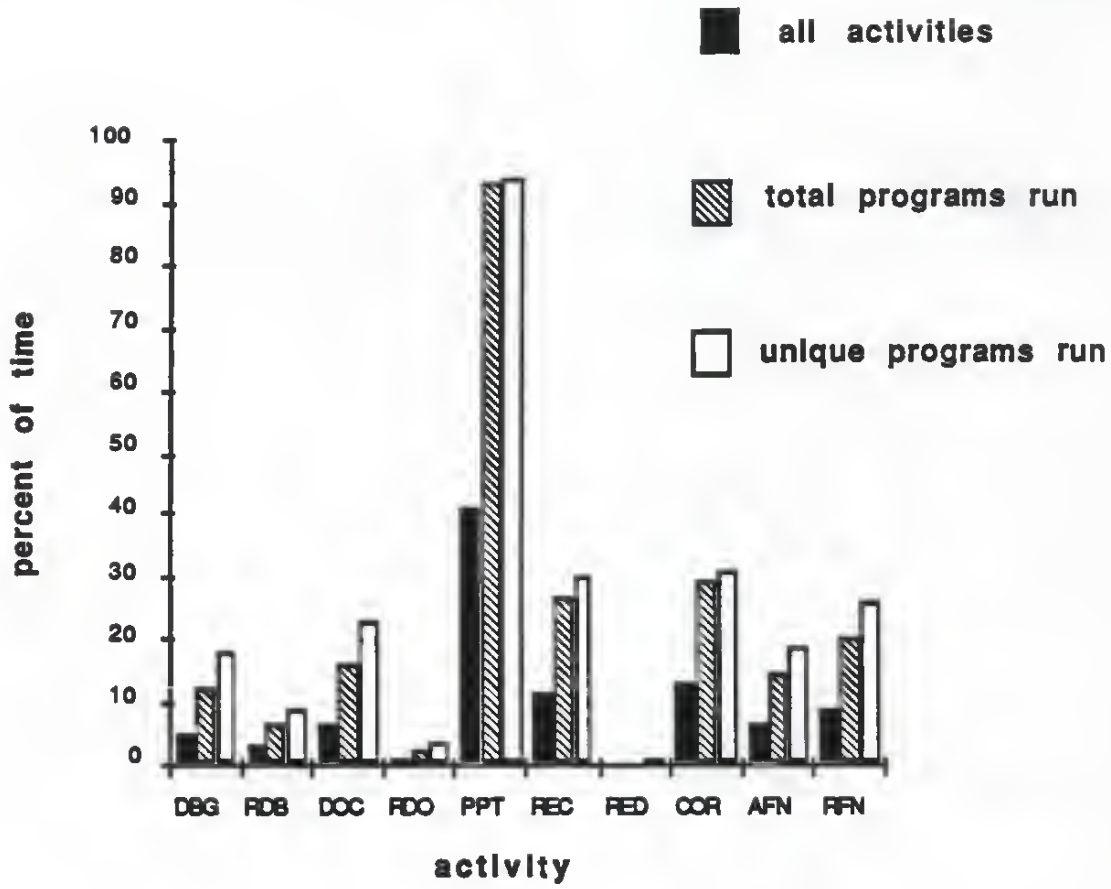

FIGURE 2. GRAPH OF percent of time VS. programmer activities FOR THE JAN DATA SET.

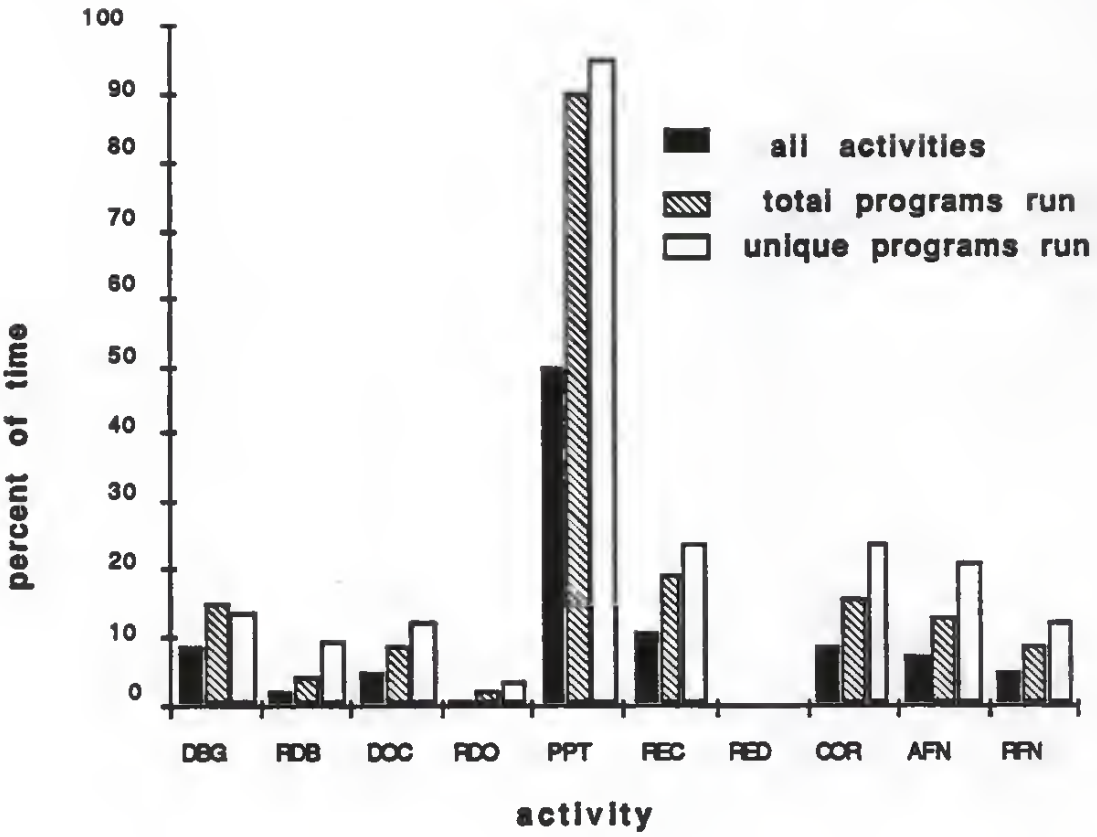

FIGURE 3. GRAPH OF percent of time VS. programmer activities FOR THE FEB DATA SET.

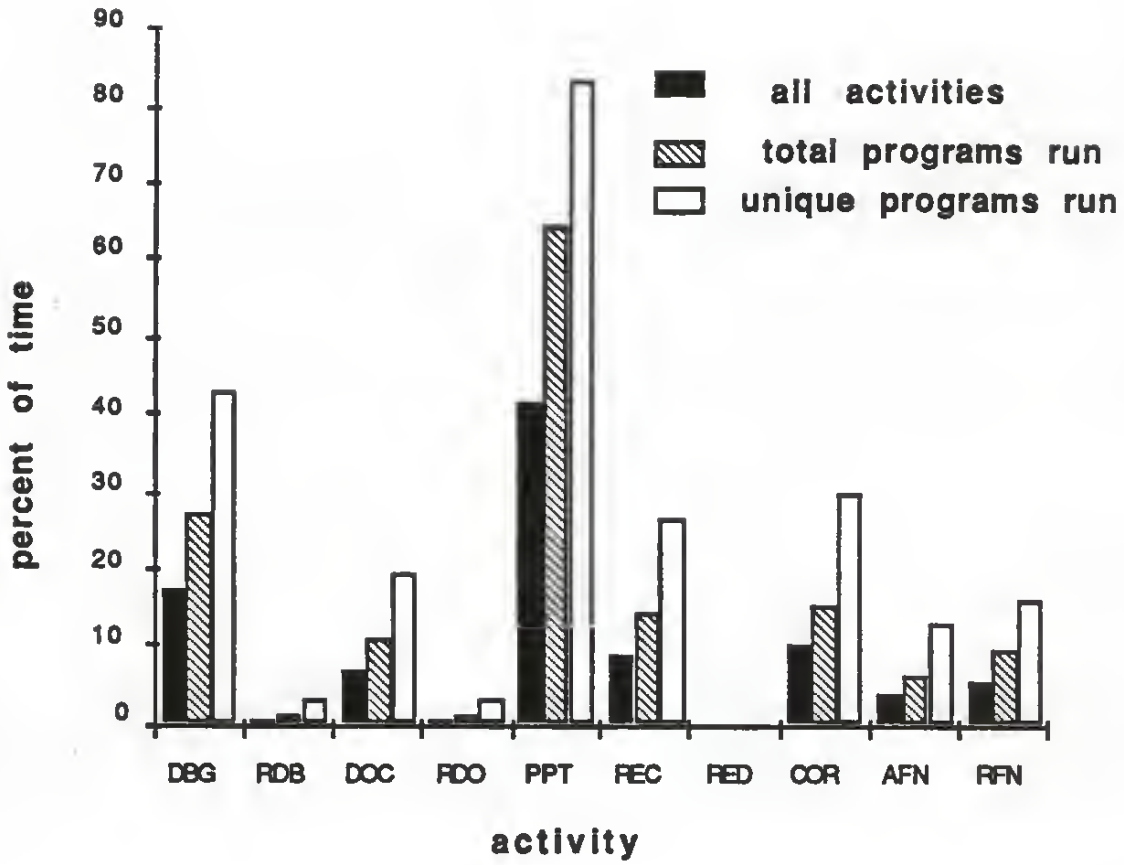

FIGURE 4. GRAPH OF percent of time VS. programmer activities FOR THE MAR. DATA SET.

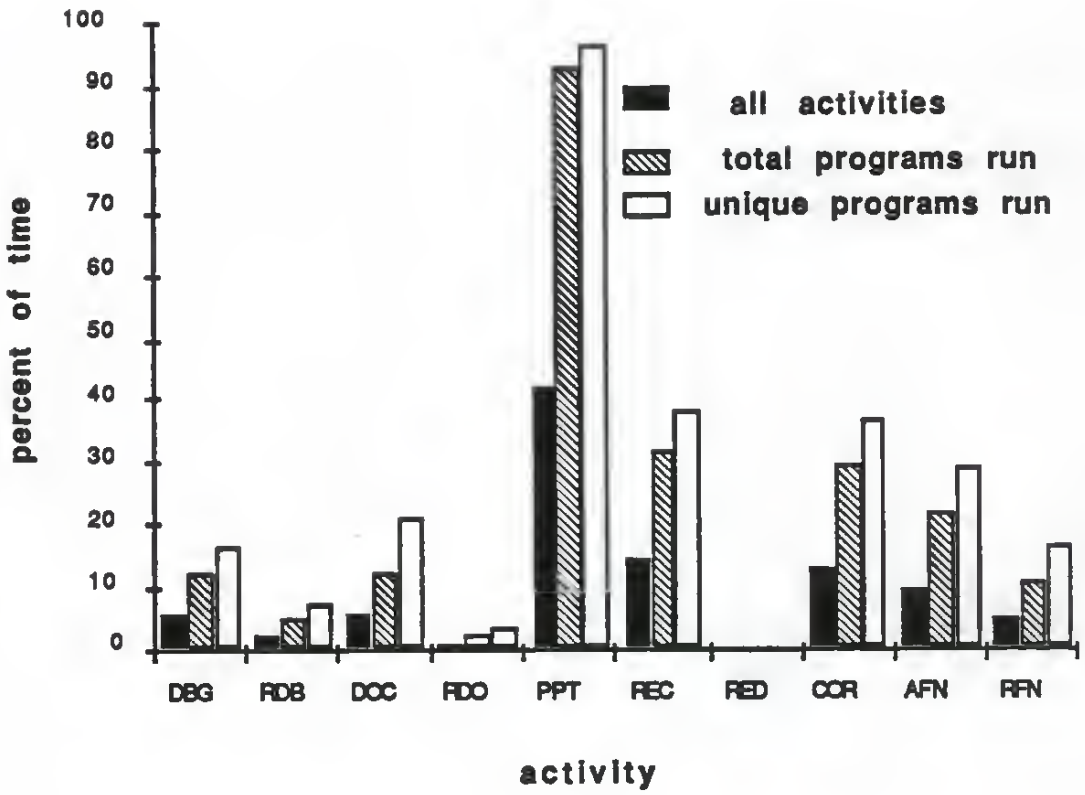

FIGURE 5. GRAPH OF percent of time VS. programmer activities FOR THE APR. DATA SET.

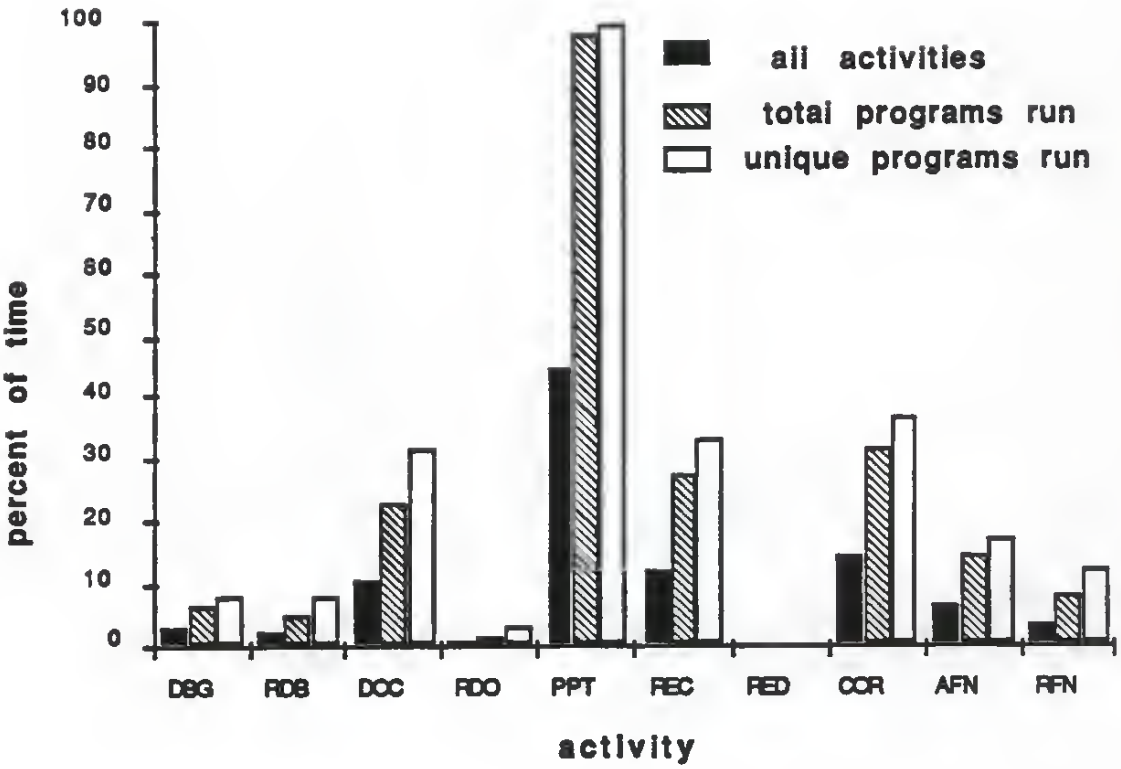

FIGURE 6. GRAPH OF percent of time VS. programmer activities FOR THE MAY DATA SET.

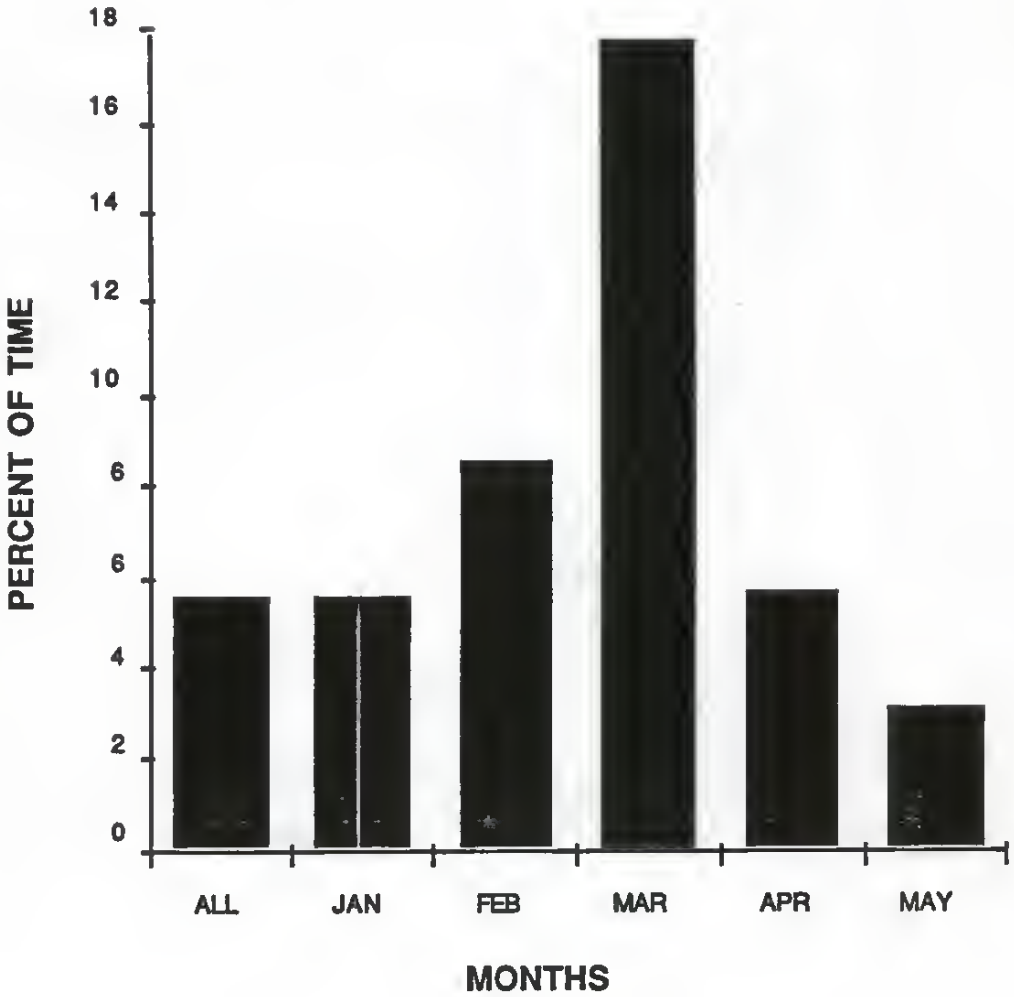

FIGURE 7. TIME DISTRIBUTION OF DEBUGGING ACTIVITY

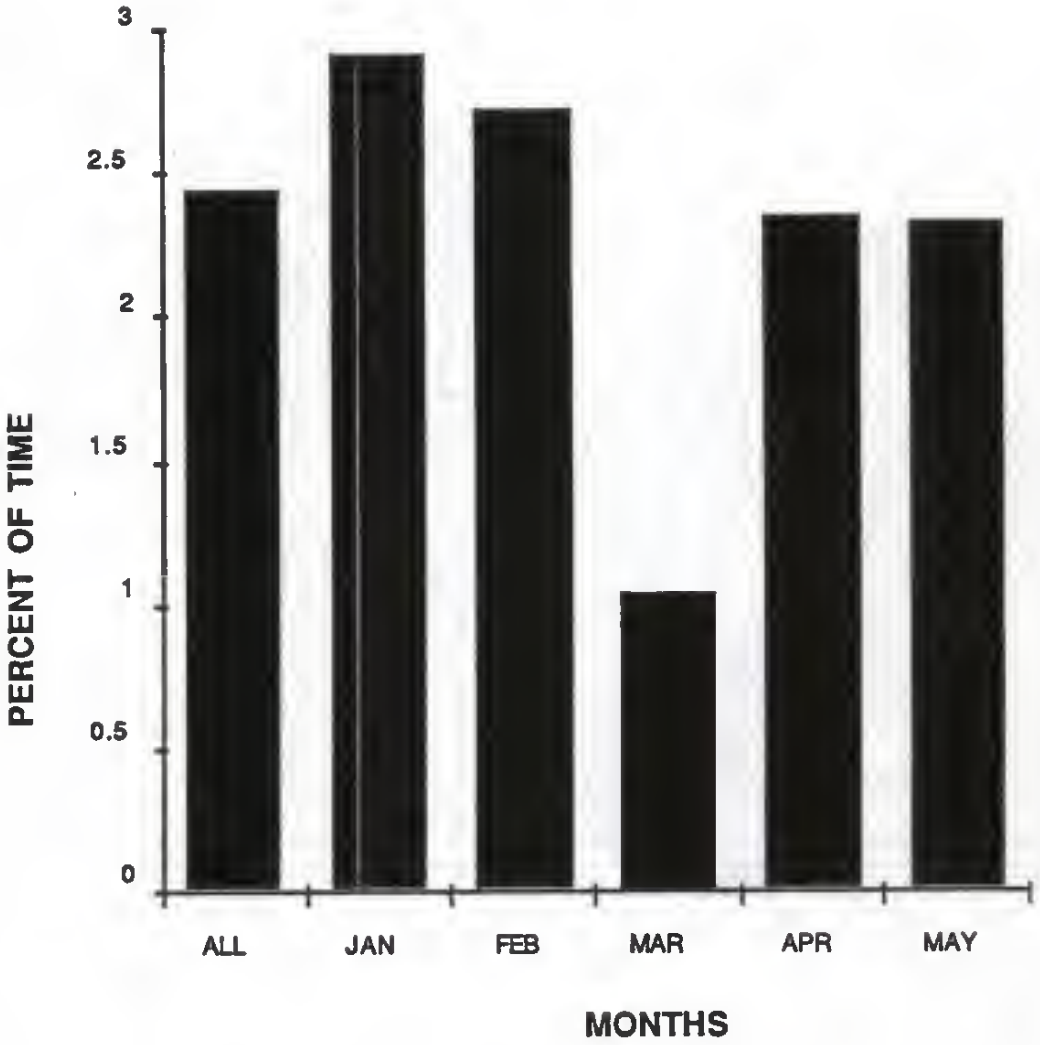

FIGURE 8. TIME DISTRIBUTION OF REMOVING DEBUGGING.

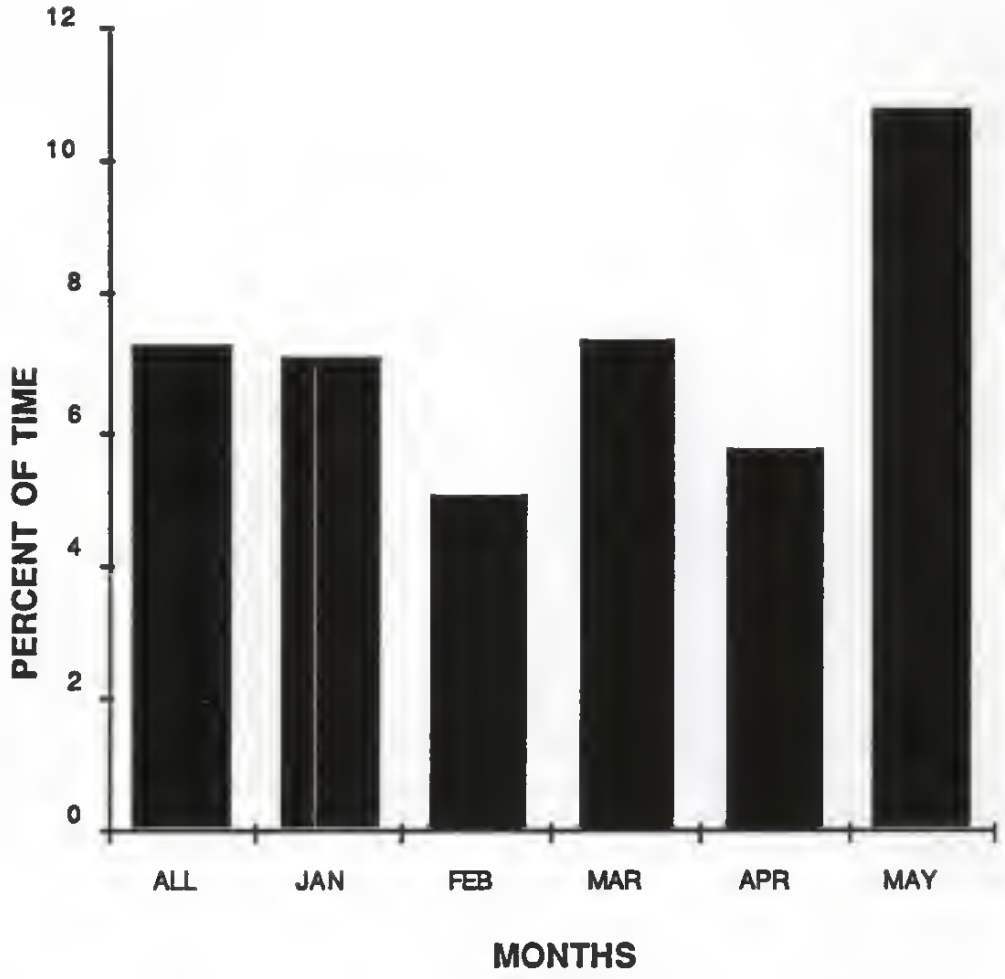

FIGURE 9. TIME DISTRIBUTION OF DOCUMENTATION.

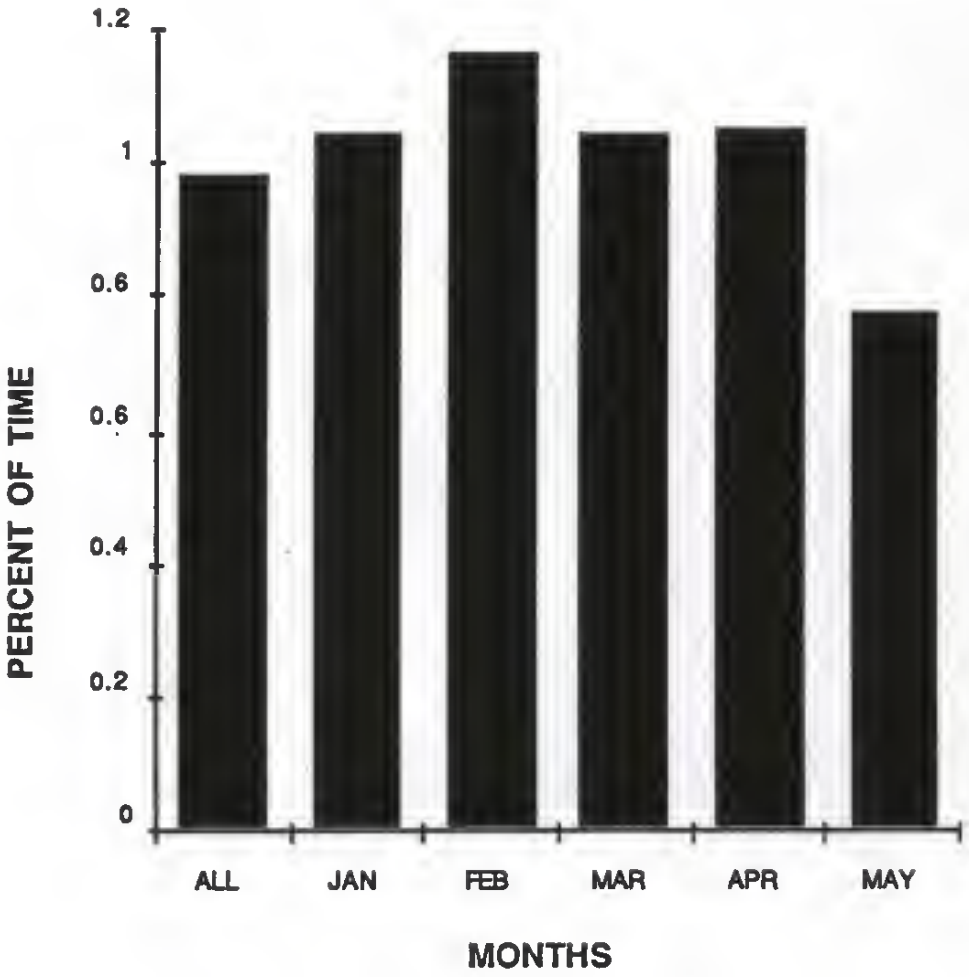

FIGURE 10. TIME DISTRIBUTION OF REMOVING DOCUMENTATION.

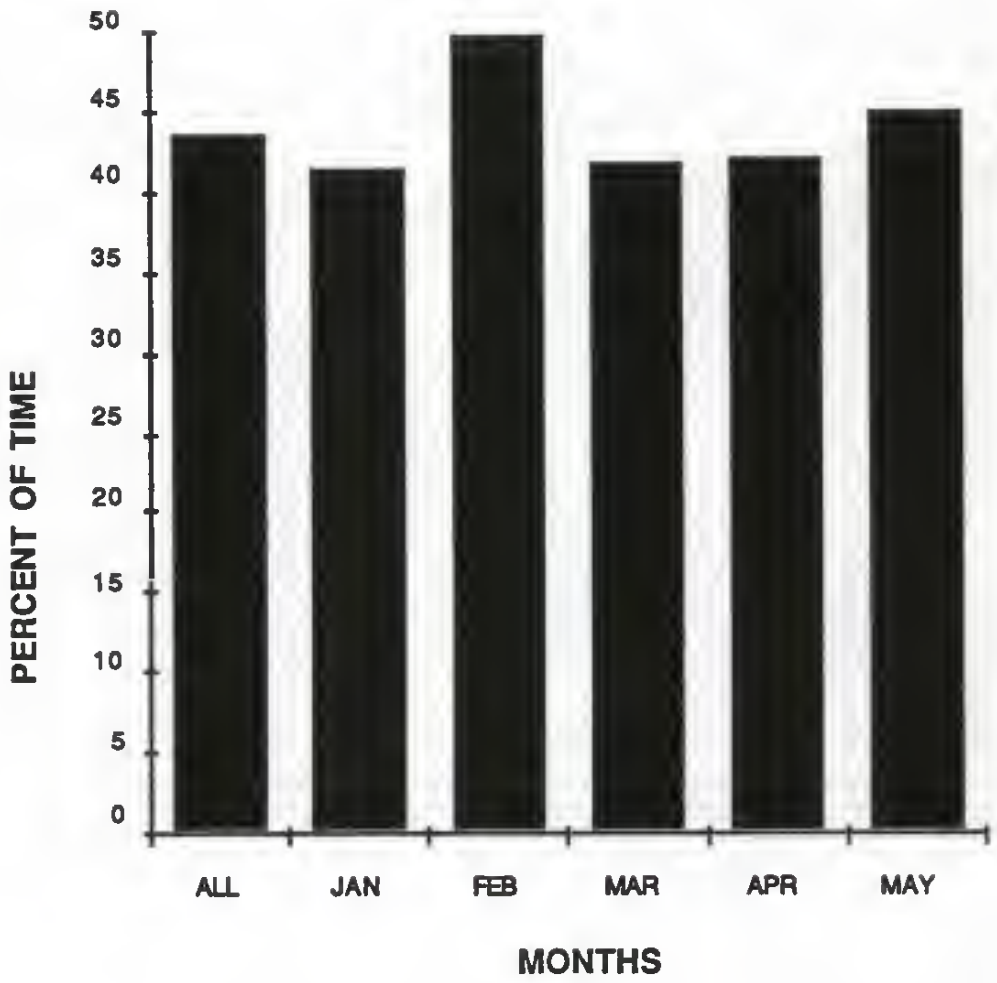

FIGURE 11. TIME DISTRIBUTION OF PRETTY PRINTING.

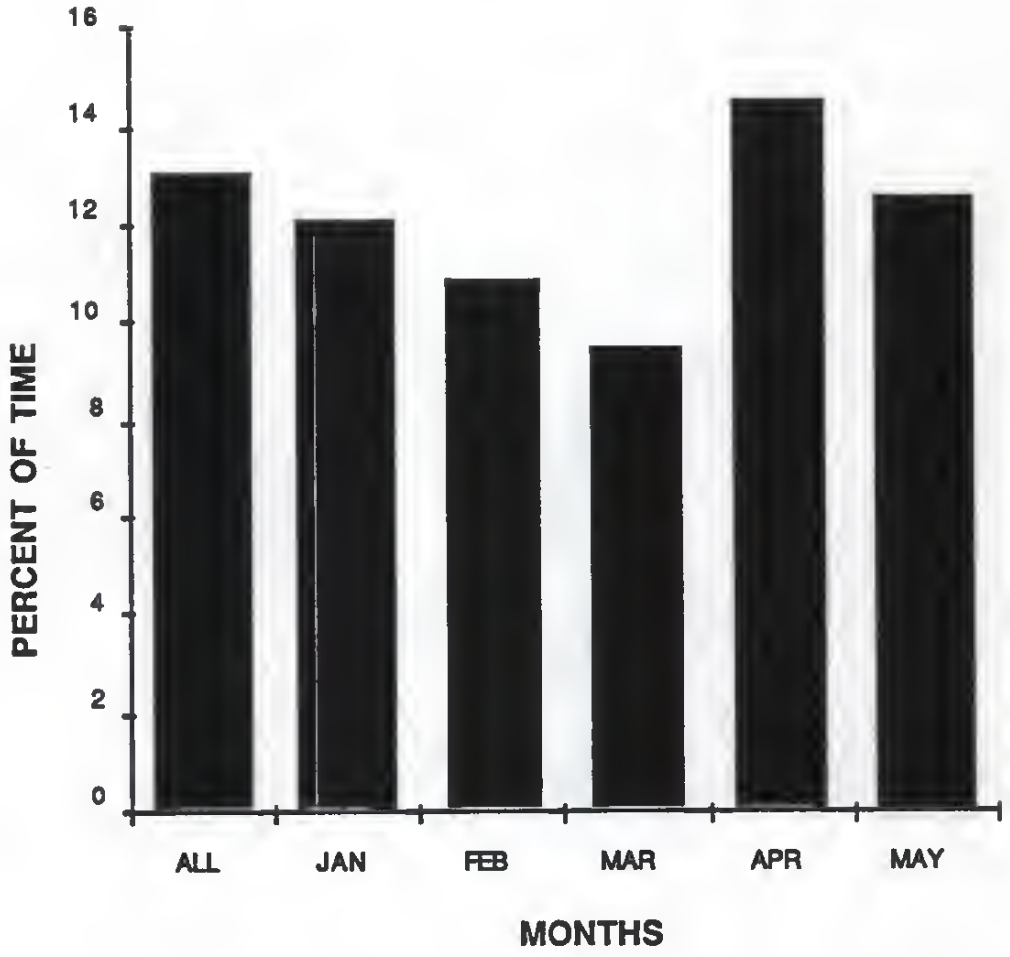

FIGURE 12. TIME DISTRIBUTION OF RECONSTRUCTION.

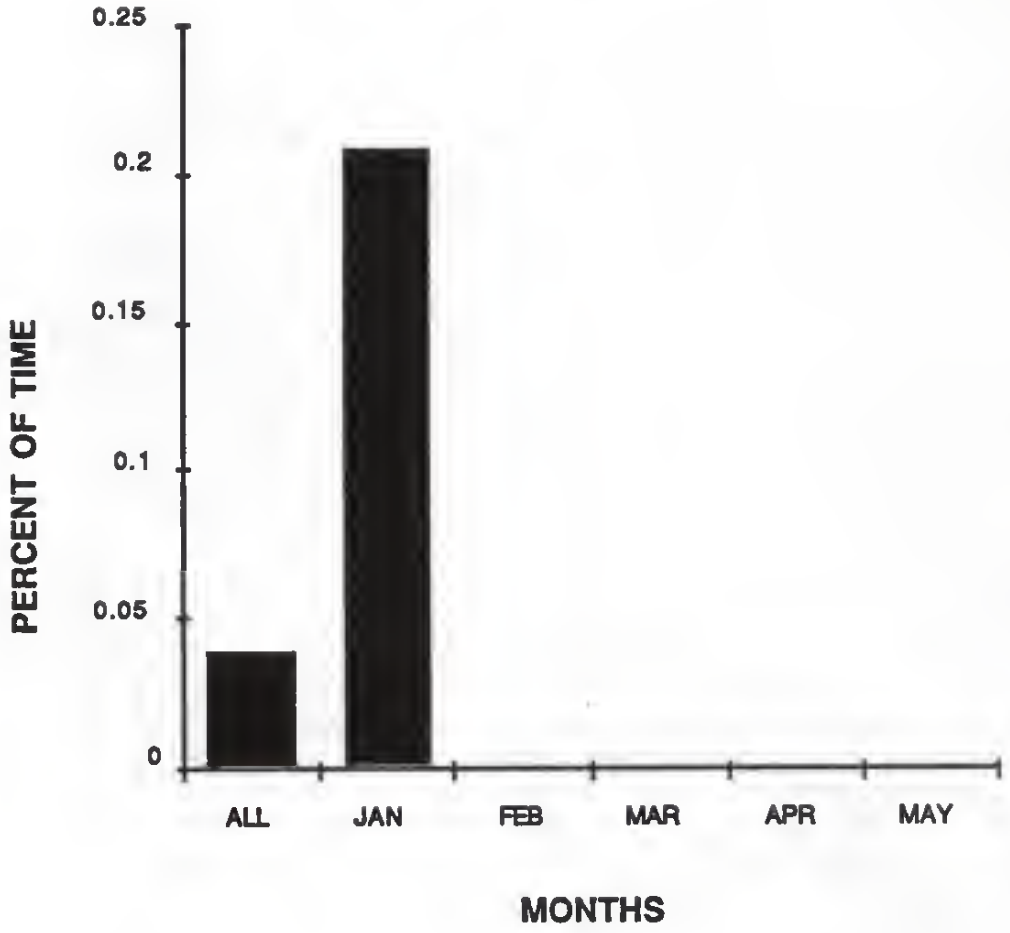

FIGURE 13. TIME DISTRIBUTION OF REDISTRIBUTION.

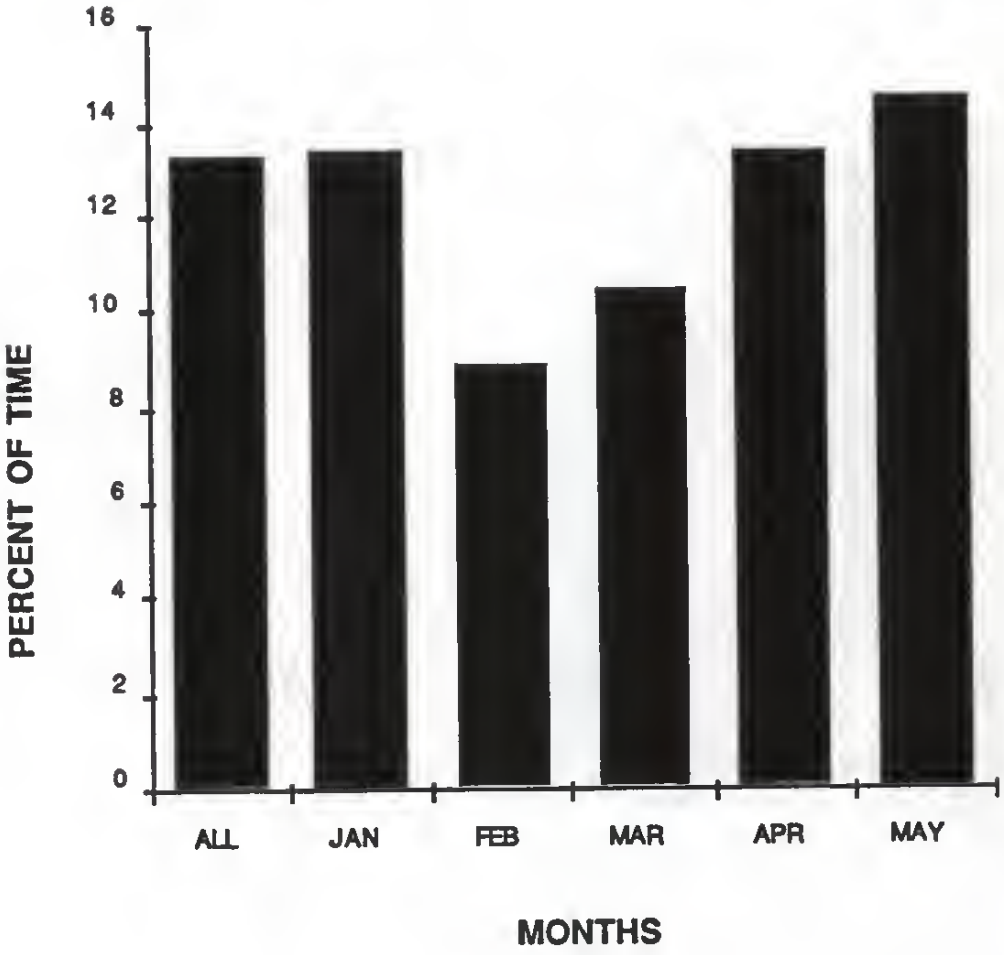

FIGURE 14. TIME DISTRIBUTION OF CORRECTION.

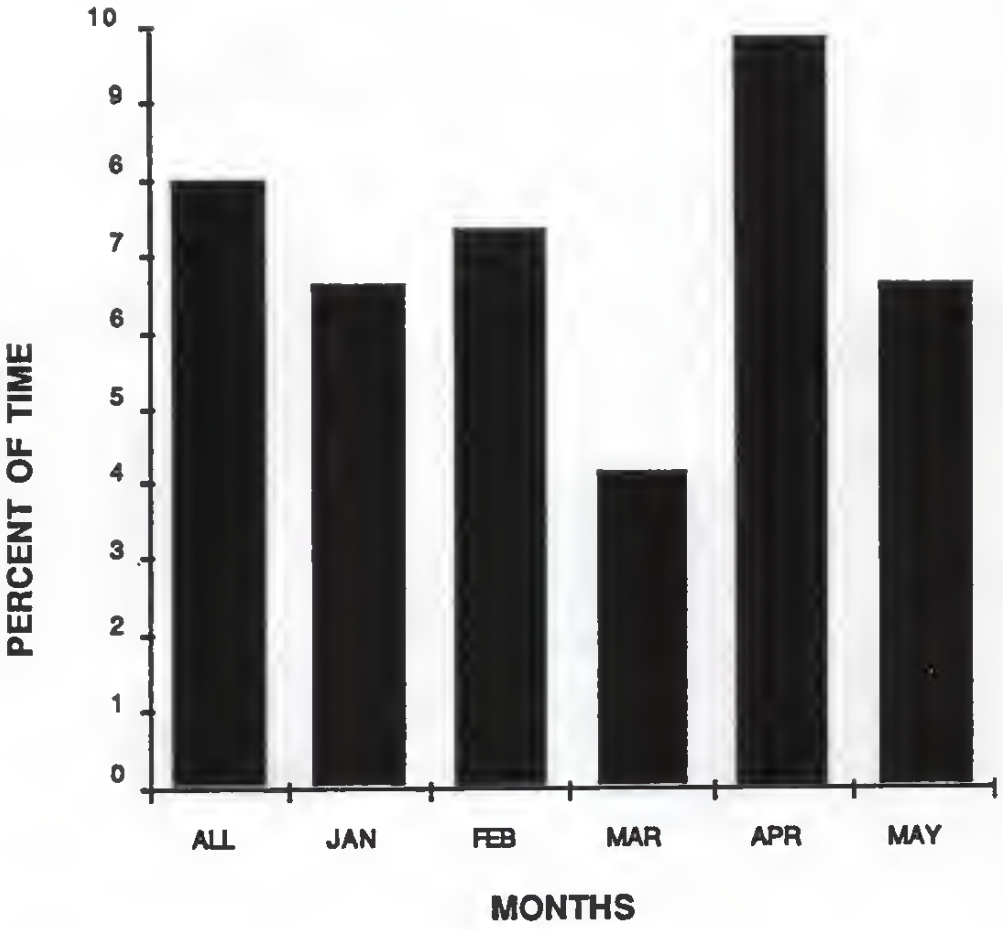

FIGURE 15. TIME DISTRIBUTION OF ADDING FUNCTIONALITY.

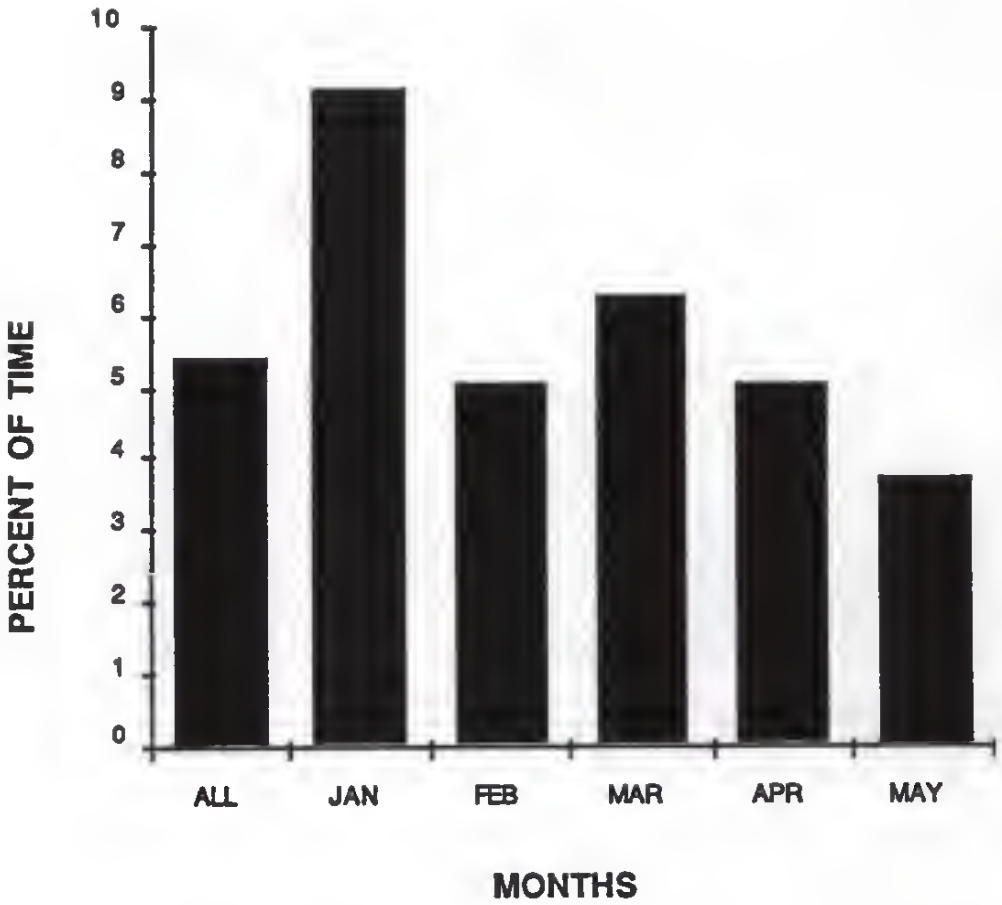

FIGURE 16. TIME DISTRIBUTION OF REMOVING FUNCTIONALITY.

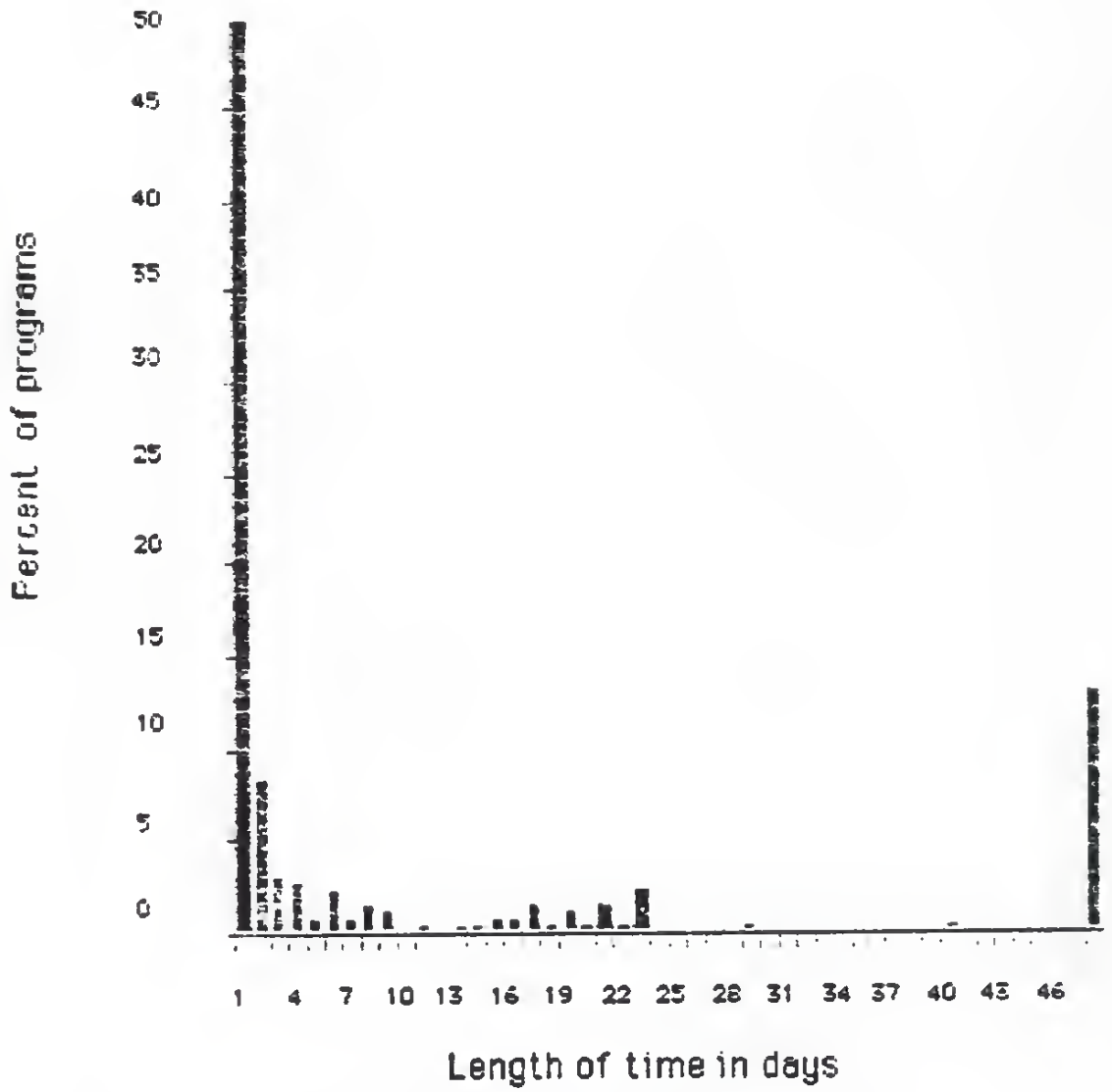

FIGURE 17: Graph of percent of programs vs. length of time

AN EMPIRICAL STUDY OF PROGRAMMER ACTIVITIES DURING SOFTWARE DEVELOPMENT AND INTEGRATION

by

KITTUR V. GANESH

B.E. (Mechanical Engg.), Bangalore University, India

M.S. (Industrial Engg.), Kansas State University, Manhattan, KS.

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer and Information Sciences

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

ABSTRACT

This project arrives at the time distribution of programmer activities during the course of the software development and integration phases of the software development cycle. It is based on an analysis of patterns of program changes between successive versions of programs, each version being the result of the synthesis of earlier versions. The programmer activities themselves are classified empirically based on past data in an earlier research conducted in the department of Computer Science, at Kansas State University.

The study involved analyzing 1321 program pairs of program versions of programs which were developed by the students of the course CMPSC 541 assigned to them as projects. The time distributions for the various activities were determined and was shown that they act as good indicators of progress made during the software development process. This approach could be used by software managers to monitor the progress of software development and take appropriate action depending on the importance of the activity within the context of the software being developed.