308

CLASSIFYING PROGRAM CHANGES DURING SOFTWARE DEVELOPMENT

by

Yu-Hua Hsu

B.S., Central State University, 1984

A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

David A. Gustafson

Major Professor

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Page

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to express her gratitute and sincere thanks to Dr. David A. Gustafson for his continuous guidance throughout the course of this research and thesis. Thanks are also due to the members of the advisory committee, Dr. Elizabeth A. Unger and Dr. Austin C. Melton.

The author dedicates this work to her parents, Mr. and Mrs. C. K. Hsu. Without their love and encouragement, this work could never have been accomplished. Last but not least, she thanks her husband, Yeewei Huang, for his patience and suggestions.

CHAPTER ONE

INTRODUCTION

Development is the central phase in the software design life cycle; it absorbs at least 75 percent of the cost of a piece of new software (program) [Pressman1982]. Decisions made in this phase will ultimately affect the success of the implementation and maintenance of the software. In spite of the importance, the management of software development is very difficult. Preset schedules and completion dates for a software system can seldom be kept. The quality of the system more often than not becomes suspect as its size grows. These difficulties can be attributed to the limited amount of historical data available to guide a software manager in controlling the progress of the software development project. Therefore, the ability to identify and evaluate the historical data of a program during its development phase is urgently desired; it renders the manager of a software development team able to not only monitor the quality of the program but also regulate the software development cost and schedule.

Traditionally, the quality of software development can be monitored by the technique of complexity measures. This technique tries to measure human factors that affect

software development. Two classical complexity measures are McCabe's Complexity Measure [McCabe1976] and Halstead's metrics [Halstead 1977]. While both measures are sophisticated and mathematically sound, neither provides a vehicle for a quick estimation of the progress of software development.

Dunsmore and Gannon [Dunsmore1977] had a very different view on estimating software complexity. They proposed a measure of complexity to be the number of "program changes" that must be made from the initial version of a program until it is in a final form. The same concepts were found to be employed later in analyzing the style of C programs [Berry1985] and in evaluating software development [Weiss1985]. Recently, Lanchbury [Lanchbury1986] proposed a model to evaluate the progress of a program during its development cycle. The model is empirically oriented; it derives software code change patterns from a successful project. The model aids a software manager to monitor the change pattern of a program.

The purpose of this work is to propose a change classification and a set of intuitive rules for effective evaluation of the program change patterns during software development. The work is basically an extension of Lanchbury's work. It is important that a software manager

sees and interprets the pattern changes during software development. The intuitive rules are designed to facilitate the interpretation of those changes.

In addition to this chapter, this thesis contains 4 chapters. Chapter 2 delineates the nature of the program change data selected to be analyzed and the procedures to collect these data. Chapter 3 presents qualitatively the process of classifying the program change data. This is followed by a quantitative discussion of the classification in Chapter 4. The discussion also leads to the proposal of a set of intuitive rules for program progress analysis using the pattern classification. Concluding remarks and recommendation to future work are given in the last chapter, Chapter 5.

CHAPTER TWO

DATA COLLECTION

The changes in a program, occurring during the software development stage, can be analyzed based on several types of data pertinent to the program. Measures for determining the progress of the software development are extracted from the analyzed results. A software manager can use these measures to evaluate the progress of a program during its development stage.

This chapter discusses the collection of data. The nature of the sample programs under examination is discussed in the first section. This is followed in the second section by a description of some utility software employed in this work. In the third section, the program CHANGES is presented in detail. CHANGES takes a pair of programs as the inputs and yields the file, *main.results*, as the output; the output file contains data about the differences between the input program pair. A discussion on the organization of this data is given in the last section.

2.1 SELECTION OF PROGRAMS TO BE ANALYZED

All the programs analyzed in the present work were written in the programming language C in the Unix

environment. The programs were written by undergraduate students in a fundamental software engineering class (course number CMPSC541, one of the core courses in the undergraduate curriculum in the Department of Computing and Information Sciences at Kansas State University). Software design methodologies were taught in the class. Students were asked to design a program based on those methodologies. Successive versions of the same program were saved during the course of development; they served as the sample programs.

A pair of programs, which are two different versions of the same program, are selected for analysis; more than sixty pairs of programs have been analyzed in this study. Programs are paired based on their size and coding date. Intuitively, two programs with the smallest differences in size and coding dates are successive versions of a program; they are grouped as a pair.

2.2 UTILITY PROGRAMS FOR DATA COLLECTION

Three utility programs, namely COUNT, NESTING, TYPEPGM, were designed as C-shell programs in Unix; each of these programs is a complex awk program. Awk is a Unix data manipulation tool [Bourne1987]; it is a pattern matching language and report generator. The three

programs are described individually in the following sub-sections.

2.2.1 COUNT (usage: awk -f COUNT *programfile*)

COUNT counts the indentation levels for each line of statement in a program. The input program must be in a file designated by *programfile*. *Programfile* must be in a pretty-print format; this can be achieved by preprocessing *programfile* using the Unix command *cb*. The result of COUNT is stored in a file, *countfile*. As an example, Appendix A is a sample C program, named *SampleProgram1*, whose indentation levels are obtained by the utility program COUNT and reproduced in Table 2.1. The source code of COUNT is given in Appendix C.

2.2.2 NESTING (usage: awk -f NESTING *countfile*)

NESTING takes the output file of COUNT, designated by *countfile*, and yields the statistics of the indentation levels of a program. The statistics are stored in an output file; they include the total number and the percentage of each indentation level. The latter is calculated by

```
percentage of level N indentation
  = (total number of level N indentation * 100)
    / total number of lines of code.
```

Moreover, the average indentation level of the *countfile* is defined as

```
indentation level
  = (zero + one*2 + two*3 + three*4 + four*5
    + five*6 + six*7) * 100
    / total number of lines of code.
```

where zero, one, ..., represent the total numbers of indentation level zero, one, ..., respectively. Table 2.2 illustrates the indentation statistics of *Sampleprogram1* in which Zeroave denotes the percentage of level zero indentation, Oneave denotes that of level one indentation, etc. The source code of NESTING is given in Appendix D.

2.2.3 TYPEPGM (usage: awk -f TYPEPGM *programfile*)

TYPEPGM calculates the total number of occurrences of each statement type in the program in the file, *programfile*. Nineteen (19) statement types have been defined. "For ...", "while ...", and "if ..." are examples of different statement types; a complete list of statement types are shown in Table 2.3. The program can be in either pretty-print format or any other free-style format. The total number of statements in the program is also recorded. Each line in the program is analyzed and the type is recorded. The weight of the program is calculated according to the following formula [Gustafson1985],

```
weight  = 18.4 * count["declaration"]
          + 11.4 * count["if"]
          + 7.9 * count["for"]
          + 8.5 * count["while"]
          + 6.8 * count["switch"]
```

```
+ 5.6 * count["case"]
+ 4.6 * count["preprocessor"]
+ 11.1 * count["goto"]
+ 2.4 * count["comment"]
```

in which the weighting of each statement type is defined based on the frequencies of change of individual statement types. Note that only 9 out of the 19 statement types are found in the formula above. This is based on the previous research result which found that the remaining 10 statement types changed in negligible frequencies compared to those listed. Subsequent research in the maintenance phase (An1987) has shown that the program weight is correlated to changes during maintenance. The average weight of an input program is also obtained in TYPEPGM; it is calculated by [Gustafson1985]

average weight = weight / total number of lines of code

Table 2.3 gives the result of processing *SampleProgram1* using TYPEPGM. The source codes of TYPEPGM are given in Appendix E.

2.3 MAIN PROGRAM FOR DATA COLLECTION

A C-shell program, CHANGES, is constructed to combine the utility programs described in the preceeding sections with Unix data manipulation tools and C-shell commands in collecting data for program change analysis. The data manipulating tools in use are

diff find the differences between two files,

grep match patterns in a set of files, and

sed edit a stream.

The C-shell commands employed include:

echo echo a message.

cb beautify a program into an appropriate indentation format.

More details on the Unix data manipulation tools and C-shell commands can be found elsewhere (see, e.g., Bourne, 1987).

CHANGES takes two programs as inputs and generates an output file containing information of individual input files and of the differences between the input files. The command

CHANGES *program1 program2*

invokes the execution of the program CHANGES. Note that for the best results *program1* and *program2* should be chosen based on the criteria discussed in section 2.1. Figure 2.1 depicts a data flow diagram of CHANGES whose complete listing is given in Appendix F. The processes found in the data flow diagram are

Calculating Occurrence of Statement Types,

Pretty-Printing a Program,

Summing the Indentation Level,

Finding the Differences, and

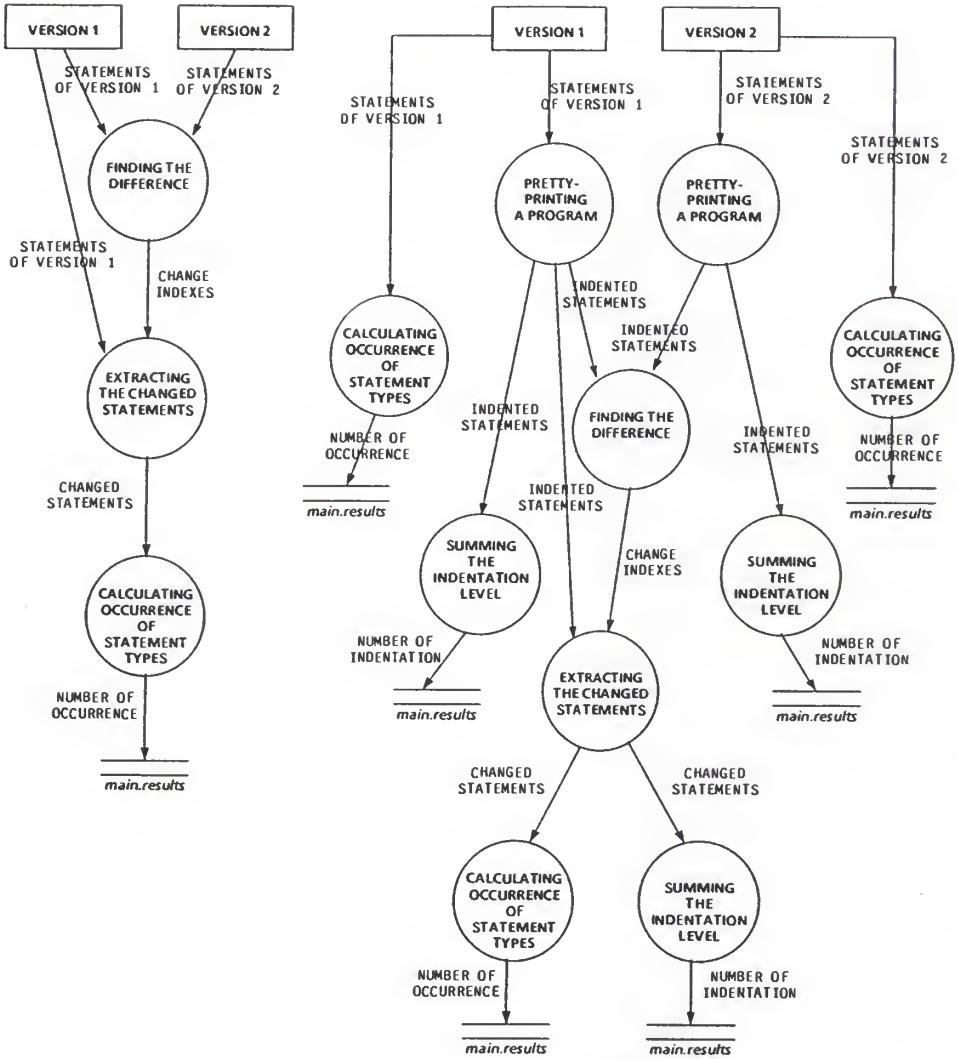


Figure 2.1 Data-flow diagram for the program CHANGES.

Extracting the Changed Statements.

The function of each process is explained, respectively, in each of the following sub-sections.

2.3.1 Calculating Occurrence of Statement Types

This process counts the total number of occurrences of various types of statements in an input program by using the utility program, TYPEPGM. The *sed* command is used inside the process to perform a global stream editing before TYPEPGM is executed. To be exact,

```
sed 's/"/ " /g
     s/}/} /g
     s/{/ {/g'
```

substitute globally """ with " " ", "}" with "} ", and "{" with " {". This assures that the key word of each statement type would not be obscured by some leading or trailing symbols. For example, a statement

```
{if (NF == 0) {if ...
```

will be transformed into

```
{ if (NF == 0) { if ...
```

where in the latter statement, the statement key word "if" can be read clearly by TYPEPGM. In this process, the input program pair are processed independently; the results are saved in the *main.results* file.

2.3.2 Pretty-Printing a Program

This process pretty-prints an input program by using the Unix C-shell command, *cb*. Specifically, it employs the command

cb <*old-file*> *new-file*.

Again, the input program pair are processed individually. The output of this process, *new-file*, is ready to be used as an input for the utility program COUNT.

2.3.3 Summing the Indentation Level

This process uses two awk programs described in section 2.2, namely, COUNT and NESTING. By executing

awk -f COUNT <*input-program*> | awk -f NESTING,

the process counts the indentation level for each line of code and yields the statistics of the indentation levels of the input program. The results of this process are also saved in the file *main.results*.

2.3.4 Finding the Differences

This process finds the differences between two versions of the same program by using Unix data manipulation tools, *diff* and *grep*. The former finds the differences between a pair of input files. For example,

diff -e *programfile1 programfile2*

lists lines that must be changed in *programfile1* to bring it into agreement with *programfile2*. The option "-e" renders the results be recorded in a script of **a**, **c** and **d** commands where a means "statement added", c means "statement changed", and d means "statement deleted"; these commands, when used in the Unix editor *ed*, will recreate *programfile2* from *programfile1*. By piping the results of *diff* to *grep*, or more specifically,

diff -e *programfile1 programfile2*|grep '^[0-9]'.

The differences between *programfile1* and *programfile2* are captured in a set of change indexes, each of which is a line in the format of

line#[a|c]

or

linestart#,lineend#d

where the former implies some codes have been added after line number line# or the specified line has been changed in *programfile1*, and the latter implies that lines number linestart# to lineend# have been deleted in *programfile1*. The results of this step are not saved; instead, it is directly piped to the following process.

2.3.5 Extracting the Changed Statements

This process has two inputs. One is a program, e.g., *programfile1*; the other is the change indexes of the

program. To facilitate the extraction of changed statements from the input program based on the change indexes, those indexes obtained via the process "Finding the Difference" need to be pre-processed by the *sed* command followed by an awk program. This awk program represents each change index in a line expression of the format

NR == NR# {print "[a|b|c|d]",$0 ;i=1}

where NR# is the line number of the statement that has been modified (added, blank, corrected, or deleted). The change indexes in their line expression format are temporarily stored in the file, *result*. By executing

awk -f *result programfile1*,

all statements that have been modified will be collected, while each statement is prefixed by an appropriate label, a, c, or d. To complete the extraction of changed statements, the prefixed statements are piped to another awk program; this awk program singles out those statements prefixed with c and stored them in a file, *temp*. *Temp* is in turn processed by *sed* command, and the result is stored in the file, *final*, which is the output of the process.

2.4 DISCUSSION

Sixty-four pairs of programs have been analyzed in the present research; each pair of the programs belongs to

one of twelve different programs designed by eight teams. After a pair of program is processed by CHANGES as described in section 2.3, the results are appended to the file, *main.results*. Table 2.4 is a sample listing of *main.results* which contains the results of analyzing one pair of programs. (Source Codes for two programs are given in Appendices A and B respectively.) Notice that the listing is divided into four blocks separated by double broken lines. The first block of data are statistics for the first program of the program pair. The second block of data are statistics for the second program. The third block of data are statistics for the changes found in the program pair with both programs being pre-processed by the command *cb*. The fourth block of data are statistics for the changes found in the program pair without each program being pre-processed by the command *cb*.

The elasped time for executing a pair of programs depends on the average size of the program pair. The larger the average size, the longer the elasped time. Table 2.5 summarizes elapsed times for executing different pairs of programs and the average sizes of respective pair of programs.

The data obtained in the VAX Unix environment have been transferred to an Apple Macintosh personal computer

for further analysis. Two awk programs, PICK and SEP, were employed to facilitate the transferring. The former strips off all descriptive part of the data and retains only the file names, the count of each statement type, and the count of each indentation level. The latter separates data pertaining to different pairs of programs into independent files. Appendices G and H are the source codes for PICK and SEP, respectively.

Excel [Townsend1985] has been chosen on Macintosh as the tool for organizing data for program change analysis; it is a spreadsheet software package. Table 2.6 gives an example of data represented in Excel. The Table can be visualized to contain two blocks of data. The first block of data, comprising those in columns A, B and C, are direct representation of data transferred from VAX environment. The second block of data, comprising those in columns D, E and F, are the same as those in the first block except WEIGHT, LOC (LINES OF CODE), TOTAL AVE, SUM, ZERO, ..., SIX. The numbers of WEIGHT, LOC, SUM and TOTAL AVE are normalized; the formulas to normalize WEIGHT and TOTAL AVE are

```
normalized number of WEIGHT or TOTAL AVE
  = (WEIGHT or TOTAL AVE) / 100
```

As an example, the data in cell 22E is calculated by

```
257.40002 / 100 = 2.57.
```

The formulas to normalize LOC and SUM are

```
normalized LOC
  = LOC / 10
```

For example, the data in cell 23E is obtained by

```
104 / 10 = 10.4.
```

Recall that ZERO, ONE, ... represent the total counts of indentation level zero, one, ... respectively; these counts are normalized in the second block by the formula

```
normalized counts of indentation level N
  = (total counts of indentation level N
    / Lines of codes) * 10
```

As an example, the data in cell 24E is calculated by

```
(41/104) * 10 = 3.94.
```

The data contained in the second block have been further expressed in a bar-chart format; the results are illustrated in Figures 2.2 to 2.4. All data collected by means described in this chapter will be analyzed and discussed in detail in the next chapter.

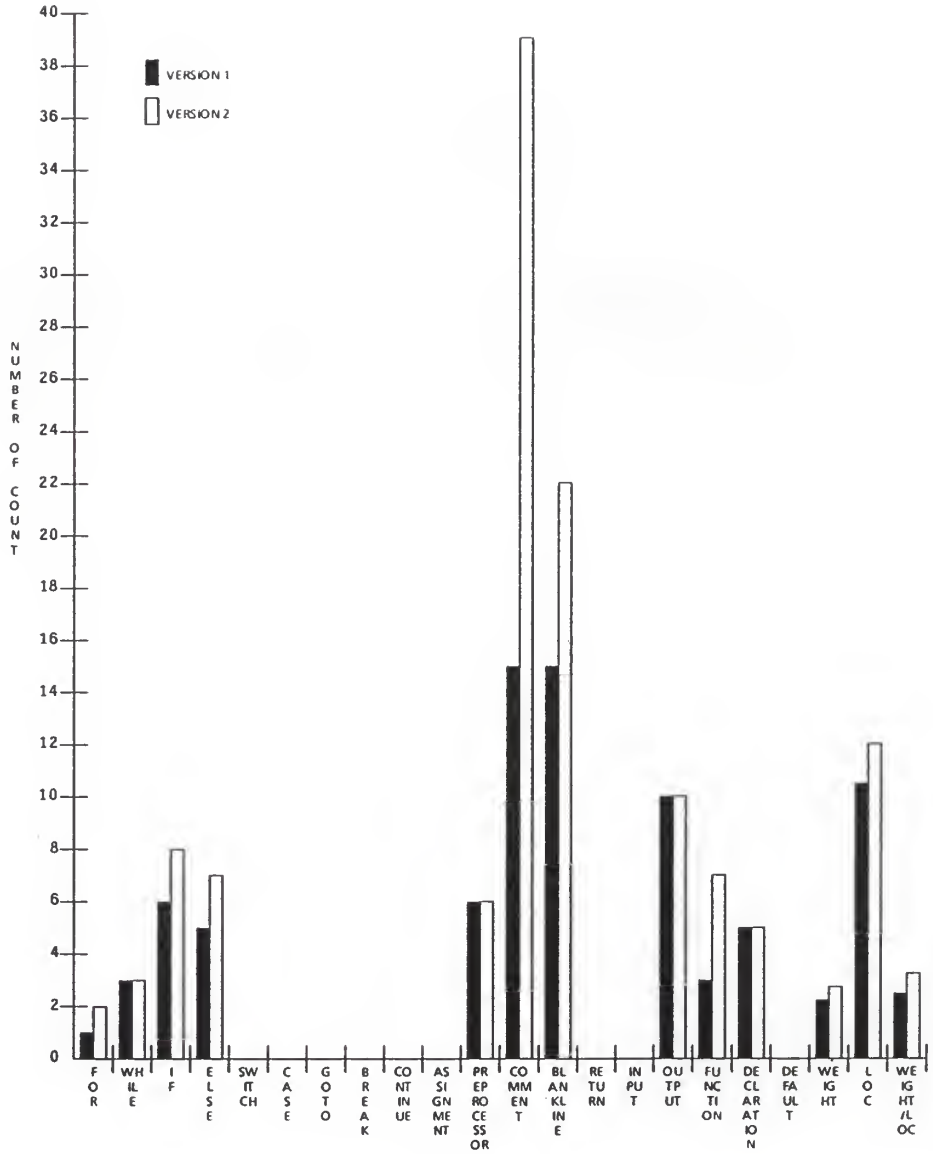


Figure 2.2 Program differences represented in a bar chart in terms of various statement types and lines of code.

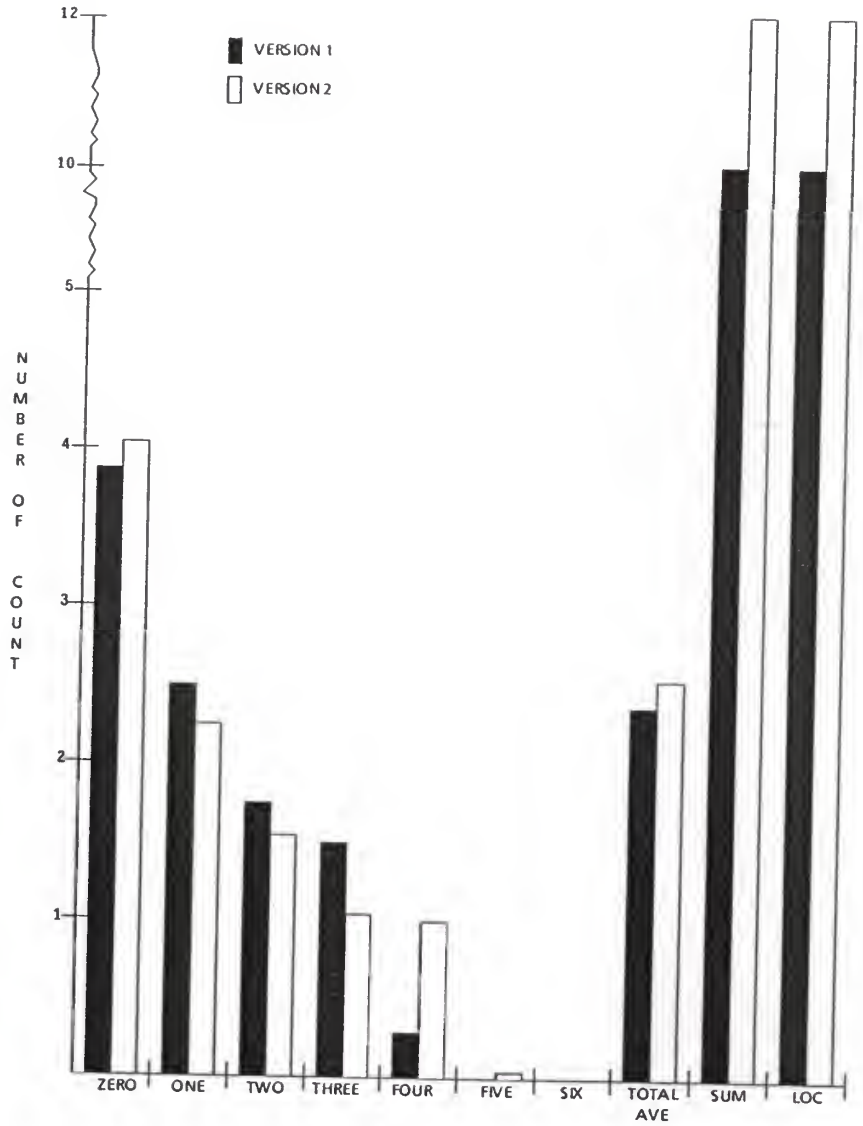


Figure 2.3 Program differences represented in a bar chart in terms of normalized indentation level.

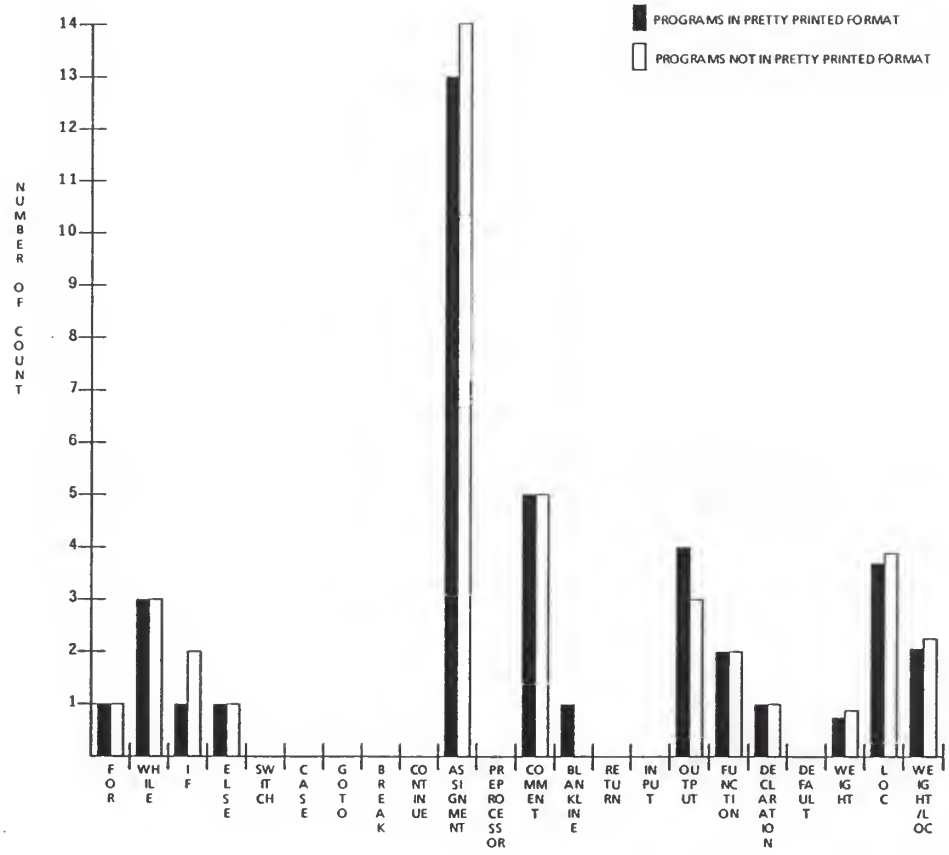


Figure 2.4 Program differences represented in a bar chart in terms of various statement types and lines of code.

Table 2.1 Results of COUNT program for *SampleProgram1*.

statement 1 - 35	statement 36 - 70	statement 71 - 104
0	0	1
0	1	1
0	0	2
0	1	2
0	0	3
0	1	3
0	0	2
0	1	2
0	2	2
0	0	3
0	1	3
0	1	2
0	0	1
0	1	0
0	1	1
0	2	2
0	2	1
0	3	2
0	3	3
0	3	2
0	3	3
0	3	4
0	3	3
0	2	4
0	2	1
0	2	1
0	3	1
0	3	2
0	4	2
0	0	1
0	3	1
1	2	1
1	1	1
1	0	0
1	1	

Table 2.2 Results of NESTING program for *SampleProgram1*.

```
Levels:
~~~~~~~~~~~~~~~

ZERO          41
ONE           26
TWO           18
THREE         16
FOUR           3
FIVE           0
SIX            0
-------------------------------
ZEROAVE =       39.423
ONEAVE =        25.000
TWOAVE =        17.308
THREEAVE =      15.385
FOURAVE =        2.885
FIVEAVE =        0.000
SIXAVE =         0.000
TOTAL AVE       =       217.308
SUM =                       104
LINES OF CODE =             104
SUM/LINES :                1.000
-------------------------------
```

Table 2.3 Results of TYPEPGM program for *SampleProgram1*.

FOR	1
WHILE	3
IF	6
ELSE	5
SWITCH	0
CASE	0
GOTO	0
BREAK	0
CONTINUE	0
ASSIGNMENT	25
PREPROCESSOR	6
COMMENT	15
BLANKLINE	15
RETURN	0
INPUT	0
OUTPUT	10
FUNCTION	3
DECLARATION	5
DEFAULT	0
WEIGHT =	257.40002
LINES OF CODE =	104
WEIGHT/LINES =	2.475

Table 2.4 Listing of *Main.results* for *SampleProgram1* and *SampleProgram2*.

```
%*% This is start of analysis data
Sun Nov  8 16:03:56 CST 1987
=============================================
File Name :    SampleProgram1
---------------------------------------------
FOR                          1
WHILE                        3
IF                           6
ELSE                         5
SWITCH                       0
CASE                         0
GOTO                         0
BREAK                        0
CONTINUE                     0
ASSIGNMENT                  25
PREPROCESSOR                 6
COMMENT                     15
BLANKLINE                   15
RETURN                       0
INPUT                        0
OUTPUT                      10
FUNCTION                     3
DECLARATION                  5
DEFAULT                      0
---------------------------------------------
WEIGHT =              257.40002
LINES OF CODE =             104
WEIGHT/LINES =            2.475
---------------------------------------------
Levels :
~~~~~~~~~~~~~~~

ZERO          41
ONE           26
TWO           18
THREE         16
FOUR           3
FIVE           0
SIX            0
---------------------------------------------
ZEROAVE =        39.423
ONEAVE =            25
TWOAVE =         17.308
THREEAVE =       15.385
FOURAVE =         2.885
FIVEAVE =         0.000
SIXAVE =          0.000
TOTAL AVE        =          217.308
SUM =                           104
LINES OF CODE =                 104
SUM/LINES :                   1.000
---------------------------------------------
```

File Name :	SampleProgram2
FOR	2
WHILE	3
IF	8
ELSE	7
SWITCH	0
CASE	0
GOTO	0
BREAK	0
CONTINUE	0
ASSIGNMENT	28
PREPROCESSOR	6
COMMENT	39
BLANKLINE	22
RETURN	0
INPUT	0
OUTPUT	10
FUNCTION	7
DECLARATION	5
DEFAULT	0

WEIGHT =	345.70001
LINES OF CODE =	120
WEIGHT/LINES =	2.88083

Levels :

ZERO	49
ONE	27
TWO	19
THREE	13
FOUR	12
FIVE	1
SIX	0

ZEROAVE =	40.496
ONEAVE =	22.314
TWOAVE =	15.702
THREEAVE =	10.744
FOURAVE =	9.917
FIVEAVE =	0.826
SIXAVE =	0.000
TOTAL AVE =	229.752
SUM =	121
LINES OF CODE =	121
SUM/LINES :	1.000

File Name :	changes.with.CB
FOR	1
WHILE	3
IF	1
ELSE	1
SWITCH	0
CASE	0
GOTO	0
BREAK	0
CONTINUE	0
ASSIGNMENT	13
PREPROCESSOR	0
COMMENT	5
BLANKLINE	1
RETURN	0
INPUT	0
OUTPUT	4
FUNCTION	2
DECLARATION	1
DEFAULT	0

WEIGHT =	75.20000
LINES OF CODE =	37
WEIGHT/LINES =	2.03243

Levels :

ZERO	7
ONE	16
TWO	9
THREE	5
FOUR	0
FIVE	0
SIX	0

ZEROAVE =	18.919	
ONEAVE =	43.243	
TWOAVE =	24.324	
THREEAVE =	13.514	
FOURAVE =	0.000	
FIVEAVE =	0.000	
SIXAVE =	0.000	
TOTAL AVE =		232.432
SUM =		37
LINES OF CODE =		37
SUM/LINES :		1.000

File Name : changes.without.CB

FOR	1
WHILE	3
IF	2
ELSE	1
SWITCH	0
CASE	0
GOTO	0
BREAK	0
CONTINUE	0
ASSIGNMENT	14
PREPROCESSOR	0
COMMENT	5
BLANKLINE	0
RETURN	0
INPUT	0
OUTPUT	3
FUNCTION	2
DECLARATION	1
DEFAULT	0

WEIGHT =	86.60001
LINES OF CODE =	39
WEIGHT/LINES =	2.22051

Sun Nov 8 16:05:10 CST 1987

Table 2.5 Summary of the elapsed times for executing the 64 pairs of programs and the average sizes of respective pair of programs.

SET OF PROGRAM	1A	1B	1C	2	3	4	5	6	7	8	9	10	11	12
elapsed time	7:35	6:59	7:39	0:43	1:10	7:40	0:48	6:43	3:31	3:39	1:38	1:20	1:17	0:53
size	435	423	431	27	56	439	63	769	105	82	195	42	134	52
elapsed time	6:43	6:28	9:53	0:49	0:55	9:26	0:46	21:27	2:15	3:42	3:58	1:00	1:30	1:15
size	433	435	452	59	56	446	68	1305	107	88	309	51	133	61
elapsed time	8:42	8:14	6:13	0:42	1:40	4:18	0:48	18:46	5:43	2:47	7:15	1:42	1:16	1:23
size	429	448	434	58	67	446	70	1795	110	106	412	82	140	87
elapsed time		8:26		0:41	2:06	7:28	0:50	11:27		3:40	15:17	1:42	1:37	2:06
size		446		36	67	464	66	2216		116	436	80	163	112
elapsed time						8:59		34:34		1:31	3:05	0:13		1:34
size						449		2413		118	424	58		125
elapsed time								19:01			4:52	1:14		1:22
size								1456			448	51		90
elapsed time											2:54			
size											472			

Table 2.6 Sample data represented in Excel; the data reveal the difference between a pair of programs.

	A	B	C	D	E	F
1		P1	P2			
2	FOR	1	2	FOR	1	2
3	WHILE	3	3	WHILE	3	3
4	IF	6	8	IF	6	8
5	ELSE	5	7	ELSE	5	7
6	SWITCH	0	0	SWITCH	0	0
7	CASE	0	0	CASE	0	0
8	GOTO	0	0	GOTO	0	0
9	BREAK	0	0	BREAK	0	0
10	CONTINUE	0	0	CONTINUE	0	0
11	ASSIGNMENT	25	28	ASSIGNMENT	25	28
12	PREPROCESSOR	6	6	PREPROCESSOR	6	6
13	COMMENT	15	39	COMMENT	15	39
14	BLANKLINE	15	22	BLANKLINE	15	22
15	RETURN	0	0	RETURN	0	0
16	INPUT	0	0	INPUT	0	0
17	OUTPUT	10	10	OUTPUT	10	10
18	FUNCTION	3	7	FUNCTION	3	7
19	DECLARATION	5	5	DECLARATION	5	5
20	DEFAULT	0	0	DEFAULT	0	0
21	WEIGHT	257.4	345.70	WEIGHT	2.57	3.46
22	LOC	104	120	LOC	10.4	12
23	WEIGHT/LOC	2.88	2.48	WEIGHT/LOC	2.48	2.88
24	ZERO	41	49	ZERO	3.94	4.08
25	ONE	26	27	ONE	2.50	2.25
26	TWO	18	19	TWO	1.73	1.58
27	THREE	16	13	THREE	1.54	1.08
28	FOUR	3	12	FOUR	0.29	0.08
29	FIVE	0	1	FIVE	0	0
30	SIX	0	0	SIX	0	0
31	ZEROAVE	39.42	40.50	TOTAL AVE	2.17	2.30
32	ONEAVE	25.00	22.31	SUM	10.4	12.1
33	TWOAVE	17.31	15.70	LOC	10.4	12.1
34	THREEAVE	15.39	10.74	SUM/LOC	1	1
35	FOURAVE	2.89	9.92			
36	FIVEAVE	0	0.83			
37	SIXAVE	0	0			
38	TOTAL AVE	217.31	229.75			
39	SUM	104	121			
40	LOC	104	121			
41	SUM/LOC	1	1			

	WITH CB	WITHOUT CB			
FOR	1	1	FOR	1	1
WHILE	3	3	WHILE	3	3
IF	1	2	IF	1	2
ELSE	1	0	ELSE	1	1
SWITCH	0	0	SWITCH	0	0
CASE	0	0	CASE	0	0
GOTO	0	0	GOTO	0	0
BREAK	0	0	BREAK	0	0
CONTINUE	0	0	CONTINUE	0	0
ASSIGNMENT	13	14	ASSIGNMENT	13	14
PREPROCESSOR	0	0	PREPROCESSOR	0	0
COMMENT	5	5	COMMENT	5	5
BLANKLINE	1	0	BLANKLINE	1	0
RETURN	0	0	RETURN	0	0
INPUT	0	0	INPUT	0	0
OUTPUT	4	3	OUTPUT	4	3
FUNCTION	2	2	FUNCTION	2	2
DECLARATION	1	1	DECLARATION	1	1
DEFAULT	0	0	DEFAULT	0	0
WEIGHT	75.2	86.6	WEIGHT	0.75	0.87
LOC	37	39	LOC	3.7	3.9
WEIGHT/LOC	2.03	2.22	WEIGHT/LOC	2.03	2.22
ZERO	7		ZERO	1.89	
ONE	16		ONE	4.32	
TWO	9		TWO	2.43	
THREE	5		THREE	1.35	
FOUR	0		FOUR	0	
FIVE	0		FIVE	0	
SIX	0		SIX	0	
ZEROAVE	18.92				
ONEAVE	43.24				
TWOAVE	24.32				
THREEAVE	13.51				
FOURAVE	0				
FIVEAVE	0				
SIXAVE	0				
TOTAL AVE	232.43				
SUM	37				
LOC	37				
SUM/LOC	1				

CHAPTER THREE

CLASSIFICATION OF PROGRAM CHANGES

The classification of program changes can help a software manager in assessing the progress of a program. Such a classification can be determined after data concerning the changes of a program development are read and the patterns are classified. While issues on the data collection have been discussed in the previous chapter and those on intuitive rules will be elaborated in the next chapter, the process of classifying the program change data will be studied in this chapter.

The first section of this chapter presents a preliminary analysis of the program change data collected. This is followed by a detailed analysis in the second section. Classification of program change patterns based on the results of the analysis is proposed in the third section. The advantages of the proposed classification types are discussed in the last section.

3.1 PRELIMINARY ANALYSIS

Three kinds of statistics were examined in the preliminary analysis; they were

(1) the difference of the counts of each statement type between a pair of non-pretty-printed programs,

(2) the difference of the counts of each indentation level between a pair of pretty-printed programs,

(3) total number of statements for each type which have been modified between a pair of pretty-printed programs, and

(4) the number of statements for each type which have been modified between a pair of non-pretty-printed programs.

These statistics provide data to estimate qualitatively the progress of a software development project.

As an example, the counts of each statement type for a non-pretty-printed program pair and those of each indentation level for a pretty-printed program pair are reproduced in Table 3.1. From the Table, we observe that on one hand the counts of two (out of nineteen) statement types, namely, DECLARATION and OUTPUT, decrease from the first version to the second version. On the other hand, six statement types, namely, FOR, IF, ASSIGNMENT, COMMENT, BLANKLINE, and FUNCTION, have their counts increase from the first version to the second version. The counts of the remaining eleven statement types are the same in the

two versions. These statistics reflect a poor design or incomplete design specifications; they indicate a lack of progress, so development would still be continuing (Lanchbury1986).

Also in Table 3.1, the counts for the indentation levels of ZERO, ONE, TWO, and THREE in the second version are less than those in the first; the counts of the remaining indentation levels in the second version are greater than those in the first. The changes in the counts of various indentation levels indicate an existence of structural changes in the program.

Table 3.2 presents a set of sample data for preliminary analyses (3) and (4). Data in column B were obtained after two versions of a program have been converted into appropriate indentation, i.e., pretty-printing. Data in column C were obtained based on two versions of a non-pretty-printed program. Comparing column B with column C, we see that the data in the latter are consistently greatly than those in the former. Such a pattern typifies the changes of a piece of software effected by pretty-printing.

3.2 DETAILED ANALYSIS

While the preliminary analyses are straightforward and enable a quick estimate to the progress of software

development, they do not provide insight into the progress. Nonetheless, it is worth mentioning that data collected in the form of Tables 3.1 and 3.2 actually contain more information than revealed in the preliminary analyses. A detailed analysis at data may suggest that a variety of activities can be emphasized during the development of a program. For example, some development may emphasize the enhancement which means that different types of statements are added in a program; others may emphasize the deletion which means that different types of statements are removed from a program. In this work, ten classes of program change patterns have been summarized to embody most activities occurring in software development. This classification of program change patterns is based on a detailed analysis of data belonging to sixty-four pairs of program; the patterns include:

(1) Debugging,

(2) Documentation,

(3) Correction,

(4) Pretty-printing,

(5) Reconstruction,

(6) Removing documentation,

(7) Removing functionality,

(8) Adding functionality,

(9) Removing debugging, and

(10) Redistribution.

The definition for each of the ten patterns can be found in Table 3.3. Notice that more than one pattern may occur when a program changes from a version to the next.

3.3 DISCUSSION ON THE CLASSIFICATION OF PROGRAM CHANGE PATTERNS

Ten types of classification have been identified. The characteristics for each type of patterns are described in the following sub-sections.

3.3.1 Debugging

When a program is debugged, the counts of output statements will be increased. The statements are added to monitor the behavior of the program; they include "putchar", "putc", "printf", "fprintf", "printw", "write", "puts" and "fputs". Figure 3.1 gives a sample program change data set belonging to a program under debugging. In the Figure, we observe that a number of output statements have been added to the second version of the program.

3.3.2 Documentation

Documentation refers to adding comments to a program; it makes the program more understandable. Needless to

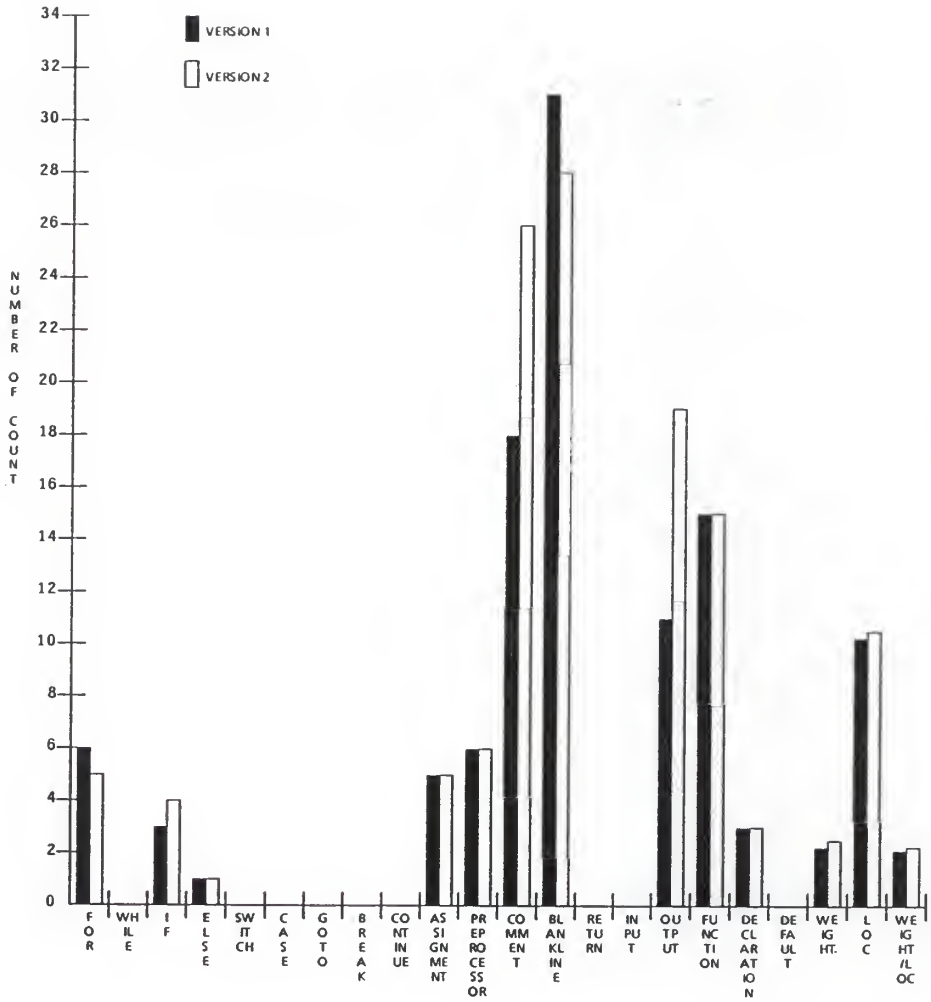


Figure 3.1 Sample program change data set belonging to a program under debugging, documentation and correction.

say, documentation results in an increase in the count of the comment statements. For example, eight additional comments have been added when the program changes from one version to the next as depicted in Figure 3.1.

3.3.3 Correction

A program change pattern is categorized as that of correction when the following conditions happen.

(1) Number of lines of code shows minor difference between two successive versions of the program.

(2) The counts of the control statements, FUNCTION, DECLARATION and ASSIGNMENT show minor changes between two successive versions of the program. The control statements includes FOR, WHILE, IF, ELSE, SWITCH, CASE, GOTO, BREAK, RETURN and CONTINUE.

The changes could be addition or deletion of a few lines of code of various statement types, including control statements, FUNCTION, DECLARATION and ASSIGNMENT.

A program change pattern of correction can be seen, again, in Figure 3.1. In the Figure, we observe, among other things, that one FOR statement has been deleted and

one IF statement added in the second version. The trivial changes indicate that the program is under correction.

3.3.4 Pretty-Printing

A program is pretty printed when its codes are displayed with proper spacing and indentation. The purpose of pretty printing is to make the structure of a program explicit.

Consider two versions of a program with a number of statements properly indented in the second version but not in the first, these statements are identified to be changed between the two versions. Nonetheless, the changes will become non-identifiable when both versions of the program are converted into their respective pretty-printed forms since pretty printing results in a unique display of the program. The analysis leads us to propose the following procedures to identify whether or not pretty printing is imposed between two versions of a program.

(1) Find the differences between two successive versions of the program based on the method outlined in section 2.3.4.

(2) Repeat step (1) except that pretty printing both versions of the programs before finding the differences.

(3) Compare the results of steps (1) and (2).

(4) If the results from steps (1) and (2) are identical, no pretty printing is involved in developing the program from version one to version two.

(5) If the results show a general trend of more statements being different in step (1) than those in step (2), we conclude that attempts were made to pretty print the program in version two.

A result of the analysis based on the proposed procedures is demonstrated in Figure 3.2. In the Figure, the height of a bar represents the degree of difference between two successive versions of program. A bar with the label "FOR" of the height of three means that three FOR statements are different between the two versions. The empty bars are obtained by step (1) while the solid ones by step (2). The fact that most of the empty bars are higher than the solid ones indicates that an attempt has been made to put the second version of the program into a properly indented format.

3.3.5 Reconstruction

In the preceding sub-section, we have shown that proper indentations is capable of fully expressing the "structure" of a program written in a structural

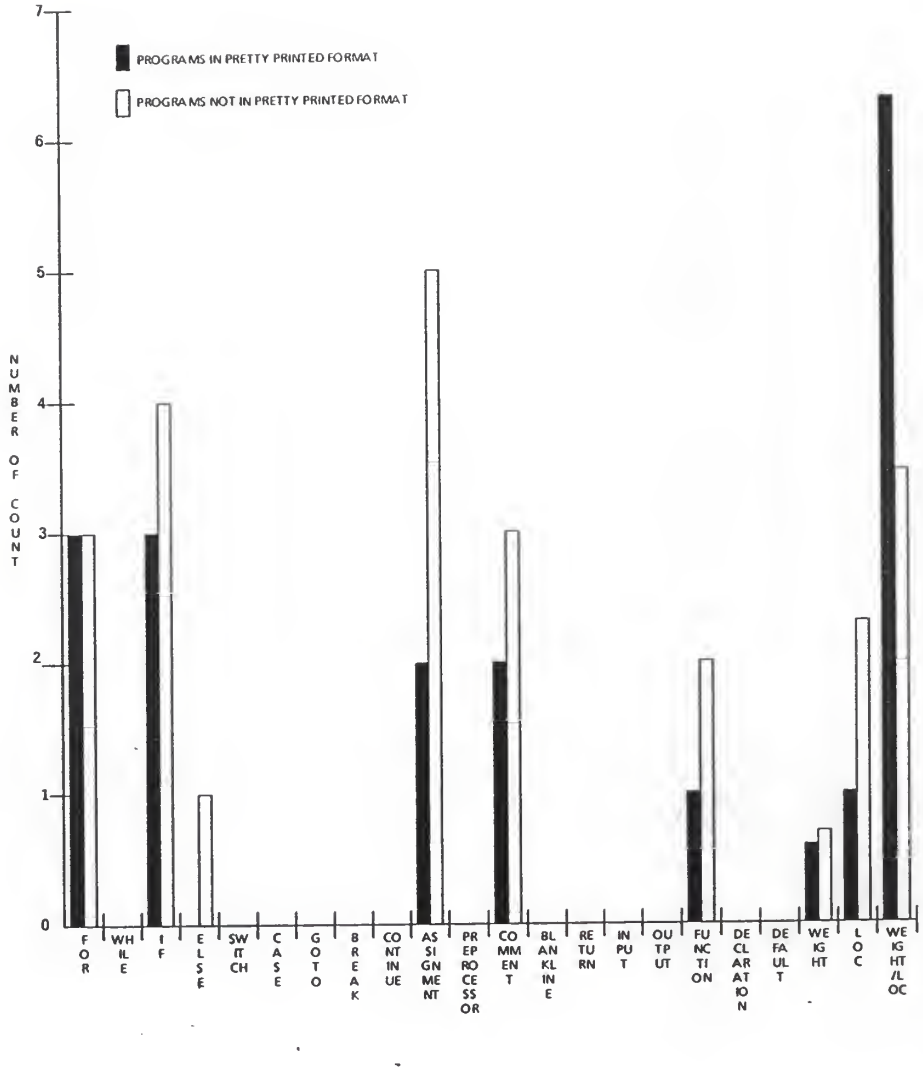


Figure 3.2 Sample program change data set belonging to a program under pretty-printing.

programming language such as C. When the structure of the program is altered, i.e., when the program is reconstructed, the alternation (reconstruction) manifests itself in changes of the indentation patterns.

Let's consider the indentation patterns of two successive versions of a program, assuming that the total lines of code in the second version is greater than that of the first by N. If the program has been "reconstructed" from the first to the second version, we should observe that the change in the count of indentation level i is n_i (i = 0 to 6) with

(1) n_i's not showing a general trend of increasing, and

(2) any of the n_i's in the second version being significantly larger or smaller than the n_i's in the first version.

Figure 3.3 depicts an example of reconstruction. In the Figure, the lines of code of the second version increase. According, we expect to see a general trend of increasing in the counts of each indentation level in the second version. However, we observe that the counts of indentation levels 2, 3 and 4 decrease, and the counts of indentation levels 0, 1, 5 and 6 increase. The further analysis tells us that the count of indentation level 3

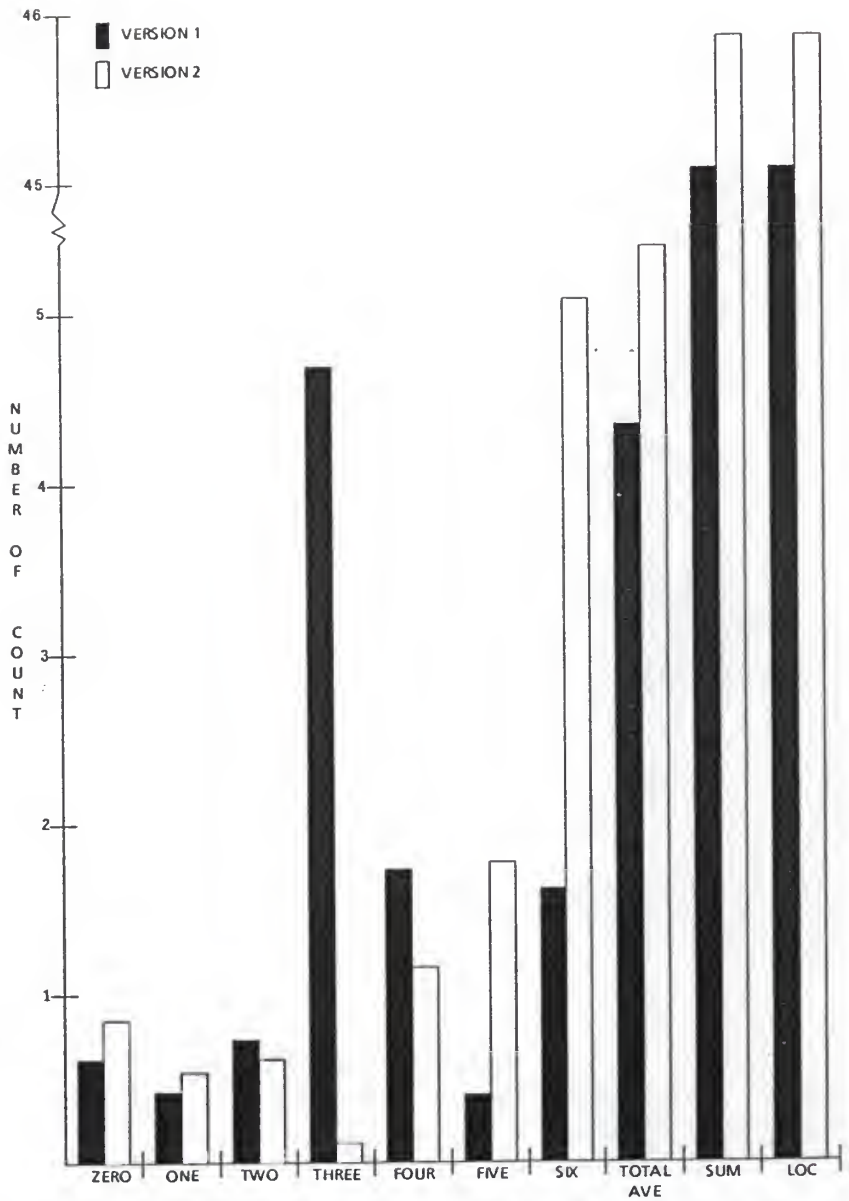


Figure 3.3 Sample program change data set belonging to a program under reconstruction.

significantly decreases and the count of indentation level 6 significantly increases.

3.3.6 Removing Documentation

Removing documentation is the reverse process of that described in section 3.3.2; it results in a decrease in the count of the COMMENT statements. Figure 3.4 depicits an example of removing documentation. In the Figure, we observe that two COMMENTS are removed from the second version of the program.

3.3.7 Removing Functionality

Removing functionality is concerned with the deletion of control statements, FUNCTION, ASSIGNMENT, include-file (PREPROCESSOR) and DECLARATION from a program. Remembering the definition of correction, the removing functionality is recognized when the lines of code have significant changes between two successive versions of a program.

In Figure 3.4, we find that the counts of FUNCTION and DECLARATION in the first version are more than those in the second version. This change pattern typifies the process of removing functionality.

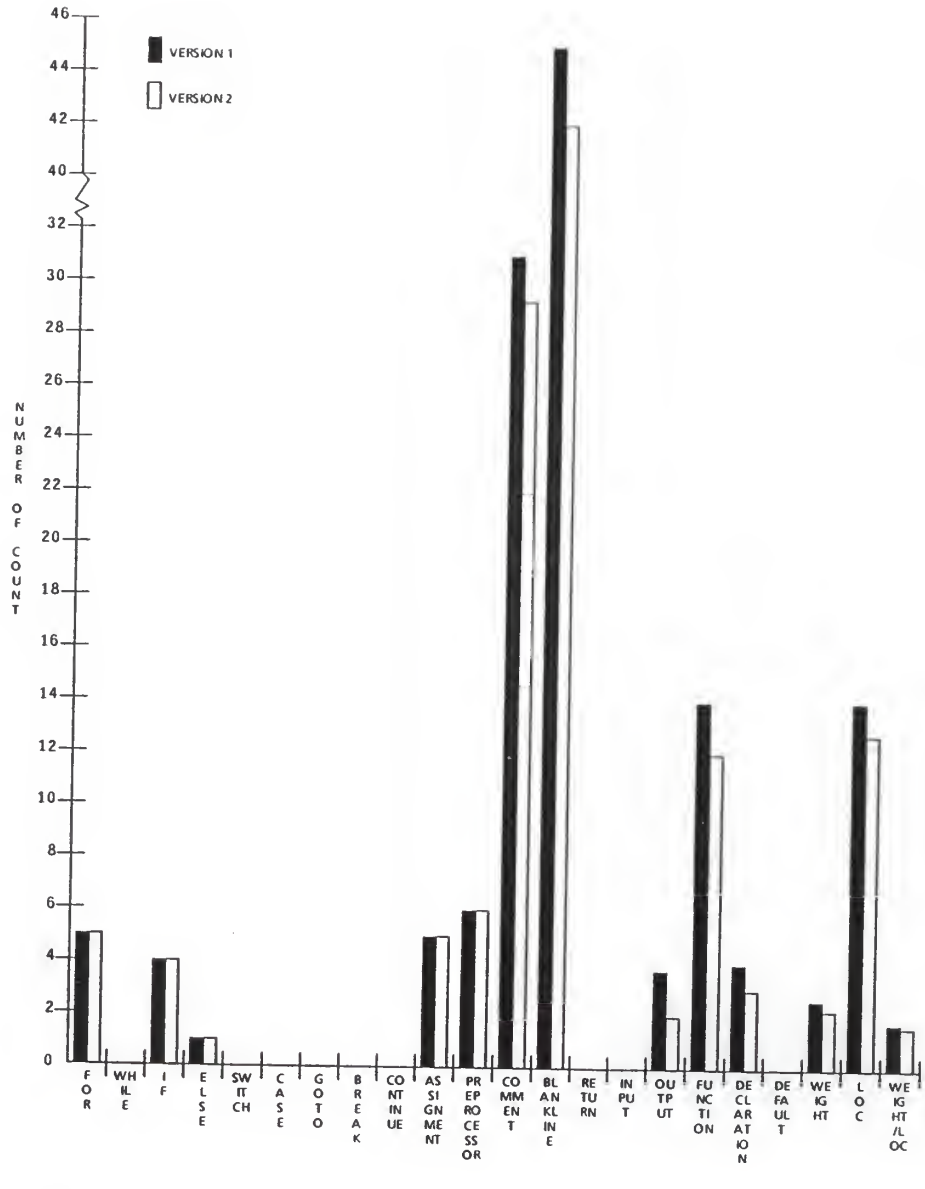


Figure 3.4 Sample program change data set belonging to a program under removing documentation and removing functionality.

3.3.8 Adding Functionality

The reverse process of "removing functionality" is "adding functionality". This process involves the addition of control statements, FUNCTION, ASSIGNMENT, include-file (PREPROCESSOR) and DECLARATION. The same as removing functionality, the addition of the mentioned various type statements with significant changes of lines of code on the second version of a program are typified as adding functionality. An example showing the program change pattern of adding functionality is illustrated in Figure 3.5. In the Figure, we observe that the counts of FUNCTION and DECLARATION increase in the second version of the program.

3.3.9 Removing Debugging

When software development reaches its final stage, those statements inserted for the purpose of debugging need to be removed. A program change pattern reflecting this process is called "Removing Debugging"; the pattern shows a decrease in the OUTPUT statements between two successive versions of the program. Figure 3.5 gives an example of removing debugging. Ten output statements are removed from the first version of program in this example.

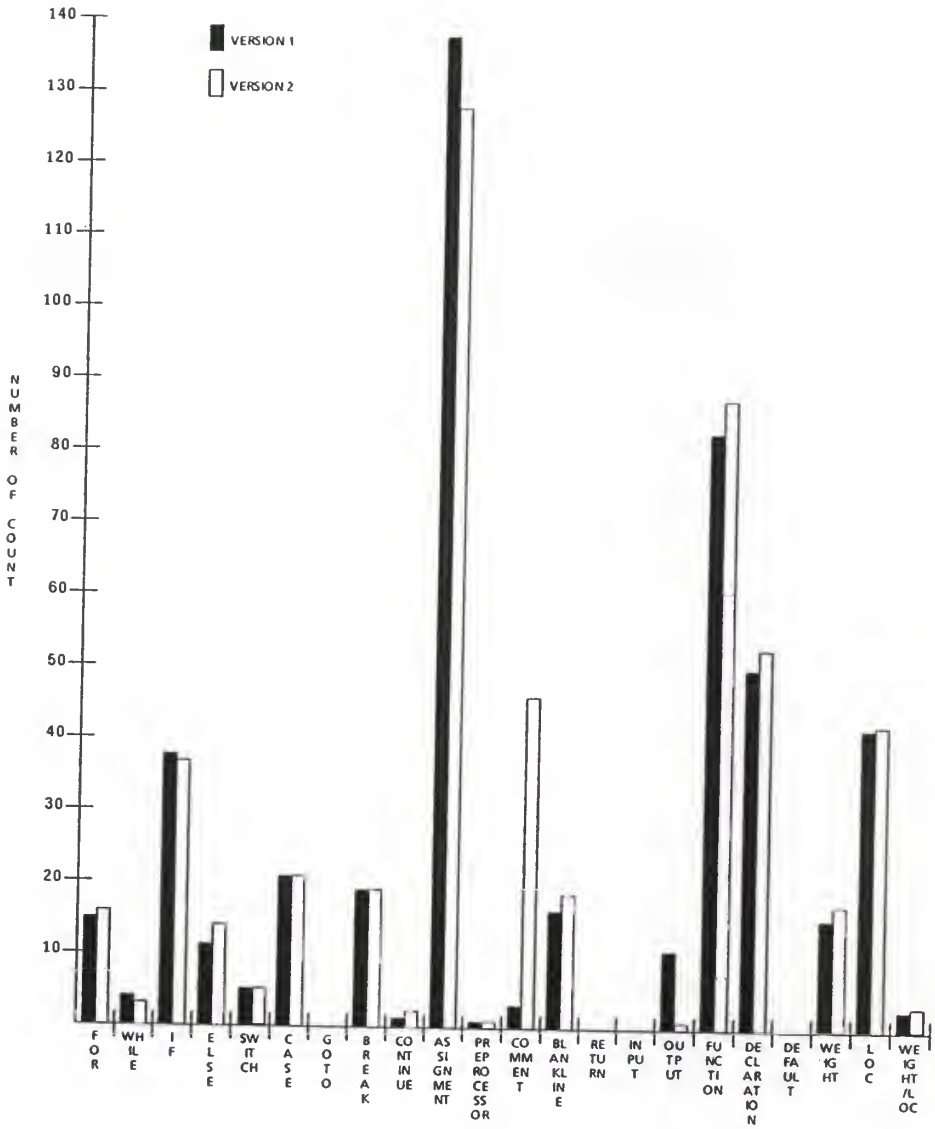


Figure 3.5 Sample program change data set belonging to a program under adding functionality and removing debugging.

3.3.10 Redistribution

Redistribution refers to changing include-file (PREPROCESSOR) and FUNCTION. Specifically, a program is recognized to be redistributed when either one of the followings happens.

(1) One or more include-file (PREPROCESSOR) is added in conjunction with one or more FUNCTION being deleted.

(2) One or more include-file (PREPROCESSOR) is deleted in conjunction with one or more FUNCTION being added.

Figure 3.6 gives an example of redistribution. In the Figure, we observe that the count of include-file (PREPROCESSOR) decreases with simultaneous increases in the count for FUNCTION. The observation tells us that redistribution has taken place.

3.4 ADVANTAGES OF PROPOSED CLASSIFICATION

The advantages of this classification are outlined as follows.

(1) Facilitate the identification of intuitive rules for program change analysis. It is worth noting that a myriad of changes can be made when a program progresses from one version to the next. To extract intuitive rules from the large amount

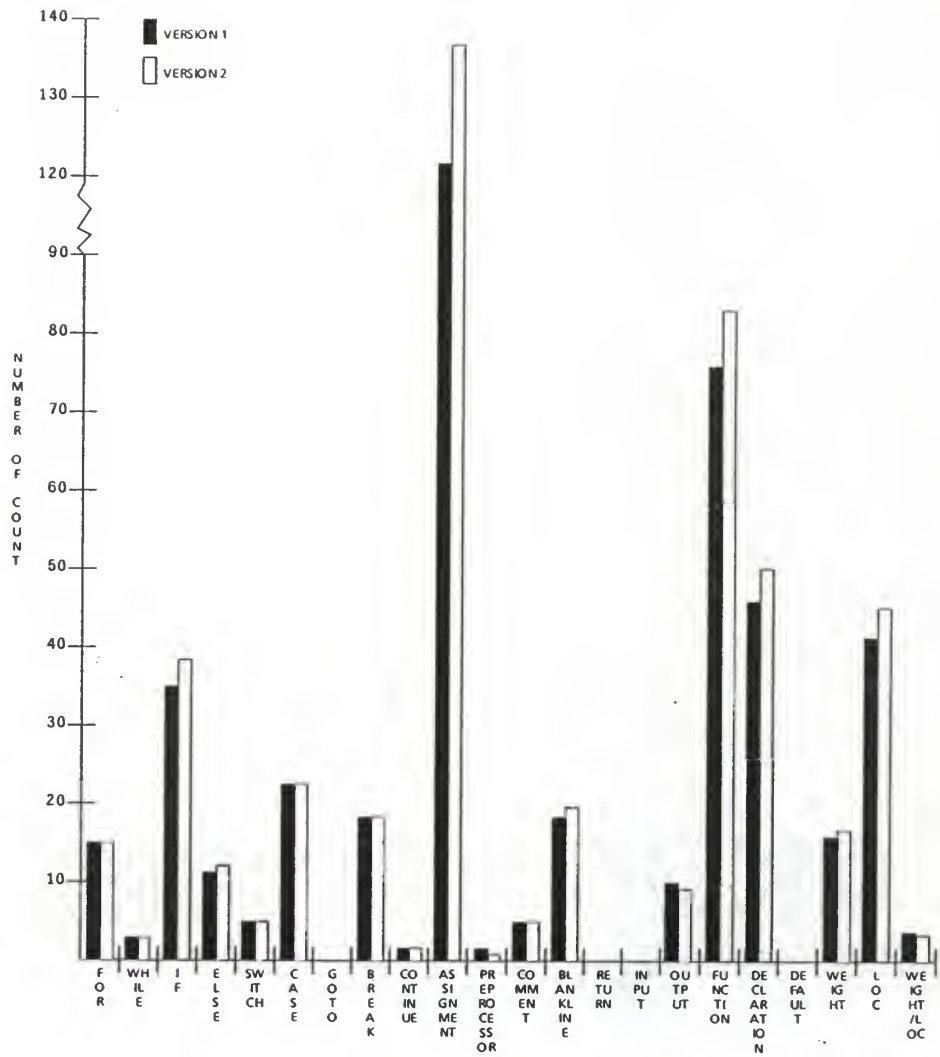


Figure 3.6 Sample program change data set belonging to a program under redistribution.

of change data is almost impossible. The proposed classification collects relevant data in small groups. Extracting rules from each group of limited amount of data is much easier.

(2) Enhance the reliability of program change analysis. When only the relevant data are collected in groups, the analysis based on each small group is more "noise-free", i.e., each step of analysis will not be influenced by irrelevant data. Notice that intuitive rules derived by the noise-free analysis are more reliable.

(3) Render the progress of software development more assessable. With program change data well organized, a technical or non-technical software manager may be able to assess the progress of the software development at a glance.

Table 3.1 Sample data collected for preliminary analysis steps (1) and (2). Rows 1 to 23 are obtained from a pair of non-pretty-printed program; rows 24-41 are from the same pair of program after they are both pretty-printed.

	A	B	C
1		Version 1	Version 2
2	FOR	5	8
3	WHILE	0	0
4	IF	4	8
5	ELSE	1	1
6	SWITCH	0	0
7	CASE	0	0
8	GOTO	0	0
9	BREAK	0	0
10	CONTINUE	0	0
11	ASSIGNMENT	5	6
12	PREPROCESSOR	6	6
13	COMMENT	31	49
14	BLANKLINE	45	59
15	RETURN	0	0
16	INPUT	0	0
17	OUTPUT	4	0
18	FUNCTION	14	20
19	DECLARATION	4	3
20	DEFAULT	0	0
21	WEIGHT	2.61	3.55
22	LOC	14	18.4
23	WEIGHT/LOC	1.86	1.92
24	ZERO	5.57	5.27
25	ONE	1.5	0.82
26	TWO	0.36	0.27
27	THREE	0.57	0.43
28	FOUR	0.5	0.71
29	FIVE	0.36	0.65
30	SIX	1.14	2.17
38	TOTAL AVE	2.46	3.11
39	SUM	14	19
40	LOC	14	19
41	SUM/LOC	1	1

Table 3.2 Sample data collected for preliminary analysis steps (3) and (4).

	A	B	C
1		WITH CB	WITHOUT CB
2	FOR	0	0
3	WHILE	0	0
4	IF	2	3
5	ELSE	0	1
6	SWITCH	0	0
7	CASE	0	0
8	GOTO	0	0
9	BREAK	0	0
10	CONTINUE	0	0
11	ASSIGNMENT	1	3
12	PREPROCESSOR	1	1
13	COMMENT	4	5
14	BLANKLINE	0	1
15	RETURN	0	0
16	INPUT	0	0
17	OUTPUT	0	0
18	FUNCTION	0	0
19	DECLARATION	1	1
20	DEFAULT	0	0
21	WEIGHT	0.55	0.69
22	LOC	1.4	1.9
23	WEIGHT/LOC	3.96	3.64
24	ZERO	2.86	
25	ONE	1.43	
26	TWO	0.71	
27	THREE	2.14	
28	FOUR	0.71	
29	FIVE	1.43	
30	SIX	0.71	
31	TOTAL AVE	3.36	
32	SUM	1.4	
33	LOC	1.4	
34	SUM/LOC	1	

Table 3.3 Classification of changes and its description.

TYPE	DESCRIPTION
Debugging	Output statements are added to monitor the behavior of a program.
Documentation	Comments are added in a program to render it more understandable.
Correction	Errors are corrected in a program.
Pretty-printing	Indenting statements are to reflect level of nesting.
Reconstruction	The number of indentation for each level does not display the consistent with the trend of decreasing/increasing of lines of code of a program.
Removing documentation	Comments are removed from a program.
Removing functionality	Function, assignment, declaration and preprocessor are remved from a program.
Adding functionality	Function, assignment, declaration and preprocessor are added from a program.
Removing debugging	Output statements are removed from a program. This is to undo the debugging.
Redistribution	Removing function plus adding preprocessor; adding function plus removing preprocessor.

CHAPTER FOUR

CLASSIFICATION RULES

The changes of a program between two successive versions are readily seen when the change pattern classification, discussed in Chapter 3, are employed to render the patterns of changes explicit. While discussions in the previous chapter are qualitative in nature, this chapter endeavors to shed light on the quantitative aspect of the classification by using PROBIT [Finney1971] and propose a set of intuitive rules for program progress analysis using the pattern classification. PROBIT is a statistical procedure which calculates maximum-likelihood estimates of the intercept, slope, and natural (threshold) response rate for biological assay data.

We commence this chapter by giving a justification of using PROBIT as the tool for quantitative analysis of the classification. This gives rise to the identification of certain threshold values crucial in quantifying each type of classification. The set of intuitive rules guiding the use of the classification in change analysis of a program is then presented. An example will be given to demonstrate the applicability of the intuitive rules.

4.1 PROBIT

PROBIT [Finney1971] is a statistical procedure specialized for dose-response problems in bioassays. For some stimulus-subject systems, measurement of a response to the action of the stimulus is impossible or impractical; all that can be done is to record whether or not the subject manifests a certain reaction. The quantal response so used can be death or any other easily recognizable change in the subject. For example, an insecticide (stimulus) may be assayed by assigning batches of insects (subjects) to various doses and then analyzing the relation between death-rate and dose. Note that each subject can be used only once. An insect that has died cannot be used again; even insects that are not dead may have been affected by the stimulus that thereafter they react differently from others not previously exposed to the stimulus.

How to determine a threshold value for each type of classification is basically a dose-response problem. The analogy can be elaborated as follows by using the change pattern class of debugging as an example:

1. The dosage in this case is the number of output statements showing up in a change pattern. The goal is to determine a number (dose) beyond which

the change pattern can be classified as debugging.

2. A subject in this case is a change pattern between two successive versions of a program. As mentioned earlier, 64 pairs of programs have been analyzed, i.e, 64 change patterns (subjects) can be identified. A batch of subjects consists of all change patterns with the same number of output statements added. This definition conforms with the requirement that all subjects in a batch receive the same level of dose. For example, all change patterns showing an increase of 2 output statements will be grouped in one batch and those showing an increase of 3 will be grouped in another batch. Note that each change pattern (like each insect) is unique in its own right and each batch of change patterns can not be reused.

3. A change pattern (subject), after closely reviewed by a software engineer, will be determine whether or not it responds to the dose. The response is quantal. A positive response means that the change pattern indeed belongs to the class of debugging; a negative response means that the change pattern does not belong to the

class of debugging although some output statements were added.

The discussion above has explained the use of PROBIT procedure for quantitative analysis of the classification.

4.2 QUANTITATIVE STUDY ON THE CLASSIFICATION

A quantitative description for each classification is defined in this section. The PROBIT procedure in the Statistical Analysis System [SAS1982] is employed to aid the quantitative analysis.

4.2.1 Debugging

Twenty-three of the sixty-four sets of programs in the current research have been observed to exhibit an increase in output statements. The increase ranges from 1 to 32; a summary of the changes is given in Table 4.1. It appears from the Table that the number of output statements added for debugging is independent on the size of the program. Further statistical analysis yields that the sample correlation coefficient between the size of the program and the number of output statements added for debugging is 0.73, implying that the two quantities indeed do not significantly correlate with each other.

To determine a threshold value for this type of change pattern, the data in Table 4.1 is reorganized

according to the discussion in the preceding section and tabulated in Table 4.2. Note that in the Table, data are in the format of DOSE-SUBJECT-RESPONSE; they are ready for PROBIT analysis. To read the Table, the 6th entry, for example, means that 2 change patterns have been found to experience an increase of 9 output statements; only one of them are found to experience debugging. The SAS program incorporating the PROBIT procedure along with the output for this analysis is reproduced in Appendix K. The most important information contained in the output is the table listing the threshold dose along with the 95% fiducial limits for different probability levels (see the last table in Appendix K). The 95% fiducial limits are computed using a t value of 1.96 since the chi-square is small. Note that in all the analyses in this study, the chi-squares are small indicating that the linearity of the data is good. However, it should be pointed out that the width of 95% fiducial intervals can be very big as the probability level increase. This is expected since human factor in software development can be very stochastic; it is extremely difficult to interpret to a high precision a program change pattern. Note that in some extreme cases, the 95% fiducial limits will be marked by a period (.) in the SAS output.

Based on the result, we can conclude that

if two successive versions of a program exhibit an increase of more than 5 (23) lines of output statements,

then there is a 50 (90) percent chance that the program has experienced a change in terms of debugging.

4.2.2 Documentation

Thirty-eight of the sixty-four sets of programs analyzed in the current research have been observed to possess an increase in comment statements. In these 38 sets of programs, a minimum of 1 and a maximum of 284 comments were added for the purpose of documentation. Table 4.3 summarizes the change in the number of comment statements in the 38 sets of programs. Notice that the correlation coefficient between the size of the program and the number of comments added for documentation is only 0.44, indicating strongly that the two quantities do not correlate. The result of PROBIT analysis of this case is given in Table 4.4, based on which we conclude that

if two successive versions of a program exhibit an increase of more than 1 (16) lines of COMMENT statements,

then there is a 50 (90) percent chance that the program has experienced a change in terms of documentation.

4.2.3 Correction

In the current research, 16 of the 64 sets of programs appear to possess this pattern of change, i.e., they have gone through some minor changes in FUNCTION, DECLARATION, ASSIGNMENT and control statements. Table 4.5 gives a breakdown of the activities involved in those 16 sets of programs. Furthermore, the result of PROBIT analysis of this case is shown in Table 4.6. Based on the result, we conclude that

if number of lines of code exhibits less than 10 lines different between two successive versions of a program

and two successive versions of a program exhibit more than a total of 1 (4) lines of change in FUNCTION, DECLARATION, ASSIGNMENT or control statements,

then there is a 50 (90) percent chance that the program has experienced, positively or negatively, a change in terms of correction.

4.2.4 Pretty-Printing

The change of a program is significant in terms of pretty-printing if certain percentage of changes in statement types is caused by print-printing. The threshold percentage has been identified by PROBIT, and the result is presented in Table 4.7. Based on the result, we conclude that

> if M statement types are changed between two successive versions of a program,
>
> and N statement types are determined to have gone through pretty-printing,
>
> and N/M > 0.1 (0.7)
>
> then there is a 50 (90) percent chance that the program has experienced, positively or negatively, a change in terms of pretty-printing.

Detail procedures to obtain M and N can be found in Section 3.3.4.

4.2.5 Reconstruction

The change of a program is significant in terms of reconstruction if changes are found in more than certain percentage of all indentation levels. In the current research, 25 of the 64 sets of programs appears to have gone through different degrees of reconstruction. Table

4.8 summarizes the activity of those 25 sets of programs. In an attempt to determine the threshold value, input data to PROBIT have been prepared (Table 4.9) based on information contained in Table 4.8. However, the result (Table 4.10) fail to yield a meaningful interpretation -- the response decreases as the dose increases. The failure can be attributed to the small and skew set of data found in Table 4.9 -- the only 3 non-100% and non-0% response data sets are all of 50% response. While more data are required before an analytical threshold value can be identified, we define subjectively at this juncture that

> if I is the highest indentation level in the second version of a program,
>
> and changes are found in J indentation levels,
>
> and J/I > 1/2,
>
> then the program has gone through, positively or negatively, a change in terms of reconstruction.

The threshold value of 1/2 in this case has been determined based on a practitioner's experiences and heuristics.

4.2.6 Removing Documentation

Nine of the sixty-four sets of programs analyzed in the current research have been observed to have a decrease

in the comment statements. Table 4.11 summarizes the change in the number of comment statements in these 9 sets of programs. Notice in the Table that 4 of the 9 sets of programs experienced removing documentation with a decreasing in the lines of code; the remaining sets of programs experienced removing documentation with an increase in the lines of code. The correlation coefficient between the size of the program and the number of comments deleted is -0.097, indicating strongly that the two quantities do not correlate. The result of PROBIT analysis (Table 4.12) reveals that

> if two successive versions of a program exhibit a decrease of more than 2 (8) lines of COMMENT statements,
>
> then there is a 50 (90) percent chance that the program has experienced a change in terms of removing documentation.

4.2.7 Removing Functionality

In the current research, 16 of the 64 sets of programs appear to possess this pattern of changes. Table 4.13 gives a detailed description of the change of these 16 sets of programs. It is interesting to note from the Table that FOR and ASSIGNMENT are changed most frequently.

A PROBIT analysis on the data derived from the Table yields that

> if two successive versions of a program exhibit a decrease of more than a total of 2 (27) lines of FUNCTION, DECLARATION, ASSIGNMENT, include-file (PREPROCESSOR) or control statements,
>
> then there is a 50 (90) percent chance that the program has experienced a change in terms of removing functionality.

The result of PROBIT analysis for this case is summarized in Table 4.14.

4.2.8 Adding Functionality

In the current research, 37 of the 64 sets of programs appear to possess this pattern of changes. Table 4.15 shows a detail description of the changes in these 37 sets of programs. A PROBIT analysis on the data derived from the Table yields that

> if two successive versions of a program exhibit addition of more than a total of 4 (18) lines of FUNCTION, DECLARATION, ASSIGNMENT, include-file (PREPROCESSOR) or control statements,

then there is a 50 (90) percent chance that the program has experienced a change in terms of adding functionality.

The result of PROBIT analysis for this case is summarized in Table 4.16.

4.2.9 Removing Debugging

Twenty-two of the sixty-four sets of programs in the current research have been observed to exhibit a decrease in the output statement. The decrease ranges from 1 to 25. A summary of the change in output statements for all the 22 sets of programs is given in Table 4.17. Observe that the removal of output statements can occur regardless of direction of change in the total lines of code.

Similar to the case of debugging, the size of the program is found to be independent of the number of statements deleted for removing debugging; the correlation coefficient between the two quantities is merely 0.4186. Using the PROBIT procedure, we conclude from the result (Table 4.18) that

if two successive versions of a program exhibit a decrease of more than 1 (10) OUTPUT statement,

then there is a 50 (90) percent chance that the program has experienced a change in terms of removing debugging.

4.2.10 Redistribution

In current research, 11 of the 64 sets of programs appear to possess this pattern of changes. Table 4.19 summarizes the activities of these 11 sets of programs in terms of redistribution. From the Table, we obtain two sets of data as described in Table 4.20. The result of PROBIT analysis of data set (a) is given in Table 4.21; it gives rise to the threshold values of using FUNCTION in a quantitative definition of the change pattern of redistribution. However, data set (b) is an invalid data set for the PROBIT procedure. This is because that the change patterns of the 11 sets of programs bear too much similarity among them -- 9 of them all have one preprocessor removed. In the light of the partial information attained, we define semi-subjectively that

if the changes in PREPROCESSOR and FUNCTION are in opposite directions (the former increases while the latter decreases or the other round),

and two successive versions of a program exhibit more than 3 lines of change in PREPROCESSOR or 3 (45) lines of change in FUNCTION,

then there is a 50 (90) percent chance that the program has gone through, positively or negatively, a change in terms of redistribution.

As in the case of reconstruction, the threshold value of 3 for PREPROCESSOR is determined based on the experiences and heuristics of a practitioner.

4.3 INTUITIVE RULES

A set of intuitive rules is proposed to guide the use of the classification in change analysis of a program. The set of rules can be best understood by the graphic representation depicted in Figure 4.1. Literally, the Figure implies that in analyzing the change of a program, a software analyzer should abide by the following rules.

(1) Focus on the pattern change in pretty-printing. If the program is found to have experienced a significant change in terms of pretty-printing, then the analysis should be halted. Otherwise, proceed with the analysis.

(2) Check the pattern change in terms of

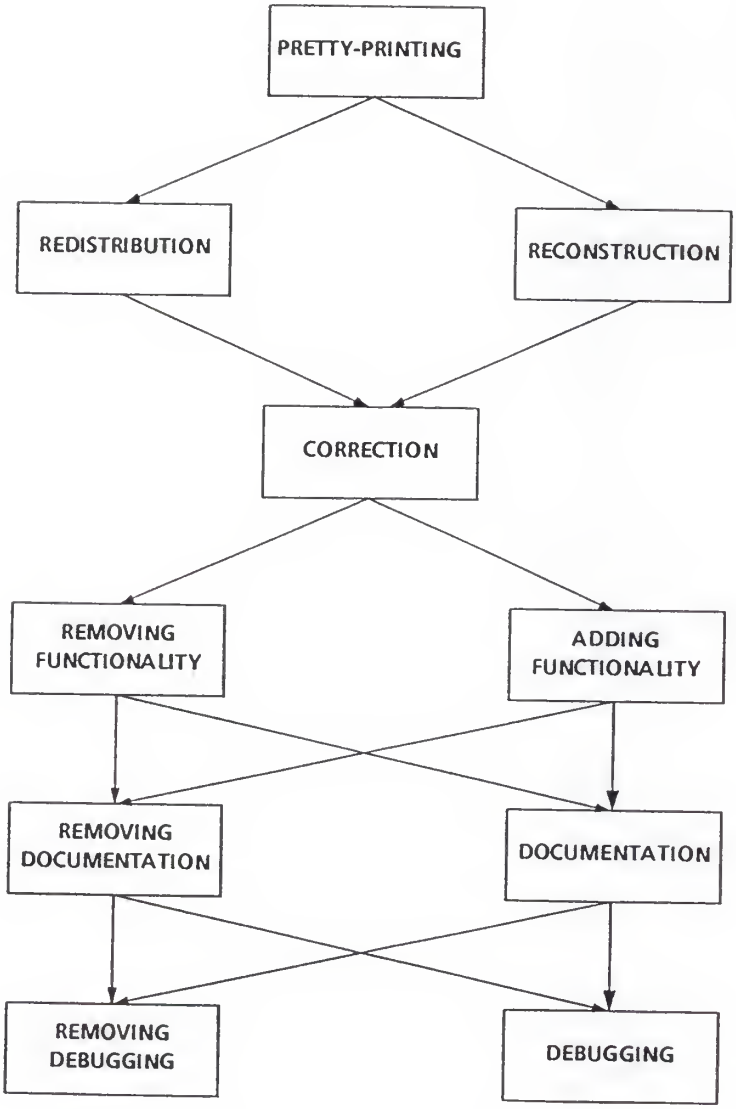


Figure 4.1 Graphic representation of the intuitive rules to guide the use of the classification in change analysis of a program.

redistribution and reconstruction simultaneously. If the program is found to not experience any significant change in terms of both types of classification, then proceed with the analysis. Otherwise, halt the analysis.

(3) Examine the pattern change in correction. If the program is found to have experienced a significant change in terms of correction, then the analysis should be halted. Otherwise, proceed with the analysis.

(4) Concentrate on the pattern change in terms of removing/adding functionality. If the program is found to not experience any significant change in terms of both types of classification, then proceed with the analysis. Otherwise, halt the analysis.

(5) Study the pattern change in terms of removing/adding documentation. If the program is found to not experience any significant change in terms of both types of classification, then proceed with the analysis. Otherwise, halt the analysis.

(6) Direct final attention to the pattern change in terms of removing/adding debugging.

The ordering of the rules is based on the previous research on the weight of a program, i.e., the frequencies of change of individual statement types [Gustafson1985]. An example is given to demonstrate an application of the intuitive rules in the following section.

4.4 EXAMPLE

Current example comprising two successive versions of the C module "Recreate_Listing" developed by students in a Software Engineering class; neither version is final. The module, upon completion, accepts data array records and counter arrays as inputs; it recreates and prints out a list containing the index values of individual entity names of the data arrays. The source codes of the two versions of the module are given separately in Appendices I and J; their sizes are 3824 and 6136 bytes, respectively.

As a preparation for change analysis, both versions of the module are used as inputs to the CHANGES program (described in Section 2.3) yielding an output file, *main.results*. Data contained in the file *main.results* is then processed by Excel on an Apple Macintosh to generate change patterns between the two successive versions of the module. Figures 4.2 to 4.4 are reproductions of those change patterns. In the following sub-section,

classification of these patterns are discussed. Changes analysis on the module will then be examined based on the classification and the intuitive rules.

4.4.1 Identifying Program Change Patterns

Figure 4.2 contrasts the changes in terms of the number of occurrence of each type of statement between the two versions of the module. The solid bars are for the first version and the empty ones for the second. The pattern of changes in this Figure indicates that redistribution, adding functionality and debugging have occurred when the module "Recreate_Listing" progressed between the two versions. First, the decrease in the number of PREPROCESSOR in conjunction with the increase in the number of FUNCTION indicate the existence of redistribution. Secondly, the increases in the numbers of IF, ASSIGNMENT and DECLARATION statements imply the presence of adding functionality. Finally, the increase in the number of OUTPUT statements suggests that debugging was involved as the development of the module progressed.

Figure 4.3 compares the indentation levels between the two versions of the module. Again, the solid bars are for the first version and the empty ones for the second. Note that the number of the ZERO indentation level in the second version is actually greater than that in the first

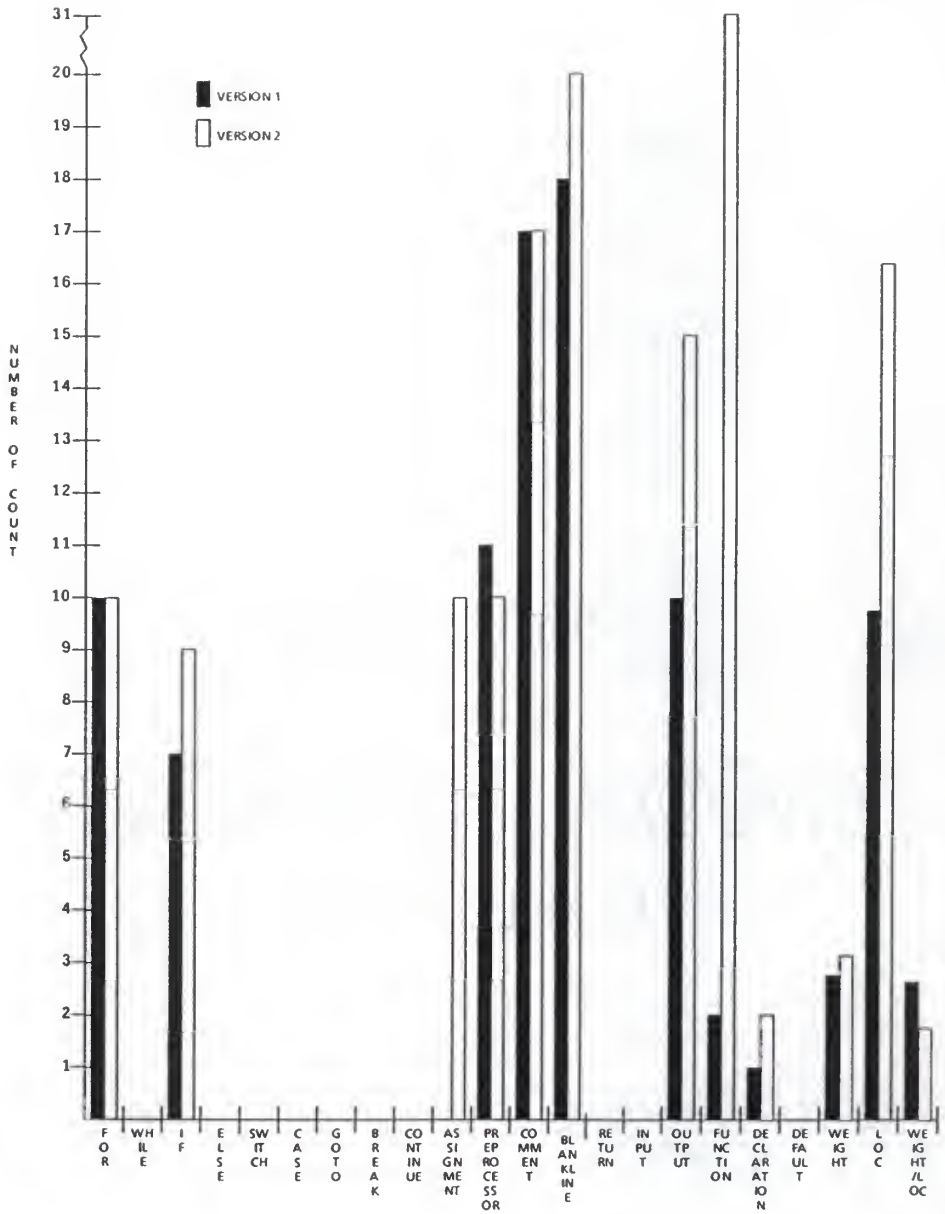


Figure 4.2 Program differences represented in a bar chart in terms of various statement types and lines of code.

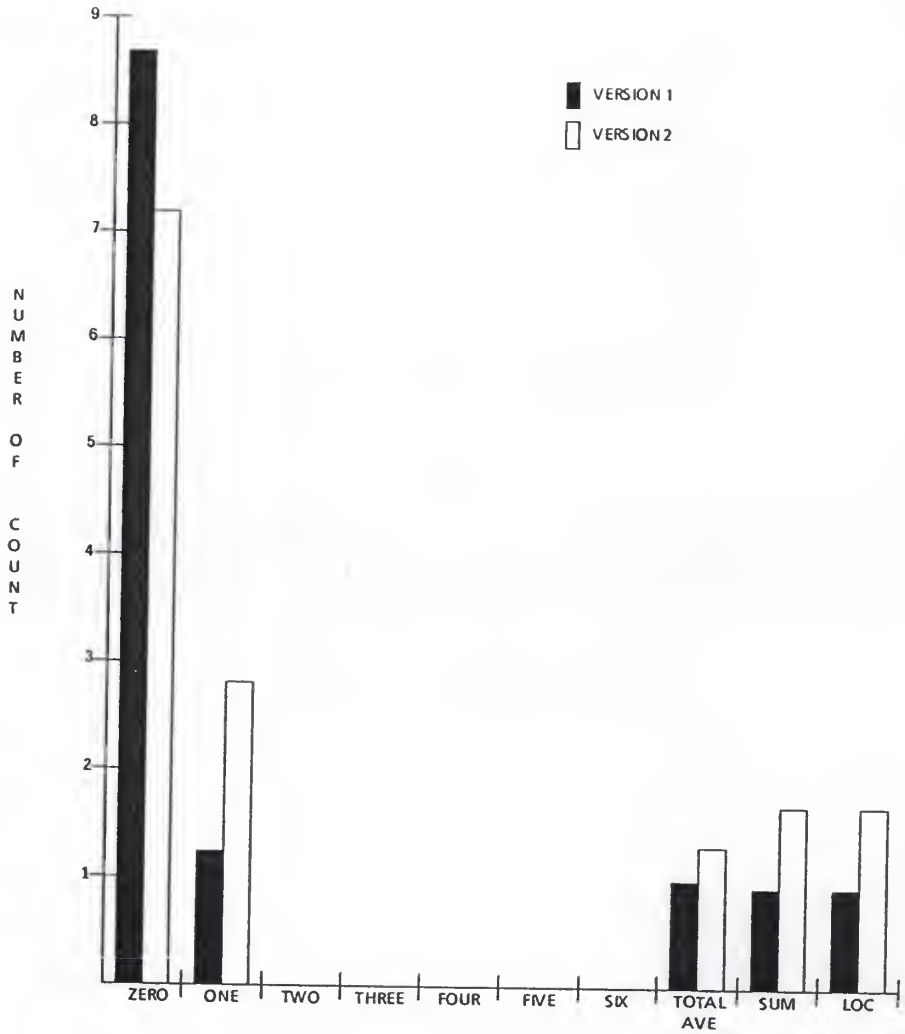


Figure 4.3 Program differences represented in a bar chart in terms of normalized indentation levels.

version, although the empty bar for the ZERO indentation level is shorter than the solid one in the Figure. This is due to the normalization scheme adopted (see Section 2.4). The consistent increases in the numbers of all indentation levels conclude that reconstruction did not happen in this case.

Figure 4.4 collates the differences between the two versions of the module with and without pretty-printing. The former are represented in solid bars and the latter in empty bars. In the Figure, more differences in the FOR and OUTPUT statements are detected between the two versions of the module when they are not pretty-printed. This hints of the occurrence of pretty-printing.

In summary, the change patterns exposited in Figures 4.2 to 4.4 lead us to conclude that redistribution, adding functionality, debugging and pretty-printing are major activities occur between the two versions. The quality of the program need be analyzed; this is discussed in the next sub-section.

4.4.2 Analyzing Program Changes

The significance of the change patterns identified in the preceeding sub-section can be appreciated when they are examined quantitatively and discussed in the context of the newly proposed intuitives rules. The explanation

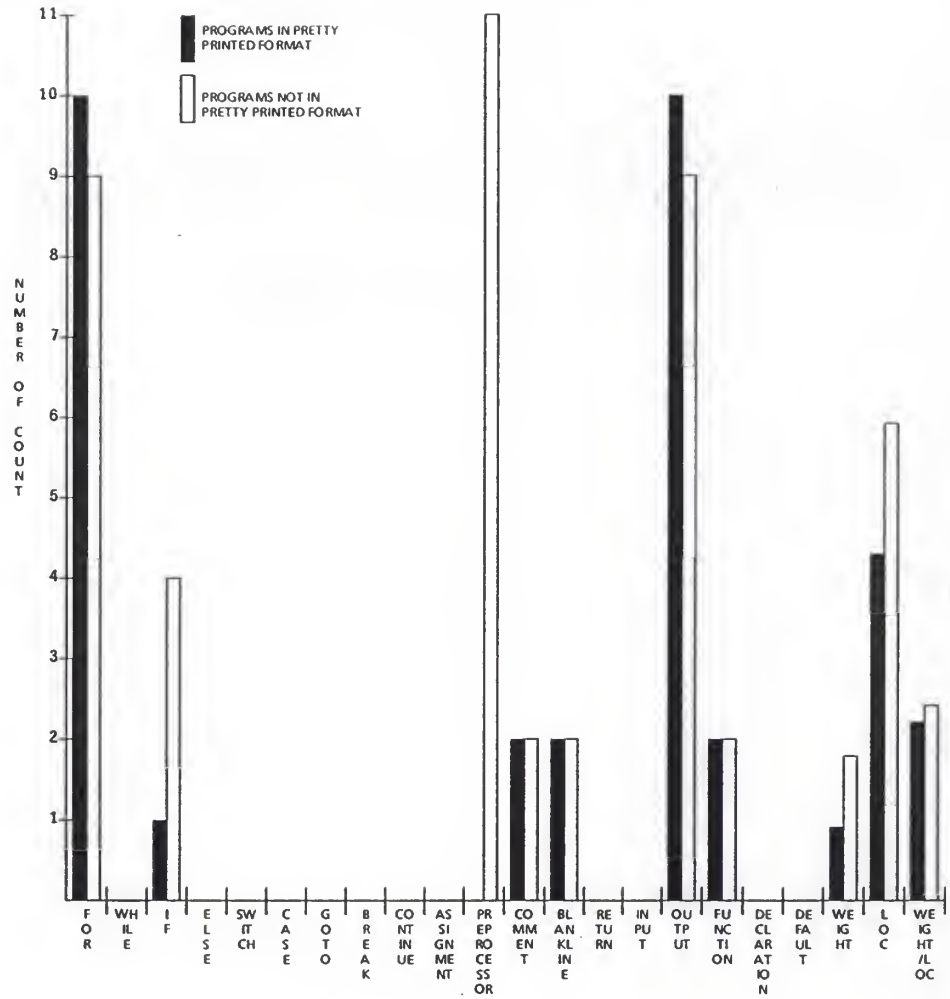


Figure 4.4 Program differences represented in a bar chart in terms of various statement types and lines of code; the programs were pretty-printed before comparison.

can be best understood with reference to Figure 4.1.

(1) The change pattern of pretty-printing is examined. From Figure 4.4, seven statements are changed between the two versions; two of them are identified to have gone through pretty-printing. The ratio of N/M is 2/7, indicating that there exists a 75% chance (see Table 4.6) for the program to have gone through a significant change in terms of pretty-printing. Considering the case when a software development should be interrupted only if we are 80% sure that the development is abnormal, we should proceed the analysis.

(2) The change pattern of redistribution is examined. From Figure 4.2, the number of PREPROCESSOR decreases by 1 while the number of FUNCTION increases by 29. Since more than 17 lines of change in FUNCTION are detected (see Table 4.19 for the significance of the threshold value of 17), we are at least 80% sure that a significant change with respect to redistribution exists. The quality of the software is in doubt; the analysis should be terminated.

Table 4.1 Increasing of OUTPUT statements between two versions of a program.

	VERSION 1		VERSION 2		Average size	Increasing of OUTPUT
	lines of code	no. of OUTPUT	lines of code	no. of OUTPUT		
1	448	9	455	18	451	9
2	412	10	455	28	433	18
3	62	0	64	2	63	2
4	64	2	72	5	68	3
5	103	11	107	19	105	8
6	15	0	39	2	27	2
7	424	13	453	16	438	3
8	438	3	453	4	445	1
9	148	0	241	1	194	1
10	377	1	447	20	412	19
11	94	2	118	5	106	3
12	128	2	138	10	133	8
13	413	0	1125	3	769	3
14	1125	3	1485	8	130	5
15	1485	8	2104	32	1794	24
16	413	0	2498	32	1455	32
17	34	7	50	16	42	9
18	112	8	47	10	79	2
19	34	7	68	18	51	11
20	51	1	53	3	52	2
21	69	3	104	10	86	7
22	47	10	68	18	57	8
23	82	1	94	2	88	1

CORRELATION COEFFICIENT = 0.7348

Table 4.2 Input data for PROBIT analysis of Debugging.

	DOSE	N	RESPONSE
1	32	1	1
2	24	1	1
3	19	1	1
4	18	1	1
5	11	1	0
6	9	2	1
7	8	3	2
8	7	1	1
9	5	1	0
10	3	4	2
11	2	4	1
12	1	3	0
13	0	41	0

Table 4.3 Increasing of COMMENT statements between two versions of a program.

	VERSION 1		VERSION 2		Average size	Increasing of COMMENT
	lines of code	no. of COMMENT	lines of code	no. of COMMENT		
1	431	6	438	41	434	35
2	438	41	427	45	432	4
3	431	6	427	45	429	39
4	433	3	437	46	435	43
5	437	46	458	189	447	143
6	413	6	458	189	435	183
7	448	5	455	46	451	41
8	412	5	455	46	433	41
9	431	46	452	184	441	138
10	64	22	72	25	68	3
11	107	18	107	26	107	8
12	106	24	113	32	109	8
13	55	12	79	48	67	36
14	55	11	79	48	67	37
15	15	13	59	24	37	11
16	453	7	438	41	445	34
17	438	41	453	48	445	7
18	453	48	474	191	463	143
19	424	7	474	191	449	184
20	148	26	241	38	194	8
21	241	38	377	55	309	17
22	377	55	447	66	412	11
23	447	66	424	67	435	1
24	424	67	424	68	424	1
25	424	68	472	123	448	55
26	141	29	184	49	162	20
27	47	0	68	2	57	2
28	413	45	1125	139	769	94
29	1125	139	1485	181	1305	42
30	1485	181	2104	275	1794	94
31	2104	275	2328	325	2216	50
32	413	45	2498	329	1455	284
33	51	1	112	29	81	28
34	51	0	53	1	52	1
35	53	1	69	16	61	15
36	104	15	120	39	112	24
37	120	39	129	42	124	3
38	51	0	129	42	90	42

CORRELATION COEFFICIENT = 0.4443

Table 4.4 Result of PROBIT for quantitative analysis of Documentation.

PROBIT ANALYSIS ON DOSE			
PROBABILITY	DOSE	95 PERCENT FIDUCIAL LIMITS	
		LOWER	UPPER
0.01	0.00620648	0.00000000	0.2
0.02	0.01127852	0.00000000	0.3
0.03	0.01647545	0.00000000	0.4
0.04	0.02191030	0.00000000	0.4
0.05	0.02762861	0.00000000	0.5
0.06	0.03365733	0.00000000	0.5
0.07	0.04001656	0.00000000	0.6
0.08	0.04672365	0.00000000	0.6
0.09	0.05379485	0.00000000	0.7
0.10	0.06124619	0.00000000	0.8
0.15	0.10479359	0.00000000	1.0
0.20	0.16059017	0.00000000	1.3
0.25	0.23161256	0.00000000	1.6
0.30	0.32180129	0.00000000	2.0
0.35	0.43645257	0.00000000	2.4
0.40	0.58280540	0.00000000	2.8
0.45	0.77095668	0.00000000	3.4
0.50	1.01533637	0.00000000	4.1
0.55	1.33718011	0.00000001	5.0
0.60	1.76887162	0.00000021	6.3
0.65	2.36201597	0.00000446	8.2
0.70	3.20355436	0.00010671	11.5
0.75	4.45100179	0.00285048	19.1
0.80	6.41949590	0.07394092	50.3
0.85	9.83750916	1.00026825	511.3
0.90	16.83219814	4.22058317	59435.9
0.91	19.16369066	5.15367069	217351.9
0.92	22.06394066	6.20764355	916930.7
0.93	25.76202996	7.41705783	4584563.9
0.94	30.62952678	8.83913182	28319660.3
0.95	37.31306265	10.57008970	230781234.6
0.96	47.05129507	12.78260853	2769176875.2
0.97	62.57236442	15.82471467	59949523007.6
0.98	91.40452554	20.55097808	3653992392227.4
0.99	166.10176166	30.07277334	2452809210542120.0

Table 4.5 Data reflectig the activity of Correction.

	FOR	WHILE	IF	ELSE	SWITCH	CASE	GOTO	BREAK	CONTINUE	ASSIGNMENT	RETURN	FUNCTION	DECLARATION	TOTAL
1		+1	+1	-1										3
2	-2		+1	+1										4
3	-1											-2	-1	4
4			-1	+2						-7		+3	+1	14
5				+3										3
6												+3	+2	5
7			+1											1
8										+4			+2	6
9			+1											2
10										-2		-1		3
11			-1										+1	2
12	-1											-2	-1	4
13												-2	-1	3
14			+1											1
15	+2									-1			-1	4
16			-1	-2							-1	-4		8

Table 4.6 Result of PROBIT for quantitative analysis of Correction.

PROBABILITY	PROBIT ANALYSIS ON DOSE DOSE	95 PERCENT FIDUCIAL LIMITS LOWER	UPPER
0.01	0.11150498	.	0.72822395
0.02	0.14676349	.	0.82456998
0.03	0.17471297	.	0.89289236
0.04	0.19919418	.	0.94845581
0.05	0.22161661	.	0.99655331
0.06	0.24267929	.	1.03971308
0.07	0.26278796	.	1.07936067
0.08	0.28220275	.	1.11638816
0.09	0.30110245	.	1.15139578
0.10	0.31961652	.	1.18480976
0.15	0.40918897	.	1.33686615
0.20	0.49796445	.	1.47672275
0.25	0.58932682	.	1.61418733
0.30	0.68557383	.	1.75578850
0.35	0.78873887	.	1.90784821
0.40	0.90095162	.	2.07858204
0.45	1.02469095	.	2.28140763
0.50	1.16304948	.	2.54354863
0.55	1.32008982	.	2.93984267
0.60	1.50139481	.	3.94117784
0.65	1.71499611	.	.
0.70	1.97306844	.	.
0.75	2.29530379	.	.
0.80	2.71642701	.	.
0.85	3.30576874	.	.
0.90	4.23220962	.	.
0.91	4.49243800	1.74922256	.
0.92	4.79330578	2.07677798	.
0.93	5.14743551	2.32312810	.
0.94	5.57395772	2.54892358	.
0.95	6.10371266	2.77624313	.
0.96	6.79078119	3.02257870	.
0.97	7.74232225	3.31128553	.
0.98	9.21676148	3.68865254	.
0.99	12.13115403	4.29788066	.

Table 4.7 Result of PROBIT for quantitative analysis of Pretty-printing.

PROBIT ANALYSIS ON DOSE			
PROBABILITY	DOSE	95 PERCENT FIDUCIAL LIMITS	
		LOWER	UPPER
0.01	0.00359507	.	0.06292549
0.02	0.00534399	.	0.07275113
0.03	0.00687219	.	0.07986129
0.04	0.00830356	.	0.08573071
0.05	0.00968507	.	0.09087535
0.06	0.01104060	.	0.09554341
0.07	0.01238436	.	0.09987576
0.08	0.01372576	.	0.10396120
0.09	0.01507146	.	0.10786001
0.10	0.01642650	.	0.11161548
0.15	0.02346113	.	0.12917096
0.20	0.03114454	.	0.14613311
0.25	0.03971307	.	0.16384922
0.30	0.04939942	.	0.18370446
0.35	0.06047244	.	0.20806020
0.40	0.07326641	.	0.24337088
0.45	0.08821543	.	.
0.50	0.10590236	.	.
0.55	0.12713546	.	.
0.60	0.15307572	.	.
0.65	0.18546148	.	.
0.70	0.22703320	.	.
0.75	0.28240854	.	.
0.80	0.36010509	.	.
0.85	0.47803794	.	.
0.90	0.68275689	0.18320956	.
0.91	0.74414212	0.22122090	.
0.92	0.81709926	0.24590687	.
0.93	0.90560228	0.26828284	.
0.94	1.01582463	0.29072952	.
0.95	1.15799995	0.31478702	.
0.96	1.35066218	0.34217866	.
0.97	1.63198366	0.37566671	.
0.98	2.09867538	0.42117163	.
0.99	3.11963954	0.49765526	.

Table 4.8 Data relecting the activity of Reconstruction.

level	0	1	2	3	4	5	6
1				x	x	x	x -
2	x	x	x	x	x	x	x -
3	x	x	x	x	x	x	-
4	x	x	x	x	x		x -
5	x	x	x	x			-
6	x						-
7	x	x	x	x	x		x -
8		x	x	x			-
9		x					-
10						x	-
11						x	-
12	x	x	x		-		
13	x			-			
14		x		x	x	x	-
15		x		x		x	x -
16				x	x		-
17	x					-	
18	x					x -	
19			x				x -
20	x	x					x -
21	x						-
22	x	x			-		
23	x				-		
24	x				-		
25					x	-	

x indicates the activity of reconstruction happening in that level.
- indicates the highest indentation level in the second version of the program.

Table 4.9 Input data for PROBIT analysis of Reconstruction.

	DOSE	N	RESPONSE	
1	1/4	1	0	
2	1/5	2	2	
3	1/6	2	1	*
4	1/7	4	4	
5	2/5	1	1	
6	1/3	1	1	
7	2/7	3	3	
8	3/4	1	0	
9	3/7	2	2	
10	4/7	4	2	*
11	6/7	2	2	
12	7/7	2	1	*
13	0	39	0	

* Only 3 sets of data are of neither 100% nor 0% response; moreover, all three sets of data are of 50% response.

Table 4.10 Result of PROBIT for quantitative analysis of Reconstruction.

PROBIT ANALYSIS ON DOSE			
PROBABILITY	DOSE	95 PERCENT FIDUCIAL LIMITS	
		LOWER	UPPER
0.01	107.02127855	3.19345977	.
0.02	64.43615097	2.61281373	.
0.03	46.70147304	2.29890731	.
0.04	36.65770134	2.08698947	.
0.05	30.10380060	1.92848786	.
0.06	25.45716481	1.80263013	.
0.07	21.97701845	1.69867638	.
0.08	19.26666462	1.61037745	.
0.09	17.09305821	1.53378888	.
0.10	15.30967137	1.46626766	.
0.15	9.70147845	1.21437523	.
0.20	6.75106227	1.04203536	.
0.25	4.94630019	0.91069888	.
0.30	3.74079577	0.80373410	.
0.35	2.88766100	0.71229031	.
0.40	2.25877368	0.63080071	.
0.45	1.78100395	0.55498907	.
0.50	1.40959181	0.48032362	.
0.55	1.11563429	0.39857046	.
0.60	0.87965832	.	.
0.65	0.68808252	.	.
0.70	0.53115679	.	.
0.75	0.40170410	.	.
0.80	0.29431651	.	.
0.85	0.20480889	.	.
0.90	0.12978391	.	0.36582117
0.91	0.11624304	.	0.33588262
0.92	0.10312886	.	0.30979057
0.93	0.09041031	.	0.28597182
0.94	0.07805068	.	0.26342504
0.95	0.06600326	.	0.24137926
0.96	0.05420277	.	0.21909146
0.97	0.04254575	.	0.19562651
0.98	0.03083594	.	0.16938581
0.99	0.01856593	.	0.13625452

Table 4.11 Decreasing of COMMENT statements between two versions of a program.

	VERSION 1		VERSION 2		Average size	Decreasing of COMMENT
	lines of code	no. of COMMENT	lines of code	no. of COMMENT		
1	413	6	433	3	423	3
2	107	26	106	24	106	2
3	56	12	55	11	55	1
4	140	31	128	29	134	2
5	34	5	50	1	42	4
6	112	29	47	0	79	29
7	34	5	68	2	51	3
8	69	16	104	15	86	1
9	3	3	82	2	82	1

CORRELATION COEFFICIENT = -0.097

Table 4.12 Result of PROBIT for quantitative analysis of Removing documentation.

PROBABILITY	PROBIT ANALYSIS ON DOSE DOSE	95 PERCENT FIDUCIAL LIMITS LOWER	UPPER
0.01	0.13030754	.	0.80370316
0.02	0.17775860	.	0.91427178
0.03	0.21646897	.	0.99479347
0.04	0.25105154	.	1.06197710
0.05	0.28321788	.	1.12166640
0.06	0.31382343	.	1.17669897
0.07	0.34336805	.	1.22872196
0.08	0.37217437	.	1.27881655
0.09	0.40046572	.	1.32776572
0.10	0.42840516	.	1.37618831
0.15	0.56639627	.	1.62847301
0.20	0.70712877	.	1.96506841
0.25	0.85542583	.	.
0.30	1.01492749	.	.
0.35	1.18916239	.	.
0.40	1.38207456	.	.
0.45	1.59845569	.	.
0.50	1.84445561	.	.
0.55	2.12831455	.	.
0.60	2.46152892	.	.
0.65	2.86085107	.	.
0.70	3.35197985	.	.
0.75	3.97698593	.	.
0.80	4.81102824	1.70248914	.
0.85	6.00642465	2.12566036	.
0.90	7.94111938	2.54177023	.
0.91	8.49515025	2.63795270	.
0.92	9.14092096	2.74215642	.
0.93	9.90778395	2.85698878	.
0.94	10.84054333	2.98619432	.
0.95	12.01201172	3.13551855	.
0.96	13.55106821	3.31454916	.
0.97	15.71595481	3.54127628	.
0.98	19.13840772	3.85630633	.
0.99	26.10759590	4.39075496	.

Table 4.13 Data reflecting the activity of Removing functionality.

	FOR	WHILE	IF	ELSE	SWITCH	CASE	GOTO	BREAK	CONTINUE	ASSIGNMENT	PREPROCESSOR	RETURN	FUNCTION	DECLARATION	TOTAL
1	2	1								9				1	13
2	2														2
3		1	1							10					12
4	2														2
5	2										1		2		5
6	3			1										2	6
7										1					1
8										2			1		3
9	1	1									1		1		2
10			8	8						4					20
11	2										4		1	7	14
12	1														1
13											1				1
14										1					1
15											1			1	2
16	1											1	1	2	5

Table 4.14 Result of PROBIT for quantitative analysis of Removing functionality.

PROBIT ANALYSIS ON DOSE

PROBABILITY	DOSE	95 PERCENT FIDUCIAL LIMITS LOWER	UPPER
0.01	0.02395598	.	0.40045266
0.02	0.04071631	.	0.51378816
0.03	0.05700605	.	0.60324999
0.04	0.07342869	.	0.68177921
0.05	0.09021888	.	0.75409786
0.06	0.10750231	.	0.82255650
0.07	0.12536044	.	0.88853750
0.08	0.14385356	.	0.95294891
0.09	0.16303094	.	1.01643884
0.10	0.18293583	.	1.07950239
0.15	0.29473461	.	1.40084269
0.20	0.43057952	.	1.75940050
0.25	0.59605259	.	2.20105839
0.30	0.79820524	.	2.82142270
0.35	1.04626445	.	3.94918746
0.40	1.35258011	.	.
0.45	1.73404900	.	.
0.50	2.21436417	.	.
0.55	2.82772210	.	.
0.60	3.62522608	.	.
0.65	4.68658635	.	.
0.70	6.14304244	.	.
0.75	8.22646987	2.24033975	.
0.80	11.38792836	3.19656253	.
0.85	16.63669120	4.26226337	.
0.90	26.80398156	5.72511159	.
0.91	30.07655233	6.11252324	.
0.92	34.08611333	6.55190307	.
0.93	39.11448321	7.05932424	.
0.94	45.61212236	7.65889260	.
0.95	54.35013947	8.38906047	.
0.96	66.77783194	9.31649800	.
0.97	86.01558798	10.57161076	.
0.98	120.42861238	12.46387882	.
0.99	204.68411661	16.06642642	.

Table 4.15 Data reflecting the activity of Adding functionality.

	FOR	WHILE	IF	ELSE	SWITCH	CASE	GOTO	BREAK	CONTINUE	ASSIGNMENT	PREPROCESSOR	RETURN	FUNCTION	DECLARATION	TOTAL
1										14				5	19
2				2									4		6
3										5				4	9
4		1	3	1						17				3	25
5	1			2									4	3	10
6			2	3						7				5	17
7			3	1						15				4	23
8										7				5	12
9											1				1
10	1		3	2						5	1		5	3	20
11	3		4							1			6		14
12													3	4	7
13	1		3	2						5	1		5	3	20
14		1	3	1						15				4	24
15	1			2									7	2	12
16		1	2	3						8				6	20
17	4	2	9	2				2		26			18	13	76
18	6		10	1				4		17				7	45
19	4		7	1						6			14	5	37
20			6					1		5					12
21			4	4											8
22	3		3							1			8		15
23										4					4
24	4	14	21	7	3					79			47	37	12
25	3	12	15	5						39			50	9	33
26	2														2
27			8	8						4					20
28	3	1									5		3	10	22
29			4	4						6		1			15
30			8	8						6				1	23
31		2		1						9	2		2	1	17
32	1		2	2						3			4		12
33		1								10			1		12
34	2	3	2	3						22	1		7	1	41
35	18	34	48	19	3					164			164	71	521
36			1											2	3
37			2										4		6

Table 4.16 Result of PROBIT for quantitative analysis of Adding functionality.

PROBIT ANALYSIS ON DOSE			
PROBABILITY	DOSE	95 PERCENT FIDUCIAL LIMITS	
		LOWER	UPPER
0.01	0.26808408	0.00000001	1.47812955
0.02	0.36936075	0.00000004	1.77241290
0.03	0.45264301	0.00000013	1.99015161
0.04	0.52745363	0.00000031	2.17232077
0.05	0.59733604	0.00000063	2.33345832
0.06	0.66406487	0.00000117	2.48061574
0.07	0.72867910	0.00000201	2.61781631
0.08	0.79185061	0.00000326	2.74760260
0.09	0.85404564	0.00000507	2.87170005
0.10	0.91560541	0.00000759	2.99134348
0.15	1.22139833	0.00004039	3.54881858
0.20	1.53575437	0.00015210	4.07659075
0.25	1.86918974	0.00047309	4.60437314
0.30	2.22989200	0.00130673	5.15190852
0.35	2.62599173	0.00333845	5.73730363
0.40	3.06672228	0.00809516	6.38150063
0.45	3.56341526	0.01896681	7.11295625
0.50	4.13071118	0.04350960	7.97527760
0.55	4.78832064	0.09869916	9.04281925
0.60	5.56384743	0.22288519	10.45760469
0.65	6.49764989	0.50170866	12.53017326
0.70	7.65183912	1.11350679	16.06254945
0.75	9.12843382	2.34594831	23.56311870
0.80	11.11035412	4.37163329	44.42163783
0.85	13.96986914	6.93743229	121.08893896
0.90	18.63551121	9.92407412	534.47594844
0.91	19.97876234	10.62048345	779.42474703
0.92	21.54797205	11.37624164	1180.08747554
0.93	23.41603453	12.21193568	1870.80595601
0.94	25.69443971	13.15786387	3144.17898370
0.95	28.56478380	14.26138255	5710.10150025
0.96	32.34933628	15.60288038	11564.99302784
0.97	37.69587611	17.33606657	27682.34437384
0.98	46.19541944	19.81618926	88886.38417365
0.99	63.64710283	24.23088995	564320.95848118

Table 4.17 Decreasing of OUTPUT statements between two versions of a program.

	VERSION 1		VERSION 2		Average size	Decreasing of OUTPUT
	lines of code	no. of OUTPUT	lines of code	no. of OUTPUT		
1	432	25	438	3	435	22
2	438	3	427	0	432	3
3	431	25	427	0	429	25
4	433	11	437	1	435	10
5	437	1	458	0	447	1
6	413	11	458	0	435	11
7	412	10	450	9	431	1
8	72	5	67	2	69	3
9	67	2	65	0	66	2
10	106	19	113	0	109	19
11	82	2	82	1	82	1
12	59	2	57	0	58	2
13	453	16	438	3	448	13
14	453	4	474	3	463	1
15	424	13	474	3	449	10
16	447	20	424	1	435	19
17	140	4	128	2	134	2
18	141	10	184	0	162	10
19	118	5	114	2	116	3
20	51	16.	112	8	81	8
21	120	10	129	0	124	10
22	51	1	129	0	90	1

CORRELATION COEFFICIENT =0.4186

Table 4.18 Result of PROBIT for quantitative analysis of Removing debugging.

PROBIT ANALYSIS ON DOSE

PROBABILITY	DOSE	95 PERCENT FIDUCIAL LIMITS LOWER	UPPER
0.01	0.00987305	.	0.24106379
0.02	0.01665264	.	0.31012384
0.03	0.02320211	.	0.36431487
0.04	0.02977738	.	0.41155477
0.05	0.03647776	.	0.45472663
0.06	0.04335614	.	0.49525537
0.07	0.05044645	.	0.53396770
0.08	0.05777351	.	0.57139492
0.09	0.06535732	.	0.60790426
0.10	0.07321522	.	0.64376403
0.15	0.11715101	.	0.81925989
0.20	0.17021341	.	0.99807044
0.25	0.23452432	.	1.18948229
0.30	0.31274365	.	1.40222998
0.35	0.40833837	.	1.64746143
0.40	0.52593629	.	1.94230557
0.45	0.67185443	.	2.31725434
0.50	0.85493131	.	2.83633204
0.55	1.08789571	.	3.67134794
0.60	1.38972642	.	5.58354746
0.65	1.78995562	.	.
0.70	2.33708203	.	.
0.75	3.11655334	.	.
0.80	4.29406555	.	.
0.85	6.23902032	1.36525446	.
0.90	9.98300049	3.01750086	.
0.91	11.18325487	3.36149229	.
0.92	12.65125828	3.73474799	.
0.93	14.48878139	4.15060798	.
0.94	16.85822524	4.62788362	.
0.95	20.03707294	5.19562959	.
0.96	24.54573128	5.90343461	.
0.97	31.50177217	6.84766259	.
0.98	43.89139787	8.25673647	.
0.99	74.03059791	10.92480081	.

Table 4.19 Data reflecting the activity of Redistribution.

	PREPROCESSOR	FUNCTION
1	-1	+2
2	-1	+6
3	-1	+7
4	-1	+9
5	-1	+7
6	-1	+7
7	-1	+3
8	-2	+16
9	+3	-1
10	-1	+1
11	-1	+4

Table 4.20 Input data for PROBIT analysis of Redistribution.

(a)
FUNCTION

dose*	N	response
15	1	1
9	1	1
7	3	1
5	1	1
3	2	1
1	3	1
0	53	0

(b)
PREPROCESSOR

dose**	N	response
3	1	1
2	1	1
1	9	4
0	53	0

* Only odd levels of dose are adopted here. The dose of level one designates that one to two FUNCTIONs have been changed between versions, the dose of level three designates that three to four FUNCTIONs have been change, etc.

** The negative signs are dropped from Table 4.19. The dose of level three designates that three PREPROCESSORs have been changed; the change can be in the positive or the negative direction.

Table 4.21 Result of PROBIT for quantitative analysis of Redistribution.

PROBIT ANALYSIS ON DOSE			
PROBABILITY	DOSE	95 PERCENT FIDUCIAL LIMITS	
		LOWER	UPPER
0.01	0.01972508	.	0.67262515
0.02	0.03537278	.	0.84156771
0.03	0.05123919	.	0.97293020
0.04	0.06771205	.	1.08724836
0.05	0.08494580	.	1.19196457
0.06	0.10302922	.	1.29077852
0.07	0.12202613	.	1.38587420
0.08	0.14198960	.	1.47870054
0.09	0.16296802	.	1.57031014
0.10	0.18500803	.	1.66152904
0.15	0.31280264	.	2.13434947
0.20	0.47483315	.	2.69705785
0.25	0.67928979	.	3.52061375
0.30	0.93693942	.	.
0.35	1.26218787	.	.
0.40	1.67464902	.	.
0.45	2.20157775	.	.
0.50	2.88177827	.	.
0.55	3.77213387	.	.
0.60	4.95903671	.	.
0.65	6.57956410	.	.
0.70	8.86358907	.	.
0.75	12.22548334	.	.
0.80	17.48960860	3.35368468	.
0.85	26.54915551	4.77313522	.
0.90	44.88802984	6.41999307	.
0.91	50.95874648	6.83526977	.
0.92	58.48770418	7.29963746	.
0.93	68.05629157	7.82864452	.
0.94	80.60476406	8.44532285	.
0.95	97.76405682	9.18603696	.
0.96	122.64650271	10.11324035	.
0.97	162.07604704	11.34788431	.
0.98	234.77502165	13.17366686	.
0.99	421.01958619	16.55821929	.

CHAPTER FIVE

CONCLUSIONS AND EXTENSIONS

A classification of program change patterns and a set of intuitive rules for effective evaluation of the program change patterns have been proposed. First, data concerning the pattern change between two successive versions of a program are identified and collected. Based on these data, certain criteria are derived to classify the change patterns. This has given rise to the ten classes of program change patterns. Further quantitative study on the classification yields the set of intuitive rules. The classification and the rules have been demonstrated to be capable of facilitating the program change analysis, enhancing the reliability of program change analysis and rendering the progress of software development more assessable. In summary, the proposed technique can help a software manager analyze the progress of a program during software development stage.

Extensions of the current research may include:

(1) Collecting and analyzing more change patterns between program sets. More data will lead to better statistical results. This is especially needed in the cases of RECONSTRUCTION and REDISTRIBUTION.

(2) Refining the definition of the change pattern Debugging. A statement of the type output statements (defined in Section 3.3.1) may be added for the purposes other than debugging. The content of the output statement need be taken into consideration in determining if it is for debugging.

(3) Identifying the average time required for a program to progress from one version to the next. It appears that the time may be dependent on the types of the change patterns involved.

(4) Extending the analysis to predict the progress of a program during its development. While this work has been aimed at determining the quality of a software development at its present state based on the historical data, attempts can be made to extend the analysis to predict the quality of the program when continuing development is compulsory.

LITERATURE CITED

[Berry1985] Berry, R. E. and B. A. E. Meekings, A Style Analysis of C Programs, Communications of the ACM, Vol. 28, No. 1, Jan. 1985, pp. 80-88.

[Bourne1987] Bourne, Stephen R., The Unix System V Environment, Addison-Wesley Publishing Company, 1987.

[Dunsmore1977] Dunsmore, Hubert E. and John D. Gannon, Experiemntal Investigation of Programming Complexity, 16th Annual Technical Symposium, June 2, 1977, pp. 117-125.

[Finney1971] Finney, D. J., Statistical Methods in Biological Assay, Second Edition, London, Griffin Press, 1971.

[Gustafson1985] Gustafson, David A., Austin C. Melton and Chyuan Samuel Hsieh, An Analysis of Software Changes During Maintenance and Enhancement, Proceedings of Conference on Software Maintenance, Nov. 11-13, 1985, pp. 92-95.

[Halstead1977] Halstead, M., Elements of Software Science, North Holland, 1977.

[Kyung1987] Kyung, Hee An, David A. Gustafson and Austin C. Melton, A Model for Software Maintenance, Proceedings of Conference on Software Maintenance, Sept. 21-24, 1987, pp. 57-62.

[Lanchbury1986] Lanchbury, Mary Lou A., A Model of Successful Patterns of Progress During the Integration of Software, Master Thesis, Department of Computing and Information Sciences, Kansas Statement University, 1986.

[McCabe1976] McCabe, Thomas J., A Complexity Measure, IEEE Transactions on Software Engineering, Vol. Se-2, No. 4, Dec. 1976, pp. 308-320.

[Pressman1982] Pressman, Roger S., Software Engineering: a Practitioner's Approach, McGraw-Hill Book Company, 1982.

[SAS1982] SAS User's Guide: Statistics, SAS Institute Inc., 1982, pp. 287-292.

[Townsend1985] Townsend, Carl, Mastering EXCEL, SYBEX, Berkeley, CA, 1985.

[Weiss1985] Weiss, David M. and Victor R. Basili, Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory, IEEE Transactions on Software Engineering, Vol. SE-11, No. 2, Feb. 1985, pp. 157-168.

APPENDIX A

SOURCE CODE OF SAMPLEPROGRAM1

```
/*****************************************************************************/
/*                                                                           */
/*                Procedure  : Syntax_Check        Last Revision : 4/30/86   */
/*                                                                           */
/*                Programmer  : Monte L. Hall                                */
/*                                                                           */
/*     Description  : This module accepts an EntityName which consists of all */
/*          those characters found after the colon on one line as distinguished */
/*          in the Read_Data module.  This EntityName string is tested for a null */
/*          string, an oversized string(one that is over 15 characters long), and */
/*          for embedded blanks within the EntityName.  A ConditionCode is passed */
/*          out indicating which error occurred; or if no error occurred, a */
/*          ConditionCode is passed out indicating such.                     */
/*                                                                           */
/*****************************************************************************/


#include   </usrb/cs340/ldb/project/define.h>
#include   <ctype.h>
#define    NoEntityName_Code 2
#define    TooLongEntityName_Code 1
#define    BlanksInEntityName_Code 8
#define    OK_Code 0

Syntax_Check(EntityName,ConditionCode)

char       *EntityName;
int        *ConditionCode;

{

       char *ch;
       int  j, i = 0, I = 0, Ch_Count = 0,
       Space = 0, Flag = 0, Error = 0;
       char  Temp[MAX_STR_LEN];

       ch = EntityName;

       printf("\nIn Syntaxcheck");

       printf("\nBefore isspace and ch is: %s",ch[I]);

       while (isspace(ch[I]) !=
              I = I + 1;

       j = I;
       printf("\nAfter isspace and ch is: %s",ch[I]);
       while (ch[I] != '\0' && ch[I] != '\n' && ch[I] != '\t' && Space == 0)
```

```
{
        if (ch[I] != ' ' && Spacc == 0)
        {
                Ch_Count = Ch_Count + 1;
                Temp[i] = ch[I];
                i = i + 1;
                I = I + 1;
                Flag = 1;
                printf("\nReading characters");
        }
        else
        {
                printf("\nRcad a space after characters");
                if (Flag == 1)
                        Space = 1;

                I = I + 1;
        }
}

while (iscntrl(ch[I]) != 0 && ch[I] != '\n'
&& ch[I] != '\t' && ch[I] != '\0' && Error == 0)
{
        if (ch[I] == ' ')
        {
                printf("\nReading spaces after all characters");
                I = I + 1;
        }
        else
        {
                printf("\nError occurred");
                Error = 1;
        }
}

if (Error == 1)
        *ConditionCode = BlanksInEntityName_Code;
else
        if (Ch_Count == 0)
                *ConditionCode = NoEntityName_Codc;
        else
                if (Ch_Count > MAX_STR_LEN - 1)
                        *ConditionCode = TooLongEntityName_Codc;
                else
                        *ConditionCode = OK_Code;
        printf("\nConditionCode in SC = %d",*ConditionCode);
        for (i=0; iAX_STR_LEN - 1; + +i)
        {
                EntityNamc[i] = ch[j];
                j = j + 1;
        }
```

```
        EntityName[MAX_STR_LEN - 1] = '\0';
        printf("\nENTITY NAME IS: %s",EntityName);
        printf("\nLeaving SyntaxCheck");
}
```

APPENDIX B

SOURCE CODE OF SAMPLEPROGRAM2

```
/*****************************************************************************/
/*                                                                           */
/*          Procedure  : Syntax_Check          Last Revision : 5/05/86       */
/*                                                                           */
/*          Programmer : Monte L. Hall                                       */
/*                                                                           */
/*   Description : This module accepts an EntityName which consists of all  */
/*     those characters found after the colon on one line as distinguished   */
/*     in the Read_Data module. This EntityName string is tested for a null  */
/*     string, an oversized string(one that is over MAX_STR_LEN characters   */
/*     long), and for embedded blanks within the EntityName. A Condition-    */
/*   Code is passed out indicating which error occurred; or if no error      */
/*     occurred, a ConditionCode is passed out indicating such.              */
/*                                                                           */
/*****************************************************************************/


#include   </usrb/cs340/ldb/project/define.h>
#include   <ctype.h>
#define    NoEntityName_Code 2
#define    TooLongEntityName_Code 1
#define    BlanksInEntityName_Code 8
#define    OK_Code 0

Syntax_Check(EntityName,ConditionCode)

char       *EntityName;
int        *ConditionCode;

{

        char *ch;
        int  i = 0, I = 0, Ch_Count = 0,     /* initialize indices, counters */
        Space = 0, Flag = 0, Error = 0,  /* and booleans */
        Null = 0;
        char Temp[MAX_STR_LEN];

        ch = EntityName;

        printf("\nIn Syntaxcheck");
        printf("\nENTITY NAME is: %s",EntityName);

        for (i=0; i<=MAX_STR_LEN; ++i)
                Temp[i] = ' ';

        i = 0;
        printf("\nBefore isspace and ch is: %s",ch[I]);
```

```
while (isspace(ch[I]) ! = 0)  /* strip off initial blanks in EntityName */
        I = I + 1;

if (iscntrl(ch[I]) = = 0 && (ch[I] ! = '\n'))
{

        printf("\nAfter isspace and ch is: %s",ch[I]);

        while (ch[I] ! = '\0' && ch[I] ! = '\n' && ch[I[ ! = '\t' && Space = = 0
        && (iscntrl(ch[i] = = 0))
        {       /* continue until eol or space encountered */

                if (ch[I] ! = ' ' && Space = = 0)
                {
                        Ch_Count = Ch_Count + 1;
                        Temp[i] = ch[I]    /* store EntityName w/o initial blanks */
                        i = i + 1;
                        I = I + 1;
                        Flag = 1;          /* reading characters */
                        printf("\nReading characters");
                } /* end inner if */
                else
                { /* read a space */
                        printf("\nRead a space after characters");
                        if (Flag = = 1)  /* read a space after characters? */
                                Space = 1;   /* turn flag on */

                        I = I + 1;
                }  /* end else */
        } /* end while */

        while (iscntrl(ch[I]) = = 0  && ch[I] ! = '\n'
        && ch[I] ! = '\t' && ch[I] ! = '\0' && Error = = 0)
        {    /* continue until eol or error occurs */

                if (ch[I] = = ' ')  /* read a space after characters */
                        I = I + 1;
                else   /* blanks between characters */
                Error = 1;  /* set flag */

        } /* end while */
} /* end outer if */
else
        Null = 1;

/* set ConditionCode */

if (Error = = 1)
        *ConditionCode = BlanksInEntityName_Code;
else
```

```
        if (Ch_Count = = 0)
                *ConditionCode = NoEntityName_Code;
        else
                if (Ch_Count  MAX_STR_LEN - 1)
                        *ConditionCode = TooLongEntityName_Code;
                else
                        *ConditionCode = OK_Code;
        printf("\nConditionCode in SC = %d",*ConditionCode);
        printf("\nTEMP IS: %s",Temp);
        if (Null = = 0)
{
                for (i=0; iAX_STR_LEN - 1; + +i)  /* copy entity name w/o initial */
                        EntityName[i] = Temp[i];       /* blanks in Temp back into */

                /* EntityName */

                EntityName[MAX_STR_LEN - 1] = '\0';
        }  /* end if */
        else
                EntityName[0] = '\0';
        printf("\nENTITY NAME after loop IS: %s",EntityName);
        printf("\nLeaving SyntaxCheck");
} /* end SyntaxCheck module */
```

APPENDIX C

SOURCE CODE OF COUNT PROGRAM

```
BEGIN {}
 {count = 0}
/       / {count = 1}
/               / {count = 2}
/                       / {count = 3}
/                               / {count = 4}
/                                       / {count = 5}
/                                               / {count = 6}
{print count}
```

APPENDIX D

SOURCE CODE OF NESTING PROGRAM

```
BEGIN { printf "Levels :  \n" > "main.results"
        printf "~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ \n" > "main.results"
                one = 0
                two = 0
                three = 0
                four = 0
                five = 0
                six = 0
                sum = 0    }
                /0/ {zero = zero + 1}
                /1/ {one = one + 1}
                /2/ {two = two + 1}
                /3/ {three = three + 1}
                /4/ {four = four + 1}
                /5/ {five = five + 1}
                /6/ {six = six + 1}
        END { printf "zero  %6d\n", zero  > "main.results"
                printf "one   %6d\n", one   > "main.results"
                printf "two   %6d\n", two   > "main.results"
                printf "three %6d\n", three > "main.results"
                printf "four  %6d\n", four  > "main.results"
                printf "five  %6d\n", five  > "main.results"
                printf "six   %6d\n", six   > "main.results"
                printf "-------------------------\n" > "main.results"
                zeroave = (zero * 100) / NR
                printf "ZERO % =  %5.3f\n", zeroave > "main.results"
                oneave = (one * 100) / NR
                printf "ONE % =  %5.3f\n", oneave  > "main.results"
                twoave = (two * 100) / NR
                printf "TWO % =  %5.3f\n", twoave  > "main.results"
                threeave = (three * 100) / NR
                printf "THREE % = %5.3f\n", threeave > "main.results"
                fourave = (four * 100) / NR
                printf "FOUR % =  %5.3f\n", fourave > "main.results"
                fiveave = (five * 100) / NR
                printf "FIVE % =  %5.3f\n", fiveave > "main.results"
                sixave = (six * 100) / NR
                printf "SIX % =  %5.3f\n", sixave  > "main.results"
                average  = 100 * (zero + one*2 + two*3 + three*4) /NR
                average + = 100 * (four*5 + five*6 + six*7) / NR
                printf"TOTAL AVERAGE = %5.3f\n",average > "main.results"
                sum = zero + one + two + three + four + five + six
                printf "SUM =      %10d\n", sum  > "main.results"
                printf "LINES OF CODE =  %10d\n", NR > "main.results"
                printf"SUM/LINES : %10.3f\n",(sum/NR) > "main.results"
                printf "-------------------------\n" > "main.results"   }
```

APPENDIX E

SOURCE CODE OF TYPEPGM PROGRAM

```
BEGIN { CommentSw = 0; StringSw = 0; LineNumber = 0 }
{
#
# process all the number of fields in the current record.
#
i = 1
if (NF = = 0)
{   count["blanklines"] + +
    LineNumber = NR
}
else { while (i < = NF)
        {  if (CommentSw = = 1)
             { if ($i ~ /\*\//)     CommentSw = 0
             }
           else {
                if (($i ~ /\/[.]*/) && ($i !~ /\//) && ($i != "/"))
                   { CommentSw = 1
                     count["comments"] + +
                     if ($i ~ /\*\//) CommentSw = 0
                   }
                else {
                       if (StringSw = = 1)
                          { if ($i ~ /\"/)       StringSw = 0
                          }
                       else {
                              if ($i ~ /\"/)
                                 { StringSw = 1
                                   if ($i ~ /\"\)/) StringSw = 0
                                 }
                              else {
                                     if ((($1 ~ /\:/) || ($2 ~ /\:/)) && ($i = = $1))
                                        { if ($1 ~ /default/)     count["default"] + +  # ... default
                                          else if ($1 != "case")   count["labels"] + +   # ... labels
                                        }
                                     if ($i ~ /\(/)
                                        { NoOfElement = split($i, Array, "(")
                                          count["functions"] =
                                          count["functions"] + NoOfElement - 1
                                          for (k = 1; k  NoOfElement; k + + )
                                               {  if (Array[k] = = "if")
                                                     {count["if"] + + count["functions"]~}
                                                  else if (Array[k] = = "for")
                                                          {count["for"] + +
                                                           count["assignments"]--
                                                           count["functions"]--
                                                          }
                                                  else if (Array[k] = = "while")
                                                                 {count["while"] + +
```

```
                count["functions"]--
               }
    else if (Array[k] = = "switch")
               {count["switch"] + +
                count["functions"]--
               }
    else if (Array[k] = = "return")
               {count["return"] + +
                count["functions"]--
               }
    else if ((Array[k] = = "getchar") ||
                (Array[k] = = "getc"))
               {count["input"] + +
                count["functions"]--
               }
    else if ((Array[k] = = "scanf") ||
                (Array[k] = = "fscanf"))
               {count["input"] + +
                count["functions"]--
               }
    else if ((Array[k] = = "gets") ||
                (Array[k] = = "fgets"))
               {count["input"] + +
                count["functions"]--
               }
    else if ((Array[k] = = "getw") ||
                (Array[k] = = "read"))
               {count["input"] + +
                count["functions"]--
               }
    else if ((Array[k] = = "putchar") ||
                (Array[k] = = "putc"))
               {count["output"] + +
                count["functions"]--
               }
    else if ((Array[k] = = "printf") ||
                (Array[k] = = "fprintf"))
               {count["output"] + +
                count["functions"]--
               }
    else if ((Array[k] = = "printw") ||
                Array[k] = = "write"))
               {count["output"] + +
                count["functions"]--
               }
    else if ((Array[k] = = "puts"] ||
                (Array[k] = = "fputs")
               {count["output"] + +
                count["functions"]--
                }
}
```

```
    } # end 'if ($i ~ /\(/)'
if ($i ~ /\=/)
    { if (($i !~ /\!=/) && ($i !~ /\= =/) && ($i !~ /\=/))
          {count["assignments"]++
          }
    } end 'if($i '/\=/)'
if  ($i = = "int") count["declarations"]++
else if ($i = = "float") count["declarations"]++
else if ($i = = "double") count["declarations'"]++
else if ($i = = "struct) count['declarations"]++
else if ($i = = "register") count["declarations"]++
else if ($i = = "static") count["declarations"]++
else if ($i = = "char") count["declarations"]++
else if ($i = = "if") {count["if"]++ count["functions"]--}
else if ($i = = "for")
        {count["for"]++
         count["assignments"]--
         count["functions"]--
        }
else if ($i = = "while")
        {count["while"]++ count["functions"]--}
else if ($i = = "switch")
        {count["switch"]++ count["functions"]--}
else if (($i = = "return") || ($i = = "return;"))
        {count["return"]++
         if (($i = = "return") && ($(i+1) ~ /\(/)
             { count["functions"]-- }
        }
else if (($i = = "getchar") || ($i = = "getc"))
        {count["input"]++ count["functions"]--}
else if (($i = = "scanf") || ($i = = "fscanf"))
        {count["input"]++ count["functions"]--}
else if (($i = = "gets") || ($i = = "fgets"))
        {count["input"]++ count["functions"]--}
else if (($i = = "getw") || ($i = = "read"))
        {count["input"]++ count["functions"]--}
else if (($i = = "putchar") || ($i = = "putc"))
        {count["output"]++ count["functions"]--}
else if (($i = = "printf") || ($i = = "fprintf"))
        {count["output"]++ count["functions"]--}
else if (($i = = "printw") || ($i = = "write"))
        {count["output"]++ count["functions"]--}
else if (($i = = "puts") || ($i = = "fputs"))
        {count["output"]++ count["functions"]--}
else if ($i = = "else") count["else"]++
else if ($i ~ /\#/) count["preprocessor"]++
else if ($i = = "case") count["case"]++
else if ($i = = "goto") count["goto"]++
else if (($i = = "break") || ($i = = "break;"))
        count["break"]++
else if (($i = = "continue") || ($i = = "continue;")
```

```
                                        count["continue"] + +
                                }
                        }
                }
        }
        LineNumber = NR
        + +i
    }
  }
}
END {
    printf "-------------------------------\n"          > > "main.results";
    printf "FOR       %10d\n", count["for"]           > > "main.results";
    printf "WHILE     %10d\n", count["while"]         > > "main.results";
    printf "IF        %10d\n", count["if"]            > > "main.results";
    printf "ELSE      %10d\n", count["else"]          > > "main.results";
    printf "SWITCH    %10d\n", count["switch"]        > > "main.results";
    printf "CASE      %10d\n", count["case"]          > > "main.results";
    printf "GOTO      %10d\n", count["goto"]          > > "main.results";
    printf "BREAK     %10d\n", count["break"]         > > "main.results";
    printf "CONTINUE  %10d\n", count["continue"]      > > "main.results";
    printf "ASSIGNMENT %10d\n", count["assignments"]  > "main.results";
    printf "PREPROCESSOR %10d\n", count["preprocessor"]  > > "main.results";
    printf "COMMENT   %10d\n", count["comments"]      > > "main.results";
    printf "BLANKLINE %10d\n", count["blanklines"]    > > "main.results";
    printf "RETURN    %10d\n", count["return"]        > > "main.results";
    printf "INPUT     %10d\n", count["input"]         > > "main.results";
    printf "OUTPUT    %10d\n", count["output"]        > > "main.results";
    printf "FUNCTION  %10d\n", count["functions"]     > "main.results";
    printf "DECLARATION %10d\n", count["declarations"]  > > "main.results";
    printf "DEFAULT   %10d\n", count["default"]       > > "main.results"
     #
     # calculate the weights
     #
     weightS  = 18.4 * count["declarations"] + 11.4 * count["if"]
     weights + =  7.9 * count["for"]        +  8.5 * count["while"]
     weights + =  6.8 * count["switch"]     +  5.6 * count["case"]
     weights + =  4.6 * count["preprocessor"] + 11.1 * count["goto"]
     weights + =  2.4 * count["comments"]
    printf "-------------------------------\n" > > "main.results"
    printf "WEIGHT/LINES =  %10.5f\n", (weights/NR) > "main.results";
    printf "WEIGHT =       %10.5f\n", weights     > > "main.results"
    printf "LINES OF CODE =  %10d\n", NR          > > "main.results";
    printf "-------------------------------\n" > > "main.results";
  }
```

APPENDIX F

SOURCE CODE OF CHANGES PROGRAM

```
echo "%*% This is start of data collection" > > main.results
date > > main.results
****** Pretty-Printing a Program ******
cb
<$1 > 1.cb
cb
<$2 > 2.cb
****** Calculating Occurrence of Statement Types ******
echo "= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = ="  > > main.results
echo "File Name :  " $1 > > main.results
sed 's/"/ " /g
     s/}/} /g
     s/{/ {/g' $1 |
awk -f TYPEPGM
****** Summing the Indentation Level ******
awk -f COUNT 1.cb |  awk -f NESTING
****** Calculating Occurrence of Statement Types ******
echo "= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =" > > main.results
echo "File Name :  " $2 > > main.results
sed 's/"/ " /g
     s/}/} /g
     s/{/ {/g' $2 |
awk -f TYPEPGM
****** Summing the Indentation Level ******
awk -f COUNT 2.cb | awk -f NESTING
****** Finding the Differences ******
diff -e 1.cb 2.cb | grep '^[0-9]' |
****** Extracting the Changed Statements ******
sed 's/\,/ /g
     s/a/ a /g
     s/c/ c /g
     s/d/ d /g' |
awk '
BEGIN {printf "BEGIN {i=0}\n"}
          NF= =2 {if ($2 = = "a")
          {printf "NR = = %d {print \"a\",$0 ;i = 1}\n",$1
printf "NR = = %d {print \"b\",$0 ;i = 1}\n",($1 + 1) }
else {printf "NR = = %d {print \"%s\",$0 ;i = 1}\n",$1,$2 }}
          NF= =3 {for (j=$1;j< =$2;j + +)
          printf "NR = = %d {print \"%s\",$0 ;i = 1}\n",j,$3}
END {}' result
awk -f result 1.cb |
awk ' /^c / {print $0 > "temp" }'
sed 's/c /  /g
     s/{/ {/g
     s/}/} /g
     s/"/ " /g' temp > final
****** Calculating Occurrence of Statement Types ******
```

```
echo "= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =" > > main.results
echo "File Name : " changes.with.TAB > > main.results
awk -f TYPEPGM final
****** Summing the Indentation Level ******
awk -f COUNT final | awk -f NESTING
rm 1.cb 2.cb result temp final
****** Finding the Differences ******
diff -e $1 $2 |grep '^[0-9]' |
****** Extracting the Changed Statements ******
sed 's/\,/ /g
     s/a/ a /g
     s/c/ c /g
     s/d/ d /g' |
awk '
BEGIN {printf "BEGIN {i=0}\n"}
          NF= =2 {if ($2 = = "a")
          {printf "NR = = %d {print \"a\",$0 ;i=1}\n",$1
printf "NR = = %d {print \"b\",$0 ;i=1}\n",($1 + 1) }
else {printf "NR = = %d {print \"%s\",$0 ;i=1}\n",$1,$2 }}
          NF= =3 {for (j=$1;j< =$2;j + +)
          printf "NR = = %d {print \"%s\",$0 ;i=1}\n",j,$3}
END {}' result
awk -f result $1 |
awk '
/^c / {print $0 > "temp" }'
sed 's/c / /g
     s/{/ {/g
     s/}/} /g
     s/"/ " /g' temp > final
****** Calculating Occurrence of Statements Types ******
echo "= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =" > > main.results
echo "File Name : " changes.without.TAB > > main.results
awk -f TYPEPGM final
rm result temp final
echo "%%% This is end of data collection" > > main.results
```

APPENDIX G

SOURCE CODE OF PICK PROGRAM

```
BEGIN {}
{
if ($1 = = "%*%") print $1 > "pick.file"
if (($NF ! = "1987") && ($1 !~ /\=/) &&
        ($1 !~ /\-/) && ($1 ! = "Levels") && ($1 !~ /\~/)
        && (NF ! = 0) && ($NF ! = "data"))
print $NF > "pick.file"
}
```

APPENDIX H

SOURCE CODE OF SEP PROGRAM

```
        BEGIN {no = 0;flag = 1}
        {
if ($1 = = "%*%")
{ + + no
flag = 1
}
if (($1 !~ /\with/) && ($1 !~ /\//) && ($1 != "%*%"))
{
print $0 > no
 + + flag
}
if (flag != 1)
{if (($1 ~ /\with/) || ($1 ~ /\//))
print "\n" no
}
        }
```

APPENDIX I

SOURCE CODE OF VERSION 1 OF EXAMPLE

```
/*************************************************************************/
/*                                                                       */
/*          Procedure   : Recreate Listing          Last Revision :      */
/*                                                                       */
/*          Programmer  : Mike McClure                                   */
/*                                                                       */
/*          Description : This module accepts the data array record and the */
/*                        counter arrayas inputs. It reads the index value */
/*                        of each entity name of the data array and recreates */
/*                        the listing asit was originally read in. This module */
/*                        then printsthe listing. The module append error is */
/*                        called by this module.                         */
/*                                                                       */
/*************************************************************************/


#include   <stdio.h>
#include   </usrb/cs340/ldb/project/structure.h>
#include   </usrb/cs340/ldb/project/counter.h>
#define    begin {
#define    end }
#define    inc + +
#define    EQ = =
#define    NE !=
#define    LE < =
#define    AND &&
#define    NULL '/0'


Recreate_Listing (Data_Record_Array, fp)
struc data_rec *Data_Record_Array;

begin  /* outer loop of data structure */
    int i,j,k; /* looping variables */
    FILE *fp;

for (i = 0; Data_Record_Array[i].Procedure[0] NE NULL AND
                i LE MAX_STRUC_ARR - 1; inc i)
     fprintf (fp,"\n PROCEDURE : %s",Data_Record_Array[i].Procedure[j]);
     if (Data_Record_Array[i].Boolean1[j] > 0)
     Append_Error;

begin  /* loping through procedure arrays */
    for  (j = 0; j LE MAX_FLD_ARR-1; inc j)
         begin
            fprintf (fp,"\n CALLS : %s",Data_Record_Array[i].Calls[j]);
            + + Calls;
         end
```

```
            if (Data_Record_Array[i].Boolean2[j] > 0)
                Append_Error;

    for (j=0; j LE MAX_STRUC_ARR -1; inc j)
        fprintf(fp,"\n EXTERNAL_INPUT : %s",Data_Record_Array[i].Ext_Input[j]);
        if (Data_Record_Array[i].Boolean3[j] 0)
            Append_Error;

    for (j=0: j LE MAX_STRUC_ARR-1; inc j)
        begin
            fprintf (fp,"\n INPUT_GLOBAL : %s",Data_Record_Array[i].Input_Global[j]);
            + +Globals;
        end
        if (Data_Record_Array[i].Boolean4[j] 0)
            Append_Error;

    for (j=0; j LE MAX_STRUC_ARR-1; inc j)
        fprintf (fp,"\n INPUT_PARAMETER : %s",Data_Record_Array[i].Input_Parameter[j]);
        if (Data_Record_Array[i].Boolean5[j] 0)
            Append_Error;

    for (j=0; j LE MAX_STRUC_ARR-1; inc j)
        fprintf (fp,"\n EXTERNAL_OUTPUT : %s",Data_Record_Array[i].Ext_Output[j]);
        if (Data_Record_Array[i].Boolean6[j] 0)
            Append_Error;

    for (j=0; j LE MAX_STRUC_ARR-1; inc j)
        begin
           fprintf (fp,"\nOUTPUT_GLOBAL : %s",Data_Record_Array[i].Output_global[j]);
           + +Globals;
        end
        if (Data_Record_Array[i].Boolean7[j] 0)
        Append_Error;

    for (j=0; j LE MAX_STRUC_ARR-1; inc j)
        fprintf(fp,"\nOUTPUT_PARAMETER:%s",Data_Record_Array[i].Output_parameter[j]);
        If (Data_Record_Array[i].Boolean8[j] 0)
            Append_Error;

    for (j=0; j LE MAX_STRUC_ARR-1; inc j)
        fprintf (fp,"\n ILLEGAL : %s",Data_Record_Array[i].Illegal[j]);
        If (Data_Record_Array[i].Boolean9[j] 0)
            Append_Error;

    for (j=0; j LE MAX_STRUC_ARR-1; inc j)
        fprintf (fp,"\n IGNORED : %s",Data_Record_Array[i].Ignored[j]);
end
```

APPENDIX J

SOURCE CODE OF VERSION 2 OF EXAMPLE

```
/*************************************************************************/
/*                                                                       */
/*        Procedure   : Recreate Listing     Last Revision : 4-30-86 mm  */
/*                                                                       */
/*        Programmer  : Mike McClure                                     */
/*                                                                       */
/*        Description : This module accepts the data array record and the */
/*                      counter arrayas inputs.  It reads the index value */
/*                      of each entity name of the data array and recreates */
/*                      the listing asit was originally read in. This module */
/*                      then printsthe listing. The module append error is */
/*                      called by this module.                           */
/*                                                                       */
/*************************************************************************/


#include   <stdio.h>
#include   </usrb/cs340/ldb/project/structure.h>
#include   </usrb/cs340/ldb/project/counter.h>
#define    begin {
#define    end }
#define    inc + +
#define    EQ  = =
#define    NE  !=
#define    LE< =
#define    AND &&


Recreate_Listing (Counter_Array,Data_Record_Array)
struct data_rec *Data_Record_Array;
rec *Counter_Array;

begin  /* outer loop of data structure */

    int i,j;  /* looping variables */
    FILE *fp, *fopen();


fp = fopen("final_report","a");

printf("\n I am in reclist & befor for loop start\n");
for (i = 0; Data_Record_Array[i].Procedure[0] NE '\0' AND
          i LE MAX_STRUC_ARR - 1; inc i)
     begin  /* looping through procedure arrays  */
printf("I am in 1st for loop for structure array\n");
          fprintf (fp,"\n"," ");
          fprintf (fp,"\n"," ");
          fprintf (fp,"\nPROCEDURE       : %s",Data_Record_Array[i].Procedure);
```

```
if (Data_Record_Array[i].Boolean1 0)
     {
      fclose(fp);
      Append_Error(Counter_Array,&Data_Record_Array[i].Boolean1);
      fp = fopen("final_report","a");
     }

for(j=0;j LE MAX_FLD_ARR-1 AND
      Data_Record_Array[i].Calls[j][0] NE '\0'; inc j)
begin
      printf("\n I am in field loop array\n");
      fprintf (fp,"\n  CALLS          : %s",Data_Record_Array[i].Calls[j]);
      if (Data_Record_Array[i].Boolean2[j]  0)
          {
           fclose(fp);
           Append_Error(Counter_Array,&Data_Record_Array[i].Boolean2[j]);
           fp = fopen("final_report","a");
      }
      inc Counter_Array[12].value;
end

for(j=0;j LE MAX_STRUC_ARR -1 AND
      Data_Record_Array[i].Ext_input[j][0] NE '\0'; inc j)
begin
      fprintf(fp,"\n  EXT_INPUT      : %s",Data_Record_Array[i].Ext_input[j]);
      if (Data_Record_Array[i].Boolean3[j] 0)
           {
            fclose(fp);
            Append_Error(Counter_Array,&Data_Record_Array[i].Boolean3[j]);
            fp = fopen ("final_report","a")
           }
end

for (j=0; j LE MAX_STRUC_ARR-1 AND
      Data_Record_Array[i].Input_global[j][0] NE '\0'; inc j)
begin
      fprintf (fp,"\n  INPUT_GLOBAL    : %s",Data_Record_Array[i].Input_global[j]);
      if (Data_Record_Array[i].Boolean4[j] 0)
           {
            fclose(fp);
            Append_Error(Counter_Array,&Data_Record_Array[i].Boolean4[j]);
            fp = fopen ("final_report","a");
           }
          inc Counter_Array[11].value;
end

for (j=0; j LE MAX_STRUC_ARR-1 AND
      Data_Record_Array[i].Input_parameter[j][0] NE '\0';inc j)
begin
      fprintf(fp,"\nINPUT_VALUE:%s",Data_Record_Array[i].Input_parameter[j]);
      if (Data_Record_Array[i].Boolean5[j] 0)
```

```
            {
             fclose(fp);
             Append_Error(Counter_Array,&Data_Record_Array[i].Boolean5[j]);
             fp = fopen ("final_report","a");
            }
end

for (j=0; j LE MAX_STRUC_ARR-1 AND
      Data_Record_Array[i].Ext_output[j][0] NE '\0';inc j)
begin
      fprintf (fp,"\nEXTERNAL_OUTPUT: %s",Data_Record_Array[i].Ext_output[j]);
      if (Data_Record_Array[i].Boolean6[j] 0)
           {
            fclose(fp);
            Append_Error(Counter_Array,&Data_Record_Array[i].Boolean6[j]);
            fp = fopen ("final_report","a");
           }
end

for (j=0; j LE MAX_STRUC_ARR-1 AND
      Data_Record_Array[i].Output_global[j][0] NE '\0';inc j)
begin
      fprintf (fp,"\nOUTPUT_GLOBAL: %s",Data_Record_Array[i].Output_global[j]);
      if (Data_Record_Array[i].Boolean7[j] 0)
           {
            fclose(fp);
            Append_Error(Counter_Array,&Data_Record_Array[i].Boolean7[j]);
            fp = fopen ("final_report","a");
           }
           inc Counter_Array[11].value;
end

for (j=0; j LE MAX_STRUC_ARR-1 AND
      Data_Record_Array[i].Output_parameter[j][0] NE '\0';inc j)
begin
      fprintf(fp,"\nOUTPUT_NO:%s",Data_Record_Array[i].Output_parameter[j]);
      if (Data_Record_Array[i].Boolean8[j] 0)
           {
            fclose(fp);
            Append_Error(Counter_Array,&Data_Record_Array[i].Boolean8[j]);
            fp = fopen ("final_report","a");       }   end
            for (j=0; j LE MAX_STRUC_ARR -1 AND
                 Data_Record_Array[i].Illegal[j][0] NE '\0';inc j)
            begin
                 fprintf (fp,"\n  Illegal      : %s",Data_Record_Array[i].Illegal[j]);
                 if (Data_Record_Array[i].Boolean9[j] 0)
                 {
                 fclose(fp);
                 Append_Error(Counter_Array,&Data_Record_Array[i].Boolean9[j]);
                 fp = fopen ("final_report","a");
                 }
```

```
                    end

        for ( j = 0; j LE MAX_STRUC_ARR-1 AND
                Data_Record_Array[i].Ignored[j][0] NE '\0';inc j)
            fprintf (fp, "\n Ignored     : %s", Data_Record_Array[i].Ignored[j]);
    end
      fclose(fp);
end
```

APPENDIX K

SAMPLE OUTPUT FROM PROBIT FOR QUANTITATIVE ANALYSIS OF DEBUGGING

```
1          SAS(R) LOG    OS SAS 5.16            OS/MVT JOB VM185600 STEP
SUBMIT    PROC SAS                18:56 WEDNESDAY, APRIL 6, 1988

NOTE: COPYRIGHT (C) 1984,1986 SAS INSTITUTE INC., CARY, N.C.  27511,
U.S.A.
NOTE: THE JOB VM185600 HAS BEEN RUN UNDER RELEASE 5.16 OF SAS AT KANSAS
STATE UNIVERSITY (03010001).

NOTE: SAS OPTIONS SPECIFIED ARE:
      NOINCLUDE NOGRAPHICS  SORT=4

NOTE:  SAS 5.16 has replaced SAS 82.3.


1           OPTIONS  LS=72;
2           DATA;
3            INPUT DOSE N RESPONSE;
4            CARDS;

NOTE: DATA SET WORK.DATA1 HAS 13 OBSERVATIONS AND 3 VARIABLES. 680 OBS/T
RK
NOTE: THE DATA STATEMENT USED 0.20 SECONDS AND 372K.

18          ;
19          PROC PRINT;
20            VAR DOSE N RESPONSE;
NOTE: THE PROCEDURE PRINT USED 0.25 SECONDS AND 422K
      AND PRINTED PAGE 1.

21          PROC PROBIT LOG10;
22           VAR DOSE N RESPONSE;
NOTE: THE PROCEDURE PROBIT USED 0.54 SECONDS AND 420K
      AND PRINTED PAGES 2 TO 6.
NOTE: SAS USED 422K MEMORY.

NOTE: SAS INSTITUTE INC.
      SAS CIRCLE
      PO BOX 8000
      CARY, N.C. 27511-8000
```

OBS	DOSE	N	RESPONSE
1	32	1	1
2	24	1	1
3	19	1	1
4	18	1	1
5	11	1	0
6	9	2	1
7	8	3	2
8	7	1	1
9	5	1	0
10	3	4	2
11	2	4	1
12	1	3	0
13	0	41	0

PROBIT ANALYSIS ON LOG10(DOSE)

ITERATION	INTERCEPT	SLOPE	MU	SIGMA
0	4.19764739	1.12527653	0.71302706	0.88867045
1	3.66586399	1.86023043	0.71718858	0.53756781
2	3.58409386	1.97505921	0.71689301	0.50631393
3	3.58208644	1.97790611	0.71687607	0.50558517
4	3.58208523	1.97790783	0.71687606	0.50558473

COVARIANCE MATRIX

	INTERCEPT	SLOPE
INTERCEPT	0.39202706	-0.44235206
SLOPE	-0.44235206	0.64162248

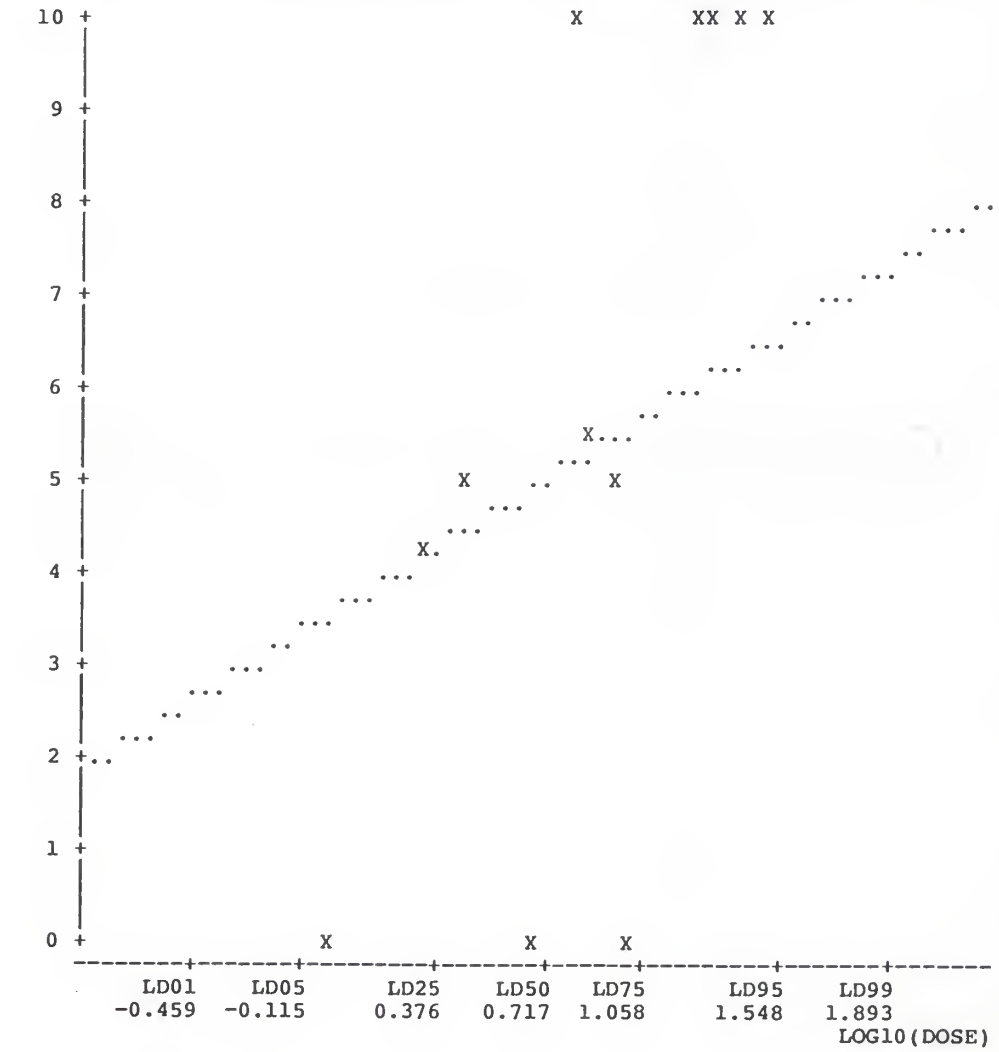
COVARIANCE MATRIX

	MU	SIGMA
MU	0.02237685	0.00227606
SIGMA	0.00227606	0.04192329

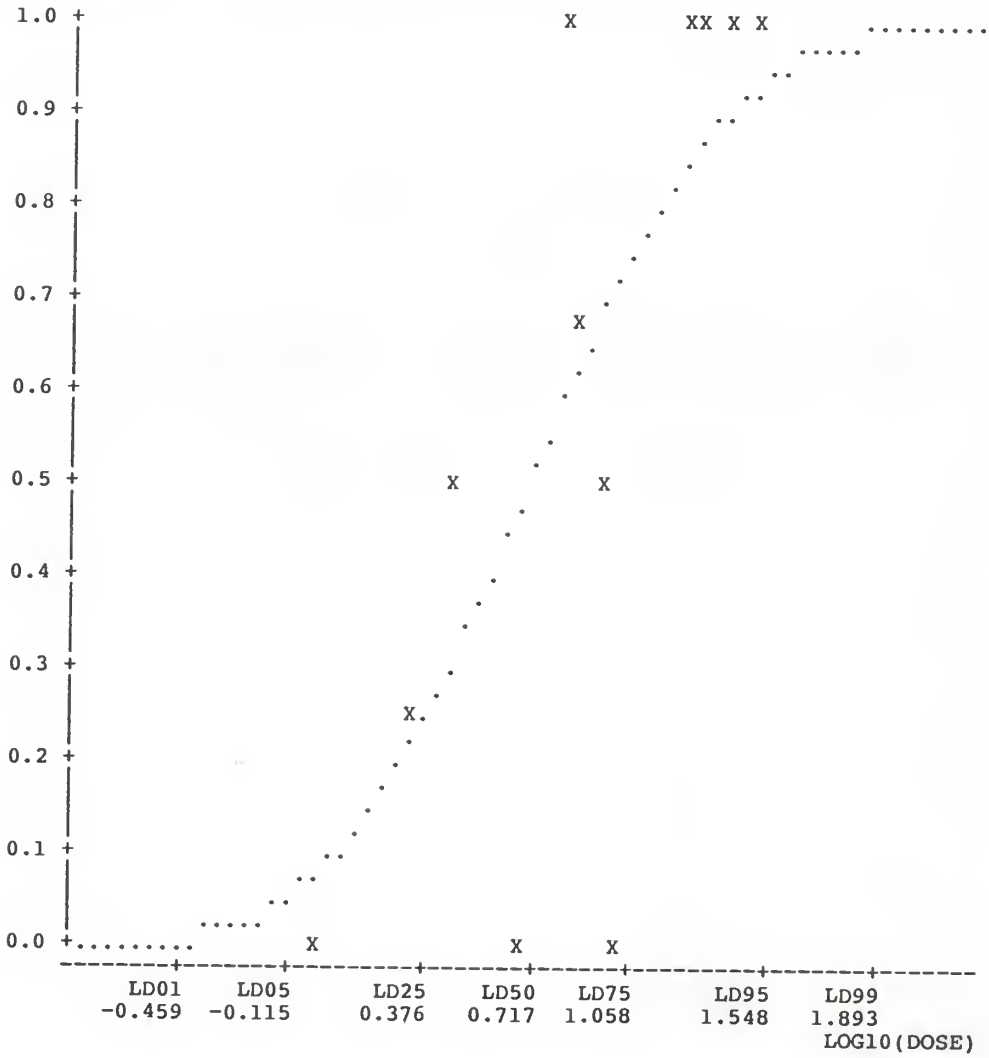
CHI-SQ = 6.1631 WITH 10 DF PROB > CHI-SQ = 0.8014

NOTE: SINCE THE CHI-SQUARE IS SMALL (P > 0.10), FIDUCIAL LIMITS WILL BE COMPUTED USING A T VALUE OF 1.96 .

PROBIT PROBIT ANALYSIS ON LOG10(DOSE)



PROBABILITY PROBIT ANALYSIS ON LOG10(DOSE)



PROBIT ANALYSIS ON LOG10(DOSE)

PROBABILITY	LOG10(DOSE)	95 PERCENT FIDUCIAL LIMITS	
		LOWER	UPPER
0.01	-0.45928991	-4.92685122	0.09547234
0.02	-0.32146803	-4.26477822	0.17849658
0.03	-0.23402447	-3.84563565	0.23209431
0.04	-0.16824409	-3.53095349	0.27303627
0.05	-0.11473682	-3.27547273	0.30682829
0.06	-0.06919373	-3.05843356	0.33600569
0.07	-0.02926135	-2.86850248	0.36195838
0.08	0.00649333	-2.69878239	0.38553646
0.09	0.03901078	-2.54475000	0.40730102
0.10	0.06894315	-2.40327159	0.42764398
0.15	0.19287116	-1.82167412	0.51603026
0.20	0.29136521	-1.36666280	0.59350078
0.25	0.37586434	-0.98505006	0.66871004
0.30	0.45174716	-0.65387609	0.74777635
0.35	0.52206391	-0.36300645	0.83705548
0.40	0.58878763	-0.10959796	0.94437098
0.45	0.65334360	0.10498151	1.07879615
0.50	0.71687606	0.27931762	1.24793134
0.55	0.78040852	0.41652275	1.45419751
0.60	0.84496448	0.52466424	1.69506067
0.65	0.91168820	0.61309224	1.96735667
0.70	0.98200495	0.68962129	2.27097637
0.75	1.05788778	0.76016205	2.61067589
0.80	1.14238690	0.82955995	2.99810000
0.85	1.24088095	0.90293269	3.45720910
0.90	1.36480896	0.98829210	4.04183343
0.91	1.39474133	1.00811672	4.18383019
0.92	1.42725878	1.02938413	4.33835972
0.93	1.46301347	1.05248274	4.50855928
0.94	1.50294585	1.07796964	4.69895615
0.95	1.54848894	1.10669008	4.91645229
0.96	1.60199620	1.14002741	5.17238774
0.97	1.66777659	1.18050666	5.48753260
0.98	1.75522015	1.23361420	5.90716537
0.99	1.89304202	1.31606856	6.56980824

PROBIT ANALYSIS ON LOG10(DOSE)

PROBABILITY	DOSE	95 PERCENT FIDUCIAL LIMITS	
		LOWER	UPPER
0.01	0.34730425	0.00001183	1.24586889
0.02	0.47701492	0.00005435	1.50833072
0.03	0.58341223	0.00014268	1.70645292
0.04	0.67882200	0.00029447	1.87515109
0.05	0.76782664	0.00053031	2.02688118
0.06	0.85271964	0.00087411	2.16773250
0.07	0.93484293	0.00135362	2.30122125
0.08	1.01506378	0.00200086	2.42960940
0.09	1.09398353	0.00285266	2.55447124
0.10	1.17204194	0.00395119	2.67697296
0.15	1.55908991	0.01507738	3.28118154
0.20	1.95598361	0.04298701	3.92193848
0.25	2.37609794	0.10350229	4.66347916
0.30	2.82974411	0.22188294	5.59469418
0.35	3.32708512	0.43350444	6.87156210
0.40	3.87960607	0.77696604	8.79773713
0.45	4.50135846	1.27344885	11.98936417
0.50	5.21045989	1.90246913	17.69829150
0.55	6.03126645	2.60929243	28.45755029
0.60	6.99784766	3.34706570	49.55194094
0.65	8.15996323	4.10291234	92.75912967
0.70	9.59411565	4.89351910	186.62781588
0.75	11.42583048	5.75654690	408.01477870
0.80	13.87991806	6.75398276	995.63464374
0.85	17.41329485	7.99710297	2865.55731738
0.90	23.16375491	9.73401698	11011.16909866
0.91	24.81654571	10.18865176	15269.68899306
0.92	26.74599656	10.70000878	21795.14307647
0.93	29.04112710	11.28451091	32252.19550293
0.94	31.83800505	11.96656871	49998.40542578
0.95	35.35810134	12.78468643	82499.68400879
0.96	39.99412541	13.80471374	148726.29000454
0.97	46.53466462	15.15328058	307278.80195588
0.98	56.91413614	17.12435399	807542.47142539
0.99	78.17034381	20.70468193	3713712.13215244

CLASSIFYING PROGRAM CHANGES DURING SOFTWARE DEVELOPMENT

by

Yu-Hua Hsu

B.S., Central State University, 1984

AN ABSTRACT OF A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ABSTRACT

The historical data of a program collected during its development phase contain important information regarding the activities of the software development. This work proposes, both qualitatively and quantitatively, a program change pattern classification and a set of intuitive rules for effective evaluation of the changes during software development. It is important that a software manager sees and interprets the pattern changes during software development. The intuitive rules are designed to facilitate the analysis of those changes; the results can be used to aid the software manager in evaluating the software development.