REAL-TIME SOFTWARE METRICS

Andreas Roesch

Computer Science Department

Oregon State University

Corvallis, Oregon 97331-3202

A project submitted to Oregon State University

in partial fulfillment of the requirements for the degree of

Master of Science

Completed May 10.1993

ACKNOWLEDGMENTS

A special note of thanks goes to Curt Cook, my advisor for the many hours of discussions required to get me going in the right direction. All of his help was invaluable.

I also owe very special thanks to my wife, Dagmar who put up with me while I went through this whole process. I also like to mention my son, Alexander Justin who gave me interesting new ideas while playing with the keyboard of my computer.

To these people and everyone else I forgot to mention, thanks.

TABLE OF CONTENTS

ABSTRACT

This study describes the software metrics analysis of 10 releases of an embedded real-time telephone switching system developed by a German telecommunications firm. The micro-controlled application was written in a C-like macro assembly language. We developed a metrics program that computes the standard complexity metrics plus a number of information flow metrics.

The releases of the real-time software satisfies published laws of software evolution, e.g. continuing change, increasing entropy, and total change is not uniform over the changed modules. The data also supports Harrison and Cook's program maintenance decision model [7]. We propose the change standard deviation as a threshold for their model.

A multivariate analysis of the metrics computed with our metric analyzer program identified four underlying complexity domains: size, information flow into functions, information flow out of functions and control flow. We also found that the information flow metrics characterize real-time complexity better than the standard software complexity metrics, e.g. Halstead's Software Science, LOC, McCabe's Cyclomatic Complexity. We also investigated the relations between programming hours for the various releases and the program changes and changes in metric values.

1. INTRODUCTION

Real-time software is generally designed to control a process interactively, as the process unfolds in time. Examples are control of airplanes (avionics), control of transportation systems (e.g. BART), and control in automobiles and appliances. The unique feature of real-time software is the time constraint - all subtasks must meet individual timing requirements.

Real-time software is considered to be different from other software. The timing constraint may mean a different design or testing methodology. In this paper we investigate the evolution and complexity of real-time software. The basic questions we address are:

1. Can we characterize the evolution of real-time programs?
2. What are the underlying complexity domains in real-time programs?
2. What types of software metrics identify the complex parts of real-time programs?
3. What is the relation between programming effort and program changes?

In an attempt to answer these questions we analyzed 10 releases of a real-time telephone switching system program developed by a German telecommunications firm. We developed a software metrics tool that computed a variety of measures. From the measures we were able to study the changes between successive releases for all 10 releases. We also investigated the relation between the programming hours for the 10 versions and software metrics.

In chapter 2 we describe the real-time program and give an overview of the system design and development process. We will show that the software can be characterized as a reactive and embedded hard real-time system.

In Chapter 3 we give an introduction to software complexity and software complexity metrics. We describe some of the most common metrics out of the four traditional classes of software complexity metrics: size metrics, data structure metrics, control flow metrics and information flow metrics.

Chapter 4 describes the metric analyzer program we developed and the metrics it computes. Note that our tool computes the traditional software complexity metrics plus a number of information flow metrics.

Chapter 5 investigates the evolution of the program. Characterizing the evolution of the program both in terms of the number of functions changed and the amount they are changed has important implications for both software maintenance and development. For example, Harrison and Cook [7] proposed a maintenance change model to determine whether a given software module can be effectively modified or whether it should be completely redesigned and rewritten. Complete redesign and rewrite is expensive, but it is even more expensive if the module structure has seriously deteriorated with severe ripple effects. From software evolution data they found that a few modules account for most of the total amount of maintenance changes. Since a module's complexity increases and its structure deteriorates with changes, it is important to detect modules that will undergo a large number of changes. The early identification of a these change prone modules will allow a complete redesign and rewrite of the module and thereby greatly reduce the cost of later changes to the module. They suggested early identification of the change prone modules through changes in software metrics across release cycles and proposed setting a threshold. Once the total change to a module exceeds the threshold the module is classified as change prone and is redesigned and rewritten the next time it undergoes maintenance. Our results confirm this maintenance change model and suggests using the standard deviation of the Halstead's Volume changes as a threshold.

In chapter 6 we studied the internal structure of the set of metrics computed with the analyzer tool. In order to understand the relation among metrics, we applied a statistical technique known as factor analysis. Munson and Khoshgoftaar [16, 17, 18] found that most metrics are measuring the same elements of a rather small set of orthogonal complexity domains. There are relatively few distinct sources of variation among metrics. Our results show that the 18 metrics map onto four underlying complexity domains: size, information flow into functions, information flow out of functions and control flow. We found that the information flow metrics contribute considerable variation to the factor model and characterize real-time complexity better than the standard complexity metrics. The real-time functions have a much higher average in and out flow of information than non-real-time functions.

In the chapter 7 we relate programming hours for the various releases to program changes and the software metrics. Our conclusions and future work are discussed in chapter 8.

2. REAL-TIME SYSTEMS

In this chapter we will give an overview of the system design and the development process of the software used in this study. We will show that the application can be classified as a reactive and embedded hard real-time system.

2.1 REACTIVE EMBEDDED REAL-TIME SYSTEMS

A real-time system is a system whose correctness depends on timeliness as well as logical correctness. Real-time systems must satisfy explicit bounded response time constraints or it is assumed that it will fail. A failure is defined as the inability of the system to perform according to system specification. In the case of the Space Shuttle or a nuclear power plant it is painfully obvious when a failure has occurred. Failure to respond quickly to a nuclear reactor over-temperature problem, could result in a melt-down. For other systems, such as a telephone switching system, the notion of a failure is less clear. A telephone switching system for example must be able to handle a peak rate of incoming internal and external calls, when all subscribers try to make a call at once.

Real-time systems are often reactive or embedded systems. Reactive systems are those which have some ongoing interaction with their environment. One system constantly reacts to buttons pressed asynchronously by an operator. Embedded systems are those used to control specialized hardware and lack an operating system and associated devices for general user interface. For example the software used to control the Space Shuttle is reactive and highly embedded.

Further, literature distinguishes between soft and hard real-time systems [10]. Soft real-time systems are systems where performance is degraded by failure to meet response time constraints. For example an airline reservation system may degrade under heavy load, but it will eventually process all passenger requests accurately. Systems where failure to meet response time constraints leads to catastrophic results are called hard real-time systems. The telephone switching system described in this section is a reactive embedded hard real-time system.

2.2 DEVELOPMENT OVERVIEW

In this study we analyzed ten versions of an embedded micro-controlled telephone switching system program developed by a German telecommunications firm over a period of two years. The programming team consisted of 3 experienced programmers. The detailed hardware design was completed before software development started. Thus, the software design was constrained by the given hardware resources.

A fully functional α-version of the size 10,577 lines of code and 223 functions in 13 modules was released in December 1990. Up to this date 1,771 programming hours were spent on system design, coding, testing and system integration. Most of the change activity for the 9 later versions can be characterized as perfective maintenance. The last version which consists of 13,621 line of code and 359 functions was released in June 1992. Over the nine releases 1,456 hours were spent on maintenance. Figure 2.1 shows the release dates and cumulative hours for each release starting with the first release.

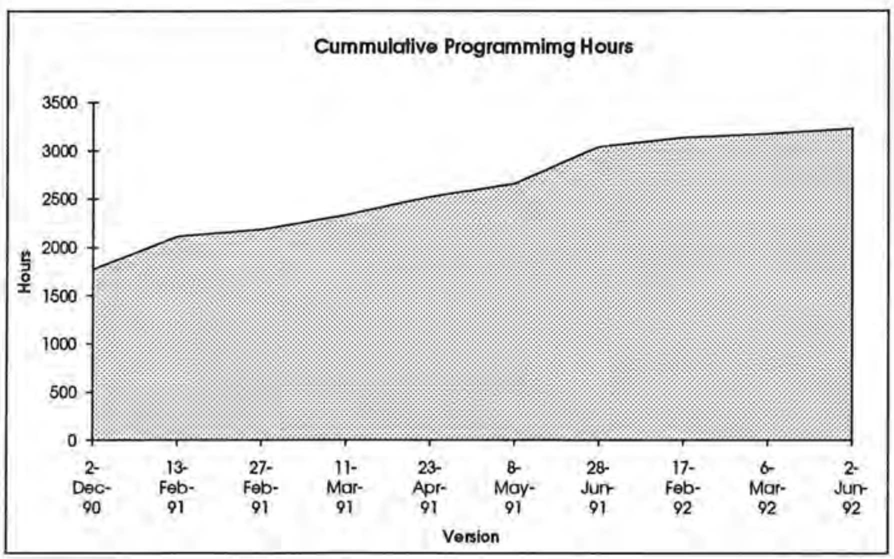

Figure 2.1. Development Schedule

2.3 System Overview

The telephone switching system supports up to four telephone sets. Each telephone set may be used independently of the other sets. A subscriber can either communicate with one of the other telephone units or connect to an external dial-up network.

The software controls about 50 input and output lines connected to a single-chip micro controller. Some of the inputs are checked frequently and deadlines to evaluate incoming events must be met within milliseconds. For example the frequency of incoming calls has to lie in a well defined range of frequencies in order to be identified as a call. Further the call must have a specific signal pattern to be valid. This is necessary to distinguish between valid calls and noise on telephone lines.

Similar time constraints are defined for all external in- and outgoing signals to ensure fail safe operation of the system. Every telephone switching system has to pass a final admission test, similar to the FCC regulations, where all time requirements are checked carefully. If response time constraints are not met within the specified range the system will fail. Accordingly the application can be classified as a hard real-time system. Since the micro controller lacks an operating system as well as system software for interface handling the software is highly embedded. Further, several hardware resources are mutually exclusive and non-preemptive. For instance the single tone dial unit is shared between the four telephone units. Hence concurrency control has to be provided among processes.

The underlying software implementation is based on finite automata with a total of 39 finite automata and 8 interrupt-handler. The main application program for all four telephone sets checks the current state of execution for each process (telephone unit) every ten milliseconds. Each process has its own set of global variables to store state, incoming external events, and additional information about the process. The real-time control part checks incoming and outgoing external signals every millisecond. All input and output drivers of the micro controlled hardware were completed prior to the first release in December 1990. Few changes to the interface modules were made during program maintenance.

Events to and from the real-time control to the main application are also passed via global memory. Because of limited storage space (typically less than 512 bytes of RAM in today's single-chip micro controller chips) parameters are not passed using the processor's stack, but instead information is exchanged through global memory or processor registers. This is one reason why information flow and information flow density turn out to be important complexity metrics for this real-time system.

2.4 The Structured Macro Assembly Language

The software was written in a structured relocatable macro assembly language. The basic instruction set of the assembler language contains the following C-like control structures: ASSIGNMENT, IF-THEN-ELSE, FOR-NEXT, DO-WHILE, SWITCH-CASE, BREAK and CONTINUE statements:

ASSIGNMENT Statement:

```
C = 0
BIT_A5 = 0
[WORK] = 10
[WORK1] = [WORK2]
```

IF - ELSE - ENDIF Statement:

```
IF [FLAG]
  [WORK] = 1
ELSE
  [WORK] = 2
ENDIF
```

FOR - NEXT Statement:

```
FOR [FLAG]
   JSR OUTPUT
NEXT
```

DO - WHILE Statement:

```
DO
   JSR OUTPUT
WHILE [FLAG]
```

SWITCH - CASE - ENDS Statement:

```
SWITCH [WORK]
   CASE 1
      JSR OUTPUT1
      BREAK
   CASE 2
      JSR OUTPUT2
      BREAK
   CASE 3
      JSR OUTPUT3
      BREAK
   DEFAULT
      JSR OUTPUT4
ENDS
```

With these structured commands it is possible to program without using GOTO statements such as BRANCH and JUMP statements of the assembler language. For example the use of an IF-THEN-ELSE statement eliminates the need to create labels. This greatly simplifies programming and leads to easy to read Single-Entry-Single-Exit structured programs. GOTO statements were almost entirely avoided in this project. Most of the code is written using only the structured macro language. Only when necessary were assembly language statements used.

A code sample is given below:

```
      .FUNC CLRKPT
;*******************************************************************
; CLRKPT          : CLEAR KPT
; PARAMETER       : VOID
; GLOBAL          : VOID
; RETURN          : VOID
;*******************************************************************

CLRKPT
      X = 0
      Y = 0
      DO
            DO
                  A = Y
                  [PORT2] = A | [KPTKTZ,X]
                  IF [PORT2] == $20
                        [STROBE] = 1
                  ELSE
                        [STROBE] = 0
                  ENDIF
            WHILE Y < 16
            Y = 0
            X = ++X
      WHILE X < 8
      [OUTBL1] = 0
      [OUTBL2] = 0
      [PROMFF] = [PROMFF] & $F0
      RTS
.ENDFUNC CLRKPT
```

3. SOFTWARE COMPLEXITY METRICS

Software complexity metrics are objective measures of how complex source code is and how difficult it may be for a programmer to test, maintain, or understand programming source code [4]. Software complexity metrics do not measure the complexity itself, but instead measure the degree to which those characteristics thought to contribute to complexity exist within the source code [19]. Many different complexity metrics have been proposed and there is no agreement as to which program characteristics contribute most to the complexity of a program. However, there are four traditional classes of software complexity metrics that characterize different aspects of program complexity: size metrics, data structure metrics, control flow metrics and information flow metrics. In the following sections of this chapter we describe some of the most common metrics out of each group.

3.1 SIZE METRICS

Almost everyone agrees that the amount of effort necessary to construct a program depends upon the number of lines that are written. Thus the line of code measure is probably the most widely used metric in software complexity analysis. It is an important factor in many models of software development and easy to compute after the program is completed. Although lines of code seem to be a simple measure, there is no general agreement about what constitutes a line of code. But most researchers agree on the following two definitions:

1. LOC (lines of code) is any line of program text that is delivered to the customer and includes comment and blank lines. Sometimes also referred to as DSL (deliverable source lines).

2. NCSL (non commentary source lines) is any line of a program that is not exclusively a comment or a blank line.

Size measure of larger and smaller granularity have been proposed. For example, in a large program, the number of functions is commonly used. At the other extreme, the number of tokens is a size measure that accounts for differences in the number of components in a line of code. The token count is like a weighted line count.

Halstead [6] proposed a large family of size metrics called Software Science based on token counts. His theory of Software Science [6] is probably the best known and most thoroughly studied composite measures of software complexity. Software Science measures are based on four counts of primitive tokens in the program:

n1 = the number of unique operators that appear in a program

n2 = the number of unique operands that appear in a program

N1 = the total number of operator occurrences

N2 = the total number of operand occurrences

One composite measure of size, called length, is the total number of tokens, which is the sum of the total operator and operand count: N = N1 + N2. Halstead also defines the term vocabulary, the sum of unique operators and operands: n = n1 + n2. Further he hypothesized that the length of a well-structured program, Nhat, is a function of the number of unique operator and operand: Nhat = n1 $\log_2$ n1 + n2 $\log_2$ n2.

Halstead suggested another commonly used measure for the size of a program, called Volume: V = N $\log_2$ n. Volume may also be interpreted as the number of mental comparisons needed to write a program of length N. Another metric from this family is Effort, which is based on the program Volume and the program Level, where Level is a measure of abstraction in a particular implementation of an algorithm. Effort is defined as: E = V / L = (n1 N2 N $\log_2$n) / (2 n2).

3.2 DATA STRUCTURE METRICS

Data structure metrics capture the amount of data, the usage of data in a module, and the degree to which data is shared among modules. Like to size metrics there are various methods to measure data structure in a program. One simple way for determining the amount of data is to count the number of entries in the cross-reference list generated by compilers and assemblers. Such a count of variables is referred to as VARS. Other popular data structure measures are Halstead's n2 and N2.

3.3 CONTROL FLOW METRICS

Control metrics measure the complexity of the logic structure of the program. By far the most popular control flow metric is the Cyclomatic Complexity V(G) proposed by McCabe [12]. V(G) is a count of the number of linearly independent paths through a program and is a measure of the programs control flow. It is calculated from the formula: $V(G) = e - n + 2$, where e is the number of edges and n is the number of nodes in the flow graph. It turns out that McCabe's Cyclomatic Complexity can be easily computed by simply adding one to the total count of decisions in a program. Other control metrics are nesting depth and number of distinct paths in a program.

3.4 INFORMATION FLOW METRICS

Information flow metrics measure directly the system connectivity by observing the flow of information or control among system components. They focus on the interface between the major levels in a hierarchically structured program. By observing communications among the system components measurements for complexity, module coupling and module interaction can be defined. Henry and Kafura [8] proposed an information flow metric based on module length, fan-in and fan-out. They defined the fan-in of a module as the number of modules that pass data directly or indirectly to the module. Similarly the fan-out of a module is the number of modules to which data is passed either directly or indirectly. They have shown that information flow of system interconnectivity gives reasonable results in measuring changes to large-scale systems.

A major drawback of Henry and Kafura's information flow metric is that it is not easily computed. A more readily available measure of interconnectivity is given by the function call chart, which reflects the hierarchical structure of modules within a program.

4. SOFTWARE METRICS TOOL

We wrote a software metrics analyzer program that computed a variety of standard software complexity metrics [3] (Lines of Code (LOC), Noncommentary Source Lines (NCSL), Halstead's Software Science measures (V,E), and McCabe's V(G)). These were straightforward to compute since much of the program was written using the C-like control structures. Since communication among system components is an important aspect in real-time systems we also included a set of information flow metrics. In the following sections we will define the metrics calculated by our analyzer tool. We will explain how the tool can be used and provide an overview of its software design.

4.1 METRICS COMPUTED BY THE ANALYZING TOOL

The software metrics analyzer program computes a number of traditional software complexity metrics. The abbreviations for the complexity metrics used in our study are given below:

n1 =	Number of unique operators
n2 =	Number of unique operands
N1 =	Total number of operator occurrences
N2 =	Total number of operand occurrences
Nhat =	Halstead's Length
V =	Halstead's Volume
E =	Halstead's Effort
V(G) =	McCabe's Cyclomatic Complexity
LOC =	Line of Code
NCSL =	Noncommentary source lines

We included two metrics that measure interconnectivity among modules within a program:

FIN =	Number of times a function is called by another function
FOUT =	Number of times a function calls another function

Because of indirect calls, FIN and FOUT counts are only approximations to the actual number of calls.

Since this was a real-time application in which considerable information is passed via global data, we counted the number of global and resource variables referenced and/or changed. Notice that we differentiate between resource and global variables. The resource variables refer to the variable identifiers through which the programmer accesses timers, I/O ports, serial interfaces, interrupt inputs, and special registers. These are assigned by the system. Global variables are programmer defined variables. The following information flow metrics are computed:

VOUT =	Number of times global variables are changed
VIN =	Number of times a global variables are referenced
UVOUT =	Number of unique global variables changed
UVIN =	Number of unique global variables referenced
VROUT =	Number of times global and resource variables are changed
VRIN =	Number of times global and resource variables are referenced
UVROUT =	Number of unique global and resource variables changed
UVRIN =	Number of unique global and resource variables referenced

4.2 HOW TO USE THE ANALYZER TOOL

The metric analyzer program *metric.exe* is written for IBM-PC and compatible systems. It analyses structured relocatible assembly language code of Mitsubishi's micro-controller series MELPS 740 [13, 14]. The metric tool requires the file *op.txt* where the operators of the assembly language are specified. All other tokens are considered as operands.

The program can analyze a single input file as well as an entire project. The output file in form of a table has one output-line for each function. The first output-line is a column header, the second is a summary for the complete module (input file) followed by the metric counts for individual functions. The printout of the detailed metric count for each function can be suppressed by specifying a command line parameter. A function is identified by the pseudo-command ***.FUNC*** *function-name* (see also example in chapter 4 and Mitsubishi's User's Manual [13]).

The metric analyzer allows the use of the following command line parameters:

metric [[@]***filename***] [***-mh***]

Where ***filename*** is a single input file, ***@filename*** is a project file that contains one or more input files. If there is no input file specified the analyzer reads from standard input. Parameter ***-m*** suppresses the output of the metrics for individual functions and reports modules only. Parameter ***-h*** prints out a help screen. The report is printed to standard output and can be easily redirected into a file. Since all table entries are separated by tabulators the report file can be read into standard spreadsheet applications.

It is important to know that the metrics reported in the summary for an entire module are not always the simple sum of the metrics for individual functions. In particular Halstead's n1, n2 and the unique information flow metrics for the module summary are based on the entire input file for the module summary. Hence, Halstead's Length Nhat, Volume V and Effort E are also different for the module summary output.

It should also be noted that the computation of the function call hierarchy (metrics FIN and FOUT) is for an entire project. Otherwise function calls to and from a single input file from other project files can not be considered.

4.3 IMPLEMENTATION DETAILS

The metric analyzer is written in C and runs under DOS and IBM-PC compatible systems. The tool contains 3 source files with 1.5K deliverable source lines. It is written and compiled with Borland C++ 3.1.

The lexical analysis of the input files is performed with a tool called FLEX. FLEX is an lexical analyzer similar to the UNIX tool lex and was developed by the University of California, Berkeley. The FLEX tool is portable to various platforms like UNIX, DOS, MACINTOSH etc. The analyzer processes each input file twice. This is necessary to calculate the function call hierarchy across multiple input-files. Since the semantics of assembly language is not complex, the entire language structure is recognized by using state variables.

The major data structure in the analyzer tool is a symbol table that holds all recognized tokens of the assembly as well as the C-like macro language. For fast access an open addressing hashing scheme is used. A double hashing algorithm is used to avoid clustering in the hash table . Once the symbol table is complete most metrics are computed by scanning through the symbol table.

The entire project consists of the following source files (a complete printout is given in Appendix A): *Metric.c* and *metric.h* contain the main program, functions to process input files, functions to output the metric counts and the data structures to count the metrics. *Metric.l* holds the lexical definitions for the analyzer tool and serves as an input to FLEX. The output file produced with the FLEX compiler is named *lexyy.c*. *Hash.c* and *hash.h* implement functions to build and manipulate the symbol table. The project build file includes the files *metric.c*, *hash.c* and *lexyy.c*. The huge memory model should be used to recompile the software.

It is interesting to note that we discovered inconsistencies in programming style in a variety of functions during development of the metric analyzer tool. When verifying the functionality of the analyzer tool we were sometimes puzzled that very different programming techniques were used. For example, within an indexed addressing scheme programmers used the index register as the base address and manipulated the base address for index calculations. The inconsistencies appear in some but not all functions and are probably due to a lack of coding standards. It is very likely that missing coding standards lead to code that is difficult to comprehend and therefore hard to maintain. Unfortunately it is very difficult to recognize these inconsistencies with a metric analyzer tool.

5. PROGRAM EVOLUTION

In this section we look at the evolution of the program. The first question we addressed was to characterize the evolution of the program. In particular we were interested in the distribution of changes that were made during program maintenance. Did the changes coincide with what other software maintenance studies have found? Or were they different because the program was a real-time application? In the last part of this section we show that our data supports the maintenance change model proposed by Harrison and Cook and suggests using the standard deviation of the changes in volume as a threshold.

5.1 EVOLUTION OF THE REAL-TIME SOFTWARE

There were ten versions of the program. The first version of the program was released in December 1990 and the tenth in June 1992. The time between versions ranged from twelve days to several months. The final version of the program is made up of thirteen modules each of which consists of one or more functions. For each version, Table 5.1 gives the release date, and number of functions in each module.

Lehman [11] and Belady and Lehman [1] studied the program maintenance changes in a variety of software systems over a period of years. Since software does not wear out or break, they felt that the term "software evolution" more accurately described the pattern of changes to the programs. From their observation they formulated Laws of Program Evolution. The two most important and universally accepted of these laws are:

1. All useful programs undergo continuing change. Useful programs are continually improved through the addition of new features as evidenced by the number of commercial products (MS DOS, Lotus 1-2-3, UNIX, etc.) that have evolved through a number of major release cycles.

2. Over time, programs exhibit increasing entropy. As changes are made to a program, its structure degrades and its size increases, resulting in increased complexity. Lehman and Belady [1] cite an IBM operating system that increased from 3,682 modules to 4,800 modules over four major release cycles. Increasing entropy makes program maintenance increasingly more difficult. Ultimately, the program will need to undergo a major and expensive overhaul or will be replaced by another program. One sign of entropy is an increasing ripple effect as a change to one part of the software affects a higher percentage of the other parts of the software

In a study of a successive versions of a real-time embedded software system, Harrison and Cook [7] noticed that most of the total change was concentrated in a few modules. This led them to propose another law of software evolution.

3. Total program change is not uniform over the changed modules. Most studies of software evolution look at the number of modules changed in successive versions. These studies have found that less than half of the modules are changed. Harrison and Cook looked more closely at the amount of changes in successive versions. They found that changes to 10% of the modules accounted for 60% of the total change.

Our data for the 10 versions supports all of these laws. Table 5.1 gives an overview of all metrics computed with our metric tool for the ten versions. The following Figures 5.1 to 5.6 show the evolution of various metrics normalized by the metric values of the last version for successive releases. The data in Table 5.1 and Figure 5.1 clearly confirms the first two laws of program evolution. The program experienced continual change and increasing entropy (complexity and size) between successive versions. The number of functions increased from 223 in version 1 to 359 in version 10; the lines of code (LOC) continually increased (Figure 5.3). With few exceptions, the Halstead measures (V, E) and McCabe's Cyclomatic complexity V(G) increased as well (Figure 5.4). However, note the unusually large increase in FIN, FOUT, and in the number of functions, between versions 4 and 5 (Figure 5.2). This occurred because by version 4 the available 16K of memory was nearly exhausted so that in version 5 macro calls were changed to function calls to recover memory. Each change from a macro call to a function call saved two bytes. Also note the drop in V(G) between versions 4 and 5 and the considerable fluctuation in E (Figure 5.4).

In figure 5.5 we can identify an evolution trend similar to the traditional metrics for information flow out of functions (VOUT, UVOUT, VROUT and UVROUT). Interestingly the evolution of metrics that measure inflowing information into functions is very different (VIN, UVIN, VRIN and UVRIN) from the metrics that measure information flow out of functions. They do not increase steadily throughout development and two metrics (VRIN, UVIN) reach their maximum value already in the second version. It suggests that information flow into and out of functions are not measuring the same attributes in program evolution and thus should be treated as two different metrics.

VERSION	DATE	FUNC	n1	N1	n2	N2	Nhat
1	2-Dec-90	223	301	11468	2118	10812	18199
2	13-Feb-91	226	297	11664	2193	11024	18899
3	27-Feb-91	226	295	11683	2191	11029	18872
4	11-Mar-91	246	294	12004	2229	11090	19234
5	23-Apr-91	300	298	12861	2340	11025	20366
6	8-May-91	310	297	13176	2387	11320	20831
7	28-Jun-91	333	289	13656	2433	11725	21247
8	17-Feb-92	353	289	13586	2492	11634	21875
9	6-Mar-92	355	286	13650	2502	11667	21965
10	2-Jun-92	359	292	13840	2507	11784	22074

VERSION	DATE	V	E	VG	LOC	NCSL	FIN	FOUT
1	2-Dec-90	183928	33364792	1070	10577	7451	372	377
2	13-Feb-91	188386	34370032	1135	10755	7425	403	400
3	27-Feb-91	188609	34501848	1134	10791	7420	397	394
4	11-Mar-91	192627	34499792	1144	11458	7613	525	524
5	23-Apr-91	201589	33159330	1092	11978	7605	791	812
6	8-May-91	208130	34760768	1147	12106	7827	813	787
7	28-Jun-91	217276	36516008	1197	12807	8132	871	819
8	17-Feb-92	217460	35628968	1244	13309	8224	926	863
9	6-Mar-92	218388	34464764	1246	13398	8240	933	865
10	2-Jun-92	221555	36357788	1280	13621	8365	953	871

VERSION	DATE	VOUT	VIN	UVOUT	UVIN	VROUT	VRIN	UVROUT	UVRIN
1	2-Dec-90	362	992	134	226	740	1570	276	404
2	13-Feb-91	354	1042	127	238	771	1661	286	436
3	27-Feb-91	354	1056	126	239	768	1670	283	436
4	11-Mar-91	393	1078	137	224	763	1513	289	413
5	23-Apr-91	403	1075	144	232	799	1474	305	427
6	8-May-91	407	1088	148	233	816	1513	310	436
7	28-Jun-91	432	1151	149	234	860	1577	328	445
8	17-Feb-92	439	1073	153	235	847	1515	330	457
9	6-Mar-92	449	1077	158	236	867	1526	339	463
10	2-Jun-92	453	1098	158	242	864	1564	327	467

Table 5.1. Project Overview

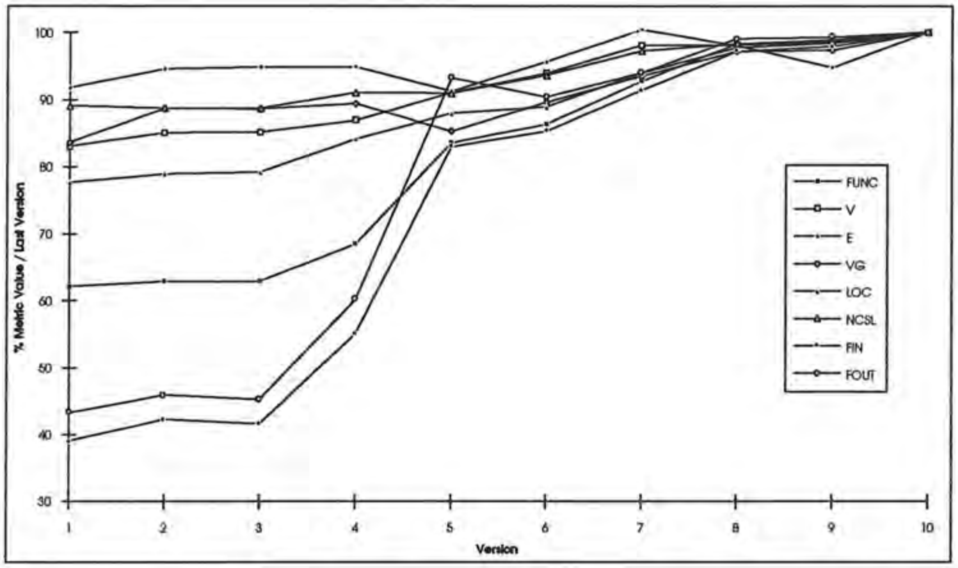

Figure 5.1. Evolution Overview

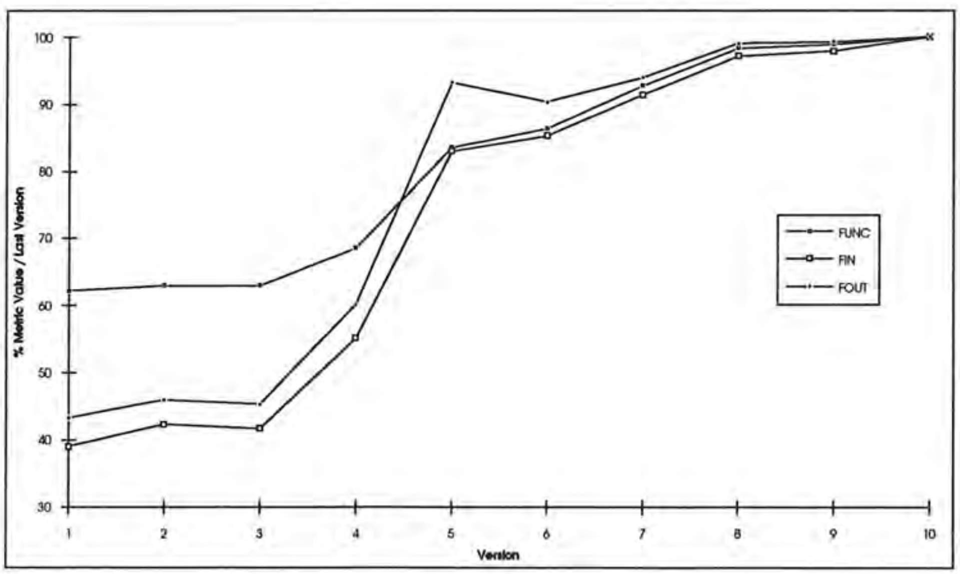

Figure 5.2. Evolution of the Metrics FUNC, FIN, FOUT

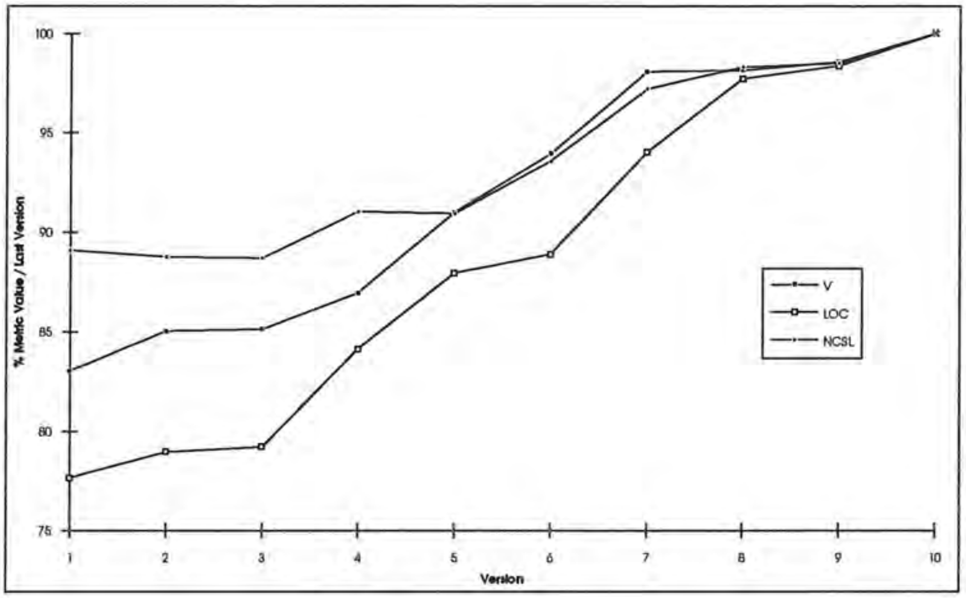

Figure 5.3. Evolution of the Size Metrics V, LOC, NCSL

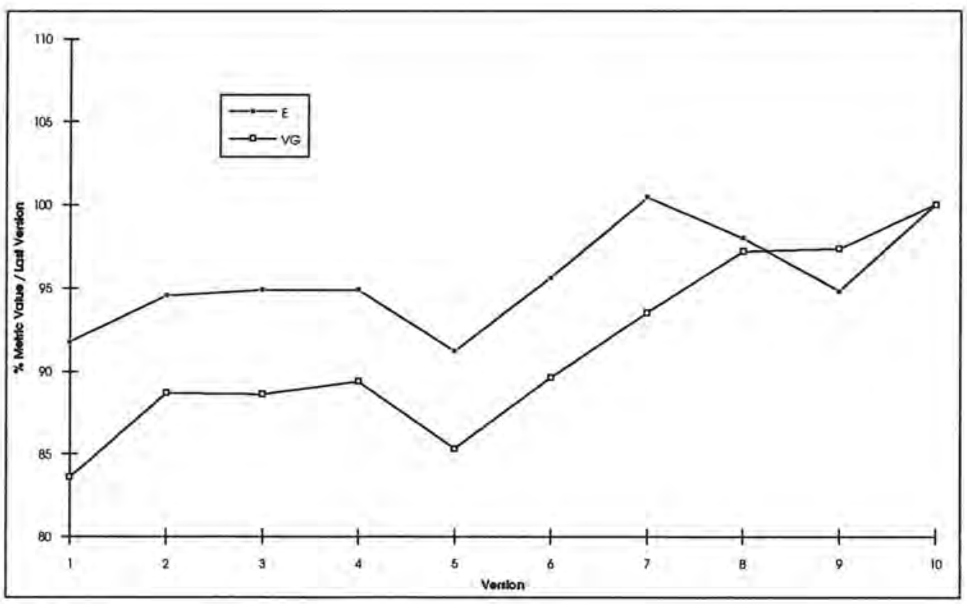

Figure 5.4. Evolution of the Metrics E and V(G)

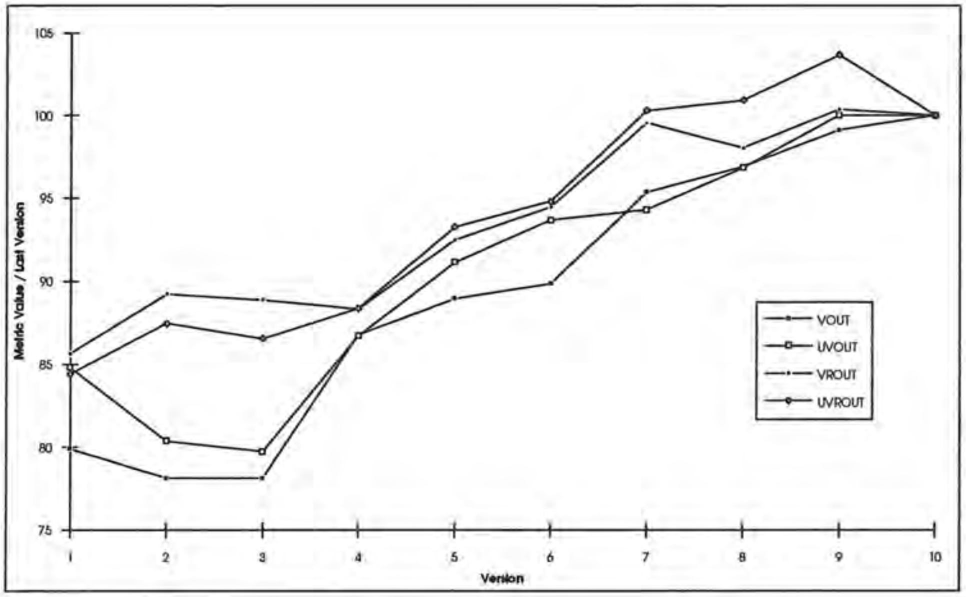

Figure 5.5. Evolution of Information Flow Metrics VOUT, UVOUT, VROUT and UVROUT

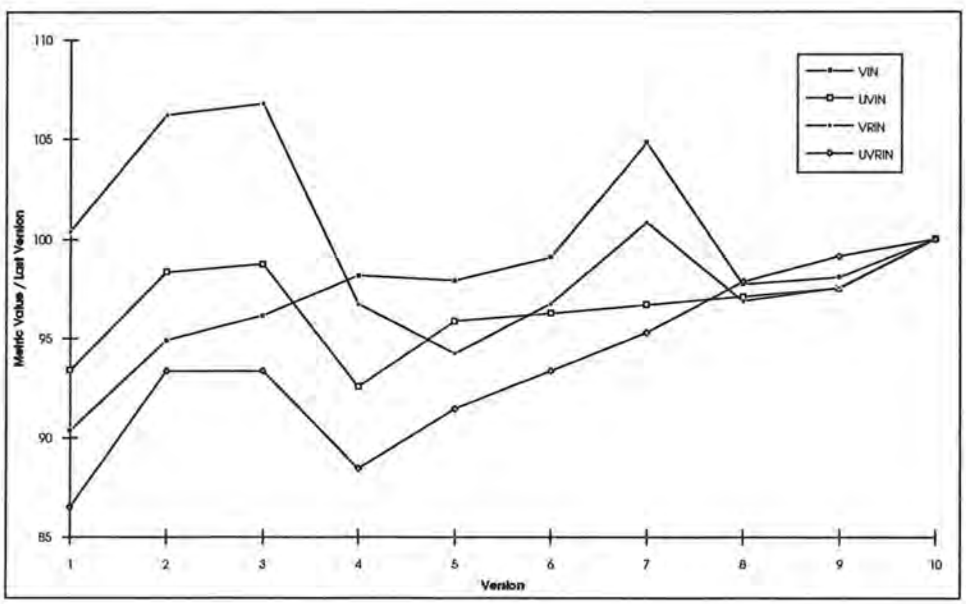

Figure 5.6. Evolution of Information Flow Metrics VIN, UVIN, VRIN and UVRIN

Harrison and Cook [7] based their law on data from only two successive versions. Our data shows that their law holds for 10 successive versions. The number of functions, number of functions changed, new functions added, deleted functions, and number of functions with major change for each version is given in Table 5.2. We define a major change to a function as an increase or decrease in Halstead's V of at least 218, the standard deviation for the volume changes. We selected V because Harrison and Cook used V in their paper although LOC and Halstead's E give the same results. The percentages in the "% MAJOR CHANGES" column are the percent of changes that were major changes. We see that between successive versions, with one exception far less than half of the functions were changed and that between 7% and 22% of the changed functions experienced major change.

VERSION	FUNC	CHANGES	% CHANGES	NEW	DELETED	MAJOR CHANGES	% MAJOR CHANGES	HOURS
1	223	-	-	223	-	-	-	1771
2	226	123	54.4	12	9	12	9.8	339
3	226	14	6.2	0	0	1	7.1	77
4	246	79	32.1	23	3	17	21.5	145
5	300	101	33.7	54	0	16	15.8	190
6	310	25	8.1	12	2	3	12.0	135
7	333	101	30.3	26	3	16	15.8	382
8	353	81	22.9	23	3	7	8.6	92
9	355	24	6.8	2	0	2	8.3	42
10	359	28	7.8	4	0	4	14.3	54
TOTAL	2931	576	19.7	379	20	78	13.5	3227

Table 5.2. Changes Between Successive Versions

Table 5.3 gives the change frequency and major change frequency over the 10 versions. Nearly 60% of the functions were changed at most once. Two functions were changed in all 9 new releases. Over the 10 versions more than 86% of the functions did not undergo a major change. 50 functions accounted for the 78 major changes. Only one function experienced four major changes and 22 functions experienced two or more major changes. Notice that the 379 totals for columns two and four include the 20 deleted functions.

FREQUENCY	CHANGES	%CHANGES	MAJOR CHANGES	% MAJOR CHANGES
0	145	38.3	329	86.8
1	88	23.2	28	7.4
2	61	16.1	17	4.5
3	30	7.9	4	1.1
4	25	6.6	1	0.3
5	13	3.4	0	0.0
6	13	3.4	0	0.0
7	1	0.3	0	0.0
8	1	0.3	0	0.0
9	2	0.5	0	0.0
TOTAL	379	100	379	100

Table 5.3. Frequency of Changes

Table 5.4 gives the total change in V, major change in V, and percentage of the total change in V that was major change between successive versions.

VERSION	VOLUME CHANGE	VOLUME CHANGE MAJOR CHANGES	% VOLUME CHANGE MAJOR CHANGES
1	-	-	-
2	11243	5549	49.4
3	1108	312	28.2
4	8957	5893	65.8
5	15895	13303	83.7
6	2803	2093	74.7
7	11009	8089	73.5
8	7279	2189	30.1
9	1679	333	19.8
10	2829	2012	71.1
TOTAL	62802	39773	63.3

Table 5.4. Changes in Volume Between Successive Versions

Tables 5.2 and 5.4 show that even though only 13.5% of the changes were major changes, the major changes account in average for over 63% of the total change in V over the 10 versions.

We also found the change concentrated in few of the 13 modules in the system. The largest module, TLNUPS, accounted for the bulk of the change. The total number of functions increased by 136 from version 1 to version 10. TLNUPS increased from 91 to 219 functions, an increase of 128. The total system increase in LOC and V between version 1 and 10 is 3,044 and 37,627 respectively; the increase in LOC and V for TLNUPS was 2,966 and 36,498 respectively. There was very little change in the eight smallest modules as the number of functions in versions 1 and 10 are identical and the LOC and V for these functions changed very little.

We also investigated the size characteristics of the changed functions. For each version we evenly divided the functions into four classes on the basis of Halstead's Volume. Class I contained the one-fourth of the functions with the largest V, Class II the one-fourth with the next largest V, and so forth. See Figure 5.7. The functions in

Class I accounted for over 60% of the total change in V for each version. Class II functions accounted for 25% or less. Hence most of the change occurs in the large functions.

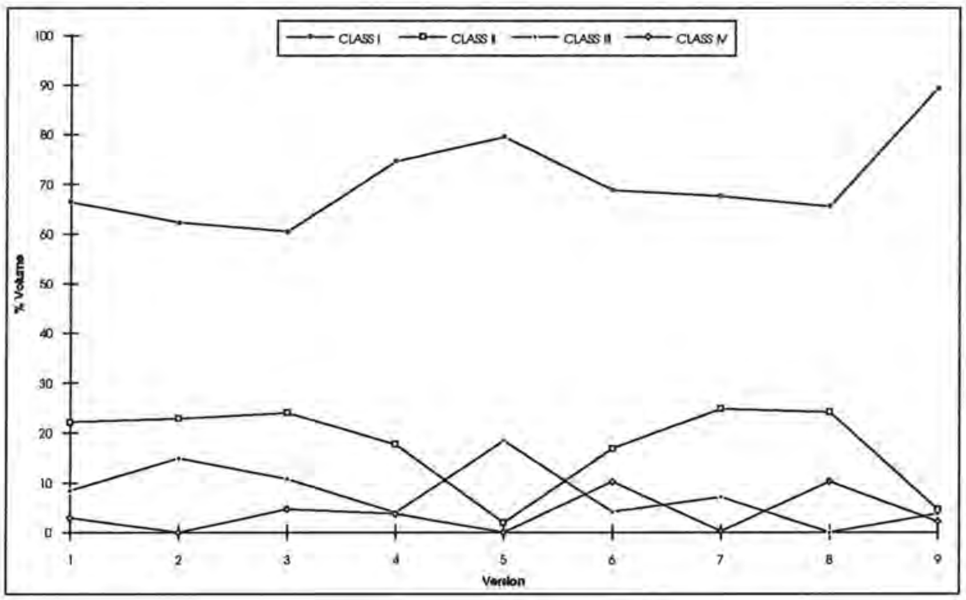

Figure 5.7. Characteristics of Changed Functions

Thus the data does support the findings of Harrison and Cook that the change is not uniform and that most of the total change is concentrated in a small number of functions and modules.

5.2 IDENTIFYING CHANGE PRONE MODULES

When making a change to a program module during program maintenance, a programmer frequently must decide whether to make an isolated change to the module or to completely redesign and rewrite the module. Complete redesign and rewrite is expensive, but it is even more expensive if entropy has taken its toll, e.g. the module structure has seriously deteriorated with severe ripple effects. This is not a simple decision and the wrong choice may have expensive consequences. Completely overhauling a module that will not be modified again may mean delaying or not servicing other maintenance requests. On the other hand, performing a series of isolated changes is wasting resources and just postponing the major overhaul.

Harrison and Cook [7] proposed a maintenance change model to determine whether a given software module can be effectively modified or whether it should be completely redesigned and rewritten. They called a module that is likely to experience significant maintenance changes change-prone. The change-prone classification identifies modules that will undergo significant maintenance activity over the release cycle. Maintenance should be performed differently on change-prone modules than on non-change-prone modules. Namely, change-prone modules should receive an early major overhaul so that the future changes to these modules will be relatively inexpensive.

Unfortunately which modules will become change-prone cannot be predicted. However, the third law of program evolution indicated that changes to a small number of modules accounted for most of the total change. Hence these modules are likely candidates for change prone modules. They measured the cumulative change to a module during program evolution. The decision rule proposed by Harrison and Cook was to establish a change threshold. Once the total change to a module exceeds this threshold it was classified as change-prone and hence should receive a major overhaul. They used Halstead's Volume (V) as the change measure and suggested the threshold value be adjusted to the "risk taking behavior of the manager".

We used V and found that using the volume change standard deviation as a threshold worked well in identifying the few functions that experienced major changes. The 78 function changes it identified as major accounted for 63.3% of the total change in V. We found that 22 of functions were involved in multiple major changes.

Thus we recommend the Halstead volume change standard deviation as a threshold for identifying change prone functions since it evolves with the program changes, is not subject to large fluctuation, and is a relative rather than absolute measure. One should have in mind that if the threshold is chosen to high we may delay recognizing the change prone modules and if it is chosen to low we may classify to many function as change prone. Therefore we were interested in identifying a safe range for the threshold value. It turned out that this value can be easily calculated by considering the volume change of previous versions. Since we used changes of all previous versions the threshold stabilizes when we move on from one to the next release (Table 5.5). Note the big jump between version 4 and 5 due to large volume changes. We also found that the LOC and Halstead's E measures with the change standard deviation as the threshold worked nearly as well as V.

VERSION	STANDARD DEVIATION
2	146
2	142
3	150
4	242
5	243
6	241
7	241
8	220
9	218

Table 5.5. Evolution of the Volume Threshold

6. Software Metrics

In this chapter we study the internal structure of the set of metrics computed with the analyzer tool. In order to understand the relation among metrics, we applied a statistical technique known as factor analysis. Our results show that the 18 metrics map onto four underlying complexity domains: size, information flow into functions, information flow out of functions and control flow. We found that the information flow metrics contribute considerable variation to the factor model. We will show that information flow metrics characterize real-time complexity better than the standard complexity metrics. We also propose new real-time complexity metrics.

6.1 Characteristics of Real Time Software

In a typical real-time application the program continually monitors sensors and upon receiving an input must complete the appropriate processing within a certain fixed time period. This time constraint is the unique feature of real-time software. All of the subtasks that are part of the processing must be scheduled to meet individual timing requirements. Hence real-time programs are characterized by a large amount of monitoring and communication.

Our real-time telephone switching application epitomizes these characteristics. The main application program for all four telephone sets is controlled by one automaton with a total of 51 different states (located in module TLNST) which checks the current state of execution for each process (telephone unit) every 10 ms and takes an action depending on the current state of the process. This action is a call to one or more of the 219 functions in module TLNUPS.

The real-time control part is in modules TLNATM and TLNINT. Incoming and outgoing external signals are controlled by 38 small but complex finite automata located in module TLNATM. Each automaton is served every 1 ms by its scheduler. Events to and from these automata to the main application are also passed via global memory and resource variables. In addition there are 8 interrupt controlled subroutines located in module TLNINT. Their functionality is similar to the automata in module TLNATM.

Thus global and resource variables play a key communication and monitoring role in this application. Each telephone unit has its own global variables. Functions share global data and parameters are passed between functions via global variables and processor registers. The program accesses timers, I/O ports, serial interfaces, interrupt inputs, and special registers through resource variables. This is the reason we developed information flow metrics that counted functions calls and global and resource variables referenced and/or changed.

6.2 Analysis of Software Complexity Metrics

As mentioned in Chapter 4 we computed a variety of standard software complexity metrics: Halstead's Software Science (n1, n2, N1, N2, Nhat, V, E), McCabe's V(G), LOC, and NCSL, and information flow metrics (Unique and total number of global and resource variables changed and/or referenced). Because of the problem with accurately computing indirect references, we omitted the FIN and FOUT metrics from our analysis.

Some of the metrics listed above are primitive and cannot be decomposed further into other metrics. The unique operator count n1, is an example of a primitive metric. Other metrics are non-primitive metrics and composites of other primitive metrics. For instance, the program length N is computed out of the primitive metrics by the sum $N = N1 + N2$. From a statistical perspective it is questionable whether the linear combination of two primitive metrics contribute any new variability in the measurement of program attributes.

Munson and Khoshgoftaar [16, 17, 18] found that in practice most metrics are measuring the same elements of a rather small set of orthogonal complexity domains. They noticed that there are relatively few distinct sources of variation among metrics and that the functional aspects of a large set of metrics can be reproduced by a small set of primitive metrics. They have shown that some metrics do not contribute anything new to the understanding of the differences among programs. Further, adding metrics that are already represented by other metrics is likely to introduce a noise component to the underlying model. In order to examine the basic sources of variation in a set of metrics Munson and Khoshgoftaar applied factor analysis. They have shown that many sets of software complexity metrics map onto less than six underlying complexity domains. All of the existing metrics appear to be representable as linear combinations of these few factor domains.

Since we introduced two new sets of information flow metrics (global and resource variables referenced and changed), we were interested in whether these metrics are measuring something not measured by the traditional metrics. In addition, we investigated how many different complexity domains are present in our data.

We computed the correlations of all of the metrics for all functions for the latest version of the program. A grouping of metrics by highest correlation partitioned the metrics into three groups: traditional metrics (Table 6.1), global and resource variables changed (Table 6.2), and global and resource variables referenced (Table 6.3). Metrics in each group are highly correlated with each other and have smaller correlation with metrics in the other groups. Note that n1, V(G) and E in Table 6.1 have noticably smaller correlation with the other metrics. They are placed in Table 6.1 because they have a higher correlation with the metrics in Table 6.1 than with the information flow metrics in Table 6.2 and Table 6.3. In order to understand the basic sources of variation in our set of metrics, a statistical technique known as factor analysis is used.

METRIC	n1	N1	n2	N2	Nhat	V	E	VG	LOC	NCSL
n1	1.00									
N1	0.47	1.00								
n2	0.26	0.78	1.00							
N2	0.36	0.93	0.94	1.00						
Nhat	0.28	0.76	0.99	0.94	1.00					
V	0.33	0.93	0.93	0.99	0.94	1.00				
E	0.50	0.86	0.54	0.76	0.54	0.76	1.00			
VG	0.63	0.53	0.25	0.38	0.23	0.37	0.63	1.00		
LOC	0.39	0.83	0.92	0.93	0.90	0.91	0.70	0.47	1.00	
NCSL	0.52	0.90	0.85	0.92	0.83	0.90	0.81	0.64	0.95	1.00

Table 6.1. Correlations Traditional Metrics

METRIC	VOUT	UVOUT	VROUT	UVROUT
VOUT	1.00			
UVOUT	0.88	1.00		
VROUT	0.76	0.78	1.00	
UVROUT	0.55	0.69	0.90	1.00

Table 6.2. Corellations Outflowing Information

METRIC	VIN	UVIN	VRIN	UVRIN
VIN	1.00			
UVIN	0.84	1.00		
VRIN	0.92	0.76	1.00	
UVRIN	0.75	0.85	0.86	1.00

Table 6.3. Correlations Inflowing Information

6.2.1 The Exploratory Factor Analysis Technique

Of various approaches for studying the internal structure of a set of indicators factor analysis is probably most powerful [20]. Factor analysis refers to a family of analytic techniques designed to identify factors, or dimensions, that underlie the relations among a set of observed variables. The observed variables are the indicators presumed to reflect the construct, i.e. factors. Factor analysis is usually applied to the correlations among indicators. An estimate of the relation between each indicator and a factor - referred to as a factor loading - is obtained. A factor loading is the weight of an indicator on the factor. Generally speaking, the higher the factor loading, the more meaningful it is, or the greater is the impact of the factor on the indicator. A factor loading may vary from zero (no relation between the indicator and the factor) to plus or minus one (perfect relation between the indicator and the factor). The square of such a factor loading indicates the proportion of variance of a given indicator accounted for by the factor. For example, a loading of .4 means that .16 ($.4^2$) , or 16% of the variance of the indicator is accounted for by the factor. Complexity metrics with similar aspects of variability will tend to have high factor loadings on a single factor and are thus associated with the underlying complexity domain represented by the factor [20].

In the following data analysis an exploratory factor analysis is used. Exploratory factor analysis is concerned with the question of how many factors are necessary to explain relations among a set of indicators and with the estimation of the factor loadings. The essential purpose of this technique is to describe the covariance relationship among variables in terms of a few underlying, but understandable, random quantities.

Factor analysis can be considered as an extension of principal component analysis [20]. Both can be viewed as attempts to approximate the covariance matrix, Σ. However, the approximation based on the factor analysis model is more elaborate. The primary question in factor analysis is whether the data is consistent with a prescribed structure. In the case of complexity metrics, this structure represents orthogonal complexity domains. That is, many existing complexity metrics map onto a reduced set of orthogonal complexity measures.

To simplify interpretation of the extracted factor loadings new common factors can be found through orthogonal rotation of the factor structure. The process of orthogonal factor rotation produces a set of new factors that also satisfy the factor model. Many different techniques are used for these orthogonal rotations. To rotate factors orthogonally, means to rotate them so that they remain at right angles to each other and that variables or vectors that are orthogonal are not correlated. By far the most widely used orthogonal rotation is the varimax rotation. Varimax is aimed at maximizing variances of the factors. Only a subset of factors from the original pattern is chosen for rotation. The selection of factors for varimax rotation is generally based on the factor's eigenvalue [20].

6.2.2 Results of The Principal Component Factor Analysis

Figure 6.1 is a plot of the eigenvalues λ in descending order of magnitude - referred to as a scree plot [2, 20]. The scree plot is an aid to determine the number of factors to be retained. Cattell [2] suggested that the plot of the λ's be examined to identify a clear break between large λ's and small ones. Considering factors with small λ's as trivial, Cattell labeled this criterion for the number of factors to be retained as a scree test. Others suggest using the criterion of eigenvalues λ larger than one.

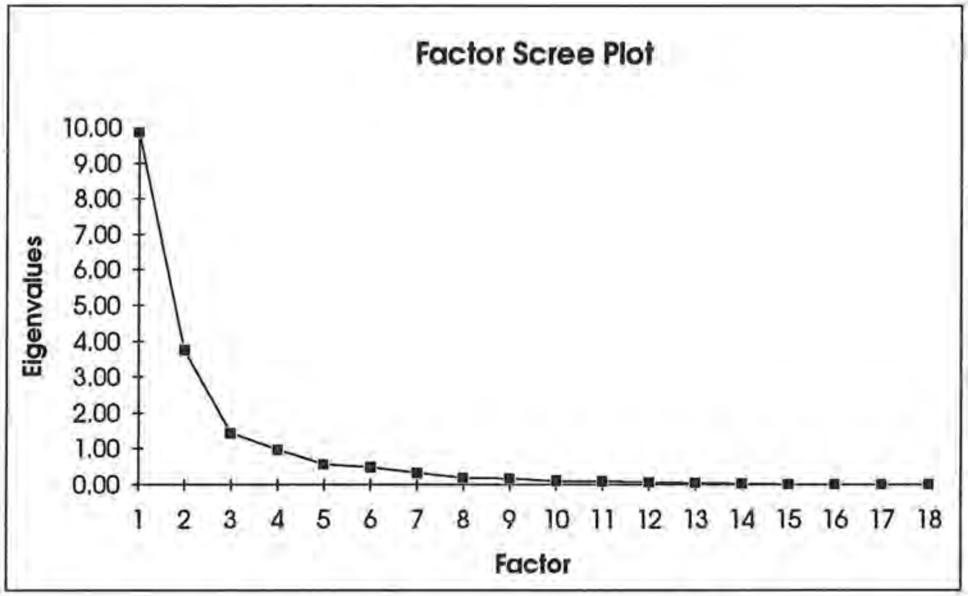

Figure 6.1. Factor Scree Plot

Table 6.4 shows the results of the principal component factor analysis. We used a varimax rotation on the original factor structure and selected four factors for rotation. In the scree plot we can identify a break, akin to an elbow, between the fourth and the fifth eigenvalue. The latter, trivial eigenvalues appear to lie on a horizontal line and are not considered. The last two rows in Table 6.4 contain the eigenvalues and the amount of variance explained by each factor in the rotated factor domain, respectively. The four factors of Table 6.4 account for 88.8% of the total amount of variance explained by the original set of metrics.

METRIC	FACTOR 1	FACTOR 2	FACTOR 3	FACTOR 4
n2	**0.973**	0.142	0.068	-0.054
NHAT	**0.970**	0.111	0.055	-0.046
V	**0.962**	0.107	0.077	0.184
N2	**0.959**	0.140	0.099	0.186
LOC	**0.913**	0.157	0.146	0.202
NCSL	**0.841**	0.262	0.155	0.401
N1	**0.822**	0.171	0.158	0.414
UVIN	0.163	**0.924**	0.179	0.065
UVRIN	0.224	**0.863**	0.201	0.177
VIN	0.166	**0.794**	0.357	0.330
VRIN	0.204	**0.765**	0.338	0.414
UVROUT	0.130	0.101	**0.913**	-0.022
VROUT	0.142	0.189	**0.906**	0.229
UVOUT	0.081	0.437	**0.799**	0.173
VOUT	0.084	0.363	**0.725**	0.330
VG	0.201	0.354	0.182	**0.792**
E	0.608	0.131	0.157	**0.662**
n1	0.183	0.364	0.338	**0.587**
EIGENVALUES	9.852	3.744	1.433	0.963
% VARIANCE	54.7	20.8	8.0	5.4

Table 6.4. Varimax Factor Analysis

It is interesting that most of the traditional metrics (n2, Nhat, V, N2, LOC, NCSL, N1) are associated with the first factor. In particular all size metrics are grouped into factor one. Hence, factor one represents the size complexity domain. Many of the traditional complexity metrics are members of this factor. This suggests for prediction purposes that combinations of members of the size domain would be just as good as simple size measures such as lines of code.

The second factor contains all metrics where global and resource variables are referenced (UVIN, UVRIN, VIN, VRIN). This factor measures the inflowing information into program modules.

The third factor consists solely of metrics that change global and resource variables (VROUT, UVROUT, UVOUT, VOUT). Similar to the previous factor the third factor measures the outflowing information.

Finally the fourth factor contains McCabe's V(G), Halstead's E and n1. The factor loading is highest for McCabe's V(G). It is conceivable that this factor represents control flow of the program.

We should mention that our results differ from those reported by Munson and Khoshgoftaar [16, 17, 18]. From their factor analysis of several metric data sets they found McCabe's V(G), and Halstead's E were placed together into the size factor. Analysis of our data separated McCabe's V(G), Halstead's E and n1 into a factor domain distinct from the size factor. However, some of this difference may be explained by there was only one information flow metric in two of their data sets and we considered six information flow metrics.

The factor analysis revealed that the traditional software metrics contribute about one half of the total variance in the set of metrics we applied to our data. Both sets of information flow metrics are associated with their own complexity domain and account for almost 29% of the total variance. The observations indicate that information flow into and out of functions is an important aspect in real-time software systems. Further, information flow into and out of functions are radically different. The results suggest to treat them separately and not combining information flow into one compound metric.

6.2.3 Relative Complexity Metrics

The initial objective of the factor analysis was to achieve a reduction in dimensionality of the problem. In addition, Munson and Khoshgoftaar [16] defined a relative complexity metric, that can be computed out of the factor score coefficient matrix given by the factor analysis. The factor score coefficient matrix **F** is constructed to send an associated matrix of standardized complexity metrics, **z**, onto the underlying orthogonal factor dimensions. The factor score coefficient matrix for mapping the original 18 metrics onto four orthogonal factor domains is shown in Table 6.5.

METRIC	FACTOR 1	FACTOR 2	FACTOR 3	FACTOR 4
n2	0.208	0.047	0.012	-0.248
NHAT	0.207	0.031	0.010	-0.230
V	0.170	-0.038	-0.012	-0.032
N2	0.168	-0.025	-0.009	-0.042
LOC	0.155	-0.028	0.008	-0.032
NCSL	0.105	-0.024	-0.038	0.121
N1	0.102	-0.077	-0.020	0.159
UVIN	-0.006	0.445	-0.111	-0.245
UVRIN	-0.008	0.375	-0.104	-0.145
VIN	-0.043	0.266	-0.037	-0.008
VRIN	-0.047	0.230	-0.053	0.069
UVROUT	0.027	-0.129	0.426	-0.202
VROUT	-0.013	-0.147	0.367	-0.010
UVOUT	-0.025	0.027	0.273	-0.100
VOUT	-0.046	-0.041	0.229	0.067
VG	-0.093	-0.064	-0.101	0.530
E	0.020	-0.158	-0.049	0.418
n1	-0.066	-0.035	0.003	0.331

Table 6.5. Factor Score Coefficient Matrix

From these new orthogonal measures of program complexity Munson and Khoshgoftaar [16] derived a relative complexity metric C_r. For each function the raw data vector is converted to a new standard score vector **z** as follows:

$$z\,score = \frac{metric\,value - \mu}{\sigma}$$

Table 6.6 shows the mean and standard deviation for the original set of metrics respectively.

METRIC	AVERAGE	STDEV
n1	10.40	5.74
N1	38.55	67.09
n2	32.59	28.87
N2	32.59	65.71
Nhat	106.32	247.74
V	391.31	989.94
E	8719.89	25028.25
VG	3.57	4.93
LOC	37.25	51.17
NCSL	23.28	32.20
VOUT	1.26	2.65
VIN	3.06	5.72
UVOUT	0.79	1.35
UVIN	1.55	2.70
VROUT	2.41	4.70
VRIN	4.36	7.61
UVROUT	1.58	3.25
UVRIN	2.42	3.75

Table 6.6. Metric Means and Standard Deviations

Then, for each data vector a new vector of factor scores, **f**, is calculated:

$$f = zF$$

The relative complexity, C_r, of the factored program modules is represented as follows:

$$C_r = zF\lambda^T = f\lambda^T$$

where λ is a vector of eigenvalues associated with the specific factor dimensions. From the vector C_r of relative complexity metrics, the i^{th} entry C_r, represents the relative complexity of the i^{th} program module.

The relative complexity metric C_r is normally distributed with a mean of zero and a variance of:

$$V(C_r) = \sum_{i=1}^{j} \lambda_i^2$$

where j represents the number of factors in the rotated factor pattern, and λ_i is the eigenvalue associated with the i^{th} factor. The relative complexity metric represents each raw complexity metric in proportion to the amount of unique variation contributed by that complexity metric.

Munson and Khoshgoftaar [16] suggested a scaled version of the relative complexity metric and defined it as follows:

$$C_r = \frac{10\, C_{ri}}{\sqrt{V(C_r)}} + 50$$

The scaled metric has a mean of 50 and a standard deviation of 10.

We computed factor scores and the relative complexity values for all 359 function of the last version of the program. Each relative complexity value is a unitary measure of program complexity. Table 6.7 shows some sample relative complexity values for the least, average and most complex modules. Figure 6.2 shows the distribution of relative complexity for all 359 modules sorted by size. It is interesting to note that there are some functions with very high relative complexity value. It suggests that these modules are extreme outliers and hence should be watched carefully in the development process. There are no outliers with exceptionally low complexity value.

FUNCTION	Cr' (LOW)	FUNCTION	Cr' (AVERAGE)	FUNCTION	Cr' (HIGH)
275	44.06	59	49.88	272	69.47
279	44.06	27	49.89	251	70.46
280	44.06	139	49.89	262	74.72
281	44.06	10	49.90	235	80.42
336	44.16	142	49.91	236	84.57
282	44.22	291	50.05	257	85.37
284	44.22	226	50.15	126	89.43
286	44.22	19	50.15	317	90.78
96	44.23	135	50.15	353	99.71
130	44.23	263	50.55	321	174.66

Table 6.7. Sample Relative Complexity Values

Four out of the ten functions with highest relative complexity value are real-time functions. Two functions (321, 353) contain data tables, are large in size and reference many gloabl variables. The reset function is also one of the functions with high relative complexity. The remaining three functions are fairly complex states of the overall control automata.

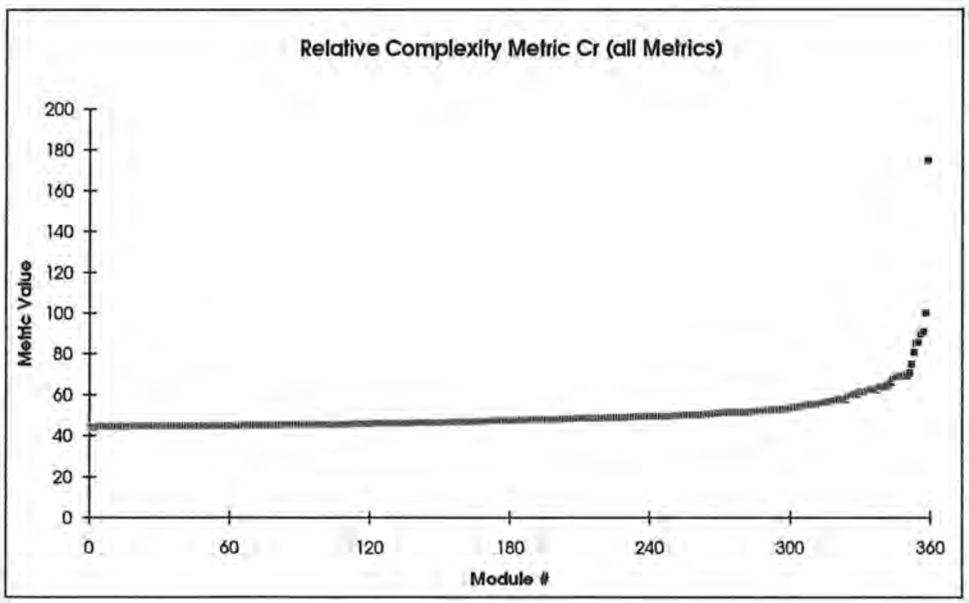

Figure 6.2. Distribution of Relative Complexity Metrics

6.2.4 METRICS REDUCTION

As mentioned earlier additional metrics appear to be contributing nothing new in the understanding of the differences among programs. It is likely that a subset of all metrics is sufficient to capture the same attributes of the program that are described by the original set of metrics.

For each complexity domain the metric with the highest strength of association was selected for a second factor analysis. Table 6.4 shows that Halstead's n2, unique variables referenced UVIN, unique global variables and resources changed UVROUT and McCabe's V(G) have the highest factor loading in its domain, respectively. Again the factor analysis was performed and the four factors n2, UVIN, UVROUT and V(G) were extracted out of the entire data set. Table 6.8 shows the matrix of the rotated factor loadings for the subset of four metrics.

METRIC	FACTOR 1	FACTOR 2	FACTOR 3	FACTOR 4
n2	**0.983**	0.087	0.105	0.127
UVROUT	0.087	**0.981**	0.116	0.129
VG	0.110	0.121	**0.965**	0.205
UVIN	0.135	0.137	0.209	**0.959**
EIGENVALUES	1.873	0.803	0.758	0.566
% VARIANCE	46.8	20.1	18.9	14.1

Table 6.8. Varimax Factor Analysis for 4 Factors

Using the factor score coefficient matrix the new relative complexity values, $C'_{r(new)}$, for the reduced set of metrics was computed. The new relative complexity values were then pairwise compared with the set previously obtained. The Spearman's correlation coefficient for ranked data $r_s = .95$ shows that the two sets of complexity metrics are highly correlated .

It can be concluded that a reduced set of only four metrics, one for each factor, is measuring everything that is given by the entire set of metrics. It indicates that the functional aspect of a large set of metrics can be reproduced entirely by a smaller set of metrics. If a metric is used in a multivariate model and its variance is already represented by other metrics, there is no need to assess additional metrics out of the same complexity domain.

6.3 Real-Time Software Complexity Metrics

One basic question we addressed is whether the software metrics for the real-time modules are different from the non-real-time modules. The factor analysis in the previous section suggested that there may be a difference.

The program was designed so that two modules (TLNST, TLNUPS) of the thirteen modules contain most of the program functionality and two modules (TLNINT, TLNATM) are responsible for most of the real-time control. These four modules include 88% (316 of 359) of the program functions and 82% of the non-comment source lines (NCSL).

Hence for our comparison of real-time and non-real-time modules we selected these four modules. The metric values for these four modules and the metric values normalized by the number of functions are given in Tables 6.9 and 6.10. Notice that each of the information flow metrics (VOUT, VIN, UVOUT, UVIN, VROUT, VRIN, UVROUT, UVRIN) in Table 6.10 are substantially higher for the two real-time modules than the non-real-time modules. For the other metrics (Halstead's Software Science, V(G), LOC, NCSL) TLNATM is highest in all instances and the results are mixed for the other three modules.

MODULE	n1	N1	n2	N2	Nhat	V	E	VG	LOC	NCSL
NON-REALTIME MODULES										
TLNUPS	0.3	24	2.3	20	22	402	117570	2.4	27	15
TLNST	0.7	39	5.5	32	49	591	63186	5.5	40	28
REALTIME MODULES										
TLNATM	1.1	88	10.4	64	96	1331	165941	10.2	64	50
TLNINT	2.5	32	12.1	32	91	440	11559	4.3	42	24

Table 6.9. Traditional Metrics Normalized by Number of Functions

MODULE	VOUT	VIN	UVOUT	UVIN	VROUT	VRIN	UVROUT	UVRIN
NON-REALTIME MODULES								
TLNUPS	1.0	2.5	0.2	0.2	1.7	3.2	0.4	0.5
TLNST	0.8	2.7	0.2	0.4	1.8	3.7	0.6	0.9
REALTIME MODULES								
TLNATM	4.0	7.9	1.8	2.3	8.1	13.2	3.3	4.3
TLNINT	2.3	5.8	1.6	4.4	3.9	7.9	2.9	6.1

Table 6.10. Information Flow Metrics Normalized by Number of Functions

These results suggest that the major difference between real-time and non-real-time functions is the real-time functions have a higher average information flow. We feel that the average in and out information flow is a good measure of real-time complexity. For example the sum of UVOUT + UVIN for each function in a module divided by the number of functions in the module. We realize that this is a preliminary result, but it does agree with our intuition that real-time modules have a the heavy information flow into and out of functions.

7. PROGRAM EFFORT ANALYSIS

A final goal of our research is to relate programmer effort to program changes. For this part of our study we obtained the number of programmer hours per day for programmers who worked on the program. The last column in Table 5.2 gives the hours worked between successive versions. Table 7.1 shows the correlations between hours worked and the number of changes, number of major changes, the total change in V, and the major change in V for successive versions. Hours worked has the highest correlation with the number of changes.

CORRELATION	CHANGES	MAJOR CHANGES	VOLUME CHANGE	VOLUME CHANGE OF MAJOR CHANGES	HOURS
CHANGES	1.00				
MAJOR CHANGES	0.86	1.00			
VOLUME CHANGE	0.92	0.91	1.00		
VOLUME CH. MAJOR CHANGES	0.75	0.87	0.94	1.00	
HOURS	0.80	0.70	0.70	0.61	1.00

Table 7.1. Correlation Between Changes and Programming Hours

We also compared hours worked between successive versions and changes in each of the metrics for successive versions. Table 7.2 shows that highest correlations were for VROUT (.80), N2 (.73), and V (.69).

METRIC	CORR.
FUNC	0.27
n1	0.54
N1	0.45
n2	0.53
N2	0.73
Nhat	0.47
V	0.69
E	0.22
VG	0.63
LOC	0.45
NCSL	0.38
FIN	0.19
FOUT	0.13
VOUT	0.36
VIN	0.48
UVOUT	0.12
UVIN	0.26
VROUT	0.80
VRIN	0.38
UVROUT	0.58
UVRIN	0.56
HOURS	1.00

Table 7.2. Correlation Between Hours Worked and Metric Changes

It should be noted that although the hours worked includes time spent on both existing and new functions, the change data is only for changes to existing functions. We attempted to seperate the effort spent on new functions. We used V of the first version divided by the number of hours as the productivity rate for new functions. To compute the hours for new functions for each version we devided the total V for new functions by the productivity rate. The corrected hours for each version is the total hours minus the hours for new function. Figure 7.1 shows programming hours and corrected programming hours for all versions.

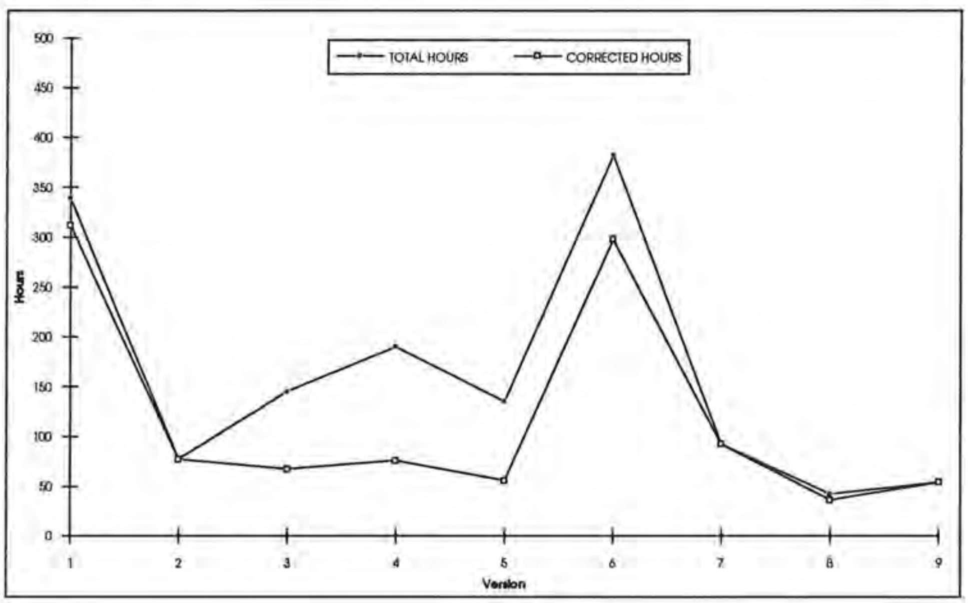

Figure 7.1. Programming Hours and Corrected Programming Hours

However, when we used the corrected hours between successive versions, we obtained lower correlations with the change data (see table 7.3).

CORRELATION	CHANGES	MAJOR CHANGES	VOLUME CHANGE	VOLUME CHANGE OF MAJOR CHANGES	HOURS
CHANGES	1.00				
MAJOR CHANGES	0.86	1.00			
VOLUME CHANGE	0.92	0.91	1.00		
VOLUME MAJOR CHANGE	0.75	0.87	0.94	1.00	
CORRECTED HOURS	0.73	0.49	0.51	0.35	1.00

Table 7.3. Correlation Between Changes and Corrected Programming Hours

It should be pointed out that the effort analysis is based on only 10 versions. Moreover, maintenance includes a variety of different change activities like correcting errors, adding functionality and adapting to a changed environment. Depending on the maintenance task, some modifications can be done quickly but cause substantial change in metric counts. Other tasks need more time and do not affect software metrics as much. For example finding an error can take a long time but fixing it may affect only one line of code and some metrics will not change at all. On the other hand a change of the program structure usually causes large changes in metric counts but it may not take as much effort. Hence, effort analysis is sensitive to the type change made between successive versions in program maintenance.

8. CONCLUSIONS

Our analysis of the ten versions of the embedded real-time software show that they obey the laws of software evolution and agree with our intuition that the information flow metrics seem to measure software complexity. We found that the data also supports Harrison and Cook's [7] program maintenance decision model and proposed the change standard deviation in Halstead's V as a threshold for their model. It is also interesting that LOC and Halstead's E measures with the change standard deviation as a threshold worked nearly as well as Halstead's V. However more studies must be performed before reliable decision rules for threshold values can be established.

We have found relatively few distinct sources of variation among the set of metrics when applied to the actual software system. The entire set of 18 metrics map onto only four underlying complexity domains: size, information flow into functions, information flow out of functions and control flow. While there are now hundreds of metrics available to measure all sorts of program attributes, we would expect that factor analysis would map these hundreds of metrics onto a small number of complexity domains, probably not more than 10. Other metrics will probably map into one of the four factors we found in our data and will not constitute a new complexity domain.

We were surprised that the factor analysis lumped most of the traditional metrics into two factors because other factor analysis studies of metrics have partitioned the traditional metrics into three or more factors. We were also mildly surprised that the global and resource variables referenced were grouped into a second factor and the global and resource variables changed into a third factor. Other factor analysis studies have grouped the information flow metrics with other traditional metrics in one factor. We found that the information flow metrics account for almost 29% of the variance and hence are an important complexity class that has to be considered in real-time systems. In view of our findings it is surprising that few information flow metrics were computed in the other studies.

We think that the relative complexity metric, C_r', is a reasonable measure to identify very complex parts of a program. However we feel that some information is lost when calculating a composite metric out of primitive metrics. We believe that further insights into why some functions are more complex than others can be obtained when considering the metric values of each domain separately. For instance, it is conceivable that a function with extreme high information flow is ranked average in complexity by the relative complexity metric C_r' because it has relative few lines of code. In this case the composite metric hides valuable information about the function. We would recommend using outliers in each domain as a method in identifying complex functions.

We were disappointed that we did not find a strong relation between the hours worked and changes, amount of change, and our metrics. We had hoped to discover a formula that would predict the hours worked based on the total change and new functions added. However, we based our effort analysis on only 10 versions and detailed information on software maintenance was not available.

We would have liked to investigate the relation between error data and metric counts. However the telecommunication firm has just begun to collect error data. In particular we expect a high error rate in complex parts of the program which would have great implications on testing and software maintenance.

Our future intention is to assist in the establishment of a metric program in the company where the software was developed and maintained. We like to encourage to use our metric tool to collect and analyze data. Correlation of the metrics and error data will help them to identify error-prone modules and in allocating testing resources. We are certain that the use of software metrics benefits the development of high quality software and are necessary to be successful in the future in a highly competitive market.

9. REFERENCES

[1] Belady, L. A. and M. M. Lehman, "A Model of Large Program Development", IBM Systems Journal, 15, 3 (1976), pp, 225-252.

[2] R. B. Cattell, The Scree Test for the Number of Factors. Sociological Methods & Research, 1 (1966), pp. 245-276.

[3] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, Software Engineering Metrics and Models. Benjamin/Cummings, Menlo Park, California, 1986.

[4] C. R. Cook, "Software Complexity Measure", Proc. 1984 Pacific Northwest Software Quality Conference, Portland, Oregon, 1984, pp. 343-363

[5] C. R. Cook, A. Roesch, "Real-Time Software Metrics", Annual Oregon Workshop on Software Metrics, Silver Falls, Oregon, 1993.

[6] M.H. Halstead, "Elements of Software Science", Elsevier, 1977

[7] W. Harrison and C. Cook, "Insights on Improving the Maintenance Process Through Software Measurement", Proceeding Conference on Software Maintenance, San Diego, 1990, pp. 37-45.

[8] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, September 1981, pp. 510-518.

[9] D. C. Howell, Statistical Methods for Psychology, Boston, Massachusetts, 1982.

[10] P. A. Laplante, "Design Issues in Real-Time", IEEE Potentials, Feb. 1993, pp. 24-26.

[11] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", Proceedings of the IEEE, 68:9 (September 1980), pp. 1060-1076.

[12] T. McCabe, "A Complexity Measure", IEEE Transaction on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320.

[13] Mitsubishi, "MELPS 740 Series Structered Relocatable Macro Assembler", User's Manual, MITSUBISHI ELECTRIC Semiconductor Software Corporation, 1989.

[14] Mitsubishi, "M37450M2-XXXSP/FP User's Manual", MITSUBISHI ELECTRIC Semiconductor Software Corporation, 1989.

[16] J. C. Munson and T. M. Khoshgoftaar, "Applications of a Relative Complexity Metric for Software Project Management", Journal Systems and Software, New York, 1990, pp. 283-291.

[17] J. C. Munson and T. M. Khoshgoftaar, "Some Primitive Control Flow Metrics", Proceedings Third Annual Oregon Workshop on Software Metrics, Portland, Oregon, 1991.

[18] J. C. Munson and T. M. Khoshgoftaar, "The Dimensionality of Program Complexity", Proceedings 11th International Conference on Software Engineering, Pittsburgh, 1989, pp. 245-253.

[19] P. Oman and C. Cook, "Design and Code Traceability", The Journal of Systems and Software", Vol 12 (3), July 1990.

[20] E. J. Pedhazur and L. Pedhazur Schmelkin, Measurement, Design and Analysis: An integrated Approach. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1991.

APPENDIX A - ANALYZER SAMPLE OUTPUT

In this section the output of the metric analyzer on the sample input file *test.txt* (provided on disk) is given.

A printout of the sample input file *test.txt* follows:

```
;***********************************************************************
;       GLOBAL DEFS
;***********************************************************************

VAR1    .blkb 1               some byte variables
VAR2    .blkb 1
INT1    = 0,IRQ1              some resource bit flags
INT2    = 0,IRQ2

;*************************************************************************
;       FUNCTION TEST1
;***********************************************************************

        .FUNC TEST1
TEST1   IF [INT2] == 1
          JSR TEST2
        ENDIF

        A = 0
        Y = 0
        JSR TEST2

        IF [INT1] == 0
          [VAR1] = 0
        ENDIF
        .ENDFUNC TEST1

;***********************************************************************
;       FUNCTION TEST2
;***********************************************************************

        .FUNC TEST2
TEST2   IF [INT1] == 1
          [VAR1] = $FF
        ENDIF

        IF [INT2] == 0
          [VAR2] = [VAR1]
        ENDIF
        .ENDFUNC TEST2
```

To run the analyzer on the input file *test.txt* type the following:

metric test.txt > testout

To suppress the report of individual functions use the *-m* option (see also chapter 4). The output of the metric analyzer is redirected into the file *testout.* This file can be easily read into a standard spreadsheet application. A formatted output is given below:

NAME	n1	N1	n2	N2	Nhat	V	E	VG	LOC	NCSL	FIN	FOUT	VOUT	VIN	UVOUT	UVIN	VROUT	VRIN	UVROUT	UVRIN
TEST.TXT	11	38	13	36	86	339	5168	6	40	23	2	2	3	1	2	1	3	5	2	3
TEST1	10	22	12	25	76	210	2183	3	18	11	0	2	1	0	1	0	1	2	1	2
TEST2	8	16	8	11	48	108	594	3	9	8	2	0	2	1	2	1	2	3	2	3

Figure A1. Sample Output

APPENDIX B - PROGRAM LISTING

FILE METRIC.L:

```
/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 *
 * This code is derived from software contributed to Berkeley by
 * Vern Paxson.
 *
 * The United States Government has rights in this work pursuant
 * to contract no. DE-AC03-76SF00098 between the United States
 * Department of Energy and the University of California.
 *
 * Redistribution and use in source and binary forms are permitted provided
 * that: (1) source distributions retain this entire copyright notice and
 * comment, and (2) distributions including binaries display the following
 * acknowledgement:  ``This product includes software developed by the
 * University of California, Berkeley and its contributors'' in the
 * documentation or other materials provided with the distribution and in
 * all advertising materials mentioning features or use of this software.
 * Neither the name of the University nor the names of its contributors may
 * be used to endorse or promote products derived from this software without
 * specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */

 #include <string.h>
 #include <stdio.h>

 #include "metric.h"
 #include "hash.h"

 // external variables
 extern struct hash_slot far *hash_table;
 extern struct metric_struct metric;
 extern int passtwo;
 extern int error;

 // initalizations
 int first_func = TRUE;
 int lookup_func_name = FALSE;
 int lookup_compound_statement = FALSE;
 int lookup_bytevar = FALSE;
 int lookup_bitvar = FALSE;
 int lookup_equate = FALSE;
 int lookup_mod_var = FALSE;
 int lookup_quote = FALSE;
 int inhibit_lookup_var = FALSE;

```

```
  int slot;

noteol          [^\n]
ws              [ \t]
identifier      [0-9A-Za-z_.?]+
comment         ;{noteol}*
bit             [01234567]
constant        [%@$O]?[0-9A-F]+[BOQH]?

%s              PASSTWO
%%
                // introduce pass two
                if (passtwo)
                  BEGIN(PASSTWO);


<INITIAL>{comment} { /* eat up comments */ }

<INITIAL>(JSR|JMP|BBC|BBS|BCC|BCS|BEQ|BMI|BNE|BPL|BVC|BVS) {
                  lookup_func_name = TRUE;
                }

<INITIAL>{identifier}{ws}*("="|".EQU"){ws}*{bit}(",") {
                  lookup_bitvar = TRUE;

                  REJECT;
                }

<INITIAL>{identifier}{ws}*(".blkb") {
                  lookup_bytevar = TRUE;

                  REJECT;
                }

<INITIAL>^{identifier}{ws}*("="){ws}*{constant} {
                  lookup_equate = TRUE;

                  REJECT;
                }

<INITIAL>{identifier} {
                  if (lookup_func_name)
                  {
                    hash_insert_token(hash_table, strupr(yytext), FUNCTION);
                    lookup_func_name = FALSE;
                  }

                  if (lookup_bytevar)
                  {
                    hash_insert_token(hash_table, strupr(yytext), BYTEVAR);
                    lookup_bytevar = FALSE;
                  }

                  if (lookup_bitvar)
```

```
                    {
                      hash_insert_token(hash_table, strupr(yytext), BITVAR);
                      lookup_bitvar = FALSE;
                    }

                    if (lookup_equate)
                    {
                      hash_insert_token(hash_table, strupr(yytext), EQUATE);
                      lookup_equate = FALSE;
                    }
                  }

<INITIAL>\n     { }

<INITIAL>.      { }

<INITIAL><<EOF>> {
                      // printf("\n\n");
                      yyterminate();
                  }




<PASSTWO>^{ws}*{comment} {
                    metric.ncsl--;
                    metric.mod_ncsl--;
                  }

<PASSTWO>^{ws}*\n {
                    inhibit_lookup_var = FALSE;
                    lookup_compound_statement = FALSE;

                    metric.loc++;
                    metric.mod_loc++;
                  }

<PASSTWO>{comment} { /* eat up comments */ }

<PASSTWO>{ws}*         { /* eat up white space */ }

<PASSTWO>^{ws}*("["){identifier}("]"){ws}*("=") {
                    lookup_mod_var = TRUE;

                    REJECT;
                  }

<PASSTWO>^{ws}*("["){identifier}(",X]"|",Y]"){ws}*("=") {
                    if (!strcmp(metric.func_name,"CHECKD0"))
                      lookup_mod_var = TRUE;

                    lookup_mod_var = TRUE;

```

```
                REJECT;
              }

<PASSTWO>^{ws}*("[(")(identifier}(",X)]"){ws}*("=") {
                lookup_mod_var = TRUE;

                REJECT;
              }

<PASSTWO>^{ws}*("[(")(identifier}("),Y]"){ws}*("=") {
                lookup_mod_var = TRUE;

                REJECT;
              }

<PASSTWO>^{ws}*{identifier}{ws}*("="|".EQU"){ws}*{bit}(",") {
                inhibit_lookup_var = TRUE;

                REJECT;
              }

<PASSTWO>^{ws}*{identifier}{ws}*(".blkb") {
                inhibit_lookup_var = TRUE;

                REJECT;
              }

<PASSTWO>^{identifier}{ws}*("="){ws}*{constant} {
                inhibit_lookup_var = TRUE;

                REJECT;
              }

<PASSTWO>("]"|")"|")") { /* eat up right paranthesis */ }

<PASSTWO>("\"") { /* eat up right qotes */
                if (lookup_quote)
                  lookup_quote = FALSE;
                else
                  REJECT;
              }

<PASSTWO>".FUNC" {
                metric.mod_func_count++;
                metric.mod_vg++;

                lookup_func_name = TRUE;
                if (!first_func)
                {
                  halstead_function(hash_table);
                  report_function();
                  hash_clear_func_count(hash_table);
                  first_func = FALSE;
                }
```

```
                    else
                      first_func = FALSE;

                    metric.loc = 0;
                    metric.ncsl = 0;
                    metric.vg = 1;
                    metric.func_call = 0;
                    metric.func_var_changed = 0;
                    metric.func_unique_var_changed = 0;
                    metric.func_var_read = 0;
                    metric.func_unique_var_read = 0;

                    metric.func_var_changed_pr = 0;
                    metric.func_unique_var_changed_pr = 0;
                    metric.func_var_read_pr = 0;
                    metric.func_unique_var_read_pr = 0;

                    hash_insert_token(hash_table, strupr(yytext), OPERATOR);
                  }

<PASSTWO>JSR      {
                    metric.func_call++;
                    metric.mod_func_call++;

                    REJECT;
                  }

<PASSTWO>(IF|LIF|FOR|LFOR|WHILE) {
                    lookup_compound_statement = TRUE;
                    metric.vg++;
                    metric.mod_vg++;

                    REJECT;
                  }

<PASSTWO>CASE|DEFAULT {
                    metric.vg++;
                    metric.mod_vg++;

                    REJECT;
                  }

<PASSTWO>ENDS     {
                    metric.vg--;
                    metric.mod_vg--;

                    REJECT;
                  }

<PASSTWO>(".REPEAT"|".REPEATC"|".REPEATI") {
                    metric.vg++;
                    metric.mod_vg++;

                    REJECT;
```

```
                    }

<PASSTWO>".IF"          {
                    metric.vg++;
                    metric.mod_vg++;

                    REJECT;
                    }

<PASSTWO>(BBC|BBS|BCC|BCS|BEQ|BMI|BNE|BPL|BVC|BVS) {
                    metric.vg++;
                    metric.mod_vg++;

                    REJECT;
                    }

<PASSTWO>{identifier} {
                    if (lookup_func_name)
                    {
                      strcpy(metric.func_name, strupr(yytext));
                      lookup_func_name = FALSE;
                    }
                    hash_insert_token(hash_table, strupr(yytext), OPERAND);

                    if (!inhibit_lookup_var)
                    {
                      if (lookup_mod_var)
                      {
                        if ((slot = hash_search(hash_table, strupr(yytext))) >= 0)
                        {
                          if ((hash_table[slot].type & (BYTEVAR | BITVAR | EQUATE)) != 0)
                          {
                            if ((hash_table[slot].type & EQUATE) == 0)
                            {
                              metric.func_var_changed++;
                              metric.mod_var_changed++;
                            }
                            metric.func_var_changed_pr++;
                            metric.mod_var_changed_pr++;
                            if ((hash_table[slot].reference & FUNCCHANGED) != FUNCCHANGED)
                            {
                              if ((hash_table[slot].type & EQUATE) == 0)
                                metric.func_unique_var_changed++;
                              metric.func_unique_var_changed_pr++;
                              hash_table[slot].reference |= FUNCCHANGED;
                            }
                            if ((hash_table[slot].reference & MODCHANGED) != MODCHANGED)
                            {
                              if ((hash_table[slot].type & EQUATE) == 0)
                                metric.mod_unique_var_changed++;
                              metric.mod_unique_var_changed_pr++;
                              hash_table[slot].reference |= MODCHANGED;
                            }
                          }
```

```

                    }
                    lookup_mod_var = FALSE;
                  }
                  else
                  {
                    if ((slot = hash_search(hash_table, strupr(yytext))) >= 0)
                    {
                      if ((hash_table[slot].type & (BYTEVAR | BITVAR | EQUATE)) != 0)
                      {
                        if ((hash_table[slot].type & EQUATE) == 0)
                        {
                          metric.func_var_read++;
                          metric.mod_var_read++;
                        }
                        metric.func_var_read_pr++;
                        metric.mod_var_read_pr++;
                        if ((hash_table[slot].reference & FUNCREAD) != FUNCREAD)
                        {
                          if ((hash_table[slot].type & EQUATE) == 0)
                            metric.func_unique_var_read++;
                          metric.func_unique_var_read_pr++;
                          hash_table[slot].reference |= FUNCREAD;
                        }
                        if ((hash_table[slot].reference & MODREAD) != MODREAD)
                        {
                          if ((hash_table[slot].type & EQUATE) == 0)
                            metric.mod_unique_var_read++;
                          metric.mod_unique_var_read_pr++;
                          hash_table[slot].reference |= MODREAD;
                        }
                      }

                    }
                  }
                }
              }


<PASSTWO>("=="|"!="|">"|"<"|">="|"<="|"||"|"&&"|"++"|"--") {
                if (lookup_compound_statement)
                {
                  if (!strcmp(yytext, "&&"))
                  {
                    metric.vg++;
                    metric.mod_vg++;
                    lookup_compound_statement = FALSE;
                  }
                  if (!strcmp(yytext, "||"))
                  {
                    metric.vg++;
                    metric.mod_vg++;
                    lookup_compound_statement = FALSE;
                  }
```

```
                    }

                    hash_insert_token(hash_table, strupr(yytext), OPERATOR);
                  }

<PASSTWO>("/"|"\""|"["|"="|"+"|"$"|"&"|","|"-"|"~"|"|"|"'"|"*"|"%"|"#"|"("|":") {
                    hash_insert_token(hash_table, strupr(yytext), OPERATOR);

                    if (!strcmp(yytext, "\""))
                      lookup_quote = TRUE;
                  }

<PASSTWO><<EOF>> {
                    halstead_function(hash_table);
                    report_function();
                    halstead_module(hash_table);
                    report_module();
                    hash_clear_func_count(hash_table);
                    hash_clear_mod_count(hash_table);

                    first_func = TRUE;
                    yyterminate();
                  }

<PASSTWO>\n       {
                    inhibit_lookup_var = FALSE;
                    lookup_compound_statement = FALSE;

                    metric.loc++;
                    metric.ncsl++;

                    metric.mod_loc++;
                    metric.mod_ncsl++;
                  }

<PASSTWO>.         {
                    error = TRUE;
                     fprintf(stderr,"undefined token in module %s: %s\n", metric.mod_name,
yytext);
                  }
%%
```

FILE METRIC.H:

```
// define some useful constants
#define TRUE 1
#define FALSE 0

// define bit constants for different types of tokens
#define OPERATOR 1
#define OPERAND 2
#define FUNCTION 4
#define BYTEVAR 8
#define BITVAR 16
#define EQUATE 32
#define PVAR 64

#define FUNCCHANGED 1
#define FUNCREAD 2
#define MODCHANGED 4
#define MODREAD 8

// extension of the report output file
#define REPORT_EXTENSION ".REP"

// data structure used to keep track of metric counts
struct metric_struct {
  char mod_name[80];
  char func_name[80];

  int mod_func_count;
  int sum_mod_func_count;
  int mod_jsr_count;
  int sum_mod_jsr_count;

  int loc;
  int mod_loc;
  int sum_mod_loc;

  int ncsl;
  int mod_ncsl;
  int sum_mod_ncsl;

  int vg;
  int mod_vg;
  int sum_mod_vg;

  int func_call;
  int mod_func_call;
  int sum_mod_func_call;

  int func_var_changed;
  int func_unique_var_changed;
  int mod_var_changed;
  int mod_unique_var_changed;
```

```
    int sum_mod_var_changed;
    int sum_mod_unique_var_changed;

    int func_var_changed_pr;
    int func_unique_var_changed_pr;
    int mod_var_changed_pr;
    int mod_unique_var_changed_pr;
    int sum_mod_var_changed_pr;
    int sum_mod_unique_var_changed_pr;

    int func_var_read;
    int func_unique_var_read;
    int mod_var_read;
    int mod_unique_var_read;
    int sum_mod_var_read;
    int sum_mod_unique_var_read;

    int func_var_read_pr;
    int func_unique_var_read_pr;
    int mod_var_read_pr;
    int mod_unique_var_read_pr;
    int sum_mod_var_read_pr;
    int sum_mod_unique_var_read_pr;

    int n1, n2;
    int N1, N2;
    float Nhat;
    float V;
    float E;

    int sum_n1, sum_n2;
    int sum_N1, sum_N2;
    float sum_Nhat;
    float sum_V;
    float sum_E;
};
```

FILE METRIC.C:

```
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>

#include "metric.h"
#include "hash.h"

// flex input/ouput files
extern FILE *yyin, *yyout;

// metric struct keeps all metric counts
struct metric_struct metric;

// pointer to hash table
struct hash_slot far *hash_table;

// error flag
int error = FALSE;

// do pass one first
int passtwo = FALSE;

// temporary file
FILE *fptmp;

// flags
int report_module_only = FALSE;

// report metric counts of function into temporary file
void report_function(void)
{
  int slot;

  // calculate jsr_count
  slot = hash_search(hash_table, metric.func_name);
  metric.mod_jsr_count += hash_table[slot].jsr_count;

  if (report_module_only)
    return;

  // if temporary file doesn't exists create it
  if (!fptmp)
  {
    if ((fptmp = fopen("TMP","w+")) == NULL)
    {
      perror("Error on creating temporary file");
      exit(1);
    }
  }
```

```
    fprintf(fptmp,"%s\t",metric.func_name);
    fprintf(fptmp," %5d\t",metric.n1);
    fprintf(fptmp," %5d\t",metric.N1);
    fprintf(fptmp," %5d\t",metric.n2);
    fprintf(fptmp," %5d\t",metric.N2);
    fprintf(fptmp," %9.0f\t",metric.Nhat);
    fprintf(fptmp," %9.0f\t",metric.V);
    fprintf(fptmp," %9.0f\t",metric.E);
    fprintf(fptmp," %5d\t",metric.vg);
    fprintf(fptmp," %5d\t",metric.loc);
    fprintf(fptmp," %5d\t",metric.ncsl);
    fprintf(fptmp," %5d\t",hash_table[slot].jsr_count);
    fprintf(fptmp," %5d\t",metric.func_call);
    fprintf(fptmp," %5d\t",metric.func_var_changed);
    fprintf(fptmp," %5d\t",metric.func_var_read);
    fprintf(fptmp," %5d\t",metric.func_unique_var_changed);
    fprintf(fptmp," %5d\t",metric.func_unique_var_read);
    fprintf(fptmp," %5d\t",metric.func_var_changed_pr);
    fprintf(fptmp," %5d\t",metric.func_var_read_pr);
    fprintf(fptmp," %6d\t",metric.func_unique_var_changed_pr);
    fprintf(fptmp," %5d",metric.func_unique_var_read_pr);
    fprintf(fptmp,"\n");
  }

  // report all metric counts
  void report_module(void)
  {
    static int first = TRUE;
    char c;

    metric.sum_mod_func_count += metric.mod_func_count;
    metric.sum_mod_jsr_count += metric.mod_jsr_count;
    metric.sum_mod_loc += metric.mod_loc;
    metric.sum_mod_ncsl += metric.mod_ncsl;
    metric.sum_mod_vg += metric.mod_vg;
    metric.sum_mod_func_call += metric.mod_func_call;
    metric.sum_mod_var_changed += metric.mod_var_changed;
    metric.sum_mod_unique_var_changed += metric.mod_unique_var_changed;
    metric.sum_mod_var_read += metric.mod_var_read;
    metric.sum_mod_unique_var_read += metric.mod_unique_var_read;
    metric.sum_mod_var_changed_pr += metric.mod_var_changed_pr;
    metric.sum_mod_unique_var_changed_pr += metric.mod_unique_var_changed_pr;
    metric.sum_mod_var_read_pr += metric.mod_var_read_pr;
    metric.sum_mod_unique_var_read_pr += metric.mod_unique_var_read_pr;
    metric.sum_n1 += metric.n1;
    metric.sum_n2 += metric.n2;
    metric.sum_N1 += metric.N1;
    metric.sum_N2 += metric.N2;
    metric.sum_Nhat += metric.Nhat;
    metric.sum_V += metric.V;
    metric.sum_E += metric.E;

    if (report_module_only)
```

```
  {
    if (first)
    {

fprintf(yyout,"%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\
t%s\t%s\t%s\n","NAME","FUNC","n1","N1","n2","N2","Nhat","V","E","VG","LOC","NCSL","FIN","F
OUT","VOUT","VIN","UVOUT","UVIN","VROUT","VRIN","UVROUT","UVRIN");
      first = FALSE;
    }
  }
  else

fprintf(yyout,"%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\
t%s\t%s\n","NAME","n1","N1","n2","N2","Nhat","V","E","VG","LOC","NCSL","FIN","FOUT","VOUT"
,"VIN","UVOUT","UVIN","VROUT","VRIN","UVROUT","UVRIN");

  fprintf(yyout,"%s\t",metric.mod_name);
  if (report_module_only)
    fprintf(yyout," %5d\t",metric.mod_func_count);
  fprintf(yyout," %5d\t",metric.n1);
  fprintf(yyout," %5d\t",metric.N1);
  fprintf(yyout," %5d\t",metric.n2);
  fprintf(yyout," %5d\t",metric.N2);
  fprintf(yyout," %9.0f\t",metric.Nhat);
  fprintf(yyout," %9.0f\t",metric.V);
  fprintf(yyout," %9.0f\t",metric.E);
  fprintf(yyout," %5d\t",metric.mod_vg);
  fprintf(yyout," %5d\t",metric.mod_loc);
  fprintf(yyout," %5d\t",metric.mod_ncsl);
  fprintf(yyout," %5d\t",metric.mod_jsr_count);
  fprintf(yyout," %5d\t",metric.mod_func_call);
  fprintf(yyout," %5d\t",metric.mod_var_changed);
  fprintf(yyout," %5d\t",metric.mod_var_read);
  fprintf(yyout," %5d\t",metric.mod_unique_var_changed);
  fprintf(yyout," %5d\t",metric.mod_unique_var_read);
  fprintf(yyout," %5d\t",metric.mod_var_changed_pr);
  fprintf(yyout," %5d\t",metric.mod_var_read_pr);
  fprintf(yyout," %6d\t",metric.mod_unique_var_changed_pr);
  fprintf(yyout," %5d",metric.mod_unique_var_read_pr);
  fprintf(yyout,"\n");

  if (report_module_only)
    return;

  fprintf(yyout,"\n");

  // get all stuff out of temporary file
  fseek(fptmp, 0L, 0);
  while (!feof(fptmp))
  {
    c = fgetc(fptmp);
    if (c != EOF)
      fprintf(yyout,"%c", c);
  }
```

```
  fclose(fptmp);
  remove("TMP");
  fptmp = NULL;

  fprintf(yyout,"\n\n");
}

// report all sums of metric counts
void report_sum_module(void)
{
  char directory[80];
  int i;

  getcurdir(0, directory);
  for (i=strlen(directory); i != 0; --i)
    if (directory[i] == '\\')
      break;

  fprintf(yyout,"\n%s\t",&directory[i+1]);
  fprintf(yyout," %5d\t",metric.sum_mod_func_count);
  fprintf(yyout," %5d\t",metric.sum_n1);
  fprintf(yyout," %5d\t",metric.sum_N1);
  fprintf(yyout," %5d\t",metric.sum_n2);
  fprintf(yyout," %5d\t",metric.sum_N2);
  fprintf(yyout," %9.0f\t",metric.sum_Nhat);
  fprintf(yyout," %9.0f\t",metric.sum_V);
  fprintf(yyout," %9.0f\t",metric.sum_E);
  fprintf(yyout," %5d\t",metric.sum_mod_vg);
  fprintf(yyout," %5d\t",metric.sum_mod_loc);
  fprintf(yyout," %5d\t",metric.sum_mod_ncsl);
  fprintf(yyout," %5d\t",metric.sum_mod_jsr_count);
  fprintf(yyout," %5d\t",metric.sum_mod_func_call);
  fprintf(yyout," %5d\t",metric.sum_mod_var_changed);
  fprintf(yyout," %5d\t",metric.sum_mod_var_read);
  fprintf(yyout," %5d\t",metric.sum_mod_unique_var_changed);
  fprintf(yyout," %5d\t",metric.sum_mod_unique_var_read);
  fprintf(yyout," %5d\t",metric.sum_mod_var_changed_pr);
  fprintf(yyout," %5d\t",metric.sum_mod_var_read_pr);
  fprintf(yyout," %6d\t",metric.sum_mod_unique_var_changed_pr);
  fprintf(yyout," %5d",metric.sum_mod_unique_var_read_pr);
  fprintf(yyout,"\n");
}

// initialize metric struct
void init_metric(void)
{
  metric.mod_func_count = 0;
  metric.mod_jsr_count = 0;
  metric.mod_loc = 0;
  metric.mod_ncsl = 0;
  metric.mod_vg = 0;
  metric.mod_func_call = 0;
  metric.mod_var_changed = 0;
  metric.mod_unique_var_changed = 0;
```

```
  metric.mod_var_read = 0;
  metric.mod_unique_var_read = 0;
  metric.mod_var_changed_pr = 0;
  metric.mod_unique_var_changed_pr = 0;
  metric.mod_var_read_pr = 0;
  metric.mod_unique_var_read_pr = 0;
}

// initialize metric struct
void init_sum_metric(void)
{
  metric.sum_mod_func_count = 0;
  metric.sum_mod_jsr_count = 0;
  metric.sum_mod_loc = 0;
  metric.sum_mod_ncsl = 0;
  metric.sum_mod_vg = 0;
  metric.sum_mod_func_call = 0;
  metric.sum_mod_var_changed = 0;
  metric.sum_mod_unique_var_changed = 0;
  metric.sum_mod_var_read = 0;
  metric.sum_mod_unique_var_read = 0;
  metric.sum_mod_var_changed_pr = 0;
  metric.sum_mod_unique_var_changed_pr = 0;
  metric.sum_mod_var_read_pr = 0;
  metric.sum_mod_unique_var_read_pr = 0;
  metric.sum_n1 = 0;
  metric.sum_n2 = 0;
  metric.sum_N1 = 0;
  metric.sum_N2 = 0;
  metric.sum_Nhat = 0;
  metric.sum_V = 0;
  metric.sum_E = 0;
}

void help()
{
  printf("metric [[@]filename] [-mh] \n\n");
  printf("-m output modules only\n");
  printf("-h help screen\n\n");
  printf("filename  is an input file.\n");
  printf("@filename is a file that contains multiple input files of a project.\n");
}

int main(int argc, char** argv)
{
  FILE *project_file = NULL;
  char fname[80];
  int first_file = TRUE;
  char directory[80];
  int i;

  // get arguments
  ++argv; --argc;
  if (argc > 0)
```

```
  {
    if (strstr(argv[0],"-h"))
    {
      help();
      exit(1);
    }

    // is it a project file ?
    if (argv[0][0] == '@')
    {
      if ((project_file = fopen(&argv[0][1], "r")) == NULL)
      {
        perror("Error on reading project file");
        exit(1);
      }
    }

    // otherwise it's a single file
    else
    {
      yyin = fopen(argv[0], "r");
      strcpy(metric.mod_name, strupr(argv[0]));
    }
  }
  // or worse, it's from stdinput
  else
  {
    yyin = stdin;
    strcpy(metric.mod_name, "STDIN");
  }

  // anything else
  if (argc > 1)
    if (argv[1][0] == '-')
    {
      if (strstr(argv[1],"m"))
        report_module_only = TRUE;
    }

  yyout = stdout;

  // initialize metric struct
  init_metric();
  init_sum_metric();

  // allocate memory for hash table
  hash_table  =  (struct  hash_slot*  far)  farmalloc(HASH_TABLE_SIZE  *  sizeof(struct
hash_slot));
  if (hash_table == NULL)
  {
    perror("Error on creating hash table");
    exit(1);
  }
```

```
  // initialize hash table
  hash_init(hash_table);

  // put in all operators
  hash_init_operators(hash_table);

  // put in all processor registers and predefined resources
  // hash_init_registers_resources(hash_table);

  // if it is a single file
  if (!project_file)
  {
    yylex();
    fseek(yyin, 0L, 0);
    passtwo = TRUE;
    yyrestart(yyin);
    yylex();

    if (error)
      fprintf(stderr,"error occurred.\n");

    return(0);
  }

  // otherwise proceed project file
  for (i=0; i<2; i++)
  {
    while (fgets(fname, sizeof(fname), project_file) != NULL)
    {
      fname[strlen(fname)-1] = '\0';
      yyin = fopen(fname, "r");
      strcpy(metric.mod_name, strupr(fname));

      if (yyin)
      {
        if (!first_file)
          yyrestart(yyin);

        init_metric();
        yylex();
      }

      fclose(yyin);
      first_file = FALSE;

    }
    // set pass two and do it again
    fseek(project_file, 0L, 0);
    passtwo = TRUE;
  }

  fclose(project_file);

  if (report_module_only)
```

```
      report_sum_module();

    if (error)
      fprintf(stderr,"error(s) occurred.\n");

    return 0;
  }
```

FILE HASH.H:

```
// define hash table size
#define HASH_TABLE_SIZE 1999

// define identifier length
#define ID_LENGTH 10

// data structure that is stored in the hash table
struct hash_slot {
        int key;
        char identifier[ID_LENGTH];
        int mod_count;
        int func_count;
        int jsr_count;
        int type;
        int reference;
};
```

FILE HASH.C:

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <values.h>

#include "metric.h"
#include "hash.h"

// needs access to metric struct
extern struct metric_struct metric;

// #define DEBUG

// double hashing function
int hash(int k, int i)
{
  long hash_value;

  hash_value = k % 1999;
  if (i != 0)
    hash_value += i * (long) (1 + (k % 1997));
  hash_value %= 1999;

  if (hash_value<0)
  {
    perror("Error on hash");
    exit(1);
  }

  return (hash_value);
}

// insert some string into hash table
int hash_insert(struct hash_slot* table, char* id)
{
  int i = 0;
  int j;
  int key;

  key = str2key(id);
  do {
    j = hash(key,i);
    if (table[j].key == -1)
    {
      table[j].key = key;
      strcpy(table[j].identifier, id);
      return(j);
    }
    else
      i++;

```

```
    } while (i<HASH_TABLE_SIZE);

    perror("Error on hash_insert");
    exit(1);
}

// lookup some string in hash table
int hash_search(struct hash_slot* table, char* id)
{
    int i = 0;
    int j;
    int key;

    key = str2key(id);
    do {
      j = hash(key,i);
      if ((table[j].key == key) && (!strcmp(table[j].identifier, id)))
        return j;
      i++;
    } while ((table[j].key >= 0) && (i<HASH_TABLE_SIZE));
    return(-1);
}

// build hash key
int str2key(char* s)
{
    int i, key;

    key = 0;
    for (i=0; i<strlen(s); i++)
    key ^= s[i];
    return key;
}

// insert a token with a given type into hash table
int hash_insert_token(struct hash_slot *table, char *s, int type)
{
    int slot;

    if ((slot = hash_search(table, s)) < 0)
      slot = hash_insert(table, s);

    if (!((table[slot].type == OPERATOR) && (type == OPERAND)))
      table[slot].type |= type;

    if ((type == OPERATOR) || (type == OPERAND))
    {
      table[slot].mod_count++;
      table[slot].func_count++;
    }

    if (type == FUNCTION)
      table[slot].jsr_count++;

```

```
  return slot;
}

// initialize hash table
void hash_init(struct hash_slot* table)
{
  int i;

  for(i=0; i<HASH_TABLE_SIZE; i++)
  {
    table[i].key = -1;
    strcpy(table[i].identifier,"");
    table[i].mod_count = 0;
    table[i].func_count = 0;
    table[i].jsr_count = 0;
    table[i].type = 0;
    table[i].reference = 0;
  }
}

// clear function counts of hash table
void hash_clear_func_count(struct hash_slot* table)
{
  int i;

  for(i=0; i<HASH_TABLE_SIZE; i++)
  {
    table[i].func_count = 0;
    table[i].reference &= (~FUNCCHANGED);
    table[i].reference &= (~FUNCREAD);
  }
}

// clear module counts of hash table
void hash_clear_mod_count(struct hash_slot *table)
{
  int i;

  for(i=0; i<HASH_TABLE_SIZE; i++)
  {
    table[i].mod_count = 0;
    table[i].reference = 0;
  }
}

// insert operators into hash table
void hash_init_operators(struct hash_slot *table)
{
  FILE *operators;
  int slot;
  char s[ID_LENGTH];

  if ((operators = fopen("OP.TXT","r")) == NULL)
  {
```

```
      perror("Error on reading operator file");
      exit(1);
    }
    while (!feof(operators))
    {
      fgets(strupr(s),ID_LENGTH,operators);
      s[strlen(s)-1] = '\0';
      slot = hash_insert(table, s);
      table[slot].type = OPERATOR;
    }
    fclose(operators);
  }

  // calculate halstaeds counts for functions
  void halstead_function(struct hash_slot *table)
  {
    int i;

    metric.n1 = metric.n2 = metric.N1 = metric.N2 = 0;

    for(i=0; i<HASH_TABLE_SIZE; i++)
    {
      if (table[i].func_count != 0)
      {
        if ((table[i].type & OPERATOR) == OPERATOR)
        {
          metric.n1++;
          metric.N1 += table[i].func_count;
        }

        if ((table[i].type & OPERAND) == OPERAND)
        {
          metric.n2++;
          metric.N2 += table[i].func_count;
        }
      }
    }

    if (metric.n2 == 0)
    {
      perror("halstaed's n2 is 0, division by zero");
      exit(1);
    }

    metric.Nhat = metric.n1 * log(metric.n1)/log(2) + metric.n2 * log(metric.n2)/log(2);
    metric.V = (metric.N1 + metric.N2) * log(metric.n1 + metric.n2)/log(2);
    metric.E = ((metric.N1 + metric.N2) * log(metric.n1 + metric.n2)/log(2) * metric.n1 *
  metric.N2) / (2 * metric.n2);
  }

  // calculate halstaeds counts for module
  void halstead_module(struct hash_slot *table)
  {
    int i;
```

```
  metric.n1 = metric.n2 = metric.N1 = metric.N2 = 0;

  for(i=0; i<HASH_TABLE_SIZE; i++)
  {
    if (table[i].mod_count != 0)
    {
      if ((table[i].type & OPERATOR) == OPERATOR)
      {
        metric.n1++;
        metric.N1 += table[i].mod_count;
      }

      if ((table[i].type & OPERAND) == OPERAND)
      {
        metric.n2++;
        metric.N2 += table[i].mod_count;
      }
    }
  }

  if (metric.n2 == 0)
  {
    perror("halstaed's n2 is 0, division by zero");
    exit(1);
  }

  metric.Nhat = metric.n1 * log(metric.n1)/log(2) + metric.n2 * log(metric.n2)/log(2);
  metric.V = (metric.N1 + metric.N2) * log(metric.n1 + metric.n2)/log(2);
  metric.E = ((metric.N1 + metric.N2) * log(metric.n1 + metric.n2)/log(2) * metric.n1 *
metric.N2) / (2 * metric.n2);
```