AN ABSTRACT OF THE THESIS OF

Warren A. Harrison for the degree of Doctor of Philosophy in Computer Science presented on July 10, 1985.

Title: A Study of Macro Level Complexity Metrics

Abstract approved: Redacted for Privacy
Curtis R. Cook

Controlling the "complexity" or "understandability" of computer software is important because of its impact on program testing and maintenance. Of the large number of complexity metrics that have been developed to measure the complexity of a computer program, most assess the "micro-complexity" of each subprogram and few assess the "macro-complexity" of the entire program. The dissertation introduces a new macro-complexity metric that incorporates global variable and parameter usage, average module micro-complexity, and internal documentation.

Validation, or demonstrating that a metric does work. is a difficult problem. Industry is seldom willing to provide researchers with actual source code because of the fear of it falling into the hands of competitors. The dissertation presents a method, called a Reduced Form. that allows researchers access to the information about the source code, but prevents reconstruction of the code from the information.

In a field study that compared the new macro-complexity metric with several other macro and popular micro metrics, the Reduced Form was used to collect empirical data. The results of the study suggest that the new metric performs significantly better than any of the others studied. However, a simple count of lines of code, while not as highly correlated with program errors as the new metric, does prove to be much easier to compute, and performs at least as well as any of the other metrics studied with the exception of the new metric. Therefore lines of code probably remains the metric of choice for most situations.

A Study of Macro Level Complexity Metrics

by

Warren A. Harrison

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed July 10, 1985

Commencement June 1986

APPROVED:

Redacted for Privacy

Professor of Computer Science in Charge of Major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented July 10, 1985

Typed by Warren Harrison

ACKNOWLEDGEMENTS

I would like to express my appreciation to each member of my committee for the time they invested in my program. 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 his help was invaluable as this work began to take shape. Also, Nancy Currans, our liason at Company A made data collection as easy as possible. She deserves a great round of thanks.

I also owe a very special thanks to my wife, Teresa who put up with me while I muddled through this whole process. Of course, a big thank-you must also go to Mom and Dad who were a constant source of encouragement throughout the entire graduate school experience.

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

A STUDY OF MACRO LEVEL COMPLEXITY METRICS

CHAPTER I INTRODUCTION

Recently, a significant emphasis has been placed on controlling the complexity, or understandability (from the programmer's standpoint) of computer software. However, in order to effectively control complexity, it must first be measured. To do this, a large number of different "complexity metrics" have been developed and studied. A complexity metric is a scheme for assessing how understandable a piece of code is to a programmer. It is usually based on certain characteristics of the software which can be statically measured, such as number of decisions, density of comments, number of GOTOs, etc.

Most popular software complexity metrics are based on characteristics which primarily affect a programmer's "micro-understanding" of a piece of software. That is, understanding a small piece of software (e.g., a subprogram or function definition) in isolation.

Recently, the idea of "macro-understanding" has been addressed [Henry & Kafura, 1981, Hall & Preiser, 1984]. In macro-understanding, the interrelationships of the various parts of a system (e.g., the effect of each subprogram on the other subprograms) is considered of primary importance, even if the detailed operation of each of the parts is not especially clear.

Unfortunately, there is no consensus regarding which approach (micro or macro) is best, nor even which metric or metrics within the two approaches are the best. The "goodness" of a metric is often based on how well it is correlated with some aspect of programmer behavior over a large number of programs. For example, the relationship between a metric's measurements and the number of errors encountered in a programming project is a frequently used measure of metric goodness.

Two general approaches for determining how well a metric works are field studies and controlled experimentation. However, for field studies, it is quite difficult for researchers to assess metric performance since most organizations are reluctant to allow "outsiders" access to their software for fear of possible misuse of the programs (e.g., disclosure of proprietary algorithms or formulas to competitors, loss of trade secret protection, etc.). This reluctance prevents researchers from gathering data from industrial software projects for use in developing and studying metrics. As a consequence, most metric research has used controlled experimentation with students as subjects. It is not clear that the results of this research can be generalized to professional programmers and large systems.

In this dissertation, we present a technique which allows organizations to share the characteristics of their software with complexity metric researchers, yet at the same time prevents the possible misuse of the software usually feared by industrial organizations. The technique, which we term the "Reduced Form" captures the program data in a format that allows most metrics to be calculated, but that prevents the original source code from being replicated.

The contribution of the Reduced Form on software complexity metric research is potentially immense. This technique could allow the empirical evaluation of the hundreds of complexity metrics currently espoused by various researchers. However, unlike most work being done today, the empirical data can be obtained from "real world" projects rather than small student programs. This should provide results more applicable to industry. Further, it will allow the convienient replication of studies involving industrial projects by others since researchers need no longer be barred from data by claims of proprietory code systems.

If the use of replicated, wide ranging studies, based on industrial software systems can be used to determine the "best" metric (or set of metrics), it will have a great impact on the current state of software development. To help control complexity, software tools which automatically calculate complexity metrics could be used as a feedback tool for programmers, much like code reviews. Further, the development of a set of accurate metrics could be used to provide gauges of software understandability when developing requirements/specifications in software contracts. Metrics could also be used to help schedule various programming tasks such as testing and maintenance.

However, the problem of acquiring reliable "performance data" is still a major roadblock to rapid advancement in the field of software complexity metrics.

The Reduced Form will at least allow one part of the metric assessment equation to be obtained:

```
code characteristics
      + performance data = metric assessment
```

Future work will involve the development of a standardized format for performance data.

This dissertation presents a new macro-complexity metric which incorporates a number of program characteristics, all of which are available from the Reduced Form. This metric is compared against several others (both micro and macro) in an analysis of a moderate-sized compiler project developed by a large industrial organization in the Pacific Northwest. Using error data collected from this project as well as Reduced Form data to calculate these metrics, we found that in some situations the new metric performs much better than any of the other metrics tested, and seldom performs worse. However, when compared to a simple measure of lines of code the increased performance of the new measure may not adequately compensate for the additional effort required to compute it. This is because the new metric, while performing significantly better than lines of code in most cases, requries a rather involved process of computation, while lines of code may be easily computed using a tool as trivial as a text editor.

The development of this metric and its subsequent performance provides more information on some characteristics which may affect macro complexity. If this metric can perform as well in future studies as it did in the one presented in this dissertation, it can be expected to set the standard of performance for new metrics to beat.

This dissertation consists of three major parts. The first part, Chapters II and III, is an introduction to software complexity and software complexity metrics. Chapter II is a general introduction to software complexity. Chapter III is a survey of currently popular complexity metrics, and introduces the new metric to measure macro-complexity. The second part, Chapter IV, describes the Reduced Form and presents the results of a survey involving industrial organizations around the nation regarding the acceptability of the Reduced Form. The third part of this dissertation, Chapter V, describes the collection of data in our field study, and evaluates several major complexity metrics (both macro and micro). Our conclusions and future work are discussed in Chapter VI.

CHAPTER II
INTRODUCTION TO SOFTWARE QUALITY AND COMPLEXITY

The Creative Freedom of Programmers

It has been said that programming is an art - not a science. This feeling is based on the great creative freedom of a programmer. Much like an artist creates an effect by applying colors and shades, a programmer creates a solution to a problem by selecting programming languages, control structures, data abstractions, variable names and so forth.

Measuring the Quality of Choices

The choices a programmer makes usually have wide-ranging effects. If a slow algorithm is selected, program performance may suffer. A misleading variable name may confuse a maintenance programmer. A restrictive data structure may make it awkward for the program to handle certain combinations of data. Thus, when there are two or more approaches to writing a program, it would be useful to determine which approach is "best", or leads to a higher quality program.

The possible criteria that may be considered when assessing the quality of a piece of software seem limitless. Boehm, Brown and Lipow [1977] suggest the following five software quality criteria:

(1) Maintainability

(2) Reliability

(3) Efficiency

(4) Human Engineering

(5) Portability

While all these factors are extremely important, many researchers consider maintainability as important as any of the others. This is because the bulk of the programming tasks (and therefore the cost) involve the maintenance of existing systems rather than the development of new ones. Thus, it is important that a piece of software be easy to maintain.

Understandability and Program Complexity

The maintenance activity is actually composed of the following three phases [Boehm, 1976]:

1) understanding the existing software

2) modifying the software

3) retesting the changed software

The American Heritage Dictionary defines understanding as "...to thoroughly perceive and comprehend the nature and significance of something..." [1973]. Understanding the software then means to thoroughly comprehend the nature and significance of the different parts of the code.

The first phase, "understanding the software", is crucial to the other two phases. It also has a significant effect on other activities within the software lifecycle:

1) Development - As a program is being developed, the portion being written is highly dependent on other

portions, some of which have already been written. This is especially true in large projects where different programmers may work on different parts of the system. It is important that the effects and interactions among previously written segments be clear. If misunderstandings occur, development time can increase, and errors may be introduced.

2) Testing - After a program has been developed, it must be tested for correctness. To this end, "test cases" must be developed to exercise its behavior for different types of input. If a clear understanding exists of how the software works, test cases which exercise different paths within the system should be easier to design. Thus, testing would be more efficient and effective since redundant test cases could be omitted, fewer test cases would be overlooked and the test cases would be more likely to uncover errors.

Hence, the ability of a programmer to understand a piece of software is an essential aspect of software quality. It then follows that an accurate evaluation of the quality of a piece of software, requires a method to measure how "understandable" it is.

Characteristics that make a program difficult to understand are said to increase the software's "complexity". These characteristics could include things such as a large number of GOTO statements, excessive use of global variables, and inadequate documentation (e.g., not enough comments).

A piece of software may be used by itself - or more often - in concert with several other pieces of software in what is known as a "system". In general, a system is a set of objects that work together to accomplish some common goal. In a software system, the objects are either programs and/or subprograms which may invoke other programs and/or subprograms in the system. A programmer may concern himself with understanding the overall system, or just the individual parts. This implies that there are at least two levels of understanding: macro understanding and micro understanding.

Micro understandability refers to how well a programmer can understand the detailed operation of an individual piece of a system. On the other hand, macro understanding refers to how well a programmer understands the operation of the overall system. Large systems may even have several levels of understanding (i.e., subsystems within systems).

It would seem that macro and micro understanding are hierarchical concepts - however, they are actually orthogonal. One does not have to know how each function or subroutine in a system works in order to understand how the overall system operates. On the other hand, one does not need to know how the overall system works in order to understand the detailed operation of one of its subparts (e.g.., a function or subroutine). As an example, consider a module that sorts a group of names in a payroll application. In order to understand the overall operation of the payroll system, there is no need to know if the sort module uses a quicksort, shellsort, or some other method. Likewise, if the decision is made to change the sorting algorithm used, the programmer making the changes need not know that the sort is being used in a payroll application.

Measuring Complexity

If those characteristics which affect the complexity of software can be identified, they may be used when comparing the quality of programs in one of two ways:

(1) An individual can examine and then compare, several systems based on the presence (or absence) of characteristics that affect complexity. This is a useful approach since special cases can be considered (e.g., sometimes GOTOs can aid understanding [Knuth, 1974]). However, this aprroach also has some limitations. It is not reasonable to expect someone to manually examine even a modest sized system (1000-10000 lines). Further, since human judgement is subjective, the results may not be consistent - ie, two evaluators may rank one system differently, by assigning different importance to the various characteristics or special cases.

(2) From the complexity characteristics, a set of consistent, objective rules can be developed to assess the degree to which these characteristics are present in a system. These rules could then be used to assess the quality of the software. This would allow a consistent ranking for a set of systems or pieces of software according to their complexity. Such rules are called complexity metrics. This approach is preferable to the first approach since it has the advantage of being consistent, and allows

the calculation to be automated, allowing thousands of lines of software to be examined in a matter of minutes. The major drawback is that special cases cannot always be considered. However, if we view software quality in a "statistical" sense - ie, over a large number of systems - the existence of special cases poses no great problem.

Unfortunately, most computer scientists seem to have many different (and sometimes conflicting) ideas about what makes a piece of software hard to understand. Even when they agree on a set of characteristics, they tend to assign varying degrees of importance to the various characteristics. Hence, it seems there is a metric for virtually every measurable characteristic [Harrison et al., 1982]. Unfortunately, the user who wishes to apply these metrics is left with the question: "Which ones (if any) work?".

Evaluating Metrics

Typically two approaches are taken to "validate", or determine if a particular characteristic or set of characteristics actually has an effect on the understandability of a program:

1) Controlled Experimentation

2) Field Studies

In controlled experimentation, two or more versions of the same program are prepared, each with differing degrees of the characteristic being studied. For example, one version may use detailed comments, one version may use high-level comments and the third version may use no comments. A number of subjects are recruited and asked to perform some task involving the program that is thought to be affected by program understanding, such as finding an error or answering some questions about what the program does. The subjects' performance (e.g., time to find the error or number of correct answers) on each version of the program is then compared to assess which version was the easiest to work with. This usually involves some sort of statistical analysis to determine the significance of the result.

Since most experimental studies are performed at universities, students are most often used as subjects.

To allow subjects to complete the experiments within a single class period, small programs are usually used. The use of experimental studies allows the researcher to tightly control the characteristics being studied. The experimental materials only vary in the characteristics being studied. Unfortunately, it is not clear that the results of such experiments generalize to larger systems written by professionals. Further, the size of the programs used often precludes dealing with macro understanding. However, experimental results can be used to suggest characteristics which warrant further study.

The weaknesses of controlled experimentation have led to field studies. In a field study, data from one or more "real world" systems is collected and analyzed to determine which characteristics they possess and to what degree (this is called a code analysis). Then performance measurements are made as programmers perform certain tasks (e.g., maintenance, debugging, etc.) that the researchers feel may be associated with program understanding. Programmer performance for the various tasks is then compared with programmer performance on other systems with differing characteristics.

Field studies suffer from the problem of how and what to measure - especially with regard to programming tasks. Also, many characteristics vary from one code system to another, so it is not always clear that the differences in performance observed are entirely due to the characteristics under study. However, the field study tends to eliminate many of the problems of controlled experimentation involving the generalization of results of students working on small programs to professionals working on realistic systems.

Attempts at Validation

In the past, many complexity metrics have been used in field studies and/or experiments. Field studies by Basili [Basili et al., 1983], [Basili & Perricone, 1984], [Basili & Hutchens, 1983], [Basili & Phillips, 1981], Henry and others [Henry et al., 1981], Baily and Dingee [Bailey & Dingee, 1981], and Feuer and Fowlkes [Feuer & Fowlkes, 1979] indicate that there is some relationship between various metrics and the number of manhours required to implement the software, or the number of errors discovered in the software during testing. However, little conclusive evidence has been presented that any one metric works much better than the others.

However, it seems that a simple count of the executable lines of a program correlates best with maintenance effort [Gremillion, 1984]. Program lines of code also works at least as good as most other metrics for many other purposes. This, coupled with its ease of calculation has resulted in the Lines of Code metric being one of the most commonly accepted metrics in use today.

Studies involving controlled experimentation by Curtis and others [Curtis et al., 1979a], [Curtis et al., 1979b] and Jorgensen [Jorgensen, 1980] conclude that certain metrics vary in predicting subjects′ performance in controlled experimentation. For example, in [Curtis et al., 1979b] the length of a set of small programs was found to be significantly correlated with the time required for the subjects to complete a modification to the programs. However, less significant correlations were observed between time to modify the programs and the number of decisions in the programs. As with the field study results, experimentation suggests that program size is at least as good as other metrics in predicting subject perfoemance. However, as of yet, no metric or set of metrics seem to be consistently good predictors of subject performance. Finding such a metric is the ultimate goal of work in this area.

CHAPTER III
MICRO AND MACRO MEASURES OF COMPLEXITY

This chapter describes a number of popular micro and macro complexity measures. In addition, it presents a new macro complexity metric which includes the effect of global variables and parameters, as well as the aid provided by internal documentation.

Two Levels of Understanding

As mentioned in Chapter II, there are two levels of understanding. There is micro understanding and macro understanding. The exact meaning of these concepts can be best described by an example programming task.

Consider the enhancement of a program containing several subprograms. As a first step, the programmer must determine what part or parts of the program will be changed. This involves macro understanding, since only the function, and not the detailed operation of any given subprogram need be known. All that is really necessary is to determine how each subprogram relates to the function of the overall system, and to the proposed enhancement. If the program is divided into relatively isolated modules, with limited side effects, this may be relatively simple. On the other hand, this can be a difficult task if the subprograms are highly interrelated with many side effects.

Once the subprogram(s) to be changed have been identified, the programmer only needs to understand the detailed operation of these subprograms to make the modification. This is micro understanding. However, if the module has side effects, a micro level comprehension of the other modules may also be neccessary.

Characteristics that Affect Micro Understanding and Metrics to Measure Micro Complexity

There are three groups of characteristics that have a significant effect on micro understanding:

1) Control Flow - the number of decisions made in a program and their interrelationships.

2) Program Size - the number of statements in the program.

3) Data Structure and Flow - the relationship and use of the data elements within a program.

Usually, the analysis of each of these groups of characteristics is based on a single piece of software (e.g., a single program or subprogram) with little explicit consideration given to how that piece of software relates to other pieces of software within the system.

Control Flow Metrics

One popular method of measuring the complexity of a computer program is to examine the program's flow of control. This is generally done by decomposing the program into a directed graph, G=(V,E), where V is the set of nodes in the graph and E is the set of edges. Each node reprsents a "block" of code which can be only entered at the "top", exited from the "bottom" and has no internal transfers of control other than to the next statement. The edges represent the flow of control between the blocks of code. The resulting directed graph is typically referred to as a flowgraph. An example of a flowgraph is given in Figure 1.

A number of so-called topological complexity metrics are currently quite popular. An extensive survey of this type of metric has already been published [Cook and Harrison, 1984]. We will briefly describe some of the more popular topological metrics.

McCabe [McCabe 1976] originally suggested that the Cyclomatic Number of a program flowgraph be used as a measure of program complexity. The Cyclomatic Number is the number of linearly independent paths from the start node of a flowgraph to the exit node and 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. The flowgraph in Figure 1 has a McCabe complexity of 4.

It has been shown that for programs with a single exit and single entry, the Cyclomatic Number can be calculated from the formula or by simply counting the number of decision statements in the program π, and adding one. More recently, it has been shown that when the program has several exits McCabe's formula no longer agrees with the shortcut formula $\pi + 1$. McCabe's formula can be calculated by using the shortcut $\pi - s + 2$ in the general case [Harrison, 1984], where π is, as before, the number of decision statements, and s is the number of "exit" statements (e.g., STOP, RETURN, etc.). Regardless of the method of computation, McCabe's metric has become quite popular with the research community because it is so simple to compute and agrees with intuitive notions of complexity.

Soon after McCabe's original work, Myers [Myers, 1977] observed that the metric did not address the problem of multiple conditions in a single IF statement. He suggested that an interval measure be used, consisting of McCabe's metric as a lower bound and the total number of predicates, plus one, as an upper bound.

Some control flow metrics are not based on an analysis of the program's flowgraph. Storm and Preiser's index of complexity [Storm & Preiser, 1979] was developed to measure the complexity of various structured programming constructs. The index is based on the following definitions:

(1) Let m[s] be the complexity of program segment s, which has l[s] lines of source code, so that m[s] = l[s].

(2) Let m[c] be the measure of complexity of a test on condition c, so that m[c] = d[c], d[c]>0.

(3) For the sequence s1;s2, let m[s1]+m[s2] = l[s1]+l[s2].

(4) For the iteration WHILE c DO s, let m[c]+m[s] = d[c]+l[s].

(5) For the alternation IF c THEN s1 ELSE s2, let m[c]+avg(m[s1],m[s2]) = d[c]+(l[s1]+l[s2])/2.

The complexity of a program p, is then calculated by determining m[p]. The program of Figure 1 has a complexity of 26. Assuming that d[c] = 1, this is calculated in the following manner:

```
if x[i] > x[i+1] ... has complexity of (1 + 4)  5
for i:=1 to n-1 ... has complexity of  (1 + 5)  6
while SwapMade ... has complexity of   (1 + 9) 10
the lines not already counted
    have a complexity of                        5

for total complexity of                        26
```

Program Size Metrics

Probably the most widely used metrics are based upon the size of the software in some way or another. Research suggests that a simple count of lines of code (LOC) is as reliable as any other metric [Gremillion, 1984].

A refinement of the size metric was developed by Halstead [Halstead, 1977]. This work spawned an entire family of metrics based on the following parameters:

n1:	count of unique operators
n2:	count of unique operands
n=n1+n2:	program vocabulary
N1:	count of total operators
N2:	count of total operands
N=N1+N2:	implementation length

Based on these counts, Halstead developed a measure of program size called Volume which is calculated as:

$$V = N(\log_2 n)$$

giving the program volume in bits. This makes program size independent of the "token representation" of the language (i.e., number of characters required to represent a token - the token representation of a variable in BASIC might be: Z1, while in a machine language, it might be the hex number AEF7). A second more widely used metric from this family is Effort, which is based on the program's Volume and the Program Level (L). Program Level is a measure of the "abstraction" inherent in a particular implementation of an algorithm. The greater the Level, the more "abstract" (or "higher level") the implementation. Level is calculated as:

$$L = \text{potential Volume} \,/\, \text{actual Volume}$$

where potential Volume is the Volume which would be observed if the program could be implemented simply as a function call. Effort is calculated as:

$$E = V/L$$

and provides an approximation of the effort (number of elementary mental discriminations) required to write the program. Halstead introduced a time element into this metric by observing that a programmer could perform between 5 and 20 mental discriminations per second. Thus, the amount of time required to write a given program should be between E/5 and E/20.

While Halstead's initial work was directed towards estimation of program development effort, the metrics can be logically extended to estimating the effort required in working with a program. They have been a popular research topic over the past several years [Curtis, et al 1979a and 1979b],[Gordon, 1979].

Data Structure and Flow Metrics

Metrics which consider the use of data within a program seem to be a neglected area of exploration. Often, a simple count of the number of variables used is considered a useful measure of program comprehensibility. Clearly, Halstead's metrics make use of this concept via the N2 parameter.

Yet another approach, suggested by Elshoff [Elshoff, 1976], is referred to as "The Span of Reference" of a data item. This is a measure of the number of lines of code occurring between consecutive references to a particular data item. The span of reference is essentially a measure of how much information one must "stack up" when trying to follow the use of a variable. It is often defined in terms of maximum or average span.

Characteristics that Affect Macro Understanding and Metrics to Measure Macro Complexity

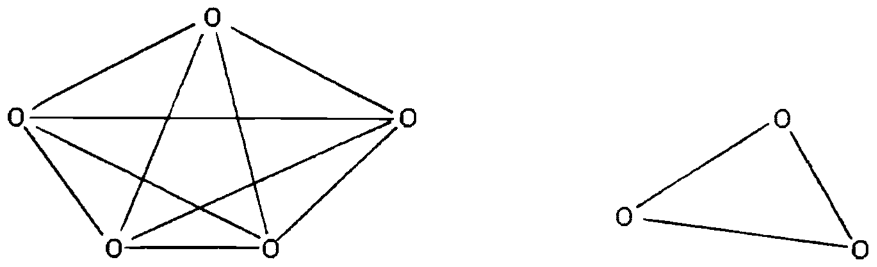
Macro understanding refers to the ease of understanding the overall operation of an entire system. A system, in general, is a set of objects that work together towards a common goal. The key to understanding the overall operation of a system is understanding the function of each part and the relationship among the parts.

In the case of a software system, an object is usually an individual piece of software. The attribute of an object is, in general, its function within the system.

Macro understanding is primarily affected by the attributes of the component objects, and their relationship to each other. The object attributes which affect understanding include the "granularity" (how many functions the object performs), and the inherent "complexity" involved in the function(s).

The number of objects in a system and the number of potential communication links affect macro understanding. Each potential communication link may introduce "side effects" which can affect other objects in the system (e.g., a modification to one part of the system may affect other parts).

Consider a system with n objects. Then, there are ((n-1)n)/2 potential communication links between objects. This is illustrated by the following example:



As can be seen, a system with 5 objects has 10 possible communication links, while a system with 3 objects has only 3 possible communication links.

In summary, we can identify four major characteristics which may affect the comprehension of a system:

(1) Granularity

(2) Inherent "complexity" of the functions

(3) Number of components

(4) Number of relationships (communication links) among objects

Comparatively little work has been done in the macro area since most software complexity metrics are based on the micro characteristics of a system. However, some macro complexity metrics have been developed.

Henry and Kafura's Information Flow Complexity

This metric, developed by Henry and Kafura [Henry and Kafura, 1981] is composed of two parts:

(1) The "detailed complexity" (i.e., micro complexity) of an individual object in a system.

(2) The complexity of the relationship between an object and each of the other objects within a system.

The detailed complexity of an object (subprogram) is determined by its size (i.e., the number of lines of code in it). Henry and Kafura justified this choice by citing work [Bowen, 1978] which observed that other characteristics of the object, while being more difficult to obtain, were not significantly better. The relationship complexity is determined by counting the "fan-in" and "fan-out" of an object.

The fan-in of an object is the number of different data items whose values can be affected by other objects in the system before being used by the object being examined. The fan-out of an object is the number of data items whose values can be affected by the object, and whose values can be subsequently used by other parts of the system. The fan-in and fan-out can be better described using the following definitions, paraphrased from [Henry, et al., 1981]:

DEF: The FAN-IN of a procedure A is the number of parameters associated with procedure A whose values are used before being changed by procedure A; the number of function calls by procedure A which return values through their names; the number of parameters associated with function calls in procedure A, which return values; and the number of global variables whose values are used before being changed by procedure A.

DEF: The FAN-OUT of a procedure A is the number of parameters associated with procedure A whose values are changed by procedure A; the number of values that can be returned through procedure A's name; the number of parameters associated with function calls in procedure A whose values are set by procedure A, before the call; and the number of global variables whose values are changed by procedure A.

Henry and Kafura considered the relationship complexity to have a greater effect on the total complexity of an object than the detailed complexity. Therefore, a weighting factor was used in the calculation of the object's complexity:

(fan-in * fan-out)**2 * length

This metric is used to determine the complexity of a single part of a system. However, it can be easily extended to assess the complexity of the entire system in several obvious ways:

$$TC = \sum_{i=1}^{\#\text{ of parts}} \text{complexity(part } \#i)$$

where each part could be a subprogram. This provides a measure of TC, the "total" system complexity. Henry and Kafura [Henry and Kafura, 1981] refer to this as "module complexity", and consider a module with multiple function/procedure definitions to be a system. Another approach to measuring overall system complexity might be:

Average Complexity = (total complexity)/(# of parts).

Other extensions are also possible.

Hall and Preiser's Combined Network Complexity

Hall and Preiser suggested this metric [Hall and Preiser, 1984] to measure the complexity of concurrent systems. However, we shall consider this metric in terms of non-concurrent systems.

They suggested that the system be represented as a network consisting of nodes that represent modules within the system, and edges which represent intermodule connections (invocation of modules and return of control). The resulting graph is referred to as the Network Control Graph.

Two different types of networks are possible. The first type of network, referred to as a hierarchy network, consists of a network of modules, in which control is always returned to the invoking module. This network is rooted at a single "monitor node". As an example, consider the following network:

$$\begin{tikzcd}
& M \arrow[dl, shift right] \arrow[dr, shift left] & \\
A \arrow[ur, shift right] & & B \arrow[ul, shift left]
\end{tikzcd}$$

Here, M invokes A, which returns control to M, and then M invokes B, which returns control to M.

The second type of network, a "pipeline network", is a network of modules in which control is transmitted directly to successor modules rather than being returned to the invoking module. Further, assume that control returns to the initial module after all the succesor modules have been activated. As an example, consider the following network:

$$\begin{tikzcd}
& A \arrow[dl] & & \\
B \arrow[drr] & & & \\
& & C \arrow[dr] & \\
& & & D \arrow[uuull]
\end{tikzcd}$$

Here, A invokes B, which invokes C, which invokes D. D returns control directly to A.

Hall and Preiser suggest two methods of assessing the "Network Complexity" of a system. The first method involves applying McCabe's Complexity Metric to the

Network Control Graph. In the case of the Hierarchy Network, the network complexity is a function of the number of modules in the system. This follows from the fact that a hierarchy network with n nodes will always have 2(n-1) edges - one in and one out of the monitor node for each of the n-1 non-monitor nodes. Thus, the McCabe complexity of a hierarchy network with n nodes is 2(n-1)-n+2 = n.

In the pipeline network, the network complexity is constant. Hall proves in her thesis [Hall, 1983], that for pipeline networks, the McCabe complexity is always two. This is based on the fact that a pipeline network forms a cycle. Thus, a pipeline network with n nodes, will also have n edges. Therefore, n-n+2 will always equal 2.

The second method of calculating the network complexity C(N) for a system N is based on the complexity of allocating resources for a given module. For a network with n module invocations ("paths"), this measure is calculated as:

$$C(N) = \sum_{i=1}^{n} (e[i] + \sum_{j=1}^{k} d[j] * r[i,j])$$

where k is the total number of resources that may be allocated; r[i,j] represents the resources that are used in the invocation of a path i (r[i,j]=0 if resource j is not used, and r[i,j]=1 if it is used); d[j] is the complexity of allocating resource j; and e[i] is the complexity of invoking and returning along each path i.

This measure is referred to as Hall and Preiser's "Generalized Measure" of Network Complexity. If it is assumed that no special resources must be allocated by a given module, then all r[i,j]=0. This results in the following expression:

$$C(N) = \sum_{i=1}^{\#\ \text{modules}} e[i]$$

which is simply the sum of the complexities of invoking and returning along each path in the system. By using this form of the calculation, and letting the complexity of each edge in the system be equal to 1 Hall and Preiser were able to compare the performance of the generalized metric to McCabe's metric (applied to networks).

Hall has shown that when using the simplified form of the generalized metric, the complexity for hierarchy networks with n modules is 2(n-1). This is because each module has an edge coming in from the monitor node and an edge returning to it. Thus, there are n-1 nodes, exclusive of the monitor node, each with an edge coming in, and an edge going out. Thus, the complexity is 2(n-1). For pipeline networks of n modules, the complexity is n since there are n nodes, each with one edge leaving.

This appears superior to the use of McCabe's metric for network complexity measurement, since the generalized metric can distinguish between different pipeline networks (recall McCabe's metric does not distinguish between pipeline networks).

Hall and Preiser point out that the simplest network structure would consist of a single module. Thus, all the network complexity may be moved into the code by using a single module. This would appear to be a very simple system when judged by the network metric, but in reality, it would be far from simple. To avoid this problem, a "combined measure" is presented by Hall and Preiser which uses the generalized measure described earlier, plus micro measures of complexity for each module in the system. This is somewhat similar to the metric suggested by Henry and Kafura [Henry and Kafura, 1981a], and is calculated as:

$$w[1] * NC + w[2] * \sum_{i=1}^{k} C[i]$$

where w[1] and w[2] are weighting factors (assigned by the user); NC is the network complexity, and C[i], (1<i<k) is a micro measure of complexity for each of the k modules in the system.

Any micro measure can be used for C[i], however, Hall and Preiser suggest the use of Storm and Preiser's index of complexity [Storm and Preiser, 1979], described earlier in this chapter.

A New Measure of Total Complexity

When maintaining a piece of software, the parts of a programmer's task that should be addressed by complexity metrics are:

(1) Examining each part of the system and their relationship to each other to determine which ones will be involved in the maintenance activity.

(2) Performing a detailed analysis of the subsytem(s)

involved in the maintenance activity in order to determine what changes are to be made and where.

A measure of the total complexity (TC) of a system, in relation to these tasks can be calculated as:

$$TC = \sum_{i=1}^{\#\,\text{subprograms}} [SC(i) * MC(i)]$$

where SC(i) is the system level complexity contributed by subprogram i, and MC(i) is the micro-level complexity contributed by subprogram i. The amount of system level complexity contributed by subprogram i is calculated as:

$$SC(i)=[Glob(i)*(\#subprograms-1)+Param(i)]*[1-DI(i)]$$

where:

Glob(i) - is the number of times global variables are used in subprogram i.

Param(i) - is the number of times parameters are used in subprogram i.

DI(i) - the "Documentation Index" for subprogram i. This is a measure of the quality of the subprogram's documentation. The better the documentation, the higher the index. A possible measure of this might be the ratio of comments to total lines within the subprogram. Obviously, if DI(i)=1, this implies that the subprogram is entirely comprised of comments, while in the case of DI(i)=0, no comments are present in the code.

The micro-complexity contributed by subprogram i is calculated using some measure of micro complexity, such as McCabe's VG. For example:

$$MC(i) = VG(i)$$

where VG(i) is McCabe's measure for subprogram i.

The components of this metric are based on the following reasoning. In order to locate the subsystem to be modified, the function of each module, plus the other modules that it affects must be determined. This is addressed by multiplying the number of data items used in each module by the number of other modules that they could affect. In a system comprised of n modules, a global data item usage could potentially affect n-1 of the modules besides the module it is in (it would be expected that the usage would affect that module). On the other hand, a parameter usage could affect only the invoking module (i.e., the one calling the module under consideration). Thus, the weight assigned to global variable usage should be n-1 and the weight assigned to parameter usage should be 1.

However, if information on the global/parameter usage of the module is readily available, this will clearly lessen the impact of the use of global items and parameters on comprehension. Thus, the System Level Complexity is weighted by the Documentation Index. This weighting ranges from 0 for the "best" documentation to 1 for the "worst". If "perfect documentation" were available, global item usage and parameters would have minimal effect on comprehension. Clearly, such a level of documentation would be very difficult to attain.

To allow automation of this metric, the Documentation Index for module i, can be defined using the following relation:

$$DI(i) = (DSL(i) - NCSL(i)) / DSL(i)$$

where DI(i) is the documentation index of module i, NCSL(i) is the number of Non-Commentary Source Lines in module i, and DSL(i) is the number of Delivered (total) Source Lines in module i. This in effect, is the ratio the number of comment lines and the number of total lines in subprogram i.

This clearly allows trivial manipulation of the metric by just inserting random blank lines. However, in a statistical context, it is felt that this method of computing the Documentation Index will be a reasonable measure of the documentation.

```
while SwapMade do begin
   SwapMade:=false;
   for i:=1 to n-1 do begin
      if x[i] > x[i+1] then begin
         hold:=x[i];
         x[i]:=x[i+1];
         x[i+1]:=hold;
         SwapMade:=true;
      end;
   end;
end;
```

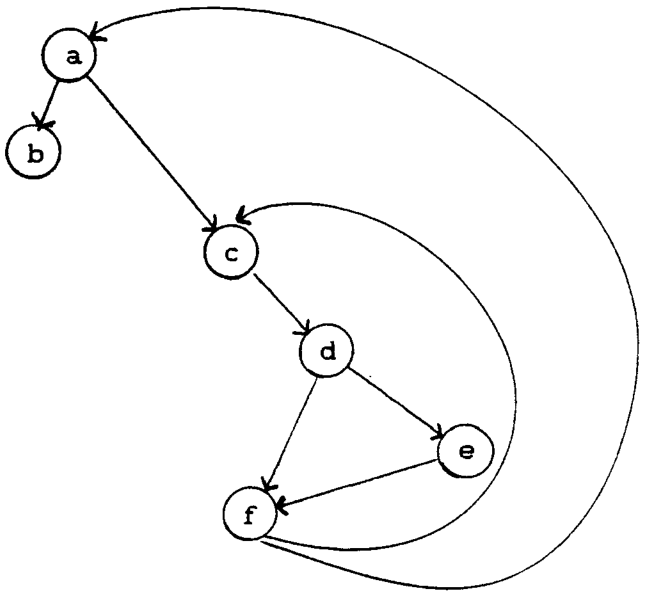


Figure 1. Example of program and associated flowgraph.

CHAPTER IV
SHARING INDUSTRIAL SOFTWARE COMPLEXITY DATA

In order to assess the reliability of the metrics discussed in earlier chapters of this dissertation, researchers need access to industrial software. However, most organizations are reluctant to provide software for use by outsiders. To overcome this difficulty, a "Reduced Form" was developed which allows researchers access to those features of the software needed for metric calculations, but which prohibits regeneration of the original software. This chapter presents the Reduced Form and discusses a survey of software managers which support the use of the Reduced Form.

Obtaining Data From Industry

Many organizations are reluctant to allow access to their code systems by "outsiders". Few organizations are prepared to circulate copies of code which has taken them thousands of man-hours to develop. Even though a researcher will not distribute the code to others, the mere giving of the code to someone may compromise any "trade secret" protection that it may possess [Graham, 1984]. Organizations dealing with scientific code systems may not necessarily care about the software itself, but may be very protective of algorithms, formulas, etc., which could be gleaned from a program.

In order to validate existing and future complexity metrics, researchers need access to "real world" programs. In this chapter we propose a Reduced Form that lists characteristics of interest to the complexity metric community, such as distribution of operators, variables, subprogram calls, etc., but prevents the reconstruction of the original source program.

Characteristics of Interest

A number of program characteristics are of interest to researchers working in the area of software complexity. It would be desirable to incorporate information about as many of these characteristics as possible into the Reduced Form.

Some of the characteristics that should be included in the Reduced Form are:

1. Information on program size. This could be partially satisfied by enumeration of operators and operands. We consider an operator to be a predefined keyword, symbol or user-defined routine that causes an action to be performed. An operand is an item acted upon by an operator. Besides enumerating each operator and operand (due to the difficulties involved in recognizing meaningful variable names, the operand names are of little interest for current complexity metrics), the number of times each is used should also be available. Information on the number of unique operators and operands appearing in the program could be used to compute the software science metrics [Halstead, 1977]. In addition, such an approach would also aid in the study of programming models [Zweben, 1977], [Elshoff, 1978]. Also of interest would be the number of lines of code in the program - in some standard form. Boehm's NonCommentary Source Lines (NCSL) [Boehm, 1981], seems to be quite useful, and easy to work with. This is a count of all lines of code, excluding comments and blank lines. Another useful standard is Delivered Source Lines, (DSL), in which all

comments, blank lines, etc. are included.

2. Information on a program's data structures and data usage. Information on the number and usage of local and global variables would be useful to support work along the lines of Henry and Kafura [Henry & Kafura, 1981] and Hall and Preiser [Hall & Preiser, 1984].

3. Information on the flow of control, both within the program and among the subprograms. This has been a rich area of research in the past [McCabe, 1976],[Chen, 1978]. An enumeration of operators provides information about the occurrences of those operators that affect flow of control within a procedure. Global program information about the interconnection between modules or subprograms has been the focus of considerable research [Henry & Kafura, 1981] [Yin & Winchester, 1978],[Gilb, 1977]. To support work in this area, the Reduced Form should provide a list of the subprogram invocations, and subprogram definitions, including formal parameters and global data.

A Reduced Form which includes this information for each subroutine or function in the code system, would have great flexibility, yet bar reproduction of the actual code through the use of aliases for variable, function and parameter names. Thus, organizations could distribute copies of it's software's Reduced Form, without fear of unauthorized use of the code. Also, such distribution would have little effect on trade secret protection, and specific algorithms and formulas would be impossible to recreate from the Reduced Form. The problems involved in attempting to reconstruct the software are addressed later in this chapter.

A Reduced Form for C

A Reduced Form for programs written in C has been implemented. This Reduced Form includes five parts:

1) An Identification Line which consists of the name of the subprogram (PROCEDURE in Figure 3).

2) A Declaratives Table, which lists the number of times each type of declarative was used in the subprogram (DCLS in Figure 3).

3) An Operand Table which lists an alias for each unique

string, constant and variable, its data type and usage (i.e., local, global or parameter), and the number of times the item was used (CONSTANTS, VARIABLES and STRINGS in Figure 3).

4) An Operator Table which lists the operators used and function calls made, plus the number of times each was used (FNCALLS and OPERATORS in Figure 3).

5) A Length Line, which indicates the number of source lines (DSL) and the number of non-commentary source lines (NCSL) in the subprogram (LENGTH in Figure 3).

An example of this Reduced Form is shown in Figures 2 and 3.

It should not be inferred that this version is the "best" Reduced Form. Rather, this version should be viewed as a prototype. Clearly, before a specific format is selected for wide use by others much input from the software metric research community must be obtained.

Extracting Software Characteristics From the Reduced Form

Almost all the characteristics used in computing the complexity metrics in Chapter II for a C program can be obtained from the Reduced Form.

1) Enumeration and count of operators and operands is available from the Operator and Operand Tables of the Reduced Form. The size of the program in "Lines of Code" (both NCSL and DSL) is available from the Length Line in the table. The number of statements may be roughly calculated by counting the number of semi-colons in the operator table.

2) The types of data structures and the manner in which data flows from subprogram to subprogram can be easily inferred from the Reduced Form tables. Variables that are global or formal parameters are flagged, and a call graph can quickly be constructed.

3) The number of decision points are available from the list of operators. A graph representing the calling structure of the system can easily be constructed

from the operator list and identification lines.

Prevention of Source Code Reconstruction

A large number of programs which may differ only in the arrangement of instructions can have the same Reduced Form. The extremely large number of programs which have equivalent Reduced Forms make it virtually impossible to reconstruct a program from its Reduced Form.

Recall that the Reduced Form contains, for each subprogram, the number and type of operands and operators, from which the number and type of statements can be inferred. From this information, the number of (not neccessarily valid) "programs" that can be constructed from a given Reduced Form can be calculated.

Suppose that program P has m subprograms P(1), ... ,P(m) and P(i) has S(i) statements. Thus, the number of total statements in P is:

$$S(P) = \sum_{i=1}^{m} S(i)$$

Then, the number of arrangements of statements of a program with the same reduced form as P is given by:

$$\sum_{i=1}^{m} S(i)!$$

To gain an idea of the magnitude of this number, let a program P consist of 50 statements, and 5 subprograms, P(1), P(2) ... P(5), each 10 statements long. Then the number of programs with the same reduced form as P is:

$$\sum_{i=1}^{5} 10! = 18{,}144{,}000$$

Furthermore, note that this calculation does not take into account the different ways that operators and operands may be arranged within a statement. To do so would require multiplying the number of arrangements of the statements in each subprogram by the factorial of the number of distinct operators and the factorial of the number of distinct operands. Many of the possible arrangements would result in valid programs that perform some computation. Thus, it would not be clear which version was, in fact, the original version. In addition, since all operands are "aliased" to be VARxxx, STRxxx, or

CONxxx, information concerning the value of constants or the use of a variable (via a meaningful variable name) cannot be obtained. Thus, these cannot be used to provide clues to determine if a given program generated from the Reduced Form is the original program.

In view of the very large number of possible programs, reproduction of the source code from the Reduced Form is quite unlikely except for trivial programs.

Acceptance by Organizations

A summary of a Reduced Form was sent to 39 individuals identified as holding positions ranging from Department Head to Vice President representing industrial organizations throughout the United States. Enclosed with the summary was an anonymous questionnaire (shown in Figure 4). This questionnaire asked three main questions:

(1) would the organization be willing to distribute copies of source code for software metric studies?

2) would they be willing to distribute copies

of the Reduced Form of their software for software metric studies?

3) would the organization be willing to distribute "performance data" describing their activities with the codes if such data were available?

In all, 14 responses were returned, a return rate of approximately 36%. The main results of the questionaires are shown in Table I. In short, over two-thirds of the respondents would be willing to provide data for use in metric studies. However, half of these respondents would agree to distribute data only in the Reduced Form. One respondent indicated that while his organization would be willing to provide source code, they would prefer supplying the Reduced Form. The responses also indicate that most organizations would be willing to provide "performance data". However, "painless" techniques must be developed to capture it as a number of respondents expressed concern with the amount of effort required to obtain the data.

The most common reasons cited by those who would not supply data for software metric studies was a concern for propietary information and trade secret protection. This either suggests that they did not understand the Reduced Form or that they did not trust it. The majority of those respondents who felt their organizations would be willing to provide data of any kind for software metric studies mentioned that anonymity would be required in any use of the data.

Number of respondents that felt their organization would share source code with researchers	5 (35%)
Number of respondents that felt their organization would not share source code, but would share the Reduced Form with researchers	5 (35%)
Number of respondents that felt their organization would share performance data with researchers	8 (57%)
Number of respondents that felt their organization would not share any data describing their code systems with researchers	4 (28%)

Table I. Major Results of the Survey

```
readfile (fname)
char *fname; {
    register FILE *f = fopen (fname, "r");
    if (f==0) {
        error ("Can't read %s", fname);
        return;
    }
    erasedb ();
    while (fgets(line,sizeof line,f)) {
        linelim = 0;
        if (line[0] != '#') yyparse ();
    }
    fclose (f);
    DBchanged = 0;
    linelim = -1;
}
```

Figure 2. Sample C Program.

```
PROCEDURE readfile()
DCLS
FILE              1
char              1
register              1
CONSTANTS
CON000020            4
CON000021            1
VARIABLES
VAR000131            1  unknown  unknown
VAR000128            4  FILE  local
VAR000127            4  FILE  formal parameter
VAR000129            3  unknown  unknown
VAR000008            2  int  global
STRINGS
STR000047            1
STR000046            1
STR000048            1
FNCALLS
erasedb()              1
error()              1
fclose()              1
fgets()              1
fopen()              1
yyparse()              1
OPERATORS
!=              1
""              2
''              1
*              2
,              4
-              1
;             10
=              4
==              1
[              1
if()              2
return              1
sizeof              1
while()              1
(              3
LENGTH          16           16
```

Figure 3. Reduced Form for C subprogram in Figure 2.

1. Would your organization be willing to supply source code for software metric studies?
   Yes ( ) No ( )

2. If the answer to #1 was "Yes", what restrictions in using the data would be required?

3. If the answer to #1 was "No", why?

4. Would your organization be willing to supply data in the Reduced Form for software metric studies?
   Yes ( ) No ( )

5. If the answer to #4 was "Yes", what restrictions in using the data would be required?

6. If the answer to #4 was "No", why?

7. Would your organization be willing to supply performance data for software metric studies, assuming it was available?
   Yes ( ) No ( )

8. If the answer to #7 was "Yes", what restrictions in using the data would be required?

9. If the answer to #7 was "No", why?

10. If your organization were to supply data in the Reduced Form for software metric studies, what language(s) would the software be written in?

11. Would your organization be interested in receiving the Reduced Form data?
    Yes ( ) No ( )

12. How would you classify your organization's primary applications?
    Scientific ( ) Commercial ( ) Systems ( )

Figure 4. Questionaire.

CHAPTER V
AN EMPIRICAL EVALUATION OF MACRO COMPLEXITY METRICS

To determine the feasibility of using the Reduced Form for assessing the performance of complexity metrics, we analyzed a moderate-sized software project of a large industrial organization. The data collected in this analysis was used to measure the performance of several popular complexity metrics as well as the macro complexity metric proposed in Chapter III. This chapter describes the data collected and compares the performance of each of the metrics observed in this study.

Use of the Reduced Form in Collecting Data in the Field

In the course of the survey described in an earlier chapter, a number of organizations became interested in our work. Two companies agreed to furnish us with data related to several projects they were involved in.

Both companies (Company A and Company B) were to provide data on the code systems being developed via the Reduced Form format, along with data relating to programmer performance. Company A provided data on a compiler project they were developing (Project A-1). Company B was to provide data on an operating system (Project B-1) and a compiler (Project B-2) they were developing.

Unfortunately, scheduling problems reduced Company B's commitment to the study. Therefore, the work in the remainder of this chapter will be limited to a study of Project A-1.

Programming Performance Data

For the purpose of our investigation the performance data had to meet several criteria:

(1) The data must be a measure of how well a programmer performed some task that required "comprehending" the program/system.

(2) Characteristics of programmer performance that managers are interested in had to be logically

inferrable from the performance data. If no conclusions could be drawn from the data on how well a programmer could be expected to do on some "pragmatic" activity (e.g., correcting errors, changing code, etc.), programming managers could not be expected to be too sympathetic towards our project.

(3) The data needed to be relatively easy to collect. Since we did not have direct access to any of the raw programmer performance data, we had to depend on others at the organization to capture the data for us and ensure its accuracy. It seems clear that as the effort to record data and verify its accuracy increases, the validity of the data collected decreases, especially if the individuals responsible for the data had little or no stake in our project.

Selection of Data

We decided to collect information on the number and type of errors discovered during system testing (roughly equivalent to integration testing). We felt that this data was both directly related to normal programmer activities, and dependent on program comprehension. In addition, it was the least difficult type of data to obtain, since most organizations tend to maintain this type of information anyway.

Data Collection

The error data collection involved obtaining seven pieces of information for each reported error correction:

1) The date the error correction was made.

2) The module changed to correct the error.

3) The system in which the error occurred.

4) The version of the system the error correction was

applied to.

5) The phase of development the error was introduced

6) How difficult the error was to repair.

7) The type of error that was fixed.

A sample error data collection form is shown in Figure 5. This form was initially presented to the organizations participating in the study. However, the organization was already collecting similar information using an on-line error correction tracking system. Thus, rather than using our form, they continued using their tracking system, and made this data available to us.

The error data was collected during a phase of development similar to what is known as "integration testing". All modules were considered to be working correctly in isolation, and their interactions with other modules were being tested.

Defining Logical Modules

A logical module was considered to be an identifiable group of subprograms that were logically related. The personnel collecting the data were given a great deal of freedom in identifying modules. The actual definition of a module is not as important as the consistency with which errors were attributed to a given module.

Distribution of Data

One employee within Company A, who was involved in the development of Project A-1, was identified as a liason between the Company and the researchers. This individual was responsible for summarizing the data available from the automated bug tracking system, and running the tool we supplied to extract the Reduced Form from C source code.

Project A-1 went through 5 major versions during development (V1.1 through V1.5). As each version was completed, a period of system testing was performed. The system consisted of 39 total modules. However, error data on only 23 of the modules was available. Further, three of the modules were not suitable for the study, for various reasons:

- o Module PARSE.C was generated via LEX. Since the module was automatically generated, we felt that the actual work would be done on the LEX source (which was not available to us).

- o The two modules DYNLD.C and LISTER.C were essential for the module testing of each of the other modules. As a result, most of the errors in these modules were found before formal system testing began. Therefore, we felt that the number of errors found in these modules would not be indicative of the actual number of errors in the modules.

Thus, our study involved the remaining 20 modules.

We used the Reduced Form data from Version V1.4 in our study since before version V1.4, many of the modules were not completed, thus a number of errors which were eventually recorded were due to modules which did not even exist in Versions V1.1 through V1.3, except perhaps as stubs.

The Data

It was agreed that Organization A was to make available the following information:

- o Reduced Form data describing the characteristics of the 20 modules analyzed.

- o Error resolution data describing the error resolutions performed by the programmers developing the software. This data summarized the resolutions for each module, in terms of:

    - Severity of the error (critical, serious, moderate, trivial).

    - Phase in which the error was introduced (requirements, design, coding, enhancement, side effect from change).

- Difficulty of fix (easy, moderate, hard).

- Type of error (data manipulation, logic, I/O, module interface, data definition, or other).

- Total number of lines changed in the module due to resolutions.

- total number of resolutions for the module

Certain unavoidable ambiguities were encountered by the personnel who recorded this data. For example, one bug may be classified as either easy to fix or hard to fix, depending on if the time required to track down the bug was included, or just the time requried to actually make the fix. Likewise, a module interface error could be mistaken for a data definition error or a data manipulation error. The significance of these ambiguities did not become obvious until after the study was finished and the results were discussed with our liason to the organization. Even if we had made a greater effort to eliminate ambiguities, they would still be possible unless a single person were to record the data, and make special efforts to maintain consistency.

The Metrics Analyzed

The intention was to compare the relationship between each of several macro complexity metric measurements and the number of resolutions (and hence, number of errors) for the 20 modules analyzed.

The metrics described in Chapter III are just a sample of metrics which are currently in use. In order to keep the scope of the analysis to a manageable level, it was decided to limit the analysis to six of the metrics described in Chapter III. The metrics involved in the analysis include some of the most commonly used metrics, as well as several macro complexity metrics:

(1) Henry and Kafura's metric (HNK). The Reduced Form does not lend itself to an exact calculation of Henry and Kafura's metric, since some of the information needed is not available (e.g., it is not clear which parameters are simply used, and which ones are changed - this is necessary to help prevent reconstruction of the code). Even so, an aggregate, rather than individual measure of FanIn/FanOut may be obtained. Thus, the metric as calculated ends up

to be:

```
(FanIn + FanOut)**2 * Line of Code
```

This was approximated by summing the number of unique global variables, parameters and functions used in each procedure, squaring this sum, and multiplying by the total number of lines in the module.

(2) Hall and Preiser's metric (HNP). Hall and Preiser's metric can be readily computed from the Reduced Form. However, it is not clear exactly what the weighting coefficients w[1] and w[2] should be. Hall and Preiser used a value of one for both weights in all examples in their paper. The purpose behind the weights seems twofold:

(i) The two different measures (i.e., network and micro complexities) may be quite different (e.g., 2(n-1) vs. lines of code differ by a much greater amount than 2(n-1) vs. average VG per function definition). The weights may serve to adjust the complexity values so they are compatible (at least the same order of magnitude). Otherwise one component may overwhelm the other component, just by the nature of the measure.

(ii) It seems that the larger the share of total complexity one attempts to put into the network, the more important the network complexity should be. Likewise, the larger the share of total complexity one attempts to put into the individual procedures, the more important the detailed complexity should be. Thus, the network complexity should be heavily weighted when the majority of the total complexity is due to the network, and lightly weigthed otherwise. Likewise, when the majority of the total complexity is due to micro-complexity, the micro-complexity should be heavily weighted.

To account for this behavior with the metric, the coefficient w[1] was set to:

4.36 / average VG per function

and the coefficient w[2] was set to:

average VG per function / 4.36

where 4.36 is the average VG per function definition for the entire suite of modules and "average VG per function" is the average VG per function definition within a particular module.

Thus, if a particular module has a lower "relative micro complexity" than the average, it suggests that the complexity has been shifted to the network structure, and thus the network complexity is more heavily weighted than the micro measure of complexity. Conversely, if a particular module has a higher "relative micro complexity" than the average, it suggests that the complexity has been shifted into the function definitions, and thus the micro complexity is more heavily weighted than the network complexity. The network complexity, NC was set to:

2(Number of Procedures - 1)

and the micro measure of complexity for each component module was set equal to the total VG for that module.

(3) The new metric introduced in an earlier chapter (HARR) was fairly simple to compute from the Reduced Form. For each subprogram definition in the module the product of the total global variable usage and the number of procedures in the module (minus 1) was added to the total parameter usage. This sum was then multiplied by a "computed Documentation Index"

calculated as:

$$\frac{\text{number of comment lines in subprogram}}{\text{total number of lines in subprogram}}$$

This result was then multiplied by the VG measure for the subprogram.

This product, for each subprogram definition in the module, was summed to arrive at a total measure of complexity for the entire module.

(4) Halstead's Effort (E) metric. While this metric is not a true system level metric, it is popular enough to warrant its inclusion in this study. The calculation of this metric from the Reduced Form is trivial. Treating each procedure independently, the total number of operators (N1) and operands (N2) were summed to arrive at N for that procedure. The total number of unique operators (n1) and operands (n2) were likewise summed to arrive at n for that procedure. The N and n values for each procedure were then summed over the entire module to yield the program "Volume" (V) for that module:

$$V = N (\log 2\ n)$$

which is then divided by an approximation to "Level"

(L′) suggested by Halstead:

$$L' = (2/n1) * (n2/N2)$$

yielding:

$$E = [N (\log2\ n)] / [(2/n1) * (n2/N2)].$$

(5) McCabe′s Cyclomatic Complexity (VG) metric. Like Halstead′s Effort metric, the cyclomatic complexity is not a true macro level complexity metric. However, like Effort, it is so popular, it would be difficult to justify leaving it out of the study. This metric was calculated from the Reduced Form by summing the occurrences of the C operators:

- for
- if
- elseif
- while
- case
- break

- exit

(6) A simple count of source lines (DSL). Since studies have shown this to be one of the most accurate measures of program complexity, it is also included in the metric analysis.

The Error Data

A total of 275 error resolutions were reported during the project. A breakdown of the error data by severity, phase of introduction, difficulty of fix, and type of error is given in Table II.

As can be seen, the majority of the resolution reports classified the errors as either of severe or moderate severity (28.7% and 40.4%). Also, the majority of the errors were introduced during the coding phase of the project (39.6%), and the requirements phase (32.0%). However, a significant number of errors were introduced as side effects to previous fixes (17.8%). The rest of the errors were either introduced during the design phase (8.0%) or during the process of making an enhancement

(2.5%).

An overwhelming majority of the error resolutions were classified as easy to correct (68.7%). A relatively insignificant number of errors were considered to be hard to correct (5.5%). The remaining errors were considered to be of moderate difficulty to correct (25.8%).

The bulk of the errors were classified as either data manipulation errors (34.5%) or logic errors (39.3%). The remainder of the errors were input/output (1.8%), module interface (2.9%) or data definition (3.6%) errors. A significant portion of the errors (17.8%) could not be associated with one of the pre-defined error types and were attributed to the "OTHER" category.

These observations are quite different than those obtained in [Basili and Perricone, 1984]. In a study of a 90,000 line FORTRAN application, their results suggest that a large portion of the errors (almost 40%) were classified as "INTERFACE" errors (similar to our "MODULE INTERFACE" errors). This could be due to the differing characteristics of each language. For example, FORTRAN uses the COMMON statement which allows global variables to be renamed, while global variables retain their original names in C. Also, the FORTRAN project was about two and a half times as large as the C project (90,000 lines vs. 38,000 lines). Thus, the FORTRAN project would be expected to contain more subprograms and functions than the C project. Another possibility is that the two projects differed in terms of design and development techniques.

A summary of the errors observed in our study for each module is provided by Tables III and IV.

The Reduced Form Data and Metric Values

Table V gives the values of the metrics computed from each module from the Reduced Form data. The correlation between each metric and the other metrics is shown in Table VI. As can be seen, some of the metrics are significantly correlated with each other. The metric VG seems to be especially highly correlated with other metrics. This is no doubt because most of the metrics contain a very large VG component. The only exception to this rule is E which does not contain VG as a component, but which is nonetheless correlated at the .93 level.

It is interesting to note that the high correlations do not appear to be "transitive" - that is, if A is highly correlated to B and C, B and C are not neccessarily highly correlated. For example, HARR is correlated with VG at .91, and PRC is correlated with VG at .92, however HARR and PRC are correlated at only .79. This lack of transitivity, precludes the use of any metric, except VG itself, as a proxy for any of the other metrics.

Relationship of the Metrics to the Errors

An extensive statistical analysis was performed on the data using the Statistical Package for the Social Sciences (SPSS, 1983). Unless otherwise stated, all measures of relationship make use of the Pearson Product-Moment Correlation Coefficient as computed by SPSS.

As can be seen from Table VII, there is a significant relationship between the metrics and the number of errors found in a module. However, the performance of the metrics is of uneven quality, ranging from .82 (HARR) down to .62 (HNK). Lines of code performed relatively well with a correlation of .76 to errors. This suggests that even though many of the metrics are significantly correlated, some still perform appreciably better than others.

Each metric reflects certain characteristics of the software being measured. Some of these characteristics may or may not overlap. Likewise, there are many different types of errors which can occur. We have identified six categories of errors, though many more are no doubt possible. These categories are: Data Manipulation (Data), Logical (Log), Input/Output (I/O), Module Interface (Xif), Data Definition (Def) and Other (Oth).

It would seem reasonable to expect the various characteristics identified by each metric to be related to different types of errors. For example, one might associate highly complex control structuring with logical errors, or complex data structures with data definition errors. One would also expect high module interconnection complexity to be present in software with a high rate of module interface errors. As a consequence we would not expect any one metric to be highly correlated with all errors. Rather, each metric should be highly correlated with one or two of the various error types.

However, as can be seen from Table VIII, the expected relationships were not observed in the study. The only category of error type that was significantly correlated with any of the metrics was DATA MANIPULATION errors. Further, this was significantly correlated with all the metrics (however, DSL was significant at only the .01 level).

The significant relationships expected (e.g., HARR, HNK and PRC should be highly correlated with MODULE INTERFACE errors and VG with LOGIC errors) did not materialize. This could be attributed to a lack of sufficient numbers of MODULE INTERFACE errors to obtain a meaningful correlation. However, LOGIC errors made up almost 40% of the total errors observed. Thus, we cannot suggest that insufficient LOGIC errors were available to obtain a meaningful correlation.

The relationship between the HARR metric and DATA MANIPULATION errors was also quite unexpected. The HARR metric was highly correlated (.93) with DATA MANIPULATION errors - in fact, the HARR correlation with DATA MANIPULATION errors was the highest correlation observed in the study. We suspect that these unexpected results may be at least partly due to some of the MODULE INTERFACE errors being incorrectly recorded as DATA MANIPULATION errors since they involved the manipulation of global data and/or parameters.

In a similar manner, the only phase of error origination whose observations are significantly correlated with any of the metrics was the REQUIREMENTS phase. These errors had an especially high correlation with the HARR metric (.86). If we assume that the modules are functionally decomposed (i.e., individual functions are isolated in separate subprograms or groups of programs) this suggests that requirements with many interconnected functions are more prone to errors. The more modest correlation of .67 between REQUIREMENTS errors and number of procedures suggests, however, that simply numbers of functions in the REQUIREMENTS may not have a similar effect.

It is possible that the effect of module size has some impact on the observed correlations. Thus, the error counts were "normalized" by the size of the modules they were associated with. The number of lines of code in each module was divided by the error count for that module, resulting in an "error density" figure (i.e., average number of lines "between errors"). The results of the normalization are shown in Table IX. Again however, only the errors classified as DATA MANIPULATION errors were significantly correlated with any of the metrics. The correlation was again especially high with the HARR metric (.91). However, in this case, the correlations between DATA MANIPULATION errors and the other metrics was not quite as high. In fact, the correlation between DSL and DATA MANIPULATION errors was not even significant.

```
-----------------------------------------------------
|  Error Report                                     |
|                                                   |
|  Date: ___/___/___          Origin of Error       |
|  Module Changed: ________   Requirements  ___     |
|  System:_________________   Design        ___     |
|  Version: _______________   Coding        ___     |
|  Comments:                                        |
|                             Repair Difficulty     |
|                             Simple        ___     |
|                             Moderate      ___     |
|                             Hard          ___     |
|                                                   |
|                             Type of Error         |
|                             Data Manipulation ___ |
|                             Logic             ___ |
|                             I/O               ___ |
|                             Module Interface  ___ |
|                             Data Definition   ___ |
|                             Other             ___ |
|                                                   |
|                                                   |
-----------------------------------------------------
```

Figure 5. Sample Error Data Collection Form

Critical	** (13.8%)
Severe	**** (28.7%)
Moderate	****** (40.4%)
Trivial	*** (17.1%)

Breakdown of errors by severity.

Requirements	**** (32.0%)
Design	* (8.0%)
Coding	****** (39.6%)
Enhancement	* (2.5%)
Side Effects	*** (17.8%)

Breakdown of errors by phase of introduction.

Easy	*********** (68.7%)
Moderate	**** (25.8%)
Hard	* (5.5%)

Breakdown of errors by difficulty of fix.

Data Manipulation	***** (34.5%)
Logic	****** (39.3%)
I/O	* (1.8%)
Module Interface	* (2.9%)
Data Definition	* (3.6%)
Other Error	*** (17.8%)

Breakdown of errors by type.

Table II. Breakdown of Error Data.

Module	Error Severity				Fix Difficulty			
	Crit.	Sev.	Mod.	Triv.	Easy	Semi	Hard	Total Bugs
alc	1	3	1	2	5	2	0	7
aprm	3	6	20	6	29	6	0	35
bcmem	0	1	0	0	1	0	0	1
cmnd	1	2	1	6	9	1	0	10
cntrl	0	1	3	2	5	1	0	6
grphrm	1	3	6	3	9	3	1	13
grphrm1	1	4	11	5	16	4	1	21
grphrm3	3	3	6	1	4	7	2	13
intrp	0	1	0	0	1	0	0	1
msrmhd	1	2	0	0	3	0	0	3
msrmtl	0	1	3	0	2	2	0	4
math	1	0	2	0	1	2	0	3
mathfn	3	5	6	1	15	0	0	15
mtrx	4	8	11	3	12	12	2	26
pug	7	17	18	2	16	22	6	44
strg	3	2	2	1	7	1	0	8
symtab	1	2	0	0	2	1	0	3
user	0	1	8	5	13	1	0	14
bscio	5	9	8	5	23	3	1	27
extio	3	8	5	5	16	3	2	21

Table III. Summary of error data for each module (Part 1).

Module	Phase of Introduction					Type of Error						Total
	Req	Des	Cod	Enh	SEf	Dat	Log	I/O	Xfc	Def	Oth	Bugs
alc	4	0	1	1	1	2	5	0	0	0	0	7
aprm	2	2	19	2	10	13	14	0	2	5	1	35
bcmem	1	0	0	0	0	1	0	0	0	0	0	1
cmnd	7	0	1	0	2	0	9	0	0	0	1	10
cntrl	6	0	0	0	0	0	6	0	0	0	0	6
grphrm	2	2	3	0	6	0	5	1	1	0	6	13
grphrm1	1	1	12	1	6	2	6	0	0	0	13	21
grphrm3	0	0	10	0	3	0	4	0	2	0	7	13
intrp	0	0	1	0	0	0	1	0	0	0	0	1
msrmhd	0	0	3	0	0	0	3	0	0	0	0	3
msrmtl	1	0	3	0	0	0	3	0	0	0	1	4
math	0	0	3	0	0	0	1	0	0	0	2	3
mathfn	0	5	6	1	3	2	4	0	0	3	6	15
mtrx	0	9	16	0	1	2	15	4	0	2	3	26
pug	27	0	9	0	8	32	8	0	1	0	3	44
strg	0	0	6	0	2	1	5	0	0	0	2	8
symtab	2	0	0	0	1	0	2	0	1	0	0	3
user	0	1	10	2	1	0	10	0	1	0	3	14
bscio	17	0	5	0	5	22	5	0	0	0	0	27
extio	18	2	1	0	0	18	2	0	0	0	1	21

Table IV. Summary of error data for each module

(Part 2).

Module	DSL	PRC	E	VG	HARR	HNK	HNP
alc	1412	31	38543544	158	159792	695839232	255
aprm	3299	51	157958528	287	574331	1762856960	500
bcmem	485	20	1977313	18	6805	10480365	12
cmnd	1098	26	14182032	91	47994	157717824	113
cntrl	1083	36	18733990	79	63306	201179168	75
grphrm	2805	3	163050	13	593	9436020	17
grphrm1	2569	4	1389543	43	19505	93719688	121
grphrm3	1214	1	8945	3	5	438254	2
intrp	207	2	181952	21	226	268272	55
msrmhd	2318	25	24125136	157	85470	363499488	295
msrmtl	1772	22	27613144	147	87024	203640016	290
math	681	24	13049362	97	75570	54926736	133
mathfn	1263	43	4966103	48	4336	92072704	34
mtrx	4497	87	269206880	356	828132	5138996224	496
pug	3058	76	187840016	478	2068100	3794965760	906
strg	1098	29	36697336	98	161122	71958528	119
symtab	1209	20	12297011	89	102644	100976888	130
user	992	31	20581646	98	83419	129278432	115
bscio	3725	84	294520704	383	1146895	6688609792	574
extio	3505	103	323265952	468	931571	7886249984	700

Table V. Metrics for each module.

	HARR	HNP	HNK	DSL	VG	E
HNP	.90**					
HNK	.77**	.84**				
DSL	.67*	.70**	.77**			
VG	.91**	.95**	.89**	.77**		
E	.80**	.87**	.98**	.81**	.93**	
PRC	.79**	.92**	.91**	.70**	.92**	.93**

* - Significance < .01 ** - Significance. < .001

Table VI. Correlations among the metrics.

	BUGS
HARR	.82**
HNP	.75**
HNK	.62*
DSL	.76**
VG	.73**
E	.69**
PRC	.64*

* Significance < .01 ** Significance < .001

Table VII. Correlation of metrics with total bug resolutions.

	HARR	HNK	DSL	VG	E	PRC	HNP
Crit	.85**	.66*	.64*	.73**	.70**	.68**	.76**
Sev	.94**	.72**	.71**	.82**	.74**	.73**	.86**
Mod	.64*	.39	.65*	.55	.49	.44	.57*
Triv	.29	.44	.52	.39	.47	.40	.31
Req	.86**	.70**	.45	.77**	.66*	.67**	.77**
Des	.15	.34	.52	.21	.39	.43	.32
Cod	.30	.18	.53	.29	.31	.23	.27
Enh	-.11	-.15	.02	-.03	-.07	-.02	.00
SEf	.45	.15	.49	.28	.23	.13	.31
Easy	.55	.56*	.68**	.59*	.63*	.59*	.61*
Semi	.81**	.43	.55	.62*	.48	.45	.61*
Hard	.82**	.52	.51	.62*	.50	.47	.63*
Data	.93**	.74**	.59*	.84**	.75**	.73**	.86**
Log	.33	.23	.53	.37	.36	.37	.32
I/O	.18	.32	.53	.24	.38	.33	.19
XInt	.11	-.10	.11	.00	-.02	-.13	-.04
Def	.09	.09	.33	.19	.22	.26	.29
Oth	-.13	-.18	.14	-.26	-.21	-.28	-.21

* - Significance < .01 ** - Significance < .001

Table VIII. Correlations between metrics bug categories.

	HARR	HNK	DSL	VG	E	PRC	HNP
Crit	.33	.14	.08	.19	.16	.22	.24
Sev	.50	.23	.04	.27	.21	.23	.39
Mod	.25	.02	.17	.14	.08	.13	.19
Trv	-.07	-.02	-.03	-.02	-.02	.04	-.08
Req	.63*	.45	.20	.53	.40	.49	.54
Des	-.03	.08	.19	.00	.09	.24	.19
Cod	-.11	-.22	-.14	-.18	-.16	-.19	-.19
Enh	-.16	-.18	-.14	-.11	-.15	-.03	-.06
SEf	.20	-.08	.21	.01	-.02	-.06	.06
Easy	.03	-.01	-.03	.00	.02	.14	.10
Semi	.55	.17	.23	.33	.20	.16	.30
Hard	.60*	.32	.32	.37	.30	.22	.37
Data	.91**	.66*	.51	.79**	.67**	.68**	.85**
Log	-.19	-.29	-.30	-.21	-.23	-.14	-.25
I/O	.15	.29	.53	.20	.34	.28	.15
Xif	-.07	-.19	-.10	-.17	-.16	-.25	-.24
Def	-.02	-.02	.11	.01	.04	.17	.21
Oth	-.28	-.32	-.15	-.41	-.35	-.35	-.34
BUGS	.38	.11	.12	.21	.15	.23	.28

* - Significance < .01 ** - Significance < .001

Table IX. Correlations among the metrics and bug density categories (number of errors normalized by number of lines of code in the modules).

CHAPTER VI
CONCLUSIONS AND FUTURE WORK

Conclusions

A number of important observations and suggestions have been made in this dissertation. This work has allowed us to conclude three major points:

(1) Industrial software complexity data, while ordinarily difficult to obtain, can be acquired by using a tool such as the Reduced Form, which eliminates any possibility of compromising the code and/or its formulas. This data can then be used to assess the performance of various software complexity metrics when applied to realistic software systems.

(2) A simple count of lines of source code is a reasonable measure of software complexity. In the study, lines of code performed better than many of the popular metrics. In addition to being highly correlated with the number of bugs found in the sample, lines of source code was also the easiest metric to compute. For these reasons, lines of code

probably remains the metric of choice in most situations.

(3) A new metric (HARR) which considers not only the micro aspects of complexity, but also the macro aspects, performs substantially better than any of the other metrics considered, including lines of code. However, the HARR metric is much more "expensive" to compute than lines of code, and thus may not be as feasible to use as other, less accurate metrics. Nevertheless, the metric's performance suggests that it is a promising new metric which should be examined further.

Future Work

Much more work must be performed before conclusive results can be obtained. However, the work reported in this dissertation is an important step in the right direction.

Rather than simply fine-tuning metrics to achieve higher and higher correlations, future work should look towards which specific characteristics are being measured, and why they are important. Of course, we still do not know which characteristics will be the true key to assessing software complexity, and thus must continue to study new and existing complexity metrics. We must study, from a psychological viewpoint, how the various characteristics being measured affect programmer understanding.

Future work should examine other application areas and programming languages, as well as other measures of programmer performance (i.e., other measures beside just "bugs"). Also, the new macro metric could be modified to use other measures besides VG as the micro component. Before this can be done, however, new methods of normalizing the measures must be developed so the various components will be of comparable magnitude.

In addition, the Reduced Form should be expanded for additional programming languages (we have already initiated work on a FORTRAN Reduced Form), and efforts to obtain a data base of Reduced Form data will be made so other researchers can have ready access to data for complexity metric studies. Most importantly, we hope to be able to develop a Reduced Form for performance data.

This will allow complexity metric evaluations using code developed in many different environments for many different applications.

BIBLIOGRAPHY

o Bailey, C. and W. Dingee, "A Software Study Using Halstead Metrics", ACM SIGMETRICS Performance Evaluation Review, Spring 1981, pp 189-197.

o Basili, V. and B. Perricone, "Software Errors and Complexity: An Empirical Investigation", Communications of the ACM, January 1984, pp 42-52.

o Basili, V. and D. Hutchens, "An Empirical Study of a Syntactic Complexity Family", IEEE Transactions on Software Engineering, Vol SE-9, November 1983. pp 664-672.

o Basili, V. and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory", ACM SIGMETRICS Performance Evaluation Review, Spring, 1981, pp 95-99.

o Basili, V., R. Selby and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects", IEEE Transactions on Software Engineering, Vol SE-9, Novemeber 1983, pp 652-663.

o Boehm, B., Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.

o Bowen, J., "Are Current Approaches Sufficient for Measuring Software Quality?", Proceedings of the ACM Software Quality Workshop, 1978, pp 148-155.

o Chen, E., "Program Complexity and Programmer Productivity", IEEE Transactions on Software Engineering, May, 1978, pp 187-194.

o Cook, C. and W. Harrison, "A Survey of Graph Theoretic Computer Program Complexity Measures", Congressus Numerantium, May 1984, pp 197-217.

o Curtis, B., S. Sheppard and P. Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics", Proceedings, Fourth International Conference on Software Engineering, 1979a, pp 356-360.

o Curtis, B., S. Sheppard, P. Milliman, M. Borst and T. Love, "Measuring the Psychological Complexity of

Software Maintenance Tasks With the Halstead and McCabe Metrics", IEEE Transactions on Software Engineering, March, 1979b, pp 95-104.

- Elshoff, J., "An Analysis of Some Commercial PL/I Programs", IEEE Transactions on Software Engineering, Vol SE-2, June, 1976, pp 113-120.

- Elshoff, J., "A Study of the Structural Composition of PL/I Programs", ACM SIGPLAN Notices, Vol 13, June, 1978, pp 29-37.

- Feuer, A. and E. Fowlkes, "Some Results from an Empirical Study of Computer Software", Proceedings of the Fourth International Conference on Software Engineering, 1979, pp 351-355.

- Gilb, T., Software Metrics, Winthrop Publishers, Cambridge MA, 1977.

- Gordon, R., "Measuring Improvements in Program Clarity", IEEE Transactions on Software Engineering, March 1979, pp 79-90.

- Graham, R., "The Legal Protection of Computer Systems", Communications of the ACM, May 1984, pp 422-426.

- Gremillion, L., "Determinants of Program Repair Maintenance Requirements", Communications of the ACM, August 1984, pp 826-832.

- Hall, N. and S. Preiser, "Combined Network Complexity Measures", IBM Journal of Research and Development, January 1984, pp 15-27.

- Hall, N., "Complexity Measures for Systems Design", PhD Dissertation, Department of Mathematics, Polytechnic Institute of New York, June 1983.

- Halstead, M., Elements of Software Science, Elsevier, New York 1977.

- Harrison, W., K. Magel, R. Kluczny and A. DeKock, "Applying Software Complexity Metrics to Program Maintenance", IEEE Computer, September, 1982, pp 65-79.

- Harrison, W., Applying McCabe's Complexity Measure to Multiple Exit Programs, Software - Practice and

Experience, October 1984, pp 1004-1007.

o Henry, S., D. Kafura and K. Harris, "On the Relationships Among Three Software Metrics", ACM SIGMETRICS Performance Evaluation Review, Spring 1981, pp 81-88.

o Henry, S. and D. Kafura, "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, September, 1981, pp 510-518.

o Jorgensen, A., "A Methodology for Measuring the Readability and Modifiability of Computer Programs", BIT, #20, 1980, pp 394-405.

o Knuth, D. "Structured Programming With GOTO Statements", ACM Computing Surveys, December 1974, pp 261-301.

o McCabe, T., "A Complexity Measure", IEEE Transactions on Software Engineering, December, 1976, pp 308-320.

o Myers, G., "An Extension to the Cyclomatic Measure of Program Complexity", ACM SIGPLAN Notices, October, 1977, pp 61-64.

o SPSS, SPSSx User's Guide, McGraw-Hill Book Company, New York, 1983.

o Storm, I. and S. Preiser, "An Index of Complexity for Structured Programming", IEEE Proceedings of the Workshop on Quantitative Software Models, 1979, pp 130-133.

o Yin, B. and J. Winchester, "The Establishment and Use of Measures to Evaluate the Quality of Software Design", ACM SIGSOFT Software Engineering Notes. 3,5, November 1978, pp 45-52.

o Zweben, S., "A Study of the Physical Structure of Algorithms", IEEE Transactions on Software Engineering, Vol SE-3, pp 250-258, May, 1977.