1990-06

An empirical study of the fault-predictive ability of software control-structure metrics

Almeida, Alberto Teixeira Bigotte de

Monterey, California: Naval Postgraduate School

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN EMPIRICAL STUDY OF THE FAULT-PREDICTIVE ABILITY OF SOFTWARE CONTROL-STRUCTURE METRICS

by

Alberto Teixeira Bigotte de Almeida

June 1990

Thesis Advisor: Timothy Shimeall

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Technology Curriculum, Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 37	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
AN EMPIRICAL STUDY OF THE FAULT-PREDICTIVE ABILITY OF SOFTWARE CONTROL-STRUCTURE METRICS(U)

12. PERSONAL AUTHOR(S)
ALMEIDA, ALBERTO T. B.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM ____ TO ____	June 1990	81

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Software Metrics, Text-based metrics, Faults, Testing, Empirical Studies

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
The increasing cost and complexity of software in recent years is causing a growing interest in the development of measurement technology to evaluate, predict and compare software complexity. Metrics can be used throughout all the development cycle providing valuable information to the software developers in order to enhance the final products. The goal of this thesis is to verify empirically the fault-predictive ability of some software complexity metrics and specifically their usefulness during the testing phase.

A set of eight programs, varying in length from 1,186 to 2,489 lines of Pascal code with 157 faults identified with specific modules, provided the data for this study. The results of the analysis of the programs using four metrics, cyclomatic complexity, bandwidth, nested complexity and the number of statements, show that control-structure metrics can be effectively used to detect the more fault-prone modules. The nested complexity of the modules seems to have some relation with the number of faults caused by wrong use of variables and overrestrictive input checks. These observations can be particularly useful during the testing phase because testers can use control-structure metrics to predict not only the modules that may cause more problems but also the more frequent types of faults and use the metrics to guide the choice of testing techniques.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Timothy J. Shimeall	22b. TELEPHONE (Include Area Code): (408) 646-2509; 22c. OFFICE SYMBOL: CS/Sm (52Sm)

Approved for public release; distribution is unlimited.

AN EMPIRICAL STUDY OF THE FAULT-PREDICTIVE ABILITY OF SOFTWARE CONTROL-STRUCTURE METRICS

by

Alberto Teixeira Bigotte de Almeida
Lieutenant, Portuguese Navy

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: [redacted]

Alberto Teixeira Bigotte de Almeida

Approved by: [redacted]

Timothy J. Shimeall, Thesis Advisor

[redacted]

Leigh W. Bradbury, Second Reader

[redacted]

Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

The increasing cost and complexity of software in recent years is causing a growing interest in the development of measurement technology to evaluate, predict and compare software complexity. Metrics can be used throughout all the development cycle providing valuable information to the software developers in order to enhance the final products. The goal of this thesis is to verify empirically the fault-predictive ability of some software complexity metrics and specifically their usefulness during the testing phase.

A set of eight programs, varying in length from 1,186 to 2,489 lines of Pascal code with 157 faults identified with specific modules, provided the data for this study. The results of the analysis of the programs using four metrics, cyclomatic complexity, bandwidth, nested complexity and number of statements, show that control-structure metrics can be effectively used to detect the more fault-prone modules. The nested complexity of the modules seems to have some relation with the number of faults caused by wrong use of variables and overrestrictive input checks. These observations can be particularly useful during the testing phase because testers can use control-structure metrics to predict not only the modules that may cause more problems but also the more frequent types of faults and use the metrics to guide the choice of testing techniques.

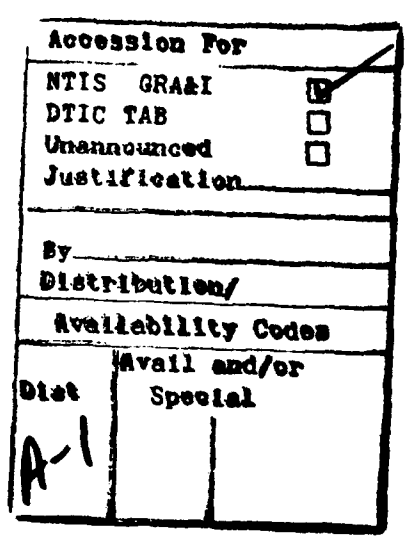

TABLE OF CONTENTS

I. INTRODUCTION

Software testing and maintenance has been estimated to consume 70% of the overall software development effort [1]. Testing and debugging costs range from 50% to 80% of the cost of producing a first working version of a software package [2]. Thus, the development of effective error detection techniques is one of the most important issues in the effort to reduce the cost and to increase the quality of software.

There have been many different approaches to software testing and error detection such as structural testing, functional testing or correctness proofs. Structural testing techniques deal with the degree to which test cases exercise or cover the structure of the program. Functional testing techniques are concerned with finding the input values with which the program does not behave according to its specifications. Correctness proofs use formal languages to specify the requirements and mathematical logic to verify that the specifications are achievable by the program. None of these approaches can guarantee to isolate all sources of program errors.

For complete confidence, structural testing strategies require that all the paths in a program are tested, but testing all the paths is usually impossible because programs often contain an infinite number of paths. This has led to the development of a number of path selection criteria. A path coverage criteria is satisfied by certain sets of paths through a program, where a path is a sequence of statements. An effective criteria requires paths with high probability of revealing faults [3].

It has been hypothesized that software errors seem to come in clusters and some areas of the programs seem to be more error prone than others [4], thus one of the goals of software testing is to detect these areas. Some studies [5] indicate that there is some relation between the number of errors found in most computer programs and their logical complexity.

Software testers should select a sufficient number of paths to achieve coverage, starting by the shorter and simpler functionally sensible paths, trying to minimize the number of decision changes from path to path. The fundamental criteria is to assure that every instruction has been exercised at least once and every decision (branch or case statement) has been taken in each possible direction at least once. Associated with each path the test plan must contain a specification of the inputs that will force that path and a specification of all the outputs and database changes expected for that path. The derivation of the path-forcing input values is called path sensitizing.

The path sensitizing process is sometimes very difficult because the input values are not obvious. Some paths are confusing, counterintuitive and hard to understand. The presence of loops and the fact that the same decision may recur several times in a routine can reduce the number of paths through the routine to the point where seemingly sensible paths are not achievable.

It has been hypothesized that one reason why errors are not identified by programmers is that they are in parts of the code that are difficult to reach. Our assumption is that the source code in those areas should be complex in terms of number of nested control structures.

One of our purposes is to verify empirically if there exists some relationship between the software complexity that can be detected by static analysis at the source code level and the actual number of errors found in the modules that have more complexity. It would be useful to find some way to identify and differentiate the areas of the program that tend to be more difficult to test and debug without having to walk through all the source code.

Another purpose of this study is to verify if the number of nested control structures used in the programs has some relation to the types of errors detected in the areas that contain them.

One of the goals of software engineering is to reduce the costs of software development using a disciplined approach. A disciplined approach needs techniques to identify or define indices of merit that can support quantitative comparisons and evaluations [6]. The software complexity measures may be useful in preparing quality specifications and making design tradeoffs between maintenance and development costs [7].

The use of measurement technology has been identified as one of the functional tasks in the Department of Defense research program Software Technology for Adaptable, Reliable Systems (STARS) [8]. This technology concerns the development of evaluation criteria, their associated measures and metrics, and the experimental evaluation of techniques, methods and tools. The goal is to find ways of measure software attributes so that we can quantify software, and develop metrics that may be used to compare and predict software complexity. Some of the more important questions in the study of software metrics are how well the metrics really represent software complexity and development effort, and how well the metrics relate to software errors and reliability.

Software complexity can be defined as a measure of how difficult a program is to understand, modify and test. The importance of software complexity is represented in Figure 1. The goal of any software project is to stay within a reasonable budget and maximize the understandability, modifiability and testability of the code. The nature of the system under development will determine the proper weighting of the different quality factors to be achieved in the delivered software. Maintainability is typically of primary importance for business systems. Testability and reliability are critical concerns for life-support systems software. Efficiency takes precedence in many embedded real-time systems. Some quality factors, however, are contradictory and difficult to maximize. Optimizing code often lowers its understandability. Software complexity metrics can be used to monitor and modify the development effort according to their values: metrics can be used to predict the resources that will be required to implement and test the code, metrics can be used to predict the number of faults that may be found in subsequent testing or the difficulty involved in modifying a section of code.

The initial budget and time scheduled for a project influence the complexity of the software developed and consequently the quality of the product. The use of more resources when the final product does not achieve the quality initially required increases the development cost and time. Metrics are tools that can be used to control phases of the software cycle, providing feedback information to the project managers and programmers about the complexity of the product being developed.

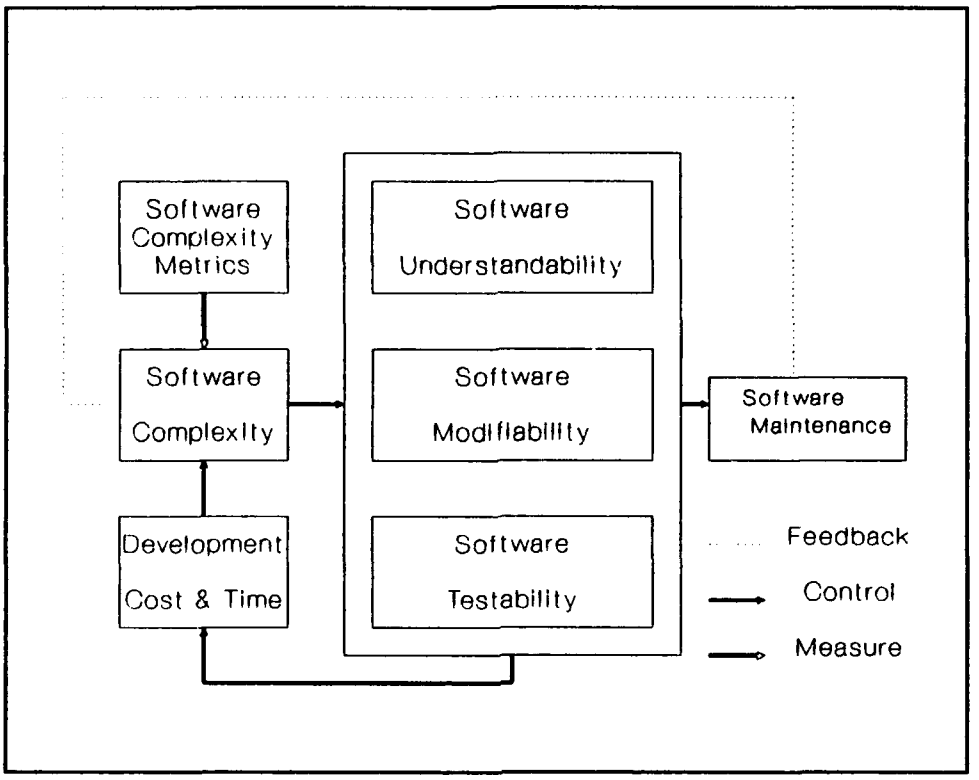

Figure 1. Importance of Software Complexity

There has been a great research effort to develop ways of measuring the complexity of programs. Using our intuitive notion of software complexity, we expect that complex programs will cost more to build and test, and will have more latent software errors.

Any useful measure of complexity must satisfy two basic requirements. First, it can be calculated for all programs to which developers apply it, and second, by adding something to a program (instructions, storage, processing time, etc.) the measured complexity can never decrease. Some complexity measures may serve as good predictors of particular properties of the programs.

Our hypothesis is that software complexity due to the number of control structures and the nesting level has some kind of relation with the degree of difficulty that programmers face when they try to test their programs, specifically during the path sensitizing phase. To test this hypothesis we analyzed the flow of control, types of control structures and levels of nesting in some faulty programs using different software complexity metrics, the average level of nesting bandwidth (BW), studied by Jensen and Vairavan [9], the cyclomatic number (v(G)), proposed by McCabe [10], the nested complexity (NC) and the number of statements (STM).

In Chapter II, we briefly describe some measures of software complexity that have been proposed, in order to provide a base of understanding for the following discussion. In Chapter III we present the description of the environment and metrics used to test our hypothesis, and the resulting data obtained from our analysis. Chapter IV details our interpretation of the results and what can be done to improve the quality of software during the development process using software metrics. Finally, in Chapter V, we provide our conclusions concerning the possible directions of future work in this area. The Appendix contains the tables with the metrics and faults, correlation coefficients and analysis of variance obtained for each version.

II. SURVEY OF SOFTWARE COMPLEXITY MEASURES

In this survey we are only concerned with complexity metrics that can be used for testing and maintenance purposes.

Many methods to measure software complexity have been proposed and explored. Software complexity metrics may be classified into two basic types, static and dynamic as shown in Figure 2.

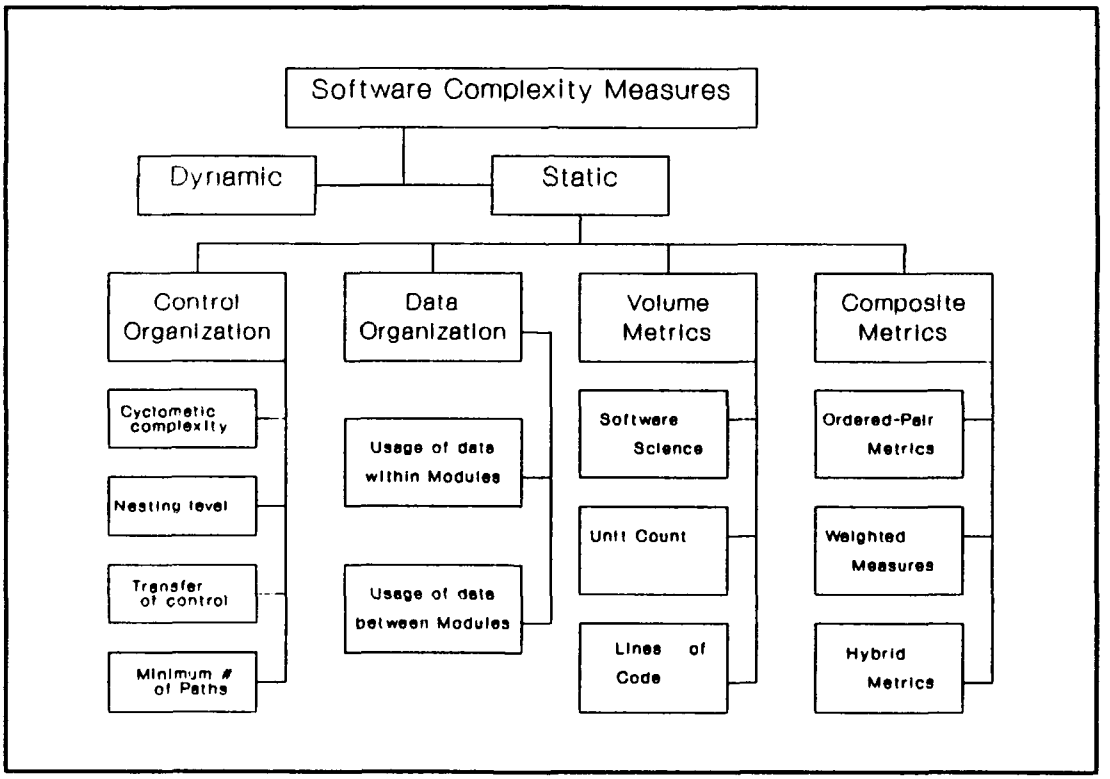

Figure 2. Classification of Software Complexity Measures

Static measures are obtained by static analysis of the source code or the high level description of the code. Dynamic measures consist of evaluation data collected at run time and may change from one execution to another. Dynamic measures may be CPU execution time, main storage used, data base size or computer turnaround time. Static measures may in turn be divided into four types, control organization metrics, data organization metrics, volume metrics and composite metrics.

The following sections overview some of the static measures that have been described in the software complexity research.

A. CONTROL ORGANIZATION METRICS

The control organization metrics are measures of the comprehensibility of control structures. These metrics use the structure of the source code to quantify the degree of complexity of the programs. Most of them use the structure of the algorithm represented by a directed graph called the control-flow graph. For each structured program module it is possible to get a directed graph with a unique entry node and a unique exit node. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program.

1. Cyclomatic complexity

The metric originally proposed by McCabe [10] uses mathematical concepts from graph theory applied to control-flow graphs. The cyclomatic complexity $v(G)$ of a control-flow graph G with n vertices, e edges and p connected components is:

$$v(G) = e - n + 2p$$

The cyclomatic number in a strongly connected graph, a graph where each node is reachable from every other node, is equal to the maximum number of linearly independent paths. In a module, this has been shown to be equal to the count of the number of decision statements in the module plus one. Thus, the cyclomatic complexity of a control-flow graph gives us the minimum number of paths that we should test to achieve independent path coverage. It has been proved that the cyclomatic complexity for a program with several modules is just the sum of the cyclomatic complexities of the individual modules.

The cyclomatic complexity metric seems to have some relation with the number of software errors and the debugging effort [2]. McCabe claimed that an upper bound for cyclomatic complexity equal to 10 seems to be a reasonable, but not magical upper limit for software modules. The intention is to keep the size of the modules manageable and allow for testing all the independent paths.

Another metric that uses the same concept of counting the number of changes in the flow of control is the count of decisions *DE*. Usually the sequential flow of control may be interrupted in three different ways: forward branches (IF-THEN-ELSE or CASE statements), backward branches (WHILE or REPEAT statements), and horizontal branches (procedure calls). An easy way of measuring the number of decisions is to count the predicates that affect the control flow. For instance an IF statement with two conditions is going to contribute two to the count of decisions. The same rule applies to the CASE statement that can be considered an IF statement with multiple predicates.

Gilb proposed two other metrics, C_L, the absolute logical complexity which is the number of binary decisions, and the relative logical complexity, c_L, which is the ratio of C_L to the number of executable statements [11]. These metrics have been supported by some empirical evidence and the latter may be considered as an improvement over pure control metrics as it also takes into account some size metrics.

Another control metric, *NPATH*, has been recently proposed by Nejmeh [12]. *NPATH* is a count of the number of acyclic execution paths through a function. It has been used with functions written in C at AT&T Bell Laboratories. The author claims that this metric overcomes the following shortcomings of $v(G)$. First, the number of acyclic paths in a flow graph varies from a linear to an exponential function of $v(G)$. Thus, the number of acyclic execution paths that may not be tested by a methodology based on that metric varies from 0 to 2^n, where n is the number of vertices in the flow graph. Second, the problem of treating different control structures (IF, WHILE, FOR) in the same way. Finally, the fact that $v(G)$ does not consider the level of nesting.

2. Nesting Level

Structure complexity can be determined by the depth of nesting [13], the average nesting level [14], and the bandwidth [9]. The basic assumption is that the higher the nesting level, the more difficult it is to determine the right data values to exercise those parts of the code. The nesting level of the first executable statement is assigned the value of one. If the following statement is sequential then it is assigned the same nesting level, otherwise it is assigned a nesting level of two. In general if the first statement is at level l and the following statement is in the range of a loop or a conditional transfer of control

then the nesting level of that statement is $l + 1$. The average nesting level *NL* is equal to the sum of all statement nesting levels divided by the number of statements. The bandwidth *BW* is equal to the sum of $(i * L(i))$ divided by the total number of nodes in the control graph, where $L(i)$ is the number of nodes at level i.

3. Transfer of Control

The idea of measuring the use of GOTO statements in FORTRAN programs was proposed by Woodward, Hennel and Hedley [15]. This measure is called *knots*. Imagine a forward arrow drawn on the left margin of a program listing indicating the flow of control from each GOTO statement to its respective LABEL and a backward arrow drawn from the end to the beginning of the respective loop. There is a *knot* every time we find an intersection of these arrows or control transfers. For equivalent programs the ones with the lower number of *knots* are believed to be better designed. Baker and Zweben [16] showed that in some cases this measure does not capture some control flow complexity differences. They present an example with two linearizations of a program that are equivalent and have different *knots* count measures. Another problem is that the addition of alternative constructs affects this measure in programs written in FORTRAN. For languages with an IF-THEN-ELSE operator the inclusion of an alternative construct does not affect the metric. Programs with arbitrary amounts of structured transfer of control may have the same complexity as any straight line code. This is an unappealing property of a general measure of control flow complexity.

Another pair of measures based on the flow graph of the program was proposed by Harrison and Magel in [17] called SCOPE and SCORT. A node is a sequential block

of code with a unique entrance and exit but no internal branch or loop. An edge is the flow of control between the various nodes. The outdegree of node u is the number of edges emanating from u, and the indegree of node u is the number of edges incident at u. Nodes with outdegree 0 or 1 are RECEIVING nodes and those with outdegree greater than 1 are SELECTION nodes. Given a selection node, we can find at least one lower bound node which succeeds every immediate successor of the selection node. The lower bound node that precedes every other lower bound is the greatest lower bound *GLB*. The number of nodes preceding *GLB* and succeeding the selection node, plus one, yields the adjusted complexity *AC* of that selection node. The SCOPE metric is calculated by summing up the adjusted complexity of each node. SCORT is the scope ratio metric and is defined as:

$$SCORT = (1.0 - N/SCOPE) * 100\%$$

where N is the number of nodes in the flow graph excluding the terminal node. SCORT increases towards 100% as complexity increases.

The SCOPE metric is dependent on the number of nodes in the flow graph. Therefore this measure cannot always be reliable, since some programs can be rearranged to give flow graphs with different scope measures [7].

4. Minimum Number of Paths

The minimum number of paths in a program, N_p, and the reachability of a node, R, were metrics proposed by Schneidewind and Hoffman [18]. The determination of N_p is done using path analysis to find a set of unique sequences of arcs from the start node to the terminal node excluding paths with backward loops traversed more than once. Since every decision leads to at least one extra path, it is always true that $N_p >= v(G)$. Reachability of

a node is defined as the number of unique ways of reaching that node. These metrics may be hard to determine on large programs because the number of paths can be very large or even infinite when loops exist.

5. Evaluation of Control Organization Metrics

Cyclomatic complexity and all the metrics that use the number of decisions are based on the assumption that software faults are proportional to control-flow complexity. This assumption seems to be well supported for v(G). There have been lots of empirical studies, since that metric was proposed in 1976, that show some relation between the higher values of this metric and the modules that are more error-prone [36], [37].

The value of v(G), however, may lead us to incorrect conclusions about the characteristics of the software product. A program may use several data structures very hard to implement and manipulate, and lots of modules calling other modules recursively and have a low value of v(G). Intuitively this program should be complex and hard to test, and consequently more error-prone. Thus, this metric is not a good predictor of error-proneness of the modules in every case. The cyclomatic complexity is an easy to use and useful rule of thumb for comparing alternate approaches and for estimating the amount of debug labor between similar programs developed in the same environment.

Jensen and Vairavan [9] have indicated that cyclomatic complexity correlates better then some of the measures based on the nesting level (e.g., bandwidth) to the number of program changes and problem reports.

The control organization metrics do not consider the contribution of any other factor except control flow complexity. These metrics, however, can differentiate between two programs of similar volume metrics and certainly are related to the software quality.

B. DATA ORGANIZATION METRICS

The data organization metrics are measures of data use and visibility, as well as the interactions between data within a program. These metrics are concerned with the amount of input data, output data and processed data internally used by software. The simplest data structure metric is the count of variables that are defined and used in a program. Another simple data structure metric is the count of the number of I/O formats in FORTRAN or COBOL code.

1. The Usage of Data Within Each Module

The usage of data within a module may be measured using the concepts of *live variables* and *variable spans*. The hypothesis is that the more data items a programmer must keep track of when constructing a statement, the more difficult it is to construct. A variable may be considered live from its first to its last references within a procedure. The average number of live variables is the sum of the count of live variables divided by the number of executable statements. The span is the number of statements between two successive references to the same variable. Thus, a large span indicates that the programmer during the construction process had to remember a variable that was last used far back in the program. These metrics have been used in a study by Elshoff reported in [19], using programs written in PL/1, to identify areas of greater complexity. Programmers

have indicated that this measure can also be applied to other languages, particularly COBOL, because the information presented is similar.

2. The Usage of Data Between Modules

Henry and Kafura [20] proposed a method to measure the complexity of code due to the flow of information from one module to another using an information flow metric. The flows of information into and out of a procedure are called *fan-ins* and *fan-outs*. Local flows represent the flow of information to or from a routine through the use of parameters and return values from function calls. *Fan-in* is the number of local flows into a procedure plus the number of global data structures from which a procedure retrieves information. *Fan-out* is the number of local flows from a procedure plus the number of global data structures which the procedure updates. The complexity of a procedure p is defined as:

$$C_p = (\textit{fan-in} * \textit{fan-out})^2$$

Another approach to the evaluation of complexity between modules is to measure the sharing of data as global variables among modules as suggested by Basily and Turner [21]. This may be done by counting the number of pairs (M, R) where M is a segment or module and R is a global variable that is read or changed by M. These pairs are called the *segment-global usage pairs*.

McClure proposed another metric focused on the complexity associated with the control structures and control variables used to direct procedure invocation in a program [22]. She claims that all predicates do not contribute the same complexity. The control variables appearing in conditional statements that determine the invocation of other

procedures contribute with a higher complexity. The complexity of a program module *P* consists of two factors: the complexity associated with the control variables invoking module *P* and the complexity associated with the control variables by which module *P* invokes other modules. The overall complexity is determined by the sum of the complexities of the modules.

3. Evaluation of Data Organization Metrics

These metrics are based on the assumption that software complexity is related with the amount of data processed and the flow of data through the program. This assumption may not be valid in some cases because there are other factors that contribute to increase the complexity of software. The structural complexity and the size are examples of those factors. There are some studies, however, that found some relation between these metrics and the number of faults.

The information flow metric was used in an objective study of the UNIX™ operating system. This study found a statistical correlation of 0.95 between faults and procedure complexity as measured by the information flow metric [20].

All these data organization measures attempt to capture a different kind of complexity of the control organization metrics. The simplest is a count of the number of entries in the cross-reference list of the program (VARS). The metric VARS seems to be robust and even slight variations in algorithm computation schemes do not seem to affect other measures based upon it.

C. VOLUME METRICS

The volume metrics are measures of the size of the product. There are many methods to measure software size. The easiest one is the count of the lines of code. This metric may include all the source lines or only the executable statements. Usually it includes the lines containing program headers, declarations, executable and non-executable statements, and excludes the lines containing comments. Other simple volume metrics are the number of statements or the number of operators and operands.

1. Software Science Measures

These measures were created by Halstead and they are part of a more complex family of metrics called Software Science [23]. We include these measures in the software size category although in his work there are several proposals of metrics to quantify other aspects of software. All the measures are functions of the count of the tokens that form the program.

The basic measures are:

n_1 = number of unique operators

n_2 = number of unique operands

N_1 = total occurrences of operators

N_2 = total occurrences of operands

Operators are symbols and keywords, and operands are variables, constants and labels. The *length* of a program, N, is expressed in tokens as:

$$N = N_1 + N_2$$

The Software Science measures defined other metrics related to size. The *vocabulary*, n, is:

$$n = n_1 + n_2$$

The *volume*, V, is:

$$V = N * log_2 n$$

There have been several studies that seem to show that these basic metrics are strongly correlated to program size and number of errors [9], [24].

Some other measures were proposed, the bug prediction formula B, and the programming effort E:

$$B = (N_1 + N_2) * log_2 (n_1 + n_2) / 3000$$

$$E = (n_1 N_2 N log_2 n) / 2n_2$$

These measures are more controversial. There are some studies that seem to confirm the bug prediction formula. They are reported by Lipow in [25] with a comparison of actual and predicted bug counts over a range of program sizes from 300 to 12,000 executable statements. These results, however, are not conclusive. Conte, Dunsmore and Shen in [26] conclude that these measures, B and E, have not been shown to have good construct validity and they probably do not measure exactly what Halstead hoped they would.

2. Unit Count

The idea behind this approach is to divide the source code in programming modules or units. These modules may be defined in many different ways. A module may be a segment of code that can be compilated separately or a procedure that executes a

particular algorithm. Each module may be divided in one or more functions. A function is defined as a collection of declarations and executable statements that performs a certain task. It is not easy to count the number of functions unless the programs are created with each module as a separate function. Studies have shown that different programmers tend to use a similar number of functions to solve the same problem using a different number of modules [27]. Another related measure is the count of the number of statements per unit, the average length of a programming module.

3. Length Estimators

There have been several proposed metrics to estimate the length of the programs. Halstead in [23] defined a program length estimator N_h as:

$$N_h = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

Jensen and Vairavan [9] proposed an empirical expression to compute the length N_j of a program, claiming that it was a more accurate estimate than Halstead's N_h:

$$N_j = \log_2 (n_1!) + \log_2 (n_2!)$$

4. Evaluation of Volume Metrics

The volume metrics were the first measures of software complexity to be used. They have the same limitations of the control organization metrics and the data organization metrics because they are based on the assumption that complexity is only related to size.

The software reliability study by Thayer, Lipow and Nelson [5] showed error rates ranging from 0.04% to 7% when measured against lines of code, with the most reliable routine being one of the largest. This seems to confirm that there is no direct correlation between faults and lines of code. Flaherty showed in [28] that there is some statistical

correlation between lines of code and maintenance costs. Another study by Li and Cheung [7] showed that the Software Science length estimator N_h overestimates the actual length for small programs and underestimates N for large programs. Thus, it cannot be a reliable measure of complexity.

The major limitation of volume metrics is that they can only be measured after the design has been carried out fully to the debugged code. By then it is usually too late and too expensive to take the necessary corrective action.

D. COMPOSITE METRICS

Since each metric is designed to capture a particular feature of a program it is impossible to determine the overall complexity of a system based exclusively on some features. This conclusion led to several attempts to incorporate different metrics. The composite metrics are an attempt to combine some aspects of the previous types.

1. Ordered-Pair Metrics

There have been several attempts to combine different metrics in ordered pairs.

Myers [29] proposed the pair (CYC-mid, CYC-max) where CYC-max is equivalent to the cyclomatic number, and CYC-mid is equal to CYC-min plus the number equal to two less than the number of selections in a CASE statement (CYC-min is the count of all conditionals and loops including CASE statements).

There are other measures proposed by Hansen [30] that consist on using an ordered pair where one coordinate is a variation of the cyclomatic number and the other coordinate is a software science measure.

Oviedo [31] proposed a complexity metric based on control flow complexity *CF* (the cardinality of a set based on the control flow graph) and data flow complexity *DF* (based on the count of variable definitions and references in each block).

This measure was defined as:

$$C = aCF + bDF$$

where a and b are some predefined weights.

2. Weighted Measures

Baker and Zweben [16] pointed out the need of a measure which should combine some of the measuring capabilities of the software sciences and the complexity number. A synthesis of software science measures and the cyclomatic number was proposed by Ramamurthy and Melton in [32]. A weighted measure is built from a software science measure by allowing certain operators and operands to contribute extra values. The purpose is to assign weights so that the length and volume measures can detect complexity produced by nonsequential control structures. If an operand or operator is part of a control structure the idea is to add a value equal to the nesting level (weight) of that control structure to the count of occurrences of that operand or operator. The authors claim that these measures can detect the different types of complexity detected by the cyclomatic number and the software science.

3. Hybrid Metrics

These metrics combine some aspects of data structure metrics and logic structure metrics.

The information flow metric of Henri and Kafura [20] may also be used as an hybrid metric. The complexity of procedure p, C_p, is defined as:

$$C_p = C_{ip} * (fan\text{-}in * fan\text{-}out)^2$$

C_{ip} is the internal complexity of procedure p determined by any code metric.

Another hybrid metric was proposed by Basily and Hutchens [33]. The definition of a syntactic complexity family that could include volume and control metrics led to a new metric called *syntactic complexity (SynC)*. A program is called a *proper* program if it has a single entry and a single exit, and every node of the program lies on some path from the entry to the exit.

The measure *SynC* of a program p is defined as:

$$SynC(p) = 1.1 * C(p_i) + 1 + \log_2 (n+1) \quad \text{if p proper statement}$$

or $$SynC(p) = 1.1 * C(p_i) + 2 * (1 + \log_2 (n+1)) \quad \text{if p not proper statement}$$

where $C(p_i)$ is the sum of all the syntactic complexities of each subcomponent p_i of the program, n is the number of decisions in program p that are not part of any subcomponent p_i. Nesting is penalized by multiplying $C(p_i)$ by 1.1, counting each statement 10% more than it would have been counted for at the next outer level. Poorly structured code is penalized twice as much as well as well structured code. Thus, this metric includes consideration of nesting level, length (statement count) and structured programming practices.

Li and Cheung propose another hybrid metric in [7]. This hybrid metric is called NEW_1 and integrates software science with the SCOPE measure. They define the raw

complexity of a node V_j as E_j:

$$E_j = N_j \log_n j \; / \; L\hat{}$$

where N_j, n_j, and $L\hat{}$ are the software science measures length, vocabulary and the estimate of the program level of the node V_j. This last one can be calculated using the following expression:

$$L\hat{} = 2/n_1 * n_2/N_2$$

The adjusted complexity for a selection node is the sum of the values of the raw complexity of every node within the SCOPE of that selection node, plus the value of the selection node itself. A receiving node has an adjusted complexity equal to its raw complexity. The complexity of the program is the sum of the adjusted complexities of every node. They define *NEW_1* as:

*NEW_1 = (1.0 - Total Raw Complexities/Total Adjusted Complexities) * 100%*

4. Evaluation of Composite Metrics

Although composite metrics have the advantage of incorporating the strengths of the primitive types of metrics and provide a more accurate measure of software complexity, they tend to be harder to calculate. The interest and quality of the information supplied may not be sufficient to justify the cost and effort of using them.

Most of the composite metrics have not been extensively tested as some of the other types because composite metrics are relatively new. The validation process must continue before these metrics can be effectively adopted in the characterization and evaluation of software.

E. THE ROLE OF SOFTWARE METRICS

Software metrics are standard ways of measuring some attribute of the software development process. Some metrics have been used in industry while others have been confined to academic environments. Those more commonly used in industry are: lines of code (the simplest metric), cyclomatic complexity metric (proposed by McCabe in [7]) and their variations, and Software Science measures (proposed by Halstead in [23]). The use of these and other software complexity metrics in the industry and the armed forces is reported in several recent studies [38], [39], [40].

The great number of software measures that have been and continue to be proposed is a good indication that the controversy that has surrounded them since their first appearance is far from ended. Some claim that metrics are useless and expensive exercises in pointless data collection, while others argue that they are valuable management and engineering tools.

The value of software measures, their limitations, their strengths, and the benefits they can provide, has to be verified through empirical studies in different kinds of environments. We cannot apply metrics without first understanding what we want to measure and how we will measure what we want to know about. Another issue when applying metrics is how to get the metrics results in a way that they can be used and understanded by the people in charge of the process.

This study analyzes the use of some control organization metrics during the software development process, specially the testing phase. Some of the specific questions that this thesis addresses are: How do these metrics relate to the number of faults detected? How do

these metrics relate to the types of faults detected? What is the relation between different control organization metrics? How can these metrics be used to predict whether a given module is error-prone?

As we collect more data about the relation between complexity metrics and potential software problems, we may be able to understand better the real importance and usefulness of metrics in issues of software reliability and cost.

III. DATA ANALYSIS

A. METRICS INVESTIGATED

Our hypothesis is that one of the major reasons why errors are not identified by programmers is that they are in parts of the code that are logically difficult to reach. In this study we decided to use three different measures of complexity to verify our hypothesis: the cyclomatic complexity v(G), the average nesting level BW and the number of statements STM. We also use another measure obtained from the product of the cyclomatic complexity and the bandwidth that we call nested complexity (NC). This is an attempt to find a metric sensitive to the level of nesting within the various control structures.

BW and v(G) were chosen because intuitively they seem to capture the structural complexity fairly well. The cyclomatic complexity metric, as a count of the number of decisions in each module plus one, is related to the number of changes of control-flow. However, it cannot detect any complexity due to nested structures. The bandwidth is a measure of the average nesting level, therefore seems logical to try a combination with v(G) to get a more accurate measure of total software complexity. That combination is the measure NC. The metric number of statements STM was used because it is a volume metric similar to lines of code, the most used measure of software complexity. In this study, using Pascal programs, STM is the count of the tokens ";" and "BEGIN".

The cyclomatic complexity and the number of statements were calculated for each module using a lexical scanner adapted to count the tokens according to the set of counting

rules for the Pascal language used by the Purdue University Software Metrics Research Group [27]. The nesting level of each module was analyzed by inspection.

B. DESCRIPTION OF THE ENVIRONMENT

A set of eight programs written from a single specification for a combat simulation problem was used in this study. The programs were designed and written in Pascal by two-person teams and the teams were assigned randomly from students in an upper division computer science course. The length of the programs varies from 1186 to 2489 lines of code and the number of modules of each program varies from 28 to 76 modules.

A previous study [34] extensively tested these programs. The number of faults detected and a brief description of their types has been previously recorded. Five different fault detection techniques have been used to detect these faults: code reading by stepwise abstraction, multi-version voting, run-time assertions inserted by the programmers, functional testing with follow-on structural testing, and static data-reference analysis. A total of 209 faults were detected in that experiment, with 157 faults identified with specific pieces of code. The remainder mainly dealt with missing code and faults with distributed causes.

The fault classification scheme used in that study was a fault taxonomy with 13 classes designed specifically to reflect the variations in faults between the techniques. The fault classification scheme is described in Table 1, drawn from [35].

C. RELATION OF METRICS WITH NUMBER OF FAULTS

The average values of the metrics and the number of faults found for each program are shown in Table 1 of the Appendix.

Our results seem to confirm that there is some relation between software complexity due to the structure of the program and the number of errors. The modules with greater

TABLE 1 - FAULT TAXONOMY

CLASSES OF FAULTS	EXAMPLES OF FAULTS	TECHNIQUE USED
1 - Overrestriction	Rejecting legal inputs	Assert, Read, Test, Vote
2 - Loop Condition	Infinite loops	Vote, Assert, Test
3 - Calculation	Incorrect formulas	Read
4 - Initialization	Variables not initialized	Statical Analysis, Test
5 - Substitution	Wrong variables used	Vote, Assert
6 - Missing Check	Divide by zero faults	Read
7 - Branch Condition	Bad condition on a branch	Vote, Read, Test
8 - Missing Branch	Localized missing code	Read, Test
9 - Missing Thread	Missing path throughout program	Vote, Test
10 - Unimplemented Requirement	Missing functionality on all paths	Test
11 - Ordering	Operations in wrong order	Vote, Test
12 - Parameter Reversal	Actual parameters permuted with formal parameters	Vote, Assert
13 - Data Structure	Linked list becomes circular	Vote, Test, Read, Assert

complexity using any of the three control structure metrics have more detected faults. These results seem to confirm other studies by Walsh [36]. The bandwidth and the nested complexity seem to have also some relation with the number of faults. However, the percentage of faults detected with these methods is not greater than the percentage obtained

using v(G). This is a useful observation because the computation of v(G) is easier than the computation of BW and NC.

We observed the following averages using the set of eight versions: 18% of the total number of modules had v(G) greater than 10, and these modules contained 51% of the total number of faults; the modules with BW greater than 2.5 were 24% of the total number of modules, and these had 47% of the faults; the modules with NC greater than 29 were 17% of the total, and these contained 47% of the faults. These values are a good indication that we may be able to detect the modules with more tendency to have errors using complexity metrics, specially these particular control structure metrics.

The modules having STM greater that 24 comprised 28% of the total number of modules and these contained 52% of the faults detected. The small percentage of modules is misleading because these modules have in average 65% of the total number of statements in each version. The metric STM do not seem to have any relation with the number of faults. This result is a confirmation of other studies [5] that did not find any relation between lines of code and software faults.

Our preliminary results seemed to indicate that the modules where the metric NC was less than 4, contained also a greater number of faults. Our first hypothesis was that this could be a consequence of the carelessness of programmers only because the modules seemed obvious and easy to implement. This assumption, however, was not validated by our data because the large number of faults actually found in those modules was a direct consequence of having many modules with small values of NC in this set of programs.

The faults in the modules with less complexity seem to have a regular distribution: 16% of the modules have NC equal to 1 and 10% of the total number of faults; 43% of the modules have NC less or equal to 4 and 23% of the faults; 56% of the modules have NC less or equal to 8 and 35% of the total number of faults. These results seem to show that the number of errors in the modules with less complexity increases proportionally at the same rate that complexity when the value of NC is less than 30. In the modules where this metric is greater than 30 the number of errors increases at an higher rate.

The complete analysis of variance of faults using the four different metrics with each version are shown in Tables 19-22 of the Appendix. The between groups variance is the estimate of variance based on the differences between the means of sets of modules with the same value of the metric. This estimate reflects the internal differences in the number of faults detected between sets of modules separated according to the values of the metrics. The estimate of variance based only on the differences between individual modules is called the within groups variance. This estimate reflects only the chance variations involved in drawing a sample. The degree of freedom of the variation between groups is the number of groups or sets of modules with the same value of the metric minus one. The degree of freedom of the variation within groups is equal to the total number of modules minus the total number of groups of modules. The F-ratio is the quotient of the two variances. The F-ratio is used to determine if the difference between groups in a sample is significant or not. This can be done using tables of the F-distribution and the values of the two degrees of freedom. The mean square is the ratio between the sum of squares and the respective degree of freedom.

The analysis of variance of the number of faults using all the metrics presented very low significance levels, which is an indication that the probability that our results were obtained by chance is very low. The only exceptions were found using NC or v(G) in version 8, and this may be a consequence of having only two modules with high complexity containing only one fault in this program. Our results indicate also that the variations in the number of faults between the different sets of modules according to the values of the metrics are significant, because in general all the versions have the variation between groups greater than the variation within groups. This is another indication of the good fault-predictive ability of software control-structure metrics.

D. RELATION OF METRICS WITH TYPES OF FAULTS

We used the values of NC to divide the modules in two sets: those with NC less or equal to 4 and those with NC greater or equal to 30. The modules with NC between 5 and 29 were not considered. Then, we identified the faults found in the two sets of modules and their respective types according to the fault classification previously described.

We found some similarities and some differences between the types of faults detected in the two sets of modules. About 43% of the total number of faults in both sets belonged to classes 3 and 6. Class 3 faults are calculation faults, for instance the use of the wrong expression in the calculation, and class 6 faults are due to missing code to deal with illegal behavior, for example divide by zero faults. The first type of faults may occur because of misunderstanding of the specifications during the translation to code, and obviously does not depend on the complexity of the structure. The second type may be the result of the

carelessness of programmers because of time constraints, so frequent during the development of any software system.

The significant differences between the two sets of programs were found in the classes 1 and 5, faults due to overrestrictive input checks and wrong variable uses, respectively. The modules with less complexity had a low incidence in these type of faults (6%) while the modules with more complexity had a high incidence (19%). These faults may be caused by different reasons. We have two hypothesis to explain the observation. Programmers tend to clutter the source code with unnecessary conditions when it is already complex from the beginning. This may happen because they do not understand exactly what the program should do in those areas, leading to class 1 faults. The reason for class 2 errors may be related to the difficulty of keeping track of the variables and their use in the modules with a large number of nested control structures.

The remaining classes of faults had no significant clusters to allow some conclusions about their relation to structural complexity. Their distribution was quite similar in both sets.

E. RELATION BETWEEN METRICS

In order to understand the relationship between the various software metrics used, Pearson correlation coefficients were computed for every pair of metrics, indicating the degree of linear relationship between them. Pearson values lie in the interval [0,1]. The correlation coefficients for each program are shown in Tables 2-9 in the Appendix.

We observed that the correlation between v(G) and BW is not very high and its value depends on the program. This observation does not confirm the earlier results of [6]. This

seems to be intuitively correct because the two metrics are measuring different aspects of software complexity.

The complexity measure NC seems to correlate well with v(G) and BW. This measure seems to bridge the gap between the two previous metrics and conceptually is a more refined measure of the complexity of the control structure.

Another observation is that the STM metric does not correlate well with any of the other metrics. This result is different from other studies [9] that presented the cyclomatic complexity correlating well with lines of code, another volume metric. Our values for the correlation between v(G) and STM are similar to the results reported in a more recent study by Henry and Selig [36] using Pascal source code (0.65 against 0.63).

The Tables 10-17 in the Appendix show the values of the metrics and the number of faults found for all modules in each version.

IV. DATA INTERPRETATION

A. DATA LIMITATIONS

It is important to consider several limitations when drawing conclusions from the data presented in this study. First, this study used several versions of only one application written in one language, Pascal, and this may not be representative of a large number of applications. Second, data gathered from programs designed and written by students should be used with caution. Lastly, the number of faults in each module may be misleading because the versions may have more faults than those that were detected. However, the final versions are relatively large and have been produced from a specification derived from an industrial specification. They have been extensively tested and the testing methods used provide a relatively good coverage.

B. USING METRICS IN SOFTWARE DEVELOPMENT

In spite of the limitations that unfortunately are common in this type of experimentation, information can be derived from this study about the software development process.

Our data indicates that the modules with higher values of software complexity using the three different control organization metrics v(G), BW or NC, have more detected faults. This is a good indication that these metrics can be used to predict the modules with more tendency to have faults. This may be useful particularly if they are used at the design stage providing feedback to the software developers, allowing the redesign of those modules. The

fact that these metrics can be computed from the high level description of the algorithms in the form of control-flow graphs or even pseudo-code is another reason why they must be used in the earlier stages of the development process to reduce the impact of the changes and consequently their cost.

The coding phase should start only after the detailed design phase has pruned out the most troublesome areas according to the value of the software complexity metrics. If it is not possible to eliminate those areas at the design stage, the project managers must be alerted to inherent levels of complexity in the source code and take appropriate actions during the reviewing and testing phases. Given a limited budget, a large project cannot afford complete branch coverage or inspection coverage. It is most effective to simplify unnecessarily complex modules and spend more time inspecting, reviewing and testing those modules that are inherently more complex.

Another observation concerns the types of faults detected in the modules according to their measured complexity. The most common types of faults detected in all the modules independently of the value of the complexity metrics were calculation faults and faults due to missing checks for obvious illegal behavior. This result seems to be an indication that these types of faults are not related to structural complexity and have to be handled in a previous stage of the development process, the requirements specification phase.

We verified also that the modules with more complexity had a relatively high incidence of faults due to overrestrictive input checks and wrong uses of variables, when compared to the modules with less complexity. This result seems to show that structural complexity at the source code level is related to these particular types of faults.

Our data suggest that software metrics can be used to divide the modules according to their structural complexity before the testing phase starts. The testing techniques used for each set of modules may be chosen according to the types of faults occurring more frequently in each set. This would allow a more efficient use of the several testing techniques because some of them are more suited to find particular types of faults.

There is no widely accepted detailed taxonomy for fault classification. Other classification schemes that may be used in similar studies were proposed by Beizer [2], Rubey [42] and Endres [43]. This raises the issue of having a standard to define the different types of software faults, even if it is evident that there is no universally correct way to categorize faults. That standard taxonomy could be only a starting point. This would allow a unified framework to all the research dealing with software faults and software reliability.

The study of the relation between software metrics, fault detection techniques and the different types of faults has to continue. The testing tools available now have to be used in the most effective way because we cannot afford the cost of testing very large and complex programs using brute-force approaches.

The maintenance phase may also get some benefits from the use of metrics. Most of the software being developed now results from changes in existing products instead of new products started from scratch. The modifications done to the programs usually consist of adding new functionalities, resulting in higher complexity of the modules at the source code level. Complexity metrics may be used to monitor changes to existing software to keep the modules in a manageable and testable form.

The creation of automated tools measuring software complexity at each stage of development to flag potential problems to the project managers may reduce costs and increase the reliability of software. These automated tools must present the metrics results in a way that all the personnel involved in the process can understand them without difficulty instead of providing just pages and pages of numbers, formulas and tables that nobody wants to look at. There are already tools that present a graphical representation of the cyclomatic complexity of the programs. This approach is giving better results than before using only the numerical values because programmers and managers respond more readily to the visual image [44].

Another useful observation of this study is that a great number of faults detected, classes 3 and 6, were found just by reading the code. These results confirm the observations of Beizer in [2] that desk checking and particularly code reading are the best catchers of private bugs and cannot be completely replaced by any other technique.

C. TESTING ANOTHER VERSION

The existence of another version of the same program where the faults had not been identified during the experiment reported in [34] gave us an opportunity to test some of the results obtained with the other versions.

Our initial approach was the determination of the values of the metrics for each module to establish different sets of modules according to their structural complexity. Then, we tried to detect the maximum number of faults just by reading the code. Using this technique we found 16 faults caused by missing code to deal with divide by zero situations. The faults were scattered throughout the program without any special incidence in the

modules with greater complexity. The modules with greater complexity had only 3 faults. These observations seem to confirm the results obtained with the other versions that indicated that this type of faults does not have any relation to structural complexity.

In our effort to detect more faults we ran the program with 100 randomly-generated test cases. Using this technique we verified that 15 cases gave us results that indicated the existence of faults in some modules. Analyzing for the detected faults, we found a missing branch in one routine, two faults involving variable initialization and use in another routine, an unused function, a loop scoping fault and two calculation faults.

The use of random tests to detect the existence of faults and the nested complexity metric to detect the modules that may contain those faults was extremely useful during this testing phase. We found more incidence in faults caused by overrestrictive checks and missing branches in the modules with greater complexity as Observation, Restoration and OutputReport. This seems to confirm our previous observation that at least the first type of faults is more frequent in the modules with higher values of nested complexity.

The values of the metrics and the faults detected for each module in this version are presented in Table 18 of the Appendix. Due to the incompleteness of the data on this version, statistical analysis was not performed on the relationship between detected faults and the metrics. Nevertheless the results support the use of this metric in realistic testing.

V. CONCLUSIONS

A. FUTURE RESEARCH

The use of metrics in software development is gaining an increasing interest in recent years as research shows their usefulness. However, there have been many proposals of new metrics, some of them complex and difficult to use and others trying to measure subjective aspects of software that cannot be measured at all. We are running the risk of spending more money implementing the metrics program to control the development process than building the software systems themselves. This may be one of the reasons why software metrics have raised so much controversy and skepticism among the software developers and researchers. As we stated at the beginning of this work, a good metric must be simple to calculate and understand by the software developers, otherwise its usefulness is completely overwhelmed by the overhead of using it. Researchers should continue to test the existing metrics with real data, with different kinds of programs and systems to verify their applicability. Project managers and programmers in industry and in the armed forces should start controlling their software development processes using different types of metrics instead of only the traditional Software Science measures and the cyclomatic complexity. More data must be collected incorporating programs of different types like operating systems, compilers and embedded real-time systems to verify the usefulness of metrics.

The application of other fault detection techniques to version 9 trying to test the results and hypothesis generated from the other versions may provide some answers to the

following questions: Are the faults due to overrestrictive input checks and wrong variable uses more frequent in the modules with greater complexity? Is the majority of faults in programs caused by wrong expressions and missing checks? We need to know if our observations are a good indication of some pattern or they are only a consequence of this particular environment.

We can never be sure that a verification is correct, thus, we need to apply the testing techniques in the most effective way to gain a reasoned and cautious assurance that the programs will run satisfactorily. To achieve this goal we need to have a better knowledge of the strengths and limitations of each testing technique. Are they particularly suited to find some types of bugs? What are those types? We need more empirical studies to test and compare the testing techniques in different environments to provide some answers to these questions. Can we develop new testing tools to help us to find obvious illegal situations? Can we build more powerful data flow analyzers to follow the use of the variables through all the program? The automation of the testing process is another area of research that needs to be addressed by the computer-science community.

The impact of using formal methods during the requirements specification phase is another interesting area of research that can find answers to some of the questions raised by our work. Can we reduce only some types of faults using that approach or can we reduce all types of faults? Is the structural complexity of the programs reduced if we use those methods or is it increased? Is it possible that when we are reducing the number of faults caused by incorrect specifications we are also increasing the number of faults caused by structural complexity? If we add more checks and more conditions to the code based on

more correct specifications to assure that nothing in the requirements is left out, our assumption is that the program is going to be more complex. Therefore, it will be more difficult to test and debug.

B. FINAL COMMENTS

This study raised new questions but also provided some answers to the questions and hypothesis presented in the introduction.

This empirical study shows that control organization metrics can be used to predict the more error-prone modules. Our data seem to indicate that the number of nested control structures used in the programs has some relation with some types of faults. Namely the faults caused by overrestrictive input checks and wrong use of variables. This observation seems to confirm our hypothesis that some faults are not detected because they are in parts of the code that are difficult to reach during the path sensitizing process. This information may be useful during the testing phase because software developers know in advance that the data flow in the modules with higher levels of nesting needs to be checked. The modules with less nested complexity show a regular distribution of faults, most of them caused by wrong expressions and missing checks that cannot be related with structural complexity. These bugs seem to be caused by faulty specifications and have to be eliminated in the requirements specification phase through the use of formal specification techniques.

The software developers should use not only formal methods to specify the requirements but also software metrics to control how those requirements are implemented. Even if we use automated tools to build the systems based on formal specification

languages, the human intervention caused by the interaction between the developers and the customers during the definition of the requirements is going to cause faults. Another problem that may arise is that usually code created by automated tools is highly optimized and consequently very complex. Our data indicate that an increase in structural complexity may create other types of faults and the detection of this increase can be done using control structure metrics. Software complexity metrics can be used to identify the improper integration of functional enhancements made to the systems. The analysis of the redesigned versions of the systems using metrics can reveal poorly structured components. This can be particularly useful to monitor maintenance activities, one of the most critical phases of the software development cycle in terms of costs.

APPENDIX - TABLES OF METRICS, CORRELATION COEFFICIENTS AND ANALYSIS OF VARIANCE OF FAULTS WITH METRICS

TABLE 1 - AVERAGES OF METRICS

Version Number	No. of Modules	No. of Faults	Average v(G)	Average BW	Average NC	Average STM
1	72	23	7.11	2.11	20.17	21.06
2	55	11	5.58	2.06	18.51	18.80
3	43	27	6.21	1.88	17.91	22.09
4	57	22	7.42	2.10	20.12	24.35
5	28	22	12.54	2.48	43.04	36.04
6	76	17	5.41	1.74	11.74	17.13
7	68	22	5.82	1.58	12.34	17.10
8	57	13	4.70	1.59	9.14	19.04

TABLE 2 - CORRELATION COEFFICIENTS FOR VERSION 1

	v(G)	BW	NC	STM
v(G)	1	0.55	0.86	0.69
BW	-	1	0.83	0.46
NC	-	-	1	0.63
STM	-	-	-	1

TABLE 3 - CORRELATION COEFFICIENTS FOR VERSION 2

	v(G)	BW	NC	STM
v(G)	1	0.95	0.96	0.67
BW	-	1	0.97	0.64
NC	-	-	1	0.65
STM	-	-	-	1

TABLE 4 - CORRELATION COEFFICIENTS FOR VERSION 3

	v(G)	BW	NC	STM
v(G)	1	0.82	0.94	0.67
BW	-	1	0.93	0.60
NC	-	-	1	0.78
STM	-	-	-	1

TABLE 5 - CORRELATION COEFFICIENTS FOR VERSION 4

	v(G)	BW	NC	STM
v(G)	1	0.61	0.84	0.69
BW	-	1	0.89	0.34
NC	-	-	1	0.52
STM	-	-	-	1

TABLE 6 - CORRELATION COEFFICIENTS FOR VERSION 5

	v(G)	BW	NC	STM
v(G)	1	0.34	0.88	0.70
BW	-	1	0.57	0.45
NC	-	-	1	0.80
STM	-	-	-	1

TABLE 7 - CORRELATION COEFFICIENTS FOR VERSION 6

	v(G)	BW	NC	STM
v(G)	1	0.60	0.97	0.58
BW	-	1	0.94	0.57
NC	-	-	1	0.64
STM	-	-	-	1

TABLE 8 - CORRELATION COEFFICIENTS FOR VERSION 7

	v(G)	BW	NC	STM
v(G)	1	0.59	0.78	0.67
BW	-	1	0.90	0.60
NC	-	-	1	0.68
STM	-	-	-	1

TABLE 9 - CORRELATION COEFFICIENTS FOR PROGRAM 8

	v(G)	BW	NC	STM
v(G)	1	0.68	0.91	0.56
BW	-	1	0.88	0.65
NC	-	-	1	0.65
STM	-	-	-	1

TABLE 10 - METRICS FOR VERSION 1

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Ceiling	4	1	4	8	-
MinI	2	1	2	3	-
MinR	2	1	2	3	-
MaxIR	2	1	2	5	-
SizeListLoc	2	1	2	13	-
OutsideRange	3	1	3	4	-
Scream	3	1	3	14	-
SquadAlive	1	1	1	4	1
BatAlive	5	1.40	7	15	-
VerifyIr ut	6	1.33	8	11	-
CheckParams	14	1	14	18	1
CheckArmyValues	50	2.78	139	58	2
CheckComMsg	6	2.17	13	10	-
CheckWeather	11	1.91	21	14	2
BatVelocV	5	2.20	11	20	-
AltitudeZ	5	1	5	13	-
DistD	1	1	1	4	-
TerrMoveTM	7	1.14	8	14	1
WeatherSevFactWF	5	1.80	9	16	-
WeatherObservWC	2	1	2	8	-
WeatherMoveWM	1	1	1	8	-
Position	20	4.35	87	38	-
HeightH	1	1	1	10	-
FindAngle	5	1.40	7	14	-
FirstCondition	4	1.25	5	30	-
SecondCondition	5	1.80	9	23	-
SlopeIntensityIS	5	1	5	12	-
IntensityLocIL	1	1	1	4	-
LocationIntensityBI	2	1	2	6	-
VisualContrast	1	1	1	7	-
ObservJam	4	1.75	7	15	1
ThirdCondition	4	1.25	5	22	-
SendReports	12	2.75	33	37	-
Observation	19	8.37	159	44	1
Movement	8	2.88	23	17	-
PrepareOutput	14	4	56	39	2
Initialization	23	2.48	57	83	-
Restoration	1	1	1	8	-

TABLE 10 - METRICS FOR VERSION 1

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
TotalRestoredCasualtiesFF	7	3.29	23	17	-
Coefficient	7	2.86	20	15	-
NumSquadsRestoringNF	4	1.75	7	10	-
RestoreSuppAmtFS	5	2.20	11	15	-
RestoreFactorF	11	3.64	40	28	2
Attrition	1	1	1	10	1
SetFiredUponCoords	12	5.33	64	21	2
AssignLLCoords	4	1.25	5	15	-
KilledK	10	4.60	46	19	-
CalcEndurE	6	2.67	16	15	-
NumKillersNK	6	2.67	16	15	-
KillersAvailKA	8	3.50	28	21	-
TimesKillersUsedKU	10	4	40	22	-
TotalWeapInUseNW	9	4	36	20	1
Communication	1	1	1	10	-
TotalSquadsSendingNS	5	2.20	11	13	-
TotalSquadsReceiveNR	5	2.20	11	13	-
TotalSquadsJammingNJ	5	2.20	11	13	-
TotalSquadsProcessingNP	5	2.20	11	13	-
PutIntoList	6	1.33	8	29	1
SendMsgs	19	4.63	88	75	-
ProcessCommandMessages	5	2.20	11	21	-
ProcessReportMessages	10	3.80	38	48	-
MsgReceiptDelayRD	6	1.67	10	22	-
ManipProcessList	18	3.72	67	44	-
PutMsgOnSentLL	4	1.75	7	16	-
ManipMsgQueue	7	2.86	20	23	-
Update	15	1.93	29	59	1
InstantiateCommandMsg	6	1.33	8	44	3
ClearDeadSquads	8	2.25	18	20	-
ForEachSquad	10	1.70	17	25	-
ForEachWeap	6	1.67	10	33	-
UpdateArmyValues	3	1.33	4	18	1
Conflict	2	1	2	63	-

TABLE 11 - METRICS FOR VERSION 2

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Conflict	1	1	1	81	-
AttritInit	5	2.20	11	31	-
MinReal	2	1	2	4	-
MinInt	2	1	2	4	-
Max	2	1	2	4	-
MaxInt	2	1	2	4	-
Roof	3	1.33	4	6	-
Floor	3	1.33	4	4	-
Dist	1	1	1	4	-
Alt	5	1	5	15	2
TMove	3	1	3	12	-
PosIntens	6	1	6	20	-
WTotal	6	2	12	17	-
WMove	3	1.33	4	11	-
WObs	3	1.33	4	12	-
ScaleSquad	10	2.30	23	43	-
Positioning	4	1.75	7	12	-
Velocity	4	1.75	7	15	-
XMove	1	1	1	6	-
YMove	1	1	1	6	-
Movement	8	2.88	23	23	-
CalcContrast	6	2.67	16	10	-
Observation	14	5.35	75	29	-
CanJSeek	1	1	1	9	-
AngleBigEnough	17	4.70	80	20	3
Slope	1	1	1	4	-
FindPt	1	1	1	7	-
NoObstacles	3	1	3	18	-
Height	2	1	2	7	-
Ojamming	5	2	10	18	-
NoObsJammed	2	1	2	17	-
Attrition	1	1	1	5	-
AttritInflict	2	1	2	8	-
Weapons	15	4.80	72	35	-
FireCoord	14	4.93	69	40	-
Suffering	20	5.65	113	58	-
Restoration	18	4.67	84	47	-
Communication	1	1	1	8	-

TABLE 11 - METRICS FOR VERSION 2

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
UpdateComm	8	3.25	26	26	1
AddToQ	7	1.57	11	15	-
CreateReports	10	3.10	31	29	1
CreateCommands	5	2.20	11	21	1
PullFromQ	9	2	18	26	-
RellayMessages	15	3.80	57	40	1
ConsumeReports	14	6	84	36	-
ConsumeCommands	12	4.92	59	40	1
Simulation	2	1	2	15	-
Initialization	3	1.33	4	15	-
PosInit	3	1.33	4	12	-
MoveInit	1	1	1	11	1
ObsInit	2	1	2	8	-
CommInit	1	1	1	15	-
MoveOut	11	3.36	37	26	-
Output	1	1	1	5	-
AttritOut	5	2.20	11	20	-

TABLE 12 - METRICS FOR VERSION 3

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
max	2	1	2	4	-
min	2	1	2	4	-
Distance	1	1	1	4	-
Cieling	2	1	2	8	-
findA	5	1	5	12	4
Altitude	1	1	1	6	-
BI	1	1	1	11	2
TM	2	1	1	13	1
WF	4	1.75	7	13	-
WM	2	1	2	7	-
WO	2	1	2	7	-
Change	1	1	1	5	-
SubAngle	13	2.46	32	54	1
InitRec	1	1	1	14	-
Output	17	3.47	59	42	-
DataUpdate	13	3.62	47	47	-
ScanQueue	14	2.86	40	16	-
PutInQueue	3	1	3	12	1
BatPosition	15	3.87	58	59	2
FollowCommandMessages	28	3.07	86	119	2
positioning	3	1.33	4	6	-
ReceiveMessages	12	[illegible]5	27	52	1
Sighting	14	[illegible]21	59	53	1
CompareRecDMessages	15	6.27	94	39	-
Observe	1	1	1	5	-
SendObservations	10	2.80	28	25	-
SendOrders	6	2.50	15	23	-
SendMessages	1	1	1	5	-
Update	2	1	2	7	-
DoDamage	1	1	1	6	-
WeaponSighting	14	3.07	43	38	5
Summation	2	1	2	13	-
Attrition	9	2.67	24	12	-
Jam	2	1	2	16	-
Move	2	1	2	13	-
Restore	4	1.75	7	14	1
PerformPassiveFunction	1	1	1	5	-
Aggression	6	2.67	16	11	-

TABLE 12 - METRICS FOR VERSION 3

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
DoAction	1	1	1	5	-
InitVals	11	3.36	37	53	3
Conflict	3	1.33	4	64	2
ScanCQueue	7	1.57	11	14	-
NM	11	3.09	34	14	1

TABLE 13 - METRICS FOR VERSION 4

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Conflict	2	1	2	106	-
IsDestroyed	2	1	2	5	-
IsCasualty	10	2.40	24	4	-
Altitude	11	1	11	19	1
Distance	1	1	1	7	-
WSeverity	5	1.60	8	23	-
WEObservation	5	2	10	16	-
WEMovement	5	2	10	15	-
CalcVelocity	5	2.20	11	18	-
MTerrain	8	1.25	10	15	-
SlopeIntensity	3	1	3	11	-
AltIntensity	1	1	1	4	-
LocIntensity	1	1	1	4	-
IntensityOfLocation	1	1	1	4	-
Ceiling	2	1	2	4	-
PositionSquads	28	1.96	55	91	-
InitVisual	3	1.33	4	10	1
Transfer	13	4.31	56	46	1
BattalionSize	6	2.67	16	14	-
PrepOutput	12	4.17	50	38	-
Attrition	1	1	1	5	-
SetCoordinates	13	5.62	73	39	1
Inflict	1	1	1	5	-
WeaponCount	1	1	1	5	-
WeapUsage	4	1.25	5	9	-
AvailableWeapons	3	1.33	4	13	-
WeaponInflict	19	5.21	99	42	-
Suffer	6	2.67	16	13	-
Endurance	2	1	2	7	-
SquadDamage	11	4.09	45	31	-
InitFireList	7	3.14	22	15	-
Communications	22	4.23	93	45	-
ProcessMsg	12	3.33	40	45	3
SendMsg	8	2	16	36	-
QueueMsg	12	2.75	33	63	1
ReceiveMsg	13	2.30	30	60	-
AddToList	8	1.63	13	46	1
CmdReplace	14	1.21	17	55	2

TABLE 13 - METRICS FOR VERSION 4

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
ReportMessages	9	1.89	17	27	-
CommandMessages	7	1.43	10	23	-
Movement	8	2.88	23	28	-
Observation	15	5.53	83	30	6
ValidObservation	4	1.75	7	12	1
CheckHeight	2	1	2	9	-
ObsJamming	7	2.71	19	18	1
Angle	18	2.61	47	45	-
Line	8	2.13	17	20	2
ObsContrast	3	1	3	5	-
LocationList	3	1.33	4	18	1
VisualContrast	8	3.38	27	19	-
NewCasualties	3	1.33	4	12	-
TotalCasualties	3	1.33	4	11	-
RestoreAmount	6	2.17	13	12	-
RestoreSupplies	4	1.25	5	13	-
SquadFixers	1	1	1	7	-
Restoration	9	3.22	29	26	-
Initialize	24	1.83	44	63	-

TABLE 14 - METRICS FOR VERSION 5

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Conflict	4	1	4	65	-
Distance	3	1	3	4	-
Altitude	1	1	1	13	3
CompWeath	5	9	45	18	-
Position	16	1.69	27	39	-
Simulate	9	2.33	21	22	-
ChangeOld	1	1	1	10	-
ChangeSquad	1	1	1	8	-
Attrition	5	1.40	7	6	-
Suffer	11	4.82	53	36	-
Inflict	27	3.52	95	94	2
Comnicat	40	3.35	134	157	5
StoreMess	5	2.20	11	23	-
ReprtMess	6	1.17	7	21	-
ComndMess	5	2.20	11	23	-
UpdateCommVars	1	1	1	12	-
Movement	11	2.09	23	29	-
WeffMov	2	1	2	7	-
TeffMov	3	1	3	13	-
Observation	49	6.98	342	151	-
SpacePoints	7	2	14	36	2
IntnstyLoc	1	1	1	13	2
WeffObs	2	1	2	6	-
Restoration	14	2.57	36	32	1
Wear	10	3.6	36	21	-
Validate	89	2.73	243	70	4
SetInitialValues	8	2.50	20	38	3
OutputResults	11	3.91	43	37	1

TABLE 15 - METRICS FOR VERSION 6

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Conflict	1	1	1	59	-
Min	2	1	2	4	-
Max	2	1	2	4	-
IMin	2	1	2	4	-
IMax	2	1	2	4	-
Ceiling	3	1.33	4	6	-
Distance	1	1	1	7	-
Height	5	1.40	7	13	-
UpdateBattalionVelocity	3	1.33	4	13	-
CheckBattConstants	24	1.17	28	24	-
AlignSquads	13	2	26	31	-
InitBattalion	23	1.96	45	43	3
Initialize	17	1.29	22	28	-
CreateLosList	2	1	2	10	-
PerformSimulation	3	1.33	4	20	1
UpdateWeather	1	1	1	6	-
UpdatePresentEvents	4	1.75	7	23	-
AddNewEvents	3	1.33	4	14	-
PerformOneDt	1	1	1	9	-
WeatherSeverity	6	1.67	10	17	-
Movement	4	1.75	7	5	-
MoveBattalion	1	1	1	15	-
TEOnMovement	2	1	2	11	-
WEOnMovement	2	1	2	10	-
Observation	4	1.75	7	10	-
GenObsList	9	3.22	29	25	-
Observable	3	1.33	4	11	1
AngleSubGreater	16	3.69	59	42	-
UpdateLOSList	2	1	2	14	1
LOSClear	3	1	3	20	1
CntrstOK	3	1	3	21	-
LocationIntensity	1	1	1	16	-
ObsJamming	3	1.33	4	12	-
WEOnObservation	2	1	2	10	-
IncludeCommObs	7	2.57	18	27	1
CollectFinishedReport	13	3.85	50	13	-
UpdateLL	8	3.88	31	26	-
SumObsToNextBatt	12	3.25	39	27	-

TABLE 15 - METRICS FOR VERSION 6

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Attrition	4	1.75	7	9	-
NumWeapons	6	1.33	8	13	1
TrackWeapons	2	1	2	12	1
UpdateUseList	5	1.40	7	14	-
ChooseTargets	14	3.29	46	33	1
SufferAttrition	11	3.55	39	38	1
Restoration	3	1.33	4	9	-
NewNumFixers	4	1.75	7	11	-
ApportionFixing	10	3	30	28	-
RemoveDestroyedSquads	8	1.50	12	19	-
Communication	1	1	1	8	-
SendCommunications	8	2.38	19	14	-
SendReport	4	1.25	5	20	-
NewNumSend	4	1.75	7	10	-
SendCommand	2	1	2	18	-
ReceiveCommunications	1	1	1	8	-
FindReceivingDelay	6	2	12	29	1
ReceiveReports	7	2.57	18	32	-
ReceiveCommands	7	2.57	18	33	-
UpdateNumVars	7	2.86	20	13	-
ProcessCommunications	1	1	1	7	-
HandleQueuing	1	1	1	7	-
QueueReports	6	2.67	16	21	-
FindQueueSpot	5	1.60	8	11	-
QueueCommands	6	2.33	14	21	-
FindQueue	5	2.20	11	11	-
ProcessingDelay	4	1.75	7	11	-
ProcessMessages	9	3.11	28	30	-
FindNextReport	6	2.16	13	11	-
FindNextCommand	6	2.16	13	11	-
TakeACommand	2	1	2	9	-
TakeAReport	2	1	2	9	-
NewNumProcessing	3	1.33	4	10	-
PrepareForNextDT	1	1	1	10	1
CollectCommands	7	2.57	18	27	-
CollectCommand	10	2.30	23	23	2
PutInCommand	3	1.33	4	20	2
DetermineOutput	7	3	21	28	-

TABLE 16 - METRICS FOR VERSION 7

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Conflict	1	1	1	23	1
float	1	1	1	4	-
WriteError	11	1	11	19	-
CheckMessages	6	2	12	16	-
CheckWeather	6	1.67	10	10	-
CheckParams	9	1	9	18	-
Process	11	2.91	32	55	-
InvalidPosition	5	1	5	7	-
CheckBatallionInfo	1	1	1	9	-
CheckNArmy	4	1.50	6	12	1
CheckPerBatallion	30	1	30	40	3
CheckPerSquad	6	1.67	10	13	-
CheckPerEnemy	5	1.80	9	9	-
CheckPerWeapon	10	2.50	25	16	-
GetTs	1	1	1	8	2
Altitude	1	1	1	16	-
Distance	1	1	1	7	-
WFactor	5	1.80	9	17	-
WXPosition	1	1	1	4	-
WYPosition	1	1	1	4	-
Height	1	1	1	9	-
MakeInt	2	1	2	4	-
Initialize	13	2.46	32	34	1
Velocity	4	1.75	7	14	-
SetSquad	3	1.33	4	27	-
SetPosition	11	2	22	39	-
InitializeWeapData	3	1.33	4	11	1
PositionSquadrons	4	1.75	7	16	-
SetPosition	11	2	22	37	-
Observation	15	3.67	55	27	3
VisibleSquad	5	1.20	6	23	1
SubAngle	9	2.56	23	11	-
GetAngle	29	1	29	23	-
Series	2	1	2	16	1
ClearView	5	1.40	7	16	-
OContrast	4	1.25	5	22	-
Intensity	1	1	1	16	-
OJamming	3	1.33	4	11	-

TABLE 16 - METRICS FOR VERSION 7

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
WObserve	2	1	2	7	-
CommandMess	24	5.29	127	56	1
RecDelay	2	1	2	13	-
JammedSquads	5	2	10	17	-
Incorporate	4	1.25	5	25	1
InitializeWeData	3	1.33	4	11	-
Attrition	1	1	1	13	-
InflictAttrition	15	3.13	47	26	3
SetFire	4	1.50	6	23	-
CalcNumObserv	8	2.75	22	12	-
CalcNumWeapToUse	11	3.55	39	27	1
Min	2	1	2	4	-
CalculateDamages	13	3.69	48	14	-
InRange	2	1	2	10	-
UpdateInfo	5	1	5	16	-
UpdateBattalion	5	1.60	8	32	1
DeltaFixSuppl	2	1	2	11	-
UpdatePosition	1	1	1	16	-
SetRestoration	3	1.33	4	18	1
ChangeSquadData	16	3.69	59	35	-
SetDamage	2	1	2	7	-
Movement	2	1	2	17	-
WMovement	2	1	2	8	-
TerrEffect	2	1	2	11	-
SetOutput	3	1.33	4	16	-
GetDifference	6	2	12	28	-
Greatest	2	1	2	4	-
Least	2	1	2	4	-
GetStatus	5	1	5	13	-
Distance	1	1	1	6	-

TABLE 17 - METRICS FOR VERSION 8

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Conflict	6	2.67	16	80	-
UpdateU	3	1.33	4	12	-
SquadPos	17	1.88	32	42	-
LinearDistance	2	1	2	4	-
Altitude	1	1	1	14	2
Velocity	5	2.20	11	20	-
WSevFactor	5	1.80	9	19	-
SetKU	2	1	2	15	-
InitVariables	6	2.17	13	45	-
CalcBI	3	1	3	17	2
VisContrast	1	1	1	7	-
Movement	1	1	1	13	1
TerrainEffect	4	1.50	6	11	-
WeatherMoveEffect	2	1	2	9	-
Observation	11	3.82	42	45	1
FindAngle	19	1.53	29	20	-
SumObJam	5	2.20	11	22	1
WObsEffect	1	1	1	6	-
Observable	5	1.60	8	17	-
Height	1	1	1	4	-
Attrition	8	2.88	23	22	1
NumOfWeapons	3	1	3	8	-
SetAttacked	3	1	3	16	-
LengthOfList	2	1	2	10	-
ResetObserveLists	3	1.33	4	15	-
Restoration	8	2.50	20	20	-
UpdateVars	10	2.60	26	46	-
UpdateFS	3	1	3	14	-
UpdateFF	4	1.50	6	12	-
UpdateK	7	3.14	22	24	-
CalcCas	6	1.67	10	15	-
CalcBK	8	2.63	21	13	-
UpdateNums	1	1	1	4	-
UpdateE	3	1.33	4	13	-
UpdateKA	5	1.80	9	21	-
UpdateKU	6	1.83	11	16	-
ClearAttackLists	3	1.33	4	15	-
PrepareOutput	3	1.33	4	10	-

TABLE 17 - METRICS FOR VERSION 8

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
SetLocation	10	2.90	29	42	1
SetStatus	3	1.33	4	15	-
Ceiling	2	1	2	6	-
InsertMsg	8	2.25	18	28	-
CommandMsg	3	1.33	4	11	-
InsertCom	1	1	1	18	-
ReportMsg	2	1	2	7	-
InsertRep	3	1	3	27	-
Communication	1	1	1	7	-
ReceiveDelay	6	1.33	8	22	-
CalRDelay	4	1.50	6	27	-
QueDelay	6	1.50	9	20	-
PutQueue	9	2.44	22	25	4
ProcessQue	5	2	10	20	-
ProcessMsg	9	2.22	20	28	-
MergeRepMsg	3	1.33	4	9	-
MergeComMsg	4	1	4	34	1

TABLE 18 - METRICS FOR VERSION 9

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Conflict	2	1	2	52	-
D	1	1	1	8	-
Ceiling	3	1.33	4	4	-
WF	6	2.17	13	18	-
WMorWO	8	2.13	17	14	-
SquadsPos	10	2	20	40	1
CalcVg	5	2	10	13	2
Initialize	11	3.36	37	33	-
CheckBattalions	33	15.17	527	15	-
CheckData	9	1.56	14	11	2
Z	1	1	1	12	2
TM	2	1	2	8	2
BI	1	1	1	11	2
Movement	5	2.20	11	23	-
Observation	16	4.56	73	39	2
Restoration	15	4.40	66	45	1
GetAngleCornerPts	7	1.57	11	23	-
Overlap	23	3.08	71	38	-
CheckAngle	3	1	3	11	2
CheckZ	4	1.25	5	14	1
GetOJ	4	1.50	6	11	-
CheckContrasts	3	1	3	8	-
CalcFgj	6	2	12	12	-
UpdateFFg	3	1.33	4	11	-
UpdateNFg	2	1	2	8	-
UpdateFSg	3	1	3	11	-
GetLocation	2	1	2	11	-
GetLenLL	2	1	2	10	-
Attrition	1	1	1	8	-
Inflict	9	4	36	25	-
GetNW	8	2.75	22	13	1
Suffer	4	1.75	7	6	-
WearTear	6	2.67	16	10	-
AddLL	4	1.75	7	15	-
Send	10	2.70	27	29	1
HasObs	2	1	2	8	-
InsertQ	5	2	10	17	-
Receive	16	3.94	63	83	1

TABLE 18 - METRICS FOR VERSION 9

NAME OF MODULE	v(G)	BW	NC	STM	FAULTS
Process	16	3.69	59	57	1
ChangeBatEnv	4	1	4	23	-
Communicate	1	1	1	6	-
OutputReport	18	2.78	50	35	2

TABLE 19 - ANALYSIS OF VARIANCE OF FAULTS WITH v(G)

Version Number ====== Source of Variation Between Groups Within Groups	Sum of Squares	Degrees of Freedom	Mean Square	F-Ratio	Significance Level
1	16.82341 14.82936	18 53	0.93463 0.27979	3.340	0.0003
2	11.18333 5.61667	16 38	0.69896 0.14781	4.729	0.0000
3	30.54651 25.50000	15 27	2.03643 0.94444	2.156	0.0399
4	41.41353 12.09524	19 37	2.17966 0.32689	6.668	0.0000
5	45.24762 5.46667	16 11	2.82797 0.49697	5.690	0.0029
6	12.14627 11.59058	17 58	0.71448 0.19984	3.575	0.0001
7	24.12936 10.75299	15 52	1.60862 0.20678	7.779	0.0000
8	8.03576 18.52564	12 44	0.66965 0.42104	1.590	0.1298

TABLE 20 - ANALYSIS OF VARIANCE OF FAULTS WITH BW

Version Number Source of Variation Between Groups Within Groups	Sum of Squares	Degrees of Freedom	Mean Square	F-Ratio	Significance Level
1	21.64520 10.00758	30 41	0.72151 0.24408	2.956	0.0007
2	11.4933 5.30667	20 34	0.57467 0.15608	3.682	0.0004
3	29.95560 26.09091	17 25	1.76209 1.04363	1.688	0.1140
4	51.37127 2.13750	32 24	1.60535 0.08906	18.025	0.0000
5	44.31428 6.40000	18 9	2.46190 0.71111	3.462	0.0309
6	14.60223 9.134615	28 47	0.52151 0.19435	2.683	0.0014
7	19.03387 15.84849	20 47	0.95169 0.33720	2.822	0.0018
8	17.69777 8.86364	22 34	0.80444 0.26069	3.086	0.0016

TABLE 21 - ANALYSIS OF VARIANCE OF FAULTS WITH NC

Version Number ====== Source of Variation	Sum of Squares	Degrees of Freedom	Mean Square	F-Ratio	Significance Level
Between Groups					
Within Groups					
1	22.04722 9.60556	32 39	0.68898 0.24629	2.797	0.0012
2	15.17592 1.58333	23 30	0.65982 0.05278	12.502	0.0000
3	48.98545 7.53636	21 21	2.32835 0.34256	6.797	0.0000
4	48.64211 4.86667	29 27	1.67731 0.18025	9.306	0.0000
5	45.41429 5.30000	19 8	2.39023 0.66250	3.608	0.0343
6	14.16322 9.57362	28 47	0.50583 0.20369	2.483	0.0029
7	26.14231 7.60769	21 42	1.24487 0.18114	6.873	0.0000
8	9.30585 17.2556	19 37	0.48978 0.46636	1.050	0.4345

TABLE 22 - ANALYSIS OF VARIANCE OF FAULTS WITH STM

Version Number Source of Variation	Sum of Squares	Degrees of Freedom	Mean Square	F-Ratio	Significance Level
Between Groups Within Groups					
1	18.06944 13.58333	34 37	0.53145 0.36712	1.448	0.1361
2	5.13333 11.66667	24 30	0.21389 0.38889	0.550	0.9316
3	41.62985 14.41667	21 21	1.98237 0.68651	2.888	0.0094
4	45.17544 8.33333	32 24	1.41173 0.34722	4.066	0.0004
5	46.04762 4.66667	22 5	2.09307 0.93333	2.243	0.1884
6	14.78880 8.94881	32 43	0.46213 0.20812	2.221	0.0075
7	19.03387 15.84849	20 47	0.95169 0.33720	2.822	0.0018
8	20.22807 6.33333	26 30	0.77800 0.21111	3.685	0.0004

LIST OF REFERENCES

[1] Avizienis, A., and Kelly, J.P., "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer*, Vol.17, No.8, pp.67-80, August 1984.

[2] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand Reinhold, 1984.

[3] Clarke, L.,Podgurski, A.,Richardson, D., and Zeil, S., "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering*, Vol.15, No.11, pp.1318-1332, November 1989.

[4] Myers, G., *The Art of Software Testing*, John Wiley & Sons, Inc., 1979.

[5] Thayer, T., Lipow, M., and Nelson, E., *Software Reliability*, Vol.2, pp.271, North-Holland Publishing Company, 1978.

[6] Perlis, A., Sayward, F., and Shaw, M., *Software Metrics: An Analysis and Evaluation*, The MIT Press, 1981.

[7] Li, H., and Cheung, W., "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, Vol.SE-13, No.6, pp.697-708, June 1987.

[8] Druffel, L., Redwine, S., and Riddle, W., "The STARS Program: Overview and Rationale," *Computer*, pp.21-29, November 1983.

[9] Jensen, H., and Vairaven, K., "An experimental study of software metrics for real time software," *IEEE Transactions on Software Engineering*, Vol.SE-11, No.2, pp.649-653, February 1985.

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

[11] Gilb, T., *Software Metrics*, Winthrop Publishers, Inc., 1977.

[12] Nejmeh, B., "NPATH: A Measure of Execution Path Complexity and its Applications," *Communications of the ACM*, Vol.31, No.2, pp.188-200, February 1988.

[13] Zolnowski, J., and Simmons, D., "Taking the Measure of Program Complexity," *Proceedings of the National Computer Conference*, pp.329-336, 1981.

[14] Dunsmore, H., and Gannon, J., "Analysis of the Effects of Programming Factors on Programming Effort," *The Journal of Systems and Software*, Vol.1, No.2, pp.141-153, 1980.

[15] Woodward, M., Hennel, M., and Hedley, D., "A Measure of Control Flow Complexity in Program TEXT," *IEEE Transactions on Software Engineering*, Vol.SE-5, No.1, pp.45-50, January 1979.

[16] Baker, A., and Zweben, S., "A Comparison of Measures of Control Flow Complexity," *IEEE Transactions on Software Engineering*, Vol.SE-6, No.6, pp.506-512, November 1980.

[17] Harrison, W, and Magel, K., "A Complexity Measure Based on Nesting Level," *ACM SIGPLAN Notices*, pp.63-74, March 1981.

[18] Schneidewind, N., and Hoffman, H., "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions on Software Engineering*, Vol.SE-5, No.3, pp.276-286, May 1979.

[19] Elshoff, J., "An Analysis of Some Commercial PL/1 Programs," *IEEE Transactions on Software Engineering*, Vol.SE-2, No.2, pp.113-120, June 1976.

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

[21] Basili, V., and Turner, A., "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Transactions on Software Engineering*, Vol.SE-1, No.4, pp.390-396, December 1975.

[22] McClure, C., "A Model for Program Complexity Analysis," *Proceedings of the Third International Conference on Software Engineering*, Atlanta, Georgia, pp.149-157, May 1978.

[23] Halstead, M., *Elements of Software Science*, Elsevier North-Holland, Inc., 1977.

[24] Christensen, K., Fitsos, G., and Smith, C., "A Perspective on Software Science," *IBM Systems Journal*, Vol.20, No.4, pp.372-387, 1981.

[25] Lipow, M., "Number of Faults per Line of Code," *IEEE Transactions on Reliability*, Vol.SE-8, pp.437-439, June 1982.

[26] Conte, S., Dunsmore, H., and Shen, V., *Software Engineering Metrics and Models*, The Benjamin/Cummings Publishing Company, Inc., 1986.

[27] Basili, V., and Reiter, R., "An Investigation of Human Factors in Software Development," *IEEE Computer*, Vol.12, No.12, pp.21-38, December 1979.

[28] Flaherty, M., "Programming Process Productivity Measurement System for System/370," *IBM Systems Journal*, Vol.24, No.2, pp.168-175. 1985.

[29] Myers, G., "An Extension to the Cyclomatic Measure of Program Complexity," *SIGPLAN Notices*, Vol.12, No.10, pp.61-64, October 1977.

[30] Hansen, W., "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)," *ACM SIGPLAN Notices*, Vol.13, No.3, pp.29-33, March 1978.

[31] Oviedo, E., "Control Flow, Data Flow, and Program Complexity," *Proceedings of the IEEE Computer Software and Applications Conference*, pp.146-152, November 1980.

[32] Ramamurthy, B., and Melton, A., "A Synthesis of Software Science Measures and the Cyclomatic Number," *IEEE Transactions on Software Engineering*, Vol.14, No.8, pp.1116-1121, August 1988.

[33] Basili, V., and Hutchens, D., "An Empirical Study of a Syntactic Complexity Family," *IEEE Transactions on Software Engineering*, Vol.SE-9, No.6, pp.664-672, November 1983.

[34] Shimeall, T., and Leveson, N., "An Empirical Comparison of Software Fault Tolerance and Fault Elimination," *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pp.180-187, July 1988.

[35] Naval Postgraduate School Report NPS52-89-047, *An Empirical Comparison of Software Fault Tolerance and Fault Elimination*, by Shimeall, T., and Leveson, N., p.35, July 1989.

[36] Henry, S., and Selig, C., "Predicting Source-Code Complexity at the Design Stage," *IEEE Software*, pp.36-44, March 1990.

[37] Walsh, T., "A Software Reliability Study Using a Complexity Measure," *Proceedings of the 1979 National Computer Conference*, pp.761-768, Montvale - New Jersey, AFIPS Press, 1979.

[38] Rambo, R., Buckley, P., and Branyan, E., "Establishment and Validation of Software Metric Factors," *Proceedings of the International Society of Parametric Analysts - Seventh Annual Conference*, pp.406-417, Germantown, Maryland, 1985.

[39] Grady, R., and Caswell, D., *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall,Inc., 1987.

[40] Butler, L., "Software Quality Assurance Cyclomatic Complexity of Computer Program," *Proceedings of the IEEE 1983 National Aerospace and Electronics Conference*, Vol.2, pp.867-873, May 1983.

[41] Fenick, S., "Implementing Management Metrics: An Army Program," *IEEE Software*, pp.65-72, March 1990.

[42] Rubey, R., "Quantitative Aspects of Software Validation," *Proceedings of the International Conference on Reliable Software*, Vol.10, No.6, pp.246-251, June 1975.

[43] Endres, A., "An Analysis of Errors and their Causes in System Programs," *Proceedings of the International Conference on Reliable Software*, Vol.10, No.6, pp.327-336, June 1975.

[44] Grady, R., "Work-Product Analysis: The Philosopher's Stone of Software," *IEEE Software*, pp.26-34, March 1990.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Computer Science Department Code CS Naval Postgraduate School Monterey, California 93943-5000	1
4.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
5.	Timothy J. Shimeall Computer Science Department Code CS/Sm Naval Postgraduate School Monterey, California 93943	10
6.	Leigh W. Bradbury Computer Science Department Code CS/Bb Naval Postgraduate School Monterey, California 93943	1
7.	Nancy Leveson ICS Department University of California, Irvine Irvine, California 92717	1

8. Richard Kemmerer — 1
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, California 93106

9. Elaine Weyuker — 1
Department of Computer Science
Courant Institute of Mathematical Sciences
New York, New York 10012

10. Phyllis Frankl — 1
Polytechnic University
333 Jay Street
Brooklin, New York 11201

11. Larry Morell — 1
Department of Computer Science
College of William & Mary
Williamsburg, Virginia 23186

12. Alberto T. B. de Almeida — 2
R. Proj. à Esc. Sec. da Parede, Lt.3 R/C Dt.
2775 Parede, Portugal

13. Direcção do Serviço de Pessoal - 1 Rep. — 1
Edifício da Administração Central de Marinha
Praça do Comércio, 1100 Lisboa, Portugal