A Methodology for Integrating Maintainability into Large-Scale Software Using Software Metrics

by

John Allan Lewis

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Science and Applications

APPROVED:

Dr. Sallie M. Henry, Chairperson

Dr. James D. Arthur

Dr. Dennis G. Kafura

Blacksburg, Virginia

A Methodology for Integrating Maintainability into Large-Scale Software Using Software Metrics

by
John Allan Lewis
Dr. Sallie M. Henry, Chairperson
Computer Science and Applications

(ABSTRACT)

Maintainability must be integrated into software as early in the development life cycle as possible to avoid overwhelming maintenance costs at later stages. This research describes a methodology which assists in the development of maintainable systems and does so without disrupting industry standard development techniques. The process uses software metrics and iterative enhancement concepts to reduce the complexity of high-level language code, making it less error-prone and more maintainable. The experiment uses large-scale system software from a major software producer.

Acknowledgements

I would like to thank my major professor, Dr. Sallie Henry, for the guidance and support she has provided throughout this research. She has been a great advisor and friend for several years. I also thank Dr. Sean Arthur and Dr. Dennis Kafura for serving on my master's committee.

Thanks also go to and provided much needed managerial backing and was my sounding board, guidance counselor, technical advisor, friend and fellow software engineering geek. I also thank and for their friendship and support during my (multiple) visits.

and from the Virginia Tech Department of Statistics supplied the important statistical framework used in the data analysis portion of this study.

provided much needed friendship which helped make the last days of this research tolerable. I also thank and for providing an anchor to sanity over the phone from afar.

Finally, I would like to thank my family, especially my parents and for their tolerance and support throughout my extended education.

Table of Contents

List of Figures

Chapter 1: Introduction

I. Introduction

Software maintenance is recognized as the single most expensive factor in a software project's life. Estimates place the cost of maintenance from 50 to 70 percent of the total life cycle cost of a software system [HALD88]. A survey of software managers indicates that they realize the crucial role which maintenance plays in their budgets. They also stress the problem that user demands for enhancements and extensions is often overwhelming. A majority even agree that software maintenance is more important than new development [LIEB78].

Given that software development is necessarily guided by budget and time constraints, the criteria for internal software acceptance is often that it satisfy the requirements of specific tests. However, correct software according to the test plan does *not* guarantee good software in terms of maintenance or other quality factors [MYEG76]. Recognizing that quality cannot be assured by testing alone, software should be delivered to a set of requirements that include criteria other than test sets.

The bottom line is that maintenance is the largest financial drain in a software system's life cycle, and the creation of software must be approached from a perspective that minimizes maintenance costs. The control of software maintenance costs should begin long before the product is delivered to the customer. Unfortunately, the cost benefits of controlling maintenance at the development stage is difficult for software management to realize [MUNJ78].

Software maintenance can be defined as any post-release change to software. Swanson [SWAE76] divides maintenance tasks into three categories based on the motivation for the changes required: (1) corrective maintenance, performed when errors have been found in the software; (2) adaptive maintenance, performed in response to changing needs of the users; and (3) perfective maintenance, performed to eliminate inefficiencies, enhance performance, or improve maintainability. All three types of maintenance require that someone, not necessarily a person familiar with the system, understand and manipulate the code.

Many techniques and criteria to promote maintainability have been proposed. The primary ones include structured design and implementation, guidelines on the use of data structures and parameters, information hiding, and appropriate documentation [LAMD88]. Used correctly, these techniques make programs easier to understand and modify. Simply put, certain software engineering concepts help to make source code less complex, and therefore more maintainable.

As the complexity level of a piece of software increases, the code becomes difficult to understand and is more likely to contain errors. Code containing errors must be modified, but that maintenance is non-trivial since the code is difficult to comprehend. To compound the problem further, it has been shown that programs cannot be made more maintainable by simply changing their code. Belady and Lehman establish that maintenance activities tend to *increase* the level of complexity of the code, so more maintenance is likely to be required [BELL76].

This snowball cycle is called the ripple effect of software maintenance which must be controlled at the earliest possible point in the life cycle [YAUS88]. Certainly, using software engineering techniques increase the maintainability of software, but they are difficult to measure and enforce. If maintainability is to be an expected consequence of software development, ways to measure it must become an integral part of the development cycle.

Software quality metrics, when defined and used correctly, have been shown to be good indicators of software complexity [CONS86]. Metrics are quantitative evaluations of software design or code based on some set of criteria which contribute to the software's complexity. If metric analysis results were required to conform to quality tolerances, the development process would not be compromised. Software metrics are discussed in greater detail later.

Maintainability is not an incidental result of other practices. It must be consciously integrated into a software project. Furthermore, deep-seated problems of maintainability require non-trivial solutions [MUNJ78]. This implies that the emphasis on controlling maintenance costs should shift toward the early stages of the development life cycle, as opposed to the post-production attention it currently receives.

Linger contends that a major problem with maintenance stems from a software management attitude which does not recognize software maintenance as a mature discipline [LINR88]. Software managers must realize that knowledgeable use of state-of-the-art tools and techniques will cause software to serve as well as survive.

This research is designed to explore the use of specific implementation techniques and tools to reduce the effort, and therefore the cost, required by the maintenance tasks of large-scale software projects.

II. Maintenance and the Software Life Cycle

Overstreet, et al. give an overview of the various software life cycles presented in current literature [OBRV86]. Software life cycle models describe the evolution of software systems as they are developed, implemented and maintained.

Probably the most popular life cycle is the waterfall model characterized by Boehm [BOEB76]. In this model, the various stages of evolution are defined as a series of steps beginning with system requirements and proceeding through preliminary and detailed designs, coding and debugging, testing, and finally operations and maintenance. Each stage is validated, and as problems are identified, previous steps are repeated as necessary.

Another model, the automation-based paradigm, incorporates program transformations from requirements-based prototypes into fully operational systems using automated techniques [BALR83]. The transformations involve repeated iterations of validation and maintenance of specification to create the final system. All of the technology needed to support this model does not yet exist, but it offers an important view of the software development process.

The two-leg model describes the process of abstracting the formal specification of software down two paths providing both an application concept and an operational system [LEAM84]. The path to the operational system is called reification and incorporates a backtracking concept for required modifications.

Boehm proposes the spiral model, incorporating refinements to the waterfall model [BOEB86]. The spiral model provides risk analysis at each stage of development which is represented by an outward spiral. Cumulative cost is represented by the radial dimension of the spiral, and the angular dimension represents the progress made in completing each stage of development.

Maintenance plays a consistent role in all of these models. An understanding of the development and maintenance processes which a software system endures throughout its existence is necessary in order to understand the proper point at which to consciously introduce maintainability.

According to the waterfall model, shown in Figure 1, and consistent with most industry development strategies, software is designed and implemented, then tests are run to verify that the software performs correctly. If the tests find errors, the code is corrected; or if an error is determined to stem from a design flaw, the design is modified, then the code is reimplemented. The tests are then run again. Finally, the system is released for consumer use. When errors are found by the users, a maintenance task is initiated to fix the problem. Once the problem is addressed, the software is reissued to the consumers, who are then likely to find additional or new problems.

Some effort has gone into the process of easing the maintenance tasks at the post-production stage. Cashman and Holt [CASP80] describe a communication-oriented system called MONSTR (MONitor for Software Trouble Reporting) which is designed to structure the environment for software maintenance. The system formulates the manner in which problems are reported, analyzed and solved based on communication protocols.

Selby and Basili describe a technique to localize errors during a maintenance activity by analyzing the source code [SELR88]. Measurements of data interaction, called data bindings, are used to estimate software coupling and cohesion. From these, a hierarchial description of a software system is obtained which helps determine error location.

Maintenance problems will never completely be eliminated and therefore such efforts are often helpful. However, concentrating on maintenance problems at the post-production stage is analogous to closing the barn door after the horse has escaped. Studies have shown that the earlier in

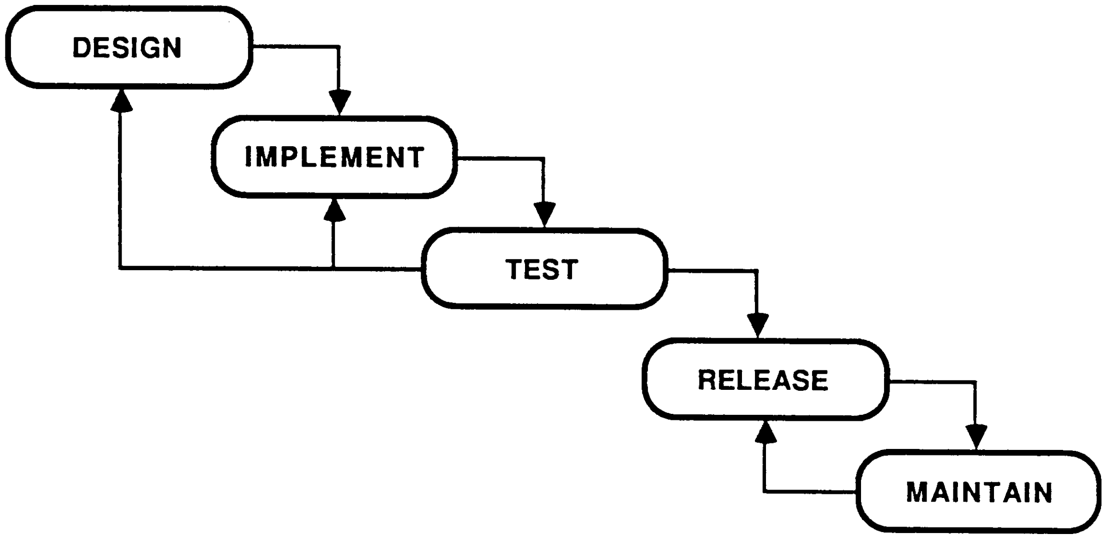


Figure 1. The classic waterfall model of software development.

the life cycle that errors are corrected, the less costly they are [LIPM79]. Therefore, the earlier that controls on the complexity of software are enforced, the less costly the error correction effort will be. The natural conclusion is to control software complexity at the time of its design.

Henry and Selig have demonstrated that it is possible to analyze software systems at the design stage under automated control [HENS88]. Using a programming design language (PDL) which can be statically analyzed with a software tool, student programs were evaluated with software quality metrics to determine complexity violations. These metrics correlate well with the metrics obtained from the actual code produced at implementation time. They also generate equations to predict source code quality from a PDL design.

Unfortunately, few software managers will deviate from their normal life cycle processes to accommodate the extra calculation of software metrics. PDL's such as the one used in Henry and Selig's research are syntactically specific and often require more time than necessary to communicate the same information in other ways. For realistic usage, the control of software complexity should disrupt as little of the current life cycle as possible.

Similar work shows that graphical design languages, which can more clearly communicate design details than text-based PDLs, can also be evaluated using metrics tools [HENS87]. Once again, however, a disruption of existing industry design processes is necessary to incorporate this technique unless the required software and hardware are already in use. Unfortunately, the main stream design techniques used in industry today do not lend themselves to automated evaluation.

Given that current design evaluation techniques are impractical, the next level of the life cycle can be investigated: implementation. All logic is eventually implemented and tested. Furthermore, high level language code is already syntactically specific. Therefore, code analysis at this stage is possible without disrupting the normal flow of software design and implementation. For large-scale software development, this analysis still produces results that can be used immediately.

Figure 2 shows a modified software development life cycle, integrating the code analysis into the basic waterfall model. In this scenario, as testing occurs, the code is analyzed to obtain complexity measures. Then, when changes are made in the design or code for error reasons, changes are also made that embrace the goal of reducing the complexity of the sections of software found to be above set complexity tolerances. Furthermore, since test results are indicative of quality to the degree that test cases exercise software attributes, the complexity measures are used to guide test case planning.

When referring to software development in industry, it is also important to realize that large-scale software projects often involve hundreds of thousands of lines of code, and that the simple life cycle presented so far does not capture the details of the process. To produce a practical methodology for integrating the use of metrics, these details must be taken into account. This is thoroughly discussed in Chapter 5.

Software metrics have been used quite successfully in the past as complexity measurements, and fit well into the scheme described here. It is now important to understand more about the use of software metrics.

III. Software Metrics

Software metrics are measurements of code complexity based on some set of characteristics of the code, providing a relative, quantitative evaluation [CONS86]. Some metrics concentrate on the amount of syntactic entities which

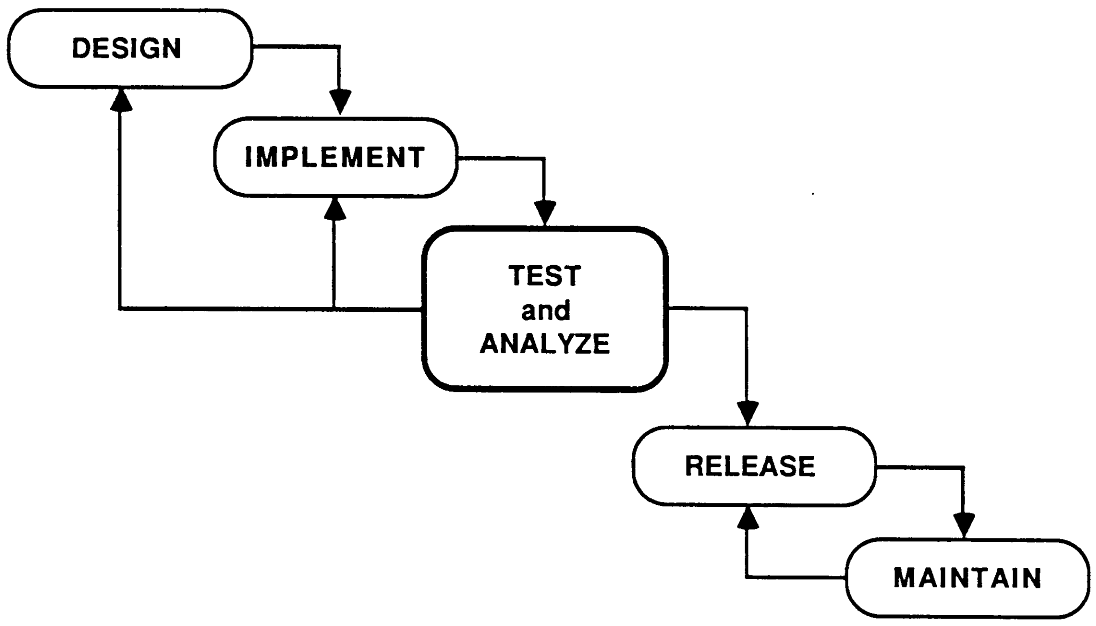


Figure 2. Metric analysis in the software development process.

must be understood in a procedure, and are called code metrics. Structure metrics are concerned with the structure of the software system being analyzed, including the quantity of information transferred among procedures. Still other evaluations combine the concepts of code and structure metrics. These are called hybrid metrics.

The metrics used in this study are fully described in Chapter 2, including how each is calculated and the exact characteristics each attempts to measure. It is very important that any use of a metric be accompanied by careful consideration of what that metric actually indicates about a procedure's complexity. For example, metrics which concentrate on length factors do not imply anything about the complexity of the communication between that procedure and other procedures, and vice versa.

There is no one metric that tells the whole complexity story about arbitrary source code, therefore several metrics are used in this research. Hybrid metrics, which combine several evaluations, are often better indicators for the same reason. Studies have indicated that no single metric is sufficient for indicating complexity [CANJ85].

Metrics can also be inconsistent among source languages, applications, or operating environments. A certain metric might be a very good indicator for one language because of certain structures or capabilities, but be mediocre for another. Similarly, a metric might be a good complexity indicator in a particular operating environment, but not as informative in another type of environment. Therefore, it is important to validate the metrics results to determine which metrics are best for a given language and system environment, and to understand why others do not yield such good results.

Calibration of metrics is also necessary. That is, certain tolerances should be determined for each metric, indicating that the procedures whose metrics fall out of this range are potential problems. Using tolerance levels is safer than a judgement based on a relative average across procedures since a large number of procedures in a system could have complexity levels above that which is actually reliable.

This calibration process is unspecific, and the values may be modified many times after the initial estimate. As the metrics are further used and understood in a particular environment, better judgements can be made concerning the safe metric tolerances.

Kafura and Reddy establish the link between the use of software metrics and the efforts of software maintenance [KAFD87]. The study explores medium-sized software systems and uses the informed input of experts who were intimately familiar with the system to validate the metric results. Kafura and Reddy conclude that the software metrics agreed with the general maintenance tasks required, that the metrics identify improper integration of functional enhancements, and that the metrics agree with the expert subjective understanding of the system.

Wake and Henry perform a similar study using software quality metrics on a small sample of production code from a major software vendor [WAKS88]. Using a maintenance history describing the modifications made to the code, Wake develops regression equations which successfully predict the need for maintenance in specific sections of the software. The equations calculate the predicted number of errors and predicted number of lines changed based on metric values.

Given the use of software quality metrics as a tool for integrating maintainability into large-scale software, we must now briefly address the current experiment.

IV. Research Data

This research is based on software designed and implemented by a major commercial software developer. The software is written in a high-level, in-house language called Mesa, which supports real-time, concurrent system activity.

Three important aspects of this research should be stressed. The first is the use of objective error data on which to base the results of the metrics validation. Second is the use of an automated tool to perform the metric collection. The third aspect is the use of realistic, large-scale data for the metric analysis, instead of the commonly seen test bed of student programs.

Within this study's software development scheme, distinct internal releases of the software are frozen for system testing and progress identification purposes. This research evaluates the code at a mid-development release point using software quality metrics to see if they correspond to actual modification needs as determined by the normal testing procedures. If the metrics identify the points at which maintenance is required, then the relationship would indicate that this technique is recognizing error-prone software.

Therefore, if code enhancement is performed as a result of metric analysis to reduce code complexity, fewer maintenance tasks would be required, and test cases concentrating on the highly complex code would uncover more errors. Furthermore, the inevitable maintenance tasks that are required eventually would be easier to handle because the complexity of the code is reduced.

V. Conclusion

Given the expensive nature of software maintenance, specific steps must be taken to reduce maintenance costs at some point in the software life cycle. Studies have shown that software which is highly complex is also error-prone and difficult to maintain. Therefore a process which controls the complexity of software reduces the maintenance effort.

Dealing with maintenance costs during post-production modifications only treats the symptoms, not the cause. However, for practical usage, disruption of industry standards for software development and testing must be avoided. Therefore, an appropriate point in the life cycle to consciously reduce software complexity and integrate maintainability is during the iterative cycles of error testing and functional enhancement.

The goals of this research are: (1) Design and implement a metrics collection tool for the Mesa language; (2) Validate that the metrics calculated by the tool indicate error-prone software; (3) Establish regression equations which predict the number of errors that are present in a section of code; and (4) Develop a methodology using these tools and techniques which integrates maintainability into large-scale software. This methodology must be as unobtrusive as possible so that it can be practically used.

This research uses an automated tool to collect software quality metrics on arbitrary source code. These measurements can direct where effort should be expended to make the code less complex and where additional testing is required. The study is performed on large-scale, real-time software written in Mesa, a system development language.

Chapter 2 describes in detail the metrics used in this study. This includes the manner in which they are calculated and what information can be determined from each metric. Chapter 3 describes the Mesa language and explains the metrics collection tool, the types of data collected, and the levels of detail at which the data was collected. Chapter 4 contains the analysis of the data from one internal release of the software. Chapter 5 describes the methodology for integrating maintainability into large-scale software development. Finally, Chapter 6 reports the conclusions of this study and discusses possible future work.

Chapter 2: Software Metrics

I. Introduction

Over the past several years, software metrics have been shown to be valuable indicators in the quantitative evaluation of software and software designs. Code which is complex is more likely to have logical errors. Once errors are found, the high complexity also makes the code more difficult to correct. Therefore complexity has a direct effect on maintenance. The metrics described here are designed to quantify the level of complexity of a piece of code.

For any measurement to be useful, it is important to understand exactly what that measurement means and how to calculate it correctly. This chapter is an overview of the metrics used in this study and a discussion of what kinds of information can be determined from each.

The metrics used in this study were chosen for specific reasons. Of the many software metrics that have been proposed, only those which are totally automatable were considered. The amount of data in this analysis immediately disqualifies any subjective evaluation of the code, or even any objective evaluation that requires individual attention or time-consuming processing.

The metrics discussed here are based on a static analysis of source code. No measurements are made pertaining to the run-time execution of the software. The metrics are discussed here at the procedure level, defined as a small collection of programming statements. Grouping procedures into higher levels of abstraction is discussed later.

Software metrics can be subdivided into three areas: code metrics, structure metrics, and hybrid metrics. Each class is now discussed and the metrics from each explained.

II. Code Metrics

Metrics which deal only with the amount of substance a given procedure contains are called code metrics. The underlying theory is simply that the more items requiring attention that exist in a procedure, the harder that procedure is to understand. A code metric measurement tends to identify the internal quality of a procedure.

Three types of code metrics are analyzed in this study. They are lines of code[CONS86], Halstead's Software Science indicators[HALM77], and McCabe's Cyclomatic Complexity [MCCT76].

Lines of Code

One popular metric is the measure of how many lines of code (LOC) exist in a given procedure. The intuition is that a procedure with twenty LOC, for example, is considerably less complex than one with fifty LOC. Most software engineers agree that the overriding consideration concerning the length of a procedure is that it perform a solitary function. This can usually be accomplished in less than fifty lines of source code [MYEG76].

The LOC metric becomes useful in the immediate identification of flagrant violators of certain complexity and modularity tolerances. When analyzing a software system whose average LOC per procedure is thirty (an established norm), a particular procedure measuring one hundred lines of code instantly warrants attention. That procedure should be extremely straight-forward at first glance, and even then most software engineers would dictate that it should be modularized further.

However, the LOC measurement is a simplistic view that immediately raises several concerns. First, it should be obvious that not all code statements are of equal complexity. The LOC metric ignores the type of statements being counted and only relies on a gross measure of volume. Procedure A with twenty straight-forward output statements cannot possibly be as difficult to understand as procedure B with twenty statements involving multiple nested loops, complicated boolean conditions and a high degree of computation. Yet the LOC measure indicates that these two procedures are equally difficult to comprehend.

Another problem with the LOC metric stems from the definition of a single line of code. Researchers have not agreed on how to count a line of code, though most agree that the measurement should not include blank lines or comments. They also debate whether variable, type, and constant declarations should be considered in the LOC count [CONS86].

Some researchers promote counting any source code line which contains a portion of source code. The trouble with this view is that programmer style becomes an issue. In high level languages such as Pascal or 'C', where end-of-line markers are not delimiters, a source code statement could be spread over ten lines in the file, or it might be kept on one line. The LOC measure would indicate a large difference in complexity from code that is identical, merely spaced differently.

Another method of counting lines of code is to use some token from the source language which acts as a line terminator or separator. In Pascal and 'C', the semicolon character serves this purpose. This way, no matter how spread out a single source code line is in the file, the counts are the same.

In general, if the research in question deals with a large enough data sample, it is sufficient to choose a method and be consistent with it's use. Programmer idiosyncrasies tend to offset one another and a valid measurement is obtained. In the Mesa analysis, the use of the semicolon character as a statement separator is similar to its use in Pascal, therefore a semicolon count is used to determine the LOC measurement. Given the complexity and versatility of Mesa declarations, they are also counted.

Halstead's Software Science

Another set of popular code metrics is defined by Halstead [HALM77]. His measurements are based on the counts of operators and operands within a body of code. The following four initial values are calculated for a procedure:

n1	=	**number of unique operators**
n2	=	**number of unique operands**
N1	=	**total number of operators**
N2	=	**total number of operands**

In making these measurements, everything in the source code is considered to be either an operator or an operand. Operands consist of any entity being operated upon, such as constants and variables. Operators consist of all other items in the procedure. These include arithmetic and boolean operators, control statements, and assignment structures. Distinct keywords which cooperate to perform one function are often logically grouped and considered to be one operator, such as the If-Then, While-Do, and Begin-End constructs. The constructs considered as Mesa operators are given in Appendix A.

Once these initial counts are taken, the following values are defined:

n	$=$	$n1 + n2$	**Vocabulary Size**
N	$=$	$N1 + N2$	**Length**

Since every entity is included in the counts as either an operator or operand, the sum of the unique elements is called the vocabulary size, and the sum of the total count of elements is considered to be the length, or total number of tokens, of that procedure.

Given these definitions, Halstead then defines program volume, program level, language level, and effort.

Program Volume

Program volume is a measure of size based on the length of the implementation and the size of the vocabulary. Given **n** distinct elements, the smallest binary representation of a particular element requires $\log_2(n)$ bits. Therefore the program volume can be expressed by the following formula:

$$V = N \times \log_2(n) \qquad \textbf{Program Volume}$$

The program volume can also be interpreted as the number of mental comparisons needed to generate the program code. This view assumes that a programmer selects an element by making a mental binary search through the program vocabulary. This interpretation is important in the derivation of the formula for effort.

A related measurement, the potential volume V^*, is defined as the size of the most succinct form in which a particular algorithm could ever be expressed. This value cannot be determined without prior existence of a language in which the necessary operations were implemented. The concept of potential volume is needed for the next measurement of interest, program level.

Program Level

The level of the code in which an algorithm is implemented is defined as the potential volume divided by the measured program volume:

$$L = V^* / V$$

Since V^* cannot be directly determined, an approximation for L is needed.

The minimum number of operators is two, consisting of a function designator and an assignment. There is no upper limit on the number of operators. Therefore, with respect to operators, program level may be approximated:

$$L \sim 2 / n1$$

Operands, however, have no lower bound. Intuitively, as operands are repeated, the implementation can be considered at a lower level. Then with respect to operands, the program level equation is:

$$L \sim n2 / N2$$

Halstead combines these ratios to yield an estimator for program level:

$$L = (2 / n1) \times (n2 / N2) \qquad \textbf{Program Level}$$

Program level is considered inversely proportional to program difficulty. The lower the level, the more difficult it is to implement the algorithm.

Language Level

Given any algorithm translated from one language to another, as volume increases, the program level decreases proportionately. Therefore, the potential volume, V^*, is the product of the program level and volume, which is seen from a simple algebraic manipulation of the first definition of program level:

$$V^* = L \times V \qquad \textbf{Potential Volume}$$

However, when the language is held constant and the algorithm changes, it is the potential volume which varies proportionately with the program level. Therefore, the language level is defined as:

$$\textbf{lambda} = L \times V^* \qquad \textbf{Language Level}$$

Substituting (L x V) for V* yields a computable value for language level:

$$\textbf{lambda} = L\,(L \times V) = L^2 \times V$$

The language level computation is a quantitative evaluation for comparing two different languages, which has previously been a subjective process.

Effort

The final Software Science metric analyzed is Halstead's effort, which attempts to quantify the effort required to generate the implemented code. Since the volume of the program dictates the number of mental comparisons needed to implement the program, and the program level is the reciprocal of the difficulty, the effort required can be expressed as:

$$E = V / L \qquad \textbf{Effort}$$

The effort calculation is the fundamental metric considered when using Halstead's Software Science for complexity analysis.

McCabe's Cyclomatic Complexity

A third code metric used in this study is defined in [MCCT76]. McCabe's cyclomatic complexity is defined as a count of the independent logical paths through a procedure. Intuitively, the more logical branches that a procedure contains, the more difficult it becomes to trace all possibilities of actual execution.

Based in graph theory, the complexity measurement is taken from a representation of a procedure as a strongly connected graph. Each node in the graph represents a sequential block of code, and each arc represents a logical branching point through the procedure. The graph must have unique entry and exit points.

Graph theory states that, given a strongly connected graph G with one component, the maximum number of linearly independent circuits V(G), also known as the cyclomatic number, is calculated:

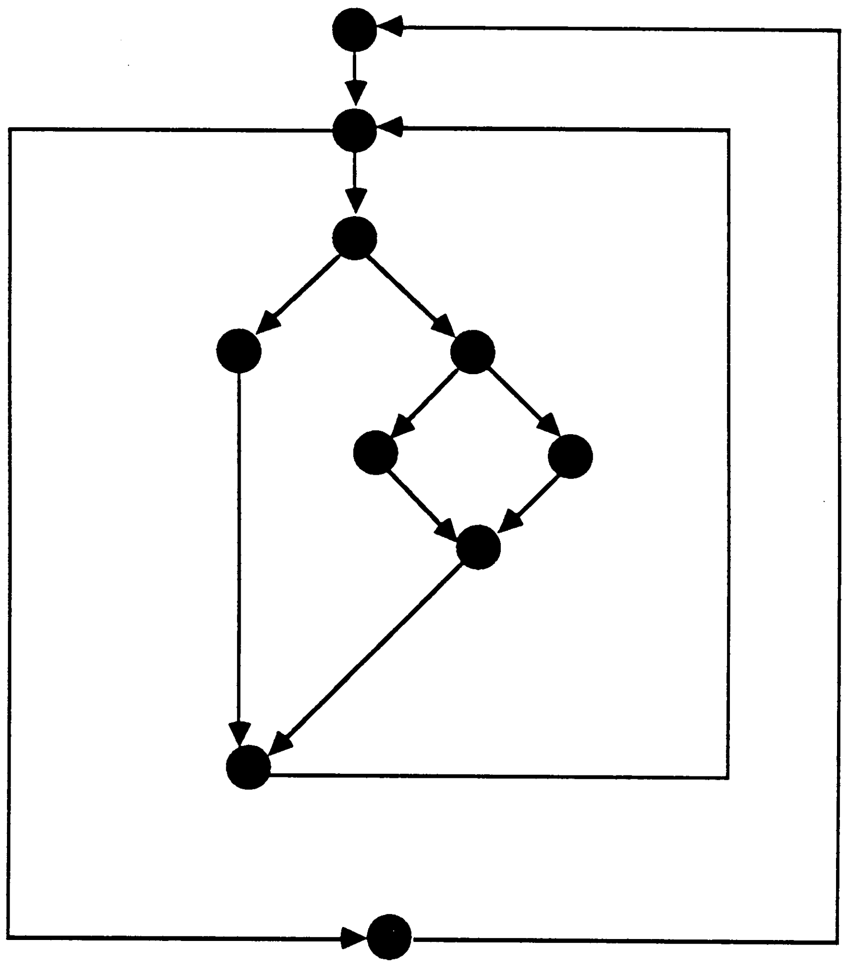
$V(G)$	=	$E - N + 2$	where
E	=	Number of edges in the graph	
N	=	Number of nodes in the graph	

Therefore, McCabe defines cyclomatic complexity to be the cyclomatic number of a procedure's strongly connected control graph.

An alternative method of calculating cyclomatic complexity is used if the analyzed code is given in structured form. In this case the cyclomatic complexity is equal to a count of the number of decision points in a procedure plus one. In the graph representation, a decision point is any node which has more than one directed arc from it.

This alternative method allows for efficient calculation of the metric based on the source language constructs. Appendix B contains a list of the Mesa constructs used as logical branch indicators for McCabe's Cyclomatic Complexity calculation.

Figure 3 shows a strongly connected program control graph for structured code. An imaginary arc is added from the exit point back to the starting point so that the graph is strongly connected. This does not change the concept or function of the graph, and is not included in the calculation. Using either method described above to calculate the complexity gives the result V(G) = 4.



$$V(G) = E - N + 2 = 12 - 10 + 2 = 4$$

$$V(G) = DP + 1 = 3 + 1 = 4$$

Figure 3. McCabe's Cyclomatic Complexity calculation.

III. Structure Metrics

Another class of metrics, structure metrics, are all based in some way on the overall structure of the system being examined. These quantitative evaluations reflect the bigger picture of the programming system structure. The structure metrics examined are Henry and Kafura's Information Flow metric and Belady's Cluster metric [HENS81a] [BELL81].

Henry and Kafura's Information Flow

The structure metric developed by Henry and Kafura is based on the amount of information which flows in and out of a procedure. Formally, Henry and Kafura define two quantities, fan-in and fan-out:

fan-in: The number of local flows into a procedure plus the number of global data structures from which a procedure retrieves information.

fan-out: The number of local flows from a procedure plus the number of global data structures which the procedure updates.

To calculate these values, structure data is generated representing the flow of information within a routine from input and output parameters, global data structures, and return values from functions. Using these relations, a flow graph is constructed representing all possible paths through a program which result in updates of global data structures.

The complexity of procedure **p** is now defined as follows:

$$C_p = (\text{fan-in} \times \text{fan-out})^2 \qquad \textbf{Information Flow}$$

The product of fan-in and fan-out is squared because the complexity between system components is non-linear.

Belady's Cluster Metric

Belady discusses the complexity of a software system in relation to how it can be subdivided into logical sections called clusters based on the communication that exists among the individual components [BELL81]. He hypothesizes that understanding interconnected elements is more difficult if their number is large. Furthermore, given the elements, complexity is proportional to the number of connections.

The cluster metric calculation is based on a graphical representation of the system, where nodes of an undirected graph represent individual elements (procedures, files, etc.) of the system, and edges represent a communication line between the elements.

The complexity of an individual cluster of elements j is given as:

$$Cj = Nj \times Ej \qquad \textbf{Cluster Metric}$$

where Nj is the number of nodes within a cluster and Ej is the number of edges between those nodes.

Belady goes on to define the complexity of the entire system as the sum of the individual cluster complexities plus the quantity representing the intercluster communication complexity. That calculation is:

$$C = \sum Cj + (N \times E_0)$$

where N is the total number of nodes in the entire system, and E_0 is the number of intercluster edges.

Belady's calculation of software complexity is performed at higher levels of data abstraction than other metrics. This higher level makes it a particularly attractive measurement when dealing with large-scale systems.

IV. Hybrid Metrics

Code metrics are based on the volume of material that make up a given procedure. Structure metrics take into account the procedure as an entity in the entire system. The concept of a hybrid metric combines both types of evaluations. Any arithmetic combination of code and structure metrics is considered a hybrid metric, although many such combinations do not make intuitive sense.

Many hybrid metrics were considered in this study, including various code metrics weighting the Henry and Kafura Information Flow. Woodfield's Review metric is defined as a hybrid and is also considered [WOOS80]. The Information Flow structure metric was originally defined as a hybrid metric with LOC weighting the Information Flow value. Further research determined the Information Flow value was a worthwhile measure by itself and is presented here accordingly [HENS81a]. The following section describes Information Flow weighted by lines of code and Woodfield's Review Complexity.

LOC and Information Flow

Using the LOC count to weight Henry and Kafura's Information Flow metric yields the following definition of the complexity of an arbitrary procedure **p**:

$$Cp = LOC \times (\text{fan-in} \times \text{fan-out})^2$$

LOC, fan-in and fan-out take on their original meanings as defined above. Any code metric could replace the lines of code count to yield another hybrid version of the Information Flow metric.

Woodfield's Review Complexity

Woodfield's Review Complexity metric is based on the number of times a component of a system needs to be reviewed to completely understand the system [WOOS80]. Two types of connections between procedures are defined. A control connection exists between a procedure and the procedure which invoked it. A data connection exists between two procedures **p** and **q** if there is some variable V such that the following conditions hold:

1. The variable V is modified in **p** and referenced in **q**.

2. There exists at least one data path set D(**pq**), between **p** and **q**, such that the same variable V is not referenced in any d_i where d_i is a member of D(**pq**).

A data path set D(**pq**) is defined as an ordered set of one or more procedures $\{d_1, d_2, d_3, ..., d_n\}$ such that one of the following conditions hold:

1. **q** calls d_1; d_1 calls d_2; ... ; d_{n-1} calls d_n; d_n calls **p**.

2. **p** calls d_1; d_1 calls d_2; ... ; d_{n-1} calls d_n; d_n calls **q**.

3. There exists some d_i in D(**pq**) such that d_i calls both d_{i-1} and d_{i+1}; d_{i-1} calls d_{i-2}; d_{i-1} calls d_{i-3}; ... ; d_2 calls d_1; d_1 calls **q**; and also d_{i+1} calls d_{i+2}; d_{i+2} calls d_{i+3}; ... ; d_{n-1} calls d_n; d_n calls **p**.

Woodfield then defines fan_in of a procedure as the count of all control and data connections for that procedure. The complexity of procedure **r** is given by:

$$C_r = \text{Effort} \times \sum RC^{k-1} \quad \text{for } k = 2 \text{ to } (\text{fan\_in}_i - 1)$$

Effort is the code metric defined in Halstead's Software Science. RC is a review constant, given as two-thirds in Woodfield's model. This constant was also suggested by Halstead [HALM77].

V. Conclusion

The metrics described in this chapter form the fundamental framework around which this study is based. Correct calculation of the metrics provides a quantitative method for evaluating software to determine which procedures are the most complex. Once identified, these procedures can be rewritten to reduce their complexity or at least tested further to raise the level of confidence in the code.

Chapter 3: Data Collection

I. Introduction

This chapter describes the experiment conducted to test the metric usage. Source code implemented in a system development language, Mesa, is analyzed using software metrics to verify that error-prone software can be identified. Following that, specific techniques are defined and are combined into a complete methodology for integrating maintainability into large-scale software.

The source code analyzed is part of a large-scale software project designed and implemented at a major software developer. Over 6000 procedures from a given internal release of the project code is analyzed. The software from the project is used to control the functions of a stand-alone machine. The system has characteristics of real-time software, in that it must respond almost immediately to user requirements and error conditions. Since it is a stand-alone system, it therefore contains it's own operating system used to control peripherals and environment. Finally, the system is highly computative and performance intensive.

For a complete understanding of what the data analysis indicates, it is important to realize the nature of the Mesa language, the various levels at which the data was obtained, and the tool which calculates the metric values.

II. Mesa

Mesa is a strongly-typed language with full system capabilities. As such, it is immensely detailed and functionally general. It supports full concurrency and contains, as part of the language itself, operating system synchronization capabilities such as monitors and semaphores. It also provides full signal processing capabilities. Many programmers familiar with Mesa compare its inherent complexity and functionality with that of the Ada programming language.

Mesa is fully modularized from the procedure level on up. Using Mes terminology, multiple procedures make up a file, and multiple files make up a module. Note that the use of the term module here is synonymous with a sub-system. Further discussion of the breakdown of the various levels of Mesa source code is presented in the following section.

The inherent complexity of a programming language is dependent on the person doing the assessment. If a programmer is very familiar with a language, he might subjectively calculate a level of complexity for a particular piece of source code much lower than someone who is not as familiar with particular constructs. However, Mesa has several functionally general constructs worthy of note.

For example, the order of parameter usage in procedure calls is completely arbitrary. By specifying the formal parameter names with the argument, the parameters may be presented to the procedure in any order. If the formal parameters are not specified, the order is assumed to be that of the procedure definition. The formal parameter names are separated from the arguments by a colon in the procedure call.

Similarly, in a procedure definition itself, default values may be presented to any particular parameters. A left arrow following the parameter designates a default value, which could be a constant or a complete expression. Any parameters with defaults can be omitted from a procedure call. If the arguments are presented, they take the value of the argument; if not, they take the value of the default. Procedures themselves are data types which can be passed as objects. An example of parameter ordering and defaults is given in Figure 4.

Also in Mesa, the programmer is completely responsible for memory allocation of all data space required, except for the base data types. Space is allocated from and returned to system or user-defined "heaps." Descriptors (masks) and user-defined sequences can be used to provide dynamic array data structures.

Signals and error data types, also inherent in the language definition, are used in a variety of ways in Mesa. Signals must be caught and properly handled using ENABLE statements or "catch phrases" in argument lists.

Mesa is a language fully capable of real-time, concurrent system development. Users of Mesa agree that it's inherent complexity exceeds the complexity of most programming languages existing today. This type of detailed language provides interesting challenges to metric analysis.

III. Levels of Data

A large amount of data was collected for this research, including the code metrics, the data from which the structure and hybrid metrics were calculated, and the error data. These various types of data were collected at different levels of granularity.

The development environment refers to three levels of abstraction in their developing systems. The lowest is the *procedure* level, as defined by a procedure of source code in the Mesa language. The second level of abstraction is a *file*, consisting of several procedures. Multiple files combine

```
DECLARATION:


SomeProc:  PROCEDURE [ i:  INTEGER, j: CARDINAL <- 1, k:  BOOLEAN ]
                  RETURNS [ x, y:  INTEGER ] =
      BEGIN
       ●
       ●
       ●
      END;


USE:


 [ q, p ]  <-  SomeProc [ k:  FALSE, i: 23 ];
```

Figure 4. An example of parameter ordering and defaults in Mesa.

to form the core of the third level of granularity, the *module*, which is a complete sub-system with a high degree of functionality.

Initially, it was hoped that all of the data could be obtained at the procedure level. That is, for each procedure analyzed, calculate all the metrics and determine the number of errors resulting from a logical mistake existing in that procedure. For various reasons this was not possible.

The code metrics are generated at the procedure level. Due to restraints on time and facilities, the structure data was only available at the file level. That is, data which defines the communications among components of the code was only obtained among individual files, and the structure and hybrid metrics were calculated on that level. Further discussion of how the structure data was collected and used is presented in the next section.

To obtain the structure metrics at higher levels, simple linear sums were used. The only problem this caused was with the Information Flow metric, which normally has values in the hundreds of thousands at the procedure level. Therefore, summing the values created overflow problems. This is solved by taking the log base 10 of the Information Flow metric values to obtain manageable numbers [CANJ85].

The error database was consistently kept at the module level, indicating only that an error occurred in a particular module. An error is defined to be a deviation from the expected output of a particular test case, or an incidental observation of incorrect action on the part of the operating code. The code analyzed was taken from a specific internal release, therefore the errors described are true defects and not indications of the need for additional functional enhancement.

Other pieces of information are stored with the error database entry, such as text descriptions of the problem, the person responsible, etc., but nothing else of interest to this study. Often individual files were noted as the possible source of the error, but this did not occur with enough consistency to be statistically relevant.

Having the data at various levels does not impede the formulation of a methodology in which metrics are used, but it is important to understand at what level each result or discussion is based.

IV. The Metrics Collection Tool

As stated before, no metric was considered for use in this research which was not automatable. When dealing with large-scale software projects, it is infeasible to address any aspect of the code manually. Therefore, a tool to perform the analysis on the code and calculate the metrics value was constructed.

The analyzer is actually sectioned into two pieces, one to generate the code metrics and a statistical report, and the second to generate the data from which the structure metrics are calculated. The metric analyzer is shown in Figure 5.

The code metric analyzer statically evaluates each component of syntactically correct source code and generates the metrics for each procedure. At any point, a statistical report can be generated which gives a breakdown by the LOC, Effort, and Cyclomatic Complexity code metrics, calculating means, standard deviations, minimums and maximums. Also, based on a parameter to the tool, the report identifies the procedures which have the highest metric values considering the top N percent of the total source

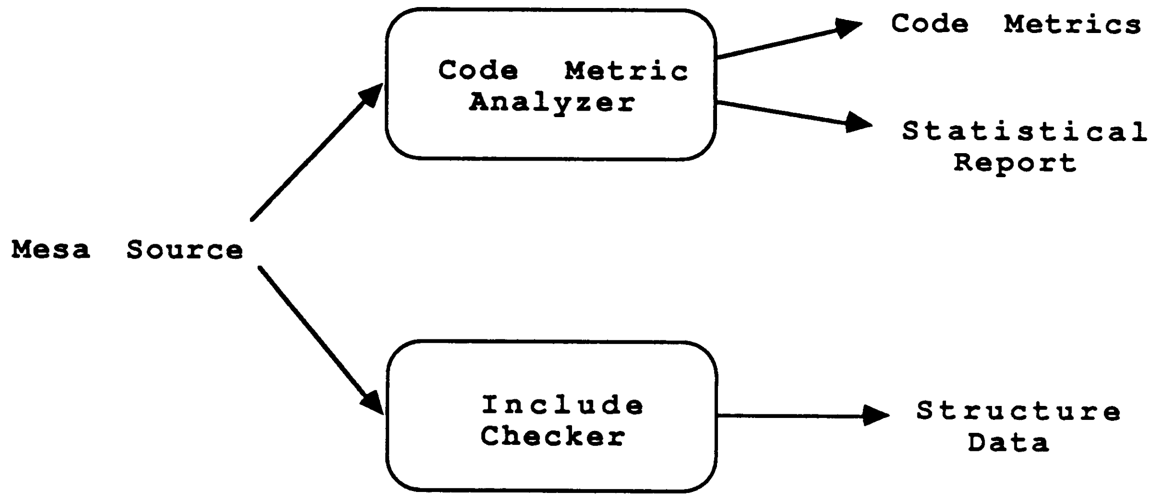


Figure 5. The Metric Analyzer Tool.

analyzed. Furthermore, from that highest N percent, the report indicates which procedures overlapped all three code metrics, or any combination of two.

This overlap report provides an immediate indicator of where potentially serious problems exist. Using threshold values with varying degrees of seriousness, the highest violators of metric tolerances can be identified and investigated. A detailed discussion of a flag / alarm system which formalizes this concept is presented in Chapter 4.

An example of the statistical report run on the code for the code metric analyzer itself is given in Appendix C. The User's Guide and Implementation Guide for the code metric analyzer is given in Appendices D and E.

The data used to calculate the structure and hybrid metrics is obtained from the output of an existing tool in the development environment called the Include Checker. This tool is used to inform a system developer of the proper order in which to rebuild the components of a system, based on the use of specific files by other files. The important aspect of the output for this research is that it identifies communication lines between files of a module (subsystem).

Basically, the Include Checker associates all files which communicate with a given file. The structure data took the form of simple lists: each file has associated with it a list of other files that it "uses" in some way. The use could be called procedures, shared data references, declarations, or a combination of these.

The Cluster structure metric fits well into this scheme. In fact, its definition lends itself to analysis at this level. However, because of the level of data available, it is actually a modified Information Flow metric that is used in this research.

The fan_in and fan_out values of the modified information flow metric are the same as those defined originally except for one detail. The deviation is that the calculation of fan_in and fan_out is an overkill count based on total communication lines, since the structure data does not distinguish between flows into and out of a file. The original definition of the metric is also an overkill of communication lines in that all procedures which reference a global data structure communicate with each other. The definition used in this study also assumes that all communication is two-way. This deviation, while not strictly as the original definition specified, still yields important aspects of the source code in terms of complexity.

V. Conclusion

This research is based on the development of a large system implemented in Mesa. To generalize analysis techniques and to make accurate conclusions from that analysis, a full understanding is needed of the type of language analyzed and of the levels of abstraction for the data collection.

In Chapter 4 the complete data analysis used to validate the study and derive further analysis tools is described.

Chapter 4: Data Analysis

I. Introduction

With the tool in place and the data gathered, analysis was performed to verify that the metrics were performing in the manner intended. Specifically, it is important to access how well the metrics indicated the error-prone software and establish a scale of "acceptable" metric values. Also, using regression analysis, predictor equations are developed to assist in the usefulness of the tool. Each of these areas of analysis and the results are described below.

II. Intermetric and Error Correlations

Correlations among the various metrics and between the metrics and the error data are essential to gain an understanding of how useful the metrics are at indicating danger areas and determining which metrics focus on similar attributes.

The correlation matrix for the code metrics is displayed in Figure 6. LOC refers to Lines of Code, N, V and EFF are Halstead's length, volume and effort, CC is McCabe's Cyclomatic Complexity, and ERR represents the number of errors.

Because of the nature of the error data, the correlations are performed at the module level. Note that all of the code metrics correlated very high (values ranging from 0.852 to 0.998) with each other. This verifies previous study's results and indicates that the code metrics all tend to measure similar aspects of the code, namely the internal complexity [HENS81b] [CANJ85] [HENS88b].

	LOC	N	V	EFF	CC	ERR
LOC	1.00					
N	0.987	1.00				
V	0.982	0.998	1.00			
EFF	0.852	0.884	0.908	1.00		
CC	0.989	0.998	0.993	0.861	1.00	
ERR	0.837	0.831	0.843	0.828	0.814	1.00

Figure 6. Intermetric and Error Correlations for Code Metrics.

The correlations between code metrics and defect occurrences (errors) is also significantly high (0.814 through 0.843). This indicates that the code metrics do successfully establish where the errors exist in the source code. The fact that no one metric shows a substantially higher correlation to errors than the others indicates that a single given metric cannot be used as "the" metric for the Mesa language or the development environment.

Figure 7 shows the correlations among two code metrics (LOC and V) and the structure and hybrid metrics. WOOD represents Woodfield's Review Complexity, CLUS is Belady's Cluster metric, INFO represents Henry and Kafura's Information Flow metric, and ERR again represents the number of errors.

Error data is once again correlated to all the metrics. Note that the information flow metric does not correlate well with the code metrics. Again, this verifies past studies and indicates that the information flow structure metric and code metrics are examining different aspects of software complexity.

The Woodfield hybrid metric correlated well with the code metrics. This is due to the fact that the hybrid metric is, by definition, affected greatly by Halstead's effort code metric. Belady's cluster metric also correlated fairly high to both the code metrics and the other structure metrics. Since this is not technically a hybrid metric and not inherently affected by code metrics, this would indicate that the cluster metric addresses aspects of both types of complexity.

The correlations to errors are considerably higher for the structure and hybrid metrics than for the code metrics. This indicates that the structure data yields a higher understanding of error-prone complexity in this environment.

	LOC	V	WOOD	CLUS	INFO	ERR
LOC	1.00					
V	0.982	1.00				
WOOD	0.925	0.901	1.00			
CLUS	0.925	0.891	0.999	1.00		
INFO	0.775	0.775	0.930	0.925	1.00	
ERR	0.837	0.843	0.936	0.919	0.884	1.00

Figure 7. Intermetric and Error Correlations including Structure and Hybrid Metrics.

III. Threshold Determination

Any use of software metrics must be accompanied by an understanding of what the metrics are attempting to indicate about complexity and therefore what appropriate values for the metrics should be. Since this study is dealing with large-scale software, individual tolerances at the procedure level can only be intuitively estimated. For higher levels, these values can easily be scaled depending on the situation.

Tolerances for Lines of Code, Halstead's Effort, and McCabe's Cyclomatic Complexity are given in Figure 8. Two levels of concern are defined, one at which you merely raise a flag and the second at which an alarm is sounded. Combinations of flags and alarms across the metrics indicate what action should be taken.

Taken individually, a flag indicates that the additional complexity should be verified to be manageable and/or justified. However, there can be little justification for a procedure whose metric's values sound an alarm.

The previous paragraphs require qualification. If a procedure raises a single flag and no alarms, this is certainly not where effort should be concentrated. Even a single alarm can be deceptive. For example, a straightforward output procedure with many display lines would make the Lines of Code metric sound an alarm, but the Effort and Cyclomatic Complexity would indicate that there is no problem. Conversely, a procedure which processes a huge Select (Case) statement would sound a Cyclomatic Complexity alarm, but not raise concern in the others.

Therefore, the metrics values must be considered in conjunction. Rules of thumb might include investigating a particular procedure individually when it raises three flags, two flags and an alarm, or two or more alarms.

	"Safe" Zone	Flag	Alarm
LOC	Under 50	50 - 100	Over 100
Effort	Under 50000	50000 - 100000	Over 100000
CC	Under 10	11 - 20	Over 20

Figure 8. Tolerances for code metrics at the procedure level.

Figures 9, 10, and 11 show the percentage of procedures from 10 modules which fall into various distributions according to Lines of Code, Halstead's Effort, and McCabe's Cyclomatic Complexity, respectively.

Note that the metric values which are not large enough to raise a flag are classified as "safe." This does not mean, of course, that a procedure with only 20 lines of code will not contain an error. It does mean, however, that the procedure is less likely to contain an error, and even if it does contain an error, the low level of complexity allows the error to be found quickly and be addressed with less difficulty.

IV. Predictor Equations

Analysis was performed to give the user of the metric tool an additional indicator of software maintainability. Regression analysis was performed with the number of errors as the dependent variable and various metric values as the independent variables. This yields equations which predict the number of errors that can be expected in a module of source code.

The "best" equations are established by evaluating their R-squared values and by examining their relative rankings using the MSE and C(p) statistics. The R-squared value of a given regression equation is an estimate of the percentage of data that is explained by that equation.

The MSE statistic is calculated by summing the squares of the difference between the actual and the predicted values. This is performed for all observations and the mean calculated. The lower the MSE value, the better the predictor equation. MSE measures the "noise" in the model, i.e. determines the unexplained variability in the model.

Module	1-30	31-50	51-100	101-200	> 200
1	97	2	1	0	0
2	91	6	2	1	0
3	94	4	2	0	0
4	90	5	4	1	0
5	94	5	1	0	0
6	93	4	2	0	1
7	85	9	4	1	1
8	94	5	1	0	0
9	94	2	2	1	1
10	93	3	2	1	1

● FLAG ● ALARM

Figure 9. Percentage Distribution of Lines of Code.

Module	10000	50000	100000	200000	> 200000
1	83	13	2	1	1
2	62	28	6	2	2
3	74	17	4	2	3
4	57	27	7	5	4
5	57	29	9	3	2
6	69	20	5	3	3
7	50	25	8	7	10
8	62	27	5	3	3
9	74	21	3	1	1
10	66	19	8	4	3

● FLAG ● ALARM

Figure 10. Percentage Distribution of Halstead's Effort.

Module	1 - 2	3 - 10	11 - 20	21 - 50	> 50
1	74	21	4	1	0
2	50	37	6	6	1
3	52	37	7	3	1
4	47	42	8	3	0
5	48	43	5	4	0
6	52	36	8	3	1
7	37	43	12	6	2
8	47	42	8	2	1
9	65	21	8	5	1
10	58	30	7	3	2

● FLAG ● ALARM

Figure 11. Percentage Distribution of McCabe's Cyclomatic Complexity.

The C(p) measurement is a compromise statistic between bias and simplicity. If too many parameters are used in a given model, the C(p) value increases. Likewise, if too few independent variables are used, the bias towards those few is large and the C(p) value increases. The C(p) statistic is measured by summing the variance over the total number of observations. This sum should be equal to the number of variables in the model.

Not all metrics need to be considered to establish a valid predictor equation, and many combinations are examined. From these, five equations are recommended, based on the R-squared values, and the MSE and C(p) statistics. These equations are given in Figure 12. The R-squared values for these equations range from 0.89 to 0.99. The complete set of equations considered with their statistical relevance is given in Appendix F.

Figure 13 shows the predicted number of errors for 10 modules using the first regression equation from Figure 12. This equation uses the metric values from Woodfield's Review Complexity, Belady's Cluster metric, Henry and Kafura's Information Flow, Halstead's Volume, and Lines of Code. This equation was chosen because of its high R-squared (0.99) value.

Note that what the equation is actually predicting is the number of errors that will be found given the same amount and concentration of testing that was performed on the data which was used to generate the predictor equations.

The level of the data used to generate these predictor equations is important. Since error data was only available at the module level, the regression analysis had to be performed at that level. Therefore, the equations presented in Figure 12 are only valid at the module level.

$$\text{Err} = 2.899\text{E-}07\ \text{WOOD} - 8.188\text{E-}04\ \text{CLUS} - 1.868\text{E-}04\ \text{V} + 1.425\text{E-}02\ \text{LOC} + 2.990\ \text{INFO}$$

$$\text{Err} = 5.001\text{E-}05\ \text{CLUS} + 8.161\text{E-}05\ \text{V} - 4.294\text{E-}03\ \text{LOC}$$

$$\text{Err} = 8.827\text{E-}07\ \text{EFFORT} + 1.530\text{E-}03\ \text{N} + 9.505\text{E-}03\ \text{LOC} - 5.004\text{E-}02\ \text{CC} - 5.986\text{E-}05\ \text{V}$$

$$\text{Err} = 9.508\text{E-}03\ \text{LOC} + 6.872\text{E-}07\ \text{EFFORT} - 4.907\text{E-}02\ \text{CC} + 1.188\text{E-}03\ \text{N}$$

$$\text{Err} = 1.706\text{E-}04\ \text{V} - 4.679\text{E-}02\ \text{CC} + 9.360\text{E-}03\ \text{LOC} + 2.662\text{E-}04\ \text{N}$$

Figure 12. Regression equations to predict the number of errors per module.

Actual Errors	Predicted Errors
34	29.181
31	31.574
147	147.000
27	25.496
53	54.005
0	-0.850
12	17.714

Figure 13. Actual vs Predicted errors using the first predictor equation.

The fact that these predictor equations are valid only at the module level restricts their use, but the technique to generate them is completely general. Therefore, additional regression equations can be easily calculated for lower (file and procedure) levels if the error data is collected at those levels.

V. Conclusion

The analysis of the internal release shows that the metrics are identifying error-prone software and that further refinement and concentrated testing can reduce the maintenance tasks required. The threshold guidelines can be used to determine violators of metrics tolerances for further analysis. The use of predictor equations can aid in the process of test case creation and takes into account the need for a view of complexity from various angles.

Now that the analysis has validated use of the metrics in the Mesa language and environment, the methodology can be formalized with unobtrusive, practical techniques which integrate maintainability into large-scale system software. The details of this methodology are presented in Chapter 5.

Chapter 5: The Methodology

I. Introduction

Tools are created, metrics established, data validated, and techniques defined. How then does all this combine into a unified whole? A major goal of this study is to define a complete methodology for integrating maintainability into large-scale software.

A methodology can be defined as a set of procedures and tools which, when used correctly, will formulate a process designed to meet specific objectives. The process in this research is large-scale software development and the objective is to develop that software so that it is maintainable.

An interesting concern came quickly to light as this research began. That was the realization that established companies which design and implement large-scale software products are reluctant to make major changes in their software development process. Often times, immediate deadlines and budget restrictions obscure long-term advantages, even when the techniques presented are proven to enhance productivity.

Maintenance is an expensive portion of the software life cycle, but it is still viewed by many software developers to be a post-production problem. Managment would certainly prefer that any effort to increase the quality of their software do so with little disruption to the current development process. Therefore, another important objective of this methodology is that it be as unobtrusive as possible.

To define such a methodology, it is helpful to examine a fundamental software development technique called iterative enhancement and how it relates to current industry practices.

II. Iterative Enhancement

The concept of iterative enhancement, as defined by Basili, describes a development scheme which inherently supports complexity analysis at the implementation stage [BASV75]. Iterative enhancement is a practical approach to the creation of new software systems. In it, a bare skeleton of the system is fully implemented, then that skeleton is augmented by a particular feature. After that feature is fully tested, the next feature is added, and so on.

Each iteration consists of designing the implementation of a selected task, coding and debugging that task, and analyzing the existing partial implementation at that stage. The analysis also defines what tasks should be added to the project list, in addition to the features of the final project which have yet to be implemented. Note that the idea of iterative enhancement as used here is different from other techniques which implement an almost complete system, then iteratively refine and reorganize until an acceptable design and implementation is achieved.

This technique is rarely used in its purest form. However, a common approach in industry borders on this concept. In the process of creating a new software system, many companies set deadlines for specific internal releases of the software. They also predict a final launch date for the first shipment to the customer. These deadlines might be manipulated as unforeseen problems arise, or the release dates can be held firm and justifications given for any feature not appearing in a particular release that was originally scheduled.

Each internal release is designed with particular goals in mind, which allows software management to assess the progress of the development. Between releases, individual programmers test their code, and when a collection of related modules are ready, integration tests are performed on larger sections of the code. The designated release points allow complete system tests to be performed.

The fundamental differences between Basili's iterative enhancement concept and industry practices is that in large-scale development, multiple features are developed at each stage, and error correction goes hand-in-hand with new development. However, these release points roughly correspond to the full system analysis stage in the iterative enhancement technique. Therefore, they also provide opportunities to measure the complexity of the system at intermediate points in its development, and determine where complexity problems are being created.

Figure 14 illustrates the integrated concepts of internal releases and complexity analysis in the waterfall model. The risk analysis stages in Boehm's spiral model can also be viewed as appropriate points for metric analysis integration.

III. Granularity

In large-scale system development, the use of the modified iterative enhancement concept can also be brought down to lower levels in the development structure. Unit managers define similar task lists for the work necessary for their section of the system. Before a system release point occurs, the smaller subsystems are frozen so that unit testing can be performed to establish confidence in the code which falls under these individual umbrellas.

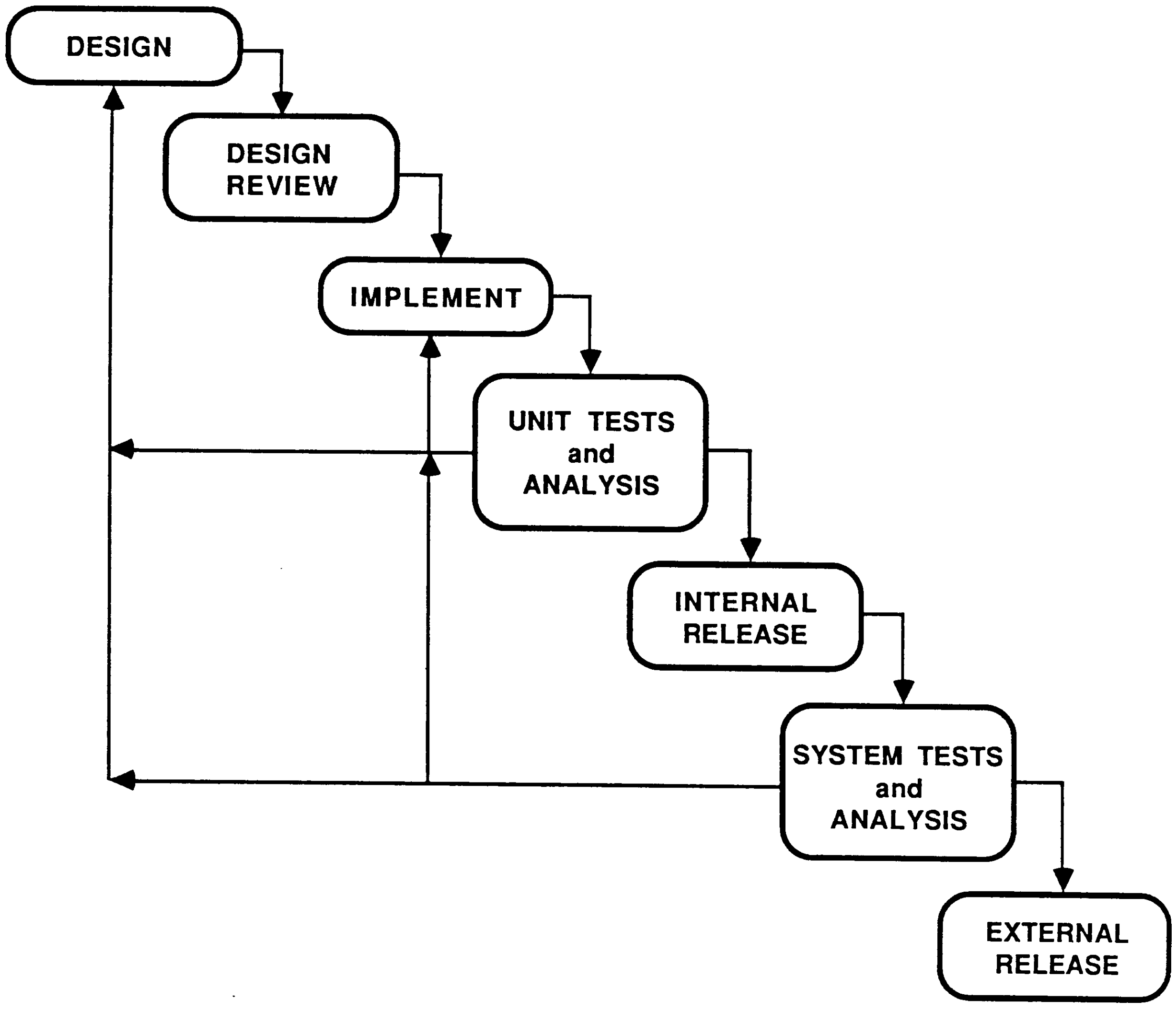


Figure 14. Large-scale development process with release points and metric analysis.

Each of these points are appropriate places for metric analysis to occur. Errors found at this level are more easily fixed since the scope of where the problem originates is smaller and better understood. Metric evaluation can indicate where that important testing should be concentrated and where error-prone code exists.

Even individual programmers can work on their code with this technique. Once a programmer determines that the code he is responsible for is fully implemented, metric evaluation can pinpoint exact procedures which will probably, if not immediately, cause problems. This identified code can be rewritten or at the very least thoroughly tested. The individual tests run by programmers can uncover many problems that might not show up until further in the testing hierarchy under regular circumstances. Programmers are highly motivated to find any problems in their code before it advances to higher levels of testing and someone else informs them they have a problem.

Complexity analysis can therefore be integrated into extremely early points of the development and testing process. And as stated in Chapter 1, the earlier an error is uncovered and fixed, the less costly the change.

IV. The Process

Now we can observe the software development process with the metrics evaluation fully integrated into all stages.

Once the functionality of an individual internal release point is defined, the system designed, and the functionality decomposed to unit and programmer levels, the implementation process begins. The methodology proposed here is based at the implementation and testing levels since this causes the least disruption to the existing development process.

Individual programmers begin implementing their relatively small portion of the logic for the overall system. Once they establish a functional subset of their code, they use the metric analysis tool to generate metric values and a statistical evaluation of their code based on the metrics. This gives immediate feedback as to the volatile portions of their code. Threshold values can be used to section off those procedures which raise flags and alarms as the situation dictates. If predictor equations are available at that level, these can indicate the number of errors that can be expected as well.

Based on their analysis, portions of their code can be rewritten to reduce the complexity levels. This rewriting process itself is highly likely to uncover logical errors in previously complex procedures. If rewriting is not considered necessary or feasible, the programmer's testing efforts can at least be concentrated on the potentially dangerous code.

Once the programmers have established their portions of the system, pieces are brought together for unit tests. At this stage, the analysis can be run on the larger subset of the overall system. Again, the metric analysis can identify areas that are potentially dangerous, this time from a slightly more global viewpoint. Structure metrics tend to become more important at this stage since there is more cohesive communication among the larger components of the code. Using predictor equations at this level can be very informative.

After unit testing, a full internal release is established. Metric analysis at this stage can give an even more global view of the developing system. Errors concerning integration problems and higher level communication are likely to be uncovered here.

Once the system internal release is tested, new tasks to fix errors and increase the growing functionality of the developing system are defined.

Progress toward the next internal release is begun and the cycle repeats itself. Eventually, the last internal release is performed with little or no problems detected, and the product is scheduled for an external release.

V. Integration

This methodology was presented to a commercial software development environment with encouraging results. Plans were made to distribute the metrics tool to various levels of the developing system process, including the individual programmers.

An important caution must be made concerning the use of the metric tool. Any evaluatory technique such as the one described in this research concludes with the result that something is good or bad. It must be made clear to the software writers that the tool is used only as a way to evaluate code and not to judge the productivity or usefulness of individuals. If software developers fear the tool, the entire concept of metric evaluation will be undermined.

If the methodology described in this chapter is used as intended, to help guide testing processes and identify error-prone software, the end result will be a software product that is more reliable and maintainable.

VI. Conclusion

A succint review of the methodology described in this chapter will give a encapsulated understanding of the processes and tools involved. Large-scale software is developed in stages, called release points, used to assess progress and perform system tests. At these points, metric analysis is performed on the implemented code, using specific techniques such as threshold determination and predictor equation analysis, to identify error-prone software due to high complexity. Once identified, the software can be rewritten to reduce the complexity, or at least the testing procedures are targeted toward that code. The same type of analysis is performed at the lower levels of the development process for each release, allowing the unit testers and even individual programmers the opportunity for metric analysis. Iterative progression through the release points will have a higher degree of confidence due to the metric analysis performed at each stage.

The methodology described in this chapter incorporates the various techniques and observations established by this research. Since it concentrates on integrating the use of the metric analysis without instigating another complete level to the development hierarchy, the use is practical as well as useful.

Chapter 6 describes the overall conclusions to this research and discusses future work in the area.

Chapter 6: Conclusions

I. Introduction

The results of this study are both encouraging and educational. The research as a whole was accepted well in the commercial environment it was targeted for, and definite plans are in the works for the fruits of this labor to be practically implemented.

Many conclusions gained from this research served to validate past research, while others shed important insight into the differences between real-world, large-scale software development and controlled, smaller studies.

The following section describes the particular conclusions which can be drawn from this research.

II. Conclusions of the Research

To review from Chapter 1, the goals of this research are: (1) Design and implement a metrics collection tool for the Mesa language; (2) Validate that the metrics calculated by the tool indicate error-prone software; (3) Establish regression equations which predict the number of errors which are present in the section of code; and (4) Develop a methodology using these tools and techniques which integrates maintainability into large-scale software.

The following sections address the weaknesses and strengths of the research, and specifically address the conclusions of the study.

Weaknesses

One area of this research which could be stronger is the concentration on Mesa as an individual language. There are probably several distinct constructs and abilities in Mesa which lend themselves higher code complexity. Not all of these are addressed by the metrics used in this study.

Another restriction in the usefulness of the predictor equations presented in this study is that they are only valid at the module level. This is a consequence of the data made available for the research. However, the techniques for generating the equations are general and the equations can easily be calculated for any level. Error database needs in industry is discussed further in the section on future work.

Strengths

Overall, this research can be considered a success. All major goals were achieved, and several other tangents were uncovered. Specifically, the following sum up the results of this research:

- Software quality metrics have again been shown to identify error-prone software due to a high level of complexity.

- Maintainability is a characteristic which can (and should) be integrated into software as it is produced, instead of dealing with it only as a post-production process.

- Techniques including the use of predictor equations and threshold analysis are developed to aid in the use of a metric analysis tool.

- A methodology is defined to integrate maintainability using software metrics as an evaluation scheme for guiding the testing processes already in place.

- Practical use dictates that such defined methodologies should disrupt as little of the existing development process as possible.

Overall the results are positive and motivating. The concentration on the maintenance aspect of the software development process must be continued in order to reduce the overwhelming cost of software production and service.

III. Future Work

The research presented here opens additional avenues for investigation. The methodology described uses techniques that can be modified and refined to provide even more insight into the complexity of large-scale software. These techniques, as well as new ones, can be used in the same overall environment to further the goals presented by this study.

This methodology should be integrated into another commercial environment to verify its validity and explore other techniques for identifying error-prone software. Specific emphasis should be given to the fact that it is intended to cause little disruption as possible to the existing software development system.

Important work for the future can be concentrated on the importance of industry-generated error databases, including the information they should contain and the levels at which it should be collected. These studies should consider programmer detail such as the number of lines changed to fix a bug, the severity of the error, specific location, sections of the code affected, the time needed to find and fix the defect, etc.

The need for these must be stressed from all points of view, including the fact that a complete error history, analyzed correctly, can lead to a better understanding of where and how errors occur at a global level. This information is necessary to make the production process more complete.

Bibliography

[BALR83] Balzer, R., Cheatham, T.E., Jr., Green, C., "Software Technology in the 1990's: Using a New Paradigm," IEEE Computer, Vol. 16, No. 11, November, 1983, pp. 39-45.

[BASV75] Basili, V.R., Turner, A.J., "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, 1975, pp. 390-396.

[BELL76] Belady, L.A., Lehman, M., "A model of large program development," IBM System's Journal, No. 3, 1976 pp. 225-252.

[BELL81] Belady, L.A., Evangelisti, C.J., "System Partitioning and Its Measure," Journal of Systems and Software, Vol. 2, 1981, pp. 23 - 39.

[BOEB76] Boehm, B.W., "Software Engineering," IEEE Transactions on Computers C-25, December, 1976, pp. 1226-1241.

[BOEB86] Boehm, B.W., "A Spiral Model of Software Development and Enhancement," ACM Software Engineering Notes 11, August, 1986, pp. 14-24.

[CANJ85] Canning, J., "The Application of Software Metrics to Large-Scale Systems," Ph.D. Dissertation, Department of Computer Science, Virginia Tech, April, 1985.

[CASP80] Cashman, P.M., Holt, A.W., "A Communication-Oriented Approach to Structuring the Software Maintenance Environment," ACM SIGSOFT, Software Engineering Notes, Vol. 5, No. 1, January, 1980, pp. 4-17.

[CONS86] Conte, S.D., Dunsmore, H.E., Shen, V.Y., *Software Engineering Metrics and Models*, The Benjamin/Cummings Publishing Company, Inc., 1986.

[HALD88] Hale, D.R., Haworth, D.A., "Software Maintenance: A Profile of Past Empirical Research," IEEE Conference on Software Maintenance, November 1988, pp. 236-240.

[HALM77] Halstead, M.H., *Elements of Software Science*, New York, Elsevier North-Holland, 1977.

[HENS81a] Henry, S.M., Kafura, D., "Software Structure Metrics Based on Information Flow," IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, September 1981, pp. 510-518.

[HENS81b] Henry, S.M., Kafura, D., Harris, K., "On the Relationships Among Three Software Metrics," Performance Evaluation Review, Vol. 10, No. 1, Spring 1981, pp. 81-88.

[HENS87] Henry, S.M., Goff, R., "Complexity Measurement of a Graphical Programming Language," Technical Report TR-87-35, Department of Computer Science, Virginia Tech, November, 1987.

[HENS88] Henry, S.M., Selig, C.L., "A Metric Tool for Predicting Source Code Quality from a PDL Design," Proceedings of the Workshop on Software Design Metrics, Melbourne, Florida, March, 1988.

[KAFD87] Kafura, D., Reddy, G.R., "The Use of Software Complexity Metrics in Software Maintenance," IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, March 1987, pp. 335-343.

[LAMD88] Lamb, D.A., *Software Engineering: Planning for Change*, New Jersey, Prentice Hall, 1988.

[LEAM84] Lehman, M.M., Stenning, V., Turski, W.M., "Another Look at Software Design Methodology," ACM Software Engineering Notes 9, April, 1984, pp. 38-53.

[LIEB78] Lientz, B.P., Swanson, E.B., Tompkins, G.E., "Characteristics of Application Software Maintenance," Communications of the ACM, June 1978, Vol. 21, No. 6, pp. 466-477.

[LINR88] Linger, R.C., "Software Maintenance as an Engineering Discipline," IEEE Conference on Software Maintenance, November 1988, pp. 292 - 297.

[LIPM79] Lipow, M., "Prediction of Software Failures," The Journal of Systems and Software, Vol. 1, 1979, pp. 71 - 75.

[MCCT76] McCabe, T.J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp. 308-320.

[MUNJ78] Munson, J.B., "Software Maintainability: A Practical Concern for Life-Cycle Costs," IEEE Conference on Computer Software and Applications, 1978, pp. 54-59.

[MYEG76] Myers, G.J., *Software Reliability, Principles and Practices*, New York, John Wiley & Sons, 1976.

[OBRV86] Overstreet, C.M., Nance, R.E., Balci, O., Barger, L.F., "Specification Languages: Understanding Their Role in Simulation Model Development," Technical Report TR-87-7, Virginia Tech, December, 1986.

[SELR88] Selby, R.W., Basili, V.R., "Error Localization During Software Maintenance: Generating Hierarchical System Descriptions from the Source Code Alone," IEEE Conference on Software Maintenance, November 1988, pp. 192 - 197.

[SWAE76] Swanson, E.B. "The Dimension of Maintenance," Proceedings of the 2nd International Conference on Software Engineering, October 1976, pp. 492-497.

[WAKS88] Wake, S., Henry, S., "A Model Based on Software Quality Factors which Predicts Maintainability," IEEE Conference on Software Maintenance, November 1988, pp. 382 - 387.

[WOOS80] Woodfield, S., *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph.D. Dissertation, Department of Computer Scinece, Purdue University, 1980.

[YAUS88] Yau, S.S., Chang, P.S., "A Metric of Modifiability for Software Maintenance," IEEE Conference on Software Maintenance, November, 1988. pp. 374 - 381.

Appendix A. Halstead Operators for Mesa

Listed on the following page are all constructs in the Mesa language that are considered operators in the calculation of Halstead's Software Science indicators. They include all reserved words and tokens which perform specific functions. Some reserved words or characters are combined as a single operator.

ABORTED	IMPORTS	PUBLIC	[ ]	
ABS	IN	READONLY	..	
ALL	INLINE	REAL	<-	(left arrow)
AND	INT	RECORD	|	
ANY	INTEGER	REJECT	( )	
APPLY	INTERNAL	RELATIVE	*	
ARRAY	ISTYPE	REPEAT	=>	
BASE	JOIN	RESIDENT	+	
BEGIN - END	LAST	RESTART	-	
BOOL	LENGTH	RESUME	/	
BOOLEAN	LOCKS	RETRY	~	
BROADCAST	LONG	RETURN	!	
CARDINAL	LOOP	RETURNS	@	
CHAR	LOOPHOLE	SELECT	#	
CHARACTER	MACHINE	SEQUENCE	<	
CODE	MAX	SHARES	<=	
COMPUTED	MDSZONE	SIGNAL	>	
CONDITION	MIN	SIZE	>=	
CONTINUE	MOD	START		
DECREASING	MONITORED	STATE		
DEFINITIONS	MONITOR	STOP		
DEPENDENT	MONITORLOCK	STRING		
DESCRIPTOR	NARROW	SUCC		
DIRECTORY	NAT	THROUGH		
DO	NATURAL	TRASH		
ENABLE	NEW	TRUE		
ENDCASE	NIL	TYPE		
ENDLOOP	NOT	UNCOUNTED		
ENTRY	NOTIFY	UNSPECIFIED		
ERROR	NULL	UNTIL		
EXIT	OF	UNWIND		
EXITS	OPEN	USING		
EXPORTS	OR	VAL		
FALSE	ORD	VAR		
FINISHED	ORDERED	WAIT		
FIRST	OVERLAID	WHILE		
FOR	PACKED	WITH		
FORK	POINTER	WORD		
FRAME	PORT	ZONE		
FREE	PRED	.		
FROM	PRIVATE	{ }		
GO	PROC	;		
GOTO	PROCEDURE	=		
IF - THEN	PROCESS	,		
IF - THEN - ELSE	PROGRAM	:		

Appendix B. McCabe Constructs for Mesa

Listed below are all constructs in the Mesa language that are used as logical decision points in the calculation of McCabe's Cyclomatic Complexity.

Construct	Comment
IF	Conditional
SELECT	Multi Decision Conditional (Case)
WHILE	Loop - precondition
UNTIL	Loop - postcondition
THROUGH	Loop - elements of a set
FOR	Loop - iterative
AND	Logical AND
OR	Logical OR

Appendix C. Statistical Report Generated by the Code Metric Analyzer

```
CODE METRICS STATISTICS
***********************


Number of procedures:  149


                          LOC         CYC. COMPLEXITY       EFFORT
                        -------      -----------------     --------

       Average :        18.208             7.932           83710.710

  St. Deviation :       34.506            26.430          456614.437

           Min :          1                1                39.302

           Max :        320              210             4662495.500

*******************************************************************


Overlaps of highest metrics values ...

Considering the top  38 procedures (  25 percent ) ...


LOC / CC / Effort :

Total procedures =  23

 LOC      CC          EFFORT        Procedure Name
-----   ------      --------      ----------------

  32      25       91114.359      AnalyzerImpl/RunIt
  49      28      192890.203      CMStatsImpl/Overlap
  35      16       81116.781      CMStatsImpl/QuickCC
  21       9       33310.777      CMStatsImpl/QuickEffort
  21       9       35515.578      CMStatsImpl/QuickLOC
  23      10       60963.832      CMStatsImpl/QuickName
  35       7       98389.132      CMStatsImpl/ReadMetrics
 320     210     4662495.500      Grammar/ProcessQueue
 231     151     1970947.625      Grammar/ProcessQueueCont
  23       6       37698.492      HalsteadImpl/HsCompute
  19      12       36466.429      ParserImpl/Accept
  20      10       29695.220      ParserImpl/ExistingConfiguration
  32      12       72341.656      ParserImpl/GrowTree
  70      23      220672.203      ParserImpl/Parse
  60      29      175504.125      ParserImpl/Recover
  20      10       38852.316      ParserImpl/RightScan
  22       6       51668.382      ProcMetricsImpl/PmImpress
 252     196     2424798.000      ScannerImpl/Atom
  51      35      294474.718      ScannerImpl/CollectNumber
  27      13       62963.859      ScannerImpl/ErrorContext
  29      19      127131.421      ScannerImpl/ScanFloating
  25       7       38278.285      UtilsImpl/CardToString
  33       7       75960.078      UtilsImpl/RealToString

LOC / CC :

Total procedures =   3

 LOC      CC          EFFORT        Procedure Name
-----   ------      --------      ----------------

  24      11       19261.224      AnalyzerImpl/ProcessFile
  16       6       25507.697      ParserImpl/ActOnStack
  19       6       23037.746      ParserImpl/SyntaxError

LOC / Effort :

Total procedures =   8

 LOC      CC          EFFORT        Procedure Name
```

```
-----   ------     --------     ----------------

  34      1       27005.613     AnalyzerImpl
  22      2       31673.464     CMStatsImpl/AllocArrays
  34      2       28923.259     CMStatsImpl/MetImpress
  37      5       31019.351     CMStatsImpl/MetricStats
 141      1      240554.500     CMStatsImpl/StatsImpress
  27      1      122438.335     EncodeImpl/InitEncodeTable
  96      1      157919.593     ParserImpl
  57      1       55864.269     ScannerImpl

CC / Effort :

Total procedures =    4

 LOC      CC        EFFORT       Procedure Name
-----   ------     --------     ----------------

  15      11      27647.445     AnalyzerImpl/RunIt/BreakupFiles
  13      11      33145.804     ScannerImpl/ScanDecimal
  15      15      57725.308     ScannerImpl/ScanHex
  13      11      33145.804     ScannerImpl/ScanOctal

.................................................................


Sort by NAME :

 LOC      CC        EFFORT       Procedure Name
-----   ------     --------     ----------------

  34      1       27005.613     AnalyzerImpl
  11      3        5190.876     AnalyzerImpl/ClearIt
   3      3         411.194     AnalyzerImpl/ClearOutputFile
   9      1        3603.606     AnalyzerImpl/EnumerateFiles
   1      1          67.500     AnalyzerImpl/EnumerateFiles/AddEnumerating
   1      1          39.302     AnalyzerImpl/ExecutiveProc
   3      1         629.469     AnalyzerImpl/FreeNameDescriptor
  14      4        7249.634     AnalyzerImpl/GenStats
   4      3         713.118     AnalyzerImpl/GenerateStats
  17      3       13351.339     AnalyzerImpl/Initialize
   3      2         548.259     AnalyzerImpl/Initialize/Cleanup
   6      1        5941.321     AnalyzerImpl/InitializeErrors
  11      1       17862.277     AnalyzerImpl/MakeForm
   6      2        9980.321     AnalyzerImpl/MakeSWs
  24     11       19261.224     AnalyzerImpl/ProcessFile
  12      7       14610.231     AnalyzerImpl/ProcessFile/CleanUp
   3      3         431.754     AnalyzerImpl/Run
  32     25       91114.359     AnalyzerImpl/RunIt
  15     11       27647.445     AnalyzerImpl/RunIt/BreakupFiles
  14      5        3848.032     AnalyzerImpl/Unload
   1      1          71.323     AnalyzerImpl/WindowOutput
  15      1       12502.323     CMFileImpl
   4      1         951.999     CMFileImpl/ClearMetFile
   3      1         512.779     CMFileImpl/CloseMetFile
   1      1         175.663     CMFileImpl/EOFMetFile
   4      1        2143.400     CMFileImpl/GetMetRec
   4      1        1367.700     CMFileImpl/OpenMetFile
   2      1         533.290     CMFileImpl/ResetMetFile
   2      1         479.541     CMFileImpl/StoreMetRec
  27      1        5267.874     CMStatsImpl
  22      2       31673.464     CMStatsImpl/AllocArrays
   4      2        1796.011     CMStatsImpl/Cleanup
   4      1         917.552     CMStatsImpl/EncodeNewLine
   1      1         284.282     CMStatsImpl/EncodeWrite
  16      3       26407.884     CMStatsImpl/MakeNewNode
  34      2       28923.259     CMStatsImpl/MetImpress
   4      1         917.552     CMStatsImpl/MetNewLine
   1      1         284.282     CMStatsImpl/MetWrite
  37      5       31019.351     CMStatsImpl/MetricStats
   8      2        1833.941     CMStatsImpl/OutEncodeMap
  29      3       25808.105     CMStatsImpl/OutSort
  49     28      192890.203     CMStatsImpl/Overlap
  35     16       81116.781     CMStatsImpl/QuickCC
 21      9     33310.777   CMStatsImpl/QuickEffort
 21      9     35515.578   CMStatsImpl/QuickLOC
 23     10     60963.832   CMStatsImpl/QuickName
 35      7     98389.132   CMStatsImpl/ReadMetrics
141      1    240554.500   CMStatsImpl/StatsImpress
  4      1       917.552   CMStatsImpl/StatsNewLine
  1      1       284.282   CMStatsImpl/StatsWrite
  8      1      1589.004   EncodeImpl
 11      5     21177.693   EncodeImpl/DecodeChar
 15      3     20339.093   EncodeImpl/DecodeId
  9      5     16213.844   EncodeImpl/EncodeChar
 14      3     26582.455   EncodeImpl/EncodeId
 27      1    122436.335   EncodeImpl/InitEncodeTable
 10      1      1821.996   Grammar
  4      1      1089.273   Grammar/AssignDescriptors
  7      1       382.378   Grammar/Initialize
320    210   4662495.500   Grammar/ProcessQueue
231    151   1970947.625   Grammar/ProcessQueueCont
  5      1       427.400   HalsteadImpl
 23      6     37898.492   HalsteadImpl/HsCompute
 13      3     10600.215   HalsteadImpl/HsCompute/HsFree
 11      3     19002.226   HalsteadImpl/HsDump
  5      1      1913.293   HalsteadImpl/HsInit
 15      4     16419.607   HalsteadImpl/HsOperand
 15      4     16419.507   HalsteadImpl/HsOperator
  4      1       313.584   McCabeImpl
  5      2      3094.849   McCabeImpl/McCompute
 11      1     59653.515   McCabeImpl/McDump
  1      1       802.894   McCabeImpl/McIncrement
 13      1      9262.550   McCabeImpl/McInit
 96      1    157919.593   ParserImpl
 19     12     36466.429   ParserImpl/Accept
 15      6     25507.597   ParserImpl/ActOnStack
 10      7     11233.250   ParserImpl/AddLeaf
  1      1       217.944   ParserImpl/Advance
  4      2      4598.145   ParserImpl/Allocate
  9      7     10718.255   ParserImpl/CheckTree
  3      2       742.585   ParserImpl/Discard
 12      2      5194.714   ParserImpl/DisplayNode
  1      2       233.022   ParserImpl/EraseQueue
  2      2       738.402   ParserImpl/EraseStack
 20     10     29695.220   ParserImpl/ExistingConfiguration
  6      3      8325.503   ParserImpl/ExpandQueue
 12      3     17530.996   ParserImpl/ExpandStack
  7      5      6001.291   ParserImpl/FindNode
  2      3      2090.743   ParserImpl/FreeRowList
  2      4      4213.928   ParserImpl/GetNTEntry
 32     12     72341.656   ParserImpl/GrowTree
  1      1        67.500   ParserImpl/InputLoc
  2      1      1608.906   ParserImpl/LinkHash
  1      1       119.589   ParserImpl/NewLine
  6      6      7728.919   ParserImpl/NextRow
 70     23    220672.203   ParserImpl/Parse
 14      1      8044.816   ParserImpl/ParseInit
  2      1       145.296   ParserImpl/ParseReset
 14      8     22408.482   ParserImpl/ParseStep
  1      1        95.097   ParserImpl/ParserID
 60     29    175504.125   ParserImpl/Recover
  4      4      5148.029   ParserImpl/RecoverInput
 20     10     38852.316   ParserImpl/RightScan
 19      6     23037.746   ParserImpl/SyntaxError
  3      3      8990.468   ParserImpl/TypeSym
  3      2       742.585   ParserImpl/UnDiscard
  4      1       713.656   ParserImpl/input
 11      1      1465.634   ProcMetricsImpl
  5      1      4319.817   ProcMetricsImpl/PmDump
  7      2      9743.224   ProcMetricsImpl/PmGetProcName
 22      6     51668.382   ProcMetricsImpl/PmImpress
  1      1       553.801   ProcMetricsImpl/PmLines
  1      1       528.000   ProcMetricsImpl/PmMcCabe
  1      1       632.279   ProcMetricsImpl/PmOperand
  1      1       632.279   ProcMetricsImpl/PmOperator
  4      1      1681.699   ProcMetricsImpl/PmPop
  4      1      2049.399   ProcMetricsImpl/PmPush
  7      1      3482.928   ProcMetricsImpl/PmPush/PmInit
 57      1     55864.269   ScannerImpl
```

```
   5       3       7494.709    ScannerImpl/AppendDecimal
   4       2       7202.639    ScannerImpl/AppendHex
   4       1       4383.412    ScannerImpl/AppendOctal
   5       3       6696.540    ScannerImpl/AppendToScale
 252     196    2424798.000    ScannerImpl/Atom
  51      35     294474.718    ScannerImpl/CollectNumber
   3       2       1391.486    ScannerImpl/CollectNumber/Accept
   2       1       1808.729    ScannerImpl/EnterLit
  27      13      62963.859    ScannerImpl/ErrorContext
   3       3       1787.488    ScannerImpl/Escape
   7       2       6539.538    ScannerImpl/ExpandBuffer
   7       4       3765.871    ScannerImpl/FillBuffer
   1       1        553.801    ScannerImpl/ItemHandleToStreamIndex
   1       2        462.857    ScannerImpl/NextChar
  11       4       8423.299    ScannerImpl/ResetScanIndex
  13      11      33145.804    ScannerImpl/ScanDecimal
   4       7       3586.635    ScannerImpl/ScanError
  29      19     127131.421    ScannerImpl/ScanFloating
  15      15      57726.308    ScannerImpl/ScanHex
  21       2      10766.739    ScannerImpl/ScanInit
  13      11      33145.804    ScannerImpl/ScanOctal
  11       8      25329.830    ScannerImpl/ScanOctalChar
   3       2        619.347    ScannerImpl/ScanReset
   1       1        119.339    ScannerImpl/ScanStats
   2       3       3273.918    ScannerImpl/ValidFraction
   4       1        326.331    UtilsImpl
  25       7      38278.295    UtilsImpl/CardToString
  33       7      75960.078    UtilsImpl/RealToString
   2       1       1162.287    UtilsImpl/StringEqual
   2       1       1306.190    UtilsImpl/StringLess


Sort by LOC :

 LOC      CC        EFFORT       Procedure Name
-----   ------     --------     ----------------

 320     210    4662495.500    Grammar/ProcessQueue
 252     196    2424798.000    ScannerImpl/Atom
 231     151    1970947.625    Grammar/ProcessQueueCont
 141       1     240554.500    CMStatsImpl/StatsImpress
  96       1     157919.593    ParserImpl
  70      23     220672.203    ParserImpl/Parse
  60      29     175504.125    ParserImpl/Recover
  57       1      55864.269    ScannerImpl
  51      35     294474.718    ScannerImpl/CollectNumber
  49      28     192890.203    CMStatsImpl/Overlap
  37       5      31019.351    CMStatsImpl/MetricStats
  35       7      96389.132    CMStatsImpl/ReadMetrics
  35      16      81116.781    CMStatsImpl/QuickCC
  34       1      27005.613    AnalyzerImpl
  34       2      28923.259    CMStatsImpl/MetImpress
  33       7      75960.078    UtilsImpl/RealToString
  32      12      72341.656    ParserImpl/GrowTree
  32      25      91114.259    AnalyzerImpl/RunIt
  29      19     127131.421    ScannerImpl/ScanFloating
  29       3      25808.105    CMStatsImpl/OutSort
  27      13      62963.859    ScannerImpl/ErrorContext
  27       1       5267.874    CMStatsImpl
  27       1     122436.335    EncodeImpl/InitEncodeTable
  25       7      38278.285    UtilsImpl/CardToString
  24      11      19261.224    AnalyzerImpl/ProcessFile
  23      10      60963.832    CMStatsImpl/QuickName
  23       6      37698.492    HalsteadImpl/HsCompute
  22       2      31673.464    CMStatsImpl/AllocArrays
  22       6      51668.382    ProcMetricsImpl/PmImpress
  21       9      25515.578    CMStatsImpl/QuickLOC
  21       2      10766.739    ScannerImpl/ScanInit
  21       9      33310.777    CMStatsImpl/QuickEffort
  20      10      29695.220    ParserImpl/ExistingConfiguration
  20      10      38852.316    ParserImpl/RightScan
  19      12      36466.429    ParserImpl/Accept
  19       6      23037.746    ParserImpl/SyntaxError
  17       3      13351.339    AnalyzerImpl/Initialize
  16       6      25507.697    ParserImpl/ActOnStack
  16       3      26407.884    CMStatsImpl/MakeNewNode
```

```
CMAnalyzer.stats          21-Jun-88 16:26:57 EDT

  15      1    12502.323   CMFileImpl
  15     15    57726.308   ScannerImpl/ScanHex
  15      3    20339.093   EncodeImpl/DecodeId
  15     11    27647.445   AnalyzerImpl/RunIt/BreakupFiles
  15      4    16419.607   HalsteadImpl/HsOperand
  15      4    16419.607   HalsteadImpl/HsOperator
  14      5     3843.032   AnalyzerImpl/Unload
  14      3    26532.455   EncodeImpl/EncodeId
  14      8    22408.482   ParserImpl/ParseStep
  14      4     7249.634   AnalyzerImpl/GenStats
  14      1     8044.816   ParserImpl/ParseInit
  13     11    33145.804   ScannerImpl/ScanDecimal
  13     11    33145.804   ScannerImpl/ScanOctal
  13      3    10600.215   HalsteadImpl/HsCompute/HsFree
  13      1     9262.550   McCabeImpl/McInit
  12      3    17530.996   ParserImpl/ExpandStack
  12      2     5194.714   ParserImpl/DisplayNode
  12      7    14610.231   AnalyzerImpl/ProcessFile/CleanUp
  11      3     5190.876   AnalyzerImpl/ClearIt
  11      3    19002.236   HalsteadImpl/HsDump
  11      4     8423.299   ScannerImpl/ResetScanIndex
  11      5    21177.593   EncodeImpl/DecodeChar
  11      1     1465.634   ProcMetricsImpl
  11      1    59663.515   McCabeImpl/McDump
  11      8    25323.930   ScannerImpl/ScanOctalChar
  11      1    17852.277   AnalyzerImpl/MakeForm
  10      1     1821.996   Grammar
  10      7    11233.350   ParserImpl/AddLeaf
   9      7    10718.255   ParserImpl/CheckTree
   9      1     3603.606   AnalyzerImpl/EnumerateFiles
   9      5    16213.844   EncodeImpl/EncodeChar
   8      2     1333.941   CMStatsImpl/OutEncodeMap
   8      1     1589.004   EncodeImpl
   7      1     3482.928   ProcMetricsImpl/PmPush/PmInit
   7      1      382.378   Grammar/Initialize
   7      5     6001.291   ParserImpl/FindNode
   7      2     9743.224   ProcMetricsImpl/PmGetProcName
   7      4     3765.871   ScannerImpl/FillBuffer
   7      2     6539.538   ScannerImpl/ExpandBuffer
   6      3     8325.503   ParserImpl/ExpandQueue
   6      2     9980.321   AnalyzerImpl/MakeSWs
   6      1     5941.321   AnalyzerImpl/InitializeErrors
   6      6     7728.919   ParserImpl/NextRow
   5      1      427.400   HalsteadImpl
   5      3     6696.540   ScannerImpl/AppendToScale
   5      1     4319.817   ProcMetricsImpl/PmDump
   5      3     7494.709   ScannerImpl/AppendDecimal
   5      2     3094.849   McCabeImpl/McCompute
   5      1     1913.293   HalsteadImpl/HsInit
   4      1      326.231   UtilsImpl
   4      1      313.684   McCabeImpl
   4      1      713.656   ParserImpl/input
   4      2     7202.639   ScannerImpl/AppendHex
   4      1     1089.273   Grammar/AssignDescriptors
   4      1     2049.399   ProcMetricsImpl/PmPush
   4      1     1367.700   CMFileImpl/OpenMetFile
   4      1     1681.699   ProcMetricsImpl/PmPop
   4      1     4383.412   ScannerImpl/AppendOctal
   4      2     4598.145   ParserImpl/Allocate
   4      1     2143.400   CMFileImpl/GetMetRec
   4      1      951.999   CMFileImpl/ClearMetFile
   4      3      713.118   AnalyzerImpl/GenerateStats
   4      2     1796.011   CMStatsImpl/Cleanup
   4      1      917.552   CMStatsImpl/StatsNewLine
   4      1      917.552   CMStatsImpl/MetNewLine
   4      4     5148.029   ParserImpl/RecoverInput
   4      7     3586.625   ScannerImpl/ScanError
   4      1      917.552   CMStatsImpl/EncodeNewLine
   3      2      742.585   ParserImpl/UnDiscard
   3      3      411.194   AnalyzerImpl/ClearOutputFile
   3      2      548.259   AnalyzerImpl/Initialize/Cleanup
   3      2      742.585   ParserImpl/Discard
   3      3      431.754   AnalyzerImpl/Run
   3      3     8990.468   ParserImpl/TypeSym
   3      2      619.347   ScannerImpl/ScanReset
   3      3     1787.488   ScannerImpl/Escape
   3        1        512.779    CMFileImpl/CloseMetFile
   3        1        629.469    AnalyzerImpl/FreeNameDescriptor
   3        2       1391.486    ScannerImpl/CollectNumber/Accept
   2        3       3273.918    ScannerImpl/ValidFraction
   2        1       1808.729    ScannerImpl/EnterLit
   2        3       2090.743    ParserImpl/FreeRowList
   2        4       4213.928    ParserImpl/GetNTEntry
   2        1       1602.906    ParserImpl/LinkHash
   2        1        479.541    CMFileImpl/StoreMetRec
   2        1       1152.237    UtilsImpl/StringEqual
   2        1        533.290    CMFileImpl/ResetMetFile
   2        1       1306.190    UtilsImpl/StringLess
   2        2        738.402    ParserImpl/EraseStack
   2        1        145.296    ParserImpl/ParseReset
   1        1        175.663    CMFileImpl/EOFMetFile
   1        2        462.857    ScannerImpl/NextChar
   1        1        553.801    ProcMetricsImpl/PmLines
   1        1        528.000    ProcMetricsImpl/PmMcCabe
   1        1        632.279    ProcMetricsImpl/PmOperand
   1        1        632.279    ProcMetricsImpl/PmOperator
   1        1         67.500    AnalyzerImpl/EnumerateFiles/AddEnumerating
   1        1        119.589    ParserImpl/NewLine
   1        1        113.339    ScannerImpl/ScanStats
   1        1        294.282    CMStatsImpl/StatsWrite
   1        1        294.282    CMStatsImpl/EncodeWrite
   1        1        284.282    CMStatsImpl/MetWrite
   1        1        553.801    ScannerImpl/ItemHandleToStreamIndex
   1        1        217.944    ParserImpl/Advance
   1        1         95.097    ParserImpl/ParserID
   1        2        233.022    ParserImpl/EraseQueue
   1        1         67.500    ParserImpl/InputLoc
   1        1        802.894    McCabeImpl/McIncrement
   1        1         39.302    AnalyzerImpl/ExecutiveProc
   1        1         71.323    AnalyzerImpl/WindowOutput


Sort by Cyclomatic Complexity :

 LOC      CC        EFFORT        Procedure Name
-----   ------     --------     ----------------

 320     210    4662495.500     Grammar/ProcessQueue
 252     196    2424798.000     ScannerImpl/Atom
 231     151    1970947.625     Grammar/ProcessQueueCont
  51      35     294474.718     ScannerImpl/CollectNumber
  60      29     175504.125     ParserImpl/Recover
  49      28     192890.203     CMStatsImpl/Overlap
  32      25      91114.359     AnalyzerImpl/RunIt
  70      23     220672.203     ParserImpl/Parse
  29      19     127131.421     ScannerImpl/ScanFloating
  35      16      81116.781     CMStatsImpl/QuickCC
  15      15      57726.308     ScannerImpl/ScanHex
  27      13      62963.859     ScannerImpl/ErrorContext
  32      12      72341.656     ParserImpl/GrowTree
  19      12      36466.429     ParserImpl/Accept
  13      11      33145.804     ScannerImpl/ScanDecimal
  15      11      27647.445     AnalyzerImpl/RunIt/BreakupFiles
  24      11      19261.224     AnalyzerImpl/ProcessFile
  13      11      33145.804     ScannerImpl/ScanOctal
  23      10      60963.832     CMStatsImpl/QuickName
  20      10      29695.220     ParserImpl/ExistingConfiguration
  20      10      38852.316     ParserImpl/RightScan
  21       9      35515.578     CMStatsImpl/QuickLOC
  21       9      33310.777     CMStatsImpl/QuickEffort
  14       8      22408.482     ParserImpl/ParseStep
  11       8      25328.830     ScannerImpl/ScanOctalChar
   9       7      10718.255     ParserImpl/CheckTree
  10       7      11233.350     ParserImpl/AddLeaf
  25       7      38278.285     UtilsImpl/CardToString
  35       7      98389.132     CMStatsImpl/ReadMetrics
  12       7      14610.231     AnalyzerImpl/ProcessFile/CleanUp
  33       7      75960.078     UtilsImpl/RealToString
   4       7       3586.635     ScannerImpl/ScanError
  16       6      25507.697     ParserImpl/ActOnStack
  22       6      51668.382     ProcMetricsImpl/PmImpress
  23       6      37698.492     HalsteadImpl/HsCompute
```

```
CMAnalyzer.stats          21-Jun-88 16:26:57 EDT

   19        6        23037.746    ParserImpl/SyntaxError
    6        6         7728.919    ParserImpl/NextRow
   14        5         3848.032    AnalyzerImpl/Unload
   11        5        21177.693    EncodeImpl/DecodeChar
    9        5        16213.844    EncodeImpl/EncodeChar
   37        5        31019.351    CMStatsImpl/MetricStats
    7        5         6001.291    ParserImpl/FindNode
   11        4         8423.299    ScannerImpl/ResetScanIndex
    7        4         3785.371    ScannerImpl/FillBuffer
   15        4        16419.607    HalsteadImpl/HsOperator
   15        4        16419.607    HalsteadImpl/HsOperand
    2        4         4213.928    ParserImpl/GetNTEntry
   14        4         7249.634    AnalyzerImpl/GenStats
    4        4         5148.029    ParserImpl/RecoverInput
    3        3          431.754    AnalyzerImpl/Run
    4        3          713.118    AnalyzerImpl/GenerateStats
    2        3         2090.743    ParserImpl/FreeRowList
    5        3         6696.540    ScannerImpl/AppendToScale
    2        3         3273.918    ScannerImpl/ValidFraction
    5        3         7494.709    ScannerImpl/AppendDecimal
   11        3        19002.236    HalsteadImpl/HsDump
   13        3        10600.215    HalsteadImpl/HsCompute/HsFree
   11        3         5190.576    AnalyzerImpl/ClearIt
    3        3         1787.489    ScannerImpl/Escape
   15        3        20339.093    EncodeImpl/Decodeid
    3        3         3990.468    ParserImpl/TypeSym
   14        3        25582.455    EncodeImpl/EncodeId
   16        3        26407.884    CMStatsImpl/MakeNewNode
    3        3          411.194    AnalyzerImpl/ClearOutputFile
   17        3        13351.339    AnalyzerImpl/Initialize
   12        3        17530.996    ParserImpl/ExpandStack
   29        3        25608.105    CMStatsImpl/OutSort
    6        3         6325.503    ParserImpl/ExpandQueue
    1        2          233.022    ParserImpl/EraseQueue
    2        2          738.402    ParserImpl/EraseStack
   12        2         5194.714    ParserImpl/DisplayNode
    4        2         4598.145    ParserImpl/Allocate
    8        2         1823.941    CMStatsImpl/OutEncodeMap
   34        2        28923.259    CMStatsImpl/MetImpress
    4        2         1796.011    CMStatsImpl/Cleanup
    3        2          548.259    AnalyzerImpl/Initialize/Cleanup
    3        2          619.347    ScannerImpl/ScanReset
    6        2         9980.321    AnalyzerImpl/MakeSWs
    5        2         3094.849    McCabeImpl/McCompute
   22        2        31673.464    CMStatsImpl/AllocArrays
   21        2        10766.739    ScannerImpl/ScanInit
    3        2          742.585    ParserImpl/Discard
    3        2          742.585    ParserImpl/UnDiscard
    4        2         7202.639    ScannerImpl/AppendHex
    3        2         1391.486    ScannerImpl/CollectNumber/Accept
    1        2          462.857    ScannerImpl/NextChar
    7        2         6539.538    ScannerImpl/ExpandBuffer
    7        2         9743.224    ProcMetricsImpl/PmGetProcName
   10        1         1821.996    Grammar
    5        1         1913.293    HalsteadImpl/HsInit
    5        1          427.400    HalsteadImpl
    7        1          382.378    Grammar/Initialize
    4        1         1089.273    Grammar/AssignDescriptors
    8        1         1589.004    EncodeImpl
   13        1         9262.550    McCabeImpl/McInit
    1        1          802.894    McCabeImpl/McIncrement
   11        1        59653.515    McCabeImpl/McDump
    4        1          313.684    McCabeImpl
   27        1       122436.335    EncodeImpl/InitEncodeTable
   27        1         5267.874    CMStatsImpl
    4        1          713.656    ParserImpl/input
   14        1         8044.816    Parserimpl/ParseInit
    2        1          145.296    ParserImpl/ParseReset
    4        1          917.552    CMStatsImpl/MetNewLine
    1        1          284.282    CMStatsImpl/MetWrite
    1        1           67.500    ParserImpl/InputLoc
    4        1          917.552    CMStatsImpl/EncodeNewLine
    1        1          284.282    CMStatsImpl/EncodeWrite
  141        1       240554.500    CMStatsImpl/StatsImpress
    1        1           95.097    ParserImpl/ParserID
    4        1          917.552    CMStatsImpl/StatsNewLine
```

```
CMAnalyzer.stats          21-Jun-88 16:26:57 EDT

    1       1          284.282    CMStatsImpl/StatsWrite
    2       1         1602.906    ParserImpl/LinkHash
    1       1          217.944    ParserImpl/Advance
    1       1          119.589    ParserImpl/NewLine
   96       1       157919.593    ParserImpl
    1       1          632.279    ProcMetricsImpl/PmOperator
    1       1          632.279    ProcMetricsImpl/PmOperand
    1       1          528.000    ProcMetricsImpl/PmMcCabe
    1       1          553.801    ProcMetricsImpl/PmLines
   15       1        12502.323    CMFileImpl
    4       1          951.999    CMFileImpl/ClearMetFile
    4       1         2143.400    CMFileImpl/GetMetRec
    2       1          479.541    CMFileImpl/StoreMetRec
    1       1          175.663    CMFileImpl/EOFMetFile
    2       1          533.290    CMFileImpl/ResetMetFile
    3       1          512.779    CMFileImpl/CloseMetFile
    4       1         1367.700    CMFileImpl/OpenMetFile
   34       1        27005.613    AnalyzerImpl
    7       1         3482.928    ProcMetricsImpl/PmPush/PmInit
    4       1         2049.399    ProcMetricsImpl/PmPush
    6       1         5941.321    AnalyzerImpl/InitializeErrors
    4       1         1681.599    ProcMetricsImpl/PmPop
   11       1        17952.277    AnalyzerImpl/MakeForm
    9       1         3602.505    AnalyzerImpl/EnumerateFiles
    1       1           67.500    AnalyzerImpl/EnumerateFiles/AddEnumerating
    5       1         4319.317    ProcMetricsImpl/PmDump
   11       1         1465.504    ProcMetricsImpl
    4       1         4383.412    ScannerImpl/AppendOctal
    2       1         1208.729    ScannerImpl/EnterLit
    1       1          119.589    ScannerImpl/ScanStats
    1       1          553.301    ScannerImpl/ItemHandleToStreamIndex
   57       1        55864.259    ScannerImpl
    2       1         1162.287    UtilsImpl/StringEqual
    2       1         1306.190    UtilsImpl/StringLess
    3       1          629.469    AnalyzerImpl/FreeNameDescriptor
    4       1          326.231    UtilsImpl
    1       1           39.302    AnalyzerImpl/ExecutiveProc
    1       1           71.323    AnalyzerImpl/WindowOutput


Sort by Effort :

 LOC      CC        EFFORT        Procedure Name
-----    ------    ---------     ----------------

 320      210     4662495.500    Grammar/ProcessQueue
 252      196     2424798.000    ScannerImpl/Atom
 231      151     1970947.625    Grammar/ProcessQueueCont
  51       35      294474.718    ScannerImpl/CollectNumber
 141        1      240554.500    CMStatsImpl/StatsImpress
  70       23      220672.203    ParserImpl/Parse
  49       28      192890.203    CMStatsImpl/Overlap
  60       29      175504.125    ParserImpl/Recover
  96        1      157919.593    ParserImpl
  29       19      127131.421    ScannerImpl/ScanFloating
  27        1      122436.335    EncodeImpl/InitEncodeTable
  35        7       98389.132    CMStatsImpl/ReadMetrics
  32       25       91114.359    AnalyzerImpl/RunIt
  35       16       81116.781    CMStatsImpl/QuickCC
  33        7       75960.078    UtilsImpl/RealToString
  32       12       72341.656    ParserImpl/GrowTree
  27       13       62963.859    ScannerImpl/ErrorContext
  23       10       60963.832    CMStatsImpl/QuickName
  11        1       59653.515    McCabeImpl/McDump
  15       15       57726.308    ScannerImpl/ScanHex
  57        1       55864.269    ScannerImpl
  22        6       51668.382    ProcMetricsImpl/PmImpress
  20       10       38852.316    ParserImpl/RightScan
  25        7       38278.285    UtilsImpl/CardToString
  23        6       37698.492    HalsteadImpl/HsCompute
  19       12       36466.429    ParserImpl/Accept
  21        9       35515.578    CMStatsImpl/QuickLOC
  21        9       33310.777    CMStatsImpl/QuickEffort
  13       11       33145.804    ScannerImpl/ScanDecimal
  13       11       33145.804    ScannerImpl/ScanOctal
  22        2       31673.464    CMStatsImpl/AllocArrays
```

```
CMAnalyzer.stats          21-Jun-88 18:26:57 EDT

  37        5        31019.351    CMStatsImpl/MetricStats
  20       10        29695.220    ParserImpl/ExistingConfiguration
  34        2        29923.259    CMStatsImpl/MetImpress
  15       11        27647.445    AnalyzerImpl/RunIt/BreakupFiles
  34        1        27005.813    AnalyzerImpl
  14        3        26582.455    EncodeImpl/EncodeId
  16        3        26407.884    CMStatsImpl/MakeNewNode
  29        3        25308.105    CMStatsImpl/OutSort
  16        6        25507.597    ParserImpl/ActOnStack
  11        8        25329.330    ScannerImpl/ScanOctalChar
  19        6        23037.746    ParserImpl/SyntaxError
  14        8        22408.482    ParserImpl/ParseStep
  11        5        21177.693    EncodeImpl/DecodeChar
  15        3        20339.093    EncodeImpl/DecodeId
  24       11        19261.224    AnalyzerImpl/ProcessFile
  11        3        19002.236    HalsteadImpl/HsDump
  11        1        17862.277    AnalyzerImpl/MakeForm
  12        3        17530.996    ParserImpl/ExpandStack
  15        4        16419.607    HalsteadImpl/HsOperator
  15        4        16419.607    HalsteadImpl/HsOperand
   9        5        16213.844    EncodeImpl/EncodeChar
  12        7        14510.231    AnalyzerImpl/ProcessFile/CleanUp
  17        3        13351.239    AnalyzerImpl/Initialize
  15        1        12502.323    CMFileImpl
  10        7        11233.350    ParserImpl/AddLeaf
  21        2        10766.739    ScannerImpl/ScanInit
   9        7        10718.255    ParserImpl/CheckTree
  13        3        10600.215    HalsteadImpl/HsCompute/HsFree
   6        2         9980.321    AnalyzerImpl/MakeSws
   7        2         9743.224    ProcMetricsImpl/PmGetProcName
  13        1         9252.550    McCabeImpl/McInit
   3        3         8990.468    ParserImpl/TypeSym
  11        4         8423.299    ScannerImpl/ResetScanIndex
   6        3         8325.503    ParserImpl/ExpandQueue
  14        1         8044.816    ParserImpl/ParseInit
   6        6         7728.919    ParserImpl/NextRow
   5        3         7494.709    ScannerImpl/AppendDecimal
  14        4         7249.834    AnalyzerImpl/GenStats
   4        2         7202.639    ScannerImpl/AppendHex
   5        3         6696.540    ScannerImpl/AppendToScale
   7        2         6539.538    ScannerImpl/ExpandBuffer
   7        5         6001.291    ParserImpl/FindNode
   6        1         5941.321    AnalyzerImpl/InitializeErrors
  27        1         5267.874    CMStatsImpl
  12        2         5194.714    ParserImpl/DisplayNode
  11        3         5190.876    AnalyzerImpl/ClearIt
   4        4         5148.029    ParserImpl/RecoverInput
   4        2         4598.145    ParserImpl/Allocate
   4        1         4383.412    ScannerImpl/AppendOctal
   5        1         4319.817    ProcMetricsImpl/PmDump
   2        4         4213.928    ParserImpl/GetNTEntry
  14        5         3848.032    AnalyzerImpl/Unload
   7        4         3765.871    ScannerImpl/FillBuffer
   9        1         3603.806    AnalyzerImpl/EnumerateFiles
   4        7         3586.635    ScannerImpl/ScanError
   7        1         3482.928    ProcMetricsImpl/PmPush/PmInit
   2        3         3273.918    ScannerImpl/ValidFraction
   5        2         3094.849    McCabeImpl/McCompute
   4        1         2143.400    CMFileImpl/GetMetRec
   2        3         2090.743    ParserImpl/FreeRowList
   4        1         2049.399    ProcMetricsImpl/PmPush
   5        1         1913.293    HalsteadImpl/HsInit
   8        2         1833.941    CMStatsImpl/OutEncodeMap
  10        1         1821.996    Grammar
   2        1         1808.729    ScannerImpl/EnterLit
   4        2         1796.011    CMStatsImpl/Cleanup
   3        3         1787.488    ScannerImpl/Escape
   4        1         1681.699    ProcMetricsImpl/PmPop
   2        1         1608.906    ParserImpl/LinkHash
   8        1         1589.004    EncodeImpl
  11        1         1465.634    ProcMetricsImpl
   3        2         1391.486    ScannerImpl/CollectNumber/Accept
   4        1         1367.700    CMFileImpl/OpenMetFile
   2        1         1306.190    UtilsImpl/StringLess
   2        1         1162.287    UtilsImpl/StringEqual
   4        1         1089.273    Grammar/AssignDescriptors
  4       1       951.999   CMFileImpl/ClearMetFile
  4       1       917.552   CMStatsImpl/EncodeNewLine
  4       1       917.552   CMStatsImpl/MetNewLine
  4       1       917.552   CMStatsImpl/StatsNewLine
  1       1       802.894   McCabeImpl/McIncrement
  3       2       742.585   ParserImpl/Discard
  3       2       742.585   ParserImpl/UnDiscard
  2       2       738.402   ParsarImpl/EraseStack
  4       1       713.356   ParserImpl/input
  4       3       713.113   AnalyzerImpl/GenerateStats
  1       1       632.279   ProcMetricsImpl/PmOperand
  1       1       632.279   ProcMetricsImpl/PmOperator
  3       1       629.469   AnalyzerImpl/FreeNameDescriptor
  3       2       619.347   ScannerImpl/ScanReset
  1       1       553.801   ProcMetricsImpl/PmLines
  1       1       553.801   ScannerImpl/ItemHandleToStreamIndex
  3       2       548.259   AnalyzerImpl/Initialize/Cleanup
  2       1       533.290   CMFileImpl/ResetMetFile
  1       1       528.000   ProcMetricsImpl/PmMcCabe
  3       1       512.779   CMFileImpl/CloseMetFile
  2       1       479.541   CMFileImpl/StoreMetRec
  1       2       462.857   ScannerImpl/NextChar
  3       3       431.754   AnalyzerImpl/Run
  5       1       427.400   HalsteadImpl
  3       3       411.194   AnalyzerImpl/ClearOutputFile
  7       1       382.378   Grammar/Initialize
  4       1       326.231   UtilsImpl
  4       1       313.684   McCabeImpl
  1       1       284.282   CMStatsImpl/StatsWrite
  1       1       284.282   CMStatsImpl/EncodeWrite
  1       1       284.282   CMStatsImpl/MetWrite
  1       2       233.022   ParserImpl/EraseQueue
  1       1       217.944   ParserImpl/Advance
  1       1       175.663   CMFileImpl/EOFMetFile
  2       1       145.296   ParserImpl/ParseReset
  1       1       119.589   ParserImpl/NewLine
  1       1       119.589   ScannerImpl/ScanStats
  1       1        95.097   ParserImpl/ParserID
  1       1        71.323   AnalyzerImpl/WindowOutput
  1       1        67.500   AnalyzerImpl/EnumerateFiles/AddEnumerating
  1       1        67.500   ParserImpl/InputLoc
  1       1        39.302   AnalyzerImpl/ExecutiveProc
```

Appendix D. Code Metric Analyzer User's Guide

Mesa Code Metrics Analyzer

User's Guide

June, 1988

John Lewis

Introduction

The Mesa Code Metrics Analyzer computes software quality metrics for arbitrary Mesa source code which indicate the level of complexity of the software. This data can be used to identify highly error-prone code, and therefore indicate where redesign and testing efforts should be concentrated.

The metrics calculated using this tool are Lines of Code, Halstead's Software Science indicators (specifically Effort), and McCabe's Cyclomatic Complexity metric. These metrics are calculated for each procedure in the analyzed software.

Once data has been collected across a set of procedures, a statistical report can be generated to assist the metrics interpretation. Sorts are performed for each metric, and overlaps are computed to indicate the set of procedures that are considered most complex by certain combinations of the metrics. The average, standard deviation, min and max are also computed for each metric.

Additionally, this tool can create data files containing the metrics information which are readable by another metrics analysis tool currently residing at Virginia Tech. This data can be encoded to protect any proprietary information.

This document is designed to assist the user of the Mesa Code Metrics Analyzer in generating and interpreting the metrics data.

See the Mesa Code Metrics Analyzer Implementation Guide for additional information on the internal makeup, techniques, and data structures used in the analyzer.

Input

The following information is used as input to the Mesa Code Metrics Analyzer:

Mesa Source Code

The source code analyzed is assumed to be syntactically correct, although the analyzer will detect syntax errors and abort processing. Any Mesa source code may be processed, including definitions files, but metrics are only calculated for executable procedures. The input file names are listed in the form subwindow of the Mesa Code Metrics Analyzer tool and may be specified using lists, wildcard characters, or by using ToolDriver.

Statistical Processing Constraints

Parameters controlling the statistical and additional reports are specified in the form subwindow of the tool. They include:

1) a numeric indicating the percentage of total procedures that should be considered when computing the overlap of highly complex procedures across the metrics,

2) a flag indicating if the additional data file for the Virginia Tech metrics analyzer should be created, and

3) a flag indicating if the output data should be encoded to protect proprietary information.

A complete explanation of the form subwindow fields which control the input and processing of the Mesa Code Metrics Analyzer is given below.

Output

The following output files are generated by the Mesa Code Metrics Analyzer:

CodeMetrics.data

This binary file contains the raw metrics data for each procedure analyzed. It serves as input to the statistical report generating software. The data for each file as processed is appended to this file until cleared by the Clear! command. Once cleared, the data is destroyed. If further analysis is desired on the current set of procedures, this file should be renamed or moved off the search path prior to executing the Clear! command.

CodeMetrics.stats

This file contains the statistical report generated to assist in the interpretation of the metric data. This file is generated as a result of executing the Stats! command. A complete explanation of the report is given in the following section.

CodeMetrics.out

If the Backend Output boolean field is set in the form subwindow of the tool when the Stats! command is executed, this file will be generated. It contains the code metrics data in a text format which is readable by another metrics analysis tool currently residing at Virginia Tech. This file should not be generated if the data is not to be processed by the other tool.

CodeMetrics.encode

If the Encode boolean field is set when the Stats! command is executed, all procedure names in the text output files will be encoded to protect proprietary information. In addition, this file will be generated. It contains a listing of the encoded procedure aliases in alphabetical order and their correct names. Encoding should be done only if the data is to be further analyzed off site or by non-cleared personnel. This file will then assist in mapping the encoded analysis back to the proper software.

CMAnalyzer.Log

This file contains a log of the processing of the Mesa Code Metrics Analyzer tool. Processing information is also output to the tool's file subwindow.

The Metrics Statistics Report

Executing the Stats! command in the form subwindow creates the text file CodeMetrics.stats which contains a statistical report of the metrics data currently stored in CodeMetrics.data. The analysis is based on the Lines of Code (LOC), Halstead's Effort, and McCabe's Cyclomatic Complexity metrics. The report is separated into three sections.

The first section gives an average, standard deviation, minimum and maximum values for each of the metrics. It also reports the total number of procedures that are considered in the statistical analysis.

The second section reports the procedures which overlap some top percentage of the highly complex procedures in various combinations of the metrics. The overlaps are computed by sorting the procedures by each of the metrics, then considering only a certain percentage of the data as specified in the Percent Overlap field of the form subwindow. Of this percentage, lists of procedures are constructed which appear in the highest values of all metrics, and of any combination of two metrics.

For example, if the Percent Overlap field contained the value 50, the top half of the procedures in each of the metrics are compared and four lists are reported: a list of the procedures who appeared in the top half of all three metrics, the procedures who appear in the top half of only LOC and Cyclomatic Complexity, the procedures who appear in the top half of only LOC and Effort, and finally the procedures who appear in the top half of only Cyclomatic Complexity and Effort.

The third section contains four sorts of all procedures. The data is sorted by procedure name and by each of the three metrics.

Any time a list of procedures is given in the report, the associated metric values are supplied as well.

Analyzer Tool Fields

The following are the fields in the form subwindow of the Code Metrics Analyzer which control its operation:

Mesa Source Files:

This field accepts the file or files of Mesa source code to be analyzed. Files may be specified using lists and/or wildcards. An extension of .mesa is assumed if none is provided.

Go!

This command item performs the source code analysis on the files specified in the Mesa Source Files field. LOC, Halstead, and McCabe metrics are calculated for each procedure in the specified files. The analysis data of each run is appended to any previous results since the last Clear! execution. Only raw data is generated with this command; no reports are created. The metrics data is stored in the binary file CodeMetrics.data.

Stats!

This command performs a statistical analysis on the metrics data generated by the source code analysis (Go! command). The report is written to the text file CodeMetrics.stats. A complete explanation of the report is given in the previous section. Each execution of Stats! overwrites any previous report.

Clear!

Execution of Clear! removes all data previously generated by the source code analysis stored in CodeMetrics.data. The data file should be saved in another file or removed from the search path prior to executing Clear! if that analysis will be needed in the future.

Percent Overlap =

This field accepts a numeric indicating the percentage of the total number of analyzed procedures that are considered when computing the overlap between the highly complex procedures. This field only affects the execution of the Stats! command. If the field contains a value that is less than one or greater than 100, a value of 100 percent is used. This field defaults to 10 percent.

BackEnd Output

This boolean item only affects the execution of the Stats! command. If the flag is set when Stats! is run, a text data file is created which serves as input to a backend metrics analysis tool. The data file is called CodeMetrics.out and will rewrite any old data in the file. This field defaults to false.

Encode

Similar to the BackEnd Output field, this boolean item only affects the execution of the Stats! command. If it is set when Stats! is executed, then any module and procedure names are encoded in the statistics output report and in the backend data file. Also, this flag will cause another text file to be created, called CodeMetrics.encode, which lists the procedure names in normal and encoded form in alphabetical order by alias. This field defaults to false.

Restrictions and Limitations

Several restrictions currently exist in the Mesa Code Metrics Analyzer. They are listed below.

1) The source code analyzed must be syntactically valid. The analyzer will abort processing on the first syntax error detected.

2) To statistically analyze groups of source code procedures at various hierarchial levels without having to reanalyze the source code, the smallest groups must be analyzed (saving the data), then manually appended together to form larger groups which can then be analyzed.

3) The largest group of procedures which can be processed in one analysis is limited by the amount of available memory on the workstation.

3) Executing the Clear! command destroys all data in the current data file. The data file must be renamed or moved off the search path if it will be needed in the future.

Appendix E. Code Metric Analyzer Implementation Guide

Mesa Code Metrics Analyzer

Implementation Guide

June, 1988

John Lewis

Introduction

The Mesa Code Metrics Analyzer processes source code written in the Mesa programming language and produces software quality metrics for individual procedures in the code. These values identify code which is highly complex and error-prone, therefore where redesign and testing efforts should be concentrated. The analyzer also produces a statistical report of the metrics data collected.

This document serves as a guide to the implementation of the analyzer, including its makeup, techniques, and data structures. It should be used to assist any maintenance activities on the analyzer.

See the Mesa Code Metrics Analyzer User's Guide for additional information on the tool's use and data interpretation.

Impl / Interface Relationships

The following describe the important files used in the Mesa Code Metrics Analyzer and their relationships:

INTERFACE	EXPORTED BY
ProcMetrics.mesa	ProcMetricsImpl.mesa HalsteadImpl.mesa McCabeImpl.mesa CMStatsImpl.mesa
Utils.mesa	UtilsImpl.mesa EncodeImpl.mesa CMFileImpl.mesa AnalyzerImpl.mesa
Parser.mesa	ParserImpl.mesa ScannerImpl.mesa Grammar.mesa

Configuration File: CMAnalyzer.config

Executable Image: CMAnalyzer.bcd

DF File: CMAnalyzer.df

Module Descriptions

The following are the modules which make up the Mesa Code Metrics Analyzer:

ProcMetricsImpl.mesa

This file contains the high level metrics data collection routines. As the Mesa grammar is perused while parsing the source file, calls are made to these routines to collect the appropriate information that will eventually yield the LOC, Halstead's effort, and McCabe's Cyclomatic Complexity metrics. These routines rely heavily on the lower level collection procedures in HalsteadImpl.mesa and McCabeImpl.mesa. Also included in this file is the routine which calculates a procedure's metrics once the data has been collected and writes it to the binary data file CodeMetrics.data. The module is exported to ProcMetrics.mesa.

HalsteadImpl.mesa

The routines in this file collect the information necessary to calculate the Halstead Software Science indicators, specifically lists of unique operators and operands in the code. These routines are called by the higher level control routines in ProcMetricsImpl.mesa. This module is exported to ProcMetrics.mesa.

McCabeImpl.mesa

The routines in this file collect the information necessary to calculate McCabe's Cyclomatic Complexity metric, specifically the count of decision points in the code. These routines are called by the higher level control routines in ProcMetricsImpl.mesa. This module is exported to ProcMetrics.mesa.

CMStatsImpl.mesa

This module controls all processing performed when the Stats! command is executed. This processing includes the generation of the statistical report of the data currently residing in CodeMetrics.data. Also, the data file CodeMetrics.out is created if the Backend Output flag is set in the form subwindow. All output is encoded if the boolean item Encode in in the form subwindow is set. The processing for the statistical report generation make up the majority of routines in this file, including procedures to sort and calculate overlaps of the highest metric indicators. The main data structures used in this file are described below. This module is exported to ProcMetrics.mesa.

EncodeImpl.mesa

This file contains the routines which encode identifier names in order to protect proprietary information. The encoding keys and mapping techniques used are described in this module. This module is exported to Utils.mesa.

UtilsImpl.mesa

This module contains utility routines used throughout the analyzer as needed. This module is exported to Utils.mesa.

CMFileImpl.mesa

This file contains the routines which control the intermediate metrics data file CodeMetrics.data. The data file is binary, and stores one record for each procedure analyzed. The record contains the essential metrics data and the procedure's name stored as a packed array of characters. As procedures are analyzed, their information is appended to the existing data in the data file. The Clear! command destroys all information in this file, essentially providing the user a clean slate to begin a new batch of processing. This module is exported to Utils.mesa.

ParserImpl.mesa

This module is a modified version of the parser used by the Mesa compiler. It has been modified to maintain a circular stack of identifiers used in the metrics analysis. This file controls the parsing of the input source code. All input is assumed to be syntactically correct, although the parser will detect syntax errors and abort processing. This module is exported to Parser.mesa

ScannerImpl.mesa

This file is a modified version of the lexical analyzer used by the Mesa compiler. Its routines are used in conjunction with the parser to return the individual language tokens. It has been modified very little. This module is exported to Parser.mesa.

Grammar.mesa

This file contains the Mesa grammar specification, and serves as input to the Parser Generator System (PGS) which creates the LALR tables used by the parser. Also, the routines are set up such that code is executed after each production in the procedure is reduced. This code is primarily calls to the routines in ProcMetricsImpl.mesa which gather the appropriate metrics data. As each procedure is exited, the data for that procedure is processed and the metrics data is appended to CodeMetrics.data. This module is exported to Parser.mesa.

AnalyzerImpl.mesa

This module creates the actual analyzer tool itself, sets up the variables associated with the fields of the form subwindow, and calls the appropriate routines as commands are executed. It is the module started when the analyzer is invoked. This module exports data to Utils.mesa.

Analyzer Tool Fields

The following are the fields in the form subwindow of the Code Metrics Analyzer which control its operation:

Mesa Source Files:

This field accepts the file or files of Mesa source code to be analyzed. Files may be specified using lists and/or wildcards. An extension of .mesa is assumed if none is provided.

Go!

This command item performs the source code analysis on the files specified in the Mesa Source Files field. LOC, Halstead, and McCabe metrics are calculated for each procedure in the specified files. The analysis data of each run is appended to any previous results since the last Clear! execution. Only raw data is generated with this command; no reports are created. The metrics data is stored in the binary file CodeMetrics.data.

This command executes the procedure Run in AnalyzerImpl.mesa. The procedure detaches a process to perform the analysis. The processing speed of the analysis obviously depends on the number of procedures analyzed and their relative size. Processing an average file of ten procedures can take between five seconds and two minutes.

Stats!

This command performs a statistical analysis on the metrics data generated by the source code analysis (Go! command). The report is written to the text file CodeMetrics.stats. A complete explanation of the report is given in the Mesa Code Metrics Analyzer User's Guide. Each execution of Stats! overwrites any previous report.

This command executes the procedure GenerateStats in AnalyzerImpl.mesa. The procedure detaches a process to perform the statistical report generation. The processing speed of this function depends on the amount of data in CodeMetrics.data and the percent overlap specified. To generate a report for 800 procedures with ten percent overlap takes ten to fifteen minutes.

Clear!

Execution of Clear! removes all data previously generated by the source code analysis stored in CodeMetrics.data. The data file should be saved in another file or removed from the search path prior to executing Clear! if that analysis will be needed in the future.

This command executes the procedure ClearMetData in AnalyzerImpl.mesa. The procedure detaches a process to perform the clearing function. The processing takes only a few seconds, no matter how much information is in the data file.

PercentOverlap =

This field accepts a numeric indicating the percentage of the total number of analyzed procedures that are considered when computing the overlap between the highly complex procedures. This field only affects the execution of the Stats! command. If the field contains a value that is less than one or greater than 100, a value of 100 percent is used. This field defaults to ten percent.

BackEnd Output

This boolean item only affects the execution of the Stats! command. If the flag is set when Stats! is run, a text data file is created which serves as input to a backend metrics analysis tool. The data file is called CodeMetrics.out and will rewrite any old data in the file. This field defaults to false.

Encode

Similar to the BackEnd Output field, this boolean item only affects the execution of the Stats! command. If it is set when Stats! is executed, then any module and procedure names are encoded in the statistics output report and in the backend data file. Also, this flag will cause another text file to be created, called CodeMetrics.encode, which lists the procedure names in normal and encoded form in alphabetical order by alias. This field defaults to false.

Main Data Structures

Several fundamental data structures are used to collect, process and statistically analyze the metrics data. These are described below:

METRICS COLLECTION

pm

defined:	ProcMetricsImpl.mesa
used:	Grammar.mesa, ProcMetricsImpl.mesa, HalsteadImpl.mesa, McCabeImpl.mesa

pm is a record of procedure metrics data, used later to calculate the complete metric values. It contains a procedure name, a lines of code count, two lists of unique operators and operands with associated counts, and counts of decision points encountered. The lists of operators and operands are used to calculate the Halstead metrics, and the decision counts are used for McCabe's Cyclomatic Complexity. Each pm record stores only one procedure's data.

_ pmStack _

defined: ProcMetricsImpl.mesa
used: Grammar.mesa, ProcMetricsImpl.mesa

This structure is a simple array and is used as a stack of the pm records so that nested procedures can be handled properly. As a new procedure is entered, the current pm record is pushed on the stack and a new one initialized. The full path name of a given procedure is obtained from all pm records on the pm stack.

STATISTICAL ANALYSIS

metList

defined: CMStatsImpl.mesa
used: CMStatsImpl.mesa

metList is a linked list of records containing the complete metrics values for the procedures to be analyzed. The data is read from the binary data file CodeMetrics.data. The name in each node of the list is the name of the procedure, and is encoded if the Encode boolean item is set in the form subwindow of the analyzer tool. This list serves as a organization of the metrics information prior to being statistically analyzed.

nameSort, locSort, ccSort, effortSort

defined: CMStatsImpl.mesa
used: CMStatsImpl.mesa

Each of these structures are unique Mesa sequences containing pointers to the nodes in metList. They are used to sort the metrics information by procedure name, Lines of Code, Cyclomatic Complexity, and Halstead's Effort, respectively. Using sequences of pointers allows the data to be stored only once and still allow the sorting to be completely performed on all criteria. Also, the sorting process only needs to swap pointers using this method.

olapall, olaplc, olaple, olapce

defined: CMStatsImpl.mesa
used: CMStatsImpl.mesa

Similar to the sorting sequences, these sequences are used to store pointers to the nodes in metList. The procedures indicated in each of these sequences represents the overlap calculations of the highest metrics values of all three metrics, LOC / CC, LOC / Effort, and CC / Effort, respectively. These sequences are sorted by procedure name prior to output to the statistical report file.

Appendix F. Regression Equation Analysis

The following pages contain the statistical evaluation of the regression equations considered for models to predict errors at the module level. The R-squared and Mean Squared Error (MSE) values are given in the first two matrices. The column titled PARAMETER ESTIMATE are the coefficients for the variables in the equation. The equations were deliberatly calculated without an intercept to be consistent with the semantics of the analysis. The actual and predicted values for each observation are given in the following matrix.

The equations presented in chapter 4 were chosen from this set of analyzed equations. Choices were made by considering the equations with the best evaluatory statistics and how well each predicted actual errors.

```
                                      MODEL INCLUDING CLUS V LOC           16:55 FRIDAY, DECEMBER 9, 1988

DEP VARIABLE: ERR
                                        ANALYSIS OF VARIANCE

                                         SUM OF          MEAN
                      SOURCE     DF     SQUARES        SQUARE      F VALUE      PROB>F

                      MODEL       3  25247.06404    8415.68801      15.578      0.0113
                      ERROR       4   2160.93596     540.23399
                      U TOTAL     7  27408.00000

                        ROOT MSE       23.24293     R-SQUARE        0.9212
                        DEP MEAN       43.42857     ADJ R-SQ        0.8620
                        C.V.           53.51991
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                                        PARAMETER ESTIMATES

                          PARAMETER        STANDARD      T FOR H0:                      VARIANCE
          VARIABLE  DF     ESTIMATE           ERROR     PARAMETER=0      PROB > |T|      INFLATION

          CLUS       1   0.000050015     0.000023188          2.157          0.0972     3.83382819
          V          1   0.000081612     0.000092060          0.887          0.4254     100.97383
          LOC        1  -0.004294828     0.007439021         -0.577          0.5946     108.67361

                          PREDICT      STD ERR    LOWER95%    UPPER95%    LOWER95%    UPPER95%                 STD ERR
 OBS           ACTUAL       VALUE      PREDICT        MEAN        MEAN     PREDICT     PREDICT    RESIDUAL    RESIDUAL

      1       34.0000      5.2424       7.0247    -14.2610     24.7459    -62.1723     72.6571     28.7576     22.1560
      2       31.0000     34.7324      17.8995    -14.9640     84.4288    -46.7176       116.2     -3.7324     14.8270
      3         147.0       145.6      23.2261     81.1311       210.1     54.3874       236.8      1.3838      0.8843
      4       27.0000     36.8534      13.8750     -1.6692     75.3759    -38.3021       112.0     -9.8534     18.6472
      5       53.0000     28.1042      12.1337     -5.5839     61.7922    -44.6917       100.9     24.8958     19.8244
      6             0     24.2680      17.2269    -23.5610     72.0970    -56.0562       104.6    -24.2680     15.6034
      7       12.0000      8.4979       8.6604    -15.5468     32.5426    -60.3680     77.3638      3.5021     21.5692

              STUDENT                           COOK'S
 OBS         RESIDUAL     -2-1-0 1 2               D

      1        1.2980   |       |**     |       0.056
      2       -0.2517   |       |       |       0.031
      3        1.5648   |       |***    |     563.065
      4       -0.5284   |      *|       |       0.052
      5        1.2558   |       |**     |       0.197
      6       -1.5553   |    ***|       |       0.983
      7        0.1624   |       |       |       0.001

SUM OF RESIDUALS                20.68556
SUM OF SQUARED RESIDUALS        2160.936
PREDICTED RESID SS (PRESS)      919291.7
```

```
                                   MODEL INCLUDING WOOD CLUS INFO         16:55 FRIDAY, DECEMBER 9, 1988
DEP VARIABLE: ERR
                                          ANALYSIS OF VARIANCE

                                      SUM OF           MEAN
                   SOURCE     DF     SQUARES         SQUARE        F VALUE       PROB>F

                   MODEL       3  26784.06726     8928.02242        57.237       0.0010
                   ERROR       4    623.93274      155.98318
                   U TOTAL     7  27408.00000

                      ROOT MSE      12.48932      R-SQUARE        0.9772
                      DEP MEAN      43.42857      ADJ R-SQ        0.9602
                      C.V.          28.75831
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                                          PARAMETER ESTIMATES

                         PARAMETER        STANDARD      T FOR H0:                          VARIANCE
        VARIABLE  DF      ESTIMATE           ERROR     PARAMETER=0      PROB > |T|          INFLATION

        WOOD       1    1.41578E-07     4.46785E-08          3.169          0.0339          454.57362
        CLUS       1   -0.000366815     0.000136087         -2.695          0.0543          457.35062
        INFO       1     2.47384346      0.68897862          3.591          0.0229         1.46258190

  OBS          ACTUAL     PREDICT     STD ERR    LOWER95%    UPPER95%    LOWER95%    UPPER95%                  STD ERR
                            VALUE     PREDICT        MEAN        MEAN     PREDICT     PREDICT    RESIDUAL     RESIDUAL

        1     34.0000     19.5364      5.5451      4.1410     34.9318    -18.4031     57.4759     14.4636      11.1909
        2     31.0000     44.3217      8.0555     21.9564     66.6869      3.0592     85.5842    -13.3217       9.5442
        3       147.0       146.3     12.4611       111.7       180.9     97.2950       195.3      0.7219       0.8393
        4     27.0000     34.3371      5.4056     19.3290     49.3452     -3.4469     72.1211     -7.3371      11.2589
        5     53.0000     40.5574      6.8303     21.5936     59.5212      1.0351     80.0797     12.4426      10.4561
        6           0      4.9632     10.1764    -23.2905     33.2169    -39.7655     49.6919     -4.9632       7.2405
        7     12.0000     13.8608      6.1319     -3.1639     30.8855    -24.7685     52.4902     -1.8608      10.8804

               STUDENT                         COOK'S
  OBS         RESIDUAL       -2-1-0 1 2            D

        1       1.2925      |      |**    |    0.137
        2      -1.3958      |    **|      |    0.463
        3       0.8601      |      |*     |   54.345
        4      -0.6517      |     *|      |    0.033
        5       1.1900      |      |**    |    0.201
        6      -0.6855      |     *|      |    0.309
        7      -0.1710      |      |      |    0.003

SUM OF RESIDUALS                0.1452705
SUM OF SQUARED RESIDUALS         623.9327
PREDICTED RESID SS (PRESS)       27011.93
```

```
DEP VARIABLE: ERR          MODEL INCLUDING WOOD CLUS V LOC          16:55 FRIDAY, DECEMBER 9, 1988

                                  ANALYSIS OF VARIANCE

                          SUM OF          MEAN
        SOURCE    DF     SQUARES         SQUARE        F VALUE      PROB>F

        MODEL      5  27347.80201     5469.56040       181.719      0.0055
        ERROR      2  60.19799066    30.09899533
        U TOTAL    7  27408.00000

            ROOT MSE     5.486255      R-SQUARE       0.9978
            DEP MEAN     43.42857      ADJ R-SQ       0.9923
            C.V.         12.63282
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                                  PARAMETER ESTIMATES

                  PARAMETER       STANDARD      T FOR H0:                    VARIANCE
VARIABLE  DF       ESTIMATE          ERROR     PARAMETER=0    PROB > |T|     INFLATION

WOOD       1   2.89970E-07    4.05482E-08          7.151        0.0190     1940.33762
CLUS       1  -0.000818814    0.000125458         -6.527        0.0227     2014.37172
V          1  -0.000186869    0.000045278         -4.127        0.0540      438.41026
LOC        1    0.01425800    0.003802236          3.750        0.0643      509.56633
INFO       1    2.99085997     0.76461233          3.912        0.0596     9.33505926

            PREDICT    STD ERR   LOWER95%   UPPER95%   LOWER95%   UPPER95%               STD ERR
OBS   ACTUAL  VALUE    PREDICT       MEAN       MEAN    PREDICT    PREDICT   RESIDUAL   RESIDUAL

  1  34.0000  29.1812   4.2968    10.6933    47.6692    -0.8027    59.1651     4.8188     3.4112
  2  31.0000  31.5741   4.8411    10.7441    52.4041     0.0921    63.0561    -0.5741     2.5812
  3    147.0    147.0   5.4858      123.4      170.6      113.6      180.4    -0.0316     0.0691
  4  27.0000  25.4956   3.5656    10.1540    40.8372    -2.6575    53.6486     1.5044     4.1696
  5  53.0000  54.0046   5.3163    31.1301    76.8790    21.1341    86.8751    -1.0046     1.3550
  6        0  -0.8502   5.0578   -22.6125    20.9120   -32.9567    31.2562     0.8502     2.1254
  7  12.0000  17.7141   3.4560     2.8442    32.5841   -10.1647    45.6130    -5.7141     4.2609

       STUDENT                     COOK'S
OBS   RESIDUAL     -2-1-0 1 2          D

  1     1.4126    |      |**   |     0.633
  2    -0.2224    |      |     |     0.035
  3    -0.4581    |      |     |   264.637
  4     0.3608    |      |     |     0.019
  5    -0.7414    |     *|     |     1.692
  6     0.4000    |      |     |     0.181
  7    -1.3411    |    **|     |     0.237

SUM OF RESIDUALS                 -0.151018
SUM OF SQUARED RESIDUALS          60.19799
PREDICTED RESID SS (PRESS)        40394.72
```

```
                                 MODEL INCLUDING HOOD CLUS LOC INFO          16:55 FRIDAY, DECEMBER 9, 1988
DEP VARIABLE: ERR
                                          ANALYSIS OF VARIANCE

                             SOURCE     DF        SUM OF          MEAN
                                                 SQUARES        SQUARE      F VALUE      PROB>F

                             MODEL       4   26835.12484    6708.78121       35.132      0.0075
                             ERROR       3     572.87516     190.95839
                             U TOTAL     7   27408.00000

                                ROOT MSE       13.81877       R-SQUARE       0.9791
                                DEP MEAN       43.42857       ADJ R-SQ       0.9512
                                C.V.           31.81953
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                                          PARAMETER ESTIMATES

                     PARAMETER        STANDARD       T FOR H0:                     VARIANCE
   VARIABLE  DF       ESTIMATE           ERROR     PARAMETER=0     PROB > |T|     INFLATION

   HOOD       1    1.43684E-07    4.96018E-08           2.897        0.0627      457.65867
   CLUS       1   -0.000363701    0.000150693          -2.414        0.0947      458.08232
   LOC        1   -0.001066217    0.002061981          -0.517        0.6408     23.62137713
   INFO       1     3.38007607     1.91119886           1.769        0.1751      9.19304757

OBS     ACTUAL    PREDICT   STD ERR   LOWER95%   UPPER95%   LOWER95%   UPPER95%                  STD ERR
                    VALUE   PREDICT       MEAN       MEAN    PREDICT    PREDICT    RESIDUAL    RESIDUAL

  1    34.0000    23.8256   10.3175    -9.0097    56.6609   -31.0583    78.7095     10.1744      9.1928
  2    31.0000    45.0256    9.0163    16.3312    73.7200    -7.4859    97.5370    -14.0256     10.4721
  3      147.0      146.7   13.8165      102.8      190.7    84.5512      208.9      0.2592      0.2486
  4    27.0000    31.4208    8.2208     5.2583    57.5834   -19.7511    82.5927     -4.4208     11.1075
  5    53.0000    37.9258    9.1113     8.9290    66.9225   -14.7515    90.6031     15.0742     10.3895
  6          0     2.0174   12.6188   -38.1421    42.1768   -57.5382    61.5729     -2.0174      5.6324
  7    12.0000    16.6612    8.6811   -10.9664    44.2888   -35.2750    68.5974     -4.6612     10.7516

         STUDENT                         COOK'S
OBS     RESIDUAL       -2-1-0 1 2             D

  1       1.1068     |      |**    |      0.386
  2      -1.3393     |    **|      |      0.332
  3       1.0427     |      |**    |    839.863
  4      -0.3980     |      |      |      0.022
  5       1.4509     |      |**    |      0.405
  6      -0.3582     |      |      |      0.161
  7      -0.4335     |      |      |      0.031

SUM OF RESIDUALS                0.3828711
SUM OF SQUARED RESIDUALS         572.8752
PREDICTED RESID SS (PRESS)       643812.8
```

```
                              MODEL INCLUDING LOC E CC             17:04 FRIDAY, DECEMBER 9, 1988
DEP VARIABLE: ERR
                               ANALYSIS OF VARIANCE

                                  SUM OF          MEAN
               SOURCE     DF     SQUARES        SQUARE       F VALUE        PROB>F

               MODEL       3  23772.14308    7924.04769        8.718        0.0315
               ERROR       4   3635.85692     908.96423
               U TOTAL     7  27408.00000

                   ROOT MSE      30.14903      R-SQUARE        0.8673
                   DEP MEAN      43.42857      ADJ R-SQ        0.7679
                   C.V.          69.42212
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                               PARAMETER ESTIMATES

                      PARAMETER        STANDARD      T FOR H0:                    VARIANCE
     VARIABLE  DF      ESTIMATE           ERROR    PARAMETER=0     PROB > |T|     INFLATION

     LOC        1    0.01246596      0.01119220          1.114         0.3278     146.20381
     E          1  0.0000011661     .00000124692         0.935         0.4026   13.51939732
     CC         1   -0.02369487      0.02540866         -0.933         0.4038     160.72348


                    PREDICT    STD ERR     LOWER95%    UPPER95%    LOWER95%    UPPER95%                 STD ERR
 OBS      ACTUAL      VALUE    PREDICT         MEAN        MEAN     PREDICT     PREDICT    RESIDUAL    RESIDUAL

    1    34.0000    14.7925     3.5058       5.0589     24.5261    -69.4775     99.0626     19.2075     29.9445
    2    31.0000    24.3799    21.0698     -34.1184     82.8781    -77.7413       126.5      6.6201     21.5645
    3      147.0      116.6    24.5708      48.3385       184.8      8.5734       224.5     30.4429     17.4711
    4    27.0000    43.0567    20.8323     -14.7823       100.9    -58.6883       144.8    -16.0567     21.7940
    5    53.0000    76.2887    26.8447       1.7568       150.8    -35.7903       188.4    -23.2887     13.7231
    6          0    38.6008    21.0920     -19.9591     97.1608    -63.5557       140.8    -38.6008     21.5428
    7    12.0000    14.4753     8.2125      -8.3260     37.2766    -72.2806       101.2     -2.4753     29.0089

         STUDENT                        COOK'S
 OBS    RESIDUAL     -2-1-0 1 2              D

    1     0.6414    |      |*     |      0.002
    2     0.3070    |      |      |      0.030
    3     1.7425    |      |***   |      2.002
    4    -0.7367    |     *|      |      0.165
    5    -1.6970    |   ***|      |      3.673
    6    -1.7918    |   ***|      |      1.026
    7    -0.0853    |      |      |      0.000

SUM OF RESIDUALS              -24.151
SUM OF SQUARED RESIDUALS     3635.857
PREDICTED RESID SS (PRESS)   28066.86
```

```
                                        MODEL INCLUDING LOC CC H                17:04 FRIDAY, DECEMBER 9, 1988
DEP VARIABLE: ERR
                                           ANALYSIS OF VARIANCE

                          SOURCE      DF        SUM OF           MEAN
                                               SQUARES         SQUARE       F VALUE       PROB>F

                          MODEL        3   24220.93998    8073.64666        10.133       0.0243
                          ERROR        4    3187.06002     796.76501
                          U TOTAL      7   27408.00000

                             ROOT MSE       28.22703       R-SQUARE         0.8837
                             DEP MEAN       43.42857       ADJ R-SQ         0.7965
                             C.V.           64.99644
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                                            PARAMETER ESTIMATES

                    VARIABLE  DF      PARAMETER        STANDARD      T FOR H0:                        VARIANCE
                                       ESTIMATE           ERROR     PARAMETER=0     PROB > |T|       INFLATION

                    LOC        1    0.008687947      0.01085061          0.801          0.4682       156.76633
                    CC         1    -0.05155654      0.03605974         -1.430          0.2260       369.29858
                    H          1    0.001469135     0.001175871          1.249          0.2796       331.95776

           OBS        ACTUAL     PREDICT     STD ERR    LOWER95%    UPPER95%    LOWER95%    UPPER95%                   STD ERR
                                   VALUE     PREDICT        MEAN        MEAN     PREDICT     PREDICT      RESIDUAL    RESIDUAL

             1       34.0000     -0.4814     13.5682    -38.1523     37.1896    -87.4349     86.4722      34.4814     24.7521
             2       31.0000     30.9530     21.0222    -27.4133     89.3193    -66.7631       128.7       0.0470     18.8369
             3         147.0       131.7     24.7844     62.9314       200.6     27.4509       236.0      15.2570     13.5092
             4       27.0000     35.7534     19.4164    -18.1544     89.6613    -59.3669       130.9      -8.7534     20.4882
             5       53.0000     55.9397     13.0185     19.7950     92.0843    -30.3636       142.2      -2.9397     25.0456
             6             0     38.6839     17.0037     -8.5253     85.8932    -52.8066       130.2     -38.6839     22.5308
             7       12.0000     -1.5502     17.7311    -50.7791     47.6787    -94.0991     90.9987      13.5502     21.9630

                      STUDENT                            COOK'S
           OBS       RESIDUAL     -2-1-0 1 2                D

             1         1.3931   |      |**     |         0.194
             2       .0024936   |      |       |         0.000
             3         1.1294   |      |**     |         1.431
             4        -0.4272   |      |       |         0.055
             5        -0.1174   |      |       |         0.001
             6        -1.7169   |   ***|       |         0.560
             7         0.6170   |      |*      |         0.083

SUM OF RESIDUALS                  12.9585
SUM OF SQUARED RESIDUALS          3187.06
PREDICTED RESID SS (PRESS)       10925.16
```

```
DEP VARIABLE: ERR                    MODEL INCLUDING V CC LOC N                 17:04 FRIDAY, DECEMBER 9, 1988

                                        ANALYSIS OF VARIANCE

                    SOURCE     DF      SUM OF         MEAN
                                      SQUARES        SQUARE        F VALUE       PROB>F

                    MODEL       4  24398.74449    6099.68612        6.081        0.0850
                    ERROR       3   3009.25551    1003.08517
                    U TOTAL     7  27408.00000

                        ROOT MSE     31.67152     R-SQUARE        0.8902
                        DEP MEAN     43.42857     ADJ R-SQ        0.7438
                        C.V.         72.92784
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                                        PARAMETER ESTIMATES

             VARIABLE  DF      PARAMETER        STANDARD      T FOR H0:                      VARIANCE
                                ESTIMATE           ERROR    PARAMETER=0     PROB > |T|      INFLATION

             V          1    0.000170647     0.000405317          0.421         0.7021     1054.14871
             CC         1    -0.04679016      0.04201406         -1.114         0.3466      398.21178
             LOC        1    0.009360493      0.01227904          0.762         0.5013      159.46517
             N          1    0.000266245     0.003147006          0.085         0.9379     1888.64722

   OBS          ACTUAL     PREDICT     STD ERR    LOWER95%    UPPER95%    LOWER95%    UPPER95%                STD ERR
                             VALUE     PREDICT        MEAN        MEAN     PREDICT     PREDICT    RESIDUAL    RESIDUAL

       1       34.0000      2.5959     16.8876    -51.1489     56.3407      -111.6       116.8     31.4041     26.7935
       2       31.0000     34.4507     25.0078    -45.1365       114.0    -93.9770       162.9     -3.4507     19.4344
       3         147.0       127.6     29.4918     33.7509       221.5    -10.1188       265.3     19.3915     11.5464
       4       27.0000     34.9129     21.8770    -34.7107       104.5    -87.5902       157.4     -7.9129     22.9015
       5       53.0000     67.0159     30.0913    -28.7497       162.8    -72.0186       206.1    -14.0159      9.8792
       6             0     34.7912     21.2010    -32.6810       102.3    -86.5021       156.1    -34.7912     23.5287
       7       12.0000     -0.8691     19.9605    -64.3934     62.6552      -120.0       118.3     12.8691     24.5899

   OBS         STUDENT                        COOK'S
              RESIDUAL     -2-1-0 1 2              D

       1        1.1721     |      |**   |      0.136
       2       -0.1776     |      |     |      0.013
       3        1.6794     |      |***  |      4.600
       4       -0.3455     |      |     |      0.027
       5       -1.4187     |    **|     |      4.669
       6       -1.4787     |    **|     |      0.444
       7        0.5233     |      |*    |      0.045

SUM OF RESIDUALS                3.493959
SUM OF SQUARED RESIDUALS        3009.256
PREDICTED RESID SS (PRESS)      48705.45
```

```
                                   MODEL INCLUDING LOC E CC N              17:04 FRIDAY, DECEMBER 9, 1988
DEP VARIABLE: ERR
                                     ANALYSIS OF VARIANCE

                                    SUM OF          MEAN
                SOURCE     DF      SQUARES        SQUARE       F VALUE       PROB>F

                MODEL       4  24451.53254     6112.88313        6.203       0.0828
                ERROR       3   2956.46746      985.48915
                U TOTAL     7  27408.00000

                   ROOT MSE       31.3925      R-SQUARE       0.8921
                   DEP MEAN      43.42857      ADJ R-SQ       0.7483
                   C.V.          72.28537
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                                     PARAMETER ESTIMATES

                          PARAMETER         STANDARD      T FOR H0:                      VARIANCE
        VARIABLE  DF       ESTIMATE            ERROR     PARAMETER=0    PROB > |T|      INFLATION

        LOC        1    0.009508376       0.01218605          0.780         0.4921      159.86312
        E          1    6.87218E-07    .00000142069          0.484         0.6617    16.18726759
        CC         1    -0.04907806       0.04042960         -1.214         0.3116      375.32677
        N          1    0.001188125      0.001430964          0.830         0.4673      397.46514

                       PREDICT    STD ERR    LOWER95%    UPPER95%   LOWER95%    UPPER95%                 STD ERR
   OBS       ACTUAL      VALUE    PREDICT        MEAN        MEAN    PREDICT     PREDICT    RESIDUAL    RESIDUAL

      1     34.0000     1.9431    15.9004    -48.6600     52.5462     -110.0       113.9     32.0569     27.0678
      2     31.0000    31.0939    23.3816    -43.3179       105.5   -93.4792       155.7     -0.0939     20.9474
      3       147.0      127.6    28.8486     35.8143       219.4    -8.0605       263.3     19.3751     12.3794
      4     27.0000    38.5572    22.3582    -32.5978       109.7   -84.0983       161.2    -11.5572     22.0363
      5     53.0000    68.3877    29.5273    -25.5828       162.4   -68.7683       205.5    -15.3877     10.6597
      6           0    32.0797    23.3240    -42.1489       106.3   -92.3840       156.5    -32.0797     21.0114
      7     12.0000    -0.4079    19.8605    -63.6138     62.7979     -118.6       117.8     12.4079     24.3116

             STUDENT                              COOK'S
   OBS      RESIDUAL      -2-1-0 1 2                  D

      1       1.1843    |      |**      |         0.121
      2     -.004482    |      |        |         0.000
      3       1.5651    |      |***     |         3.326
      4      -0.5245    |     *|        |         0.071
      5      -1.4435    |    **|        |         3.997
      6      -1.5268    |   ***|        |         0.718
      7       0.5104    |      |*       |         0.043

SUM OF RESIDUALS               4.721444
SUM OF SQUARED RESIDUALS       2956.467
PREDICTED RESID SS (PRESS)     41299.33
```

```
                              MODEL INCLUDING V CC LOC N E            17:04 FRIDAY, DECEMBER 9, 1988
DEP VARIABLE: ERR
                                  ANALYSIS OF VARIANCE

                                  SUM OF          MEAN
               SOURCE     DF     SQUARES        SQUARE      F VALUE      PROB>F

               MODEL       5  24454.75420   4890.95084        3.312      0.2480
               ERROR       2   2953.24580   1476.62290
               U TOTAL     7  27408.00000

                  ROOT MSE     38.42685     R-SQUARE       0.8922
                  DEP MEAN     43.42857     ADJ R-SQ       0.6229
                  C.V.         88.48288
NOTE: NO INTERCEPT TERM IS USED. R-SQUARE IS REDEFINED.

                                  PARAMETER ESTIMATES

                       PARAMETER        STANDARD     T FOR H0:                       VARIANCE
        VARIABLE  DF    ESTIMATE           ERROR   PARAMETER=0     PROB > |T|        INFLATION

        V          1  -0.000059867     0.001281682       -0.047         0.9670      7160.44980
        CC         1   -0.05004513      0.05364521       -0.953         0.4494       441.01660
        LOC        1   0.009505829      0.01491676        0.637         0.5892       159.86526
        N          1   0.001530184     0.007529678        0.203         0.8578      7344.74464
        E          1   8.82720E-07    .00000453238        0.195         0.8636       109.95424

                       PREDICT    STD ERR    LOWER95%    UPPER95%    LOWER95%    UPPER95%                   STD ERR
 OBS         ACTUAL      VALUE    PREDICT        MEAN        MEAN     PREDICT     PREDICT     RESIDUAL     RESIDUAL

     1      34.0000     1.5532    21.1775    -89.5673     92.6737      -187.2       190.3      32.4468      32.0646
     2      31.0000    29.9069    38.2745      -134.8       194.6      -203.5       263.3       1.0931       3.4190
     3        147.0      127.9    35.8143    -26.1944       282.0    -98.1123       353.9      19.0962      13.9269
     4      27.0000    39.6496    36.0008      -115.3       194.6      -186.9       266.2     -12.6496      13.4376
     5      53.0000    68.0432    36.8887    -90.6776       226.8      -161.1       297.2     -15.0432      10.7634
     6            0    31.5666    30.5909      -100.1       163.2      -179.8       242.9     -31.5666      23.2556
     7      12.0000    -0.3219    24.3804      -105.2       104.6      -196.1       195.5      12.3219      29.7022

             STUDENT                        COOK'S
 OBS        RESIDUAL     -2-1-0 1 2              D

     1        1.0119    |       |**     |    0.089
     2        0.3197    |       |       |    2.562
     3        1.3712    |       |**     |    2.487
     4       -0.9414    |      *|       |    1.272
     5       -1.3976    |     **|       |    4.589
     6       -1.3574    |     **|       |    0.638
     7        0.4148    |       |       |    0.023

SUM OF RESIDUALS               5.698523
SUM OF SQUARED RESIDUALS       2953.246
PREDICTED RESID SS (PRESS)     97693.21
```