Definition and Validation of Software Complexity Metrics for Ada

by

Bryan L. Chappell

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. Dennis G. Kafura

Dr. James D. Arthur

November, 1989

Blacksburg, Virginia

Definition and Validation of Software Complexity Metrics for Ada

by

Bryan L. Chappell

Dr. Sallie M. Henry, Chairperson

Computer Science and Applications

(ABSTRACT)

One of the major goals of software engineering is to control the development and maintenance of software products. With the growing use and importance of the Ada programming language, control over the software life cycle of Ada systems is becoming even more important. Software complexity metrics have been developed to aid software engineers in the design and development of software systems. This research defines metrics for Ada and uses an automated analysis tool to calculate them. This tool can be used by the software engineer to help maintain control over Ada software products. The validation of this tool was performed by analyzing a medium-sized commercial Ada product. The flow of control and flow of information through the use of Ada packages can be measured. The results show that software complexity metrics can be applied to Ada and produce meaningful results.

This work was supported in part by the Software Productivity Consortium Grant #041-4-42182 to Virginia Polytechnic Institute and State University.

Acknowledgements

I would like to thank my advisor, Dr. Sallie Henry, for her support throughout this research effort. Her guidance in writing this thesis has been invaluable. I would also like to thank my other committee members, Dr. Dennis Kafura and Dr. Sean Arthur.

My thanks to for letting me use him as a sounding board to get through some of the tough coding parts of the Ada Translator. His assistance in pushing the data all the way through the Software Metric Analyzer was very helpful. The analysis could not have been done without the help of computer system manager She made our micro behave like a mainframe.

Also, thanks goes out to the Software Productivity Consortium (SPC) for funding part (but unfortunately not all) of this research. The data used in my analysis was provided by Software Productivity Solutions (SPS), located in Melbourne, Florida.

Finally, I would like to thank my family and friends. I thank my parents, and for without their support and interest in my education throughout my life, I would not have pursued graduate school. Thanks to my sister and family, and and especially my nephews, ("my little buddy") and (who should have been born after I completed my degree, but I postponed my graduation until after he was born so he could see his name in print). They provided me with the daydreams that helped maintain my sanity. Thanks to for her support and caring throughout the last few months of my research, which gave me the strength to finish. Thanks go out to and without whose help quite a few courses would have been much more difficult and less enjoyable. Thanks to for putting up with a Computer Science graduate student living in their basement. Thanks to Anheuser-Busch, Domino's Pizza, and Carol Lee Donuts for

making my few relaxing hours enjoyable. Finally, thanks to countless friends who helped me survive.

Table of Contents

List of Illustrations

List of Tables

Chapter 1 Introduction

The Importance of Measuring Software

Tom DeMarco says it best: "You can't control what you can't measure" [DEMAT82]. With the increasing size and cost of software development projects, measuring software effectively is necessary in order to manage it. Measurement results in better software [GRADR87] and a better understanding of the software development process. Measuring the software development process directly is helpful and guides programmer effort. However, process metrics that have been validated and automated do not exist. Therefore, the software engineering community is left with measuring the software product. Measuring the quality of the software is what is desired. Measures of quality are faced with the same problems as measures of the software process. These measures are very difficult to quantify. Therefore, the metrics used in this study focus on the complexity of the software. There are other metrics available that concentrate on aspects of software other than complexity. Complexity metrics are used because most software engineering researchers believe that if the complexity of software can be controlled, then the resultant software will be a quality product. The motivation behind software measurement is the ability to predict certain aspects of the software development process (such as reliability and maintenance) that are believed to be affected by the characteristics being measured in the software product [CURTB83].

Software Metrics That Have Been Defined

Software complexity metrics are used to characterize certain features of the software quantitatively so that comparisons and analyses can be done among modules in a software system. Many metrics have been defined over the last ten to fifteen years [CONTS86], [HALSM77], [HENRS81b], [KOKOP88], [MCCAT76], [MCCLC78], [PIWOP82], [RAMAB88], [WOODS80]. These metrics range from a simple count of lines of code to

data and control flow measurements between modules in a programming system. The lines of code metric has existed since the beginning of computer programming. It has been used to measure programmer productivity (lines of code per unit time) and source code complexity (based on the belief that long code segments are more complex than short code segments). Most software researchers and managers now realize that complexity is not simply a function of the number of lines of code. Basing productivity and complexity solely on lines of code is inaccurate. A short recursive procedure is more complex and takes longer to code than a lengthy output procedure.

Most metrics are based on a simple count of some source code feature. Halstead developed a series of metrics called *Software Science* based on the count of operators and operands used in a program [HALSM77]. McCabe defined a metric that is the number of execution paths through a program called the *Cyclomatic Complexity* [MCCAT76]. These counts are proposed to be a measurement of program complexity. Most highly correlate to one another [ELSHJ84], [EVANW83], [HENRS81A], [LIH87], [SUNOT81]. Halstead's Software Science and McCabe's Cyclomatic Complexity are among the most popular of the metrics available.

Studies Supporting the Use of Software Metrics

Many studies have been done supporting the use of software complexity metrics. Fitzsimmons and Love [FITZA78] validated Halstead's Software Science and determined that the measures are good predictors of the number of errors, programming time, and length. Basili, Selby, and Phillips [BASIV83b] also validated Halstead's Software Science measures against several production FORTRAN projects. The metrics correlated well against developmental effort and developmental errors. Kafura and Canning [KAFUD85] validated several metrics against component errors and component coding time. Shen, Conte, and Dunsmore [SHENV83] conducted a very detailed, critical analysis of several

Software Science measures. They determined that although some of the measures are not supported theoretically, there is a lot of empirical evidence that supports their validity. Davis and LeBlanc [DAVIJ88] studied lines of code, Halstead's effort metric, McCabe's Cyclomatic Complexity, and variations of Woodfield's Review Complexity. They found that the Review Complexity measures out-performed the other metrics as a predictor for debugging time, construction time, and number of errors.

Using Metrics Throughout the Software Life Cycle

Most metrics are defined in terms of features in the source code. Therefore, measurement of these metrics can only take place during the testing and maintenance phases of a software product. Software metric researchers have realized that more time, effort, and money can be saved if the measurement is performed throughout the software development life cycle, especially during the design phase. The use of software metrics at design time can greatly affect the effort in producing a software product. These measures can be used to compare different designs and to guide the designer in the design effort. Also, since most of the cost of a software product occurs in the maintenance phase, metric analysis of software maintenance can be very beneficial.

Several studies have been done that support the use of measurement throughout the software life cycle. Ramamoorthy et al. [RAMAC85] suggest the use of metrics throughout the software life cycle, emphasizing that different metrics need to be used during different phases of software development. Kafura and Canning [KAFUD85] suggest the use of structure metrics in the design and development phases of the software life cycle. Pollock and Sheppard [POLLG87] propose a design methodology that utilizes metrics in various phases of the life cycle.

Several studies have been performed that apply the use of software complexity metrics to specific phases of software development such as design, testing and

maintenance. Metrics applied in the design of a software system can be used to determine the quality of a design, and can also be used to compare different designs for the same requirement specifications. Szulewski et al. [SZULP81] applied Halstead's Software Science to design graphs, defining a mapping between graphs and the Software Science parameters. They manually computed these measures. Card and Agresti [CARDD88a] developed a design complexity measure based on structural complexity and local complexity that estimates the overall developmental error rate. Their measurement also agrees with a subjective interpretation of design quality. Henry and Selig [HENRS90b] apply complexity metrics at design time to accurately predict the resultant source code complexity.

Using metrics during the testing of a software product helps to determine the reliability of the software. Ottenstein [OTTEL81] used Halstead's Software Science to predict the number of errors to expect during the entire software development process. Her predictions worked best for those programmers that had to correct the fewest errors. Lew, Dillon, and Forward [LEWK88] used software metrics to help improve software reliability. The software metrics were able to quantify the design and provide a guide for designing reliable software.

As previously noted, maintenance efforts cost the most of any part of the software development life cycle. Curtis et al. [CURTB79a] correlated Halstead's effort metric, McCabe's Cyclomatic Complexity and lines of code to programmer performance on two maintenance tasks. They produced empirical evidence that these metrics were related to the difficulty that programmers experienced in understanding and modifying software. Curtis, Sheppard, and Milliman [CURTB79b] extended the previous work by using improved experimental procedures. They showed that these metrics are related to the difficulty programmers experience in locating errors in code. Kafura and Reddy [KAFUD87] used

seven metrics in a software maintenance study. Among other results, they found that the growth in system complexity that was determined by the metric measurements agreed with the general character of the maintenance tasks that were performed. Other research shows that metrics applied during the coding phase can predict the maintainability of the software product [LEWIJ89], [WAKES88].

The Importance of Applying Metrics to Ada

Ada is a result of the Department of Defense's (DoD) common programming language effort. This effort was based on the premise that costs could be reduced by using a common programming language. In 1975, over 400 different programming languages were being used in DoD software [LIEBE86]. Excessive costs included providing translators, software tools and the application software for each language. Costs and programming effort that were needed for the projects themselves were being expended on the development of new programming languages [FISHD78].

There are several reasons that distinguish Ada from other programming languages. First of all is the effort put forth by the Department of Defense in its creation. Now, Ada is not just a DoD programming language. Its use has grown outside the defense community. Ada has some technical features that make it unique [SAMMJ86]:

- packages
- strong data typing
- generics
- tasking
- numeric processing
- real-time processing
- exceptions
- overloading

- separate compilation
- representation clauses

The use of packages, generics, and tasking make Ada much more difficult to measure than Pascal or C. Ada also supports and enforces several software engineering principles, including structured programming, top-down development, strong data typing, abstraction (of data and actions), information hiding and encapsulation, separation of specification from implementation, reusability, separation of logical from physical concerns, portability, modularity, readability, and verifiability [SAMMJ86].

Because of the Department of Defense's initiative and the above-mentioned features, Ada is growing in popularity. As of July 1989, 52 vendors have produced 257 validated Ada compilers [ADAIC89]. For Department of Defense contracts, not only is Ada the required language, but DOD STD 2167 A requires that code be evaluated for maintainability, along with design and coding standards [DOD88]. More importantly, Ada will be used for some time to come, as it is proving useful in improving productivity and reliability [MYERW87], [MYERW88].

Certainly, metric analysis of Ada programming projects contributes in determining reliability and productivity. Metric analysis of Ada code can also guide programmers in their testing and maintenance duties.

The Need for Automatable Metric Analysis Tools

It is obvious from the preceding sections that software metrics can be a useful guide in the life of a software product. In order to facilitate the use of software metrics, automatable metric analysis tools must be developed. It would be unreasonable to consider measuring software manually due to the size of most software systems. Also, automatable measuring is more accurate and consistent. Gilb [GILBT77] stresses the importance of an automated metric analysis tool. He cites a TRW study where metrics-guided testing is half

the cost of conventional testing. Grady [GRADR87] also mentions the need for tools to collect metrics.

Several metric analysis tools are available. The Software Complexity Metrics Tool [COOKC87] collects metrics for C programs. The metrics that are calculated are lines of code, Halstead's Software Science, McCabe's Cyclomatic Complexity, data structure metrics, and information flow metrics based on function calls. The Software Metrics Data Collection system was developed "to help researchers investigate the practical applications of new and existing software metrics" [YUT88]. Process metrics are entered into the system from report forms and the SMDC calculates some product metrics. The product metrics collected are lines of code, Halstead's Software Science, and McCabe's Cyclomatic Complexity. PC-Metric is another tool that collects these same three code metrics for C, Pascal, or Modula-2 on an IBM PC [MCAUD88].

The Ada Measurement and Analysis Tool (ADAMAT) is a metric collection tool for Ada developed by Dynamics Research Corporation. ADAMAT computes more than 150 different code metrics for Ada. These code metrics include several measures of lines of code and various ways to count tokens. The Complexity Measures Tool (CMT) is another tool that collects metrics for Ada. It was developed by EVB Software Engineering, Inc., and computes five different measures of lines of code, Halstead's Software Science, and McCabe's Cyclomatic Complexity. The fact that these code metrics highly correlate was mentioned previously. These "powerful" tools do nothing but provide several different measurements of the same code features. Since all of these measures are based entirely on the source code, these Ada tools can only be used in the testing and maintenance phases of software development.

This research provides a software metric tool for Ada that can be used throughout the entire software development life cycle by providing structure metrics, in addition to

code metrics. The structure metrics are based on the structure of the code, which can be seen early in the software design. Therefore, if Ada is used as a program design language, the structure metrics generated by this tool still yield a good measure of the design, regardless of whether or not the design is written with a low level or a high level of refinement.

It is important to note that there is not a single metric that measures all aspects of a program's complexity at all phases of development. Some metrics perform better at design time. Other metrics perform better during maintenance. The metrics that perform best for one organization using a certain development environment and certain design methodologies are most likely different from the metrics that perform best for other organizations using different development environments and design methodologies. Also, metrics need to be tailored to the type of software that is being analyzed. Basili and Selby [BASIV85] describe how to determine an environment's "characteristic software metric set." This set of metrics are the ones that perform the best in a particular software development/maintenance environment.

The Use of Ada as a Program Design Language

There is a benefit of having a design language that is a subset of a programming language while also having design language features. Many researchers have proposed the use of Ada as a program design language (PDL) [BONDR84], [CHASA82], [GABBE83], [GORDM83], [HOWEB84], [LINDL83], [SAMMJ82]. Each of the military services under the Department of Defense is currently issuing requests for proposals (RFPs) which require the use of an Ada-based PDL [HOWEB84].

The Software Metric Analyzer that was developed as part of this research can be used to analyze Ada code written with any level of refinement. Therefore, the analyzer supports the use of Ada as a PDL. Design quality can be determined and different designs

for the same requirement specifications can be compared. Henry and Selig [HENRS90b] have been able to predict the complexity of the resultant source code from measurement of designs written in an Ada-like PDL. Many metrics are calculated by the Software Metric Analyzer, including the code and structure metrics that have already been mentioned. The structure metrics produce more relevant information than code metrics when analyzing design code. This is because most of the details of the design code concerns the system's hierarchy and calling structure. The Software Metric Analyzer should be of great benefit as a life cycle support tool.

Conclusion

The need for measuring software, and specifically the need for measuring software written in Ada, has been discussed. This research effort concentrates on defining metrics for Ada and developing an automated metric analysis tool. This tool can be used throughout the software development life cycle. It also supports the use of Ada as a program design language. Chapter 2 defines the metrics used in this research and how they are applied to Ada. Chapter 3 discusses the Software Metric Analyzer, the tool used to calculate Ada metrics. Chapter 4 presents an analysis of an Ada programming system. The conclusions of this research are presented in Chapter 5.

Chapter 2 Metrics for Ada

Introduction

Once it is realized that software needs to be measured, it is important that the measuring process be automatable. It would be unreasonable to consider measuring software manually due to the size of most software systems. Also, automatable measuring is more accurate and consistent than manual, qualitative measurement [BERAE83]. Two examples of qualitative metrics are coupling and cohesion. These concepts are meaningful and attempts have been made to measure them, but without much success. Qualitative metrics are not considered in this study. Automatable metric analysis of Ada is the main thrust of this research. Therefore, this work concentrates on only those metrics that are quantitative and automatable.

The available quantitative metrics can be grouped into three rather broad categories. These categories are *code metrics*, *structure metrics* and *hybrid metrics*. Code metrics produce a count of some feature of the source code. Structure metrics attempt to measure the logic and interconnectivity of the source code. Hybrid metrics combine one or more code metrics with one or more structure metrics.

Of the three categories, code metrics are the easiest to calculate. However, since structure and hybrid metrics measure different aspects of the original source code, they are very important and worth the effort of the more complex computations [HENRS81a], [HENRS90b].

This chapter discusses the code metrics, structure metrics, and hybrid metrics that are used in this research. Of the many metrics available, only nine are incorporated in this study. These nine metrics are used because they are popular and have been validated [HENRS81b], [HENRS90a], [HENRS90b], [KAFUD85], [KAFUD87], [LEWIJ89], [WAKES88]. Towards the end of this chapter, *interface metrics* and the need for dynamic

metrics are discussed. Interface metrics attempt to more fully measure the interface complexities between communicating modules by taking into consideration variable types and how variables are used within a module.

Code Metrics

The code metrics calculated in this study are now defined. Many code metrics have been proposed and defined but only those used in this study are discussed. The code metrics defined here can be grouped into three categories of metrics. They are Size, Software Science Composite, and Logic Structure metrics.

Lines of Code

The lines of code (LOC) metric is perhaps the most widely used software metric. It is a size metric. However, since it is widely used does not mean that it is the most useful metric in determining program complexity. On the surface, lines of code is the easiest measurement to compute since it is available in almost any programming environment. A compiler, an editor or a utility provided by an operating system can give the number of lines of code within a file. This measure is readily accessible to anyone, but the definition that is usually used for a line of code by these methods is a total count of the lines in the program file, including blank lines and comments. Many metrics researchers have realized that blank lines and comments should not be included in the lines of code count since they do not contribute to the complexity of a piece of software (although, when used properly they can contribute to its understanding). The question that needs to be answered is what exactly is a line of code. By having a concise definition, lines of code may be measured accurately and consistently.

There are many different definitions of a line of code. As previously mentioned, blank lines and comments may or may not be counted. Other definitions count executable statements or, in a language such as Ada, semicolons. The definition for a line of code that

was used in this study follows that suggested by Conte et al., as the prevailing definition in use today [CONTS86]:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.

Firesmith suggests that before computing the number of lines of code, the program should be run through a code formatter or pretty printer [FIRED88]. This is a good suggestion since virtually any definition for a line of code depends on coding style. The code formatter would impose a single coding standard that would allow comparisons to be made among subprograms in a programming system.

Another point brought up by Firesmith is how to count generic units and generic instantiations. The alternatives presented include counting the generic unit and only one line for each generic instantiation, counting the generic unit and only the lines of the first generic instantiation, and counting the generic unit and all the lines of all generic instantiations. In this study, the length of the generic unit is used for the length of each generic instance.

Halstead's Software Science

Some of the problems with defining a line of code is that many statements can be placed on one line of text and a single statement can be spread across several lines of text. Halstead's Software Science is a means to measure a program without having this variation make a difference. Halstead defined several metrics based on token counts [HALSM77]. A token is a basic syntactic unit that is the smallest character grouping distinguishable by a compiler. The metrics defined for Software Science are all based on the number of operators and operands within a program. Therefore, tokens are grouped as either operators or operands. Measuring a program based on token counts is not affected by the

placement of statements on one line or across many lines. Regardless of the number of lines per statement or the number of statements per line, the number of tokens remains the same (for a given statement sequence).

There are four basic measures of tokens that are the basis for all of the Software Science metrics. These measures are:

n_1 number of unique or distinct operators

n_2 number of unique or distinct operands

N_1 total occurrences of all the operators

N_2 total occurrences of all the operands

Although counting tokens and grouping them as either operators or operands seems straightforward, it is not. For each programming language, there are different keywords and symbols that must be defined to be either an operator or an operand. Halstead defined his metrics before Ada was developed. Therefore, there is no universally accepted set of definitions of operators and operands for Ada. Berard in [BERAE83] and [BERAE84] offers a light discussion of Halstead's Software Science applied to Ada. Mehndiratta and Grover [MEHNB87] also apply Software Science to Ada. Part of this research involved defining what the operators and operands are for Ada. While mathematical expressions are easy to divide into operators and operands, other Ada constructs are not. The approach used for defining operators and operands for Ada was to define all the operators first and define everything else to be an operand. Overloaded operators and how they are treated with regards to Software Science are discussed in the next chapter.

There are a few points that need to be discussed about how these operators are defined. The first point considers multiple tokens that are counted as a single operator. This is because one of these tokens can not be used by itself. For example, `BEGIN END` (two tokens separated by a sequence of statements) is generally considered to be only one

operator. Another point that needs to be made is that a single token can be counted as one of several operators. A "-" is the first example that comes to mind. In its unary usage it is counted as a negation operator. In its binary usage it is counted as a subtraction operator. Another example is the semicolon. A semicolon can be counted as one of two different operators, depending upon its use. A semicolon is either a statement terminator operator (i.e., when the semicolon is at the end of a statement) or a special item separator (e.g., appearing between parameter specifications within a subprogram declaration). A complete list of operators defined for Ada is in Appendix A.

Of the many metrics defined by Halstead, this research concentrates on his N, V, and E. Halstead defined the *length* of a program, N, as the sum of N_1 and N_2. Length, like lines of code, is also a size metric. N is the total number of tokens used in the program. Another metric defined by Halstead is *vocabulary*. The vocabulary n is defined as the sum of n_1 and n_2. The term vocabulary is used since the program measured can be constructed by using only n operators and operands as a programming vocabulary [CONTS86]. Another size metric defined by Halstead is the *volume* V, where

$$V = N \log_2 n$$

The unit of measurement for V is a bit. Since n operators and operands are used in the program, they could be encoded using $\log_2 n$ bits. Each of the N tokens could then be encoded by a bit code of length $\log_2 n$. Therefore, a program could be encoded as a string of $N \log_2 n$ bits. Note that this is the volume for a binary translation of the original program, not the size of the compiled version of the original program.

Halstead's programming *effort*, E, is not a size metric but a Software Science Composite metric, following the terminology presented by Conte et al. [CONTS86]. Halstead proposed that the effort required to program an algorithm would increase with that

program's volume and *difficulty*. He defined difficulty as the reciprocal of program *level*, *L*.

The program level is dependant on the volume and the *potential volume* of a program. The potential volume is the volume that would be calculated for the most efficient way to code an algorithm. Halstead states that this most efficient method would be in the form of a procedure call to a previously defined subroutine. Therefore, the operator and operand count would be a count of those involved in invoking that subroutine. However, computing the potential volume in general is very hard to do, so Halstead estimated the program level by

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

As previously stated, effort depends on volume and difficulty, the reciprocal of level. Therefore, effort is defined as

$$E = \frac{n_1 N_2 N \log_2 n}{2 n_2}$$

and is the total number of elementary mental discriminations required to generate a given program module.

McCabe's Cyclomatic Complexity

McCabe defines a metric that is based on the logic structure of a program and has its basis in graph theory [MCCAT76]. McCabe's Cyclomatic Complexity metric measures the number of basic paths through a program. When all of these basic paths are taken in combination, every possible path through the program will be generated. McCabe believes that the Cyclomatic Complexity is related to the testability and maintainability of a program. The Cyclomatic Complexity is defined similarly to the cyclomatic number of a directed graph *G*:

$$V(G) = e - n + p$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components in the graph. A procedure is considered to be a connected component. The cyclomatic complexity has a branch from the exit node to the entry node of the graph. Therefore, the cyclomatic complexity is

$$V(G) = e - n + 2$$

The Cyclomatic Complexity can be calculated by defining a node to be a block of code and an edge to be a line connecting blocks of code (indicating control flow). According to McCabe, 10 seems "like a reasonable, but not magical, upper limit" for $V(G)$.

Developing a software tool to construct a flowgraph representation of a procedure would be difficult. Fortunately, construction of such a tool is not necessary since the cyclomatic complexity only depends on the number of edges and nodes in the graph, not their arrangement. There is a relationship between edges and nodes. Note that a node generally has only one edge leading out of it, unless it is the terminal node of the graph (in which case it has no edges leading out) or a conditional node (in which case there will be two edges leading out). Therefore, $e = n - 1$ for a graph with no conditionals. The "- 1" accounts for the terminal node with no edge leading out. Each conditional contributes two to the number of edges, one that is included in n and an extra one. Therefore, $e = n - 1 + c$ where c is the number of conditionals. By algebraic manipulation, $V(G) = e - n + 2 = c + 1$. Therefore, the Cyclomatic Complexity can be computed by simply counting the number of simple conditions within each decision and adding one.

For Ada, the Cyclomatic Complexity is calculated by accumulating the number of occurrences of `IF`, `ELSIF`, `FOR` and `WHILE` statements and `CASE` labels, as well as the binary boolean connectives. It is necessary to count boolean connectives since `IF C1 AND C2 THEN` would have to be written as `IF C1 THEN IF C2 THEN` without the connective. For loop statements, the conditionals associated with each `EXIT` statement are also counted.

See Appendix B for a complete list of the conditions in Ada which contribute to the Cyclomatic Complexity.

Structure Metrics

In this section the structure metrics implemented for this research are defined. Structure metrics attempt to measure the logic and control flow of a program. These metrics take into account the interconnectivity among program modules that code metrics ignore. Interaction between modules (through subprogram and task invocations) contributes to the overall complexity of the modules. The structure metrics that this study incorporates are Henry and Kafura's Information Flow metric [HENRS81b] and McClure's Invocation Complexity [MCCLC78].

Henry and Kafura's Information Flow Metric

Henry and Kafura's Information Flow metric is based on the information flow connections between a module and its environment. In Ada, subprograms, packages and tasks are considered to be modules.

There are several ways that information can flow between modules. Henry and Kafura define and name the following flows of information [HENRS81b]:

> There is a global flow of information from module *A* to module *B* through a global data structure *D* if *A* deposits information into *D* and *B* retrieves information from *D*.
>
> There is a local flow of information from module *A* to module *B* if one or more of the following conditions hold:
>
> 1) if *A* calls *B*,
> 2) if *B* calls *A* and *A* returns a value to *B*, which *B* subsequently utilizes, or
> 3) if *C* calls both *A* and *B* passing an output value from *A* to *B*.
>
> In Ada, a global flow of information is defined as follows:

> There is a global flow of information from module *A* to module *B* through a data structure *D* inside the specification of package *P* if *A* deposits information into *D* and *B* retrieves information from *D*.

Henry and Kafura then define two other terms based on the previous definitions:

> The *fan-in* of procedure *A* is the number of local flows into procedure *A* plus the number of data structures from which procedure *A* retrieves information.

> The *fan-out* of procedure *A* is the number of local flows from procedure *A* plus the number of data structures which procedure *A* updates.

Henry and Kafura define the complexity of a procedure *P* as $(fan\text{-}in \times fan\text{-}out)^2$. The term *fan-in* x *fan-out* is the total number of combinations of an input source to an output destination. This term is squared because Henry and Kafura believe that the relationship between the information flows of a procedure and its environment is more than linear.

McClure's Module Invocation Complexity

McClure wants to be able to measure and control complexity in well-structured programs. McClure defines program complexity as follows [MCCLC78]:

> Program complexity is an indicator of program readability. . . . Program complexity is introduced by the difficulty of the programming problem and the size of the solution program. It is a function of the number of possible execution paths in the program and the difficulty of determining the path for an arbitrary set of input data.

McClure discusses McCabe's Cyclomatic Complexity and presents a few problems. McClure believes that predicates contribute differing amounts of complexity to a module, while McCabe believes that all predicates contribute the same amount of complexity. As an example, see Figure 1. Both procedure *A* and *B* have the same Cyclomatic Complexity ($V(G) = 4$), but McClure would assert that procedure *B* should have a greater complexity

```
Procedure A (param : integer) is
Begin
   IF param = 1 THEN
      C;
   ELSIF param = 2 THEN
      D;
   ELSIF param = 3 THEN
      E;
   ELSE
      F;
End A;

Procedure B (param1, param2 : integer) is
   var : integer;
Begin
   var := Function_X(param1);
   IF (param1 = 1) OR (var = 0) THEN
      C;
   ELSIF (param2 = 2) THEN
      D;
   ELSE
      E;
End B;
```

Figure 1. Difference Between McCabe's Complexity and McClure's Complexity

than procedure *A*. This is because understanding procedure *B* requires knowledge of three variables, while understanding procedure *A* only requires knowledge of one variable.

McClure believes that a more complete technique for measuring complexity "must include an examination of the number of possible execution paths and the control structures and the variables used to direct path selection" [MCCLC78]. Therefore, in the previous example McClure would compute a complexity greater than McCabe's Cyclomatic Complexity for procedure *B*.

As stated previously, McClure's interest is the complexity of well-structured programs. In order to facilitate the complexity calculations, McClure defines the Program Control Hierarchical System (PCHS) which governs module invocations [MCCLC78]. It is a triple denoted by $PCHS = (P, \gamma, f)$ where

P is a finite set of well-structured modules, i.e., $P = \{p_1, ..., p_n\}$,

γ is the root module,

$f: P \rightarrow X$ is the invoking function such that $P \supset X$, and

$X = f(p_i)$, where $X = \{x_1, ..., x_k\}$ for $0 \leq k < n$, $1 \leq i \leq n$,$1 \leq j \leq k$.

All of this means that module p_i invokes each $x_j \in X$, for $1 \leq j \leq k$. Module p_i is the *direct ancestor* of x_j, and x_j is a *direct descendant* of p_i.

For each $p_i \in P$, McClure defines the following sets [MCCLC78]:

F_{p_i} denotes the set of direct ancestors of p_i

G_{p_i} denotes the set of direct descendants of p_i

H_{p_i} denotes the set of ancestors of p_i

L_{p_i} denotes the set of descendants of p_i.

McClure then defines a *common module*. If $\left|F_{p_i}\right| > 1$, p_i has more than one direct ancestor and is called a common module.

The PCHS can be represented graphically by a hierarchical structure where ancestor modules are listed above their descendants. A PCHS *sub-hierarchy S* is a set of modules that have exactly one common ancestor, $s_i \ni s_i \in S$. This common ancestor, s_i, is the *local root* of S.

With these preliminary definitions at hand, McClure postulates that the most likely source of complexity in a well-structured program is the use of *control variables*, i.e., variables that are used to direct program path selection. All of the procedure invocations in procedures A and B in Figure 1 are governed by control variables.

With the importance of control variables in mind, McClure then defines the complexity of a control variable. The control variable complexity is a function of the set of modules which access the control variable and the PCHS invocation relations of these modules. This is intuitive, as it is more difficult to understand how a control variable is used and modified when many different modules access the control variable.

To be precise about control variable accesses, McClure states that a module accesses a control variable if it is referenced in a conditional expression or modified within the module. A_v denotes the set of program modules which access control variable v. A_v can then be thought of as being composed of two disjoint sets, R_v and M_v, where R_v is the set of modules that only reference the control variable v, and M_v is the set of modules where the control variable v is modified. McClure also defines another set of modules denoted by E_v, as the set of modules whose invocation is dependent upon control variable v.

The *owner* of a control variable, α_v, is the local root of the smallest PCHS sub-hierarchy which contains all members of the set A_v. McClure then refines the concept of an owner into *degrees of ownership* for control variable v:

Degree 1	The value of control variable v is modified exclusively in α_v or never modified in the program.
Degree 2	The value of control variable v is modified in α_v and in at least one descendant of α_v.
Degree 3	The value of control variable v is strictly referenced in α_v and modified in at least one descendant of α_v.
Degree 4	The value of control variable v is not accessed in α_v and is modified in at least one descendent of α_v.

By using all of the preceding preliminary definitions and notation, the control variable complexity function for control variable v, $C(v)$, is defined as

$$C(v) = \frac{D(v) \times I(v)}{n}$$

where

$D(v)$ is the degree of ownership for control variable v,

$I(v) = q_v + u_v + w_v + t_v$, where

$q_v = \left| M_v \cap E_v \right|$, the invocation complexity,

$u_v = \left| U_v \right|$ where $U_v = \{p_j \mid p_j \in R_v \text{ and } \exists\, p_k \in L_{p_j} \ni p_k \in M_v\}$, the descendent complexity,

$w_v = \left| W_v \right|$ where $W_v = \{p_i \mid p_i \in A_v \text{ and } \exists\, p_j \in M_v \ni p_j \notin H_{p_j} \text{ and } p_j \text{ is listed above } p_i \text{ in the PCHS structure}\}$, the path complexity,

$t_v = \left| T_v \right|$ where $T_v = \{p_j \mid p_j \in A_v \text{ and } \exists\, p_k \in F_{p_j} \ni p_k \notin A_v\}$, the intermittent complexity, and

n is the number of unique modules in the PCHS.

$D(v)$ measures access and modification complexity. $I(v)$ measures the module interaction via v. Control variable complexities range from zero to eight.

McClure defines several sources that contribute to a module's complexity. Module complexity attempts to measure the difficulty of understanding how program control is

passed between modules in a well-structured program. As discussed previously, control variables contribute to the complexity. Also, the control structures (e.g., `CASE`, `WHILE LOOP`, `FOR LOOP`, `IF`, or `ELSIF`) that are used in the module's invocation contribute to its complexity. The commonality of the module contributes to its complexity as well.

One last definition needs to be presented before the module complexity function can be explained. An *invocation control variable set* is the set of control variables that governs a particular invocation of a module. Note that a module can have more than one invocation control variable set if it is conditionally invoked in multiple places within the program.

McClure's Module Invocation Complexity function, denoted by $M(p)$, is used to determine the complexity of invoking module p in a well-structured program. $M(p)$ is defined as

$M(p) = [f_p \times X(p)] + [g_p \times Y(p)]$, where

$f_p = |F_p|$,

$X(p)$ measures the complexity of the control structures and control variables used to invoke module p,

$g_p = |G_p|$, and

$Y(p)$ measures the complexity of the control structures and control variables used by module p in invoking its direct descendants.

Let x be the number of invocation control variable sets used in the invocation of module p. Let e be the number of control variables in the j[th] invocation control variable set. Let v_{ji} be the i[th] control variable in the j[th] invocation control variable set. Let $b_j = 1$ if the j[th] invocation is within a selection structure (e.g., `IF`, `ELSIF`, or `CASE`). Let $b_j = 2$ if the j[th] invocation is within a repetition structure (e.g., `FOR LOOP` or `WHILE LOOP`). If $x = 0$, $X(p) = 0$; otherwise

$$X(p) = \frac{\sum_{j=1}^{x}\left(b_j \times \sum_{i=1}^{e} C(v_{ji})\right)}{x}$$

$Y(p)$ is defined similarly. Let y be the number of invocation control variable sets referenced by module p to invoke its direct descendants. Let k be the number of control variables in the j[th] invocation control variable set. If $y = 0$, $Y(p) = 0$; otherwise

$$Y(p) = \frac{\sum_{j=1}^{y}\left(b_j \times \sum_{i=1}^{k} C(v_{ji})\right)}{y}$$

The values for McClure's Module Invocation Complexity range from 0 to $16sn$, where s is the total number of control variables and n is the number of unique modules in the program.

Obviously, it is very difficult to calculate McClure's metric. However, due to its basis in the environment under which modules are invoked, it is quite useful and more meaningful than code metrics. According to McClure, the number of possible execution paths contributes to program complexity. Her module complexity reflects the differences between different invocations of a module by not treating each invocation the same as the next. Her idea of control variables and having their complexity contribute to a module's complexity is what makes her metric more accurate at determining complexity than just simply counting the number of module invocations.

Hybrid Metrics

As previously stated, hybrid metrics are a combination of one or more code metrics with one or more structure metrics. The hybrid metrics that this study incorporates are the hybrid form of Henry and Kafura's Information Flow metric [HENRS81b] and Woodfield's Review Complexity metric [WOODS80].

Henry and Kafura's Information Flow Metric

Henry and Kafura's Information Flow metric can also be used as a hybrid metric. This is the approach that Henry and Kafura used in a study of the UNIX operating system [HENRS81b]. The complexity of a procedure P is $C_{ip} \times (\textit{fan-in} \times \textit{fan-out})^2$. C_{ip} is the internal complexity of procedure P and may be any code metric. (Henry and Kafura used lines of code as the internal complexity in their study, but suggested that any code metric may be used.) The definitions for *fan-in* and *fan-out* are the same as they are in the definition of the structure form of the Information Flow metric.

Woodfield's Review Complexity

Woodfield's Review Complexity metric attempts to measure effort in terms of the time required to understand a module [WOODS80]. Recall that this is similar to Halstead's effort metric, which measures the number of elementary mental discriminations necessary to program a module. In fact, Halstead's effort is the code metric used by Woodfield, but as with Henry and Kafura's Information Flow metric (the hybrid form), any code metric may be used. Where Halstead's effort measures the complexity involved in creating a programming module, Woodfield's Review Complexity measures the complexity involved in understanding a programming module that has been previously written (presumably by another programmer).

Woodfield believes that a module must be reviewed several times before it can be completely understood. The number of times that a module needs to be reviewed depends

on the number of flows into that module. Woodfield defines a *fan-in* as a flow into module *A* when either of the following occur:

Any module invokes module *A*, or

A data structure *D* is modified by module *A* and referenced elsewhere.

Woodfield's Review Complexity for module *A* is defined as follows [WOODS80]:

$$C_i \times \sum_{k=2}^{fan\text{-}in - 1} RC^{k-1}$$

where C_i is the internal complexity of module *A* and *RC* is a review constant. The review constant used by Woodfield is $\frac{2}{3}$, which was previously suggested by Halstead [HALSM77].

Note that the time involved to review a module decreases exponentially each time it is reviewed. Woodfield's model accurately reflects this. Also notice that Woodfield's *fan-in* is different from the *fan-in* used by Henry and Kafura in their Information Flow metric.

Proposed Ada-Specific Metrics

Gannon, Katz, and Basili [GANNJ86] propose a way of measuring the complexity of Ada packages. They define a component access metric and a package visibility metric. The component access metric attempts to quantify the complexity involved in referencing entities (constants, types, objects, subprogram declarations, etc.) that are declared in package specifications but referenced outside of that package. Referencing entities in this way definitely contributes not only to the complexity of the package in question, but also to the program unit that contains the reference. Both McClure's Module Invocation Complexity and Henry and Kafura's Information Flow Metric measure this complexity.

The package visibility metric determines if `WITH` clauses have been placed at the lowest possible level within the Ada code. By using subunits, `WITH` clauses may be placed such that they are only in effect when needed. Program units that do not need information

contained in certain package specifications do not have access to that information when they are outside the scope of the corresponding WITH clauses. This is a good design criterion and most likely helps to make maintenance tasks easier, but it is not a measure of program complexity and therefore not included in this study.

Shatz [SHATS88] discusses some preliminary ideas involving the complexity of tasks and task interaction, specifically concentrating on task communication complexity. His complexity is based on the maximum number of concurrently active rendezvous determined by a static analysis. This research also measures task communication complexity statically, but by using the measures proposed by McClure and Henry and Kafura. Each task entry is treated as a unit of analysis. The communication paths for these metrics are determined by the accept and entry call statements. The details of how tasks are measured are discussed in the next chapter.

Interface Metrics

Although McClure's Module Invocation Complexity and Henry and Kafura's Information Flow metric both measure a subprogram's interfaces, many details that are in the source code do not affect their complexity calculations. Interface metrics attempt to more fully measure the interfaces among subprograms by incorporating more details into complexity measurement. These details are that variables of different types have a different complexity inherent in their types (e.g., an integer variable is less complex than a record variable), operations vary in complexity (e.g., addition is less complex than division), and conditionals vary in complexity (e.g., a FOR loop is different in complexity than an IF statement). Also, how variables are referenced (either they are read from or written to, or both) contributes to complexity.

All of these facts combine to yield a better measure of the complexity of subprograms for several reasons. As previously mentioned, more details are incorporated

in the calculation of interface metrics. Also, since most errors involve interfaces, it is felt that interface metrics may provide a more accurate complexity measure.

Interface metrics were defined with the Ada programming language in mind. With the possibilities available through the use of packages and generic units, program complexity needs to be viewed in a new way [GANNJ86], [SHATS88].

Part of this research effort deals with calculating some of the information necessary to derive the interface metrics. The interface metrics themselves are actually computed in another phase of research, different from this one. The details of the interface metrics that directly pertain to this research are discussed in the next chapter.

The Need for Dynamic Metrics

All of the metrics discussed so far have dealt with a program's source code in a static way. Without knowing the values of all input data, there is no way of knowing the execution path of any non-trivial programming system. Complexity analysis is done to all program modules, without regard to how many times each module is executed.

Since one of the goals of software metrics research is not only to calculate complexity but also to use this complexity to help guide programmers in their testing efforts to reduce errors, the dynamic behavior of a program needs to be measured. This is true of any programming system written in any language, but with the advent of Ada and its tasking construct, dynamic metrics now really need to be pursued. Tasking introduces complexity of its own that cannot be measured statically. Although this research effort does not produce any dynamic metrics, it is hoped that dynamic metrics for Ada (and other languages) will be forthcoming.

Conclusion

Several metrics and how they are calculated for Ada have been presented. The metrics measure different parts of the source code, some dealing with size and others

dealing with interactions between program modules. The next chapter discusses the tool that was developed to calculate these metrics.

Chapter 3 The Software Metric Analyzer

Introduction

The need for automatable metric analysis is discussed in Chapter 2. Towards that end, the software metric research group at Virginia Tech has developed a Software Metric Analyzer. A diagram of the Software Metric Analyzer is in Figure 2.

Phase One of the Software Metric Analyzer is the language-dependant translator. In addition to Ada, translators exist for Pascal, C, FORTRAN, THLL, and ADLIF. THLL is a language used by the United States Navy and ADLIF is an Ada-like Design Language based on Information Flow [HENRS90b]. Phase Two is the Relation Manager. Phase Three is the Metric Generator.

The Software Metric Analyzer has been developed over the last several years, with the different language-dependent translators being written and additions being made in all aspects of metric generation. All of these different language-dependent translators do the same thing for a different source language: produce code metrics and translate their source language into a common object language. Therefore, although several different translators exist, only one Relation Manager and only one Metric Generator are necessary.

Phase One

The most time-consuming part of this research was in developing the Ada Translator. The Ada Translator takes as input Ada code, as well as information on the pre-defined language environment (e.g., the package STANDARD). There are two outputs from the Ada Translator. They are code metrics and *relation language code*. Note from Figure 2 that the Ada Translator is the only part of the analyzer that has access to the Ada code. Therefore, all the code metrics are produced by the Ada Translator. The Ada Translator is written in Lex [LESKM75], YACC [JOHNS75], and C under the UNIX operating system.

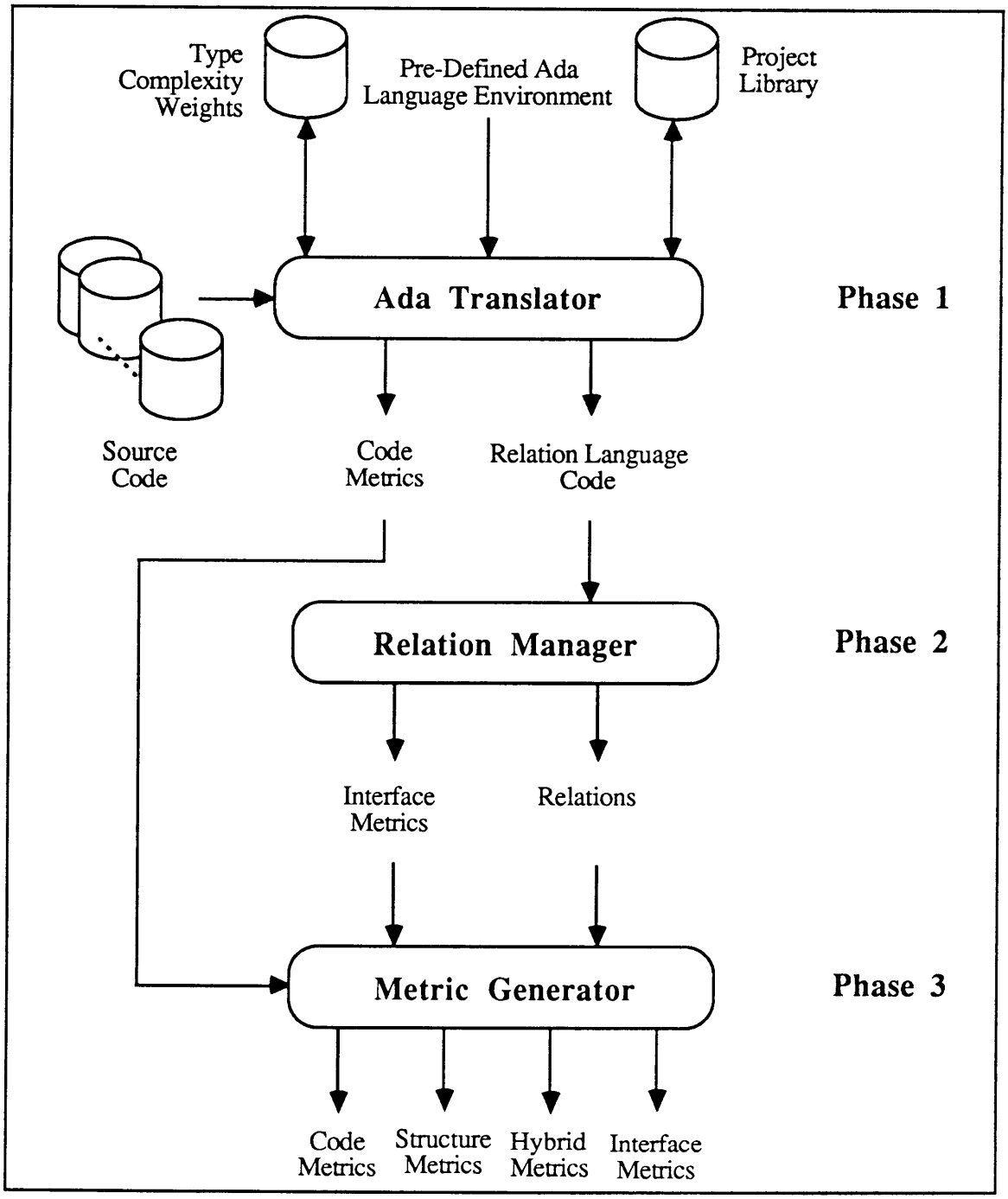

Figure 2. The Software Metric Analyzer

Both the lexical analyzer and the parser [FISHG84] are circulating in the public domain. The entire Ada Translator is over 28,000 textual lines of code in length.

There are two major differences between the Ada Translator and an Ada compiler. The first difference is that the Ada Translator, as part of an analysis tool, assumes all Ada source code input is syntactically correct and that files are submitted in the correct order. The order that is necessary is identical to the compilation order that an Ada compiler enforces. Any references in a context clause to other compilation units must be to a previously compiled library unit. The other difference between the Ada Translator and a compiler is that no executable code is generated. The relation language code that is output is necessary to compute the structure and hybrid metrics in later phases of the analyzer. It is not executable, nor will it be translated into an executable form in a later phase of the analyzer.

The Project Library

The Ada Translator accepts and processes full Ada. With that in mind, it has many implementation details similar to an Ada compiler. The Ada Translator accepts any number of source code files, each with any number of compilation units within them. Therefore, like an Ada compiler maintains a project library to keep information at hand about previously compiled library units, the Ada Translator needs to keep similar information in its own project library. (The project library is created and maintained by the Ada Translator; it is not dependent on any information from an Ada compiler.) As an example, the name of each compilation unit, its parameters (for subprograms) and its declarations (for packages) are contained in the project library, among other information. The project library is maintained for use in processing other compilation units for help in name resolution and symbol table processing. It is a very central part to the Ada Translator. In effect, it is another symbol table.

Details concerning exactly what is stored in the project library and when it is used is explained below with the discussions about specific Ada constructs.

Relation Language

The reason that the Software Metric Analyzer is written in three phases is to protect proprietary information. The relation language that is produced by the Ada Translator is our own, in-house object language [HENRS88]. The Ada source code is translated into the equivalent relation language. The relation language code contains all the information necessary to compute the structure metrics, although they are not calculated until the third phase. The relation language keeps the same structure as the original Ada code, but does not have any superfluous details or syntax. For example, no type declarations are included in the relation language. Enough details are removed from the original source code that it is impossible to generate Ada code from the relation language code. Software developers need not worry about losing any proprietary details from releasing any relation language code for academic research purposes.

The relation language has declaration statements which include a corresponding type complexity for each variable declared. The executable statements consist of simple assignment statements, procedure invocation statements, and condition statements. It is the translation to relation language that makes the Ada Translator so difficult and complex. Calculation of code metrics could be done with little more than just a lexical analyzer. As stated previously, it is important to compute structure and hybrid metrics since they measure different aspects of the source code. Therefore, it is necessary to generate relation language in order to calculate the structure and hybrid metrics in Phase Three of the Software Metric Analyzer.

The different relation language constructs are discussed in detail where appropriate in the next several sections. A complete description of the entire relation language is in Appendix C.

Units of Analysis

The relation language was designed with Pascal in mind as the language to analyze. In Pascal, statements are grouped into procedures and functions. For simplicity, the relation language groups statements into procedures only. That is, both procedures and functions in Pascal are translated into procedures in the relation language. These procedures become the *units of analysis* for which metrics are generated. These units of analysis are the same as the modules that McClure's Module Invocation Complexity, Henry and Kafura's Information Flow metric, and Woodfield's Review Complexity discuss. In Ada, units of analysis are defined to be procedures, functions, packages, exception handlers, and tasks. Therefore, metrics are computed for each procedure, function, package, exception handler, and task.

Code Metric Calculations

Code metrics are also calculated for each unit of analysis. In the sections that follow, each type of statement in Ada is discussed and the resultant relation language code is given. Due to their simplicity, code metrics are discussed here and not with the sections pertaining to the translation to relation language.

The lexical analyzer computes the lines of code metric. The Ada Translator does not use a code formatter to make the lines of code metric consistent, as was mentioned in the previous chapter. The software engineer always has the option to use a code formatter before running the Software Metric Analyzer. As each statement is processed, the tokens that contribute to Halstead's Software Science and McCabe's Cyclomatic Complexity are

counted. The Cyclomatic Complexity for a package that has only a specification is always 1, for the path that is present for the elaboration of the specification's declarations.

After each unit of analysis is processed, the code metrics are calculated from these token counts and output to the code metrics file. See Appendix A for the tokens that are counted as operators in Halstead's Software Science and Appendix B for the conditions that contribute to McCabe's Cyclomatic Complexity.

Pre-Defined Language Environment

The Ada Translator has all of the pre-defined language features available to it that an Ada compiler has. The package `STANDARD` is available with all its type declarations, operator declarations and exceptions, as well as the package `ASCII`. Also part of the pre-defined language environment are the following library units:

the package `CALENDAR`

the package `SYSTEM`

the generic procedure `UNCHECKED_DEALLOCATION`

the generic function `UNCHECKED_CONVERSION`

the generic package `SEQUENTIAL_IO`

the generic package `DIRECT_IO`

the package `TEXT_IO`

the package `IO_EXCEPTIONS`

the package `LOW_LEVEL_IO`

For each of these library units, entries are made in the project library to make them available to the user's program. Relation language is generated for the subprogram declarations in these pre-defined units. The concern with processing pre-defined subprogram declarations is to prevent metrics from being generated for them. The relation language that is generated for a pre-defined subprogram declaration is an *intrinsic*. This

declaration defines the subprogram to the relation manager (which allows the subprogram to be called by the user's code) but will not cause any metrics to be generated for it. Therefore, calls to pre-defined functions do not enter in to any of the complexity calculations.

For example, some of the subprogram declarations in the package TEXT_IO will generate the following declarations in the relation language:

```
intrinsic text_io/create;
intrinsic text_io/open;
intrinsic text_io/close;
```

Type Complexities and Type Declarations

In order to compute the interface metrics in Phase Two, type complexities must be computed by the Ada Translator and passed to the Relation Manager embedded within the relation language. The only type information that is contained in the relation language is in the form of type complexities associated with each variable declaration. The rules for the calculation of type complexities are from Mayo [MAYOK89].

A complexity is assigned to each pre-defined type in Ada (i.e., those defined in the package STANDARD). These complexities are in an external weight file instead of being hard-coded within the Ada Translator. All values in this external file can be easily changed according to the software engineer's preference. User-defined types are composed of pre-defined types and previously defined user-defined types. Mayo defines rules for the calculation of complexities for user-defined types [MAYOK89]. Type complexities are computed as floating-point values for accuracy reasons.

The major problem in computing type complexities deals with incomplete type declarations. An incomplete type declaration is of the form

```
TYPE identifier;
```

and is necessary to construct any recursive data structure. As an example, see Figure 3. Note that a full type declaration must eventually follow an incomplete type declaration. With a recursive structure, two problems occur. The first problem is that type complexities cannot always be calculated as soon as the type definitions are encountered. This is obvious because an incomplete type declaration has absolutely no information to use to compute a complexity (e.g., the type *node* in Figure 3). Also, the type complexities for type declarations that depend on other types whose complexities are not known cannot be computed (e.g., the type *pointer* in Figure 3). Only when the Ada Translator has processed an entire set of declarations can all the type complexities be computed, since the full type declarations for all incomplete type declarations have been processed by this time.

When calculating the type complexities that could not be computed as the types were being processed, the other problem becomes apparent. Incomplete type declarations are only necessary in recursive data structures. Calculation of the type complexity of recursive data structures becomes an unending cycle. In the example, the type complexity of *node* depends on the complexity of *integer* (which is known) and the complexity of *pointer* (which is not known). The complexity of *pointer* depends on *node*. This is a recursive loop and is quite a problem. The solution is to use a Self Reference value that is in the external weight file. When a loop such as this is encountered, the Ada Translator uses the Self Reference value and backs out of the loop.

The Ada Translator makes a symbol table entry for each type declaration. It also assigns a unique number to each type that is declared. This helps in evaluating the type of expressions, but is also necessary to process enumeration type declarations.

The only relation language that is generated from a type declaration is for the enumeration literals of an enumeration type declaration. Recall that the relation language was defined with Pascal in mind as the language of analysis. All identifiers within the

```
TYPE node;
TYPE pointer is ACCESS node;
TYPE node is
   RECORD
      value : integer;
      next : pointer;
   END RECORD;
list : node;
```

Figure 3. Incomplete Type Declaration Example

same declarative part must be unique in Pascal, and therefore in the relation language too. Since enumeration literals may be overloaded in Ada, the Ada Translator must make them unique. The Ada Translator appends the enumeration type number to each enumeration literal whenever it is used (i.e., in its declaration and when it is referenced). See Figure 4 for an example of how enumeration type declarations are translated into relation language. The complexity of each enumeration literal is taken from the external weight file.

Object Declarations

All object declarations are translated into relation language code with one of the following formats:

```
local (n) variable_name;
```

or

```
const (n) constant_name;
```

where *n* is the type complexity of *variable_name* and *constant_name*. No distinction is made between declarations of variables of different types, with the exception that their type complexities vary. The following object declaration would be generated for the variable *list* from Figure 3:

```
local (2) list;
```

Note that although type complexities are computed as floating-point values, they always appear as integer values in the relation language. These type complexities are used in the computation of the interface metrics. If the user feels that type complexities contain too much proprietary information, the Ada Translator has an option that will turn off the type complexity output. The interface complexities will not be as accurate when the type complexities are turned off. The Ada Translator makes a symbol table entry for each object declaration.

```
TYPE flag is (red, white, blue);
TYPE primary is (red, green, blue);
```

(a) Ada Code

```
const (2) red_1;
const (2) white_1;
const (2) blue_1;
const (2) red_2;
const (2) green_2;
const (2) blue_2;
```

(b) Relation Language Code

Figure 4. Enumeration Type Declaration Translation

Statements

The Ada Translator translates statements from Ada into relation language. The translation is straightforward and the details are presented in the next few sections. Very few problems occurred in this part of the development of the Ada Translator.

Assignment Statements

Assignment statements in Ada are translated into the relation language with the following format:

```
variable_name = expression;
```

Procedure Call Statements

Procedure call statements are translated into the relation language in the following format:

```
procedure_name (parameter, ..., parameter);
```

One difference between an Ada procedure call and a procedure call in the relation language is that the parentheses are required in the relation language, even if there are no parameters. This is a restriction imposed by the Relation Manager.

IF Statements

All conditional statements in Ada are translated into COND statements in the relation language. The COND statement consists of the keyword COND followed by a condition, followed by a block of statements that depend on that condition. A single digit can be appended to the keyword COND to facilitate calculation of interface metrics. When computing interface metrics, the Relation Manager makes a distinction between different conditionals based on this digit.

Since appending this digit uniquely identifies the Ada statement that corresponds to the COND statement, the relation language that is generated does not hide as much proprietary information as it could if a COND without the digit were generated instead. Any

COND with the appended digit can be traced back to an IF, CASE, or LOOP statement. If the user believes that this is releasing too much proprietary information, the option of appending this digit may be turned off. However, the interface metrics that are generated will be weaker since turning this option off reduces the amount of information that is available to use in their calculation.

As an example, see Figure 5 for the translation of an IF statement from Ada to relation language. Note that no distinction is made between the if-part and the else-part of an IF statement since both parts depend on the same condition. Statements *stmt1*, *cond5* and *stmt3* all depend on *var1* and *var2*. Statement *stmt2* depends on *var1*, *var2*, *var1* (again) and a literal constant (all literal constants are translated to *100*).

CASE Statements

CASE statements are translated into relation language similar to the way IF statements are translated. The major difference is that the condition that guards the COND statement consists of the CASE expression and all the choice lists. See Figure 5 for the translation of a CASE statement from Ada to relation language. As mentioned in the IF statement discussion, the digit following the COND may be removed if necessary.

Null Statements

Null statements appear in the relation language exactly as they do in Ada.

LOOP Statements

LOOP statements are translated into COND statements in a direct manner. A WHILE LOOP becomes a *cond1*, a FOR LOOP becomes a *cond2* and a basic LOOP becomes a *cond3*. As mentioned in the IF statement discussion, the digit following the COND may be removed if necessary. Since the FOR LOOP parameter is declared implicitly, the Ada Translator explicitly generates a declaration for it. This is necessary because the Relation Manager

```
IF var1 = var2 THEN
      stmt1;
ELSIF var1 = 0 THEN
      stmt2;
ELSE
      stmt3;
END IF;
```

```
cond4 (var1 & var2)
begin
      stmt1;
      cond5 (var1 & 100)
      begin
            stmt2;
      end;
      stmt3;
end;
```

```
CASE State IS
      WHEN VA =>

      WHEN MD =>

      WHEN OTHERS =>
END CASE;
```

```
cond6 (state & va & md & 100)
begin
stmt1;              stmt1;
      stmt2;
stmt2;              stmt3;
end;
stmt3;
```

```
FOR i in 1..3 LOOP
      process_number(i);
END LOOP;
```

```
local (1) i;
cond2 (i & 100 & 100)
begin
      process_number(i);
end;
```

(a) Ada Code

(b) Relation Language Code

Figure 5. `IF`, `CASE` and `FOR` Statement Translation

expects all variables to have been explicitly declared. See Figure 5 for the translation of a FOR LOOP from Ada to relation language.

EXIT Statements

If there is a condition associated with an EXIT statement, then that condition is translated and added to the condition of the COND statement that corresponds to the LOOP statement that encloses the EXIT statement. If there is no condition with the EXIT statement, nothing is added to the COND statement's condition. The EXIT statement itself is translated into a null statement in the relation language. See Figure 6 for an example of the translation of an EXIT statement from Ada to relation language.

GOTO Statements

GOTO statements are not translated into a control structure in the relation language. They are translated into NULL statements (as a place holder). This is because Ada provides many constructs that make GOTO statements unnecessary in most cases. Also, GOTO statements are not used too often by software developers that are concerned enough about producing quality code that they are using the Software Metric Analyzer.

DELAY Statements

DELAY statements are also translated into NULL statements. This is because they do not contribute to the complexity of a program.

Block Statements

Block statements are translated directly into the relation language, except the BEGIN END keywords are not necessary. The declarations in the declarative part are processed just like other declarations. The sequence of statements are processed normally as well.

```
count := 0;
LOOP
      count := count + 1;
      EXIT WHEN count > 10;
END LOOP;
```

(a) Ada Code

```
count = 100;
cond3 (count & 100)
begin
      count = count & 100;
      null;
end;
```

(b) Relation Language Code

Figure 6. EXIT Statement Translation

Expressions

In some of the previous examples, expressions in Ada, as part of a statement, were translated into relation language code without any explanation. This section explains how Ada expressions are translated into relation language.

Literals

All numeric literals, string literals and character literals are translated into a *100* in the relation language. Enumeration literals are translated as if they were declared as constants. (See the next section on names for an example.)

Names

Name resolution is a large and complex part of the Ada Translator. As mentioned previously, the relation language does not allow overloading of enumeration literals. The method used to make all enumeration literals unique was discussed previously in the Type Complexities and Type Declarations section. The Ada Translator removes any ambiguities that may exist for a name reference by fully qualifying the name when necessary. That is, the full path name of a variable is used instead of the simple variable name. The path name is composed of the names of the units of analysis that reflect the lexical scoping of the variable's declaration. This is always done when referencing any entity that was declared within a package. If an ambiguity exists in a reference, the full path name is also used. See Figure 7 for an example of the Ada Translator's name resolution. Procedure *inner* references procedure *outer's* local variable *var*. Procedure *demonstrate* references package *example's* variable *var*.

Recall that the relation language makes no distinction between variables that were declared in Ada with different types. As a result, variables that are records are treated as if they were simple variables. Therefore, a reference to a record component is translated into

```
Procedure outer is
      var : integer;
      Procedure inner is
            var : integer;
      Begin
            var := 1;
            outer.var := 1;
      End inner;
Begin
      null;
End outer;

Package example is
      var : integer;
end example;


WITH example;
USE example;
Procedure demonstrate is
      TYPE rec is
            RECORD
            val1,
            val2 : integer;
            END RECORD;
      the_rec : rec;
Begin
      var := 1;
      the_rec.val1 := 1;
End demonstrate;
```

(a) Ada Code

```
procedure outer ()
begin
      local (1) var;
      procedure inner ()
      begin
            local (1) var;
            var = 100;
            outer/var = 100;
      end;
      null;
end;

procedure example ()
begin
      local (1) var;
      null;
end;


procedure demonstrate ()
begin
      local (1) the_rec;
      example/var = 100;
      the_rec = 100;
end;
```

(b) Relation Language Code

Figure 7. Name Resolution Example

just a reference to a simple variable. See procedure *demonstrate* in Figure 7 for the translation of a record component reference.

Array references are translated into relation language just like the array reference in Ada, except that square brackets are used instead of parentheses. Attributes are translated into a *100* since they are really a special form of a function call to a pre-defined function. Calls to pre-defined functions do not enter in to any of the complexity calculations. They could also have been translated to a call to an intrinsic function, which also would not contribute to the complexity computations.

Aggregates

Aggregates allow several components of records and arrays to be referenced in a very concise manner. Since the relation language makes no distinction between the components of a record or an array, only the simple variable name of the record or array and the values of the aggregate appear in the relation language. The relation language only needs to convey what variables get updated by what values. The translation of aggregates produces exactly this information. See Figure 8 for the translation of a record aggregate.

Qualified Expressins and Type Conversions

Qualified expressions and type conversions are expressions with an associated type. Since the relation language has no types, the expressions associated with these constructs are simply translated as usual.

Allocators

Allocators are used to dynamically allocate a variable and optionally, to assign an initial value to this variable. A call to the intrinsic function NEW is generated and if there is an initial value for this variable, that expression is translated as well.

```
TYPE person_record is
     RECORD
          name : string(1..20);
          age : positive;
          status : boolean;
     END RECORD;
matthew : person_record;
.
.
.
matthew :=
     (name   => "Matthew",
      age    => 3,
      status => TRUE);
```

(a) Ada Code

```
local (2) matthew;
          .
          .
          .
matthew = 100 & 100 & true;
```

(b) Relation Language Code

Figure 8. Record Aggregate Translation

Function Calls

A function call is generated exactly like a procedure call is generated, except that a function call is an expression (i.e., part of a statement) and not a complete statement.

Operators

Operators in Ada (when they refer to the built-in functions defined in the package `STANDARD`) are translated to ampersands in the relation language. A number may be appended to the ampersand to uniquely identify which operator was used in the Ada code. This information is used by the relation manager in the interface metric calculations to be able to assign different complexities to different operators. Just like the digit following a `COND` for conditional statements is optional and may be turned off by the user, this operator number may also be turned off if the user desires. Turning off this option will affect the interface metric calculations as it will for the conditional statements.

Subprograms

Subprogram bodies are translated into the relation language with a `procedure` header and become a unit of analysis. An entry is made in the symbol table for each subprogram declaration and subprogram body. If the subprogram declaration or subprogram body is a compilation unit, then the following information is stored in the project library:

- a procedure/function flag,
- subprogram name,
- number of parameters,
- for each parameter:
    - parameter name,
    - type number, and
- type of the return value if the subprogram is a function

RETURN statements with expressions in Ada are translated directly to the relation language using a RETURN statement. RETURN statements without expressions (such as those allowed in procedures) are translated into a NULL statement in the relation language to act as a place holder.

See Figure 9 for an example of a subprogram in Ada and Figure 10 for the resultant relation language code. Note that subprogram parameters have a type complexity associated with them, just like object declarations. Procedure *subprogram_overloading_example* will have an entry in the project library. Notice that the relation language requires a parameter list for each procedure declaration, even if it is empty.

Overloading Subprograms

Subprograms can be overloaded in Ada. That is, as long as the number and type of parameters (and the type of the result for functions) of each subprogram are different, subprograms may have the same name. This presents a problem for the translation to relation language, since overloading is not allowed. The solution to this problem is to assign a unique number to each subprogram and append it to the subprogram declaration and all subprogram invocations. This removes the overloading from the relation language.

Translating subprogram invocations are more difficult when overloading is considered. If procedure X is overloaded, a call to procedure X needs to be translated using the correct unique number that applies to the appropriate version of procedure X. The only way to determine which procedure corresponds to the call is to evaluate the type of the parameters. The rules of Ada require that no ambiguities exist, so once one procedure declaration is found that matches the invocation, the Ada Translator can stop searching and generate the call. See Figure 9 for an example of subprogram overloading in Ada and Figure 10 for the resultant relation language code.

```
Procedure subprogram_overloading_example is
   int1, int2, int_result : integer;
   real1, real2, real_result : float;

   Function max (a, b : integer) return integer is
   Begin
      IF a < b THEN
         return b;
      ELSE
         return a;
      END IF;
   End max;

   Function max (a, b : float) return float is
   Begin
      IF a < b THEN
         return b;
      ELSE
         return a;
      END IF;
   End max;

Begin --subprogram_overloading_example
   int1 := 0;
   int2 := 1;
   int_result := max(int1, int2);
   real1 := 0.0;
   real2 := 1.0;
   real_result := max(real1, real2);
End subprogram_overloading_example;
```

Figure 9. Subprogram Overloading in Ada

```
procedure subprogram_overloading_example_1 ()
begin

   local (1) int1;
   local (1) int2;
   local (1) int_result;
   local (3) real1;
   local (3) real2;
   local (3) real_result;

   procedure max_2 ((1) a, (1) b)
   begin

      cond4 (a &11 b)
         begin
         return b;
         return a;
         end;
   end {max_2};

   procedure max_3 ((3) a, (3) b)
   begin

      cond4 (a &11 b)
         begin
         return b;
         return a;
         end;
   end {max_3};
   int1 = 100;
   int2 = 100;
   int_result = max_2(int1, int2);
   real1 = 100;
   real2 = 100;
   real_result = max_3(real1, real2);
end {subprogram_overloading_example_1};
```

Figure 10. Translation of Subprogram Overloading

Operators can also be overloaded in Ada. Again, the relation language does not allow this. It also does not allow procedure designators to be anything but a normal identifier. To solve these problems, the Ada Translator names each overloaded operator as an "op" and appends the unique number to this as usual.

Operator translation is more difficult when overloading is considered. If an operator is overloaded and its use is really a call to a user-defined function, a function call is generated. If the operator is a call to one of the pre-defined functions in package `STANDARD` (e.g., "+" for two integer arguments), the normal ampersand notation previously discussed is used. The types of the arguments are used to determine which operator is being invoked. See Figure 11 for an example of operator overloading.

Packages

Packages are translated into the relation language with a `procedure` header and become a unit of analysis. An entry is made in the symbol table for each package specification. If the package specification is a compilation unit, then the package name is stored in the project library along with all its declarations. Packages contribute more information to the project library than any other compilation unit because of the declarations that must be stored.

The name, unique number and complexity is stored for each type declaration. The name and type number is stored for each object declaration. The same information is stored for subprogram declarations in this case as is stored when a subprogram is a compilation unit (i.e., procedure/function flag, name, number of parameters, name and type number of each parameter, and result type if the subprogram is a function).

Subunits

Subunits cause more information to be stored in the project library than usual. For example, if a subprogram has a body stub as part of its declarations, all of the

```
--Ada Code:
Procedure operator_overloading_example is
   TYPE array_type is ARRAY (1..10) of integer;
   array1, array2, result_array : array_type;

   Function "+"(left, right : array_type) return array_type is
      result : array_type;
   Begin
      FOR i in 1..10 LOOP
         result(i) := left(i) + right(i);
      END LOOP;
      return result;
   End "+";

Begin  --operator_overloading_example
   result_array := array1 + array2;
End operator_overloading_example;
```

```
{relation language code:}
procedure operator_overloading_example_1 ()
begin

   local (3) array1;
   local (3) array2;
   local (3) result_array;

   procedure op_2 ((3) left, (3) right)
   begin
      local (3) result;
      local (1) i;
      cond2 (i &27 100 &26 100)
         begin
         result[i] = left[i] &15 right[i];
         end;
      return result;
   end {op_2};
   result_array = op_2(array1, array2);
end {operator_overloading_example_1};
```

Figure 11. Operator Overloading

subprogram's declarations are stored in the project library, just as if this subprogram were a package. The body stub is stored in the project library too, as well as in the current symbol table. It is necessary to store the subprogram's declarations because they need to be visible to the subunit when it is processed later.

When the subunit is processed, the information stored in the project library on all the declarations is used to manipulate the current symbol table to act as if the subunit were lexically nested where the body stub appeared.

Relation language is generated for a subunit in the usual way, except that the name in the `procedure` header reflects the nesting level. See Figure 12 for an example of the translation of a compilation unit with subunits.

Exceptions

Exceptions provide a way to handle unexpected situations at run time. When an exception is raised (either automatically by the run-time system or when explicitly raised by the program), control is passed to the appropriate exception handler. The dynamic execution path of the program determines which exception handler is executed. Therefore, it is impossible to always know which exception handler gets executed in a static analysis. Since the Software Metric Analyzer examines the programming system statically, exceptions are processed only when they occur in one type of situation. This situation is when the exception is raised explicitly within a frame and a matching exception handler is defined within this frame. A *frame* is a sequence of statements and the associated list of exception handlers. For example, a subprogram body with statements and exception handlers is considered a frame.

The exception handler is translated into a procedure and becomes a unit of analysis. See Figure 13 for an example of exceptions and exception handlers in Ada and Figure 14 for the translation of exceptions and exception handlers. Notice that procedure *common*

```
Procedure outer is
   outer_var : integer;
   procedure inner is separate;
Begin
 inner;
End outer;

separate (outer)
procedure inner is
 inner_var : integer;
Begin
 outer_var := 1;
 inner_var := 1;
End inner;
```

(a) Ada Code

```
procedure outer_1 ()
begin
   local (1) outer_var;
   outer_1/inner_2();
end {outer_1};


procedure outer_1/inner_2 ()
begin
   local (1) inner_var;
   outer_1/outer_var = 100;
   inner_var = 100;
end {outer_1/inner_2};
```

(b) Relation Language Code

Figure 12. Subunit Translation

```
Package define_exceptions is
   except1 : exception;
   except2 : exception;
End define_exceptions;

With define_exceptions;
Use define_exceptions;
With Text_IO;
Procedure exception_example is
   Procedure common (param : integer) is
   Begin
      IF param = 0 THEN
         RAISE except1;
         RAISE except2;
      END IF;
   Exception
      WHEN except1 => text_io.put_line("error 1 occurred");
   End common;

   Procedure one (param : integer) is
   Begin
      common(param);
   Exception
      WHEN except2 => text_io.put_line("error 2 occurred");
   End one;

   Procedure two (param : integer) is
   Begin
      common(param);
   Exception
      WHEN except2 => text_io.put_line("error 2 occurred");
   End two;

Begin
   one (1);
   two (2);
End exception_example;
```

Figure 13. Exceptions and Exception Handlers in Ada

```
procedure define_exceptions_1 ()
begin

   null;
end {define_exceptions_1};

procedure exception_example_2 ()
begin

   procedure common_3 ((1) param)
   begin

      cond4 (param &9 100)
         begin
         except1_4;
         null;
         end;
      procedure except1_4 ()
      begin
         text_io_100/put_line_101(100);
      end {except1_4};
   end {common_3};

   procedure one_5 ((1) param)
   begin
      common_3(param);
   end {one_5};

   procedure two_6 ((1) param)
   begin
      common_3(param);
   end {two_6};

   one_5 (100);
   two_6 (100);
end {exception_example_2};
```

Figure 14. Translation of Exceptions and Exception Handlers

raises *except1* and *except2*, but only has an exception handler for *except1*. If this segment of Ada code were to be executed, the exception handler for *except1* in procedure *common* would be executed when *except1* is raised. Therefore, the Ada Translator generates a procedure call for the raise of *except1* and generates a procedure body for the exception handler of *except1*.

Since procedure *common* does not have an exception handler for *except2*, it is unknown during a static analysis which exception handler is executed when *common* raises *except2*. This is only known at run time. (The run time system could determine which of procedures *one* or *two* called procedure *common* by the execution path and execute the appropriate exception handler.) Therefore, the raise statement for *except2* is translated into a null statement to serve as a place holder. Note that the exception handlers in procedures *one* and *two* are not translated into the relation language at all, since it cannot be statically determined that they are ever called. Exceptions are a part of Ada that would benefit greatly from dynamic metric analysis.

Generic Units

The processing of generic units is the most complex and involved part of the Ada Translator. Both subprograms and packages can be defined as a generic unit. When a generic declaration is processed, an entry is made in the symbol table and, if the generic declaration is also a compilation unit, information is stored in the project library. The information that is stored is the name of the generic declaration and a flag indicating whether the generic declaration is for a procedure, function, or a package. If the declaration is for a package, the declarations within the package are stored as well. The generic formal parameters are also stored. Generic formal parameters may be objects, types, or subprograms.

When a generic body is processed, its source code is saved by the Ada Translator to use when processing its generic instantiations. Since generic units are simply templates, neither code metrics nor relation language is generated for them. The metrics and relation language translation is performed for all of the generic instantiations, based on the generic body.

Once a generic declaration has been processed, instances of the generic unit may be declared, assigning actual parameters to the generic formal parameters. The body of the generic unit may be processed either before or after any instances of it. The only requirement is that a generic unit be declared before any of its instantiations. Therefore, instances may be fully processed at the time they are presented to the Ada Translator (if the body of the generic unit has already been processed), or saved until the very end of the Ada Translator's execution. At this time, the generic body is guaranteed to have been processed.

The Ada Translator processes generics just as one might think. The actual parameters are matched to the generic formal parameters. The values of these actual parameters are used whenever the formal parameters are referenced within the generic body. This is easy to explain, but much harder to implement. See Figure 15 for an example of generic units and Figure 16 for their translation. Procedures *gen_inst1* and *gen_inst2* cannot be fully processed when their instantiations are encountered because the body of procedure *template* has not yet been submitted to the Ada Translator. They are processed at the end of the Ada Translator's execution. Procedure *gen_inst3* is processed fully when its instantiation is encountered because procedure *template's* body has been processed. Note that the formal parameters have been replaced by the actual parameters when the body of *template* is translated into *gen_inst3, gen_inst1* and *gen_inst2*.

```
package container is
   type enum1 is (a);
   type enum2 is (a, b);
   type enum3 is (a, b, c);
   type array_type is array (1..100) of integer;
   procedure bubble_sort(array_to_sort : IN OUT array_type);
   procedure merge_sort (array_to_sort : IN OUT array_type);
   procedure quick_sort (array_to_sort : IN OUT array_type);
end container;

with container;
use container;
generic
   type enum_type is (<>);
   with procedure sort(array_to_sort : IN OUT array_type);
procedure template;

with container;
use container;
with template;
procedure gen_inst1 is new template
   (enum_type => enum1,
    sort      => bubble_sort);

with container;
use container;
with template;
procedure gen_inst2 is new template
   (enum_type => enum2,
    sort      => merge_sort);

with container;
use container;
procedure template is
   the_array : array_type;
   enum_var : enum_type;
begin
   sort (the_array);
end template;

with container;
use container;
with template;
procedure gen_inst3 is new template
   (enum_type => enum3,
    sort      => quick_sort);
```

Figure 15. Generic Units in Ada

```
procedure container_165 ()
begin

   procedure bubble_sort_166((3) array_to_sort)
   begin
   ...
   end {bubble_sort_166};

   procedure merge_sort_167((3) array_to_sort)
   begin
   ...
   end {merge_sort_167};

   procedure quick_sort_168((3) array_to_sort)
   begin
   ...
   end {quick_sort_168};

end {container_165};

procedure gen_inst3_173 ()
begin

   local (3) enum_var;
   local (3) the_array;
   container_165/quick_sort_168(the_array);
end {gen_inst3_173};

procedure gen_inst1_171 ()
begin

   local (1) enum_var;
   local (3) the_array;
   container_165/bubble_sort_166(the_array);
end {gen_inst1_171};

procedure gen_inst2_172 ()
begin

   local (2) enum_var;
   local (3) the_array;
   container_165/merge_sort_167(the_array);
end {gen_inst2_172};
```

Figure 16. Translation of Generic Units

Tasks

Tasks are translated into the relation language with the emphasis on having them contribute to the complexity of programming systems due to the flow of information. Therefore, task bodies and entries are translated into procedures in the relation language and entry call statements are generated just like procedure call statements. See Figure 17 for an example of tasks in Ada and Figure 18 for the translation of tasks.

Ada Features That Are Ignored

The Ada Translator accepts full Ada. Some constructs in Ada, however, are ignored and not processed by the Ada Translator. The situations where some exceptions and exception handlers are not processed were mentioned previously. The Ada Translator ignores parts of Ada for one of several reasons: either the item in question does not contribute to a program's complexity, the construct is machine dependent, or (as is the case with exceptions) the correct translation cannot be determined statically.

Pragmas, which are basically compiler directives, do not contribute to a program's complexity and are ignored by the Ada Translator. The pre-defined library package `MACHINE_CODE` is not used by the Ada Translator since Ada compilers are not required to provide it [ADARM83]. Also, the Ada constructs dealing with low-level programming are ignored since they are machine dependent.

Phase Two

The Relation Manager reads the relation language code output from the Ada Translator and computes the interface metrics. Also, the Relation Manager creates *relations*. Both the interface metrics and the relations are then passed to the Metric Generator for final processing.

The interface metrics are computed based on all of the relation language code, but a few parts of the code need to be emphasized to show that they contribute directly to the

```
--Ada Code:
Task Type Change_Variable is
      Entry Increment(value : IN OUT integer; amount : IN integer);
      Entry Decrement(value : IN OUT integer; amount : IN integer);
End Change_Variable;

Task Body Change_Variable is
Begin
      LOOP
            SELECT
                  Accept Increment( value : IN OUT integer;
                                    amount :  IN integer) do
                        value := value + amount;
                  End Increment;
            OR
                  Accept Decrement( value : IN OUT integer;
                                    amount :  IN integer) do
                        value := value - amount;
                  End Decrement;
            OR
                  Terminate;
            End SELECT;
      End LOOP;
End Change_Variable;
.
.
.
Change_It : Change_Variable;
val : integer;
.
.
.
      Change_It.Increment(val, 3);
```

Figure 17. Tasks in Ada

```
{relation language code:}
procedure change_variable ()
begin

      procedure increment( (1) value, amount)
      begin
            value = value &15 amount;
      end {increment};

      procedure decrement( (1) value, amount)
      begin
            value = value &16 amount;
      end {decrement};

      null;
end {change_variable};
.
.
.
local (3) change_it;
local (1) val;
.
.
.
      change_variable/increment(val, 100);
```

Figure 18. Translation of Tasks

interface metric calculations. The type complexities associated with each variable declaration are only used in the calculation of interface metrics. Also, the unique numbers appended to the end of conditional statements (`CONDn`) and to the end of ampersands (denoting a particular operator) are only used in the interface metric calculations. These unique numbers are used to determine which conditional or which operator was used in Ada, so that different complexity weights can be applied. For instance, this is how multiplication is given a higher complexity than addition.

As mentioned previously, the interface metric calculations themselves are not a part of this research. The role that this research played in regards to interface metrics was simply to gather the necessary information and pass it on to the Relation Manager.

Relations are grouped by units of analysis. A set of relations is computed for each variable within a unit. Relations are simply a list of where each variable gets its information. Examples would be from a literal constant, another variable, the result of a function invocation, or an output parameter from a procedure invocation. For a complete explanation of relations, see [KAFUD82].

Phase Three

The third phase of the Software Metric Analyzer is where all the information produced thus far is accumulated and assembled into a format accessible to the user. The code metrics from the Ada Translator and the interface metrics and relations from the Relation Manager are included in this final phase. From the relations generated in Phase Two, the Metric Generator calculates structure metrics. The code metrics, together with these newly calculated structure metrics, allow the hybrid metrics to be calculated. Therefore, all metric values are available to the user from this third and final phase of the Software Metric Analyzer.

Conclusion

The Software Metric Analyzer has now been presented. Much emphasis has been placed on the details of the Ada Translator. The next chapter discusses the analysis and verification of the Software Metric Analyzer.

Chapter 4 The Analysis

Introduction

The Software Metric Analyzer was used to analyze a commercial Ada product. The metric analysis was able to determine those units of analysis with the greatest complexity. After outlier data was removed, intermetric correlations produced similar results to other metric studies. The method of analysis and the detailed results are discussed below.

Data Description

Software Productivity Solutions (SPS) is a small company (less than 40 employees) interested in producing quality software products. Located in Melbourne, Florida, most of their contracts are from the Department of Defense. The data used in the analysis of the Software Metric Generator is an Ada-based Design Language (ADL) Processor produced by SPS. The system was donated to the Virginia Tech Metrics Research Group for use in the validation of the Software Metric Analyzer. The ADL programming system provides for syntax checking and limited semantic checking of ADL source files. Also, it generates a variety of data dictionary and cross reference reports and has an on-line help facility. The system consists of four subsystems: support, database, report, and analyzer. In total, this system has 83,799 textual lines of code and 5,355 units of analysis. (A unit of analysis was defined in detail in chapter 3.) The support section contains 6,702 lines of code. The database section contains 43,019 lines of code. The report section contains 3,874 lines of code. The analyzer section contains 30,204 lines of code.

Analysis of the Data

The Software Metric Analyzer processed the ADL Processor's source code and the resultant code, structure, and hybrid metrics were analyzed. See Table 1 for a list of

Table 1. List of Abbreviations

Abbreviation	Meaning
LOC	Lines of Code
N	Halstead's Length
V	Halstead's Volume
E	Halstead's Effort
CC	McCabe's Cyclomatic Complexity
RC	Woodfield's Review Complexity
INFO	Henry and Kafura's Information Flow

abbreviations used in the following tables and figures. See Figures 19 through 25 for the histograms of the individual complexity metrics. In each histogram, the last column of data represents all of the procedures with complexity values greater than or equal to the indicated value. For example, in Figure 19, there are approximately 100 procedures with at least 120 lines of code. See Table 2 for some summary statistics of the raw data and Table 3 for the intermetric correlations of the raw data. All correlations in this thesis are Pearson correlations.

After the data was gathered, the metrics had to be validated. Usually this is done by using error history analysis or development time information but none of this was available. Therefore, the validation was performed by a subjective hand inspection of some of the units of analysis that were flagged as potential problem areas.

In Figure 19, the distribution of the lines of code metric values are shown. The unit of analysis that has the largest lines of code value is a package that has only a package specification. It is a part of the analyzer subsystem of the ADL Processor system. The reason for its length is that it contains table initializations in constant declarations.

The distribution of Halstead's Software Science measures N, V, and E are presented in Figures 20, 21 and 22. The same unit of analysis is responsible for all three maximum values of Halstead's Software Science measures. It is a procedure also in the analyzer subsystem. The reason for its large Software Science metric values is that it contains a lot of initialization statements (using a lot of operators and operands). It is interesting to note that the same unit of analysis also accounts for the smallest values calculated for all of Halstead's Software Science measures. This unit of analysis is an exception that only propagates the exception to the unit that called the surrounding frame.

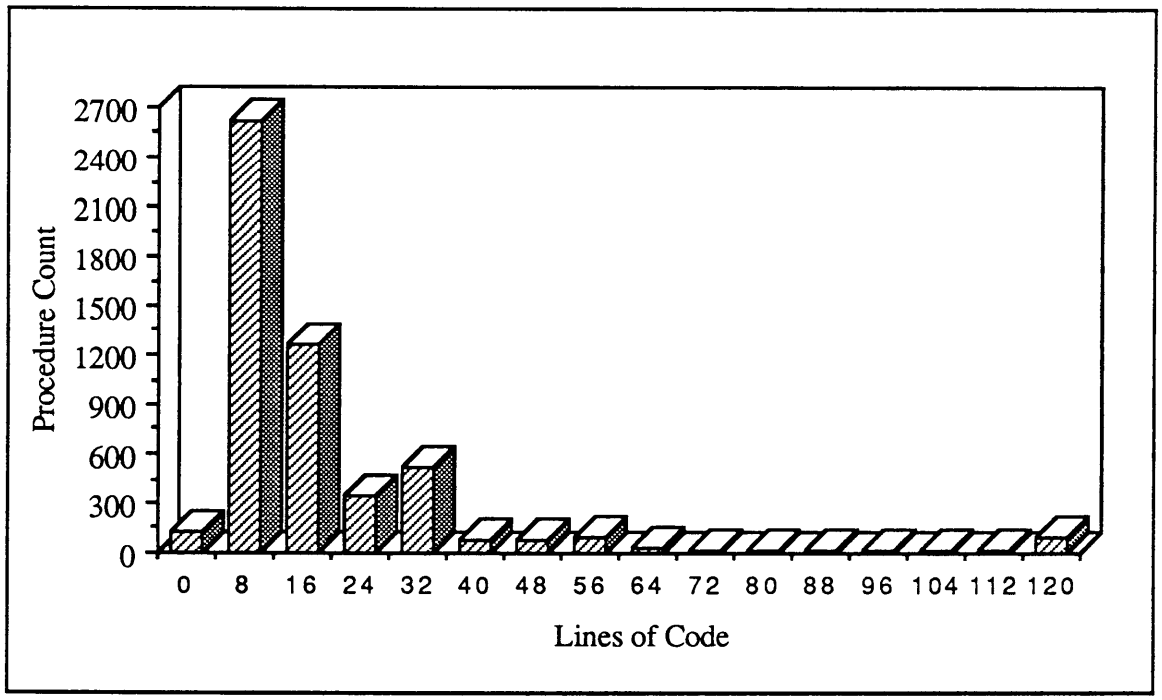

Figure 19. Lines of Code Histogram

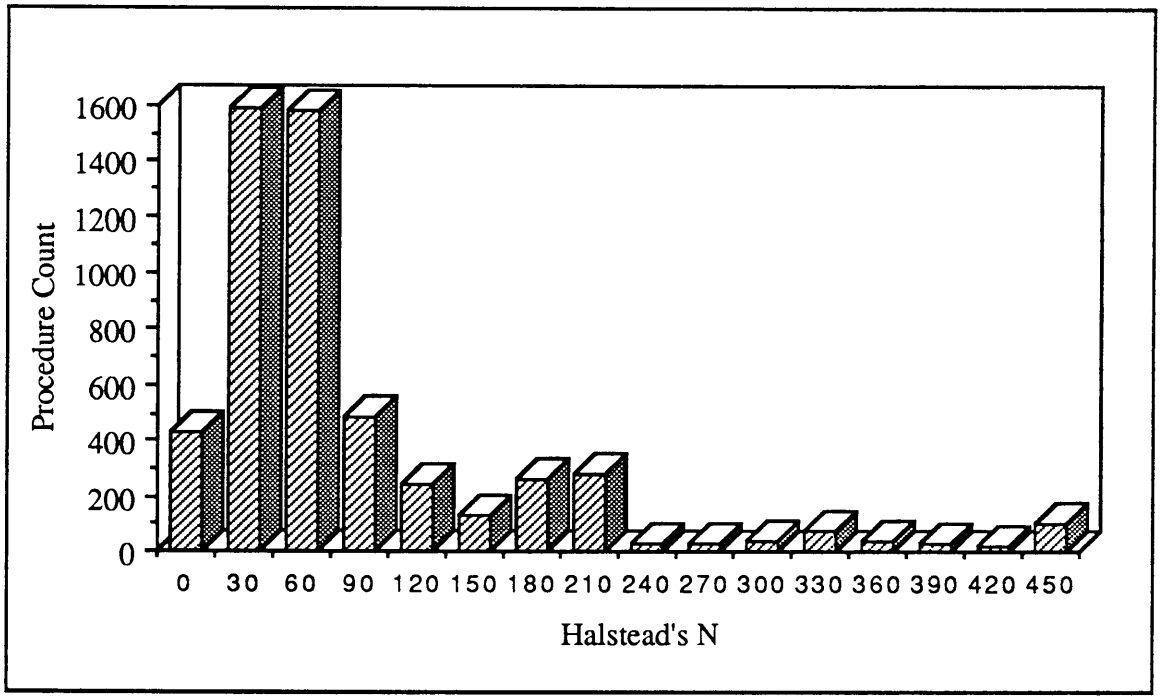

Figure 20. Halstead's Software Science N (Length) Histogram

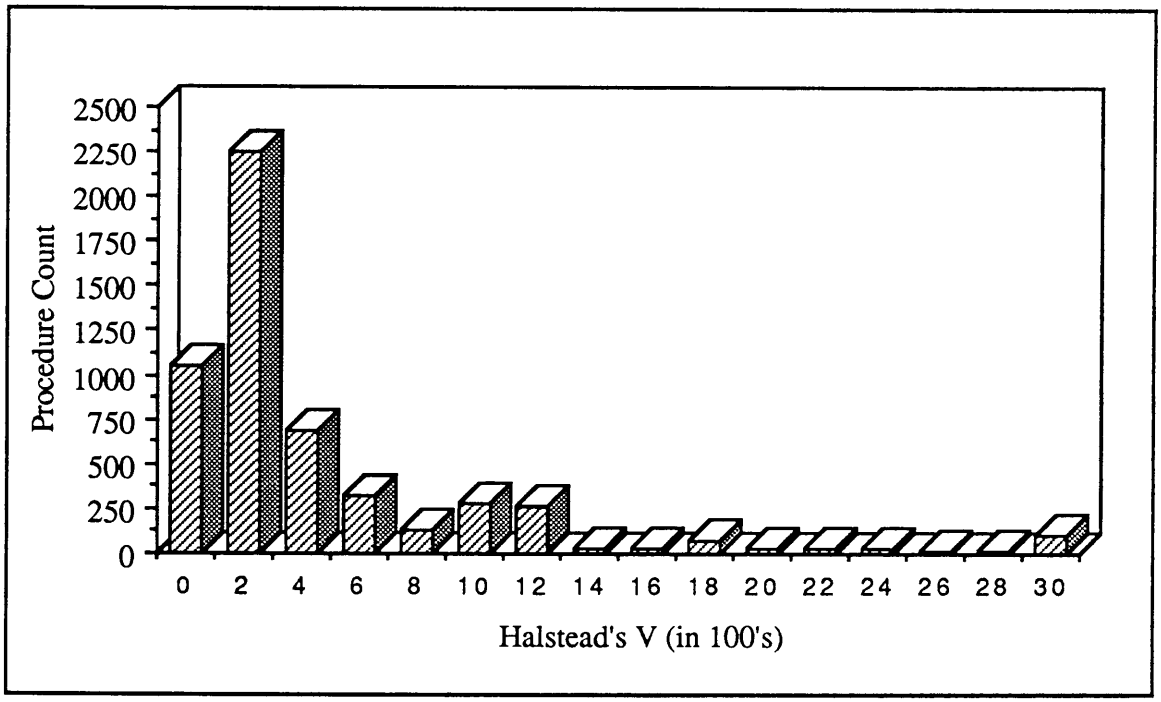

Figure 21. Halstead's Software Science V (Volume) Histogram

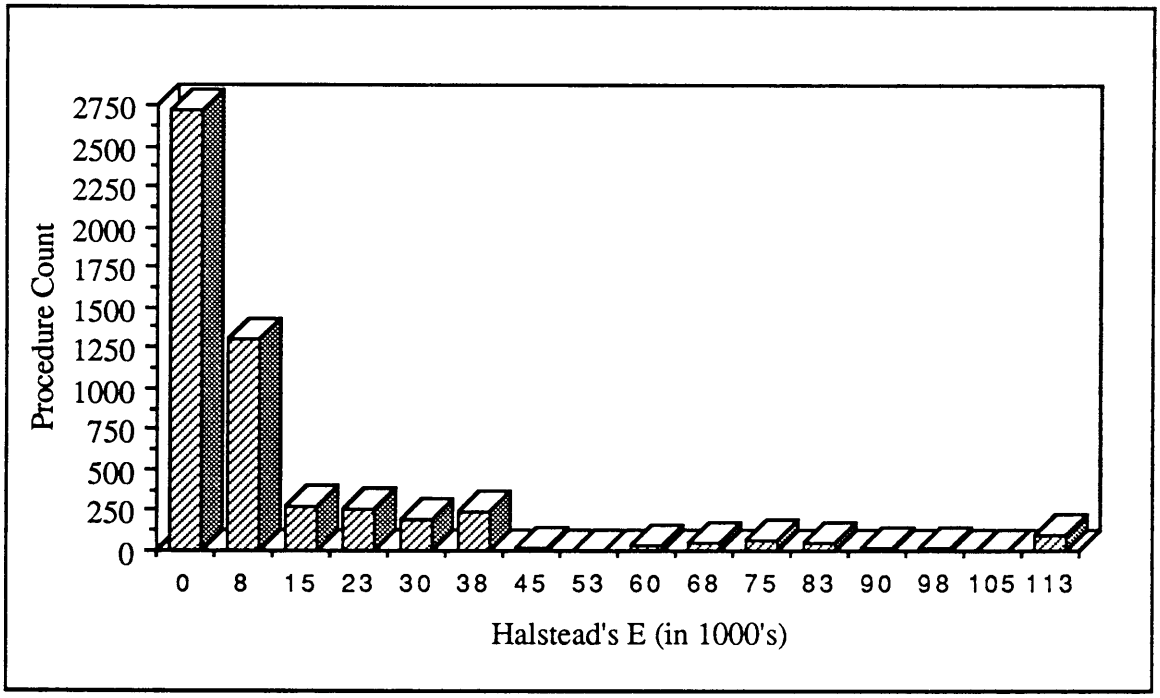

Figure 22. Halstead's Software Science E (Effort) Histogram

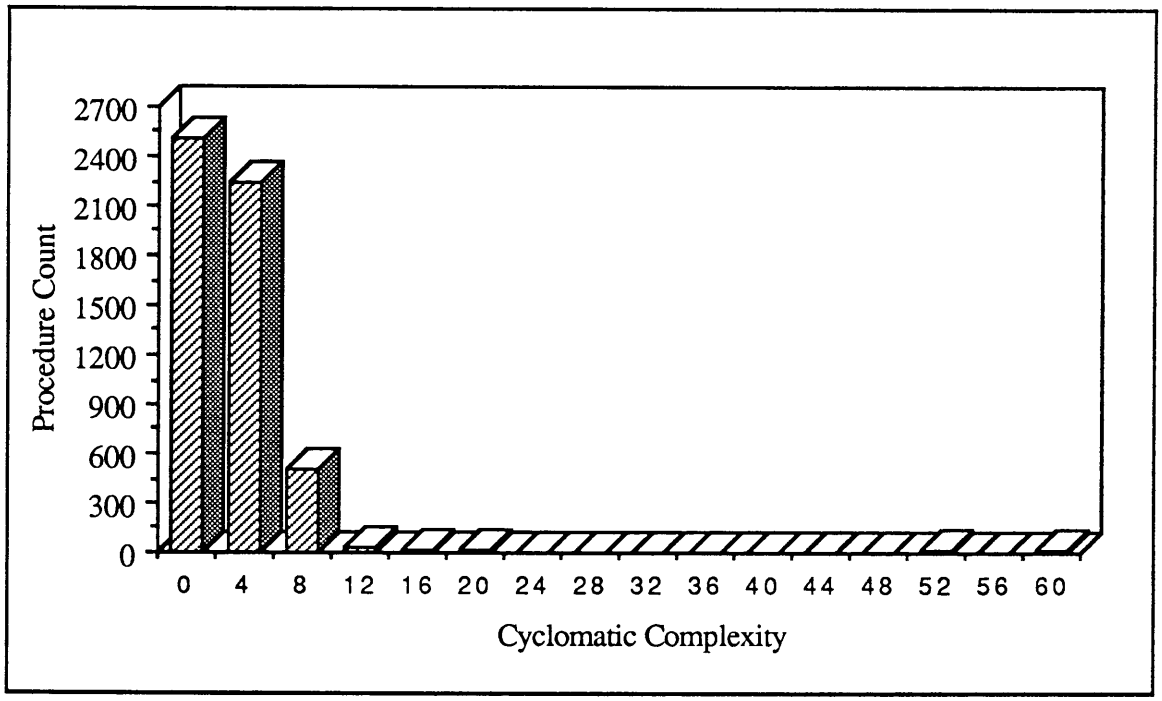

Figure 23. McCabe's Cyclomatic Complexity Histogram

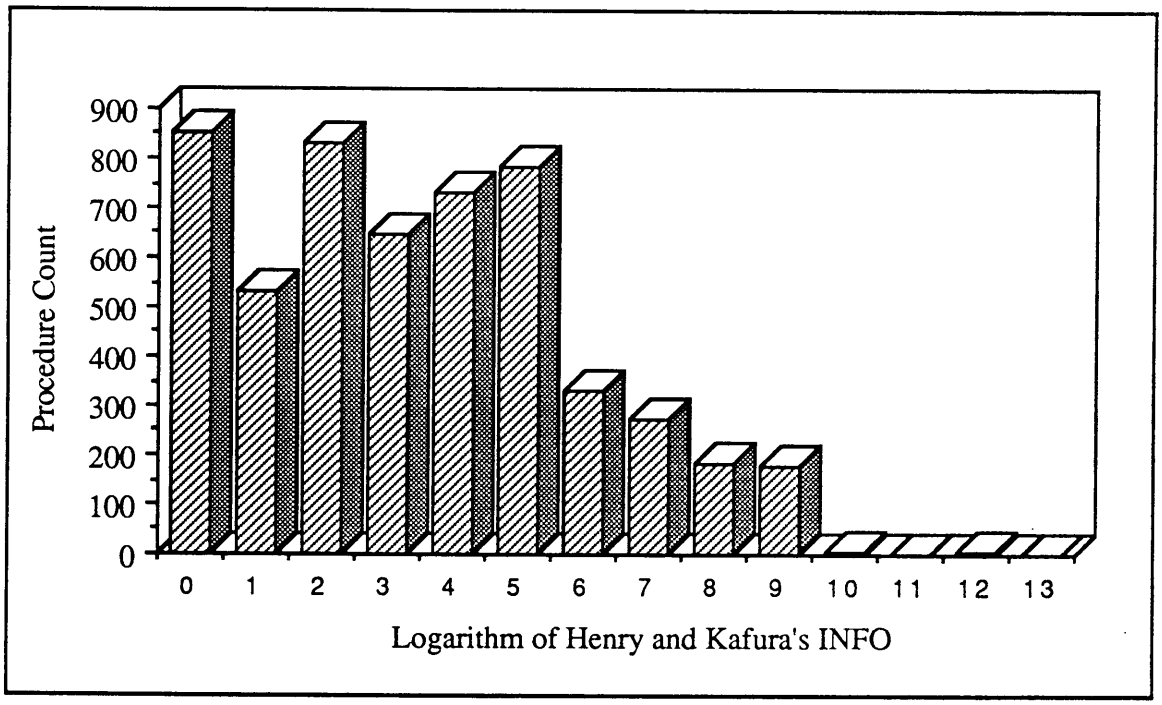

Figure 24. Henry and Kafura's Information Flow (INFO) Histogram

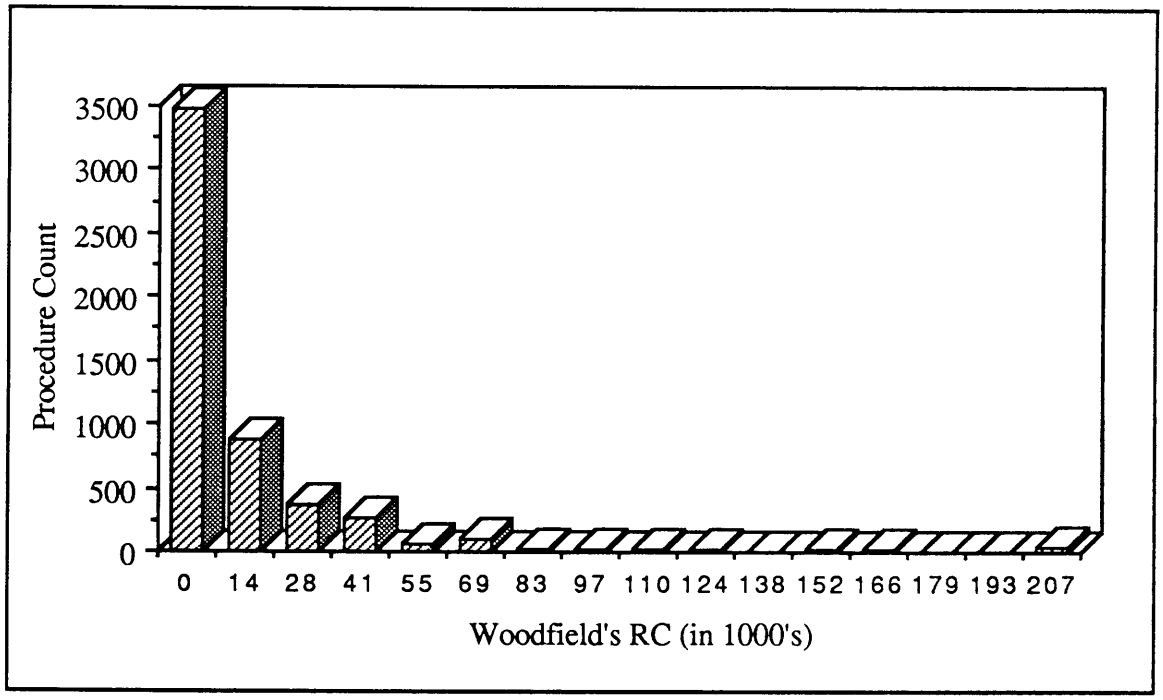

Figure 25. Woodfield's Review Complexity (RC) Histogram

Table 2. Summary Statistics of Raw Data

Metric	Mean	Standard Deviation	Minimum	Maximum
LOC	20.0	52	1	2391
N	99.6	247	4	10892
V	553.4	2030	8	93218
E	21370.1	201593	12	9961191
CC	3.0	9	1	354
RC	21260.0	197702	12	9961191
INFO	16959213555.0	831363564057	1	55959169202181

Table 3. Intermetric Correlations of Raw Data

Metric	LOC	N	V	E	CC	RC	INFO
LOC							
N	0.644						
V	0.616	0.991					
E	0.571	0.927	0.952				
CC	0.525	0.571	0.543	0.600			
RC	0.559	0.930	0.960	0.991	0.534		
INFO	0.029	0.066	0.060	0.029	0.001	0.058	

Figure 23 shows the distribution of McCabe's Cyclomatic Complexity values. The unit of analysis that has the largest Cyclomatic Complexity value is a procedure in the analyzer subsystem that contains a (very large) CASE statement.

In Figure 24, the distribution of Henry and Kafura's Information Flow metric values are presented. Note that this distribution is on a log-scale. It is necessary to use a log-scale for the Information Flow metric due to the great difference in magnitude among the metric values [HENRS81b], [KAFUD85]. The unit of analysis that has the largest Information Flow value is a procedure in the database subsystem. It only has two parameters, but its fan-in is 917 and its fan-out is 687. This indicates a high level of coupling with its environment. Most of its subprogram invocations are to a user-defined input procedure. There are also a lot of assignment statements. Most of the fan-ins and fan-outs are not from subprogram invocations but are from data structures declared in external packages being referenced within this procedure. This procedure is also considered to be an outlier for all of Halstead's Software Science measures and Woodfield's Review Complexity. This is not surprising since this is a rather large procedure.

Finally, Figure 25 shows the distribution of Woodfield's Review Complexity metric. The unit of analysis with the largest Review Complexity value is the same unit that has the largest Halstead's *E* value (a procedure with a lot of initialization statements). This procedure must have a Woodfield *fan-in* that does not exceed 3, since the Review Complexity value and Halstead's *E* would differ otherwise. This is expected since an initialization procedure should not, in general, be called more than once.

From the histograms in Figures 19 through 25 and Table 2, it is clear that most of the units of analysis in this system are of a reasonable size and complexity. This may be because the developers of this Ada system at Software Productivity Solutions have short,

open lines of communication (due to being a small company) and that they strive to produce quality software. Using Ada to write this system might also be a factor influencing this product's reasonable complexity measures.

As can be seen in Table 2, the maximum metric values are 40-70 standard deviations away from the mean. There are some units of analysis in the system that produce exceptionally large complexity values and therefore are adversely affecting the analysis of the system as a whole. Also, the intermetric correlations in Table 3 are not extremely definitive. Previous studies show that code metrics highly correlate for languages other than Ada [HENRS81a], [LEWIJ89], [HENRS90b]. The code metrics in the analysis of the raw data do not correlate as high as expected. The Information Flow metric does not correlate at all to any of the other metrics. This is an expected result, as it conforms to previous studies for other languages [HENRS81a], [HENRS90b]. Since the Information Flow metric does not correlate to the other metrics, it must be measuring some different aspect of complexity.

With the observation that the code metrics do not correlate as high as expected, outlier data was removed and the system was again analyzed. A unit of analysis is considered to be an outlier if any *one* of its metric values falls in the top 1% of that metric's values. Since the data is clustered towards small complexity values, only the largest 1% of the data was removed. (This is true for all the metric values *except* Henry and Kafura's Information Flow metric, for which the largest 4% of the data was removed.) Outliers were defined this way because the distribution is very sparse for very large values for each of the metrics. The range of metric values would definitely vary from system to system, so the definition of an outlier would need to vary as well. Different companies would most likely define outliers in different ways as a method of tuning the metrics for their specific environment.

Outlier removal resulted in removing the data for 5% of the units of analysis. See Table 4 for some summary statistics of the data and Table 5 for the intermetric correlations of the data with the outliers removed. Table 6 contains information concerning the overlap of the outlier data among the metrics.

With the outlier data removed, the maximum metric values are only 7-13 standard deviations from the mean. The mean did decrease some (as expected), but for most of the metrics it did not decrease that much. Therefore, removing 5% of the data is not detrimental in this case. The main point to be said about the summary statistics with the outlier data removed in Table 4 is that the maximum values and standard deviations are more in-line with the mean value of each metric.

The intermetric correlations of the data with the outliers removed in Table 5 are more consistent with previous results than the correlations using the raw data. The code metrics correlate highly, which seems to validate the code metric definitions for Ada that this research defined. McCabe's Cyclomatic Complexity metric does not correlate to the other code metrics as well as all of the other code metrics correlate to one another. Part of the reason for this could be the fact that package definitions are not required to have a body of code with them. A package that only has a specification will always have a Cyclomatic Complexity value of 1; all the other code metric values can vary greatly.

This should not be a concern to anyone using the Software Metric Analyzer. The analyzer has two different methods of grouping units of analysis. Modules can be defined in order to group a package together with all of its nested subprograms and packages. In this way, complexity analysis can be performed on larger sections of code. The other grouping mechanism of the analyzer is more flexible and allows any combination of units of analysis to be grouped together, regardless of their lexical nesting level.

Table 4. Summary Statistics of Data with Outliers Removed

Metric	Mean	Standard Deviation	Minimum	Maximum
LOC	16.3	15	1	150
N	83.0	83	4	645
V	429.2	518	8	4055
E	11152.5	19049	12	184177
CC	2.4	2	1	18
RC	11506.6	19568	12	182989
INFO	12510577.0	67495181	1	900486748

Table 5. Intermetric Correlations of Data with Outliers Removed

Metric	LOC	N	V	E	CC	RC	INFO
LOC							
N	0.947						
V	0.942	0.997					
E	0.916	0.957	0.965				
CC	0.668	0.726	0.717	0.727			
RC	0.904	0.940	0.946	0.968	0.700		
INFO	0.065	0.090	0.098	0.072	-0.007	0.106	

Table 6. Overlap of Outliers

Number of Metrics in Overlap	Percentage of the Outlier Data
1	76.0
2	5.5
3	2.5
4	2.9
5	5.6
6	5.8
7	1.7

It was previously mentioned that Henry and Kafura's Information Flow metric does not correlate at all to the other metrics. This is because the Information Flow metric measures the flow of information between units of analysis and does not measure size. The Information Flow metric is probably a better indicator of complexity than the size metrics, because although size contributes to complexity, module interactions make programs difficult to write, test and understand. These interactions are also the source of many errors [HENRS81a].

It should also be noted that since the Information Flow metric does not depend on size but rather on the flow of information, it can produce meaningful results from analyzing a design written in Ada or an Ada-like PDL. This is because designs written in a PDL are broken down by procedures and contain calls to other procedures. With just this calling structure, the Information Flow metric can compute a meaningful complexity measure [HENRS90b].

Woodfield's Review Complexity correlates very high (0.968) to the code metric used (Halstead's *E*). This is because most of the units of analysis have a Woodfield *fan-in* value that is less than or equal to 3. This yields a Review Complexity value equal to Halstead's *E*. Packages could be part of the reason for Woodfield's *fan-in* not exceeding 3 that often. A package with only a package specification will never have a Woodfield *fan-in* that exceeds 3 because packages can not be called, nor can they modify a data structure that is declared external to them.

It can be seen from the information in Table 6 that 76% of the outliers are in the top 1% of *only one* metric's value range. Most of these units of analysis probably should not be considered problem areas since only one metric considers them to have a high complexity. This is a good example of not wanting to use only one metric in analyzing a programming system.

On the other end of the spectrum, 1.7% of the outliers are considered to be an outlier based on *all* of their metric values. Surely these units of analysis are the ones that should be examined carefully, tested rigorously or perhaps be redesigned. One of these units of analysis is the procedure that has the maximum Cyclomatic Complexity value discussed previously. In addition to the large `CASE` statement, there are numerous subprogram invocations and references to data structures defined in external packages. Both of these points contribute equally to its large Information Flow complexity. The size metrics are large due primarily to the `CASE` statement and its contents.

Two of the units of analysis that are considered outliers based on all of their metric values are procedures in the database subsystem. Both contain about 30 `IF` statements and approximately 75 subprogram invocations. Another unit of analysis considered an outlier by all metric values is also a procedure in the database subsystem. It contains about 25 `WHILE LOOPS` and `IF` statements as well as many subprogram invocations. It also contains about 700 lines of code and a lot of subprogram invocations. These units of analysis are considered to be complex and in need of stringent testing because the connectivity of a subprogram to its environment is one of the main sources of errors and program complexity.

The last unit of analysis that is considered to be an outlier based on all the metric values is a procedure in the analyzer subsystem. Although it is just barely an outlier based on its lines of code value, it contains a large nested `IF THEN ELSIF ELSE` statement referencing a lot of data structures from external packages. It also has eight formal parameters and many subprogram invocations that contribute to its Information Flow complexity.

Conclusion

The analysis of a commercial Ada product using the Software Metric Analyzer has been presented. The results from the analysis can be summed as follows:

- the intermetric correlations agree with other studies using other languages
- the metrics were able to identify the units of analysis having high complexity
- the Software Metric Analyzer can be used on software with any level of refinement and produce meaningful results

There is no single metric that can be used to measure source code complexity. This point was made in the discussion of the overlap of the outlier data. This research uses several metrics and they all perform differently. Since the code metrics are easy to compute and are the most familiar to the software engineering community, it is tempting to just calculate them instead of any others. For better coverage of complexity analysis, it is best to use many different types of metrics as done in this research effort. Henry and Kafura's Information Flow metric is used in addition to the code metrics (as is Woodfield's Review Complexity). The reason for using the Information Flow metric is that it is measuring a different (and perhaps more important) aspect of complexity. Also, since the Information Flow metric measures the structure and connectivity of the code (which is present at the beginning stages of software development), the Software Metric Analyzer supports the use of Ada as a PDL. The complexity of designs can be measured and meaningful results can be produced using the Information Flow metric.

Chapter 5 Conclusions

Research Conclusions

This research attempts to define and validate complexity metrics for Ada. The conclusions from the research can be summarized as follows:

- code metrics for Ada were defined
- it was determined how to apply Henry and Kafura's Information Flow, Woodfield's Review Complexity, and McClure's Module Invocation Complexity metrics to Ada
- the code metric definitions were validated since the intermetric correlations agree with other studies using other languages
- the metrics were able to identify the units of analysis having high complexity, verified by hand inspection
- the Software Metric Analyzer can be used on software with any level of refinement and produce meaningful results

Future Work

There is definitely a need for more validation of these complexity metrics for Ada. A thorough validation would include using commercial Ada systems with their error histories. This would show how accurate the metrics are at computing the complexity of different parts of a programming system. (The more complex parts should be the ones with the most errors.) The validation should also include a varied mixture of programming systems, e.g., real-time applications, batch applications, etc. Analyzing an Ada system written by a larger company and developed under different conditions would probably yield different results.

The results attained by this research from analyzing a programming system developed by SPS are probably typical of Ada systems written by small groups of talented people. Metric measurement is not only dependent on the programming language and the

type of programming system being analyzed, but also on the development process. This is why analyzing a system produced by a larger company under a different development environment will probably show different results.

As mentioned in Chapter 2, there is a need for dynamic metrics. Programs need to be measured during execution to be able to give more meaningful results. The complexity that concurrency introduces via the task construct in Ada is important only if it is measured during program execution. Also, since Ada is very communication oriented, a complexity measure that focuses directly on the interfaces is needed. (The Ada Translator portion of the Software Metric Analyzer generates some information that is useful for producing interface complexities, but none of these complexities are produced yet.)

Finally, the third phase of the Software Metric Analyzer, the Generator, can be improved to make it easier to use. Heuristics and tolerances can be added to help interpret the metrics, as well as incorporating a graphical interface to provide menus and multiple displays. Options to link to a statistics package and to generate graphs and charts would definitely improve the Software Metric Analyzer.

Bibliography

[ADAIC88] *Ada Information Clearinghouse Newsletter*, Ada Information Clearinghouse, Vol. 6, No. 4, Dec. 1988.

[ADAIC89] *Ada Information Clearinghouse Newsletter*, Ada Information Clearinghouse, Vol. 7, No. 2, July 1989.

[ADARM83] *Reference Manual for the Ada Programming Language*, Ada Joint Program Office, ANSI/MIL-STD-1815 A, 1983.

[AJPO 83] "DoD Directive 5000.31," *Ada Letters*, --AJPO, Vol. 3, No. 2, Sept./Oct. 1983, pp. 29-30.

[BAKET82] Baker, T. P., "A One-Pass Algorithm for Overload Resolution in Ada," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, Oct. 1982, pp. 601-614.

[BASIV83a] Basili, Victor R. and Elizabeth E. Katz, "Metrics of Interest in an Ada Development," *IEEE Computer Society Workshop on Software Engineering Technology Transfer*, Apr. 1983, pp. 22-29.

[BASIV83b] Basili, Victor R., Richard W. Selby, Jr., and Tsai-Yun Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, Vol. 9, No. 6, Nov. 1983, pp. 652-663.

[BASIV85] Basili, Victor R., and Richard W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings: 8th International Conference on Software Engineering*, Aug. 1985, pp. 386-391.

[BELMP80] Belmont, Peter A., "Type Resolution in Ada: An Implementation Report," *SIGPLAN Notices*, Vol. 15, No. 11, Nov. 1980, pp. 57-61.

[BERAE83] Berard, Edward V., "Halstead's Metrics and Ada," *Ada Letters*, Engineering Ada, Vol. 3, No. 3, Nov./Dec. 1983, pp. 33-44.

[BERAE84] Berard, Edward V., "Halstead's Metrics and Ada—Part II," *Ada Letters*, Engineering Ada, Vol. 3, No. 5, Mar./Apr. 1984, pp. 44-49.

[BONDR84] Bond, Rodney M., "Ada as a Program Description Language (PDL): A Project Software Management Perspective," *Ada Letters*, Vol. 4, No. 1, July/Aug. 1984, pp. 67-73.

[CARDD88a] Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *The Journal of Systems and Software*, Vol. 8, No. 3, June 1988, pp. 185-197.

[CARDD88b] Card, David N., "Major Obstacles Hinder Successful Measurement," *IEEE Software*, QualityTime, Vol. 5, No. 6, Nov. 1988, p. 82, 86.

[CARLW80] Carlson, William E., et al., "Introducing Ada," *Proceedings: ACM 1980 Annual Conference*, Oct. 1980, pp. 263-271.

[CHASA82] Chase, Anna I. and Mark S. Gerhardt, "The Case for Full Ada as a Design Language," *Ada Letters*, Vol. 2, No. 3, Nov./Dec. 1982, pp. 51-59.

[COHEN86] Cohen, Norman H., *Ada as a Second Language*, New York: McGraw-Hill Book Company, 1986.

[CONTS86] Conte, S. D., H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, Reading, MA: The Benjamin/Cummings Publishing Company, Inc., 1986.

[COOKC87] Cook, Curtis R., "Prototype Software Complexity Metrics Tool," *Software Engineering Notes*, Vol. 12, No. 3, July 1987, pp. 58-60.

[COTEV88] Cote, V., et al., "Software Metrics: An Overview of Recent Results," *The Journal of Systems and Software*, Vol. 8, No. 2, Mar. 1988, pp. 121-131.

[CURTB79a] Curtis, Bill, et al., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," *IEEE Transactions on Software Engineering*, Vol. 5, No. 2, Mar. 1979, pp. 96-104.

[CURTB79b] Curtis, Bill, Sylvia B. Sheppard, and Phil Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," *Proceedings: 4th International Conference on Software Engineering*, Sept. 1979, pp. 356-360.

[CURTB83] Curtis, Bill, "Software Metrics: Guest Editor's Introduction," *IEEE Transactions on Software Engineering*, Vol. 9, No. 6, Nov. 1983, pp. 637-638.

[DAVIJ88] Davis, John Stephen and Richard J. LeBlanc, "A Study of the Applicability of Complexity Measures," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, Sept. 1988, pp. 1366-1372.

[DEMAT82] DeMarco, Tom, *Controlling Software Projects: Management, Measurement & Estimation*, Englewood Cliffs, NJ: Yourdon Press, 1982.

[DOD88] Department of Defense, *Defense System Software Development*, DOD-STD-2167 A, 1988.

[ELSHJ84] Elshoff, James L., "Characteristic Program Complexity Measures," *Proceedings: 7th International Conference on Software Engineering*, Mar. 1984, pp. 288-293.

[EVANW83] Evangelist, W. M., "Software Complexity Metric Sensitivity to Program Structuring Rules," *The Journal of Systems and Software*, Vol. 3, No. 3, Sept. 1983, pp. 231-243.

[FIRED88] Firesmith, Donald G., "Mixing Apples and Oranges: Or What is an Ada Line of Code Anyway?," *Ada Letters*, Vol. 8, No. 5, Sept./Oct. 1988, pp. 110-112.

[FISCC88] Fischer, Charles N. and Richard J. LeBlanc, Jr., *Crafting a Compiler*, Reading, MA: The Benjamin/Cummings Publishing Company, Inc., 1988.

[FISHD78] Fisher, David A., "DoD's Common Programming Language Effort," *Computer*, Vol. 11, No. 3, Mar. 1978, pp. 24-33.

[FISHG84] Fisher, Gerry and Philippe Charles, "A LALR(1) Grammer for ANSI Ada," *Ada Letters*, Vol. 3, No. 4, Jan./Feb. 1984, pp. 37-50.

[FITZA78] Fitzsimmons, Ann, and Tom Love, "A Review and Evaluation of Software Science," *Computing Surveys*, Vol. 10, No. 1, Mar. 1978, pp. 3-18.

[GABBE83] Gabber, Eran, "The Middle Way Approach for Ada Based PDL Syntax," *Ada Letters*, Vol. 2, No. 4, Jan./Feb. 1983, pp. 64-67.

[GANNJ86] Gannon, J. D., E. E. Katz, and V. R. Basili, "Metrics for Ada Packages: An Initial Study," *Communications of the ACM*, Vol. 29, No. 7, July 1986, pp. 616-623.

[GILBT77] Gilb, Tom, *Software Metrics*, Cambridge, MA: Winthrop Publishers, Inc., 1977.

[GORDM83] Gordon, Michael, "The Byron Program Design Language," *Ada Letters*, Vol. 2, No. 4, Jan./Feb. 1983, pp. 76-83.

[GRADR87] Grady, Robert B. and Deborah L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

[HALSM77] Halstead, Maurice H., *Elements of Software Science*, New York: Elsevier North-Holland, Inc., 1977.

[HENRS81a] Henry, Sallie, Dennis Kafura and Kathy Harris, "On the Relationships Among Three Software Metrics," *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981, pp. 81-88.

[HENRS81b] Henry, Sallie and Dennis Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, Sept. 1981, pp. 510-518.

[HENRS88] Henry, Sallie, "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers; or, Do You Recognize This Well-Known Algorithm?," *The Journal of Systems and Software*, Vol. 8, No. 1, Jan. 1988, pp. 3-11.

[HENRS89] Henry, Sallie and Roger Goff, "Complexity Measurement of a Graphical Programming Language," to appear in *Software—Practice and Experience*, 1989.

[HENRS90a] Henry, Sallie, and John Lewis, "Integrating Metrics into a Large-Scale Software Development Environment," to appear in *The Journal of Systems and Software*, 1990.

[HENRS90b] Henry, Sallie, and Calvin Selig, "Design Metrics which Predict Source Code Quality," to appear in *IEEE Software*, 1990.

[HOWEB84] Howe, Bob and Jean E. Sammet, "Ada Policy Material from Dallas Meeting in October 1983," *Ada Letters*, Vol. 3, No. 4, Jan./Feb. 1984, pp. 131-136.

[JENSH85] Jensen, Howard A., and K. Vairavan, "An Experimental Study of Software Metrics for Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 11, No. 2, Feb. 1985, pp. 231-234.

[JOHNS75] Johnson, Stephen C., "YACC: Yet Another Compiler-Compiler," *Computing Science Technical Report, No. 32*, Bell Laboratories, Murray Hill, NJ, 1975.

[KAFUD82] Kafura, Dennis and Sallie Henry, "Software Quality Metrics Based on Interconnectivity," *The Journal of Systems and Software*, Vol. 2, No. 2, June 1982, pp. 121-131.

[KAFUD85] Kafura, Dennis and James Canning, "A Validation of Software Metrics Using Many Metrics and Two Resources," *Proceedings: 8th International Conference on Software Engineering*, Aug. 1985, pp. 378-385.

[KAFUD87] Kafura, Dennis and Geereddy R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering*, Vol. 13, No. 3, Mar. 1987, pp. 335-343.

[KEARJ86] Kearney, Joseph K. et al., "Software Complexity Measurement," *Communications of the ACM*, Vol. 29, No. 11, Nov. 1986, pp. 1044-1050.

[KOKOP88] Kokol, P., B. Ivanek, and V. Zumer, "Software Effort Metrics: How to Join Them," *Software Engineering Notes*, Vol. 13, No. 2, Apr. 1988, pp. 55-57.

[LESKM75] Lesk, M. E. and E. Schmidt, "Lex—A Lexical Analyzer Generator," *Computing Science Technical Report, No. 39*, Bell Laboratories, Murray Hill, NJ, 1975.

[LEWK88] Lew, Ken S., Tharam S. Dillon, and Kevin E. Forward, "Software Complexity and Its Impact on Software Reliability," *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, Nov. 1988, pp. 1645-1655.

[LEWIJ89] Lewis, John, and Sallie Henry, "A Methodology for Integrating Maintainability Using Software Quality Metrics," *Proceedings: IEEE Conference on Software Maintenance*, Oct. 1989, pp. 32-37.

[LIH87] Li, H. F. and W. K. Cheung, "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, Vol. 13, No. 6, June 1987, pp. 697-708.

[LIEBE86] Lieblein, Edward, "The Department of Defense Software Initiative—A Status Report," *Communications of the ACM*, Vol. 29, No. 8, Aug. 1986, pp. 734-744.

[LINDL83] Lindley, Lawrence M., "Ada Program Design Language Survey Update," *Ada Letters*, Vol. 2, No. 4, Jan./Feb. 1983, pp. 61-63.

[MAYOK89] Mayo, Kevin A., "Improving Software Quality Through the Use of Interface Metrics," M.S. Thesis, Computer Science Department, Virginia Polytechnic Institute and State University, Dec. 1989.

[MCAUD88] McAuliffe, Daniel, "Measuring Program Complexity," *Computer*, Vol. 21, No. 6, June 1988, pp. 97-98.

[MCCAT76] McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, Dec. 1976, pp. 308-320.

[MCCLC78] McClure, Carma L., "A Model for Program Complexity Analysis," *Proceedings: 3rd International Conference on Software Engineering*, May 1978, pp. 149-157.

[MEHNB87] Mehndiratta, B., and P. S. Grover, "Software Metrics Applied to Ada," *Proceedings: Computer Software & Applications Conference*, Oct. 1987, pp. 368-373.

[MYERW87] Myers, Ware, "Ada: First Users—Pleased; Prospective Users—Still Hesitant," *Computer*, Vol. 20, No. 3, Mar. 1987, pp. 68-73.

[MYERW88] Myers, Ware, "Large Ada Projects Show Productivity Gains," *IEEE Software*, Vol. 5, No. 6, Nov. 1988, p. 89.

[OTTEL81] Ottenstein, Linda, "Predicting Numbers of Errors Using Software Science," *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981, pp. 157-167.

[PIWOP82] Piwowarski, Paul, "A Nesting Level Complexity Measure," *SIGPLAN Notices*, Vol. 17, No. 9, Sept. 1982, pp. 44-50.

[POLLG87] Pollock, Guylaine M., and Sallie Sheppard, "A Design Methodology for the Utilization of Metrics Within Various Phases of Software Life Cycle Models," *Proceedings: Computer Software & Applications Conference*, Oct. 1987, pp. 221-230.

[POSTR86] Poston, Robert M., "Engineering Software Engineering Metrics," *IEEE Software*, Software Standards, Vol. 3, No. 5, Sept. 1986, pp. 52-54.

[RAMAB88] Ramamurthy, Bina and Austin Melton, "A Synthesis of Software Science Measures and the Cyclomatic Number," *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, Aug. 1988, pp. 1116-1121.

[RAMAC84] Ramamoorthy, C. V., et al., "Software Engineering: Problems and Perspectives," *Computer*, Vol. 17, No. 10, Oct. 1984, pp. 191-209.

[RAMAC85] Ramamoorthy, C. V., et al., "Metrics Guided Methodology," *Proceedings: Computer Software & Applications Conference*, Oct. 1985, pp. 111-120.

[SAMMJ82] Sammet, Jean E., Douglas W. Waugh, and Robert W. Reiter, Jr., "PDL/Ada—A Design Language Based on Ada," *Ada Letters*, Vol. 2, No. 3, Nov./Dec. 1982, pp. 19-31.

[SAMMJ86] Sammet, Jean E., "Why Ada Is Not Just Another Programming Language," *Communications of the ACM*, Vol. 29, No. 8, Aug. 1986, pp. 722-732.

[SHATS88] Shatz, Sol M., "Towards Complexity Metrics for Ada Tasking," *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, Aug. 1988, pp. 1122-1127.

[SHENV83] Shen, Vincent Y., Samuel D. Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," *IEEE Transactions on Software Engineering*, Vol. 9, No. 2, Mar. 1983, pp. 155-165.

[SUNOT81] Sunohara, Takeshi, et al., "Program Complexity Measure for Software Development Management," *Proceedings: 5th International Conference on Software Engineering*, Mar. 1981, pp. 100-106.

[SZULP81] Szulewski, Paul A., et al., "The Measurement of Software Science Parameters in Sofware Designs," *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981, pp. 89-94.

[WAKES88] Wake, Steve and Sallie Henry, "A Model Based on Software Quality Factors Which Predicts Maintainability," *Proceedings: Conference on Software Maintenance-1988*, Oct. 1988, pp. 382-387.

[WEYUE88] Weyuker, Elaine J., "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, Sept. 1988, pp. 1357-1365.

[WOODS80] Woodfield, S. N., "Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors," Ph.D. Dissertation, Computer Science Department, Purdue University, Dec. 1980.

[WOODS82] Woodfield, S. N., V. Y. Shen, and H. E. Dunsmore, "A Study of Several Metrics for Programming Effort," *The Journal of Systems and Software*, Vol. 2, No. 2, June 1982, pp. 97-103.

[YOURE89] Yourdon, Ed, "Software Metrics: You Can't Control What You Can't Measure," *American Programmer*, Vol. 2, No. 2, Feb. 1989, pp.3-11.

[YUT88] Yu, T. J., et al., "SMDC: An Interactive Software Metrics Data Collection and Analysis System," *The Journal of Systems and Software*, Vol. 8, No. 1, Jan. 1988, pp. 39-46.

Appendix A Halstead's Software Science Operator Definitions

Operator Description	Token(s) Specifying the Operator
Miscellaneous Operators	
statement terminator	;
list delimiters	( )
general item separator	,
special item separator	;
assignment	:=
Operators Associated with Declarations	
declaration operator	:
constant declaration	CONSTANT
type declaration	TYPE IS or
	TYPE
discrete range	RANGE low..high or
	low..high (low and high are operands)
floating-point accuracy constraint	DIGITS
fixed-point accuracy constraint	DELTA
array definition	ARRAY ( ) OF
unconstrained range	RANGE <>
record declaration	RECORD END RECORD
renaming declaration	RENAMES
Operators Associated with Access Types	
access declaration	ACCESS
dynamic allocation	NEW
dynamic allocation initialization	'
Operators Associated with Subtypes and Type Equivalence	
subtype declaration	SUBTYPE IS
derived type declaration	NEW

Operator Description	Token(s) Specifying the Operator
Operators Associated with Statements	
subprogram calls	name of procedure or function invoked
if-then	IF THEN END IF
elsif	ELSIF THEN
else	ELSE
case	CASE IS END CASE
case selection statement	WHEN =>
choice separator	|
others option	OTHERS
null statement	NULL
loop and block statement identifier	:
loop statement	LOOP END LOOP
while loop	WHILE
for loop	FOR
for direction ascending	IN
for direction descending	IN REVERSE
exit statement	EXIT or EXIT WHEN
statement identifier delimiters	<< >>
goto statement	GOTO
delay statement	DELAY
sequence of statements delimiters	BEGIN END
block statement	DECLARE
Operators Found in Expressions	
string delimiters	" "
character delimiters	' '
component selector	.
select all	ALL
attribute designation	'

Operator Description	Token(s) Specifying the Operator
association	=>
qualified expression	'
based literal delimiters	# #
exponentiation	**
absolute value	ABS
multiplication	*
division	/
remainder REM	REM
remainder MOD	MOD
unary positivity	+
unary negation	-
binary addition	+
binary subtraction	-
concatenation	&
equal	=
not equal	/=
less than	<
less than or equal	<=
greater than	>
greater than or equal	>=
membership	IN
logical negation	NOT
logical and	AND
logical or	OR
logical exclusive or	XOR
logical short-circuit and	AND THEN
logical short-circuit or	OR ELSE

Operators Associated with Subprograms

procedure declaration	PROCEDURE IS or PROCEDURE

Operator Description	Token(s) Specifying the Operator
mode in	nothing or
	IN
mode out	OUT
mode in out	IN OUT
function declaration	FUNCTION RETURN IS or
	FUNCTION RETURN or
	FUNCTION
return statement	RETURN
overloaded operator delimiters	" "

Pragma Declaration Operator

pragma declaration	PRAGMA

Operators Associated with Packages

package specification	PACKAGE IS END or
	PACKAGE IS or
	PACKAGE
package body specification	PACKAGE BODY IS END or
	PACKAGE BODY IS
context clause	WITH
use clause	USE

Operators Associated with Private Types

private type declaration	PRIVATE
limited type declaration	LIMITED

Operators Associated with Separate Compilation

body stub indication	SEPARATE
subunit indication	SEPARATE ()

Operator Description	Token(s) Specifying the Operator
Operators Associated with Exceptions	
exception definition/declaration	EXCEPTION
exception selection statement	WHEN =>
others option	OTHERS
raise statement	RAISE
Operators Associated with Generic Units	
generic declaration	GENERIC
generic instantiation of a package	PACKAGE IS NEW
generic instantiation of a procedure	PROCEDURE IS NEW
generic instantiation of a function	FUNCTION IS NEW
generic formal constant declaration	nothing or IN
generic formal variable declaration	IN OUT
generic type declaration	TYPE IS
generic formal integer declaration	RANGE <>
generic formal floating-point declaration	DIGITS <>
generic formal fixed-point declaration	DELTA <>
generic formal discrete declaration	(<>)
generic formal subprogram specification	WITH or WITH IS or WITH IS <>
Operators Associated with Tasks	
task declaration	TASK TYPE IS END or TASK TYPE or TASK IS END or TASK
entry declaration	ENTRY
task body declaration	TASK BODY IS

Operator Description	Token(s) Specifying the Operator
accept statement	ACCEPT DO END or ACCEPT

Operators Associated with Task Interaction

selective wait	SELECT END SELECT
select choice separator	OR
select guard	WHEN =>
terminate alternative	TERMINATE
else alternative	ELSE
abort statement	ABORT

Operators Associated with Low-Level Programming

length clause	FOR USE
enumeration clause	FOR USE
record clause instantiation	FOR USE
record clause record	RECORD END RECORD
record alignment clause	AT MOD
record storage unit	AT RANGE
address clause	FOR USE AT

Appendix B McCabe's Cyclomatic Complexity Conditions

Each of the following counts as a condition when calculating the Cyclomatic Complexity:

IF

ELSIF

WHILE

FOR

EXIT (only when there is a condition associated with the exit statement)

each CASE statement label

AND

OR

XOR

AND THEN

OR ELSE

Appendix C Relation Language Constructs

The following is a brief description of each of the relation language constructs:

```
local (n) variable_name;        -- local variable declaration
const (n) constant_name;        -- constant declaration
struct (n) variable_name;       -- variable declaration for
                                -- variables within a package
                                -- specification
                                -- (n is the type complexity)
intrinsic procedure_name;       -- declaration for pre-defined
                                -- subprograms
extern procedure_name;          -- subprogram declaration for
                                -- subprograms within a package
                                -- specification
procedure procedure_name (parameter, ..., parameter)
                                -- subprogram declaration
procedure_name (parameter, ..., parameter);
                                -- procedure call statement
condn expression statement_list
                                -- conditional statement
                                -- (n is a unique condition number)
variable_name = expression;     -- assignment statement
begin sequence_of_statements end;
                                -- block statement
null;                           -- null statement
return expression;              -- return statement
```

```
100                          -- all literal constants
&n                           -- all operators
                             -- (n is a unique operator number)
```

The vita has been removed from the scanned document