

THE APPLICATION OF COMPLEXITY METRICS  
TO A SOFTWARE REQUIREMENTS SPECIFICATION

by

ALICE JOSEPHINE LEMIEUX

B. A., University of Maine, 1979

---

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY  
Manhattan, Kansas

1984

Approved by:

  
Major Professor

LD  
2668  
R4  
1984  
L45  
c. 2

111202 662231

TABLE OF CONTENTS

|                                                             |    |
|-------------------------------------------------------------|----|
| 1. INTRODUCTION.....                                        | 1  |
| 1.1 Overview.....                                           | 1  |
| 1.2 Software Life Cycle.....                                | 1  |
| 1.3 Software Requirements Specifications.....               | 2  |
| 1.4 Application of Metrics Early in Life<br>Cycle.....      | 3  |
| 1.5 Software Metrics and Their Uses.....                    | 4  |
| 1.6 Software Complexity Metrics.....                        | 6  |
| 1.7 McCabe's Control Flow Metric.....                       | 8  |
| 1.8 Halstead's Software Science Metrics.....                | 9  |
| 1.9 Henry and Kafura's Information Flow<br>Metrics.....     | 12 |
| 1.10 Scope of the Implementation.....                       | 14 |
| 2. IMPLEMENTATION REQUIREMENTS AND DESIGN<br>DECISIONS..... | 17 |
| 2.1 Overview.....                                           | 17 |
| 2.2 Application of Halstead's Complexity<br>Metrics.....    | 19 |
| 2.3 Application of Procedure Complexity<br>Metric.....      | 21 |
| 2.4 Design Decisions Influenced by the<br>Requirements..... | 22 |
| 3. DESIGN OF COMPLEXITY METRICS APPLICATION.....            | 23 |
| 3.1 Overview.....                                           | 23 |
| 3.2 Scanning the Specification Input.....                   | 25 |
| 3.3 Calculation of the Complexity Metrics.....              | 26 |
| 3.4 Output From The Analyzer.....                           | 27 |
| 4. IMPLEMENTATION OF A COMPLEXITY METRIC ANALYZER.....      | 28 |
| 4.1 Overview.....                                           | 28 |
| 4.2 Scanning the Requirements Specification.....            | 28 |
| 4.3 Calculation of the Complexity Metrics.....              | 30 |
| 4.4 Testing the Analyzer.....                               | 32 |
| 5. CONCLUSIONS AND EXTENSIONS.....                          | 33 |
| BNF For The ERA Requirements Specification.....             | 37 |
| Example of an ERA Requirements Specification.....           | 41 |
| Requirement Specification for the Implementation.....       | 47 |
| Header File Of Tables For Implementation.....               | 56 |
| Source Code For Implementation.....                         | 58 |

## LIST OF FIGURES

|                                                       |    |
|-------------------------------------------------------|----|
| Figure 1 - Complexity Analyzer Data Flow Diagram..... | 24 |
| Figure 2 - Afferent Flow.....                         | 24 |
| Figure 3 - Efferent Flow.....                         | 24 |

## CHAPTER 1

### 1. INTRODUCTION

#### 1.1 Overview

This report describes the application of complexity metrics to the software requirements specification. A complexity analyzer is developed which applies the complexity metrics of Halstead [8] and Henry and Kafura [9] to the requirements specification. The application of complexity metrics to the software requirements specification is an attempt to predict the complexity of a software project early in its development.

#### 1.2 Software Life Cycle

The software life cycle is an important concept of software engineering. The software life cycle is a macro view of the process through which software is developed. Requirements analysis, requirements specification, preliminary design, detailed design, coding, testing, and operations and maintenance are the phases which make up the software life cycle[15].

Each phase in the software life cycle is supported by the documentation which it uses and produces. The documentation produced during the software life cycle phases are an important part of the software product. This documentation is the means by which the software product

moves from one software life cycle phase to another during its development and maintenance.

### 1.3 Software Requirements Specifications

The software requirements specification is a product of the requirements specification phase of the software life cycle. The software requirements specification provides the foundation for the remainder of the software development process. The software requirements specification provides an important means of communication among the user or requester, the analyst, the software developers, and the management.

The software requirements specification describes the meaning of the software to be developed and the constraints which must be adhered to by the software developers. The flow and structure of information, descriptions of the functions of the software, interfaces between functions, constraints on the design, and validation criteria are provided in the software requirements specification. The software requirements specification should provide all of the information required for the design of the software and the validation of that design.

Many formats for the software requirements specification have been proposed and, in some cases, marketed. The requirements specification may describe the software through the flow of information, the structure of

information, or the functions performed on the information. An information flow approach is of particular interest for this report.

The software requirements specification which will be analyzed by this implementation is in an entity-relationship-attribute form. The entity-relationship-attribute form is most closely aligned to the information flow approach to software requirements specification. The information flows between entities which have attributes and relationship to other entities.

The entity-relationship-attribute form of the software requirements specification is a text description of the software to be developed. The flow of the information through the entities is described by the attributes and relationships which are associated with the entities. There is no graphical representation of the flow of information through the entities as there is in some other formats of information flow software requirements specifications.

#### 1.4 Application of Metrics Early in Life Cycle

Since the software requirements specification guides the further refinement and detail of the software being developed, it is important for the specification to be understandable and maintainable. The terms understandable and maintainable are often considered to be synonyms for complexity[5]. Complexity introduced in the requirements

specification may proliferate through the following phases of the software life cycle making the software difficult to understand, test, modify, and maintain.

White and Gustafson [13] have shown that the complexity of the software design and the complexity of the programs developed from that design correlate in terms of some complexity metrics. Halstead's complexity metrics were applied to both the Warnier-Orr diagrams of the software and the PL/I code generated from the diagrams. Halstead's length and volume metrics were shown to have a fairly consistent relationship between the diagrams and programs.

Applying software metrics to the products of the early phases of the software life cycle may provide an early indication of the quality of the software being developed [9]. Software metrics, which may be used to predict characteristics of the software product of later software life cycle phases, would be particularly useful. Problems and errors which are detected early in the software life cycle are less costly to fix than those detected later in the software life cycle.

#### 1.5 Software Metrics and Their Uses

Software metrics are a quantitative measure of some characteristic of the software. Boehm et al [2] have defined eleven (11) characteristics which should be measured. Understandability, completeness, conciseness,

portability, consistency, maintainability, testability, usability, reliability, structuredness, and efficiency are characteristics for which they feel software should be measured.

Software metrics can provide useful information throughout the software life cycle[5]. Software metrics which may be used to predict something about the software product (i.e. complexity of the code to be produced) or resources required by the development process (i.e. size, cost, or estimated time) would be particularly useful. Three valuable uses of software metrics are: 1) management tools, 2) measures of the quality of the software, and 3) feedback to software developers[5].

Software metrics used as management tools include those developed to measure the cost and size of projects as well as those which measure productivity [5]. In the former case, the cost and size metrics help predict the resources which will be required to complete the project. In the latter case, the productivity metrics evaluate the development process which is being managed.

Software quality metrics may be used by the organization requesting the software as well as by the software developers [5]. For example, the requesting organization may set a minimal mean time to failure to be achieved before accepting a software product. This same

quality metric could be used by the developers as a guide to potential problems during software validation and verification.

Feedback to developers is an important use of software metrics [5]. Complexity metrics, McCabe's cyclomatic complexity or Henry and Kafura's procedure complexity metric, may be used to indicate whether or not the software product may be passed to the next phase of the software life cycle. If the software product is considered to be too complex, it may be required to be redesigned prior to being passed to the subsequent software life cycle phase.

#### 1.6 Software Complexity Metrics

The software metrics of particular interest for this report are complexity metrics. Curtis [5] suggests the following definition of software complexity:

Complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software.

This definition allows for different operational definitions of the abstract construct - complexity - to be developed depending on the criterion under study.

Software complexity measures may be classified several different ways. A panel of software engineers has suggested the following classifications for complexity metrics: 1) product/process, 2) quality/invariant, and 3)

priori/posteriori[1]. The classification of a particular complexity metric will depend on the characteristic(s) the metric is attempting to quantify[1].

The process/product classification depends on what aspect of the software life cycle is to be measured - the software development methodology itself or the documentation developed by the methodology. Process metrics would be such measures as development time and number of errors. Product metrics would be based on such numbers as the number of decisions, number of interfaces, or size calculations[1].

The split between the quality and invariant classification of complexity metrics is based on whether or not the metric evaluates a software product as being good or bad. Quality metrics evaluate the software product as being good or bad relative to a specific model. A complexity metric which does not make a good or bad evaluation, such as effort in terms of lines of code, is classified as invariant[1].

The third classification for complexity metrics is priori or posteriori. A priori metric is one that is used to estimate and predict what the software product will eventually look like. A complexity metric which measures the software product after completion is classified as posteriori[1].

The metrics of Halstead, McCabe, and Henry and Kafura are product metrics. These complexity metrics are calculated for the completed program code or design specification. The software development process is not evaluated by the complexity metrics of concern for this report.

Complexity metrics may be used as quality metrics. A program or other software life cycle product may be classified as good or bad in relation to a complexity measure. Those software life cycle products whose complexity measure is greater than an acceptable number would be classified as being bad.

McCabe's complexity metric and the complexity measures of Henry and Kafura are posteriori while Halstead's metrics are split between priori and posteriori. Halstead's volume, length, program level, and effort are calculated after the software product is developed and therefore are classified as posteriori. Halstead's potential volume metric could be used as a priori metric to estimate the actual size of the final product[1].

#### 1.7 McCabe's Control Flow Metric

Complexity of software may be measured by assessing the control flow of the software. McCabe in particular defines complexity in relation to the decision structure of a program. In McCabe's complexity metric, the decision

structure is the number of linearly independent paths in a program.

McCabe has attempted to assess the complexity of software as it affects testability and reliability. Since his complexity metric is a representation of the control paths which must be exercised during testing, his complexity metric assesses the difficulty of testing a program. McCabe states that testing and reliability become greater problems in sections of code in which his complexity measure,  $V(G)$ , is greater than 10[10].

Experiments have shown that McCabe's complexity metric correlates to some important criteria of programs. Some of the criteria to which the complexity metric has been correlated are: 1) number of errors existing in a segment of code, 2) the time to find and repair errors in a segment of code, and 3) the ability to discriminate significantly between procedures with errors and procedures without errors[11]. These experiments have all correlated McCabe's metric to some aspect of the program code.

#### 1.8 Halstead's Software Science Metrics

The metrics of Halstead's Software Science are derived from four (4) quantities. The four quantities include counts of unique operands and operators and frequency totals of operands and operators. From these four quantities, Halstead developed metrics for overall program length,

potential smallest volume of an algorithm, actual volume of an algorithm in a particular language, program level, language level, programming effort, and programming development time.

The four (4) quantities, upon which the Software Science metrics are based, are derived from the operands and operators. An operand is defined by Halstead as a variable or a constant. An operator according to Halstead is that which can either change the value of an operand or change the order in which the operand is changed.

Halstead's Software Science is a composite measure of software complexity. Several of his metrics contribute to the abstract construct of complexity. Halstead did not propose one metric which alone measures complexity.

Halstead's metrics which calculate size contributes to the concept of program complexity. The length and volume metrics are measures which indicate the size of the program being evaluated. The size of a program contributes to complexity in terms of the number of operators and operands which must be understood and remembered.

The programming level and programming effort metrics also contribute to the measure of program complexity. Programming level is a measure of the difficulty of understanding a program. The number of mental discriminations required to generate a program is measured

by Halstead's programming effort metric.

Halstead's size metrics may be considered to predict the program characteristics of understandability and maintainability. Large programs are more difficult to understand and maintain because of the numbers of operators and operands which must be comprehended, remembered, and modified.

Halstead's metrics have been subjected to a considerable amount of research and experimentation. Halstead's metrics have received attention in terms of their theoretical derivation as well as validation experiments. Coulter [4] has questioned Halstead's application of cognitive psychology results to software science while Fitzsimmons and Love[7] have shown correlations between estimates and actual results calculated using Halstead's metrics.

Coulter feels that Halstead has not accurately applied the theories and findings of cognitive psychology to his Software Science. Halstead has used cognitive psychology in three areas of his Software Science. The limit placed on the number of parameters which should be passed to a module, the number of elementary mental discriminations required to write a program, and the use of the Stroud number are the areas in which Halstead has turned to cognitive psychology. Although Coulter points out discrepancies between the way

Halstead has applied some of the cognitive psychology theories, experiments continue to show that Halstead's metrics provide some valuable correlations.)

Halstead's programming effort metric has been shown to correlate with the number of errors reported for a program. Fitzsimmons and Love point to the linear relationship which Funami and Halstead found to hold for effort and reported errors. The number of errors reported in a program increases as the amount of effort required to produce the program increases.

Fitzsimmons and Love[7] also cite experiments which indicate that Halstead's estimated time metric provides useful correlations. The estimated time required for generating a program is shown to correlate to the actual time required to generate the program. The estimated time metric may be useful for determining the amount of time a project should require[3].

#### 1.9 Henry and Kafura's Information Flow Metrics

Software complexity may be quantified by observing the flow of information among software system components. Henry and Kafura [9] have developed complexity metrics based on the measurement of information flow between components of a system. Measures for procedure complexity, module complexity, and module coupling have been developed.

Henry and Kafura's procedure complexity metric is based on length (number of non comment source lines), fan-in, and fan-out. Fan-in is the number of local flows into a procedure plus the number of data structures from which a procedure receives information. Fan-out is defined to be the number of local flows from a procedure plus the number of data structures which a procedure updates.

The length component of the procedure complexity metric of Henry and Kafura is a weak factor of the metric [9]. An estimate of the length may be used when applying the metric to a software design. Additionally, Henry and Kafura state that the length may be eliminated from the metric without significant loss of accuracy.

The procedure complexity metric of Henry and Kafura may be used to: 1) show procedures which lack functionality, 2) show stress points of a system, and 3) show inadequate refinement of a procedure [9]. A high degree of fan-in and fan-out reveals a large number of connections between a procedure and its environment. This large number of connections between a procedure and its environment makes change difficult to implement and may indicate the procedure performs more than one function.

The second complexity metric of Henry and Kafura measures module complexity. Henry and Kafura define a module to be those procedures which either directly update

or directly retrieve information from a data structure [9]. The complexity of a module is the sum of the complexities of the procedures within a module.

Henry and Kafura's complexity metrics have been correlated to changes to a software product [9]. There is a strong correlation between procedures needing change and procedure complexity. Procedures, which are in more than one module, are more likely to require change.

#### 1.10 Scope of the Implementation

This implementation project applies the complexity metrics of Halstead and the procedure complexity measure of Henry and Kafura to the software requirements specification. This implementation calculates complexity metrics for the requirements specification and reports the results. No attempt is made by this implementation to provide validation of the complexity metrics calculated.

The application of complexity metrics to the software requirements specification allows software developers to use the design-measure-redesign cycle [9]. This cycle is shorter and less expensive than the design-implement-test-redesign cycle. The earlier problems are detected in the software life cycle the easier and less costly the problems are to fix.

The ERA format of the software requirements

specification describes the inputs and outputs of a software product and the functions to be performed by the software product. The functions of the software product are counted as operators for Halstead's complexity metrics. The inputs and outputs of the software product are counted as operands for Halstead's metrics and as fan-in or fan-out for Henry and Kafura's procedure complexity metric.

This implementation calculates Halstead's potential volume for each function of the software product specified in the software requirements specification. Halstead's potential volume for the entire requirements specification is calculated considering the functions to be the modules of the requirements specification. Halstead's estimated length, volume, programming level, programming effort, and estimated time metrics are applied using the counts of operators and operands for the entire requirements specification.

The procedure complexity metric of Henry and Kafura is calculated for each function described in the requirements specification. The procedure complexity metric is calculated using two values for the length factor. The procedure complexity of each function is calculated eliminating the length factor and using Halstead's potential volume calculated for each function.

McCabe's complexity metric is not part of this

implementation. The ERA format of the software requirements specification is an information flow representation of the software. McCabe's complexity measurement is based on control flow which is not readily derived from the grammar of the ERA format [10].

The software requirements specifications, which may be analyzed, must be provided in the entity-relationship-attribute format. The software requirements specification analyzer must parse the text of the document into the elementary counts on which the metrics are built. The program must count unique operators and operands and the frequency of operators and operands for the application of Halstead's metrics. Henry and Kafura's procedure complexity metric requires counting the inputs and outputs of the functions defined by the requirements specification.

The results of the complexity metric calculations are provided by each program. The results of the calculations of the complexity metrics as applied to the software requirements specification are presented but no interpretation of the results is attempted. This implementation is an exercise in whether or not the complexity metrics developed for program code can be applied to a form of the software requirements specification.

## CHAPTER 2

### 2. IMPLEMENTATION REQUIREMENTS AND DESIGN DECISIONS

#### 2.1 Overview

The complexity analyzer for software requirements specifications provides an estimate of the complexity of the software project to be developed. These complexity measures may be used in the decision making processes of management and software developers. Software complexity measures may be used for estimating the resources required for the remainder of the software development process, an acceptance criterion for passing the requirements specification on to the design phase, or an early indication of difficulties to be anticipated in the remaining software life cycle phases.

The input to the analyzer is a software requirements specification in the entity-relationship-attribute format. The ERA format of the software requirements specification is a text description of the meaning of the software. The analyzer must be able to apply the complexity metrics of Halstead and Henry and Kafura to this form of the requirements specification.

The input to the analyzer is required to be generated according to the grammar of the ERA format (Appendix 1). This requirement will ensure that the analyzers are lexically scanning a requirements specification which uses

valid entity and attribute types. An example of the entity-relationship-attribute format of the requirements specification is provided in Appendix 2.

The analyzer must scan the requirements specification for the entity types and attribute types which are used in the calculation of the complexity metrics. Some of the entity types and attribute types which are part of the ERA form of the requirements specification make the requirements specification more readable and understandable for the human being, but are not required for the automated calculation of the complexity metrics. For example, the Type entities of the requirements specification provide further description of input and output attributes of the Activity and Periodic function entities, but do not contribute to the counting of operators, operands, fan-in, or fan-out.

The requirements specification analyzer must be flexible. Flexibility is required in terms of the order and number of entities and attributes which are used in a requirements specification. Flexibility is also required in terms of the types of entities and attributes which are found in the ERA format of the requirements specification.

The output of these software requirements specification analyzer is the values of the complexity metrics in text format. The text format for the output allows the output to be used by software developers and managers as well as other

software development tools. At this time however there is no software development tool which requires the output of the analyzer.

The analyzer is to be developed on the KSU Computer Science department's Perkin-Elmer 8/32 computer system which runs Version 7 of the operating system. This analyzer of the software requirements specification is part of a research tool being developed by Dr. Gustafson of the Computer Science Department at KSU. The prototype of this research tool is to be implemented on a UNIX\* system.

## 2.2 Application of Halstead's Complexity Metrics

The analyzer of the software requirements specification, which applies Halstead's Software Science metrics, uses the Activity, Periodic\_function, Input, Output, Input\_output, and Data entities for its computations while ignoring the Type, Comment, and Constant entities. The Activity and Periodic function entities are essentially the operators of the ERA format of the requirements specification, while the Input, Output, Input\_output, and Data entities are the operands. The Type, Comment, and Constant entities provide further description of the attributes which are associated with the Activity and

---

\* UNIX is a trademark of AT&T Bell Laboratories.

Periodic function entities.

The Activity and Periodic function entities and verbs from the action and assertion attributes are considered to be operators. The Activity and Periodic function entities describe the functions that the software product being described is to perform. The action and assertion attributes also describe action that is to occur in the software product.

The input and output attributes associated with the Activity and Periodic function entities are the operands of the requirements specification. The input and output attributes are the items which are operated on by the Activity and Periodic function entities. The number of times a particular input or output is consumed or produced by an Activity or Periodic function must be tabulated.

The number of unique operators and operands and the frequencies of occurrence of the operators and operands are used to calculate seven (7) of Halstead's software complexity metrics. The potential volume of each Activity and Periodic function entity is computed and the result printed as well as the combined potential volume for the entities. Halstead's estimated length, volume, effort, level, and estimated time metrics are computed and the results printed.

### 2.3 Application of Procedure Complexity Metric

The Activity and Periodic\_function entities are considered to be the procedures of the software requirements specification for the application of Henry and Kafura's procedure complexity metric. The analyzer is required to store the names of the Activity and Periodic\_function entities. The Input, Output, Input\_output, Data, Type, Comment, and Constant entities are ignored by the analyzer for purposes of the procedure complexity metric application.

The inputs and uses attributes of the Activity and Periodic\_function entities are considered to be the fan-in to these entities. The fan-out from the Activity and Periodic\_function entities is the output attributes. The keywords, required\_mode, necessary\_condition, occurrence, comment, media, structure, type, units, subpart\_is, and subpart\_of attributes are ignored for the application of the procedure complexity metric of Henry and Kafura to the ERA format of the requirements specification.

The procedure complexity metric is calculated for each Activity and Periodic\_function entity. The procedure complexity metric is calculated two ways: first by eliminating the length factor (using 1 instead) and second by using Halstead's potential volume for the length of the Activity and Periodic\_function entities. The results of the application of Henry and Kafura's procedure complexity

metric to the Activity and Periodic\_function entities of the requirements specification are printed in a table.

#### 2.4 Design Decisions Influenced by the Requirements

The analyzer of the requirements specification is implemented in a "C" language program, which is table driven in order to facilitate change which may occur in the evolving ERA format of the software requirements specification. The currently acceptable entity types, attribute types, verbs to be searched for in the action and assertion attributes, and fan-in/fan-out types are placed in four tables. These tables are in a header file which is loaded at compile time.

The entity and attribute tables contain the name of the entity or attribute, a flag indicating whether or not the entity or attribute is to be ignored, and a flag indicating whether or not the stored entity or attribute should be counted as an operator or an operand. The table for the verbs is a list of the verbs to be searched for in the action and assertion attributes. The fan-in and fan-out tables contain the names of the attributes to be counted for Henry and Kafura's procedure complexity metric calculation.

## CHAPTER 3

### 3. DESIGN OF COMPLEXITY METRICS APPLICATION

#### 3.1 Overview

The input to the analyzer, which applies complexity metrics to the requirements specification, is a software requirements specification in the ERA format. The file containing the requirements specification is specified to the analyzer by redirecting the UNIX standard input to the file. The analyzer scans the input file until an end-of-file condition is encountered.

The output from the analyzer is the results of the complexity metric calculations. The output from the analyzer will be directed to the UNIX standard output. The user may choose to redirect the UNIX standard output to a file or pipe the output to a line printer.

The design of this implementation was approached from a data flow perspective. The input of the software requirements specification is transformed by the scanning functions into the elementary counts required for the metric calculations. The metrics are calculated and the values of the metrics are output.

The diagrams in Figure-1, Figure-2, and Figure-3 depict the afferent-transaction-efferent flow of the data through

the analyzer.

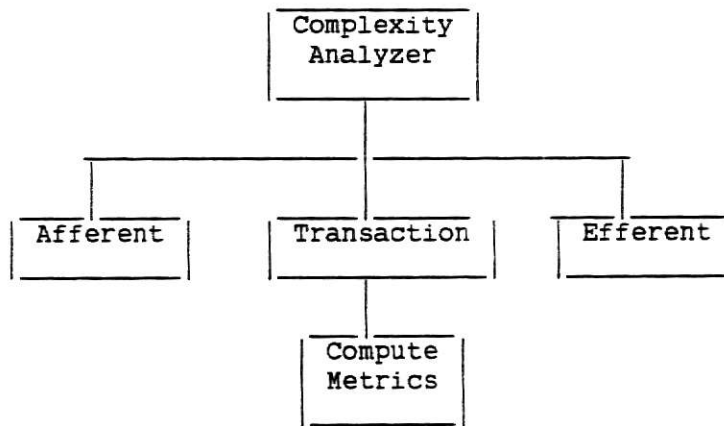


Figure 1 - Complexity Analyzer Data Flow Diagram

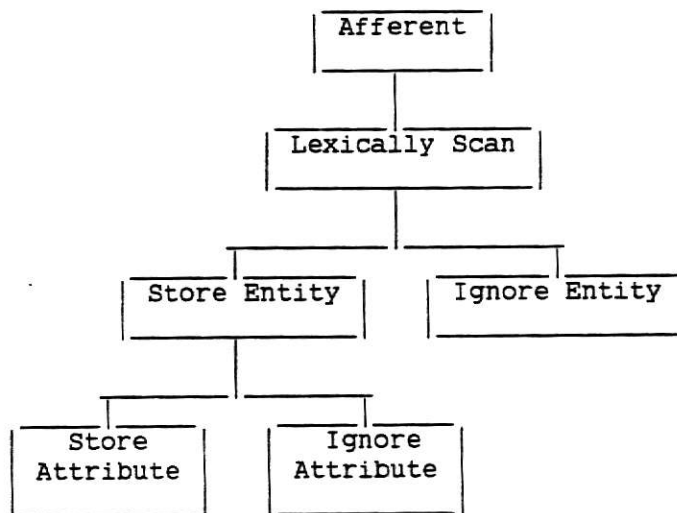


Figure 2 - Afferent Flow

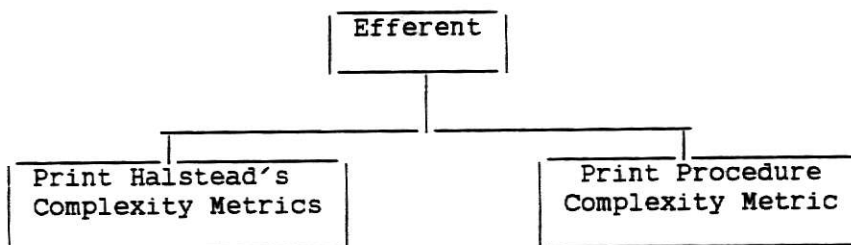


Figure 3 - Efferent Flow

### 3.2 Scanning the Specification Input

The analyzer must scan the text input of the ERA format of the requirements specification. The analyzer uses tables of the entity types, attribute types, verbs, and fan-in/fan-out types to determine which pieces of the ERA format are to be stored and which are to be ignored. The analyzer scans through the entire requirements specification before the complexity metrics are computed.

The software requirements specification is scanned for those pieces of the ERA format which contribute to the elementary counts required for the calculation of the complexity metrics. The number of unique operators and operands and the frequencies of the operators and operands are required for the calculation of Halstead's metrics. Counts of the fan-in and fan-out for each procedure are required for the calculation of Henry and Kafura's procedure complexity metric.

As the input requirements specification is scanned, lists of the unique operators and operands for the entire requirements specification and their frequencies are generated. The specification operator and operand lists are arrays of structures, which have a character array name element and an integer count element. The numbers of unique operators and unique operands are indicated by the subscripts of the operator and operand arrays of structures

upon completion of scanning the requirements specification.

The operators for Halstead's metrics are the Activity and Periodic\_function entities and verbs which are used in the phrases of the action and assertion attributes. The Input, Output, Input\_output, and Data entities are counted as operands. The input, output, and uses attributes of the Activity and Periodic\_function entities are also counted as operands.

The Activity and Periodic\_function entities are counted as the procedures for the application of Henry and Kafura's procedure complexity metric. The fan-in to the procedures is the input and uses attributes of the Activity and Periodic\_function entities. The output attributes of the entities are counted as the fan-out factor of the procedure complexity metric.

The Activity and Periodic\_function entities are the procedure entities of the requirements specification. The fan-in and fan-out lists are arrays of structures, which have a character array element for the procedure entity name and a character array element for the fan-in or fan-out name.

### 3.3 Calculation of the Complexity Metrics

Halstead's Software Science metrics are calculated from the number of unique operators, the number of unique

operands, frequency of operators, and frequency of operands. The potential volume for each stored entity, the combined potential volume for the entities, the volume of the specification, effort, level, and estimated time metrics are calculated.

Henry and Kafura's procedure complexity metric is calculated from the length, fan-in, and fan-out factors. This implementation computes the procedure complexity for the Activity and Periodic\_function entities using two values for the length factor. The length factor is eliminated (value of 1 is used) in one calculation, and Halstead's potential volume for the entity is used as the length estimate in the other calculation.

### 3.4 Output From The Analyzer

The output of the analyzer is a table of the individual potential volumes followed by the value of the combined potential volume. The five metrics, whose values are for the entire requirements specification, are listed in a table. A table of the procedure complexity metric values is generated for Henry and Kafura's metric.

## CHAPTER 4

### 4. IMPLEMENTATION OF A COMPLEXITY METRIC ANALYZER

#### 4.1 Overview

This implementation of the application of complexity metrics to the software requirements specification is a "C" language program. The analyzer was developed on the KSU Computer Science department's Perkin-Elmer 8/32 computer, which runs Version 7 of the UNIX operating system. The analyzer applies the complexity metrics of Halstead and the procedure complexity metric of Henry and Kafura to the software requirements specification.

The complexity metric analyzer program receives its input from the UNIX standard input and outputs the complexity metric values to UNIX standard output. The input to the complexity metric analyzer is a software requirements specification in the ERA format. The results of the complexity metrics are listed in three tables: 1) Halstead's individual module potential volume, 2) Halstead's metrics applied to the entire specification, and 3) Henry and Kafura's procedure complexity metric.

#### 4.2 Scanning the Requirements Specification

The analyzer scans the input requirements specification prior to the calculation of the complexity metrics. The

title of the requirements specification is read and used as part of the analyzer output heading. The function body of the requirements specification is scanned until the MODE\_TABLE is encountered.

Tables of the entity types, attribute types, verbs, and fan-in/fan-out types are used to determine which elements of the function body of the requirements specification should be stored and counted and which should be ignored. The five tables used by the analyzer are in a header file which is included prior to the main() function of the analyzer at compile time. The use of the header file allows the analyzer to be table driven and at the same time allows the user to be concerned only with directing the input to the analyzer and the output from the analyzer.

The tables of entity and attribute types are arrays of structures. The era\_table structure contains a type character array, an integer ignore flag, and an integer operator/operand flag. The verbs, fan-in, and fan-out tables are arrays of structures which contain a character array for the name of the verb to be searched for or attribute type to be considered fan-in or fan-out.

The entities and attributes, which are stored and counted for Halstead's complexity metrics, are stored in a list of operators or operands. Lists of the fan-in and fan-out for each Activity and Periodic\_function entity are

generated as the requirements specification is scanned. When an entity is ignored, the entire body of the entity is ignored.

An error message is sent to the UNIX standard error when an entity or attribute type is encountered in the requirements specification which is not in the entity or attribute table. When an invalid entity type is found, the entire body of the entity is ignored. If an invalid attribute type is encountered, the line of the specification containing the invalid attribute type is ignored, but the other attributes associated with the entity being scanned are not ignored.

#### 4.3 Calculation of the Complexity Metrics

Halstead's individual module potential volume, combined potential volume, volume, estimated length, programming level, programming effort, and estimated time metrics are calculated by the analyzer. The counts of the unique operators and operands and the frequencies of the operators and operands for the entire requirements specification are used for the calculation of the volume, estimated length, programming level, programming effort, and estimated time metrics. The values computed for these five metrics are listed in a table which is directed to standard output.

Halstead's individual module potential volume is calculated for each Activity and Periodic\_function entity.

The Activity and Periodic\_function entities are considered to be the modules of the requirements specification for the application of Halstead's module potential volume. Halstead's combined potential volume for the entire requirements specification is calculated from the individual potential volumes of the Activity and Periodic\_function modules of the requirements specification.

Henry and Kafura's procedure complexity metric is calculated for each procedure of the requirements specification. The procedures of the requirements specification are the Activity and Periodic\_function entity types. Each of these entity types describes a function or procedure the software product described by the requirements specification is to perform.

The procedure complexity metric of Henry and Kafura is computed using two values for the length factor of the metric. The length factor is eliminated in one calculation of the procedure complexity metric. Halstead's individual potential volume, which was calculated for the Activity and Periodic\_function entities, is used as the estimated length of the procedure for the second calculation of the procedure complexity.

#### 4.4 Testing the Analyzer

The analyzer, which applies complexity metrics to the requirements specification, was tested using a variety of ERA format requirements specifications. The size of the requirements specifications were varied by the number of entities in the function body of the requirements specification. The size of the requirements specification was also varied by the number of attributes associated with the entity types.

The detection of invalid entity and attribute types in the requirements specification was tested. Entity and attribute types, which are not valid according to the BNF of the ERA format, were placed in requirements specifications. The display of the expected error messages and proper handling of the remainder of the requirements specification were tested with the invalid entities and attributes.

## CHAPTER 5

### 5. CONCLUSIONS AND EXTENSIONS

The application of complexity metrics to the software product early in the software life cycle allows software developers and the managers of software developers to have an early indication of the quality of the software product under development. The measurement of software quality early in the software life cycle allows software developers to use the design-measure-redesign cycle during the development of software products. The design-measure-redesign cycle is shorter and less expensive than the design-implement-test-redesign cycle, which is frequently used during software development.

The complexity measures of Halstead and the procedure complexity metric of Henry and Kafura may be applied to the ERA format of the requirements specification. The ERA format of the requirements specification lends itself well to the application of these complexity metrics. The function body of the ERA format of the requirements specification may be broken down to the elementary counts used by these complexity metrics.

Whether or not the results of the application of Halstead's complexity metrics to the software requirements specification predict the complexity of the software design

specification and program code generated from the requirements and design specifications should be pursued. The work of White and Gustafson [13] showed correlations between some of Halstead's complexity metrics applied to the Warnier-Orr format of the design specification and the program code generated from these design specifications. A similar experiment could be performed tracking the values of the complexity metrics calculated for the requirements specification, design specification, and program code.

It would also be interesting to see whether or not the procedure complexity metric of Henry and Kafura applied to the requirements specification correlates to those procedures requiring change during maintenance. Henry and Kafura found that their procedure complexity metric applied to the program code correlated to the changes made to procedures.

## REFERENCES

- [1] Basili, Victor R. "Quantitative Software Complexity Models: A Panel Summary". Tutorial On Models and Metrics for Software Management and Engineering , pp.232 Computer Society Press, Long Beach, 1980.
- [2] Boehm, B. W. et al. Characteristics of Software Quality Elsevier North-Holland, New York, 1978.
- [3] Cook, Michael L. "Software Metrics: An Introduction and Annotated Bibliography" Software Engineering Notes , April 1982, vol. 7, no. 2, pp.41
- [4] Coulter, Neal S. "Software Science and Cognitive Psychology" IEEE Transactions on Software Engineering , March 1983, vol. SE-9, pp.166
- [5] Curtis, Bill "Measurement and Experimentation in Software Engineering", Proceedings of the IEEE , September 1980, Vol. 68, No. 9, pp. 1144
- [6] Curtis, Bill, Sylvia B. Sheppard, Phil Milliman, M.A. Borst, and Tom Love "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and Mc Cabe Metrics" IEEE Transactions on Software Engineering , March 1979, vol. SE-5, pp.96
- [7] Fitzsimmons, Ann, and Tom Love "A Review And Evaluation Of Software Science" ACM Computing Surveys , March 1978, vol. 10, pp.3
- [8] Halstead, Maurice H. Elements of Software Science Elsevier North-Holland, New York, 1977
- [9] Henry, Sallie and Dennis Kafura "Software Structure Metrics Based on Information Flow" IEEE Transactions on Software Engineering , September 1981, vol. SE-7, pp.510
- [10] Mc Cabe, Thomas J. "A Complexity Measure" IEEE Transactions of Software Engineering , December 1976, vol. SE-2, pp.308
- [11] Potier, D., J.L. Albin, R. Ferreol, and A. Bilodeau "Experiments with Computer Software Complexity and Reliability" 6th International Conference On Software Engineering , pp.94, Computer Society Press, Long Beach, 1982.
- [12] Pressman, Roger S. Software Engineering: A Practitioner's Approach McGraw-Hill, New York, 1982

- [13] White, B. and D. Gustafson, "Application Of Halstead's Complexity Measures To Programs Designed With Warnier-Orr Diagrams", Symposium on Application and Assessment of Automated Tools for Software Development, San Francisco, 1983.
- [14] Woodward, Martin R., Michael A. Hennell, and David Hedley "A Measure of Control Flow Complexity in Program Text" IEEE Transactions on Software Engineering, January 1979, vol. SE-5, pp.45-50
- [15] Zelkowitz, Marvin V. "Perspectives on Software Engineering" Computing Surveys, Vol. 10, No. 2, June 1978.

## Appendix 1

### BNF For The ERA Requirements Specification

#### General Description

The era specification will consist of a set of frames. The order of the frame is not fixed. Each frame will contain information about one entity. Each frame will start on a newline. The first line in the frame will contain the keyword that describes the type of the entity and the name of the entity. The first letter in the type is capitalized. The type and the name are separated by a colon. At least one blank line will separate each frame.

The information in a frame is generally in the form of relations between this entity and other entities. Some of the information is in the form of attributes. An attribute gives information about this entity without referring to other entities. The order of these relations/attributes is not fixed.

Each relation/attribute is specified by a keyword that specifies the relation/attribute and its value. The value is either the name of the entity that has that relation or a text description of the attribute value. A colon separates the keyword and its value. Each relation/attribute starts on a new line. If a relation/attribute continues on to another line, the continuation line starts with a blank field followed by a colon. Multiple occurrences of a relation/attribute is represented by multiple occurrences of the keyword.

#### Entity Types

These entity types are not fixed. Additional entity types may be defined in the future. All entity types will start with a capital letter.

Activity  
Type  
Input  
Output  
Periodic\_function  
Input\_output  
Data  
Constant  
Comment

\* additional entity types may be added at any time

#### Relations/Attributes

## Appendix 1

keywords  
input  
output  
required\_mode  
necessary\_condition  
occurrence  
assertion  
action  
comment  
media  
structure  
type  
units  
subpart\_is  
subpart\_of  
uses

\* additional entity types may be added at any time

### Syntax Description

```
<era_spec> ::=
    <era_title> <era_body> <mode_table>

<era_title> ::=
    PROCESS : <text>

<era_body> ::=
    <frame> | <frame> <era_body>

<frame> ::=
    <NL> <NL> <frame_header> <frame_body>
    | <NL> <NL> Comment : <text_lines>

<frame_header> ::=
    <i_o_data_header> : <i_o_data_name>
    | <function_header> : <CAPITAL_WORD>

<i_o_data_header> ::=
    Type | Input | Output | Input_output | Data
    | Constant | <CAPITAL_WORD>

<function_header> ::=
    Activity | Periodic_function | <CAPITAL_WORD>

<frame_body> ::=
    <relation> | <relation> <frame_body>

<relation> ::=
    <NL_B> <relation_type> : <relation_value>

<relation_type> ::=
    keywords | input | output | required_mode
    | necessary_condition | occurrence | assertion
```

## Appendix 1

```

    | action | comment | media | structure | type
    | units | subpart_is | subpart_of | uses | <WORD>

<relation_value> ::=
    <text_lines> | <structure>

<structure> ::=
    <struct> | <struct> <NL_B> : <structure>

<struct> ::=
    <name> | <text> | <name> <structure>
    | <text> <structure>

<name> ::=
    <mode_name> | <i_o_data_name>

<i_o_data_name> ::=
    $ <WORD> $

<mode_name> ::=
    * <WORD> *

<mode_table> ::=
    <NL> <NL> MODE_TABLE <mode_list> <initial_mode>
    <transition_body>

<mode_list> ::=
    <mode> | <mode> <mode_list>

<mode> ::=
    <NL_B> Mode : <mode_name>

<initial_mode> ::=
    <NL> <NL_B> Initial_Mode : <mode_name>

<transition_body> ::=
    <NL> <NL_B> Allowed_Mode_Transitions :
    <transition_list>

<transition_list> ::=
    <transition> | <transition> <transition_list>

<transition> ::=
    <NL_B> <event> : <mode_name> -> <mode_name>

<event> ::=
    <i_o_data_name>
    | <i_o_data_name> = ' <text> '
    | <function_header>

<text_lines> ::=
    <text> | <text> <text_cont>

```

## Appendix 1

```
<text> ::=
    <WORD> | <WORD> <text>

<text_cont> ::=
    <NL_B> : <text> | <NL> : <text> <text_cont>

<NL> ::=
    '0 | '0 <NL>

<NL_B> ::=
    <NL> ' ' ' '
```

### LEXICAL SCANNER INFORMATION

Tokens used in the productions above may begin with <char> or one of the following characters: \*\$,':-={}. Blanks can delimit tokens as well.

The following tokens are important above:

```
<WORD>          ::= <char> | <char> <WORD>
<CAPITAL_WORD> ::= <capital_letter> <WORD>

<char> ::=
    <lower_case_char> | <symbol>

<lower_case_char> ::=
    a | b | ... | z | 0 | 1 | ... | 9

<symbol> ::=
    # | % | & | ( | ) | ? | _

<capital_letter> ::=
    A | B | ... | Z
```

There exists a set of "reserved word" tokens which includes:  
{keyboard,crt,internal,secondary\_storage,NONE,every,mode}.

## Appendix 2

### Example of an ERA Requirements Specification

PROCESS : Requirements specification for the chess program

Comment : as of 6/25/84 1:40 in ksu832:/usrb/we/era

:  
: comment on specification:  
: Not all of the activities necessary for this  
: program to be implemented are included in this  
: description. Some activities are not included if  
: their activities were determined by the other  
: activities. The activity of interpreting the user's  
: command was not included.

Type : \$piece\$

structure : a string from the set {Kr,Kk,Kb,K,Q,Qb,Qk,Qr,p}

Type : \$rank\$

structure : a string from the set {1,2,...8}

Type : \$position\$

structure : \$piece\$ \$rank\$

Type : \$piece\_position\$

structure : \$piece\$ ',' \$position\$

Type : \$board\_matrix\$

structure : array[1..8,1..8] of \$piece\$ OR ' '

Input : \$board\_description\$

media : keyboard

structure : 'white' set of \$piece\_position\$

structure : 'black' set of \$piece\_position\$

structure : 'end'

Input : \$name\_of\_game\$

media : keyboard

structure : 1 to 20 alphanumeric characters

Input : \$new\_user\_input\$

media : keyboard

structure : any string

Input\_output : \$stored\_board\$

media : secondary\_storage

structure : information to recreate the board configuration

Input\_output : \$chess\_board\$

media : internal

structure : \$board\_matrix\$

## Appendix 2

```
Input : $move$
      media      : keyboard
      structure  : 'm' $position$ '-' $position$

Input : $display_board$
      media      : keyboard
      structure  : 'display'

Input : $create$
      media      : keyboard
      structure  : 'create'

Input : $concede$
      media      : keyboard
      structure  : 'concede'

Input : $store$
      media      : keyboard
      structure  : 'store' $name_of_game$

Input : $retrieve$
      media      : keyboard
      structure  : 'retrieve' $name_of_game$

Input : $mate_stale$
      media      : keyboard
      structure  : 'stalemate'

Input : $limit_time$
      media      : keyboard
      structure  : 'time_limit'

Input : $out_time$
      media      : keyboard
      structure  : 'time_out'

Input : $check_input$
      media      : keyboard
      structure  : 'input_check'

Output : $status$
      media      : crt
      structure  : string from the set {'your move','check',
      : 'checkmate','concede'}

Output : $board_display$
      media      : crt
      structure  : visually oriented display of current chess board

Output : $syntax_error$
      media      : crt
      structure  : <cr> 'illegal, try again'
```

## Appendix 2

```
Output : $store_message$
  media      : crt
  structure  : 'board stored'
  structure  : 'storage failed'

Output : $retrieve_message$
  media      : crt
  structure  : $name_of_game$ 'retrieved'
  structure  : 'retrieval failed'

Output : $stalemate$
  media      : crt
  structure  : 'stalemate occurred'

Output : $time_warning$
  media      : crt
  structure  : 'this is a warning - 5 minutes elapsed'

Output : $time_out$
  media      : crt
  structure  : 'too much time - game over'

Output : $move_message$
  media      : crt
  structure  : <cr>
  structure  : 'illegal move'

Output : $computer_move_message$
  media      : crt
  structure  : 'computer moves from' $position$ 'to' $position$

Activity : Initialize_board
  keywords      : Standard_board,Initialize,Place_pieces
  input         : NONE
  output        : $chess_board$
  required_mode : *START*
  necessary_condition : $start$
  assertion     : The output board is a correct
                  : representation of the standard
                  : starting configuration for chess

Activity : Create_special_board
  keywords      : Assign_positions,Place_pieces
  input         : $board_description$
  output        : $chess_board$
  required_mode : *START*
  necessary_condition : $create$

Activity : Store_board
  keywords      : Store_game_status,Save_board
  input         : $name_of_game$
  input         : $chess_board$
  output        : $store_message$
```

## Appendix 2

required\_mode : \*NORMAL\*  
necessary\_condition : \$store\$  
assertion : the game is stored in file  
: '\$name\_of\_game\$'

### Activity : Retrieve\_board

keywords : Retrieve\_board  
input : \$name\_of\_game\$  
output : \$chess\_board\$  
output : \$retrieve\_message\$  
required\_mode : \*START\*  
necessary\_condition : \$retrieve\$  
assertion : Retrieves game stored in file  
: '\$name\_of\_game\$' if successful

### Activity : Validate\_user\_move

keywords : Check\_move,Check\_m\_status,  
keywords : Move\_validation  
input : \$chess\_board\$  
input : \$move\$  
output : \$move\_message\$  
required\_mode : \*NORMAL\*  
assertion : If the move is illegal,  
: the mode changes to \*ILLEGAL\*

### Activity : Computer\_Move

comment : used to be Move  
keywords : Select\_move,Select\_status  
input : \$chess\_board\$  
output : \$chess\_board\$  
output : \$computer\_move\_message\$  
output : \$status\$  
required\_mode : \*NORMAL\*  
action : mode may change to \*END\*  
: if \$status\$ = 'checkmate' OR 'concede'

### Activity : Update\_board

keywords : Update\_position,Update\_status  
input : \$chess\_board\$  
input : \$move\$  
output : \$chess\_board\$  
required\_mode : \*NORMAL\*

### Activity : Display\_board

keywords : Display  
input : \$chess\_board\$  
output : \$board\_display\$  
required\_mode : \*NORMAL\*  
required\_mode : \*END\*  
necessary\_condition : \$display\_board\$

### Periodic\_function : Stalemate

required\_mode : \*NORMAL\*

## Appendix 2

```

occurrence      : whenever a board configuration is
                  : repeated 3 times
input           : $mate_stale$
output          : $stalemate$
action          : change mode to *END*

```

```

Periodic_function : Time Limit
required_mode     : *NORMAL*
occurrence        : whenever the user response time
                  : exceeds 5 minutes
input            : $limit_time$
output           : $time_warning$
action           : NONE

```

```

Periodic_function : Time Out
required_mode     : *NORMAL*
occurrence        : whenever the user response time
                  : exceeds 10 minutes
input            : $out_time$
output           : $time_out$
action           : change mode to *END*

```

```

Periodic_function : Input_Check
required_mode     : every mode
occurrence        : whenever user input does not match
                  : allowed syntax
input            : $check_input$
output           : $syntax_error$
action           : change mode to *ILLEGAL*

```

### MODE\_TABLE

```

Mode : *ILLEGAL*
Mode : *NORMAL*
Mode : *START*
Mode : *END*

```

```
Initial_Mode : *START*
```

### Allowed\_Mode\_Transitions :

```

$create$          : *START* -> *NORMAL*
$start$           : *START* -> *NORMAL*
$retrieve$        : *START* -> *NORMAL*

$status$ = 'checkmate' : *NORMAL* -> *END*
$status$ = 'concede'   : *NORMAL* -> *END*
$stalemate$        : *NORMAL* -> *END*
$time_out$         : *NORMAL* -> *END*

$move_message$ = 'illegal move' : *NORMAL* -> *ILLEGAL*
$syntax_error$      : *NORMAL* -> *ILLEGAL*

$syntax_error$     : *START* -> *ILLEGAL*

```

## Appendix 2

|                    |                         |
|--------------------|-------------------------|
| \$syntax_error\$   | : *END* -> *ILLEGAL*    |
| \$new_user_input\$ | : *ILLEGAL* -> *NORMAL* |
| \$'<cr>'\$         | : *END* -> *START*      |

## Appendix 3

### Requirement Specification for the Implementation

PROCESS : Application of Complexity Metrics to Requirements

Comment : This is the requirements specification for the  
: application of complexity metrics to a software  
: requirements specification in the ERA format.

Activity : Read\_Spec\_Title

keywords : read title; print output heading  
input : \$line\_of\_spec\$  
output : \$analyzer\_output\_heading\$  
output : \$title\$  
required\_mode : \*START\*

Activity : Scan\_Spec

keywords : parse spec; scan for entities  
input : \$line\_of\_spec\$  
input : \$entity\_table\$  
output : \$line\_of\_spec\$  
output : \$eof\$  
required\_mode : \*START\*  
action : If entity type is invalid,  
: mode changes to \*ILLEGAL\*.

Activity : Store\_Entity

keywords : store entity for calculations  
input : \$line\_of\_spec\$  
input : \$entity\_table\$  
input : \$attribute\_table\$  
input : \$n1\_n2\_switch\$  
output : \$spec\_n1\_list\$  
output : \$spec\_n2\_list\$  
output : \$line\_of\_spec\$  
output : \$eof\$  
required\_mode : \*START\*  
action : If entity format is invalid,  
: mode changes to \*ILLEGAL\*.

Activity : Ignore\_Entity

keywords : entity type not used for calculations  
input : \$line\_of\_spec\$  
output : \$eof\$  
required\_mode : \*START\*  
comment : All attributes associated with the entity  
: are ignored.

Activity : Store\_Attribute

keywords : store attribute for calculations  
input : \$line\_of\_spec\$  
input : \$n1\_n2\_switch\$  
input : \$verb\_table\$

### Appendix 3

```

input      : $fan_in_table$
input      : $fan_out_table$
output     : $spec_n1_list$
output     : $entity_n1_list$
output     : $entity_in_list$
output     : $entity_out_list$
output     : $spec_n2_list$
required_mode : *START*
action     : If attribute format is invalid,
            : mode changes to *ILLEGAL*.

Activity : Ignore_Attribute
keywords  : attribute type is not used for calculations
input     : $line_of_spec$
required_mode : *START*

Periodic_function : Invalid_Entity
required_mode : *START*
occurrence   : whenever the entity type does not match
              : allowed syntax.
input        : $line_of_spec$
output       : $inv_ent_msg$
action       : change mode to *ILLEGAL*;
              : read through remainder of entity

Periodic_function : Invalid_Attrib
required_mode : *START*
occurrence   : whenever the attribute type does not match syntax
input        : $line_of_spec$
output       : $inv_attrib_msg$
action       : change mode to *ILLEGAL*

Activity : Entity_P_Vol
keywords  : calculate entity potential volume
input     : $entity_in_list$
input     : $entity_out_list$
output    : $mod_pot_vol$
output    : $pot_vol_msg$
required_mode : *NORMAL*

Activity : Comb_Pot_Vol
keywords  : calculate combined potential volume for entire spec
input     : $mod_pot_vol$
input     : $num_mods$
output    : $comb_pot_vol$
output    : $comb_p_vol_msg$
required_mode : *NORMAL*

Activity : Print_Halstead_Heading
keywords  : print heading for table of Halstead's metrics
input     : NONE
output    : $H_table_head$
required_mode : *NORMAL*

```

### Appendix 3

#### Activity : Calc\_Volume

keywords : calculate the specification volume  
input : \$spec\_length\$  
input : \$spec\_vocab\$  
output : \$spec\_volume\$  
output : \$spec\_vol\_msg\$  
required\_mode : \*NORMAL\*

#### Activity : Calc\_Est\_Length

keywords : calculate specification estimated length  
input : \$num\_spec\_n1\$  
input : \$num\_spec\_n2\$  
output : \$est\_length\$  
output : \$est\_len\_msg\$  
required\_mode : \*NORMAL\*

#### Activity : Calc\_Level

keywords : calculate programming level for the specification  
input : \$comb\_pot\_vol\$  
input : \$spec\_volume\$  
output : \$spec\_prgm\_level\$  
output : \$level\_msg\$  
required\_mode : \*NORMAL\*

#### Activity : Calc\_Effort

keywords : calculate effort in elementary mental discriminations  
input : \$spec\_volume\$  
input : \$spec\_prgm\_level\$  
output : \$effort\$  
output : \$effort\_msg\$  
required\_mode : \*NORMAL\*

#### Activity : Calc\_Est\_Time

keywords : Calculate estimated time required to understand  
: the specification  
uses : \$STROUD\$  
input : \$num\_spec\_n1\$  
input : \$num\_spec\_n2\$  
input : \$length\_n2\$  
output : \$est\_time\$  
output : \$est\_time\_msg\$  
required\_mode : \*NORMAL\*

#### Activity : Complexity Head

keywords : print heading for table of Henry and Kafura's  
: procedure complexity metric  
input : NONE  
output : \$hk\_head\_msg\$  
required\_mode : \*NORMAL\*

#### Activity : Calc\_HK\_Metric

keywords : calculate Henry and Kafura's complexity metric  
input : \$num\_ent\_in\$

### Appendix 3

```

input      : $num_ent_out$
input      : $mod_pot_vol$
output     : $num_ent_in$
output     : $num_ent_out$
output     : $hk_one$
output     : $hk_pot_vol$
output     : $hk_calc_msg$
required_mode : *NORMAL*
comment    : Complexity metric is calculated using one (1) for
            : length and using potential volume for length

Activity : Print_N1N2_Heading
keywords  : print heading for table of n1 and n2 values
input     : $num_spec_n1$
input     : $spec_n1_list$
input     : $num_spec_n2$
input     : $spec_n2_list$
output    : $length_n1$
output    : $length_n2$
output    : $spec_length$
output    : $n1n2_msg$
required_mode : *NORMAL*

Input_output : $line_of_spec$
media       : internal
structure   : 1 to 85 alphanumeric characters

Output : $analyzer_output_heading$
media   : crt
structure : 'Halstead's Software Science Analysis of'
         : $title$

Output : $title$
media   : internal
structure : 1 to 85 alphanumeric characters

Output : $eof$
media   : internal
structure : true OR false

Input : $entity_table$
media   : internal
structure : $e_table$

Type : $e_table$
structure : array of structure
         : $e_name$ $ignore_flag$ $n1_n2_flag$

Type : $e_name$
structure : 'Activity' OR 'Periodic function' OR 'Type' OR
           : 'Input' OR 'Output' OR 'Input_output' OR 'Data'
           : OR 'Comment' OR 'Constant'

```

### Appendix 3

Type : \$ignore\_flag\$  
structure : '0' OR '1'

Type : \$n1\_n2\_flag\$  
structure : '0' OR '1' OR '2'

Input : \$attribute\_table\$  
media : internal  
structure : array of structure \$a\_name\$ \$ignore\_flag\$  
: \$n1\_n2\_flag\$

Type : \$a\_name\$  
structure : 'keywords' OR 'input' OR 'output' OR 'required\_mode'  
: OR 'necessary\_condition' OR 'occurrence' OR  
: OR 'assertion' OR 'action' OR 'comment' OR 'media'  
: OR 'structure' OR 'type' OR 'units' OR 'subpart\_is'  
: OR 'subpart\_of' OR 'uses'

Input : \$n1\_n2\_switch\$  
media : internal  
structure : '1' OR '2' OR ':'

Output : \$spec\_n1\_list\$  
media : internal  
structure : array of structure \$name\$ \$count\$

Type : \$name\$  
structure : 1 to 40 alphanumeric characters

Type : \$count\$  
structure : integer

Input\_output : \$spec\_n2\_list\$  
media : internal  
structure : array of structure \$name\$ \$count\$

Input : \$verb\_table\$  
media : internal  
structure : array of \$v\_name\$

Type : \$v\_name\$  
structure : 'stored' OR 'Stored' OR 'retrieves' OR 'Retrieves'  
: 'changes' OR 'Changes' OR 'change' OR 'Change'

Input : \$fan\_in\_table\$  
media : internal  
structure : array [1..2] of \$fan\_in\_name\$

Type : \$fan\_in\_name\$  
structure : 'input' OR 'uses'

Input : \$fan\_out\_table\$  
media : internal

### Appendix 3

```

    structure : array [1..1] of $fan_out_name$

Type : $fan_out_name$
    structure : 'output'

Input_output : $entity_n1_list$
    media      : internal
    structure : $elist$

Type : $elist$
    structure : array [1..20] of structure $name$ array [1..10]
                : of $name$

Input_output : $entity_in_list$
    media      : internal
    structure : $elist$

Input_output : $entity_out_list$
    media      : internal
    structure : $elist$

Output : $inv_ent_msg$
    media      : crt
    structure : 'Invalid entity type detected in SRS.'
                : 'Entity line was: ' $line_of_spec$
                : 'Entire entity ignored.'

Output : $inv_attrib_msg$
    media      : crt
    structure : 'Invalid attribute type detected in SRS.'
                : 'Attribute line was: ' $line_of_spec$
                : 'Execution continues.'

Input_output : $mod_pot_vol$
    media      : internal
    structure : array [1..20] of $vmstar$

Type : $vmstar$
    structure : real number

Output : $pot_vol_msg$
    media      : crt
    structure : $name$ $vmstar$

Input : $num_mods$
    media      : internal
    structure : integer

Input_output : $comb_pot_vol$
    media      : internal
    structure : $vMstar$

Type : $vMstar$

```

### Appendix 3

```

structure : real number

Output : $comb_p_vol_msg$
  media      : crt
  structure  : 'Combined potential volume is ' $vMstar$

Output : $H_table_head$
  media      : crt
  structure  : 'Halstead's Metrics For Entire Specification'
              : 'Metric Name          Computed Value'
              : '-----'
Input_output : $spec_length$
  media      : internal
  structure  : integer
  comment    : Halstead's length, N.

Input : $spec_vocab$
  media      : internal
  structure  : real number
  comment    : Halstead's vocabulary;
              : number of unique operators and operands

Input_output : $spec_volume$
  media      : internal
  structure  : real number

Output : $spec_vol_msg$
  media      : crt
  structure  : 'Volume V' $spec_volume$

Input : $num_spec_n1$
  media      : internal
  structure  : real number

Input : $num_spec_n2$
  media      : internal
  structure  : real number

Output : $est_length$
  media      : internal
  structure  : real number

Output : $est_len_msg$
  media      : crt
  structure  : 'Estimated length N~' $est_length$

Input_output : $spec_prgm_level$
  media      : internal
  structure  : real number

Output : $level_msg$
  media      : internal

```

### Appendix 3

```

structure : 'Specification level L' $spec_prgm_level$

Output : $effort$
media    : internal
structure : real number

Output : $effort_msg$
media    : crt
structure : 'Programming effort E' $effort$

Constant : $STROUD$
media    : internal
structure : 18

Input_output : $length_n2$
media    : internal
structure : real number

Output : $est_time$
media    : internal
structure : real number

Output : $est_time_msg$
media    : crt
structure : 'Estimated time T~' $est_time$

Output : $hk_head_msg$
media    : crt
structure : 'Henry and Kafura's Procedure Complexity Metric'
          : 'Entity Name  Fan_in  Fan_out Length=1 Length=PV'
          : '-----  -----  -----  -----'

Input_output : $num_ent_in$
media    : internal
structure : array [1..20] of integer

Input_output : $num_ent_out$
media    : internal
structure : array [1..20] of integer

Output : $hk_one$
media    : internal
structure : real number
comment  : Henry and Kafura's metric using length equal to 1.

Output : $hk_pot_vol$
media    : internal
structure : real number
comment  : Henry and Kafura's metric using potential volume
          : for length.

Output : $hk_calc_msg$
media    : crt

```

### Appendix 3

```
structure : $name$ $num_ent_in$ $num_ent_out$ $hk_one$
           : $hk_pot_vol$
```

```
Output : $length_n1$
media   : internal
structure : real number
```

```
Output : $n1n2_msg$
media   : crt
structure : 'Table of Operators and Counts'
           : 'Operator Name      Counts'
           : '-----'
           : $spec_n1_list$
           : '-----'
           : $num_spec_n1$ $length_n1$
           : 'Table of Operands and Counts'
           : 'Operand Name      Counts'
           : '-----'
           : $spec_n2_list$
           : '-----'
           : $num_spec_n2$ $length_n2$
           : 'Entity Potential Volume Table'
           : 'Entity Name          v*'
           : '-----'
```

#### MODE\_TABLE

```
Mode : *ILLEGAL*
Mode : *NORMAL*
Mode : *START*
Mode : *END*
```

```
Initial_Mode : *START*
```

```
Allowed_Mode_Transitions :
```

```
$eof$           : *START* -> *NORMAL*

$inv_ent_msg$   : *START* -> *ILLEGAL*
$inv_attrib_msg$ : *START* -> *ILLEGAL*

$inv_ent_msg$   : *ILLEGAL* -> *START*
$inv_attrib_msg$ : *ILLEGAL* -> *START*

$hk_calc_msg$   : *NORMAL* -> *END*
```

## Appendix 4

### Header File Of Tables For Implementation

```

#define MAXENTITY      9
#define MAXATTRIB     16
#define MAXVERBS       8
#define MAXFIN         2
#define MAXFOUT        1

/* The structure of the tables of ERA entities */
/* and attributes consists of the name of the */
/* entity or attribute followed by an ignore */
/* flag, and an operator/operand indication. */
/* The ignore flag is set to 1 if the element */
/* of the era_table is to be ignored and set */
/* to 0 if the element is to be stored for the */
/* computations. The operator/operand flag is */
/* used to determine whether the stored */
/* element should be considered an operator or */
/* an operand. A 1 indicates an operator, a 2 */
/* indicates an operand, and a 0 is entered */
/* for elements which are ignored. */

struct era_table {
    char type[TYPESIZE];
    int ignore;
    int n1 n2;
} e_table[MAXENTITY] = {
    "Activity",0,1,
    "Periodic function",0,1,
    "Type",1,0,
    "Input",0,2,
    "Output",0,2,
    "Input_output",0,2,
    "Data",0,2,
    "Comment",1,0,
    "Constant",0,2},

a_table[MAXATTRIB] = {
    "keywords",1,0,
    "input",0,2,
    "output",0,2,
    "required_mode",1,0,
    "necessary_condition",1,0,
    "occurrence",1,0,
    "assertion",0,1,
    "action",0,1,
    "comment",1,0,
    "media",1,0,
    "structure",1,0,
    "type",1,0,
    "units",1,0,

```

#### Appendix 4

```
    "subpart_is",1,0,
    "subpart_of",1,0,
    "uses",0,2};

/* The following table contains common verbs */
/* which are found in the action and */
/* assertion attributes in the ERA spec. */

struct verbs {
    char verb[MAXNAME];
} v_table[MAXVERBS] = {
    "stored",
    "Stored",
    "retrieves",
    "Retrieves",
    "changes",
    "Changes",
    "change",
    "Change"};

/* The following table contains the names */
/* of the attributes which are considered */
/* to be fan-in and fan-out for Henry and */
/* Kafura's complexity metric calculation */
/* from the ERA specification. */

struct fan_table {
    char ftype[MAXNAME];
} fan_in[MAXFIN] = {
    "input",
    "uses"},
fan_out[MAXFOUT] = {
    "output"};
```

## Appendix 5

### Source Code For Implementation

```
/* This program calculates Halstead's Software */
/* Science metrics and Henry and Kafura's */
/* Procedure Complexity metric for a software */
/* requirements specification. The input to */
/* this software development tool is a */
/* requirements specification of the ERA type. */
/* */
/* The ERA (entity-relationship-attribute) */
/* format is a keyworded text description of */
/* the software to be developed. */

#include <stdio.h>
#include <math.h>

#define TYPESIZE 24
#define MAXNAME 40
#define MAXOPERSPEC 50
#define MAXOPERENT 10
#define MAXLINE 85
#define MAXWORD 80
#define PROCESS 8
#define MODE_T 11
#define MODULES 20
#define STROUD 18

#include "comp_tables.h"

/* The following lists are used to store the */
/* operators and operands found in the spec */
/* and their occurrence count. */

struct slist {
    char name[MAXNAME];
    int count;
} spec_n2[MAXOPERSPEC],
  spec_n1[MAXOPERSPEC];

struct elist {
    char ename[MAXNAME];
    char oper[MAXOPERENT][MAXNAME];
} ent_n1[MODULES],
  ent_in[MODULES],
  ent_out[MODULES];

int num_s_n1, num_s_n2, num_mod,
    num_e_n1[MODULES], num_e_in[MODULES],
    num_e_out[MODULES];

char title[MAXLINE], process[PROCESS] = {"PROCESS"},
    mode_t[MODE_T] = {"MODE_TABLE"}, dollar = '$';
```

## Appendix 5

```
double mod_p_vol[MODULES];

main()
{
    int i, s, N1, N2, N;
    double vMstar, V, L, vocab;

    scan_spec();
    vocab = num_s_n1 + num_s_n2;
    print_nln2(&N1,&N2,&N);
    ent_p_vol();
    vMstar = comb_p_vol();
    m_heading();
    V = volume(N,vocab);
    n_hat();
    level(vMstar,V,&L);
    effort(V,L);
    e_time(N2);
    hk_heading();
    calc_hk();
}

/* This function lexically scans the ERA spec */
/* for entity types. If the ignore flag is */
/* true for the entity, ignore_entity() is */
/* called. If the ignore flag is false, then */
/* store_entity() is called. If the entity */
/* type is not found in the entity_table, */
/* the inv_entity() function is called. */

scan_spec()
{
    char entity[TYPESIZE], specline[MAXLINE], *lptr;
    int i, match, s;

    r_title();
    while ((lptr=fgets(specline,MAXLINE,stdin)) != NULL)
    {
        if (specline[0] == '\n') continue;
        s = sscanf(specline,"%s",entity);
        if (s == 1)
        {
            s = strcmp(mode_t,entity);
            if (s == 0) break;
            for (i=0; i<MAXENTITY; i++)
            {
                match = 0;
                s = strcmp(e_table[i].type,entity);
                if (s == 0)
                {
                    match = 1;
                    if (e_table[i].ignore)
                        ignore_entity(specline);
                }
            }
        }
    }
}
```

## Appendix 5

```

        break;
    }
    else
    { store_entity(specline,
                    e_table[i].n1_n2);
      if (e_table[i].n1_n2 == 1) ++num_mod;
      break;
    }
}
}
if (!match)
    inv_entity(specline);
}
return;
}

/* This function reads the title of the era spec */
/* from the specification file. The heading for */
/* the potential volume table is printed as well */
/* as the title of the specification. */

r_title()
{
    char proc[PROCESS], colon[2], *lptr, blank = ' ';
    int i, s;

    s = scanf("%s %[:]", proc, colon);
    if (s != 2)
        fprintf(stderr,
                "Error reading PROCESS :\n");
    if ((lptr=fgets(title,MAXLINE,stdin)) == NULL)
        fprintf(stderr,
                "Error reading title\n");
    else s = strlen(title);
    printf("\n\n");
    for (i=0; i<20; i++)
        printf("%c", blank);
    printf("Halstead's Software Science Analysis of\n");
    for (i=0; i<(80 - s)/2; i++)
        printf("%c", blank);
    printf("%s\n\n", title);
    return;
}

/* This function stores the names of the entities */
/* in the list of operators for the spec. The */
/* number of occurrences of the operator is also */
/* updated. */

store_entity(line,sw)
    char line[];
    int sw;

```

## Appendix 5

```

{
    char *lptr, attrib[TYPESIZE], colon[2], name[MAXNAME];
    int cont,i, match, s;

    cont = 0;
    colon[0] = ':';
    colon[1] = '\0';
    s = sscanf(line,"%*s %*[:] %s",name);
    if (s == 1)
    {
        switch(sw + '0')
        {
            case '1':
                match = 0;
                for (i=0; i<MAXOPERSPEC; i++)
                {
                    s = strcmp(spec_n1[i].name,name);
                    if (s == 0)
                    {
                        match = 1;
                        ++spec_n1[i].count;
                        break;
                    }
                }
                if (!match)
                {
                    strcpy(spec_n1[num_s_n1].name,name);
                    ++spec_n1[num_s_n1++].count;
                    strcpy(ent_n1[num_mod].ename,name);
                    strcpy(ent_in[num_mod].ename,name);
                    strcpy(ent_out[num_mod].ename,name);
                }
                break;

            case '2':
                match = 0;
                for (i=0; i<MAXOPERSPEC; i++)
                {
                    s = strcmp(spec_n2[i].name,name);
                    if (s == 0)
                    {
                        match = 1;
                        ++spec_n2[i].count;
                        break;
                    }
                }
                if (!match)
                {
                    strcpy(spec_n2[num_s_n2].name,name);
                    ++spec_n2[num_s_n2++].count;
                }
                break;
            default :

```

# Appendix 5

```

        break;
    }
}
else inv_entity(line);
while ((lptr = fgets(line,MAXLINE,stdin)) != NULL)
{
    if (line[0] == '\n') break;
    s = sscanf(line,"%s",attrib);
    if (s == 1)
    {
        s = strcmp(colon,attrib);
        if (s == 0)
            if (a_table[cont].ignore)
            {
                ignore_attrib();
                continue;
            }
            else
            {
                store_attrib(line,colon[0]);
                continue;
            }
        }

    match = 0;
    for ( i=0; i<MAXATTRIB; i++)
    {
        s = strcmp(a_table[i].type,attrib);
        if (s == 0)
        {
            match = 1;
            if (a_table[i].ignore)
            {
                ignore_attrib();
                cont = i;
                break;
            }
            else
            {
                store_attrib(line,a_table[i].n1_n2
                               + '0');
                cont = i;
                break;
            }
        }
    }
    if (!match)
        inv_attrib(line);
}
return;
}

/* This function stores the names and counts of */
/* the operands and operators of the entities */
/* and entire specification. */

```

## Appendix 5

```

store_attrib(ln,i)
    char ln[];
    int i;
{
    char word[MAXWORD], *lptr, name[MAXNAME], fname[MAXNAME];
    int j, k, match, s;

    switch (i)
    {
        case '1':
            k=0;
            while (ln[k] != ':')
                ++k;
            k = k + 2;
            while (ln[k] != '\n')
            {
                j=0;
                while (ln[k] != ' ' && ln[k] != '\n')
                    word[j++] = ln[k++];
                word[j] = '\0';
                match = 0;
                for (j=0; j<MAXVERBS; j++)
                {
                    s = strcmp(v_table[j].verb,word);
                    if (s == 0)
                    {
                        match = 1;
                        break;
                    }
                }
                if (match)
                {
                    match = 0;
                    for (j=0; j<num_s_n1; j++)
                    {
                        s = strcmp(spec_n1[j].name,word);
                        if (s == 0)
                        {
                            ++spec_n1[j].count;
                            match = 1;
                            break;
                        }
                    }
                }
                if (!match)
                {
                    strcpy(spec_n1[num_s_n1].name,word);
                    ++spec_n1[num_s_n1++].count;
                }
                strcpy(ent_n1[num_mod].oper[num_e_n1[num_mod]],
                    word);
                ++num_e_n1[num_mod];
            }
        }
    }
}

```

## Appendix 5

```

        if (ln[k] != '\n') ++k;
    }
    break;

case '2':
    s = sscanf(ln,"%s %*[:] %s",fname,name);
    if (s == 2)
    {
        if (name[0] != dollar) break;
        match=0;
        for (i=0; i<MAXFIN; i++)
        {
            s = strcmp(fan_in[i].ftype,fname);
            if (s == 0)
            {
                match=1;
                strcpy(ent_in[num_mod].oper[num_e_in[
                    num_mod]],name);
                ++num_e_in[num_mod];
                break;
            }
        }
        if (!match)
        {
            for (i=0; i<MAXFOUT; i++)
            {
                s = strcmp(fan_out[i].ftype,fname);
                if (s == 0)
                {
                    match=1;
                    strcpy(ent_out[num_mod].oper
                        [num_e_out[num_mod]],
                        name);
                    ++num_e_out[num_mod];
                    break;
                }
            }
            if (!match) inv_attrib(ln);
        }
    }

    match = 0;
    for (i=0; i<MAXOPERSPEC; i++)
    {
        s = strcmp(spec_n2[i].name,name);
        if (s == 0)
        {
            match = 1;
            ++spec_n2[i].count;
            break;
        }
    }
    if (!match)
    {
        strcpy(spec_n2[num_s_n2].name,name);
        ++spec_n2[num_s_n2].count;
    }

```

# Appendix 5

```

        }
    }
    else inv_attrib(ln);
    break;
case ':':
    k=0;
    while (ln[k] != ':')
        ++k;
    k = k + 2;
    while (ln[k] != '\n')
    {
        j = 0;
        while (ln[k] != ' ' && ln[k] != '\n')
            word[j++] = ln[k++];
        word[j] = '\0';
        match = 0;
        for (j=0; j<MAXVERBS; j++)
        {
            s = strcmp(v_table[j].verb,word);
            if (s == 0)
            {
                match = 1;
                break;
            }
        }

        if (match)
        {
            match = 0;
            for (j=0; j<num_s_n1; j++)
            {
                s = strcmp(spec_n1[j].name,word);
                if (s == 0)
                {
                    ++spec_n1[j].count;
                    match = 1;
                    break;
                }
            }
            if (!match)
            {
                strcpy(spec_n1[num_s_n1].name,word);
                ++spec_n1[num_s_n1++].count;
            }
            strcpy(ent_n1[num_mod].oper[num_e_n1[num_mod]],
                word);
            ++num_e_n1[num_mod];
        }
        if (ln[k] != '\n') ++k;
    }
    break;
default: break;

```

## Appendix 5

```

    }
    return;
}

/* This function reads the specification until */
/* a line containing only a <newline> character */
/* is found. */

ignore_entity(line)
    char line[];
{
    char *lptr;

    while ((lptr=fgets(line,MAXLINE,stdin)) != NULL)
        if (line[0] == '\n') break;
    return;
}

/* This function is essentially a dummy function */
/* provided for readability of scan_spec(). It */
/* contains only a return statement and thereby */
/* ignores the attribute line of the spec. */

ignore_attrib()
{
    return;
}

/* This function prints the error message when */
/* an invalid entity type has been encountered.*/

inv_entity(line)
    char line[];
{
    char *lptr;

    fprintf(stderr,
        "Invalid entity type detected in SRS.\n");
    fprintf(stderr,
        "Entity line was: %s", line);
    fprintf(stderr,
        "Entire entity ignored.\n\n");
    while ((lptr=fgets(line,MAXLINE,stdin)) != NULL)
        if (line[0] == '\n') break;
    return;
}

```

## Appendix 5

```

/* This function prints the error message when an */
/* invalid attribute type has been encountered. */

inv_attrib(line)
char line[];
{
    fprintf(stderr,
        "Invalid attribute type detected in SRS.\n");
    fprintf(stderr,
        "Attribute line was: %s", line);
    fprintf(stderr,
        "Execution continues.\n\n\n");
    return;
}

/* The ent_p_vol function computes the potential */
/* volume for each function entity. The average */
/* entity potential volume is returned. The */
/* table of potential volumes is printed. */
/* */
/*          *          *          *          */
/*          V  = (2 + n ) log (2 + n )          */
/*          m          2          2          2          */
/*          */
ent_p_vol()
{
    double nln2, lgnln2, vmstar;
    int i;

    for (i=0; i<num_mod; i++)
    {
        nln2 = 2 + num_e_in[i] + num_e_out[i];
        lgnln2 = log(nln2);
        vmstar = nln2 * lgnln2;
        printf("%-40s      %5.1f\n", ent_in[i].ename, vmstar);
        mod_p_vol[i] = vmstar;
    }
    return;
}

/* The comb_p_vol() function computes a potential */
/* volume for the entire specification from the */
/* potential volumes of the entities. */
/* */
/*          *          *          */
/*          V  = MV  + Mlog M          */
/*          M          m          2          */
/*          */
comb_p_vol()

```

## Appendix 5

```

{
    double M, logM, vMstar, vmavrg;
    int i;

    M = num_mod;
    vmavrg = 0;
    for (i=0; i<num_mod; i++)
        vmavrg = vmavrg + mod_p_vol[i];
    vmavrg = vmavrg / M;
    logM = log(M);
    vMstar = M * vmavrg + M * logM;
    printf("\n\n Combined potential volume is %6.1f .\n\n\n\n",
        vMstar);
    return(vMstar);
}

```

```

/* The m_heading() function sets up the table for */
/* printing the values of the metrics calculated */
/* for the entire specification. */
/*

```

```

m_heading()
{
    printf("Halstead's Metrics For Entire Specification\n\n");
    printf("Metric Name \t\t Computed Value \n");
    printf("----- \t\t ----- \n");
    return;
}

```

```

/* The volume() function computes the volume for */
/* the specification based on the length and the */
/* vocabulary of the specification. */
/*
    V = N log n
    2
*/

```

```

volume(N,n)
    double n;
    int N;
{
    double logn, V;

    logn = log(n);
    V = N * logn;
    printf("Volume V                %6.1f\n", V);
    return(V);
}

```

## Appendix 5

```

/* The n_hat() function computes the estimated length */
/* of the specification from the vocabulary. */
/* */
/*      ^ */
/*      N = n log n  + n log n */
/*           1   2 1   2   2 2 */
/* */
/* */

n_hat()
{
    double Nhat, logn1, logn2, n1, n2;

    n1 = num_s_n1;
    n2 = num_s_n2;
    logn1 = log(n1);
    logn2 = log(n2);
    Nhat = n1 * logn1 + n2 * logn2;
    printf("Estimated length N^      %6.1f\n", Nhat);
    return;
}

/* The level() function calculates the programming */
/* level of the specification from the ratio of the */
/* volume to the potential volume. */
/* */
/*      * */
/*      L = V  / V */
/* */
/* */

level(Vstar,V,L)
    double Vstar, V, *L;
{
    *L = Vstar / V;
    printf("Specification level L      %6.3f\n", *L);
    return;
}

/* The effort() function calculates the number of */
/* elementary mental discriminations required to */
/* understand the specification. */
/* */
/*      * */
/*      E = V / L */
/* */
/* */

effort(V,L)
    double V, L;
{
    double E;
    E = V / L;
    printf("Programming effort E      %6.1f\n", E);
}

```

## Appendix 5

```

    return;
}

/* The e_time() function calculates the estimated */
/* time required to understand the specification. */
/*                                          */
/*      ^                                          */
/*      T = n N (n log n + n log n ) log n      */
/*            1 2 1 2 1 2 2 2 2                */
/*      -----                                  */
/*                        2n S                    */
/*                        2                      */
/*                                          */
e_time(N2)
{
    int N2;

    double n1, n2, logn1, logn2, n, T_hat, logn;
    int S;

    S = STROUD;
    n1 = num_s_n1;
    n2 = num_s_n2;
    n = n1 + n2;
    logn = log(n);
    logn1 = log(n1);
    logn2 = log(n2);
    T_hat = (n1 * N2 * (n1 * logn1 + n2 * logn2) * logn) /
            (2 * n2 * S);
    printf("Estimated time T^          %6.1f\n\n\n\n", T_hat);
    return;
}

/* The hk_heading() function prints the heading for */
/* the table of Henry and Kafura's complexity      */
/* measure calculation results.                      */

hk_heading()
{
    printf("\t\tHenry and Kafura's Procedure");
    printf(" Complexity Metric\n\n");
    printf("Entity Name                               ");
    printf(" Fan-in");
    printf(" Fan-out Length=1 Length=PV\n");
    printf("-----");
    printf("--- -----");
    printf(" ----- \n");
    return;
}

```

## Appendix 5

```

/* The calc_hk() function calculates the complexity */
/* measure of Henry and Kafura using a length of 1 */
/* and a length equal to Halstead's potential */
/* volume for each module. */

calc_hk()
{
    double length, fin, fout, inout, one, pvol, iopow, two;
    int i;

    two = 2;
    for (i=0; i<num_mod; i++)
    {
        fin = num_e_in[i];
        fout = num_e_out[i];
        inout = fin * fout;
        iopow = pow(inout,two);
        one = 1.0 * iopow;
        pvol = mod_p_vol[i] * iopow;
        printf("%-40s    %2d    %2d    %5.2f    %7.2f\n",
            ent_in[i].ename, num_e_in[i], num_e_out[i],
            one, pvol);
    }
    return;
}

/* The print_nln2() function generates a table of the */
/* operators and operands for the specification and */
/* their counts. */

print_nln2(N1,N2,N)
    int *N1, *N2, *N;
{
    int i;

    (*N1)=(*N2)=(*N)=0;
    printf("\t\t Table of Operators and Counts\n\n");
    printf("Operator Name\n");
    printf("Count\n\n");
    printf("-----\n");
    printf("-----\n");

    for (i=0; i<num_s_n1; i++)
    {
        printf("%-40s    %3d\n", spec_n1[i].name,
            spec_n1[i].count);
        *N1 = *N1 + spec_n1[i].count;
    }
    printf("\n-----\n");
    printf("-----\n");
    printf("n1 = %3d\n");
}

```

## Appendix 5

```

printf("N1 = %4d\n", num_s_n1, *N1);

printf("\t Table of Operands and Counts\n\n");
printf("Operand Name          ");
printf("      Count\n");
printf("-----");
printf("      ----\n");

for (i=0; i<num_s_n2; i++)
{
    printf("%-40s      %3d\n", spec_n2[i].name,
          spec_n2[i].count);
    *N2 = *N2 + spec_n2[i].count;
}
printf("\n-----");
printf("-----\n");
printf("n2 = %3d          ");
printf("N2 = %4d\n", num_s_n2, *N2);

printf("\n\n\t\t Entity Potential Volume Table\n\n");
printf("Entity Name          ");
printf("      V*\n");
printf("-----");
printf("      m--\n");

*N = *N1 + *N2;
return;

```

THE APPLICATION OF COMPLEXITY METRICS  
TO A SOFTWARE REQUIREMENTS SPECIFICATION

by

ALICE JOSEPHINE LEMIEUX

B. A., University of Maine, 1979

---

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY  
Manhattan, Kansas

1984

#### ABSTRACT

A measure of the complexity of a software system early in the software life cycle may provide an indication of the quality of the software under development. Complexity metrics, which may be applied to the software requirements specification, may provide this early indication of the software quality. The complexity of software affects the software quality characteristics of understandability and maintainability. The characteristics of understandability and maintainability of software may be measured in terms of the flow of information and size. The metrics developed by Maurice Halstead and Sallie Henry and Dennis Kafura provide measures of the complexity of software. The application of the complexity metrics of Halstead and Henry and Kafura to a software requirements specification is provided by the implementation described in this report.