

Definition and Validation of Interface Complexity Metrics

by

Kevin A. Mayo

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

APPROVED:

Dr. Sallie M. Henry, Chairperson

Dr. Dennis G. Kafura

Dr. J. A. N. Lee

December, 1989
Blacksburg, Virginia

Definition and Validation of Interface Complexity Metrics

by

Kevin A. Mayo

Dr. Sallie M. Henry, Chairperson

Computer Science and Applications

(ABSTRACT)

A software system is an aggregate of communicating modules, and there are several different types of communication among these modules (direct, indirect, and global). Therefore, understanding the interfaces among these modules can characterize the system and are a major factor in the system's complexity. These interfaces could possibly also show and predict inadequacies in the reliability and maintenance of a system. Interfaces are defined early in the development life cycle at a detailed or high level design stage. Knowing that these interfaces exist and their structure leads us to measure them for an indication of the designed interface complexity. This designed interface complexity can then be utilized for software quality assurance by allowing users to choose from among several designs. With data provided by an Ada software developer, the interface complexity metrics correlated with established metrics, but also found complex interfaces that established metrics missed.

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

Acknowledgements

I would like to thank my advisor, Dr. Sallie Henry, for her support throughout this research effort. Her guidance and help in writing this thesis has been without measure (metric?). I would also like to thank Dr. J.A.N. Lee for reviewing my manuscript, and Dr. Dennis Kafura for his insightful criticism within the SPC working group.

was instrumental in the creation of this work. His help and aid can not be overstated. , I think it is safe to say, shared in my pain, as I shared in his. and also lent their help as time came down to the wire, barbed wire that is.

Of course, the constant emotional, spiritual, and financial aid of my parents, _____, kept me at my station and duty. With the conclusion of this I relieve my mother of "yet one more thing to worry about."

And to my friends, pets, and others who listened to me grumble and curse, thank you.

Table of Contents

Abstract.....	ii
Acknowledgements.....	iii
Table of Contents.....	iv
Table of Figures.....	vii
Chapter 1 Introduction.....	1
Introduction.....	1
Metrics.....	2
Code Metrics.....	3
Code Length.....	3
Halstead's Software Science.....	5
Volume.....	5
Program Level.....	6
Difficulty.....	6
Effort.....	7
Cyclomatic Complexity (McCabe).....	7
Myer's Modification.....	8
Hansen's Modification.....	11
Relationships Among Code Metrics.....	11
Structure Metrics.....	12
Module Invocation Complexity (McClure).....	12
Chapin's Module Complexity.....	19
Knots (Woodward).....	20
Information Flow (Henry and Kafura).....	21
Hybrid Metrics.....	22
Information Flow (Henry and Kafura).....	23
Interconnection Model (Woodfield).....	23
Communication Complexities.....	26
Coupling.....	26
Cohesion.....	28
Interface Complexity Metric.....	29
Conclusions.....	30
Chapter 2 Tools Used.....	32
Introduction.....	32
Relation Language.....	32
Metrics Analyzer.....	38
Phase 1.....	39
Phase 1 Modifications.....	42
Phase 2.....	43
Phase 2 Modifications.....	43

Phase 3	47
Phase 3 Modifications	47
Overview	47
Chapter 3 Interface Metric Generation	51
Introduction	51
Type Complexity	52
Base Types	53
Special Cases	54
Characters and Strings	54
Boolean Values	54
Enumerated Types	55
Access Types	55
Aggregate Types	56
Arrays	56
Records	57
Operator Complexity	64
Expression Complexity	67
Flow-Control Complexity	68
Environment Complexity	71
Variable Complexity	73
Ref_Read Complexity	73
Ref_Write Complexity	74
Interface Complexity	76
Output from Analyzer	77
Conclusion	82
Chapter 4 Data, Results, and Analysis	83
Introduction	83
Data	83
Small Experiment	85
Current Weights	88
Possible Uses	88
Interface Complexities	89
Method of Analysis	93
Analysis	94
Known Metrics Correlation	94
Known Metrics VS Interface Complexity Metrics	97
Functions of the Interface Complexity Metrics	101
Subjective Examination	108
Strategy for Examination	108
Examination Results	112
Measures with Current Procedure	112
Extreme AccN Measures	112
Extreme Acc Measures	114
Extreme P1 Measures	115
Extreme P2 Measures	115
Measures from Calling Procedures	116
Extreme FAccN Measures	117
Extreme FAcc Measures	118

Extreme FP1 Measures	119
Extreme FP2 Measures	120
Reverse Examination.....	120
Conclusions.....	123
Chapter 5 Conclusions	125
Research Conclusions	125
Weaknesses	127
Future Work	127
Bibliography	128
Appendix A	
Top 1% Interface Complexity Measures.....	134
Appendix B	
Metrics for Top 1% Interface Complexity Procedures.....	142
VITA.....	149

Table of Figures

Figure 1 Comparison for Code Length (LOC).....	4
Figure 2 Example of a Flow Graph	9
Figure 3 Example of Myer's Modification.....	10
Figure 4 Declaration Statements	35
Figure 5 Language Comparison	36
Figure 6 Conversion of Selection Routines	37
Figure 7 Phase 1 of Analyzer	40
Figure 8 Modified Phase 1 of Analyzer.....	41
Figure 9 Unmodified Phase 2 of Analyzer	45
Figure 10 Modified Phase 2 of Analyzer	46
Figure 11 Modified Analyzer Overview	49
Figure 12 System Usage Chart	50
Figure 13 Calculation of Aggregate Types	59
Figure 14 Contents of Pre-Defined Types File.....	60
Figure 15 (a,b) Example Ada Code	61
Figure 16 Relation Language Output for Example Ada Code.....	63
Figure 17 Operator Correspondence and Weights.....	66
Figure 18 Selection/Repetition Correspondence and Weights.....	70
Figure 19 Example Relation Language.....	80
Figure 20 COMPLEX.OUT Example.....	81
Figure 21 Access Complexity Correlations.....	87
Figure 22 View of Interface Complexity.....	92
Figure 23 List of Abbreviations.....	95
Figure 24 Correlations Among Established Metrics.....	96
Figure 25 Correlations Among Established Metrics and Interface Complexity Metrics	99
Figure 26 Correlations Among Interface Complexity Metrics.....	100
Figure 27 Function F1+ vs Established Metrics	103
Figure 28 Functions F1*& F2* vs Established Metrics.....	105
Figure 29 Functions F2+& F2* vs Known Metrics.....	107
Figure 30 Extreme Values of Interface Complexity Metrics.....	110
Figure 31 Frequency Count Percentile Breakdown of Extreme IC Procedures.....	111

Figure 32 Correlations Among Extreme Established Metric Procedures and
Interface Complexity Metrics 122

Chapter 1: Introduction

Introduction

The computer science field is growing as fast as new applications for computers can be defined, and it is generating software for applications and systems that touch every part of our lives. It is therefore highly necessary to have standards that show how to verify and validate the results of these software projects. Software engineering specialists try to fill this gap, and they attempt to construct standards and goals for the generation and maintenance of computer projects. With these standards, the software life cycle follows along a path paralleling the minimization of costs for development. Thus, the need for these guidelines is apparent.

There are many different factors that make up a good development process: design methodologies (functional decomposition, bottom-up design, and stepwise refinement, etc), the use of design tools (graphical [HENS89] or a design language [HENS90]), design reviews, and the use of software quality metrics. If these factors are maintained throughout the life cycle then some degree of software quality may be attained.

This thesis is focused on the development of a particular software quality metric which measures the communication (or interface) connectivity between two modules. It should be stressed that the definition of interface within this research is based on communication between two modules.

Metrics

The basis of a software metric is that source code has some definite attributes that can be measured. These measures can then be exploited in such a way as to classify software properties such as maintainability, reliability or complexity. There are many different measures that have been addressed. Many of these metrics are subjective measures, and thus they are considered non-automatable. This research is only interested in automatable measures.

The sub-field of software engineering that deals with automatable metrics is divided into: code metrics, structure metrics, and hybrid metrics. Code metrics are measures that are derived directly from the source code of a given software system. Structure metrics are concerned with the complexity of a software system as seen through the interconnections of the source code elements. The third category, hybrid metrics, is composed of both code and structure metrics. The interface metric developed in this study is a hybrid metric.

This chapter provides a short overview of selected metrics that are automatable. The metrics discussed are automated within a tool at Virginia Tech, the 'analyzer' (which is discussed in chapter 2).

Code Metrics

Code metrics are directly 'countable' from the software. In the past many different schemes have been created; however, only a few are discussed within this thesis. The code metrics discussed are: Code Length, Halstead's Software Science, McCabe's Cyclomatic Complexity, and Myer's and Hensen's extensions to Cyclomatic Complexity. All of these, except for Myer's and Hensen's extensions, are generated within the 'analyzer' tool.

Code Length

The most basic, yet non-trivial, metric is that of code length (LOC). This can be interpreted as the number of lines of code for a given module or unit of code under inspection [CONS86]. While this may seem trivial, it is the most intuitive metric. It implies, that the more lines there are to understand, the more complex the program. Yet, this metric does imply that Program A and Program B in Figure 1 are equivalent. However, the complexity of the equations should dictate otherwise.

This causes ambiguity when comments and blank lines are encountered. Should these lines be counted? If comments are counted then that adds to the complexity of the code. The inclusion of declaration statements is also debatable.

Program A	Program B
<pre>x := y; y := x * y / 3.14; z := x / y;</pre>	<pre>a := 2; b := 4; c := a + b;</pre>
LOC = 3	LOC = 3

Figure 1: Comparison for Code Length (LOC)

Halstead's Software Science

Halstead introduced a series of measures based on the countable aspects of source code. Software Science [HALM77], measures are based on:

n_1 = The number of unique operators

n_2 = The number of unique operands

N_1 = The total number of operators

N_2 = The total number of operands

$n = n_1 + n_2$ = Size of Vocabulary

$N = N_1 + N_2$ = Length of the Program

From the manipulation of these four countable measures, Halstead created software science indicators (among others) for: Volume (V), Program Level (L), Difficulty(D), and Effort (E).

Volume

The volume of a program is defined, by Halstead, to be a function of the length (N) and the vocabulary size. The volume is representative of the storage requirements of the program, and is defined as:

$$V = N * \text{Log}_2(n)$$

The $\log_2(n)$ component of the equation represents the storage requirements of the vocabulary symbols.

Program Level

The volume measure can change depending upon the size of the algorithm chosen for the task. Assuming that there are n different algorithms (and thus n different volumes) it can be shown that there is a minimum volume over these code sections. This minimum volume, or potential volume of the most concise algorithm (V^*), is defined but can never be calculated due to its theoretical nature. With this, Halstead defines the program level to be:

$$L = \frac{V^*}{V}$$

Due to the fact that V^* can never be calculated an estimator must be attained:

$$L' = \frac{(2 * n_2)}{(n_1 * N_1)}$$

Difficulty

The difficulty of an algorithm is defined in terms of elementary mental discriminations (EMD). EMD represents the number of comparisons required to implement the algorithm with respect to the given language. Therefore, Halstead defined it as the reciprocal of the program level estimator:

$$D = \frac{(n_1 * N_1)}{(2 * n_2)} = L^{-1}$$

Effort

Halstead's effort measure is a function of the difficulty and the volume. It is representative of the mental effort required to design the algorithm from conception to implementation. Effort (E) is defined as:

$$E = V * D = V * L^{-1} = \frac{V^2}{V^*}$$

Effort, Difficulty, Program Level, and Volume are the measures that compose the Halstead Software Science indicators.

Cyclomatic Complexity (McCabe)

McCabe's cyclomatic complexity metric is built upon the selection and repetition structure of the code. In this theory [MCCT76], the code is represented by a strongly connected graph G. Within this graph the nodes represent the code and the arcs represent the control flow. If the code is a selection statement, then several arcs may extend from that node. This graph maintains the path or flow information of the algorithm and has a unique entry and exit point. The graph is strongly connected since there is an arc from the exit to the entry. With this framework, McCabe defined:

$$V(G) = E - N + 2 = \text{Cyclomatic Complexity, where:}$$

E = Number of Edges in G

N = Number of Nodes in G

$V(G)$ represents the number of control paths in the algorithm. McCabe suggested an upper limit of $V(G)=10$, stating that this limit would also limit program complexity [MCCT76]. It should be noted that in this calculation, $V(G)$, is the cyclomatic number found within classical graph theory, and it is the maximum number of linearly independent circuits or paths within G .

The calculation of this metric may seem, at first, to be complex. However, the generation of this metric reverts to the enumeration of the decisions within the code plus one [WOOM79]. These decisions include the selection as well as the repetition constructs. Also, the total complexity of the set of modules (the program) is equal to the sum of all module $V(G)$'s.

Other metrics have been defined in terms of McCabe's cyclomatic complexity; of these, Myer's and Hansen's work are summarized.

Myer's Modification

Myers suggested an alteration to McCabe's complexity measure [MYEG77]. Myers proposed that the complexity measure be a spanning region from the cyclomatic complexity to the number of individual conditions (plus one). The reasoning behind this is that McCabe's complexity would rate a selection (or repetition) construct with one condition equivalent to a construct with multiple conditions. Figure 2 shows two pieces of code, both of which create the same graph.

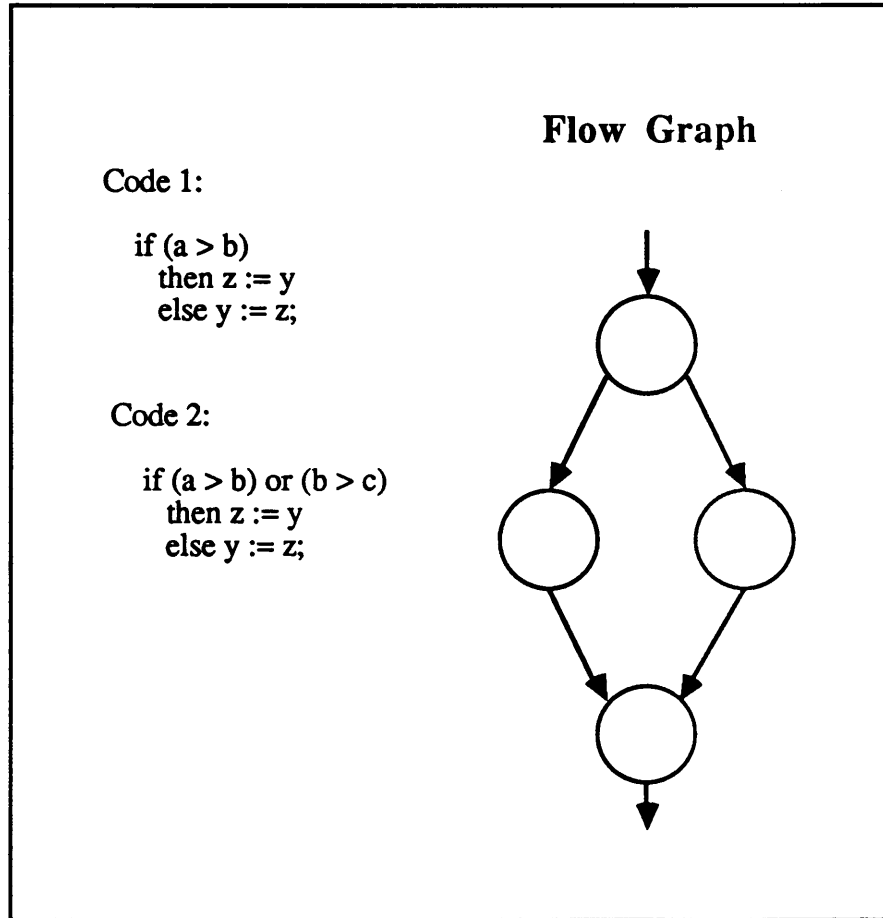


Figure 2: Example of a Flow Graph

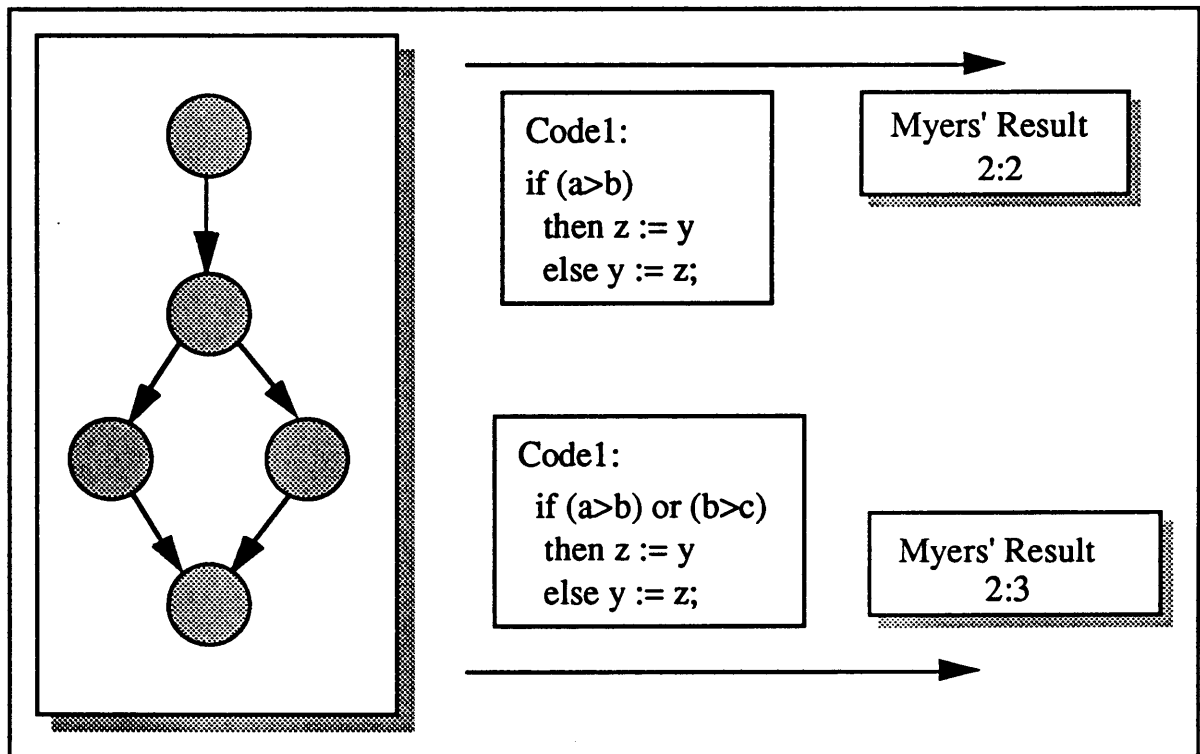


Figure 2: Example of Myer's Modification

Therefore, Myers counts the number of simple conditions and creates the spanning region from cyclomatic complexity to the number of individual conditions. The notation is (cyclomatic complexity : individual conditions). Figure 3 shows Myer's calculations on the code from figure 2. The regions associated with each code segments are 2:2 and 2:3 respectively. This is a better estimator for program complexity in that the complexity of the selection's conditions are not lost as it is in McCabe's work.

Hansen's Modification

Hansen also suggested a metric that included aspects of McCabe's cyclomatic complexity [HANW78]. In this scheme, Hansen combined the cyclomatic complexity and a measure of the expression complexity into a two-tuple. The expression complexity is much like Halstead's N_1 calculation; this complexity is the sum of various operators within the program. This metric measures both flow control and program complexity.

Hansen's complexity is a 2-tuple:

(a,b), where:

a = count of selection/repetition constructs

b = operator count

The operator count is the count of: primitive operators (i.e.: +, -, *, and, or, etc.); assignments; function calls; input/output statements; using subscripts within an array.

Relationships Among Code Metrics

Several code metrics have been validated and correlated through research. This correlation provides the software engineer with a viable basis for using these metrics in their

work. Often these correlations are to show how certain metrics relate to one another under different problems. Bill Curtis, while at General Electric in their 'Software Management Research' group, did work on comparisons between LOC, Halstead's, and McCabe's work. Albert Baker's work included comparisons between McCabe's cyclomatic complexity and 'Knots' (which are introduced later). A small reading list might include: [BAKA80], [BELL80], [CURB79a], [CURB79b], [HENS81a], and [WOOS79].

Structure Metrics

Structure metrics, as defined above, measure complexity through the interconnections found in the source code. In these cases, quantification is applied to these interconnections, and the measures incorporate aspects of program structure. Code metrics are applied over separate modules, sometimes ignoring the program structure, but always ignoring the interconnection complexities. Thus structure metrics must be used and examined to retain the information lost in code metrics.

Module Invocation Complexity (McClure)

McClure introduced a formalism for the calculation of the complexity associated between two communicating modules. McClure defines program complexity as [MCCC78]:

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

Through rigorous definition McClure introduces the Module Complexity Function $M(p)$, which represents the "... complexity of invoking module p in a well structured program." This theory is studied in detail because of its use of a calling hierarchy in its complexity definition.

As noted within the complexity definition, McClure is interested in the structure of the program. To this end, McClure introduces the concept of the PCHS (Program Control Hierarchical System) to be the invocation relationships or calls between all of the modules. It is defined as [MCCC78]:

$PCHS = (P, \gamma, f)$, where:

$P = \{ p_1 \dots p_n \}$	• Set of modules
$\gamma = \text{module}$	• Root Module
$f : P \Rightarrow X \mid P \supset X$	• Invoking Function

$X = f(p_i \mid 0 \leq i \leq n)$, where:

$X = \{x_1, x_2 \dots x_k\} \quad 0 \leq k \leq n$

The above definition of PCHS includes information about the modules and their relationship to each other. The definition of the invoking function implies that f maps a module, p_i , onto a set of modules, X . In other words, this function f relates that p_i passes logical control to each module within X . This defines an invocation relation.

From the PCHS the invoking relation from module to module defines a 'direct ancestor' and a 'direct descendant'. There is also a relationship called 'descendant' ('ascendant') which implies one or more intermediate direct descendants (direct ascendants) between pairs of modules. With these definitions McClure defines these additional sets [MCCC78]:

F_{p_i} = Set of direct ancestors of p_i

G_{p_i} = Set of direct descendants of p_i

H_{p_i} = Set of ancestors of p_i

L_{p_i} = Set of descendants of p_i

From these sets the notion of a common module can be defined:

if $|F_{p_i}| > 1$ then p_i is a common module.

This is obvious that if there is more than one direct ancestor to p_i then two or more modules are invoking p_i , and thus, p_i is a common module. For example, if $F_{p_i} = \{ p_j \}$ then p_i is not common; however, if $F_{p_i} = \{ p_j, p_k \}$ then logical control is passed from both p_j and p_k to p_i thus making p_i a common module.

With the above definition McClure defines the structure of the code; however, the control variables are also of importance within the definition of McClure's complexity. A relationship between the control variables and the modules is defined as [MCCC78]:

v = Program Control Variable

$A_v = R_v \cup M_v$ = Set of modules where v is accessed,

where:

R_v = Set of Modules where v is referenced

M_v = Set of Modules where v is modified

E_v = Set of Modules whose invocation relies on v

α_v = Owner module of variable v

With these definitions variable locality and scope can be witnessed through set membership.

Along with defining a control variable's scope, a degree of ownership is associated with each module over the variable. McClure introduces four different types of ownership:

Degree 1	α_v only modifies control variable v
Degree 2	control variable v is modified in α_v and at least 1 descendant
Degree 3	control variable v is referenced in α_v and is modified in at least 1 descendant
Degree 4	control variable v is not accessed in α_v but is modified in at least 1 descendant

With hierarchical degrees of ownership it is possible to trace the propagation of errors in control variables throughout the modules effected by the variable v .

It is now possible using the aforementioned definitions to introduce the Control Variable Complexity Function. McClure defines the complexity of control variable v as:

$$C(v) = D(V) \times I(V) \times (1/n), \text{ where:}$$

$D(V)$ represents the referencing (Access /Modification) degree of v

$$I(V) = q_v + u_v + w_v + t_v, \text{ where:}$$

q_v = Invocation Complexity

$$= |M_v \cap E_v|$$

u_v = Descendent Complexity

$$= |U_v| \text{ for } U_v = \{ p_j \mid p_j \in R_v \text{ and } \exists p_k \in L_{p_j} \ni p_k \in M_v \}$$

w_v = Path Complexity

$$= |W_v| \text{ for } W_v = \{ p_i \mid p_i \in A_v \text{ and } \exists p_j \in M_v \ni p_j \notin H_{p_i} \text{ and } p_j \text{ is above } p_i \text{ in PCHS} \}$$

t_v = Intermittent Complexity

$$= |T_v| \text{ for } T_v = \{ p_j \mid p_j \in A_v \text{ and } \exists p_k \in F_{p_j} \ni p_k \notin A_v \}$$

In this calculation q_v , the invocation complexity, is simply the number of modules where v is modified and where the invocation relies on v . The descendant complexity, u_v , consists of the number of modules where v is modified within the sub-hierarchy of u . The next term, w_v or the path complexity, is a count of the number of modules that are effected (in the sub-hierarchy) by modules that modify the control variable v . The final term, t_v , calculates the intermittent complexity which is the number of modules that have direct ancestors that do not reference the control variable v .

The value calculated by $C(v)$ is between 0 and 8, where the higher value implies the greater complexity.

Using the Variable Control Complexity it is now possible to calculate the Module Complexity Function. This function, as noted before, is "... used to determine the complexity of invoking module p ..." [MCCC78]. McClure stipulates there are three areas that add to complexity within a module:

- Control structures used in module invocation
- Control variables referenced in module invocation
- Commonality of a module

While to a casual observer these seem obvious, it is McClure's method of measurement that is of interest. The Module Complexity Function is defined as:

$$M(p) = [f_p * X(p)] + [g_p * Y(p)], \text{ where:}$$

$$f_p = |F_p|$$

$X(p)$ = Complexity of Control Structures and Variables used to call the module p

$$g_p = |G_p|$$

$Y(p)$ = Complexity of Control Structures and Variables used within p to call direct descendants of module p

Within this function, $M(p)$, the complexities of both the descendants and ascendants are taken into account. This function appears simple, however, the calculation of $X(p)$ and $Y(p)$ is somewhat involved. They are defined as:

$$X(p) = \left[\left[\sum_{j=1}^x b_j * \left[\sum_{i=1}^e C(v_{ji}) \right] \right] / x \right]$$

$$Y(p) = \left[\left[\sum_{j=1}^y b_j * \left[\sum_{i=1}^k C(v_{ji}) \right] \right] / y \right]$$

and,

- $b_j =$
- 1 if j^{th} invocation is within selection structure or set of selection structures
 - 2 if j^{th} invocation is within repetition structure

x = number of invocation control variable sets used in the invocation
of module p , if $x = 0$ then $X(p) = 0$
 y = number of invocation control variable sets referenced by module
 p to invoke its direct descendants, if $y = 0$ then $Y(p) = 0$
 e = number of control variables in j^{th} invocation
 k = number of control variables in j^{th} invocation
 v_{ji} = i^{th} control variable in j^{th} invocation
 s = total number of control variables
 n = number of unique modules in the PCHS

As can be seen, this calculation gets complicated. It should also be noted that $M(p)$ is bounded in $[0 \dots (16*s*n)]$.

McClure also defines the complexity of a partitioning scheme used for a given piece of code. This result is of not much interest to this thesis and is not covered.

Chapin's Module Complexity

Chapin argued that within a module, data can take on different roles. These different roles yield different complexities to the over all picture of the module. Chapin defined the data roles to be [CHAN79]:

Role P	Data used to produce the output
Role M	Data created or modified
Role C	Data that controls flow through module
Role T	Data not accessed (transitory)

It can be seen that role C data should contribute greatly to the complexity in relation to role T data. This implies an ordering among the different data roles. The counts of the various data used work into the equation for data complexity:

$$W' = 3*C + 2*M + 1*P + \frac{T}{2}$$

The weighting factors, defined by Chapin, are the coefficients of the role types. Chapin then calculates a repetition factor, R, for each module. This repetition factor only looks for loops that invoke other segments (groups of modules). Chapin's Q is then defined for modules within the segments as:

$$Q = R * \sqrt{W'} = R * \sqrt{3*C + 2*M + 1*P + \frac{T}{2}}$$

Finally, the complexity of the entire model is the arithmetic mean of the segment complexities. This model of complexity fails to be comprehensive because it only watches for data communication, whereas other factors (flow complexity) are ignored.

Knots (Woodward)

Woodward [WOOM79] introduced the concept of a "knot". This measure of complexity is based on the number of flow transitions through a 'jump'. A jump could easily be defined in terms of an example: a GOTO statement. A change in control flow through a jump is represented by an ordered pair: (a,b). This implies transfer of control from line a to line b. Originally this study was in terms of FORTRAN line numbers; however, it can be extended to other languages. Therefore, a knot is defined as:

A knot occurs when either conditions 1 or 2 hold:

1. $\min(a,b) < \min(p,q) < \max(a,b)$ AND $\max(p,q) > \max(a,b)$
2. $\min(a,b) < \max(p,q) < \max(a,b)$ AND $\min(p,q) < \min(a,b)$

For points (a,b) and (p,q) in source code.

This theory gives higher complexities to procedures where control flow is highly interwoven. However, in the era of structured programming, knots should be few and far between.

Information Flow (Henry and Kafura)

Information flow is the basis for the structure metric defined by Henry and Kafura [HENS81b]. In this case the complexity is defined in terms of the relationship of a procedure to its environment. This relationship represents the flow of information into (fan-in) and out-of (fan-out) a module. The terms are defined as:

Fan-in	The number of local flows into the procedure plus the number of data structures from which the procedure retrieves information.
--------	---

Fan-Out	The number of local flows from the procedure plus the number of data structures which the procedure updates.
---------	--

With the above definitions defining the flow relationship, the complexity of procedure p was defined to be:

$$C_p = (\text{Fan-in}_p * \text{Fan-out}_p)^2,$$

$\text{Fan-in}_p = \text{Fan-in for procedure } p$

$\text{Fan-out}_p = \text{Fan-out for procedure } p$

Where $(\text{Fan-in}_p * \text{Fan-out}_p)$ represents the number of all possible data paths into and out of procedure p , and it is squared because the complexity relationship is not linear. This complexity points out areas (procedures) with very high information correspondence, and thus a location for errors and error propagation.

Other complexity measures associated with this model are those of module complexity and level complexity. The possibility of an error propagating through some global data must be addressed, and the module complexity with respect to a global data item, G , is defined as the sum of C_p for all p in which G is referenced. The level complexity identifies the locations of missing levels of abstraction. This level complexity is derived from the sum of the module complexities over a given level of a hierarchical ordering of the modules.

Hybrid Metrics

As mentioned above, hybrid metrics are composed of both code and structure metrics. This study only introduces two different hybrid models: Information flow, and the Syntactic Interconnection Model.

Information Flow (Henry and Kafura)

The information flow metric was introduced within the scope of the structure metrics; however, this metric scheme can also be modified to be of a hybrid type. In this case the metric is defined as:

$$C_p = C_{ip} \times (\text{Fan-in}_p \times \text{Fan-out}_p)^2, \text{ where:}$$

C_{ip} = Internal complexity of p

Fan-in_p and Fan-out_p are defined as before.

With this form C_{ip} can be any of the code metrics (e.g. Lines of Code). This metric has been used in a study of the UNIX operating system [HENS81b].

Interconnection Model (Woodfield)

As another example of a hybrid metric is Woodfield's interconnection model [WOOS81]. This model defines the complexity associated with programmer time. The assumption is that certain modules must be addressed several times due to their interconnections within the program. The interconnection scheme is defined with respect to module connections of which there are three different types:

- | | |
|---------------------|---|
| $A \Rightarrow_c B$ | Control Connection: Module A invokes module B |
| $A \Rightarrow_d B$ | Data Connection: Module A modifies some variable that is used within module B |

$A \Rightarrow_i B$ Implicit Connection: Assumptions made within module A are used within module B.

If any of these three types of connections hold, then there is a connection from module A to module B. These connections form:

Fan-in_i The number of times that a module needs to be reviewed. This is made up from a combination of the connections for module i.

It should be noted that the concept of Fan-in here differs from the Henry and Kafura model by including the aspect of implicit connections.

With these definitions in mind, Woodfield continued on to build a rating called the Review Factor:

$$RF_i = \sum_{k=1}^{Fan-in_i} RC^{k-1} \qquad RC = \frac{2}{3}$$

Where RC is the review constant that has been used by Halstead [HALM77].

Woodfield then goes on to define the complexity of a module as a function of the module complexity and the review factor. It is defined as:

$$\hat{T}_{iC_i} = C_i + m \ln(C_k, C_i) \sum_{k=2}^{Fan-in_i - 1} RC^{k-1}$$

where:

C_i is the complexity of module i

C_k is the complexity of an optimal module

It should be noted that C_i is a scaled code metric. Woodfield stipulates that T_{LOC} (C_i is lines of code metric) and $T_{V(G)}$ (C_i is McCabe's cyclomatic complexity) do not measure modular complexity due to the fact that these measures are not from a ratio scale.

Woodfield further divides the calculation of T_{iC_i} into two different models. Both of these models calculate T_{iC_i} in the same manner; however, these two models differ in the calculation of $Fan-in_i$. In the first case, the Syntactic Model, $Fan-in_i$ is calculated using only the control and data connections. Whereas in the second model, the Semantic Model, $Fan-in_i$ is from all three different module connection schemes.

These two models are compared with varying results over different (C_i) metrics [WOOS81]. Woodfield supports the Semantic Model with Halstead's Software Science Effort metric as the best estimator of programming effort.

Communication Complexities

In the previous section several different measuring techniques were introduced. These quantified values in programs and modules were often ratio-ed with respect to one another; however, often it is not the ratio of these values that is of interest to software engineers, but it is what they represent to a standard scale that defines levels of 'goodness' or 'badness'. It is very hard to devise an standard scale that can be applicable to each different metric. It is the mapping from these metrics to the standard (outside) scale that introduces debate. This section is concerned with two outside scales.

This thesis is interested in the communication costs between two different modules, defined as the interface complexity. Therefore, this section introduces the concept of Coupling and Cohesion as scales to judge the quality of software.

Coupling

As two modules communicate there is an interface that has associated with it a complexity. This interface in turn, effects the interaction strength between the two different modules. The amount or strength of correspondence is referred to as coupling. Specifically, coupling is defined to refer to [STEW74]:

- How complicated the connection is
- Whether the connection refers to the module itself or something inside of it, and
- What is being sent or received

Therefore, as coupling increases so does the complexity of the interface.

If a design goal is to keep the coupling of a system down then several different approaches must be considered. For example, the use of global data sets increases the coupling between different modules, and thus should be avoided. In other words, every module that accesses the global data is in turn coupled with every other. This has the effect of coupling modules that can have totally different functionality.

Another aspect that contributes to higher coupling is that of interface complexity. Stevens defines [STEW74]:

Interface Complexity: How much information is needed to state or to understand the connection. Thus, obvious relationships result in lower coupling than obscure or inferred ones.

This interface complexity would then include: the number of parameters, whether the connection refers to the module name or something inside of the module, and how the module is invoked (environment).

The final aspect toward coupling is: the effects of what is being passed. This implies that "elements of control" increase the coupling. These elements include: switches, flags, signals, etc.

Cohesion

While coupling measures the strength between two different modules through calls and data, cohesion represents the binding amongst related modules. The modules are related to one another and fall into a category defined by their binding of their elements (data, and the modules themselves). This binding falls into one of six different categories defined as [STEW74]:

- | | | |
|---|-----------------|--|
| 1 | Coincidental | There is no meaningful relationship among the elements in a module. |
| 2 | Logical | There is some logical relationship between the elements of a module. |
| 3 | Temporal | This implies logical cohesion with the added clause that these elements are also related in time. |
| 4 | Communicational | The elements are related by a reference to the same set of input and/or output data. |
| 5 | Sequential | The output from an element is the input to another element of this module. |
| 6 | Functional | By far the strongest, this implies the elements are related to the performance of a single function. |

These are listed in order of strength of binding, from least to greatest. When it is possible for a module to contain more than one type of binding, then the higher binding applies.

Interface Complexity Metric

The purpose of this thesis is to introduce the Interface Complexity Metric and give its definition, implementation, and uses in the area of software quality.

The Interface Complexity Metric has elements contained within it much like that of Stevens interface complexity. A set of metrics is defined that make up the Interface Complexity Metric. These metrics, defined within chapter 3, are: Type Complexity, Operator Complexity, Flow-Control Complexity, Expression Complexity, Environment Complexity, and Variable Complexities (Ref_Read and Ref_Write Complexities). This set of metrics finds its basis in the information flow concept as put forth by Henry and Kafura. These measurements try to capture how much information is passed in a similar manner to that of the Interconnection Model. This metric set also takes into account the structure of the code somewhat like that of McClure's weighting of the selection and repetition constructs.

The motivation behind the interface metric is that previous metrics fall short of maintaining all the information needed to model a non-trivial indicator as to the complexity of code and interfaces. In other words, past metrics only viewed specific problem areas, and thus were applicable to those areas only. For example, while Henry and Kafura [HENS81b] modeled metrics within the scheme of data flow analysis, they ignored the inherent complexities that different data structures introduce into the picture. McClure weights the different structures by either selection or repetition; however, this approach neglects the complexity associated with each construct (selection/repetition) control clause.

Also, various styles of repetition constructs (Test-Before, Test-After, etc.) were put into a single class; whereas, there could possibly be a difference in complexity among these structures.

Within the calculation of the Interface Complexity Metric there are several underlying code metrics that must be calculated. Instead of using the predefined 'classical' metrics, several metrics have been defined to include properties of Halstead Software Science indicators. These new measurements define expression complexities of mathematical and relational relationships so as to further enhance the measures associated with selection and repetition clauses.

Conclusions

This chapter has given an overview of previous work in software quality metrics. Of the three types, code, structure, and hybrid, this thesis feels that the third needs more consideration. In past research many aspects of complexity have been left out or overlooked due to specific needs and abilities of researchers and their computers, respectively. Yet with the analyzer tool, discussed within chapter 2, a whole new hybrid metric can be introduced. This metric, the interface complexity, takes many different aspects into account in order to be robust.

The introduction to the different metrics and several software engineering classification categories in this chapter is designed to give a framework for the work presented here. This study tries to introduce a new metric, the Interface Complexity metric. Chapter 2 gives a brief introduction to the metric gathering tool used for this research. Chapter 3

discusses the implementation of the interface complexity metrics. Chapter 4 presents this studies results, and chapter 5 shows new roads for the future in this topic.

Chapter 2: Tools Used

Introduction

While considering the interface metric, a tool must be used to gather the underlying measures and generate the interface complexities. Currently at Virginia Tech there is a software quality metrics gathering tool, hereafter referred to as the analyzer. This tool measures the three different classes (code, structure and hybrid) of metrics that were introduced within chapter 1, and the metrics defined in chapter 2. This tool is a three phase operation, and to incorporate the interface complexities all three phases were modified.

Underlying this tool is what is known as the 'Relation Language'. The analyzer converts source language into this intermediate code. A discussion of the relation language precedes the description of the metrics analyzer.

Relation Language

The relation language is an intermediate code form which the Henry and Kafura flow relations are generated. This language is designed to be sufficiently generic to repre-

sent several different high-level procedural-style source languages (FORTRAN, Pascal, C and Ada); however, it still contains the information needed to calculate the structure and hybrid metrics. Due to its 'generic' design, the relation language protects the proprietary nature of source code. In this way, proprietary software can be analyzed by Virginia Tech without loss of metric quality and companies do not have to relinquish their software [HENS88].

There are two divisions of statements in the relation language: Declaration and Executable. Declaration statements define the locations to which data may flow 'into' or 'out of'. These include: local variables, data structures, procedures, and functions. It should be noted that in the case of procedures the data flows 'into' and 'out of' the parameters; whereas, functions return values through themselves (through the function name). The declaration statements are outlined in figure 4 with examples. An example of the translations of variables can be seen in figure 5.

The executable statements within the relation language try to capture the elements that are represented within the class of high-level procedural languages. There are only a few: COND, Block Delineators, and an assignment operator '='. This COND statement is used to model the various flow control constructs. These include the IF-THEN-ELSE and CASE like statements, as well as the repetition statements: REPEAT-UNTIL, WHILE-DO, and FOR-LOOPS. In order to retain the scoping and nesting structure of these high-level languages the BEGIN and END statements represent block limits within the relation language.

Figure 6 demonstrates the conversion of various Pascal and Ada flow control structures into the relation language. It should be noted that all the conditionals, both the sets of selection and repetition constructs, are converted into COND statements. Also, because the primary interest is a static analysis of the data flow within the relation language, there is no need or provision for the ELSE clause. This ELSE-omission can be seen in Figure 6.

STATEMENT	EXAMPLE	USAGE
LOCAL	local X;	Defines a local variable X.
STRUCT	struct X;	Defines a non-local variable X.
CONST	const X;	Defines a constant variable X.
PROCEDURE	procedure X (...)	Defines procedure X with any number of parameters.
FUNCTION	function X (...)	Single value return procedure X with any number of parameters.
INTRINSIC	intrinsic ABS, LOG;	A function defined within the language specifications, i.e. ABS, LOG, TAN, SIN, COS.
EXTERNAL	extern X;	Defines a procedure that is declared within another module or sub-system.

Figure 4: Declaration Statements

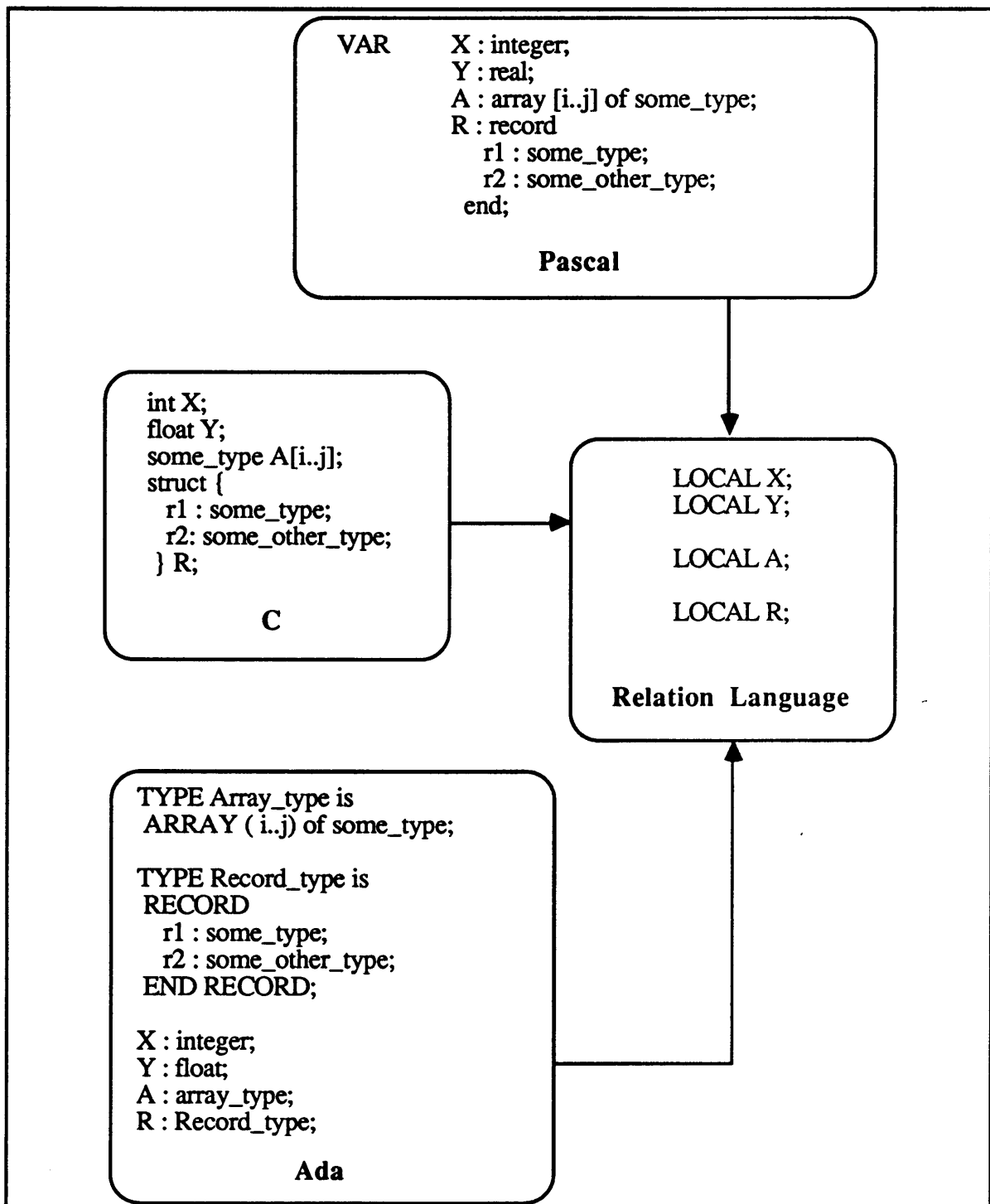


Figure 5: Language Comparison

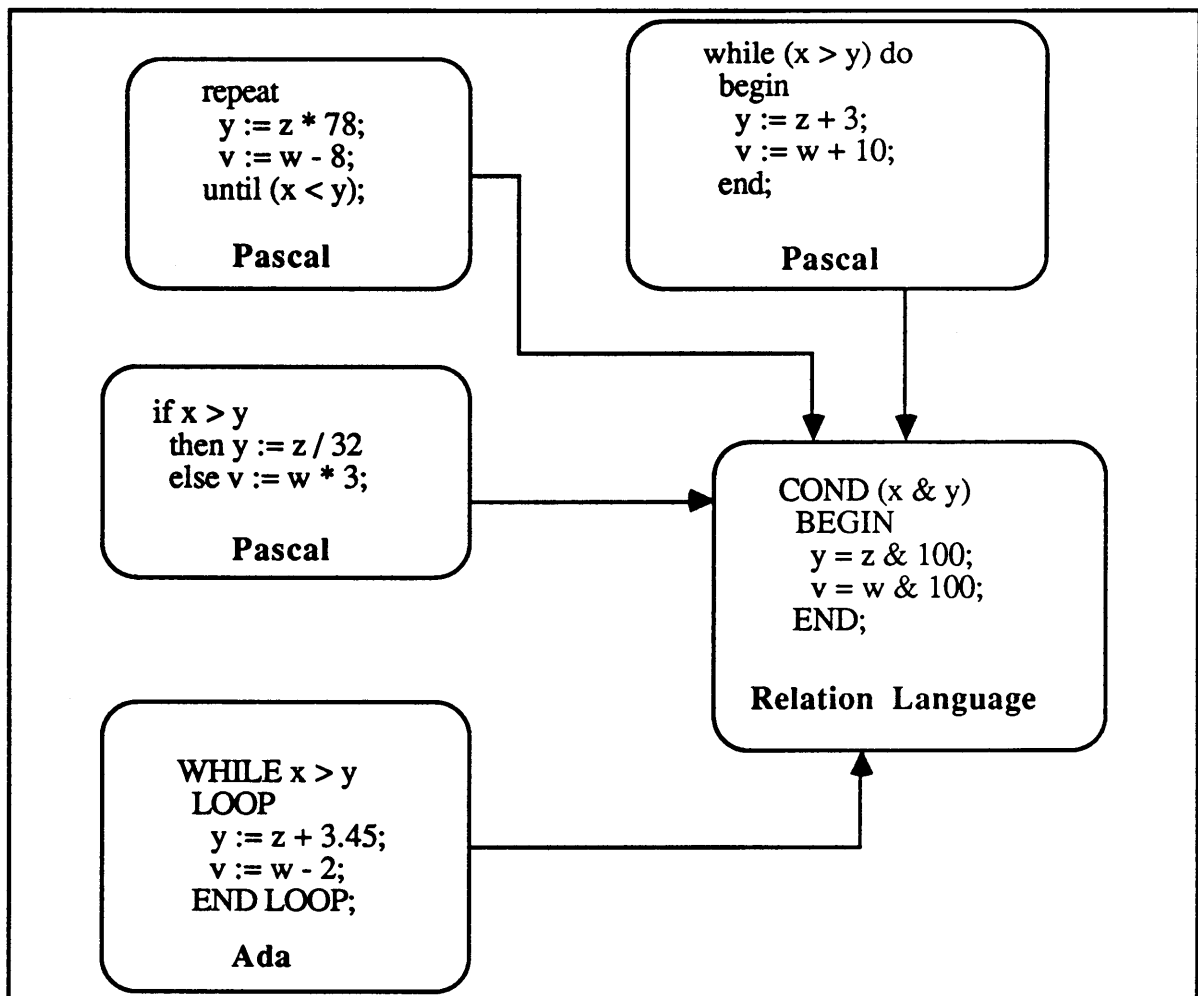


Figure 6: Conversion of Selection Routines

Due to the fact that only data flow is considered within the scope of the relation language, the operations on the data become secondary in nature to the data flow. What matters is where the data is 'flowing to' more so than how it gets there. This leads to the mapping of all operators like '+', '*', '-', etc. into the generic relation language operator '&'. For example, consider:

- $X = Y * Z + 2.45$
- $X = Y + Z - 10$

It should be clear, from a data flow perspective, that in all cases the data flows from Y and Z into X regardless of the operator. Numeric constants are all mapped to the constant 100 since the representation of data flowing from a constant does not require the knowledge of the value of the constant. Therefore, the correct relation language derivative of both from the above example is:

- $X = Y \& Z \& 100$

It can be seen through this example that proprietary data is well-masked.

Metrics Analyzer

The relation language is used as an intermediate form for input to the metrics analyzer. This section defines the three phases of the metrics analyzer, and also the changes that must be made in order to incorporate the interface complexity generation.

The operation of the analyzer, as mentioned above, is a three-phase process. The first phase converts a source language program into a relation language equivalent and generates code metrics. The second phase currently calculates the Henry and Kafura Flow relations from the relation language. The final phase calculates the structure and hybrid metrics and allows a user to compare and contrast all the various metrics generated for the source code.

Phase 1

Phase 1 converts a source language programs into the relation language and calculates code metrics. This tool is built with LEX and YACC within the UNIX environment. LEX generates a lexical analyzer for use with YACC (Yet Another Compiler Compiler). This phase is the only language dependent aspect of the three phase process. A translator is built for each language (ie: Pascal, FORTRAN, C and Ada) by using the BNF (Backus-Naur Form) grammar and a file with the language-specific intrinsic functions. Figure 7 describes the relation language translator.

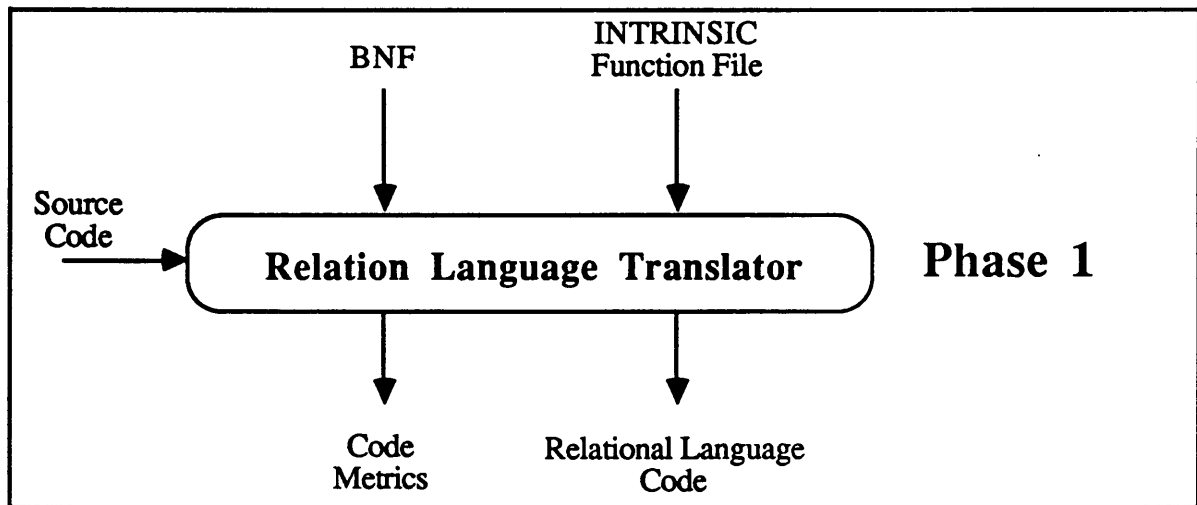


Figure 7: Phase 1 of Analyzer

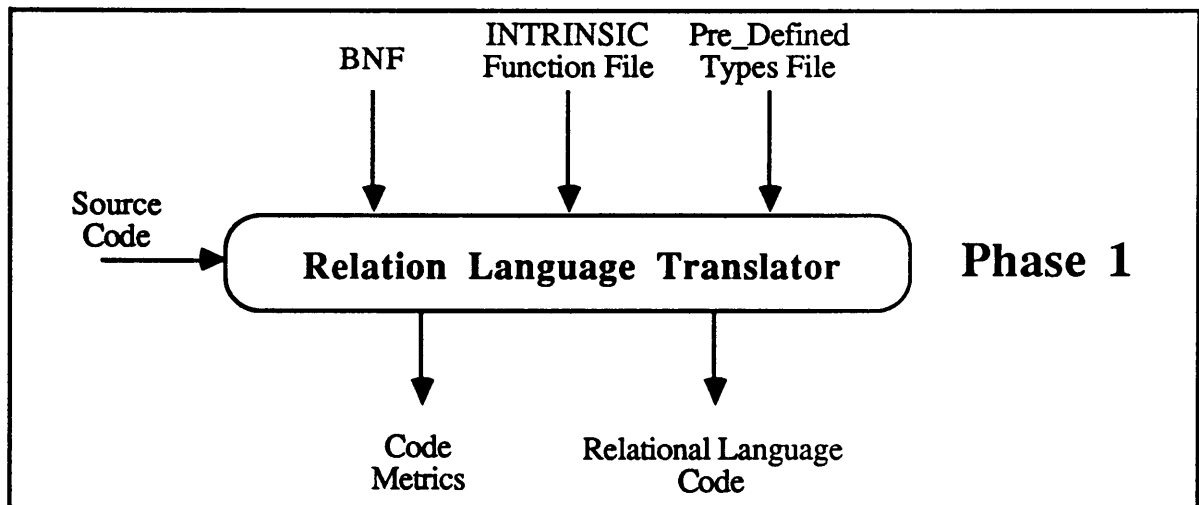


Figure 8: Modified Phase 1 of Analyzer

Phase 1 Modifications

The modifications that need to take place for the incorporation of the interface complexities within the relation language, and thus, within the first phase include maintaining the type, operator and flow-control complexities. This is handled by associating an integer valued complexity with each type, operator and flow-control statements. The operation and flow-control integer valued complexities are actually a code that is converted to the corresponding operation and flow-control complexities within the complexity manager in phase 2. The type complexity is, however, generated within the first phase. This type complexity is located with the declaration of the variable. For example: each variable has a complexity, suppose that variable 'x' has a complexity of 24, then in the relation language this variable would be defined as:

LOCAL (24) X;

Where it can be seen that the general format is:

LOCAL (*type complexity*) *variable* ;

This format is also used for constants and structures.

There must also be modifications to the generic operator (&) and the generic conditional statement (COND). Therefore, each operator has to be carried into the relation language, and this is achieved by concatenating a unique integer to '&' for each unique operator. For example '+' might be defined as '&1', while '-' would be '&2'. This format fol-

lows for the selection and repetition statements represented by the statement COND. For example, IF-THEN-ELSE might be 'COND1', while 'CASE' would be 'COND2'. These appended numbers are the code in which the complexity manager will weight the operator/selection/repetition complexity. In this way the integrity of the information is maintained. If there are no integers attached to '&' or 'COND' then a default value is selected, and this default does not effect the calculations of the interface complexities.

Phase 2

This phase, referred to as the 'Relation Manager', takes as its input the relation language translation from phase 1. The output of this phase is a set of flow relations that are used in the calculations of the Henry Kafura Information Flow metric.

The Relation Manager, much like the source language translators, is built on a LEX-YACC base. The LEX-YACC base calls a set of routines referred to as the relation manager. Each procedure is analyzed at a time and its relations are written to a file for use in phase 3.

Phase 2 Modifications

The interface complexities are generated in phase 2. It is from the relation language that these complexities are derived; therefore, the LEX-YACC base must be modified to call a set of routines referred to as the 'Complexity Manager'. The complexity manager is a module that encapsulates all the information for the generation of the interface complexity metrics. This manager is easily 'attached' to the LEX-YACC scheme, and due to its modular nature it does not interfere with the generation of relations.

The complexity manager is a series of routines, as noted above, that controls the calculations of the interface metrics. Within the LEX-YACC phase 2 scheme, the routines are called when variables, procedures, and structures are seen through the parse. In this way the complexity manager keeps track of the status of the relation language code.

Associated with the complexity manager is the complexity file COMPLEX.DAT. This complexity file contains the default weights for the operators (&1, &2, etc.) and flow control structures (COND1, COND2, etc.) COMPLEX.DAT may be modified to test the relation language code for different characteristics. For example, the software engineer may wish to weight repetition structures greater than (or less than) selection structures to assess the effects of loops within the code. COMPLEX.DAT will be discussed in more depth in chapter 3.

The complexity manager analyzes the relation code procedure by procedure and generates two output files: CODEOUT and COMPLEX.OUT. The first file, CODEOUT, is used as input to phase 3 of the analyzer. This file is coded information about the interface complexities. The second file, COMPLEX.OUT, is a document that defines all the interface complexities as well as the ref_read and ref_write metrics and other important information. This file can be printed out and analyzed by the software engineer. COMPLEX.OUT will be discussed in more depth in chapter 3.

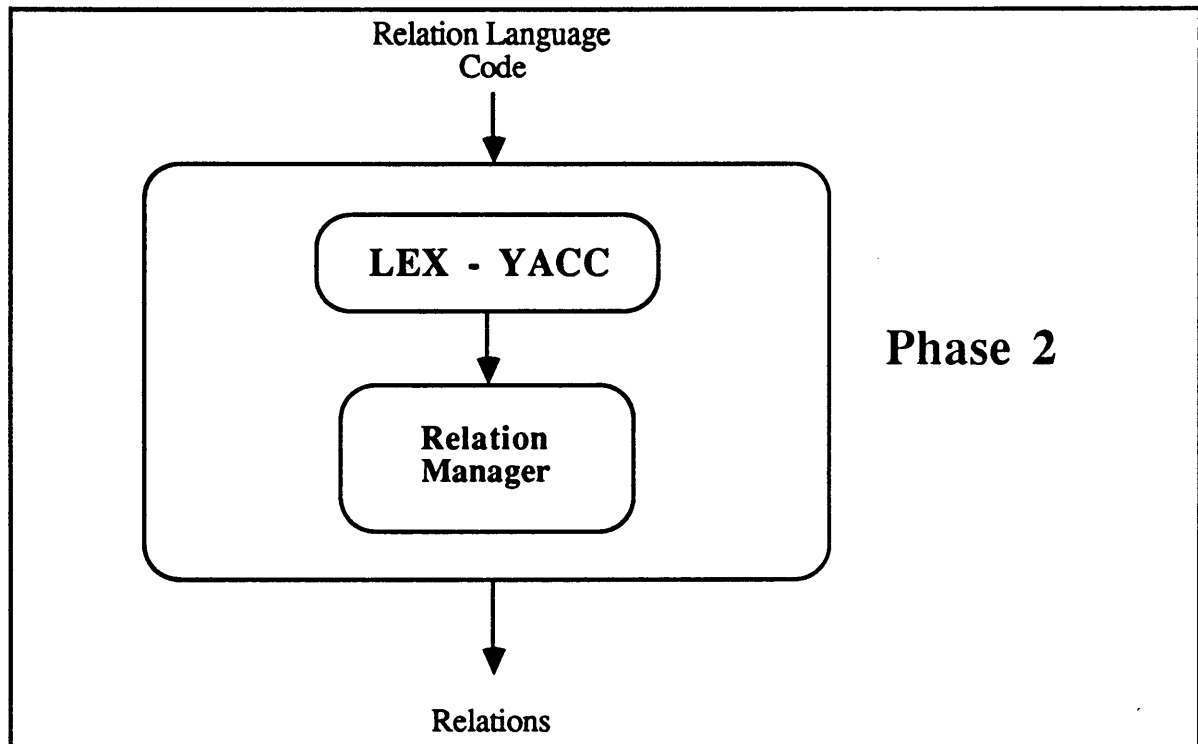


Figure 9: **Unmodified Phase 2 of Analyzer**

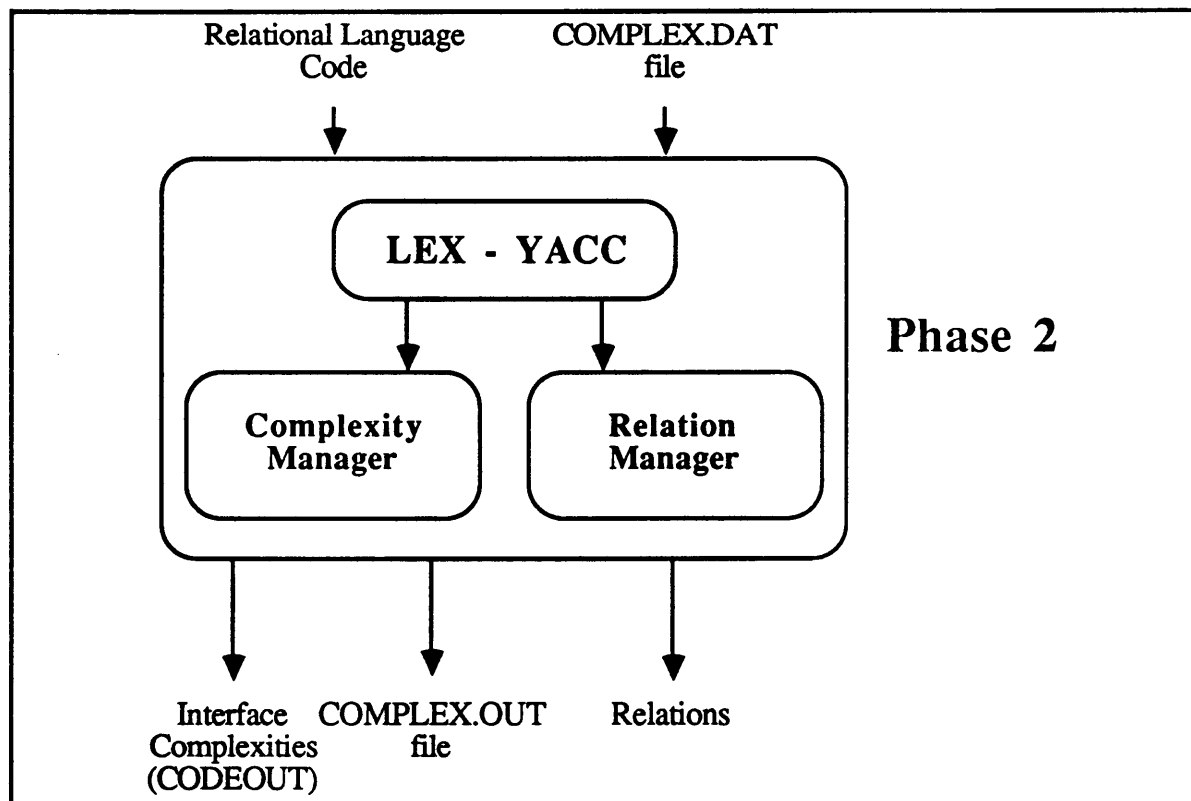


Figure 10: Modified Phase 2 of Analyzer

Phase 3

The final phase uses the relation file generated in Phase 2 for further calculations of structure and hybrid metrics. These metrics include the McClure Invocation metric, Henry and Kafura Information Flow metric, and Woodfield's Review complexity. Written in standard Pascal, this is also a tool for the viewing of the metrics. This analysis can be performed at the procedure level or at user defined modules or levels.

Phase 3 Modifications

Phase 3 displays the various code, structure and hybrid metrics. Due to the fact that the interface complexities are already generated in the modified Phase 2, the only modifications in this phase are involved with the display of the data.

The interface complexities are located in the CODEOUT file that is created in Phase 2. This file must be read in and displayed in a format that the user can understand and use. There are two different viewing schemes implemented: Called Data, and Called-By Data. Every procedure can be viewed as to the procedures it calls (and the calls associated complexity) and by the procedures that call it (and these complexities, also.) This gives the user the ability to globally locate procedures with high 'fan-in' or 'fan-out' complexities.

Overview

Figure 11 is an overview of the analyzer. This figure details how each of the parts works together to produce the tool. This figure represents the modified analyzer. Figure

12 is a graphical description of the file usage by the three phases, and it also defines the files that can be modified by the software engineer to meet his/her needs.

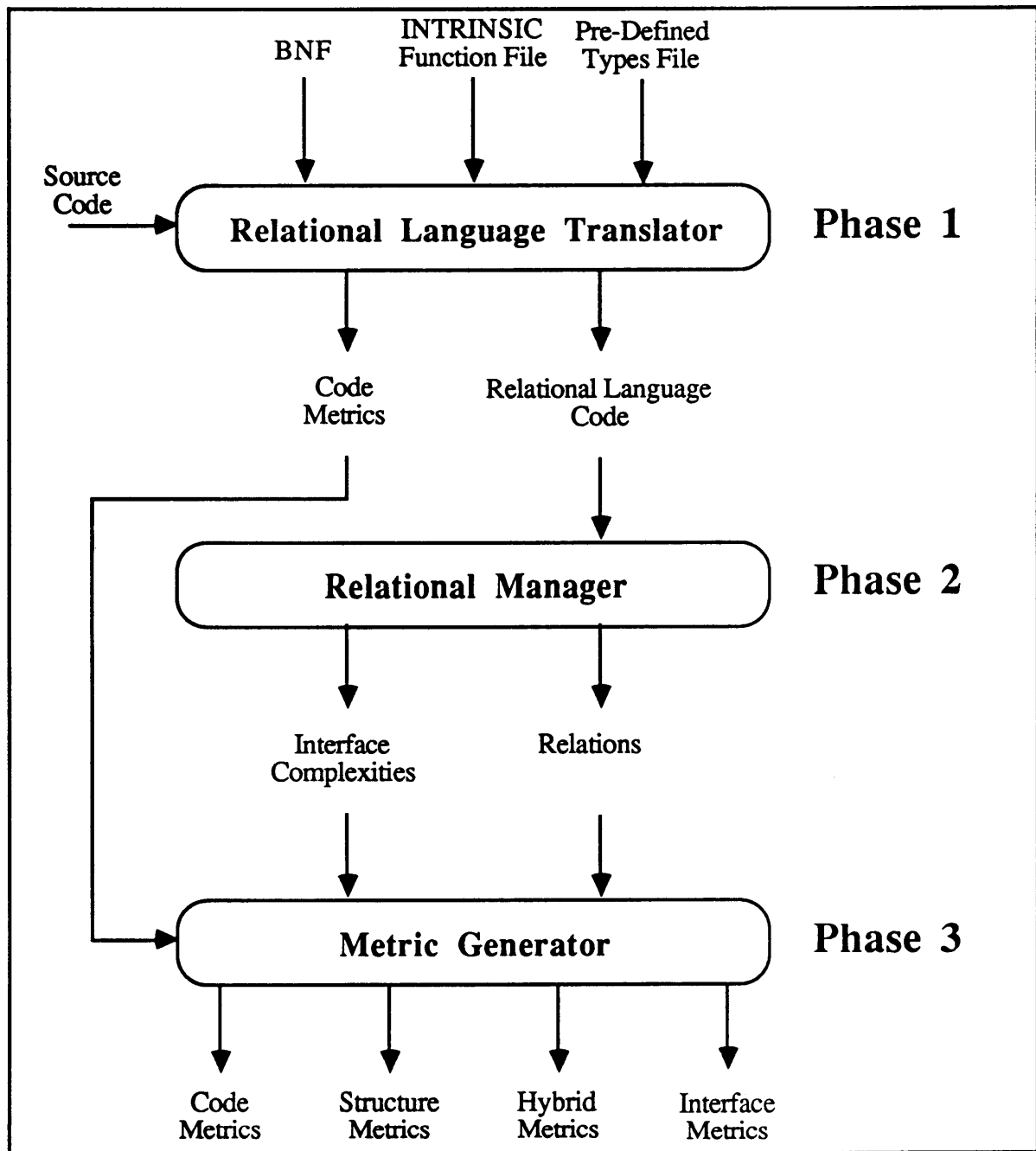


Figure 11: **Modified Analyzer Overview**

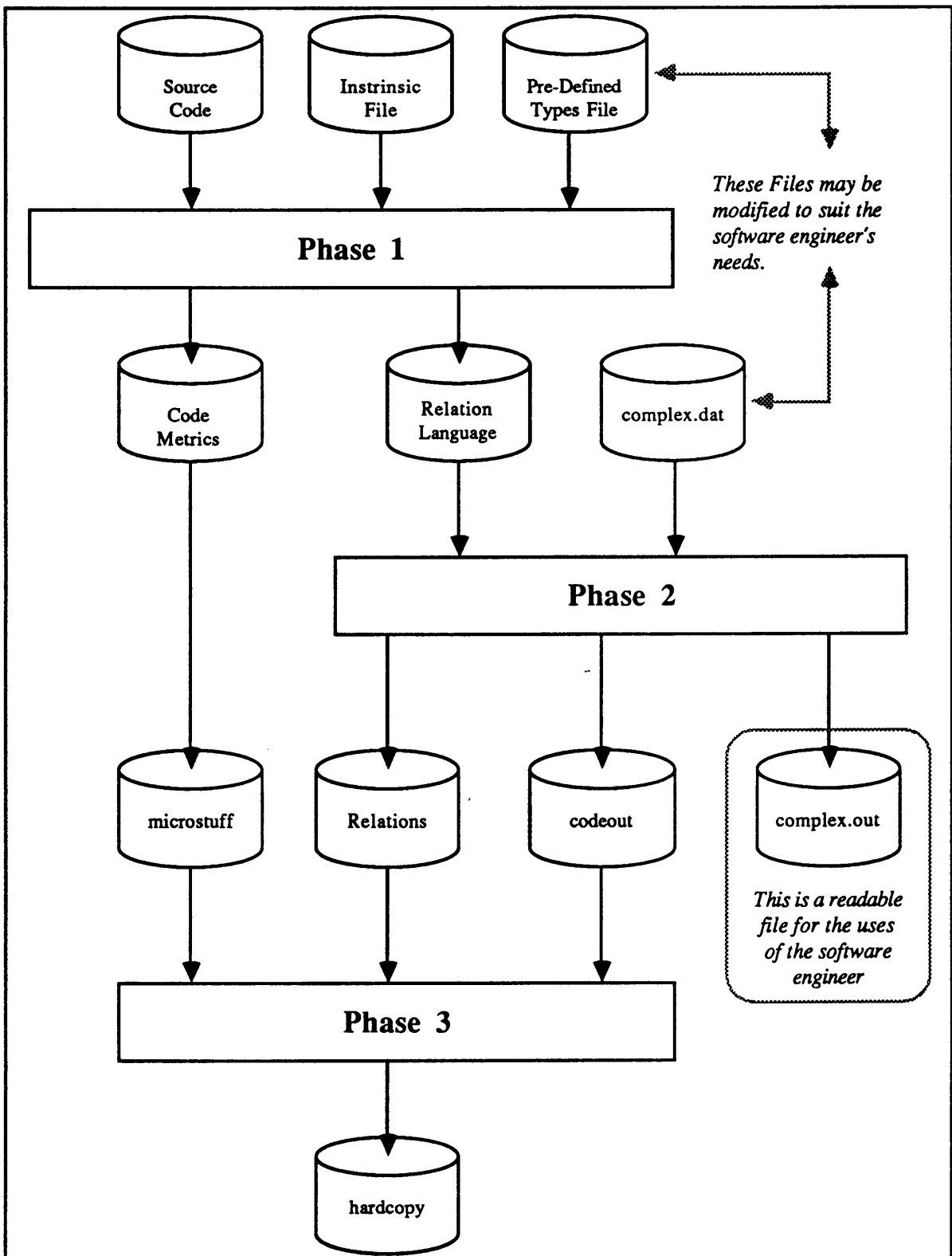


Figure 12: System Usage Chart

Chapter 3: Interface Metric Generation

Introduction

To facilitate the needs of the interface complexity outlined in chapter 1, several different underlying complexities (metrics) must be defined. These metrics try to capture the 'weight' of data being passed through the 'complexity' of the given code at the instant of the communication invocation. This implies that the interface complexity must take into account the types, expression, and flow-control. Incorporating these complexities into meaningful software quality metrics is the goal of this research.

These metrics consider the differences inherent in types, operators, and flow structures. The complexities to be defined are: Type, Operator, Expression, Flow-Control, Environment, Variable and Interface Complexities. These complexities are broken down into two different classes:

- 1 Type, Operator, and Flow-Control
- 2 Expression, Environment, Variable (Ref_Read and Ref_Write), and Interface

The first class of complexities are all computable directly from the source code, while the second class is derived from functions on the first class.

Type Complexity

Type complexity attempts to capture the inherent differences among variable types. This allows the metric to differentiate between the inherently easier INTEGER type from the more complex FLOATING-POINT type variables.

Most high-level procedural programming languages contain two classes of types: Basic Types and Aggregate Types. The basic types define single value variables such as: BOOLEAN, INTEGER, REAL (FLOATING-POINT), FIXED-POINT, CHARACTER, etc. While the aggregate types define aggregates of data. These aggregates, as good programming practice dictates, are specified with regards to a single logical unit (i.e. `player_record`, `system_page_map_record`, etc.).

The type complexities are generated within the first phase of the analyzer. These type complexity integer values are generated following specific formulae; however, the constants within the formulas can be changed by the software engineer. This ability allows the user to weight different types by his own measurements, and thus allow for measuring of different aspects of the code. It is the contention of this study that different types have inherently different complexities, and these measures represent a metric value associated with the complexity of the type.

This chapter introduces the different functions used to calculate the type complexities, and an analysis of which weights can better serve the software engineer. There are two classes of types: Base and Aggregate. Base types include single attribute types such as integers, reals, etc. Aggregate types are collections of other group types and base types, like arrays, records, etc. Both of these classes are addressed and their complexity formulae given.

It should be noted that this study is performed over data from an Ada environment. It should also be noted that the types that are being dealt with here are representative of most procedural language types. Therefore, it can easily be seen how this type study can be applied to other languages with similar type constructs.

Base Types

The base type (single attribute) variables have predefined type complexities. In other words, the complexities of the base types are contained within a file (Pre-Defined Types file) accessible by the software engineer. This file allows the software engineer to apply his own weights to base types and thus create his own specific measurements. The benefit is the ability to single out one type and display, through metric analysis, its effects on the software package.

Within Ada the user is given the ability to apply specific attributes on base types. An example of this is the use of DELTA, DIGITS, and RANGE appended to type or variable declarations. For each base type modifier there is an entry in the pre-defined types file, and these are used in the calculation of the base types when necessary. The complexity of these types is the sum of the base type complexity and all modifier complexities.

Special Cases

Contained within the simple domain of the base types are several special cases. These types are special because they represent non-numerical concepts. These are types widely used but with specific restrictions. These would include: Characters, Boolean, and Enumerated Types. For completeness, their type complexity definitions follow.

Characters and Strings

Characters are single attribute types with specific restrictions. A single character is inherently easily to conceptualize, and thus its complexity should be relatively low. The character type complexity is maintained within the predefined type complexity file.

Strings as a type are defined to be sequences of concatenated characters with an attribute of length. Therefore, this type is more readily viewed as an array of characters, and the complexity reflects this view.

Boolean Values

George Boole (1815-1864), a self-educated British logician and mathematician, is the namesake for this special case of base types. A boolean variable can represent a TRUE or FALSE state. All the operations associated are boolean algebraic specific, such as AND, OR, XOR, etc. This is a simple and distinctly different algebraic scheme than numerical schemes.

The type complexity for boolean values is also contained within the predefined types file. The software engineer can, therefore, adjust the boolean type complexity. For

example, if the software engineer wishes to test the effects of boolean variables on code, he/she can modify all other weights to zero and the resultant interface complexity analysis will reflect only the boolean variables contribution.

Enumerated Types

Enumerated types represent user-defined classes of objects. Users can define operations upon these classes aside from the standard operations. The complexity of the set is as complex as the number of elements within that aggregate. Therefore, the complexity of an enumerated set is proportional to the number of members within that set, while, the complexity of a single element within the group is contained in the predefined type file.

Access Types

Access types within Ada are the related to pointers within Pascal. An access typed variable contains a value, called *access value*, that points to another variable. The complexity of this access type is derived from the complexity of the variable that is located in regards to the *access value*. Therefore, the access type complexity is the *access value* variables complexity plus an ACCESS Adder. The ACCESS Adder is a value from the predefined types file that can be modified by the software engineer.

It should be noted that a distinction is made for self-referencing pointers. A self-referencing pointer is an access type that refers back to itself. In this case, self referencing pointers are defined to be easier (less complex) than non self-referencing pointers. A self-referencing pointer is a predefined type, and its complexity is contained within the predefined type complexity file.

The method of resolution is to generate a type 'tree' of incomplete specified types. This is where much of the self-referencing comes into play. Next the 'tree' is resolved by applying the rule of self-referencing. In other words, if a type name is found that is identical to a name found previously then there is self-referencing, and the predefined type complexity can be applied. Thus, the tree is simplified, and repeated manipulations will reduce the 'tree' to isolated completely specified types.

Aggregate Types

Aggregate types are comprised of other aggregate types and base types. Of the various structures that are present in high order procedural-style languages this study centered on: Arrays and Records, noting, of course, that files are aggregate types that imply different external storage requirements.

Arrays

Arrays are collections of similar data where the components are identified by one or more subscripts. To compute a complexity based on this type, several different concepts must be addressed. These aspects include: number of indices and the complexity of the array elements. The formula that this study implemented is:

$$\text{Array Complexity} = \text{Component Type} * (\text{Indices} + 1) * \text{Multiplier} + \text{Adder}$$

It can be seen the effect that the component complexities and number of indices have on the complexity; however, the multiplier and adder are weights defined by the software engineer

to fine tune the metrics. The definition of these variables is within the pre-defined type file where they are referred to as: Array Multiplier and Array Adder.

Figures 13-16 have several examples of this calculation. In figure 13, *players* is defined to be an array of player records. Therefore, by the formula above the complexity of *players* is the component type complexity of *player_rec* times the number of indices plus one (1 + 1), times the multiplier, plus the adder.

In figure 15, *ASDF*, *QWER*, and *ZXCV* are example arrays. Using the example pre-defined values in figure 14 the weights are calculated to be 3, 10, and 9 respectively.

Records

Records are aggregates of base and other aggregate types, referred to here as components. These components add complexity in a different manner than array elements. Each record can be made up of different types of fields that have different type complexities, so each field must have some bearing on the complexity. However, simply summing the field complexities is inadequate because records logically group related information and thus reduce its complexity. It is not sufficient to divide by the number of fields, because this would only yield the average type complexity of the fields. Therefore, like the array complexity, a multiplier, adder, and component adder have been added to the equation. The formula used in this study is:

$$\text{Record Complexity} = \frac{\left(\sum_{1}^{\# \text{ sub-fields}} \text{Sub-Field Type Complexity} \right) * \text{Multiplier} + \text{Adder}}{(\# \text{ sub-fields} + \text{Component Adder})}$$

As with the array complexity, the multiplier, adder, and component adder are user defined within the predefined types file.

Figures 13-16 demonstrate this equation. In figure 13, *player_rec* is a record, and the calculation of its complexity is outlined. The complexity is the sum of component complexities (integer complexity, 3 for enumerated group complexity, and real/float complexity), times the record multiplier, plus the record adder. This value divided by the number of components (3) plus the record component adder.

Figures 15 and 16 demonstrate the calculation with the pre-defined types values contained within figure 14. Figure 15 defines several records: *date*, *buffer*, and *peripheral*. Their complexities, shown in figure 16, are 2, 2, and 2 respectively.

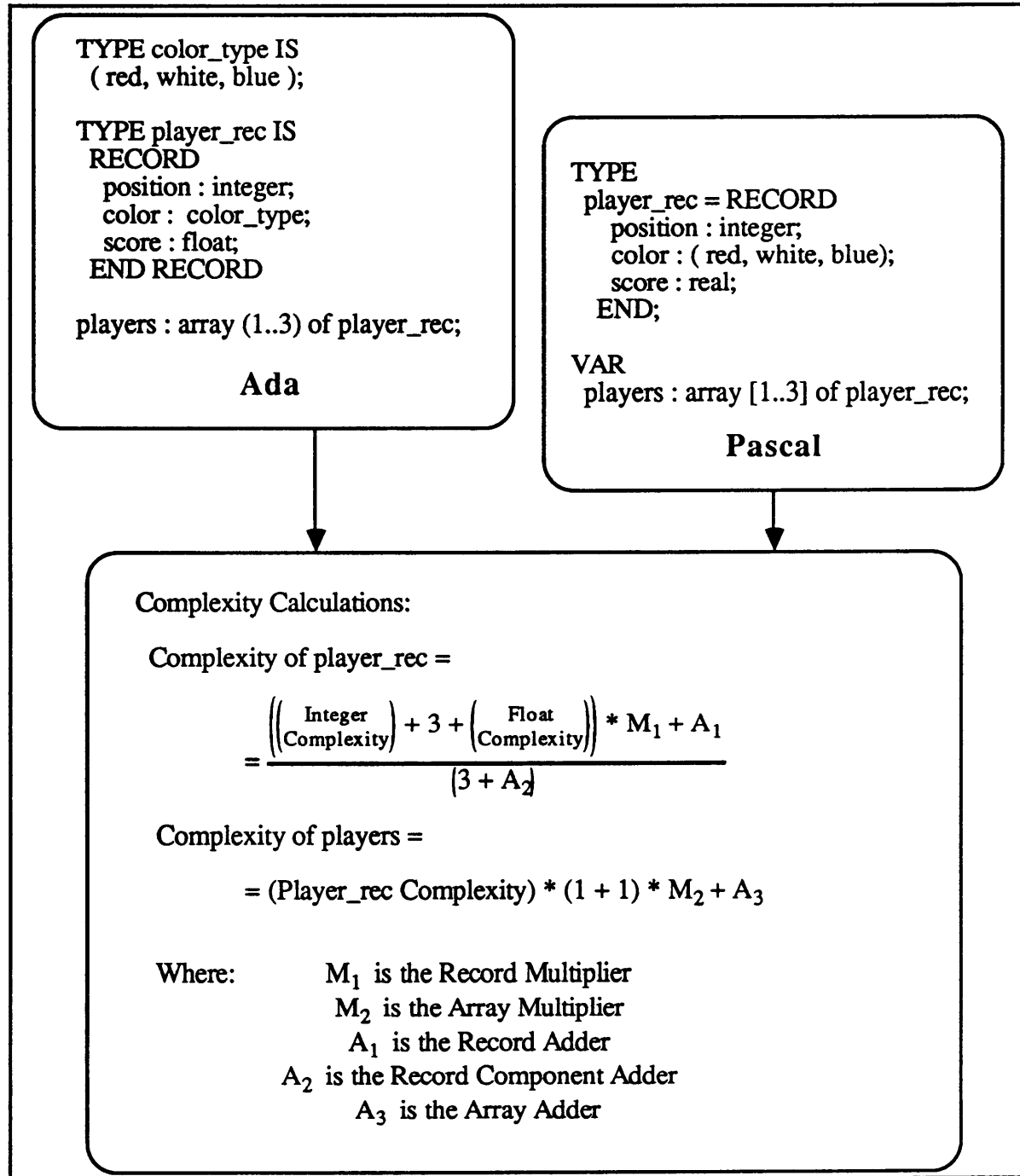


Figure 13: Calculation of Aggregate Types

NATURAL	1
FLOAT	3
CHARACTER	1
DURATION	3
INTEGER	1
SHORT_INTEGER	1
POSITIVE	1
SHORT_FLOAT	2
STRING	3
TINY_INTEGER	1
BOOLEAN	1
DELTA	1
DIGITS	1
RANGE	1
\$\$FIXED	2
\$\$RECORD_TYPE_MULT	1
\$\$RECORD_TYPE_ADDER	1
\$\$RECORD_TYPE_COMP_ADDER	1
\$\$ARRAY_TYPE_MULT	1
\$\$ARRAY_TYPE_ADDER	1
\$\$ACCESS_TYPE	1
\$\$DISCRIMINANT	1
\$\$ENUM_CONST	2

Figure 14: Contents of Pre-Defined Types File

```

procedure main_program is

  x : constant integer := 0;
  int, int2, int3 : integer;
  si : short_integer;
  ti : tiny_integer;
  na : natural;
  po : positive;
  fl : float;
  sf : short_float;
  ch : character;
  st : string(1..5);
  du : duration;
  bo : boolean;

  int_range : integer range 0..20;
  floating_point : float digits 6;
  type fixed_point_type is delta 0.1 range 0.0..1.0;
  fixed_point : fixed_point_type;

  asdf : array (1..3) of integer;
  qwer : array (1..5, 2..5) of float;
  zxcv : array (1..4) of float digits 5;

  type enum_type is (red, white, blue);
  enum_var : enum_type;

  type int_type is range 0..45;
  int_var : int_type;

  type float_type is digits 7;
  float_var : float_type;

  type fixed_type is delta 0.001 range 0.0..1.0;
  fixed_var : fixed_type;

  type array_constr_type is array (1..10, 0..10) of
    float range 0.0..100.0;
  array_constr_var : array_constr_type;

  type array_unconstr_type is array
    (integer range <>, integer range <>) of float;
  array_unconstr_var : array_unconstr_type(1..2, 1..2);

  type access_type is access integer range 1..3;
  access_var : access_type;

  type new_type is new access_type;
  new_var : new_type;

```

Figure 15-a: Example Ada Code

```

type date is
  record
    day : integer range 1..31;
    month : string(1..15);
    year : integer range 0..4000;
  end record;
tomorrow : date;

type buffer(size : integer) is --size is a discriminant
  record
    pos : integer;
    value : string(1..size);
  end record;
buf : buffer(size => 100);

type device is (printer, disk, drum);
type state is (open, closed);
page_size : constant := 256; -- this is a number declaration
type cylinder_index is range 1..128;
type track_number is range 1..500;

--this is a variant record declaration
type peripheral(unit : device := disk) is
  record
    status : state;
    case unit is
      when printer => line_count : integer range 1..page_size;
      when others =>
        cylinder : cylinder_index;
        track : track_number;
      end case;
  end record;

-- note: these are subtypes

subtype drum_unit is peripheral(drum);
subtype disk_unit is peripheral(disk);

writer : peripheral(unit => printer);
archive : disk_unit;

begin
  null;
end;

```

Figure 15-b: Example Ada Code

```

intrinsic maxint;
intrinsic maxfloat;
intrinsic sin;
intrinsic cos;

procedure MAIN_PROGRAM ()
begin
  const (1) X;
  local (1) INT, INT2, INT3;
  local (1) SI;
  local (1) TI;
  local (1) NA;
  local (1) PO;
  local (3) FL;
  local (2) SF;
  local (1) CH;
  local (3) ST;
  local (3) DU;
  local (1) BO;
  local (2) INT_RANGE;
  local (4) FLOATING_POINT;
  local (4) FIXED_POINT;
  local (3) ASDF;
  local (10) QWER;
  local (9) ZXCV;
  const (2) RED; {enumerated literal}
  const (2) WHITE; {enumerated literal}
  const (2) BLUE; {enumerated literal}
  local (3) ENUM_VAR;
  local (2) INT_VAR;
  local (4) FLOAT_VAR;
  local (4) FIXED_VAR;
  local (13) ARRAY_CONSTR_VAR;
  local (12) ARRAY_UNCONSTR_VAR;
  local (3) ACCESS_VAR;
  local (3) NEW_VAR;
  local (2) TOMORROW;
  local (2) BUF;
  const (2) PRINTER; {enumerated literal}
  const (2) DISK; {enumerated literal}
  const (2) DRUM; {enumerated literal}
  const (2) OPEN; {enumerated literal}
  const (2) CLOSED; {enumerated literal}
  const (2) PAGE_SIZE;
  local (2) WRITER;
  local (2) ARCHIVE;
  null;
end;

```

Figure 16: Relation Language Output for Example Ada Code

Operator Complexity

It should be obvious that there are complexity differences among operators due to the different semantic meanings of each. This is most easily seen when '+' (plus) and '**' (exponentiation or involution) are considered. While both are binary operators, their associated complexity is far different, with exponentiation being the more complex. Therefore, a weighting scheme is needed to characterize the relationship among the operators and their complexities.

Operators have set weights. However, these values are contained within an external file that can be modified or 'fine-tuned' to specific needs. The file, COMPLEX.DAT, is used by the complexity manager (within phase 2 of the analyzer) and assigns weights to the various operators. Figure 12 in chapter 2 shows the relationship of this file to the analyzer very clearly. An example line from this file might be:

	OP Number	Weight	Semantics
O	1	1	• This implies that &1 has weight 1
O	2	3	• Operator &2 has weight 3
			... etc ...

The 'O' represents operator data; this specification is necessary because the file COMPLEX.DAT is used to hold not only the operator weights, but also the flow-control weights. The next number represents the code of the operator. This code is stripped from the '&1' operator in relation language by the complexity manager in phase 2. The final

value is the weight associated with the operator. The correspondence between the codes and the operators is shown in figure 17.

OP	Ada	Relation	Semantics	WT
	Source	Language		
&1	ABS(x)	&1 x	Absolute Value	2
&2	NOT x	&2 x	Boolean Not	1
&3	x ** y	x &3 y	Raise x to the power y	3
&4	+ x	&4 x	Unary Plus	2
&5	- x	&5 x	Unary Minus	2
&6	x AND y	x &6 y	AND	1
&7	x OR y	x &7 y	OR	1
&8	x XOR y	x &8 y	Exclusive-OR	3
&9	x = y	x &9 y	Equality Test	1
&10	x /= y	x &10 y	Not Equal	2
&11	x < y	x &11 y	Less Than	1
&12	x <= y	x &12 y	Less Than or Equal To	2
&13	x > y	x &13 y	Greater Than	1
&14	x >= y	x &14 y	Greater Than or Equal To	2
&15	x + y	x &15 y	Binary Plus	1
&16	x - y	x &16 y	Binary Minus	1
&17	x & y	x &17 y	'&' Operator	2
&18	x * y	x &18 y	Multiplication	3
&19	x / y	x &19 y	Division	3
&20	x MOD y	x &20 y	Modular Remainder of x by y	3
&21	x REM y	x &21 y	Remainder of x by y	3
&22	x IN y	x &22 y	Set Membership Test	2
&23	x NOT IN y	x &23 y	Set Exclusion Test	3
&24	IF x AND THEN y	x &24 y	Boolean Shortcut	3
&25	IF x OR ELSE y	x &25 y	Boolean Shortcut	3

Figure 17: Operator Correspondence and Weights

Expression Complexity

Given that there are type and operator complexities, then the next logical step is to define a complexity for a given expression. This complexity is defined as a function of the operator complexities and the operand-type complexities. With this backdrop, it is possible to rank expressions in a way so as to define highly complex expressions. The motivation behind the expression complexity is that highly complex expressions yield more probable locations of errors.

The expression complexity is a function of the type and operator complexities in the expression. Because both the type and operator complexities can be fine-tuned through the predefined types file, this makes this measure very useful for many applications. In actuality, this metric is the sum of type and operator complexities for a given function. Consider the relation language expression:

$W = X \&3 Y \&5 Z;$ { Valid Relation Language }

The expression complexity for this example is:

Expression Complexity ($X \&3 Y \&5 Z$) =

Type Complexity (X)
+ Type Complexity (Y)
+ Type Complexity (Z)
+ Operator Complexity ($\&3$)
+ Operator Complexity ($\&5$)

Of course, it might seem that apples and oranges are being added together; however, this is not the case. Within the complexity manager is a set of dimensional analysis conversion factors. These factors convert all the variables into a unitless expression. There are factors for converting types and operators. Therefore, the true equation is:

$$\text{Expression Complexity (X \&3 Y \&5 Z)} =$$

$$\begin{aligned} & (\text{Type Complexity (X)} \\ & + \text{Type Complexity (Y)} \\ & + \text{Type Complexity (Z)}) * \text{Type Conversion} \\ & + (\text{Operator Complexity (\&3)} \\ & + \text{Operator Complexity (\&5)}) * \text{Operator Conversion} \end{aligned}$$

Thus, all the factors of the complexity yield unitless expressions. Throughout the rest of this thesis this conversion will be assumed.

Flow-Control Complexity

The possible complexity associated with different paths through the source code must also be considered. There is a complexity difference among the flow control statements. In general there are two classes: selection and repetition. Selection statements, in most languages, include IF-THEN-ELSE (and all the derivatives) and CASE, while repetition statements include the two looping constructs: Test-Before and Test-After cases.

The flow-control complexity, like the operator complexity, is not measured but set by the software engineer. These are also contained within the COMPLEX.DAT file, and their format is:

	COND Num	Weight	Semantics
S	1	4	• Operator COND1 has weight 4
S	2	6	• Operator COND2 has weight 6
... etc ...			

Also, the list of correspondence between the selection/repetition statements and the codes is also kept confidential by the software engineer. This list correlates the COND statements with the selection/repetition statements as shown in figure 18.

COND STMT	Semantics	WT
COND1	WHILE - LOOP	2
COND2	FOR - LOOP	2
COND3	LOOP - END LOOP (<i>Basic Loop</i>)	1
COND4	IF - THEN	1
COND5	IF - THEN - ELSE	2
COND6	CASE	4

Figure 18: Selection/Repetition Correspondence and Weights

Environment Complexity

The environment of the communication invocation is very important to the interface complexity measure. It is at the point of the invocation all aspects of the source code come together to produce correct communication, and thus an error at invocation results in the propagation of the error throughout other sections of the source code.

The environment complexity is a function of the expression complexities and the flow-control complexities at each point of communication, and thus a hybrid metric. To simply look at expression or flow-control complexities leaves out important information on the complexity of the path taken to reach the invocation. The environment complexity is defined for each line in the source code, therefore, it can be different from line to line.

The environment complexity is highly interested with the flow-control statements within the code. These flow-control statements dictate the complexity of the environment, and this metric tries to measure this complexity. In other words, this metric weights the flow through various selection and repetition statements with respects to their complexities. This function is the sum of the COND and the related expression complexity of the control expression. For example, consider the relation language code:

x = y &6 z ;	{ Line 1 }
COND1 (a &1 b)	{ Line 2 }
BEGIN	
x = a &9 y;	{ Line 3 }
END ;	
a = x &7 y;	{ Line 4 }

Assume, for the sake of argument, that the environment complexity was initialized to '1' before this section of code was analyzed. At line 1 the environment complexity is still '1', because this line does not alter the environment complexity. However, line 2 does alter the environment complexity because it changes the flow. Line 2 modifies the environment complexity by:

Environment Complexity (*After Line 2*) =

Flow Control Complexity (COND1) +	{ The effect of the COND (selection/repetition) statement on the com- plexity }
Type Complexity (a) +	
Type Complexity (b) +	
Operation Complexity (&1)	{ The effect of the COND control expression on the complexity }

Because the flow control only effects line 3, the expression complexity reverts back to its previous value before line 4.

Variable Complexity

Each variable that effects the communication between two modules (used loosely) must be analyzed as to the extent that its complexity will add to the interface. Variables become more complex to understand as they are modified, because of data-flow; and as variables modify others their complexity must be represented in the modified variable. There are two complexity metrics associated with each variable: Ref_Read and Ref_Write complexities. Each tries to capture different information that will be used to calculate other metrics. Ref_read and ref_write may be viewed in the COMPLEX.OUT file, figure 20, which is the file generated for the user by the complexity manager in phase 2 of the analyzer.

Ref_Read Complexity

The ref_read complexity captures the status of the environment at every 'read' or 'reference' to a variable. Because this is a static analysis, this complexity is simply the sum of all environment complexities at the variable references. All references of the variable are addressed within this expression, including:

- assignment to variable
- use of variable in assignment expression
- use of variable in COND expression
- use of variable in procedure as a parameter

For example, consider the following relation language code:

x = y &6 z ;	{ Line 1 }
COND1 (a &1 b)	{ Line 2 }
BEGIN	
x = a &9 y;	{ Line 3 }
END ;	

If the environment complexity is defined, as above, to be '1' before line 1, then the ref_read complexity of variable a is:

Ref_Read(a <i>after line 2</i>) =	
Environment Complexity (Line 1)	{ Defined as '1' }
Ref_Read(a <i>after line 3</i>) =	
Ref_Read(a <i>after line 2</i>) +	{ Defined above }
Flow-Control Complexity (COND1) +	
Type Complexity (a) +	
Type Complexity (b) +	
Operation Complexity (&1) +	{ Expression Complexity
	of COND control }

This complexity is produced for each variable in the COMPLEX.OUT file. This is useful in finding variables that are greatly used and the (environment) complexity in which they are used.

Ref_Write Complexity

At each modification of a variable the variable becomes more complex through data-flow. Therefore, at each modification the ref_write complexity retains information as to the expression complexity of the right-hand side of the assignment. This complexity can order

the variables by difficulty in regards to data-flow. In this way a variable may be singled out for possible errors if the ref_write complexity is unusually high with regards to other module-related ref_write complexities.

This measurement tries to capture the expression complexity associated with an assignment statement. For each assignment statement, the variable being modified will increase its ref_write complexity by the expression complexity of the right hand side of the modification. In other words, the ref_write complexity is the sum of all assignment expression complexities for a given variable. For example, consider the following piece of relation language code:

```

x = y &6 z ;           { Line 1 }
COND1 (a &1 b)         { Line 2 }
    BEGIN
        x = a &9 y;     { Line 3 }
    END ;

```

The only variable that is written to is 'x', therefore, the ref_write complexity of x is:

$$\begin{aligned}
 \text{Ref_Write}(x \text{ after line 1}) = & \\
 & \text{Type Complexity (y)} + \\
 & \text{Type Complexity (z)} + \\
 & \text{Operation Complexity (\&6)} \quad \{ \text{Expression Complexity} \\
 & \quad \text{of the first assignment --} \\
 & \quad \text{Line 1} \}
 \end{aligned}$$

$$\begin{aligned}
\text{Ref_Write}(x \text{ after line } 3) = & \\
& \text{Ref_Write}(x \text{ after line } 1) + \\
& \text{Type Complexity (a)} + \\
& \text{Type Complexity (y)} + \\
& \text{Operation Complexity (\&9)} \quad \{ \text{Expression Complexity} \\
& \quad \text{of second assignment} \\
& \quad \text{statement -- Line 3} \}
\end{aligned}$$

This complexity can also be found in the COMPLEX.OUT file.

Interface Complexity

The interface complexity is defined to be a 2-tuple containing information as to the environment of the call and the nature of the parameter complexity. The environment element is a function of the environment complexity at each invocation of the communication or interface; the parameter complexity is a function of the expression complexities and the ref_write complexities of the parameters at each invocation.

The first element defines the environment at the time of the call or calls (if multiple calls are present in the code). This is just the sum of all environment complexities at the various times within the source code corresponding to the calls. In this way, the complexity of the environment may be viewed in relation to the time of the calls, as opposed to the complexity of the entire procedure. Figures 19 and 20 show an example.

The second term within the 2-tuple traps the parameter complexity. Currently this is measured in two ways: (1) by the ref_write complexity of all parameters, and (2) by the expression complexity of the parameters. In the first case, the measurement reflects how

much computation went into the generation of the parameters, and thus their complexity. The second alternative reflects the inherent complexity associated with the understanding of the parameters in terms of the parameters expression complexities.

Output from Analyzer

All of the above metrics can be viewed by: examining the file COMPLEX.OUT, the translated relation language, and working with phase 3. The type complexities, because they are generated in the first phase, can be viewed within the relation language as well as within COMPLEX.OUT. The flow-control and operator complexities can be viewed inside the COMPLEX.DAT file. The variable metrics, ref_read and ref_write, can be found within the COMPLEX.OUT file, as well as the interface complexities.

Figures 19 and 20 show an example output of a COMPLEX.OUT file. Figure 19 gives the relation language code that is represented by the COMPLEX.OUT file in figure 20. It is easy to note the type complexities within the relation language code; the type complexities are contained within the variable definitions. This example is a simple two-line procedure that calls a nested one-line procedure.

Figure 20 is the COMPLEX.OUT file for figure 19. Figure 20 tries to 'point-out' all the components contained within the COMPLEX.OUT file. To begin with, because the flow-control (structure) and operator complexities can be changed, they are enumerated with their values at the top of all the COMPLEX.OUT files. This is to insure that comparisons between relation language code will not occur unless the weights are equal.

The next section within the COMPLEX.OUT file is an enumeration of all the procedures. For each procedure all the variables associated with it are listed along with: ref_read complexity, number of accesses, average ref_read complexity, ref_write complexity, number of modifications, average ref_write complexity, variable type, variable type complexity, and of course the variable name. The variable type is one of the following: local, parameter, procedure, structure, or extern; while, the variable type complexity is the complexity calculated within phase 1. There is also a special variable name 'zzz-Numeric-Constants' that represents all the numeric constants within the procedure. This variable can be important in assessing the effects or influence of constants on a procedure.

Two other aspects of a variable are if it is a parameter and/or if it is a structure. If the variable is a parameter to a procedure(s) then indirect communication may be observed. For example, if procedure A1 passes variable X to procedure A11 and also to procedure A12 then there is indirect communication between A11 and A12. This is identified by the 'Link to' procedure number within the variable information section of the COMPLEX.OUT file. A variable that has more than 1 link implies indirect communication between the listed procedures. The other aspect, whether or not a variable is a relation language structure, implies global communication. This 'STRUCT' variable could be accessed or modified anywhere within the code (subject to visibility rules). A breakdown is given for each of these variables, denoted by: (BrDn) followed by the procedure number and name and the associated ref_read and ref_write numbers. These numbers give the software engineer the vision to see where in the code these global variables are mostly accessed and modified.

After each procedures' variables are described in terms of these ref_read and ref_write metrics, then there is a listing of the interface complexities associated with that

procedure. In figure 20 there is an interface complexity associated with main and jj, and it is described in main (the calling procedure).

Finally at the end of all the procedure listings is an enumeration of all the procedure names, their nesting lexical nesting level, and a unique identifier. These numbers are used within the structure breakdowns and the links.

```
const (2) j;  
struct (10) i;  
  
procedure jj ( (24) tt )  
begin  
  tt = 100;  
  null;  
end;  
  
i = i &6 i;  
  
jj(i & i & i);  
  
null;  
null
```

Figure 19: Example Relation Language

Conclusion

While many of the variables are left for the software engineer to define (for his/her particular applications) it is the intention of this study to give specific weights that yield good measures of the interface complexities. It is possible, however, that the weights can be modified.

This chapter introduced and defined the metrics that will be of interest in this research. Chapter 4 discusses the data and the analysis used within the study, and chapter 5 discusses some conclusions.

Chapter 4: Data, Results, and Analysis

Introduction

This chapter introduces the data and shows the results derived in this research. The analysis of the data is also handled within this section, while conclusions are presented in the next chapter.

The analysis of the data was accomplished in three stages. Stage 1 is a small experiment designed to compare different type weights so as to show a correspondence among the interface complexity metrics. Stage 2 involves correlating calculated interface complexity metrics with calculated known metrics, such as lines of code or any of Halstead's Software Science indicators. Finally, stage 3 involves a subjective evaluation of source code so as to further support interface complexity metrics.

Data

In order to validate the metrics that have been proposed in this research, a large system was used. This system was donated to The Virginia Tech Metrics Group for re-

search purposes by Software Productivity Solutions (SPS), Inc, Melbourne, Florida. This system proved large enough to demonstrate the usefulness of the metrics.

It should be stressed that this research spanned over only this one project. Yet, in the same vain, this is not student data but a system that is currently in use. This system is an SPS product, and that implies the confidence and quality that is present in SPS work. SPS was very helpful in supplying this system.

The system, written in Ada, is a design language processor. This system works over an Ada-based Design Language (ADL). It is engineered to fit into version 1.0 of the ADL Development System (ADS). The system is referred to ADS.

The system is divided into four separate sections, by design, called subsystems. These are:

- ADL Processor 195 Files 30204 Lines
- ADL Report Generator 28 Files 3874 Lines
- ADL Database Manager 137 Files 43019 Lines
- Support Subsystem 39 Files 6702 Lines

The program is controlled by the ADL Processor system, which is a parser of an Ada like PDL.

While the functionality of this system is interesting, it is a side note to the structure and interfaces for this study.

Small Experiment

The formulae introduced within the previous chapters rely on a predetermined set of weights. The possible combinations of these weights is infinite. This research proposed the new interface complexity metric measurement, implemented it, and showed preliminary results. Due to the wide diversity of these weights and the absence of an error history to substantiate findings, an optimum set of weights was not attained.

To show the effects of a single predefined weight, a complexity was selected and three test cases were generated. These tests were generated from the Ada system, each with the `RECORD_TYPE_MULTIPLIER` of 1.0, 1.5, and 2.0 respectively referred to as test 1, test 2, and test 3. Test 0, the control case, was generated from the same Ada system without any of the new complexity measures (i.e. type, operator, flow-control, and the hybrids of these.)

The four test cases, test 0, 1, 2, and 3, are all derivatives of the same Ada system. Therefore, they contain the same number of interface complexities and these can be compared. The interface complexity is a 2-tuple: access complexity and parameter complexity. There were 20,975 observations of the interface complexity. In other words, there were 20,975 communication invocations within the system. The analysis also includes the sum of all interface complexities into each procedure and the sum of all interface complexities out of each procedure.

The access complexity of the interface complexity metrics is defined by the sum of environment complexities and the number of accesses. For this experiment the number of accesses is constant throughout the test cases, it is only the relationship among the access complexities that is interesting. Figure 21 shows the information that is important to define this relationship. ACC_1, ACC_2, and ACC_3 refer to the access complexities of the three sample test cases, respectively. ACC_0 represents the access complexity without considering environment complexity, in other words, it is a control case.

Figure 21 shows that there is a high correlation among ACC_1, ACC_2, and ACC_3. This finding should not be a surprise -- these tests only differ in the calculation of the RECORD type complexity. The point to notice is the degree of correlation of ACC_0 to the other three access complexities -- it is not outstanding, hence, these three access complexities measure a different aspect.

	ACC_3	ACC_2	ACC_1	ACC_0	Accesses
Accesses	0.69788	0.74398	0.72507	0.93337	1.0000
ACC_0	0.75893	0.81443	0.79148	1.0000	
ACC_1	0.99532	0.99637	1.0000		
ACC_2	0.98404	1.0000			
ACC_3	1.0000				

Figure 21: Access Complexity Correlations

Current Weights

The current weights for this experiment are defined in Figures 14, 17 and 18. These numbers are defined with intuition in mind. They represent what we consider as possible, plausible weights. Of course, because these weights can be changed, this represents only one such mapping out of an infinite number.

As mentioned before, an error history of the system analyzed is not available. If an error history were available, then a better set of weights could be attained through an iteration process over possible values of the weights correlated to the error history. This would then model an error predictor equation regards to the interface complexity. This equation could be used starting at design time, when the interfaces are first known, on through maintenance. However, no error history is available.

Possible Uses

As mentioned before, these metrics are tunable. The weights that are adjustable include: predefined types, operator and control-flow weights. The weights that a software engineer might tune would depend upon his/her needs.

One such example of tuning these metrics for a specific need could be in database management. In this case it might be important for the software engineer to weight selection structures higher than looping structures, and give a much higher weighting to array and record types. In this way the software engineer could define areas of comparison between the larger types. This is an example just for demonstration.

Each different environment and application domain may need its own set of weights that do not correspond to other sets. Also, it might be interesting in the future to test these weights over different programming languages. Intuitively, the weights should be different over different languages.

Interface Complexities

As mentioned, there were 20,975 observations of the Interface Complexity. In other words, there were 20,975 calls within the ADS system. Each of these Interface Complexities were of the form:

p1 p2 accn acc para1 para2

This information contains the directional communicating procedure numbers (p1 communicates to p2), the number of calls (accn), the sum of the environment complexities at each call (acc), and two different parameter complexities. These parameter complexities are based on different views of the data. Para1 represents the sum of the type and operator complexities of the parameter expression, while, para2 captures the modification and operator complexities at the time of invocation.

These numbers capture the interface complexities at a calling level. However, the interface complexity should represent the complexity of the entire procedure to other procedures. Therefore, the data needed to be converted and compacted.

In order to view the interface complexity on a per procedure basis the data was collapsed. Reducing this data was a multi-step process. First, the interface complexities for equal calling and called procedures were collected together. This first compaction took place within the complexity manager section of phase 2 of the analyzer. Next all the interface complexity metrics directed from procedure p1 to any other procedure were collected into the p1 measures, and, likewise, all the interface complexity metrics directed from any procedure into procedure p1 were collected into the p1 measures. Therefore, at the end, each procedure had 8 new measures that represented the interface complexity.

These new measures are shown pictorially in Figure 22. Notice that there are four measures associated from the calling procedure (FAccN, FAcc, FP1, FP2), and four measures from the procedure (X in this case) to the called procedures (AccN, Acc, P1, P2). As with the interface complexity, these numbers represent accn, acc, para1 and para2 respectively for the calling and called procedures.

Four measures are derived from the current procedures code: AccN, Acc, P1, and P2. The first measure, AccN, represents the number of accesses to other procedures. This number should correlate highly with metrics that count code invocations. Acc examines the environment complexity at the time of each call. This environment complexity is the sum of flow-control and expression complexities, therefore, this measure intuitively should correlate with code metrics that count environment tokens (Halstead's Software Science, for instance.) This same logic can be applied to the parameter complexities, P1 and P2, for both of these are counts of tokens. Granted, these tokens are given a weight. P1 represents the complexity associated with the type of the parameters, while, P2 measures the modification complexity of the parameters.

The other measures, FAccN, FAcc, FP1, and FP2, are derived from all the procedures that call the given procedure. They are the sum of all the AccN, Acc, P1 and P2 measures respectively, with the notation of an 'F' to imply 'from procedure'. Therefore, because these are based on code contained within other procedures, the correlations of these measures with established metrics should be low.

The instances these metrics for all the procedures are contained within Appendix A (top 1%), and Appendix B contains the calculated established metrics for each procedure listed within Appendix A.

For Each Procedure:

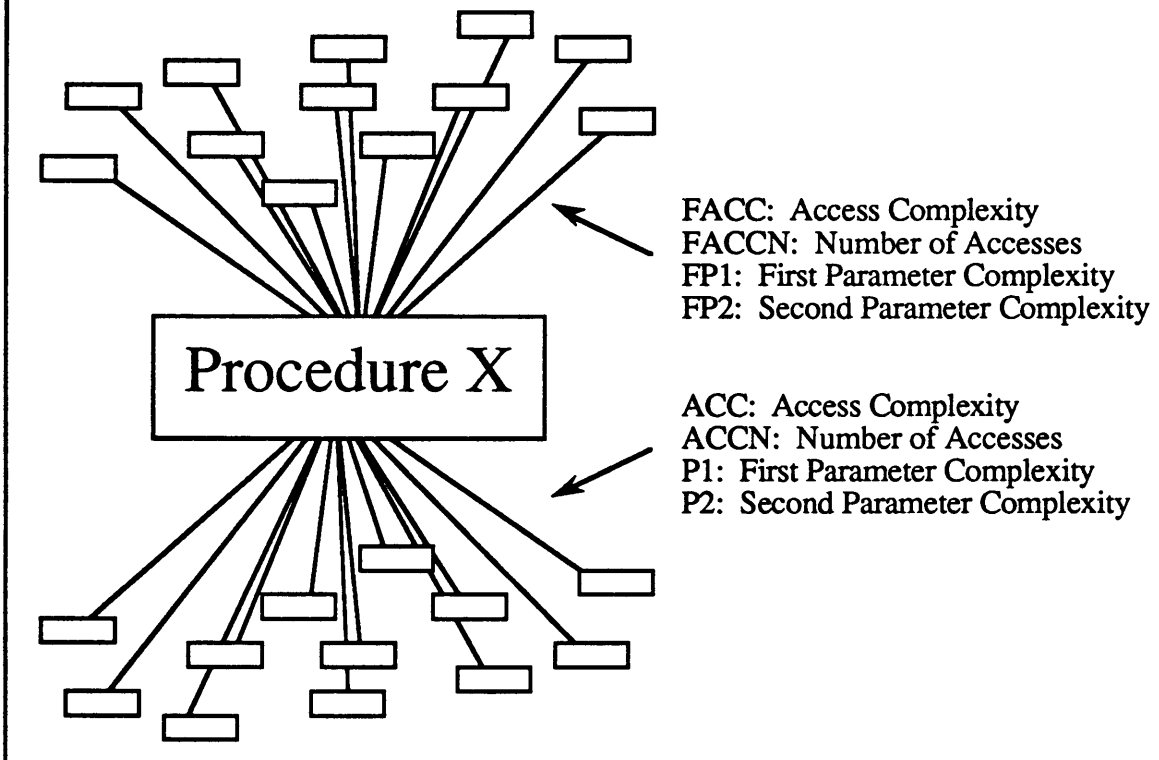


Figure 22: View of Interface Complexity

Method of Analysis

The interface complexity metrics are designed to overcome the many weaknesses of currently used metrics. The interface complexity metrics capture the degree of relationship among communicating procedures differently than Information Flow [HENS81b] and McClure's Invocation Complexity [MCCC78]. Not only is the environment of the invocation captured, but also the modification and type complexity of the parameters are captured as well.

This research set out to elucidate the following:

- Definition of the Interface Complexity Metrics
- Implementation of Interface Complexity Metrics
- Validation of Interface Complexity Metrics

Chapters 2 and 3 define and show the implementation of the interface complexity metrics.

The validation of the interface complexity metrics follow these lines:

- Getting the data (ADS) from SPS
- Analyzing the data using the Analyzer presented in Chapter 2
- Gathering and reducing the interface complexity data
- Correlating the interface complexity data to other well-known metrics

This validation is admittedly only on one system with subjective initial predefined type weights due to the lack of an error history. If, however, an error history were present, then a more thorough investigation could be accomplished with the fine tuning ability of the complexity manager. Yet, even without an error history correlations of these metrics with

other well known and validated metrics can be used so as to increase the credibility of these metrics.

Without an error history, these numbers and functions presented here lack a degree of believability. Therefore, the final phase of the analysis is a subjective examination of the source code.

Analysis

The ADS system was run through the analyzer with output for the code metrics, interface complexity metrics, and hybrid metrics. The next step is to correlate these interface complexity metrics with well known and previously validated metrics. The statistics generated are Pearson correlation coefficients.

Known Metrics Correlation

First it is important to give an understanding of the metrics that are used to correlate to the interface complexity metrics. These metrics were discussed in chapter 1, and figure 23 gives an abbreviation of each of the metrics.

Figure 24 gives the correlations between these metrics. This figure points out the relationship among the code metrics: LOC, N, V, E, and CC. And also, the differences among the code metrics and the hybrid metrics: WOOD and INFO. The metrics generated for this system are discussed in great length in [CHAB89].

Abbreviation	Meaning
LOC	Lines of Code
N	Halstead's N
V	Halstead's V
E	Halstead's E
CC	McCabe's Cyclomatic Complexity
WOOD	Woodfield's Review Complexity
INFO	Henry-Kafura Information Flow Complexity

Figure 23: List of Abbreviations

	LOC	N	V	E	CC	WOOD	INFO
INFO	0.030	0.067	0.061	0.029	0.000	0.058	1.000
WOOD	0.561	0.935	0.962	0.991	0.535	1.000	
CC	0.538	0.581	0.549	0.602	1.000		
E	0.574	0.932	0.954	1.000			
V	0.610	0.991	1.000				
N	0.634	1.000					
LOC	1.000						

Figure 24: Correlations Among Established Metrics

Known Metrics VS Interface Complexity Metrics

The first question is how well do the eight interface complexity measures correlate to established metrics. This question is important because of the need not to generate useless or redundant metrics. Figure 25 shows the correlation among LOC, N, V, E, CC, WOOD, and INFO with AccN, Acc, P1, P2, FAccN, FAcc, FP1 and FP2.

In figure 25 it is interesting to note the moderate to high correlations between N, V, and E and AccN, Acc, and P1. This is due to the fact that these measures are generated partially from counts of code token weights, while Halstead only counts the number of tokens. In the same vein, it is very understandable why INFO correlates very poorly to these measures, as would be expected. This shows that AccN, Acc, and P1 are gathering data somewhat like Halstead's Software Science Indicators, but still differently.

Another interesting fact that figure 25 brings to light is the very low correlation of FAccN, FAcc, FP1 and FP2 to the code metrics. This is logical and expected. These metrics measure the complexities associated with the procedures calling the current procedure under inspection. Therefore, no correlation between the calling procedures interface complexity with the current procedures code or hybrid metrics are found.

Also with regards to figure 25, it should be noted that the first parameter complexity (P1 and FP1) in general show higher correlations than the second parameter complexity (P2 and FP2). Later there is a discussion on the correlation of functions of the interface complexity metrics with the well known metrics, and these functions involving P1 and FP1 show a higher correlation to selected metrics, and are thus more meaningful. However,

this is just a by-product of the type weights defined for this experiment and should not always be taken as axiomatic. Different type weights yield different values of P1.

Finally, figure 25 also shows that if Ada is used as a design language, then several established metrics for the final code can be estimated from this design. This is due to the high correlation between AccN and the established metrics N, V, E, and WOOD. Therefore, given an adequately detailed design with all interfaces in place, Halstead's Software Science Indicators can be gathered. Thus, helping to choose between designs by just knowing the number of accesses (AccN).

Figure 26 shows the correlation among the interface complexity metrics. These numbers help to show the differences within the two groups of interface complexity metrics: (AccN, Acc, P1 and P2) and (FAccN, FAcc, FP1 and FP2). The interesting note is that while P1 and P2 correspond highly, FP1 and FP2 correspond poorly. This can be attributed to the variation, within the procedure calls environments and parameters, which spans over all the procedures calling the procedure in question.

These correlations are interesting, however, if taken singularly they fail to take advantage of the richness of the all the interface complexity metrics. In order to exploit these metrics, and thus create more meaning, functions of these metrics are created.

	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
LOC	0.575	0.507	0.610	0.394	0.025	0.057	0.033	-0.006
N	0.924	0.695	0.943	0.566	0.031	0.062	0.048	0.001
V	0.949	0.694	0.953	0.622	0.027	0.054	0.042	0.001
E	0.930	0.648	0.935	0.642	0.004	0.011	0.030	-0.001
CC	0.620	0.651	0.690	0.545	-0.017	-0.008	0.049	-0.011
WOOD	0.933	0.646	0.922	0.656	0.019	0.031	0.043	0.005
INFO	0.064	0.034	0.063	0.018	0.448	0.389	0.655	0.166

Figure 25: Correlations Among Established Metrics and Interface Complexity Metrics

	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
FP2	0.002	-0.006	0.002	0.000	0.257	0.271	0.401	1.000
FP1	0.039	0.026	0.048	0.006	0.714	0.482	1.000	
FAcc	0.012	0.000	0.010	0.003	0.801	1.000		
FAccN	-0.001	-0.011	-0.002	0.001	1.000			
P2	0.790	0.742	0.718	1.000				
P1	0.961	0.769	1.000					
Acc	0.808	1.000						
AccN	1.000							

Figure 26: Correlations Among Interface Complexity Metrics

Functions of the Interface Complexity Metrics

It is very important to create functions that intuitively hold meaning for the researcher or software engineer. In many cases metrics are viewed by managers, and their knowledge of a system might be limited to only these measures. Thus, a function that is easily understood, by software engineers and managers alike, can be incorporated more easily into a working environment. The functions that are derived and addressed are all correlated with lines of code (LOC), Halstead's E (E), Woodfield's review complexity (WOOD), and Henry-Kafura's information flow complexity (INFO).

This research formed two different sets of equations for two different parameter complexities, for a total of four equations. These equations differ in that one set is based on a summation operator, while the second set is based over a multiplication operator. Summation and multiplication were used because of their intuitive applicability. It is the intent of this research to capture the entire interface complexity over a procedure, hence, the operands of these functions represent the interfaces into and out-of the procedure in question.

One such function investigated incorporates both the interface complexity coming into and out of the procedure in question. This function, referred to as $F1_+$, is $(Acc * P1) + (FAcc * FP1)$. The first product represents the complexity into the procedure, while the second product stands for the complexity out of the procedure. Figure 27 shows the correlations to established metrics. The interesting aspect of this function is that it correlates poorly to both code, structure and hybrid metrics. This implies that this function is

measuring something different. It cannot be said whether this difference is good or bad but that it is different.

$F1_+ : (Acc * P1) + (FAcc * FP1)$	
Correlation	
Coefficient	
LOC	0.28539
E	0.47745
WOOD	0.49844
INFO	0.50984

Figure 27: **Function $F1_+$ vs Established Metrics**

The function $F1_*$, $(Acc * P1) * (Facc * FP1)$ is a product function similar to the sum function above. However, its correlations are very different. In this case it is compared to another function, $F2_*$, $(Acc * P2) * (Facc * FP2)$, which represents logically the same concept, only with the second parameter complexity. Table 28 shows the relationship between these two functions and established metrics.

The first aspect to notice, from figures 27 and 28, is the difference in the correlations between functions $F1_+$ and $F1_*$. With the only difference being to sum or take the product, there is bewildering difference between the correlations. In one case, $F1_+$, there is poor correlation to all measures, while with $F1_*$ there is much better correlation to INFO and poorer to the code metrics. However, this could be a reaction to the squared term within INFO, and the product grows much faster in $F1_*$ than the sums within LOC, E and WOOD.

The next aspect, derived from figure 28, is the similarities in correlations between the two functions $F1_*$ and $F2_*$. If the correlation between $P1$ vs. $P2$, and $FP1$ vs. $FP2$ are considered, then the similarities of the correlations fall into line and are not surprising.

F1*: (Acc * P1) * (FAcc * FP1)		
F2*: (Acc * P2) * (FAcc * FP2)		
Correlation Coefficients		
	F1*	F2*
LOC	0.03311	0.03270
E	0.02996	0.02950
WOOD	0.06208	0.06174
INFO	0.91971	0.91970

Figure 28: Functions F1* & F2* vs Established Metrics

The final two functions that are examined are concerned with the second parameter complexity of the interface complexity metrics. Functions F2₊ and F2_{*} mirror functions F1₊ and F1_{*} respectively, with the sole exception that they deal with the second parameter complexity of the interface complexity metrics, not the first. These two functions are of interest because of their inverseness to the established metrics. Function F2₊ correlates moderately to E and WOOD, while function F2_{*} correlates very poorly to E and WOOD. Furthermore, function F2_{*} also correlates very high with INFO while F2₊ correlates very low. The second parameter functions were "inverses" to each other, and because of the mirroring of the first and second parameter functions, this trait could be expected in the first parameter function correlations. Yet, this "inverseness" is not present in the first parameter functions. This is because of some property of F1₊.

$F2_+ : (Acc * P2) + (FAcc * FP2)$		
$F2_* : (Acc * P2) * (FAcc * FP2)$		
Correlation Coefficients		
	$F2_+$	$F2_*$
LOC	0.34048	0.03270
E	0.57726	0.02950
WOOD	0.59038	0.06174
INFO	0.06788	0.91970

Figure 29: Functions $F2_+$ & $F2_*$ vs Known Metrics

Examining these numbers helps to show the credibility of the interface complexity metrics. Yet, a subjective examination of the code is needed to determine the impact of the interface metrics.

Subjective Examination

The data that from SPS included all the source code for the ADS system. Therefore, it was determined that some rationale be developed to correspond to the metrics that were calculated. Due to the lack of an error history a subjective examination of the code was deemed necessary.

Strategy for Examination

To implement the subjective examination on the interface complexity metrics the eight measures (Acc, AccN, P1, P2, FAcc, FAccN, FP1, and FP2) were considered. Each of the eight measures were rank ordered, and several procedures corresponding to extremely high values were examined. All of these procedures were within the top 1% of the interface complexity metric in question. These procedures were examined from an intuitive stand point, as well as from a more rigorous stance that involved the examination of the established metrics.

Figure 30 displays the procedure numbers, generated within the Analyzer, with extreme values of the eight interface complexity metrics. The bold-italicized numbers represent procedures that are extremes in more than one interface complexity metric (this includes all eight measures.) One procedure number per cell represents that the complexity

metric was summed up over that procedure, while two procedure numbers represent a specific communication between the two procedures.

Figure 31 lists the procedures, in numerical order, found in figure 30 with its corresponding percentile to established metrics. It should be noted that this is a frequency percentile, in other words, being in the top 1% (> 99.0 in figure 31) corresponds to having one of the top 1% values. This is not to be confused with having a value only 1% different from the maximum.

Complexity	Measure	Procedure	Complexity	Measure	Procedure
	Value	Number		Value	Number
AccN	1634	2754	FAccN	1085	1937
	1125	6829		841	6661
	605	1939		829	3932
				819	6651
Acc	8868	2754		727	2333
	3544	4377			
	3380	1939	FAcc	11214	3932
				8552	992
P1	6887	2754		8474	6651
	5235	6829		7494	2521
	3551	1939		6880	2733
	2722	3915			
	2075	1174	FP1	5839	6651
				5775	2521
P2	49187	2754		5667	1937
	16994	1939		5041	2333
	8769	1978			
	2842	4377	FP2	12170	2754 $\Rightarrow$ 4961
	1454	5760		11597	6651
				10466	4922
				10225	2754 $\Rightarrow$ 4958

Figure 30 Extreme Values of Interface Complexity Metrics

Procedure Number	LOC	E	CC	WOOD	INFO
992	> 99.0	98.3	22.9	> 99.0	> 99.0
1174	> 99.0	> 99.0	> 99.0	> 99.0	> 99.0
1937	58.3	29.3	22.9	35.4	> 99.0
1939	> 99.0	> 99.0	> 99.0	> 99.0	> 99.0
1978	> 99.0	> 99.0	73.4	> 99.0	> 99.0
2333	33.2	32.4	22.9	42.1	> 99.0
2521	> 99.0	> 99.0	83.0	> 99.0	> 99.0
2733	94.6	87.7	22.9	95.2	> 99.0
2754	> 99.0	> 99.0	> 99.0	> 99.0	> 99.0
3915	> 99.0	> 99.0	22.9	> 99.0	91.3
3932	84.9	74.0	22.9	83.5	> 99.0
4377	> 99.0	> 99.0	> 99.0	> 99.0	90.9
4922	78.1	79.0	59.0	78.3	96.7
4958	41.6	53.2	22.9	50.6	86.3
4961	41.6	48.3	22.9	49.0	89.4
5760	> 99.0	> 99.0	> 99.0	> 99.0	95.7
6651	33.2	32.4	22.9	42.1	> 99.0
6661	41.6	43.4	22.9	50.6	> 99.0
6829	> 99.0	> 99.0	22.9	> 99.0	91.8

Figure 31: Frequency Count Percentile Breakdown of Extreme IC Procedures

Examination Results

Each interface complexity metric is addressed by examining the extremes within the two areas: measures from within the current procedure (AccN, Acc, P1, and P2), and measures associated with the calling procedures (FAccN, FAcc, FP1, and FP2).

Measures within Current Procedure

The four interface complexity metrics, AccN, Acc, P1 and P2 are discussed here according to the extreme values contained within figure 30.

Extreme AccN Measures

Examination of figure 30 shows that procedure 2754 dominates in all four measures. This is not surprising because procedure 2754 is the main parser driver (yyparse within the adl_parser package). Why should this generate large interface complexity measures? This procedure can be described as a very large CASE statement, which accounts for the access complexity, Acc. Also for the most part, each alternative within the CASE statement is a communication invocation of the either a call or reference to a variable within another package, often both, thus increasing AccN. This is a very complex routine; however, it should be noted that it is the nature of parsers to be complex in this manner, especially one for Ada. This is the location in which the parse tree is derived, therefore there are great amounts of highly complex information flowing out, hence the high parameter complexities for P1 and P2. This procedure, with regards to the other known metrics, was noticeable in LOC, N, V, E, CC, WOOD, and INFO (all in top 1% as shown in figure 31).

The second procedure under this category, AccN, 6829, is an initialization procedure (`initialize_standard_scope` from `visibility_tree_manager` package). This procedure initializes the standard (root) scope of a parsing visibility tree. It can be seen that this procedure is simply an enumeration of many procedure calls to create a tree that houses Ada Pragma, Reserved Words, Exceptions, Types, and a set of characters. For each individual item several calls are made with often many parameters. During the subjective investigation into this procedure it was noted that the character initializations into the tree were enumerated out; yet, this task could be accomplished by a looping structure at a savings of 276 procedural invocations (3 procedures called for 92 different characters). The calls would still be made, but within a loop not from straight line code which is better programming than the enumeration style. The interface complexity metrics pointed this programming style out. This procedure's INFO measure failed to capture the necessary complexity, while the code metrics were noticeable (top 1%, excluding CC). It can be said, however, that the code metrics are only noticeable because of the length not because of the interconnections and communication. Therefore, the code metrics reached the same conclusion as the interface complexities, but the reasoning behind the code metrics is much different.

The third, and final, procedure listed as an extreme case of AccN is a database routine (`copy` within `adl_database` package). Procedure 1939, as AccN testifies, is full of reference and invocation communications. As above, the code metrics are high for this case (top 1%).

Each of the procedures flagged by AccN were also noted by the established metrics. This fact, along with the data from figure 25, helps further demonstrate the correspondence of AccN and the known metrics.

Extreme Acc Measures

Figure 30 lists two procedures under both AccN and Acc: 2574 (parser) and 1939 (database copy routine). It should be easy to see why both of these procedures would also have a great interface complexity access metric. Yet, what about procedure 6829, an initialization (`initialize_standard_scope` within `visibility_tree_manager` package), why is it not included here? Examination of this procedure shows that while it contains many communications through references and calls, all of these are contained within straight line code. Thus, procedure 6829 has a relatively low interface complexity access metric.

The interesting question is where did procedure 4377 come from? This procedure does not have an extreme AccN, therefore, the communication is complex because of the environment -- not just a side product of the number of calls. Procedure 4377 is a lexical analyzer (`yylex` within `lexical_analyzer` package). The code contains, like the parser, a large CASE statement nested within a WHILE-DO statement. Also within this procedure (function) the code metrics generated are high (top 1%), and CC, as expected, is very high (top 1%). However, as before, these numbers just reflect the effect of the length as opposed to the communication environment which Acc models. INFO also failed to take notice of this procedure.

These extreme Acc measures relate well with the code metrics, as shown here and in figure 25.

Extreme P1 Measures

The P1 measure is generated from the sum of the type and operator complexities of the parameters, summed over all calls. Therefore, this metric is sensitive, in a manner, to the number of calls. So it can be easily seen why procedures 2754 (parser), 6829 (initialization of standard scope), and 1939 (database copy) are effected, and thus in the extremes. Yet, the meaning of P1 should not be lost, because it is actually the type weights that contribute the most to this metric, and these procedures pass very complex types.

Procedure 1174 is not listed in either the AccN or Acc extremes, so this leads the conclusion that the parameters are complex as opposed to any influence from the number of communications. This procedure (meets_constraints within ada_unit package) is composed of 20 inline CASE statements that decode and take action on given operators. The parameter complexity is generated from the fact that much of the manipulation within this procedure is in regard to record fields, and therefore, the entire complexity of the record is considered. Records have higher complexity, and this translates into a higher complexity for the parameters.

Extreme P2 Measures

The P2 measure examines, by sum, the modification complexity of the parameters along with any operator complexities. This parameter complexity is more intuitively aligned with the data flow concept, in other words, this measure tries to capture the modification complexity associated with each parameter. Therefore, procedures with high P2 values are modifying their data to a great extent, so by this reasoning, it is not strange to see procedures 2754 (parser) and 1939 (database copy) with high values.

It is remarkable, however, to see procedure 4377 (lexical analyzer) presented here. Recall that this procedure did not demonstrate an overly high AccN, but did show a high Acc, leading to the speculation that there were not relatively many communications, but they were taking place in very highly complex environments. The inclusion of 4377 in the P2 extreme list adds to this hypothesis that the parameters were modified greatly. This should fall in line with our concept of a lexical analyzer -- the modifications come from the I/O and internal manipulation associated with token fetching.

Procedure 1978 represents a routine with intuitively a very high interface complexity. This routine (Create within ADL_Database package) contains many invocations through procedure and function calls.

Measures from Calling Procedures

This next section of subjective analysis centers around the four measures attained from the calling procedures. These are in fact the AccN, Acc, P1 and P2 measures summed from all the calling procedures -- which create FAccN, FAcc, FP1 and FP2 respectively. Therefore, it is not the procedures calls or code that has the influence so inspecting these procedures must include examination of the procedures that communicate with them.

Within this section there are several instances of utilities being flagged as extreme. This should is not alarming because, while utilities are often simple, they are widely used and abused. The data provided by SPS was broken into for subsystems, one of which was utility support, perhaps the utilities should not have been included in the analysis. This

would lead to the monitoring of other procedures which might be more useful. However, this study is executed over the entire data set, including utilities.

Extreme FAccN Measures

FAccN defines the number of calls, or communications, to the current procedure. Therefore, intuitively this defines the procedure that is called, or communicated with, the most. In this study, the procedure 1937 had the greatest FAccN value. This procedure (convert within `adl_database` package, renamed `unchecked_convert`) has many flows into the procedure, as FAccN and INFO (top 1%) depict. This procedure is also an extreme for the interface complexity metric FP1.

The next procedure examined, 6661, is a procedure within an instantiation of a generic package (`variable_string_of` within the `text_utilities` package, renamed `string_utilities`). This is a simple one line routine that converts a string to a variable string type; however, it is not the simplicity of this procedure that FAccN measures, but the high degree of calls to this procedure from other procedures. This is a support function that is widely called, and thus this degree of FAccN is not surprising.

The third procedure on the FAccN list demonstrates a characteristic of Ada. This procedure, 3932, is called `definitions`, and it is just that: a package of definitions. However, due to the makeup of Phase 1 of the analyzer, all references to a variable within this package are treated as communication into the package. And, like wise, the interface complexity metrics capture this communication. Within this procedure are definitions of null variables, used through out the code for testing and comparisons, therefore, the value of 829, which is relatively high, is explainable.

Procedure 6651 is interesting since it is found under extreme in all four metrics: FAccN, FAcc, FP1 and FP2. Due to this trait this procedure is dealt with later. Also, procedure 2333 is shown later.

Extreme FAcc Measures

The FAcc measures the access complexities within all calling procedures for a given procedure. The highest access complexity is assigned to procedure 3932 (definitions). This is easily understood since the variables defined within the definitions package are used throughout the code (high FAccN) and this usage adds to the FAcc complexity. This intuitively stands to reason that a procedure with a high fan in would have a high access complexity; however, this is not always the case. For example, procedure 1937 produced the highest FAccN value, yet it was not selected as an extreme FAcc value. This would tend to imply that while there is a great deal of communication, the environments of the communications are not relatively great. This reasoning also applies to the second routine, 992 (adl_database package).

Procedure 2521 is a utility within the database system. This procedure (open within adl_database package) has two formal parameters: a database type parameter and a string parameter, which represent a database and its name respectively. The database type parameter is highly complex, however, it is the environment (access) complexity of the calls into this procedure that is of interest. The discussion of this procedure continues within the discussion of extreme FP1 measures.

The final procedure within the extreme list of the complexity interface metric FAcc is 2733 -- a package of type definitions (`adl_database_types`). This, again, is another particularity of Ada. This file contains several sets of enumerated types, and due to the translation of the Ada source, the values of the enumerated types reside within this package. Therefore, all references to an enumerated type element are references to this package. In other languages, this case might not be possible, hence, it is important to understand the process by which Ada is translated by the analyzer. [CHAB89]

Extreme FP1 Measures

The FP1 variable measures the weight of types through a communication. Procedure 2521 (open within `adl_database` package), as noted above, has a very complex data structure that is passed into it. Therefore, it should not be strange that this procedure be marked as an extreme given the complex parameter and the fact that this procedure is also an extreme on the FAcc metric. It is very interesting to note that this procedure generates the maximum INFO metric over the entire system, and INFO coupled with extreme interface complexity measures serves to reason out the unusually high (but still moderate) correlations of INFO vs FAcc and FP1 within figure 25.

Another procedure of interest, 1937 (`convert` within `adl_database` package) is extreme in FP1, but also in FAccN. Intuitively this means that there is a great deal of communication into this procedure, and that this communication involves parameters of great complexity. However, it should be noted that because of the additive property of these metrics, the large value of FAccN could have contributed vastly. Yet, on inspection, there are two parameters into this procedure both of great complexity: `actual_db_type` and `db_type` (database types).

Procedure 2333, deferred from above, is a function within the database manager (is_open within adl_database package). This procedure is an extreme for both FAccN and FP1. As above, a variable of high complexity (db_type) is passed into the procedure.

Extreme FP2 Measures

The interface complexity metric FP2 measures the modification complexity of all the parameters passed to each procedure. Two procedures are of special interest here, 4961 and 4958. These procedures (set_scope_kind and set_name_kind, respectively, in parse_stack package) have extreme values, but with the added information that the procedure 2754 (parser) is responsible for the communication. All the other extreme values were measured over an entire procedure, here, however, the communication from procedure 2754 (parser) to these routines is highlighted. There is a great deal of modification that goes into the parameters for these procedures within the parser.

Finally, procedure 6651 is addressed. This procedure (string_of within text_utilities, renamed string_utilities) is present in all four measures. This is a utility procedure with the fourth highest INFO of the system. However, it should be stressed, the procedure is a utility so it is widely called. This brings up a point about information flow and interface measures. Often the procedures with the greatest values are utilities, and a large fan in is the nature of utilities.

Reverse Examination

The subjective examination demonstrated many relationships between the eight interface complexity measures and the established metrics. For example, in the Acc case, a

high Acc corresponded to high established code metrics. However, the question arises: Does a high code metric imply a high interface measure? Figure 32 addresses this question.

Figure 32 enumerates the correlations among the established code metrics and the interface complexity measures. These correlations are taken over all procedures that has at least one established metric within the top 1%. In this way, this table shows the reverse examination of the previous section.

An interesting point about figure 32 is the correspondence among AccN and the established metrics: LOC, N, V, E, CC, and WOOD. This relationship was pointed out by the subjective evaluation, and figure 32 just gives credence to this relationship. The moderate to high correlations among Acc, P1, and P2 with the established metrics are also reflected within the subjective evaluation study.

Many of the relationships within this table do not carry the weight of association that is expected. This lack of correlation is reasonable considering these measures are taken over 275 different structured procedures.

	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
LOC	0.524	0.446	0.555	0.389	-0.036	-0.004	-0.027	-0.037
N	0.924	0.671	0.940	0.583	-0.024	0.007	-0.005	-0.023
V	0.946	0.684	0.948	0.631	-0.021	0.005	-0.007	-0.019
E	0.933	0.679	0.946	0.640	-0.036	-0.030	-0.008	-0.021
CC	0.586	0.653	0.664	0.551	-0.068	-0.064	0.001	-0.040
WOOD	0.937	0.676	0.932	0.653	-0.019	-0.008	0.006	-0.011
INFO	0.047	0.017	0.044	0.001	0.491	0.402	0.718	0.317

Figure 32: Correlations Among Extreme Established Metric Procedures and Interface Complexity Metrics

Conclusions

There are three stages to the analysis of this research. The first stage shows that by changing one predefined weight the interface complexity metric values still correlate highly. This was expected, and the small experiment proved it.

The second stage of analysis is, by far, the most important. Correlations among four functions of interface complexity metrics are examined. Each of the functions show different characteristics:

- F1₊ This function shows poor to moderate correlation with established metrics. This function measures aspects found in both code and hybrid metrics.
- F1_{*} While this function does not correlate to code metrics, it does correlate well to INFO.
- F2₊ This functions correlates to established metrics in much the same way as F1₊ -- yet, the correlations are higher.
- F2_{*} For all practical purposes this function and F1_{*} measure and correlate the same.

Therefore, looking at these four functions, F1₊ measures different aspects from the rest. These functions are also very dependent upon the weights a software engineer might set. So accepting one of these functions as axiomatic without proper verification and validation would be in error.

The final stage of this analysis is a subjective inspection of the code provided by SPS. This stage is necessary to lend credibility to the interface complexity metrics. This stage shows that indeed each interface complexity metric (AccN, Acc, P1, P2, FAccN, FAcc, FP1 and FP2) measures what it is designed to measure. These metrics also point out Ada specific programming styles. This aspect is seen in procedures: 6661 (generics), 3932 (definitions within a package), and 2733 (definitions within a package).

In short, the set of interface complexity metrics was calculated over a large Ada system. These metrics proved to be valuable for the given predefined weights, and functions of these metrics were also meaningful.

Chapter 5: Conclusions

Research Conclusions

In conclusion, this research set out with the following goals:

- 1 Define Interface Complexities
- 2 Specify Interface Complexity Metrics
- 3 Implement Interface Complexity Metric Calculations
- 4 Produce Metrics from the Data Provided by SPS
- 5 Verify Interface Complexity Metrics

This research attained all five goals.

The first goal is to define interface complexities. It was hypothesized that between any two communicating modules a complexity exists. This point is not debatable; however, the concept of interface complexity used within this research is debatable. Interface complexity, as defined by this research, is a function of the communication environment and the parameters. Established metrics, to date, do not address interfaces by examining both of these aspects. This research investigated both aspects.

The second goal of this research is to define a set of interface complexity measures. These measures are presented within chapter 3. The interface complexity metric is defined as a 2-tuple: (communication environment, parameter complexity). The communication environment is a measure of the environment at all points of dataflow. While the parameter complexity is measured in two ways: (1) in terms of the type complexity, and (2) in terms of the modification complexity.

The third goal of this research is to implement the process of generating the interface complexity metrics. This goal is discussed within chapter 2. The complexity manager was developed and incorporated within the analyzer. This manager calculates all the interface complexity metrics. However, the grouping and compaction of the data, as discussed within chapter 4, is external to the analyzer. Also, the calculation of the type complexities is handled within Phase 1 of the analyzer.

The fourth goal is to produce metrics from the data provided by SPS. This goal took many man-hours of work, not to mention computer-hours.

The final goal was to verify the interface complexity metrics. This was done using a three stage analysis. The first stage demonstrated the relationships involved in changing weights. The second stage correlated and examined functions of the interface complexity metrics. The final phase was a subjective examination of the code provided by SPS. This analysis, discussed within chapter 4, showed several positive results, including:

- The interface complexity metrics can be, and are, generated.
- The interface complexity metrics are tunable for specific needs.

- The interface complexity metrics measure meaningful aspects of source code.
- The interface complexity metrics can help in software quality.

Weaknesses

The weaknesses of this research include:

- Only one system was analyzed.
- Lack of any error data limits the meaning of the functions.
- Code examination was subjective.

Future Work

The future work for these metrics would include:

- Addressing the weaknesses of the research, i.e. attain error data for predictor equations, analyze more than one system (but from the same environment), and interaction with programmers on subjective examination.
- Compare interface metrics for different languages, different Environments, different problem domains, and different design methodologies.
- Develop a methodology for deriving optimum weights for different languages, environments, problem domains, and design methodologies.

BIBLIOGRAPHY

- [BAKA80] Baker, A. L., and Zweben, S. H., "A Comparison of Measures of Control Flow Complexity", IEEE Transactions on Software Engineering, Vol. SE-6, pp. 506-512, 1980.
- [BASV80] Basili, V. R., "Qualitative Software Complexity Models: A Summary", Tutorial on Models and Metrics for Software Management and Engineering, IEEE Computer Society Press, Los Alamitos, California, 1980.
- [BASV83] Basili, V. B., and Hutchens, D. H., "An Empirical Study of a Syntactic Complexity Family", IEEE Transactions on Software Engineering, SE-9, Nov. 1983.
- [BASV84] Basili, V. R., and Perricone, B. T., "Software Errors and Complexity: An Empirical Investigation", Communications of the ACM, Vol 27, No. 1, pp. 42-52, January 1984.
- [BASF83a] Bastaini, F. B., "An Approach to Measuring Program Complexity", Proceedings of COMPSAC '83, Chicago, IL, pp. 90-94, 1983.
- [BASF83b] Bastaini, F. B., and Iyengar, S. S., "The Effect of Data Structures on the Logical Complexity of Programs", Technical Report, Department of Computer Science, Louisiana State University, 1983.
- [BECL83] Beck, L. L., and Perkins, T. E., "A Survey of Software Engineering Practice: Tools, Methods, and Results", IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, pp. 541-561, September 1983.
- [BELL80] Belady, L. A., "On Software Complexity", IEEE Workshop on Quantitative Software Models, pp. 90-94, 1980.
- [BERE80] Berlinger, E., "An Information Theory Based Complexity Measure", Proceedings of the 1980 National Computer Conference, pp. 773,779.
- [BOOG83] Booch, G., Software Engineering with ADA, Menlo Park, CA, 1983.
- [BORA72] Borodin, A., "Computational Complexity and the Existence of Complexity Gaps", Journal of the ACM, Vol. 19, No. 1, pp. 158-174, January 1972.

- [CATS84] Cater, S. C., Iyengar, S. S., and Fuller, J., "Computation of Logical Effort in High Level Languages", Computer Languages, Vol. 9, No. 3-4, pp. 133-148, December 1984.
- [CHAB89] Chappell, Bryan T. S., "Definition and Validation of Software Complexity Metrics for Ada," Masters Thesis, Virginia Tech, 1989.
- [CHAN79] Chapin, N., "A Measure of Software Complexity", Proceedings of the 1979 National Computer Conference, New York, pp. 995-1002, 1979.
- [CHAB83] Chaudhary, B. D., and Sahasrabudde, H. V., "Two Dimensions of Program Complexity", International Journal of Man-Machine Studies, Vol. 18, No. 5, pp. 505-511, 1983.
- [CHET78] Chen, E. T., "Program Complexity and Programmer Productivity", IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, pp. 187-194, May 1978.
- [CONS86] Conte, S. D., Dunsmore, H. E., and Shen, V. Y., Software Engineering Metrics and Models, The Benjamin Cummings Publishing Company, Inc., 1986.
- [COUP85] Courtois, P. -J., "On Time and Space Decomposition of Complex Structures", Communications of the ACM, Vol. 28, No. 6, June 1985.
- [CRAS85] Crawford, S. G., McIntosh, A. A., and Pregibon, D., "An Analysis of Static Metrics and Faults in C Software", The Journal of Systems and Software, Vol. 5, No. 1, February 1985.
- [CURB79a] Curtis, B., Sheppard, S., Milliaman, P. M., Borst, M. A., and Love, T., "Measuring the Psychological Complexity of Software Maintenance Tasks with Halstead and McCabe Metrics", IEEE Transactions on Software Engineering, Vol. SE-5, pp. 95-104, 1979.
- [CURB79b] Curtis, B., Sheppard, S., and Milliman, P., "Third Time Charm: Stronger Replication of the Ability of Software Complexity Metrics to Predict Programmer Performance", Proceedings of the 4th International Conference on Software Engineering, pp. 356-360, September 1979.
- [CURB79c] Curtis, B., "In Search of Software Complexity", Workshop on Quantitative Software Models for Reliability, Complexity, and Cost, pp. 95-106, October 1979.
- [DAVJ84] Davis, J. S., "Chunks: A Basis for Complexity Measurement", Information Process and Management, Vol. 20, No. 1-2, pp. 119-127, 1984.
- [DAVJ88] Davis, J. S., LeBlanc, R., "A Study of the Applicability of Complexity Measures," IEEE Transactions on Software Engineering, Vol. 14, No. 9, pp. 1366-1372, Sept. 1988.

- [EJIL84] Ejiogu, L. O., "A Simple Measure of Software Complexity", Computer World, Vol. 12, No. 14, pp. ID/10-ID/16, April 2, 1984.
- [EJIL85] Ejiogu, L. O., "A Simple Measure of Software Complexity", ACM SIG-Plan Notices, Vol. 20, No. 3, pp. 16-31, March 1985.
- [EVAW83] Evangelist, W. M., "Software Complexity Metric Sensitivity to Program Structuring Rules", J. Syst. Softw. 3, pp. 231-243, September 1983.
- [GLIT77] Glib, T., Software Metrics, Winthrop Publishers, Cambridge, MA, 1977.
- [GORG83] Gorsline, G. W., and Fainter, R., "Program Complexity: A Literature Review and Synthesis", Computer Science Department, Virginia Tech, Blacksburg, 24061, 1983.
- [GOUJ84] Gouguen, J. A., "Parameterized Programming", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pp. 474-477, September 1984.
- [HALM77] Halstead, M. H., Elements of Software Science, Elsevier, New York, N.Y., 1977.
- [HANW78] Hansen, W. J., "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)", ACM SIGPlan Notices, Vol. 13, No. 3, pp. 29-33, March 1978.
- [HARW81a] Harrison, W., A Hybrid Metric to Measure Software Complexity, Masters Thesis, University of Missouri-Rolla, Department of Computer Science, 1981.
- [HENS81a] Henry, S. M., Kafura, D., and Harris, K., "On the Relationships Among Three Software Metrics", Proceedings fo ACM SIGMETRICS, pp. 81-89, 1981.
- [HENS81b] Henry, S. M., Kafura, D., "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, Vol. Se-7, September 1981.
- [HENS84] Henry, S. M., and Kafura, D., "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics", Software -- Practice and Experience, Vol. 14, No. 6, pp. 561-573, June 1984.
- [HENS89] Henry, S. M., Goff, R., "Complexity Measurement of a Graphical Programming Language," to appear in Software -- Practice and Experience, 1989.
- [HENS90] Henry, S. M., Selig, C. "Design Metrics Which Predict Source Code Quality," to appear in IEEE Software, 1990.
- [INCD85] Ince, D. C., "The Influence of System Design Complexity Research on the Design of Module Interconnection Languages", ACM SIGPLAN Notices, Vol. 20, pp. 36-43, October 1985.

- [IYES85] Iyengar, S. S., Bastaini, F. B., and Fuller, J. W., "An Experimental Study of the Logical Complexity of Data Structures", Proceedings of the Symposium on Empirical Foundations of Information Science and Software, pp. 225-238, November 1985.
- [JENR79] Jensen, R. W., and Tonies, C. C., Software Engineering, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.
- [KAJD84] Kafura, D., and Henry, S., "A Viewpoint on Software Quality Metrics: Criteria, Use, and Integration", The Journal of Systems and Software, pp. 121-131, June, 1984.
- [KAJD87] Kafura, D., and Geereddy, R. R., "The Use of Software Complexity Metrics in Software Maintenance", Transactions on Software Engineering, Vol. SE-13, No. 3, pp. 335-343, March 1987.
- [KEAJ86] Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gary, M. A., and Adler, M. A., "Software Complexity Measurement", Communications of the ACM, Vol. 29, No. 11, pp 1044-1050, November 1986.
- [LEWK88] Lew, K. S., Dillon, T. S., and Forward, K. E., "Software Complexity and Its Impact on Software Reliability", IEEE Transactions on Software Engineering, Vol. SE-14, No. 11, pp. 1645-1655, Nov. 1988.
- [LIHF87] Li, H. F., Cheung, W. K., "An Empirical Study of Software Metrics," IEEE Transactions on Software Engineering, Vol. SE-13, No. 6, pp. 697-708, June 1987.
- [MCAD88] McAuliffe, D., "Measuring Program Complexity," Computer, Vol. 21, No. 6, pp. 97-98, June 1988.
- [MCCT76] McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320, 1976.
- [MCCT85] McCabe, T. J., and Schulmeyer, G. G., "System Testing Aided by Structured Analysis: A Practical Experience", IEEE Transactions on Software Engineering, Vol. SE-11, No. 9, pp. 917-921, September 1985.
- [MCCC78] McClure, C. L., "A Model for Program Complexity Analysis", Proceedings of the 3rd Conference on Software Engineering, pp. 149-157, 1978.
- [METD85] Metayer, D. L., "Mechanical Analysis of Program Complexity", ACM SIGPlan Notices, Vol. 20, No. 7, pp. 69-73, July 1985.
- [MILS73] Mills, H. S., "The Compleixty of Programs", Program Test Methods, Hetzel, W. C., ed., Prentice-Hall, Inc. Englewood Cliffs, NJ, pp. 225-238, 1977.

- [MYEG77] Myers, G. J., "An Extension to the Cyclomatic Measure of Program Complexity", ACM SIGPlan Notices, Vol. 12, No. 10, pp. 61-64, October 1977.
- [PARD71] Parnas, D. L., "Information Distribution Aspects of Design Methodology", Proceedings IFIP Congress, pp. 339-344, August 1971.
- [PARD85] Parnas, D. L., Clements, P. C, and Weiss, D. M., "The Modular Structure of Complex Systems", IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, March 1985.
- [PIWP80] Piwowarski, P., "A Nesting Level Complexity Measure", ACM SIGPLAN Notices, Vol. 17, pp. 44-50, 1980.
- [PRAR84] Prather, R., "An Axiomatic Theory of Software Complexity Measure", The Computer Journal, Vol. 27, No. 4, pp. 340-347, 1984.
- [RABM77] Rabin, M. O., "Complexity of Computations", Communications of the ACM, Vol. 20, pp. 625-633, 1977.
- [SCHB80] Schneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishing, Inc., Cambridge, MA, 1980.
- [SCHB86] Schneiderman, B., "Empirical Studies of Programmers: The Territory, Paths, and Destinations", Empirical Studies of Programmers, Soloway, E., and Iyengar, S. Eds., Ablex Publishing Corp., Norwood, New Jersey, pp. 1-12, June 1986.
- [SIMM73] Sime, M., Green, T., and Guest, D., "Psychological Evaluation of Two Conditional Constructs Used in Computer Languages", International Journal of Man-Machine Studies, Vol. 5, No. 1, pp. 105-113, 1973.
- [SIMM77] Sime, M., Green, T., and Guest, D., "Scope Marking in Computer Conditionals -- A Psychological Evaluation", International Journal of Man-Machine Studies, Vol. 9, No. 1, pp. 107-118, 1977.
- [SIMH62] Simon, H. A., "The Architecture of Complexity", Proceedings of the American Philosophical Society, Vol. 106, No. 6, pp. 467-482, December 1962.
- [SHEB81] Sheil, B. A., "The Psychological Study of Programming", ACM Computing Surveys, pp. 308-320, March 1981.
- [STEW74] Stevens, W. P., Myers, G. L., and Constantine, L. L., "Structured Design", IBM Systems Journal, Vol. 13, No. 2, 1974.
- [WAKS88] Wake, S., Henry, S., "A Model Based on Software Quality Factors Which Predicts Maintainability", Proceedings: Conference on Software Maintenance - 1988, pp. 382-387, Oct. 1988.

- [WALJ85] Wallace, J. C., and Nance, R. E., "The Control and Transformation Metrics: A Basis for Measuring Model Complexity", Technical Report TR-85-15, Computer Science Department, Virginia Tech, Blacksburg, 24061, March 1985.
- [WEIG71] Weinberg, G. W., The Psychology of Computer Programming, Van Nostrand Reinhold Co., New York, NY, 1971.
- [WEIL74] Weissman, L., "Psychological Complexity of Computer Programs: An Experimental Methodology", ACM SIGPlan Notices, Vol. 9, No. 6, pp. 25-30, June 1974.
- [WOOS79] Woodfield, S. N., "An Experiment on Unit Increase in Problem Complexity", IEEE Transactions on Software Engineering, Vol. SE-5, pp. 76-79, March 1979.
- [WOOS82] Woodfield, S. N., and Shen, V. Y., Dunsmore, H. E. "A Study of Several Metrics for Programming Effort", The Journal of Systems and Software, Vol. 2, No. 2, June 1982, pp. 97-103.
- [WOOM79] Woodward, M. R., Hennell, M. A., and Hedley, D., "A Measure of Control Flow Complexity in Program Text", IEEE Transactions on Software Engineering, Vol. SE-5, No. 1, January 1979.
- [YOUR89] Yourdon, Editor. "Software Metrics: You Can't Control What You Can't Measure", American Programmer, Vol. 2, No. 2, pp. 3-11, Feb. 1989.
- [ZOLJ77] Zolnowski, J. M., and Simmons, D. B., "Measuring Program Complexity", Proceedings of the 1977 Fall COMPCON, IEEE, pp. 336-340, 1977.

Appendix A: Top 1% Interface Complexity Measures

This appendix contains the interface complexity measures: AccN, Acc, P1, P2, FAccN, FAcc, FP1, and FP2, for the procedures with at least one of these measures within the top 1%. This is added to show the diversity of the interface complexity metrics.

Top 1% Interface Complexity Metric Cut-Offs								
Proc	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
Numb.	46	517	253	148	40	340	273	238

58	3	3	14	3	3	21	30	270
70	2	2	12	0	3	13	41	303
152	2	2	12	0	7	17	121	257
175	6	13	8	1	20	368	144	119
222	3	3	14	3	3	21	30	294
234	2	2	13	0	7	17	133	430
304	3	3	14	3	3	21	30	278
316	2	2	12	0	3	13	41	311
585	6	13	8	1	20	368	144	119
644	2	2	13	0	7	17	133	364
726	2	2	12	0	6	16	101	251
878	3	3	14	3	3	21	30	246
890	2	2	13	0	5	15	89	334
913	6	13	9	1	50	996	356	269
919	0	0	0	0	55	694	0	0
954	1	1	3	0	8	465	49	11
964	2	2	14	0	43	243	933	714
972	2	2	14	0	22	32	465	537
992	0	0	0	0	380	8552	0	0
1034	67	346	495	55	2	22	108	5
1042	28	609	203	94	3	3	35	3
1106	61	308	394	90	2	22	118	5
1116	28	650	219	101	1	1	14	1
1147	26	70	121	212	4	21	33	1
1172	11	151	67	48	28	422	140	26
1174	220	1166	2075	251	2	1175	0	0
1180	0	0	0	0	54	54	0	0
1196	28	1693	371	101	13	145	1	1197
1202	25	752	334	148	10	32	239	31
1266	40	206	328	68	2	22	98	5
1275	28	580	209	101	3	8	40	3
1309	40	206	328	68	2	22	98	5
1318	28	580	209	101	3	8	40	3
1352	40	206	326	68	2	22	98	5
1361	28	580	209	101	2	2	26	2
1395	51	258	373	93	2	22	118	5
1405	28	650	219	101	2	2	26	2
1443	40	204	298	68	2	22	92	5
1452	28	553	195	94	3	8	40	3
1486	40	204	300	68	2	22	92	5
1495	28	553	195	94	4	22	53	25
1557	22	67	110	170	3	31	15	3
1577	11	160	71	52	31	474	155	33
1579	143	770	1475	159	2	22	1580	0

Top 1% Interface Complexity Metric Cut-Offs								
Proc	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
Numb.	46	517	253	148	40	340	273	238

1593	28	1252	311	101	6	6	85	11
1599	25	564	277	151	12	24	237	24
1649	40	206	330	68	2	22	98	5
1658	28	580	209	101	3	8	40	3
1692	40	206	330	68	2	22	98	5
1701	28	580	209	101	3	8	40	3
1735	40	206	328	68	2	22	98	5
1744	28	580	209	101	2	2	26	2
1778	40	204	300	68	2	22	92	5
1787	28	553	195	94	3	8	40	3
1819	42	208	254	84	2	22	96	5
1829	28	567	197	94	5	20	68	5
1917	28	567	197	94	2	19	23	2
1935	121	454	1232	459	1936	0	0	0
1937	0	0	0	0	1085	1085	5667	0
1939	605	3380	3551	1699	1940	0	0	0
1978	312	315	988	8769	3	1979	0	0
1999	105	552	905	107	2	22	190	2000
2013	28	902	255	101	3	3	35	3
2044	22	66	106	142	13	13	316	6
2064	125	663	1151	141	2	22	2	2065
2081	28	1084	284	101	3	3	35	3
2156	0	0	0	0	37	370	592	740
2157	0	0	0	0	37	37	592	222
2166	269	1601	1306	459	2167	0	0	0
2169	20	63	92	68	98	98	1756	0
2179	58	297	396	46	2	22	90	5
2192	28	546	194	94	2	2	28	2
2231	77	402	656	92	2	22	150	5
2245	28	762	235	101	4	41	41	4
2290	95	469	579	193	2	22	204	2291
2298	0	0	0	0	46	46	0	0
2309	28	951	262	101	2	19	23	2
2333	0	0	0	0	727	2498	5041	3229
2403	27	72	130	200	6	6	72	0
2424	150	796	1273	195	2	22	2425	0
2431	0	0	0	0	42	42	0	0
2443	25	564	277	151	0	0	0	0
2447	28	1259	315	101	0	0	0	0
2520	2	6	11	0	52	329	156	0
2521	166	521	774	93	2522	0	0	0
2556	85	444	691	73	2	22	144	5
2574	28	735	221	94	0	0	0	0
2608	96	497	744	98	2	22	166	5

Top 1% Interface Complexity Metric Cut-Offs								
Proc	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
Numb.	46	517	253	148	40	340	273	238

2629	28	812	232	94	2	2	28	2
2663	47	244	345	21	2	22	68	5
2700	40	204	300	68	2	22	92	5
2716	28	553	195	94	4	15	54	4
2733	0	0	0	0	603	6880	0	0
2735	38	194	86	374	6	78	0	0
2736	22	110	21	0	102	1346	1133	2737
2749	0	0	0	0	567	2805	0	0
2754	1634	8868	6887	2755	10	69	328	6713
2761	0	0	0	0	456	3031	0	0
3072	2	2	12	0	6	16	101	243
3154	2	2	12	0	6	16	101	319
3236	2	2	12	0	5	15	81	253
3259	6	13	10	1	36	746	290	199
3265	0	0	0	0	41	497	0	0
3310	2	2	15	0	29	172	682	504
3318	2	2	15	0	15	25	339	387
3342	1	1	0	0	79	412	0	0
3395	3	3	14	3	3	21	30	254
3407	2	2	12	0	6	16	101	359
3429	5	18	17	0	39	155	692	554
3432	33	285	99	175	1	5	22	21
3450	23	653	28	0	1	5	0	0
3453	22	50	64	288	1	1	12	0
3455	0	0	0	0	152	918	0	0
3507	10	14	25	0	212	713	664	21
3620	6	13	8	1	28	528	200	159
3626	0	0	0	0	33	364	0	0
3671	2	2	13	0	21	111	448	340
3679	2	2	13	0	11	21	221	282
3801	6	13	9	1	32	624	230	179
3807	0	0	0	0	37	424	0	0
3852	2	2	14	0	25	135	537	408
3860	2	2	14	0	13	23	267	338
3915	383	383	2722	0	1	1	3	3916
3931	4	43	17	21	32	348	54	61
3932	0	0	0	0	829	11214	0	0
3933	0	0	0	0	78	1265	0	0
3934	73	788	338	35	5	15	107	3935
3937	52	543	253	36	2	6	47	0
3938	115	1685	453	54	1	1	35	3939
3940	107	813	297	100	1	6	8	3941
3943	64	306	165	116	3	33	24	14
3945	59	751	123	87	2	42	22	5

Top 1% Interface Complexity Metric Cut-Offs								
Proc	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
Numb.	46	517	253	148	40	340	273	238

3946	31	544	99	35	2	35	30	5
3947	104	1300	282	31	1	1	38	0
3948	66	710	209	47	5	48	78	3
3978	0	0	0	0	83	478	0	0
3983	27	129	50	432	99	1037	3984	0
3997	29	133	51	432	20	397	176	46
4001	70	222	437	23	4	30	29	3
4002	65	237	376	28	3	24	21	3
4099	81	520	332	224	0	0	0	0
4108	34	216	188	198	0	0	0	0
4136	6	13	8	1	24	444	172	139
4187	2	2	13	0	17	87	360	272
4195	2	2	13	0	9	19	177	382
4207	7	7	0	0	49	735	0	0
4210	8	23	30	28	66	908	594	303
4211	3	18	18	15	3	13	86	365
4212	64	281	146	142	2	4	8	4
4218	62	1352	329	747	3	14	79	78
4235	6	13	8	1	28	542	200	159
4241	0	0	0	0	33	364	0	0
4282	3	3	14	3	3	21	30	262
4286	2	2	13	0	21	111	448	340
4294	2	2	13	0	11	21	221	502
4309	41	1035	68	0	0	0	0	0
4313	0	0	0	0	83	112	0	0
4315	14	37	44	123	76	76	152	0
4319	0	0	0	0	36	560	0	0
4328	106	2142	374	33	0	0	0	0
4329	59	1931	231	28	0	0	0	0
4343	53	753	211	10	2	7	32	9
4375	0	0	0	0	22	1631	0	0
4376	0	0	0	0	115	217	0	0
4377	357	3544	428	2842	1	4378	41	140
4612	2	2	11	0	2	12	19	245
4682	3	3	14	3	3	21	30	286
4694	2	2	11	0	2	12	19	293
4706	0	0	0	0	53	226	0	0
4709	38	194	86	374	1	1	3	0
4710	19	19	50	176	1	1	0	0
4750	8	14	4	0	28	55	78	280
4752	24	162	106	475	2	15	12	5
4826	2	2	12	0	5	15	81	269
4849	6	13	11	1	34	724	276	189
4855	0	0	0	0	39	466	0	0

Top 1% Interface Complexity Metric Cut-Offs								
Proc	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
Numb.	46	517	253	148	40	340	273	238

4900	2	2	16	0	27	159	635	468
4908	2	2	16	0	14	24	317	551
4922	9	25	39	9	8	40	359	10466
4924	9	25	39	9	3	15	141	4650
4948	0	0	0	0	80	670	0	0
4952	2	14	8	0	188	1707	1593	184
4953	2	14	8	0	88	690	744	123
4958	1	1	0	0	7	35	104	10225
4960	1	1	0	0	3	15	24	5029
4961	1	1	0	0	10	42	110	12170
4977	1	1	3	0	27	178	297	9
4990	0	0	0	0	176	322	0	0
5006	6	13	7	1	24	416	148	133
5057	2	2	12	0	17	80	327	256
5170	6	13	7	1	26	458	160	143
5221	2	2	12	0	19	91	367	288
5229	2	2	12	0	10	20	181	261
5241	0	0	0	0	99	721	0	0
5248	6	18	12	6	6	6	96	314
5377	1	1	3	0	31	137	496	66
5379	3	37	0	0	9	9	171	240
5392	1	1	0	0	49	99	63	0
5393	1	1	16	3	43	223	129	0
5407	34	887	119	3	0	0	0	0
5410	31	182	141	184	1	44	4	0
5515	47	368	234	416	1	44	4	0
5585	46	368	224	377	1	44	4	0
5620	47	385	234	416	1	44	4	0
5655	54	639	236	335	1	44	4	0
5760	77	1123	338	1454	1	44	4	5761
5765	6	24	20	0	7	107	70	484
5795	53	581	216	277	1	44	4	0
5830	53	581	216	277	1	44	4	5831
5865	36	302	188	469	1	44	4	0
5900	46	192	203	382	1	44	4	0
5935	97	1226	331	400	1	44	8	5936
6051	130	1077	337	457	4	20	6052	15
6055	61	134	263	63	1	23	6	0
6062	52	462	262	27	2	26	12	10
6065	91	418	306	138	3	27	18	20
6072	49	454	223	12	1	10	6	0
6145	2	2	12	0	6	16	101	265
6239	0	0	0	0	61	1190	0	0
6363	1	1	30	0	13	13	382	246

Top 1% Interface Complexity Metric Cut-Offs								
Proc	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
Numb.	46	517	253	148	40	340	273	238

6367	1	1	3	0	44	68	132	10
6454	19	338	214	607	3	5	63	66
6520	47	710	244	15	39	208	6521	4
6523	121	648	293	18	1	3	0	0
6526	10	38	85	0	10	50	262	245
6527	45	508	154	74	2	10	92	6528
6547	0	0	0	0	52	52	520	0
6555	0	0	0	0	124	1149	0	0
6560	0	0	0	0	27	357	74	0
6569	0	0	0	0	52	52	260	0
6616	0	0	0	0	35	178	83	504
6618	0	0	0	0	26	54	117	862
6620	0	0	0	0	113	1323	345	1452
6644	5	45	13	2	8	8	38	383
6646	0	0	0	0	63	439	501	1772
6651	0	0	0	0	819	8474	5839	1159
6654	8	44	50	164	0	0	0	0
6656	5	29	30	53	254	345	1549	84
6661	0	0	0	0	841	4473	1149	211
6662	0	0	0	0	58	365	0	0
6670	0	0	0	0	45	45	590	42
6675	7	7	60	25	99	136	1569	294
6681	53	342	110	33	45	350	6682	111
6690	11	66	57	3	26	57	300	0
6694	1	1	3	0	73	378	407	90
6702	0	0	0	0	91	1687	0	0
6703	9	45	36	12	27	376	302	80
6708	0	0	0	0	30	249	292	48
6709	1	1	13	0	44	585	574	27
6715	2	2	10	0	50	534	125	113
6717	7	107	30	0	31	155	310	21
6719	9	115	10	0	56	428	584	38
6723	7	108	30	10	43	149	409	27
6726	1	1	3	0	185	570	552	0
6732	1	1	0	0	114	242	174	0
6735	1	1	3	0	43	356	346	86
6767	0	0	0	0	81	1625	0	0
6769	7	19	38	0	79	395	1486	1515
6771	15	63	47	0	8	40	292	330
6776	52	132	254	21	1	5	74	70
6779	55	177	217	9	1	5	39	21
6780	70	270	379	162	1	5	76	6781
6782	50	144	192	9	2	10	66	42
6785	49	104	252	13	1	5	83	91

Top 1% Interface Complexity Metric Cut-Offs								
Proc	AccN	Acc	P1	P2	FAccN	FAcc	FP1	FP2
Numb.	46	517	253	148	40	340	273	238

6787	61	218	387	22	2	10	128	6788
6789	58	165	323	22	1	5	68	4
6794	0	0	0	0	50	228	0	0
6805	10	10	3	0	30	99	341	71
6829	1125	1125	5235	783	1	6830	6	19
6831	0	0	0	0	62	250	186	71
6839	50	818	262	117	1	1	3	0
6840	128	907	575	51	1	3	3	0
6845	71	428	303	6	3	5	9	0
6860	46	734	153	7	1	1	3	0
6864	6	30	12	3	57	149	111	14
6921	0	0	0	0	170	248	2306	496
6928	1	1	0	0	42	137	0	0

Appendix B: Metrics for Top 1% Interface Complexity Procedures

This appendix contains all the established metrics for each procedure contained within Appendix A.

Also, the following table lists the mean, standard deviation, minimum and maximum values over all units of analysis of the system. This is included for comparison purposes.

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

Metrics for Top 1% Interface Complexity							
Proc	LOC	N	V	E	CC	WOOD	INFO

58	10	67	311	5170	1	3446	1243671040
70	8	51	220	2976	1	3306	48412800
152	8	51	220	2976	1	5429	282268800
175	13	46	208	3246	3	6477	9414613
222	10	67	311	5170	1	3446	253210240
234	8	51	220	2976	1	5429	5445000
304	10	67	311	5170	1	3446	853776000
316	8	51	220	2976	1	3306	32772608
585	13	46	208	3246	3	6477	8613748
644	8	51	220	2976	1	5429	153475200
726	8	51	220	2976	1	5168	276595200
878	10	67	311	5170	1	3446	2598544000
890	8	51	220	2976	1	4776	107399168
913	13	46	208	3246	3	6492	176625748
919	19	69	304	2385	1	4769	316436659
954	6	29	113	1275	1	2400	697250400
964	8	51	224	3327	1	6653	42320000
972	8	51	220	2976	1	5950	214659200
992	255	1072	7666	99136	1	198300	16730550000
1034	185	769	4681	459321	76	306214	4736000
1042	67	372	2310	86034	8	95593	32830000
1106	185	693	4248	365197	66	243464	4736000
1116	68	372	2310	86034	8	86034	4318272
1147	101	646	4264	201978	5	224420	301584384
1172	30	172	966	25794	5	50992	175692000
1174	677	2605	17231	3192102	287	2128068	17331200
1180	2	5	12	23	1	23	2
1196	97	505	3370	114004	8	182969	237271312
1202	80	428	2818	87595	6	164936	349448000
1266	147	526	3168	237582	53	158388	1843968
1275	66	363	2240	84383	8	93759	26195400
1309	147	526	3168	237582	53	158388	1843968
1318	66	363	2240	84383	8	93759	27542856
1352	147	526	3168	237582	53	158388	1843968
1361	68	363	2240	84383	8	56256	12514788
1395	181	635	3838	333059	64	222039	2270464
1405	68	371	2296	86712	8	57808	13647872
1443	147	526	3168	237582	53	158388	1843968
1452	67	363	2240	84383	8	93759	29185200
1486	147	524	3132	245404	53	163603	1843968
1495	66	362	2226	85086	8	119750	54895104
1557	77	502	3251	136631	5	151813	51774800
1577	30	172	966	25794	5	50992	64123320
1579	445	1721	11001	1689060	188	1126040	11392000
1593	83	443	2869	100554	8	174624	45450800
1599	66	366	2333	74288	6	142779	213840000
1649	147	526	3168	237582	53	158388	1843968

Metrics for Top 1% Interface Complexity							
Proc	LOC	N	V	E	CC	WOOD	INFO

1658	66	363	2240	84383	8	93759	15206400
1692	147	526	3168	237582	53	158388	1843968
1701	66	363	2240	84383	8	93759	15846600
1735	147	526	3168	237582	53	158388	1843968
1744	68	363	2240	84383	8	56256	6120000
1778	147	526	3168	237582	53	158388	1843968
1787	67	363	2240	84383	8	93759	17426700
1819	149	513	3078	231660	49	154440	1869056
1829	66	362	2226	85086	8	136557	23760000
1917	65	363	2240	84383	8	56256	3744000
1935	165	1320	9491	324621	39	632353	34171645365
1937	10	26	84	461	1	924	22904461500840
1939	751	4994	39606	4591799	78	4591799	447115360000
1978	207	2056	15716	713908	3	793231	10449572175
1999	315	1246	7808	999171	132	666114	8064000
2013	75	408	2594	92870	8	103188	39421875
2044	73	464	2958	125352	5	83568	28151793
2064	381	1477	9338	1328517	158	885678	9753600
2081	79	425	2724	96978	8	107754	41639004
2156	0	0	0	0	0	0	0
2157	0	0	0	0	0	0	0
2166	204	1727	12417	522165	39	348110	119273169600
2169	49	322	1974	74628	5	49752	5112121
2179	153	647	3911	343166	61	228777	3916800
2192	65	363	2240	84383	8	56256	3146000
2231	247	948	5868	629744	100	419829	6323200
2245	71	390	2451	89408	8	125834	8797184
2290	317	1075	6675	793649	108	529099	3976448
2298	2	5	12	23	1	23	2
2309	75	405	2553	94748	8	63166	4320000
2333	6	20	74	651	1	1304	5077229827416
2403	90	545	3555	151965	5	151965	5490810
2424	423	1622	10451	1449872	163	966581	10828800
2431	2	5	12	23	1	23	2
2443	64	358	2276	72086	6	72086	906304
2447	81	435	2810	98198	8	98198	746496
2520	11	44	196	2355	2	4708	364953600
2521	141	1388	10212	455644	4	912590	55959169202181
2556	249	1013	6250	725967	106	483978	6374400
2574	71	390	2451	89408	8	89408	654336
2608	283	1122	6945	870192	117	580128	7244800
2629	73	398	2509	91748	8	61165	3895353
2663	119	533	3122	245744	50	163829	2677500
2700	147	526	3168	237582	53	158388	1843968
2716	66	363	2240	84383	8	118762	34214400
2733	46	394	2762	25587	1	51191	34434201600
2735	33	313	1793	79863	5	112399	1670625

Metrics for Top 1% Interface Complexity							
Proc	LOC	N	V	E	CC	WOOD	INFO

2736	29	255	1512	22037	22	44074	368360784
2749	50	309	1858	19668	1	27681	737280000
2754	1218	8313	78395	8139773	354	8139773	4862260872
2761	18	118	608	7010	1	9866	11606562
3072	8	51	220	2976	1	5168	276595200
3154	8	51	220	2976	1	5168	189891072
3236	8	51	220	2976	1	4776	254837888
3259	13	46	208	3246	3	6492	59479173
3265	19	69	304	2385	1	4769	90627264
3310	8	51	224	3327	1	6653	11829248
3318	8	51	220	2976	1	5931	116494848
3342	4	14	44	222	1	444	13483584
3395	10	67	311	5170	1	3446	2129264640
3407	8	51	220	2976	1	5168	88817792
3429	12	63	296	4188	2	8231	425115648
3432	69	378	2442	92531	10	92531	655411956
3450	32	227	1261	34166	19	34166	18432
3453	31	200	1135	34354	4	34354	642816
3455	10	143	815	4076	1	8010	174240
3507	12	81	405	6096	2	12193	89303808
3620	13	46	208	3246	3	6492	26620893
3626	19	69	304	2385	1	4769	35349424
3671	8	51	224	3327	1	6651	4147200
3679	8	51	220	2976	1	5848	174620672
3801	13	46	208	3246	3	6492	40681693
3807	19	69	304	2385	1	4769	58187500
3852	8	51	224	3327	1	6653	7250432
3860	8	51	220	2976	1	5905	170755200
3915	445	4653	39343	1644068	1	1644068	17095120
3931	12	78	397	10556	3	21003	6500352
3932	25	120	600	7513	1	15044	1360593938025
3933	190	834	4912	153767	1	305954	53774560
3934	101	690	4845	201893	13	324026	388397621
3937	83	565	3839	153554	10	102369	109386032
3938	108	887	6126	297794	13	297794	105423552
3940	109	813	5709	274041	15	274041	294956725
3943	58	483	3120	125300	6	139222	65661568
3945	47	419	2610	107356	7	71571	44404848
3946	41	288	1771	61104	6	40736	4739600
3947	115	861	6102	383037	39	383037	182574000
3948	68	520	3432	129865	7	182773	282988800
3978	63	350	1990	33120	1	66241	334139967
3983	49	272	1573	56021	15	112047	2642782464
3997	52	279	1634	63542	16	122125	52000000
4001	207	843	6102	186203	15	206892	1789225200
4002	165	722	5125	151832	12	168703	848730960
4099	95	516	3119	140351	35	140351	14897520

Metrics for Top 1% Interface Complexity							
Proc	LOC	N	V	E	CC	WOOD	INFO

4108	66	271	1442	57690	18	57690	511104
4136	13	46	208	3246	3	6489	16453125
4187	8	51	224	3327	1	6643	2508800
4195	8	51	220	2976	1	5719	220668032
4207	65	331	1870	32179	1	58709	2106000
4210	31	179	1040	27143	3	54261	378882496
4211	11	72	363	8872	3	5915	13475
4212	87	537	3395	180822	13	120548	176175
4218	154	876	5870	528424	34	587137	1540000
4235	13	46	208	3246	3	6492	24685492
4241	19	69	304	2385	1	4769	35349424
4282	10	67	311	5170	1	3446	1673400960
4286	8	51	224	3327	1	6651	4718592
4294	8	51	220	2976	1	5848	74908800
4309	29	172	928	22156	9	22156	35525
4313	21	106	514	7938	1	13786	75600
4315	22	134	732	21947	5	14631	8159382
4319	118	518	2989	62982	1	118592	3330432
4328	108	721	4748	264518	24	264518	15925248
4329	74	486	3155	139709	19	139709	4262400
4343	78	410	2554	105050	16	105050	23766912
4375	288	343	2339	27625	1	27625	288
4376	33	185	890	9764	1	6509	232848
4377	330	2081	15673	1379407	72	1379407	11539770
4612	8	51	220	2976	1	1984	116494848
4682	10	67	311	5170	1	3446	518400000
4694	8	51	220	2976	1	1984	18580608
4706	42	257	1388	20286	1	38988	5443200
4709	33	310	1767	80073	5	80073	20625
4710	18	131	692	15146	1	15146	3528
4750	11	48	217	3670	3	7212	101376
4752	40	274	1612	86356	11	57571	256000
4826	8	51	220	2976	1	4776	233280000
4849	13	46	208	3246	3	6492	49432500
4855	19	69	304	2385	1	4769	73064899
4900	8	51	224	3327	1	6653	8323200
4908	8	51	220	2976	1	5921	298852352
4922	18	106	544	9968	2	9968	543312648
4924	18	106	544	9968	2	9968	171532242
4948	62	330	1977	66128	1	132253	35435232
4952	10	40	178	2355	2	4710	1018888360
4953	10	40	178	2355	2	4710	1487912040
4958	7	44	196	2826	1	2826	1258432
4960	4	21	78	684	1	684	509796
4961	7	36	158	2108	1	2343	5899068
4977	12	54	251	4468	2	8936	4915200
4990	17	97	443	7546	1	14918	1184832

Metrics for Top 1% Interface Complexity							
Proc	LOC	N	V	E	CC	WOOD	INFO

5006	13	46	208	3246	3	6489	16453125
5057	8	51	224	3327	1	6643	2163200
5170	13	46	208	3246	3	6491	21099988
5221	8	51	224	3327	1	6648	3035648
5229	8	51	220	2976	1	5796	208080000
5241	370	1539	10792	367409	1	734849	2578752000
5248	11	69	332	6137	2	10657	53900
5377	12	54	251	4468	2	8936	20031168
5379	28	118	579	21579	6	41475	36288
5392	5	17	59	463	1	926	1220180
5393	13	57	261	3649	1	7296	445196752
5407	98	424	2759	122811	31	122811	1786050
5410	122	389	2532	108401	12	108401	40758248
5515	154	569	3895	157967	10	157967	256271400
5585	155	566	3875	164690	10	164690	306409580
5620	156	577	3957	169186	11	169186	305760000
5655	140	540	3710	148930	9	148930	187087040
5760	175	708	4763	284084	13	284084	242020800
5765	13	66	310	9669	3	10743	464373
5795	155	572	3944	162630	10	162630	205701120
5830	155	572	3944	162630	10	162630	213996875
5865	136	433	2818	107224	6	107224	42497416
5900	138	476	3142	121077	5	121077	81395712
5935	170	710	4904	262678	15	262678	165608730
6051	122	928	6557	260785	18	260785	3168244352
6055	97	536	3417	116862	13	116862	17110800
6062	90	556	3710	156385	15	104257	87852960
6065	98	689	4416	209933	8	233259	114307200
6072	61	417	2629	81058	8	81058	18586944
6145	8	51	220	2976	1	5168	268748928
6239	165	555	3812	93603	1	187206	152380965
6363	11	51	220	3637	1	7218	633600
6367	8	31	129	1625	2	3250	40716288
6454	102	560	3550	277198	18	307997	1234200
6520	110	576	3899	210962	16	234402	2272272750
6523	120	676	4520	204025	18	204025	33327480
6526	40	218	1322	26137	2	26137	101888640
6527	107	574	3978	156361	12	156361	507574188
6547	6	23	88	701	1	1394	264600
6555	3	15	39	128	1	256	45302988
6560	15	51	220	3502	3	7004	11353500
6569	15	68	316	9544	2	19088	40344000
6616	10	54	241	6966	1	13861	6400000
6618	10	52	238	5364	1	5960	169000
6620	10	53	243	6561	1	13122	51347560
6644	22	102	527	15324	4	24594	970200
6646	15	85	417	9802	1	19603	93675015

Metrics for Top 1% Interface Complexity							
Proc	LOC	N	V	E	CC	WOOD	INFO

6651	6	20	74	651	1	1305	4260010340736
6654	41	201	1097	51599	2	51599	53136
6656	31	144	756	25845	2	51692	281983936
6661	7	28	112	1400	1	2801	29641467303
6662	256	1225	8664	343162	1	686350	1784217600
6670	10	26	84	461	1	921	1444000
6675	22	155	870	13985	1	15539	2664288
6681	219	1192	7776	673891	42	1347512	594061371
6690	29	159	878	23713	5	47416	67886100
6694	11	46	205	2308	1	4616	1002177275
6702	76	327	1849	65831	1	131666	384750000
6703	21	115	595	14507	3	29005	39530064
6708	8	25	95	888	1	1763	10543232
6709	12	48	217	3198	1	6395	182988300
6715	9	53	243	3645	1	7290	215032896
6717	19	97	489	9374	5	18747	587147139
6719	16	71	348	5062	7	10117	22429696
6723	19	107	544	13790	5	27581	59122224
6726	9	29	116	1193	1	2386	71368704
6732	5	20	72	552	1	1104	19503125
6735	11	51	231	3066	1	6132	2479952475
6767	48	390	2747	30470	1	59884	2811072
6769	22	106	515	9814	4	9814	317011552
6771	27	137	685	19341	7	19341	471754800
6776	96	607	4076	148638	4	148638	1558352736
6779	92	624	4137	134645	3	134645	1774649088
6780	123	811	5713	259624	7	259624	3102119532
6782	92	592	3898	118118	3	118118	2081872508
6785	86	575	3829	129958	3	129958	1835618400
6787	119	749	5173	197101	5	197101	4635059639
6789	104	682	4651	162649	4	162649	2415167144
6794	223	1209	8396	278212	1	556434	725734768
6805	30	217	1255	19602	1	39204	1182775230
6829	907	10892	93218	9961191	1	9961191	21510412
6831	7	23	88	759	1	1518	186957568
6839	77	514	3361	109600	13	109600	108673488
6840	150	1029	6613	574659	28	574659	84375000
6845	78	620	3995	172306	13	172306	80199288
6860	70	415	2577	77860	7	77860	25032280
6864	14	62	281	4742	3	9485	199191776
6921	10	44	187	2492	1	4790	258267240
6928	4	11	31	124	1	247	10969344

**The vita has been removed from
the scanned document**