Communicational Measurement

Kevin A. Mayo
Sallie Henry

TR 91-30

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

October 11, 1991

Communicational Measurement

Kevin A. Mayo
Sallie Henry

Computer Science Department
Virginia Tech
Blacksburg, VA 24061

(703) 231-6931

mayoka@vtodie.cs.vt.edu
henry@vtodie.cs.vt.edu

Abstract

A software system is an aggregate of communicating modules. The interfaces supporting module (procedure, etc.) communication characterize the system. Therefore, understanding these interfaces (areas of communication) gives a better description of system complexity. Understanding in an empirical sense implies measuring, and measuring interfaces involves examining both the communicational environment and the exchanged data. These measures can lead to better design and aid in software quality assurance.

There are several different measures associated with the communication environment. Obviously, the structure or nesting level at the communication point is very interesting. The need to measure the data communicated also raises some very interesting questions concerned with data type and expressional form.

This paper reports on the efforts at Virginia Tech to measure, and thus, capture the complexities of software interfaces. Analyzing an Ada system of 85,000 lines of code validated the measures proposed here. The results of this research are very encouraging.

This research was sponsored in part by: Software Productivity Consortium (SPC), The Center for Innovative Technology (CIT), and Software Productivity Solutions (SPS).

Keywords: Software Design, Software Design Tools, Metrics, Software Quality Assurance, Interface Complexity, Ada Design Metrics

I. Introduction

In todays age the cost of software development is sky rocketing. Software Engineering tries to cap these costs with methodologies and tools to perfect the software development process. One such tool is the use of software quality measures. These product measurements can be taken at different stages of the life cycle, and have shown to be useful in error prediction and maintenance activity. [WAKS88] [LEWK88] [KAFD87] [BASV84] [CURB79a]

Metrics, as software quality measurements have become known, attempt to capture different aspects of designs or codes. The aspect depends upon the type of metric applied. There are many different measures that fall into three types: code, structure, and hybrid metrics. Within these different categories exist automatable and nonautomatable metrics, and it can be seen that if a metric is used it must be automated.

A metric only measures a single aspect of code or design written in a PDL. While this aspect can be meaningful, a single measure cannot suggest total system quality. A collection of metrics spanning over different aspects must be examined to derive meaning [YOUR89]. For example, a count of the system's procedures is meaningful, but does not indicate to total system quality because it neglects the complexity of each procedure.

This paper introduces a new set of metrics and shows their applicability to measuring communicational interfaces. It is our contention that viewing the communicational structure with weights associated with each data flow or invocation between modules gives an excellent indication to overall system quality and complexity. The next section gives background by introducing work already done in software quality metrics. Section III describes the metrics derived within this research and their importance. Because this research was performed over the Ada language, a section, IV, was devoted to Ada specific issues. Section V analyzes the data, while section VI discusses the conclusions and gives direction for future work.

II. Past Work in Metrics

Within code there exist objects, or tokens, that can be enumerated. A simple example could be a count of the relational operators (<, >, <=, etc) within a piece of code. Token counts, or functions of counts, from either a source or PDL representation are code metrics. Code metrics are well defined and tested throughout the literature [CONS86]. These metrics are often applied on a per module/procedure basis. These measurements include: Lines of Code (LOC), Halstead's Software Science (N, V, E) [HALM77], and McCabe's Cyclomatic Complexity (CC) [MCCT76], among others. The reader is directed to Appendix A for a full definition of: LOC, N, V, E, and CC. There also exist modifications to these measures, for example, modifications on CC include [HANW78] and [MYEG77] among others.

Examining token counts singularly ignores the internal structure of the code. Internal structures can be examined by viewing the data flow or nesting/looping structures within PDL or source code. Structure metrics try to capture the internal structure (interconnectivity) of source or PDL representation. Obviously, structure measurements are much more costly (in time and computing power) to gather than code metrics, but this effort pays for itself. Research in this area includes:

McClure's Module Invocation Complexity [MCCC78], Chapin's Module Complexity [CHAN79], Woodward's Knots [WOOM79], and Henry-Kafura's Information Flow [HENS81b] among others.

Finally, hybrid metrics are a function of code and structure metrics. Through a combining function, aspects captured by both code and structure metrics are distilled into a single hybrid metric. The cost to generate these metrics is the cost of generating both the underlying code and structure metrics, and then combining them to form the final hybrid metric. These include: Henry-Kafura's Information Flow (INFO) [HENS81b] and Woodfield's Review Complexity (WOOD) [WOOS82] among others. The reader is directed to the appendix.

Each category of metric (code, structure, and hybrid) has its own strengths and weaknesses. Applying code metrics over singular modules, as done in the past, does not maintain information on design or source code innerconnections or structure. On the other hand, structure metrics ignore syntactic complexity only to measure interconnectivity. Hybrid metrics attempt to measure both aspects. It is the contention of this research that hybrid metrics are the most useful. They contain the most information and are not trivial.

The motivation behind this research is that previous metrics are inadequate. They fall short of maintaining all the information needed to model a non-trivial indicator of source and PDL code interfaces. In other words, past metrics only viewed specific problem areas, and thus were applicable only to those areas. For example, while INFO is modeled within the scheme of data flow analysis, it ignores the inherent complexities that different data structures introduce. McClure weights the different structures by selection or repetition; however, this approach neglects the complexity associated with each of the constructs' (selection/repetition) control clause. Also, various styles of repetition constructs (Test-Before, Test-After) were put into a single class; whereas, there could possibly be a difference in complexity among these structures. The interface complexity metric attempts to gather these aspects that INFO and McClure overlook, i.e.: how much data is passed (a weight), and a better understanding of the nesting/looping structures within the source or PDL representation.

It is the contention of this research that measuring the complexity of a communication interface involves understanding what complexity is and how the human mind deals with it [SCHB80] [WEIG71]. This aspect, psychological complexity, is not the theme of this report so it will not be addressed directly, only indirectly through the definition of the interface complexity measures.

III. Interface Measurements

Defining a new set of measures must be predicated by a need. All too often researchers define and test measures that are not necessarily meaningful. It is not the intention of this research to present just another set of metrics, but to present a useful set of metrics that can be manipulated and modified for the software engineer's use.

To measure the interface between modules, this communication must be understood and easily recognizable. There are three types of communication that can occur: direct, indirect, and global. Direct communication is data passage through a procedure call. Indirect communication is common data passed between sub-procedures through procedure calls (e.g., A calls B sending 'x', then A calls C sending 'x' -- indirect communication between B and C has occurred). Global communication involves data passage from one procedure to another through a non-local data item. The interface complexity captures all three types of communication.

Once communication has been determined to exist between two modules, then the interface measurement is gathered. As mentioned before, to measure correctly the interface complexity both the data (parameters to communication, Parameter Complexity) and the environment (Environment Complexity) must be considered. This implies the interface measure is a two-tuple. To gather both metrics several underlying measurements are taken.

Parameter Complexity

To measure the data passed within a communication (e.g. parameters), a weighting convention must be adopted with respect to the data. Measuring a data expression (e.g. 'x+y') implies that both operands and operators must be considered which has not been done in the past. This research uses a weight associated with the data's defined type as the contribution of the operands. Psychologically is should be obvious that INTEGERs and RECORDs have different complexities. Therefore, there is a need for a consistent data type measure. This ranks types according to psychological complexity as perceived by this research.

Single valued base types, or parent types like INTEGER or REAL, have user defined constant complexity levels. Type complexities of aggregates of component data are calculated as functions of the components' data type. Aggregates include arrays, records, and files. To calculate array and record type complexities use the following formulas:

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

$$\text{Record Complexity} = \frac{\sum_{1}^{\#} \text{Component Type Complexity} * \text{Multiplier} + \text{Adder}}{\#\text{ of Components} + \text{Component Adder}}$$

The design of the formulas centered on each structures' cognitively difficult aspects. Within the array formula there is consideration for the array element type with the number of dimensions. Yet, no consideration is given to each dimension's span -- understanding a dimension implies understanding the span. The record type formula regards each component's type (sub-fields) and the number of components.

As mentioned, the base type complexities may be modified, also the aggregate type formulas contain 'tuning' parameters. These 'tuning' parameters include: Array Adder / Multiplier, and Record Multiplier / Adder / Component Adder. By raising or lowering these constants the software engineer may derive an optimum set of weights based on environment and experience.

While not as important, the impact of operators must be considered on the data expression complexity. Clearly, to show the difference in psychological complexity consider addition (+) and division (/). Figure 1 depicts the relationship between types and operators as they influence the data expression complexity.

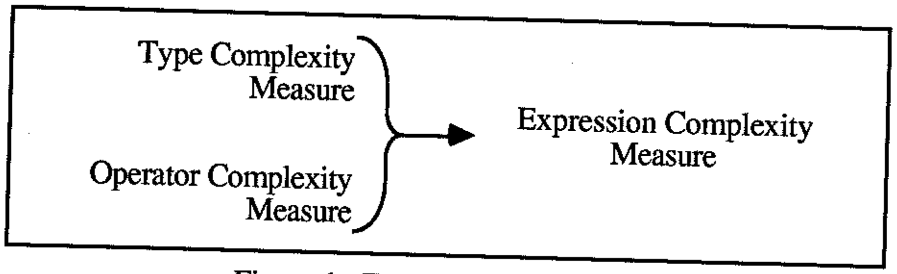

Figure 1: Expression Complexity

It is not necessary to limit the expression complexity to data within module communication. The expression complexity can be applied to all expressions within the code or PDL. This affords the ability to rank expressions, and find complex expressions. (With the software engineer defining what ranks are 'complex'.)

This measure represents the first of two different measures of the parameter complexity. The second measure captures the modifications to the variables. This other measure is described later.

Environment Complexity

The second interface complexity aspect is capturing the complexity at the place or environment of the communication. This involves analyzing the code or PDL for complex structures. Program or PDL code is a union of three structures: concatenation (straight line code), repetition (path through code is repeated several times), and selection (path is determined by validity of a condition). Of these structures repetition and selection adds to the complexity by changing the execution flow.

This research defines the environment complexity as a function of the code or PDL structure as seen through the selection and repetition statements. In other words, the further down within nested code (either selection or repetition) the more complex the environment. To measure these structures' influences upon the environment, a weighting scheme is needed. These weights try to capture the psychological complexity of the given structure. These weights can be changed by the software engineer to test for special effects or aspects.

Examining just the structures are slightly misleading. There also must be an investigation of the structure control variables or expression. Combining the structure complexity with the expression complexity of the control variables therefore defines the environment complexity. Nested structures add their environment complexity to the environment complexity of the outer level. Figure 2 sketches the relationship between expression complexities, structure complexities and environment complexity.

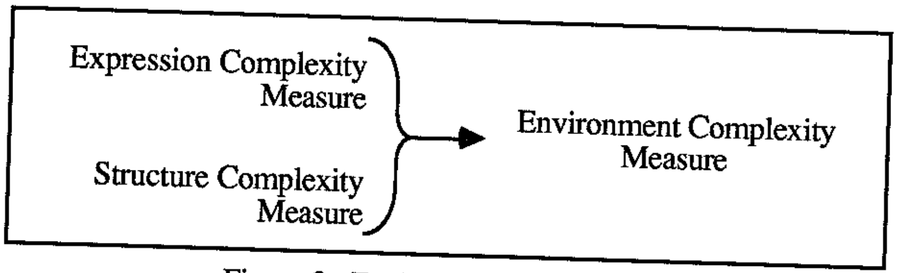
Expression Complexity Measure
Structure Complexity Measure
Environment Complexity Measure

Figure 2: Environment Complexity

This defines the environment aspect of the interface complexity tuple.

Reference Complexities

As mentioned above, there are two different ways to measure the parameter complexity. The first method, already described, looks at a function of the variables type and the operators within the data expression. The alternative method is measured from the modification complexity of the variables and the operators within the data expression. In other words, the second accounts for how often a variable is modified and to what extent.

To measure how often or to what extent a variable is modified, separate measures for each variable within source or PDL code must be maintained. Two measures are taken for each variable within each procedure/module: Ref_Read and Ref_Write. Ref_Read maintains the number of accesses and the sum of environment complexities for each access. While Ref_Write keeps up the number of modifications and the sum of expression and operator complexities for each expression. In these ways, Ref_Read indicates how often and in what context a variable is used. Ref_Write reports on how often and to the extent of modifiability. Figure 3 shows an example of the measures.

```
x = x + 1;
y = x * z;
```

Ref_Read to variables: x, z, Numeric_Costant

Ref_Write to variables: x, y

Figure 3: Ref_Read and Ref_Write

IV. Ada

Until this point the discussion has been language unspecific. The formulas and ideas can be applied to any language. Yet, this study was performed over an existing Ada system currently in use [CHAB90]. Ada, large as it is, caused several problems. This section addresses a few of these problems and presents solutions.

Ada has an extensive typing system supporting all manner of structures and variations. The first area of concern is that of type modifiers, e.g., DELTA, RANGE, etc. These, like the base or parent types, have weights defined by the software engineer. Adding these weights to the base type complexities yield the modified type complexity. Another aspect is that of private types, raising the question "Is there a type complexity associated with a variable of a private type, and if so, is the complexity less because of the unknown implementation?" This research took the position that the complexity is known though the implementation is not.

The next area of significance concerns overloaded operators and operands. Many cases are resolved through parameter type checking. Overloaded operators have an operator complexity defined by the software engineer. In this research, overloaded operator defined complexity differed from standard operator defined complexity by at least an order of magnitude.

Packages also added different concepts to this intra-module communication model. For example, several packages within the system analyzed contained only variable definitions, e.g., a null package body. Yet, these packages, and their variables, are throughout the system creating data flows from procedures to these null-package bodies. This situation easily exists within Ada, but other languages do not allow for this circumstance.

V. Data and Results

To validate the measurements proposed here, a system of considerable length was to be analyzed. Software Productivity Solutions (SPS), Inc., in Melborne, Florida came to the rescue with a system of about 85,000 lines of code. Written entirely in Ada, this system contains several thousand procedures. The function of this sample system is to analyze an Ada PDL.

Data Gathering Tool

At Virginia Tech there exists a system that can gather several validated metrics. These metrics include: LOC, N, V, E, CC, WOOD, INFO. This system is a three phase automatic metric generating tool. Figure 4 depicts the outline of the system:

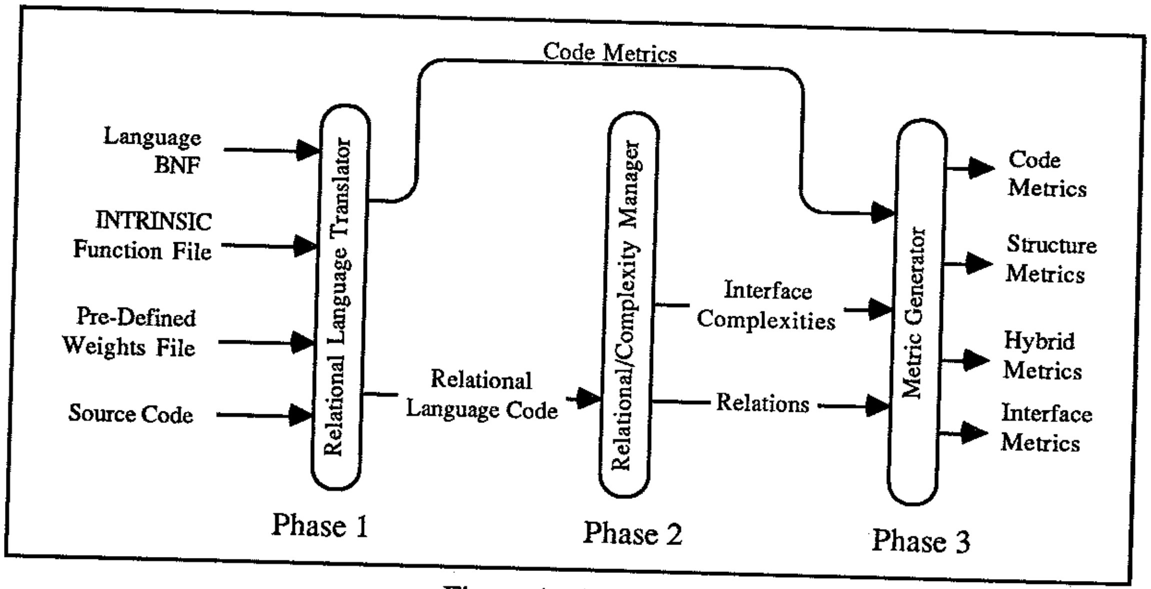

Figure 4: Analysis Tool

This system takes source code, in phase 1, and generates a generic representation in relation language along with code metrics. This representation contains all the necessary information to process the structure metrics. This representation also offers several advantages, such as hiding proprietary data. There are several languages that phase 1 can work over: Ada, C, Pascal, and FORTRAN.

The second phase calculates all interface complexity metrics as well as the information flow relations. [KAFD82] The actual calculation of the Henry-Kafura Information flow metric takes place in phase 3. Phase 3 is also a display tool to show all metrics generated. These may be displayed by procedure level, or user defined module level.

Data Reduction

Analysis shows 20,975 observations of communication throughout the system. Each communication has associated with it the interface complexity tuple:

Interface = (proc1, proc2, accn, acc, para1, para2), where:

• proc1	The communicating procedure
• proc2	The invoked procedure, e.g., p1 communicates with p2
• accn	Number of calls (or accesses)
• acc	Access complexity (environment aspect of interface)
• para1	Data complexity calculated with expression complexity
• para2	Data complexity calculated with modification (ref_write) complexity.

These 20,975 tuples capture all the communication within the system in the above measures. While it is possible to examine each of these tuples individually, the interface complexity should represent the complexity of a single procedure to all other procedures. Collapsing the data is necessary to view the data by per procedure basis. For each procedure, all calls to and from are joined. This produced eight measures for each procedure:

Collapsed interface for each procedure *i* is:

(CallsTo, CompTo, Para1To, Para2To,
CallsFrom, CompFrom, Para1From, Para2From),

where:

CallsTo	The sum of CALLS (or accesses) from i TO all other procedures (accn), i.e., Proc1 = i, and Proc2 = k, for all k.
CompTo	The sum of the access COMPlexity (acc) from procedure i TO all other procedures, i.e., Proc1 = i, and Proc2 = k for all k.
Para1To	The sum of the first PARAmeter complexity (para1) from procedure i TO all other procedures, i.e., Proc1 = i, and Proc2 = k for all k.
Para2To	The sum of the second PARAmeter complexity (para2) from procedure i TO all other procedures, i.e., Proc1 = i, and Proc2 = k for all k.
CallsFrom	The sum of CALLS (or accesses) to procedure i FROM all other procedures (accn), i.e., Proc1 = j, and Proc2 = i, for all j.
CompFrom	The sum of access COMPlexity (acc) to procedure i FROM all other procedures, i.e., Proc1 = j, and Proc2 = i, for all j.
Para1From	The sum of the first PARAmeter complexity (para1) to procedure i FROM all other procedures, i.e., Proc1 = j, and Proc2 = i for all j.
Para2From	The sum of the second PARAmeter complexity (para2) to procedure i FROM all other procedures, i.e., Proc1 = j, and Proc2 = i for all j.

Figure 5 shows relationship among the interface complexity and the collapsed interface complexity measures.

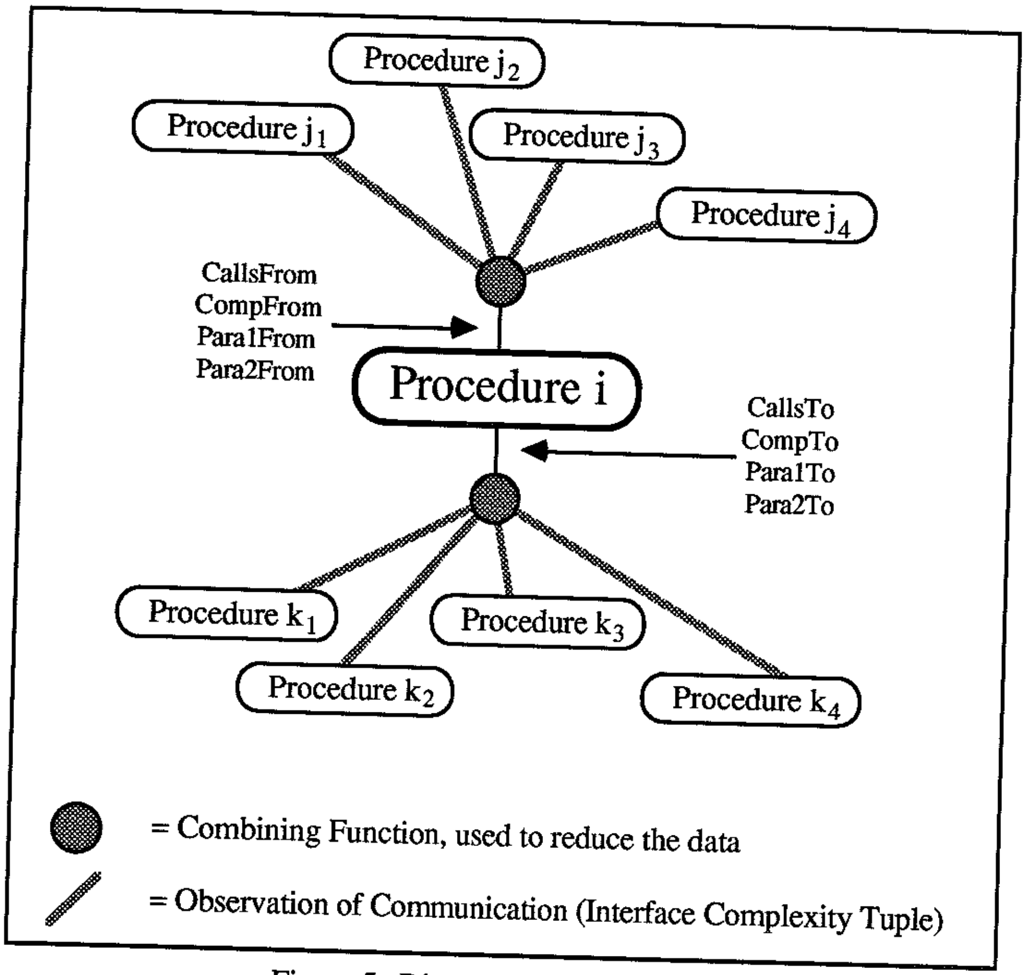

Figure 5: Diagram of Data Reduction

The analysis of this data is in two parts: numerical analysis (correlational analysis, functional correlation), and subjective examination. A three part approach is because no error history for the system is available, and therefore no causative study can be included. Also, the type and operator weights used within this study were set by the researchers. Since there was no data history for reverse analysis -- the researchers choose the most intuitive set of weights they could.

Numerical Analysis

The first step is correlation analysis. Pearson correlations serve to show whether the interface complexity measures correlate to established metrics. This question is important because of the need not to generate useless or redundant measures. Table 1 lists the correlations among the established metrics defined within Appendix A -- LOC, V, V, E, CC, WOOD, and INFO, with CallsTo, CompTo, Para1To, Para2To, CallsFrom, CompFrom, Para1From and Para2From.

	CallsTo	CompTo	Para1To	Para2To	CallsFrom	CompFrom	Para1From	Para2From
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

Table 1: Correlations Among Established Metrics and Interface Measures

There are several interesting points that Table 1 raises. The first, consider the correlational distinction between the two different methods of calculating the data complexities. Table 1 shows higher correlations among the first method (Para1To and Para1From) to the code metrics than the second method (Para2To and Para2From). Yet, this is a by-product of the type and operator weights selected for this study. These measures and their correlations are different for different sets of weights.

Notice that there is a very high correlation between N, V, and E with CallsTo, CompTo, and Para1To. Token counts make up these measures to some degree, and this is the cause of the correlation. Apparently CallsTo, CompTo, and Para1To gather data somewhat like Halstead's Software Science, and therefore considering these separately would be redundant. However, given a system design with only the interfaces an indication to Halstead's Software Science can be attained. Thus, at a design stage Halstead's metrics can be predicted for the system.

Another point to address is the low correlation of CallsFrom, CompFrom, Para1From and Para2From to the code metrics. Logically these metrics measure the complexity associated with the communicating procedure, not the current procedure.

The correlations are interesting, yet, if taken singularly they fail to take advantage of the interface complexity measures taken as a whole. Functions of the interface complexity measures exploit their richness. Managers and software engineers both view these metrics. Hence, it is important to create functions that intuitively hold meaning for both manager and software engineer. Once attained, this function could easily be used by both parties, and quickly enters a working environment.

This study examines several functions of the interface complexity measures. These functions are:

- $F1_{+}$ = (CompTo * Para1To) + (CompFrom * Para1From)
- $F2_{+}$ = (CompTo * Para2To) + (CompFrom * Para2From)
- $F1_{*}$ = (CompTo * Para1To) * (CompFrom * Para1From)
- $F2_{*}$ = (CompTo * Para2To) * (CompFrom * Para2From)

In each function there are two influences: data flowing into the procedure, and data flowing out of the procedure -- represented by the first and second term respectively. The equations are coded using the following format: F(parameter complexity used)$_{\text{Combination Method}}$. Therefore, $F1_{+}$ and $F1_{*}$ both deal with the first parameter complexity measurement method, while $F2_{+}$ and $F2_{*}$ deal with the second. Like wise, $F1_{+}$ and $F2_{+}$ are combined using addition, while $F1_{*}$ and $F2_{*}$ use multiplication. This gives several combinations of the interface complexity measures. These combinations were derived to be simplistic but meaningful, and their format was based upon the 'style' of metric functions derived in the past (i.e., the use of addition and multiplication). Most of all, these functions must be usable by both software engineers and managers alike.

Table 2 lists the correlations among these four functions and selected code (LOC, E, and WOOD) and structure metrics (INFO). Established metrics that were not chosen (N, V, CC) correlated very high with the metrics listed (LOC, E, WOOD, and INFO), and therefore, left out due to redundancy.

	$F1_{+}$	$F2_{+}$	$F1_{*}$	$F2_{*}$
LOC	0.285	0.340	0.033	0.033
E	0.477	0.577	0.030	0.030
WOOD	0.498	0.590	0.062	0.062
INFO	0.510	0.068	0.920	0.920

Table 2: Functions of Interface Complexity Measures *vs.* Established Metrics

The first function, $F1_{+}$, correlates moderately to both code and structure metrics. These correlations imply that this function measures aspects of both code and structure metrics as a hybrid metric should.

Functions $F1_{*}$ and $F2_{*}$ are similar down to the hundredths place. Therefore, it is not important to examine both of them with these type and operator complexity weights. Yet, an interesting aspect of these functions is their high correlation to the structure metric INFO. This implies that these functions capture information like INFO, yet, on second inspection this correlation could be the influence of the rapid growth due to the multiplication. Also, this multiplicative growth produces poor correlations to additive growth code metrics, i.e., LOC, E, and WOOD. Therefore, these functions do not prove to be of much interest on their own.

Comparing the functions $F1_+$ and $F1_*$ shows questionable relationships when compared to $F2_+$ and $F2_*$. Functions $F2_+$ and $F2_*$ correlations demonstrate an inverse effect. Function $F2_+$ has moderate correlations with code metrics (i.e., LOC, E, and WOOD,) and low correlation with the structure metric INFO; while, function $F2_*$ has the reverse (i.e., very low code metric correlation and very marked correlation with structure metrics.) This characteristic is interesting in itself, yet, this pattern is not continued over the functions $F1_+$ and $F1_*$. This difference lies in the calculation of the different methods of data complexity.

Finally, the question "What of $F2_+$?" Obviously this function does not correlate with the INFO structure metric, but can meaning be derived from the correlations with the code metrics? A better question is "Should meaning be derived?" With all the metrics available today, it is not wise nor profitable to produce redundant metrics.

These correlations depend upon the underlying complexity functions. These complexity functions rely on the type, operator, and structure weights. Therefore, changing these weights can change these correlations. To 'tune' these weights correctly to an optimum set an error history or access to the programmers is crucial. This study had neither of these available, so there was a subjective examination of the code.

Subjective Examination

The SPS data included the source code for the entire system. Therefore, a comparison among the eight interface complexity measures per procedure with the procedure's source code was possible. The procedures were ranked ordered for each eight measures. Procedures selected from these lists corresponded to extremely high measures with relation to their respective lists. In fact, all the selected procedures were within the top 1% of the measure in question.

The subjective evaluation produced several interesting results. First, code metrics relate well to the interface complexity measures in pin-pointing procedures with questionable measures. In other

words, procedures that were flagged by the established metrics, were also marked by the interface
complexity measures.

A second intriguing point brought out by the subjective evaluation is the influence of the source
language. The original system is written in Ada, therefore, Ada specific characteristics were flag-
ged by the interface complexity measures. One such example is that of a package that contain only
declarations. While this package does not have any code to execute (null package body), there are
many data flows into and out of the package. This is done by accessing or modifying a variable
that is defined within the package.

Utility routines were also flagged with questionable interface complexity measures. This is be-
cause they are greatly used throughout the system. In future examinations, utility routines should
be left out to search for more meaningful measures.

VI. Discussion

The subjective examination of the code helped to verify that the interface complexity measures were
collecting the correct data. Therefore, this adds validity to the functions of the interface complexity
measures. These functions indicate problem areas or 'hot spots' within the code that may need
further consideration. This consideration may include recoding or even redesigning that particular
section. These measures also can be used at design time to differentiate between two different de-
signs, and thus, save money in development further down the life cycle.

A strong point of this research is that the system analyzed was an in use commercial system. This
evades the 'student data' problem that plagues many other studies. The tunability of the type, op-
erator, and structure weights are also strong points of this project. While it may seem that the
weights could be tuned to produce any results, it is the contention that proper weights should be
derived by reverse engineering with error histories.

A weak point of this research is that there was no error history available to the researchers. This
diminished its impact, but did not kill its validity or need. It was for this reason that a subjective
evaluation was undertaken.

This research needs to be extended and validated in other languages besides Ada. This would al-
low a comparison of languages on these measures that might prove interesting or useful. Also,

each development environment and methodology needs to be assessed to attain an optimum set of weights.

Bibliography

[BASV84] Basili, V. R., and Perriconne, B. T., "Software Errors and Complexity: An Empirical Investigation", Communications of the ACM, Vol 27, No. 1, pp. 42-52, January 1984.

[CHAB90] Chappell, B. T. S., Henry, S., Mayo, K., "Measurement of Ada Throughout the Software Development Life Cycle", Proceedings of the Eigth Annual National Conference on Ada Technology, pp. 525-532, March, 1990.

[CHAN79] Chapin, N., "A Measure of Software Complexity", Proceedings of the 1979 National Computer Conference, New York, pp. 995-1002, 1979.

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

[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.

[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.

[HENS81a] Henry, S. M., Kafura, D., and Harris, K., "On the Relationships Among Three Software Metrics", Proceedings of 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., Kafura, D., "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics", *Software Practice and Experience*, June 1984, pp. 561-573.

[KAFD87] 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.

[KAFD82] Kafura, D., Henry, S. M., "Software Quality Metrics based on Innerconnectivity" Journal of Systems and Software, pp. 121-131, 1982.

[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.

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

[MCCC78] McClure, C. L., "A Model for Program Complexity Analysis", Proceedings of the 3rd Conference on Software Engineering, pp. 149-157, 1978.

[SCHB80] Schneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishing, Inc., Cambridge, MA, 1980.

[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.

[WEIG71] Weinberg, G. W., The Psychology of Computer Programming, Van Nostrand Reinhold Co., New York, NY, 1971.

[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.

[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.

Appendix A: Metric Definitions

This appendix defines the metrics that are used within this research:

LOC:	Lines of Code
N, V, E:	Halstead's Software Science Indicators
CC:	McCabe's Cyclomatic Complexity
WOOD:	Woodfield's Review Complexity
INFO:	Henry-Kafura's Information Flow Metric

Lines of Code (LOC)

This metric, most probably the oldest, is an enumeration of the lines of code. Simple, but indicative of the complexity. There is, however, some debate what constitutes a line of code from one language to the next.

Halstead's Software Science (N, V, E)

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).

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

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

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

The volume measure can change depending upon the size of the algorithm chosen for the task. If 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:

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

Because V^* can never be calculated an estimator must be attained:

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

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:

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

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

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

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

McCabe's Cyclomatic Complexity (CC)

McCabe's cyclomatic complexity metric is built upon the selection and repetition structure of the code. [MCCT76] In the model, code is represented by a strongly connected graph G. 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 graph G
N = Number of Nodes in graph G

V(G) represents the number of control paths in the algorithm. McCabe suggested an upper limit of V(G)=10, stating that this bound would curtail 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. Yet, the generation of this metric reverts to the enumeration of the decisions within the code plus one [WOOM79]. These decisions include the selection and repetition constructs. Also, the total complexity of the set of modules (the program) is equal to the sum of all module V(G)'s.

Henry-Kafura's (INFO)

Information flow is the basis for the structure metric defined by Henry and Kafura [HENS81b]. Here 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 that 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.

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

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

C_{ip} = Internal complexity of p
Fan-in_p and Fan-out_p are defined as before.

The hybrid form of the Henry-Kafura metric is used within this study as INFO.

Woodfield's Review (WOOD)

Woodfield's review complexity 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 idea 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:

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

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