Software Structure Metrics Based on Information Flow

SALLIE HENRY, MEMBER, IEEE, AND DENNIS KAFURA

Abstract—Structured design methodologies provide a disciplined and organized guide to the construction of software systems. However, while the methodology structures and documents the points at which design decisions are made, it does not provide a specific, quantitative basis for making these decisions. Typically, the designers' only guidelines are qualitative, perhaps even vague, principles such as "functionality," "data transparency," or "clarity." This paper, like several recent publications, defines and validates a set of software metrics which are appropriate for evaluating the structure of large-scale systems. These metrics are based on the measurement of information flow between system components. Specific metrics are defined for procedure complexity, module complexity, and module coupling. The validation, using the source code for the UNIX operating system, shows that the complexity measures are strongly correlated with the occurrence of changes. Further, the metrics for procedures and modules can be interpreted to reveal various types of structural flaws in the design and implementation.

Index Terms—Complexity, design methodologies, information flow, software metrics, UNIX.

I. Introduction

THE ANNUAL cost of software is currently estimated at $20 billion and rising rapidly [40]. Of this amount, a frustratingly large fraction, as high as 50-75 percent over the lifetime of a typical software system, is spent on maintaining unreliable, poorly structured software. This high maintenence factor is due in large part to errors made during the design phase [40] and to the choice of a system structure which has an overly complex interconnectivity among its components [4], [25], [35]. In the face of these problems one of the most significant and practical challenges facing software engineers today is the development of tools and techniques to restrain software costs while enhancing software quality.

Much of the recent work in software engineering has recognized that reducing costs and increasing quality are compatible goals which can be achieved when the complexity of the software structure is properly controlled. For example, the use of structured design methodologies allow the controlled introduction of complexity via levels of abstraction, virtual machines, or layered hierarchies [11], [23], [28], [29]. By establishing an ordered discipline during the design phase these techniques have had notable success in producing higher quality, lower cost software. Research has also concentrated on the development of specification techniques [31], testing and debugging methods [26], and the construction of programmer-workbench style software development systems [12], [21]. These efforts have in common the attempt to maintain control of the emerging software product either by making explicit the decisions involved with the specification and design of the system or by automating (making implicit) the mechanical operations involved in its assembly. Because of the fundamental relationship between complexity and software quality several recent studies have focused attention on the development and validation of a set of quantitative metrics to measure the complexity of software structures [8], [17], [36], [38]. These metrics are useful management aids, important design tools, and also basic elements in research efforts to establish a quantitative foundation for comparing language constructs and design methodologies.

Manuscript received August 7, 1979; revised August 7, 1980 and March 12, 1981.

S. Henry is with the Department of Computer Science, University of Wisconsin, LaCrosse, WI 54601.

D. Kafura is with the Department of Computer Science, Iowa State University, Ames, IA 50011.

Previous research in software metrics has been concentrated in one of three major areas. First, many studies explored metrics based on the lexical content of a program. These studies included Halstead's original work [17], which counted operators and operands, and its subsequent development, the cyclomatic complexity measure by McCabe [24], which counted the number of branch points in a program, and the extensive study by Thayer *et al.* [36], which counted the occurrence of a wide variety of statement types. The common thread in this work is the counting of lexical tokens without specific regard for the structure created by those tokens. These simple lexical measures have proven to be surprisingly robust in predicting various aspects of software quality.

A second major research direction is based on the application of information theoretic concepts, such as entropy, to the formulation of software metrics. This approach grew out of work by Alexander and others [1] in the field of architecture and design. Channon [10] was the first of several [33], [34] who used this idea to analyze the design of software structures. One difficulty with Channon's measure is that each assumption of each procedure must be explicitly determined. Not only is this level of detail excessive but, since the assumptions cannot be deduced automatically, it requires a manual analysis by the designer/implementor. For large-scale systems such a time-consuming operation is not feasible. The other information-based measurements also require a considerable degree of parameter estimation—such as determining the probability of a change in one module causing a change in any other module. While these techniques are thoroughly grounded in an underlying theoretical base, they have not proven to be as practical as the lexical measures.

The third major type of metric deals directly with the system connectivity by observing the flow of information or control among system components. The work of Yin and Winchester [38] focused on the interface between the major levels in a large, hierarchically structured system. The work of Henry using information flow [19] also falls into this category and is the subject of this paper. The information flow approach is more detailed than the Yin-Winchester study because it observes all information flow rather than just those across level boundaries. However, in contrast to the information theoretic measures, the information flow method is a completely automatable process using fairly standard data flow analysis techniques [2], [18].

We believe that the information flow technique is an appropriate and practical basis for measuring large-scale systems for three reasons. First, as will be seen in the subsequent sections of this paper, the major elements in the information flow analysis can be directly determined at design time. The availability of this quantitative measurement early in the system development process allows the system structure to be corrected with the least cost. Second, the research of Henry [19] has shown that the information flow technique, illustrated in Section II of this paper, reveals more of the system connections than other ordering relations such as "calls," "uses" [29], or "dependency" [20]. In this sense the information flow measurements are more sensitive than measurements based on other relations to the nuances of system structures. Third, by observing the patterns of communication among the system components we are in a position to define measurements for complexity, module coupling, level interactions, and stress points. Evaluation of these critical system qualities cannot be derived from the simple lexical measures. Thus, the information flow method leads to automatable measures of software quality which are available early in the development process and produce quantitative evaluation of many critical structural attributes of large-scale systems.

The most important practical test of a software metric is its validation on real software systems. There are many scales against which a complexity metric, such as the one proposed in this paper, may be validated (e.g., determining the correlation of the measure to programming time, comprehension time, error rate, etc.). In this paper we will validate the complexity metric by demonstrating that it is highly correlated with the occurrence of system changes. The software system used in this validation study is the UNIX operating system (version 6). We chose to use an operating system because they are important objects of study among large-scale systems. The operating system forms the interface between the hardware and applications programs and also creates the virtual machine environment which is critical to the success of the applications which it supports. In addition, operating systems tend to be among the more complex of software systems, both in their design and in the interrelationships of their parts. Thus, an operating system is a sound basis for the measurement and validation of software metrics. The UNIX operating system was chosen as the vehicle for the information flow analysis for several reasons. First, UNIX is written in a high-level language and its internal mechanisms are well documented [22], [32]. Second, UNIX is a large enough operating system to use as a reasonable experimental basis, yet is small enough for a manageable research project with limited resources. The third reason for selecting UNIX is the fact that UNIX software was designed for users and not as a toy or experimental system. In particular, it was not written to illustrate or defend a favored design methodology. Fourth, UNIX is universal in that it is installed in many environments and on several different machines. Thus, we expect our results to be understood, and perhaps criticized, in detail by a larger audience. The fifth reason is the functionality of UNIX. UNIX, containing a powerful I/O system, a simple virtual memory structure, dynamic task creation and deletion, simple interprocess communication, and some protection features, possesses the typical spectrum of operating system functions.

Section II of this paper defines the basic notions of the information flow technique and illustrates these ideas with a simple example. Section III applies this technique in analyzing the UNIX operating system. A structural complexity measure is presented and used to evaluate the procedure and module structure of UNIX. Section IV contains a validation of the complexity metric by showing that this metric is strongly correlated with the existence of errors in UNIX. In Section V we describe our initial, and as yet incomplete, attempt to develop a set of module interface measurements (e.g., coupling). This material is presented primarily to show the wide range of structures which can be based on the information flow method.

II. Information Flow Concepts and Definitions

In this section various types of information flows are informally presented by example. The mechanisms for deriving these flows are bypassed here but are given in [19]. Following the example, formal definitions of information flow are given.

Fig. 1 shows six modules, A,B,C,D,E,F, a data structure DS, and the connections among these modules and the data structure. Module A retrieves information from DS and then calls B; module B then updates DS. C calls D and module D calls E and E returns a value to D which D then utilizes and then passes to F. The function of F is to update DS.

Generated from this example are the following direct flows of information (termed direct local flows).

$A \rightarrow B$,
$C \rightarrow D$,
$D \rightarrow E$,
$D \rightarrow F$.

These flows are simply the ones observed in a calling structure. There are also some indirect flows of information (termed indirect local flows). These indirect flows are

$E \rightarrow D$,
$E \rightarrow F$.

The first flow results when E returns a value which is utilized by D and the second flow results when the information that D receives from E is passed to F. Notice that this is an exam-

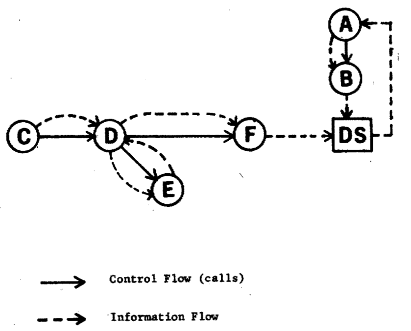


Fig. 1. An example of information flow.

ple of an information flow existing when no control flow exists. There also exists a flow of information through the global data structure *DS*, again without a matching flow of control. These global flows are

$$B \rightarrow A,$$
$$F \rightarrow A.$$

Both *B* and *F* update *DS* and *A* retrieves information from *DS*. The information flow relations generated from this example are summarized in Fig. 2.

The following definitions describe precisely the various types of information flow presented informally above. Additional details regarding these definitions can be found in [19].

Definition 1: There is a global flow of information from module *A* to module *B* through a global data structure *D* if *A* deposits information into *D* and *B* retrieves information from *D*.

Definition 2: There is a local flow of information from module *A* to module *B* if one or more of the following conditions hold:

1) if *A* calls *B*,

2) if *B* calls *A* and *A* returns a value to *B*, which *B* subsequently utilizes, or

3) if *C* calls both *A* and *B* passing an output value from *A* to *B*.

Definition 3: There is a *direct* local flow of information from module *A* to module *B* if condition 1) of Definition 2 holds for a local flow.

Definition 4: There is an *indirect* local flow of information from module *A* to module *B* if condition 2) or condition 3) of Definition 2 holds for a local flow.

An important property of the information flows defined above is that the knowledge necessary to construct the complete flows structure can be obtained from a simple procedure-by-procedure analysis. The example presented in Fig. 1 will be used to illustrate this process. The left part of Fig. 3 shows possible skeleton code for procedures *D*, *E*, and *F*. The corresponding set of relations generated for each procedure are shown in the right part of the figure. Each relation is named (e.g., *D*1,*D*2, · · ·, *F*1,*F*2) merely for convenience and this is not a necessary part of the relation. In this example it is assumed that the only operations involving the values of the variables *P* and *Q* are those explicitly shown.

In general, a flow of information is represented by a relation of the form

destination ← source1,source2, · · ·, source*N*.

```
Global Flows       B -> A
                   F -> A

Local Flows
  Direct           A -> B
                   C -> D
                   D -> E
                   D -> F
  Indirect         E -> D
                   E -> F
```

Fig. 2. Information flow relations generated from Fig. 1.

The source(*s*) and destination are denoted in one of four possible ways: 1) *X.n.I* denotes the value of the *n*th parameter of procedure *X* at the procedure's invocation; 2) *X.n.*0 denotes the value of the *n*th parameter of procedure *X* at the termination of the procedure; 3) if *X* is a function, *X.*0 denotes the value returned by that function; and 4) *X.D* denotes an access by procedure *X* to the global data object *D*. As an example, consider procedure *F*. Relations *F*1 and *F*2 indicate that the output (i.e., final) values of its parameters are the same as their original input values. Relation *F*3 indicates that the global data structure *DS* is updated based on the input values of its two parameters.

Notice that the form of the relations is in no way affected by the syntax of the source language being used. Also note that no distinction is made between a flow of information established by a passed parameter and one established by a shared global data structure.

Using the relations generated on a procedure-by-procedure basis it is possible to combine these relations to form the complete structure of information flow by a simple substitution process. For example, we can derive the flow emanating from procedure *D* which ultimately is used to update *DS* by working backward from *DS* in the following manner:

$$\begin{array}{lll}
F.DS & \leftarrow F.1.I,F.2.I & (F3) \\
 & \leftarrow F.1.I,E.0 & (D3) \\
 & \leftarrow F.1.I,E.1.I,E.\text{constant} & (E1) \\
 & \leftarrow F.1.I,D.1.I,E.\text{constant} & (D1) \\
 & \leftarrow E.1.0,D.1.I,E.\text{constant} & (D2) \\
 & \leftarrow E.1.I,D.1.I,E.\text{constant} & (E2) \\
 & \leftarrow D.1.I,E.\text{constant} & (D1)
\end{array}$$

The first line is just relation *F*3. The second line is obtained by replacing *F.*2.*I* by *E.*0 from relation *D*3, and so on. This substitution process can also be seen in the information flow structure shown in Fig. 4. This structure presents all of the possible paths of information flow in the subset of the procedures we are considering.

It should be clear from this brief presentation that the information flow paths are easily computable from the relations which have been generated individually for each procedure. The current techniques of data flow analysis [2], [18] are sufficient to produce these relations automatically at compile time. Furthermore, if a sufficiently precise design language is used, such that the relations can be generated from the design code, then the information flow analysis can be done at design time. This justifies our earlier claim that the information flow metrics, based on this analysis, can be made available early in the system development process. Having illustrated the underlying method of capturing the flows of information in an

```
Procedure D(P);
    .
    .
    .                        D1: E.1.I <-- D.1.I
Q := E(P);                   D2: F.1.I <-- D.1.I,E.1.O
F(P,Q);                      D3: F.2.I <-- E.O
    .                        D4: D.1.O <-- E.1.O,F.1.O
    .
    .
end D;

Procedure E(P);
    .
    .
    .
E := P + 3;                  E1: E.O   <-- E.1.I,E.constant
    .                        E2: E.1.O <-- E.1.I
    .
    .
end E;

Procedure F(P,Q);
    .
    .
    .                        F1: F.1.O <-- F.1.I
DS(P) := Q;                  F2: F.2.O <-- F.2.I
    .                        F3: F.DS  <-- F.1.I,F.2.I
    .
    .
end F;
```

Fig. 3. Possible code and relations for Example 1.

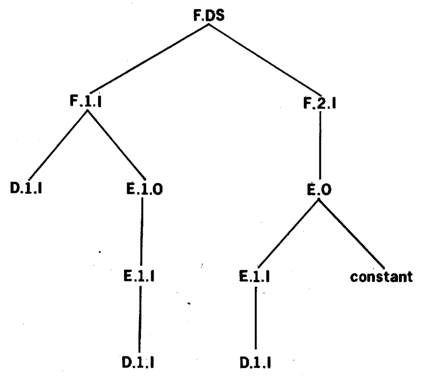


Fig. 4. Portion of the flow structure for Example 1.

automatable way we will proceed in the next section to show how these flow paths can be used as the basis for measuring the complexity of software structures.

III. Complexity Measurement

An information flow analysis was performed on the UNIX operating system and the global flows and local flows derived from the information flow analysis were used to obtain measurements for the UNIX system. These measurements, including the complexity measurements for each procedure, are presented in this section. The local flows and global flows are used to operationally define the modules in the system and to reveal possible weaknesses in the design and implementation of these modules. Also in the next section, the interfaces between the modules are examined placing emphasis on the "coupling" between the modules.

The metrics presented below operationally define a "complexity value" which attempts to measure the "degree of simplicity of relationships between subsystems" [16]. It should be kept in mind that the complexity of a given problem solution is not necessarily the same as the unmeasurable "complexity" of the problem being solved. In the following text the terms "complexity value," "complexity of the problem solution," and "complexity" are all used interchangably.

The terms fan-in, fan-out, complexity, and module are specifically defined for information flow in order to present the measurements in this section. Fan-in and fan-out are described with respect to individual procedures.

Definition 5: The *fan-in* of procedure A is the number of local flows into procedure A plus the number of data structures from which procedure A retrieves information.

Definition 6: The *fan-out* of procedure A is the number of local flows from procedure A plus the number of data structures which procedure A updates.

The complexity of a procedure depends on two factors: the complexity of the procedure code and the complexity of the procedure's connections to its environment. A very simple length measure was used as an index of procedure code complexity. The length of a procedure was defined as the number of lines of text in the source code for the procedure. This measure includes imbedded comments but does not include comments preceeding the procedure statement. Because of the spare use of imbedded comments and the typical occurrence of only one statement per line, this length measure is essentially equivalent to counting the number of statements in the procedure. Simple length measures, such as the number of source statements, are known to have positive correlations, albeit not the strongest correlations, with the occurrence of errors [8]. For example, Halstead's length or McCabe's cyclomatic complexity have shown stronger correlations to errors and programming time than the simple length. We have chosen to use the simplest possible procedure measurement for convenience. Furthermore, we speculate that using any of the other procedure measurements, such as Halstead's or McCabe's, should only improve the accuracy of the complete information flow complexity measure given below. The connections of a procedure to its environment are determined by the fan-in and the fan-out. The formula defining the complexity value of a procedure is

$$\text{length} * (\text{fan-in} * \text{fan-out}) ** 2.$$

The term fan-in * fan-out represents the total possible number of combinations of an input source to an output destination. The weighting of the fan-in and fan-out component is based on the belief that the complexity is more than linear in terms of the connections which a procedure has to its environment. The power of two used in this weighting is the same as Brook's law of programmer interaction [9] and Belady's formula for system partitioning [6]. These interrelationships are generally very complicated for operating and other large-scale systems. The validity of the approach used in this paper, emphasizing the connections to the environment, is supported by the material in this section and the following one showing a correlation between this measurement and system changes.

It is important again to note that the information flow analysis may be performed immediately after the design phase when the external specifications have been completed but before the implementation has begun. The measurements taken at this point rely only on: 1) sufficient information to generate the information flow relations shown above, and 2) estimates of the code length in order to compute the complexity measures defined next. As we will see in the next section, the code length is only a weak factor in the complexity measure. Thus, a precise estimate of length is not necessary and, in fact, this factor may be omitted without significant loss of accuracy. Based on these measures the design can be evaluated for possible flaws before the investment in implementation has begun. This permits a design-measure-redesign cycle which is considerably shorter and less expensive than the more common design-implement-test-redesign cycle. The measurements may also be taken after the implementation phase using the exact code length for each procedure. The measurements taken at either point will show possible areas where redesign or reimplementation is needed, and where maintenance of the system might be difficult. These measurements allow the design and implementation decisions to be evaluated for potential reliability problems.

The UNIX procedures and their associated complexities have been computed [19]. Procedures written in assembly language and certain "memoryless" procedures were eliminated from the information flow analysis. A memoryless procedure is one "which is guaranteed to have kept no record of data supplied after it has completed its task" [14]. Memoryless procedures are incapable of communicating information across successive invocations and, hence, serve to terminate an information flow path. The set of local flows from all of the data structures was used to compute the procedure complexities. Later in this section the complexity of a procedure contained in a specific module is computed using only the local flows for the data structure associated with that module.

The procedure complexities varying between 4 and 27 432 000 represent a broad spectrum of complexities. Readers unfamiliar with the UNIX system should find this wide range believable—UNIX contains some very simple and easy to understand procedures and some extremely difficult to understand procedures. A distribution of the order of complexity of UNIX procedures is given in Fig. 5.

The procedure complexities reveal three potential problem areas in a given procedure. First, the measurements show procedures which possibly lack functionality [25]. A high fan-in and fan-out reveals a large number of connections of that procedure to its environment indicating the procedure may perform more than one function. Second, a high complexity shows stress points in a system (i.e., a procedure with high information traffic through it). At such a stress point it is difficult to implement a change to the specific procedure because of the large number of potential effects on its environment and, indirectly, on other procedures. The third area indicated by these measurements is that of inadequate refinement. The inadequate refinement could be caused by either a problem in implementation or design. An implementation difficulty would be indicated by a large procedure, i.e., many lines of code. Perhaps the procedure should be divided into two or more separate procedures. The inadequate refinement could also appear as a missing level of abstraction in the design process. This would be indicated by a large fan-in or fan-out.

The procedure complexities are used, in turn, to establish module complexities. The operational definition of module adopted for use in this paper reflects Parnas' theory of hiding design decisions within a module [30].

Order of Procedure Complexity	Number of UNIX Procedure
10**0	17
10**1	38
10**2	41
10**3	27
10**4	26
10**5	12
10**6	3
10**7	1

```
procedure complexity = length * (fan-in * fan-out)**2
```

Fig. 5. Distribution of procedure complexities.

Definition 7: A *module* with respect to a data structure *D* consists of those procedures which either directly update *D* or directly retrieve information from *D*.

The information flow analysis was performed for each data structure in the UNIX operating system. The modules in UNIX simply consist of those procedures which read from or write to a data structure. A module is named by the data structure which it contains. Only a subset of the UNIX modules is presented in this section, since the other modules are relatively trivial. Fig. 6 displays the modules discussed in this paper and their descriptions.

Examination of the global flows in each module reveals the number of procedures in each module and all possible interconnections between the module procedures and the data structure. Fig. 7 displays the number of read only procedures, write only procedures, and read_write procedures for each module. Another interesting measurement easily calculated for a given module is the number of paths of information possible among these procedures through the module's data structure. The formula used to calculate the number of global flows is

$$(\text{write} * \text{read}) + (\text{write} * \text{read\_write}) + (\text{read\_write} * \text{read}) + (\text{read\_write} * (\text{read\_write} - 1)).$$

This shows all possible flows of information from those procedures which can update the data structure (write and read_write) to all procedures which retrieve information from the data structure (read and read_write).

The global flows measurement will indicate overloaded data structures. Fig. 7 shows that the *U* data structure has 3303 global flows with 84 procedures. Clearly, the *U* data structure, as compared to all other data structures, is extremely overloaded. The primary reason for this is that one function of this structure is to pass error codes across levels in the system. Since over half of the procedures have access to the *U* data structure, the module is so complex that is distorts other measurements. Accordingly, the *U* structure will not be given further consideration in this analysis. UNIX is not the only operating system with a complex, overloaded data structure. IBM OS/360 has a large global data structure called the communications vector table. The inability to control access to this table has led to a number of reliability and adaptability problems [25].

The complexity of a module is defined to be the sum of the complexities of the procedures within the module. It is interesting to note that the majority of a module's complexity is

MODULE	DESCRIPTION
buf	buffer information for the block I/O systems
file	information for each open file
filesys	super block information for disk allocation
inode	active file information for the general disposition of the file
kl11	status information for each terminal
lp11	device information for the line printer
mount	super block information for mounted files
proc	process information for each active process
text	segment information for unaltered code and data
u	process information not needed for swapping

Fig. 6. UNIX module descriptions.

Module Name	Global Flows	Number of Procedures in Module	Procedures (Read, Write, Read_Write)
buf	226	23	5,10,8
file	36	10	3,4,3
filesys	46	11	1,6,4
inode	448	28	1,16,11
kl11	106	17	7,6,4
lp11	8	5	2,2,1
mount	11	6	3,2,1
proc	425	30	7,11,12
text	11	6	2,3,1
u	3303	84	13,37,34

Fig. 7. Global flows for UNIX modules.

Module Name	Procedure Complexity	Procedure Complexity of Three Largest Procedures	Percent
buf	3541083	3468024	98
file	33062	29425	89
filesys	268807	254080	95
inode	13462921	12984995	96
kl11	3262	2120	65
lp11	855	829	97
mount	135503	135084	99
proc	436151	379693	87
text	24886	24831	99

Fig. 8. Percent of module complexity for largest procedures.

due to a few very complex procedures. Fig. 8 shows the modules, their complexity, the sum of the three largest procedures' complexities, and the percentage of that sum to the module complexity. In all but one case the three most complex procedures constitute more than 85 percent of the module complexity.

The global flows and the module complexities show four areas of potential design or implementation difficulties for the module. First, as with the *U* structure the global flows indicates a poorly refined (i.e., overloaded) data structure. Redesign of the data structure to segment it into several pieces may be a solution to this overloading. Second, the module complexities indicate improper modularization. It is desirable that a procedure be in one and only one module and, as a by-product of computing the module complexities, violations of this property will be revealed. This is particularity important when implementation languages are used which do not contain a module construct and violations of the module property are not enforcable at compile time. High global flows and a low or average module complexity indicate a third area of difficulity, namely, poor internal module construction. Numerous procedures have direct access to the data structure but there is little communication among the procedures. Fourth, a low global flows and high module complexity may reveal either a poor functional decomposition within the module or a complicated interface with other modules.

In this section we have presented the definitions of procedure and module complexity based on the connections established by an information flow analysis. Several examples have been given illustrating how these complexity measurements can be interpreted to reveal design or implementation problems. Finally, the measurements of a specific operating system were shown. In the next section we will examine the relationship between these complexity measurements and changes in the UNIX system.

IV. Correlation of Information Flow Complexity Measurements to Changes

The correlation between the complexity measurements and a collection of changes in the UNIX system is investigated in order to further validate the set of metrics presented in the previous section. These changes, 80 of which involved procedures used in the information flow analysis, were obtained by the authors from the UNIX users group [15]. These data will be used to determine the ability of the complexity measurement to predict procedures which, with a high probability, will contain needed alterations. The "program changes" measure of software quality has been used in previous studies [5], [13]. Basili and Reiter [5] refer to program changes as "... a reasonable measure of the relative number of programming errors ..." and cite the prior work of Dunsmore and Gannon [13] showing a high (rank order) correlation between program changes and error occurrences. Because of this relationship between program changes and errors we will use these two concepts interchangeably in this section.

One of the design features observed by the module measurements was that of improper modularization, i.e., those procedures located in more than one module. It is the goal of modularization to ensure each procedure in one and only one module [30]. It is expected that procedures which violate this principle should be more prone to errors due to their connections to more than one module. In the UNIX operating system there are 53 procedures in more than one module and 38 of these were in the list of procedures containing changes. Fig. 9 displays the distribution of UNIX procedures in zero or one module and those contained in more than one module, the number of changes associated with these procedures, and the percentage of procedures to be changed. A procedure may, according to our operational definition, be in no module because it does not directly access any data structure. The conclusion from this figure is simply that procedures which violate the modularity principle are more likely to require alteration. This conclusion is not at all surprising and documents the validity of the principles advanced under the title of abstract data typing.

The primary validation effort involved a correlation of procedure complexity with the occurance of changes in the UNIX code. In ranking the procedures according to their order of complexity we discovered that there is a strong correspondence between the changes to UNIX and high procedure complexity. Eleven out of the twelve procedures with complexity 10**5, and two out of the three procedures with complexity 10**6 required changes. The single procedure with complexity 10**7 did not require a change. However,

	Procedures in more than one module	Procedures in one or less modules
Number of procedures	53	112
Number of changes	38	42
Percent	72	38

Fig. 9. Relationship of module violations to changes.

Order of Complexity	Number of Procedures	Number of Procedures with Changes	Percentage
0	17	2	12
1	38	12	32
2	41	19	46
3	27	19	70
4	26	15	58
5	12	11	92
6	3	2	67
7	1	0	0

Fig. 10. Relationship of procedure complexity to changes.

this procedure, NAMEI, has connections to the system which are very complex and NAMEI would be extremely difficult to change [22]. Fig. 10 shows the number of procedures for each order of complexity, the number of procedures with corrections, and the percentage of those procedures containing changes.

Levels 6 and 7 only involve four procedures and because of this small sample size, these levels will be eliminated from the following analysis. The Spearman's r test [7] was used to correlate the order of complexity with the percentage of procedures containing an error. The resulting correlation coefficient is 0.94. The significance level of this correlation is 0.0214 indicating that there is only a 2 percent change of arriving at this correlation through mere coincidence in the data. The significance levels for the correlation coefficients reported in the rest of this section are similar to this 2 percent level. If the procedures at level 6 and level 7 are included, by combining them with the procedures at level 5, the correlation coefficient remains at 0.94.

Additional analysis was performed to determine which of the factors in the complexity formula contributed the most to this high correlation. The factors considered were length, (length**2), (fan-in * fan-out), and (fan-in * fan-out) **2.

The length of the UNIX procedures, measured in lines of source code, varied between 3 and 180. Using intervals of 20 lines of code, the distribution of procedures over those intervals showed that 53 percent of the UNIX procedures contain less than 20 lines of code, and that only 28 percent of those procedures contain an error, while 78 percent of the procedures with more than 20 lines of code contained errors. Due to the density of this distribution of length it was not possible to obtain a meaningful correlation coefficient using the Spearman r method. However, since most of the UNIX procedures are quite small we did not expect length to be a significant factor in the correlation. This expectation was confirmed by a subsequent test.

The second factor analyzed was (fan-in * fan-out). Again, using Spearman's r correlation, a coefficient of 0.83 was obtained. Thus, fan-in * fan-out does contribute significantly to the complexity analysis performed above.

The third factor considered was (fan-in * fan-out)**2. The

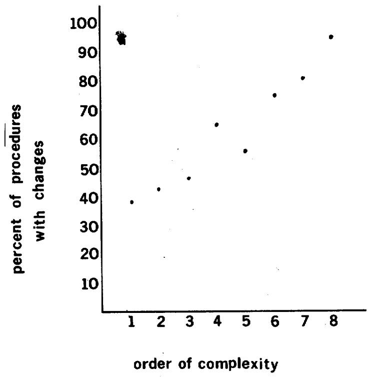


Fig. 11. Graph of (fan-in * fan-out) **2 correlation.

intervals used in analyzing this factor were 10**(n/2). Fig. 11 displays a distribution of the eight intervals used. The Spearman correlation coefficient derived was 0.98.

The final factor considered was (length**2). The intervals used for this calculation were 20 lines of code. As anticipated, a low correlation coefficient, only 0.60, was obtained. This result is significantly lower than the correlation coefficients of those measures which incorporated aspects of the information flow.

After considering the above factors, we find that length actually detracts from the predictive accuracy of the complexity formula indicating that at least as far as UNIX is concerned length is not a reliable indicator of procedure complexity. However, the connections of a procedure to its environment, namely (fan-in * fan-out) **2, is an extremely good indicator of complexity. The correlation coefficients and the level of significance for each coefficient are summarized in Fig. 15 in the final section.

V. Interface Measurements

In the previous sections we have concentrated on the measurement of procedure and module complexity and how to interpret these measurements in locating possible design and implementation problems. In this section we present a set of measurements which focus on the interfaces which connect system components. The interface between modules is important because it allows the system components to be distinguished and also serves to connect the components of the system together.

An often used design goal is to minimize the connections among the modules [27]. Myers has defined six categories of coupling based on the data relationships among the modules [25]. The information flow metrics can recognize two of these categories, namely those modules that are termed "content" coupled and those that are termed "common" coupled. Content coupling refers to a direct reference between the modules. This type of coupling is equivalent to the direct local flows. Common coupling refers to the sharing of a global data structure and this is equivalent to the global flows measure.

The connections between two modules is a function of the number of procedures involved in exporting and importing information between the modules, and the number of paths used to transmit this information. A simple way to measure the strength of the connections from module A to module B is (the number of procedures exporting information from module A + the number of procedures importing information into module B) $*$ the number of information paths.

Occasionally, the information flowing from A to B will pass through another procedure (see Fig. 12). These additional procedures between A and B are not in any module and their only purpose is to transfer information from A to B perhaps in an altered form (e.g., changing the scale of a value passed from A to B). In order to measure the coupling from module A to module B, the following three factors must be taken into consideration: 1) the direct flow of information from module A to module B, 2) the flow of information from module A to the transfer procedures, and 3) the flow of information from the transfer procedures to module B. The coupling measurements are derived by applying the above mentioned formula to these three factors.

Fig. 13 displays these measurements for the UNIX modules. In this figure the coupling through direct flows is given first and the coupling through the transfer procedures is in parenthesis. Note that the coupling is not symmetrical. INODE is tightly coupled with BUF, but BUF is only loosely coupled with INODE.

The coupling measurements show the strength of the connections between two modules. It can be observed that the coupling measurements for the UNIX modules reveals very strong connections between INODE and BUF, FILE, FILESYS, and PROC. This strong coupling of INODE to the rest of the system indicates that a substantial change to the INODE module would strongly affect other system components. This measurement agrees with the fact that the INODE module plays a key role in the UNIX system. Coupling also indicates a measure of modifiability. If modifications are made to a particular module, the coupling indicates which other modules are affected and how strongly the other modules are connected. These measurements are useful during the design phase of a system to indicate which modules communicate with which other modules and the strength of that communication. During implementation or maintenance, the coupling measurement is a tool to indicate what effect modifying a module will have on the other components of a system.

We believe that the interface measurements presented in this section are significant and further demonstrate the wide range of measurements which can be derived from an information flow basis. However, we have found no satisfactory way to thoroughly validate these measurements using the UNIX data we currently possess. Additional experiments, using other large-scale systems as vehicles, will be necessary in order to assess the utility of the interface measurements.

VI. Summary

The procedure, module, and interface measurements presented in this paper reveal potential design and implementation difficulties. Fig. 14 summarizes the measurements together with the particular design and implemention features associated with the measurements.

In addition to locating possible design and implementation weaknesses, the complexity measurements obtained through

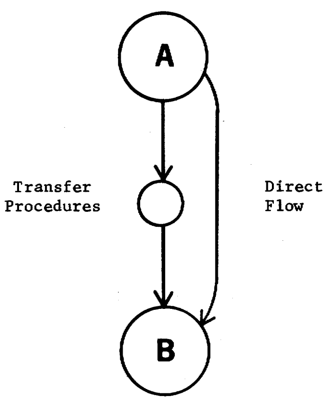


Fig. 12. Description of the interface between two modules.

an information flow analysis reveal a high correlation to actual changes for the UNIX operating system. The correlations reported in this paper are summarized in Fig. 15.

In conclusion, we believe that the measurement of software quality for large-scale systems using information flow to represent the system interconnectivity is an important and viable technique. This paper represents only the beginning of the work that should be undertaken to explore this approach.

References

[1] C. Alexander, *Notes on the Synthesis of Form.* Cambridge, MA: Harvard Univ. Press, 1964.
[2] F. E. Allen, "Interprocedural analysis and the information derived by it," in *Lecture Notes in Computer Science*, vol. 23. Wildbad, Germany: Springer, 1974, pp. 291-321.
[3] F. T. Baker, "Chief programmer team management of production programming," *IBM Syst. J.*, vol. 11, pp. 56-73, 1972.
[4] V. R. Basili and A. J. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 390-396, July 1975.
[5] V. R. Basili and R. W. Reiter, Jr., "Evaluating automatable measures of software development," in *Proc. Workshop Quantitative Software Models*, Oct. 1979, pp. 107-116.
[6] L. A. Belady and C. J. Evangelisti, "System partitioning and its measure," IBM Res. Rep. RC7560, 1979.
[7] H. M. Blalock, *Social Statistics.* New York: McGraw-Hill, 1960.
[8] J. B. Bowen, "Are current approaches sufficient for measuring software quality?," in *Proc. ACM Software Quality Assurance Workshop*, vol. 3, no. 5, 1978, pp. 148-155.
[9] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1975.
[10] R. N. Channon, "On a measure of program structure," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, Nov. 1974.
[11] E. W. Dijkstra, "The structure of the T.H.E. multiprogramming system," *Commun. Ass. Comput. Mach.*, vol. 11, no. 5, pp. 341-346, 1968.
[12] R. C. Dolotta *et al.*, "The programmer's workbench," *Bell Syst. Tech. J.*, vol. 57, no. 6, pp. 2177-2200, 1978.
[13] H. E. Dunsmore and J. D. Gannon, "Experimental investigation of programming complexity," in *Proc. ACM/NBS 16th Annu. Tech. Symp. Syst. Software*, Washington, DC, June 1977, pp. 117-125.
[14] J. S. Fenton, "Memoryless subsystems," *Comput. J.*, vol. 17, no. 2, pp. 143-147, 1974.
[15] M. Ferentz, Rockefeller Univ., private correspondence, 1979.
[16] T. Gilb, *Software Metrics.* Cambridge, MA: Winthrop, 1977.
[17] M. H. Halstead, *Elements of Software Science.* New York: Elsevier, 1977.
[18] M. Hecht, *Flow Analysis of Computer Programs.* New York: North-Holland, 1978.
[19] S. M. Henry, "Information flow metrics for the evaluation of operating systems' structure," Ph.D. dissertation, Iowa State Univ., Ames, IA, 1979.
[20] P. A. Janson, "Using type-extension to organize virtual memory mechanisms," IBM Res. Rep. RZ858, 1977.

	buf	file	filesys	inode	kl11	lp11	mount	proc	text
buf		2(0)	24(0)	12(0)	0(4)	0(4)	4(0)	6(4)	0(0)
file	0(15)		0(18)	90(68)	0(28)	0(8)	0(0)	12(15)	0(0)
filesys	49(3)	2(14)		35(32)	0(14)	0(8)	0(0)	12(6)	0(0)
inode	323(20)	18(20)	88(20)		0(39)	0(10)	4(0)	56(45)	6(12)
kl11	0(0)	0(0)	0(0)	0(0)		0(0)	0(0)	6(0)	0(0)
lp11	0(0)	0(0)	0(0)	0(0)	0(0)		0(0)	0(0)	0(0)
mount	42(0)	0(0)	42(8)	48(0)	0(0)	0(0)		6(2)	0(0)
proc	8(3)	2(2)	0(3)	12(4)	2(14)	0(8)	0(3)		0(2)
text	0(0)	0(0)	0(0)	0(0)	0(2)	0(2)	0(0)	0(0)	

Fig. 13. Coupling through direct flows (coupling through transfer procedures).

Measurement	Features
Procedure measurements	1. lack of functionality
	2. stress points in the system
	3. inadequate refinement
Module measurements	1. poorly designed data structures
	2. improper modularization
	3. poor module design
	4. poor functional decomposition
Interface measurements	1. strength of the coupling between modules
	2. measure of modifiability

Fig. 14. Summary of the measurements and corresponding features.

Measure	Correlation to Changes	Level of Significance
(fan-in * fan-out)**2	.98	.028
length*(fan-in * fan-out)**2	.94	.021
(fan-in * fan-out)	.83	.042
(length**2)	.60	.078

Fig. 15. Summary of correlation coefficients.

[21] E. L. Ivie, "The programmer's workbench–A machine for software development," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 746–753, Oct. 1977.

[22] J. Lions, "A commentary on the UNIX operating system," Dep. Comput. Sci., Univ. of New South Wales, 1977.

[23] B. Liskov, "The design of the Venus operating system," *Commun. Ass. Comput. Mach.*, vol. 15, no. 3, pp. 144–149, 1972.

[24] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, Dec. 1976.

[25] G. J. Myers, *Software Reliability Principles and Practices.* New York: Wiley-Interscience, 1976.

[26] —, *The Art of Software Testing.* New York: Wiley-Interscience, 1979.

[27] D. L. Parnas, "Information distribution aspects of design methodologies," in *Proc. IFIP 1971*, New York: North-Holland, pp. 339–344.

[28] —, "On the design and development of program families," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 1–9, Jan. 1976.

[29] —, "Some hypothesis about the 'USES' hierarchy for operating systems," Tech. Hochschule Darmstadt, Fachbereich Inform., Darmstadr, West Germany, Res. Rep. BSI 76/1, 1976.

[30] —, "Use of abstract interfaces in the development of software for embedded computer systems," Naval Res. Lab., Washington, DC, NRL Rep. 8047, 1977.

[31] C. V. Ramamoorthy and R. T. Yeh, *Tutorial: Software Methodology.* New York: IEEE Computer Society, 1978.

[32] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 365–375, July 1974.

[33] D. Schuster, "On the specification and quantification of software performance objectives," in *Proc. ACM 1977 Annu. Conf.*, Oct. 1977, pp. 181–188.

[34] N. L. Soong, "A program stability measure," in *Proc. ACM 1977 Annu. Conf.*, Oct. 1977, pp. 163–173.

[35] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 2, pp. 115–139, 1974.

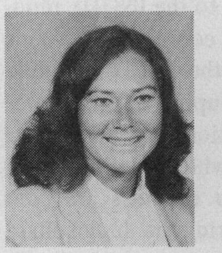
[36] T. A. Thayer, M. Liplow, and E. C. Nelson, *Software Reliability.* New York: North-Holland, 1978.

[37] E. Yourdon, *Techniques of Program Structure and Design.* Englewood Cliffs, NJ: Prentice-Hall, 1975.

[38] B. H. Yin and J. W. Winchester, "The establish and use of measures to evaluate the quality of software designs," in *Proc. ACM Software Quality Assurance Workshop*, vol. 3, no. 5, 1978, pp. 45–52.

[39] E. Yourdon, *Techniques of Program Structure and Design.* Englewood Cliffs, NJ: Prentice-Hall, 1975.

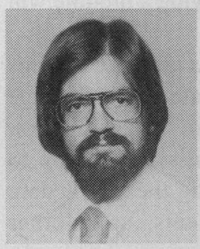
[40] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, pp. 1221–1241, Dec. 1976.



Sallie Henry (M'81) received the B.S. degree in mathematics from the University of Wisconsin, LaCrosse, in 1972 and the M.S. and Ph.D. degrees in computer science from Iowa State University, Ames, in 1977 and 1979, respectively.

She is currently an Assistant Professor of Computer Science at the University of Wisconsin, LaCrosse. Her current research interest is the quantitative measurement of the structure and quality of large-scale systems.

Dr. Henry is a member of the Association for Computing Machinery, SIGOPS, and SIGSOFT.



Dennis Kafura received the M.S. and Ph.D. degrees in computer science from Purdue University, West Lafayette, IN, in 1972 and 1974, respectively.

He is currently an Associate Professor of Computer Science at Iowa State University, Ames. His recent research, involving the quantitative evaluation of software structure, has been reported in the 1981 SIGMETRICS Symposium on Software Quality. Other papers dealing with operating system performance analysis have previously appeared in *Performance Analysis*, *Journal of the ACM*, and *SIAM Journal on Computing*.