An information theory based complexity measure

by ELI BERLINGER
Nassau Community College
Garden City, New York

INTRODUCTION

There have been numerous measures proposed to measure program complexity. Some are completely heuristic, comparing certain measurable program features against a set of predefined standards. Some are topological, based on the number of regions on the control or data graph of the programs or a combination of the above, and of course, there is Halstead’s Software Physics.

All of these measures have their deficiencies and, no doubt, so will ours. We have, however, set ourselves the goal of eliminating some of them and to provide a measure which has mathematical and intuitive correctness and which will have a good correlation with observed facts.

Goals

A complexity measure based on all elements of the program. Everything can produce an error, even a simple assignment statement or MOVE instruction. They should all have some effect on the measure. This requirement is not met in the topological measures, for instance.

To satisfy both intuitive desires and psychological realities, we want the measure to be more sensitive to infrequently used elements of the language being used than to commonly used ones. It is reasonable to suppose one can do more accurately what one does more frequently.

A particular element of a language used in a complex manner should contribute more to a complexity measure than the same element used in a simpler manner. For example, a nested loop would seem to be more complex than a sequence of two simple loops. Few of the measures seen to date have adequately addressed this situation.

The measure should allow for automated techniques in its calculation.

These are our primary goals. As will be seen later, additional advantages will follow as a direct result of the measure definition.

STATE OF THE ART

A number of complexity measures have been proposed. A review of some of them will be presented in this section, together with a brief critique of some of their strengths and weaknesses. As will be noted, a number of the measures seem to be completely heuristic with no apparent mathematical justification.

Hellerman proposed a measure having some relation to information theory. His measure is based entirely on the number of inputs and outputs. Let $X$ be the domain of inputs and $Y$ be the range of outputs. Each element, $Y_i$, of the range is the output of a class of inputs, $X_i$. The $X_i$ form a mutually exclusive set of $N$ classes. If we conceive of a program as reducible to a table lookup, let $|X_i|$ be the number of elements of $X_i$, and $|X|$ be the number of elements of $X$. Then each $Y_i$ appears $|X_i|$ times. The probability of its occurrence is $P(Y_i)=|X_i|/|X|$ and its information content is $-\log_2|X_i|/|X|$. The total information content, $w(f)$ which Hellerman calls the “computational work” is $w(f)=\sum_{i=1}^{N} |X_i| \log_2|X_i|/|X| = |X| H(P_1, \ldots, P_N)$ where $H$ is the entropy function.

There are several objections to the use of Hellerman’s measure. Two will be given.

First, it is necessary to know precisely how many input values yield a specific output value. In most programs, the domain input values is an enormous set, as is the range of output values. In all likelihood, neither domain, nor range, will ever be exhausted.

Second, even assuming all elements of the domain can be used, we would require the computer to evaluate all elements of the range to determine the domain classes. This means that every input value has to be used to obtain the output values, which will determine the domain classes, which can then yield the measure. By this time, the program is completely debugged and delivered and we no longer need the measure.

McCabe proposed a measure which is easily obtained at an early stage. Known as the “cyclomatic number,” it is based on the number of linearly independent paths through the control graph of the program. Let $N$ be the number of nodes, $E$ the number of edges, and $P$ the number of connected components. Then the cyclomatic number, $V$, is $V=E-N+2P$.

McCabe derives a number of simplifications for the calculation of $V$. Using Euler’s formula, $N-E+R=2$, where $R$ is the number of regions of a plane control graph, we get $R=E-N+2$. Therefore, if there is only one connected com-
ponent, \( R = V \). In words, \( V \) is the number of regions on a plane control graph, counting the external region.

Using a result of Mills,\(^3\) McCabe achieves another simplification. If \( \pi \) is the number of predicates in the program, then \( V = \pi + 1 \). Therefore, to calculate \( V \), one need only count the number of binary conditions contained in IF’s and DO’s to calculate \( V \).

Myers\(^4\) proposes a modification to McCabe’s cyclomatic measure. He shows that difficulties arise if one calculates \( V \) by counting the number of predicates (conditions) and that other difficulties arise if one counts only the number of decision statements, IF’s and DO’s. He proposes to make \( V \) an interval whose lower bound is the number of decisions + 1, and whose upper bound is the number of predicates + 1. By using an interval, Myers removes the question of whether or not the statements

\[
\text{IF (A & B) THEN ... ELSE ... ; are more complex than \ IF A THEN IF B THEN ...; ELSE ...; ELSE :}
\]

The cyclomatic measure is a good one. It is easily obtained and comparisons between programs are possible. However, two programs consisting only of sequences will both have \( V = 1 \) regardless of the number of statements or variables contained in the program. The measure is sensitive only to branching or looping. Nothing else which might contribute to the complexity of a program is considered. Some purely sequential programs can be quite complex, however, and their measures should reflect this.

A much more serious problem appears to exist. A sequence of IF-THEN-ELSE’s has a higher cyclomatic number than a nest. Consider the following two diagrams:

![Diagrams A and B]

Both control graphs have a cyclomatic measure of 4, yet, intuitively, a nest should be more complex than a sequence since several predicates are operative simultaneously.

McTap\(^5\) proposes a measure which compares certain measurable features of a program to a set of predetermined standard values. The greater the number of features which deviate from the norm on the side of complexity, the larger the measure. The measure is calculated in two phases, as follows:

### Phase 1

1. Select those features, which are to be measured. The features must be “normalized” so that a given value means the same thing in all programs or modules. For example, the number of decisions is not a satisfactory feature, while the proportion of decisions, compared to all statements, is.

2. Select, arbitrarily or by experience, a reference level for each feature.

3. Form reference vector, \( R \), comprised of these levels.

4. Form a direction vector, \( D \), where \( d_i = +1 \) means the feature contributes to simplicity and \( d_i = -1 \) means it adds to complexity. +1 and −1 are the only values used in \( D \).

5. Form a vector, \( W \), of weights for each feature.

### Phase 2

1. Form feature vector, \( F \), by studying the documentation and measuring each feature.

2. Form the magnitude vector \( M = F - R \). Here, if the feature exceeds the reference level, \( m_i > 0 \), else \( m_i = 0 \).

3. Form product vector \( P = M * D \). If \( P_i > 0 \), the \( i \)-th feature in the program contributes less complexity than the reference level. If \( P_i < 0 \), it contributes to greater complexity.

4. Form change vector, \( C \):

\[
C_i = \begin{cases} 0 & \text{if } P_i \geq 0 \\ -P_i & \text{if } P_i < 0 \end{cases}
\]

That is, \( C_i = 1 \) for each feature which contributes more complexity than the reference level.

5. Form the score vector, \( S = C * W \). If particular feature of the program is more complex than the reference value, the corresponding score vector element is the weight assigned to that feature. If not, the element is zero.

6. Form the complexity measure \( M = S_1 \), that is, the measure is the sum of the elements of the score vector.

From a theoretical viewpoint, there is absolutely no mathematical correspondence between this measure and the number of bugs, or any other factor of importance. From a production viewpoint, this criticism is not important. If it has been experimentally determined that programs whose measure exceeds a certain value tend to be poorly written, then the measure can be used to detect such faulty programs so that they can be rewritten at an early stage of production. For that purpose, the measure appears suitable as a guide.

The measure will provide no correlation between the complexity measure and the number of bugs. The elements of the score vector are only two-valued. If the program feature is below the reference level, the corresponding element is zero. If above, it is equal to the weight. It would seem that the measure might be improved by redefining the change vector, \( C \) as follows:

\[
C_i = \begin{cases} 0 & \text{if } P_i \geq 0 \\ -P_i & \text{if } P_i < 0 \end{cases}
\]

The correlation between the measure and the number of bugs should now be considerably improved.

Chapin\(^6\) proposes a measure which is based on the role played by the variables in each module. He considers four
types of variables:

- \( P \) — input data needed for processing
- \( M \) — output data assigned a value
- \( C \) — data used as control (in loops, decisions, etc.)
- \( T \) — data which remains unchanged in the module (passes through)

An I-O table is constructed for each module on which is indicated the role of each I-O variable. (Here, an I-O variable is any non-local to the module.) The measure is calculated as follows:

For each module:
1. Count, in the I-O table, the number of items in \( C, P, \) or \( T \) roles (output).
2. Multiply each by a weighting factor; 3 for \( C, 2 \) for \( M, 1 \) for \( P, \frac{1}{2} \) for \( T \).
3. Sum the weighted counts.
4. Initialize \( E \) to 0 for all modules.
5. If a module serves in the exit test of an iteration then add to its \( E \) measure as follows: (a) if the control data item for the exit test comes from within this module, add 0 to \( E \); (b) if it comes from the loop body, add 1 to \( E \); (c) if it comes from outside the loop, add 2 to \( E \). For example, if the control data is initialized outside the loop and modified within the loop, add 3 to \( E \).
6. Convert \( E \) into a repetition factor, \( R \), by adding 1 to the square of \( E \): \( R = (E^2 + 1) \).
7. Multiply the sum of the weighted counts from step 3 by the modules respective \( R \).
8. Find the square root of the products from step 7. This is \( Q \), the index of module complexity.
9. Calculate the program \( Q \) by finding the mean of the module \( Q \).
10. Calculate the system \( Q \) by finding the mean of the component module \( Q \) within the component programs.

It is felt that this measure has a number of disadvantages:
1. While the measure is based on the set theoretic definition of a function, some of the specific calculations appear to be experimentally derived. For example, step 6 requires the square of one third of the \( E \) value. Why this particular value is used is not theoretically clear.
2. It requires a detailed I-O table or equivalent for each module specifying precisely how each variable is to be used in the module. This may be desirable in some cases. A clear idea of the meaning of each data item is certainly essential. However, it would require close supervision to ensure that each variable is properly identified.
3. The calculation of the program and system \( Q \) leaves much to be desired; the use of the mean has the effect of making large programs with many modules no more complex than an individual module. This does not seem to be desirable. As a minimum, the program and system \( Q \) should be some additive function of the module \( Q \).

Chen defines a measure on the topological properties of a graph. Given a "strongly connected proper flowchart," one which contains no dividing edges or bridges, nor weakly connected subparts, he defines the "maximal intersect number," \( MIN \), as the maximum number of edges which can be intersected by a continuous line drawn so as never to enter any region, including the external region, more than once.

If the graph of the flowchart is made up of serially connected subparts, the \( MIN = \sum_{i=1}^{n} MIN_{i} = (2 \times \text{number of subparts} + 2) \), for which he gives a proof.

The topological attribute \( MIN \) can be given analytically by the expression \( MIN = Z_{\infty} + 1 \) where

\[
Z_{\infty} = 1 + \sum_{i=1}^{n} \log_{2}(p_{i}X + q_{i}), \quad X = 2
\]

where \( n \) is the number of decision symbols on the flowchart or graph, \( q_{i} = \) probability that the \( i \) th \( IF \) symbol is forming a serial relation with any of its preceding and adjacent \( IF \) symbols, and \( p_{i} = 1 - q_{i} \).

For a given proper flowchart, \( q_{i} \) is either 1 or 0 depending on whether or not it is in the specified serial relationship. \( Z_{\infty} \) is called the "control structure entropy." The programmer is considered a source emitting \( IF \) symbols and \( Z_{\infty} \) is defined as the entropy of the "source" when it emits \( n \) \( IF \) symbols in a specified manner.

For a given total number of \( IF \) symbols, the programming job is to determine the exact flowchart structure. This will determine the \( p \) and \( q \) values. Since there are many different alternatives which are seemingly equal, we can, a priori, set \( p_{i} = q_{i} = \frac{1}{2} \). This gives \( Z_{\infty} = 1 + \sum_{i=1}^{n} \log_{2}(1 + \frac{1}{2}) = (n - 1) \log_{2}3 - n + 2 = Z_{\infty} \). Given the number of \( IF \) symbols a program is to have, \( Z_{\infty} \) can be calculated since it depends only upon \( n \).

Chen now derives a relationship between \( Z_{\infty} \) and \( r \), the productivity in source statements per busy hour. The relationship is heuristic, derived to fit his observed points and is

\[
r(Z_{\infty}) = \frac{1}{b/a + (c-b/a)e^{-a/c}n}
\]

where \( a = 0.3187, \quad b = 9.2451 \times 10^{-1} \) and \( 0.7695 \times 10^{-1} \leq c \leq 0.7714 \times 10^{-1} \).

For a given flowchart the measure \( Z_{\infty} \) has the property that any number of serially connected \( IF \) symbols do not contribute to complexity. Only if the \( IF \) symbols are "nested," will the complexity increase. While it is true that serially connected \( IF \) symbols seem to be inherently less complex than other connections, yet it seems to be an oversimplification to ignore them completely.

Using the "a priori" formula improves this situation somewhat. Here, each \( IF \) statement add \( \log_{2}3 \) to the entropy \( Z_{\infty} \).

It should be noted that Chen's maximal intercept number is quite similar to McCabe's measure when there is only one connected component.

Mohanty proposes the Entropy Loading measure based on the amount of information shared between modules or subsystems.

Assume \( A \) and \( B \) are two subsystems of \( S \) such that \( A \cap B = \emptyset, A \cup B = S \). A table is made with every subsystem
of A and B along one dimension and every system attribute along the other. An entry is 1 if a subsystem makes an assumption about the attribute, 0 otherwise. For each subsystem of S(A or B) we consider the submatrix consisting of attributes which have a nonzero entry for that subsystem. Applying Schutt's Entropy Metric to each submatrix, we get \( H(A) \) and \( H(B) \). If we calculate the Entropy Metric for the entire system \( S \), we get \( H(S) \). The Entropy Loading Measure is now defined as:

\[
C(S) = H(A) + H(B) - H(S).
\]

It is a measure of the information shared between the subsystems A and B.

This measure is useful at the system design stage where the shared attributes are known, or, at least, hinted at. It provides no help in determining the complexity of a given module or subsystem, which would, however, be desirable at a later stage in program development.

Schutt proposes the Entropy Metric. It is similar to Hellerman's Work Function is divided by \( |X| \), the number of input classes. The definition of the Entropy Metric is

\[
H = \sum \frac{|X|}{|X|} \log_2 \frac{|X|}{|X|}.
\]

Halstead has proposed a rather comprehensive theory in which he not only proposes a measure to correlate with the bugs in a program, he also attempts to predict this measure at the most primitive state of the programming when all that is known is the number of input and output variables.

Let \( \eta_1 \) = no. of operators
\( \eta_2 \) = no. of operands
\( N_1 \) = total usage of operators
\( N_2 \) = total usage of operands
and \( N = N_1 + N_2 \) be the total length of the program then \( N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \) and is called the program length.

While this relationship is interesting and Halstead demonstrates that it gives correct results for a number of programs, it is not the measure he uses. He defines program volume, \( V \), as

\[
V = N \log_2 \eta \quad \text{where} \quad \eta = \eta_1 + \eta_2
\]

The potential volume, \( V^* \), is the volume a program would have if a function existed which would solve the problem. There would be two operators; one an assignment statement, the other a call to the function. The potential volume is therefore

\[
V^* = (2 + \eta_1^*) \log_2 (2 + \eta_2^*)
\]

where \( \eta_1^* \) is the number of input/output variables, each used once in the function parameter list.

The program level, \( L \), is now defined as the ratio of the potential volume to the program volume:

\[
L = \frac{V^*}{V}
\]

The larger the volume of the program, the lower the level of its coding.

The program effort, \( E \), is defined to be proportional to the volume and inversely proportional to the level: \( E = V/L \).

Here it is assumed that for a given volume, it is more difficult to comprehend a program written at a low level than at a high one, perhaps because of increased complexity for each statement, and for two programs written at the same level (remember that the level is unitless, being a ratio of like quantities) it is more difficult to comprehend the "larger" one.

It is the effort, \( E \), which is used as the measure to correlate with the number of bugs in the program.

Halstead defines another measure, \( I \), called the information content, defined \( I = LV \). This measure, Halstead feels, correlates best with total programming and debugging time.

Halstead has given a comprehensive and, on the whole, a rather beautiful theory to explain program behavior. There is but one basic flow. He "derives" the volume equation, \( V = N \log_2 \eta \), as follows: Given \( \eta \) types, \( \log_2 \eta \) bits are required to uniquely identify them all. It requires \( N \log_2 \eta \) "mental discriminations" to select \( N \) symbols from a table of \( \eta \) symbols using a binary search technique since each search requires \( \log_2 \eta \) choices. Hence the volume.

This scheme seems to be based loosely on information theory, and this is where the measure fails from a theoretical viewpoint.

From information theory, we have the total information as \( I = - \sum \eta_i \log_2 P_i \), where \( P_i \) is the probability of the \( P \) type, and \( N_i \) is the number of times it is used. To get Halstead's volume, it would be necessary to assume that all types have equal probability, and that the types existent in a particular program constitute the universe of types, giving \( P_i = 1/N \). Then \( I = - \sum \eta_i \log_2 \eta = \sum \log_2 \eta = N \log_2 \eta \). These two assumptions are untenable. Therefore, the Halstead measure fails to have a theoretical basis and must be thought of as completely heuristic.

Sullivan reports a number of measures, some of which he rejects as either unworkable or not sufficiently studied, and others he proposes as measuring the quantities which contribute to complexity.

The C1 complexity at any node of an elementary scheme is the number of paths containing more than \( X \) contiguous repetitions of a sequence of modes, where \( X = 2 \) unless otherwise stipulated. The complexity of the entire scheme is defined as the local complexity of the terminal node. A problem with this measure is that the measure can be very large or even infinite, the difficulty being in the notion of a loop as the immediate repetition of the same sequence of steps. If there are intermediate steps, this definition allows for the repetition of the same loop virtually an infinite number of times.

The C2 complexity at any node is similar to the C1 measure but eliminates the difficulty mentioned above. It is defined as one less than the number of paths from the start node to the given, not counting paths where any node occurs more than \( X \) times, \( X = 2 \) unless otherwise stipulated. The complexity of the entire elementary scheme is defined as the
local complexity at the terminal node. The complexity of a composite scheme is defined as the sum of the complexities of its elementary subschemes. Compared to $C_1$, only the contiguity condition has been removed, but this removes the retracing of loops move than once for different sequences of paths. The subtraction of unity is to give unitary subschemes (sequences) a $C_2$ measure of 0 so that the sum of the measures for the elementary schemes of a scheme would not be sensitive to the incorporation of unitary decomposition or to decompositions along a single path.

The $C_2$ measure is always finite and if the program is structured, all of its elementary subschemes have measures of either 0 or 1. If the program is written using IF-THEN-ELSE for alternation and DO WHILE for iteration, then the $C_2$ measure is the number of IFs and WHILEs.

Sullivan feels that one weakness of the $C_2$ measure is that all paths are counted even when some paths do not contribute as much to complexity as others. He gives the example of an $N$-way case structure which is not necessarily $N$ times as complex as a simple IF-THEN structure.

Measure $C_3$ at any mode of an elementary scheme is defined as the first derivative of the $C_2$ measure with respect to $X$, the maximum no. of repetitions of any node. This eliminates the problem mentioned above with respect to the $N$-way CASE structure. The $C_3$ measure has not been studied in depth and Sullivan makes no further use of it.

Sullivan now uses another approach. The $C$ measures are defined on properties of the control graph but completely ignore data. He defines two new values, the $P$ measures, which describe complexity in terms of how the program interacts with the data.

Let the entire scheme be decomposed into subschemes and let the subschemes be modules (either actually or conceptually). Let the data subgraph proper to the module be that part of the data graph referenced from the nodes of the module or subscheme. If we now assume that any data element so referenced is relevant over any path through the subscheme, then it is reasonable to define measure $P_1$ as the product of the number of paths through the subscheme ($C_2 + 1$) and the number of data items (nodes of the subgraph) referenced.

As Sullivan himself states, the underlying assumption is not intuitively satisfying. Not all nodes of the subscheme reference every node of the data subgraph. Also, no distinction is made between set and use references.

Sullivan now changes perspective and examines each data item individually to see how it is used. For a given data graph nodes, define the process node set as all nodes which either set or use that data item, together with the start and terminal nodes. For nodes $n_i$, $n_j$ of the set, let edge $(n_i, n_j)$ exist if $n_j$ can be reached from $n_i$ without passing through another node, $n_k$, of the set. For this new control graph, the $PD_2$ data node complexity at this particular data node is the $C_2$ complexity of the control graph just defined except that any subscheme which has only use references and no set references in nodes other than its start node is assigned value 0. Sullivan has not determined how the $P_2$ measure should be defined from the $PD_2$ data node complexities. It may be possible to have $P_2$ be the sum of all the $PD_2$ measures but he suggests that some other function makes more sense.

The various measures Sullivan suggests may be correlated with the number of bugs, or difficulty, of a program. At the time the paper was published, the measures had not been tested. This writer is not aware that experimental justification exists. The $P_2$ measure seems a bit difficult to implement. For each variable, it is necessary to create a control graph from the nodes of the program or system control graph which use that variable. Where there are subscripted or qualified variables, the problem is compounded greatly since the data graph will have many nodes, each of which requires the creation of a control graph. From a practical viewpoint, the calculation of this measure may require a program of considerable complexity to measure it.

**BACKGROUND ON INFORMATION THEORY**

Suppose an alphabet of symbols is given $(e_1, e_2, \ldots, e_n)$ whose respective probabilities of occurrence are $(p_1, p_2, \ldots, p_n)$. Then the following quantities can be defined or proven.

Uncertainty and information

Where symbol $e_i$ has probability $p_i$, the uncertainty associated with the occurrence of symbol $e_i$ is $- \log p_i$. If the logarithm is base 2, the information is in bits. The uncertainty is a measure of the information provided by the symbol. Note that a symbol with low probability contains more information than one with high probability. Uncertainty and information are, for practical purposes synonymous in the present context.

Entropy

The entropy, $H$, is defined $H = - \sum p_i \log p_i$. It is the average information provided by the alphabet of symbols $(e_i)$. We will discuss entropy again.

Total message information

Let a measure be composed from an alphabet $(e_i)$ whose frequencies of occurrence are $(f_i)$ and probabilities are $(p_i)$. Since $e_i$ contains $-\log p_i$ bits of information, the total information contained in the message is

$$I = - \sum f_i \log p_i \text{ bits.}$$

This will form the basis of our complexity measure.

Ideal symbol length

To minimize expected message length, where a message is defined as a string of symbols, it can be shown that the
encoded symbol should have a length in bits given by
\[ l_i = -\log_2 p_i. \]
For example, if \( e_i \) has probability \( p_i = \frac{1}{4} \), and if we use base 2, then the encoded length of symbol \( e_i \) should be
\[ l_i = -\log_2 \frac{1}{4} = 2 \text{ bits}. \]

Entropy, \( H \)
Entropy of an alphabet is defined as
\[ H = -\sum p_i \log p_i, \]
summed over all \( i \). If \(-\log p_i\) represents the length of symbol \( e_i \), then the entropy represents the weighted average symbol length in an ideal coding scheme. The entropy will be used later in a complexity measure prediction formula.

Message length
Let a message have symbol \( e_i \), occur with a frequency \( f_i \), and let an ideal coding scheme be used so that symbol \( e_i \) has coded length \(-\log p_i\), then the coded length of the message will be
\[ L = -\sum f_i \log p_i, \]
where \( \sum f_i = N \) is the total number of symbols used in the message.

We see that the two calculations \(-\sum p_i \log p_i\), and \(-\sum f_i \log p_i\), can be interpreted in two ways. The first, entropy, is both the average information content supplied by the alphabet and the average length of the encoded alphabet where an ideal coding symbol is used. The second is both the total information contained in a message and the total message length if encoded with an ideal coding scheme.

These interpretations will be used again later when discussing the complexity measure.

ASSUMED PROGRAMMING ENVIRONMENT
We require an installation at which statistics have been gathered on all items used in programs written at the installation. Specifically, the following are known: (1) the percentage of time each operator is used; for some operators several distinct percentages are known; these include operators which have nesting levels such as DO’S, IF’S, parentheses, and so on; (2) for operands such as variables, constants, arrays, function names, etc. we rank their probabilities according to frequency of use; that is, the probability of the most frequently used taken first, the next most frequent second, etc.; separate sets of probabilities are kept for arrays of differing dimensions and, perhaps, of different data types or structures; (3) frequencies for labels are kept as for variables, ranked in order of usage.

The required frequencies can be automatically accumulated as programs run at the installation. Different probabilities are maintained for each language for which a complexity measure is required. A number of programs have been written, which can be easily incorporated into a system, that count frequencies for FORTRAN, PL/I, and perhaps some others.

A COMPLEXITY MEASURE
We now define a new complexity measure based on information theory.
Let a program be given. We count the frequency of occurrence of all tokens in the program: operands and operators. We assume that the long term probabilities of these are known and are represented by \( p_i \). The complexity measure is defined as:
\[ M = -\sum f_i \log p_i. \]

There are several interpretations we can assign to this measure. First, from an information theory viewpoint, the measure represents the total information contained in the program, assuming the program to be a message.

Second, if we were to code each token of the program using an ideal coding scheme, the measure would be the total length required to encode the program.

The measure as defined is sensitive to the frequency of usage of all symbols and to the proportion of times the symbol has been used in the past.

The \( p_i \) are an accumulated statistic gathered at the installation over a period of time. It is quite possible that a given program may have a different measure at different installations. This would reflect different programming techniques at these installations. There are both positive and negative aspects to such a situation.

On the positive side, if a program manager has a known correlation between this complexity measure and programming difficulty at his installation, he can predict how long it will take his programmers to finish the job. This would be of primary importance to him.

On the negative side, the manager has no way of knowing from the measure whether his programmers are working efficiently. If it is important to know this, and one would think it is, a comparison of the measure would have to be made with another installation’s. If another installation has a different set of probabilities, the measure can be recalculated using those probabilities. It is somewhat unclear, however, whether the results are meaningful.

A better way to measure programmer efficiency might be with the entropy measure to be defined later.

This measure may appear, to some, to be similar to Halstead’s program length measure. There is, however, a major difference. Halstead’s “probabilities” are calculated as \( f_i/n \) for a given program. The measure here uses long term probabilities obtained over a period of time. Halstead makes no distinction between programs using difficult constructs and those using simpler ones. The measure proposed here does.

ADVANTAGES
The measure, as defined, has the following advantages: (1) once the programming has been done, calculation of the measure can be completely automated; (2) the measure can
be estimated at an early stage to detect programs or modules which may be overly complex; (3) the measure is more sensitive to infrequently used tokens than more frequently used ones; this is intuitively satisfying as it is reasonable to suppose that people will tend to make more errors in using things with which they are less familiar; (4) most importantly, the measure makes distinctions between tokens which are used at different levels; the probability of a DO used as a second level is lower than the one used at the first level; the level two DO will contribute more to the measure than the level one; this also seems quite satisfying as it is reasonable to suppose that nesting is more complex than sequencing; very few of the measures proposed so far have considered this aspect of programming.

**ENTROPY**

We can define a language entropy as:

\[
H = - \sum p_i \log p_i
\]

A language with a high entropy has many tokens with low probabilities. One with low entropy has few tokens with high probabilities.

Note that a given language, such as PL/I, will have different entropies at different installations. At one place, programmers might write programs containing a large number of variables resulting low frequencies. The other installation might use a great deal of segmentation with the result that each module has only a few variable of high probability. The second installation would tend to write programs with lower complexity than the first.

The entropy might, therefore, be a measure of an installation's ability to reduce complexity through segmentation. It could also, perhaps, be used to compare with other installations. For example, if one installation used the various language features more uniformly than another, then the entropy would be larger at the installation. This might indicate a better knowledge of the language, perhaps.

**ESTIMATING THE COMPLEXITY MEASURE**

We have seen that the entropy, \( H \), is the average length of a program token and the complexity measure, \( M \), is the encoded length of the entire program.

Suppose \( \hat{N} \) is the expected number of tokens, operators or operands, in a program. Then we can define an estimated complexity measure, \( \hat{M} \), as

\[
\hat{M} = \hat{N}H
\]

The only problem is obtaining \( \hat{N} \). Some work is being done along these lines. For example, Professor Laemmel,\(^{12}\) at PINY, has been using a modified Zipf's law to obtain the number of tokens if the number of types is known. The number of types might be estimated at an early program state. Halstead, using software physics, attempts to estimate the number of tokens, which he calls the length, from the number of I-O parameters.

Whichever scheme is used, an estimate can be obtained.

**CONCLUSION**

A measure has been proposed which, it is to be hoped, correlates well with difficulty in programming. The measure has mathematical foundation in information theory and is intuitively satisfying as well. It can be easily found after some initial programs have been added to a system. Furthermore, a predictor for the measure is available.

The measure is based on the probability with which various tokens of a program are used. The probabilities are collected over a period of time and are cumulative.

The measure is installation dependent but comparisons between installations is possible by use of an accompanying measure. This other measure is also used in predicting to complexity measure.

**BIBLIOGRAPHY**

Software Engineering Education

Several studies have indicated that the data processing industry's most critical problem during the 1980's will be a shortage of qualified software engineers. This panel brings together a number of people who have been actively addressing the problem of increasing the supply of qualified software engineers through both University and industry programs in software engineering education. They will address the following issues:

• What skills will be needed by the software engineer of the future?
• What software engineering programs are currently underway in universities, industry, and professional societies to meet these needs?
• What particular approaches have been tried to date, and how well have they worked out?
• What are some resulting guidelines for mounting a successful software engineering education program?
Special Topics

In this session, an integral EDP auditor discusses his function and involvement with data processing and the users of data processing. He shares his views on a wide range of specific EDP areas of activity and involvement. He will relate his concerns to control objectives in the audit methodology utilized for determining that the control objectives are effectively assured.

Gene Smith
Area Director