Measuring Prime Program Complexity

MARVIN V. ZELKOWITZ

Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742

and

JIANHUI TIAN

IBM PRGS Toronto Lab. 895 Don Mills Road, North York, Ont. M2J 1M6, Canada

ABSTRACT

This paper uses the prime program decomposition of a program as the basis for a measure that closely correlates with our intuitive notion of program complexity. This measure is based upon the information theory ideas of randomness and entropy such that results about structured programming, data abstractions, and other programming paradigms can be stated in quantitative terms, and empirical means can be used to validate the assumptions used to develop the model. As a graph-based model, it can be applied to several graphical examples as extensions not otherwise available to source-code based models. This paper introduces the measure, derives several properties for it, and gives some simple examples to demonstrate that the measure is a plausible approximation of our notions concerning structured programming.

1. INTRODUCTION

This paper develops a program complexity metric based upon the information theoretic concepts of complexity and randomness to quantify programming concepts like structured programming and data abstractions. The graph-based prime program decomposition of a flow graph is used to develop these ideas.

Why do we need such a measure? Program complexity is of significant interest within software engineering. Numerous "truths" are afforded daily:

(a) Complex programs are larger to build, thus cost more.
(b) Complex programs are more unreliable.
(c) Data abstraction, object-oriented design, and structured programming are all techniques which lower program complexity, and hence improve development cost and performance of such software.

All of these have serious economic consequences; however, anecdotal data are only loosely correlated. An effective measure could tie together many of these concepts. In addition, without quality metrics, ad hoc solutions may be imposed which are not optimal. For example, since size is often used as a measure, several guidelines limit module size to 100 lines of code or a page of text. However, at least one study has shown that such artificial limits to module size or complexity are suboptimal with respect to error rates [5]. What is clearly needed is an effective measure for such complexity.

We would also like to predict system size and cost—factors crucial to any engineering discipline. Source code measures are generally ineffective since the numbers are only available once the project is completed. We would like to base our measurement notion on a graphical structure to be able to develop quantitative results about the specifications and design phases of development [16].

Our notion of complexity comes from three sources: (1) We base our definition of complexity on the measures of information content and randomness that grew out of information theory. (2) This research builds upon the work of Halstead, McCabe, Basili, Boehm, and others who have studied related measures. (3) The graph-based prime program decomposition of a flowgraph is the formal model we propose as a basis for our ideas of programming complexity.

The remainder of this section will briefly describe these separate roots for program complexity, while Section 2 will describe our prime program complexity measure in greater detail. Section 3 develops several properties of this measure, and Section 4 gives some examples of its use.

1.1. INFORMATION THEORY

Aspects of information theory and theoretical computer science have been converging for the past 30 years. Fundamental work in this area is due to Chaitin [10] and Kolmogorov [13]. The randomness (or *algorithmic complexity*) of a string of numbers is determined by the minimal length of a program which can compute that string. Thus, the 192 digits in the string

"1234...9899100" can be described by the 27-character pseudo-Pascal program "for I := 1 to 100 do write(I)." However, a random string of 192 digits would, in general, exhibit no such structure, and its shortest representation would be a program to write each digit in turn. If we increase the string to the 5,888,896 digits "1234...9999991000000," then we only need to marginally increase the program complexity to the 31-character string "for I := 1 to 1000000 do write(I)." However, a random string of 5,888,896 digits would need a significantly larger program describing its structure.

The actual bound on a random string of length n is actually $n+c$ for some constant c (i.e., the n digits encoding the string and an "*interpreter*" of length c to decode and print out the digits). Unfortunately, we cannot determine the minimal complexity of a given string since minimal program size for a given function runs into undecidability and the limits of our axiom systems [9]. However, we use such complexity as a theoretical limit to compare against specific program complexity.

1.2. PROGRAM COMPLEXITY MEASURES

The most common way to measure a program is to give its size, either in terms of source lines, with or without comments, or object code. Numerous studies have shown that as poor as it is, ease of collection and approximate reliability have kept its usefulness as a complexity metric in spite of its numerous shortcomings (e.g., application and language specific, known only after project finished, ignores all other developmental factors like tools, environment, quality of staff, etc.) [4].

Because of such problems, other complexity metrics have been proposed [7]. The software science measure of Halstead is based upon variable and token counts in a program, but has no component that addresses the control structure such as loop or conditional statements [12]. McCabe's cyclomatic complexity addresses aspects of the control flow (its loop structure), but has no data component [15]. Knots are related to cyclomatic structure and evaluates the linearity of the control graph. On the other hand, measures like data bindings [3] or the def-use testing criteria of Rapps and Weyuker [17] look at data interaction with little control flow information. Productivity measures like the COCOMO model [6] were developed to address development time and effort, and hence do not directly address complexity or reliability. Other measures look at assorted attributes of a project (e.g., size, development environment, machines available, programming language used, function points) and use a statistical approach for developing projections [1, 18].

While all of these measures capture some aspects of the programming process, and some are useful for large scale projections of cost, effort, time, and reliability in large developments, we are interested in studying the science of programming by identifying those characteristics of the source program that reflect on the programming process. In addition, we believe that both control and data interactions affect complexity, which some of the other models do not address. We believe that the prime program decomposition adds this to the previous discussion of information theory complexity.

1.3. PRIME PROGRAMS

The prime program was developed by Maddux [14] as a generalization of structured programming to define the unique hierarchical decomposition of a graph. We will assume that graphs contain two classes of nodes. *Function nodes* represent computations by a program, and are pictured as circles or boxes with a single arc entering such a node and a single arc leaving such a node. *Conditionals* are represented as function nodes with multiple output arcs. A second class of node, *joins*, is represented as a point where two arcs flow together to form a single output arc. We can describe the decomposition of a graph via its function nodes alone.

A *proper program* is a graph containing one input arc, one output arc, and for each (function) node in the graph, there is a path from the input arc through that node to the output arc. A *prime program* is a proper program of more than one node that contains no proper subprograms of more than one node (i.e., no two arcs can be cut to extract another proper subprogram). For example, Figure 1(a) is a prime, while Figure 1(b) is not. In Figure 1(b), arcs A–C and F–G can be cut to extract proper program

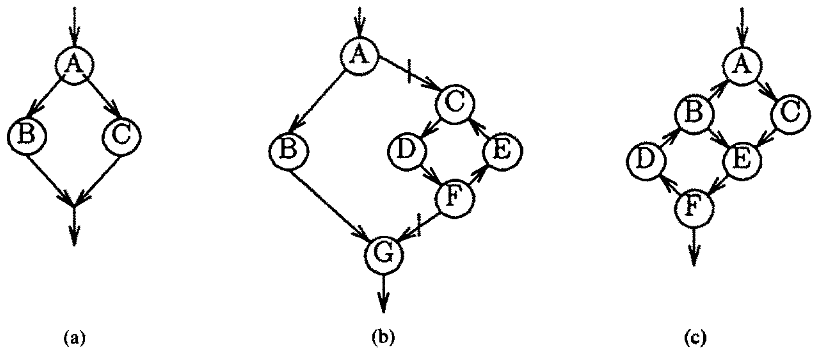


Fig. 1. Proper programs.

C–D–E–F. Note that for simplicity, we often omit the join nodes. Thus, the entrance to node *C* in Figure 1(b) is given by Figure 2(a), but actually means a graph with the added join node of Figure 2(b).

Replacing any prime subprogram in a graph by a single function node creates a unique hierarchical decomposition of a graph into primes [11]. The prime programs containing up to three nodes are the usual structured programming constructs such as *if*, *while*, and *repeat* statements (Figure 3), but the number of such primes is infinite (e.g., Figure 1(c) is a prime of six nodes). In particular, we can create primes of arbitrary sizes (e.g., the lattice pattern of Figure 1(c) can be extended indefinitely).

In order to assure uniqueness of the prime decomposition, we will always represent linear sequences of function nodes as a single function node. Thus, the functionality which could be represented as:

$$\rightarrow \textcircled{A} \rightarrow \textcircled{B} \rightarrow \textcircled{C} \rightarrow$$

will be represented by the single node:

$$\rightarrow \textcircled{D} \rightarrow$$

where D is the functional composition of A, B, and C (i.e., $D = A \circ B \circ C$).

Since our goal is to understand the complexity of programs from some application domain (e.g., all Pascal, C, or Ada programs), we will assume that function nodes represent constructs from that domain. For example, for Pascal programs, we will assume that conditional nodes can only be Boolean expressions (representing *if*, *while*, and *repeat* statements) or expressions that evaluate to an enumerated type (representing a *case* statement), and other function nodes can only be sequences of assignment and procedure call statements. Similarly, although our model will permit

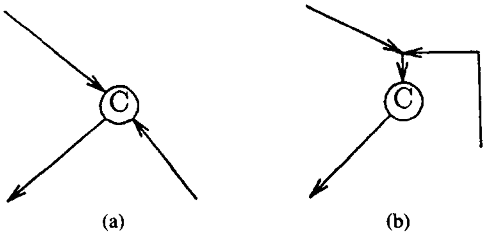


Fig. 2. Omission of join nodes.

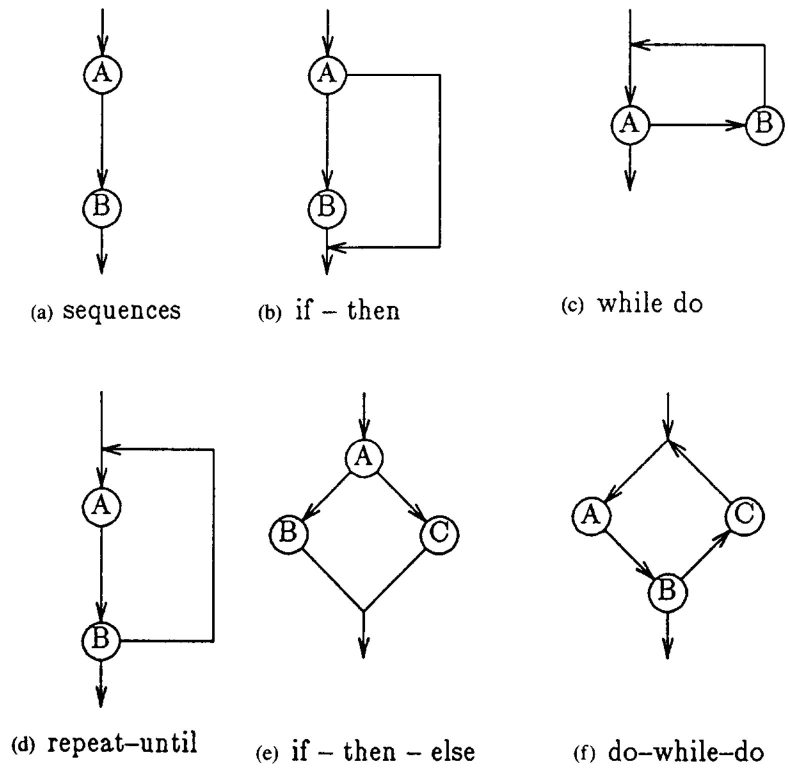


Fig. 3. Primes of up to three function nodes.

side effects in conditionals, we will assume the usual programming language restrictions about no side effects in such Boolean expressions.

2. PRIME PROGRAM STRUCTURAL COMPLEXITY

In order to compare two programs, we present here a modification of the earlier *Hierarchical Abstract Computer* (HAC) measure [2]. Programs are represented by their graphs, and we hypothesize the existence of a computer designed to execute that particular graph. Instructions for this machine, like most machine languages, have an operation code, operand addresses, and next instruction locations, but will be variable in size to minimize storage costs. As programs become more complex, the operation field, operand fields, and next location fields must become larger to handle the increased number of names relevant to the program. Computing the sizes of these fields is our concept of program complexity.

Assume a program is represented as a hierarchical decomposition of its graph. Each node in the graph is either a primitive operation or a function implemented by another graph (e.g., Figure 4 consists of a sequence of two function nodes, the first of which is an if–then–else graph).

Instructions (e.g., function nodes) have the syntax:

$$\langle \textit{label} \rangle : \langle \textit{opcode} \rangle \langle \textit{data list} \rangle (\langle \textit{label list} \rangle)$$

where:

⟨*label*⟩ represents the label on that instruction (e.g., name of node),
⟨*opcode*⟩ represents the function to be performed,
⟨*data list*⟩ is a list of data names, and
⟨*label list*⟩ are the following instructions (i.e., the set of successor nodes). Label *1* is the unique entry node to the proper program graph, and exit arcs (those that exit without going to another function node, although they may pass through other joins) have the unique label *exit*.

We cannot determine the frequency of execution of any program node —in fact, such determination is undecidable and will vary depending upon

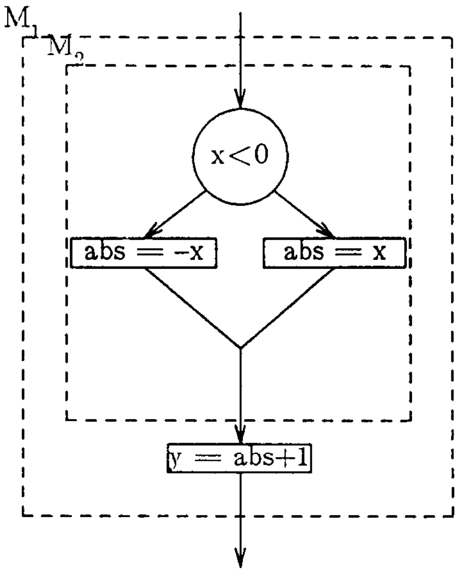


Fig. 4. Prime decomposition of program.

data values. Therefore, we will assume equal distribution of each HAC instruction, data reference, and label. For this reason, if there are n objects of a given type in a HAC program, the size of the relevant field is $\log_2 n$.

Given the graph representation for program P, we define its HAC program as follows:

DEFINITION. Let program P consist of subgraphs $\{G_1,\ldots,G_k\}$. Let L be the set of nodes and I be the set of instructions in P (unique functions performed by the program nodes or the name of a subgraph of P). Let D_{Gi} be the set of data objects in G_i. For each node $j \in G_i$, each instruction has an assembly language-like syntax of:

$$\text{label}_{j1}\text{:opcode}_j\{\text{data list}_j\}(\text{label list}_{j2})$$

for *label*$_{j1}$, *label list*$_{j2} \in L$, *opcode*$_j \in I$, and *data list*$_j \in D_{Gi}$. If execution control is at *label*$_{j1}$, then *opcode*$_j$ executes (i.e., accesses and modifies) data items *data list*$_j$, and the next instruction is chosen from one of the nodes in *label list*$_{j2}$. If the next node is the label *exit*, the machine halts.

The flowgraph containing the program entry arc of P is graph G_1.

Consider the (prime) decomposition of the program in Figure 4 into the two prime HACs:

HAC M_1:	L_1: $m2$		(L_2)	/*call M_2 */
	L_2: add	abs, y, 1	(exit)	/* y = abs + 1 and halt */
HAC M_2:	L_3: test	x,0	(L_4, L_5)	
	L_4: assign	x, abs	(exit)	/* x = abs and exit */
	L_5: negassign	x, abs	(exit)	/* x = −abs and exit */

In this case,

$$D_{M1} = \{y, \text{abs}, 1\} \qquad D_{M2} = \{0, x, \text{abs}\}$$

$$L_{M1} = \{L_2, \text{exit}\} \qquad L_{M2} = \{L_4, L_5, \text{exit}\}$$

$$I_{M1} = \{m2, \text{add}\} \qquad I_{M2} = \{\text{test}, \text{assign}, \text{negassign}\}$$

To compute program complexity, consider two computers built exactly for M_1 and M_2. The size of an instruction is the number of bits required to encode instructions on these computers, and the program size is the number of bits necessary to contain these programs. If there are n

possibilities for any instruction field, the field must be $\log_2 n$ bits wide, and it can be viewed as the amount of information contained within that field.

For program (i.e., graph) M which consists of the subgraphs $\{G_1, \ldots, G_k\}$, let I_{Gi} be the set of opcodes in G_i, let D_{Gi} be the set of data objects referenced in G_i, and let L_{Gi} be the set of labels in G_i. The number of nodes in M will be given by $|M|$.

DEFINITION. *Instruction size:* If instruction j in graph G (i.e., node in G) has data list $\{k_j\}$ and label list $\{l_j\}$, we define *instruction size* for instruction j as:

$$I_j = \log_2 |I_G| + |k_j| \times \log_2 |D_G| + |l_j| \times \log_2 |l_j|.$$

(For easy reading, we omit $|\cdots|$ when it is clear what the meaning is.) Since the HAC machine is tailored to a specific graph to execute, the label field only needs information about the specific nodes that follow a given executing node.

DEFINITION. *Module complexity*: If G is a graph (i.e., program module), then $C(G) = \Sigma(I_j)$ where I_j is the size of instruction j in G.

DEFINITION. *Program complexity*: If M consists of the subgraphs $\{G_1, \ldots, G_k\}$, then $C(M) = \Sigma(C(G_j))$ where $C(G_j)$ is the complexity of module G_j.

We can define such a complexity for any decomposition of a flowgraph. However, we are particularly interested when the decomposition is prime.

DEFINITION. *Prime program structural complexity:* If $M = \{G_1, \ldots, G_k\}$ is the prime program decomposition of a program graph, we call $C(M)$ the *prime program structural complexity*, and write it as $C'(M)$.

Since the prime decomposition of a program is unique, we will not always indicate this decomposition when discussing $C'(M)$ for a particular program M. In the general case, however, since $C(M)$ depends upon the decomposition, it must be indicated. If no decomposition is given, we mean the HAC program consisting of the entire program M.

Out initial model does not differentiate among parameters, local, and global variables. Data references are assumed to be as specified by the programmer. Arguments to functions will simply be given as data objects on the HAC instruction that calls another HAC subprogram, while all other data references in instructions will be considered global references.

From the decomposition of Figure 4, we can compute the size of the HAC program necessary to represent this graph:

For M_1:

The instruction field size will be $\log_2 I = \log_2 2 = 1.00$.
Each data field size will be $\log_2 D = \log_2 3 = 1.58$.

For M_2:

The instruction field size will be $\log_2 I = 1.58$.
Each data field size will be $\log_2 D = 1.58$.

Complexity of each instruction is then:

Instruction L_1: $1.00 + 0 + 0 = 1.00$.
Instruction L_2: $1.00 + 3 \times 1.58 + 0 = 5.74$
The complexity of M_1 is then $1.00 + 5.74 = 6.74$ bits.
Instruction L_3: 6.74
Instruction L_4: 4.74
Instruction L_5: 4.74
The complexity of M_2 is then $6.74 + 4.74 + 4.74 = 16.22$ bits.
The complexity of the program is then $6.74 + 16.22 = 22.96$ bits.

We believe that this model captures:

- *The overall modular structure of a program in terms of the size of each module*—Highly complex code will have complex primes and numerous opcodes, increasing the size of the HAC instruction, and hence its complexity.
- *The data connectivity among program modules in terms of the number of data objects*—With few data items referenced in each module, data fields in each instruction are kept small.

3. PROPERTIES OF THIS MEASURE

Using $C(M)$ and $C'(M)$, we can develop several quantitative results about program structure.

DEFINITION. Let $C_I(M)$ be the total size of all operation fields in HAC M, let $C_D(M)$ be the total size of all data fields in HAC M, and let $C_L(M)$ be the size of all label fields in HAC M.

THEOREM 1. $C(M) = C_I(M) + C_D(M) + C_L(M)$. *Similarly,* $C'(M) = C'_I(M) + C'_D(M) + C'_L(M)$.

Proof. $C(M)$ comes from summing over all instructions j in M. ■

DEFINITION. If M is a graph, then the hierarchical decomposition of M into graphs M_1 and M_2 can be written as $M = M_1 | M_2$. We will assume M_2

is a subgraph of M (i.e., M_1 contains a call to M_2). M will contain $M_1 + M_2 - 1$ nodes (i.e., one of the nodes of M_1 will be a call to M_2, which is replaced by the body of M_2 in M).

An important rationale for computing complexity measures is the desire to choose among several programs that compute the same function—Which one is better? Unfortunately, this is a difficult problem, as the following shows:

THEOREM 2. *The minimal complexity of a graph which computes the same function as a given graph M is undecidable.*

Proof. This is an application of Theorems 4.1 and 4.2 from [9]. From Theorem 4.1, if $\mathscr{T}$ computes the minimal complexity of a set of programs P of sizes up to N, then for $p \in P$, $\mathscr{T}(p) \leqslant |p| + c$ for some constant c.

However, P cannot be the set of all programs of size less than or equal to N since there must be programs with lesser complexity which cannot be computed by $\mathscr{T}$. We show this using Theorem 4.2 of [9]: There must be some program q of size less than N with complexity less than or equal to $|q| + c$ where $\mathscr{T}(q)$ does not halt. Therefore, q is not in P and not computed by $\mathscr{T}$. Therefore, there are programs whose minimal complexity is not greater than $N + c$ which we cannot compute. ∎

Although this result does not permit us to always calculate this minimal complexity, we can still determine the relative complexity between two program graphs in many cases. For most of what follows, we will consider those graphs where each instruction node is unique. This will avoid many of the undecidability problems of general graphs. For example, most of the programs in the next section of this paper obey this property. We will call such programs *Unique-instruction* programs:

DEFINITION. *Unique instruction programs*: Let U be the set of all programs such that every instruction in each node is unique.

Let us first ignore data complexity and investigate various properties of control flow complexity (C_I and C_L).

THEOREM 3. *If* $M = M_1 | M_2$, *then* $C_L(M) = C_L(M_1) + C_L(M_2)$, *i.e., control flow complexity is independent of modular decomposition.*

Proof. If M_2 is extracted from M, then proper subprogram M_2 is replaced in M by a single function node with the instruction name m_2. The number of arcs in either the remainder of M or in M_2 are the same except for the following renaming:

(1) Arcs in M that previously went to the entry node of M_2 are relabeled with the name of the new instruction node m_2; and

(2) The exit arcs from M_2 are relabeled *exit*.

Since the number of labels in any instruction does not change, every instruction has the same label complexity, and for the new node in M for the instruction m_2, its label complexity is $1 \times \log_2 1 = 0$.

THEOREM 4. *If $M \in U$, and if $M = M_1 | M_2$, then $C_I(M) \geqslant C_I(M_1) + C_I(M_2)$.*

Proof. For each instruction in M, the operation field is $\log_2 M$ and C_I is $M \log_2 M$. We need to show that:

$$M \log_2 M = (M_1 + M_2 - 1)\log_2(M_1 + M_2 - 1) \geqslant M_1 \log_2 M_1 + M_2 \log_2 M_2$$

Let $F(x,y) = (x+y-1)\log_2(x+y-1) - x\log_2 x - y\log_2 y$. We need to show that $F(x,y) \geqslant 0$ for $x, y \geqslant 2$.

$$\begin{aligned}
F(x,y) &= (x+y-1)\log_2(x+y-1) - x\log_2 x - y\log_2 y \\
F(2,2) &= 3\log_2 3 - 2\log_2 2 - 2\log_2 2 > 0 \\
\frac{\partial F(x,y)}{\partial x} &= \log_2(x+y-1) + (x+y-1)\frac{1}{(x+y-1)}\frac{1}{\log_e 2} \\
&\quad - \log_2 x - x\frac{1}{x}\frac{1}{\log_e 2} \\
&= \log_2(x+y-1) - \log_2 x \\
&> 0, \text{ because } y \geqslant 2. \\
\frac{\partial F(x,y)}{\partial y} &= \log_2(x+y-1) + (x+y-1)\frac{1}{(x+y-1)}\frac{1}{\log_e 2} \\
&\quad - \log_2 y - y\frac{1}{y}\frac{1}{\log_e 2} \\
&= \log_2(x+y-1) - \log_2 y \\
&> 0, \text{ because } x \geqslant 2.
\end{aligned}$$

Because F is a rational function, from the above we have $F(X,Y) > 0$ for $x, y \geqslant 2$. ∎

Given all N-node U graphs, we would like to characterize the minimal complexity that can be achieved. We show that the intuitive structured programming constructs achieve this minimal value. We will first look at instruction complexity (C_I) minimization, and then study label branching minimization (C_L).

THEOREM 5. *If $P \in U$ is a prime program such that $|P| \geqslant 3$, then there is another program $Q \in U$ such that $|P| = |Q|$ and $C_I'(Q) < C_I'(P)$.*

Proof. Assume $|P| = N$. Since we can create primes of arbitrary size, consider a prime P' such that $|P'| = N - 1 \geqslant 2$. Replace some function node in P' by a two-node prime Q', and build the program $Q = P'|Q'$ which results in $|Q| = N$.

For $N \geqslant 4$, we have: $C_I'(Q) = C_I'(P') + C_I'(Q') = (N-1)\log_2(N-1) + 2\log_2 2 < (N-1)\log_2 N + 2 \leqslant (N-1)\log_2 N + \log_2 N = N\log_2 N = C_I'(P)$.

For $N = 3$, we have: $C_I'(Q) = C_I'(P') + C_I'(Q') = 2\log_2 2 + 2\log_2 2 = 4 < 3\log_2 3 = C_I'(P)$. ■

THEOREM 6. *For all programs $P \in U$ such that $|P| = N$, the minimal possible instruction complexity is bounded by* $2N - 2$.

Proof. From Theorem 5, we know that in the prime decomposition of any program P with N nodes, for any prime with at least k nodes such that $k \geqslant 3$, we can reduce the complexity of P by creating program P' of N nodes by replacing the prime with two primes of $k-1$ and two nodes. We can apply this process $N-1$ times. This leads to the decomposition:

(1) $|P| = N$, *and*
(2) $P = P_1|\cdots|P_{N-1}$ *and*
(3) Each P_i is a two-node prime.

The resulting complexity is: $C_I'(P) = (N-1) \times (2\log_2 2) = 2N - 2$. ■

Note that minimal instruction complexity is linearly related to the number of nodes, that is, the traditional lines of code measure.

Let us now consider the additional label component to complexity.

THEOREM 7. *If program $P \in U$ is a program of N nodes that minimizes instruction complexity C_I', then $C_L'(P) = N - 1$.*

Proof. From Theorem 6, $P = P_1|\cdots|P_{N-1}$ and each P_i is a two-node prime. In order to minimize C_L', we wish to maximize the number of two-node sequence primes (Figure 3(a)) because there will only be a single output arc (with label complexity of 0) for each node in the sequence.

However, we cannot have a sequence prime embedded within another prime. Consider that case:

```
BEGIN
A;
   BEGIN B; C END
END
```

This condition is equivalent to the single prime:

```
BEGIN
A;
B;
C
END
```

However, we can embed a two-node sequence within other two-node primes. That is, if we use CONDITION...END to represent any of *if*, *while*, or *repeat* primes (i.e., any two-node prime consisting of a single binary conditional and a single function node) or a default single function node, we minimize C'_L with the following construct applied recursively:

```
CONDITION
     CONDITION...END
     CONDITION...END
END
```

This represents an alternating pattern of sequence and `CONDITION` primes. (Note that we are limiting the length of sequences to 2. However, in this case, adding more CONDITION...END primes internal to the outer prime does not add to C'_L since each new node adds label complexity of 0.)

Therefore, given the $(N-1)$ two-node primes, we minimize C'_L if:

(1) Half of the primes are two-node CONDITION primes with total label complexity of:

$$2\log_2 2 + 0 = 2$$

(2) Half of the primes are two-node sequence primes with label complexity of:

$$1\log_2 1 + 1\log_2 1 = 0 + 0 = 0$$

$$C'_L = ((N-1)/2 \times 2) + ((N-1)/2 \times 0) = N-1. \qquad \blacksquare$$

THEOREM 8. *For all programs $P \in U$ such that $|P| = N$, and P is composed of two-node primes,*

$$C'(P) = 3N - 3 + C'_D(P).$$

Proof. This is an obvious consequence of Theorems 6 and 7. ■

Consider the more general case of minimizing C'_L for arbitrary programs P in U. We minimize the label complexity by maximizing the number of function nodes relative to the number of conditionals. We can do this by having arbitrary long sequences of function nodes with a total label complexity of 0. To avoid this situation, we define our set U of programs to be those with only two function nodes in a given prime program sequence.

For each conditional node, on each arc leaving that node we can attach this two-node sequence of function nodes. The label complexity of that conditional, for k outgoing arcs, is $k\log_2 k$, and we get a total label complexity for the conditional plus $2k$ function nodes of:

$$(k\log_2 k + 0 + \cdots + 0) = k\log_2 k.$$

The average complexity per node in P is then

$$k\log_2 k/(2k+1) \qquad (\S)$$

which is minimized for $k=1$ (i.e., the single two-node sequence prime). This obviously limits program size, so since (§) increases for increasing k, the minimal practical control structure is $k=2$, which states that C'_L is minimized when we have two outgoing arcs for each conditional node, and each outgoing arc has two function nodes on it.

If we let BEGIN...MIDDLE...END represent either of the two three-node primes of Figure 3(e) and 3(f) consisting of a single conditional expression and two function nodes (e.g., the *if–then–else* and the *do–while–do* constructs), we can demonstrate the minimal C'_L with the structure:

```
BEGIN
    A;
    B;
MIDDLE
    C;
    D;
END
```

This structure consists of two sequence primes as function nodes within a three-node prime, or a total of four function nodes and one conditional node with a total label complexity of $2\log_2 2 = 2$. Structures A, B, C, and D can be either primitive instructions or recursive repetitions of this basic pattern. We summarize this discussion with the following:

THEOREM 9. *The minimal average label complexity for any program that limits prime sequences to length two is* $1/3$.

Proof. Assuming we limit sequences to length 2, from the above discussion we get the minimal program P with the structure that maximizes the

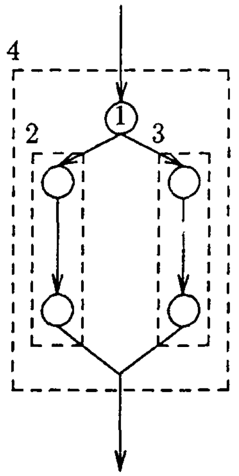


Fig. 5. Construction of minimal complexity.

number of function nodes for a single conditional node:

```
A;
BEGIN
   B;
   C;
MIDDLE
   D;
   E;
END
```

which has one conditional and five function nodes, and $C'_L(P) = 2/6 = 1/3$. ∎

THEOREM 10. *For all programs $P \in U$ with no prime function sequence greater than 2 such that $|P| = N$, the minimal possible label complexity has an upper bound of $N/2$ as N gets large.*

Proof. From Theorem 9, Figure 5 demonstrates the structure that has the maximum number of function nodes (4) for each conditional (1). As more complex programs are built, each of the function nodes can be recursively replaced by a prime program with the structure of Figure 5. Therefore, if P has minimal label complexity x for an N-node program, then we build a larger minimal program P' which consists of four function nodes similar to P and a new conditional. At each level in order to create

P', we combine four of the primitive five-node structures with a new conditional to create a new subgraph with one three-node prime. That is, given the primitive five-node structure, we add:

one conditional node (Node 1 of Figure 5)
two sequence primes (to contain the four structures—Boxes 2 and 3 of Figure 5) *and*
one function node for the newly created structure (Box 4 of Figure 5)

Thus, we add two labels (with complexity 2) to node 1 and a total of four nodes, or an average complexity of 0.5. The total label complexity becomes $(x+0.5)/2$, which since x starts at $1/3$ (from Theorem 9), approaches 0.5 as a ratio as the number of nodes N increases. ■

THEOREM 11. *For all programs $P \in U$ with prime function sequences not greater than 2, the minimal $C'(P)$ is bounded above by* (2.5) $|P| - 2 + C'_D(P)$.

Proof. This is an obvious consequence from Theorems 6 and 10. ■

Note that Theorem 11 is not a strict bound on minimal label complexity since the decomposition of all programs that minimizes C'_I is different from the minimization of C'_L. Therefore, different programs minimize each complexity measure, and no one program achieves this bound.

Our final result shows that data complexity is also positively affected by our modularization technique—modularization reduces complexity.

THEOREM 12. *If $M = M_1 | M_2$, then $C_D(M) \geqslant C_D(M_1) + C_D(M_2)$.*

Proof. Since modularization does not add any data references to any HAC instruction, $D_{M1} \subset D_M$ and $D_{M2} \subset D_M$.

$$C_D(M) = \Sigma d_j \log_2 D_M, \text{for } d_j \text{ data references in instruction } j \in M$$

$$= \Sigma d_j \log_2 D_M, \text{for } d_j \text{ data references in instruction } j \in M_1$$

$$+ \Sigma d_j \log_2 D_M, \text{for } d_j \text{ data references in instruction } j \in M_2$$

$$\geqslant \Sigma d_j \log_2 D_{M1}, \text{for } d_j \text{ data references in instruction } j \in M_1$$

$$+ \Sigma d_j \log_2 D_{M2}, \text{for } d_j \text{ data references in instruction } j \in M_2$$

$$= C_D(M_1) + C_D(M_2).$$ ■

3.1. SUMMARY OF RESULTS

With the previous results, we know that instruction complexity is minimized when we use only two node primes such as the *if*, *repeat*, and *while* constructs (i.e., Figures 3(a)–3(d)). In addition, we know that label complexity is reduced when we use a combination of two and three node primes. Thus, using three node primes (Figures 3(e) and 3(f)), we increase the complexity of the program. Therefore, our model is consistent with the body of anecdotal evidence that structured "gotoless" programming is effective. Based upon this theory, we know that it is impossible to find a "new" control structure that is inherently simpler than what we already have since any new control structure will have to involve a prime of more than three nodes. Although we have limited our results to unique instruction programs, the results suggest that our anecdotal evidence is appropriate.

4. FEASIBILITY

We present several examples that demonstrate that this model is consistent with our intuitive notions of complexity, although we certainly do not view this as a proof of the correctness of this model. We first give examples of the usefulness of this measure using using textual (e.g., source program) formats, and then give a graph example where other textual measures are not applicable.

4.1. EXAMPLE A. DATA ABSTRACTIONS

Current programming practice encourages the use of data abstractions as a means to enhance the quality of the program by encapsulating clusters of data activity into small defined objects and operations. While the practice has existed for many years, only recently have programming languages been available that provide sufficient facilities to create the abstractions conveniently and efficiently.

Figure 6(a) contains an Ada program which adds two rational numbers. Each number is implemented as pairs of integers. Figure 6(b) contains a modified version of the program which contains an abstraction of the data using a new data type *rational* and a new operator + which sums data objects of this new type. Figure 7 contains the HAC representations of these programs. Note that the model can be extended to include parameters to procedures as data items on the function call. (Contrast this it to

```
procedure add_rational(x1, x2, y1, y2:
  in integer;
  z1, z2: out integer) is
  begin
    z1 := x1 * y2 + y1 * x2;
    z2 := x2 * y2;
  end add_rational;
procedure main is
  x1, x2, y1, y2, z1, z2: integer;
  begin
  —
  add_rational (x1, x2, y1, y2, z1, z2);
  —
end main;
```

(a) Add rationals (no abstraction)

```
package rational_arith is
   type rational is record
            numerator,denominator:integer;
            end record;
function " + " (x, y: in rational) return rational;
end rational_arith;
package body rational_arith is
            function " + " (x, y: in rational) return rational is
            begin
            return(x.numerator * y.denominator +
               y.numerator * x.denominator,
               x.denominator * y.denominator);
            end " + ";
   end rational_arith;
with rational_arith; use rational_arith;
procedure main is
   x, y, z: rational;
   begin
   —
   z := x + y
   —

   end main;
```

(b) Add rationals (data abstraction)

Fig. 6. Ada data abstraction example.

the previous example where node L_4's data (0, x, and abs) were static data inside machine M_1.)

By applying the complexity measure, we observe that for the initial program (Figure 7(a)). Prime program structural complexity is $15.48 + 40 = 55.48$ bits, while for typed rationals (Figure 7(b)), it is $4.74 + 40 = 44.74$ bits. The addition procedures have the same complexity (40 bits) since they perform the same transformation on the same primitive data objects. The reduction in complexity is due to the abstraction in one of the calling procedures caused by encapsulating the rational data type as a single concept. The program is able to deal with a single data object x, instead of multiple data objects, x_1 and x_2.

4.2. *EXAMPLE B. DATA COUPLING*

Minimizing data coupling should reduce complexity. There are many possible ways to modularize the structure in Figure 8(a), two of which are shown in Figures 8(b) and 8(c). The first decomposition, however, is

```
MAIN:
—
  ADD  RATIONAL x1, x2, y1, y2, z1, z2
—
  ADD  RATIONAL:
   * x1, y2, t1
 * x2, y1, t2
 + t1, t2, z1
 * x2, y2, z2
```

(a) HAC for nonabstraction program

```
MAIN:
—
+x, y, z
—
"+":
* x.num, y.den, t1
* y.num, x.den, t2
+ t1, t2, z.num
* x.den, y.den, t.den
```

(b) HAC for abstraction program

Fig. 7. Prime sequential form for Ada example.

accompanied by nine dependencies between the modules, while the second realizes a small coupling (two dependencies), implying a more natural modularization and a smaller complexity. Figure 9(a) illustrates a program which possesses this coupling pattern, Figure 9(b) illustrates a modularization of this program according to the first decomposition, while Figure 9(c) illustrates a modularization according to the improved decomposition. Application of the HAC complexity on these programs yields a complexity of 143.48 for the original program, a complexity of $2+53.18+73.32=128.50$ for the complex decomposition, showing that some modularization is better than none, and a complexity of $2+49.74+61.18=112.92$ for the improved decomposition, demonstrating that our measure is sensitive to data references.

4.3. EXAMPLE C. TOOL COMPLEXITY

This measure has applications beyond source programs. For example, it can form the basis for measuring the integration across tools within a software engineering environment. It may provide the basis for evaluating alternative design decisions before implementation begins [8].

For example, a common feature for many of these tools is a graphically-based specifications methodology using data flow diagrams, "bubble charts," or other pictorial information. Measuring the structure of these designs would be an important analysis tool, but such measurements are lacking.

Consider data flow through a typical compiler in Figure 10. In this example, the compiler has two inputs (grammatical description of the programming language and a given source program to compile), and has

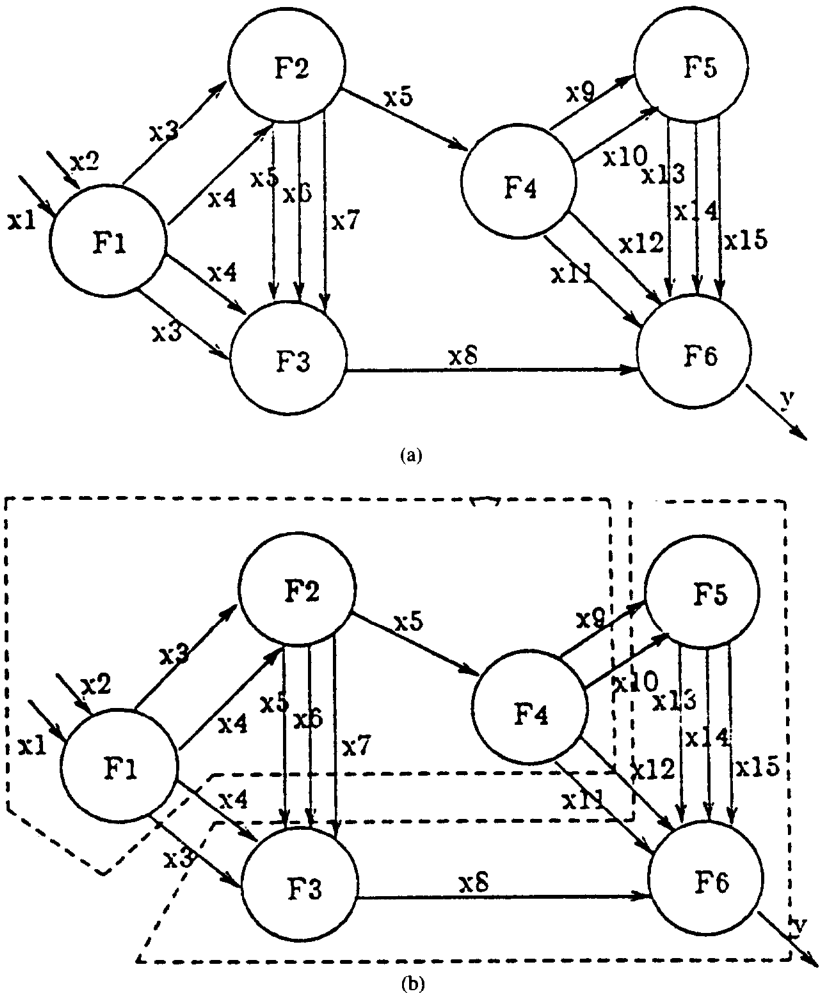
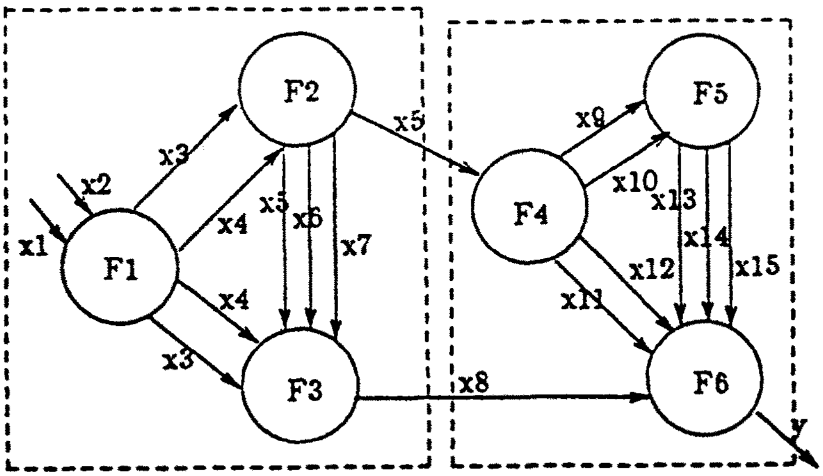


Fig. 8. Data coupling.

one output (the machine language translation of the input source program). We can represent this graphically and compute the complexity measure on the result:

(1) Convert the structure chart to a proper program by adding one input node with no input data items that branches to the given input nodes



(c)

Fig. 8. *Continued.*

(dashed arrows and lines in Figure 10). Do a similar transformation to the output data.

(2) Compute the prime program decomposition of this graph (dotted lines surrounding boxes M_1, M_2, and M_3 in Figure 10):

$$\text{Compiler} = M_1 | M_2 | M_3$$

PROG:	L_1:	$F_1 X_1, X_2, X_3, X_4$
	L_2:	$F_2 X_3, X_4, X_5, X_6, X_7$
	L_3:	$F_3 X_3, X_4, X_5, X_6, X_7, X_8$
	L_4:	$F_4 X_5, X_9, X_{10}, X_{11}, X_{12}$
	L_5:	$F_5 X_9, X_{10}, X_{13}, X_{14}, X_{15}$
	L_6:	$F_6 X_8, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, Y$

a. Original Program before Decomposition

F_7:	F_1	X_1, X_2, X_3, X_4
	F_2	X_3, X_4, X_5, X_6, X_7
	F_4	$X_5, X_9, X_{10}, X_{11}, X_{12}$
F_8:	F_3	$X_3, X_4, X_5, X_6, X_7, X_8$
	F_5	$X_9, X_{10}, X_{13}, X_{14}, X_{15}$
	F_6	$X_8, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, Y$
PROG:	F_7	
	F_8	

b. Decomposition of Figure 8(b)

F_7:	F_1	X_1, X_2, X_3, X_4
	F_2	X_3, X_4, X_5, X_6, X_7
	F_3	$X_3, X_4, X_5, X_6, X_7, X_8$
F_8:	F_4	$X_5, X_9, X_{10}, X_{11}, X_{12}$
	F_5	$X_9, X_{10}, X_{13}, X_{14}, X_{15}$
	F_6	$X_8, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, Y$
PROG:	F_7	
	F_8	

c. Decomposition of Figure 8(c)

Fig. 9. Prime descriptions of Figure 8.

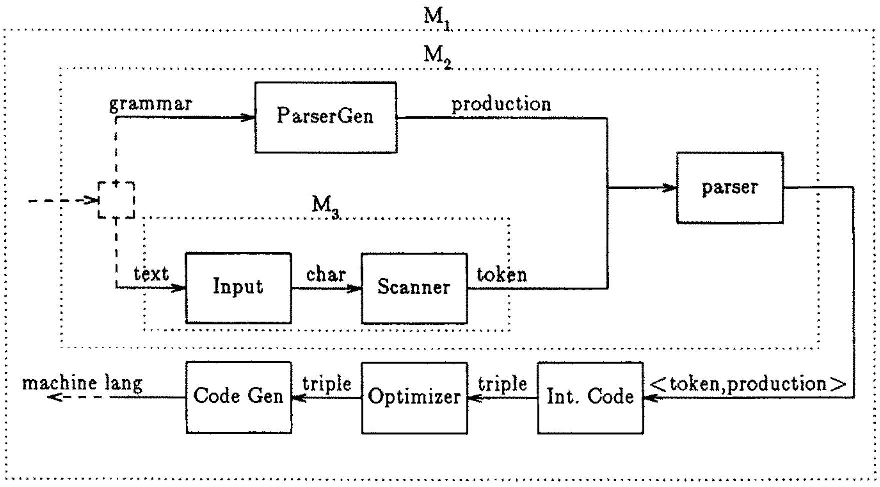


Fig. 10. Compiler structure.

M_3 scanner module—

L_1:	Input	text, char	(L_2)
L_2:	Scanner	char, token	(exit)

which has $|I|=2$, $|D|=3$, $|I_1|=4.16$, $|I_2|=4.16$, and complexity(M_3) = 8.32.

M_2 parser module—

	Start	text, grammar	(L_1, L_2)
L_1:	M_1	text, token	(L_3)
L_2:	ParserGen	grammar, production	(L_3)
L_3:	Parser	token, production, ⟨token, prod_number⟩	(exit)

where ⟨token,prod_number⟩ is the record structure of token produced by the scanner and the grammatical production number from the language grammar. This HAC will have complexity of:

$|I|=4$, $|D|=5$, $|I_1|=8.64$, $|I_2|=6.64$, $|I_3|=6.64$, $|I_4|=8.96$, and

Complexity(M_2) = 30.88.

M_1 Entire compiler—

L_1:	M_2	⟨token,prod_number⟩	(L_2)
L_2:	IntCode	⟨token,prod_number⟩, triple	(L_3)
L_3:	Optimizer	triple, triple	(L_4)
L_4:	CodeGen	triple, machine_language	(exit)

where triple is the intermediate translation of the parsed program. This has complexity:

$$|I|=4,\ |D|=3,\ |I_1|=3.58,\ |I_2|=5.16,\ |I_3|=5.16,\ |I_4|=5.16, \text{ and}$$

Complexity(M_1) = 19.06.

The total complexity of this top level design is then $8.32+30.88+19.06=58.26$.

Using this measure, alternative designs can be evaluated early in the development cycle, and give indications of alternative strategies that can be applied. All too often, the only criterion used in such an evaluation is the experience of the designer. In this case, we have objective criteria that are easily programmable into such an environment that can be used as a design aid before source (or even design) code has been written.

5. CONCLUSIONS

In this paper, we have presented a model of program complexity that uses information theoretic ideas of complexity. Using a subset of realistic program graphs, we have demonstrated that intuitive notions of structured programming are consistent with this model, and for this subset of program graphs, we have demonstrated bounds on program complexity.

We have given several examples in various programming domains that indicate that the measure is so far consistent with our intuitive notions of complexity. This measure can be applied to either source program text or as a measure of the integration within a programming environment. As a design structural complexity measure before the source program is written, we can use it as a means to evaluate alternative specifications.

While this is a useful beginning, this paper only explored the set of unique instruction programs. While an interesting subset, we obviously need to expand the results to more complete sets of program graphs. The model also considers data interactions in its C_D component, and obviously that component needs further work. Tradeoffs between data and control complexity would be a needed aspect of this work.

The measures proposed here may not be optimal for these activities. However, we firmly believe that a combination of information theoretic ideas of complexity combined with the prime program decomposition will provide a basis for an effective programming language complexity measure.

Research support for this paper was provided in part by National Science Foundation Grant CCR-8819793 to the University of Maryland. Early work on the HAC model was aided by William Bail.

REFERENCES

1. A. J. Albrecht and J. E. Gaffney, Software function, source lines of code and development effort prediction: A software science validation, *IEEE Trans. on Software Engineering* 9(6):639–647 (Nov. 1983).
2. W. Bail and M. V. Zelkowitz, Program complexity using hierarchical abstract computers, *J. of Computer Languages* 13(3):109–123 (1988).
3. V. Basili and A. J. Turner, Iterative enhancement: A practical technique for software development, *IEEE Trans. on Software Engineering* 1(4):390–396 (Dec. 1975).
4. V. R. Basili and D. H. Hutchens, An empirical study of a syntactic complexity family, *IEEE Trans. on Software Engineering* 9(6):664–672 (Nov. 1983).
5. V. R. Basili and B. T. Perricone, Software errors and complexity: An empirical investigation, *Comm. of the ACM* 27(1):42–51 (Jan. 1984).
6. B. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
7. D. N. Card and R. L. Glass, *Measuring Software Design Quality*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
8. S. Cardenas and M. V. Zelkowitz, A management tool for the evaluation of software designs, *IEEE Trans. on Software Engineering* 17(9):961–971 (Sept. 1991).
9. G. Chaitin, Information-theoretic limitations of formal systems, *J. of the ACM* 21:403–424 (1974).
10. G. Chaitin, A theory of program size formally identical to information theory, *J. of the ACM* 22:329–340 (1975).
11. J. Gannon, M. S. Hecht, and R. S. Herbold, Prime program decomposition, in *16th Hawaii International Conference on System Science*, Jan. 1983, pp. 25–29.
12. M. Halstead, *Elements of Software Science*, Elsevier, New York, 1977.
13. A. N. Kolmogorov, Three approaches to the quantitative definition of information, *Problems of Information Transmission I* 1:1–7 (1965).

14. R. Maddux, A study of program structure, Ph.D. dissertation, University of Waterloo, Canada, July 1985.
15. T. J. McCabe, A complexity measure, *IEEE Trans. on Software Engineering* 2(6):308–320 (1976).
16. H. D. Rombach, Design measurement: Some lessons learned, *IEEE Software* 7(2):17–25 (Mar. 1990).
17. S. Rapps and E. Weyuker, Data flow analysis techniques for test data selection, in *Sixth ACM/IEEE International Conference on Software Engineering*, Tokyo, Japan, 1982, pp. 272–278.
18. C. E. Walston and C. P. Felix, A method of programming measurement and estimation, *IBM Systems J.* 16(1):54–73 (1977).

Received 20 June 1991; revised 3 July 1992, 10 September 1992