Evaluating Software Complexity Measures

ELAINE J. WEYUKER

Abstract—**A set of properties of syntactic software complexity measures is proposed to serve as a basis for the evaluation of such measures. Four well-known complexity measures are evaluated and compared using these criteria. This type of formalized evaluation should help to clarify the strengths and weaknesses of existing and proposed complexity measures, aid in the selection of appropriate measures, and ultimately lead to the definition of better measures by emphasizing important properties.**

Index Terms—**Cyclomatic number, data flow, software complexity, software metrics, software science.**

Introduction

In the last several years, there has been a great deal of interest in defining appropriate ways to measure the complexity of software. Most of the proposed measures are syntactic in nature and frequently involve counting one or more textual properties of the program. In most cases, the author presents arguments to show that as the frequency of the selected features increases, while everything else remains the same, so does the complexity of the program.

Rather than informally discussing the pros and cons of various proposed measures, or doing an empirical study to see how well each of the proposed measures correlate with actual data as in [3], we present instead abstract properties that permit us to formally compare software complexity models. This should allow one to determine the most suitable measure for various purposes and to evaluate newly proposed measures. We then check whether such well-known complexity measures as McCabe's cyclomatic number, Halstead's programming effort, statement count, and Oviedo's data flow complexity satisfy our properties.

Similar attempts to abstract properties of software metrics, and thereby facilitate the comparison and evaluation of competing models have recently been reported. Weyuker [30] has looked at properties of software test data adequacy criteria, and Iannino *et al.* [14] have done similar research for software reliability models. In [23], Prather presents a set of three axioms for software complexity measures. Our intent is somewhat different and more pragmatic than his. Traditionally when a mathematician axiomatizes some notion, the goal is to define a set of properties which describe all and only objects that might *plausibly* be considered to be of the desired type. Prather shows that, in spite of explicitly identified weaknesses in McCabe's cyclomatic number, this complexity measure satisfies all of his axioms. As mentioned above, we are attempting to present a formal framework in which proposed measures can be compared and contrasted. By doing this, an assessment can be made of the suitability of proposed measures for a given purpose. This allows one to identify the weaknesses of a measure in a concrete way and ultimately should lead to the definition of really good notions of software complexity.

One of the difficulties in assessing complexity measures is that it is not always clear what the measure is supposed to be measuring. Frequently mentioned characteristics include the difficulty of implementing, testing, understanding, modifying, or maintaining a program. But in most cases, these terms are themselves vague. By formalizing the properties we use for evaluation, measures can be compared and assessed based on the particular needs of the user of the proposed metric.

We view this work as an initial step. We believe the properties we propose are generally desirable and relevant, but certainly are not complete. Hopefully this work will provide a foundation which can be built upon.

Definitions

Although most of the ideas of the paper are not really dependent on the particular details of the programming language, it is nonetheless necessary to have an explicit syntax in order to make our definitions precise. Our language will contain a finite number of identifiers. *Arithmetic expressions* are to be constructed using constants, identifiers, and the arithmetic operators $+$, $-$, $*$, $/$, in the usual manner.

An *assignment statement* has the form:

$$\text{VAR} \leftarrow \text{EXP}$$

where VAR is an identifier and EXP is an arithmetic expression.

A *predicate* is a Boolean expression having one of the forms:

$$B1 = B2,\ B1 \neq B2,\ B1 < B2,\ B1 \leq B2,$$

where $B1$ and $B2$ are each either a constant or an identifier. A *program body* is defined recursively:

1) An assignment statement is a program body.

Manuscript received February 4, 1986; revised May 28, 1986. This was supported in part by the National Science Foundation under Grant DCR8501614 and by the Office of Naval Research under Contract N00014-85-K-0414.

The author is with the Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012.

IEEE Log Number 8822457.

0098-5589/88/0900-1357$01.00 © 1988 IEEE

2) **IF** PRED **THEN** P
ELSE Q
END

is a program body if PRED is a predicate and P and Q are program bodies.

3) **IF** PRED **THEN** P **END**

is a program body if PRED is a predicate and P is a program body.

4) **WHILE** PRED **DO** P
ENDWHILE

is a program body if PRED is a predicate and P is a program body.

5) P
Q

is a program body if P and Q are program bodies.

We shall refer to program bodies of the form 2), 3), or 4) as *conditionals*. The program body formed as in 5) will be said to be *composed from* P and Q, and will be denoted by P; Q.

A *program statement* has the form:

PROGRAM(variables)

where variables is a list of the input variables.

An *output statement* has the form:

OUTPUT(variables)

where variables is a list of the output variables.

Finally, a *program* consists of a **PROGRAM** statement, followed by a program body, followed by an **OUTPUT** statement. We will frequently call this program body a program, provided no confusion results. Since our language consists of entirely familiar locutions, there is no need for us to specify further its formal semantics.

For a given program P, we write $P(c) = b$ to mean that the program P on input c halts with output b. We write $P \equiv Q$ (P is *equivalent to* Q), to mean that P and Q halt on the same inputs and that $P(c) = Q(c)$ for all such inputs c.

One way to think of a program is as an object made up of smaller programs. Certainly this is the perspective used in our definition of a program body, or any recursive definition. Using this point of view, the basic operation in constructing programs is composition. Because of this perspective, we will apply measures of complexity to program bodies, rather than programs. This will not affect the results in any substantial way for any of the measures considered.

We use the following notation: P, Q, and R will denote program bodies, and $|P|$ will denote the complexity of P, with respect to some hypothetical measure. We shall assume that for any complexity measure being considered and any program body P, $|P|$ is a nonnegative number. It follows immediately, therefore, that for any P and Q,

$$|P| \leq |Q| \quad \text{or} \quad |Q| \leq |P|.$$

That is, the complexity of any pair of program bodies can be compared and ordered. It is true that measures have been proposed [2], [11], [19] which are vectors of nonnegative numbers. A primary advantage of such a measure is that it allows the user to consider different (and not necessarily related) aspects which contribute to the program's overall complexity. In such a case the user can select whichever components are most relevant for the current assessment, and lexicographically order the vectors accordingly. Alternatively, the user could use a scheme which encodes a vector into a single number. In either case, the above property holds for complexity measures which are vectors of numbers. We consider this property to be essential since a primary reason for measuring program complexity is to be able to compare the relative complexities of any set of programs.

Complexity Measures

A large number of software complexity measures have been proposed in recent years, and there have been a number of interesting comparisons of the usefulness of these complexity measures [1], [6], [8], [16], [27], [31]. Among the most frequently cited measures are the number of program statements, McCabe's cyclomatic number [18], Halstead's programming effort [10], and the knot measure [32]. We will not consider this last measure, since for a structured language such as ours, the knot measure of every program is 0.

Probably the oldest and most intuitively obvious notion of complexity is the number of statements in the program, or the *statement count*. A primary advantage of this measure is its simplicity. Although there are a number of different ways to define a statement, once one has been chosen, it is a straightforward and easily automated task to compute the statement complexity of a program.

McCabe [18] defines the complexity of a program to be:

$$v = e - n + 2p$$

where e is the number of edges in a program flow graph, n the number of nodes, and p the number of connected components. It is further demonstrated that if $p = 1$, then $v = \pi + 1$ where π is the number of predicates in the program.

Questions have been raised as to the most appropriate way to treat compound predicates [19], but once the decision has been made, the cyclomatic complexity of a single component program is trivial to compute. Since our programming language permits only simple predicates, the question of compound predicates is moot for us.

Halstead [10] introduced software science in an attempt to measure properties of programs. Following Halstead's notation:

η_1 = Number of distinct operators.
η_2 = Number of distinct operands.
N_1 = Total number of operators.
N_2 = Total number of operands.

The program volume is defined to be:

$$V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2).$$

The potential volume V^* is defined as the minimum possible volume for a given algorithm. Programming effort is then defined to be:

$$E = V^2/V^*.$$

Since V^* is obviously difficult to compute, and not a purely syntactic notion, an approximate computational formula is frequently used [6], [9], [10] as a measure of program complexity:

$$E = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)}{2\eta_2}.$$

This is a purely syntactic, implementation-dependent notion, and we shall refer to this measure as the *effort measure*. When several program bodies are being compared, we shall sometimes wish to indicate the program body for which a property is being computed. Thus we shall write $E(P)$ to denote the effort for program body P or $\eta_2(Q)$ to denote the number of distinct operands in Q.

A substantially different type of complexity measure, based on the data flow characteristics of a program, was proposed by Oviedo [21]. The programming language he defined is similar to the one given above, although **GOTO** statements are permitted in his language but do not contribute to the data flow complexity of the program. Oviedo uses terms familiar from compiler optimization literature [12], [26], defined below:

A program can be uniquely decomposed into a set of disjoint *blocks* of ordered statements having the property that whenever the first statement of the block is executed, the other statements are executed in the given order. Furthermore, the first statement of the block is the only statement which can be executed directly after the execution of a statement in another block. Intuitively, a block is a chunk of code which is always executed as a unit.

A *program flow graph* is a directed graph in which each node corresponds to a block of the program and the edges correspond to the program branches. If the nodes n_i and n_j of the flow graph correspond to the program blocks n_i and n_j then there is an edge (n_i, n_j) from node n_i to node n_j if it is possible for control to transfer directly from block n_i to block n_j in the program.

A *variable definition* takes place in a PROGRAM statement or in an assignment statement. A *variable reference* takes place when the variable is used in an expression (i.e., in an assignment statement or predicate) or an OUTPUT statement.

A *locally available* variable definition for a program block is a definition of the variable in the block. A *locally exposed* variable reference in a block is a reference to a variable which is not preceded in the block by a definition of that variable.

A variable definition in block n_i is said to *reach* block n_k if the definition is locally available in block n_i and there is a path from n_i to n_k (i.e., n_k is a successor of n_i) along which the variable is not locally available in any block on the path (i.e., the variable is not redefined along that path.) A variable definition in a block *kills* all other definitions of this variable that might otherwise reach the block. Let:

R_i = the set of variable definitions that reach n_i.

Oviedo makes the following assumptions:

1) a programmer can determine the definition-reference relationships *within* blocks more easily than the definition-reference relationships *between* blocks, and

2) the number of different variables which are locally exposed in each block is more important than the total number of locally exposed variable references in each block.

Let V_i be the set of variables whose references are locally exposed in block n_i. Then block n_i's *data flow complexity* is

$$DF_i = \sum_{j=1}^{\|V_i\|} \mathrm{DEF}(v_j)$$

where $\mathrm{DEF}(v_j)$ represents the number of available definitions of variable v_j in the set R_i, and $\|V_i\|$ denotes the cardinality of the set V_i. That is, DF_i counts all prior definitions of locally exposed variables in n_i which reach n_i.

Finally, the *data flow complexity* of a program body is:

$$DF = \sum_{i=1}^{\|S\|} DF_i$$

where S is the set of blocks in the program body. That is, the data flow complexity of a program body is the sum of the data flow complexities of each block of the program body. By this definition, only *interblock* data flow contributes to the complexity of a program body. We will write $DF(P)$ to denote the data flow complexity of P. Since we consider the complexity of program bodies, and a variable referenced within a program body may have been defined in a different program body, we assume that for every variable referenced in the program body, there is a single definition at the entry node (i.e., we assume there is a **PROGRAM** statement which defines each variable). Recall that every program body is single entrant, and hence it makes sense to speak of "the entry node."

This definition is closely related to the test data selection or adequacy criterion, *all-uses*, defined in [24], [25] which requires that a test case be included which exercises every definition-reference pair. Similar testing criteria have also been proposed in [13], [17], [20]. Other complexity measures have been proposed which depend at least in part on data flow in [4], [15], [16], [28].

Desirable Properties of Complexity Measures

We now begin our investigation of desirable properties of complexity measures. All of the measures we consider depend only on syntactic features of the program. This is desirable, even necessary, as virtually all semantic questions about a program are in general recursively undecidable [7], [29].

Our first property reflects the intuition that a measure which rates *all* programs as equally complex is not really a measure. We therefore propose:

Property 1: $(\exists P)(\exists Q)(|P| \neq |Q|)$.

Clearly, this is a property which is satisfied by each of the measures we consider. Recall, however, that it was the failure to satisfy this property for structured languages that led us to exclude the knot metric [32] from this study.

Our next property is a strengthening of Property 1 which requires that the measure not be too "coarse". Property 1 states that a measure should not rank all programs as equally complex. Similar intuition implies that a measure is not sensitive enough if it divides all programs into just "a few" complexity classes. Property 2 is an attempt to formalize this intuition. In a much more abstract vein, Blum [5] presented a pair of axioms, which, it is generally believed, should be satisfied by any reasonable definition of complexity. Property 2 is Blum's first axiom.

Property 2: Let c be a nonnegative number. Then there are only finitely many programs of complexity c.

Our language permits only finitely many identifiers. In addition, it is reasonable to assume there is some largest possible number that can be represented and an upper bound on the length of an instruction (perhaps measured in terms of the number of bits needed to represent the instruction, or the number of operators or operands permitted, or some similar syntactically determinable characteristic). This upper bound may be a function of the particular machine used, and will be assumed to exist.

With these assumptions, it follows that statement count fulfills Property 2, but cyclomatic number does not. This reflects one of the obvious intuitive weaknesses of the cyclomatic number measure: it makes no provision for distinguishing between programs which perform very little computation and those which perform massive amounts of computation, provided that they have the same decision structure. This was, at least in part, Hansen's motivation for defining the complexity measure in [11].

Since our language permits only finitely many identifiers and constants, there are at most finitely many distinct operands in a program. Similarly, the language contains only a fixed, finite number of distinct operators. Therefore, for given values of η_1 and η_2, there are only finitely many program bodies having that number of distinct operators and operands. Since $N_2(P) \geq \eta_2(P)$, it then follows that for a given value e of E, there are only finitely-many different program bodies P such that $E(P) = e$.

For data flow complexity, Property 2 does not hold, since a program body could contain arbitrarily many assignment statements of the form:

$$\text{VAR} \leftarrow C$$

where VAR is an identifier and C is a constant. These statements contribute nothing to the data flow complexity of the program body. Of course, one might argue that such a counterexample of this property is not really a reasonable one, especially since there are only finitely many identifiers and constants. But since *intrablock* data flow does not contribute at all to the complexity of a program body, a block could contain the statements:

$$\begin{aligned} X &\leftarrow C \\ X &\leftarrow f(X) \\ &\vdots \\ X &\leftarrow f(X) \end{aligned}$$

where f is some function of one variable and there are arbitrarily many copies of the statement "$X \leftarrow f(X)$" in the block. These statements add nothing to the complexity of the program body in which they appear, and thus the complexity of the program body would be the same whether there were one or one million copies of the statement.

Just as we have argued (Property 2) that it is undesirable for a measure to be too "coarse," in the sense of rating too many programs as being of equal complexity, we also do not want a measure to be too "fine" and assign to every program a unique complexity.

Property 3: There are distinct programs P and Q such that $|P| = |Q|$.

Clearly, each of the complexity measures we consider satisfies Property 3. An interesting question, then, is what type of measure would fail to satisfy Property 3? It is easy to see that any measure which assigns a unique numerical name to each program (sometimes known as a Gödel numbering [7]), and treats this name as the program's complexity, would fail to satisfy this property. For example, if the binary representation of the program was considered as its complexity, such a measure would not satisfy Property 3.

The first three properties we have proposed are really properties of measures and do not directly reflect the fact that we are dealing with programs which have a syntax and semantics. The next property we consider is another strengthening of Property 1, and reflects the fact that we are considering *syntactic* complexity measures.

Property 4: $(\exists P)(\exists Q)(P \equiv Q \,\&\, |P| \neq |Q|)$.

The intuition behind Property 4 is that even though two programs compute the same function, it is the details of the implementation that determine the program's complexity. This is, we are measuring the complexity of the *program*, not the function being computed by the program. Since all the measures we consider are entirely implementation dependent, they all satisfy this property.

As mentioned earlier, the knot measure [32] would not satisfy Property 1 for our language, since for a structured language such as ours, the knot measure of every program is 0. It therefore follows that the knot measure would also fail to satisfy Property 4. Of course, any measure that satisfies Property 4, automatically satisfies Property 1. Under what circumstances, then, might a measure satisfy Property 1 but not Property 4? Since program equivalence is an undecidable question [7], it is clear, intuitively, that no *usable* measure could divide programs into complexity classes based on the equivalence of computations. Therefore, from a pragmatic point of view, Properties 1 and 4 are essentially equivalent.

Our view of programs is that they are objects composed

from simpler programs (or more properly program bodies). Thus it is important to consider the relative complexities of program bodies related in this way. Central to any notion of syntactic program complexity, should be the property that the components of a program are no more complex than the program itself. We believe that "monotonicity" is another fundamentally important property and it is difficult to imagine the sense in which a measure which fails to satisfy the monotonicity property is measuring program complexity. That is:

Property 5: $(\forall P)(\forall Q)(|P| \leq |P; Q|$ and $|Q| \leq |P; Q|)$.

It is easy to see that for statement count, $(\forall P)(\forall Q)(|P; Q| = |P| + |Q|)$ and for cyclomatic number $(\forall P)(\forall Q)(|P; Q| = |P| + |Q| - 1)$. It follows immediately from these relationships that Property 5 holds for these two complexity measures. It is very disappointing to discover, however, that Property 5 does *not* hold for data flow complexity or the effort measure.

For the data flow measure, the program arises because only *interblock* data flow contributes to the program's complexity. Thus, if when P and Q are concatenated they form a single block, the complexity, as computed by this measure, could well decrease. For example, the program body $X \leftarrow 0$ has a data flow complexity of 0, while the program body $Y \leftarrow X$ has a data flow complexity of 1. The program body formed by concatenation:

$$X \leftarrow 0$$
$$Y \leftarrow X$$

has a data flow complexity of 0 (since all data references are to variables defined within the block), which is less than the complexity of one of its parts.

The problem appears to arise only in cases in which two program bodies (or parts of program bodies) are composed to form a single block. If this is not the case, then $(\forall P)(\forall Q)(|P; Q| \geq |P| + |Q|)$, and hence Property 5 holds.

For the effort measure, however, the program seems to be far more fundamental. Consider the case of a program body P with $\eta_1(P) = 12$, $\eta_2(P) = 4$, $N_1(P) = 35$, and $N_2(P) = 44$. For such a program, $E(P) = 20{,}856$. Assume $P; Q$ is a program body composed from (the above) P and Q with $\eta_1(P; Q) = 12$, $\eta_2(P; Q) = 20$, $N_1(P; Q) = 60$, and $N_2(P; Q) = 60$. Then $E(P; Q) = 10{,}800$. We see that for this, and many other easily constructed cases, $E(P) > E(P; Q)$ (i.e., $|P| > |P; Q|$). Notice that this is a feasible set of values. If P and Q use exactly the same set of operators, then $\eta_1(P) = \eta_1(Q) = \eta_1(P; Q)$. In addition, assume that each of the 4 distinct operands of P are each used 11 times, and hence $N_2(P) = 44$. If Q contains 16 distinct operands (i.e., $\eta_2(Q) = 16$) each used once, then $N_2(Q) = 16$. If, furthermore, the operands of P are all different from the operands of Q, it follows that $\eta_2(P; Q) = 4 + 16 = 20$, and $N_2(P; Q) = 44 + 16 = 60$.

Programming effort was proposed as a measure of the amount of effort (time) needed to construct a given program. It is difficult to imagine an argument that it is reasonable that it would take more effort to produce the initial portion of a program, than to produce the entire program. The failure of the effort measure to fulfill this property is therefore of fundamental importance, and makes questionable its usefulness as a complexity measure.

Two variants of this measure have been proposed and used [10], [33] and it is interesting to notice that this property does not hold for any of these versions of the effort measure. Halstead speaks of "impurities" (which are frequently interpreted to be instances of poor programming style), and uses the following measure of effort to minimize their effects when present:

$$E = \frac{\eta_1 N_2 (\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2) \log_2 (\eta_1 + \eta_2)}{2\eta_2}.$$

That is, the estimator $\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$ replaces $N_1 + N_2$. Using this definition of E, and the values of η_1, η_2, and N_2 for P and $P; Q$ of the example above, it follows that:

$$E(P) = 13{,}469$$
$$E(P; Q) = 11{,}648$$

and thus Property 5 does not hold for this version of the effort measure.

A third variant of the effort measure is based on defining:

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$$

where η_2^* is the number of input/output operands needed by the program. Then:

$$E = \frac{\left((N_1 + N_2) \log_2 (\eta_1 + \eta_2)\right)^2}{(2 + \eta_2^*) \log_2 (2 + \eta_2^*)}.$$

Considering again the above example, with $\eta_2^*(P) = 2$, $\eta_2^*(P; Q) = 14$, for this measure of effort,

$$E(P) = 12{,}482$$
$$E(P; Q) = 5625.$$

This demonstrates that Property 5 does not hold for any of the three proposed definitions of the effort measure.

Another related question to consider is whether or not the concatenation of a given program body with other program bodies should always affect the complexity of the resultant program body in a uniform way? Although a given program body R has a fixed complexity in isolation, R may not interact at all with a program body P with which it is concatenated, while R might interact with Q in subtle and important ways which affect the complexity of the resulting program body. Similarly, if R was P, but R and Q were different program bodies of the same complexity, the complexity of $P; R = P; P$ might well be different than that of $Q; R$. These intuitions are reflected in the next property:

Property 6a: $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \ \& \ |P; R| \neq |Q; R|)$

b: $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \ \& \ |R; P| \neq |R; Q|)$.

Since both cylcomatic number and statement count view program bodies as having inherent complexities which are static, regardless of their context, they are not able to reflect this possible difference in interaction, and hence neither measure satisfies Property 6.

To see that these properties hold for the data flow measure, let P be a program body using a given set of variables, while Q is a program body using a different set of variables. If R uses some of the same variables as P, then the computation in R might depend directly on the computation in P and this should be reflected in the complexity. In contrast, assume Q and R use totally disjoint sets of variables. In that case, they can be considered to call for two totally independent computations and thus Q; R might be expected to have a lower complexity than P; R. A similar example can be constructed to show that Property 6b holds for data flow complexity.

It is also easy to show that Property 6 holds for the effort measure. Let P and Q be program bodies using exactly the same set of operators and the same total number of operators. Then $\eta_1(P) = \eta_1(Q)$ and $N_1(P) = N_1(Q)$. Assume too that P and Q have the same number of distinct operands, but that no operand is used in both P and Q (that is, the sets of operands in P and Q are disjoint), and that the total number of operands used in P and Q is the same. Then $\eta_2(P) = \eta_2(Q)$ and $N_2(P) = N_2(Q)$. Therefore it follows that $E(P) = E(Q)$.

Now assume that R uses exactly the same set of operators as P and Q, and the same set of operands as P (and hence a set disjoint from Q's set of oprands). Then:

$$\eta_1(P; R) = \eta_1(Q; R)$$

$$N_1(P; R) = N_1(Q; R)$$

$$\eta_2(P; R) = \eta_2(P)$$

$$\eta_2(Q; R) = \eta_2(Q) + \eta_2(R) = \eta_2(P) + \eta_2(R) > \eta_2(P; R)$$

$$N_2(P; R) = N_2(P) + N_2(R) = N_2(Q) + N_2(R) = N_2(Q; R).$$

Therefore, in computing the efforts for P; R and Q; R, the only different value of the factors is $\eta_2(Q; R) > \eta_2(P; R)$. Hence, $E(P; R) \neq E(Q; R)$ even though $E(P) = E(Q)$.

Consider now two program bodies P and Q which contain exactly the same statements but in different orders. Are P and Q always of equal complexity? We believe the answer to this question should be "not necessarily." Other researchers have argued convincingly that the order of statements may well affect the complexity. Piwowarski [22] argued, for example, that the depth of nesting of loops play a critical role in the complexity of software. For a nonstructured language, Woodward *et al.* [32] argued that the complexity of a program is determined by the flow of control through the program, and hence the order of the statements.

Property 7 asserts that program complexity should be responsive to the order of the statements, and hence the potential interaction among statements.

Property 7: There are program bodies P and Q such that Q is formed by permuting the order of the statements of P, and $|P| \neq |Q|$.

Neither statement count, cyclomatic number, nor the effort measure, satisfy this property since the complexity of a program is completely independent of the placement, and therefore potential interaction among, the program's statements using these measures. In contrast, since the location of statements may affect their interaction and hence the program's complexity if evaluated using the data flow measure, one would expect this property to hold.

To verify that this property does indeed hold for data flow complexity, consider the following program bodies.

```
P: WHILE X≥0  DO X ← X − Y
                ENDWHILE
   WHILE Y≥10 DO X ← X + 1
                Y ← Y − 1
                ENDWHILE

Q: WHILE X≥0  DO X ← X − Y
                  WHILE Y≥10 DO X ← X + 1
                                Y ← Y − 1
                                ENDWHILE
                ENDWHILE
```

For these program bodies, $DF(P) = 12$ while $DF(Q) = 14$. The only difference between P and Q is that in P the two loops are sequential, whereas in Q the loops are nested. In P, therefore, the assignments to X and Y in the second loop can have no influence on their values in the first loop. In Q, however, the assignments to X and Y in the inner loop may affect their values in the outer loop.

An obvious question is: what kinds of syntactic modifications should leave the complexity of a program unchanged? We shall call P *a renaming* of Q if P is identical to Q except that all instances of an identifier x_i of Q have been replaced in P by x_j where x_j does not appear in Q, or if there exists a sequence $Q = P_1, P_2, \cdots, P_n = P$ where P_{i+1} is a renaming of P_i for $i = 1, \cdots, n-1$.

Although we believe that in general renaming represents a reasonable criterion for deeming two programs of equal complexity, we do recognize that there may be circumstances under which one might not wish to rank such programs as equally complex. In particular, if a complexity measure is intended to assess the difficulty of understanding (sometimes called the psychological complexity), then, in fact, the care with which variable names are chosen might well affect the measured complexity. However, how does one quantify the usefulness of mnemonics since this would presumably vary with the individual programmer? We therefore propose:

Property 8: If P is a renaming of Q, then $|P| = |Q|$.

This property is clearly satisfied by each of the considered complexity measures. We therefore ask: what type of measure would fail to satisfy Property 8? Again, as was the case for Property 3, if a Gödel numbering were used as a complexity measure, it would fail to satisfy Property 8. Since using a program's name as its complexity does not correspond to one's intuition about what is a reasonable complexity measure, it is interesting to note that such an encoding scheme considered as a measure of complexity not only fails to satisfy Properties 3 and 8, but also Properties 6 and 7. We consider it corroboration of our theory that such an encoding scheme is rated as a very inappropriate way to measure program complexity by our properties.

The next property asserts that, at least in some cases, the complexity of a program formed by concatenating two program bodies is greater than the sum of their complexities. This reflects the fact that there may be interaction between the concatenated subprograms.

Property 9: $(\exists P)(\exists Q)(|P| + |Q| < |P; Q|)$.

Properties 5 and 9 allow for the possibility that as a program grows from its component program bodies, additional complexity is introduced due to the potential interaction among these parts. Both cyclomatic number and statement count view program bodies as having inherent complexities which are static, regardless of their context, and hence Property 9 does not hold for either of these measures.

It is clear that Property 9 does hold for the data flow complexity measure since it is exactly this type of interaction that the measure is designed to capture. Let P be the program body:

```
IF X<0 THEN Y ← −X
       ELSE Y ← X
       END
```

and Q be the program body:

```
IF Y≤100 THEN Y ← 0
         ELSE Y ← 1
         END
```

Then $5 = DF(P; Q) > DF(P) + DF(Q) = 3 + 1$.

This example demonstrates that data flow complexity recognizes that there may be interaction between program parts and that this interaction will add to the difficulty in implementing, testing, understanding, maintaining, or modifying the resulting program.

For the effort measure, Property 9 also holds. To see this, consider the case for which the set of operators and operands in P is identical to the set of operators and operands in Q. Then it follows that:

$$\eta_1(P) = \eta_1(Q) = \eta_1(P; Q)$$

and

$$\eta_2(P) = \eta_2(Q) = \eta_2(P; Q).$$

However:

$$N_1(P; Q) = N_1(P) + N_1(Q)$$

and

$$N_2(P; Q) = N_2(P) + N_2(Q).$$

Therefore:

$$\begin{aligned}
E(P; Q) &= \frac{\eta_1(P; Q)(N_2(P) + N_2(Q))(N_1(P) + N_1(Q) + N_2(P) + N_2(Q)) \log_2 (\eta_1(P; Q) + \eta_2(P; Q))}{2\eta_2(P; Q)} \\
&= \frac{\eta_1(P)\, N_2(P)(N_1(P) + N_2(P)) \log_2 (\eta_1(P) + \eta_2(P))}{2\eta_2(P)} \\
&\quad + \frac{\eta_1(Q)\, N_2(Q)(N_1(Q) + N_2(Q)) \log_2 (\eta_1(Q) + \eta_2(Q))}{2\eta_2(Q)} \\
&\quad + \frac{\eta_1(P)\, N_2(P)(N_1(Q) + N_2(Q)) \log_2 (\eta_1(P) + \eta_2(P))}{2\eta_2(P)} \\
&\quad + \frac{\eta_1(P)\, N_2(Q)(N_1(P) + N_2(P)) \log_2 (\eta_1(P) + \eta_2(P))}{2\eta_2(Q)} \\
&= E(P) + E(Q) + c
\end{aligned}$$

where $c > 0$.

The last question we discuss is, given that the complexity of a program body should be no less than the complexities of each of its parts (Property 5), can we make a stronger statement? For example, should the complexity of a program body be no less than the sum of the complexities of its components? Intuitively, in order to implement a program, each of its parts must be implemented. Thus one might want to require:

Property: $(\forall P)(\forall Q)(|P| + |Q| \leq |P; Q|)$.

Since neither the effort measure nor data flow complexity fulfill Property 5, it follows immediately that they do not satisfy this property. From the relationship stated ear-

lier, it follows that this property does hold for statement count (and data flow complexity if as a result of composing P and Q, two blocks are not merged into one). Technically, this property *never* holds for cyclomatic number since for that measure, since $|P; Q| = |P| + |Q| - 1 < |P| + |Q|$. But if cyclomatic number is changed so that

$$v' = e - n + 2p - 1 = \pi$$

then $|P; Q| = |P| + |Q|$, and this property holds. Note that the measure is not changed in any fundamental way by this modification, and thus we consider this property to hold for cyclomatic number. (Alternately, we could consider the modified property:

$$(\exists c)(\forall P)(\forall Q)(|P| + |Q| \leq |P; Q| + c)$$

where c is a nonnegative constant. Clearly with $c = 1$, this modified property holds for cyclomatic number.)

But is this really what we want to require? Consider the program body P; P (that is, the same set of statements repeated twice.) Would it take twice as much time to implement or understand P; P as P? Probably not. In general, a measure which views the complexity of a program body as independent of its context (such as statement count or cyclomatic number) will satisfy this property. Although it seems reasonable that the complexity of a program body be related to the complexities of all of its parts, it is difficult to determine the precise desired relationship. We consider this an interesting open question.

Conclusions, Summary, and Future Directions

We have introduced several properties which we believe a syntactic complexity measure should fulfill. We have closely examined four proposed syntactic complexity measures to see which properties they have in common, and which properties distinguish them.

We summarize our findings:

Property Number	Statement Count	Cyclomatic Number	Effort Measure	Data Flow Complexity
1	YES	YES	YES	YES
2	YES	NO	YES	NO
3	YES	YES	YES	YES
4	YES	YES	YES	YES
5	YES	YES	NO	NO
6	NO	NO	YES	YES
7	NO	NO	NO	YES
8	YES	YES	YES	YES
9	NO	NO	YES	YES

By viewing a program as an object built up from smaller programs, important differences between the measures become clear. Conceptually, both statement count and cyclomatic number view a program's components as having inherent complexity, regardless of their context in the program. In contrast to this, the complexity of a program using the data flow measure depend directly on the placement of statements and how the components interact via the potential flow of data. Programming effort falls somewhere between these two views. A given group of statements will yield the same effort regardless of their order, but depending on the other program bodies with which a program body is composed to build a program, the amount that is contributed to the complexity by various textual units could vary.

The failure to satisfy Property 2 is an important weakness of both the cyclomatic number and data flow measures. The problem is that they rate too many programs as equally complex. That is, they are not sensitive enough to what might reasonably be considered differences in program complexity.

Even more fundamental, however, is the failure of the effort measure to satisfy Property 5. We believe it is so important that it calls into question its usefulness as a syntactic complexity measure, especially since $E(P)$ is supposed to directly predict the amount of time needed to implement P. It is difficult to imagine how it is possible that it take longer to produce the initial part of a program, than the entire program.

The data flow complexity measure also failed to satisfy Property 5, but in this case it was due to the fact that the measure only includes the flow of data between blocks, not within blocks. Since two concatenated program bodies (or their parts) may form a single block, it is possible for the data flow complexity of P; Q to be less than that of P or Q. Oviedo's assumption that it is easier to determine definition-reference relationships within blocks than between blocks seems reasonable. But perhaps his conclusion that *all* intrablock data flow should be discounted is too strong. Similarly, the assumption that multiple references to the same variable within a block add *nothing* to the complexity may also be too strong. It seems likely that minor modifications of this measure can solve its failure to fulfill all of our proposed properties.

Properties 6–9 point out subtle differences between programs that neither statement count nor cyclomatic number are responsive to. Their failure to satisfy Properties 6, 7, and 9 reflect their lack of responsiveness to the interaction among program units. This is a weakness shared to a lesser extent, by the effort measure. Specifically, the failure to satisfy Property 7 shows that none of these three measures have any provision for differentiating between nested and sequential loops. Properties 6–9 also identify positive aspects of using data flow to measure program complexity.

We have provided the foundation for comparing and evaluating software complexity measures in a formal way. We have considered four of the most widely cited measures and have shown that there are substantial differences among them.

We believe that several questions and future areas of study are opened by our investigation. The most obvious one is whether the properties proposed here can serve as a foundation for the definition of new complexity measures. Such measures would presumably avoid the weaknesses of previously proposed measures.

Another interesting question to consider is whether the importance of a given property depends on the particular aspect of complexity being studied. For example, as noted

in our discussion of Property 8, the use of well-chosen mnemonics might affect the understandability of a program. Therefore, if the measurement of the comprehendability of a program were the primary feature being assessed by a complexity measure, and the measure failed to satisfy Property 8 *because* it took mnemonic use into account, then the failure to satisfy Property 8 might be acceptable. A related question is: can additional properties be added which are relativized for the particular intended use of the measure?

We hope that this work will encourage a more rigorous look at complexity measures and ultimately lead to the definition of good meaningful measures. We also hope that others will be encouraged to add to our list of properties which we view as an initial step in the development of a theory of software complexity measures.

ACKNOWLEDGMENT

I am grateful to M. Davis, P. Frankl, and S. Weiss for making many helpful and interesting suggestions.

REFERENCES

[1] A. L. Baker and S. H. Zweben, "A comparison of measures of control flow complexity," *IEEE Trans. Software Eng.*, vol. SE-6, no. 6, pp. 506–512, Nov. 1980.

[2] V. R. Basili and E. E. Katz, "Metrics of interest in Ada development," in *Proc. 1983 IEEE Computer Society Workshop Software Engineering Technology Transfer*, 1983, pp. 22–29.

[3] V. R. Basili, R. W. Selby, and T.-Y. Phillips, "Metric analysis and data validation across Fortran projections," *IEEE Trans. Software Eng.*, vol. SE-9, no. 6, pp. 652–663, Nov. 1983.

[4] J. M. Bieman and W. R. Edwards, "Measuring data dependency complexity," Dep. Comput. Sci., Univ. Southwestern Louisiana, Tech. Rep. 83-5-3, July 1983.

[5] M. Blum, "On the size of machines," *Inform. Contr.*, vol. 11, pp. 257–265, 1967.

[6] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Trans. Software Eng.*, vol. SE-5, no. 2, pp. 96–104, Mar. 1979.

[7] M. D. Davis and E. J. Weyuker, *Computability, Complexity, and Language*. New York: Academic, 1983.

[8] W. M. Evangelist, "Software complexity metric sensitivity to program structuring rules," *J. Syst. Software*, vol. 3, pp. 231–243, 1983.

[9] R. D. Gordon, "Measuring improvements in program clarity," *IEEE Trans. Software Eng.*, vol. SE-5, no. 2, pp. 79–90, Mar. 1979.

[10] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

[11] W. J. Hansen, "Measurement of program complexity by the pair (cyclomatic number, operator count)," *SIGPLAN Notices*, vol. 13, no. 3, pp. 29–33, Mar. 1978.

[12] M. S. Hecht, *Flow Analysis of Computer Programs*. Amsterdam, The Netherlands: North-Holland, 1977.

[13] P. M. Herman, "A data flow analysis approach to program testing," *Australian Comput. J.*, vol. 8, no. 3, pp. 92–96, Nov. 1976.

[14] A. Iannino, J. D. Musa, K. Okumoto, and B. Littlewood, "Criteria for software reliability model comparisons," *IEEE Trans. Software Eng.*, vol. SE-10, no. 6, pp. 687–691, Nov. 1984.

[15] S. S. Iyengar, N. Parameswaran, and J. Fuller, "A measure of logical complexity of programs," *Comput. Lang.*, vol. 7, pp. 147–160, 1982.

[16] J. L. Knox and K. C. Tai, "An empirical evaluation of program complexity metrics," North Carolina State Univ., Tech. Rep. TR-84-06, 1984.

[17] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Software Eng.*, vol. SE-9, no. 3, pp. 347–354, May 1983.

[18] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, No. 4, pp. 308–320, Dec. 1976.

[19] G. J. Myers, "An extension to the cyclomatic measure of program complexity," *SIGPLAN Notices*, vol. 12, no. 10, pp. 61–64, Oct. 1987.

[20] S. C. Ntafos, "On required element testing " *IEEE Trans. Software Eng.*, vol. SE-10, no. 6, pp. 795–803, Nov 1984.

[21] E. I. Oviedo, "Control flow, data flow and program complexity," in *Proc. IEEE COMPSAC*, Chicago, IL, Nov. 1980, pp. 146–152.

[22] P. Piwowarski, "A nesting level complexity measure," *SIGPLAN Notices*, vol. 17, no. 9, pp. 44–50, Sept. 1982.

[23] R. E. Prather, "An axiomatic theory of software complexity measure," *Comput. J.*, vol. 27, no. 4, pp. 340–346, 1984.

[24] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," in *Proc. 6th Int. Conf. Software Engineering*, Tokyo, Japan, Sept. 1982, pp. 272–278.

[25] ——, "Selecting software test data using data flow information," *IEEE Trans. Software Eng.*, vol. SE-11, no. 4, pp. 367–375, Apr. 1985.

[26] M. Schaeffer, *A Mathematical Theory of Global Program Optimization*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[27] T. Sunohara, A. Takano, K. Uehara, and T. Ohkawa, "Program complexity measure for software development management," in *Proc. 5th Int. Conf. Software Engineering*, San Diego, CA, Mar. 1981, pp. 100–106.

[28] K.-C. Tai, "A program complexity metric based on data flow information in control graphs," in *Proc. 7th Int. Conf. Software Engineering*, Orlando, FL, Mar. 1984, pp. 239–248.

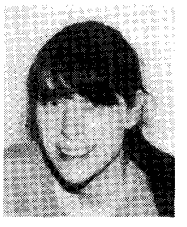
[29] E. J. Weyuker, "The applicability of program schema results to programs," *Int. J. Comput. Inform. Sci.*, vol. 8, no. 5, Nov. 1979.

[30] ——, "Axiomatizing software test data adequacy," *IEEE Trans. Software Eng.*, vol. SE-12, no. 12, pp. 1128–1138, Dec. 1986.

[31] S. N. Woodfield, V. Y. Shen, and H. E. Dunsmore, "A study of several metrics for programming effort," *J. Syst. Software.*, vol. 2, pp. 97–103, 1981.

[32] M. R. Woodward, M. A. Hennell, and D. Hedley, "A measure of control flow complexity in program text," *IEEE Trans. Software Eng.*, vol. SE-5, no. 1, pp. 45–50, Jan. 1979.

[33] S. H. Zweben and K-C. Fung, "Exploring software science relations in COBOL and APL," in *Proc. IEEE COMPSAC*, Chicago, IL, Nov. 1979, pp. 702–207.



Elaine J. Weyuker received the Ph.D. degree in computer science from Rutgers University, New Brunswick, NJ, in 1977.

She has been on the faculty of the Courant Institute of Mathematical Sciences of New York University since 1977, and is currently an Associate Professor of Computer Science. Before coming to NYU, she was a Systems Engineer for IBM and was on the faculty of the City University of New York from 1969 to 1975. Her research interests are in software engineering, particularly software testing and reliability and software complexity measures, and in the theory of computation. She is the author of a book (with Martin Davis), *Computability, Complexity, and Languages*, published by Academic Press.

Prof. Weyuker is a member of the IEEE Computer Society and the Association for Computing Machinery. She has been an ACM National Lecturer and was formerly a member of the Executive Committee of the IEEE Computer Society Technical Committee on Software Engineering.