A measure of software complexity

by NED CHAPIN
Infocci Inc.
Menlo Park, California

IMPORTANCE OF COMPLEXITY

In recounting the copious reasons or excuses for our traditional problems in developing and maintaining computer software, many authorities have mentioned complexity (e.g., References 2, 8, 15). The complexity pointed to sometimes is the inherent complexity of the jobs the computer is to do; sometimes it is the complexity of the systems or programs that direct the computer to do the jobs.

Yet thoughtful observers have also long noticed that the inherent complexity of jobs may differ greatly from the apparent complexity of the software for those same jobs (e.g., References 26, 29). Anyone who has had the opportunity to study the diversity of approaches, designs, and codes resulting when different people independently produce software for the same job, keenly senses the difference between apparent software complexity and inherent job complexity (e.g., Reference 25).

Intuition and common sense generally agree that software which appears simple is superior to software that appears complex, whatever the inherent complexity of the job (e.g., Reference 20). This position is in fact incorporated in the appraisal guidelines of structured design as "simplicity." 23,24 But applying intuition and common sense is not really sufficient to obtain consistently simple software. What is needed is objective, quantitative, reliable, valid and convenient ways of measuring either the complexity or the simplicity in software. To that end, a number of proposals have been advanced. 1,7,11,17,19,21,22,26,32

This paper proposes an alternative measure of software complexity. The background of the measure is briefly given, and its computational procedure described. Then it is applied to a given software design of a small modular structured program. Afterward, the measure is compared with other alternative measures and with programmer ratings of the program. The paper closes with a discussion of the validity of the proposed measure of software complexity.

BASIS OF MEASURE

The proposed measure of software complexity, \( Q \), is a index of the difficulty people have in understanding the function implemented by the software. The proposed measure is quantitative, highly reliable, shows reasonable validity and is easily computed from system and program documentation. Since source code is not required, the measure can be applied to designs before writing the source code, as long as they display some modularization.

The theoretical basis for the proposed \( Q \) measure springs from the set theoretic definition of function. The function is what the software is to direct the computer to do. Briefly put, a function is a correspondence between sets of input data of specified domains, and sets of output data of specified ranges. Hence, any difference in functions must be expressed as differences in the input or in the output or both.

These differences may take two forms—differences in the membership in the sets, or differences in the domains and ranges. Changes in domains—that is, the legal, allowable values the input may take for which the function is defined—and in ranges, are usually of smaller impact in the specification of a function than are changes in the component sets of the input or output. Thus, expanding the domain of a function such as "find a square root" from a domain of one- and two-digit positive integers to a domain of one-, two- and three-digit positive integers is usually a minor matter. But changing the components of the input or output to include a new variable, such as the natural log of the root as an additional output, usually makes a non-trivial change in the function of the software. At the loss of some validity, the proposed measure of software complexity ignores domains and ranges, and concentrates on the components of the sets of input and output.

A listing of the sets of input and output define the function coarsely. A more refined specification is possible if the role of the data is recognized. 5 Some input data are needed for processing (role "P" data)—that is, for the production of output data. The data changed, created, or modified (role "M" data) in value or identity by the performance of the function are the output data. The data used to select or decide which functions to perform serve in a controlling role (role "C" data). Or, data may pass through (role "T" data) a function unchanged in value and identity, when a function of the software is to communicate data from one part of the software to another part, as from one module to another.

To improve the validity of the proposed measure, data roles are recognized in it. Data in a C or control role contribute the most complexity. Data in an M or modified role are major contributors. Data in the P or processing role
contribute some complexity. The $T$ role data that are only passed through contribute the least to software complexity.

The modules or segments of the software can be visualized as communicating data among themselves. Such intermodule data exhibit a simple life history. They start as $M$ role data in one module, become $T$ role data as they are communicated through other modules, and then terminate as $C$ or $P$ role data in a using module. If only two modules are involved in the communication, then the data skip the $T$ role. But the more modules involved in communicating any item of data, the higher is the apparent complexity of the software.

The data communication among modules may be further complicated by the presence of iteration. In modularized programs and systems, the iteration control is observed to be the psychologically most difficult aspect of the "interior" complexity. Modularization can reduce most kinds of "interior" complexity, but makes no reduction in iteration complexity when more than one module is involved.

If the duration or extent of iteration is determined by $C$ or control role data arising outside of the loop exit module, the complexity of the software increases, but not in a linear manner. This non-linearity at first is of little importance, but becomes critical as the number and arrangement of the $C$ role data become larger and more diverse. This is approximated in the computation of the proposed $Q$ measure by a conversion formula involving the square of one-third of a weighted count.

In review, the proposed $Q$ measure appears, because of its stress on function, to reflect the "exterior" rather than the "interior" complexity. This appearance is deceptive for several reasons. First, as Halstead has pointed out with his "$n_A$," the externals place a lower limit on the potential interior complexity of a module. Second, assuming the presence in the module of an algorithm that avoids work redundant and extraneous to the function within the module, the data (as noted in the input-output table) with their associated domains and ranges, place an upper limit on the interior complexity of a module. And third, the process of identifying modules, apportions the interior complexity of a system into the programs' component modules, and their interrelationships. Both what is apportioned where, and the interrelationships, are described by the data flow among the modules, and are reflected in the proposed $Q$ measure of complexity because of its stress on function.

**COMPUTATION OF MEASURE**

The high reliability of the proposed measure arises from the simple computational procedure used on the documentation for the program or system. A measure is reliable when different people using the same computational procedure consistently come to the same result.

The ten steps in the computational procedure for the proposed measure $Q$ are:

1. For each module, count in the input-output table or the equivalent, the number of data items shown in $C$, $P$, or $T$ roles as input, and in $M$ or $T$ roles as output. When one data item appears in multiple roles or has multiple sources or destinations, each is to be counted. Data serving as program-wide or system-wide constants or literals are not counted. The reason for distinguishing the roles of data was described earlier.

2. Multiply for each module the total count for each role by the appropriate weighting factor $W$, as follows: 3 for $C$, 2 for $M$, 1 for $P$, and $\frac{1}{2}$ for $T$. The reason for the weights was described earlier.

3. Sum the weighted counts by module.

4. Assign an initial $E$ of 0 to all modules. Then examine the documentation to determine which modules are to contain the exit tests for iterations where subordinate modules are part of the iteratively-invoked loop body. The tree structure chart usually shows this most conveniently. The loops or iterations are ignored when they are to be performed entirely within a module with no subordinate modules iteratively invoked. The reason for the concern with iteration control was described earlier.

5. For each iteration-exit module identified in Step #4, examine the $C$ items to determine which are to serve in the exit test for the iteration of the subordinate modules that comprise the loop body. Determine where these $C$ data come from. If they come from within this module only, or are constants, add 0 to $E$ for each such $C$ data item. If they come from within the subordinate loop body, add 1 to $E$ for each such $C$ data item. If they come from outside of the loop body, add 2 to $E$ for each such $C$ data item. An example (starting with $E$ as 0) is an item of data which is initialized to a starting value outside of the loop (add 2 to $E$), and also modified within the loop body (add 1 to $E$ to total 3). Note that $E$ for any one module cannot exceed three times the count of the number of data items serving in a $C$ role for determining the exit from iteration.

6. Convert $E$ for each module into a repetition factor $R$ by adding 1 to the square of one-third of $E$. For example, if $E$ is 6, then one-third of $E$ is 2, and 2 squared is 4, 4 plus 1 is 5. Hence $R$ is 5. The reason for this formula was described earlier. $R$ values for common $E$ counts are: $E$ of 0 gives $R$ of 1.00; $E$ of 1 gives $R$ of 1.11; $E$ of 2 gives $R$ of 1.44; $E$ of 3 gives $R$ of 2.00; $E$ of 4 gives $R$ of 2.78; $E$ of 5 gives $R$ of 3.78; $E$ of 6 gives $R$ of 5.00; and $E$ of 7 gives $R$ of 6.44.

7. Multiply the sum of the weighted counts from Step 3 by the modules' respective $R$ values.

8. Find the square root of the products from Step #7. This is $Q$, the index of module complexity. The computation in this step is easily done on most pocket calculators, and effectively is a computation of the geometric mean of the total weighted counts and the inter-module iteration control.

9. Calculate the $Q$ of the program by finding the arithmetic mean (average) of the component modules. This is a simple averaging computation.
10. Calculate the $Q$ of the system by finding the arithmetic mean of the component modules within the component programs, or by weighting the programs' $Q$ from Step #9 by the relative sizes (in terms of modules) of the programs. Note that if in execution a system will iterate the execution of a program, then the $E$ of the module containing the loop exit control is rarely zero.

The practical lower bound on module complexity is 1.0. The exception with a lower bound of zero is a module that has no $C$, $P$, $T$, or $M$ role data—a module that does no function! However, in some delayed-time software, such modules may appear as the root of a system implemented at the root (top) level with JCL (Job Control Language). No upper bound exists for $Q$, but values beyond 11.0 are uncommon with structured programming and structured design.

EXAMPLE OF COMPUTATION

Figure 1 provides a tree structure chart of a program designed with structured programming techniques. The input-output tables are shown in Figure 2. The computation of module $Q$ and program $Q$ are summarized later. Figure 3 lists in its leftmost column, the identifications of the module.

The numbered paragraphs to follow refer to the step numbers described previously.

1. The counts from the input-output (I-O) tables are shown in the second through fifth column of Figure 3. Thus, for instance, Module S-4 has one $C$ item, one $P$ item, two $M$ items, and two $T$ items of data shown in Figure 2. These counts are the "raw counts" entries for line S-4 in Figure 3.

2. The raw counts are multiplied by the weights. Thus, for S-4, the count of 2 for $M$ is multiplied by the weight $W$ of 2 to give 4.

3. The sum of the weighted counts is shown as W-TOT in Figure 3. Thus, for Module S-4, the total of 9 is the sum of the weighted counts of $3+1+4+1$.

4. A review of Figure 1 shows Modules S-3, S-4 and S-7 to contain iteration exits. The broken ring flags them. If the documentation were less specific, other clues would have to be used to locate repeated functions, such as the reading or writing of records.

5. In S-3, iteration exit items as flagged by a dot in the input-output table are the Record Key East, Record Key West, and Out of Sort. In S-4, it is a Bad End Switch. In S-7, no evidence appears from the input-output table about what the iteration control might...
be. It probably is internal within this S-7 module. Care should be taken in such non-appearances to verify them to be reasonable, and not oversights in preparing the input-output tables. In the case of S-7, since the validation appears to be done in S-7 itself, no explicit loop exit data appears reasonable. Hence, $E$ is 1 for all modules except S-3 and S-4.

In S-3, each of the three C items identified comes from a subordinate module which is within the loop body (hence, $E$ is 3). But one C item (Out-of-Sort) also comes from outside the loop. Hence, $E$ totals to 5 for the module S-3. In S-4, the Bad End Switch comes from the subordinate (loop body) module. Hence, $E$ is 1 for S-4.

6. The $E$ to $R$ conversion for all modules is very easy for all but two modules. Since $E$ is 0, $R$ is 1.0. For module S-3, it is 3.78, and for S-4, it is 1.11, by applying the formula.

7. The column PROD is the product of the entries in the W-TOT column times the corresponding entry in the $R$ column. Thus, for row S-4, the product of 9 times 1.1 is 9.9.

8. The square roots of the entries in the PROD column are entered in the $Q$ column. Thus, the square root of 9.9 is 3.1, which is the $Q$ entry for module S-4. The $Q$ entry is the complexity index for a module.

9. The sum of the $Q$ entries for the modules is 35.2, which when divided by 12, the number of modules, yields a program complexity index of 2.9.

10. Since this example system consists of one program, the system index of complexity $Q$ is also 2.9.

The interpretation of the $Q$ measure is easy. Low-complexity is indicated by a low $Q$. But also, a relatively even distribution of the complexity is desirable among the modules. Software prepared using the ideas of both structured design and structured programming show low complexity compared to traditionally prepared software. The highest complexity in the structured software is usually along the main branches of the tree-like structure. But even there, the $Q$ rarely exceeds 11. Leaf modules rarely exceed a $Q$ of 5.

Traditionally prepared software usually shows a higher average complexity. If the software is modular in design, the $Q$ can be fairly easily determined. If it is not modular, two alternatives are open. When source code is available, the modules may be taken to be equivalent to the lexical units used to group code, such as paragraph, section, subroutine, procedure, block, etc. Or when no source code is available, the design documentation may be examined, broken arbitrarily into pieces, and input-output tables prepared. This is rarely an easy process if done in an attempt to highlight and separately recognize functions jumbled together in skimpily-documented traditional designs.

**COMPARISON OF COMPLEXITY MEASURES**

Five major measures of software complexity have been proposed. McCabe has offered a graph-theoretic measure, which others have elaborated. McClure has offered a carefully-thought-out and well tested measure. An application of it is given in the McClure column in Figure 4. Myers has given a basis for the measurement of software quality. While not labeling it a complexity measure, his connectivity matrix measure can be not unreasonably interpreted in that way. An application of it is given in the Myers column in Figure 4. A group of entropy-based measures have been proposed but are not shown in Figure 4. The Zolnkowski measure has been claimed by Zolnkowski to be not applicable to an individual program and to individual modules as it has been presented thus far.

---

**Figure 2**—Input-output tables for example.
<table>
<thead>
<tr>
<th>S#</th>
<th>C</th>
<th>M</th>
<th>P</th>
<th>T</th>
<th>C</th>
<th>P</th>
<th>M</th>
<th>T</th>
<th>W-TOT</th>
<th>R</th>
<th>PROD</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>1.0</td>
<td>10.0</td>
<td>3.2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>1.0</td>
<td>6.0</td>
<td>2.4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>14</td>
<td>9</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>22</td>
<td>3.8</td>
<td>83.6</td>
<td>9.1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>1.1</td>
<td>9.9</td>
<td>3.1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1.0</td>
<td>3.0</td>
<td>1.7</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>21</td>
<td>7</td>
<td>12</td>
<td>4</td>
<td>44</td>
<td>1.0</td>
<td>44.0</td>
<td>6.6</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>1.0</td>
<td>5.0</td>
<td>2.2</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1.0</td>
<td>2.0</td>
<td>1.4</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1.0</td>
<td>2.0</td>
<td>1.4</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1.0</td>
<td>2.0</td>
<td>1.4</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1.0</td>
<td>3.0</td>
<td>1.7</td>
</tr>
</tbody>
</table>

N = 12 = number of modules

Program complexity index 2.9

Figure 3—Computation of proposed complexity index Q for the example shown in Figures 1 and 2.

The McClure measure has well defined limits, depending on the number of control variables (C role data items). But as system or program size increases, larger numbers can be expected. This is intuitively reasonable (the growth generally is less than in the McCabe graph-theoretic measure), but a measure that gave "complexity density" would probably be more useful than just "complexity extent." The proposed Q measure does do this.

At the module level, all the measures are suitable for making comparisons between modules from different programs and systems. The McCabe graph-theoretic measure reflects primarily the amount of logic expressible as conditional transfers in the flow of control. In any modular approach, this is a useful type of complexity to control during design, implementation and maintenance.

The McTap-Zolnowski features-measure of complexity depends for its interpretation on the particular features included. A high measure indicates that some of the features are present to an important degree.

The Myers connectivity-matrix measure reflects the relative dependence of the modules. The higher the measure,
the more interdependent are the modules. But a high measure may also reflect functional variety and code packaging.

The proposed $Q$ measure reflects what will take the form of actual code complexity in implementation because the more data that are used and produced, the more likely is the processing to be complex. Complexity is harder to achieve with only a few items of data going in and out!

The McClure measure is the most sensitive to the use of data for directing the flow of control. Historically, this has been the source of the toughest bugs in our software. And the McClure measure reflects well the complexity of the patterns of use and value-assignment for the data serving for control.

All of the measures can be used with modular designs characterized by properly nested functions. The Zolnowski measure does not require it. The McClure measure requires it, making provision for only one exception, the equivalent of the job abort. The Myers connectivity matrix and the proposed measures do not require it, but are enhanced by it. More significant for both of them is the size of the module. Both work best for relatively small (less than about 60 imperative instructions expected for the implementing code) and fairly even-sized modules. The McCabe graph-theoretic measure is comparatively independent of the design and implementation philosophy and practice. In fact, it can be used to limit the size of modules for the aid of the other measures (such as not more than an expected 10 measure and only rarely over an expected 8 measure for the modules).

Some usage difficulties and conveniences distinguish the measures of complexity. For the Zolnowski measure, they depend upon the features selected. The entropy and McCabe graph-theoretic measures are almost always an underestimation of the ultimate complexity until the design has been carried fully to debugged code. But by then, it is usually too late to take much corrective action. This can be offset by a tight early discipline in design, but few designers welcome it.

The preparation of the Myers connectivity matrix is a separate additional step—not a normal by-product of design. With experience the matrix preparation can be done fairly rapidly, and it does not require either source code or detailed charts to be available. A tree-structure chart may be sufficient, but the availability of input-output tables strengthens the preparation of the connectivity matrix.

To use the McClure measure takes the same detailed review of the design as needed for the Myers connectivity matrix. But the factors looked for are far more objective in

<table>
<thead>
<tr>
<th>MODULE</th>
<th>McCabe</th>
<th>McTAP - ZOLNOWSKI</th>
<th>McClure</th>
<th>MYERS</th>
<th>$Q$</th>
<th>GROUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-1</td>
<td>7</td>
<td>6.5</td>
<td>2.20</td>
<td>0.70</td>
<td>3.2</td>
<td>2</td>
</tr>
<tr>
<td>S-2</td>
<td>1</td>
<td>0.0</td>
<td>0.55</td>
<td>0.70</td>
<td>2.4</td>
<td>5</td>
</tr>
<tr>
<td>S-3</td>
<td>3</td>
<td>6.5</td>
<td>2.75</td>
<td>0.65</td>
<td>9.1</td>
<td>3</td>
</tr>
<tr>
<td>S-4</td>
<td>2</td>
<td>2.5</td>
<td>0.55</td>
<td>0.55</td>
<td>3.1</td>
<td>4</td>
</tr>
<tr>
<td>S-5</td>
<td>1</td>
<td>4.8</td>
<td>0.55</td>
<td>0.30</td>
<td>1.0</td>
<td>10</td>
</tr>
<tr>
<td>S-6</td>
<td>1</td>
<td>1.0</td>
<td>1.10</td>
<td>0.25</td>
<td>1.7</td>
<td>7</td>
</tr>
<tr>
<td>S-7</td>
<td>5</td>
<td>7.5</td>
<td>3.74</td>
<td>0.65</td>
<td>6.6</td>
<td>1</td>
</tr>
<tr>
<td>S-8</td>
<td>1</td>
<td>1.0</td>
<td>0.00</td>
<td>0.65</td>
<td>2.2</td>
<td>8</td>
</tr>
<tr>
<td>S-8-9</td>
<td>1</td>
<td>1.0</td>
<td>0.00</td>
<td>0.25</td>
<td>1.4</td>
<td>6</td>
</tr>
<tr>
<td>S-10</td>
<td>1</td>
<td>1.0</td>
<td>0.72</td>
<td>0.25</td>
<td>1.4</td>
<td>11</td>
</tr>
<tr>
<td>S-11</td>
<td>1</td>
<td>1.0</td>
<td>0.72</td>
<td>0.25</td>
<td>1.4</td>
<td>12</td>
</tr>
<tr>
<td>S-12</td>
<td>1</td>
<td>1.0</td>
<td>0.92</td>
<td>0.55</td>
<td>1.7</td>
<td>9</td>
</tr>
<tr>
<td>PROG</td>
<td>24</td>
<td>5.5</td>
<td>1.25</td>
<td>0.58</td>
<td>2.9</td>
<td>--</td>
</tr>
</tbody>
</table>
the McClure measure than in the Myers measure. Yet the McClure measure, like the McCabe measure, is a retrospective measure of the complexity already present in the design. By contrast, the proposed Q measure can be used on-the-fly during design, implementation and maintenance. It does not require source code. It rarely misstates the final complexity of the software as coded if the input-output tables were conscientiously prepared.

A more serious limitation for the McClure measure and to a lesser extent the Myers connectivity matrix measure, is a failure to define data usage consistently. Use of the input-output tables helps reduce the problem but lacking good definition, the effect is to understate the complexity for all of these measures. The McClure measure requires a full and precise anticipation of the C and M roles in every module of every item of data used anywhere for control. The proposed Q method requires that any item of data be accessed or assigned a value in any module be separately identified, such as a record and the key field within that record. But this is only a statement in data terms of the function of the module. And that is knowledge available to the designer.

The complexity measures differ considerably in computational convenience. The most difficult is the Zolnowski measure because it requires extensive data gathering, computation, measurement and further computation. Somewhat easier is the McClure measure because it involves fewer operations to gather the needed data and fewer computational steps. The next easiest is the Myers connectivity matrix. While the arithmetic is easy, the estimates of strength and coupling involve significant human judgments. Some of the entropy measures, such as the Halstead, are as easy to compute, and are free of the need for such extensive judgments. The McCabe graph-theoretic is still easier, and would be the easiest were it not for the difficulty in obtaining firm counts for the lines, the nodes and branches. The proposed Q method is clearly the easiest of all to use if input-output tables or the equivalent, or code, or Chapin charts, be available, since simple objective counts and simple arithmetic then yield the proposed Q measure. HIPO can be used but adjustments are needed since a HIPO detail chart is not limited in its view to a single module except at the leaf position in the tree. When input-output tables are available from the design effort, the proposed Q measure can be calculated by someone without even a knowledge of computers or data processing.

DISCUSSION

The validity of any proposed measure of software complexity cannot be assessed with precision. As Zolnowski has well pointed out, people view software differently and see its complexity differently. In general, an index is said to be valid if it measures what it purports to measure. Thus, changes in what it purports to measure should be reflected faithfully in the measure. The validity of some measures is open to question or left unaddressed. The other measures cited offers a measure of validity, as shown in Figure 4. On that basis the proposed measure seems as good as any of the others in terms of validity. The column in Figure 4 identified as GROUP represents the rank of the average rankings of the modules in the example by 206 programmers and analysts who had access to a full set of documentation. A rank of 1 represents the most complex, and of 12 the least complex. A ranking does not discriminate the extent of the differences in complexity. Thus, modules S-10 and S-11 were ranked virtually identically. The module S-1 includes a four-way CASE structure which some people regarded as complex. Module S-1 nearly tied for second place with module S-3 in the rankings. No ranking was made of the program overall.

It is surprising that the theory of computational complexity, long a part of the mathematical side of computer science, has contributed so little to measuring software complexity. Perhaps in the future, some contribution will be forthcoming to help assess the validity of measures of software complexity. In the meantime, field experience can help evaluate the contribution in the development and maintenance of systems and programs, of the use of measures of complexity.

REFERENCES