Edgar H. Sibley
Panel Editor

Software engineering is a discipline in search of objective measures for factors that contribute to software quality. NPATH, which counts the acyclic execution paths through a function, is an objective measure of software complexity related to the ease with which software can be comprehensively tested.

NPATH: A MEASURE OF EXECUTION PATH COMPLEXITY AND ITS APPLICATIONS

BRIAN A. NEJMEH

Perhaps the "biggest bang for the buck" in software testing comes from assuring quality at the function or unit level. This is because detecting and correcting unit-level defects during integration and system testing can be very costly. Assuring quality at the function level depends, at least in part, on how thoroughly a function can be tested. Thus, the thoroughness with which functions can be tested is an important design concept. Designers and developers can increase the quality of their software systems by making sure the functions that constitute their systems can be comprehensively tested.

In addition to factors such as modularity and input space size that affect the testability of a software system, an equally important factor that affects testability is the number of execution paths through its functions. Functions with more execution paths are more difficult to test than functions with fewer execution paths. The number of execution paths in a function has received considerable attention in software measurement research [16, 17]. A difficulty of this research is that functions that contain any looping construct can have an infinite number of execution paths. Any meaningful measure of the number of paths in a function must be based on some finite subset of the (usually) infinite set of execution paths. A major criticism of existing software complexity metrics based on the number of acyclic execution paths is that there is a poor relationship between the finite subsets of paths selected by various measures and the set of all execution paths. Thus, the accurate measurement of the acyclic execution path complexity for functions must be addressed.

The author is currently a senior member of the technical staff at the Software Productivity Consortium. The author's present address is B. Nejmeh, SPC, 1880 North Campus Commons Drive, Reston, VA 22091.

SHORTCOMINGS OF McCABE'S MEASURE OF PATH COMPLEXITY

Perhaps the most widely discussed complexity metric is McCabe's cyclomatic complexity metric [11]. McCabe's metric "attempts to determine the number of execution paths in a function" [11, p. 309]. The cyclomatic complexity number, $V(G)$, is the number of logical conditions in the function plus 1. McCabe argued that $V(G)$ represents the number of fundamental circuits in the flow-graph representation of a function. (Evangelist [5] points out that this formula is correct only when each predicate vertex in the flow graph has outdegree 2).

Finally, McCabe argued that the number of fundamental circuits acts as an index of testing effort.

There are several problems with McCabe's metric. First, the number of acyclic paths in a flow graph varies from a linear to an exponential function of $V(G)$ [5]. Thus, to assert that $V(G)$ is a reasonable estimate of the number of execution paths through a function is unjustified. Moreover, the number of acyclic execution paths that may not be tested by a methodology based on McCabe's metric varies from 0 to 2^N, where N is the number of vertices in the flow graph [5]. The poor relationship between the number of acyclic execution paths and the number of execution paths tested, based on McCabe's metric, suggests that McCabe's assumption that total testing effort is proportional to $V(G)$ should not be accepted.

A second problem with McCabe's metric is that it fails to distinguish between different kinds of control flow structures (i.e., the measure treats the `if`, `while`, `for`, etc., structures the same). Certain control flow structures, however, are more difficult to understand and use properly.

Finally, Curtis [2] argues that McCabe's metric does not consider the level of nesting of various control structures (e.g., three `for`-loops in succession will result in the same metric value as three nested `for`-loops). Curtis argues that nesting may influence the psychological complexity of the function. Moreover, many researchers [4] argue that psychological complexity has a large impact on software quality.

In light of the problems with McCabe's metric, a new and intuitively more appealing measure of software complexity has been developed. The new metric of software complexity, NPATH, overcomes the shortcomings associated with McCabe's metric.

BACKGROUND DEFINITIONS

The following definitions are pertinent to the discussion of NPATH.

- A *control flow graph* is a graph in which each vertex represents either a basic block of code (statement sequence that contains no branches) or a branch point in the function, and each edge represents possible flow of control. More formally, the *control flow graph* of a function can be represented as a directed graph four-tuple (V, E, *Ventry*, *Vexit*), where
  - —V is a set of vertices representing basic blocks of code or branch points in the function;
  - —E is a set of edges representing flow of control in the function;
  - —*Ventry*, an element of V, is the unique function entry vertex; and
  - —*Vexit*, an element of V, is the unique function exit vertex.
- A *path* P in a control flow graph G is a sequence of vertices $\langle V_0, V_1, \ldots, V_j \rangle$, such that there is an edge from V_i to $V_i + 1$ for $i = 0, \ldots, j - 1$.
- A *possible execution path* is any path from *Ventry* to *Vexit* in a flow graph.
- An *elementary cycle* is any path P from vertices V_1, $V_2, \ldots, V_k$ such that $V_1 = V_k$ and $V_i <> V_j$ for $1 < i < j <= k$. In short, an elementary cycle is a cycle that contains no other cycles within it.
- A *loop control vertex* is a vertex V with the following two properties: (1) V has an out-edge that lies on at least one elementary cycle that begins and ends at V, and (2) V has a second out-edge that lies on a path leading out of the loop. An appealing property to determine the loop control vertices in a control flow graph is that any execution path through a function can be constructed once the loop vertices are known. This happens when zero or more of the cycles that comprise the loop for a vertex are substituted until the entire execution path is constructed.
- A *loop* is a cycle that begins and ends at a given loop control vertex.
- A *range* of a statement V is the set of statements whose execution may be determined by the truth value of the expression in statement V.

BACKGROUND OF THE NPATH MEASURE

To keep the execution path measure finite and eliminate redundant information, a measure of execution path complexity should not reflect every possible iteration of a loop. This suggests that a good characterization of the number of execution paths in a function should only count a single iteration of each loop. Such a metric counts all paths where a loop is not iterated more than twice. It is a count on the number of acyclic execution paths through a function.

This approach has initiated the development of *NPATH*, a metric that counts the number of execution paths through functions written in the C programming language [9]. Although the NPATH metric is defined for the C programming language, most of the control structures available in C are similar to the control structures in other high-level languages (e.g., Pascal, PL/1, Fortran). Thus, the NPATH approach is applicable to other programming languages.

THE EXECUTION PATH COMPLEXITY OF C CONTROL FLOW STRUCTURES

The acyclic execution path complexity expressions for each of the control flow structures in the C programming language are defined in the following subsections.

`if` Statement (Figure 1, next page)

Syntax

```
if (⟨expr⟩)
  ⟨if-range⟩
S;
```

The semantics of the `if` statement are as follows: If the expression ⟨`expr`⟩ is `True`, then the statement com-

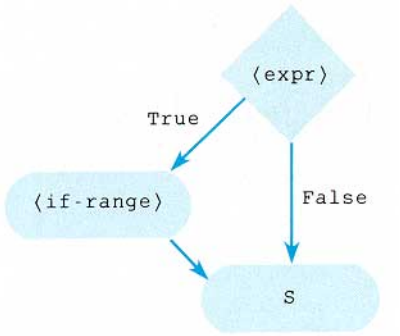

FIGURE 1. Flow Graph for the `if` Statement

prising the ⟨if-range⟩ is executed; otherwise, the statement following the **if** statement is executed.

The acyclic execution path complexity (*NP*) for the **if** statement is

$$NP(\textbf{if}) = NP(\langle \texttt{if-range} \rangle) + NP(\langle \texttt{expr} \rangle) + 1.$$

This expression is derived from the flow-graph representation of the statement. In particular, the number of acyclic execution paths through the **if** statement is the number of paths through the ⟨if-range⟩ plus 1 for the case when the ⟨expr⟩ is False. The complexity of the logical expression ⟨expr⟩ is also added to the complexity of the **if** statement. A definition for the complexity of the logical expression ⟨expr⟩ appears below. The same reasoning applies to each of the following acyclic execution path complexity expressions.

`if-else` Statement (Figure 2)

Syntax

```
if (⟨expr⟩)
   ⟨if-range⟩
else
   ⟨else-range⟩
S;
```

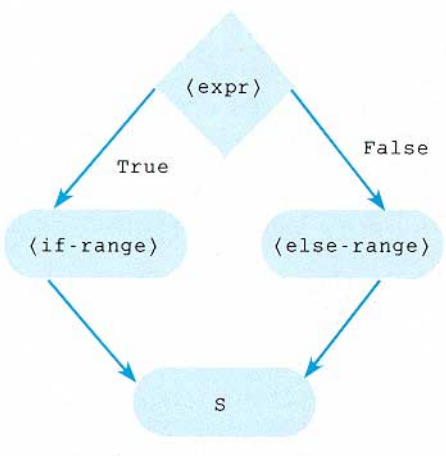

FIGURE 2. Flow Graph for the `if-else` Statement

The semantics of the **if-else** statement are as follows: If the expression ⟨expr⟩ is True, then the statement comprising the ⟨if-range⟩ is executed; otherwise, the statement comprising the ⟨else-range⟩ is executed.

The acylic execution path complexity for the **if-else** statement is

$$NP(\textbf{if-else}) = NP(\langle \texttt{if-range} \rangle) + NP(\langle \texttt{else-range} \rangle) + NP(\langle \texttt{expr} \rangle).$$

`while` statement (Figure 3)

Syntax

```
while (⟨expr⟩)
   ⟨while-range⟩
S;
```

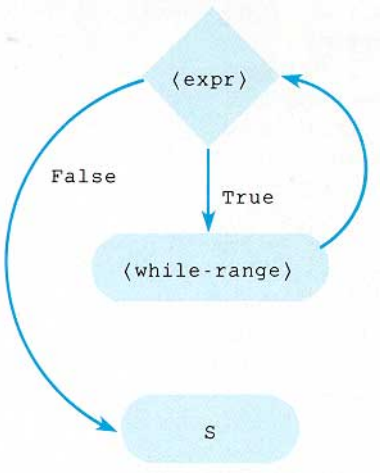

FIGURE 3. Flow Graph for the `while` Statement

The semantics of the **while** statement are as follows: If the expression ⟨expr⟩ is True, then the statement comprising the ⟨while-range⟩ is executed, and control branches back to the ⟨expr⟩ logical evaluation; otherwise, the statement following the **while** statement is executed.

The acyclic execution path complexity for the **while** statement is

$$NP(\textbf{while}) = NP(\langle \texttt{while-range} \rangle) + NP(\langle \texttt{expr} \rangle) + 1.$$

`do while` Statement (Figure 4)

Syntax

```
do
   ⟨do-range⟩
while (⟨expr⟩);
S;
```

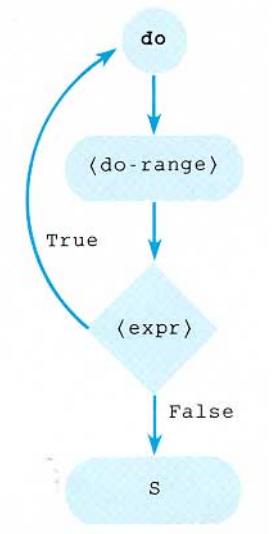

FIGURE 4. Flow Graph for the `do while` Statement

The semantics of the **`do while`** statement are as follows: The statement comprising the ⟨do-range⟩ is executed, and then, if the expression ⟨expr⟩ is `True`, control branches back, and the statement comprising the ⟨do-range⟩ is reexecuted; otherwise, the statement following the **`do while`** statement is executed.

The acyclic execution path complexity for the **`do while`** statement is

$$NP(\textbf{do}) = NP(\langle \texttt{do-range} \rangle) + NP(\langle \texttt{expr} \rangle) + 1.$$

`for` Statement (Figure 5)

Syntax

```
for (⟨expr1⟩; ⟨expr2⟩; ⟨expr3⟩)
   ⟨for-range⟩
S;
```

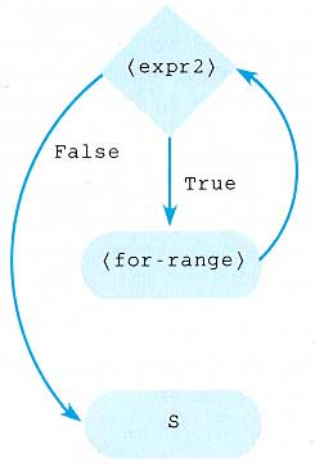

FIGURE 5. Flow Graph for the `for` Statement

The semantics of the **`for`** statement are as follows: The expression ⟨expr1⟩ is used to initialize a loop control variable, ⟨expr2⟩ is used as the termination condition for the loop, and ⟨expr3⟩ is used as the increment/decrement value for the loop variable upon each iteration of the loop. The sequence of statements denoted by the ⟨for-range⟩ are executed as long as the expression ⟨expr2⟩ is `True`.

The acyclic execution path complexity for the **`for`** statement is

$$NP(\textbf{for}) = NP(\langle \texttt{for-range} \rangle) + NP(\langle \texttt{expr1} \rangle) + NP(\langle \texttt{expr2} \rangle) + NP(\langle \texttt{expr3} \rangle) + 1.$$

`switch` Statement (Figure 6, next page)

Syntax

```
switch (⟨expr⟩)
  {
    ⟨case-range1⟩
    .
    .
    .
    ⟨case-rangen⟩
    ⟨default-range⟩
  }
```

The semantics of the **`switch`** statement are as follows: The **`switch`** statement transfers control to one of several statements depending on the value of the expression ⟨expr⟩. When the **`case`** statement is executed, ⟨expr⟩ is evaluated and compared with the value of each case. If a **`case`** value is equal to the value of ⟨expr⟩, then control is transferred to the statement following the matched **`case`** value. If there is neither a **`case`** match nor a default, then the statements in the **`switch`** are not executed. Note that a ⟨case-range⟩ is delimited by either another ⟨case-range⟩ or a **`break`** statement.

The acyclic execution path complexity for the **`switch`** statement is

$$NP(\textbf{switch}) = NP(\langle \texttt{expr} \rangle) + NP(\langle \texttt{default-range} \rangle) + \sum_{i=1}^{i=n} NP(\langle \texttt{case-range}_i \rangle).$$

In the case of a null ⟨case-range$_i$⟩, a situation where the **`case`** statement falls through to the next case or ⟨case-range$_{(i+1)}$⟩, the complexity of ⟨case-range$_i$⟩ is 1.

? Operator (Figure 7, next page)

Syntax

⟨expr1⟩ **?** ⟨expr2⟩ : ⟨expr3⟩

The semantics of the **?** statement are as follows: The expression ⟨expr1⟩ is evaluated first. If ⟨expr1⟩ is nonzero (`True`), then the expression ⟨expr2⟩ is evaluated and returned as the result of the expression; otherwise, ⟨expr3⟩ is evaluated. Only one of ⟨expr2⟩ and ⟨expr3⟩ is evaluated.

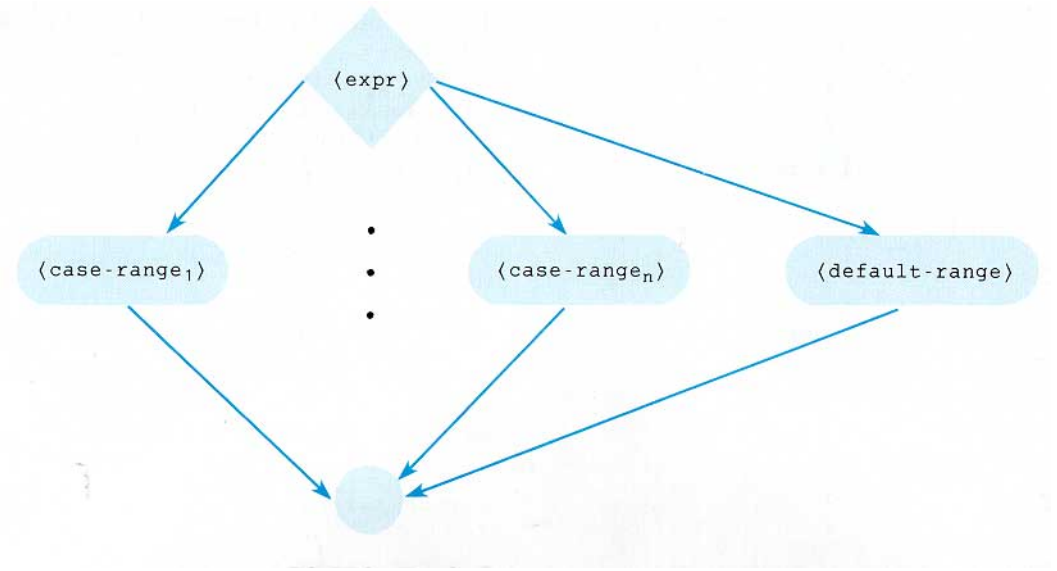

FIGURE 6. Flow Graph for the `switch` Statement

The acyclic execution path complexity for the **?** operator is

$$NP(?) = NP(\langle \texttt{expr1} \rangle) + NP(\langle \texttt{expr2} \rangle) + NP(\langle \texttt{expr3} \rangle) + 2.$$

For our purposes, the **?** operator can be treated similarly to the **if-else** statement. The 2 that is included in the *NP*(**?**) expression reflects the execution path complexity resulting from this statement (i.e., one path is traversed if ⟨expr1⟩ is `True`, and another path traversed if ⟨expr1⟩ is `False`).

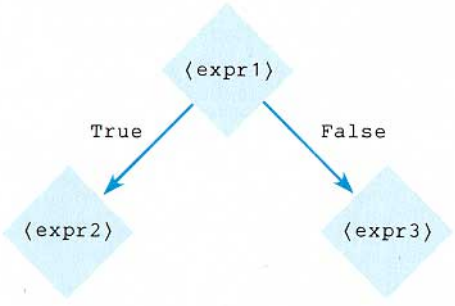

FIGURE 7. Flow Graph for the ? Operator Statement

goto Statement

When the statement **goto** `label` is executed, transfer of control goes to the "labeled" statement, where program execution continues. A **goto** statement is referred to as *forward referencing* when the "labeled" statement being referenced appears textually after the **goto** statement. Similarly, a **goto** statement is referred to as *backward referencing* when the "labeled" statement being referenced appears textually before the **goto** statement.

The acyclic execution path complexity expression for the **goto** statement is difficult to define. In the case of a forward referencing **goto**, accounting for the complexity of the code beginning at the target of the **goto** may overstate the complexity of the code between the **goto** statement and the target statement. On the other hand, a backward referencing **goto** would create a cycle in the program flow graph; it would thereby enable the execution path complexity to be infinite.

Given the inherent ambiguity and difficulty in accounting for the execution path complexity created by the **goto** statement, our path complexity metric does not account for the execution path complexity introduced by the **goto** statement. Although the number of acyclic paths resulting from the use of the **goto** statement could be significant in theory, in practice the **goto** statement is rarely used. Moreover, the use of the **goto** statement is generally considered poor programming practice [3].

break Statement

A **break** statement causes exit from the innermost enclosing loop (**while**, **do**, **for**) or **switch** statement in which it appears. If and when the **break** statement is reached, it ends the execution of statements within the basic block of code where it occurs. In the context of execution path complexity analysis, the **break** statement can be thought of as the last statement on the execution path containing the basic block of code in which it occurs. As such, the execution path complexity of the **break** statement is 1.

Expressions

The syntax for a logical expression is as follows:

⟨expr1⟩ op1 ⟨expr2⟩ op2 . . . op(N − 1)⟨exprN⟩,

where ⟨expr1⟩, ⟨expr2⟩, . . . , ⟨exprN⟩ are expressions and op1, op2, . . . , op(N − 1) are any one of the logical operators **and** (**&&**) or **or** (**||**).

The complexity of logical expressions can have a tremendous impact on the number of execution paths in a function. This is because of the way logical expressions are evaluated in C. In particular, logical expressions are evaluated only until the final truth value of the expression can be determined. Consider the two logical expression operators **&&** (**and**) and **||** (**or**). In the case of the **and** operator, the truth value of the logical expression ⟨expr1⟩ **&&** ⟨expr2⟩ is determined as follows: If ⟨expr1⟩ is `False`, then the value of the entire logical expression is `False`, and the evaluation of the logical expression is terminated; otherwise, ⟨expr2⟩ is evaluated. If ⟨expr2⟩ is `True`, then the value of the entire logical expression is `True`; otherwise, the value of the logical expression is `False`. In the case of the **or** operator, the truth value of the logical expression ⟨expr1⟩ **||** ⟨expr2⟩ is determined as follows: If ⟨expr⟩ is `True`, then the value of the entire logical expression is `True`, and the evaluation of the logical expression is terminated; otherwise, ⟨expr2⟩ is evaluated. If ⟨expr2⟩ is `True`, then the value of the entire logical expression is `True`; otherwise, the value of the logical expression is `False`.

The number of expressions that may conditionally be executed in a logical expression grows linearly with the number of **&&** and **||** operators in the logical expression. That is, every expression within a logical expression may have to be evaluated in order to determine the truth value of the entire logical expression. Therefore, the number of acyclic execution paths added as a result of each logical operator in a logical expression is 1.

To illustrate, consider the function segment to the left, as well as a more explicit but logically equivalent form of the function segment to the right (also see Figure 8):

```
if( (A && B) && C )          if( A )
  {                            {
    S1;                        if( B )
  }                              {
else                               if( C )
  {                                  {
    S2;                              S1;
  }    which is equivalent to      }
SN;                                else
                                     {
                                     S2;
                                     }
                                 }
                               else
                                 {
                                 S2;
                                 }
                               else
                                 {
                                 S2;
                                 }
                           SN;
```

The flow-graph representation of the statement indicates that there are four different acyclic execution paths through this flow graph (assuming `S1` and `S2` are **sequential** statements). The path complexity expressions defined in this article lead to the same conclusion about the number of acyclic execution paths in this statement. That is, the complexity of the **if-else** statement, *NP*(**if-else**), has been previously defined to be *NP*(⟨if-range⟩) + *NP*(⟨else-range⟩) + *NP*(⟨expr⟩). In the above case, *NP*(⟨if-range⟩) and *NP*(⟨else-range⟩) are each 1 since both `S1` and `S2` are **sequential** statements. The complexity of the logical expression (`A` **&&** `B`) **&&** `C` is 2 (the number of **&&** and **||** operators in the expression). Thus, *NP*(**if-else**) = 1 + 1 + 2 = 4.

The acyclic execution path complexity for any logical expression is

NP(expression)

= number of **&&** and **||** operators in the expression.

continue Statement

The **continue** statement forces the next iteration of an enclosing loop (**for**, **while**, **do**) to begin. Thus, the **continue** statement represents a back edge in the control flow graph of a function. NPATH does not account for the complexity of this construct.

return Statement

The **return** statement terminates the execution of a C function. A **return** statement can also contain an expression. Therefore, the complexity of the **return** statement is *NP*(⟨expr⟩).

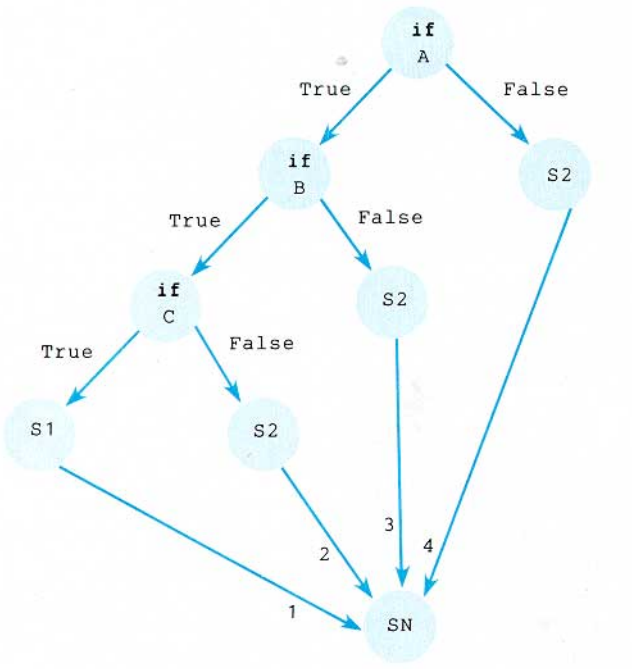

FIGURE 8. A Logical Expression and Its Corresponding Flow Graph

`sequential` Statements and Function Calls

The execution path complexity for the **`sequential`** statement is 1 because there is only one path created by consecutive **`sequential`** statements. Note that function calls are treated as **`sequential`** statements. That is, it is assumed that the code within the function being called has been unit tested and is functioning properly. Therefore, the execution path complexity of function calls is also 1.

THE EXECUTION PATH COMPLEXITY OF C FUNCTIONS

The composite acyclic execution path complexity for a C function, NPATH, is

$$\text{NPATH} = \prod_{i=1}^{i=N} NP(\texttt{Statement}_i),$$

where N denotes the number of statements in the body of the function and $NP(\texttt{Statement}_i)$ denotes the acyclic execution path complexity of statement `i`. Note that the complexity of any statement range is the product of complexities of the statements in the range.

The complete algorithm to compute NPATH is listed in Appendix A.

An Example of NPATH

A segment of C source code and its corresponding NPATH measure follows (also see Figure 9):

```
if ( ch == 'a' )
  {
    actr++;
    Func_a( );
  }
if ( ch == 'b' )
  {
    bctr++;
    Func_b( );
  }
if ( ch == 'c' )
  {
    cctr++;
    Func_c( );
  }
if ( ch == 'd' )
  {
    dctr++;
    Func_d( )
  }
```

NPATH = 16.

The NPATH value of 16 is obtained as follows: $NP(\langle \texttt{if} \rangle) = NP(\langle \texttt{if-range} \rangle) + NP(\langle \texttt{expr} \rangle) + 1$. In the above example, $NP(\langle \texttt{expr} \rangle) = 0$ for each **`if`** statement. Also note that ⟨`if-range`⟩ for each of the above **`if`** statements is a **`sequential`** statement. Therefore, $NP(\langle \texttt{if-range} \rangle) = 1$ for each **`if`** statement. Thus, $NP(\langle \texttt{if} \rangle) = 1 + 0 + 1 = 2$ for each **`if`** statement. $NP(\langle \texttt{code segment} \rangle) = 2 \times 2 \times 2 \times 2 = 16$.

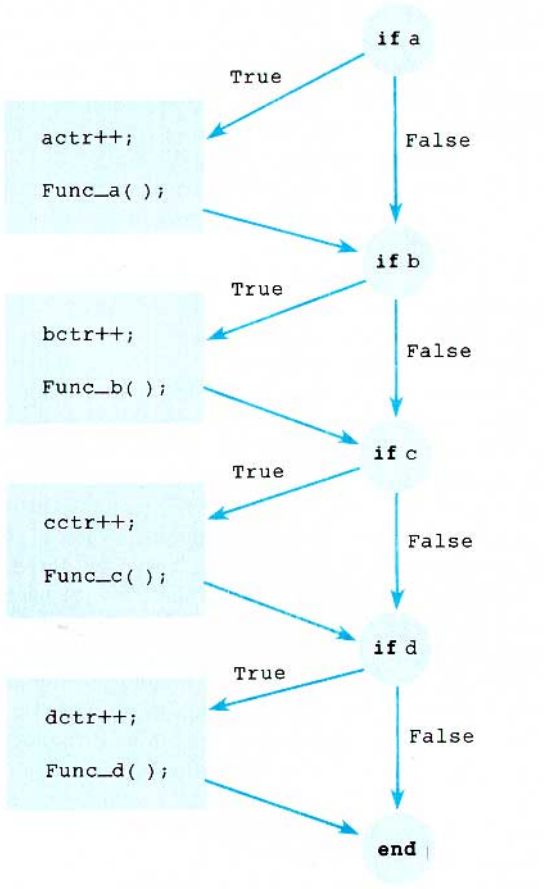

FIGURE 9. Example C Code Segment with NPATH = 16

Characteristics of NPATH

We now demonstrate that NPATH overcomes the shortcomings of McCabe's measure. It was noted earlier that the number of acyclic execution paths in a function varies from a linear to an exponential function of V. NPATH is a measure that is more closely related to the number of acyclic execution paths through a function. In particular, the NPATH measure differs from the actual number of acyclic execution paths by the number of acyclic execution paths resulting from **`goto`** statements. Although the number of acyclic paths resulting from the use of the **`goto`** statement could be significant in theory, in practice the use of the **`goto`** statement is minimal and generally not thought to be good programming practice [3].

McCabe's measure fails to distinguish between different kinds of control flow structures. NPATH, on the other hand, is based on unique expressions of acyclic execution path complexity for each C control flow structure. Thus, the NPATH measure clearly distin-

guishes between different kinds of control flow structures.

Another criticism of McCabe's measure is that it does not account for nesting levels within a function, whereas the NPATH measure does. In particular, the number of acyclic execution paths through a function is dependent, in part, on the level of nesting among statements in the function. That is, acyclic execution path complexity is additive if one statement is nested within another; acyclic execution path complexity is multiplicative if statements are consecutive.

Anomaly of the NPATH Measure

An anomaly arises in the NPATH definition when a certain class of control flow structure sequences appears in a function. Consider the following segment of C source code and its corresponding flow graph (also see Figure 10):

```
if ( A == B )
  S0;
if ( A == B )
  S1;
```

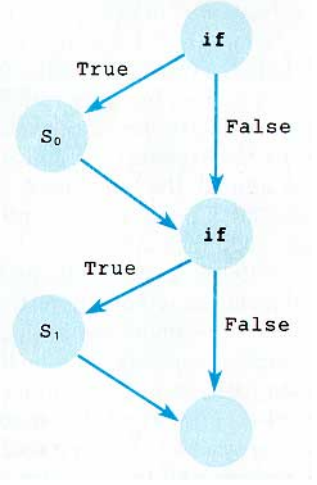

FIGURE 10. Multiple `if` Flow Graph with NPATH = 4

The NPATH measure for this segment of code is 4. In the above segment of code, `S1` is executed *if and only if* `S0` is executed. Therefore, the above segment of code is equivalent to the following segment of code provided `S0` does not alter `A` or `B` (also see Figure 11):

```
if ( A == B )
  {
  S0;
  S1;
  }
```

The NPATH measure for this segment of code is 2. Thus, there are only two unique execution paths possible through the first code segment. The NPATH definition, however, does not detect this anomaly.

Given any two logical expressions ⟨`expr1`⟩ and ⟨`expr2`⟩ governing the execution of two different sequential sequences of code, *if* ⟨`expr1`⟩ *is identical to* ⟨`expr2`⟩ *or* ⟨`expr1`⟩ *is not logically equivalent to* ⟨`expr2`⟩, *and the values of the control variables in* ⟨`expr1`⟩ *and* ⟨`expr2`⟩ *are the same, then the NPATH measure overstates the acyclic execution path complexity by a factor of* 2.

Comparing NCSL, TOKENS, *V*(*G*), and NPATH

In order to assess whether traditional measures of software complexity are closely related to execution path complexity, we computed the following measures of software complexity for 821 functions in a UNIX® C software application.

- *NCSL* is the number of noncommentary source lines of code in a function; that is, any line of program text that is not a blank or comment.
- *TOKENS* is the number of lexical tokens in a function. TOKENS for the C programming language include

  —keywords (e.g., **`while`** and **`if`**),
  —operator symbols (e.g., + and <=),
  —identifiers (e.g., X and Msg), and
  —punctuation symbols (e.g., `(`, `)`, and `;`).

  The TOKENS metric is the basis for the Halstead collection of metrics referred to as Software Science [6].
- *V*(*G*), McCabe's [11] cyclomatic complexity number, represents the number of fundamental circuits in the flow-graph representation of a function. It is the number of logical conditions (**`if`**, **`while`**, **`for`**, **`case`**, **`default`**, **`&&`**, **`||`**, **`?`**) in a function plus 1.

UNIX is a registered trademark of AT&T Bell Laboratories.

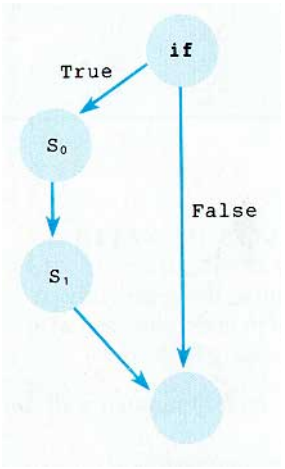

FIGURE 11. Single `if` Flow Graph with NPATH = 2

- *NPATH* is the number of acyclic execution paths through a function.

The correlation matrix shown in Table I summarizes the R^2 correlations among the metrics. R^2 represents the percentage of variance in one variable that is explained by the other variable. Note that the three metrics NCSL, TOKENS, and $V(G)$ are highly correlated; the correlation between these metrics is at least 0.97. These correlations show that NCSL, TOKENS, and $V(G)$ appear to be measuring the same thing, namely, lexical complexity. These three measures do not measure the semantic content of code. Also, when NPATH is correlated to NCSL, TOKENS, and $V(G)$, the resulting correlations are 0.57, 0.53, and 0.56, respectively. These correlations show that NPATH is somewhat independent of the NCSL, TOKENS, and $V(G)$ measures, with at least 40 percent of the variance in NPATH not accounted for by any one of the other measures. Because NPATH is measuring different factors than those measured by NCSL, TOKENS, and $V(G)$, the lower correlations between NPATH and the other measures are expected. These correlations do not suggest that any of the measures is superior to the others, but they do show that NPATH is measuring different factors of complexity than the other measures. The correlation measures show that NCSL, TOKENS, and $V(G)$, which are measuring the lexical content, are not particularly sensitive to the number of execution paths through a function. Thus, if we accept the premise that an important property of software to be measured is the number of execution paths through a function, then these data highlight the importance of the NPATH measure.

TABLE I. R^2 Correlation Matrix for NCSL, TOKENS, $V(G)$, and NPATH

	NCSL	TOKENS	$V(G)$	NPATH
NCSL	1.00	0.99	0.97	0.57
TOKENS	0.99	1.00	0.97	0.53
$V(G)$	0.97	0.97	1.00	0.56
NPATH	0.57	0.53	0.56	1.00

PRACTICAL USES OF NPATH

Many software complexity metrics lack practical value. Measures are often designed without any particular use in mind [8]. Such is not the case with NPATH. Software developers are using NPATH to

- select functions for thorough walk-through/inspection,
- allocate functional testing resources, and
- define module design criteria.

Walk-throughs/Inspections

Code walk-through and inspections have become an integral part of the software development process. When schedule and resource constraints preclude the comprehensive review of all functions, it is important to identify functions that would be most useful to thorough walk-throughs and inspections. Since NPATH measures the functional complexity and testability of code, it follows that NPATH can be used in determining the level of review/inspection of a function. Functions with high (i.e., in the top 25 percent) NPATH values are candidates for thorough review and/or inspection. Thus, NPATH is aiding in the process of deciding which functions should be thoroughly reviewed and/or inspected.

Testing

Software development organizations have limited testing resources. To make good use of limited resources, testing effort might best be allocated to functions proportional to the testability of the function. A widely used criterion in deciding how to apportion testing resources among functions in a software system is the number of NCSL in each function; the larger the NCSL count of a function, the greater the resources allocated to it. Testing a function in proportion to its NCSL count, however, could lead to an inappropriate use of testing resources. This is because there is a weak relationship between the NCSL of a function and the testability of the function, as evidenced by the correlation matrix.

A more appropriate criterion of relevance when apportioning testing resources is the number of unique execution paths through the function. A relatively large number suggests that a relatively large proportion of testing resources should be allocated to the function. The rank order of the functions based on NPATH is being used by developers to allocate testing resources using this criterion. The amount of testing resources allocated to a function is proportional to the number of acyclic execution paths through the function. In short, we agree with others [8] that the extent to which a complexity measure can be used as a guide in testing effort depends on how well the measure specifies what is contributing to the complexity of a program. NPATH is used as a guide in the testing process because it characterizes a significant factor contributing to the complexity of functional testing—the number of acyclic execution paths through a function. Note the NCSL, TOKENS, and $V(G)$ do not capture this property of a function.

Design Criterion

NPATH is also being used to establish a functional design criterion and identify functions appropriate for redesign early in the development process. The NPATH value for a detailed design specification can be computed provided the specification uses the control flow structures of C, such as in the program design language PDL-C [15].

Software quality can be increased by designing software that requires manageable levels of functional testing to assure its correctness. Thus, the testability of each function in a software system is an important design criterion. Along these lines, an NPATH threshold value has been established to define a functional design criterion and identify candidate functions for redesign. An NPATH threshold value of 200 has been established for a function. The value 200 is based on studies done at AT&T Bell Laboratories [14]. For functions that exceed the threshold value of 200, methods to reduce NPATH complexity are provided to developers.

Additional Considerations

Any decision to allocate inspections, testing, or design effort based on NPATH must also take into account

- the criticality of the function: Whereas even moderately high NPATH values in a heavily used function would identify the function for thorough inspection and testing as well as possibly redesign, a noncritical function of similar NPATH complexity might not warrant the same level of attention; and
- factors other than complexity impact on software quality: Requirements volatility, software development environment, developer experience, reuse, and the use of code generators all impact on software quality.

Thus, the use of NPATH cannot provide absolute principles for software development, but is a useful adjunct to traditional and intuitive measures of software complexity.

METHODS TO REDUCE COMPLEXITY

If a method is to be useful in controlling software complexity, then it must index a function's complexity level, as well as suggest ways to reduce complexity [8]. Such is the case with NPATH. Many strategies to reduce the NPATH complexity of functions are being used by software developers. Some of the most effective methods of reducing the NPATH value include

- distributing functionality,
- implementing multiple **if** statements as a **switch** statement, and
- creating a separate function for logical expressions with a high count of **and** (**&&**) and **or** (**||**) operators.

Distributing Functionality

To reduce NPATH for a function, divide the function into blocks of code that logically belong together. Create a new function for each block of code. Then, replace each block of code with a call to the appropriate newly created function. The original functionality is thus distributed, reducing the NPATH value for the original function because function calls are treated as **sequential** statements.

Generally, the more functions defined, the greater the reduction in NPATH. This is not true in all cases; for example, in the case of sequential code, NPATH is not changed by making the sequential code a function and then calling the function.

Multiple if Statements

Another way to reduce NPATH for a function is to implement multiple **if** statements via the **switch** and **case** statements. The following simple example illustrates this strategy: The original sequence of **if** statements

```
if ( c == 'a' )
  ca++;
if ( c == 'b' )
  cb++;
if ( c == 'c' )
  cc++;
if ( c == 'C' )
  cC++;
if ( c != 'a' && c != 'b' && c != 'c' &&
     c != 'C' )
  cOther++;
```

has an NPATH value of 80 (2 × 2 × 2 × 2 × (2 + 3)).

An equivalent, less complex **case** statement implementation of the same sequence

```
switch ( c )
{
  case 'a':
    ca++;
    break;
  case 'b':
    cb++;
    break;
  case 'c':
    cc++;
    break;
  case 'C':
    cC++;
    break;
  default:
    cOther++;
    break;
}
```

has an NPATH value of 5 (1 + 1 + 1 + 1 + 1).

Operators per Logical Expression

NPATH can be reduced for a function with a high count of **and** (**&&**) and **or** (**||**) operators in a logical expression by creating a separate function for the logical expression. Suppose the following logical expression occurs several times in a function:

```
if ((v1 && v2) || ((v3 || v4) && (v5 &&
    v6)) ).
```

The new separate function for the logical expression looks like the following:

```
if ( v_check ( ) )
  :
  :
v_check ( )
{
  /* assuming v1, v2, v3, v4, v5, v6 are
     global to the module */
    return ( (v1 && v2) || ((v3 || v4)
    && (v5 && v6)) );}
}
```

Although strategies to reduce the NPATH complexity of functions are important, care must be taken not to distort the logical clarity of the software by applying a strategy to reduce the complexity of functions. That is, there is a point of diminishing return beyond which a further attempt at reduction of complexity distorts the logical clarity of the system structure.

FUTURE DIRECTIONS AND SUMMARY

NPATH counts the acyclic execution paths through a C function. The practical applications of NPATH and methods to reduce NPATH have been discussed.

The success of NPATH suggests other possible applications. For example, monitoring the coverage of code during the testing process has proved to be an effective method of improving and verifying testing. Most coverage monitors report either (1) the percentage of NCSL executed during code execution, or (2) the percentage of branches executed during code execution. Although both measures of code coverage are useful, in practice, systems that have close to 100 percent NCSL and branch coverage may not be adequately tested. The reason for this is that, although every line of code and branch point in a program might be executed, there could be a substantial number of execution paths in a program that have not been. Path coverage is much more difficult to achieve than branch coverage; branch coverage is much more difficult to achieve than code coverage. Monitoring path coverage would more accurately reveal the completeness of software testing. Future work on developing an NPATH-based coverage monitor is an important next step of this research.

Another potentially useful application of NPATH is in the area of software reliability modeling. A vast majority of software reliability models [13] require a priori estimates for the model parameters T_0 and R. T_0 is the MTTF (mean time to failure) at the start of testing; R is the rate at which MTTF is assumed to increase over time. To date, no acceptable means of estimating T_0 *and* R prior to entering system test has been established. NPATH could be used to develop initial approximations for T_0 *and* R. The basic approach would be to define T_0 complexity classes based on NPATH; as NPATH increases, T_0 would decrease. Table II illustrates this point. Obviously, NPATH ranges and T_0 values need to be found through empirical study.

The MTTF rate (R) *changes* over time; it does not monotonically increase or decrease. Therefore, reliability models need to take into account the dynamic nature of MTTF rates. NPATH and NPATH-coverage monitors could be used to estimate the way MTTF changes over time.

- Let NP_{total} denote the total NPATH for a system.
- Let NP_{exec} denote the number of unique acyclic execution paths executed thus far.
- Let NP_{fail} denote the total number of failures covered thus far.
- The *path failure rate* (*PFR*) at time T is defined as $PFR = NP_{fail}/NP_{exec}$.
- Let NP_{est_fail} denote the estimated number of failures remaining in the software; then $NP_{est\_fail} = PFR \times (NP_{total} - NP_{exec})$.

Both *PFR* and NP_{est_fail} could be extremely valuable in predicting how MTTF changes over time. Note that, as $| NP_{total} - NP_{exec} |$ approaches 0, the software is more completely tested, and the reliability of the system should behave more consistently (i.e., in a near monotonic fashion).

There are also several useful extensions to the NPATH measure. First, the NPATH measure considers any call to a function as a **sequential** statement; function calls do not add complexity. An extension to the current model of software complexity would be to capture the complexity of subsystems and entire systems by accounting for the acyclic execution path complexity of the calling sequences within each function making up the system. Another extension to the model would be to apply the proposed notion of acyclic execution path complexity to hardware, and to define the notion of system complexity as a function of the number of hardware and software execution paths through a system.

Finally, in comparison to code, there is little known about measuring the complexity of requirements and designs. This is partially because programming languages provide a formal notation on which to base measures, whereas formal notations for expressing requirements and designs have only emerged recently. Formal notations such as Structure Charts [18], Booch Diagrams [1], and Data Flow Diagrams [12] now provide computational models and notations on which to base measures of requirement and design complexity. Future software measurement research should focus on defining and analyzing measures of requirements and design complexity based on such notations.

TABLE II. T_0 Complexity Classes Based on NPATH

NPATH range	T_0
1–1000	100
1001–2500	75
2501–5000	60
⋮	⋮

APPENDIX A. NPATH Algorithm

- `Next V` is either (1) the first statement in the compound statement that follows `V`, or (2) `LAST` if it is the last statement in some compound statement.
- `Bool_Comp` of `V` is the complexity of the expressions in statement `V`.

```
NPath ( V )
statement V;
{
if ( V is LAST )
  return ( 1 );
else
  {
  switch ( statement type of V )
    {
    case IFST:   /* if statement */
      return ( (NPATH(if-range of V) + Bool_Comp of V + 1) * NPATH(Next V) );

    case IFEST:   /* if-else statement */
      return ( (NPATH(if-range of V) + NPATH(else-range of V) + Bool_Comp of V)
                * NPATH(Next V) );

    case WHST:   /* while statement */
      return ( (NPATH(while-range of V) + Bool_Comp of V + 1) * NPATH(Next V) );

    case DOST:   /* do statement */
      return ( (NPATH(do-range of V) + Bool_Comp of V + 1) * NPATH(Next V) );

    case FORST:   /* for statement */
      return ( (NPATH(for-range of V) + Bool_Comp of V + 1) * NPATH(Next V) );

    case SWST:   /* switch statement */
    CompSW = Bool_Comp of V;
    for ( each case and default range in switch )
        CompSW = CompSW + NPATH(case-range);
      return ( CompSW * NPATH(Next V) );

    case QUESST:   /* ? statement */
      return ( (Bool_Comp of V + 2) * NPATH(Next V) );

    case GOTO:   /* goto statement */
      return ( NPath(Next V) );   /* skip to next statement */

    case RET:   /* return or exit statement */
    case BRST:   /* break statement */
    case CONTST:   /* continue statement */
      if ( Bool_Comp > 0 )
        return ( Bool_Comp );
      else
        return ( 1 );

    case SEQ:   /* sequential statement */
      if ( Bool_Comp > 0 )
        return ( Bool_Comp * NPATH(Next V) );
      else
        return ( NPATH(Next V) );
    }   /* end of switch statement */
  }   /* end of else */
}   /* end of NPath function */
```

APPENDIX B. A Summary of Execution Path Expressions

Structure	Complexity expression
`if`	$NP(\langle \texttt{if-range} \rangle) + NP(\langle \texttt{expr} \rangle) + 1$
`if-else`	$NP(\langle \texttt{if-range} \rangle) + NP(\langle \texttt{else-range} \rangle) + NP(\langle \texttt{expr} \rangle)$
`while`	$NP(\langle \texttt{while-range} \rangle) + NP(\langle \texttt{expr} \rangle) + 1$
`do while`	$NP(\langle \texttt{do-range} \rangle) + NP(\langle \texttt{expr} \rangle) + 1$
`for`	$NP(\langle \texttt{for-range} \rangle) + NP(\langle \texttt{expr1} \rangle) + NP(\langle \texttt{expr2} \rangle) + NP(\langle \texttt{expr3} \rangle) + 1$
`switch`	$NP(\langle \texttt{expr} \rangle) + \sum_{i=1}^{i=n} NP(\langle \texttt{case-range}_i \rangle) + NP(\langle \texttt{default-range} \rangle)$
`?`	$NP(\langle \texttt{expr1} \rangle) + NP(\langle \texttt{expr2} \rangle) + NP(\langle \texttt{expr3} \rangle) + 2$
`goto` label	1
`break`	1
Expressions	Number of **&&** and **||** operators in expression
`continue`	1
`return`	1
`sequential`	1
Function call	1
C function	$\prod_{i=1}^{i=N} NP(\texttt{Statement}_i)$

Acknowledgments. The author is grateful for the many stimulating conversations he had with Christopher Fox of the Quality Software Technology Group at AT&T Bell Laboratories about NPATH.

REFERENCES

1. Booch, G. *Software Engineering With Ada.* Benjamin/Cummings, Menlo Park, Calif., 1987.
2. Curtis, B., et al. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe Metrics. *IEEE Trans. Softw. Eng. SE-5*, 3 (Mar. 1979), 96–104.
3. Dijkstra, E.W. GO TO statement considered harmful. *Commun. ACM 11*, 3 (Mar. 1968), 147–148.
4. Dunn, R. *Software Defect Removal.* McGraw-Hill, New York, 1984.
5. Evangelist, M. An analysis of control flow complexity. In *Proceedings of IEEE COMPSAC '84.* 1984, pp. 388–396.
6. Halstead, M. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.
7. Johnson, S. *YACC: Yet Another Compiler Compiler.* Bell Laboratories, Murray Hill, N.J., 1975.
8. Kearney, J.K., Sedlmeyer, R.L., Thompson, W.B., Gray, M.A., and Adler, M.A. Software complexity measurement. *Commun. ACM 29*, 11 (Nov. 1986), 1044–1050.
9. Kernighan, B., and Ritchie, D. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1978.
10. Lesk, M., and Schmidt, E. *LEX: A Lexical Analysis Generator.* Bell Laboratories, Murray Hill., N.J., 1975.
11. McCabe, T. A complexity measure. *IEEE Trans. Softw. Eng. SE-2*, 4 (Apr. 1976), 308–320.
12. Mellor, S., and Ward, P.T. *Structured Development for Real-Time Systems.* Yourdon Press, New York, 1986.
13. Musa, J. Software reliability modeling. In *Handbook of Software Engineering*, C. Vick and C. Ramamoorthy, Eds. Van Nostrand Reinhold, New York, 1984.
14. Nejmeh, B. Software complexity metrics study summary. Tech. Memo., AT&T Bell Laboratories, Holmdel, N.J., Oct. 1986.
15. Nejmeh, B., and Dunsmore, H. A survey of program design languages (PDLs). In *Proceedings of IEEE COMPSAC '86* 1986, pp. 447–456.
16. Paige, M. An analytical approach to program testing. In *Proceedings of IEEE COMPSAC '80.* 1980, pp. 527–531.
17. Rapp, S., and Weyuker, E. Selecting software test data flow information. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr. 1985), 367–375.
18. Stevens, W.P. *Using Structured Design.* Wiley-Interscience, New York, 1981.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.2.4 [**Software Engineering**]: Program Verification—*reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*restructuring*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; D.2.9 [**Software Engineering**]: Management—*software quality assurance (SQA)*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*control primitives*

General Terms: Algorithms, Design, Management, Measurement, Reliability

Additional Key Words and Phrases: Execution path complexity, NPATH, software testing

Author's Present Address: Brian A. Nejmeh, SPC, 1880 North Campus Commons Drive, Reston, VA 22091.