Franz Stetter

A measure of program complexity
We propose a measure of program complexity which takes into account both the relationship between statements and the relationships between statements and data objects (constants and variables). This measure, called program flow complexity, can be calculated from the source text of a program in an easy way.

Index terms: Complexity measures, software metrics, program graph.
1. Introduction

Complexity measures of programs have been related to the following concepts:

- the cyclomatic complexity /McCa76/ is based on the notion of program graph and is defined as the cyclomatic number of that graph,
- in software science /Hals77/ the notion of programming effort may be viewed as a measure of complexity,
- the number of program knots /Wood79/ is a useful measure for unstructured programs only (thus, we do not refer to it in what follows),
- the logical complexity /Iyer82/ measures the complexity of the relationship amongst both the instructions and the memory cells in a program.

Obviously, the cyclomatic complexity does not take data flow into account in any way, but data flow certainly contributes to complexity. On the other hand, the programming effort does not distinguish between various kinds of control structure but a loop is certainly more complex than an if-then-else clause. Baker /Bake79/ has shown that the drawbacks of cyclomatic complexity and programming effort can be eliminated in a synthesized program complexity measure.

The logical complexity measure /Iyer82/ makes use of the variable dependency of sequence of computation, inductive effort in writing loops, and complexity of data structures. Essentially, the dependence of a computation at a node upon the computation of other nodes is described with the aid of a graph.

In this paper we extend the notion of program graph by taking into account the constants and variables used in the program. Thus, we view programs to consist of operations (statements) and data (constants and variables), and both components contribute to the complexity measure of a program. It is shown that the measure proposed is more appropriate than the measures mentioned above.
2. Cyclomatic complexity

Since the complexity measure to be proposed may be viewed as an extension of the cyclomatic complexity measure /McCa76/, it is useful to describe the cyclomatic complexity briefly.

Statically, a program \( P \) consists of a sequence of statements \( s_i \), \( 1 \leq i \leq n \), i.e.

\[
(2.1) \quad P = (s_1, s_2, \ldots, s_n).
\]

The program graph \( G = (N, E) \) of \( P \) consists of a set \( N \) of nodes and a set of \( E \) of edges where each node \( n_i \in N \) represents a statement \( s_i \). An edge \( e_{ij} = (s_i, s_j) \) indicates that, after the execution of statement \( s_i \), the control of execution can pass immediately to statement \( s_j \). Without loss of generality, we assume that every node in \( G \) lies on some path from an entry node to an exit node. The cyclomatic complexity \( C(P) \) /McCa76/ is defined to be the number of linearly independent paths through the corresponding program graph of \( P \), i.e.,

\[
(2.2) \quad C(P) := e - n + 2
\]

for a single-entry/single-exit program, or more generally /Stet83/:

\[
(2.3) \quad C(P) := e - n + n_s + n_t,
\]

where \( e = |E| \) : number of edges,
\( n = |N| \) : number of nodes (= statements),
\( n_s \) : number of entry nodes,
\( n_t \) : number of exit nodes.

For a strongly connected graph \( G \), the cyclomatic number \( c(G) \) which expresses the number of linearly independent circuits is given by \( c(G) = e - n + 1 \). Equation (2.2) and (2.3) can be derived easily by adding nodes and edges to the program graph so that the number of linearly independent paths does not change and the program graph becomes strongly connected.
Now, if we look at the outgoing edges of a node of the program graph we see that the number of outgoing edges indicates the number of branches originating in that statement. Let $b(s_i)$ be the number of branches at statement $s_i$. Then we get

$$e = \sum_{i=1}^{n} b(s_i) = n + \sum_{i=1}^{n} (b(s_i)-1)$$

which yields in (2.3).

$$C(P) = \sum_{i=1}^{n} (b(s_i)-1) + n_s + n_t.$$  

The term $(b_i(s)-1)$ vanishes for all nodes $n_i$ which have one outgoing edge only.

Equation (2.5) can be simplified for D-structured programs (which have a small set of control structures only):

- sequence: $b(s) = 1$,
- alternative: $b(s) = 2$,
- loop: $b(s) = 2$.

Since $b(s_i) = 0$ for all exit nodes, we get from (2.5).

$$C(P) = \text{number of alternatives} + \text{number of loops} + n_s$$

for D-structured programs (if we include the case-statement branching $N$-times, we have $b(s) = N$; in this case (2.6) has to be modified slightly).
3. Flow Graph

From the equations above we see some serious drawbacks to the notion of cyclomatic complexity:

- It is \( C(P) = 1 \) for a linear sequence of any length but experience tells us that the complexity of a linear sequence increases with its length.

- The conditions which control the branching may be very "complex" expressions but the cyclomatic complexity does not reflect this fact.

- The cyclomatic complexity of an if-then-else statement and of a while-do loop have the same value. But again, experience tells us that a loop is more complex than an alternative.

In order to remedy these deficiencies of the cyclomatic complexity we extend the notion of a program graph, arriving at the notion of a flow graph.

Observing that a program consists of both declarations \( d_j \), \( 1 \leq j \leq k \), and statements \( s_j \), \( 1 \leq j \leq m \), we now view a program statically as a sequence of declarations and statements, i.e.,

\[
(3.1) \quad P = (d_1, d_2, \ldots, d_k, s_1, \ldots, s_m).
\]

In what follows we assume that each \( d_j \) declares one variable or one constant only (thus, the number of variables and constants of \( P \) equals the number of declarations).

The flow graph \( H = (N, E) \) of a program \( P \) consists of a set \( N \) of nodes and a set \( E \) of edges where each node \( n \in N \) represents a declaration \( d_j \) or a statement \( s_i \). In what follows a node representing a statement (declaration) is called a statement (declaration) node. The number of nodes equals the sum of the number of constants, variables, and statements.
An edge $e_{gh} = (n_g, n_h)$ is defined in the following three cases:

a) If $n_g$ and $n_h$ are statement nodes, $e_{gh}$ indicates that control can switch immediately from the statement represented by $n_g$ to the statement represented by $n_h$.

b) If $n_g$ is a statement node and $n_h$ a declaration node, $e_{gh}$ indicates that there may be a write access to the variable represented by $n_h$ during the execution of the statement represented by $n_g$.

c) If $n_g$ is a declaration node and $n_h$ a statement node, $e_{gh}$ indicates that there may be a read access to the constant or variable represented by $n_g$ during the execution of the statement represented by $n_h$.

In other words: an edge from a statement node to a statement node indicates the possible flow of control of execution, an edge from a statement node to a declaration node shows that the value of the corresponding variable may be changed during the execution of that statement, and an edge from a declaration node to a statement node indicates a possible read access. Note that there are no edges from declaration nodes to declaration nodes. Thus, an edge from a node $n_i$ to a node $n_j$ indicates that node $n_j$ is immediately "influenced" by node $n_i$. Paths through the flow graph show the flow of information during the execution of the program.

Without loss of generality, we assume that each node lies on some path from an entry node to an exit node. It may be noted that both statement nodes and declaration nodes can be entry or exit nodes.

The number $n_s$ of entry nodes of $H$ consists of three components:

- the number $n_{sc}$ of constants used in the program $P$,
- the number $n_{sv}$ of variables referenced by read only (these are the input parameters whose values are not changed in $P$),
- the number $n_{ss}$ of entry statements which do not have read references to data.
Thus, we have the relationship

\[ n_s = n_{sc} + n_{sv} + n_{ss}. \]

Analogously, we conclude that the number \( n_t \) of exit nodes of \( H \) has two components:

- the number \( n_{tv} \) of variables referenced by write only (these are the output parameters otherwise not used in \( P \))

- the number \( n_{ts} \) of exit statements which do not have write references to variables.

Thus,

\[ n_t = n_{tv} + n_{ts}. \]
4. CYCLOMATIC FLOW COMPLEXITY

As for the cyclomatic complexity based on the program graph, we now define the *cyclomatic flow complexity* $F(P)$ to be the number of linearly independent paths through the flow graph $H$ of a program $P$ (confer (2.3)):

\[(4.1) \quad F(P) := e - n + n_s + n_t,\]

where
\[e : \text{number of edges of } H,\]
\[n : \text{number of nodes of } H (= \text{number of constants, variables, and statements}),\]
\[n_s : \text{number of entry nodes of } H,\]
\[n_t : \text{number of exit nodes of } H.\]

Equation (4.1) can be transformed in such a way that we do not have to refer to the flow graph but only to the program itself. We denote by

\[b(s) : \text{number of branches originating at a statement node},\]
\[r(d) : \text{number of statements where the data object represented declaration } d \text{ is referenced by a read access},\]
\[w(d) : \text{number of statements where the data object represented by declaration } d \text{ is referenced by a write access},\]
\[a(d) := r(d) + w(d).\]

As for (2.4) we can express $e$ in (4.1) in terms of $b(s)$ and $a(d)$:

\[(4.2) \quad e = \sum_{i=1}^{m} b(s_i) + \sum_{j=1}^{k} a(d_j)\]
\[= n + \sum_{i=1}^{m} (b(s_i) - 1) + \sum_{j=1}^{k} (a(d_j) - 1),\]

so we get from (4.1), (3.2), and (3.3).
\[ F(P) = \sum_{i=1}^{m} (b(s_i)-1) + \sum_{j=1}^{k} (a(d_j)-1) + n_{sc} + n_{sv} + n_{ss} + n_{tv} + n_{ts}. \]

\( F(P) \) can be counted easily by looking at the source code of the program \( P \) (so we do not actually have to plot the flow graph).
5. Example

We illustrate the notion of cyclomatic flow complexity by means of the following simple example.

```pascal
program EUCLID;
var A, B, GCD, H : integer;
begin
  read (A, B); GCD := 0;
  if A > 0 and B > 0 then
    begin
      GCD := A; H := B;
      while GCD ≠ H do
        if GCD > H then
          begin
            GCD := GCD - H;
          end
        else
          begin
            H := H - GCD;
          end;
    write (A, B, GCD)
  end.
end.
```

We immediately see:

\[ b(s_1) = b(s_2) = b(s_4) = b(s_5) = b(s_9) = b(s_{10}) = 1, \]
\[ b(s_3) = b(s_6) = b(s_7) = 2, \quad b(s_{4,0}) = 0, \]
\[ a(A) = 4, \quad a(B) = 4, \quad a(GCD) = 8, \quad a(H) = 6, \quad a(0) = 2, \]
\[ n_{sc} = 1, \quad n_{sv} = 0, \quad n_{ss} = 1, \quad n_{tv} = 0, \quad n_{ts} = 1. \]

Thus,

\[ F(P) = 24. \]
6. FEATURES OF THE CYCLOMATIC FLOW COMPLEXITY

The notion of cyclomatic flow complexity has been derived by means of theoretical considerations. Of course, we now have to examine how it fits with reality. To do this we look at some features of this notion.

a) A major drawback of the cyclomatic complexity is the fact that a linear sequence has complexity 1 regardless of the number of elements in the sequence. This feature contradicts empirical evidence that the complexity of a linear sequence increases as a function of the number of elements in that sequence.

For the cyclomatic flow complexity, looking at equation (4.3) we see that for any linear sequence:

\[ F(P) = \sum_{j=1}^{k} (a(d_j) - 1) + n_{sc} + n_{sv} + n_{ss} + n_{tv} + n_{ts} - 1. \]

If we increase the length of sequence by one statement, the number of accesses to data objects (6.1) increases in almost all cases we can imagine. On the other hand, if we increase the number of declarations, \( F(P) \) increases too since \( a(d_j) \geq 1 \).

b) Intuitively, a loop is more complex than an alternative (the cyclomatic complexity yields the same value for both program constructs):

(6.2) LOOP: while \( B \) do
   \hspace{1cm} begin ... \( B := ... \) end;

(6.3) ALTERNATIVE: if \( B \) then \( S_1 \) else \( S_2 \);

Since in (6.2) the variable \( B \) is referenced (at least) twice, we get \( F(\text{LOOP}) > F(\text{ALTERNATIVE}) \).
(Of course, it is assumed that the loop body and the statements \( S_1 \) and \( S_2 \) have the same complexity).
c) Myers /Myer77/ contends that any adequate program complexity measure should produce the ordering (6.4)<(6.5)<(6.6) for the complexity of the following three program segments:

(6.4) \[ \text{if } x=0 \text{ then } S_1 \text{ else } S_2; \]
(6.5) \[ \text{if } (x=0) \land (y=1) \text{ then } S_1 \text{ else } S_2; \]
(6.6) \[ \text{if } x=0 \text{ then } \]
\[ \text{if } y=1 \text{ then } S_1 \text{ else } S_2 \]
\[ \text{else } S_2; \]

The cyclomatic complexity \( C(P) \) yields the numbers (2,2,3) if one counts entire predicates and it yields (2,3,3) if one counts each simple predicate as one decision. In either case, the complexity is equal for two of the three cases. But, with respect to cyclomatic flow complexity we get a different picture. We see that \( F(6.4) = F(6.5) - 2 < F(6.5) \). Furthermore, we have \( F(6.5) = F(6.6) - 2 < F(6.6) \). Thus, the requirement of Myers is met.

d) The number of constants and variables contributes to the cyclomatic flow complexity in addition to the number of references from and to a variable. However, reaching the same data object several times within one statement does not increase the complexity.
7. **Conclusion**

The measure of cyclomatic flow complexity can be viewed as an extension of the cyclomatic complexity measure. Whereas the cyclomatic complexity is based on the control structures only, the cyclomatic flow complexity takes into account the flow of the data in the program as well. This is achieved by an extension of the well-known notion of program graph. The program flow complexity measure does not have some of the serious deficiencies of other complexity measures. Furthermore, it is in line with our experience since it yields the following:

- the complexity of a linear sequence increases as a function of the number of statements,
- a loop is more complex than an alternative,
- the number of variables contributes to the complexity,
- the number of references from and to a variable is taken into account.

Finally, the program flow complexity can be calculated from the source text of the program in an easy way.


