S/C: a Software Size/Complexity Measure

Y.R. Pant, J.M. Verner and B. Henderson-Sellers

a MacDonald Dettwiler Pty. Ltd., Sydney
b Department of Information Systems, City Polytechnic of Hong Kong
c School of Computing Systems, University of Technology, Sydney

Abstract

A new software structural complexity metric is proposed which is a simple, yet powerful, extension of the normal lines of code count. This measure is shown to account for the differences in complexity of program statements in high-order languages and is useful in assessing control complexity of source programs objectively when the source code is available. The proposed measure also satisfies the seventh axiom of Weyuker's software complexity measures (complexity is dependent on permutation of source statements). The measure has the potential (not fully evaluated here) for use in assessing the effort related to understanding software for software maintenance, debugging and generalization. In our pilot project, we found that the time taken to trace artificial bugs introduced in programs is explained better by S/C counts than by lines of code counts.

Keyword Codes: D.2.8, D.2.5
Keywords: Metrics, Testing

1. INTRODUCTION

The lines of code measure does not account for the complexity of an individual line of code. This may reduce the effectiveness of the lines of code measure as a representation of structural complexity on which models of external characteristics such as understandability may be based.

Program understandability and maintainability/generalizability are parallel concepts. The more difficult a program is to understand, the more difficult it is to enhance/modify. A program fragment has to be understood and analyzed fully before it can be generalized and enhanced. Program analysis consists of an examination of source code to find out what the program is supposed to do and how various program segments are connected together. Without a proper understanding of a program segment, it may be difficult to predict or detect the ripple effects of change within the program and affecting other programs (although the reverse is not true — we may understand the segment and still not be able to predict e.g. maintenance costs and ripple effects). Both intra- and inter-module structures may well affect the cost of software maintenance. There are several factors (in addition to the following) affecting program maintenance costs:

- control flow: the consequences of executing each path of the program logic.
- program structure types (e.g. procedural, control, data and input/output);
- data flow, their origin, aliases and uses; and
Table 1
Classification of C++ control keywords.

<table>
<thead>
<tr>
<th>Class</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class I</td>
<td>break, case, continue, default, return, switch, throw</td>
</tr>
<tr>
<td>Class II</td>
<td>do, else, for, goto, if, while</td>
</tr>
</tbody>
</table>

1.1 Control complexity

Control statements specify the order in which a computation is carried out. Traditionally, cyclomatic complexity was used to assess control flow complexity of programs, especially for providing a "basis set" for testing. The cyclomatic number of a single-entry, single-exit program is equal to the total number of individual decisions (of the program) plus one. The problem with this approach is that the complexity of different control statements constructed with the keywords like if/else, while, for and switch is deemed to be the same. However, intuitively, some of these statements/keywords (SKs) are inherently more complex than others. Furthermore, in cyclomatic complexity, program text excluding control statements do not add any complexity; for extrapolation to program comprehension this is frequently inappropriate.

In a high-order programming language, decision-making and iterative statements are normally more complex than assignment statements. Ledgard and Marcotty discuss the relative power of control structures. They introduce the idea of a hierarchy of difficulty of understanding different executable statements. Iyengar et al. classify control statements into three categories: sequencing, conditional and repetitive. They conclude that the repetitive statements are the most logically complex structures, since they demand the values of the variables during the ith iteration to be expressed recursively in terms of the values of the variables at the (i - 1)th iteration.

There are 13 control keywords in C++: break, case, continue, default, do, else, for, goto, if, return, switch, throw and while. Some of these SKs do not take any optional parameters (e.g. continue, break) while others take a varying number of parameters (e.g. if, for) in different contexts. Thus (inherently) different control keywords have differing complexity. Furthermore, the SKs e.g. if and for (taking a varying number of parameters) also group a number of statements to make a block of code.

A block of code is similar to a subprogram but without a name. In some sense, the abstraction mechanisms provided by a subprogram and a block of code can be considered to be equivalent. Thus, the SKs e.g. if and for are at a higher abstraction level than the SKs e.g. break and continue. Therefore, to assess control complexity, it may be necessary to analyze the specific uses of SKs. Based on these observations, we classify (Table 1) the control keywords of the C++ programming language into the following two categories:

- **Class I**: those that do not take any optional parameters or only fixed parameters (e.g. switch) and
- **Class II**: those that take a varying number of parameters.

We think that Class II SKs carry more 'functionality' or contain more 'logical stuff' (and hence are more difficult to comprehend) than Class I SKs. In order to avoid the problems in straightforward application of SLOC counts and cyclomatic complexity (discussed by many authors), complex statements realized with Class II keywords should be broken into simple or primitive statements for which the effort for understanding is approximately the same.

Indeed, the classification of C++ keywords is difficult as the members of Class I and Class II can overlap. For example, the keyword goto does not introduce any repetitions nor conditions, and supports a low-level programming style. Since the goto only sequences the flow of control, it should have been a member of Class I. However, it may introduce an arbitrary transfer of control that may span a large amount of program text. As a result, the program may be difficult to understand and maintain since immediacy of resolving variable dependencies is lost. This makes it a suitable member of Class II.

Since a for loop allows grouping of statements like declarative, assignment, comparison and arithmetic, we have placed it into Class II. The following for loop

```cpp
for (int i = 0; i <= 50; i++) {
    /* Repeats 50 times */
}
```

(E1)
has a declaration/assignment statement, comparison and arithmetic statement and takes three parameters. These parameters can have wide ranges of values which makes some for statements much more complex than others. For instance, to create an infinite loop (E2), we do not specify any parameters:

```c
for (;;) {
    /* An infinite loop */
}
```

(E2)

The following statement (E3) has two declaration/assignment, two comparison and two arithmetic statements. Because of the differences in the number of parameters, (we believe that) this for loop (E3) will take greater effort to understand and modify than the previous for loops (E1 & E2).

```c
for (int i = 0, j = 500; i <= 50, j >= 70; i++, j--) {
    /* A more complex loop */
}
```

(E3)

1.2 The repetitive and conditional constructs

There are three repetitive statements in C++/C: for, while and do/while. As they are generally equivalent, only the for loop will be discussed in detail here. All the loops have three parts: expr1, expr2 and expr3.

for loop:
```c
for (expr1; expr2; expr3) {
}
```

(E4)

while loop:
```c
expr1;
while (expr2) {
    expr3;
}
```

(E5)

do/while loop:
```c
do{
    expr1;
    expr3;
}while (expr2);
```

(E6)

Generally, expr1 and expr3 initialize, increment or otherwise change some variables that are used by the loop; expr2 is normally a conditional statement that is used for controlling the repetition of the loop.

```c
for (int i = 0; i < 10; i++) {
    /* do something */
}
```

(E7)

To illustrate the method of obtaining the S/C measure, we consider the for loop (E7). We can think of this loop as consisting of the following four parts:
- initialization: int i = 0
- comparison: i < 10
- increment: i++
- binder: for (; ;){ ... }

Thus, the S/C counts of the loop (E7) is four. We have named the group of elements for (; ;){ ... } as a "binder". The binder logically groups the first three parts (int i = 0, i < 10 and i++) together. We consider that a binder contributes one unit of S/C. However, it is possible to assign varying subjective weights (similar to that of ref 12) to the control keywords.

Had there been two comparisons (E8) like:

```c
for (int i = 0; i < 10 && j > 2; i++) {
    /* For loop with two comparisons */
}
```

(E8)

then the S/C counts would be seven because the two comparisons can have four outcomes, creating four distinct execution (or mental) paths. This should be reasonable because E8 is intuitively more difficult to construct, comprehend and debug than E7, as all the four distinct paths must be traversed mentally. In other words, because of the four possible (different) execution paths, E8 requires more effort (for understanding, for maintenance and for testing) than E7.
The proposed model differs from the complexity measure which is simply a count of the total number of individual conditions (not decisions) plus one whilst having some similarity with the NPATH measure.

A more complex for loop (E9) is shown below:

```c
for (int i = 0; ((i < lim - 1) && ((c = getchar()) != EOF) && (c != '8')); i++) {
  /* More effort is required to understand this loop */
}
```

In this loop, the initialization and assignment parts are no different to those discussed above (e.g. E7, E8). However, the decision making part is complex and has the three conditions \( i < \text{lim} - 1 \), \( (c = \text{getchar}()) \neq \text{EOF} \) and \( c \neq 8 \). The three comparisons are anded together such that they have 8 different outcomes. With the keyword for and the statements \( \text{int } i = 0 \) and \( i++ \), the S/C count of this line is 11.

```c
if (a < b) {
  /* A simple if statement */
}
```

The simple if statement (E10) can be broken into two parts:

- comparison: \( a < b \)
- binder: if (){}

Thus it contributes 2 units of S/C.

### 1.3 Nested statements

Nested statements have been a source of considerable debate in the context of software complexity metrics. The following nested if (E11)

```c
if (expr1) {
  if (expr2) {
    s2;
  }
}
```

can be theoretically considered as a subset of (E12)

```c
if ((expr1) logical-operator (expr2)) {
  /* logical-operators can be & & or || */
  s2;
}
```

when the logical-operator is &&, and commonalities can be traced. For example, in the C language, the predicates are always evaluated from left to right, and evaluation stops as soon as the truth or falsehood of the result is known. In this case, if the first predicate (expr1) evaluates to be false, the second will not be evaluated. The same criterion is also true in the nested if (E11); the second if statement will not be evaluated if the first predicate evaluates to be false. In other words, for the statements s2 to be executed, both the predicates must evaluate to true. We can also consider a logical and (&&) to be implicit in the first nested construct.

One of the important differences between the nested and the anded construct is that the execution history of the statements s1 may alter the evaluation of the second predicate, thus causing the statements s2 to be executed. McCabe's metric fails to account for differences such as this. In McCabe's measure, each conditional expression in a complex Boolean expression adds the same level of difficulty as an entire conditional structure. McCabe's measure does not have any provisions for distinguishing between a series of nested if statements and a single (equivalent) if statement in which all the conditions are grouped together.

The differences can be accounted for by identifying the property that a nested construct has one more if and has more static properties than its corresponding anded version. Neglecting s1 and s2 and assuming that expr1 and expr2 each has only one predicate, the S/C counts of the first construct is 6; while that of the second is 5.

As a second example, the statements equivalent to the following construct (E13) is 4.
if (expr1) {
}

if (expr2) {
}

while (((c = getchar()) != EOF) {
    s1;
    if (a >= b) {
        /* do something */
    }
}

In E17, the execution of the inner if loop is controlled by the outer while loop. When programmers attempt to understand the inner if block, they should consider the effect of the variable 'c' on the condition 'a >= b'. For example, the statements s1 might affect the values of the variables 'a' and 'b' as a result of different values of 'c'. The programmers might think that if the variable 'c' is true and the variable 'a' is greater than or equal to the variable 'b' then only the code inside the inner if loop will be executed. This is similar to logical anding (&&) of 'c' and (a <= b) which creates four execution paths (total S/C counts = 6).

Thus, this model is in accordance with our intuitive understanding that a module gets more complex if there are more embedded or nested control statements in it.

2. EXPERIMENTAL SUPPORT

For our experiment, we selected two programs (Figure 1) each of 41 source lines of code in C. The S/C counts of the first program was 50; while that of the second was 81.

The first program computes some simple statistics (mean, standard deviation, maximum and minimum value) from a set of data. The second program prompts for a number to be entered and validates it. If a user enters characters other than the digits (including one decimal point), the number will not be accepted and the user is prompted again to enter a correct real number.

2.1 Experiment

As the 'effort for understanding a program' is a difficult measure to quantify, we sought to measure the time required to trace bugs artificially introduced in the programs. The buggy statements are shown within a framebox for both the programs (Figure 1). Different types of bugs were introduced in the programs to reduce the likelihood of formulating similar debugging strategies (in the subsequent program) when the subjects found the bug(s) in the initial program.

- In the ‘statistical’ program (Figure 1(a)), the statement for (i = 0; i < size; i++) was changed to for (i = 1; i <= size; i++) and the statement sum = 0.0 was deleted.
- In the 'input validation' program (Figure 1(b)), the statement only_digits = 1 was changed to only_digits = 0; the statement if (!isdigit(buffer[i])) was changed to if(!isdigit(buffer[i])

Let \( \mu \) be the mean difference of the time taken to debug the ‘input validation’ and the ‘statistical’ program, then

- Null hypothesis: \( H_0 : \mu = 0 \),
- Alternative hypothesis: \( H_a : \mu > 0 \).

The null hypothesis states that there is no difference between the debugging times, and \( H_a \) states that the ‘input validation’ program takes more time to debug (its S/C counts are greater) than the ‘statistical’ program.

The debugging order of the two programs was randomized by tossing a coin. The programs were given to 15 subjects, who were asked to record exactly the time taken to trace the bugs. The subjects were either currently working on C projects or had previously developed programs in C. The subjects were closely monitored without any interference while they were fixing the bugs.
# include <stdio.h>
# include <math.h>
# define MAXSIZE 500

void compute_stat(void)
{
    int i, size;
    float sum = 0.0, temp, avg, stddev, min, max, data[MAXSIZE];
    printf("Enter data size : ");
    scanf("%d", &size);
    printf("Enter data separated by a space : ");
    for (i = 0; i < size; i++)
    {
        scanf("%f", &data[i]);
    }
    max = min = data[0];
    for (i = 0; i < size; i++)
    {
        if (data[i] > max)
        {
            max = data[i];
        }
        if (data[i] < min)
        {
            min = data[i];
        }
        sum = sum + data[i];
    }
    avg = sum / size;
    stddev = sum / size;
    stddev = sqrt(stddev);
    printf("mean = %5.2f std. dev. = %5.2f max = %5.2f, min = %5.2f\n", avg, stddev, max, min);
}

int main()
{
    compute_stat();
    return 0;
}

Figure 1 (a) ‘Statistical’ program; and (b) ‘Input validation’ program.

2.2 Results and interpretation

The results are shown in Table 2. The first column assigns an identification number to the different subjects. The second column indicates the time (in minutes) spent to trace the bugs of the program in Figure 1(a) while the last column indicates the time needed to find the bugs of the program in Figure 1(b). In columns 2 and 3, the letter N within the brackets indicates that the corresponding subject did not identify the correct answer.

A program may be difficult to debug until it is fully understood. In addition, the ‘understanding’
component may take a major portion (up to 70%) of the total debugging time. Thus we assume that debugging time is highly correlated with time for understanding a piece of code. This means that it would be incorrect to ignore the pairs and analyze the data as if we had two samples, one for the 'statistical' and the other for the 'input validation' programs. The difference in debugging time of programs in Figure 1(a) and 1(b) was significant (Wilcoxon signed ranks test, $T^+ = 115$, no. of subjects = 15). For a sample size of 15, it is not unreasonable to use a normal approximation with a mean of $n(n+1)/4 = 60$ and a variance of $n(n+1)(2n+1)/24 = 310$. This gives a normal test statistic (z-value) of 3.10 which is statistically significant at the 1% level ($Z_{crit.} = 2.516$ for a two-tailed test).

The 'input validation' program (Figure 1(b)) had $U = 1.62$ times more units of S/C than the 'statistical' (Figure 1(a)) program although both had the same size in terms of lines of code (41 LOC). The ratio of average values of debugging time for the two similarly sized (in LOC) programs was $1.48$. These results are encouraging, albeit from a small pilot project. Further large scale empirical experiments are planned as a full validation of the newly proposed S/C metric.

3. CONCLUSIONS

In high-order programming languages, program statements (constructed even with the same keyword) can have a varying degree of "complexity". In this paper, an objective size/complexity measure is proposed which is sensitive to the complexity of individual control statements as well as the length of the code ("number of lines") and nesting levels.

A pilot study undertaken in this paper indicates that the S/C metric is highly correlated with the effort (time) for finding artificially introduced bugs in C programs (than with the source lines of code measure). A large scale experiment is planned as a full validation of the newly proposed S/C metric.

The proposed metric can be useful for assessing the control complexity of program statements for the purpose of software maintenance, generalization, debugging and testing. The proposed metric may also be more suitable than the normal lines of code for quantifying the effort related to understanding a piece of code for a number of post-delivery activities and, incidentally, can be shown to satisfy Weyuker's seventh axiom.

4. ACKNOWLEDGEMENTS

We wish to acknowledge the support of a Small Research Grant (grant number C450.301) from the Australian Research Council. We also wish to thank Graham Tate of City Polytechnic of Hong Kong for his useful comments on an earlier draft of this manuscript.

REFERENCES

11. LIPPMAN, S., 'C++ Primer', (Addison-Wesley, Reading, Massachusetts, 1991)