A STUDY OF A FAMILY OF STRUCTURAL COMPLEXITY METRICS

Victor R. Basili
David H. Hutchens
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

A family of structural complexity metrics which contains a number of current metrics is developed. The family may be used to give a framework for experimental analysis of metrics. By implementing the family or a suitable subset as an automatic metric tool, many metrics become readily available and may even be merged to form new metrics in response to information obtained during exploratory analysis.

CR Category: 4.6

Introduction

Many people have made attempts at quantifying the complexity of computer programs. A good complexity metric could be used as a quality assurance test by software developers and could even be used as part of the contractual obligation. Current complexity metrics, based upon the physical attributes of the software product, fall into three basic categories: volume, control organization, and data organization. Each of these categories will be discussed briefly below.

Volume metrics are measures of the size of the project; for example, the number of lines of code, the number of statements, or the number of operators and operands [Halstead]. The cyclomatic complexity [McCabe] of structured programs, when viewed as the number of decisions plus one, can be considered a member of this category. Other metrics which relate to volume include the number of variables, the number of procedures, the average length of procedures, and the number of input/output formats [Carriere & Thibodeau]. Note that these are measures of the logical size, rather than just the physical size, of the programs.

Control organization metrics are measures of the comprehensibility of the control structures. Thus cyclomatic complexity, when viewed as the number of control paths, is also a control metric. The average nesting level has been shown to be a useful control organization metric [Dunsmore]. Essential complexity [McCabe], which will be discussed later, falls in this category as well.

Data organization metrics are measures of the data visibility and use and the interactions between data within the program. Data Binding [Basili & Turner; Stevens, Myers & Constantine] is an example of a module interaction metric. PCB [McClure] concentrates on those interactions which affect the flow of control. Spans [Elloff] and Slicing [Weiser] are concepts which relate closely to data complexity.

A Family of Control Structure Metrics

None of the above metrics has gained widespread acceptance as a good measure, i.e., one which can be used for quality assurance and contractual obligations, for two reasons. First, there is a lack of experimental evidence to determine what aspects of the system life cycle the metric actually explains. The metric could, in fact, correlate well with debugging time and say little about the difficulty of maintenance. Thus the experimental evidence should be focused on the intended use or uses of the metric. Second, the metrics are static (non-parameterized) so they cannot be tuned to the results of exploratory analysis.

In light of the second comment, it is reasonable to consider defining a parameterised family of complexity metrics. Although one would like to include elements from each of the three categories, the current research has concentrated on volume and control concepts with the hope of later including the data measures or treating them in a separate family.

The structural complexity family should incorporate length, nesting, control paths, types of control structures, and ease of understanding the decomposition. The family should transcend language. Various numbers should emphasize different aspects of software development and maintenance.

The length could have been measured by lines of code, with or without comments. However, in a free format language this measure can be altered by cosmetic revisions of the code, so the number of statements is a more consistent measure. The nesting factor should be included as a multiplier for each construct at a given level. Control paths and types of structures are closely related and are handled in a variety of ways by current metrics. The ease of understanding the decomposition is intended to measure the relative difficulty (a maintenance) programmer encounters when he must understand the structure of control.

With these concepts in mind, a recursive definition of a family of control structure complexity metrics could be given by

\[ c(p) = 1 + \sum c(p_i) + f(n, t, s) \]

where \( p \) is a program which is decomposed in some fashion into \( k \) components \( p_1, p_2, \ldots, p_k \). The parameter \( b \) is used to generate the multiplier for nesting level. The function \( f \) is the key to the measure. It has four arguments: \( n \), the number of decisions in program \( p \) which are not part of a particular subcomponent; \( t \), the nesting level of component \( p \); \( s \), the type of structure instantiated by \( p \); and \( a \), the structural "niceness" of \( p \).

Some discussion of, and restrictions on, the parameter will clarify their meaning. \( b \) is intended to penalize nesting \( n > 1 \), where \( b = 1 \) obviously removes \( f \) from the formula. Since an increase in the number of decisions should not decrease the complexity, \( f \) should be a non-decreasing function of \( n \). At first glance, one might be tempted to place a non-decreasing condition on \( f \) with respect to the level. However, there is reason to believe that a concave up function of \( n \) may be better [Dunsmore]. An example
will be given later.

It should be noted that \( b \) is in fact superfluous, for the metric

\[
c(p) = b + \frac{1}{2} c(p') + \frac{1}{2} c(p) + f(n, t, i, s) =
\]

\[
\frac{1}{2} c(p) + f'(n, t, i, s) \quad \text{where} \quad f'(n, t, i, s) = b' +
\]

\[
f(n, t, i, s). \] Here \( b \) is reduced to a constant in the

function \( f' \). The definition is stated the way it is

because the explicit inclusion of \( b \) helps clarify the

next penalties. Indeed, many instantiations will

drop \( b \) instead of \( b \).

The parameters \( t \) and \( s \) may appear redundant, but

they have different purposes. The values of \( t \)

normally range over syntactic entities, such as while,

case, and if statements. On the other hand, \( s \) is used

to answer the question "Is this structured program-

ning?" A more precise statement of this question will

be given, but some background must be presented first.

The control flow of a program may be described by

digraph. A program (equating the program and its
digraph) is called a proper program if it has a

single entry, a single exit, and every node of the

program lies on some path from the entry to the exit.

A proper program is called a prime program if it con-
tains no proper subprograms with two or more nodes.

Some common prime programs are the usual while do

and if then else fi. A prime decomposition is found

by continually replacing prime subprograms by function

nodes (a node with a single entry and a single exit).

A proper program has a unique prime decomposition

if sequences are treated as a unit (Linger, Mills & Witt).

By letting the parameter \( s \) have the two values

1) proper and 2) not proper, the resulting (sub)family

is given by:

\[
c(p) = b + \frac{1}{2} c(p') + f(n, t, i, s) \quad \text{p proper}
\]

\[
= \frac{1}{2} c(p) + f(n, t, i, s) \quad \text{p not proper.}
\]

This restricted family will be the subject of the rest

of this paper. If the decomposition of \( p \) into \( p_1 \), \( p_2 \),

..., \( p_k \) is in some sense reasonable, it would be

expected that proper programs would be easier to under-

stand than non-proper programs. Thus it is assumed

that \( f(n, t, i) \leq g(n, t, i) \) for all \( n, t, i, \) and \( t \).

Some Members of the Family

Consider the member obtained by letting \( b = 1 \) and

\( f(n, t, i) = g(n, t, i) = n \), where the subcomponents are
determined by prime program decomposition. Note that
at each level of the recursion each of the \( p_i \)'s are
determined by prime decomposition and are, therefore,
necessarily proper. Hence, the \( g(n, t, i) \) branch is never
used. The measure is just

\[
c(p) = \frac{1}{2} c(p) + n
\]

and eventually each decision will be counted exactly

once. Therefore, the member is just the cyclomatic

complexity minus one.

Essential complexity is defined as the cyclomatic

complexity minus the number of subprograms in the prime

decomposition (ignoring sequences). Thus, we may use

a sibling of the previous measure where

\[
f(n, t, i) = \begin{cases} 0 & n = 0 \\ 1 & n \neq 0 \end{cases}
\]

minus one. Note that for a program constructed accord-
ing to standard structured programming techniques,
this measure is zero and the essential complexity is
one, since each prime program will have either zero
or one decision node.

The decomposition of \( p \) into \( p_1, p_2, \ldots, p_k \) can
be based on the syntactic structure of the language.

One major benefit of this approach is the ease with

which a compiler can be changed into an automatic

metric tool. As a simple example, consider the de-

composition of programs into statements (and state-

ments into substatements) where

\[
c(p) = \begin{cases} \frac{1}{2} c(p) + 1 & \text{p a statement} \\ 0 & \text{otherwise.} \end{cases}
\]

Note that this uses the \( t \) parameter of the family.

The resultant measure is nothing more than a state-

ment count volume metric.

In fact, by methods similar to the above, many

of the volume metrics can be derived. If we count

expressions and appropriate subparts, we get

\[
c(p) = 2c(p) + \begin{cases} 1 & \text{p an expression,term,\ldots,identifier} \\ 0 & \text{otherwise} \end{cases}
\]

which is Halstead's operator-operandom count. It is also

easy to see that a statistics to get an identifier
count.

By combining the proper,not proper distinction with

syntactic decompositions at the statement level

(grouping sequences), an interesting measure may be

derived. It is assumed that the language allows nested

statement constructs such as the usual while and if

Then the member defined by

\[
c(p) = \frac{1}{2} c(p) + \begin{cases} 0 & \text{p proper} \\ n & \text{p not proper} \end{cases}
\]

is very similar to essential complexity where complex

predicates are treated as a single decision. This

measure counts the number of decisions in all state-

ments which are abnormally exited (e.g., with a GOTO)

while essential complexity would sometimes subtract

one for each exit. For example, consider the follow-

ing program:

\[
1: \quad i := 1
\]

while \( i \leq \text{max} \) do

begin

if \( A[i] \) = key then goto 1 ;

\[
i := i + 1
\]

end

1: \quad \text{return}

The essential complexity is the number of decisions

minus the number of programs in the prime decomposi-

tion, which is 2+1 (dropping the plus one from

McGabe's definition). The other terms of the above

measure gives \( c(p) = c(\text{while}) + 0 = c(\text{if}) + 1 + 1 = 2 \).

The difference occurs because the code begins with

the while and ending with the return forms a prime

program even though it is not a syntactic statement.

The point of this digression is that essential com-

plexity can be approximated using the statement decom-

position motif. The next two examples also use the

statement decomposition idea.

For a set of small programs, it was found that

those with a central average nesting level tended to

have fewer program changes during development

[Buneman]. The family may generate metrics which

describe this phenomenon. For example, if

\[
c(p) = \frac{1}{2} c(p) + 1 + 2
\]

then the metric need not be positive. The interpreta-

tion would be that a measure close to zero is good

14
while those on either side are progressively worse. Note that this metric does achieve the quality of averaging, at least if we divide the result by the number of statements. On the other hand, the measure could directly penalize deviation from "best" depth by using $|z - 2|$ or perhaps $|z - 2|^2$. These are examples of concave up functions with respect to $z$ which were mentioned earlier.

The last example of the members of the family follows:

$$c(p) = 1.10c(p) + \left\lceil \frac{\log_{10} n + 1}{2} \right\rceil$$

This member exhibits some of the flexibility of the family. The $b$ value of 1.1 penalizes nesting by counting each statement 10% more than it would be at the next outer level. Furthermore, poorly structured code costs twice as much as well structured code. Each statement must contribute at least one to the measure due to the addition of 1 in each of the functions $f$ and $g$. The use of the logarithm encourages the use of case statements, the only standard control structure with more than one decision node. Thus, this metric includes consideration of nesting level, length (statement count), structured programming practices, and bonuses for use of the organizing construct (the case statement).

Experimenting with the Family

There are two ways to use the structural complexity family in the analysis of software engineering. The first is for testing the correlation of a given program property with some software development or maintenance aspect. Given the property in question, one must develop the family member which depicts the property. An experiment must be performed to measure the development or maintenance aspect and then the metric is calculated for the program used in the experiment. Standard statistical methods may then be used to determine the correlation.

The second use of the family has been mentioned before. The family may be viewed as a function from parameters into metrics. Given a software engineering aspect and data from an experiment, the parameters are manipulated in an attempt to maximize the correlation between the resultant metric and the aspect. This is exploratory data analysis. The analyst must be careful not to become excessively detailed in the parameter changing as the result of the analysis is limited by the accuracy of the data. Having determined a candidate metric, it should be tested against new data in a standard confirmatory experiment.

Current research takes the second approach where the aspect being studied is program changes during development. The number of changes has been shown to be closely related to the number of errors [Danmore & Gannon]. The structural complexity family with proper versus not proper statement distinctions has been implemented in the SIMPL-T [Basili & Turner] compiler. SIMPL-T is a GO-TO-less non-block language which allows statement nesting. Loops may be abnormally exited using the EXIT statement and RETURNS are allowed at any point. SIMPL-T is used in many courses at the University of Maryland. The experiment data was collected from class projects ranging in size from small projects to larger programs. The string manipulation routines are being used for exploratory analysis. The analysis will begin by considering some of the metrics mentioned previously (e.g., cyclomatic complexity, essential complexity, number of statements) and others. The insights gained from this preliminary analysis will then be used to generate other members of the family. The best candidates will then be tested on the compilers.

The compilers were written under three different development methodologies: ad hoc individuals, ad hoc teams, and disciplined teams. Many metrics have already been tested to see if they detect the differences in methodologies [Basili & Reiter]. The metrics which have already been analyzed included, among others, statement counts (broken into types), variables declared (global, local and parameter), and Data Findings. This work will be continued with the structural complexity family.

Conclusions

A family of structural complexity metrics has been defined which encompasses many of the current metrics. Much work remains to be done in comparing and evaluating the various members of the family. This evaluation will be based on the correlation with aspects of the development cycle, in particular, program changes and team organization.

References


