7-7

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 subfamily 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. PCAM [McClure] concentrates on those interactions which affect the flow of control. Spans [Elshoff] 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 parameterized 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 members 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

$$c(p) = b\sum_{i=1}^{k} c(p_i) + f(n, \ell, t, s)$$

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; ℓ, the nesting level of component p; t, the type of structure instantiated by p; and s, the structural "niceness" of p.

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

It should be noted that b is in fact superfluous, for the metric $c(p) = b \sum_i c(p_i) + f(n,\ell,t,s) = \sum_i c(p_i) + f'(n,\ell,t,s)$ where $f'(n,\ell,t,s) = b^\ell + f(n,\ell,t,s)$. 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 nexting penalties. Indeed, many instantiations will drop ℓ 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 programming?" 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 a digraph. A program (equating the program and its digraph) is called a <u>proper program</u> 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 <u>prime program</u> if it contains no proper subprograms with two or more nodes. Some common prime programs are the usual <u>while</u> <u>do</u> <u>od</u> and <u>if</u> <u>then</u> <u>else</u> <u>fi</u>. A <u>prime decomposition</u> 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:

$$c(p) = b\sum_{i=1}^{k} c(p_i) + \begin{cases} f(n,\ell,t) & ; \text{ p proper} \\ g(n,\ell,t) & ; \text{ p not proper.} \end{cases}$$

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 understand than non-proper programs. Thus it is assumed that $f(n,\ell,t) \leq g(n,\ell,t)$ for all n, ℓ, and t.

Some Members of the Family

Consider the member obtained by letting b = 1 and $f(n,\ell,t) = g(n,\ell,t) = n$, where the subcomponents are determined by prime program decomposition. Note that at each level of the recursion each of the pi's are determined by prime decomposition and are, therefore, necessarily proper. Hence, the $g(n,\ell,t)$ branch is never used. The measure is just

$$c(p) = \sum_{i=1}^{k} c(p_i) + n$$

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

$$f(n,\ell,t) = \begin{cases} n-1 & n>0 \\ 0 & n=0 \end{cases}$$ to obtain the essential complexity

minus one. Note that for a program constructed according 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, ..., 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 decomposition of programs into statements (and statements into substatements) where

$$c(p) = \sum_{i=1}^{k} c(p_i) + \begin{cases} 1 & ; \text{ p a statement} \\ 0 & ; \text{ otherwise.} \end{cases}$$

Note that this uses the t parameter of the family. The resultant measure is nothing more than a statement 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

$$c(p) = \Sigma c(p_i) + \begin{cases} 1 & ; \text{ p an expression,term,...,identifier} \\ 0 & ; \text{ otherwise} \end{cases}$$

which is Halstead's operator+operand count. It is also possible to count declarations 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 <u>while</u> and <u>if</u>. Then the member defined by

$$c(p) = \sum_{i=1}^{k} c(p_i) + \begin{cases} 0 & ; \text{ p proper} \\ n & ; \text{ p not proper} \end{cases}$$

is very similar to essential complexity where complex predicates are treated as a single decision. This measure counts the number of decisions in all statements which are abnormally exited (e.g., with a GO TO) while essential complexity would sometimes subtract one for each exit. For example, consider the following program:

```
i := 1
while i < max do
    begin
        if A[i] = key then goto ℓ ;
        i := i + 1
    end
ℓ:  return
```

The essential complexity is the number of decisions minus the number of programs in the prime decomposition, which is 2-1=1 (dropping the plus one from McCabe's definition). On the other hand, the above measure gives c(p) = c(while) + 0 = c(if) + 1 = 1 + 1=2. The difference occurs because the code beginning 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 complexity can be approximated using the statement decomposition 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 [Dunsmore]. The family may generate metrics which describe this phenomenon. For example, if

$$c(p) = \sum_{i=1}^{k} c(p_i) + \ell - 2$$

then the metric need not be positive. The interpretation would be that a measure close to zero is good 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 $|\ell-2|$ or perhaps $(\ell-2)^2$. These are examples of the concave up functions with respect to ℓ which were mentioned earlier.

The last example of the members of the family follows:

$$c(p) = 1.1\Sigma c(p_i) + \begin{cases} 1+\log_2(n+1) & ;\ p \text{ proper} \\ 2*(1+\log_2(n+1)) & ;\ p \text{ not proper.} \end{cases}$$

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 programs 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 analyzer 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 [Dunsmore & Gannon]. The structural complexity family with proper verses not proper statement distinctions has been implemented in the SIMPL-T [Basili & Turner] compiler. SIMPL-T is a GOTO-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 experimental data was collected from class projects ranging in size from string manipulation routines to small compilers. 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 Bindings. 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

[Basili & Reiter] V. R. Basili & R. W. Reiter, Jr., "An Investigation of Human Factors in Software Development," Computer Magazine, Dec. 1979, pp. 21-38.

[Basili & Turner] V. R. Basili & A. J. Turner, SIMPL-T A Structured Programming Language, Paladin House Publishers, Geneva, Ill., 1976.

[Carriere & Thibodeau] W. M. Carriere & R. Thibodeau, "Development of A Logistics Software Cost Estimating Technique for Foreign Military Sales," General Research Corporation, Santa Barbara, California, June '79.

[Dunsmore] H. E. Dunsmore, "The Influence of Programming Factors on Program Complexity," Ph.D. diss., Dept. of Computer Science, University of Maryland, July '78.

[Dunsmore & Gannon] H. E. Dunsmore & J. D. Gannon, "Experimental Investigation of Programming Complexity," Proc. ACM-NBS Sixteenth Annual Technical Symposium: Systems and Software, Wash., D. C., June '77, pp.117-125.

[Elshoff] J. L. Elshoff, "An Analysis of Some Commercial PL/1 Programs," IEEE-TSE June 1976.

[Halstead] M. Halstead, Elements of Software Science, Elsevier Computer Science Library, 1977.

[Linger, Mills & Witt] R. C. Linger, H. D. Mills, B. I. Witt, Structured Programming: Theory and Practice, Addison-Wesley, Reading, Mass. 1979.

[McCabe] T. J. McCabe, "A Complexity Measure," IEEE-TSE Vol.], No. 4, Dec. 1976, pp. 308-320.

[McClure] C. L. McClure, "A Model for Program Complexity Analysis," 3rd International Conference on Software Engineering, May 1978, pp. 149-157.

[Stevens, Myers & Constantine] W. P. Stevens, G. J. Myers & L. L. Constantine, "Structured Design," IBM Systems Journal, Vol. 13, No. 2, 1974, pp. 115-139.