An Empirical Study of a Syntactic Complexity Family

VICTOR R. BASILI, MEMBER, IEEE, AND DAVID H. HUTCHENS

Abstract—A family of syntactic complexity metrics is defined that generates several metrics commonly occurring in the literature. The paper uses the family to answer some questions about the relationship of these metrics to error-proneness and to each other. Two derived metrics are applied; _slope_ which measures the relative skills of programmers at handling a given level of complexity and _r square_ which is indirectly related to the consistency of performance of the programmer or team. The study suggests that individual differences have a large effect on the significance of results where many individuals are used. When an individual is isolated, better results are obtainable. The metrics can also be used to differentiate between projects on which a methodology was used and those on which it was not.

Index Terms—Control structure metrics, development methods, program changes, software experiments, software metrics, structural complexity, syntactic complexity.

I. Introduction

As COMPUTER scientists attempt to understand the software process and product, it is natural to try to measure those aspects of software that seem to affect cost. A major problem in computer science is the intellectual control of design that is directly related to the complexity of the product. Many attempts at quantifying the complexity of computer programs have been made [2], [8], [10], [13], [15], [18]. A good complexity metric could be used as a quality assurance test by software developers and even as a contractual obligation. Current complexity metrics may be roughly divided into two basic groups: 1) static metrics that are measures of the product at one particular point in time and 2) history metrics that are measures of the product and process taken over time. Static complexity metrics are based on the physical attributes of a software product. These fall into three basic categories: *volume*, *control organization*, and *data organization*. Each of these categories will be discussed briefly below. This paper will deal predominately with the volume and control subclasses of static complexity metrics.

Some *volume* metrics are measures of the size of a product: the number of lines of code, the number of statements, or the number of operators and operands [15]. The software science volume metric is in this group. Even cyclomatic complexity [18] can be placed in this category since it is the number of decisions plus one. The number of procedures, the average length of procedures, and the number of variables are examples of volume metrics. The number of input/output formats [7] and other abstraction metrics are volume metrics as well. Note that these latter metrics are measures of the logical size, rather than just the physical size, of a program.

Manuscript received October 3, 1981; revised July 14, 1983. This work was supported in part by the U.S. Air Force Office of Scientific Research under Contract AFOSR-F49620-80-C-001. Computer support was provided in part by the Computer Science Center at the University of Maryland.

V. R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

D. H. Hutchens was with the Department of Computer Science, University of Maryland, College Park, MD 20742. He is now with the Department of Computer Science, Clemson University, Clemson, SC 29631.

Control organization metrics are measures of the comprehensibility of control structures. Thus cyclomatic complexity, when viewed as the number of control paths, is also a control metric. Knots [24] and Maximal Intersection Number [8] attempt to measure the control complexity by visual properties of program control, either as it is written (in a computer language) or as it would appear in a planer flow graph. Average nesting level has been shown to be a useful control organization metric [11]. Essential complexity [18] falls in this category as well. Control organization metrics may also include interprocedural metrics such as calling level and distinct calls [6].

Data organization metrics are measures of data use and visibility as well as the interactions between data within a program. Data binding [4], [21] is an example of a module interaction metric. A span [14] is an attempt to measure the proximity of references to each data item. As such it qualifies as a data organization metric. Slicing [23] can also be considered a data organization metric. A slice is that (not necessarily consecutive) portion of code that is necessary to produce some prescribed partial output from the program.

As seen above, it is not always clear which category a particular metric belongs to. For example, we may view cyclomatic complexity as a volume or a control metric depending upon the desired emphasis.

II. Definition of a Structural Metric Family

The above metrics have failed to gain full acceptance as valid measures of program complexity for at least two reasons. First, there is a lack of experimental evidence to determine what aspects of the system life cycle the metric explains. While a metric could correlate well with debugging time, it might still be a poor predictor of the effort required to do maintenance. We need experimental evidence that is focused on the expected uses of the metrics. Second, existing metrics are not parameterized and thus cannot be tuned to the results of exploratory analysis.

A complete development of the structural family of complexity metrics may be found in [1]. The structural family includes many metrics from the volume and control organization groups. The data organization group is a subject for other

If the family is to include many of the metrics in the literature it must incorporate length, nesting level, control paths, types of control structures, and decomposition simplicity. The family should transcend languages (although specific members may not). The various members may relate to many aspects of software development and maintenance although any one metric may only be useful in a limited way.

Length can be 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 seems to be a more consistent measure. Nesting level might be included explicitly or as a factor to be multiplied with the complexity of the lower levels. Control paths and types of control structures are closely related and are handled in a variety of ways by current metrics, so the family must allow a general mechanism for these concepts. Decomposition simplicity is intended to measure the naturalness with which the intended function is broken into smaller functions.

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

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

where p is a program that is decomposed in some fashion into k components $p1, p2, \cdots, pk$. The parameter b generates the multiplier for nesting level. The function f, the key to the metric, has four arguments: n, the number of decisions in program p that are not part of any subcomponent pi; lev, the nesting level of component p; t, the type of structure instantiated by p; and s, the structural "niceness" of p. The range of f is the positive real numbers.

Some discussion of, and restrictions on, the parameters will clarify their meaning. b is intended to penalize nesting so $b \geq 1$; where, of course, $b = 1$ just 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 lev. However, there is reason to believe that a concave up function (one that falls at first and later rises) of lev may be better [11]. An example may be found in [1].

It should be noted that b is superfluous for the metric

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

$$= \sum_{i=1}^{k} c(pi) + f'(n, lev, t, s)$$

where

$$f'(n, lev, t, s) = b^{lev} f(n, lev, t, s).$$

In this example, b is reduced to a constant in the function f'. The use of the constant b makes the penalties more explicit than does hiding that information in the function. Indeed, many instantiations may use b instead of lev.

The values of t normally range over syntactic entities, such as assignment, while, case, and if statements. The parameter s describes whether the control structure is "structured" or "nonstructured."

The control flow of a program may be described by a digraph. A program (equating the program and its digraph) is called a *proper program* if it has a single entry and 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 contains no proper subprograms with two or more nodes. The usual *while do od* and *if then else fi* are examples of common prime programs. 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 successive sequences are treated as a unit [17].

By letting the parameter s have the two values 1) proper and 2) not proper, the resulting (sub)family is given by

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

Both f and g are functions from (INTEGER × INTEGER × VOCABULARY) to REAL, where VOCABULARY is that of the grammar for the language. This restricted family will be used throughout the rest of this paper. If one assumes that proper programs are less complex than nonproper programs then $f(n, lev, t) \leq g(n, lev, t)$ for all n, lev, and t. The restricted family might reasonably be called a *syntactic complexity* family since it is based on the syntactic decompositions of the program.

The syntactic complexity family covers most of the volume metrics and some of the control flow metrics in the literature. It does not contain any aspects of data organization.

III. Some Members of the Family

One major benefit of basing the decomposition on the syntactic structure 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

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

Note that this uses the t parameter of the family. The resultant measure is nothing more than a count of the executable statements (STMT), a member of the volume subfamily of metrics.

The call count (CALL), the number of calls to any procedure or function whether user defined or language predefined, is easily produced as another member of the volume subfamily as follows:

$$c(p) = \sum_{i=1}^{k} c(pi) + \begin{cases} 1; & p \text{ a func or proc call} \\ 0; & \text{otherwise.} \end{cases}$$

Likewise, the decision statement count (DecS) is another volume metric.

$$c(p) = \sum_{i=1}^{k} c(pi) + \begin{cases} 1; & p \text{ an IF, WHILE, or CASE} \\ 0; & \text{otherwise.} \end{cases}$$

Cyclomatic complexity ($v(G)$) may be generated by adding the number of decisions to the number of segments [18]. The measure is

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

Eventually each decision will be counted exactly once. Therefore, the member is just the cyclomatic number of the program, a member of the control organization subfamily of metrics. Note that this formulation uses predominately the n parameter.

For a final example, consider this more complex member of the family:

$$c(p) = 1.1 \sum_{i=1}^{k} c(pi) + \begin{cases} 1 + \log 2(n+1); & p \text{ proper stmt} \\ 2 * (1 + \log 2(n+1)); & p \text{ not proper stmt.} \end{cases} \tag{1}$$

This member exhibits some of the flexibility of the family. The b value of 1.1 penalizes nesting by counting each statement 10 percent more than it would be at the next outer level. Furthermore, poorly structured code is penalized twice as much as well structured code. Each statement must contribute at least 1 to the measure since 1 is added in each of the functions f and g. The use of the logarithm encourages the use of *case* statements. Thus, this metric includes consideration of nesting level, length (statement count), structured programming practices, and bonuses for use of an organizing construct (the case statement). This metric, which we will call Syntactic Complexity (SynC), is a hybrid of volume and control organization families.

Several other members of the family, including essential complexity and the software science count of total operators and operands, are derived in [1].

IV. Experimentation

This research focuses on the ability of product metrics to explain the number of program changes made during development as well as the differences in the metrics caused by different development strategies. Given the above family of syntactic metrics one would like to 1) evaluate their use in specific environments and 2) analyze and compare members of the various subfamilies.

In the first case a set of questions to be asked might include the following. Are the metrics useful in measuring or predicting the error proneness of programs? Are they effective in predicting the effort that goes into program development? Are they useful in characterizing methodological approaches? Are they useful in evaluating the software development process and product?

Questions generated by the second concern include the following. Are there any differences in these subfamilies? Are they all measuring the same thing? Is one member of a subfamily better than others under some set of conditions? Are the instances of anomalies between measures an indication of error proneness or extra effort? Which are useful in evaluating and characterizing methodological approaches? Does one have to go outside this family to find metrics which capture different aspects of complexity?

It is impossible to answer all of these questions within the scope of this paper and on a single database of small programs. However, this paper will present experimental evidence for evaluating and classifying metrics.

In order to investigate the error proneness, the program changes made during the development of the projects have been counted. A *program change* [12] is defined to be either a textual change to one or more adjacent lines of source code, or the insertion of one or more lines of source code with the following exceptions. An insertion together with an adjacent changed line of code is considered to be only one program change. The insertion of output statements is not considered a program change as this activity is usually concerned with temporary debug code. The deletion of code is not considered a program change as the code is usually either moved (in which case the insertion is counted) or the code is debug code being removed after it is no longer needed. Changes and insertions of comments are not counted as program changes. Program changes have been shown to be closely related to the number of errors made during development [12].

The syntactic complexity family has been implemented in the SIMPL-T compiler [5]. SIMPL-T is a GOTO-less nonblock structured language that 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; therefore, the experiment participants were fairly familiar with the language.

The research reported in this paper uses a database of 19 compilers written by upperclassmen and graduate students. The compilers were written under three different development methodologies: ad hoc individuals (AI), ad hoc teams (AT), and disciplined teams (DT). Each team consisted of three students. The ad hoc individuals and ad hoc teams were not given any particular methodologies or techniques to be used in the implementation. They were free to organize their work in any way they desired. The disciplined teams were required to use a list of methodologies and techniques that were taught in their class. These methodologies included chief programmer teams, walkthroughs, and top down design with PDL, among others. Several metrics have already been tested to see if they detect the differences among the groups [2], [3].

The results reported here deal with the metric defined in (1) (SynC), statement count (STMT), call count (CALL), cyclomatic complexity ($v(G)$), and decision statement count (DecS). We will focus on the relationship between these metrics and the program changes made during the project development.

TABLE I
CORRELATION MATRIX FOR THE PRODUCT METRICS

	STMT	SynC	CALL	$v(G)$	DecS
STMT	1.000				
SynC	0.975	1.000			
CALL	0.845	0.770	1.000		
$v(G)$	0.879	0.893	0.747	1.000	
DecS	0.873	0.939	0.617	0.832	1.000

A. The Effects of Individuals

In attempting to validate the various metrics as useful predictors of error-proneness, we compared each metric against the number of program changes for each project. That is, the number of program changes for each project was determined, and the metrics were computed on each segment (procedure or function) in each project. The complexities of the individual segments were combined into a project complexity in several different ways. These included summing the complexity of each segment as well as summing only the most complex segments (such as the top 10 or 20 percent). All these attempts at correlating the number of program changes to the complexity of the project were unsuccessful, i.e., no significant correlation was found.

The five metrics considered here are highly correlated as may be seen in the correlation matrix in Table I. Thus, multiple regression equations tended to be erratic, with the coefficients changing greatly with the addition of new variables, while producing minimal increases in r square. Therefore, the rest of this paper deals only with simple regression equations.

A second, more detailed analysis was made of the error-proneness of the metrics. Each of the complexities of the individual segments in one project was correlated to the number of program changes made in that particular segment. Appendix I contains the coefficients of determination (r square) and the slope of the lines for each of the projects and each of the metrics using simple regression analysis [19] on the metrics with the dependent variable program changes.

Examining only the six projects that were developed by ad hoc individuals, the coefficient of determination (r square) for SynC as a predictor of program changes ranged between 0.475 and 0.866. The other metrics had slightly lower values but a similar spread (see Appendix I). Therefore, when an individual is isolated, it appears that these metrics do correlate well with the number of program changes. For an example plot, see Fig. 1.

It is somewhat surprising that a linear fit does better for almost all cases with respect to both r square and the distribution of the residuals than a regression based on log-log transformations which yields an exponential curve in the original data. Many have argued that segments should be made small to control their complexity. An exponential fit would imply that the argument is valid, since the sum of the complexities of several small segments would be much smaller than the complexity of one larger segment with the same amount of code. However, a linear fit implies that there is no advantage to splitting a large segment into many smaller segments unless duplication of code could be reduced.

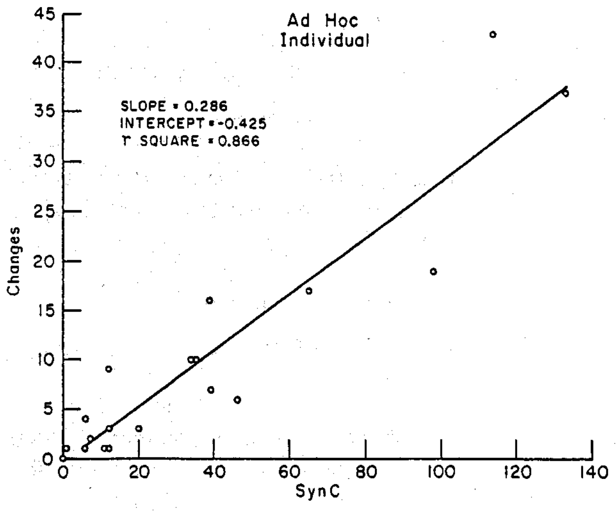

Fig. 1.

The 19 projects did fit linearly for all five metrics. Only a couple of projects yielded minor improvement using log-log transformations (exponential fits). The straight lines intersect close to the origins; therefore, the poor fit of the exponential is not caused by missing the low valued points due to forcing the curve through the origin.

It is possible that the linear model appears to fit best because the segments are so small (the average "maximum segment size" for the 19 projects is 66 statements). The exponential tail might appear if there were larger (more complex) segments. It is also possible that programmers naturally limit themselves to smaller segments where they can handle the complexity level.

More interesting, however, is that the slopes of the fitted lines varied from 0.16 to 0.73 for SynC (see Fig. 2). Similar variation exists for the slopes of the other metrics. The slope of the line may be viewed as a measure of a programmer's ability to cope with complexity since it estimates the number of program changes he makes in developing a program for each unit of complexity. This interpretation is possible because the intercepts of the regression lines are close to zero. It is the variation in the slopes that accounts for the lack of results using several projects produced by different people.

Experimentation that combines the work of different people is likely to contain a large amount of noise resulting from individual differences among participants. This phenomenon alone may be the cause of many failed experiments.

B. Slope Metrics

In general, the slope of the regression line is not sufficient to determine which of two results is better. The intercept may also play a role. For example, if one individual has a high slope but a low intercept, he may still have produced code while making fewer program changes than a person with an average slope but a much higher intercept. In the data for this study the intercepts were all close to zero, and what little variation did exist tended to be in the same direction as the differences in slope.

Using the slope to indicate a programmer's ability to cope with complexity gives hope of producing an experiment that

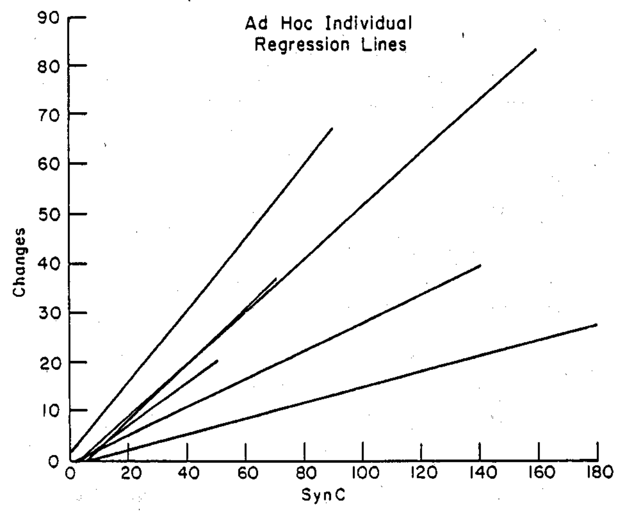

Fig. 2.

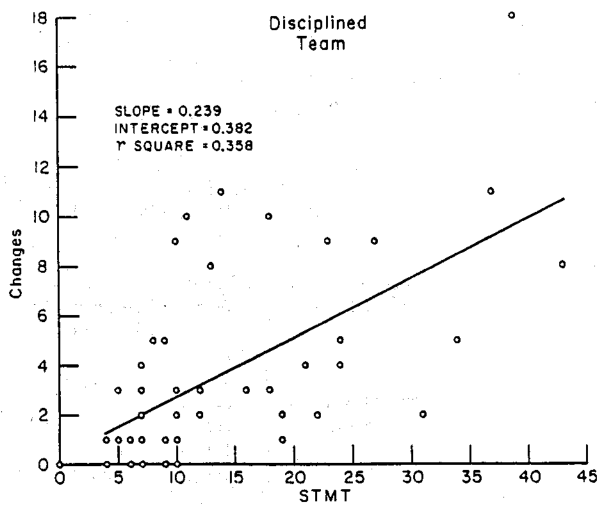

Fig. 3.

can quantify a programmer's limitations with respect to the complexity of various applications. The results might be used for management decisions such as assignment of tasks to different programmers.

The results presented here, however, do not give a total picture of the individual's ability to cope with complexity. One complexity metric is not powerful enough to represent the difficulty of the task.

Since a single complexity metric is unlikely to cover all aspects of complexity, it may be possible for a programmer to shift the difficulty of development to unmeasured aspects of the program, i.e., to the data structure if the metric is a volume metric. A vector of metrics (and corresponding slopes) might give a better indication of the ability of the programmer to cope with complexity. Such a vector may be useful in determining how to allocate the available programmer resources so that each is working on problems where the complexity is expected to be of the type that he is most capable of handling.

One advantage of a slope metric is its independence of the specification (as long as the specification is not changing during development). Note that in this experiment, the specification for each of the segments in a given product is different. It therefore might be possible to take measurements from the regular work of the programmers over a long period of time and avoid the construction of a special experiment. Thus the programmers will not need to be specifically aware of the experiment so their performance would not be affected by any reactions to the experimental situation.

The benefit of a derived metric like slope, might still be realizable even if the fits are nonlinear. For example, if the relationship is exponential, the value of the exponent might provide a measure of the programmer's limitations. The use of metrics in the evaluation of programmer's ability to copy with complexity is an area that warrants considerable research attention.

C. Comparison of Methodologies

The five metrics were used to compare the different groups of teams. This part of the study uses the Kruskal-Wallis test and the Mann-Whitney U test [20] to determine if a particular group appears to have a better slope and/or coefficient of determination than another. For an example of the difference, compare Fig. 1 with Fig. 3. Note that the slope of the line has units of changes per unit of complexity. Thus the larger the slope, the more changes made in the face of a given level of complexity and (supposedly) the less effective in handling complexity the methodology or programmer that produced it. Statistically, the coefficient of determination is a measure of the amount of variation in the dependent variable (program changes) that may be explained by the variation in the independent variable (the product metric). That is, a high coefficient of determination leaves less variability to be accounted for by other factors; individual differences in particular. Thus we would expect them to be higher for ad hoc individuals than for teams.

It also acts as a measure of uniformity in team efforts. Under the hypothesis that methodology makes a group act more like an individual with respect to consistency, one would expect that disciplined teams would have a coefficient of determination that is slightly lower (less consistent) than ad hoc individuals but larger (more consistent) than ad hoc teams. The results appear in Tables II and III. The CALL metric does not appear in these tables because none of the statistics are significant with regard to it. Appendix II shows the raw data sorted and displayed to illustrate the contribution of each group.

The Kruskal-Wallis test yields a significance level of between 0.02 and 0.05 (depending on the metric) in favor of there being some difference among the slopes of the groups.

As may be seen in Table II, the slope of the line is larger for ad hoc individuals than for disciplined teams. This means that disciplined teams do a better job (by requiring fewer program changes) for a given amount of complexity than ad hoc individuals. Disciplined teams appear better than ad hoc teams for the $v(G)$ metric. The ad hoc teams appear to have done better than the ad hoc individuals by the DecS metric.

It should be noted that there is no statistically significant differences among the intercepts of the three groups with respect to any of the metrics except STMT. The Kruskal-Wallis test for STMT yields a significance level of 0.05. The

TABLE II
METHODOLOGY COMPARISONS USING THE MANN-WHITNEY U TEST ON SLOPES

SynC		
	Kruskal-Wallis at 0.05 level	
	Mann-Whitney	AI = AT
		AI > DT at 0.014 level
		AT = DT
STMT		
	Kruskal-Wallis at 0.05 level	
	Mann-Whitney	AI > AT at 0.094 level
		AI > DT at 0.014 level
		AT = DT
$v(G)$		
	Kruskal-Wallis at 0.02 level	
	Mann-Whitney	AI = AT
		AI > DT at 0.008 level
		AT > DT at 0.074 level
DecS		
	Kruskal-Wallis at 0.02 level	
	Mann-Whitney	AI > AT at 0.026 level
		AI > DT at 0.008 level
		AT = DT

TABLE III
METHODOLOGY COMPARISONS (USING THE MANN-WHITNEY U TEST) ON r SQUARE

SynC		
	Kruskal-Wallis at 0.03 level	
	Mann-Whitney	AI > AT at 0.016 level
		AI = DT
		AT < DT at 0.052 level
STMT		
	Kruskal-Wallis at 0.10 level	
	Mann-Whitney	AI > AT at 0.026 level
		AI = DT
		AT < DT at 0.034 level
$v(G)$		
	Kruskal-Wallis at 0.10 level	
	Mann-Whitney	AI > AT at 0.016 level
		AI = DT
		AT = DT
DecS		
	Kruskal-Wallis at 0.10 level	
	Mann-Whitney	AI > AT at 0.042 level
		AI = DT
		AT = DT

Mann-Whitney U test shows AI < AT at a 0.026 level of significance and DT < AT at a 0.052 level of significance. Note that this tends to support the claim that disciplined teams were more able to cope with the complexity. It makes the distinction between ad hoc individuals and ad hoc teams less clear.

For the coefficient of determination, the Kruskal-Wallis test gives a significance level of 0.03 to 0.10 in favor of there being some difference among the groups (see Table III). The ad hoc teams seem to have a lower coefficient of determination than ad hoc individuals. It is conjectured that this results from the differing abilities of the members of ad hoc teams causing different parts of the system to be assembled with varying degrees of effectiveness. It is interesting to note that disciplined teams also have a larger coefficient of determination than ad hoc teams (for Sync and STMT). This also indicates that a team that works with a set of methodologies tends to be more consistent with respect to uniformly spreading the errors through the code than a team that does not. The data indicate that disciplined teams have a lower coefficient of determination than ad hoc individuals. This would also be expected given our conjecture. Since both disciplined teams and ad hoc individuals were more consistent than ad hoc teams, we may say that the discipline allowed the teams to perform more like an individual than a group.

TABLE IV
SEGMENTS OF METHODOLOGY REGRESSIONS (METRICS WITH PROGRAM CHANGES)

	b	c	r square
Call			
AI	1.288	0.822	0.337
AT	1.684	0.655	0.275
DT	1.221	0.516	0.253
$v(G)$			
AI	1.562	0.822	0.337
AT	1.761	0.763	0.357
DT	1.680	0.399	0.144
DecS			
AI	1.507	1.026	0.359
AT	1.689	0.892	0.366
DT	1.560	0.555	0.163
SynC			
AI	0.575	0.801	0.463
AT	0.911	0.645	0.386
DT	0.672	0.538	0.242
STMT			
AI	0.539	0.921	0.473
AT	0.974	0.707	0.400
DT	0.658	0.608	0.257

D. Regression by Methodology

We treated all of the segments developed by all ad hoc individuals in one regression model for each of the five metrics to see if there were any consistencies within the group. We did likewise for the other two groups, ad hoc teams and disciplined teams. Since the projects were known to have a large variation in the slopes of the regression lines, it was no surprise that all 15 data plots gave a fan that was close at the origin but became more spread out as the value of the metric increased. For this reason, the regression model that gave similar variance of residuals across the scale of the independent variable was an exponential model of the form

$$\text{changes} = b\ \text{metric}^{c}$$

or

$$\log(\text{changes}) = \log(b) + c \log(\text{metric}).$$

The results of the 15 regressions are given in Table IV.

We note that 14 of the exponents are less than 1.0 (one was 1.026). This indicates that the larger segments are less costly in program changes than smaller segments for each unit of complexity. There are at least three possible interpretations of this result.

1) Larger segments cause less problems (so we should encourage larger segments).

2) Programmers tend to write larger segments when the problem is trivial and smaller segments when the problem is more difficult.

TABLE V
METRIC COMPARISONS
(USING THE SIGN TEST)

Friedman yields a 0.02 level		
"SynC = STMT"		(10/19)
"SynC = $v(G)$"		(13/19)
"SynC > DecS"	at 0.063 level	(14/19)
"SynC > CALL"	at 0.019 level	(15/19)
"$v(G)$ < STMT"	at 0.019 level	(4/19)
"$v(G)$ = DecS"		(7/19)
"$v(G)$ = CALL"		(10/19)
"DecS = STMT"		(7/19)
"DecS = CALL"		(11/19)
"CALL < STMT"	at 0.063 level	(5/19)

3) The less capable programmers felt a need to reduce the size of their segments in order to maintain control. Thus, the larger segments were written by the better people.

Because the third interpretation also explains the linear fits for the single projects, it appears to be the best explanation. More experiments are needed before any definitive conclusions can be reached.

A confusing point is that the r square for the DT group is lower than for the AI and AT groups. This seems to contradict some of the earlier conclusions. It does suggest that a disciplined team is less predictable than an ad hoc individual or team, given data from random individuals or teams of the appropriate type. However, given history data from the specific team (i.e., the past performance of the team in dealing with complexity), a specific disciplined team seems more predictable than a specific ad hoc team. More experiments are needed to resolve these points.

V. COMPARISON OF METRICS

We now turn to the second set of questions. The five members of the family have been compared to see how well they predict the number of changes that were made to each segment. Many other members of the family were investigated but not reported because they are unfamiliar and yield no new insights. The results are summarized in Table V.

For each project, the coefficient of determination was compared over the five metrics. Friedman's test [9] is employed to determine globally (over all five metrics) whether there is reason to believe that any of the metrics performs significantly differently from the others. After concluding that there is a difference in the metrics at the 0.02 level of significance, a two-tailed sign test [20] was used pairwise to test the null hypothesis that the metrics have equal predictive value. If the level of significance was less than 0.10, the alternative hypothesis (that there is a difference) with the direction of difference was listed in Table V. Otherwise, the two metrics are listed as "=", indicating that we may not reject the null hypothesis. The last column contains the ratio of the times that the first listed metric had a better (higher) r square than the second metric, to the total number of data points in the group.

The results show that STMT does better than $v(G)$ and indicate that it may be better than CALL in explaining the number of program changes. Moreover, SynC is better than CALL and there is an indication that it may be better than $v(G)$ or DecS. There is no distinguishable difference between SynC and STMT or between CALL, $v(G)$, and DecS.

Since the statement count is easy to calculate and many researchers have found that it does a credible job of measuring the complexity, it must be considered the baseline for comparison in studies of this kind. This study has failed to find a metric that is significantly better than statement count.

VI. ORTHOGONALITY OF THE METRICS

If the complexity of computer programs is to be measured, it is necessary to develop metrics that have a degree of orthogonality, i.e., metrics that measure different aspects of the complexity. As was seen in the correlation matrix of Table I, the metrics considered so far lack this property. One possible way to gain some orthogonality is to normalize the metrics. For example, if cyclomatic complexity is normalized with respect to length (by computing $v(G)$/STMT) the resulting metric is a measure of decision density in the code. One might then ask if code with a high decision density requires more program changes than code with a low decision density. For our data, the answer is no. Similar results (or lack thereof) hold for CALL and DecS normalized by STMT.

A mild relationship does seem to exist between SynC/STMT and program changes, but little predictive value is gained. The normalized metrics were also tried in multiple regression equations with all of the original metrics, using incremental regression techniques [19]. The normalized metrics proved to yield little additional information in the equations.

Another approach, more closely resembling [22], is to regress a metric [such as $v(G)$] with STMT and select those points that are sufficiently far from the regression line (e.g., 1 or 2 standard errors of estimate). Then, considering the regression of STMT with program changes, inquire about the residuals associated with the outliers of the first regression. If the anomalies have an effect on the residuals (if they tend to be larger), then the anomalies tend to cause more than their share of program changes. This approach was tried. None of the metrics had anomalies where the associated residual populations have means significantly different from 0. In fact, the means tended to be very close to 0. Therefore, the anomalies do not seem to explain program changes in our data.

No orthogonal metrics within this study of syntactic metrics have been successful at explaining program changes.

We believe that orthogonal metrics may exist outside the realm of syntactic complexity. Metrics that measure other properties of programs and program development, specifically data metrics [4], [11], [14], [16], [21], [23], [25], may prove orthogonal to the control structure metrics studied here. We are currently investigating a variety of metric classes.

VII. CONCLUSIONS

A family of syntactic complexity metrics has been defined that encompasses many of the current metrics. The family has been used in comparing different individuals, metrics, and development methodologies.

It was found that individuals differ widely in the number of program changes required to implement a program of some given complexity. This leads to the possibility of measuring a programmer's ability to cope with complexity. The concept of an ability measure should be pursued with complexity metrics from other groups of metrics (such as data complexity metrics).

Furthermore, we have some evidence that a disciplined team acts more capably than an individual as measured by the slopes of the fitted regression lines. This lends support to the argument that even small projects that one person might be able to do will be done better if more than one person cooperates in the development (at least when they take active steps, such as the use of various methodologies to aid in their communication). This should not be construed to mean that many programmers should be assigned to the task. Rather, it might be possible to gain the same advantages by assigning the project to one programmer and allowing him to use other programmers in design and code reading in return for providing the same service to them.

Several metrics in the family have been evaluated with respect to their suitability in correlating with program changes; none seems significantly better than statement count. Metrics which count specific parts of the code (such as CALL or DecS) appear to be less well related to program changes than the metrics which count more things (such as SynC and STMT). This suggests the hypothesis that program changes are distributed randomly through the code and the closer a metric comes to counting all of the syntactic attributes of the program the better the metric will correlate with program changes. One experiment cannot prove or disprove this hypothesis. As more data are examined, we may begin to understand this relationship more fully.

Acknowledgment

We would like to thank H. E. Dunsmore and G. Sutton for their efforts in counting the program changes in the projects. This work would not have been possible without the work done by R. W. Reiter in developing the experiment.

Appendix I

This Appendix contains the values of the slope and coefficient of determination data for each of the 19 projects. The data are presented in groups with each column representing a given metric. The projects are ordered in the same manner in each of the two tables.

Slope and Coefficient of Determination Data

slope

	SynC	STMT	CALL	v(G)	DecS
AI	.729	1.162	1.411	1.776	4.567
	.286	.397	.443	1.013	3.044
	.157	.277	.469	.440	1.076
	.576	.809	1.460	2.121	3.114
	.499	.927	3.859	2.788	2.950
	.437	.684	.406	1.318	2.774
AT	.204	.319	.547	.531	1.060
	.492	.785	1.775	1.244	2.904
	.085	.121	.118	.289	.640
	.173	.244	.242	.799	.841
	.456	.743	1.736	1.992	2.840
	.128	.254	.502	.621	.811
DT	.155	.239	.510	.258	1.035
	.142	.193	.362	.372	1.078
	.161	.278	.390	.583	.887
	.102	.143	.181	.281	.932
	.297	.524	.823	.694	1.627
	.189	.320	.542	.499	1.319
	.141	.210	.323	.431	.812

Slope and Coefficient of Determination Data

r square

	SynC	STMT	CALL	v(G)	DecS
AI	.475	.447	.104	.288	.368
	.866	.800	.556	.595	.852
	.717	.679	.487	.525	.733
	.521	.469	.454	.396	.372
	.739	.838	.489	.683	.712
	.592	.638	.075	.627	.450
AT	.490	.504	.289	.257	.376
	.322	.287	.380	.177	.325
	.170	.149	.078	.187	.207
	.054	.051	.024	.086	.042
	.585	.551	.533	.589	.515
	.207	.227	.319	.210	.232
DT	.335	.358	.257	.065	.302
	.351	.309	.312	.163	.382
	.724	.790	.705	.522	.480
	.660	.725	.872	.735	.531
	.499	.558	.734	.321	.336
	.682	.625	.398	.494	.672
	.469	.484	.337	.350	.288

Appendix II

The following tables present the results of the regression models on each of the 19 compiler projects. The data are sorted by the *r* square or the slope and is divided into three columns under the headings AI (ad hoc individual), AT (ad hoc team), and DT (disciplined team). This is intended to provide a picture of the statistical results found in the paper by illustrating which groups have higher or lower values for these measures.

Sorted Raw Data
(used by Mann-Whitney U test)

r square

v(G)			DecS			SynC			STMT			CALL		
AI	AT	DT	AI	AT	DT	AI	AT	DT	AI	AT	DT	AT	AT	DT
		.065		.042			.054			.051			.024	
	.086			.207			.170			.149		.075		
		.163		.232			.207			.227			.078	
	.177				.288		.322			.287		.104		
	.187				.302			.335			.309			.257
	.210			.325				.351			.358		.289	
	.257				.336			.469	.447					.312
.288			.368			.475			.469				.319	
		.321	.372				.490				.484			.337
		.350		.376				.499		.504			..380	
.396					.382	.521				.551				.398
		.494	.450				.585				.558	.454		
		.522			.480	.592					.625	.487		
.525				.515				.660	.638			.489		
	.589				.531			.682	.679				.533	
.595					.672	.717					.725	.556		
.627			.712					.724			.790			.705
.683			.733			.739			.800					.734
		.735	.852			.866			.838					.872

slope

v(G)			DecS			SynC			STMT			CALL		
AI	AT	DT	AI	AT	DT	AI	AT	DT	AI	AT	DT	AI	AT	DT
		.258		.640			.085			.121			.118	
		.281		.811				.102			.143			.181
	.289				.812		.128				.193		.242	
		.372		.841				.141			.210			.323
		.431			.887			.142			.239			.362
.440					.932			.155		.244				.390
		.499			1.035	.157				.254		.406		
	.531			1.060				.161	.277			.443		
		.583	1.076				.173				.278	.469		
	.621				1.078			.189		.319			.502	
		.694			1.319		.204				.320			.510
	.799				1.627	.286			.397					.542
1.013			2.774					.297			.524		.547	
	1.244			2.840		.437			.684					.823
1.318				2.904			.456			.743		1.411		
1.776			2.950				.492			.785		1.460		
	1.992		3.044			.499			.809				1.736	
2.121			3.114			.576			.927				1.775	
2.788			4.567			.729			1.162			3.859		

References

[1] V. R. Basili and D. H. Hutchens, "A study of a family of structural complexity metrics," in *Proc. ACM-NBS 19th Annu. Tech. Symp.: Pathways to System Integrity*, Gaithersburg, MD, June 1980, pp. 13-15.

[2] V. R. Basili and R. W. Reiter, "An investigation of human factors

in software development," *IEEE Computer*, pp. 21-38, Dec. 1979.

[3] —, "A controlled experiment quantitatively comparing software development approaches," *IEEE Trans. Software Eng.*, vol. SE-7, May 1981.

[4] V. R. Basili and A. J. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 390-396, Dec. 1975.

[5] —, *SIMPL-T: A Structured Programming Language*. Geneva, IL: Paladin House, 1976.

[6] G. Benyon-Tinker, "Complexity measures in an evolving large system," in *Proc. IEEE Workshop on Quantitative Software Models*, Kiamesha Lake, NY, Oct. 1979, pp. 117-127.

[7] W. M. Carriere and R. Thibodeau, "Development of a logistics software cost estimating technique for foreign military sales," General Res. Corp., Santa Barbara, CA, June 1979.

[8] E. T. Chen, "Program complexity and programmer productivity," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 187-193, May 1978.

[9] W. J. Conover, *Practical Nonparametric Statistics*. New York: Wiley, 1971.

[10] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 96-104, Mar. 1979.

[11] H. E. Dunsmore, "The influence of programming factors on program complexity," Ph.D. dissertation, Dep. Comput. Sci., Univ. Maryland, July 1978.

[12] H. E. Dunsmore and J. D. Gannon, "Experimental investigations of programming complexity," in *Proc. ACM-NBS 16th Annu. Tech. Symp.: Systems and Software*, Washington, DC, June 1977, pp. 117-225.

[13] H. E. Dunsmore and J. D. Gannon, "Analysis of the effects of programming factors on programming effort," *J. Syst. Software*, vol. 1, pp. 265-273, 1980.

[14] J. L. Elshoff, "An analysis of some commercial PL/1 programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 113-120, June 1976.

[15] M. Halstead, *Elements of Software Science*. New York: Elsevier Comput. Sci. Library, 1977.

[16] S. Henry and D. Kafura, "Software quality metrics based on interconnectivity," *J. Syst. Software*, vol. 2, pp. 121-131, 1981.

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

[18] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, pp. 308-320, Dec. 1976.

[19] J. Neter and W. Wasserman, *Applied Linear Statistical Models*. Homewood, IL: R. D. Irwin, Inc., 1974.

[20] S. Siegel, *Nonparametric Statistics*. New York: McGraw-Hill, 1956.

[21] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structural design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.

[22] T. Sunohara, A. Takano, K. Vehara, and T. Ohkawa, "Program complexity measure for software development management," in *Proc. 5th Int. Conf. Software Eng.*, San Diego, CA, Mar. 9-12, 1981, pp. 100-106.

[23] M. D. Weiser, "Program slicing," presented at the 5th Int. Conf. Software Eng., San Diego, CA, 1981.

[24] M. R. Woodward, M. A. Hennell, and D. Hedly, "A measure of control flow complexity in program text," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 45-50, Jan. 1979.

[25] S. S. Yau and J. S. Collofello, "Some stability measures for software maintenance," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 545-552, Nov. 1980.

Victor R. Basili (M'83), for a photograph and biography, see this issue, p. 663.

David H. Hutchens received the B.S. degree in mathematics from Western Carolina University, Cullowhee, NC, in 1977, the M.S. degree in mathematical sciences from Clemson University, Clemson, SC, in 1979, and the Ph.D. degree in computer science from the University of Maryland, College Park, in 1983.

He is currently an Assistant Professor of Computer Science at Clemson University. His research interests include measurement, evaluation, and modeling of the software development process and its product.