EVALUATING AUTOMATABLE MEASURES OF SOFTWARE DEVELOPMENT

Victor R. Basili and Robert W. Reiter, Jr.

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

There is a need for distinguishing a set of useful automatable measures of the software development process and product. Measures are considered useful if they are sensitive to externally observable differences in development environments and their relative values correspond to some intuition regarding these characteristic differences. Such measures could provide an objective quantitative foundation for constructing quality assurance standards and for calibrating mathematical models of software reliability and resource estimation. This paper presents a set of automatable measures that were implemented, evaluated in a controlled experiment, and found to satisfy these usefulness criteria. The measures include computer job steps, program changes, program size, and cyclomatic complexity.

I. Proposing Automatable Measures

There is a need for analyzing software development phenomena and for developing measures to facilitate such analysis. Appropriate measurement is the key to providing vital information to the developer, manager, and customer with regard to making an individual development effort more visible. Measurement is equally important for investigating techniques and methodologies in an attempt to recommend more effective software development methods. The appropriateness of any measurement depends not only on its usefulness, but also on the manner in which it is performed. Mensurative concepts and specific metrics for software have been devised in abundance (e.g., [Gilb 77]) but these software measurements can prove to be expensive, inaccurate, or completely inadequate in practical application.

The Software Engineering Laboratory [Basili et al. 77; Basili & Zelkowitz 78; Basili & Zelkowitz 79], a joint venture of the University of Maryland and NASA's Goddard Space Flight Center, has been analyzing several multi-man-year software projects. Data collection in the early stages has been predominantly manual; i.e., participants in the project development periodically fill out forms which provide information on what is being done, the level of effort involved, the types and numbers of errors encountered, etc. This kind of data collection can impair an analysis effort to some degree since it introduces some overhead and, more importantly, sources of error and bias due to human reporting inaccuracies and the phenomenon that people behave differently when being observed.

It was recognized early that these problems could be alleviated by developing automatable measures and collection mechanisms. However, the kind of data that can be collected automatically is not always as informative as the kind that can be collected manually. For example, it is generally desirable to record the incidence of errors during software development and to classify them according to various criteria. However, this task typically requires extensive and expensive human supervision and is quite error prone itself. On the other hand, the textual changes made to source code (once it exists in a programming-product library) could be collected automatically and an algorithm (based on heuristics for identifying combinations of textual changes that correspond to the correction of individual errors) could be employed to count potential error corrections automatically. Such an automatable measure is relatively inexpensive to implement and might approximate the true error count, but it cannot yield much insight into the types and causes of errors.

We define a measure to be automatable if it satisfies the following criteria: (1) the data can be collected without interfering with individuals involved in the development, (2) the measures are computed algorithmically from quantifiable sources normally available, and (3) the measures can be reproduced on other projects by essentially the same algorithms. Note that these criteria stress the feasibility of measuring software development in an unobtrusive, objective, and transportable fashion and they ignore the issue of how automatable measures are actually implemented. There are clear advantages to computerizing software data collection and measurement

This research was supported in part by the Air Force Office of Scientific Research Grant AFSOR-77-3181A to the University of Maryland. Computer time was provided in part through the facilities of the Computer Science Center of the University of Maryland.

procedures; in fact, all of the particular measures presented below were computerized to some degree in order to study them empirically. However, the fundamental research issue addressed in this paper is how to determine whether or not an automatable software measure is indeed a worthwhile measure of anything after all.

In order to discover which measures are worth implementing, we need to do some experimenting. First we must decide what characteristics of software development we want to analyze and then we must choose some automatable means of quantifying them. As in the error-counts/textual-changes example above, this approach often involves some kind of intuitive approximation and requires an independent validation (to ensure that the automatable measure does in fact reflect the designated software characteristic). In analyzing software development, we have focused on two distinct facets of the problem: the process and the product. With respect to the process, we are predominantly interested in measuring efficiency of development; i.e., the less effort expended to develop a given system, the better. With regard to the product, we are predominantly interested in measuring quality of developed software; i.e., quality in the sense of readability, modifiability, testability, etc. In both cases these general characteristics of software development are extremely difficult to quantify by any practical automatic means. To overcome this obstacle we must transform them into more specific characteristics that have a chance of being quantified and measured automatically.

The efficiency of the software development process is determined by the level of effort expended, as manifested in time spent and cost incurred, both by human personnel and on computing machinery. For this paper, we have choosen two aspects of the process that can be measured automatically: computer usage by type of activity and program changes [Dunsmore & Gannon 77] as an approximation to error counts. In some sense, both of these measures effectively quantify the effort expended, time spent, and cost incurred during software development.

The quality of a developed software product can be manifested in its structure. Clearly, structure has a direct bearing on readability, modifiability, testability, etc. For this paper, we have isolated two aspects of program structure, size and control flow, to be quantified and measured automatically. Specific items examined are the number of routines, the number of source lines, the number of decisions, and several variations of cyclomatic complexity [McCabe 76].

The following criteria were used to evaluate the measures as useful: (1) the measure should be sensitive to externally observable differences in the development environment, i.e., if the observable difference between two developments is the methodology used, then the measure should yield different values for these two developments; (2) the relative values of the measure for specific environments should correspond to some intuitive notion about the characteristic differences for those environments, i.e., if a methodology is supposed to improve various attributes of the development process or product, then the value of the measure should be greater for a development using that methodology than for a development not using it. For example, if the measure is the number of textual changes as an approximate error count, and if the environmental difference is the use of a particular methodology, then to satisfy the first criterion the number of textual changes should be different for those developments using and not using the methodology. The second criterion would be satisfied if the methodology purports to reduce errors and the measure yields a lower value for the methodology development than the nonmethodology development.

II. An Evaluation Environment

The evaluation of these measures took place under controlled and slightly varying conditions. Nineteen independent replications of the same software development project were conducted and closely monitored [Reiter 79]. Six were performed by single individuals using an ad hoc approach to software development, six were performed by three-person teams also using an ad hoc approach, and seven were performed by three-person teams using a specific disciplined methodology. The following mnemonics denote the three distinct programming environments represented by these groups: AI (ad hoc individuals), AT (ad hoc teams), and DT (disciplined teams).

Each individual or team in groups AI and AT was allowed to develop software in a manner of their own choosing, which is referred to as an ad hoc approach. The disciplined methodology imposed on teams in group DT consisted of an integrated set of techniques, including top down design of the problem solution using a Process Design Language (PDL), function expansion, design and code reading, walk throughs, and chief programmer team organization [Baker 72; Linger, Mills & Witt 79]. The participants in group DT had just recently been taught and trained in this disciplined methodology; thus, they probably were not yet able to exercise the techniques to maximum capacity and advantage.

Beyond these controlled variations in size of programming team and degree of methodological discipline, other software development factors were explicitly held constant across all replications wherever possible. Each individual or team worked independently to develop their own software system, using the same project specifications, computer resource allocation, calendar time allotment, implementation language, debugging tools, etc. Although it was not possible to assign participants randomly to the programming environments, at least one independent indicator of the native programming ability of the participants was observed to be distributed quite homogeneously among the three groups.

The particular programming application was a compiler for a small high-level language and a simple stack machine. The development task required between one and two man-months of effort and the resulting software systems averaged about 1200 source lines, or 600 executable statements, in a high-level language. The participants were advanced undergraduate and graduate students in the University of Maryland Computer Science Department, some with up to three years' professional experience. The implementation language was the high-level structured-programming language SIMPL-T [Basili & Turner 76], which is used extensively in course work at the University and has string-processing capabilities similar to PL/1.

III. Evaluating Automatable Measures

A series of controlled experiments have been conducted within the environment described above in order to evaluate several automatable measures. A legitimate experimental design and rigorous nonparametric statistical methods were employed; details appear in [Basili & Reiter 78; Basili & Reiter 79; Basili & Reiter 80]. In each case, the null hypothesis that the metric shows no distinction in its expected value among the software developments performed by the AI, AT, and DT groups was tested against the empirical data. The experimental results, in the form of statistical conclusions, are presented in Tables 1 and 2. Each conclusion consists of the comparison outcome (as a symbolic inequality) and the associated critical level (as a probability) for a particular automatable measure.

The outcome indicates the observed distinction among the programming environments represented by the AI, AT, and DT groups, and the direction thereof. For example, the outcome AI < AT = DT indicates that the expected value for the metric was noticeably lower for the ad hoc individuals than for either the ad hoc teams or the disciplined teams, which both exhibited about the same expected value for the metric; in other words, the metric differentiates between individual programmers and programming teams, with individuals scoring lower. The critical level indicates the statistical "strength" of each conclusion, since it is the risk of having claimed the corresponding conclusion erroneously (i.e., concluding that the data support some distinction when in fact no systematic difference truly exists for that metric). For example, a critical level of 0.1 means that the chances are one in ten that the same measurement scores could have occurred due to purely random fluctuations among the groups. The critical level may also be interpreted as indicating the relative degree of differentiation observed between the groups.

For each set of measures, we will define the individual metrics, explain how they are computed, motivate their general intuitive appeal, and briefly interpret their experimental results. It will be demonstrated that almost all of the metrics presented in this paper qualify as both automatable and useful measures, according to the criteria established above.

In the remainder of this paper, the term "module" refers to a separately compiled source code component of the entire software system, and the term "routine" refers to an individual procedure or function.

A. Process Measures

Essentially two aspects of the software development process were examined: job steps (or runs) and program changes [Dunsmore & Gannon 77]. These were selected as quantifiable, automatable approximations to two important dimensions of the process: level of effort expended in development and errors occurring in program source code during development.

Job Step Metrics. A "computer job step" is a conceptually indivisible programmer-oriented activity that is performed on a computer at the operating system command level, is inherent to the software development process, and involves a nontrivial expenditure of computer or human resources. Typical examples of job steps would include editing symbolic texts, compiling source modules, link-editing (or collecting) object modules, and executing entire programs; however, activities such as querying the operating system for status information or requesting access to on-line files would not qualify as job steps. In this study, consideration as COMPUTER JOB STEPS was limited exclusively to the activities of compiling source modules and executing entire programs.

Several subcategories and normalizations of the basic computer job steps measure were also investigated. A "module compilation" is an invocation of the implementation-language processor on the source code of an individual module. In this study, only compilations of modules comprising the final software product (or logical predecessors thereof) are counted in the MODULE COMPILATIONS metric. All module compilations are categorized as either "identical" or "unique" depending on whether or not the source code compiled is textually identical to that of a previous compilation. During the development process, each unique compilation was necessary in some sense, while an identical compilation could conceivably have been avoided by saving the (relocatable) object module from a previous compilation for later reuse (except in the situation of undoing source code changes after they have been tested and found to be erroneous or superfluous).

A "program execution" is an invocation of a complete programmer-developed program (after the necessary compilation(s) and link-editing) upon some test data. In this study, only executions of programs composed of modules comprising the final product (or logical predecessors thereof) are counted in the PROGRAM EXECUTIONS metric. A "miscellaneous job step" is an auxiliary compilation or execution of something other than the final software product. In this study, the MISCELLANEOUS JOB STEPS metric counts exactly those COMPUTER JOB STEPS not already categorized as MODULE COMPILATIONS or PROGRAM EXECUTIONS. An "essential job step" is a computer job step which involves the final software product (or logical predecessors thereof) and could not have been avoided (by off-line computation or by on-line storage of previous compilations or results). In this study, the number of ESSENTIAL JOB STEPS is the sum of the number of UNIQUE MODULE COMPILATIONS plus the number of PROGRAM EXECUTIONS.

Finally, both the average and worst-case number of unique compilations per module were examined. The number of AVERAGE UNIQUE COMPILATIONS PER MODULE is simply the number of unique module compilations divided by the number of modules. The number of MAXIMUM UNIQUE COMPILATIONS FOR ANY ONE MODULE is computed in the obvious way: each unique compilation is associated (either directly or as a logical predecessor) with a particular module of the final product, a total number of unique compilations is obtained for each module, and the maximum of the totals is taken.

On the whole, these job step metrics directly quantify the frequency of computer system activities during software development. They are one possible way of measuring machine costs, in units of basic activities rather than monetary charges. Assuming that each computer system activity also involves a certain expenditure of the programmer's time and effort (e.g., effective terminal contact, test result evaluation), these metrics indirectly quantify the human costs of development (at least that portion exclusive of design work).

<u>Program Changes Metric</u>. The "program changes" metric [Dunsmore & Gannon 77] is defined in terms of textual changes in the source code of a module during the development period, from the time that module is first presented to the computer system to the completion of the project. The rules for counting program changes--which are reproduced below from the paper referenced above with the kind permission of the authors--are such that one program change should represent approximately one conceptual change to the program.

The following each represent a single program change:

(1) one or more changes to a single statement,
(A single statement in a program represents a single concept and even multiple character changes to that statement represent mental activity with a single concept.)

(2) one or more statements inserted between existing statements,
(The contiguous group of statements inserted probably corresponds to a single abstract instruction.)

(3) a change to a single statement followed by the insertion of new statements.
(This instance probably represents a discovery that an existing statement is insufficient and that it must be altered and supplemented in order to achieve the single concept for which it was produced.)

However, the following are not counted as program changes:

(1) the deletion of one or more existing statements,
(Statements which are deleted must usually be replaced with other statements elsewhere. The inserted statements are counted; counting deletions as well would give double weight to such a change. Occasionally statements are deleted but not replaced; these are probably being used for debugging purposes and their deletion takes no great mental activity.)

(2) the insertion of standard output statements or special compiler-provided debugging directives,
(These are occasionally inserted in a wholesale fashion during debugging. When the problem is discerned, these are then all removed, and the actual statement change takes place.)

(3) the insertion of blank lines, insertion of comments, revision of comments, and reformatting without alteration of existing statements.
(These are all judged to be cosmetic in nature.)

Program changes are counted according to these rules by symbolically comparing the source code from each pair of consecutive compilations of a particular module (or logical predecessor thereof).

The program changes metric directly quantifies, at an appropriate level of abstraction, the amount of textual revision to source code during (postdesign) development. Arguing that textual revisions are generally necessitated by errors encountered while building, testing, and debugging software, independent research [Dunsmore & Gannon 77] has demonstrated a high (rank order) correlation between total program changes (as counted automatically according to a specific algorithm) and total error occurrences (as tabulated manually from exhaustive scrutiny of source code and test results) during software development. This metric is thus a reasonable measure of the relative number of programming errors encountered outside of design work. Assuming that each textual revision involves a certain expenditure of the programmer's effort (e.g., planning the revision, on-line editing of source code), this metric indirectly quantifies the level of human effort devoted to implementation.

<u>Automatbility and Usefulness</u>. These process measures are quite automatable. The raw data for computer job steps might be obtainable directly from operating system logs and accounting information, or the processors involved in each job step could be instrumented to maintain similar logs. The subcategorization of job steps is quite objective and the normalizations are defined mathematically, so that they can all be automated easily. Program changes are computed algorithmically from symbolic comparisons of successive versions of source code modules. If a programming-product library is maintained on-line during development, the comparisons can be made as part of regular updating procedures. Alternatively (as was done in our study), the language processor could be instrumented to record automatically the

Table 1. Conclusions for Process Measures

N.B. A simple pair of equal signs (= =) appears in place of the null outcome AI = AT = DT in order to avoid cluttering the table.

process measure	comparison outcome	critical level
COMPUTER JOB STEPS	DT < AI = AT	0.003
MODULE COMPILATION	DT < AI = AT	0.022
UNIQUE	DT < AI = AT	0.011
IDENTICAL	= =	
PROGRAM EXECUTION	DT < AI = AT	0.022
MISCELLANEOUS	DT < AI = AT	0.144
ESSENTIAL	DT < AI = AT	0.003
AVERAGE UNIQUE COMPILATIONS PER MODULE	DT < AI = AT	0.088
MAX. UNIQUE COMPILATIONS F.A.O. MODULE	DT < AI = AT	0.118
PROGRAM CHANGES	DT < AI = AT	0.003

MAX. is an abbreviation for MAXIMUM
F.A.O. is an abbreviation for FOR ANY ONE

complete source code of every compilation in an historical data bank of source code versions; comparisons can then be made from the data bank in an off-line fashion without interfering with the programmer's normal activities. Many computing systems already support some kind of symbolic comparator utility which isolates the literal differences between the line images of two files; such a utility is easily augmented or extended to implement the rules for identifying program changes as defined above.

These process measures satisfy our usefulness criteria, since they have differentiated between the different programming environments in our evaluation environment and they have done so in a manner that corresponds intuitively to the arranged, observable differences among those environments. The experimental results for these process measures (see Table 1) are strong. As a whole, their strength derives from the near unanimity of comparison outcomes and the low critical levels involved. Except for one instance of no distinction (IDENTICAL MODULE COMPILATIONS), each metric distinguishes the disciplined teams from both the ad hoc teams and the ad hoc individuals. Further, the direction of the distinction indicates that the DT group required noticeably fewer job steps and textual revisions than the AI or AT group to complete the exact same project. This corresponds directly to our intuition about how efficiency should be affected by disciplined methodology, since it is widely held that the disciplined methodology has a beneficial effect on software development in terms of minimizing development effort/time/costs and reducing errors during implementation and testing by enhancing their detection at earlier phases.

Thus, the process measures presented here do meet the criteria established above as both automatable and useful measures.

B. Product Measures

Essentially two aspects of the developed software product were examined: program size and control flow complexity. These were selected as quantifiable, automatable facets of program structure. Program size is measured in terms of the number of routines, the number of symbolic lines, and the number of programmer-coded decisions in the final product source code. Control flow complexity is measured in terms of cyclomatic complexity [McCabe 76], a graph-theoretic metric which has been shown to be independent of physical size (adding or subtracting functional statements leaves the measure unchanged) and dependent only on the decision structure of a program.

Program Size Metrics. The ROUTINES measure counts the individual procedures and functions of a program, since they are the smallest packaging unit of contiguous statements that the programmer deals with conceptually. The LINES measure counts every textual line of delivered source code, including compiler directives, documenting comments, data variable declarations, executable statements, etc. The DECISIONS measure counts each of the decisions coded by the programmer to govern flow of control;

in a structured programming language, this is simply the number of constructs, such as IF-THEN-ELSE, WHILE-DO, CASE-OF, etc., that appear in the source code.

On the whole, these program size metrics directly quantify the sheer frequency of certain "elements" within a software product. Simple size metrics are usually the most readily available measures of a software product and are often a central factor in large-scale software productivity studies (e.g., [Walston & Felix 77]) and resource estimation models (e.g., [Putnam 78]). Traditionally, sheer size has been used as an indirect indicator of the logical complexity of software systems; i.e., the larger a system is or the more "elements" it contains, the more complex it is likely to be and therefore of lower quality.

Cyclomatic Complexity Metrics. In standard graph theory [Berge 73], the cyclomatic number $v(G)$ of a graph G having n nodes, e edges, and p connected components is defined as

$$v(G) = e - n + p$$

and is equal to the minimum number of basic paths in a strongly connected graph, from which all other paths may be constructed as linear combinations. By modeling the control flow of a computer program as a graph in the traditional manner, an analogously defined number serves as a software metric [McCabe 76]. This "cyclomatic complexity" measure $v(P)$, for a well-formed (i.e., all statements are on some path from start to finish) program P with π predicates strewn among r routines, is computed as

$$v(P) = \pi + r ,$$

where a predicate is a Boolean expression governing control flow, and is equal to the minimum number of basic execution paths through the routines, from which all execution paths may be "constructed."

This measure originated as an absolute count of the minimum number of program paths to be tested and thus served as a quantitative indicator of the difficulty of testing a given program to a certain degree of thoroughness. The cyclomatic complexity measure is one particular way to directly quantify the complexity of a program's flow of control, something that has been identified as a key factor of software quality.

Several variations of the basic cyclomatic complexity measures were considered, because there are at least two definitional issues for which intuitively motivated alternatives lead to meaningful variations. One of these issues is the weighting given to instances of CASE statement constructs. The original definition of cyclomatic complexity views a CASE statement as the semantically equivalent series of nested IF-THEN-ELSE statements: each CASE statement contributes n units of cyclomatic complexity, where n is the number of individual cases involved. It can be argued, however, that a CASE statement deserves a smaller contribution to cyclomatic complexity since its inherent uniformity and readability have a moderating effect on control flow complicatedness (relative to an explicit series of nested IF-THEN-ELSE statements). One reasonable alternative defines each CASE statement as contributing $\lfloor \log_2(n) \rfloor$* units of cyclomatic complexity, where n is the number of individual cases involved. This logarithmic weighting is appropriate since the CASE statement's moderating effect seems to increase with the number of cases involved.

The other issue is the manner of counting predicates. The original definition counts simple predicates individually so that the compound predicate

(I < J) <u>and</u> ((A(I) = A(J)) <u>or</u> (<u>not</u> SORTED))

would contribute three units of cyclomatic complexity, for example. An alternative definition considers each full compound predicate as an indivisible part of a program, contributing one unit of cyclomatic complexity, since it represents a single abstract condition governing the flow of control. Note that this issue is the basis for a proposed extension [Myers 77] to the original cyclomatic complexity measure. This issue also affects the way individual cases of a CASE statement construct are identified and counted. The original definition counts each case label separately, since multiple case labels on the same case branch are semantically equivalent to simple predicates joined by <u>or</u>'s to form the Boolean expression governing the case branch. The alternative definition counts only the case branches themselves, regardless of case label multiplicity, since multiple case labels could represent a single abstract case designator (e.g., case labels <u>0</u>, <u>1</u>, <u>2</u>, ..., <u>9</u> may be abstracted simply to <u>digit</u>).

This study examined the four variations of cyclomatic complexity defined as follows:

- SIMPPRED-NCASE -- Simple predicates contribute 1 unit; CASE statements contribute 1 unit for each case label.
- SIMPPRED-LOGCASE -- Simple predicates contribute 1 unit; CASE statements contribute $\lfloor \log_2(n) \rfloor$ units, where n is the number of case labels.
- COMPPRED-NCASE -- Compound predicates contribute 1 unit; CASE statements contribute 1 unit for each case branch; multiple case labels on the same case branch are disregarded.
- COMPPRED-LOGCASE -- Compound predicates contribute 1 unit; CASE statements contribute $\lfloor \log_2(n) \rfloor$ units, where n is the number of case branches; multiple case labels on the same case branch are disregarded.

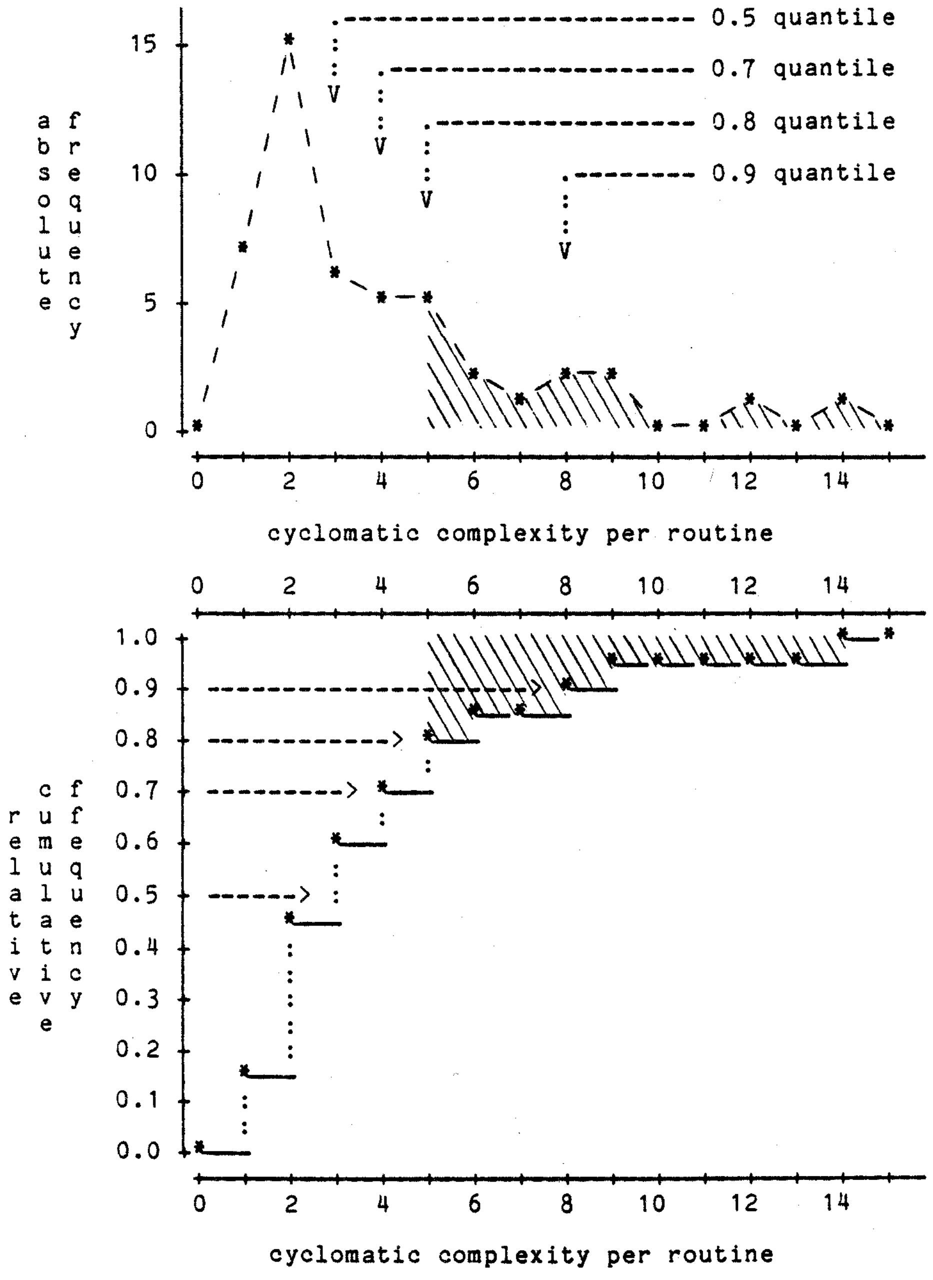
Note that the SIMPPRED-NCASE variation of cyclomatic complexity is McCabe's original proposal.

Because of the way it is defined, the cyclomatic complexity metric is sensitive to system modularization, particularly to the number of routines. Hence, a fairer comparison of cyclomatic complexities can be made on the basis of individual

* The notation $\lfloor x \rfloor$ signifies the smallest integer less than or equal to x .

Figure 1. Frequency Distributions for Cyclomatic Complexity

Both the absolute and relative-cumulative frequency distributions for cyclomatic complexity values from 47 routines comprising a complete system are plotted. The tail region associated with the 0.8 quantile is shaded on each plot.



routines, rather than globally for entire systems. It is also desirable to focus attention on instances of higher cyclomatic complexity since most systems usually contain several small, easily understood routines with very low cyclomatic complexity values (e.g., a routine which computes the average of a vector) while the remaining routines are really the heart of the system and embody most of the true "complexity." One general technique for applying such routine-oriented metrics to entire systems is in terms of the sample quantiles [Conover 71, pp. 31-32, pp. 72-73] of the empirical distribution of the metric's values across the routines comprising each system (see Figure 1). Both the quantile point value (i.e., the largest integer x such that the fraction of cyclomatic complexity values which are less than x is less than or equal to some fixed fraction) and the quantile tail average (i.e., the average of cyclomatic complexity values greater than or equal to the quantile point value) are normalized ways to quantify just how high the cyclomatic complexity is for the typical nontrivial routines. Several different quantiles were examined: the 0.5 quantile is closely related to the median of the distribution, and the 0.7, 0.8, and 0.9 quantiles provide a series of increasingly smaller tails of the distribution.

Automatability and Usefulness. These product measures are all clearly automatable since the required information is routinely determined within he front end (lexical and syntactic analysis phases) of a typical language processor. A compiler can easily be instrumented to compute the values of the measures for each module of source code. The technique of applying cyclomatic complexity to entire systems is readily computed according to its mathematical formulation.

The experimental results for these product measures (see Table 2) are rather interesting. As a whole, they each make some sort of distinction among the programming environments. But there is a subtle shifting among the observed distinctions, making it more difficult to draw connections with our intuitive notions about how program size and structure are affected by differences in programming team size and degree of methodological discipline. This is not surprising in view of the poorly understood nature of program structure in general, and program complexity in particular. In any event, the measures presented here certainly satisfy the first part of our usefulness criteria, and we will proceed to delineate some interpretations by which the second part will also be satisfied.

The results for size aspects exhibit distinctions with a common underlying trend, namely, AI $\leq$ DT $\leq$ AT. This trend says that the disciplined teams produced software smaller in size than the ad hoc teams, but still larger than the ad hoc individuals. This is reasonable, and indeed expected, under the following assumption: disciplined methodology serves to mitigate the organizational overhead experienced in team programming, enabling a team to function with greater conceptual integrity. Conceptual integrity (which is easily achievable by an individual programmer working alone) certainly has an impact on the structural quality of the software being produced, resulting in a closer-knit design and implementation. Thus, we would expect the AI group to always measure less than the AT group, and the DT group to be measured closer to AT on some product aspects and closer to AI on other product aspects, but generally between the two. The experimental results obtained on the size aspects display this kind of behavior, and thus correspond to a reasonable intuition.

In a similar vein, the results for cyclomatic complexity exhibit another common underlying trend, namely, DT $\leq$ AT $\leq$ AI. In fact, the non-null outcomes were all either DT = AT < AI or else DT < AT = AI. This says that either the teams were differentiated from the individuals or else the disciplined methodology was differentiated from the ad hoc approach, depending on the particular variation of cyclomatic complexity involved. This corresponds well with the intuition that team programming alone should force a general reduction of cyclomatic complexity for individual routines, and that use of the disciplined methodology within team programming should promote this effect even further. The observed results for the cyclomatic complexity metrics seem to display this kind of behavior, and thus correspond to a reasonable intuition. The generally weaker differentiation (i.e., larger critical levels) observed for the cyclomatic complexity metrics relative to the other metrics presented above is quite understandable in light of the fact that all 19 systems were coded in a structured-programming language which greatly restricts potential control flow patterns. We would expect cyclomatic complexity metrics to be even more useful in the context of unrestrictive programming languages such as FORTRAN.

Thus, the product measures presented here do meet the criteria established above as both automatable and useful measures.

IV. Summary and Future Research Directions

Empirical data and statistical conclusions from a controlled experiment in software development have been used to evaluate certain software measures. Criteria have been established for the automatability and usefulness of software measures in general. Several measures of the software development process and product have been presented that satisfy these criteria for useful automatable measures. Both process metrics (job steps and program changes) have proven, in our evaluation environment, to differentiate strongly on the basis of efficiency of software development. The program size metrics (lines of code, number of routines, and number of decisions) have demonstrated reasonable utility in differentiating among our programming environments. Several variations of the cyclomatic complexity metric have shown very encouraging potential for usefulness as measures of software product quality. As a side issue, a technique was described for circumventing the problem of applying routine-oriented measures

Table 2. Conclusions for Product Measures

N.B. A simple pair of equal signs (= =) appears in place of the null outcome AI = AT = DT in order to avoid cluttering the table.

product measure	comparison outcome	critical level
ROUTINES	AI < AT = DT	0.063
LINES	AI < DT < AT	0.119
DECISIONS	DT = AI < AT	0.146
CYCLOMATIC COMPLEXITY :		
SIMPPRED-NCASE VARIATION :		
0.5 QUANTILE POINT VALUE	AT = DT < AI	0.185
0.5 QUANTILE TAIL AVERAGE	= =	
0.7 QUANTILE POINT VALUE	AT = DT < AI	0.072
0.7 QUANTILE TAIL AVERAGE	AT = DT < AI	0.180
0.8 QUANTILE POINT VALUE	= =	
0.8 QUANTILE TAIL AVERAGE	= =	
0.9 QUANTILE POINT VALUE	= =	
0.9 QUANTILE TAIL AVERAGE	= =	
SIMPPRED-LOGCASE VARIATION :		
0.5 QUANTILE POINT VALUE	= =	
0.5 QUANTILE TAIL AVERAGE	DT < AI = AT	0.185
0.7 QUANTILE POINT VALUE	= =	
0.7 QUANTILE TAIL AVERAGE	DT < AI = AT	0.188
0.8 QUANTILE POINT VALUE	DT < AI = AT	0.114
0.8 QUANTILE TAIL AVERAGE	= =	
0.9 QUANTILE POINT VALUE	AT = DT < AI	0.103
0.9 QUANTILE TAIL AVERAGE	= =	
COMPPRED-NCASE VARIATION :		
0.5 QUANTILE POINT VALUE	= =	
0.5 QUANTILE TAIL AVERAGE	AT = DT < AI	0.155
0.7 QUANTILE POINT VALUE	DT < AI = AT	0.140
0.7 QUANTILE TAIL AVERAGE	AT = DT < AI	0.154
0.8 QUANTILE POINT VALUE	= =	
0.8 QUANTILE TAIL AVERAGE	AT = DT < AI	0.154
0.9 QUANTILE POINT VALUE	AT = DT < AI	0.134
0.9 QUANTILE TAIL AVERAGE	AT = DT < AI	0.136
COMPPRED-LOGCASE VARIATION :		
0.5 QUANTILE POINT VALUE	= =	
0.5 QUANTILE TAIL AVERAGE	DT < AI = AT	0.197
0.7 QUANTILE POINT VALUE	DT < AI = AT	0.082
0.7 QUANTILE TAIL AVERAGE	DT < AI = AT	0.197
0.8 QUANTILE POINT VALUE	= =	
0.8 QUANTILE TAIL AVERAGE	= =	
0.9 QUANTILE POINT VALUE	AT = DT < AI	0.089
0.9 QUANTILE TAIL AVERAGE	= =	

to entire software systems (i.e., how to measure an entire system using a metric that is only appropriate for individual routines).

Work is progressing on developing and evaluating additional software measures. Another measure of control flow complexity, called essential complexity [McCabe 76], is currently being evaluated. The program length, volume, and effort measures from software science theory [Halstead 77] have recently been automated within our compiler. We are interested in developing several product measures based on declarative scope of data variables, data variable usage span [Elshoff 76], and interprocedural communication via global variables [Stevens, Myers & Constantine 74; Basili & Turner 75]. We are also interested in developing process measures based on the concept of error-day [Mills 76]. All of these software measures can be evaluated using the same test bed of empirical data acquired from the experimental study that was conducted in our evaluation environment. Furthermore, by exploring various comparisons and correlations, we intend to probe for quantitative relationships between automatable software metrics.

There is a great deal of work to be done. Automatable measures must themselves be evaluated according to some empirical criteria regarding their usefulness. Only then will there be any hope of employing them to provide the user and the researcher with cost-effective and accurate feedback on software development. Only then will their application to medium-scale software development, such as the projects in the Software Engineering Laboratory, be warranted.

References

[Baker 72] F.T. Baker. Chief programmer team management of production programming. IBM Systems Journal 11, 1 (1972), 56-73.

[Basili & Reiter 78] V.R. Basili and R.W. Reiter, Jr. Investigating software development approaches. Technical Report TR-688, Department of Computer Science, University of Maryland, August 1978.

[Basili & Reiter 79] V.R. Basili and R.W. Reiter, Jr. An experimental investigation of the effects of human factors on software development. Computer Magazine, to be published.

[Basili & Reiter 80] V.R. Basili and R.W. Reiter, Jr. A controlled experiment quantitatively comparing software development approaches. IEEE Trans. Software Eng., to be published.

[Basili & Turner 75] V.R. Basili and A.J. Turner. Iterative enhancement: a practical technique for software development. IEEE Trans. Software Eng. 1, 4 (December 1975), 390-396.

[Basili & Turner 76] V.R. Basili and A.J. Turner. SIMPL-T, A Structured Programming Language. Geneva, IL: Paladin House, 1976.

[Basili & Zelkowitz 78] V.R. Basili and M.V. Zelkowitz. Analyzing medium-scale software development. in Proc. 3rd Int. Conf. Software Eng. (IEEE Catalog No. 78CH1317-7C), Atlanta, Georgia (May 1978), 116-123.

[Basili & Zelkowitz 79] V.R. Basili and M.V. Zelkowitz. Measuring software development characteristics in the local environment. Computers & Structures 10 (1979), 39-43.

[Basili et al. 77] V.R. Basili, M.V. Zelkowitz, F.E. McGarry, R.W. Reiter, Jr., W.F. Truszkowski, and D.L. Weiss. The software engineering laboratory. Technical Report TR-535, Department of Computer Science, University of Maryland, May 1977.

[Berge 73] C. Berge. Graphs and Hypergraphs. Amsterdam, The Netherlands: North-Holland, 1973.

[Conover 71] W.J. Conover. Practical Nonparametric Statistics. New York, NY: John Wiley & Sons, 1971.

[Dunsmore & Gannon 77] H.E. Dunsmore and J.D. Gannon. Experimental investigation of programming complexity. in Proc. ACM/NBS 16th Annual Tech. Symp.: Systems and Software, Washington, D.C. (June 1977), 117-125.

[Elshoff 76] J.L. Elshoff. An analysis of some commercial PL/1 programs. IEEE Trans. Software Eng. 2, 2 (June 1976), 113-120.

[Gilb 77] T. Gilb. Software Metrics. Cambridge, MA: Winthrop, 1977.

[Halstead 77] M. Halstead. Elements of Software Science. New York, NY: Elsevier, 1977.

[Linger, Mills & Witt 79] R.C. Linger, H.D. Mills, and B.I. Witt. Structured Programming: Theory and Practice. Reading, MA: Addison-Wesley, 1979.

[McCabe 76] T.J. McCabe. A complexity measure. IEEE Trans. Software Eng. 2, 4 (December 1976), 308-320.

[Mills 76] H.D. Mills. Software development. IEEE Trans. Software Eng. 2, 4 (December 1976), 256-273.

[Myers 77] G.J. Myers. An extension to the cyclomatic measure of program complexity. ACM SIGPLAN Notices 12, 10 (October 1977), 61-64.

[Putnam 78] L.H. Putnam. A general empirical solution to the macro software sizing and estimation problem. IEEE Trans. Software Eng. 4, 4 (July 1978), 301-316.

[Reiter 79] R.W. Reiter, Jr. Empirical investigation of computer program development approaches and computer programming metrics. Ph.D. Dissertation, Department of Computer Science, University of Maryland, December 1979.

[Stevens, Myers & Constantine 74] W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured design. IBM Systems Journal 13, 2 (1974), 115-139.

[Walston & Felix 77] C.E. Walston and C.P. Felix. A method of programming measurement and estimation. IBM Systems Journal 16, 1 (1977), 54-73.