Software Metrics: Guest Editor's Introduction

This Special Section is the culmination of a call for papers made two years ago among the software metrics community. In proposing this section to Les Belady, then the Editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, I stressed that the section would consist primarily of papers reporting empirical results. Les was enthusiastic because he felt the TRANSACTIONS was receiving too many papers for review which proposed new metrics, but which failed to report any data demonstrating their benefit. The four papers included in this section all present empirical research evaluating and comparing different metrics.

For the purpose of this section, "software metric" will refer only to its more limited definition as a measure derived from the requirements, specification, design, code, or documentation of a computer program. These metrics are computed for the purpose of predicting some outcome of the software development process which is believed to be affected by the software characteristics being measured. A hallmark of engineering practice is the ability to predict product behavior from measures taken on the product or its component parts. This measurement prediction is the motivation behind most software metrics.

Software performance evaluation represents a successful engineering application of measurement techniques and statistical analysis in predicting an important software criterion, execution efficiency. Greater difficulty has been experienced in attempting to predict criteria such as the amount of time required to develop a program or the number of defects a given component of the program is likely to possess. Nevertheless, prediction of these criteria is important if the process of developing software and the quality of the delivered product are to be managed effectively.

Systematic research on metrics to predict these latter criteria began when Maurice Halstead proposed his theory of software science in 1972. He presented a number of equations using counts of operators and operands to predict a wide range of criteria. Although his assumptions and analyses have been debated over the past decade, research in industry and academia has demonstrated that measurement at the level of tokens is effective for predicting important outcomes.

In 1976 Thomas McCabe developed a metric derived from graph theory and based on the decision structure of the program. Since his initial proposal, a blizzard of refinements have been described, but typically without accompanying data to verify their purported improvements. Accordingly, most of the empirical research to date has been performed on the metrics developed by Halstead and McCabe.

Frequently, empirical research has been criticized on any of several methodological issues. These criticisms typically revolved around measurement theory, research design, or statistical analysis. Although related, these are separate skills that must be mastered by the serious metrics researcher.

Skill in measurement theory begins with the conceptual ability to identify critical attributes of software and define them in quantifiable terms. Yet, it also includes the mathematical understanding of allowable operations for the type of measurement scale employed and the statistical means of assessing and reducing measurement error. Although software metricians exhibit considerable skill in quantifying software characteristics, their greatest failures in measurement technique are displayed in measuring the criteria their software metrics supposed predict. Poor reliability can mask important predictive relationships which would have been detected with better data.

Research design is a critical skill for ensuring that the data collected provide legitimate answers to the questions asked. Poor research designs result in explanations given at the wrong level of analysis, causal explanations provided where only associations were demonstrated, and their failure to rule out alternate explanations of the results. The extensive planning required for a rigorous research design is as critical in field studies on actual projects as in controlled laboratory experiments.

Problems in statistical analysis have ranged from an inadequate number of data points for detecting a significant effect, to begging the existence of an effect by employing an arbitrarily low significance level. The greatest problem in statistical analysis, however, is in the limited repertoire of statistical methods most researchers have been able to apply. Thus, rather than use a multivariate statistical procedure, too many researchers have reported pages of univariate tests. Beyond anesthetizing the reader, these armies of univariate tests capitalize on chance relationships in the data (unless corrected on an experimentwise basis) and fail to assess the size of the combined effects and interactions of important characteristics. On the other hand, some researchers having learned a new technique are too eager to apply it to available data without regard to its relevance to the original research question. A good statistician is as sensitive to conceptual issues in analyzing data, as to the mathematical assumptions underlying the various techniques.

It is too easy to blame shortcomings in measurement theory, research design, and statistical analysis on individual researchers. The problem is more fundamental, since these three areas are not included in the curricula of computer science departments (with the occasional exception of introductory statistics). Thus, those best able to define the important research issues in software metrics are least prepared to design rigorous empirical studies. One solution to this problem is the development of interdisciplinary research teams. Such teams, coupled with better research education, are critical if the field of software metrics is not to suffer irreparable harm from poor research methods.

Methodological issues can be raised with the four papers included in this Special Section, as they can be raised with any paper reporting empirical results. Nevertheless, these papers were judged to make conceptual and practical contributions to our understanding of software measurement and the prediction of important development outcomes.

The size of a computer program (typically reported in state-
ments) cannot be determined until the program has been coded. However, this figure is needed much earlier in the development cycle for sizing the programming effort. Further, when productivity is assessed as statements per person-year, comparisons cannot be made among projects using different programming languages for determining the amount of function delivered per unit of development time. Al Albrecht developed function points to address these measurement problems. His quantification scheme is primarily designed for business systems, and has been adopted by over 200 commercial institutions for measuring their programming effectiveness and sizing their projects. The advantages of function points are their computability early in the development cycle and their independence from differences in programming languages. In their paper, Albrecht and Gaffney described the use of function points in assessing a number of programming projects. They report the reassuring results that function points are related to measures of software science, statements, and development effort.

In a supplementary paper to that of Albrecht and Gaffney, Behrens demonstrates the use of function points in a software production environment. His paper shows how software metrics can be used in management decision making. The types of decisions to which he applies these metrics are resource estimation analysis during project planning, and benefits analysis of various programming methods and languages. He also suggests that function points can be used to track productivity trends over time.

Using data from the NASA Software Engineering Laboratory, Basili, Selby, and Phillips demonstrate that Halstead’s and McCabe’s metrics, in addition to statements, calls, and jumps, are related to development effort and errors. None of the measures were found superior to the others in correlating with development effort. However, source statements were found to be more related to errors than were the Halstead or McCabe metrics. The size of the correlations observed improved when analyses were restricted to the modules developed by a single programmer or those within a single project. These results emphasize the importance of determining the appropriate level of analysis for a specific measurement technique. They also demonstrate improved results from data analysis by considering the reliability of the data.

In a companion paper, Basili and Hutchens delve into the analysis of data at the individual and project levels. Their analyses demonstrate characteristic slopes for different programmers in relationships between a software metric and the changes to a module. Further, the software metrics also differed among programming teams using different methodologies. This latter result suggests the possibility of using software metrics to evaluate programming techniques. As in the previous study, they found that statement counts correlated as well or better with program changes than metrics did.

Several conclusions can be drawn from the results reported in these papers. First, function points can be a valuable programming management aid in a business data processing environment. Further research needs to identify similar quantification schemes for scientific or embedded systems programming. Second, it is not clear that metrics such as those proposed by Halstead and McCabe are better than statement counts at predicting important criteria in actual programming environments. These metrics should not be discarded, however, until empirical data are reported from other environments demonstrating the same pattern of results obtained by Basili and his colleagues. Currently, the available data are mixed. Third, there are ways in which software metrics may be used to evaluate individual differences among programmers and the level of complexity they can handle effectively. Fourth, software metrics offer one source of data evaluating the comparative benefits of programming techniques and tools. Finally, the reliability of the data affects the strengths of the relationships observed. Careful attention must be paid to the accuracy of the data during data collection, and this attention must be an active process. These conclusions support the current use of some metrics in programming evaluation and management, and continued research on others.

Bill Curtis (M’79) received the B.A. degree in psychology, mathematics and drama from Eckerd College, St. Petersburg, FL, in 1971, the M.A. degree in clinical psychology from the University of Texas, Austin, in 1974, and the Ph.D. degree in industrial psychology and statistics from Texas Christian University, Ft. Worth, in 1975.

He currently directs the Human and Knowledge Engineering Department at the ITT Programming Technology Center, Stratford, CT. He is responsible for directing human factors research and providing guidance on user interface technology to ITT units. He joined ITT in 1980 as Manager, Programming Trends Analysis, and was instrumental in developing ITT’s programming productivity and quality measurement system. This system provides measurement standards, measurements for management tracking, annual baselines, and analyses of major factors in productivity and quality. From 1978 to 1980, he managed the Software Management Research Group in General Electric’s Space Division, Arlington, VA. His group performed experimental research on software metrics and the psychological aspects of programming. From 1975 to 1977, he was a Research Assistant Professor at the University of Washington where he taught behavioral statistics and performed research on organizational behavior and sports psychology.

Bill Curtis

Guest Editor