Concise Papers

Cyclomatic Complexity Density and Software Maintenance Productivity

Geoffrey K. Gill and Chris F. Kemerer

Abstract—**While the need for software metrics to aid in the assessment of software complexity for maintenance has been widely argued, little agreement has been reached on the appropriateness and value of any single metric. McCabe's cyclomatic complexity metric, a measure of the maximum number of linearly independent circuits in a program control graph, has been widely used in research. The current research validates previously raised concerns about the metric on a new data set. However, a simple transformation of the metric is investigated whereby the cyclomatic complexity is divided by the size of the system in source statements, thereby determining a "complexity density" ratio. This complexity density ratio is demonstrated to be a useful predictor of software maintenance productivity on a small pilot sample of actual maintenance projects.**

Index Terms— **Management, measurement, performance, software productivity, software maintenance, software complexity, McCabe metrics, cyclomatic complexity.**

I. Introduction

A critical distinction between software engineering and other, more well-established branches of engineering is the shortage of well-accepted measures, or metrics, of software development. Without metrics, the tasks of planning and controlling software development and maintenance will remain stagnant in a "craft"-type mode, wherein greater skill is acquired only through greater experience, and such experience cannot be easily communicated to the next system for study, adoption, and further improvement. With metrics, software projects can be quantitatively described, and the methods and tools used on the projects to improve productivity and quality can be evaluated. These evaluations will help the discipline grow and mature as progress is made at adopting those innovations which work well, and discarding or revising those which do not.

Of particular concern is the need to improve the software maintenance process, given that maintenance is estimated to consume 40–75% of the software effort [23]. What differentiates maintenance from other aspects of software engineering is, of course, the constraint of the existing system. It constrains the maintenance work in two ways—the first through the imposition of a general design structure that circumscribes the possible designs of the system modifications, and the second, more specifically, through the complexity of the existing code which must be modified.

A metric of this latter, more specific type of system influence is McCabe's cyclomatic complexity metric, a measure of the maximum number of linearly independent circuits in a program control graph [14]. As described by McCabe, a primary purpose of the metric is to " . . . identify software modules that will be difficult to test or maintain" [14, p. 308], and it is therefore of particular interest to researchers and practitioners concerned with maintenance. The McCabe metric has been widely used in research, and a recent article by Shepperd cites some 63 articles which are directly or indirectly related to the McCabe metric [20].

There is some question, however, as to the practical usefulness of the metric [4]. As noted by Shepperd, concerns about the external validity of the data and analyses in some of the previous studies can be used to mitigate some of the results, particularly those using data on small programs, often using student subjects. Therefore these results bear validation on data from actual systems, and, in particular, from data on maintenance projects, since use of the metric for testing and maintenance was one of its author's main stated purposes. In particular, this research seeks not to determine whether cyclomatic complexity captures all aspects of complexity in one figure of merit, but rather to answer the question raised by Shepperd as to whether cyclomatic complexity can serve as a "useful engineering approximation" [20].

An additional question he raised regards the practical utility of a number of "variants" on the original McCabe metric [15], [9]. This research investigates these variants in the context of new development.

One area that is of considerable practical importance for research is the effect of existing system complexity on maintenance productivity. It is often assumed that: (a) more complex systems are harder to maintain, and (b) that systems suffer from "entropy" and therefore become more chaotic and complex as time goes on [1]. Knowledge that a metric such as cyclomatic complexity accurately reflects the difficulty in maintaining a set of source code would allow management to make rational choices in the repair/replace decision as well as aid in evaluating CASE tools designed to reduce existing complexity by automatically restructuring source code [16], [22]. The empirical evidence linking software complexity to software maintenance costs has been criticized as being relatively weak [11]. However, studies have shown that a significant fraction of the staff resources used in maintenance is spent in understanding the existing code [5]. This paper will report a study of the relationship between McCabe's cyclomatic complexity and software maintenance productivity, given that a metric which measures complexity should prove to be a useful predictor of maintenance costs.

The rest of this paper is organized as follows: Section II outlines the data-collection procedures and summarizes the data set used in the research. Results are presented in Section III, and concluding remarks are presented in Section IV.

II. Data Collection

A. Data Overview and Background of the Data Site

The approach taken in this study was to collect detailed data about a number of completed software projects at a single firm. All the projects studied were small customized programs primarily for real-time defense applications. The projects were undertaken in the period from 1984–1989, with the majority of the work performed in the last three years of that period. The company considers the data contained in this study proprietary and therefore requested that its name not

Manuscript received November 1, 1990; revised June 10, 1991. Recommended by S. H. Zweben. This work was partially supported by the Center for Information Systems Research and the International Financial Services Research Center.

The authors are with the Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139.

IEEE Log Number 9104158.

0098–5589/91$01.00 © 1991 IEEE

be used and that any identifying data which could directly connect it with this study be withheld. The numeric data, however, have not been altered in any way. The company employed an average of about 30 professionals in the software department.

One benefit of this approach was that because all the projects were undertaken by a single organization, neither interorganizational differences nor industry differences should confound the results. A possible limitation is that this concentration on one industry and one company may somewhat limit the applicability of the results. At a minimum, however, the defense industry is very important for software[1]and the results will clearly be of interest to researchers in that field. More generally, these results may provide suggestions for research at other sites.

B. Data Sources

The data for this study were derived from three sources:

- *Source Code*—A copy of the source code developed or modified by each project was obtained
- *Accounting Database*—This database contained the number of hours every software engineer worked on each project for every week of the study period
- *Activity Reports*—A description of the work done by each engineer for each week for every project [7].

Of the source code developed for the projects, 74.4% was written in Pascal and the other 25.6% in a variety of other third-generation languages, primarily FORTRAN. Following Boehm, only deliverable code was included in this study, with "deliverable" defined as any code which was placed under a configuration management system [2]. This definition eliminated any purely temporary code, but retained code which might not have been part of the delivered system but was considered valuable enough to be saved and controlled. (Such code included programs for generating test data, procedures to recompile the entire system, utility routines, etc.) The source code was the main data source for addressing the counting variants research question.

In addition to the source code, data from the accounting database and activity reports were used in addressing the maintenance productivity question. Because the hourly charges were used to bill the project customer and were subject to Department of Defense (DoD) audit, the engineers and managers took extreme care to insure proper allocation of charges. For this reason, these data are believed to be accurate representations of the actual effort expended on each project. The activity reports were weekly summaries of the work done by each software engineer every week as part of his or her standard recordkeeping for DoD reporting. These activity reports were used to ensure that only work related to software maintenance was included when calculating maintenance productivity.

C. Data Definitions

For the purposes of this research, a software module was defined as the source code that was placed in a single physical file (which was often compiled separately). The size of the software modules was measured in noncomment source lines of code (NCSLOC). The definition of a line of code adopted for this study is that of Conte *et al.* [3, p. 35]:

> "A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and nonexecutable statements."

This set of rules has several major benefits. It is very simple to calculate, and it is very easy to translate for any computer language. Because of its simplicity, it has become the most popular SLOC metric [3, p. 35]. Through consistent use of this metric, results become comparable across different studies.[2]

This research project had access to a total of 834 modules, of which 771 were written in Pascal and 63 in FORTRAN. They averaged 176.2 NCSLOC, with a standard deviation of 257.9. In total, approximately 150 000 lines of code (NCSLOC) from 19 software systems were analyzed [8].

The definition of software maintenance adopted by this research is that of Parikh and Zvegintzov—namely, "work done on a software system after it becomes operational" [17]. The projects described below as maintenance work pertain to all of the three Lientz and Swanson maintenance types (corrective, adaptive, and perfective) [12].

Project data were gathered for seven maintenance projects in order to test the hypothesis concerning the effect of complexity on maintenance productivity. The average size of a maintenance project was 1006 NCSLOC added (standard deviation, 1158), and required an average of 1059 work hours (standard deviation, 982).

III. Results

A. Empirical Tests of Analytic Critique

Before embarking on the actual study of complexity and maintenance productivity, a study was done to determine whether proposed variants of McCabe's original metric, suggested to deal with perceived weaknesses in the original metric, would be likely to affect the results. The original McCabe metric is defined as:

$$V(G) = e - n + 2$$

where $V(G)$ = the cyclomatic complexity of the flowgraph G of the program in which we are interested, e = the number of edges in G, and n = the number of nodes in G.

McCabe showed that $V(G)$ is also equal to the number of binary decision nodes in G plus one. There are four basic rules that can be used to calculate $V(G)$ [9]:

1) Increment one for every IF, CASE or other alternate execution construct
2) Increment one for every Iterative DO, DO-WHILE or other repetitive construct
3) Add two less than the number of logical alternatives in a CASE
4) Add one for each logical operator (AND, OR) in an IF.

The three variants studied in this research are defined as follows:

a) CYCMAX: all four rules are used, as in the original McCabe version
b) CYCMID: only rules 1–3 apply, as proposed by Myers [15]
c) CYCMIN: only rules 1 and 2 apply, as suggested by Hansen [9].

[1]*Fortune* magazine reported in its September 25, 1989 issue that the Pentagon spends $30 billion annually on software.

[2]As several authors have described, SLOC have many deficiencies as a metric. In particular, they are difficult to define consistently, as Jones has identified no fewer than 11 different algorithms for counting code [10]. Also, SLOC do not necessarily measure the "value" of the code (in terms of functionality and quality). However, Boehm also points out that no other metric has a clear advantage over SLOC. Furthermore, SLOC are easy to measure, conceptually familiar to software developers, and are used in most productivity databases and cost estimation models. For these reasons, it was the metric adopted by this research.

TABLE I
MEANS OF COMPLEXITY METRICS

CYCMIN	CYCMID	CYCMAX
20.3	22.2	24.0

TABLE II
CORRELATIONS METRICS FOR CODE COMPLEXITY

	NCSLOC	CYCMIN	CYCMID	CYCMAX
NCSLOC	1.000	0.934	0.949	0.949
CYCMIN		1.000	0.985	0.986
CYCMID			1.000	0.998
CYCMAX				1.000

Note: All correlations are significant at better than the 99.99% level ($n = 834$).

TABLE III
COMPARISON OF FORTRAN AND PASCAL AVERAGE COMPLEXITY DENSITY

Variable	FORTRAN ($n = 63$)	Pascal ($n = 771$)
CMINDENS	0.10	0.10
CMIDDENS	0.11	0.11
CMAXDENS	0.12	0.11

Table I gives the average value for each metric. Table II gives the Pearson correlation coefficients for the three complexity metrics used in this study.

The most striking result is the extent to which similar metrics are highly correlated with each other. These results indicate that the metrics proposed by Myers (CYCMID) and Hansen (CYCMIN) measure complexity in a very similar manner to McCabe's metric, CYCMAX. In fact, the empirical data suggest that there are unlikely to be any practically significant different results using CYCMIN or CYCMID instead of CYCMAX, which is consistent with Shepperd's prediction [20].

As was also suggested by previous research, the length metric, NCSLOC, and the complexity measure, CYCMAX, turned out to be highly correlated. Table II gives the Pearson correlation coefficients for the complexity metrics with the SLOC metrics.

B. Pilot Test of the Impact of Complexity on Maintenance Productivity

An obvious research question is whether the code length (as measured by NCSLOC) has a significant effect on maintenance productivity. A possible argument that NCSLOC should be a significant factor rests on the theory that since the developer has more code to understand, maintaining that code would be more difficult. On the other hand, a slightly different argument suggests that the program length effect would be relatively minor if the changes were localized to a fairly small number of modules. The engineer would therefore only need detailed knowledge of a small fraction of the code. The size of this fraction depends more on how well the code is modularized than the total size of the program. Thus the length of the entire code is less relevant. There is no way to determine *a priori* which of these hypotheses is correct, and therefore the relationship between NCSLOC and maintenance productivity requires an empirical test.

Knowledge of the impact of cyclomatic complexity on maintenance productivity is potentially more valuable than that of NCSLOC, because managers typically do not have a great deal of control over the size of a program since it is intimately connected to the size of the application. However, by measuring and adopting complexity standards and/or by using CASE restructuring tools, they can manage unnecessary cyclomatic complexity.

Because of the high correlation of cyclomatic complexity with lines of code, it is likely that a test of the impact of complexity on productivity using cyclomatic complexity would produce similar results to those obtained using lines of code. Therefore a transformed metric complexity density, CMAXDENS, is defined as the ratio of cyclomatic complexity (CYCMAX) to thousand lines of executable code (KNCSLOC). This complexity density measure is similar to Gilb's logical decisions per statement [6], but such measures have not often been published. One exception is a by-product of Selby's research on reused code [19]. In his study of 25 NASA projects, his mean was 81.4, with a standard deviation of 57.2. Other work by Potier *et al.* [18] and Sunohara *et al.* [21] also uses length-adjusted complexity measures to model software quality. CMAXDENS, the average McCabe complexity per thousand lines of code, was 121.3 for the current sample (standard deviation of 62.7).

The code analyzed in this research was written in both FORTRAN and Pascal; therefore it is important to determine whether there exists any effect of programming language used. The average complexity density of FORTRAN modules was compared with Pascal modules (see Table III).

No statistical difference between the means of the modules from the different languages was found. These results show that there is little difference that can be attributed to the different languages, at least for the data in this sample. It is also an indication of the general robustness of the complexity density metric.

Ultimately, however, the value of such a complexity metric lies in whether it can predict how difficult a piece of software will be to maintain. Such a prediction can be used either in estimating the resources required or in trying to restructure the code so that the maintainability is improved. The current best measure of maintainability is the productivity of the maintenance project on a piece of software. In order to test the complexity density metric, its average value was computed for seven application systems before they were maintained. Logically, this initial (premaintenance) complexity density should only affect those phases that are directly related to maintaining the code. Tasks such as user support, system operation, and management functions should not be directly affected by the complexity of the code. Therefore maintenance productivity was defined as the total number of lines added divided by the time (in hours) spent in coding and testing upgrades to the software. These data are presented in Table IV.

Three regressions were performed in order to estimate relations of three statistics of the premaintenance software with the associated maintenance productivity. (Obviously, the small number of data points requires that any results must be treated cautiously.) The three statistics examined were CMAXDENS and its components, KNCSLOC and CYCMAX. The results for CMAXDENS were:

$$\text{PRODUCTIVITY} = 28.7 - 0.134^{*}\ \text{CMAXDENS}$$
$$(t = 3.53)\ (t = -2.69)$$
$$R^2 = 0.59$$
$$F\text{-Value} = 7.22\ (p = 0.04)$$
$$n = 7.$$

TABLE IV
SUMMARY OF MAINTENANCE PROJECT DATA VALUES

CMAXDENS	Productivity	NCSLOC Added	Labor Hours	Initial CYCMAX	Initial NCSLOC
0.132	10.1	451	45	882	6682
0.134	10.3	3143	305	958	7133
0.197	3.6	395	110	1054	5343
0.196	2.0	347	174	1126	5738
0.193	6.1	341	56	1176	6085
0.152	0.3	221	737	310	2036
0.108	19.0	2147	113	370	3435

The t-values are significant at normal levels, suggesting that CMAXDENS is a useful predictor of variations in maintenance productivity. The R^2 value of 0.59 may be interpreted to mean that more than half of the variation in maintenance productivity for these data is accounted for by this simple model.

Neither of the two other regressions, however, yielded statistically significant (at normal levels) results:

PRODUCTIVITY = 5.52 + 0.346* KNCSLOC
$(t = 0.65)$ $(t = 0.22)$
$R^2 = 0.01$
F-Value = 0.050 $(p = 0.83)$ $n = 7$
PRODUCTIVITY = 11.8 − 0.0053 * CYCMAX
$(t = 1.70)$ $(t = -0.69)$
$R^2 = 0.09$
F-Value = 0.48 $(p = 0.52)$ $n = 7$

These results indicate that for this small sample, the complexity and length of the premaintenance code (CYCMAX and KNCSLOC) are not as useful predictors of the productivity of the maintenance project as is the complexity density.[3]

These results suggest that maintenance productivity declines with increasing complexity density. While it would be speculative to ascribe too much importance to results based on only seven data points, they do seem sufficiently interesting and the implications sufficiently important that further study is indicated. If these results continue to hold on a larger independent sample, then such results would provide strong support for the use of the complexity density measure as a quantitative method for analyzing software to determine its maintainability. It might also be used as an objective measure of this aspect of software quality.

IV. SUMMARY AND CONCLUDING REMARKS

Based upon the data used in this study, both the Myers and Hansen counting variants of cyclomatic complexity seem to result in values that are equivalent to the original version for all practical purposes. Therefore these data support Shepperd's hypothesis that the variations do not represent a significant difference over the original formulation [20]. The main research question revolved around the empirically derived value added by the metric, given the high correlations found by previous research between the metric and standard size measures, most particularly NCSLOC. The data from this study provide additional evidence of this high correlation. This is particularly noteworthy, since these data are from actual systems and, in particular, are from maintenance projects, which is the main intended domain of the metric. Going beyond this correlation, this paper has investigated the use of a transformed version of the metric, referred to as "complexity density," whereby the ratio of the cyclomatic complexity of the module to its length in NCSLOC is calculated. This ratio is meant to represent the normalized complexity of a module and hence its likely level of maintenance task difficulty. This proposed complexity density ratio was tested on a small sample of projects and shown to be a statistically significant single-value predictor of maintenance productivity. Therefore it is proposed that this transformed version of the cyclomatic complexity metric can provide a useful approximate measure of the difficulty of maintaining a program. Of course, further empirical research will be required to test these pilot results on a larger sample drawn from a different environment in order to validate the usefulness of the complexity density ratio. The complexity density ratio proposed by this research could prove to be a simple, but practically useful measure of complexity. It incorporates the McCabe cyclomatic complexity metric, a well-known and well-understood measure of complexity. In addition, in the intervening years since its introduction, the collection of the data necessary to compute the metric has been automated by a number of tools.[4] Thus the early difficulties in collecting these data have been largely resolved, and therefore data collection should not prove to be a practical barrier to the ratio's use. The continued search for useful metrics of software product complexity is a necessary first step in the ongoing process of moving software development firmly into the realm of engineering. With well-founded complexity metrics, developers will have objective and useful yardsticks with which to evaluate their initial products. These metrics will also be used by each maintenance team as it seeks to enhance the initial product without increasing the level of unnecessary complexity. Both of these aspects should make a contribution toward the improved management of the software development and maintenance process.

[3] Upon closer examination of the data, one project appeared to be an outlier. It was determined that there were two reasons why that project might not be representative. It was the smallest project examined, with only 221 lines of code (NCSLOC) added. Furthermore, 21% of the hours in the project could not be directly assigned because the activity reports had not been filed. The remaining hours were therefore assigned proportionately based upon the other activity reports. This was the largest percentage for any of the projects studied (no other project had more than 15%). Regressions with the outlying project removed for sensitivity analysis yielded improved fits in all three cases, although the CMAXDENS (the complexity density) model retains the highest R^2.

[4] For example, see Language Technology Incorporated's INSPECTOR product, and SET Laboratories' PC-METRIC product [13].

ACKNOWLEDGMENT

Helpful comments were received from B. Curtis on an earlier draft. This paper has also benefited from the useful suggestions of the Associate Editor and anonymous referees.

REFERENCES

[1] L. A. Belady and M. M. Lehman. "A model of large program development," *IBM Syst. J.*, vol. 15, no. 1, pp. 225–252, 1976.
[2] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[3] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin/Cummings, 1986.
[4] W. M. Evangelist, "Software complexity metric sensitivity to program structuring rules," *J. Syst. Software*, vol. 3, pp. 231–243, 1982.
[5] R. K. Fjeldstad and W. T. Hamlen. "Application program maintenance study: report to our respondents," in *Proc. GUIDE 48*. Philadelphia, PA: Guide Corp., 1979.
[6] T. Gilb, *Software Metrics*. Cambridge, MA: Winthrop, 1977.
[7] G. K. Gill, "A study of the factors that affect software development productivity," Master's thesis, Sloan School of Management, MIT, Cambridge, MA, 1989 (unpublished).
[8] G. K. Gill and C. F. Kemerer. "Cyclomatic complexity metrics revisited: an empirical study of software development and maintenance." Ctr. Inform. Syst. Res., Sloan School of Management, MIT, Cambridge, MA, CISR WP No 218. Sloan WP No. 3222–90, 1991.
[9] W. J. Hansen, "Measurement of program complexity by the pair (cyclomatic number, operator count)," ACM SIGPLAN Notices, vol. 13, no. 3, pp. 29–33, Mar. 1978.
[10] C. Jones, *Programming Productivity*. New York: McGraw-Hill, 1986.
[11] J. K. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, and M. A. Adler, "Software complexity measurement," *Commun. ACM*, vol. 29, no. 11, pp. 1044–1050, Nov. 1986.
[12] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*. Reading, MA: Addison-Wesley, 1980.
[13] D. McAuliffe, "Measuring program complexity," *IEEE Computer*, Oct. 1988.
[14] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, 1976.
[15] G. J. Myers, "An extension to the cyclomatic measure of program complexity," *SIGPLAN Notices*, vol. 12, no. 10, pp. 61–64, 1977.
[16] G. Parikh, "Restructuring your COBOL programs," *Computerworld Focus*, vol. 20, no. 7a, pp. 39–42, Feb. 19, 1986.
[17] G. Parikh and N. Zvegintzov, Eds., *Tutorial on Software Maintenance*. Silver Spring, MD: IEEE Computer Soc., 1983.
[18] D. Potier, J. Albin, V. Ferreol, and A. Bilodeau, "Experiments with computer software complexity and reliability," in *Proc. 6th Int. Conf. on Software Eng.*, 1982, pp. 94–101.
[19] R. W. Selby, "Empirically analyzing software reuse in a production environment," in *Software Reuse—Emerging Technologies*, W. Traz, Ed. Los Alamitos, CA: IEEE Computer Soc., 1988.
[20] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Eng. J.*, vol. 3, no. 2, pp. 30–36, Mar. 1988.
[21] T. Sunohara, A. Takano, K. Uehara, and T. Ohkawa, "Program complexity measure for software development management," in *Proc. 5th Int. Conf. on Software Eng.*, 1981, pp. 100–106.
[22] "Parallel test and evaluation of a Cobol restructuring tool," Fed. Software Management Support Ctr., Office of Software Development and Inform. Technol., U.S. General Services Admin., Falls Church, VA, Sept. 1987.
[23] I. Vessey and R. Weber, "Some factors affecting program repair maintenance: an empirical study," *Commun. ACM*, vol. 26, no. 2, pp. 128–134, Feb. 1983.