Full citation: MacDonell, S.G. (1991) Reliance on correlation data for complexity metric use and validation. ACM SIGPLan Notices 26(8), pp.137-144.
doi: 10.1145/122598.122613

Reliance on Correlation Data for Complexity Metric Use and Validation

Stephen G. MacDonell[a, b]

[a]*Computer and Information Science, University of Otago, New Zealand*
[b]*Department of Engineering, University of Cambridge, England*
stevemac@commerce.otago.ac.nz

Abstract

This paper reports the results of an experiment to illustrate the hazards of using correlation data as the sole determinant for software metric use and validation. Three widely cited complexity metrics have been examined in relation to the frequency of software development errors.

I. INTRODUCTION

The ultimate aim of quantitative software assessment is the same as that for any measurement procedure, i.e. to attain control over aspects of the operating environment. In the domain of software development there are many aspects over which we would like to maintain control e.g. development time, resources required, costs incurred and maintenance effort.

Similarly, the measurement function over many application areas is often achieved in a common way, particularly when estimation is a priority. Relationships are proposed, a model or technique is developed to reflect these relationships, prediction or estimation based on the model is performed, comparisons are made between actual and predicted results and the model is adjusted and tuned (if required).

II. COMPLEXITY MEASUREMENT TECHNIQUES

Such has been the perceived importance of software complexity assessment that well over fifty models and techniques which are said to quantify software complexity have been proposed in the literature. (This in itself is an indication of the inadequacy of many of the methods.)

Three of the most widely cited and investigated measurement approaches have been evaluated in this study. These are the *lines of code* measures, Halstead's *software science* [1] and McCabe's *cyclomatic complexity* [2].

a. Lines of code measures:

Line-based measures are still widely promoted, particularly as an easily derived baseline approach which can be useful when applied consistently. The many variations of this technique are based on a common assumption; that a larger program (in terms of the number of lines) is likely to be more difficult to understand than a smaller counterpart. In turn this means that the larger program will be harder to construct and change.

Five sub-techniques are widely employed to indicate complexity in lines:

1. total lines (TLOC) - all lines excluding blank lines
2. executable lines (ELOC) - quantifying all occurrences of program verb clauses
3. non-commentary lines (NCLOC) - all lines except blank and comment lines
4. lines as separated by code delimiters
5. statement count - this usually has the same form as the ELOC or delimiter-separated counting method.

Criticism of this overall approach has, however, been widespread. Probably the most significant factor which has impaired the use of this method is the lack of consistency in the counting methods used - although five counting schemes were outlined above, up to twelve different methods have been identified [3], [4]. This clearly reduces the likelihood of obtaining valid comparisons for results obtained under different schemes.

This method is also susceptible to variations dependent on the programming style employed (particularly the TLOC and NCLOC measures). For example:

```
IF X = 70 THEN
   GOSUB 500                      TLOC = 4
ELSE                              NCLOC = 4
   X = X + 10
```

```
IF X = 70 THEN GOSUB 500 ELSE X = X + 10

                                  TLOC = 1
                                  NCLOC = 1
```

Furthermore, the method appears to lack some degree of comprehensiveness, as only size is evaluated in the assessment of complexity. In addition, LOC counts cannot be determined until late in the project, so that useful estimation for the current project is virtually impossible.

b. Halstead's metrics:

Token counts form the basis for all of Halstead's metrics, the collection of which is widely known as *software science*. Each element in the code representation is classified as an operand (label, constant, variable etc.) or an operator (a symbol which affects the value or order of an operand). Thus the basic parameters of the theory are:

n_1 = number of unique or distinct operators in that implementation
n_2 = number of unique or distinct operands in that implementation
N_1 = total usage of all the *operators* in that implementation
N_2 = total usage of all the *operands* in that implementation.

By combining psychological processing principles with these token counts, Halstead developed a number of size and complexity estimation formulae. For example, the *vocabulary* is derived from the initial counts as:

$$n = n_1 + n_2$$

and the implementation length as

$$N = N_1 + N_2.$$

Another of the primary measures formulated was the size measure, *volume*:

$$V = N \log_2 n.$$

Although this overall technique appears to be comprehensive, several problems have been encountered in its use. Significant anomalies have arisen in the consistent classification of tokens, particularly for languages which have emerged since the theory was developed [5], [6]. This has resulted in a situation where researchers have often had to use their own counting schemes, introducing a degree of subjectivity into what is said to be an objective quantification. This is further compounded by the fact that many studies fail to publish the counting rules which were employed. Thus validation through further experimentation cannot be performed.

Criticism of the psychological assumptions utilized in the formulation of the theory is also widespread, particularly relating to the model of program construction which Halstead adopted [7], [8].

Furthermore because of the theory's actuary nature, some very erratic results have been observed for empirical work involving very large and very small programs [9], [10]. This clearly lessens the general applicability of the theory.

Halstead's work has also been criticized for failing to take account of the many other aspects which are thought to contribute to software complexity. For example, software science makes no consideration of nesting levels or control flow in the code [11], [12]. Samson et al [13] and Vessey [14] also comment that the measures are only derivable after coding is complete.

c. McCabe's cyclomatic complexity:

McCabe's measure uses the number of execution paths through the code as an indication of complexity, as it is suggested that each path must be traced if the program is to be completely understood.

All procedural programs can be represented by directed flowgraphs, using nodes to indicate blocks of sequential statements and edges to illustrate selection and iteration structures. For all single-entry single-exit code modules, McCabe's measure equates to one plus the number of decision structures (alternation and repetition) in the module:

$$v(G) = e - n + 2 \quad \approx \quad v(G) = \pi + 1$$

e = number of edges π = number of decision structures

n = number of nodes.

A significant criticism of the two previous assessment schemes (lines of code and software science) was their failure to consider the contribution of aspects such as code structure to the overall complexity of the program. In comparison, McCabe's technique appears to be quite promising, as control flow is clearly assessed. It is evident however, that this is somewhat to the detriment of the evaluation of other aspects. In particular, the complexity of all functional code blocks is considered to be equivalent, irrespective of the size or internal nature of the blocks. This means in effect that a two-line segment of sequential statements is considered to be as difficult to understand as a two-hundred-line sequential code block.

Two other counting anomalies arising from the original metric definition have also been identified. The first concerns the consideration of multiple-exit code modules [15]; the second is related to the consistent interpretation of compound predicate structures [16], [17].

In addition, no recognition is made of system size, nesting levels, data flow or program modularization. It therefore seems doubtful that this metric can be considered as an adequately comprehensive indicator of total complexity. Moreover, due to its foundation in code decisions or code-based flowgraphs, determination of the measure can only occur after program development is complete.

d. Summary:

All of the three techniques investigated here have positive aspects; each has been successfully validated both empirically and subjectively, and each is intuitively acceptable as being in some way related to software complexity. However, criticism of the methods is also extensive. Several failings are evident for all of the techniques, particularly (i) their single-aspect consideration, (ii) the counting procedure anomalies associated with each and (iii) the late availability of results under each of the schemes.

(i) Since there are many attributes thought to contribute to complexity, it would not be an easy task to design a measure which would consider every aspect. What is more, such a measurement scheme is likely to be so comprehensive as to become impractical for efficient project management. Also, combining several approaches may have the undesirable result of simply compounding the problems inherent in each. It may also be difficult to achieve an appropriate `balance' between the approaches employed so that the scheme is not dominated by one aspect e.g. size or module structure.

(ii) Generally the causes of counting problems are two-fold: the first is the lack of succinct underlying theory behind the actual measurement techniques and the subsequent looseness in their definition; the second is the frequency of change in technology which afflicts the computing industry. When the measurement schemes were developed, they were certainly relevant to development methods of their day. However with new techniques for software production constantly emerging, the appropriateness of the metrics has been reduced.

(iii) To obtain an objective measure, the assessment of a tangible product was considered to be necessary - the obvious software product was the source code. Hence this has been used in most quantification techniques. However this does mean that measurement extraction can only be performed very late in the development process. Furthermore, since design conventions and notations are still very broad and lacking in standards, measurement from these representations may also be difficult.

Despite these extensive problems the three metrics are still widely supported, based solely on the very strong empirical evidence which has been observed in many studies. Most of this evidence has been provided using linear correlation data, supporting the existence of linear relationships between the metrics and various aspects of the software development process and/or the final software product. These aspects are said to be indicative of complexity e.g. the number of development errors, development time duration or the time needed for error location or system enhancement. Using Pearson's product-moment correlation coefficient (r), relationship levels of greater than 0.90 have been reported [18], [19]. In many cases this has been the extent of the validation undertaken and predictions based on these findings have been subsequently performed.

Reliance on conclusions based solely on the correlation data may be questionable, however. Lister [20] points out that although the correlation procedure is appropriate for investigating the relationship between random variables, this random nature has been seldom proved in measurement studies. Furthermore, a high correlation may indicate the existence of a linear relationship, but it provides no insight into the validity of the relationship itself.

III. EXPERIMENTAL WORK

To examine whether such confidence should be placed solely on correlation data, a set of twenty-eight high-level programs were analyzed in conjunction with development information.

(i) Sample - The software used in this investigation was a financial database statistics extraction package written by a professional programmer with eleven years of programming experience. The system was written in Clipper[1] and consisted of twenty-eight newly developed program modules. The smallest was twelve lines long, the largest was 123 lines.

(ii) Procedure - All logical compilation errors which occurred during the development of each program were recorded as they arose until the program was complete. Measures were then extracted from the modules for the three metric techniques discussed previously. Total and non-commentary lines of code values were derived as representative of the line-based measures (TLOC and NCLOC), n_1, n_2, N_1, N_2, n and N were chosen for software science and the number of decisions and v(G) were extracted for the topological measures (DEC and VG). Correlation and regression techniques were then employed to determine the existence and significance of any relationships.

(iii) Results - Correlations between the number of development errors and the values of the specific metrics are shown in Table 1.

TABLE 1. Correlation between development errors and complexity metrics

	ERR	TLOC	NCLOC	n_1	N_1	n_2	N_2	n	N	DEC	VG
ERR	1.00	0.80	0.82	0.83	0.78	0.65	0.74	0.73	0.76	0.51	0.50

Apart from the two decision-based metrics (DEC and VG) all of the metrics showed fairly good correlations ($0.65 \le r \le 0.83$) with the development errors. If we were to end the experiment there, it would be tempting to suggest that these measures, particularly TLOC, NCLOC and n_1, were adequate indicators of complexity in terms of development error occurrence, and that they may in fact be used to predict the likelihood of development errors (due to complexity) in other projects, as has been done in the past.

Regression analysis, however, lessens the validity of such a suggestion. Using the R^2 statistic (the square of the r correlation measure) as an indication of the explanatory power of prediction models, the following levels (Table 2) were obtained for the metric-based estimation of development errors:

[1] Clipper is a trademark of Nantucket.

TABLE 2. R^2 levels for metric-based error prediction

TLOC	NCLOC	n_1	N_1	n_2	N_2	n	N	DEC	VG
0.64	0.67	0.69	0.61	0.42	0.54	0.54	0.58	0.26	0.25

The explanatory capabilities of the various metrics appear to be lower than we would require to obtain accurate estimates of error occurrence. (This result can, of course, be derived from the correlation statistics, because of the direct square relationship between *r* and R^2; this is seldom performed, however.) Furthermore, the regression procedure allows the examination of the residual plots associated with the prediction models. These should show a constant band of data points, evenly dispersed about the mean (at 0 on the vertical) with a constant variance and a random nature. All of the plots derived from the above prediction models failed to conform to these requirements. This is likely to be due at least in part to the impossibility of obtaining a negative value for the number of errors, resulting in a skewed distribution for this variable. This suggests that this commonly used method of complexity metric validation may be flawed, particularly for smaller samples.

A stepwise linear regression procedure was then performed to determine whether a combination of metrics could provide more effective error prediction (Table 3).

TABLE 3. Summary of stepwise regression procedure for dependent variable errors

	VARIABLE	NUMBER	PARTIAL	MODEL			
STEP	IN		R**2	R**2	C(P)	F	PROB>F
1	N1A	1	0.6925	0.6925	32.4982	58.5538	0.0001
2	TLOC	2	0.0277	0.7202	29.4102	2.4742	0.1283
3	N2A	3	0.0909	0.8111	14.7133	11.5439	0.0024
4	N1B	4	0.0712	0.8823	3.6336	13.9059	0.0011

An R^2 of 0.88 appears promising at first; however, the *beta* coefficient of the N2A variable (n_2) in the predictive equation is negative (-0.09902982). This implies that (all other variables being held constant) the incidence of errors should actually decrease with a corresponding increase in the number of distinct operands in the representation. This would seem to encourage the use of a large number of operands in order to reduce the incidence of errors in the code development phase. This is an interesting finding, but one which is most counter-intuitive.

These results provide a difficult choice. If we choose to use single variable based predictions (such as TLOC, NCLOC or n_1), the explanatory powers are low and the residual plots reveal a lack of adequacy in the models. If we therefore utilize the full stepwise model, counter-intuitive parameters are employed.

IV. CONCLUSIONS

Despite widespread acknowledgement of the problems associated with the three techniques, support for each has continued because of the often strong correlative experimental evidence obtained. As a result, estimation of process and product attributes is frequently performed based on this data. This study has attempted to illustrate the problems inherent in this procedure. It is acknowledged that this study, like many others in this area, has several limitations. Only one small system implemented in one language by one programmer was evaluated. This, however, does not completely invalidate the results achieved - the same problems are likely to occur in experiments involving large, team-developed systems, if adequate statistical procedures are not employed. It seems clear that the sole use of correlation data as evidence for metric use and validation is misdirected, and that other statistical methods should be applied if truly valid results are to be obtained.

REFERENCES

[1] M.H. Halstead, Elements of Software Science. New York: Elsevier North-Holland, 1977.

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

[3] R.W. Osborn, "Theories of productivity analysis'" Datamation, pp. 212-216, Sept. 1981.

[4] R.E. Carlyle, "High cost, lack of standards is slowing pace of CASE," *Datamation*, pp. 23-24, Aug. 1987.

[5] J.L.F. De Kerf, "APL compared with other languages according to Halstead's theory," *ACM SIGPlan*, pp. 31-39, Jan. 1986.

[6] D.M. Miller, J.W. Howatt, R.S. Maness and W.H. Shaw, "A software science counting strategy for the full Ada language," *ACM SIGPlan*, pp. 32-41, May 1987.

[7] B. Curtis, I. Forman, R. Brooks, E. Soloway and K. Ehrlich, "Psychological perspectives for science," *Information Processing & Management*, vol. 20, pp. 81-96, 1984.

[8] N.S. Coulter, "Software science and cognitive psychology," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 166-171, Mar. 1983.

[9] N. Beser, "Foundations and experiments in software science," *ACM SIGMetrics*, pp. 48-72, 1982.

[10] G. Davies and A. Tan, "A note on metrics of Pascal programs," ACM SIGPlan, pp. 39-44, 1987.

[11] B. Beizer, *Software System Testing and Quality Assurance*. New York: Van Nostrand Reinhold, 1984.

[12] B. Ramamurthy and A. Melton, "A synthesis of software science measures and the cyclomatic number," *IEEE Trans. Software Eng.*, vol. SE-14, pp. 1116-1121, Aug. 1988.

[13] W.B. Samson, D.G. Nevill and P.I. Dugard, "Predictive software metrics based on a formal specification," *Information and Soft. Tech.*, vol. 29, pp. 242248, June 1987.

[14] I. Vessey, "On program development effort and productivity," *Information & Management*, vol. 10, pp. 255-266, 1986.

[15] W.A. Harrison, "Applying McCabe's complexity measure to multiple-exit programs," *Software - Practice and Experience*, vol. 14, pp. 1004-1007, Oct.1984.

[16] G.J. Myers, "An extension to the cyclomatic measure of program complexity," ACM SIGPlan, pp. 61-64, Oct. 1977.

[17] E.J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Software Eng.*, vol. SE-14, pp. 1357-1365, Sept. 1988.

[18] P.M. Zislis, An Experiment in Algorithm Implementation. Purdue University CSD-TR 96, 1973.

[19] K. Christensen, G.P. Fitsos and C.P. Smith, "A perspective on software science," *IBM Syst. Jnl.*, vol. 20, pp. 372-387, 1981.

[20] A.M. Lister, "Software science - the emperor's new clothes?," *Australian Comp. Jnl.*, vol. 14, pp. 66-70, May 1982.