1981

An Investigation of the Relationship between Initial and Final Programming Effort Estimates

Andrew S. Wang

Report Number:
80-365

An Investigation of the Relationship between Initial and Final Programming Effort Estimates

CSD TR-365
May 18, 1981

Andrew S. Wang

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

This study considered the ability of several effort measures to predict both total effort and debugging effort when applied to the first compiled version of a program. The data came from twenty-seven Fortran programs ranging in size from several lines to nearly two hundred and requiring from one to over four hours to produce. All measures were found to be consistent (i.e., the measures applied to final versions correlated well with the same measures applied to initial compiled versions). Measures based on the software science E, cyclomatic complexity v(G), and lines of code all were good predictors of both debugging time and total time. A measure "program changes" was a moderately good estimator of debug time. The "number of runs" was the worst predictor in both situations.

Keywords and Phrases: programming effort measures, software science, cyclomatic complexity, lines of code, programming time estimation

1. Introduction

Because of the continuing increase in software cost and the lack of understanding of software production and maintenance [Boeh73, Wood80], there is an urgent need to develop techniques for estimating the total amount of effort involved in producing software. An acceptable measure should meet the requirements of being accurate, objective, cost-effective, and automatically collectable. One of the first effort measures which was widely used was "lines of code" [Jone78]. We define lines of code (*LOC*) as the total number of lines in a listing excluding all comment lines, continuation lines, and blank lines. Although

LOC is actually a size measure, previous studies have indicated that it may be related to programming time as well [Feuc79, Bail80].

Two other effort measures which have achieved some degree of acceptance are McCabe's cyclomatic number $v(G)$ [McCa76] and the software science effort measure E [Hals77]. In graph theory, the cyclomatic number of a strongly connected graph is the maximum number of linearly independent circuits. McCabe proposed that the control structure of a program can be interpreted as a strongly connected graph. Looking at a program from this graphical point of view, he claimed that $v(G)$ measures the number of independent paths through a program. The use of $v(G)$ as an effort measure was based on the assumption that programming effort is primarily a function of the number of unique paths in a program. Since each path and its criteria for selection must be understood and tested, the presence of many paths would require a large amount of mental effort. Previous experiments have shown that the number of unique paths is significantly related to effort [Schn79].

The software science effort measure E incorporates two complexity factors: program size and program difficulty. Program size is measured by the software science volume V which can be thought of as a count of total mental steps required to write a program. Program difficulty is represented by the difficulty measure D which indicates the average mental units per mental step for a particular program. Thus, the total number of mental units E required to generate a program should be given by:

$$E = D * V \tag{1.1}$$

where the units for E are claimed to be elementary mental discriminations. With an assumption that each elementary mental discrimination requires a constant unit of time [Stro56], E can be converted to desired time units through

division by an appropriate constant.

Various studies and experiments have shown that the software science effort measure E may be useful. It is able to estimate both programming time and program comprehension with acceptable accuracy [Hals77, Fitz78]. Woodfield [Wood80] extended this effort measure to account for modularity factors as well. The original software science effort measure was developed using single module programs. To extend the effort measure to modularized programs, it was assumed that the effort measure should be applied as if the entire program were a single unit and not as if it were a set of modules. This was the integrated model, E^{INT}, as proposed by Hunter and Ingojo [Hunt77].

Another method to estimate effort for multi-module programs using software science is to view the entire program as a set of "physical modules", such as subroutines or procedures, and to compute the sum of the estimates of individual modules. This resulted in the physical module model, E^P [Wood80]. Furthermore, we can divide each physical module into a number of "logical modules" (each of which is small enough to comprehend and implement) such that the effort estimate for a physical module is simply the sum of the effort estimates of the individual logical modules. The extended software science effort measure based on this scheme is called the logical module model, E^L. It should be clear that every program contains at least one physical module and each physical module contains at least one logical module. Since the software science effort measure grows faster than linearly with respect to program size, the summation of the efforts for the parts should be less than the effort for the program taken as a single unit. Therefore, for multi-module programs, we will find that

$$E^P < E^{INT} \tag{1.2}$$

Similarly, if a program contains a large physical module which can be divided into several logical modules, the following condition will be true:

$$E^L < E^P \tag{1.3}$$

In general,

$$E^L \leq E^P \leq E^{INT} \tag{1.4}$$

The extended models, E^P and E^L, were validated using two sets of experimental data which were collected on two different occasions. Results indicate that the extended effort measures yielded a significant improvement over the original one.

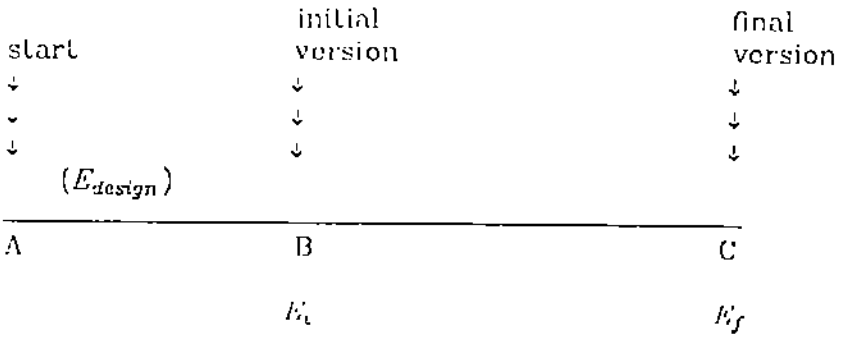
Besides the effort measures described above, two others considered in this study are total number of runs (*RUNS*) and program changes (*PC*). Previous research [Basi79] has shown that the number of job submissions can be an indication of program development effort and that a good development methodology leads to a small number of job submissions. Program changes [Duns78] were used as a programming effort measure and appeared to correlate with total error occurrences in developing a program. A "program change" is a result of textual changes between successive versions of a program. The rationale for using program changes as an effort measure is that one program change is a representation of effort expended on a single abstract instruction. In this study, we included these two effort measures as well.

Unfortunately, all of the measures mentioned above suffer one common problem. They cannot be used predictively. *LOC*, $v(G)$, and the two software science metrics V and D, from which E is computed, are all derived from a program's final version. Similarly, since *RUNS* and *PC* are "process-oriented measures" [Basi79] (representing characteristics of the development process

itself) they cannot be collected until the program is completed and the entire development process terminates.

For a measure to be truly useful in effort prediction, it should be obtainable as early as possible. However, it is quite difficult to derive an objective programming effort estimator without a great deal of information about the program before coding is started. Nevertheless, it is reasonable to expect that as the problem specification is established, algorithms and data structures selected, and the initial version prepared, we can use appropriate methods at various points to make progressively more accurate estimates. We now have several well-developed effort estimators that may be applied to the final version. In order to predict effort earlier in the process we propose using a "back-to-front" approach. As a first step, we can move back to the stage of the first clean compiled version (later referred to as the "initial version") and investigate the behavior of different effort estimators at this stage. By analyzing the first clean compiled version of the program, we might be able to estimate the *total* programming effort at this earlier stage, and thus use the results to predict the amount of effort remaining.

The completion of the initial version of a program represents a major commitment on the part of the programmer. That is, the programmer has spent a substantial amount of effort in the design process and normally has no intention of making drastic changes in the program after this point. The final program evolves through intermediate versions each representing only changes of limited nature. Thus, the effort measure derived from the initial version could be a good indicator of the entire programming effort and could be used to predict the effort remaining after the initial version. The idea stated above can be shown graphically as follows:



Let E_{design} = actual time spent up to B

Then, predicted remaining effort at B
= predicted total effort at B - actual time spent up to B
= $E_i - E_{design}$

and analyzed actual remaining effort at B
= predicted total effort at C - actual time spent up to B
= $E_f - E_{design}$

If $E_i \simeq E_f$ then $(E_i - E_{design}) \simeq (E_f - E_{design})$

hence predicted remaining effort at B

$\simeq$ analyzed actual remaining effort at B

Our ultimate goal is to develop a family of effort estimators to be applied to different milestones during program development. It is clear that since less information is available at earlier milestones, greater error tolerances must be allowed.

2. Source of data and experimental procedures

The data analyzed in this study consists of all submitted program versions of twenty-seven Fortran programs. This set of data was collected in three four-hour programming competition sessions conducted during the summer of 1980 [Wood80]. The participants were twelve computer science graduate students experienced in Fortran programming. They were divided into three teams of four programmers each. During each session, each team was given four problems selected from diverse areas. All subjects worked in an isolated environment that permitted a very high degree of concentration. They worked

on their programs in a competitive atmosphere to see which team could solve their four problems the fastest. An observer was present during each session to record the actual programming times in a consistent fashion.

The actual programming time of each program was recorded via two major components, the design time and the debug time. That is,

$$TOTAL\ TIME = DESIGN\ TIME + DEBUG\ TIME \tag{2.1}$$

DESIGN TIME started when the programmer finished reading the problem specification and ended when the first clean compiled version was achieved. *DEBUG TIME* was defined to be the time elapsed between the first clean compiled version (initial version) and the final version. All versions of the twenty-seven programs were examined using a Fortran analyzer. Several measurements were obtained from all versions of each program. They included lines of code (LOC), cyclomatic number ($v(G)$), software science program length (N), and the effort measure E in terms of the integrated model (E^{INT}), physical module model (E^{P}), and logical module model (E^{L}). Since we collected all versions of each completed program, the information on total number of runs was readily available. Program changes were collected using an algorithmic counting procedure.

In summary, the data have the following general properties:

No. of completed programs	27
No. of distinct programs	11
No. of program versions	197
Range of program sizes (LOC)	17-195 lines of code
Range of program length (N)	90-1114
Range of cyclomatic number ($v(G)$)	6-14
Range of $RUNS$	2-17

Range of *PC*	0-22
Range of *TOTAL TIME*	0.57-4.37 hours
Range of *DESIGN TIME*	0.33-3.00 hours
Range of *DEBUG TIME*	0.02-2.60 hours

3. Predicted properties of effort measures

As explained before, when the initial verison is similar to the final version, effort measures derived from initial and final versions should be similar. In such a case we propose that

$$E_i \simeq E_f \ , \ N_i \simeq N_f \ , v(G)_i \simeq v(G)_f \ , \ LOC_i \simeq LOC_f$$

Since the software science effort measure is claimed to estimate the total amount of programming time instead of debug time, we predicted that E would correlate with *TOTAL TIME* better than with *DEBUG TIME*. On the other hand, McCabe's complexity measure $v(G)$ was proposed as a measure of the amount of work required to test a program. Thus we anticipated that $v(G)$ would correlate with *DEBUG TIME* better than with *TOTAL TIME*. *RUNS* and *PC* are both related to program testing effort. Thus, we assumed that they would correlate with *DEBUG TIME* better than with *TOTAL TIME*.

4. Comparison criteria

In previous studies [Wood80] five comparison criteria were used in evaluating effort measures' ability to estimate programming effort. For a set of programs, we computed the correlation coefficient between the estimated programming times and the actual times. A higher correlation coefficient indicates a stronger realtionship between the two. One correlation coefficient is the *Spearman rank coefficient*, a non-parametric statistic [Holl73] measuring the association between two different sets of corresponding values from rank

scales. Another way of characterizing correlation is the *Pearson product moment correlation coefficient.*

The actual implementation time is assumed to be a linear function of an effort measure which can be converted to time using some conversion formula. The *average relative error* for n programs in the estimate is computed as follows:

$$RE = \frac{\sum_{i=1}^{n} \frac{T_i - C_i}{T_i}}{n} \qquad (4.1)$$

where T_i is the actual programming time for the i^{th} program, C_i is the corresponding estimated time calculated using the appropriate conversion formula for a particular measure, and n is the number of programs in the sample. If the *average relative error* is low, the effort measure tends to predict the correct time on the average. The *average absolute relative error* is calculated as follows:

$$|RE| = \frac{\sum_{i=1}^{n} \frac{|T_i - C_i|}{T_i}}{n} \qquad (4.2)$$

A small average absolute relative error shows that the errors in estimation are consistently small without compensating cases of over- and under-estimation. The *mean square error* is an estimate of the variance between the actual programming times and the estimated times from different models. The mean square error is computed as follows:

$$MSE = \frac{\sum_{i=1}^{n} (T_i - C_i)^2}{n - 2} \qquad (4.3)$$

A small *mean square error* indicates that the estimated values are, in an

absolute sense, close to the actual value and that there are very few outliers.

5. Results

TABLE 1.

Relationships between initial and final estimates						
init.	final	*Spearman*	*Pearson*	*RE%*	*|RE%|*	*MSE*
E_i^{INT}	E_f^{INT}	.98	.97	5.49	11.16	2.59
E_i^P	E_f^P	.99	.97	3.55	10.37	2.30
N_i	N_f	.96	.97	4.45	6.38	4214.76
$v(G)_i$	$v(G)_f$	.95	.95	2.16	5.51	7.72
LOC_i	LOC_f	.98	.99	2.64	5.01	52.80

Table 1 shows the relationships between initial and final estimates for five effort estimation models. Each row corresponds to one model. It should be evident that the five effort measures derived from the initial versions are not substantially different from those derived from the final versions. The correlation coefficients between these two sets of measures are uniformly high and the average relative differences are below 12%. The three comparison criteria, RE, $|RE|$, and MSE, indicate the relative difference (instead of relative error) between two sets of values. Regression was not applied at all in this case. The magnitude of MSE depends on the units of the measurement used in the particular model. This explains the high MSE value in the third row for the two program length measures, N_i, and N_f. From the above observations, we can conclude that initial effort estimates are very close to final effort estimates. This is exactly what we expected.

TABLE 2.

Comparisons of 13 measures with respect to *TOTAL TIME*						
Measure	*Spearman*	*Pearson*	*RE%*	*|RE%|*	*MSE*	Reg. Coeff?
E_i^{INT}	.84	.59	-112	137	46.83	No
E_i^P	.81	.42	-86	83	24.05	No
E_f^{INT}	.86	.62	-126	147	52.67	No
E_f^P	.82	.43	-89	91	29.22	No
E_f^L	.88	.79	30	36	.68	No
N_i	.77	.68	-16	35	.53	Yes
$v(G)_i$	.77	.71	-15	35	.50	Yes
LOC_i	.82	.71	-16	34	.49	Yes
N_f	.79	.70	-15	34	.51	Yes
$v(G)_f$	.80	.73	-15	34	.48	Yes
LOC_f	.82	.73	-15	33	.47	Yes
RUNS	.22	.09	-34	60	1.00	Yes
PC	.43	.19	-33	58	.97	Yes

A comparison of several models for estimating programming time (*TOTAL TIME*) is shown in Table 2. The first five rows labeled E_i^{INT}, E_i^P, E_f^{INT}, E_f^P, and E_f^L represent the software science E metric derived from the initial version (i) and final version (f) using the integrated model (INT), physical module model (P), and logical module model (L) respectively. The next six rows correspond to the analysis of the estimates derived from initial and final versions using regression models based on program length (N), cyclomatic complexity ($v(G)$), and lines of source code (LOC) respectively. The last two rows represent regression models based on *RUNS* and *PC*. The last column in Table 2 indicates whether linear regression analysis was applied to each model.

Among the thirteen effort estimation models, *RUNS* and *PC* yield the lowest correlation values. This result is not surprising since *RUNS* and *PC* are only associated with program debugging effort and are expected to correlate with

DEBUG TIME better than with *TOTAL TIME*. It should be noted that the *MSE*, *RE*, and |*RE*| values for all models based on regression analysis will be relatively low. This is because regression minimizes the *MSE* value and affects *RE* and |*RE*| as well.

TABLE 3.

Comparisons of 13 measures with respect to *DEBUG TIME*

Measure	*Spearman*	*Pearson*	*RE%*	|*RE%*|	*MSE*	Reg. Coeff?
E_i^{INT}	.68	.30	-1086	1086	62.45	No
E_i^P	.65	.13	-674	674	31.90	No
E_f^{INT}	.74	.35	-1123	1123	69.69	No
L_f^P	.65	.15	-713	715	38.28	No
E_f^L	.69	.57	375	377	1.50	No
N_i	.63	.37	-109	128	.24	Yes
$v(G)_i$	.69	.40	-111	131	.23	Yes
LOC_i	.71	.44	-97	117	.22	Yes
N_f	.67	.39	-99	118	.23	Yes
$v(G)_f$	.71	.45	-101	121	.22	Yes
LOC_f	.73	.46	-89	109	.21	Yes
RUNS	.46	.40	-125	149	.23	Yes
PC	.63	.45	-110	131	.22	Yes

Table 3 is similar to Table 2 except that all models are compared with respect to their ability in estimating *DEBUG TIME* instead of *TOTAL TIME*. In terms of *Spearman rank correlation coefficient*, *RUNS* still reports the lowest value and *PC* is no better than the other models (except *RUNS*).

Comparing Table 2 and Table 3, it is easily seen that for each model except *RUNS* and *PC* the performance in estimating programming time is better for *TOTAL TIME* than *DEBUG TIME*. It is interesting to notice that although McCabe's complexity measure, $v(G)$, is claimed to be closely correlated with program testing effort, it seems to work better for our data in estimating total

program development effort than in estimating debugging effort.

6. Conclusions

We found that *RUNS* and *PC* are not good metrics for estimating program debugging effort. Our result shows that the actual time spent in program debugging seems to correlate poorly with the number of runs. Dunsmore and Gannon [Duns78] proposed the effort measure of program changes and demonstrated that program changes were highly correlated with errors. Basili and Reiter [Basi79] showed that program changes were minimal when a good software development method was used. According to Basili [Basi80], monitoring program changes and errors during the software development process can provide us with information about the quality of the product. Our result indicates that several other effort measures are better for estimating program debugging time than either number of runs or program changes.

Our results suggest that initial effort estimates based on E, LOC, $v(G)$, or their combinations can be good approximations of their final estimates and can be used for effort prediction at the point when the first clean compiled version is achieved. We believe that this situation will hold in cases where the program is carefully, rigorously designed. In such a case the major program structure should remain invariant during the entire development process. More experimental analysis on the relationship between initial and final effort estimates needs to be done with other languages and programs in different size ranges.

7. References

[Bail80] Bailey, J. W., and Basili, V. R., "A meta-model for software development for resource expenditures," *Technical Report TR-935*, Computer Science Technical Report Series, University of Maryland, Aug. 1980.

[Basi79] Basili, V. R., and Reiter, R. W., "An investigation of human factors in software development," *Computer*, pp. 21-38, December 1979.

[Basi80] Basili, V. R., "Changes and errors as measures of software development," *Tutorial on Models and Metrics for Software Management and Engineering*. V. R. Basili (ed.), Computer Science Press, Los Alamitos, Calif., pp. 62-64, 1980.

[Boeh73] Boehm, B. W., "Software and its impact: a quantitative assessment," *Datamation* 19, 5, pp. 48-49, May, 1973.

[Duns78] Dunsmore, H. E. and Gannon, J. D., "Programming factors - language features that help explain programming complexity," *Proceedings of ACM 78*, Washington, D.C., pp. 554-560, December 1978.

[Feue79] Feuer, A. R., and Fowlkes, E. B.,"Some results from an empireal study of computer software," *Proc. of the 4th International Software Eng. Conf.* Munich, Germany, pp. 351-355, September 1979.

[Fitz78] Fitzsimmons, A., and Love, T., "A review and evaluation of software science," *Computing Surveys*, 10, 1, pp. 3-18, March 1978.

[Hals77] Halstead, M. H., *Elements of Software Science*, New York: Elsevier, 1977.

[Holl73] Hollander, M., and Wolfe, D. A., *Nonparametric Statistical Methods*, John Wiley & Sons, 1973.

[Hunt77] Hunter, L., and Ingojo, J., "Conservation of software science parameters across modularization," *Proc. ACM National Conf.*, pp. 189-194, October 1977.

[Jone78] Jones, T. C., "Measuring programming quality and productivity," *IBM Systems Journal 17*, pp. 39-63, 1(1978).

[McCa76] McCabe, T. J., "A complexity measure," *IEEE Trans. Softw. Eng.*, SE-2, 4, pp. 308-320, December 1976.

[Schn79] Schneidewind, N. F., and Hoffman, H., "An experiment in software error data collection and analysis," *IEEE Trans. on Soft. Eng., SE-5*, 3, pp. 276-286, May 1979.

[Stro56] Stroud, J. M., "The fine structure of psychological time," *Information Theory in Psychology*. The Free Press, Chicago, Ill., 1956.

[Wood80] Woodfield, S. N., "Enhanced effort estimation by extending basic programming models to include modularity factors," Ph.D. Th., Purdue Univ., December 1980.