1978

Quantitative Estimates of Debugging Requirements

Linda M. Ottenstein

Report Number:

78-282

QUANTITATIVE ESTIMATES OF DEBUGGING REQUIREMENTS

Linda M. Ottenstein

Computer Sciences Department
Purdue University
West Lafayette, Indiana 47907[1]

August 1978
CSD-TR 282

Abstract: A major portion of the problems associated with software development might be blamed on the lack of appropriate tools to aid in the planning and testing phases of software projects. As one step towards solving this problem, this paper presents a model to estimate the number of bugs remaining in a system at the beginning of the testing and integration phases of development. The model was tested using data currently available in the literature. Extensions to the model are also presented which can be used to obtain such estimates as the expected amount of personnel and computer time required for project validation.

[1]Current Address:
Dept. of Math. and Computer Sciences
Michigan Technological University
Houghton, Michigan 49931

QUANTITATIVE ESTIMATES OF DEBUGGING REQUIREMENTS

INTRODUCTION

The management of large-scale software system development is a significant problem. The problem is not with the type of management organization, but with scheduling and planning: determining how long a project is going to take and determining when the project is done. Upper management, in general, is accustomed to dealing with tested models and concrete numbers when planning and reporting on progress. Software people, on the other hand, have tended to rely on ad hoc methods. For instance, timing estimates may be made by comparing the assumed difficulty of the current project with the difficulty of an earlier project. Frequently, however, many factors have changed and cannot be accounted for. These might include new personnel, a different programming language, or a change in programming techniques.

The reputation of these methods is not good. As Zelkowitz summarizes in a recent Computing Surveys, "Software is often delivered late. It is frequently unreliable and usually expensive to maintain" [Zel 78]. Despite the inadequacy of the ad hoc methods, there have been no generally accepted alternatives, and until recently, the idea of being able to solve this problem was considered preposterous.

The importance of the scheduling and planning problem becomes clearer when one considers the ever-increasing role software plays in everyday living. Medical equipment, defense systems, air traffic control, bank accounts and an ever increasing number of other vital functions in our lives are controlled by computers. Thus it is vitally important for the software to be available as promised.

A significant aid in solving a portion of these problems would be an accurate estimate of the number of bugs in the software at the beginning of the validation phase. This could then be used to predict the amount of personnel and computer time needed for the validation of the project, as well as assess the product's reliability. Finding a model to provide this estimate was the purpose of this research.

BACKGROUND

In the early days of computing, managers obtained rough estimates for the number of bugs in a module by assuming there was one bug in every 60 lines of code or perhaps in every 100 lines of code depending on their optimism and experience. As Shooman and Bolsky's [ShB 75] data indicates this may have actually been reasonably accurate for some languages and projects. In this decade, however, a more thorough understanding of what is actually happening and a more reliable estimate for the number of bugs expected in a program is needed. Work related to this problem has been approached from two main directions.

Some research has been aimed at establishing complexity[1] measures of software. One test of a good measure is its indication of the error-proneness of the software. Most of the complexity measures suggested so far have been based on the control-flow graph of the program [BeS 74, McC 76, Mye 77, ScH 77a, ScH 77b, Sul 73]. These complexity measures have been shown to be indicative of characteristics such as the number of errors in a program. Thus they can be used to determine which of several programs probably has more errors. There does not, however, appear to be a way of extending them to be predictive, that is to be able to use them to say how long a particular program should take to understand or how many bugs should be found in a particular module.

In contrast other research is based on a phenomenological approach to the study of programming [Aki 71, LiT 77, MoB 77, Tha 75, Tha 76]. In these works it is hypothesized that there exist measurable phenomena which are correlated with characteristics of the software. Given metrics to measure

---

[1]In this thesis when we use the term complexity, we are not referring to the technical meaning of programming complexity which measures time and space of executing programs. Instead, we are using the term in the more general English usage to denote the amount of human effort required to understand the program text. Thus in a sense what we mean is the static complexity of the program.

the appropriate attributes of the software, statistical analysis can be performed to find the relationships between the attributes of the software and the desired characteristics. It might be possible for the results of this type of study to be used in a predictive sense for similar projects in the same language, although this is not yet clear. It does not appear feasible, however, to be able to generalize the results obtained from this type of study.

To be able to make predictions, a general theory based on a complexity measure which is more encompassing and discriminating than the control-flow measures is needed. The complexity measure, E, as derived in software science has been found to predict well the time to implement and to understand programs which were written in several languages [GoH 75, Hal 75]. A modification to this complexity has also been used to calculate error rates for estimating delivery dates [Klo 77]. Several reports have also presented data that show a high correlation between E and the number of bugs found in the measured module [FuH 76, LoB 76, Fit 78].

These studies indicate that software science could provide a basis for more reliably estimating parameters of the software development including the expected number of bugs. In the next section, we will pursue this idea. A model which predicts the number of bugs to be found during integration and testing will be presented.

DEVELOPMENT OF HYPOTHESIS

Before beginning it is necessary to emphasize what is meant by the term validation bug. These are the bugs that remain after the initial module tests and are delivered with the module to the testing team for system integration. Thus they are the bugs found during the phase of system development commonly called either validation or test and integration. These are frequently characterized by being those bugs for which software problem reports are generated.

Estimating E from Akiyama's data, Funami and Halstead found a .98 correlation between E and the total number of bugs reported [FuH 76]. One would not expect this correlation to always be this high. Many other factors, such as programmer experience, method of programming, and amount of available machine time, must also have an effect on the number of bugs. It appears, however, that many of the complicating factors were relatively constant in Akiyama's experiment. The data, therefore, should be useful in discovering basic relationships.

After a cursory inspection of Akiyama's data, it was determined that the modules in his sample had undergone varying amounts of initial testing. The percentage of the total errors found during the integration ranged from 9.6% to 44% and the correlation between number of steps and these validation bugs was .51.

It is important for anyone attempting to estimate the number of bugs to be found during validation to be aware of this source of variation. If controls are not instituted to insure a uniform level of initial testing, this factor alone could cause a tremendous amount of variation between the actual number of bugs to be found and the predicted number. In Akiyama's data, for example, the mean of the percentage of total errors found during validation is 29.2% with a standard deviation of 14.9%. This gives a coefficient of variation of 50.7%. Thus predictions could easily be off by as much as 50% due just to this factor.

In order to eliminate the effects of this nonuniform initial testing, a decision was made to look at an adjusted number of delivered bugs. An adjustment factor was obtained by taking the ratio of the total validation bugs to the total number of bugs. This factor multiplied by the original total number of bugs for each module then gives an adjusted number of validation bugs which should be a better data set with which to obtain a general approximation. For Akiyama's date, the ratio was .2425. A summary of Akiyama's data including the adjusted number of validation bugs is presented in Table 1.

Because of the apparently close relationship between E and the number of bugs, it was hypothesized that an approximation to the expected number of validation bugs could be obtained from E and some constant, E_0. This

Table 1: Akiyama's data along with predictions for the delivered number of bugs.

Module	Number Statements	No. Mental Discriminations (in millions)	Total Number Bugs Found	Delivered Bugs	Adjusted Delivered Bugs	Predicted Delivered Bugs
	S	E			B_V	$\hat{B}_V$
MA	4032	170.3	102	40	25	26
MB	1329	15.3	18	8	4	8
MC	5453	322.6	146[1]	14	35	37
MD	1674	28.2	26	5	6	10
ME	2051	100.2	71	14	17	12
MF	1513	65.5	37	16	9	15
Totals	17052	702.1	400	97	96	108

[1]The 53 bugs reported in the text by Akiyama, but not included in his tables, are included here.

constant represents the average number of mental discriminations per validation bug. This approximation could be stated as[2]

$$\hat{B}_V \simeq E/E_0$$

Working with this equation, however, did not lead to satisfactory results. On closer examination of the data, it was observed that the error rate varied depending on the size of the module. That is, the larger modules requiring more mental discriminations to complete had a lower rate of errors per discrimination, and modules requiring fewer discriminations had a higher rate of errors per discrimination. Similar results were reported in the study by Motley and Brooks [MoB 77]. They found a negative correlation between the number of statements in a module and the error rate where error rate was defined as the number of errors per 100 lines of code. This indicates that as the size of the module increased the errors found per 100 lines of code decreased. Motley and Brooks felt this might have indicated that the larger modules were not as fully debugged as the smaller modules. This does not seem to be a warranted conclusion in the case of Akiyama's data, however. Table 2 shows that the error rate is generally increasing for the more traditional complexity measures, Akiyama's

[2]Since bugs occur in discrete units, what we really mean is $\hat{B}$ = round(E/E_0). For simplicity, the round is assumed and does not appear in any of the equations.

measure, C (the sum of decision symbols and subroutine calls), and the number of statements, S, as would be expected. A different conclusion, therefore, is drawn.

It is hypothesized that this decreasing error rate is the result of learning. The larger a program is the more likely there is to be duplication in the code and the more familiar the programmer will become with the operators and operands with which he is working. An approximation to the amount of redundancy should allow us to account for this learning phenomenon. To a great extent, the level of the language dictates the required repetition in the code. The level of a program, L, is inversely related to the amount of repetition. That is, a program of the highest level, 1, would have no repetition. Using L then to obtain an approximation to the portion of nonrepetitive mental discriminations, an estimate for the number of bugs might be

$$\hat{B}_V \simeq LE/E_0 \ . \qquad (1a)$$

Since L is described in [Hal 77] as being inversely related to the difficulty of the program, it appears that what we are saying here is that as the difficulty decreases, the expected number of bugs increases. It is necessary to realize, however, that as L changes, E is not independent and therefore E also changes. To see the actual effect of a change in L, we can modify (1a) using (A.4) and (A.7) which gives

Table 2: Error rates for Akiyama's data.

Module	No. Mental Discriminations (in millions)	Number Statements	Decisions and Calls	$\frac{\text{bugs}}{E}$	$\frac{\text{bugs}}{S}$	$\frac{\text{bugs}}{C}$
	E	S	C			
MB	15.3	1329	259	1.18	.0136	.069
MD	28.2	1674	241	.922	.0156	.108
MF	65.5	2513	403	.565	.0147	.092
ME	100.2	2051	512	.709	.0346	.139
MA	170.3	4032	655	.599	.0253	.156
MC	322.6	5453	914	.453	.0268	.160

$$\hat{B}_V \simeq V^*/(L^*E_0) \ .$$

It is now clear that our intuition has not misled us. As the level is increased for a given V^* (and therefore the difficulty is decreased), the estimate for $\hat{B}_V$ decreases as expected.

Using (A.7), (1a) becomes

$$\hat{B}_V \simeq V/E_0 \tag{1b}$$

An estimate for E_0 is now needed. One possibility is based on a psychological theory. Approximately 20 years ago, Miller conceptualized the idea of a basic unit of information that the short term memory of the human brain could hold for immediate recall [Mil 56]. He called these basic units 'chunks' and concluded that the human short term memory can hold approximately seven of them. More recently, however, Simon has shown the short term memory chunk capacity to be closer to five [Sim 74].

One can deduce that if a person holds 5 chunks of information in his short term memory for immediate recall, he can also operate on these same five chunks of information at any one time. Each time an operation is performed on the available information in the short term memory a result is obtained. Thus the number of input and output operands, or η_2^*, for each of these operations should be 6. From this and the basic equations, the volume of information processed and the required effort for each of these operations can be determined.

Solving (A.4) for V and substituting into (A.7):

$$E = V^*/L^2.$$

Solving (A.6) for L and substituting:

$$E = (V^*)^3/\lambda^2. \qquad (2)$$

Knowing η_2^*, we can determine V^* using (A.3). That is

$$\begin{aligned} V^* &= (2+\eta_2^*)\log_2(2+\eta_2^*) \\ &= 8 \log_2(8) \\ &= 24. \end{aligned}$$

Given the value of λ, one can determine the number of elementary mental discriminations in each operation from (2). It is assumed that the level of the language used in thought processes is approximately the same as the level used in communication. The value of λ for English has been found to be approximately 2.16 [Hal 77]. Now substituting into (2), one gets

$$E_0 = 24^3/2.16^2 \simeq 3000.$$

If the assumptions are correct, this implies that after every 3000 mental discriminations a decision has been completed. The result of this decision, whether correct or incorrect, is almost certainly either used as an input for the next operation or as an output to the environment. If incorrect the error should become apparent. Thus, an opportunity for error occurs every 3000 mental discriminations.

Using $E_0 \approx 3000$, (1b) becomes

$$\hat{B}_V \approx V/3000. \tag{3}$$

Predictions for the number of bugs found from (3) using Funami and Halstead's estimates for the software science parameters are presented in Table 1 and also in Figure 1. The correlation between the predictions and the actual data is .95 which is significant at the .005 level.

VALIDATION

Bell and Sullivan's data.—A technical report by Bell and Sullivan concerning complexity measures of programs provides the needed data to try the model in a slightly different situation [BeS 74]. During a study of complexity measures, they found that all the correct algorithms sampled from CACM had a length as defined in software science of less than 237 and all the incorrect algorithms, with one exception, had a length greater than 284.

Since these programs had undergone individual module testing by the authors, but had not been integrated and tested as part of a larger system, the above hypothesis

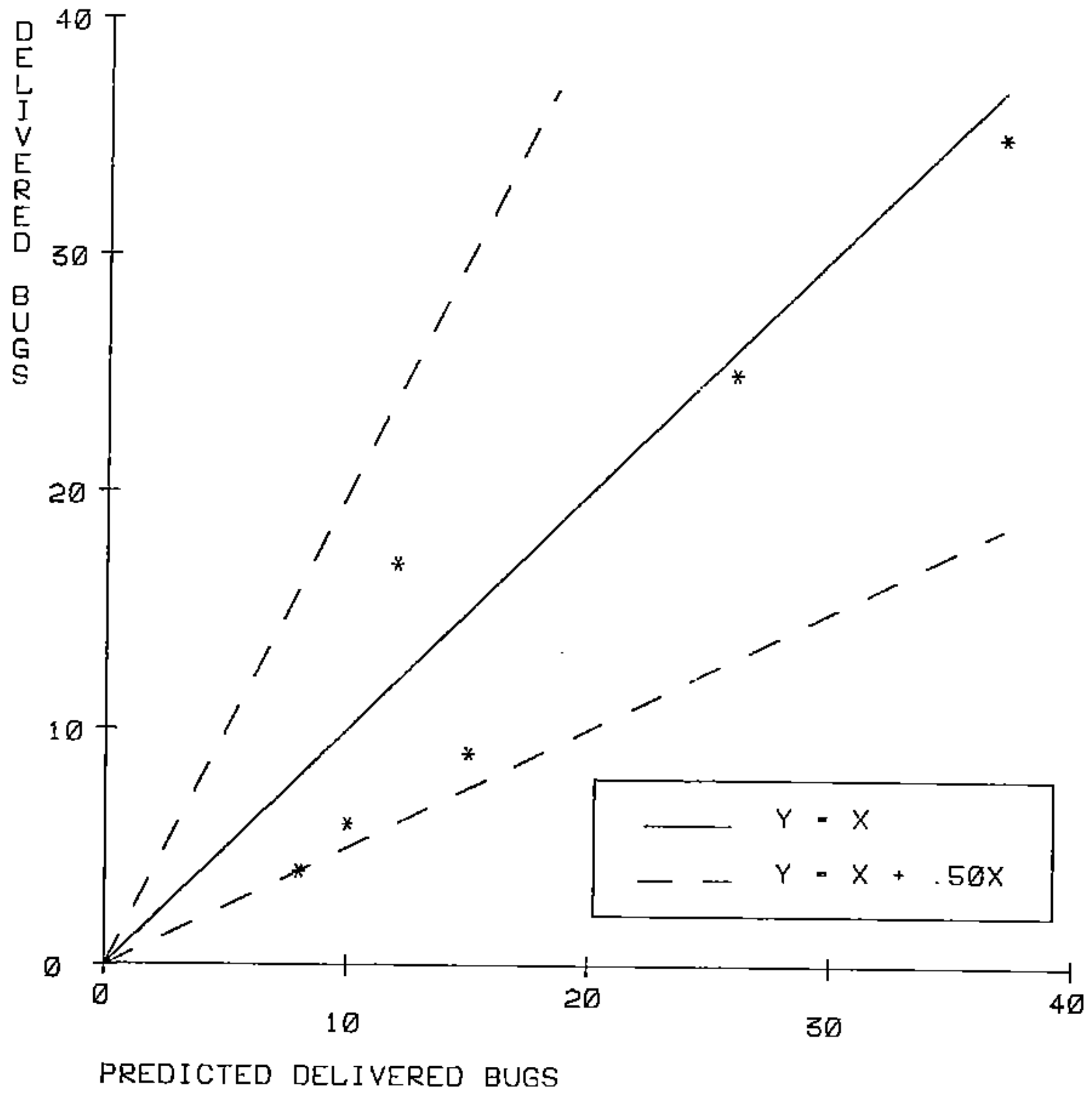

FIGURE 1: PLOT OF AKIYAMA'S DATA

might explain this phenomenon. Because equation (3) requires rounding, by setting the right hand side of it equal to 1/2, the largest number which would round to zero, an estimate of V for the largest program which can be written with no expected delivered errors is obtained. Substituting into (3), we get

$$1/2 = V/3000$$

$$V = 3000 * 1/2$$

$$V = 1500$$

Using (A.11), (A.2), and V=1500, an estimate for $\hat{N}$ can be obtained, namely,

$$\hat{N} \simeq 260$$

which is between 237 and 284.

Shooman and Bolsky's data.—Relevant data is also found in a study done by Shooman and Bolsky [ShB 75]. The information presented by them was gathered from the test and integration phase of a moderate sized control-type program designed to interphase with many other programs in a large system. It was written in a special-purpose language which was described as essentially an assembly language with powerful macro features added.

Although the number of operators and operands are not given, one can estimate N from the program length, P using (A.9). The substitution of P = 4000 given by Shooman and

Bolsky into (A.9) results in

$$N \simeq 10700.$$

Using (A.11), one obtains

$$\eta \simeq 1160.$$

Substituting into (A.2),

$$V \simeq 109000.$$

Finally

$$\hat{B}_V \simeq V/3000 = 36.$$

This estimate of the number of bugs found is within 20% of the published value of 45. The language used in the study is not truly an assembly language, therefore one can not expect to make a better prediction using approximations based on assembly languages.

Lipow and Thayer's data.—The most extensive data set available was that published by Lipow and Thayer [Tha 76, LiT 77]. Their data was gathered during the validation phase of a 115,000 statement command and control program written in Jovial. Again the software science parameters were not measured, but can be estimated from the number of executable statements and (A.10) and (A.11).

Using these approximations, the hypothesis was applied to Lipow and Thayer's data as presented in [LiT 77]. The relevant subset of their data along with the estimated $\hat{B}_V$'s are presented in Table 3. The correlation beetween the reported number of problems and $\hat{B}_V$ is .962 which is significant at the .001 level and the slope of the linear regression line forced through the origin is .994. Thus, not only is there a high degree of association between the predictions and the actual number of problems, but also the best fit coefficient differs from the actual one in the model by less than 1%.

From Figure 2, it can be seen that the majority of the predictions are within 50% of the actual values. Recalling that in the analysis of Akiyama's data, up to 50% of the variation in the reported number of validation bugs could be attributed to the lack of uniform initial testing, these results are significant.

LEARNING AND THE GROUPING OF MODULES

Although the model fits Lipow and Thayer's data as shown above, it was not immediately obvious that this was the appropriate grouping of the data to apply the model to. In [Tha 76] the data was presented both in terms of the approximately 250 individual procedures and in terms of the 25 mutually exclusive groups of procedures. Each group corresponded to a function of the system. In the previous section, we applied the model to the functions. In [Tha 76]

Table 3: Lipow and Thayer's data along with predictions for the delivered number of bugs.

Routine	Total Executable Statements	Delivered Bugs	Predicted Delivered Bugs
	EX	B_v	$\hat{B}_v$
A1	1711	26	45
A2	2327	67	63
A3	2312	54	62
A4	1789	41	47
A5	4185	79	121
B1	2438	105	66
B2	2839	95	78
C1	7227	239	221
C2	3704	69	105
C3	1324	55	33
C4	848	27	20
C5	2578	50	70
C6	1973	48	52
D1	6002	87	180
D2	842	13	20
E1	4646	144	136
F1	440	4	10
F2	1002	8	24
F3	1132	8	28
F4	1267	30	32
F5	2151	30	58
G1	7801	238	241
G2	1169	22	29
H1	340	1	7
H2	12541	466	406

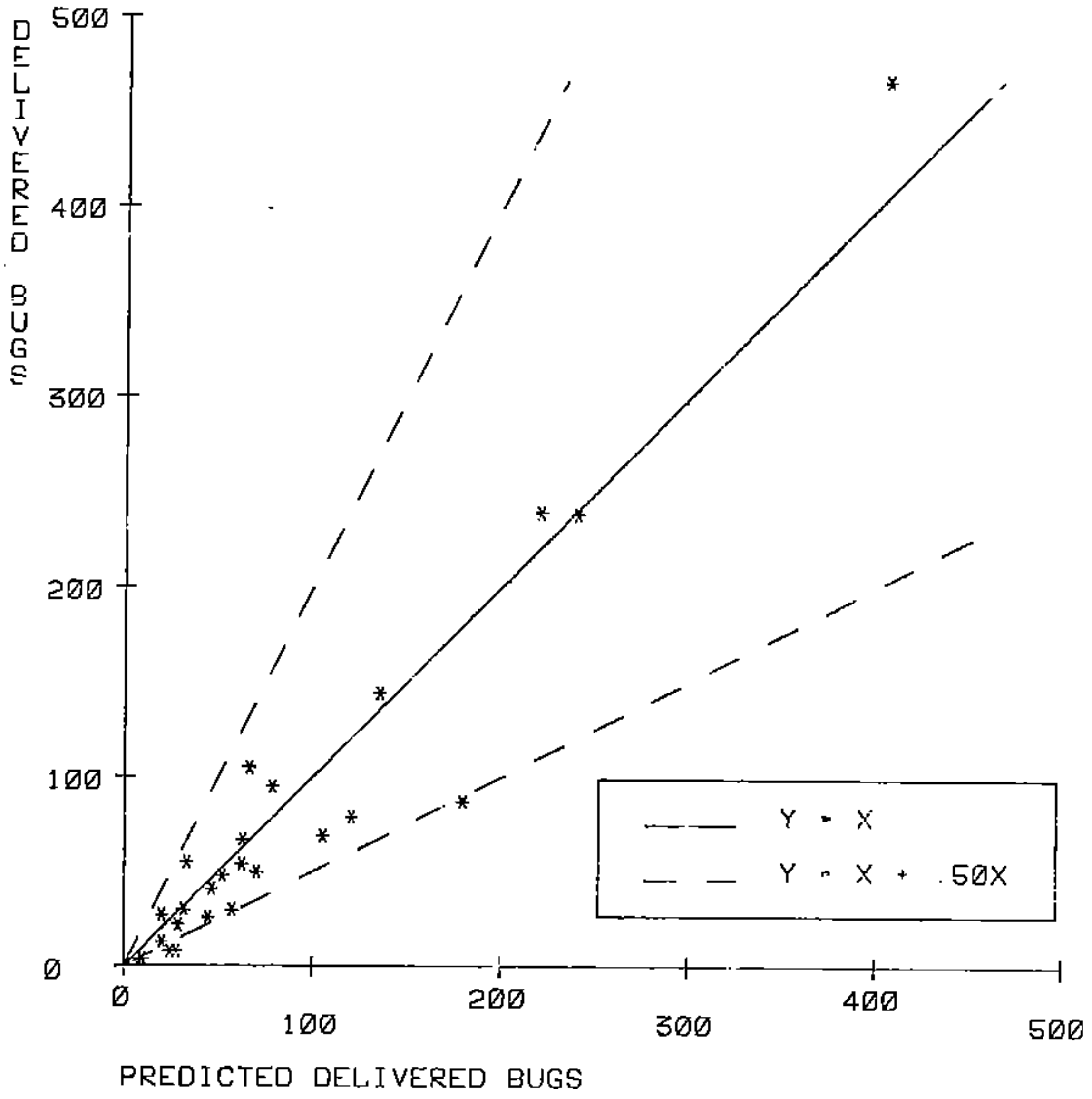

FIGURE 2: PLOT OF LIPOW AND THAYER'S DATA

it was reported that procedures in the same function tended to have the same manpower, implementation, and schedule problems. Thus, they hypothesized it was not surprising that correlations improve when the unit used for each data point is the function. We would like to propose that this occurs for a different reason.

Since the model presented is not linear,[3] it is important to determine to what grouping of the procedures to apply the model. Since the data from [Tha 76] is the only available data presented both in terms of individual procedures and the functional grouping, an experiment was performed on it. As reported earlier, the correlation between the predicted number of bugs and the actual number of problems was .962. When the model is applied to the individual procedures, the correlation is .757. Thus only 57% of the variation is accounted for in this case as opposed to 93% in the former one. It appears that a major factor is accounted for in the first case but is unaccounted for when dealing with the individual procedures. Groups of procedures that were subject to the same manpower and schedule requirements would have the variance due to these factors compounded. Therefore, that must not be the answer.

---

[3]This can be seen more clearly by noting that using (A.4) and (A.6), one gets $V = V^{*2}/\lambda$ and therefore, $\hat{B}_V = V^{*2}/(3000*\lambda)$.

Another possiblity is that the decrease in variance is explained simply because many sources of variation are averaged out by any grouping of the procedures. That is, when looking at individual procedures, one would expect to find much variation due to the individual programmer's skill, to the tightness of the schedule on which it was completed, and other factors too numerous to mention. By combining these procedures into larger groupings, these factors might average out, however, and have a lesser effect on the variation. This, then, might explain the increased correlations when dealing with the functional groupings.

Another explanation is also possible. In the development of the model to predict bugs, there was some evidence that programmers learn and improve as they work on a particular project. This was based on the finding that the more effort programs require, the lower the error rate. It is logical to assume that if learning occurs while working on individual procedures, programmers would also learn and improve their performance while working on a number of procedures that are part of the same function. As a programming team works on the second, third, or even later procedures of a function, they should be more familiar with the concepts involved, the operators and operands being used and overall how the elements are fitting together. By grouping the procedures into the functional units, the variance due to this form of learning would be decreased.

If this form of learning did not occur and the increase in variance when dealing with the individual procedures was due only to factors such as differences in individual programmer's skills, any random grouping of the procedures would have a similar decrease in the unexplained variance. To test this, an experiment was performed.

The procedures were grouped into 25 random groupings to correspond in number to the 25 functional groupings. The average correlation between the predicted number of bugs and the actual number of bugs for 10 such random groupings was .876 with a high and low of .950 and .788, respectively. Since the square of the correlation, r, is the amount of the variation that is explained, by looking at $(1-r^2)$ we see that on the average 23% of the variation was unexplained for the random groupings compared to 7% for the functional groupings. These results are summarized in Table 4. This indicates that there was some factor which was controlled in the functional grouping, but which was unaccounted for in the random groupings. A very likely candidate for this factor is the learning which it was hypothesized above would have an effect on our calculations.

Table 4: Correlations of several possible groupings using Lipow and Thayer's data.

Grouping	r	$1-r^2$
Individual procedures	.757	43%
Average of 10 random groupings	.876	23%
Functional grouping	.962	7%

OTHER MODELS CONSIDERED

Although the model presented in the previous sections fits the data quite satisfactorily, several other models based on software science parameters were investigated briefly. The high correlations found when relating the predictions from the first model to the actual number of bugs indicated that most likely the right metrics were being used. It is possible, however, to combine these measurements in many different ways, and so a few others were considered to see if an improved model could be found. For more details concerning the development of these models see [Ott 78].

Since all the model parameters are based on the same basic metrics (software science parameters), the correlation of the predictions from any model with the actual number of bugs for any of the data sets is quite high. Because of this, a criterion other than a high correlation is needed to determine if a model is usable. Each of the models requires the use of a constant to represent the error rate. For a model to be applicable to any data set, a single error rate must be found that can be used universally. Thus a test of the usefulness of a model can be obtained by applying it to several data sets, obtaining the regression coefficient from the linear best-fit equation, and determining if the coefficients are approximately equal.

In the work reported here, the form used for the best-fit equation was a linear regression equation with the regression line forced through the origin. The alternate models were applied to the four data sets used to validate the original model. To determine the closeness of the coefficients obtained from these four data sets, the mean, $\bar{x}$, standard deviation, s, and coefficient of variation, CV, of the coefficients were calculated. The coefficient of variation which is the standard deviation as a percentage of the mean is a relative measure of variation. Therefore, the lower the coefficient of variation is, the better the fit that can be expected when using the mean error rate on the individual data sets.

The regression coefficient for each of the four data sets used in the preceding sections was calculated for each of the alternate models. The results are presented in Table 5. Similar calculations were also done for our original hypothesis,

$$\hat{B}_V \approx V/c$$

and are included for comparison. The correlations of the predictions from each of the alternate models with the actual number of bugs found for Akiyama's data and the data of Lipow and Thayer are presented in Table 6.

As can be seen from Tables 5 and 6, a high correlation between predictions and actual bugs found for a particular data set does not necessarily indicate a useful model. For any particular data set, all the models considered produced predictions which were very highly correlated with the actual numbers of bugs. However, the error rates calculated from the individual data sets varied considerably for all except the original model. The analysis of the linear regression coefficients indicated that the coefficient of variability for our original model was 12% while for the other models investigated, it ranged from 39% to 182%. Thus, out of all of these models, only the original one relating V to bugs has an error rate which is reasonably constant across several data sets. This is not to be construed as proof that it is the best model for predicting validation bugs. It only means that given our current level of understanding and ability to measure, the model based on V is the most satisfactory of the currently available measures.

Table 5: Linear regression coefficients and mean statistics obtained in analysis of various models and how they relate to validation bugs.

	V	Total Stmts.	N	E	$\eta \log_2 \eta$
Akiyama	3277	167.6	335.2	$82.8 * 10^5$	372.5
Lip & Tha	2982	32.8	246.4	$264.0 * 10^5$	264.8
Shooman	2422	88.9	237.8	$8.49 * 10^5$	262.4
Bel & Sul	3000	3900.	520.	$1.06 * 10^5$	635.9
$\bar{X}$	2920	1047.	334.9	$89.1 * 10^5$	383.9
s	358	1903.	131.1	$122.3 * 10^5$	175.7
CV	12.3%	182%	39.1%	137%	45.8%

	$(\eta \log_2 \eta)/L$	$\eta \log_2 \eta'$	$(\eta \log_2 \eta')/L$	log(E)/L
Akiyama	$9.10 * 10^5$	272.6	$.676 * 10^5$	2897
Lip & Tha	$21.9 * 10^5$	198.7	$2.89 * 10^5$	809
Shooman	$41.6 * 10^5$	193.1	$.680 * 10^5$	197
Bel & Sul	$.224 * 10^5$	458.7	$.162 * 10^5$	1105
$\bar{X}$	$18.2 * 10^5$	280.0	$1.10 * 10^5$	1252
s	$18.0 * 10^5$	124.0	$1.22 * 10^5$	1160
CV	98.9%	44.2%	100%	92.6%

Table 6: Correlations of various metrics with the actual number of validation bugs found in Lipow and Thayer's and Akiyama's data.

	V	Total Stmts.	N	E	$\eta \log_2 \eta$
Akiyama	.951	.951	.951	.977	.949
Lip & Tha	.962	.957	.957	.935	.957

	$(\eta \log_2 \eta)/L$	$\eta \log_2 \eta'$	$(\eta \log_2 \eta')/L$	log(E)/L
Akiyama	.983	.949	.942	.900
Lip & Tha	.940	.958	.964	.972

USES OF THE MODEL

An expected-bug predicting model should be found quite useful during the development of software products. A primary advantage of the model that has been presented here is that it can be applied very early in the developmental effort. Because it is based on software science measures, as soon as the implementation language has been selected and the parameters to a procedure have been determined, estimates for V, and therefore $\hat{B}_V$ and $\hat{B}_T$ can be obtained. These estimates can then be revised, if necessary, after implementation when V can be measured directly.

Since predictions can be made with or without an implementation, the types of applications are varied. Those which will be discussed here briefly include estimating debugging times, computer usage and reliability and also comparing programming styles and languages. It should be emphasized at this time that the model presented here cannot be expected to always work on individual programs. Rather, like most management tools, it provides estimates for the average case. Because of this, its usefulness to management increases as the number of projects to which the model is applied increases.

Timing Estimates—One of the most obvious applications of an expected-bug predicting model is in producing better project schedules. Knowing an estimate of the number of bugs that need to be found and corrected during the validation phase of the project should encourage budgeting a more realistic amount of time for that phase.

The estimate can also be used to determine if the project is currently on schedule. Shooman and Bolsky found that, although there appeared to be hard and easy bugs to find and correct, there was no apparent correlation between the difficulty of a bug and at what time of the validation phase it was discovered [ShB 75]. If this is indeed the case, then on the average the ratio of the number of validation bugs found to $\hat{B}_V$ should be the same as the portion of time in the validation phase that has elapsed. If it is lower, the system may be behind schedule.

It may also be possible to estimate the average amount of time needed to find and correct a bug. One possible model is based on the assumption that the expected average amount of effort required to find a bug is proportional to the total effort required to understand the program divided by the expected number of bugs. That is, if there are 10 bugs expected, one would have to understand to some degree 1/10 of the program on the average for each bug found. Therefore, the time to find one bug during the validation phase of a project, T_V^1, might be approximated using (A.7) by

$$T_V^1 = K*T/\hat{B}_V$$

where $K<1$ is an estimate of the portion of complete understanding of the program the programmer needs to find the bug. The reason for introducing this factor K is that a report by Gould and Drowngowski [GoD 72] suggests that programmers are able to find and correct bugs without fully understanding the program.

Substituting for T and $\hat{B}_V$, one gets

$$T_V{}^1 = K * \frac{E/S}{V/3000}$$

$$= K * \frac{V/(L*S)}{V/3000}$$

$$= K * \frac{3000}{L*S} . \qquad (4)$$

In order to use this model, a value for K is needed. Although there is no experimentally verified value available, a rough estimate can easily be obtained based on the known characteristics of K. An upper bound for the total amount of time to find and correct all the delivered bugs would be the total validation time. This can be approximated by 40% of the total implementation time.[4] Thus,

[4]Data presented by Barry Boehm indicates that the amount of time needed for check-out and testing is 45% - 50% [Boe 73]. Wolverton's data indicates that this percentage is closer to 35% [Wol 74]. These figures do not seem to dispute the 40-20-40 rule of thumb which states that analysis and design account for 40%, coding and debugging is 20%, and test and integration is 40%.

K is less than or equal to 0.40. Since validation time includes preparing test data and verifying the correct runs, K must be somewhat less than 0.40 but, of course, significantly greater than 0. Arbitrarily using 1/4 for K, therefore, should give reasonable approximations. As an example, estimates for $T_V{}^1$ for Shooman and Bolsky's project can be made.

Using (A.4) and (A.5), one finds that

$$L = (\lambda/V)^{1/2}.$$

Recalling that for Shooman and Bolsky's sample, $V \simeq 109{,}000$ and that for assembly language, $\lambda \simeq .88$ [Hal 77], one obtains

$$L = (.88/109000)^{1/2} = .00283 \ .$$

Substituting into (4) and converting from seconds to hours,

$$T_V{}^1 = \frac{1}{4} * \frac{3000}{(.00283)(18*60*60)}$$
$$= 4.09 \text{ hours} \ .$$

This compares with the 4.44 hours given as the sum of the average time to find and to correct a bug.

<u>Computer Usage Estimates</u>—The expected-bug predicting model might also be used to estimate the amount of computer time needed to validate a system. One might assume that the total number of computer runs needed to validate a system is twice the number of bugs; that is, for each bug one run is

needed to determine that there is a bug and another one to show that the bug has been corrected. This assumption is confirmed by data from Shooman and Bolsky [ShB 75]. They found that on the average bug detection required .61 runs and bug correction required 1.35 runs, or approximately two runs altogether to detect and correct each bug.

The number of runs, R_V, needed during the validation of a project might then be estimated as

$$R_V = 2*\hat{B}_V \ . \qquad (5)$$

Given (5) and the average amount of computer time that one run takes, an estimate for the total amount of computer time needed for validation can be made.

Another possibility is that, by assuming again that validation is 40% of the total implementation time, an estimate for the average number of runs needed per day during testing and integration can be obtained. That is,

$$R_V/\text{day} = 2*\hat{B}_V/(.4*T) \ .$$

Simplifying, one gets

$$R_V/\text{day} = \frac{2*V/3000}{.40*E/(S*60*60*8)}$$

$$= 48*S*L$$

It is surprising to find that the average number of runs needed per day during validation is solely a function of the implementation level of the project and the speed of the programmer's brain. This is by no means counterintuitve, however, since the higher the level of an implementation, the faster bug detection and correction should be.

Although we have no data with which to actually test this estimate for runs needed per man-day, we can use the values of L from the previous section and check if the results are at least plausible. Substituting L=.00283 for Shooman and Bolsky's data, we get

$$R_V/\text{day} = 48*18*.00283 = 2.45 .$$

This is indeed a reasonable value.

This average number of runs per day is probably not a very important number for the programmer concerned with a single project. However, for the management of a large software development center, these numbers could be quite useful. In a large enough system, a steady-state equilibrium should exist, which would mean that the the sum over all the projects of the average number of runs needed per day might give a good approximation to the average daily work-load. This could be especially useful when expansion, either in terms of software development or hardware, is contemplated.

<u>Reliability Predictions</u>—Given the exact number of errors to be found during the debugging of a system, one could make very accurate statements about its reliability. If all the bugs had been found, the system would be considered quite reliable. If some bugs remain, the exact number of bugs still in the system would be known. Obviously no technique currently available, including the model presented here, can determine the exact number of bugs in a system. Knowing a good approximate number, however, can lead to improved reliability estimates. If, for instance, predictions indicated that the expected number of bugs was much higher than the number that had been found, one ought to be cautious about declaring the product reliable and making delivery without insuring that it had indeed been thoroughly tested.

The estimates could also be used in improved reliability models. Several reliability estimating models have been proposed. Examples of reliability models are presented in [Mus 75, Mus 77, Sho 73, Sho 76, Suk 76, Sch 75]. In general, a reliability model is based on the debugging history of the project and predicts the mean time to the next failure. Most models require an estimate of the initial number of bugs. This estimate, however, is not very critical since it is revised using a maximum likelihood estimate based on the debugging history. In other words, the entire prediction relies almost entirely on the debugging history of the project. Very little of the original characteristics of the program, other than possibly the length, are taken into account. Perhaps with the improved estimates for the number of errors to be found, reliability estimating models could be devised that used more information about the project and therefore were more reliable themselves.

Comparing Programming Styles and Languages—An interesting application of the models presented here is in comparing programming languages and programming styles. Recent work by Gordon indicated that E can be used to measure the clarity of programs [Gor 77, Gor 78]. He found that, in general, if two programs are implemented to solve the same problem, the one with the lower value of E was considered by experts to be the easier to understand. Minimizing E would be especially important when maintenance is to be the longest phase of a project.

On the other hand, one might be more interested in correctness than in understandability. The models presented here might be used to compare the expected number of bugs using varying programming styles, or varying programming languages, to implement the algorithm.

Intuitively, one would expect that error-proneness would generally decrease with increasing clarity. Decreasing V to reduce the error-proneness of a procedure, does not guarantee a similar decrease in E and therefore an increase in clarity. Examining the data presented in [Gor 78] shows, however, that in almost all of the 46 comparisons of programs presented there, V and E behave in the same manner. In the less than 10% of the cases where a decrease in E was not accompanied by a decrease in V, V did not change significantly from the original measures. In other words, at no time was an increase in clarity, as measured by E, found accompanied by an increase in error-proneness.

CONCLUSION

We have presented a model based on software science metrics to predict the number of bugs that will be found during the validation phase of a software project. The model has been tested on the data available in the literature. This included projects written both in assembly language and higher level languages to solve a wide range of problems. This is the only model that we are aware of that fits these diverse data sets.

Several extensions to the model were also presented which increase its usefulness. These include estimating the average time to find and correct a bug during validation, as well as the average number of computer runs needed per man-day during this phase of development. The preliminary results obtained from these extensions, although not always as initially expected, are not counterintuitive, and therefore, do not invalidate the hypothesis.

ACKNOWLEDGEMENTS

I am grateful to my major professor, Dr. M. H. Halstead, for his insight and guidance during the duration of this research. I am also grateful to the other members of my committee, Drs. D. E. Denning, S. D. Conte, and V. Y. Shen, for their encouragement and helpful suggestions.

REFERENCES

[Aki 71] Akiyama, F., "An Example of Software System Debugging," Proceedings of IFIP Congress, 1971.

[Boe 73] Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973, Vol. 19, No. 5, pp. 48-59.

[BeS 74] Bell, D. E. and J. E. Sullivan, "Further Investigations into the Complexity of Software," MITRE MTR-2874, Vol. II, June 30, 1974.

[Fit 78] Fitzsimmons, Ann Bowman, "Relating the Presence of Software Errors to the Theory of Software Science," presented to The 11th Hawaii International Conference of Systems Sciences, January 1978.

[FuH 76] Funami, Yasao, and M. H. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data," Proceedings Microwave Research Institute XXIV International Symposium: Computer Software Engineering, (Jerome Fox, Ed.), Polytechnic Press, (Halsted Press, distributor), New York, 1976.

[GoH 75] Gordon, R. D. and M. H. Halstead, "An Experiment Comparing FORTRAN Programming Time With the Software Physics Hypothesis," Proceedings of the NCC, 1976, pp. 935-937.

[Gor 77] Gordon, Ronald D., "A Measurement of Mental Effort Related to Program Clarity," Ph.D Thesis, Purdue University, August 1977.

[Gor 78] Gordon, Ronald D., "Measuring Improvements in Program Clarity," CSD-TR 265, Purdue University, May 1978.

[Gou 75] Gould, John D., "Some Psychological Evidence on How People Debug Computer Programs," International Journal of Man-Machine Studies, 7, 1975, pp. 151-182.

[Hal 75] Halstead, M. H. , "Towards a Theoretical Basis for Estimating Programming Effort," Proceedings of ACM 75, October 20-22, 1975, Minneapolis, Minnesota, pp. 222-224.

[Hal 77] Halstead, M. H. , Elements of Software Science, Elsevier, New York, 1977.

[Han 78] Hansen, Wilfred J., "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)," ACM SIGPLAN Notices, Vol. 13, No. 3, March 1978, pp.29-33.

[Klo 77] Klobert, R. K., "Calculation of Error Proneness of Computer Programs," Proceedings of AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, Los Angeles, CA., October 31-November 2, 1977, pp. 422-426.

[LiT 77] Lipow, M., and T.A. Thayer, "Prediction of Software Failures," Proceedings of 1977 Annual Reliability and Maintainability Symposium, January 18-20, 1977.

[LoB 76] Love, L. T. and A. B. Bowman, "An Independent Test of the Theory of Software Physics," ACM SIGPLAN Notices, Vol. 11, No. 11, November 1976, pp.42-49.

[McC 76] McCabe, T. J. , "A Complexity Measure," IEEE Transanctions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp. 308-320.

[Mil 56] Miller, G. A., "The Magical Number Seven, Plus or Minus Two," Psychological Review, 1956, pp. 311-329.

[MoB 77] Motley, R. W. and W. D. Brooks, "Statistical Prediction of Programming Errors," Final Technical Report, RADC-TR-77-175, May 1977.

[Mus 75] Musa, J. D., "A Theory of Software Reliability and Its Application," IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, September 1975, pp. 312-327.

[Mus 77] Musa, J. D., "Measuring Software Reliability," Proceedings of TIMS-ORSA Joint National Meeting, San Francisco, May 9-11, 1977.

[Mye 77] Myers, Glenford J., "An Extension to the Cyclomatic Measure of Program Complexity", ACM SIGPLAN Notices, Vol. 12, No. 10, October 1977, pp. 61-64.

[Ott 78] Ottenstein, Linda M., "Predicting Parameters of the Software Validation Effort," Ph.D. Thesis, Purdue University, August 1978.

[Sch 75] Schneidewind, N. F., "Analysis of Error Processes in Computer Software," Proceedings of 1975 International Conference on Reliable Software, April 21-23, 1975, Los Angeles, CA, reprinted in ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 337-346.

[ScH 77a] Schneidewind, N. F. and H. M. Hoffman, "Software Structure and Error Properties: Models vs. Real Programs," Proceedings of TIMS-ORSA Joint National Meeting, San Francisco, May 9-11, 1977.

[ScH 77b] Schneidewind, N. F. and H. M. Hoffman, "An Experiment in Software Error Data Collection and Analysis," Proceedings of the Sixth Texas Conference on Computing Systems, November 14-15, 1977.

[ShB 75] Shooman, M. L., and M. I. Bolsky, "Types, Distributions and Test and Correction Times for Programming Errors," Proceedings of 1975 International Conference on Reliable Software, April 21-23, 1975, Los Angeles, CA., reprinted in ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 347-357.

[Sho 73] Shooman, M., "Operational Testing and Software Reliability Estimation during Program Development," Proceedings IEEE Symposium on Computer Software Reliability, New York, April 30-May 2, 1973, pp. 51-57.

[Sho 76] Shooman, M. L., "Structured Models for Software Reliability Prediction," Proceedings of 1976 IEEE Second International Conference on Software Engineering, October 13-15, 1976, San Francisco, pp. 268-280.

[Sim 74] Simon, Herbert A., "How Big is a Chunk?," Science, February 1974, Vol. 183 (4124), pp. 482-488.

[Suk 76] Sukert, Captain Alan N., "A Software Reliability Modeling Study," RADC-TR-76-247, August 1976.

[Sul 73] Sullivan, J. E., "Measuring the Complexity of Computer Software," MITRE MTR-2648, Vol. V, June 25, 1973.

[Tha 75] Thayer, T. A. , "Understanding Software through Empirical Reliability Analysis," AFIPS Conference Proceedings, Vol. 44, 1975, pp. 335-341.

[Tha 76] Thayer, T. A., et.al., "Software Reliability Study," Final Technical Report, RADC-TR-76-238, August 1976.

[Wol 74] Wolverton, Ray W., "The Cost of Developing Large-Scale Software," IEEE Transactions on Computers, Vol. C-28, No. 6, June 1974, pp. 615-636.

[Zel 78] Zelkowitz, Marvin V., "Perspecttives on Software Engineering," ACM Computing Surveys, Vol. 10, No. 2, June 1978.

APPENDIX A

The following tested hypotheses from software science are used in this paper [Hal 77]:

Estimated program length, $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$ (A.1)

Program volume, $V = N \log_2 \eta$ (A.2)

Potential, or
Minimum volume, $V^* = (\eta_1^* + \eta_2^*) \log_2 (\eta_1^* + \eta^*)$ (A.3)

Program level, $L = V^*/V$ (maximum value of 1) (A.4)

Estimated program level, $\hat{L} = (\eta_1^*/\eta_1)(\eta_2/N_2)$ (A.5)

Language level, $\lambda = LV^*$ (A.6)

Number of mental discriminations needed to implement or understand an algorithm, $E = V/L$ (A.7)

Time to implement an algorithm, or
To fully understand an implementation, $\hat{T} = E/S$ (A.8)

where

η_1 = number of unique operators used in a program

η_2 = number of unique operands used in a program

η_1^* = minimum number of unique operators needed to express the algorithm in a "Potential Language"

= 2

η_2^* = minimum number of conceptually unique operands needed to express the algorithm in a "Potential Language"

= the number of input—output parameters

$\eta = \eta_1 + \eta_2$

N_1 = total number of operator usages

N_2 = total number of operand usages

$N = N_1 + N_2$ = actual length

S = the Stroud number $\simeq$ 18 elementary mental discriminations per second for the average programmer.

When the software science metrics are not available, estimates can be obtained from the number of executable statements in the implementation, P. To estimate N for an assembly language

$$N \simeq 8/3 * P \tag{A.9}$$

is used and for Fortran (and similar higher level languages)

$$N \simeq 7.5 * P \tag{A.10}$$

is used. Also, when necessary, it is assumed that $\eta_1 \simeq \eta_2$,[1] which reduces (A.1) to

$$\hat{N} = \eta \log_2(\eta/2) \ . \tag{A.11}$$

[1]The maximum error would occur if either $\eta_1=0$ or $\eta_2=0$. In this case, $\hat{N} = \eta \log_2 \eta$. Assuming $\eta_1 \simeq \eta_2$, the estimate becomes $\hat{N} = \eta \log_2 \eta - \eta$ giving a relative error of $1/\log_2 \eta$. Since the difference between η_1 and η_2 can not be this large, the relative error introduced by this assumption is always less than $1/\log_2 \eta$.