Computer Science Technical Reports

Department of Computer Science

1981

Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support

V. Y. Shen

S. D. Conte

Herbert E. Dunsmore
Purdue University, dunsmore@cs.purdue.edu

Report Number:
81-376

Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support

V. Y. Shen
S. D. Conte
H. E. Dunsmore

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

CSD-TR-376

ABSTRACT

The theory of Software Science was developed by the late Prof. M. H. Halstead of Purdue University during the early 1970's. It was first presented in unified form in the monograph *Elements of Software Science* published by Elsevier in 1977. Since it claimed to apply the methods of science to the very complex and important problem of software production, and since experimental evidence supplied by Halstead and others seemed to support the theory, it drew widespread attention from the computer science community.

Some researchers have raised serious questions about the underlying theory of Software Science. At the same time, experimental evidence supporting some of the metrics continues to mount. This paper is a critique of the theory as presented by Halstead and a review of experimental results concerning Software Science metrics published since 1977.

Keywords and Phrases: Software science, software management, software metrics, software engineering, software measurement, software complexity.

CR Categories: 4.0, 4.6

1. Introduction

The ever-increasing cost of program development has made the measurement of software complexity more important than it has ever been before. The critical role that software metrics can play in analyzing and evaluating software is emphasized in the recent book *Software Metrics: An Analysis and Evaluation* [Perl81] produced by a study panel commissioned by the Office of Naval Research. This book and numerous studies reported in the literature indicate the inadequacy of simple measures such as lines of code in predicting programming effort. It is generally agreed that there are a multitude of factors that affect programmer productivity. These include the type of program being developed, the interface complexity among modules in the program, the experience of the programmers involved, the computer environment, etc. (cf. [Moha81]). However, it is not evident that the inclusion of a great many factors in a measure, either intuitively or by using regression methods, leads to a *useful* estimator of total programming effort [Farr65, Wolv74, Jame77, Basi79, Moha81]. In our opinion what is needed is a model of the programming process based upon a manageable number of major factors that affect programming. This will, we believe, lead to reasonable estimators that can be useful to software project managers.

Software Science, as presented in [Hals77], was purported to be such a model. It was an attempt to analyze the complex problem of software production using established scientific methods. It drew the immediate attention of many researchers. Papers supporting the theory were published in numerous journals and conference proceedings (see, for example, [Fitz78] and a collection of papers in *IEEE Transactions on Software Engineering*, March 1979). As interest in Software Science mounted, some evidence was reported which supported some of the metrics while other results raised serious questions about them.

This paper is a critical review of the current state of Software Science both as a theory and as a practical tool for software management. We focus especially on research that has appeared since the publication of Halstead's book in 1977, and on studies done by the Software Metrics Research Group at Purdue.

2. The Theory of Software Science

We begin with a brief summary of the theory of Software Science as presented by Halstead in his 1977 monograph *Elements of Software Science* [Hals77]. A computer program is considered in Software Science to be a series of tokens which can be classified as either "operators" or "operands"*. All Software Science measures are functions of the counts of these tokens. The basic metrics are defined as:

$$\eta_1 = number\ of\ unique\ operators \tag{1}$$

$$\eta_2 = number\ of\ unique\ operands \tag{2}$$

$$N_1 = total\ occurrences\ of\ operators \tag{3}$$

$$N_2 = total\ occurrences\ of\ operands \tag{4}$$

Generally, any symbol or keyword in a program that specifies an algorithmic action is considered an operator, and a symbol used to represent data is considered an operand. Most punctuation marks are also considered as operators. The Length of a program is defined as

$$N = N_1 + N_2 \tag{5}$$

and the Vocabulary of a program is defined as

$$\eta = \eta_1 + \eta_2 \tag{6}$$

Additional metrics are defined using these basic terms. Of interest is another measure for the size of the program, called the Volume:

*This is based on the fact that all programs can be reduced into a sequence of machine language instructions each of which contains an operator and a number of operand addresses.

$$V = N \times \log_2 \eta \qquad (7)$$

The unit of measurement of Volume is the common unit for size -- "bits". It is the actual size in a computer if a uniform binary encoding for the Vocabulary is used. Volume may also be interpreted as the number of mental comparisons needed to write a program of Length N, assuming a binary search method is used to select a member of the Vocabulary of size η. Since an algorithm may be implemented by many different but equivalent programs, a program that is minimal in size is said to have the Potential Volume V^*. Any given program with Volume V is considered to be implemented at the Program Level L, which is defined by

$$L = V^* / V \qquad (8)$$

The value of L ranges between zero and one, with $L=1$ representing a program written at the highest possible Level (i.e., with minimum size). The inverse of the Program Level is termed the Difficulty. That is,

$$D = 1 / L \qquad (9)$$

As the Volume of an implementation of a program increases, the Program Level decreases and the Difficulty increases. Thus, programming practices such as the redundant usage of operands, or the failure to use higher level control constructs will tend to increase the Volume as well as the Difficulty.

The effort required to implement a computer program increases as the size of the program increases. It also takes more effort to implement a program at a lower Level (higher Difficulty) when compared with another equivalent program at a higher Level (lower Difficulty). Thus the Effort in Software Science is defined as

$$E = V/L = D \times V \tag{10}$$

The unit of measurement of E is "elementary mental discriminations".

A sound theory should have not only an intuitive set of definitions, but should also contain an intuitive model for which a useful set of hypotheses may be derived and validated. The model, although never explicitly stated by Halstead, is that most programs are produced by concentrating programmers through a process of mental manipulation of the unique operators and operands. The basic assumption that leads to the hypotheses presented in the following subsections is an implicit limit on the mental capacity of a programmer.

2.1. Length equation

The first hypothesis of Software Science is that the Length of a well-structured program is a function only of the number of unique operators and operands. It is called the "Length equation" where $\hat{N}$ is the predicted Length of the program*.

$$\hat{N} = \eta_1 \times \log_2 \eta_1 + \eta_2 \times \log_2 \eta_2 \tag{11}$$

The Length equation, like many other software metrics, may not be a precise equality for a specific program yet may be considered valid in a statistical sense. Such relationships are common in experimental sciences dealing with human subjects. Since programming is a very demanding *human* activity, it certainly falls into the realm of activities that must be approached with the idea of explaining "typical" performance while perhaps failing to achieve precision in specific instances. It is also known that certain poor programming practices,

*The Length equation estimates the total length, not the individual estimates of N_1 and N_2, as some have believed [Moha79].

referred to as the "introduction of impurities" in Software Science, can make the Length

equation a very poor predictor of N.

2.2. Potential Volume

As we have discussed earlier, a program that implements an algorithm in its most succinct form has the Potential Volume V^*. If the desired operation on data is already defined in the programming language or its subroutine library as a "built-in" procedure, the Potential Volume is achieved by specifying the name of the procedure and by giving a list of input/output parameters. The Vocabulary of this program consists of two operators and $\eta_2{}^*$ operands. One operator is the name of the procedure, since it defines some action; the other operator is a grouping symbol needed to separate the list of parameters from the procedure name. Thus

$$V^*=(2+\eta_2{}^*)\times\log_2(2+\eta_2{}^*) \qquad (12)$$

where $\eta_2{}^*$ is the number of input/output parameters to the procedure. This formula, although useful for many programs, is not applicable universally since there are programs that do not have an explicit list of input/output parameters. An example is a compiler whose output consists of several files and messages to the operating system. Moreover, as Halstead himself observed, the concept of $\eta_2{}^*$ may have to be extended to include certain "information-packed" constants and, perhaps, other implicit variables.

2.3. Program Level (Difficulty) estimator

The Level of a particular implementation depends on the ratio of the Potential Volume and the actual Volume (equation (8)). Since the Potential Volume is usually not available, an alternate formula which *estimates* the Level is defined as

$$\hat{L}=\frac{1}{\hat{D}}=\frac{2}{\eta_1}\times\frac{\eta_2}{N_2} \qquad (13)$$

An intuitive argument for this formula is that programming difficulty increases if additional operators are introduced ($\eta_1/2$ increases) and if an operand is used repetitively (N_2/η_2 increases). Every parameter in equation (13) may be obtained by counting the operators and operands in a computer program. The Potential Volume $V*$ may then be deduced using equation (8) with L equal to $\hat{L}$. This formula can also be used with equation (10) to determine the Software Science Effort estimate for a given program.

2.4. Programming time

A major claim for Software Science is its ability to relate the basic metrics to actual implementation time. A psychologist, John Stroud, suggested that the mind is capable of making a limited number of elementary discriminations per second [Stro67]. Stroud claimed that this number S (now called the "Stroud number") ranges between 5 and 20. Since Effort$E$ has as its unit of measure the "number of elementary mental discriminations", the Programming Time T of a program in seconds is simply

$$T=E/S. \qquad (14)$$

S is normally set to 18 since this seemed to give the best results in Halstead's experiments comparing the predicted times using equation (14) with observed programming times. The Software Science claim is that this formula can be used to estimate programming time when a given problem is solved by a single, proficient, concentrating programmer writing a single-module program [Hals77].

2.5. The Language Level

The proliferation of programming languages suggests the need for a metric that expresses the power of a language. Halstead hypothesized that if the programming language is kept fixed, then as V^* increases, L decreases in such a way that the product $L \times V^*$ remains constant. Thus this product, called the Language Level λ, can be used to characterize a programming language. That is,

$$\lambda = L \times V^* = L^2 V \qquad (15)$$

Analyzing a number of programs written in different languages using equation (13) for L, Language Levels were determined to be 1.53 for PL/1, 1.21 for Algol, 1.14 for Fortran, and 0.88 for CDC assembly language. These average values follow most programmers' intuitive rankings for these languages, but they all have large variances. Such fluctuations in a hypothesized fixed value are not entirely unexpected since the Language Level depends not only on the language itself, but also on the nature of the problem being programmed as well as on the proficiency and style of the programmer. Equation (15) can be useful in comparing programming languages if the same set of problems is programmed in different languages by the same programmer. Algebraic manipulation of equations (10) and (15) yields another formula for E:

$$E = V^{*3} / \lambda^2 \qquad (16)$$

This formula can be used for Effort *prediction* if the Potential Volume V^* and the Language Level are known. Thus, for a given problem, the Effort (and the resulting implementation time) varies according to the squared inverse of the Language Level.

3. Criticisms of Software Science

The publications of [Hals77] and [Fitz78] generated significant interest in the research community. Criticism of Software Science first appeared as letters and later as papers [see, for example, Mora78, Feni79, Male80, Lass81]. Some of the objections concerned the theory; others reported empirical results that failed to support Software Science formulas. These criticisms are summarized in the following subsections.

3.1. Defining and counting operators and operands

The original theory of Software Science was intended for analyzing *algorithms* (not programs). Most supporting data was drawn from algorithms written in Algol and Fortran. It was not difficult to accept that "an algorithm consists of operators and operands, and of nothing else" ([Hals77], p.6). Nor did it seem very difficult to classify the tokens used in Algol and Fortran programs into operators and operands. Variable declaration sections and other non-executable statements were excluded from the counts in computer programs.

However, in other languages it is sometimes impossible to determine whether a token is to be interpreted as an operator or operand [Lass81]. The meaning may depend on the use of the token at execution time, or it may depend on the information given in the declaration section. A function reference may serve as both an operator and an operand at the same time. The inconsistent results from counting the same program on different occasions in

[Hals77] further illustrate the difficulty in classifying tokens [Male80]. Since the variable declaration section in some languages (e.g., Data Division in Cobol) takes a significant portion of the programming effort, it does not appear reasonable to ignore it [Elsh78, Fits79, Shen81]. Some (cf., [Male80]) have suggested that operators should be divided into two groups - control operators and process operators - because of their significantly different impact.

Another objection raised by [Lass81] questions the count of GO TO's in Fortran. Halstead proposed that each "GO TO label" be counted as a unique operator for each unique label. On the other hand n IF statements are considered to be n occurrences of one unique IF operator. Ambiguities, both theoretical and practical, in the classification and treatment of some operators and operands may lead to substantially different values of some Software Science metrics. These ambiguities are normally resolved by the designer of automatic counting tools or analyzers using some convenient strategy (see, for example, [Fits79] for a strategy on counting IBM's assembly language and PL/S).

3.2. The derivation of formulas

Rigorous algebraic derivations are given in [Hals77] for all the Software Science formulas. However, several implied assumptions are made for which there seem to be no theoretical justifications. For example, in deriving the Length equation in chapter 2 of [Hals77], Halstead divides a program of Length N into N/η substrings of Length η. Assuming there are no duplications of these substrings and assuming that operators and operands alternate, he concludes that N must satisfy the inequality

$$N < {\eta_1}^{\eta_1} \times {\eta_2}^{\eta_2} \qquad (17)$$

No reason is given for dividing a program of Length N into N/η substrings of

Length η. The assumption that operators and operands alternate seems reasonable, but this would imply that for all programs $N_1 \approx N_2$ which has not been observed in general. Furthermore, the derivation of (17) assumes that an operator or operand (which one is not made clear in the text) always comes first. Allowing either operators or operands to appear first would double the upper limit in (17). Thus, the Length equation cannot be justified on theoretical grounds from these considerations, although its use and value as an experimental metric may still be valid.

Another example is the derivation of the relationship between operators and operands. After defining V** as the Boundary Volume

$$V^{**}=(2+\eta_2^{*}\times\log_2\eta_2^{*})\times\log_2(2+\eta_2^{*}) \qquad (18)$$

at the end of chapter 3 of [Hals77], Halstead sets

$$\frac{d\eta}{d\eta_1}=\frac{V^{**}}{V^{*}} \qquad (19)$$

in chapter 4. No justification for this formula is given in the book. Furthermore, since η is a discrete variable, treating it as a continuous variable in order to differentiate it is highly questionable.

In deriving equation (14) for programming time, Halstead relies on the work of [Stro67] to convert Effort, given in "elementary mental discriminations", into time in seconds. Among psychologists there is no general acceptance of Stroud's hypothesis that the mind is capable of making a constant number (S) of mental discriminations per second. As a theoretical concept, the Time equation must therefore be considered suspect.

The presence of these and other unverifiable assumptions in the derivation of formulas in [Hals77] casts serious doubt on the underlying theoretical foundations of Software Science.

3.3. Validity of experimental data

Even though the theoretical foundations of Software Science are weak, it is still possible that the Software Science metrics may be useful. In order to decide if the formulas are acceptable approximations to reality, it is necessary to examine the empirical work that has been conducted.

A first observation is that the validating data reported in [Hals77] and in some early papers that followed were not presented in the classical form of hypothesis testing [Zweb79]. Indeed, Halstead frequently and incorrectly inferred that because two sets of numbers were highly correlated, one can be used as a substitute for the other.

Secondly, the experiments conducted by Halstead and others to validate his claims have been criticized on the following grounds:

> i. The sample sizes in most cases were too small. Good experimental technique requires as many data points as possible (at least on the order of 20 to 30) before making an inference from a sample. Many of Halstead's conclusions were based on sample sizes less than 10.
>
> ii. The programs involved were small (especially in the programming time experiments). All except one were single modules of less than 50 statements. It is probably not possible to generalize results with such programs to large, multi-module industrial programs.
>
> iii. Many of the experiments, especially those concerning programming time, involved only single subjects. Unless the single subjects were perfectly "typical", the results may not generalize to other programmers.

iv. The subjects (even when there were several) were generally college students. There is a real concern that results based on this type of subject may not generalize to professional programmers.

These criticisms of the experimental results contained in [Hals77] are certainly valid. However, we recognize that this pioneering work was done in a university environment, which made it difficult to conduct experiments involving large software projects. Additional research since [Hals77] has provided more data of better quality. The following sections contain discussions of each of the hypotheses (or claims) of Software Science in light of recent research results.

4. The Length Equation

Software Science claims that the length of a program is a function of the unique operators and operands. This is the hypothesis that has received the most attention, since it can be easily tested. Such extensive testing leads to the results outlined in the following subsections.

4.1. The effects of counting rules on the Length equation

Although it is easy to construct a pathological program to make $\widehat{N}$ a poor predictor of N, there is overwhelming evidence using existing analyzers to suggest the validity of the Length equation in several languages. A misclassification of any token has virtually no effect on the final estimate since

$$\widehat{N}=\eta_1\times\log_2\eta_1+\eta_2\times\log_2\eta_2\approx\eta\times\log_2\frac{\eta}{2} \qquad (20)$$

regardless of how the Vocabulary of size η is divided into operators and operands.

The Length equation indicates that the total length of a program is a function of the counts of its basic tokens. As the number of unique operators and operands increases, the Length estimator ($\hat{N}$) increases as well. It will significantly over-estimate the Length if each unique token is used only once or twice in the body of the program. This, in fact, is somewhat characteristic of many Cobol programs. The Length equation yields much better estimates if we include the counts for the declarations [Shen81]. The same observation has been made in constructing automatic counting tools or analyzers for PL/1 and PL/S [Elsh78, Fits79]. For consistency, we suggest that all Software Science analyzers should count operators and operands in declaration sections as well as procedure sections. Software Science theory also excluded input/output statements from the counts. However, since a significant portion of many programs deal with input and output, we feel that these statements should be considered by Software Science analyzers as well.

4.2. The derivation of the Length equation

The problems in the derivation were discussed in the previous section. The derivation as given in [Hals77] was actually established *after* the relationship (i.e., equation (11)) was proposed and tested. Since the Length equation has been found to be a valid and useful formula in many different environments, there should be a better way to support it theoretically than that offered by Halstead.

4.3. The prediction of program Length using unique operands

It is likely that a complicated program will use nearly all of the predefined operators or keywords in the language. Thus, for large enough programs the count for η_1 should be a constant plus the count of procedure calls, function references, and direct transfer (i.e., GO TO) statements. In languages such as Pascal and PL/S, where the use of direct transfers are discouraged, η_1 should be nearly constant for large programs. An analysis of 490 PL/S modules shows that η_1 has a mean value of 46 with a standard deviation of 18 [Fits80]. Thus, for programs written in languages in which the use of direct transfers are restricted, it may be possible to predict the eventual program length using equation (11) after the declaration section is completed, since at that time an estimate of η_2 will also be available [Chri81].

4.4. The error of the Length equation

The metric $\widehat{N}$ has proven to be an acceptable estimator of N when applied to a wide range of programs [Fits80]. In an analysis of 1637 modules the relative error between N and $\widehat{N}$ was less than 6%, although this error can be much larger for individual modules [Smit80]. The usefulness of the estimator $\widehat{N}$ *does* appear to be somewhat sensitive to the actual program Length N; i.e., $\widehat{N}$ tends to overestimate for small programs and underestimate for large ones. It appears to work best for programs in the range $2000 < N \leq 4000$ [Smit80, Shen81]. For programs of size $N > 4000$ the average relative error is -20% in the analysis of 231 modules, and for $100 < N \leq 2000$ the error is 40% from 1032 modules. Under these circumstances the relative error of the Length equation can be minimized by dividing a program into modules of reasonable size and then summing the individual estimates.

5. The Program Level / Difficulty

The Program Level (L) was intended to be a measure related to the effort in writing a program, the error-proneness of a program, and the ease of understanding a program. Although it depends to some extent on the language being used, the Program Level might vary greatly even for equivalent programs written in the same language since it is dependent on the experience and style of the programmer. Thus the validity of the concept as defined in equation (8) and the computation formula as defined in equation (13) can only be tested indirectly. We shall comment on the derivation of the Level estimator and its use as a complexity metric for error-proneness. The effects of using equation (13) in the computation of the Potential Volume, the Effort, and the Language Level will be discussed in later sections.

5.1. The derivation of the Program Level estimator

Equation (13) for $\hat{L}$ establishes the Program Level as the product of two terms. The first term, $2/\eta_1$, decreases as the number of unique operators increases. The Program Level L exhibits the same behavior; i.e., L decreases as η_1 increases. Similarly, the second factor in $\hat{L}$, namely η_2/N_2, decreases as operand redundancy increases. Since operand redundancy also increases the Volume, it is evident that L too will decrease. Halstead decided to use the product of these two factors as the Level estimator. He considered other possible combinations of these two factors, but discarded them because they did not agree as well with available data. The data presented in [Hals77] showing the validity of the Level equation depend on values for $\eta_2{}^*$, which were determined using a subjective method.

We cannot test the Level equation objectively on large sets of programs since we do not have an objective method to compute $\eta_2{}^*$. Its validity can only be inferred by applying the equation to other metrics, such as the Difficulty, Potential Volume, etc., and comparing the results with observed values. The testing problem is made more complicated since an unfavorable result may mean either a poor Level estimator formula or an improper definition of the metric being studied. On the other hand, a positive result may not be conclusive, since errors in the Level estimator may be compensated by errors in the metric definition.

5.2. *D* as a complexity metric for error-proneness

It is very difficult to define an error in a program. Certain errors are simple and may require the change of only one statement. Others are more complex and may require changes in many different places in order to eliminate the errors. There are even errors which are denoted as errors only after some specification is changed. However, it is generally believed that if a programmer is careful in the design phase, a program will be easy to understand and will contain few errors. If a program is easy to understand, it should also be easy to correct if an error is discovered. The inverse of the Level, called the Difficulty (D), is a candidate for a measure of "error-proneness". An equivalent of equation (13) is the following:

$$\hat{D}=\frac{\eta_1}{2}\times\frac{N_2}{\eta_2} \qquad (21)$$

Equation (21) is the product of two ratios. The first ratio, $\frac{\eta_1}{2}$, increases when more unique operators are used. Although programming languages in general have a fixed set of operators, it is still possible to increase this ratio arbitrarily by introducing a large number of direct transfer (i.e., GO TO) statements, if they are counted as unique operators. Advocates of structured programming generally agree that the use of direct transfers is a harmful practice [Dijk68]. Thus, this intuition of difficulty is supported as $\hat{D}$ increases when the program uses more transfer statements. The second ratio, on the other hand, is the average operand usage. It is 1 if each operand is used only once. The more times an operand is referenced in a program, the more candidates there are for changing its value, possibly requiring more mental effort to remember its current meaning at any point during programming. Thus a program with a high value of $\hat{D}$ is likely to be more difficult to construct and this may lead to more errors in the future.

There have been a number of recent studies comparing complexity metrics that are potentially related to program maintenance. A study of 197 PL/1 programs for which error data was available showed that the $\hat{D}$ metric yielded a better correlation with both the error density and the average number of errors than did cyclomatic complexity, the nesting structure of control flow, or the Software Science E metric [Feue79]. Another study using 30 program modules at IBM (for which there are records on the reported errors after release) showed that $\hat{D}$ is a good measure of relative error-proneness [Smit79]. The latter study suggested the establishment of threshold values of $\hat{D}$ which could be used by programmers as a guide for developing software products. For example, for PL/S programs, the average value of η_1 was found to be 46, and the ratio N_2/η_2 was less than 5 [Fits80]. Two threshold values for the Difficulty metric can be determined as follows:

$$\hat{D}_1 = \frac{46}{2}\times 5 = 115 \qquad (22)$$

$$\hat{D}_2 = \frac{46+18}{2}\times 5 = 160 \qquad (23)$$

(The value 18 used in equation (23) is the standard deviation of η_1). If, for a certain module, $115<\hat{D}<160$, the programmer is advised to review his code for certain poor programming practices such as the use of too many GO TO statements, too much embedded assembly language code, or unwarranted redundant usage of operands. If $\hat{D}\geq 160$, more drastic action such as a team review would be recommended. The IBM study suggests that similar threshold values may be established for other high-level languages.

6. The Invariance of the Potential Volume/Intelligence Content

Equation (8) suggests that for a given algorithm, different implementations may have different Volumes and Levels; yet the product of those two may remain constant. That is, the Potential Volume $V^*=L\times V$ is dependent only on the algorithm, not on the characteristics of a particular implementation. When $\hat{L}$ from equation (13) is used to estimate L, the product $\hat{L}V$ is called the Intelligence Content; i.e.,

$$I=\hat{L}\times V. \qquad (24)$$

The Intelligence Content I is also expected to remain constant over different implementations of the same problem since it is an estimate of V^*. In chapter 6 of [Hals77] there are some examples in which all of the I values are within 10% of the average value for six or seven implementations of the same problem. If this invariance of I could be shown to be universal, it would establish an extremely important complexity metric.

The invariance of such a quantity can be tested on programs that all solve the same problem using essentially the same algorithm. We have analyzed hundreds of student programs written in Fortran and Cobol. The range of I

values are normally far more variable. For example, in the analysis of 237 Cobol programs from four separate assignments, the *best* result we found for I is a standard deviation about 13% of the average [Shen81]. It is not unusual to find individual cases where the Intelligence Content varies almost 100% from the average. There are also four versions of twelve programs presented (for another purpose) in Table 8.1 of [Hals77]. Only two of the twelve cases have Intelligence Contents within 10% of the average [Male80]. The worst case (Algorithm 24) has one version that differs by 51% from the average. Thus, this data fails to support the claim of the invariance of I, although student programs admittedly will show greater variability than professionally-written programs. Furthermore, percentage variations on small programs tend to be more pronounced. The data, however, does not invalidate the hypothesis that V^* is invariant since it uses equation (13) for the Program Level and this equation depends on some additional assumptions. But it does suggest that if the Potential Volume as originally defined does exist, it should be computed in some way other than that proposed by Halstead.

7. The Effort Measure

An important role for software complexity measures is to predict the cost of software development. If there are two designs to solve a particular problem, then a complexity measure is very useful if it can identify the design that will take less time to program. It is even better if the complexity measure can be used to predict the actual time required to implement each design.

The software metrics area is now replete with a large number of complexity measures. These may be divided into three classes: (1) those that are dependent on the size of the program, called *extensive* measures; (2) those that are dependent on the structure of the program, called *intensive* measures; and

(3) those that are dependent on a combination of a number of factors. For most complexity measures, statistically-derived constants are required to relate the complexity measures to actual programming times. The limited number of data points and the variable quality of data points used in deriving these constants frequently restrict the application of these measures to very limited types of programs and environments. For example, a common complexity measure is the simple size measure lines of code (*LOC*). It is generally accepted that a program requiring more lines of code will take proportionally longer to implement than another program requiring fewer lines. To relate the lines-of-code measure to actual programming time, a formula of the following type can be derived using regression analysis:

$$T = a \times LOC^{b} + c \tag{25}$$

The formula is of limited usefulness, since different environments lead to different constants. There are published reports that stipulate the value of *b* from as low as .91 [Wals77] to as high as 1.83 [Boeh81].

The Software Science Effort measure depends on the number of unique operators and operands and how they are used. It can be related directly to programming time using the so-called "Stroud constant". It can also be related to the effort needed to comprehend an existing program [Gord79]. The *E* metric has the potential of being an effort predictor; i.e., the factors that it depends upon may be available before the program is constructed.

7.1. The derivation of the E metric

Implementing an algorithm of Length N can be considered to be the selection of N tokens from a Vocabulary of size η. According to Hick's law [Hick52], the selection process in humans approximates that of the binary search. Thus the definition of the program Volume $V=N \times \log_2 \eta$ (equation (7)) is directly related to the effort required to implement the algorithm, which is measured in the number of "mental comparisons". The time to make each mental comparison is not constant, however. It depends upon how difficult it is for the particular implementation; thus it is dependent on the D or L measure (equation (10)). These considerations led Halstead to hypothesize that the Effort E is the product of D and V. Since D was interpreted as the number of elementary mental discriminations per comparison, Halstead made the unit of measurement of E "elementary mental discriminations".

The conversion from mental comparisons to elementary mental discriminations has no theoretical basis. The conversion from elementary mental discriminations to time using $S=18$ is also controversial as mentioned in Section 3. Even if these assumptions are valid, one would expect a large range in time estimates since S ranges from 5 to 20 according to Stroud.

The measure E is basically an *extensive measure*. Although it includes the count of unique operators, it cannot take into account the different functions of these operators. Thus, a program using a proportionally large number of conditional instructions (implying a more complex internal structure) may not yield a higher value for E. This was considered as a weakness in using E as a measure of control flow complexity [Bake80].

7.2. Programming effort

The experiments reported in [Hals77] showing the comparison of actual programming times and estimated programming times using E involved only one subject. Another small experiment conducted later in which four subjects built eight modules also found $S = 18$ to be a reasonable factor to convert E to programming time (see Table 1, which is taken from [Wood80]).

Table 1

Sorting Experiment Results

Program Number	Actual Time (minutes)	Est. Time (minutes)
1	6	7
2	12	6
3	13	10
4	14	14
5	15	15
6	95	44
7	127	164
8	173	174

This second experiment supports E as a metric for programming effort when individual programmers construct small modules.

When the Effort measure is applied to large programs with multiple modules, it consistently overestimates programming time [Wood80]. A study of four projects with a total of 416 modules showed that the correlation of the E metric with actual time was only about .65 [Basi81]. This was not better than the correlation coefficients of other traditional measures such as lines of code.

A recent set of experiments suggests that larger modules in multi-module programs should be conceptually broken into smaller parts (called "logical modules") before applying the E measure [Wood81b]. Using $S=18$ to convert the E measure to T works best for modules which take less than two hours to produce and which are less than 50 lines of code in length. Under these circumstances, the E measure is a better effort measure than those produced by regression formulas using only lines of code or cyclomatic complexity (v(G)) [McCa76] (Table 2).

Table 2

A Comparison of Several Models

for Estimating Programming Times

Model	Pearson Corr. Coeff.		Avg. Relative Error(%)	Avg. Abs.Rel. Error(%)	Mean-Squared Error	Regression Coeff. Used
	r	r^2	RE	$\|RE\|$	MSE	
$T_{v(G)}$	.66	.43	-37	53	.59	Yes
T_{loc}	.78	.60	-22	37	.39	Yes
T_L	.83	.69	16	26	.40	No

The last row in Table 2 is the E measure converted to times based on the "logical module" concept. It is a better estimator of programming time since E is a highly nonlinear function of program length. For example, Schneider

[Schn78] showed that E as a function of N behaves like

$$E \approx a \times N^{1.83}$$

Thus, for large programs it is necessary to first modularize the program, obtain E for each module, and then sum these to obtain the total Effort estimate. Of course this then raises questions about what module size to choose as well as the role of module interconnection complexity.

Although the modules produced in the Woodfield experiments [Wood81b] were small according to industry standards, the data were nonetheless difficult and costly to collect. The lack of controlled experiments on the production of larger software is the main factor preventing more extensive tests of the E metric.

7.3. Program comprehension

A large portion of a programmer's time is spent in modifying existing programs to correct errors or to meet new specifications [see, for example, Dona80]. This activity, popularly called "maintenance", requires a thorough understanding of at least part of the existing code before modifications or additions can be made. The effort required to understand a piece of software is often non-trivial. As a result, there are those who advise reprogramming from scratch when the effort to comprehend an existing program is believed to be comparable to the effort to start anew.

The importance of writing programs which are easy to comprehend led to the publication of many books on good programming style [see, for example, Kern78]. In these books examples are normally given as two alternate means of implementing a program segment: one poorly written, the other significantly improved. In an analysis of 46 pairs of program segments written in Fortran, Cobol, Pascal, Algol, and PL/1, it was discovered that the E measure decreased when 40 of the 46 program segments were improved -- implying that E is highly correlated with good programming practice. The number of executable statements, on the other hand, decreased in only 31 of the 46 cases suggesting that size alone is not a good predictor of comprehensibility [Gord79].

When the E measure was used by the Software Management Research group at General Electric in a small experiment designed to measure software maintenance effort, it produced unimpressive results but differences that were in the expected direction [Curt79a]. Another experiment using more programmers and larger programs was conducted by the same group several months later. It showed that the E metric was better than $v(G)$ or lines of code in estimating the maintenance effort [Curt79b]. Specifically, the correlation between maintenance performance and E was .75 while for $v(G)$ and LOC the correlations were .65 and .52 respectively.

Another experiment was conducted at Purdue University by asking 48 programmers to study eight versions of the same program for a fixed amount of time [Wood81a]. The subjects then were asked to answer a twenty-question quiz designed to measure comprehension [Wood81a]. The subjects who studied the version with the lowest predicted effort (using an E-based measure) had the highest quiz scores [Wood80]. Thus, these two studies tentatively support the conclusion that a program with a lower E measure is easier to comprehend than an equivalent program with a higher E value.

7.4. Effort prediction

Most suggested effort measures depend on factors that are available only after the completion of the program; e.g., lines of code, cyclomatic number, and counts of operators and operands. Such measures are only useful for determining whether actual programming time is close to the "predicted" time. An effort measure would be more useful if the factors it depends upon were available *before* the program was completed. One approach is to try to estimate these factors at earlier milestones in the development process and then to try to predict the remaining effort. For example, a total time estimate might be made at the end of the design phase or at the first time when the program compiles correctly (called the "first clean compile"). In a study conducted at Purdue University, we analyzed the first clean-compiled version of 27 programs for which the total programming time was known. We predicted total programming time from measures based on lines of code, cyclomatic complexity, and the Software Science Effort measure. The E measure was the best at predicting total programming time using the first clean-compiled versions. Its correlation with total time was .84 compared to .77 for $v(G)$, .82 for lines of code, and .22 for number of runs [Wang81].

Software science also permits *a priori* effort estimation based on $E = V^{*3} / \lambda^2$ (equation (16)). The use of E as a predictor from this formula requires knowledge of η_2^* (the number of conceptually-unique input variables) and of the constant Language Level λ. Unfortunately this formula is of limited usefulness since η_2^* cannot always be determined precisely and any errors in η_2^* will be magnified in computing E. In addition, as will be shown in the next section, the Language Level λ as proposed by Halstead is subject to large variability.

8. The Language Level

It is an interesting hypothesis of Software Science that it should be possible to rank languages on a linear scale based on a simple count of operator and operand usage. Such a Language Level metric, if it exists, could be used in selecting a language for a new application, in testing the potential power of a proposed language, and even in predicting relative effort to produce software in different programming languages. For example, equation (16) shows that for a fixed problem (i.e., fixed V^*), the Effort measure varies inversely as the square of the Language Level. Thus, if one language has a λ twice that of another, then the E measure associated with the program in the first language would be 1/4 of that for the second language.

8.1. The derivation of the Language Level

The generic equation for the Language Level in [Hals77] was

$$\lambda = L^b V \qquad (26)$$

Regression analysis using some sample sets of programs in different languages seemed to indicate that $b \approx 2$. (Note that the formula for $\hat{L}$ (equation (13)) was used in the study). This result led Halstead to hypothesize that $\lambda = L^2 V$ would remain essentially constant for all programs written in a fixed language. Using this formula Halstead determined the Language Level for various languages using essentially the same sample sets of programs and arrived at the Language Levels shown in Table 3. Although the λ values follow most programmers' intuitive ranking of the powers of these languages, the large standard deviations relative to the mean values lend only weak support to the hypothesis that λ is essentially constant for a fixed language.

Table 3

Language Levels

Language	λ	σ
PL/1	1.53	.92
Algol	1.21	.74
Fortran	1.14	.81
CDC assembly	.88	.42

8.2. The Length dependency of λ

There have been a number of recent studies of Language Level in several languages [Smit80, Shen81] based on much larger sets of data. Table 4 comparing the Language Levels of IBM Assembly Language and PL/S is taken from [Smit80]. It is evident from Table 4 that the wide range of λ values within each language and their large standard deviations do not support the claim of Language Level constancy. This particular study also indicated that the average Language Level exhibited a strong inverse dependence on the Length of the program. If the sample programs used in the study are grouped according to their N values, the average λ's are shown in Table 5. From these results it seems that the Language Level is a strongly exponentially decreasing function of the program Length, shattering the validity of the claim for constancy. Such Length dependency was also observed in Fortran [Cont81], in Cobol [Shen81], and in the ESS programming language [Bail81].

Table 4

Language Levels for IBM Projects

Project	#Modules	Language	Avg. λ	σ
A	211	BAL	.51	.53
B	514	BAL	.90	.76
C	176	BAL	1.49	.76
J	93	BAL	.79	.94
D	63	PL/S	1.59	.87
E	82	PL/S	2.71	1.16
F	54	PL/S	4.08	1.98
G	354	PL/S	2.13	1.16
H	90	PL/S	1.47	.80

6.3. Alternate formulas for λ

Although experimental evidence has failed to confirm the hypothesis of Language Level constancy based on the formula $\lambda = L^2 V$, the cause may again lie with the use of equation (13) for the Program Level (see Section 5). Nevertheless, it may still be possible that a generalized version of this hypothesis of the form

$$\lambda = \hat{L}^{\alpha} V^{\beta} \qquad (27)$$

Table 5

Length Dependence of Language Level

Language	$N \leq 100$	$100 < N \leq 2000$	$2000 < N \leq 4000$	$N > 4000$
BAL	2.5	1.2	0.4	0.3
PL/S	4.2	2.2	1.8	1.2

might be statistically valid. Indeed an investigation currently underway [Cont81] shows that constants α and β can be found such that equation (27) leads to statistically-valid Language Level metrics.

9. Summary and Conclusion

In this paper we have presented criticisms (both ours and other researchers) of the theory of Software Science. We have also examined Software Science measures in light of recent data that has been published. We have concentrated on the basic properties and relations of Software Science (Part I of [Hals77]), since there has been little interest demonstrated in some of the more esoteric claims made in Part II (the error equation, application to hardware, etc.).

The early experiments to validate Software Science claims have been criticized on grounds of sample sizes and programs that were very small. It has also been suggested that the very base of Software Science (counting operators and operands) is shaky due to ambiguities concerning what should be counted and how. We concluded that serious deficiencies have been the failure to consider declarations and input/output statements, and (possibly) counting a "GO TO label" as a unique operator for each unique label.

Furthermore, we have shown that the Length equation (11) cannot be justified theoretically in the manner proposed by Halstead. On the other hand there is a large amount of empirical evidence to suggest its validity, although it appears to work best in the range of N between 2000 and 4000. The Intelligence Content I was claimed to be constant over different implementations of the same problem, but this does not appear to be supported empirically. Published data does seem to sustain the usefulness of D (the so-called Difficulty metric) as a measure of error-proneness.

Results also suggest that the Software Science E is a better effort measure than most others being used. The Time equation (14) is suspect in theory because it relies on the very questionable hypothesis that the mind is capable of making a constant number of elementary mental discriminations per second. However, we report some results that support its utility. Other data suggests that the Language Level λ is anything but invariant, but recent work [Cont81] may lead to a formulation for λ that is statistically constant for a language regardless of programmer or problem.

Thus, the current state of Software Science seems to be that of a still-evolving theory. There are those who question (with good reason in most cases) some of its underlying assumptions. However, there is a large body of published data that suggest that Software Science metrics may be useful. It is possible that several of the formulas (eg. (11), (12), (14), (16)) may only be first approximations of the real relationships concerning program length, potential volume, programming effort, and programming time.

To our knowledge Software Science is the only complete theory which attempts to explain the programming development process. As such it is deserving of continued investigation by researchers in spite of the many theoretical shortcomings that have been described in this paper. In practice, we conclude that the "real world" use of Software Science measures in their current state must be done very carefully. On the other hand, we believe that researchers should continue to refine these metrics (and to eliminate those that appear unsalvageable). The goal should be a set of measures that can be justified theoretically, that can be supported empirically, and that can be used with confidence by programmers and project managers.

10. Acknowledgements

The authors express their sincerest appreciation to Scott Woodfield, Steve Thebaut, Andrew Wang, Ken Dickey, and Carl Burch for their contributions to software metrics research at Purdue University. We want to thank Ken Christensen, George Fitsos, and Chuck Smith for several useful discussions concerning this topic. Software metrics research at Purdue is supported by the International Business Machines (IBM) Corporation, by the Army Institute for Research in Management Information and Computer Systems (AIRMICS), and the TRW Company.

11. References

[Bail81]

Bailey, C. T. and Dingee, W. L. A software study using Halstead metrics. *ACM Sigmetrics* 10, 1 (1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality) (March, 1981) 189-197.

[Bake80]

Baker, A. L. and Zweben, S. H. A comparison of measures of control flow complexity. *IEEE Transactions on Software Engineering* 6, 6, (November, 1980), 506-512.

[Basi79]

Basili, V. R. and Reiter, R. W. An investigation of human factors in software development. *Computer* 12, 12 (December 1979), 21-38.

[Basi81]

Basili, V. and Phillips, T. Evaluating and comparing software metrics in the Software Engineering Laboratory. *ACM Sigmetrics* 10, 1 (1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality) (March, 1981) 95-106.

[Boeh81]

Boehm, B. W. Software life cycle factors. TRW-SS-81-03, TRW Software Series.

[Chri81]

Christensen, K., Fitsos, G., and Smith, C. P. A perspective on Software Science. *IBM Systems Journal* 20, 4, (1981).

[Cont81]

Conte, S. D. The Software Science level metric. CSD TR-373, Department of Computer Sciences, Purdue University, (1981).

[Curt79a]

Curtis, B., Sheppard, S. B., Milliman, P. M., Borst, M. A., and Love, T. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering* 5, 2, (March, 1979), 96-104.

[Curt79b]

Curtis, B., Sheppard, S. B., and Milliman, P. Third time charm: stronger replication of the ability of software complexity metrics to predict programmer performance. *Proceedings of the Fourth International Conference on Software Engineering* (September, 1979), 356-360.

[Dijk68]

Dijkstra, E. W. Go to statement considered harmful. (letter to the editor). *Communications of the ACM* 11, 3 (March, 1968), 147-148.

[Dona80]

Donahoo, J. D., et al. A review of software maintenance technology. RADC-TR-80-13, Rome Air Development Center, (February, 1980).

[Elsh78]

Elshoff, J. L. An investigation into the effect of the counting method used on Software Science measurements. *ACM SIGPLAN Notices*, 13, 2, (February, 1978), 30-45.

[Farr65]

Farr, L. and Zagorski, H. Quantitative analysis of programming cost factors: a progress report. *International Computation Center Symposium Proceedings: Economics of Automatic Data Processing.* (A. B. Frielink, ed.), North-Holland, Amsterdam, (1965), 167-177.

[Feue79]

Feuer, A. R. and Fowlkes, E. B. Some results from an empirical study of computer software. *Proceedings of the Fourth International Conference on Software Engineering* (September, 1979), 351-355.

[Feni79]

Fenichel, R. Heads I win, tails you lose. (in Surveyors' Forum). *ACM Computing Surveys* 11, 3 (September, 1979), 277.

[Fits79]

Fitsos, G. P. Software Science counting rules and tuning methodology. Technical Report 03.075, IBM Santa Teresa Laboratory, (September, 1979).

[Fits80]

Fitsos, G. P. Vocabulary effects in Software Science. Technical Report 03.082, IBM Santa Teresa Laboratory, (January, 1980).

[Fitz78]

Fitzsimmons, A. B. and Love, L. T. A review and evaluation of software science. *ACM Computing Surveys* 10, 1 (March, 1978), 3-18.

[Gord79]

Gordon, R. D. Measuring improvements in program clarity. *IEEE Transactions on Software Engineering* 5, 2, (March, 1979), 79-70.

[Hals77]

Halstead, M. H. *Elements of Software Science*, Elsevier North-Holland, New York, N.Y., (1977).

[Hick52]

Hick, W. E. On the rate of gain of information. *Quarterly Journal of Experimental Psychology* 4, (1952), 11-26.

[Jame77]

James, T. Software cost estimating methodology. *IEEE Proceedings of National Aerospace Electronics Conference*. (1977), 22-28.

[Kern78]

Kernighan, B. W. and Plauger, P. J. *The Elements of Programming Style*. McGraw-Hill Book Co., (1978).

[Lass81]

Lassez, J., Van Der Knijff, D., and Shepherd, J. A critical examination of Software

Science. *Journal of Systems and Software* 2, 2, (December, 1981) (in press). .

[Male80]

Malenge, J. P. Critique de la physique du logiciel (Critique of software science). Publication Informatique (Technical Report) IMAN-P-23, Universite de Nice, France, (October, 1980).

[McCa76]

McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering* 2, 4 (December, 1976), 308-320.

[Moha79]

Mohanty, S. N. Models and measurements for quality assessment of software. *ACM Computing Surveys* 11, 3 (September, 1979), 251-275.

[Moha81]

Mohanty, S. N. Software cost estimation: present and future. *Software Practice and Experience* 11 (1981), 103-121.

[Mora78]

Moranda, P. B. Is Software Science hard? (in Surveyors' Forum). *ACM Computing Surveys* 10, 4 (December, 1978), 503-504.

[Otne80]

Otnes, W. Quantitative analysis of software projects. Technical Report No. 15/80, Division of Computer Science, The Norwegian Institute of Technology, The University of Trondheim, N-7034 NTH - Trondheim, Norway, (1980).

[Perl81]

Perlis, A. J., Sayward, F. G., and Shaw, M. (Ed.). *Software Metrics: An Analysis and Evaluation*. MIT Press, 1981.

[Schn78]

Schneider, V. Prediction of software effort and project duration: four new formulas. *ACM SIGPLAN Notices* 13, 6 (June, 1978), 49-59.

[Shen81]

Shen, V. Y. and Dunsmore, H. E. Analyzing Cobol programs via Software Science CSD TR-348, Department of Computer Sciences, Purdue University, (August, 1980; revised September, 1981).

[Smit79]

Smith, C. P. Practical applications of software science. Technical Report 03.067, IBM Santa Teresa Laboratory, (June, 1979).

[Smit80]

Smith, C. P. A software science analysis of programming size. *Proceedings of the ACM National Computer Conference* (October, 1980), 179-185.

[Stro67]

Stroud, J. M. The fine structure of psychological time. *Annals of New York Academy of Sciences* 138, 2 (1967), 623-631.

[Wals77]

Walston, C. E. and Felix, C. P. A method of programming measurement and estimation. *IBM Systems Journal* 16, 1, (1977), 54-73.

[Wang81]

Wang, A. S. An investigation of the relationship between initial and final programming effort estimates. CSD TR-365, Department of Computer Sciences, Purdue University, (May, 1981).

[Wolv74]

Wolverton,R. W. The cost of developing large scale software. *IEEE Transactions on Computers*. 23, 6 (1974), 615-636.

[Wood80]

Woodfield, S. N. Enhanced effort estimation by extending basic programming models to include modularity factors. Ph. D. Thesis, Department of Computer Sciences, Purdue University, December, 1980.

[Wood81a]

Woodfield, S. N., Dunsmore, H. E., and Shen, V. Y. The effect of modularization and comments on program comprehension. *Proceedings of Fifth International Conference on Software Engineering* San Diego, California (March, 1981), 215-223.

[Wood81b]

Woodfield, S. N., Shen, V. Y., and Dunsmore, H. E. A study of several metrics for programming effort. *The Journal of Systems and Software* 2, 2 (December, 1981) (in press).

[Zweb79]

Zweben, S. H. Response to Fenichel's comment. (in Surveyors' Forum). *ACM Computing Surveys* 11, 3 (September, 1979), 277.