CONSERVATION OF SOFTWARE SCIENCE PARAMETERS ACROSS MODULARIZATION

Lawrence Hunter
Certifiable Minicomputer Project, University of Texas
Austin, Texas 78712

Jose C. Ingojo
Department of Computer Science, Purdue University
Lafayette, Indiana 47907

Current results in software science research provide a potentially powerful tool for software engineering management. Software science parameters including time required to write a program and program length can be estimated from parameters available at the time of program design specification. The application of these results to modularized programs is not straightforward since the derived parameters are nonlinear in vocabulary size. We define an integrated vocabulary for a modularized program. A parameter is said to be conserved across modularization if the parameter value derived from the integrated vocabulary equals the sum of the parameter values derived from the modules; three parameters have been found to be conserved across modularization in well-modularized programs. Failure to exhibit conservation of length can be used to detect excessive or insufficient intermodule communication.

1. BACKGROUND

Software science is an inherently empirical discipline. The methodology of much of the research in this area follows three stages:

(1) Quantitatively describe observed phenomena or relationships involving software measurements.

(2) Hypothesize a mechanism to explain the observations.

(3) Support or disprove the hypothesized mechanism by applying it to additional data.

The empirical study of software requires well-defined quantitative measurements applied to actual programs written in actual programming languages. We refer to these measurements collectively as the software science parameters. The parameters fall into three classes:

(1) Four fundamental parameters directly counted from source code:

N_1, the number of operators used (total operator usage);

N_2, the number of operands used (total operand usage);

n_1, the number of distinct operators used (operator vocabulary);

n_2, the number of distinct operands used (operand vocabulary).

From these we obtain the program length $N = N_1 + N_2$ and the vocabulary size $n = n_1 + n_2$.

(2) Other parameters directly measured from the overall software development cycle. These include, for example, the time required to write a program and the number of delivered bugs. (Most of these parameters are well-defined but not necessarily easy to measure.)

(3) Many useful secondary parameters, derived indirectly from the two directly measured classes. These include:

V^*, potential or minimal possible volume, defined by $V^* = n^* \log_2 n^*$ where n^* is the vocabulary size required to express the program as a single operation in a language which has the program function as an operator. (The length of such a program is n^*.)

V, program volume, defined by $V = N \log_2 n$ (V^* a special case of V).

L, program level, defined by $L = V^*/V$.

E, program effort, defined by $E = V/L$. (E essentially measures the number of mental decisions required to select the operators and operands from the vocabulary used.)

λ, language level, essentially a measurement of average program level in a given language, defined by $\lambda = V^* \times L$.

We refer to the four directly counted parameters as the fundamental parameters, and to the others as derived parameters.

Previous work has resulted in the discovery and validation of many relationships among the para-

meters. The first one reported [1] was the length equation,

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2 \qquad (1)$$

This unexpected but not really counterintuitive relationship showed that program length tends to be characterized by the vocabulary used; as length increases, vocabulary also increases in a directly related way. Another relationship of interest is the invariance of the LV product: for a given algorithm, LxV remains constant regardless of the language in which the algorithm is written. While it is trivially true from the definitions that

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

regardless of language, the potential volume V* is in practice only an unreachable limit value. Yet, the invariance of the LV product has been empirically validated. A complete introduction to software science is given in [2].

The first effort at giving a formal mechanism for the programming activity which would explain the results mentioned above was [3]. While the models given were greatly oversimplified, they gave surprisingly close agreement with observed data. That situation is typical of the current state of the art in software science: the relationships currently known are only approximately true of observed programs, but they are very close to being exact. Due to the simplifications of the models, software science still deals largely with first-order effects.

A brief consideration of the length equation above will reveal an important phenomenon: it is very easy to write programs for which the software science relationships (such as the length equation eq. (1) or the invariance equation eq. (2)) do not hold, but doing so requires the use of such practices as unwarranted assignments and redundant subexpression evaluation. An early study [4] of program features which prevented the length equation eq. (1) from holding led to the discovery of program impurity classes, a well-defined classification of what are informally known as "poor programming practices". Because of phenomena like these, software science research efforts must follow two related but distinct goals: first, the quantitative description of relationships which hold (approximately) in well-written programs; and second, the study of those program features which cause the relationships not to hold in poorly-written programs.

2. APPLICATION TO SOFTWARE ENGINEERING MANAGEMENT

2.1 Estimating Parameters from Program Specification Information

It is shown in [2] that most of the software science parameters for a given program can be estimated surprisingly well from only two independent parameters: the language level λ of the language used, and n_2^*, the number of distinct items of input information. Informally, this means that a program is largely characterized by the absolute minimum vocabulary required to express it.

In particular, two of the parameters which can be estimated from n_2^* and λ are T, the time required to write the program, and N, the length of the program. The time T is estimated from the program effort E (defined earlier) and known estimates of the rate at which a concentrating human being can perform elementary mental discriminations (approximately 18 per second). The length N is estimated by first obtaining approximations of the number of distinct operators n_1 and operands n_2, then using the length equation.

These results hold obvious potential for the development of powerful software engineering management tools. Estimates of the writing time T and the program size N can be made from information which is known at the time the detailed program design specifications are made. It has also been shown [5] that estimates of the number of bugs remaining in the program when it is delivered for system integration can be made. These estimates could be used to decide how much more testing and debugging effort was likely to be productive.

2.2 Application to Modularized Programs

The estimation methods described above are still rather crude, and they depend on knowing n_2^* and λ more precisely than may currently be possible. These appear to be limitations of refinement rather than basic method. Previous work in software science has largely dealt with program modules as separate programs. In order to be usable as a software engineering management tool, parameter estimation must be applicable to highly modularized programs. In practice the modules will be interrelated. It is important to manage the development of both the individual modules and the project as a whole. And it is important to insure that good programming practices are used both within the individual modules and in the use of the modules on a higher level.

All of the derived parameters can be derived (or estimated) from the four fundamental, directly-counted parameters. But we observe that the derivations are nonlinear in the fundamental parameters. This means that we cannot simply add up the lengths and vocabulary sizes of individual modules, and then derive the parameters for the overall project from their sums. Yet the length N, writing time T, and other parameters of the overall project are directly related to the parameters of the modules. Thus we are led to the central question of this research: how are the derived parameters of a modularized program related to the parameters of the separate modules?

3. INTEGRATED PARAMETERS

3.1 Integrated Parameter Definition

The length of a modularized program must be the sum of the lengths of its modules, but the size of the vocabulary of a modularized program is not the sum of the sizes of the vocabularies of its modules. This is because there must be some duplication of vocabulary elements in different

modules in order to have intermodule communication. Duplication of vocabulary elements may be in the form of either global items or formal and actual parameters.

Let a program consist of M modules. Define the integrated vocabulary VOC_{int} of the program as

$$VOC_{int} = \bigcup_{i=1} (VOC_i - FP_i) \bigcup PO \quad (3)$$

where

VOC_i is the vocabulary of the i^{th} module;

FP_i is the set of formal parameters (operands) of the i^{th} module;

PO is a set of pseudo-operands containing one element for each actual parameter expression which is not an operand in the set VOC_i of the calling site.

The effect of the definition is to make VOC_{int} a "global" vocabulary. Formal parameters are deleted since they do not add operands to the overall vocabulary; they only serve to define operations. Actual parameter pseudo-operands are added, since each is conceptually an operand of the procedure call where it is used; in most cases, a compiler will treat such an expression as a temporary value-operand.

We define the integrated versions of the software science parameters as being those counted or derived from the integrated vocabulary. We use a subscript "int" to designate an integrated parameter. The integrated length, for example, is given by the predicted value $\hat{N}_{int}$, where

$$\hat{N}_{int} = n_{1_{int}} \log_2 n_{1_{int}} + n_{2_{int}} \log_2 n_{2_{int}} \quad (4)$$

When computing the length equation for the integrated program, therefore, the integrated parameters must be used.

3.2 Conservation across Modularization

Let M be the number of modules in a program. Let P designate any software science parameter; P_{int} is the integrated version of P, and P_i is P for module "i", $1 \leq i \leq M$. Parameter P is conserved across modularization if

$$P_{int} = \sum_{i=1}^{M} P_i \quad (5)$$

The following parameters have been found to be conserved across modularization:

(1) Conservation of Length via the Length Equation

$$\hat{N}_{int} = \sum_{i=1}^{M} \hat{N}_i \quad (6)$$

(2) Conservation of Potential Volume

$$V^*_{int} = \sum_{i=1}^{M} V^*_i \quad (7)$$

(3) Conservation of Modular Effort

$$E_{int} = \sum_{i=1}^{M} E_i \quad (8)$$

This list of conserved properties is not meant to imply an exclusive list of such properties, since further ones may yet be uncovered.

4. CONSERVATION PROPERTIES

4.1 Empirical Data

A set of 14 programs were selected for study to investigate conservation properties. The criteria for selection were that the program be readily available in published form and modularized. Seven were selected from the Algorithms section of CACM [6]; four were selected from a set of programs for use in chemical engineering education [7]; one is a published compiler [8]; and one is a program for counting software science parameters [9]. The last program in the data set is one used in a tutorial on software science integrated parameters [10]. This is tabulated in Table 1 below.

Table 1. Programs in the Data Set

Algorithm Designation	Source	Number of Modules	Language Utilized
A	CACM	3	FORTRAN
B	[10]	3	FORTRAN
C	CACM	4	FORTRAN
D	CACM	4	ALGOL 60
E	CACM	4	FORTRAN
F	CACM	5	FORTRAN
G	CACHE	5	FORTRAN
H	CACM	7	ALGOL 60
I	CACHE	7	FORTRAN
J	CACHE	7	FORTRAN
K	CACM	8	ALGOL 60
L	[8]	14	PILOT
M	[9]	24	FORTRAN
N	CACHE	36	FORTRAN

The integrated versions of the fundamental parameters are given in Table 2. From these values, the other integrated parameters may be computed with one exception: the program level L. In order to compute L, both for modules and the integrated program, the program level equation will be used to approximate it [2],

$$\hat{L} = \frac{n^*_1}{n_1} \times \frac{n_2}{N_2} \quad (9)$$

where $n^*_1 = 1 + m$ and m is the number of modules in the code. For individual modules of a program, $m = 1$ and hence, $n^*_1 = 2$. For the integrated program, $m = M$ and hence, $n^*_1 = 1 + M$.

Table 2. Fundamental Parameters for the Integrated Programs

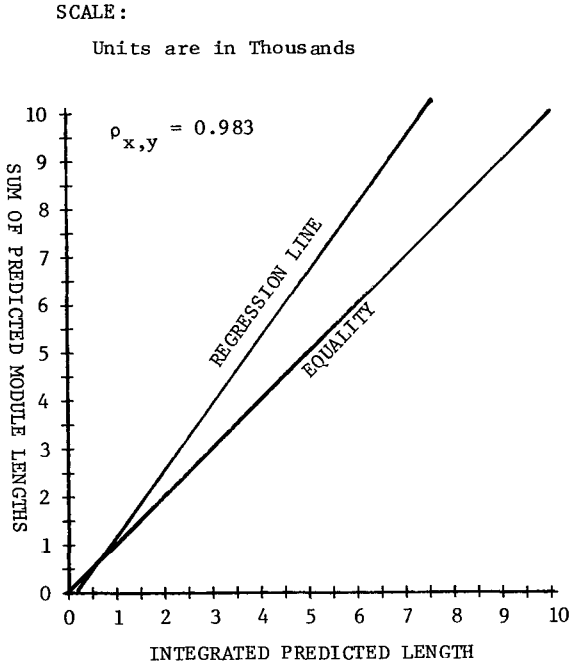
Algorithm Designation	n_1	n_2	N_1	N_2	N
A	18	43	171	185	356
B	14	17	69	49	118
C	61	133	759	677	1 436
D	32	65	483	439	922
E	32	70	468	378	846
F	64	160	1 007	942	1 949
G	80	150	846	623	1 469
H	31	167	1 124	1 239	2 363
I	132	233	1 646	1 211	2 857
J	185	345	2 394	1 732	4 126
K	32	76	439	405	844
L	83	96	763	571	1 334
M	247	185	2 644	1 666	4 310
N	320	368	3 561	2 204	5 765

Extensive tables containing the fundamental parameters for the individual modules of each program may be found in [10,11,12].

4.2 Conservation of Predicted Length

Conservation of observed length is trivially true across modularization. Predicted length, as computed by the length equation eq. (1), is not, due to the nonlinearity of the equation. Nevertheless, predicted program length appears to be conserved across modularization. The third column of Table 3 gives the integrated predicted length (obtained from the length equation using the integrated vocabulary). The fourth column gives the sums of the predicted module lengths. Figure 1 is a graph of integrated predicted lengths against sums of predicted module lengths. The coefficient of correlation between the two is 0.983, indicating a linear relationship; the slope of the least-squares line shown in Figure 1 is 1.358. The coefficient of correlation between the integrated predicted length and the observed length is 0.978, and between the sums of the predicted module lengths and the observed length is 0.971. The corresponding slopes of the least-square lines are 0.977 and 1.341 respectively. Thus, conservation of predicted length is also consistent with the observed values of length.

Figure 1. The Conservation of Predicted Length Across Modularization



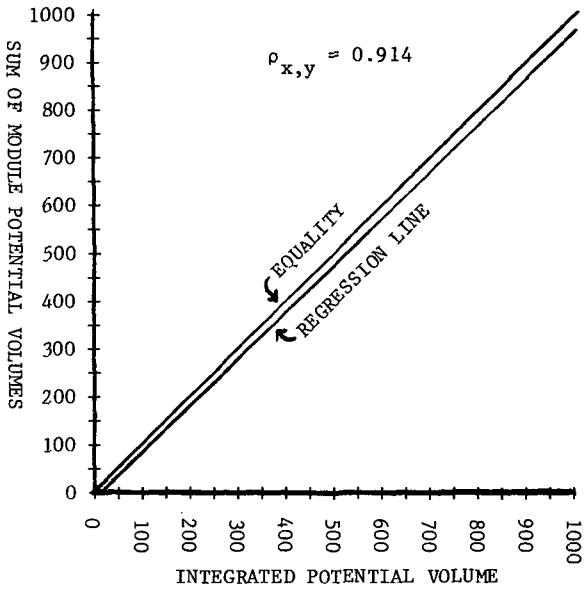
4.3 Conservation of Potential Volume

Potential volume also appears to be conserved. In the fifth and sixth columns of Table 3, the integrated potential volumes and sums of module potential volumes are given. Figure 2 is the corresponding graph. The coefficient of correlation is 0.914 and the slope of the least-square line is 0.972, indicating approximate equality.

Table 3. Integrated Parameter Values vs. Summed Modular Parameter Values

Program Attributes:		(1) Predicted Length:		(2) Potential Volume:		(3) Modular Effort:	
Algorithm Designation	Modules M	$\hat{N}_{Int}$	$\hat{N}_{Sum}$	V^*_{Int}	V^*_{Sum}	E_{Int}	E_{Sum}
A	3	308	390	109	91	40 877	50 006
B	3	123	123	58	42	5 898	7 416
C	4	1 300	1 376	176	190	677 735	513 242
D	4	551	662	141	120	263 027	288 338
E	4	589	636	163	109	195 087	245 409
F	5	1 556	1 535	242	214	955 598	913 194
G	5	1 590	1 731	206	212	638 230	538 499
H	7	1 387	1 580	627	275	518 296	861 318
I	7	2 762	3 046	282	235	2 075 638	2 944 344
J	7	4 302	4 678	325	332	4 334 919	7 890 650
K	8	635	979	301	200	108 021	132 116
L	14	1 161	1 461	303	326	328 572	230 300
M	24	3 357	4 830	434	586	3 357 284	5 989 572
N	36	5 800	8 426	1 048	1 111	2 814 222	4 903 301

Figure 2. The Conservation of Potential Volume Across Modularization



4.4 Conservation of Modular Effort

The effort involved in constructing a modularized program is composed of the effort needed to write the individual modules and the effort needed to integrate the modules into a functional whole. The first is the modular effort, and the second component is the integrating effort. The data suggests that modular effort is also conserved. The seventh column of Table 3 gives the modular effort as computed from the integrated parameters. The eighth column gives the modular effort as the sum of the module efforts. The coefficient of correlation is 0.993 and the slope of the least-squares line is 1.828. Figure 3 is a graph of the modular efforts computed by the two methods.

5. ANALYSIS

5.1 Evaluation of Results

The work reported here is based on a small sample. The sample does include a wide range of program lengths, several authors, and several languages. Conservation of properties is exhibited as equality of an integrated measurement and the sum of the corresponding measurements over modules. We have taken two pieces of evidence for conservation: a high coefficient of correlation (indicating linearity) and a least-squares line with slope near 1 (indicating equality).

Within our sample, there is convincing evidence for conservation of predicted length and potential volume. The evidence for conservation of modular effort is less convincing. Effort measurements for total module construction show high correlation, but the slope of the least-squares line is nearly twice the value of unity.

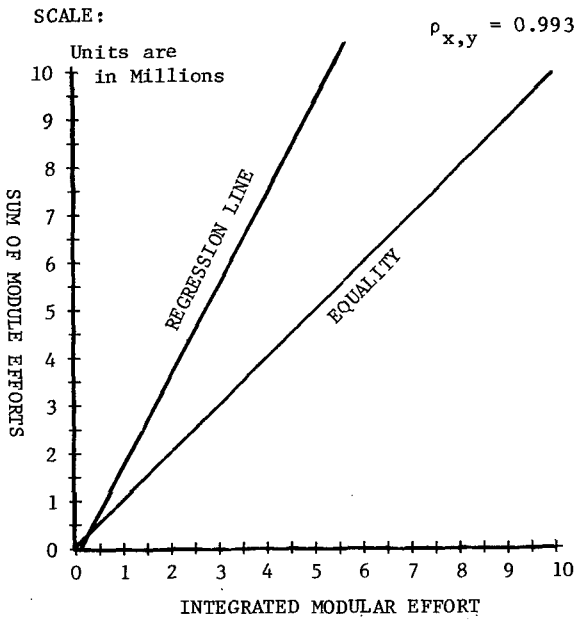
The conservation of software science properties hold in the same sense as other software science relationships, as discussed in section 1. The conservation of such properties is not exactly a true relationship for any programs in general. But the conserved integrated parameters are approximately equal to the summed corresponding module parameters over a variety of modularized programs.

5.2 Toward a Theory of Modularization Impurities

We have only found convincing evidence for the conservation of two parameters: predicted length and potential volume. Additional evidence has been found for the conservation of modular effort. Work currently in progress has shown that these are significant. In particular, conservation of length can be shown to be governed by the sharing of global information among module vocabularies, and conservation of potential volume can be shown to be governed by the parameterization of modules. Furthermore, it can be shown that instances of what are generally considered to be poor practices in modularization will cause one or the other conservation not to hold. By examining ways in which conservation of length or potential volume can be violated, we are developing a theory of "modularization impurities". This appears to be a natural extension of what Bulut and Halstead's theory of "algorithm impurities" [4] to the situation of several modules.

For example, algorithm H is the only one in our sample which conserves neither length nor potential volume. It can be shown that an excessive amount of global information will cause the integrated length to be much smaller than the sum of module lengths. Algorithm H violates conservation of length in this direction, and it does appear to

Figure 3. The Conservation of Modular Effort Across Modularization



have a lot of global information shared among routines. It can also be shown that if many call sites have the same actual parameter expression corresponding to a given formal parameter, the integrated potential volume will be much larger than the sum of module potential volumes. Algorithm H violates conservation of potential volume in this direction, and it has several formal parameters for which the same actual parameter expression is used at all call sites.

6. CONCLUSIONS

The conservation of software science properties reported here do not represent rigorous conditions for good modularization. But our empirical study shows that they are exhibited by programs which we subjectively feel are well modularized. The conservation of such properties as predicted length, potential volume, and modular effort apparently represent a balance between overhead due to intermodule communication and the higher level of abstraction provided by modularization.

ACKNOWLEDGEMENT

Special thanks are due to Prof. M.H. Halstead, who guided the research on conservation of software science parameters. Further documentation for the research for this paper can be found in [12].

REFERENCES

[1] M.H. Halstead, Natural laws controlling algorithm structure?, ACM Sigplan Notices, vol. 7, no. 2, February 1972.

[2] M.H. Halstead, Elements of software science, American Elsevier, New York NY, 1977.

[3] M.H. Halstead and Rudolph Bayer, Algorithm dynamics, Proc. ACM National Conference, Atlanta, 1973.

[4] Necdet Bulut and M.H. Halstead, Impurities found in algorithm implementation, ACM Sigplan Notices, vol. 9, no. 3, March 1974.

[5] Linda Cornell and M.H. Halstead, Predicting the number of bugs expected in a program module, Proc. Computer Management Group International Conference, Atlanta, 1976.

[6] "Algorithms" section, Communications of the ACM, vols. 16 - 17.

[7] CACHE, Computer Programs for Chemical Engineering Education, vols. 1 - 7. Aztec Publishing Co., Austin TX, 1972.

[8] M.H. Halstead, A laboratory manual for compiler and operating system implementation, American Elsevier, New York NY, 1974.

[9] Karl Ottenstein, A program to count operators and operands for ANSI FORTRAN modules, Computer Sciences Report TR 196, Purdue University, June 1976.

[10] J.C. Ingojo, Deriving the integrated vocabulary for a modularized program, Computer Sciences Report TR 77-4, Indiana University-Purdue University at Indianapolis, February 1977.

[11] J.C. Ingojo, Modularization in the PILOT compiler and its effect on the length, Computer Sciences Report TR 169, Purdue University (complete); also in abridged form, Proc. ACM National Conference, Houston, 1976.

[12] J.C. Ingojo, Modularity properties in software science, Ph.D Thesis, Purdue University, October 1977.