An Overview of Halstead Language and Program Complexity Metrics (August 1, 2011)

Peter J. Inslee, *Undergraduate, CS410 Mastery in Programming, Summer Term 2011*

***Abstract*—This paper provides an overview of "Software Science", the term given by Maurice Halstead to his collection of software metrics.**

***Index Terms*—Software Science, software complexity metrics, Maurice Halstead, programming effort, static code analysis**

I. Introduction

THIS document describes one of the first software metrics ever to be presented to the computing community at large, the Halstead software complexity metrics, also known collectively as "Software Science". Halstead introduced the theory of software science in 1977, when his book "Elements of Software Complexity"[1] was published. His work brought attention to the need for empirical methods in the field of computer science. The Halstead software complexity metrics are still considered and used to this day.

The author's goal in writing this paper was to summarize Maurice Halstead's theory, to provide some historical context for understanding what problems it was invented to address, and to discuss the extent to which it has fulfilled its purpose. In addition, some of the strengths and weaknesses of the Halstead software metrics are highlighted and compared with those of other alternative software metrics of comparable popularity.

II. Motivation

To improve the quality of software, it is necessary to define or quantify one's notion of "software quality". To the extent it is possible to pinpoint and clarify this concept, it then becomes possible to measure it, study it, and conduct repeatable experiments which reveal what factors and relationships affect it. This, in turn, provides the means for evaluating various implementations, and justification for making claims about the fitness or suitability of one method of software development over another.

This basic recognition, that the full force of the scientific method could be (and needed to be) applied in the field of software development to the evaluation of programs, was Maurice Halstead's great contribution. His theory was a direct response to the "software crisis". This was the phrase coined in the late 1960s to describe the perceived lack of effective policies and methodologies for managing and ensuring the timely and cost-effective design, implementation, delivery and maintenance of bug-free software. This crisis forced the conception of "software evaluation as a science" to the surface. It is interesting to note that M. Halstead's background was in meteorology and the physical sciences, and for years he worked in the Navy developing weather prediction software. It is not surprising that someone with an education in applied physics is responsible for introducing the idea of experimental measurement to the field of computing.

Completed and submitted August 1, 2011. This research was done without sponsorship, in fulfillment of the requirements of CS410, *Mastery in Programming*, under the instruction of Herb Mayer, Ph.D.

P. J. Inslee is an undergraduate student majoring in Computer Science, currently enrolled at the Maseeh College of Engineering & Computer Science of Portland State University, Portland, OR 80305 USA (503-888-2754; email: pjinslee@gmail.com).

TABLE I
HALSTEAD SOFTWARE COMPLEXITY METRICS

Symbol	Quantity	Equivalent Expression
η_1	Unique operator count	Primary measure
η_2	Unique operand count	Primary measure
N_1	Total operator count	Primary measure
N_2	Total operand count	Primary Measure
η	Vocabulary size	$\eta_1 + \eta_2 = k\eta'$
N	Program length	$N_1 + N_2$
$\hat{N}$	Program length	$\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
V	Program volume	$N \log_2 \eta$
V^*	Potential volume	$\eta^* \log_2 \eta^*$
V^{**}	Boundary volume	$\eta^* \log_2 \eta^*????$
η^*	Potential vocabulary	$\eta_1^* + \eta_2^* = 2 + \eta_2^*$
L	Program level	V^* / V
$\hat{L}$	Program level	$\eta_1^* \eta_2 / \eta_1 N_2 = 2\eta_2 / \eta_1 N_2$
I	Intelligence content	$\hat{L}V \approx V^*$
λ	Language level	$LV^* = L^2 V$
E	Effort	$V / L = V^2 / V^*$
f	Frequency	Primary measure
k	Redundancy Factor	Primary measure
M	Number of modules	Primary measure
β	Block size	Primary measure
v	Branch count	Primary measure
T	Implementation Time	E / S
D	Difficulty	$1 / L$

These are the same symbols used by Halstead in [].

III. THE SOFTWARE SCIENCE APPROACH

The content of this section is taken in its entirety from [1], in which Halstead presents his approach to constructing software metrics. Much of the theory he proposes is based on common- sense intuitions and assumptions about software quality, and is accompanied by an emphasis on the need for experimental verification of results. He proposes to count just four fundamental quantities which can be derived from the lexical analysis of any given source file; the number of distinct operators (η_1), the number of distinct operands (η_2), the total number of operators, including repetitions, (N_1) and the total number of operands, also including repetitions (N_2). He defines the vocabulary η of the program to be the sum of the former two quantities, $\eta = \eta_1 + \eta_2$, and the length N of the program to be the sum of the latter two quantities, $N = N_1 + N_2$. He justifies the partitioning of lexical constructs into just the two distinct classes, since assembly instructions (to which all code is typically reduced during at least some stage of the compilation/interpretation process) can be viewed as a simple pairing of a machine instruction with the operand(s) upon which the instruction operates. Aside from a few ambiguities associated with how one chooses to differentiate “operator” from “operand” and how to associate tokens with these categories (both issues will be addressed later in this section), these metrics provide a straightforward way to characterize the source code of a program, and serve as a (potentially) well-defined foundation on which to build a theory of software evaluation.

A. *Metrics*

From the aforementioned purely syntactic textual measurements, Halstead derives and advises the adoption of the following metrics, summarized in TABLE I.

Program Length

In addition to the theoretical program length N just given, a second derivation of this metric is possible. First note that N has η as a lower limit. Some additional assumptions allow for placing an upper bound on N as well, without having to count it directly. If we suppose that any given source program or text will be written with a mind to avoiding unnecessary redundancy, we may conclude that it will not contain any identical substrings of length η. That is, if we were to divide the given source into N/η statements, each of length η, we may conclude that no two of these statements would be the same; any reasonable programmer would assign a name (in the form of a macro, variable, or procedure) to any such duplicate expression, thereby eliminating the redundancy, and increasing the vocabulary count η by 1. We assume, for example, that a program with vocabulary {a, b, c}, $\eta = 3$ and length $N = 9$, will not have the form "abc abc bca", since it contains two occurrences of "abc", each of length η. As the program in question is constrained to consist of distinct substrings of length η, then there can be at most η^η such substrings in the program, since there are only that many possible combinations of η things taken η at a time.

Moreover, it is observed that operators and operands tend to alternate. This further restricts the combinatorial possibilities for the program, imposing on N the upper bound $\eta \times \eta_1^{\eta 1} \times \eta_2^{\eta 2}$. By an argument presented in Chapter 2 of [1], it makes sense to equate the total number of possible combinations of operators and operands with the number of elements in the power set of N, namely 2^N. This allows the following derivation of $\hat{N}$, an alternative, experimentally testable, definition of program length:

$$2^N = \eta_1^{\eta 1} \times \eta_2^{\eta 2} \tag{1}$$

$$\hat{N} = \log_2 (\eta_1^{\eta 1} \times \eta_2^{\eta 2}) \tag{2}$$

$$\hat{N} = \eta_1 \times \log_2 \eta_1 + \eta_1 \times \log_2 \eta_1 \tag{3}$$

Program Volume

The volume V of a program is defined to be

$$V = N \times \log_2 \eta. \tag{4}$$

Two different rationalizations for its use can be provided.

Note that if a given program has a total of $\eta = 8$ distinct operators and operands, only 3 bits per element are needed to represent each of them unambiguously. In general, the minimum number of bits required to represent each of the distinct elements in a given program vocabulary is $\log_2 \eta$. Therefore, if there are N total such instances of these elements in a given program, the product $N \times \log_2 \eta$ gives the minimum number of bits needed to represent the given program.

Alternatively, program volume can be viewed as the number of “mental comparisons” required by the programmer to generate the program. This view is based on the assumption that the efficiency of a binary search algorithm approximates the performance of a programmer in choosing which element from an ordered list of η operators and operands to commit to the page. Since the performance of binary search is known to take $\log_2 \eta$ comparisons for an ordered list of η items, and since the writing of a program is basically a series of “N non-random selections from a list of η items”, the total number of comparisons required is $N \times \log_2 \eta_2$.

Potential Volume

Using the formula for program volume derived above, another metric, called the minimum, or potential volume V^* can be defined. It is intended to quantify the volume of a given algorithm when expressed in its tersest conceivable form. A program in this minimal form would employ the minimal number of operators and operands necessary, denoted η_1^* and η_2^* respectively. It would also be free of repetitions, so the total number of operators N_1 would be equal to η_1^*, and likewise, the total number of operands N_2 would be equal to η_2^*. Substituting these into the formula for volume gives

$$V^* = (\eta_1^* + \eta_2^*) \times \log_2 (\eta_1^* + \eta_2^*). \tag{5}$$

This result can be refined by considering that the most concise expression of an algorithm is nothing more than a call to a procedure which implements the algorithm, along with the arguments that it requires. As an example, consider a function "hypotenuse(leg1, leg2)" which computes the hypotenuse of a right triangle, given the lengths of its legs. Note that this procedure call is composed of 2 operators, the function name "hypotenuse" and the grouping operator "(...)". Since any procedure call will necessarily have this essential logical form, regardless of the language within which it is written, one may conclude that, for any program, $\eta_1^* = 2$. This insight yields

$$V^* = (2 + \eta_2^*) \times \log_2 (2 + \eta_2^*). \tag{6}$$

Program Level

Although they are related conceptually, the program level L should not be confused with the language level λ, which will be introduced later. In the same way that a high-level language is able to express more using fewer statements, the epitome of a high-level program is written in such a way that it expresses as much as is necessary in the most succinct way possible. Using this criterion, the level of a given program is defined as the ratio $L = V^*/V$, so that the level of the program increases to 1 as its volume is reduced to the minimum possible, the potential volume V^*, which would be nothing more than a procedure call. Hence L is always a value between 0 and 1. This fits with our expectation that a call to a procedure will be easier to write than the entire procedure itself. Using the previous example, a low-level program for computing the hypotenuse of a right triangle would express each step of the computation in voluminous detail, ostensibly multiplying each given length with itself, adding the results, and calculating the square root of the sum. A higher level language might instead make procedure calls to perform the squaring and rooting operations, hiding some of the details of the computation, but making the procedure more readable overall. A program with volume equal to its potential volume will simply have the form "hypotenuse(leg1, leg2)", the most succinct possible for this computation, (naming considerations aside).

Adding more unique operators to a program is seen as reducing the program level, so L is proportional to η_1^*/η_1. Also, if we consider the duplication of operands within a program to diminish the level of the program, we have that L is proportional to η_2/N_2. By combining these ratios into a single equation, we derive an alternative definition of L, denoted $\hat{L}$, given by

$$\hat{L} = (\eta_1^*/\eta_1) \times (\eta_2/N_2) = (2/\eta_1) \times (\eta_2/N_2) = 2\eta_2/(\eta_1 N_2) \tag{3}$$

Programming Effort

…

Programming Time

…

Language Level

There is a tradeoff at work here, with a higher level language being more comprehensible to someone fluent in the language or domain, but perhaps too cryptic for someone unfamiliar with it, who would then require lower-level descriptions of the same procedures to render them comprehensible, at the expense of more volume.

B. Predicting Programming Effort and Time

…

C. Predicting Error Rates

…

D. Other Applications

…

IV. Strengths and Weaknesses

A. Strengths

The greatest strength of Halstead’s metrics is their simplicity. It is easy to automate the counting of operators and operands in source code, as it can be done concurrently during the compilation process. Most of his expressions are simple enough even to be calculated on “the back of the envelope”, as a first order approximation that can be used in initial attempts at predicting software production costs and expected bugs. This simplicity endows it with many of the same beneficial properties that are shared by static code analysis techniques in general.

Another strength of this evaluation method is its platform agnostic scoring of code quality. In the same way that portability of program source code is viewed as a virtuous attribute, so too is the consistent evaluation of software across different architectures, in Halstead's view. After all, his metric is a "software" metric, and should not be tainted by the choice supporting hardware. This consistency is guaranteed by virtue of the measurements relying solely upon the tokens present in the source code. (Whether these metrics are truly invariant across platforms is another matter, but at least their measurement as specified is independent of the machine on which the program in question is executed.

This reliance of the metric on the artifact of the source code alone for its computation implies that, unlike the hardware, the language of the implementation may be taken into account, and *does* play a role when compared with functionally equivalent code written in another language. In Halstead's view, not all languages are created equal, (or at least not equally appropriate for application to a given problem), and so his metric is devised in such a way that it can be used not only to draw conclusions about a specific program, but also about entire programming languages, when applied across a sufficiently large sample of functionally equivalent programs. This feature of the metric is part of its appeal, as in some sense it provides the hope that certain invariants or underlying patterns of software complexity might be revealed, in the same way that data obtained from barometers and thermometers in the field, when analyzed appropriately, yields insight into general patterns of emergent weather patterns.

B. *Weaknesses*

Unfortunately, many of the hopes for Halstead's complexity measures remain unrealized, apparently because of unsupportable hidden assumptions implicit in their prescription. (Most of my sources include reasons for taking Halstead's assumptions with a grain of salt, which I will lay out in all their gory detail here. The following paragraph provides a glimpse of what I hope to include.)

One of the main weaknesses inherent in the "software science" approach is the flip-side of its strength; simple token counting has the benefit of providing objective statistics (that may or may not be useful) for describing a given program, but it does nothing to provide insight into the semantic structure of the program. For instance, [53] points out that all permutations of operators and operands in a given program will result in the same Halstead complexity measure, whether they are sensible programs or not. Note how this tradeoff is analogous to the tradeoff in software QA between "black-box" and "white-box" testing. Halstead's approach mirrors the benefits of a "black-box" approach, which enforces objectivity at the expense of ignoring possibly insightful structural details of a system. This same quality Halstead's metrics difficult to apply effectively to the analysis of object-oriented code, although several practitioners, have found ways of reconciling this issue.

V. Alternatives

This section lists contending theories and software metrics that have been proposed since 1977. Most of them are constructed to address some specific flaw or shortcoming discovered in those previously put forth, but each has its own inherent ambiguities and weaknesses as well. The following sections exhibit some of the most notable of these candidate software metrics, and compare their respective strengths and weaknesses with those of Halstead.

A. *Source Lines of Code (SLOC)*

This is worth mentioning, as it is one of the few software metrics that was used before Halstead suggested improving upon it. Surprisingly, it is still commonly used, and some claim (yet to be cited) that most of Halstead's metrics correlate highly with this measure; in essence, demonstrating that his metrics don't reveal much more about a given program than its length.

B. *Cyclomatic Complexity*

Introduced by Thomas J. McCabe in 1976, this metric takes the structure of a programs flow of control into consideration.

C. *Function Point Analysis*

According to [13], "an ISO recognized software metric to size an information system based on the functionality that is perceived by the user of the information system."

D. *Instruction Path Length*

A way of evaluating program performance, basically by counting the number of machine instructions required to execute it.

VI. Applications, Uses, Implementations

Several software evaluation systems include Halstead metrics as part of their analysis, and many more perform calculations similar to those proposed by Halstead. Some examples include:

VerifysoftTechnology's "TestwellCMT++" and "CMTJava"

Virtual Machinery's "JHawk"

IBM's "Logiscope"

Dan Galorath's "SEM-SEER", and

Geonius' "nPath", among others.

Acknowledgment

P. J. Inslee thanks Herb Mayer for introducing the author to this fascinating topic, and inspiring him to understand and explore the work of M. H. Halstead as it applies to measuring and reducing code complexity.

References

[1] Halstead, Maurice H. (1977). Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc.. ISBN 0-444-00205-7.

[2] http://answers.oreilly.com/topic/2258-do-we-need-more-software-complexity-metrics/

[3] http://books.google.com/books?hl=en&lr=&id=vtNWAAAAMAAJ&oi=fnd&pg=PR3&dq=halstead+complexity+measure+tutorial&ots=aXY_rMH6eC&sig=t7q-Ums3yVzyjUhyfaoaJDkIQTQ#v=onepage&q&f=false

[4] http://code.google.com/p/prest/

[5] http://delivery.acm.org/10.1145/80000/76382/p1415-mccabe.pdf?ip=131.252.212.118&CFID=35207185&CFTOKEN=21771008&__acm__=1310793638_2fa50fb62ef41121c9b5bd406797b441

[6] http://delivery.acm.org/10.1145/950000/947916/p58-salt.pdf?ip=131.252.212.111&CFID=35342169&CFTOKEN=50084565&__acm__=1310870378_7dba826353539a0046cde91e0c3b16a5

[7] http://dl.acm.org/author_page.cfm?id=81100290291
[8] http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1062&context=cstech
[9] http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1166&context=cstech
[10] http://en.wikipedia.org/wiki/Code_review
[11] http://en.wikipedia.org/wiki/Comparison_of_development_estimation_software
[12] http://en.wikipedia.org/wiki/Cyclomatic_complexity
[13] http://en.wikipedia.org/wiki/Function_point
[14] http://en.wikipedia.org/wiki/Halstead_complexity_measures
[15] http://en.wikipedia.org/wiki/Software_metric
[16] http://en.wikipedia.org/wiki/Software_metrics
[17] http://groups.engin.umd.umich.edu/CIS/tinytools/cis375/f03/halstead-php/index.php
[18] http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/chapter2.htm
[19] http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00087287
[20] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1703032
[21] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1703110
[22] http://nsfcac.rutgers.edu/TASSL/Thesis/Hilda_Thesis.pdf
[23] http://portal.acm.org/citation.cfm?id=800508
[24] http://portal.acm.org/citation.cfm?id=805693
[25] http://portal.acm.org/citation.cfm?id=807762
[26] http://publib.boulder.ibm.com/infocenter/rassan/v5r5/index.jsp?topic=/com.ibm.raa.doc/common/cstats.htm
[27] http://publib.boulder.ibm.com/infocenter/rassan/v6r0/index.jsp?topic=/com.ibm.help.raa.doc/common/chalstd.htm
[28] http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.logiscope.doc/pdf66/ReviewerJava.pdf
[29] http://scssummary.googlecode.com/svn-history/r2/trunk/Papers/Halstead_Paper.pdf
[30] http://sluglug.ucsc.edu/pipermail/sluglug/2002-June/018269.html
[31] http://sunnyday.mit.edu/16.355/kearney.pdf
[32] http://umrefjournal.um.edu.my/filebank/published_article/1686/10.pdf
[33] http://web.cs.wpi.edu/~gpollice/cs562-s05/Readings/DefectPredictionCritique.pdf
[34] http://www-01.ibm.com/software/awdtools/logiscope/
[35] http://www.cs.toronto.edu/~yijun/csc408h/handouts/tutorial5.pdf
[36] http://www.cs.ttu.edu/fase/v8n06.txt
[37] http://www.cs.umd.edu/class/spring2006/cmsc735/slides/CMSC735%204%20Product%20Models%20and%20Metrics.pdf
[38] http://www.emunix.emich.edu/~ahmad/compmeas.pdf
[39] http://www.exforsys.com/tutorials/testing/metrics-used-in-testing.html
[40] http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/CT-91-1-SQM.pdf
[41] http://www.fi.muni.cz/~sochor/PA103/Slajdy/OO_metriky_navrhu.pdf
[42] http://www.geonius.com/software/tools/npath.html
[43] http://www.horst-zuse.homepage.t-online.de/halstead.html
[44] http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/h/Halstead:Maurice_H=.html
[45] http://www.it.kau.se/cs/education/courses/davddiss/Uppsatser_2008/E2008-01.pdf
[46] http://www.it-smc.com/Knowledge/Metrics.pdf
[47] http://www.leepoint.net/notes-java/principles_and_practices/complexity/complexity_measurement.html
[48] http://www.mccabe.com/iq_research_metrics.htm
[49] http://www.nickerson.to/visprog/ch7/ch7.htm
[50] http://www.niwotridge.com/Resources/PM-SWEResources/MetricsTools.htm
[51] http://www.program-transformation.org/Transform/FatherOfDecompilation
[52] http://www.semdesigns.com/Products/Metrics/CSharpMetrics.html
[53] http://www.systemsarchitecture.org/mane/Text/MitThesis.pdf
[54] http://www.verifysoft.com/en_halstead_metrics.html
[55] http://www.verifysoft.com/en_software_complexity_metrics.pdf
[56] http://www.virtualmachinery.com/sidebar2.htm
[57] http://yunus.hacettepe.edu.tr/~sencer/complexity.html