THE MEASUREMENT OF SOFTWARE SCIENCE PARAMETERS IN SOFTWARE DESIGNS

Paul A. Szulewski
Mark H. Whitworth
Philip Buchan
J. Barton DeWolf

ABSTRACT

Metrics of software quality have historically focused on code quality despite the importance of early and continuous quality evaluation in a software development effort. While software science metrics have been used to measure the psychological complexity of computer programs as well as other quality related aspects of algorithm construction, techniques to measure software design quality have not been adequately addressed. In this paper, software design quality is emphasized. A general formalism for expressing software designs is presented, and a technique for identifying and counting software science parameters in design media is proposed.

ACKNOWLEDGEMENT

This work was funded in part by the National Bureau of Standards' Institute of Computer Sciences and Technology, Contract NB79SACA0220. The U.S. Government retains the right to reproduce all or portions of this paper and to authorize others to do so for the U.S. Government purposes.

The authors are with the Charles Stark Draper Laboratory Inc., Cambridge, MA., with the exception of Mark H. Whitworth, who is with ITP boston Inc., Cambridge, MA.

If a design is sufficiently well developed, translation of the design into a UDD may require the omission of some design detail. In these cases, extension of the UDD to allow the expression of transformations and predicate evaluation functions may be desirable. For this discussion, the UDD provides sufficient power of expression.

Before proceeding to a discussion of design quality measurement, a significant limitation of the present formalism should be noted. Although several design media deal with concurrency, the design diagraph structure can be used to express only sequential control flow. It is planned to extend this work to concurrent system models (such as Petri net or other graph models(HOLT70)), since many applications suggest designs which employ concurrency.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1981 ACM 0-89791-038-9 /81/0003/0089 $00.75

3. Software Science Metrics

Software science is a branch of experimental and theoretical science dealing with the analysis of computer programs and other types of written material (HALS77). Although lacking a firm mathematical foundation, experimental evidence suggests that the application of software science to computer software production provides useful indicators of software quality. Unfortunately, most efforts to date support its utility only at the code level of software representation.

In this section, software science is briefly outlined and extended to the design level of software representation.

3.1 Software Science Background.

Software science (HALS77), formerly called software physics (but not to be confused with Kolence's software physics) (KOLE72) was developed by M.H. Halstead to fill the need for a theory which would provide quantitative and objective measurements of software quality and complexity. Software science deals with those properties of algorithms that can be measured, particularly relationships that remain invariant under translation from one language to another.

Drawing on intuitive notions from information theory and the laws of thermodynamics, this theory is based on the measurement of four fundamental parameters that are directly available from the language used to express the algorithm.

1. INTRODUCTION

Software quality has many attributes, some of which are amenable to static quality measurement. Simplicity, as a function of human understandability, is one that has been identified as measureable. Software development methodologies comprised of tools, techniques, and standards that provide and environment for software production, require an analysis technique that can measure software simplicity. Early visibility of software quality, particularly in the design phase

of software development, would provide both designers and managers confidence of a quality end product. Unfortunately, most quality metrics to date focus on the quality of computer code (HALS77, MCCA76, MCCA77, SULL75).

There is a need to develop quality metrics specifically oriented to design specification media. The distinction between a design specification medium and a programming language is primarily one of intent. The objective of a design medium os to allow representation of the overall structure of a system, without at the same time introducing a forest of details. Design specification media (e.g., HIPO (STAY76), PSL (TEIC74), and DARTS (CSDL80)) generally present a more explicit portrayal of control flow, data flow, modularization than do programming languages.

In this paper (which is based on a study conducted for the National Bureau of Standards (SZUL80)), measuring the quality of software designs is emphasized. Software science has been shown to be an effective software quality indicator, yet most work with this technique has been at the code level of software representation. This work applies software science to software design media and proposes an identification and counting method for software science parameters. Two functionally equivalent but alternative designs for a simple example are presented in a generalized design media representation, and an analysis of design quality is made.

2. A Design Representation Medium

In seeking measurement techniques to quantify design quality, one is immediately confronted with the diversity of design media. The following discussion is based on a generalized formalism called an uninterpreted design digraph[1] (UDD) in which control and data flow can be expressed. Most of the media currently in use are compatible with the UDD formalism.

An uninterpreted design digraph is defined to be a 4-tuple

$$G = \langle N, E, D, M \rangle$$

where

N = a finite set of nodes which includes a unique initial node, s, and a unique terminal node, t.

$E \subseteq N \times N$
= a set of directed edges joining nodes of N.

D = a finate set of variable names

M = (INPUT:$N \quad 2^D$, Output:$N \quad 2^D$).

The nodes of a UDD represent uninterpreted data transformations and control decisions. The edges impose a control structure on the transformations represented by the nodes (As in a flowchart or flowgraph). Nodes have any in-degree (number of input edges) or out-degree (number of output edges), except that there is only one node of zero out-degree (the terminal node). A node having an out-degree of two or more is said to be a decision node. Other nodes are referred to as functional nodes. The initial and terminal nodes, s and t, are present only for notational convenience and are neither functional nor decision nodes (they may be viewed as the system's environment).

The graph may have loops but, by definition, may not have parallel edges. The functions INPUT and OUTPUT are used to associate data names with the input and output item sets of each node.

Each functional node i represents an uninterpreted data transformation that can be expressed as

$$f_i(\text{Input}(i)) = \text{Output}(i)$$

Each decision or branch node j represents an uninterpreted predicate evaluation function b_j (Input (j)) that directs the flow of control through exactly one of its out-edges. The Output set of a decision node is always null. The Output set of the initial node s is defined to contain inputs to the digraph. The Input set of the terminal node t contains the outputs of the digraph.

1. n_1 number of unique operators
2. n_2 number of unique operands
3. N_1 total number of operators
4. N_2 total number of operands.

Other parameters, which are derivable from these basic quantities, are shown in Table 1.

N	LENGTH	$N=N_1+N_2$.
$\hat{N}$	LENGTH ESTIMATOR	$\hat{N}=\eta_1\log_2\eta_1+\eta_2\log_2\eta_2$.
n	VOCABULARY SIZE	$\eta=\eta_1+\eta_2$.
V	VOLUME	$V=N\log_2\eta$.
V*	POTENTIAL VOLUME	$V^*=\eta^*\log_2\eta^*$.
L	LEVEL OF ABSTRACTION	$L=V^*/V$.
E	EFFORT	$E=V/L$
λ	LANGUAGE LEVEL	$\eta=V^*L$

Table 1: Software Science Equations

Halstead observed a relationship between the vocabulary size η and the algorithm length N such that an estimate of the length $\hat{N}$ could be made knowing only the algorithm's vocabulary. This

[1] A digraph is a directed graph.

relationship, the length equation, is defined as

$$N = \eta_1 \log \eta_1 + \eta_2 \log \eta_2.$$

Although no mathematical justification for this relationship is known, it has been validated for a number of programming languages (ELSH76). The accuracy of this estimator is, however, dependent on the purity of the algorithm. Halstead defined six classes of algorithm impurities, which address specific flaws in programming style. When present, impurities cause discrepancies between the observed and predicted lengths N and $\hat{N}$.

The volume V can intuitively be related to the number of bits required to encode the algorithm. In this context, algorithms specified in more abstract languages (e.g., design languages) occupy less volume than those specified in less abstract languages (e.g., assembler). Halstead hypothesized a conservation law between level of abstraction and volume such that: LV = constant. This interesting result allows alternative implementations to be compared, even is the level of abstraction (representation) differs between implementations. The potential volume V* is defined informally as a measure of the algorithm's most succinct form. In this form, the required function is reduced to a single operation. In translations from one language to another, actual volume V may change, but the potential volume V* does not. To find the potential volume, it is necessary to consider the potential vocabulary η^*.

The potential vocabulary η^* connotes the number of operators and operands in the algorithm's minimal form. The minimum number of operators η_1^*, for any algorithm reduced to a single statement, is $\eta_1^*=2$ (i.e., function and assignment operators). The minimum number of operands η_2^* is just the number of input and output parameters.

In order to predict the volume at which an algoritm is implemented in a given language, it is necessary to determine the language level λ. The language level λ measures the ability of the language to express algorithms. This number has been obtained experimentally for a number of different languages (HALS77), although the observed variances are large.

As a measure of the mental effort required to create a computer program, Halstead introduced the effort metric E. This number represents the number of elementary decisions that an experienced programmer would be expected to make in constructing the program. It has been used as a measure of psychological complexity (CURT78). When coupled with the Stroud Number, a psychological concept defining the time required by a human brain to make an elementary decision, Halstead was able to provide programming time estimates.

3.2 Using Software Science To Assess The Quality Of Software Design

The software science relationships found by Halstead and others appear to be independent of any particular implementation language. Extensions to natural language (HALS77A) and technical writing (COME79) provide some evidence that the relationships may be valid over a wide range of languages. Since early assessment of software quality, prior to code generation, is particularly advantageous, this section explores the extension of software science to design media.

In order to compute software science metrics prior to coding it is necessary to identify and count the operators and operands in the design medium. In code media, despite the apparent simplicity associated with counting these parameters (since code when abstracted to the machine level is only operators and operands), counting methods can vary for the same language. Elshoff (ELSH78) reported that when different counting methods were employed, some properties of the algorithm varied, while others remained stable, and no single method could be shown to be best.

To illustrate a generalized technique for counting operators and operands in a design medium, it is assumed that the design can be translated to the design digraph structure of Section 2. Rules can be specified for identifying and counting operators and operands for the design digraph.

The assignment of values to η_1, η_2, N_1, N_2 requires the identification of operators and operands and the adoption of conventions for counting their occurrences. The definitions of the operand and operator sets and the four software science quantities are presented in Table 2 and are explained as follows.

The operators in a design digraph are uninterpreted transformations, uninterpreted predicate evaluation functions, an assignment operator, and a flow-control operator. Each functional node i of a design digraph represents a unique transformation

$$f_i(\text{Input}(i)) = (\text{Output}(i)).$$

The set $F \subseteq$ Operators consists of all such transformations f. For each functional node i, f and the assignment operator "= " represent two operator occurrences which contribute a count of two to N_1.

Each decision node j of a digraph represents a unique presicate evaluation function $b_j(\text{Input}(j))$.

The set $B \subseteq$ Operators consists of all such functions. Each decision node contributes a count of one to N_1.

$$\text{OPERATORS} = F \cup B \cup \{CTL, =\}$$

$$\eta_1 = |\text{OPERATORS}|$$

$$N_1 = 2(|F|) + |B| + (|E| - g)$$

where

g = out-degree(s) + in-degree(t)

$$\text{OPERANDS} = \sum_{n \epsilon N} (\text{INPUT}(n) \cup \text{OUTPUT}(n))$$

$$\eta_2 = |\text{OPERANDS}|$$

$$N_2 = \sum_{n \epsilon N} |\text{INPUT}(n)| + \sum_{n \epsilon N} |\text{OUTPUT}(n)|$$

Table 2: Software Science Identification and Counting Definitions

The members of a digraph's edge set E define the flow of control over the nodes. Flow control is considered to be affected by an operator which is denoted CTL. The number of occurrences of CTL is defined to be the number of digraph edges which do not originate at the start node s or terminate at the terminal node t (the nodes s and t were added to the node set for notational convenience, and edges involving them are superflouous to the design).

The operands in a design digraph are those data items which are input to or output from the nodes. The operand count is best described by the definition in Table 2.

The use of this identification and counting method is illustrated by the example in Section 4.

4. Application Of Software Science To Design Media

To illustrate the ideas presented in the previous sections, the software science metrics will be applied to two example designs. The designs represent alternative solutions to the "stores movement summary" problem discussed by Jackson (JACK75). The software is intended to process the product distribution file of an inventory control system and provide a report summarizing product movement. As Jackson demonstrates, the structure of the input data and by subjective assessment, design 1 is of lower quality than design 2. The designs were translated from the DARTS design specification medium (CSDL80) into UDD form. The structure of the UDD for design 1 is shown in Figure 1 and the UDD for design 2 is similarly displayed in Figure 2.

To apply the software science metrics described in Section 3 to the design digraph representation, it is necessary to

1. identify operators and operands in the design.

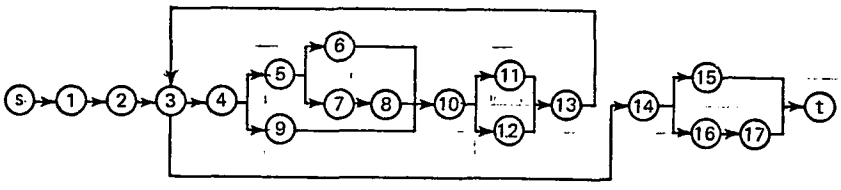

Figure 1: UDD for Design 1

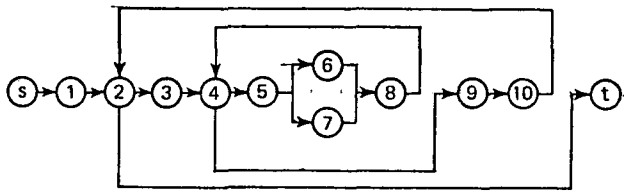

Figure 2. UDD for Design 2

2. count the number of occurrences of each operator and operand
3. calculate the metrics based on the formulas listed in Table 1.

Identifying and counting operators in the design digraphs of the "stores movement summary" example (see Figure 1 and 2) is simply a matter of counting edges, functional nodes, and decision nodes. Since digraph functions are not interpreted, each functional node and each decision node are assumed to be unique.

To identify and count operands, The data-flow tables available from the DARTS medium are examined and are summarized in Table 3. In table 4, raw counts of operators and operands are shown for designs 1 and 2. The minimum number of unique operands, η_2^* (see Section 3), is observed to be 3. This value is obtained by counting external input and output variables (identified by the prefix #). Table 5 summarizes the remaining software science calculations. The important complexity indicators (length, volume, and effort) suggest that design 2 is less complex than design 1. This result agrees with Jackson's analysis based on a simplicity argument.

Node	Inputs	Outputs
s		#MVTREC
1	FIRSTTIME	
2	#MVTREC	PN, ATYPE, Q, EOF
3	EOF	
4	FIRSTTIME, PARTNO, PN	
5	FIRSTTIME	
6		FIRSTTIME
7	PARTNO	#PARTNO
8	NETMVT	#NETMVT
9	PN	PARTNO, NETMVT
10	ATYPE	
11	NETMVT, Q	NETMVT
12	NETMVT, Q	NETMVT
13	#MVTREC	PN, ATYPE, Q, EOF
14	FIRSTTIME	
15		
16	PARTNO	#PARTNO
17	NETMVT	#NETMVT
t	#NETMVT, #PARTNO	

Node	Inputs	Outputs
S		#MVTREC
1	#MVTREC	PN, ATYPE, Q, EOF
2	EOF	
3	PN	NETMVT, PARTNO
4	PN, PARTNO, EOF	
5	ATYPE	
6	NETMVT, Q	NETMVT
7	NETMVT, Q	NETMVT
8	#MVTREC	PN, ATYPE, Q, EOF
9	PARTNO	#PARTNO
10	NETMVT	#NETMVT
t	#NETMVT, #PARTNO	

Table 3: Input/Output Table for Designs 1 and 2

DESIGN 1				DESIGN 2			
OPERATORS	NO.	OPERANDS	NO.	OPERATORS	NO.	OPERANDS	NO.
CTL	20	FIRSTIME	5	CTL	12		
=	12	# MVTREC	3	=	7	# MTREC	3
f_1	1	EOF	3	f_1	1	EOF	4
f_2	1	PARTNO	4	b_2	1	PARTNO	3
b_3	1	PN	3	f_3	1	PN	3
b_4	1	NETMVT	6	b_4	1	NETMVT	6
b_5	1	ATYPE	3	b_5	1	ATYPE	3
f_6	1	Q	4	f_6	1	Q	4
f_7	1	# NETMVT	3	f_7	1	# NETMVT	2
f_8	1	# PARTNO	3	f_8	1	# PARTNO	2
f_9	1			f_9	1		
b_{10}	1			f_{10}	1		
f_{11}	1						
f_{12}	1						
f_{13}	1						
b_{14}	1						
f_{15}	1						
f_{16}	1						
f_{17}	1						
(η_2=19)	(N_1=48)	η_2=10	N_2=37	(η_1=12)	(N_1=29)	η_2=9	N_2=30

Table 4: Operator and Operand Counts for Designs 1 and 2

	DESIGN 1	DESIGN 2
η_1	19	12
η_2	10	9
η	29	21
η_1^*	2	2
η_2^*	3	3
η^*	5	5
N_1	48	29
N_2	37	30
N	85	59
$\hat{N}$	113.9	71.5
$N-\hat{N}$	28.9	12.5
$\frac{N-\hat{N}}{N}$	0.34	0.21
V	412	259
V^*	11.6	11.6
L	.028	.048
λ	.326	.578
E	14686	5784

Table 5: Summary of Software Science Metrics for Designs 1 and 2

5. Conclusions

The application of the software science metrics to software designs has produced evidence that such metrics can provide designers with useful feedback during system development. Prior to commitment to code, alternative designs can be compared, using the proposed technique, to assess the relative static quality of the designs with respect to simplicity, or alternatively their lack of complexity. This technique is general, in that it compensates for the level of design abstraction, yielding a tool that can be used throughout the software development effort. For this experiment, manual methods were used to identify and count operators and operands in a design medium, although automation of this method to complement modern design aid tools is possible. More experience with this quality assessment technique is needed to validate its utility and provide insight into its limitations.

6. References

(COME79) Comer, D., and M.H. Halstead, "A Simple Experiment in Top-Down Design," IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, March 1979, pp. 105-109.

(CURT78) Curtis, B., S.B. Sheppard, M.A. Borst, P. Milliman, and T. Love, "Some Distinctions Between the Psychological and Computational Complexity of Software," Proceedings of the 2nd Software Life Cycle Management Workshop, August 1978, pp.166-170.

(CSDL80) "Design Aids for Real-Time Systems (DARTS): A Designer's Manual "Preliminary, The Charles Stark Draper Laboratory, Inc., Cambridge, MA, January 1980.

(ELSH78) Elshoff, J.L., "An Investigation into the Effects of the Counting Method Used on Software Science Measurements," SIGPLAN Notices, Vol. 13, no. 2, February 1978, pp. 30-45.

(HALS77) Halstead, M.H., Elements of Software Science, Elsevier North-Holland, Inc., New York, 1977.

(HALS77A) Halstead, M.H., "A Quantitative Connection Between Computer Programs and Technical Prose," Proceedings of Fall COMPCON 1977, pp. 332-335.

(HOLT70) Holt, A.W., and F.G. Commoner, "Events and Conditions," Applied Data Research, Inc., New York, 1970.

(JACK75) Jackson, M.A., Principles of Program Design, Academic Press, New York, 1975.

(KOLE72) Kolence, K.W., "Software Physics and Computer Performance Measurements, "Proceedings of the ACM 1972 Annual Conference, pp. 1024-1040.

(MCCA76) McCabe, T.J., "A Complexity Measure," IEEE Trans. on Software Engineering, Vol. SE-2, NO. 4, December 1976, pp. 308-319.

(MCCA77) McCabe, T.J., et al., "Factors in Software Quality," RADC-TR-77-357, Vol. I, II, and III (AD-A049-014, -015, -055), General Electric Company, Sunnyvale, CA, 1977.

(STAY76) Stay, J.F., "HIPO and Integrated Programs Design," IBM Systems Journal, Vol. 15, No. 2, 1976.

(SULL75) Sullivan, J.E., "Measuring the Complexity of Computer Software," MTR-2648, Vol. V, The MITRE Corporation, Bedford, M.A., January 1975.

(SZUL80) Szulewski, P.A., M.H. Whitworth, P. Buchan, and J.B. DeWolf, "Quality Assurance Guidelines and Quality Metrics for Embedded Real-Time Software Designs," NBS Contract NB76SBCA0220, Report R-1376, The Charles Stark Draper Laboratory, Inc., Cambridge, MA., May 1980.

(TEIC74) Teichroew, E., M.J. Bastarache, and E.A. Hershey III, "An Introduction to PSL/PSA." ISDOS Working Paper No. 86, University of Michigan, Ann Arbor, Michigan, March 1974,