Science and Technology, 4 (1999) 45-53

Applying Halstead's Metric to Oberon Language

Fawaz Ahmad Masoud

Department of Computer Science, College of Science, Sultan Qaboos University, P.O.Box 36, Al-Khod 123, Muscat, Sultanate of Oman. Email: fawaz@squ.edu.om.

تطبيق مقياس هالستد على لغة أوبرون

فواز أحمد مسعود

خلاصة : لغة أوبرون من اللغات البسيطة والتي يسهل فهمها واستعمالها . إن هذه اللغة تؤيد نمطين من لغات الحاسب - نمط اللغات الإجرائية ونمط اللغات الموجهة الهدف . إن تطبيق مقياس هالستيد على لغة أوبرون يجعل بالإمكان تحليل وقياس صعوبة صيانة البرامج المكتوبة بهذه اللغة . وهذا النوع من التحليل يزود المبرمج والمدير بمعلومات كافية عن مدى الصيانة اللازمة للبرامج المكتوبة بهذه اللغة . ويتم قياس هذه الصعوبة عن طريق عد مفردات جمل اللغة المستعملة في البرنامج المكتوب . لقد تمت عملية العد هذه باستخدام برنامج كتب أعد بلغة خاصة من أجل هذا الغرض ولقد تم تحليل ومناقشة النتائج بالتفصيل .

ABSTRACT: Oberon is a small, simple and difficult programming language. The guiding principle of Oberon was a quote from Albert Einstein: "Make it as simple as possible, but not simpler". Oberon language is based on few fundamental concepts that are easy to understand and use. It supports two programming paradigms: the procedural paradigm, and the object-oriented paradigm. This paper provides the application of Halstead's software science theory to Oberon programs. Applying Halstead's metric to the Oberon language has provided the analysis and measurements for module and within module maintenance complexity of programs written in Oberon. This type of analysis provides a manager or programmer with enough information about the maintenance complexity of the Oberon programs. So they can be aware of how much effort they need to maintain a certain Oberon program. The maintenance complexity of the programs written in Oberon or any other language is based on counting the number of operators and operands within the statements of the tested program. The counting process is accomplished by a program written in C language. Results are obtained, analyzed, and discussed in detail.

Oberon is a general-purpose programming language, the successor of Pascal and Modula-2, developed by Niklaus Wirth in 1985-1988. The Oberon was purposely designed to serve as an implementation tool for the Oberon Operating System. Reiser, *et al* (1992) stated that the Oberon language is, however, not tied to the Oberon operating system. An Oberon program, like any other computer program, consists of statements in which a statement may have many different forms. Ideally, the Oberon source program is made up of operators and operands. Counting the number of operators and operands with a frequency measure of each one provides an idea of how large the program is? It gives also an indication of the most important or heavily used operands. This type of analysis provides a manager or programmer with good indication about the maintenance complexity of the Oberon programs, so they can be aware of how much effort they need to maintain a certain Oberon program. The next section provides an overview of the theory of software science. Section 3 provides the related software science equations to this study. The application of the theory is then discussed in section 4, particularly when the theory is applied to Oberon. This section also discusses the problems, which occur during the process of counting the operators and operands. Finally, the results are obtained, analyzed, and discussed in detail in section 5.

Overview of the Software Science

Halstead (1972) is the first to suggest the use of the lexical analysis in the theory of software science. The basic idea of Halstead's theory is that a program in a language can be thought of as being composed of operands and operators. In a given program, we can count the number of unique or distinct operators, the number of unique or distinct operands, the total usage of all operators, and the total usage of all operands. Halstead drew an analogy with thermodynamics, where one can determine a number of physical properties of a pure substance simply from knowledge of its pressure and volume. That means we can infer properties and characteristics from knowledge of their operators, operands and their

usage frequencies of programs. Program operators can be divided into three groups:

1. Fundamental operators such as; +, -, /, *, =, >, <, <>, AND, OR, .NE.,.etc.

2. key words operators such as; IF, THEN, DO, GOTO, END, WHILE, ...etc.

3. Specific operators such as: names of procedures, functions, subroutines and entry points and tasks activation, etc.

The software science theory has attracted enormous interest since its first appearance. One reason for this interest may be the apparent success of many early experiments with this theory. For example, it has been claimed that software science theory can accurately predict programming time and the mean number of bugs in a piece of software. The number of bugs gives an indication of maintenance required for the software. Fitzsimmons and Love (1978) have provided an interesting review of these early results.

Early experiments with software science theory were applied to programs written in CDC Assembly language and Fortran. The high correlation between predicted and observed values prompted further study, and a special issue of IEEE Transactions on Software Engineering was devoted to the subject (Yeh, 1979). However, Woodward, (1984) reported difficulties in applying the science theory to ALGOL 68 programs. It is probably true to say that software science theory represents the wrong level of detail for most measurement applications. This paper will add more evidence to the continuing debate by reporting the results of applying the theory to Oberon programs.

Software Science Theory

Software science theory is concerned with algorithms and their implementations as computer programs. As an experimental science, it deals only with those properties of algorithms that can be measured either directly or indirectly, statically or dynamically, and with relationships among those properties that remain invariant(unchanged) under translation from one language to another.

Halstead, (1977) argued that algorithms and their implementations in programming languages consist of operators and operands and of nothing else. Woodward, (1984) provides a simple view of algorithms and their implementation in programming languages. This view can be easily verified by considering a simple digital computer whose instruction format consists of only two parts: an operator code (instruction) and one or more operand addresses. Computer programs must be translated (compiled) to the level of computer instructions before being executed. That means, every part of the program would be either an operator or an operand. A higher-level language view of operands and operators would be as follows:

- Operands are the variables, subprogram calls, generic instantiation etc., and constants of the program,
- Operators are the symbols and statements that affect the values or ordering of the operands.

The software science theory provides the following variables definitions:

- n_1 the number of distinct operators in the specified program.
- n_2 the number of distinct operands in the specified program.
- N_1 the total number of occurrence of operators.
- N_2 the total number of occurrence of operands.

From the above variables Halstead, (1977) defined many measures for computer programs such as, the vocabulary size n and the program length N that can be obtained using the following equations:

- $n = n_1 + n_2$
- $N = N_1 + N_2$

Then Halstead, (1972) argued that the formula:

- $N^{\wedge} = (n_1 * \text{Log}_2 n_1) + (n_2 * \text{Log}_2 n_2)$

would provide a reasonable estimate of the program length N for what he called a pure program. In fact he proved this estimate to fall within 10% of the real length N (only for pure programs). Further more, Halstead, (1977) identified six classes of impurity which might cause the estimated length N^ to deviate from the observed length N. The six impurities are given below:

1. Complementary operations: the successive application of complementary operators with one canceling the effect of the other as when saying (not (not (true))).
2. Ambiguous operands: the use of an operand for more than one purpose.
3. synonymous operands: the use of more than one identifier for the same object.
4. common sub-expressions: the use of the same sub-expression more than once.
5. Unwarranted assignment: the assignment of a name to a sub-expression, which is used only once.
6. unfactored expression: the use of an expression, which can be put more succinctly.

A number of definitions and procedures were devised to measure quantities for programs and algorithms. Some of these definitions are given below:

1. program volume (V) = $N*Log_2 n$.
2. program level (L) = V*/V, where V* is the potential volume (the volume of the minimal size implementation of a program).
3. language level (LL) = L* V* = $((L)^2)*V$.
4. programmer's effort(E) = V/L=$(V)^2/V*$.
5. program's intelligence content(I) = L^ *V, where L is an estimated program level.

The following discussion seemed to be adequate to mention here as we are discussing Halstead's own formulas:

1. The volume V of an algorithm program should decrease as the language implementation grows from low level languages such as Assembly to higher level languages such as FORTRAN, COBOL, ALGOL, Pascal, C, Oberon etc. This is because n and N decrease since low level languages require a great amount of detail to say the same thing than do in high level languages.

2. , The potential volume of the most compact algorithm (V*) which corresponds to a single procedure call (with appropriate passed parameters) can be determined

From: $V^* = N^ * Log_2 n^ = (N^_1 + N^_2)*Log_2(n^_1 + n^_2)$, where $n^_1$ and $n^_2$ are the numbers of unique operators and operands, $N^_1$ and $N^_2$ are the estimates of the total numbers of occurrence of operators and operands. It is, in some circumstances, difficult to determine V* directly. For this reason, the following formula can provide an estimate L^ for the program level:

$L^ = 2* n_2/(n_1 * N_2)$.

Many researchers have used this estimated level L^ in place of the previously defined quantity L. In particular, the language level has frequently been calculated using L^ in place of L. In this paper, LL^ will be used to denote this estimated value of language level. In other words:

$LL^ = ((L^)^2)*V$

The equation $E = V^2/V^*$ quantifies the programming effort, which is defined as the mental activity required to reduce and translate a preconceived algorithm to an actual implementation in a language in which the implementer is fluent and knowledgeable. Halstead tried to give a quantity to this measure by using primitive metrics so he came up with the above equation of effort in which he related the program volume and the program level. This relation sounds reasonable enough because as the program volume increases the more effort it needs to be implemented and debugged, but this is only valid for the kind of effort defined previously. However, there are kind of efforts that cannot be measured by the effort equation. These are things such as the mental effort by the programmer to produce an algorithm for the first time. The main concern of this paper is to discuss existing software science theory as it is, and not to enhance it. Therefore, the discussion will continue only its use of effort as defined by Halstead. Finally, the equation: I = L^*V
Defines the intelligence level of a program. This equation relates the program level to the program volume. From the above relation, it seems that the intelligence increases as both or either of the volume and/or the level increases.

APPLYING SOFTWARE SCIENCE THEORY TO THE OBERON LANGUAGE: It is known that Halstead's formulae are strongly based on counting the number of operators and number of operands in a given program or algorithm. Knowing the

importance of such counts Halstead gave a fair definition of what constitutes operators and operands. In fact, he gave some suggested counting rules for some languages such as Fortran, CDC Assembly, etc. Nevertheless, when applying the above simple definition of problems and operands to Oberon programs, many problems arise. These problems are some constructs of Oberon which give ambiguous counting results. The ambiguous language constructs are given below:

PROCEDURE DEFINITION AND CALLS: A procedure in Oberon is a group of statements that have a name and may be invoked from other locations in a program. Formally, the syntax of a procedure in Oberon is given by :

```
ProcedureDeclaration=
    ProcedureHeading";"ProcedureBody identifiers
ProcedureHeading=
    "PROCEDURE"ident.["*"][FormalParameters]
ProcedureBody= DelarationSequence
        ["BEGIN" StatementSequence]
        "END".
```

Example:

```
PROCEDURE SqRoot(a,b,c:REAL;VAR r1,r2,i1,i2:REAL);
VAR det:REAL;
BEGIN
b:=b/2;det:=b*b-a*c;
IF det >= 0 THEN (* real roots *)
r1:=(ABS(b)+sqrt(det))/a;
IF b>=0 THEN r1:=-r1 END;
    r2:=c/(a*r1); i1:=0; i2:=0
ELSE
    r1:=-b/a;r2=r1;i1:=sqrt(-det);i2:=-i1
END
END SqRoot;
```

When counting a procedure name at the time of its definition it should be counted as an operand according to the operands simple definition since "PROCEDURE" is an Oberon keyword (operator) whose action falls upon the procedure name. So, the procedure name is an operand of "PROCEDURE", but when a procedure is called from the program it is called by just writing its name and its actual parameters. Formally, the syntax of the procedure call in an Oberon language is given by :

```
ProcedureCall= designator[ActualParameters].
```

Example of procedure calls include:

```
ReadInt(i)
WriteInt(j*2+1,6)
INC(w[k].count)
```

It is clear that counting the procedure name as an operand at its definition and as an operator at its call causes an ambiguity.

Oberon Module

An Oberon module is a collection of declarations of constants, types, variables, and procedures, and a sequence of statements for the purpose of assigning initial values to the variables. A module typically constitutes a text that can be compiled as a unit. Like procedures, modules have a type and they store values of the same type as they are defined in. This property can qualify modules to be in the same level of variables (storing values) and variables are always counted as operands. Further, modules could have parameters and they can change the values of their parameters. Another ambiguity is that, module names have the same ambiguity count as procedures at the time of their definition. Oberon modules observe the syntax:

```
module = "MODULE" identifiers ";"
         [ImportList]
         DeclarationSequence
         ["BEGIN"StatementSequence]
         "END" identifiers".".
```

Example:

```
Module IFS;
VAR
a1,b1,c1:REAL;
a2,b2,c2:REAL;
PROCEDURE Draw
...(*Procedure body*)
END Draw;
BEGIN
:
...(*Statement sequence*)
ENDIFS.
```

The module names are considered as operands of the operator keyword "Module".

The solution to the above ambiguity is counting the module name according to the way it occurs in the program, that is:

1. module names are counted as operands at the time of their definition.
2. module names are counted as operands when they are used as ordinary variables.
3. module names are counted as operators when they have a parameter list and used as procedure calls.

Declaration of Variables and Constants

Every identifier occurring in an Oberon program must be introduced by a declaration, unless it is a predefined identifier. For example, variable declarations in Oberon observe the following syntax:

VariableDeclaration = identlist":"type.

Examples of variable declarations include:

```
i, j, k:   INTEGER
x, y:      REAL
p, q:          BOOLEAN
s:         SET
F:         Function
a:         ARRAY OF 50 OF REAL
w:         ARRAY OF 15 OF
               RECORD ch:CHAR;
                          count:INTEGER
               END
t:         Tree
```

The above language construct has no change made against the values of the given variables. So, some researchers do not count the declaration sections while others do count declarations on the basis that they are there in the program and the primary rule for counting is to count the items of the program.

In this paper, we support counting constant declarations since the constant section in Oberon is defined as follows:

CONST [constant-name := value];

Example:

CONST x = 5; y = 100;

The above definition looks like an ordinary assignment statement with both constant value and name as operands and the equal sign as the operator. Therefore, this is a good support for counting the constant declarations. The suggested count for the above example is as follows:

1. declaration section is counted with variables considered as operands and predefined types as operators.
2. constant declarations (names and their values) are counted as operands of the operator "=".

Other Language Constructs

Each language has its own reserved words (keywords). These keywords are classified for the purpose of counting into two parts:

1. simple keywords: which are counted as single operators.
2. compound keywords, which consist of more than one keyword.

The compound keywords include:

WHILE...DO
REPEAT...UNTIL
BEGIN...END
CASE...END
RECORD...END
WITH...END
LOOP ...END
RETURN
EXIT
IF...THEN
IF...THEN...ELSE

Some components of the above compound keywords are optional. For example,

IF...THEN..ELSE
and
IF...THEN

Therefore, a problem arises here with the optional component when exists and when does not exist. To solve this problem, we have the following suggestions:

1. the above compound keywords are counted as one occurrence.
2. if the compound keyword has an optional component it is counted as two different keyword occurrences when the optional part is omitted and when it exists. For example, IF...THEN is one keyword and IF...THEN...ELSE is another.

With this point we conclude the discussion of designing a counting scheme for Oberon. The complete design is written in the appendix (1).

Analysis and Discussion of the Results

The counting scheme discussed in section 4 (see appendix 1) was used together with a counting tool program that was specially designed for this purpose to apply the theory to selected programs From Videki, .(1989-1991). The results obtained from the above raw data are presented in Table (1) below which shows for every module the number of unique operators (n_1) and unique operands (n_2) and total occurrences of operators (N_1) and operands (N_2). The quantities with an asterisk are corresponding counts for calls of the procedures, which are included in the selected programs.

TABLE 1

Module	n_1	n_2	N_1	N_2	n_1*	n_2*	N_1*	N_2*
M1	11	36	76	54	4	4	5	4
M2	8	40	64	47	4	4	5	4
M3	7	36	55	41	4	6	7	6
M4	8	32	54	39	4	4	5	4
M5	10	40	63	43	2	2	2	2
M6	10	51	91	55	4	3	5	3
M7	8	47	67	49	4	4	5	4
M8	9	42	68	42	3	2	3	2
M9	7	27	41	29	4	2	4	2
M10	18	85	141	108	0	0	0	0
M11	10	37	52	38	6	7	10	7
M12	14	70	106	82	2	2	2	2
M13	10	29	61	31	0	0	0	0
M14	8	72	104	73	0	0	0	0

Then Table (2) shows the derived software science quantities using the same counting scheme, specifically the observed and estimated program length (N and N^), observed and estimated program level (L and L^), observed and estimated language level (LL and L L^) for the selected modules.

TABLE 2

Module	N	N^	L	L^	LL	LL^
M1	130	155.0	.03739	.1212	.6997	7.352
M2	111	164.2	.04355	.2128	.8149	19.45
M3	96	142.6	.8292	.2508	2.482	22.71
M4	93	127.5	.05455	.2051	1.021	14.44
M5	106	170.6	.01337	.1861	.0742	14.36
M6	146	223.6	.02594	.1855	.4037	20.65
M7	116	197.6	.04025	.2398	.7532	26.73
M8	110	176.8	.01861	.2222	.1497	21.36
M9	70	102.6	.04354	.2666	.4680	17.47
M10	249	481.7	0.0000	.0875	0.000	8.835
M11	90	156.6	.12584	.1947	5.487	13.13
M12	188	334.3	.00667	.1219	.0370	12.37
M13	92	120.7	0.0000	.1871	0.000	11.80
M14	177	324.6	0.0000	.2466	0.000	47.16

Results of Table (2) are used to calculate the Pearson correlation coefficient between observed and estimated values for program length, program level, and language level. The Pearson Correlation Coefficient is a measure of linear association between 2 variables. Values of the correlation coefficient (r) range from -1 to +1. The absolute value of the correlation coefficient indicates the strength of the linear relationship between the variables, with larger absolute values indicating stronger relationships. The sign of the coefficient indicates the direction of the relationship. The Pearson Correlation Coefficient is calculated from two variables (X and Y), usually with interval or ratio level data. Each variable is assigned a score based on its distance from the mean and these scores are then cross-multiplied for each subject, and then summed.

The results of these calculations are summarized in Table (3) below. It should be noted that, since language level LL is calculated using L and estimated language level LL^ is calculated using L^, the correlation between L and L^ is not independent of that between LL and L L^. Hamer *et al*, (1981) have warned of pitfalls in using the Pearson correlation coefficient to confirm a relation between two variables. However, it has become the (de-facto) standard practice in work on software science theory and so the results have been given here without detailed comment.

TABLE 3

Language	
correlation of N and N^	0.9782
correlation of L and L^	00.2922
correlation of LL and L L^	00.0998
minimum LL	00.0000
maximum LL	05.4870
mean LL	00.88
minimum L L^	06.96
maximum L L^	47.16
mean L L^	18.41

Halstead (1977) published the list of language levels shown in table (4), which appears to confirm the intuitive ordering from high to low level languages. The results of the experiments reported here depart radically from what might have been anticipated, putting Oberon (with LL = 0.88) equal to assembly language in Halstead's table of language level irrespective of which counting scheme for Oberon is considered.

TABLE 4

Language	Language Level
English	2.16
PL/I	1.53
Fortran	1.14
PILOT	.92
Assembly (CDC)	.88

Summary and Conclusion

The important results of this study are the following:

1. The counting scheme given by Halstead, (1977) needs further investigation.
2. The programs are pure and well structured or at least do not exploit the high Level nature of Oberon.
3. The theory of software science as it is does not apply to Oberon.
4. Undue significance is being attached to quantities with high standard deviation.

With regard to the first point, there are clearly many reasonable modifications one could make to the counting scheme employed here, although it is doubtful whether this itself would change the situation much. Elshoff, J. L. (1978) has performed an extensive set of experiments with PL/I programs. In fact he has used eight different counting strategies to determine the effect on the calculated software science properties. He discovered that, although some measures vary considerably, depending on the counting scheme used, other measures, which include the language level, do not alter significantly. In particular his results for L appear to be broadly in agreement with the language level of 1.53 for PL/I quoted by Halstead. With regard to the second point we say that the tested modules are pure and they are very structured so they do reflect Oberon's high level nature. With regard to the third and fourth points, the theory of software science is doubtful and care should be taken when dealing with it.

References

ELSHOFF, J. L. 1978. An investigation into the effects of counting method used on software science measurement, *ACM Sigplan Notices*, **13 No.2.**

FITZSIMMONS, A. and LOVE, T. 1978. A review and evaluation of software science, *ACM, computing surveys* **10** No. 1.

HALSTEAD, M. H. 1972. Natural laws controlling algorithm structure, *ACM Sigplan Notices*, **7** N0. 2.

HALSTEAD M. H. 1977. *Elements of Software Science*, Elsevier-North Holland, Newyork.

HAMER, P.G. and FREWIN, G. D. 1981. M. A. HALSTEAD's Software Science a Critical Examination, ITT technical report no. STL 1314, STL Ltd. Harlow, Essex, U.K.

REISER, M. and WIRTH N. 1992. *Programming in Oberon, Steps beyond Pascal and Modula*, Wokingham: Addison-Wesley.
VIDEKI, E. R. 1989-1991. *Oberon-M(tm) version 1.2 for MSDOS.*
WOODWARD, M. R. 1984. The application of Halstead's software science theory to ALGOL68 programs, *Software Practice and Experience*, **14**, 263-276.
YEH, R. T. 1979. In the memory of Maurice H. Halstead, Editorial in commemorative issue in honour of Dr. Maurice H. Halstead, *IEEE Transaction on Software Engineering*, **5** No. 2.

Appendix 1 (The Software Science Counting Scheme for Oberon)

1. All program constructs should be considered for counting scheme such as: statement parts, program heading, and declaration parts.
2. Comments should be ignored.
3. Variables, constants, literals, file names, and the reserved word NIL are counted as operands. All operands are counted as if they were global in scope. In other words. local variables with the same name in different procedures arecounted as multiple occurrences of the same operand.
4. The following entities are always counted as single operator (* is not differentiated between set and arithmetic use):
   + - * / ~ & . , ; | ( ) [ ] { } Div Mod := ^ = # < > <= >= : .. ARRAY BEGIN CASE DO ELSE ELSIF END EXIT IF IMPORT IN IS LOOP MODULE OF OR POINTER TO PROCEDURE RECORD REPEAT RETURN THEN TYPE UNTIL VAR WHILE WITH
5. The following multiple entities are counted as single operator:
   BEGIN...END CASES...END WHILE...DO REPEAT...UNTIL IF...THEN IF.. THEN...ELSE FOR...DO WITH...DOSET OF FILE...OF RECORD...END ARRAY...OF
6. The following entities or pairs entities are counted as single operator subject to the accompanying conditions:
   VAR is counted as an operator in identifiers list and is not counted as a section label.
   - = is counted as either a relational operator in expressions or a efinition operator in non-executable sections of the program.
   - + is counted as either a uninary + or binary + depending on its module. The binary + is not differentiated between arithmetic and set usage.
   - - is counted as either a uninary - or binary - depending on its module. The binary - is not differentiated between arithmetic and set usage.
   - . is counted as either a record component selector symbol or a program terminator depending on its module.
   - : is a definition operator in the VAR section and the parameter lists. It is a separation operator following CASE or GOTO labels.
   - ( ) is counted as either an argument list operator or expression operator depending on its module.
   - [ ] is counted as either a subscript operator or a set operator depending on its module.
7. Procedure and module calls are counted as operators. The subprogram name following Module or PROCEDURE is not counted, though it actually is the operand for the Module or PROCEDURE operator.
8. GOTO statement (i.e. GOTO and accompanying label) are counted the operator GOTO and the operand label.
9. Declaration of labels are not enumerated (all tokens after the label operator through the next semi-colon inclusive a semi-colone) are ignored.
10. The following are syntactic devices and are not counted:
    CONST TYPE VAR(for variable section).
11. The following are rules pertaining to Oberon:
    VALUE is syntactic device and is not counted.
    Commas, ( ) and = in VALUE section are counted as in the TYPE section.
    OF in VALUE section is a syntactic device and is not counted.

Received 1 December 1997
Accepted 8 February 1999