SIMPLIFIED READABILITY METRICS

Chung Yung
Department of Information Systems
New York University
Leonard N. Stern School of Business
44 West 4th Street, Suite 9-170
New York, NY 10012-1126
yung@edgar.stern.nyu.edu

January 6, 1997

Working Paper Series
Stern #IS-97-1

Extended Abstract

This paper describes a new approach to measuring the complexity of software systems with considering their readability. Readability Metrics were first proposed by Chung and Yung [8] in 1990. Software industry uses software metrics to measure the complexity of software systems for software cost estimation, software development control, software assurance, software testing, and software maintenance [3], [7], [9], 15], [18]. Most of the software metrics measure the software complexity by one or more of the software attributes. We usually classify the software attributes that software metrics use for measuring complexity into three categories: size, control flow, and data flow [5], [7]. All the three categories concern with the physical activities of software development. Readability Metrics have been outstanding among the existing software complexity metrics for taking nonphysical software attributes, like readability, into considerations [8]. The applications of Readability Metrics are good in indicating the additional efforts required for less readable software systems, and help in keeping the software systems maintainable. However, the numerous metrics and the complicated formulas in the family usually make it tedious to apply Readability Metrics to large scale software systems. In this paper, we propose a simplified approach to Readability Metrics. We reduce the number of required measures and keep the considerations on software readability. We introduce our Readability model in a more formal way. The Readability Metrics preprocesses algorithm is developed with compilers front-end techniques. The experiment results show that this simplified approach has good predictive power in measuring software complexity with software readability, in addition to its ease of applying. The applications of Readability Metrics indicate the readability of software systems and help in keeping the source code readable and maintainable.

Simplified Readability Metrics

Chung Yung[*]

Department of Information Systems
Leonard N. Stern School of Business
New York University

Abstract

This paper describes a new approach to measuring the complexity of software systems with considering their readability. Readability Metrics were proposed by Chung and Yung [8] in 1990. Readability Metrics have been outstanding among the existing software complexity metrics for taking nonphysical software attributes, like readability, into considerations. The applications of Readability Metrics are good in indicating the additional efforts required for less readable software systems, and help in keeping the software systems maintainable. However, the numerous metrics and the complicated formulas in the family usually make it tedious to apply Readability Metrics to large scale software systems. In this paper, we propose a simplified approach to Readability Metrics. We reduce the number of required measures and keep the considerations on software readability. We introduce our Readability model in a more formal way. The Readability Metrics preprocesses algorithm is developed with compilers front-end techniques. The experiment results show that this simplified approach has good predictive power in measuring software complexity with software readability, in addition to its ease of applying. The applications of Readability Metrics indicate the readability of software systems and help in keeping the source code readable and maintainable.

1. Introduction

It is widely conceived that more time and money are spent on maintaining existing software systems than on developing new ones [12]. More and more modern companies use a maintenance-based software development paradigm, in which

[*] Chung Yung is currently reachable at yung@edgar.stern.nyu.edu.

software is developed mainly by modifying the source code of existing software systems [21]. The maintainability of software systems becomes one of the most important issues in software industry.

A lot of researches investigate in the techniques of improving software maintainability, software quality, and software reliability, and propose new methodologies of software metrics, which are applied to measuring software complexity and to monitoring the process of software development [3], [9], [15], [18].

The maintainability of software systems is driven by their complexity [2]. The cost of maintain legacy program is enormous because of the program's complexity [21]. There are plenty of materials dedicated to measure and analyze the complexity of software systems [4], [5], [12], [14], [22], [23]. Many metrics are famous with their power in predicting software complexity, such as software science [15], cyclomatic measurement [17], and so on. They measure the complexity of software systems by quantifying certain attributes of software, such as software size, control flow, data flow, and others [5], [7].

Size is one of the most important attributes of software systems [25]. It dominates the cost for the systems both in man-power and in budget, and both for development and for maintenance. Size based software metrics indicate the complexity of a software system mainly by its size attributes. These size base metrics help in predicting the cost for maintaining the system [5].

Control flow and data flow are two of the most important attributes, other than size, of software systems [5], [7], [9]. Control flow metrics capture the relation between the logic structures in a program with its complexity, while data flow metrics indicate the complexity of software system by their data dependency.

Readability is another important attributes of software systems that gives substantial affect on software maintainability [7], [8]. The software systems with less readable source code are recognized as more difficult to maintain than those with more readable source code. In contrast to the software attributes like size, control flow, and data flow, software readability attribute is more about psychological activities rather than physical ones. Readability Metrics are a family of software metrics that measure software complexity with taking readability into considerations.

The family of Simplified Readability Metrics is a new approach to measuring the complexity of software systems. In addition to simplifying Readability Metrics, we develop a new readability model. With the new readability model, we expand the software science metrics with a family of Readability Metrics. In particular, the magnitudes of Readability Metrics indicate the readability of software systems in percentage with respect to the readability of the algorithm implemented. And, we include the algorithms of Readability Metrics preprocesses and a few modifications we have made since Readability Metrics was published.

This paper is organized as such. The following section describes the attributes used by existing software metrics for measuring software complexity. We classify them into three categories and give examples of the metrics in each category. Section 3 is a more formal description on our Readability model. In Section 4, we introduce our new approach to Readability Metrics. We include the Readability Metrics preprocesses algorithm. We also show how the simplified approach measures the complexity of software systems by their readability. We include the results of one experiment set in Section 5. And, at last is a brief conclusion.

2. Attributes for Measuring Software Complexity

We distinguish the existing software complexity metrics by the attributes they use for measuring and we usually classify them into three categories: size based metrics, control flow based metrics, and data flow based metrics [5]. Please note that all the three categories of software attributes used for measuring software complexity are more about the physical activities in the software development life-cycle while software readability is more about the physical activities.

The size of a software system is a popularly conceived software attribute that affects software complexity [7], [25]. The size based metrics measure the complexity of software systems by their sizes. However, it is still arguable what is the basic unit of software size. The popularly used size based metrics include token counts, lines of code, software science, and so on.

A few empirical results indicated that the control flow complexity is well correlated with the overall complexity of a software system [12], [18], [27]. The control flow based metrics measure the complexity of software systems on their control flow graphs. Two of the most famous control flow based metrics are knots metrics and cyclomatic metrics. McCabe and Butler showed that cyclomatic metric also contributes in software testing and software maintenance, in addition to software complexity measurement [19].

Data flow based metrics measure the complexity of software systems by the inter and intra data dependency among modules [5], [20]. The results of numerous studies and experiments indicated that data dependency of a software system has a significant effect on the software complexity [13], [14]. The widely used data flow based metrics include Oviedo's metrics, live variable metrics, variable span metrics and Chung's metrics [4].

In the following of this section, we briefly introduce one family of software complexity metrics in each category as an example and as a comparison.

2.1 Software Science

In the early 1970's, Halstead investigated on measuring the complexity of software by analyzing its source code, which is called *software science*. Software science used a series of simple formulas to measure a few characteristics of a software system [16]. There are many papers published and concluded that the predictive power of software science is pretty well by Gordon [14], Woodfield [27], and other researchers. The details about software science are originally appeared in a monograph, *Elements of Software Science* [15]. A few later materials also described the metrics [9], [11], [23], [24].

Halstead defined four basic metrics computable from the program source code:

n_1 = the number of unique operators
n_2 = the number of unique operands
N_1 = the total number of operator occurrences
N_2 = the total number of operand occurrences

where, the operands are the variables or constants, and the operators are the symbols or combinations of symbols that affect the values or the ordering of operands.

For example, the C subprogram MatrixChain1 in Fig. 1 has 10 distinct operators and 12 distinct operands. The total number of operator occurrences is 44, and the total number of operand occurrences is 52.

```
MatrixChain1 ()
{  int n = length;
   int i, j, k, l, q;
   for (l=2; l<=n; l++) {
      for (i=1; i<=n-l+1; i++) {
         j = i + l - 1;
         m[i][j] = 50000;
         for (k=i; k<j; k++) {
            q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
            if (q < m[i][j]) {
               m[i][j] = q;
               s[i][j] = k;
}  }  }  }  }
```

Fig. 1: MatrixChain1 subprogram

The vocabulary *n* of a program, and the length *N* of a program were defined as

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

The volume *V* of a program, and the effort *E* of a program can be derived as

$$V = N * \log_2 n$$
$$E = V^2 / V^*$$

where V^* is the volume of the algorithm implemented in a procedure call. Since V^* is not easy to calculate, Halstead proposed measuring the effort *E* by its standard approximation *E^* as following:

$$E^{\wedge} = V * [(n_1 * N_2)/(2 * n_2)]$$

By Halstead's definition, the effort of a program is the mental activity required for reducing a preconceived algorithm to an actual implementation in a programming language [15].

measure	value
n_1	10
n_2	12
N_1	44
N_2	52
n	22
N	96
V	428
$\hat{E}$	9273

Table 1 : The software science metrics applied to MatrixChain1

The subprogram MatrixChain1 in Fig. 1 is a C program implementing the dynamic programming algorithm solving the matrix-chain multiplication problem. The metrics of software science applied to MatrixChain1 are listed in Table 1.

2.2 Cyclomatic Metric

In 1976, McCabe proposed cyclomatic metric by adapting a mathematical concept from graph theory [18]. Cyclomatic metric is popularly applied in measuring the software complexity for its simplicity and its mathematical background. In addition to measuring software complexity, cyclomatic metric also contribute to software testing and software maintenance [19].

For each structured software module, we may derive a directed graph from its control flow, called control flow graph. A node in the graph corresponds to a block of sequential code, and an edge in the graph corresponds to control flow in the module. The cyclomatic metric, derived from control flow graph G [4], [18], is defined as :

$$v(G) = e - n + 2$$

where *e* is the number of edges and *n* is the number of nodes in G.

Note that from the view point of graph theory *v(G)* is the number of linearly independent paths in G, and that *v(G)* depends only on the decision structure of the control flow graph G. The cyclomatic metric indicates the number of linearly independent circuits in a strongly connected control flow graph G.

For example, the cyclomatic metric of MatrixChain1 in Fig.1 is 4.

2.3 Variable Span Metrics

Variable span metrics measure software complexity by the number of statements between two successive references of a variable, based on the observation that large variable span results in higher software complexity [7], [9]. Two of the most important metrics in this family are program total span and program average span. Program total span metric is the total of the average span of each variable; while program average span metric is derived from dividing program total span by the number of variables in the program.

Variable span metrics capture the essence of how often a variable is used in a program. Furthermore, the size of a span indicates the number of statements that pass between successive uses of a variable. A large span can require the programmer to remember during the constructing process a variable that was last used far back in the program [5], [9].

For MatrixChain1 in Fig. 1, the span metrics of the variables are shown in Table 2.

3. Readability Model

If an algorithm is implemented by a few programmers with different programming styles, many different versions of its implementation will appear. The different versions of implementation have different complexity, different readability, and so on. One of our goals is to propose a readability model that shows the readability and the complexity of software systems with considering the readability difference in different implementations. So that, we have a standard to indicate the readability of the software system in order to keep it maintainable, and a criterion for measuring the complexity of software systems with respect to their readability.

variable	# of span	span sequence	average
n	1	1	1.000
i	10	0,0,1,1,1,1,0,1,1,1	0.700
j	7	1,1,1,0,1,1,1	0.857
k	6	0,0,1,0,0,3	0.667
l	4	0,0,1,1	0.500
q	2	1,1	1.000
m	1	1	1.000
p	2	0,0	0.000
s	0	0	0.000

```
program total span   = 5.724
program average span = 0.636
```

Table 2 : The variable span metrics applied to MatrixChain1

We start introducing our Readability model with presenting a motivating example which shows that the software complexity measured by Software Science does not indicate the relative readability between two versions of implementation of the same algorithm. And then, we introduce our Readability model in a more formal way.

3.1 A Motivating Example

There are a few attributes which make a software system more readable, such as proper comments; while some others make a software system less readable, such as badly named variables. One of the generally recognized attributes which make program source code difficult to read is the highly compound expressions. In such cases, splitting the highly compound expressions usually helps in making the program more readable.

The subprogram MatrixChain1 in Fig.1 is an implementation of the dynamic programming algorithm solving the *matrix-chain multiplication* problem, stated as follows: given a chain $<A_1, A_2, \ldots, A_n>$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1, A_2, \ldots, A_n$ in a way that minimizes the number of scalar multiplications [10]. In the implementation of

MatrixChain1, there is an statement with a highly compound expression which includes three additions, one subtraction, and two multifications.

Suppose that we have another version of implementation of the same algorithm as MatrixChain1 implements with a different programming style, shown as MatrixChain2 in Fig.2. The only difference is on the statement calculating *q*. In MatrixChain2, the statement with the highly compound expression is split into two statements, and the new variable has a logically clear definition. To most of software engineers, MatrixChain2 is much more readable than MatrixChain1. When maintaining software systems, the long compound statements, like the one in MatrixChain1, usually take much more time for software engineers to understand what it does.

```
MatrixChain2 ()
{  int n = length;
   int i, j, k, l, q, r;
   for (l=2; l<=n; l++) {
      for (i=1; i<=n-l+1; i++) {
         j = i + l - 1;
         m[i][j] = 50000;
         for (k=i; k<j; k++) {
            r = p[i-1] * p[k] * p[j];
            q = m[i][k] + m[k+1][j] + r;
            if (q < m[i][j]) {
               m[i][j] = q;
               s[i][j] = k;
}  }  }  }  }
```

Fig. 2 : MatrixChain2 subprogram

The software science metrics applied to MatrixChain2 are listed in Table 3. Compared with those metrics to MatrixChain1 in Table 1, we found that MatrixChain2 has larger program volume and larger program effort. MatrixChain2 uses simpler statements which are more readable and thus reduces the effort of maintenance. However, software science metrics failed in showing the complexity with considering software readability.

measure	value
n_1	10
n_2	13
N_1	45
N_2	54
n	23
N	99
V	448
$\hat{E}$	9305

Table 3: The software science metrics applied to MatrixChain2

In the following section, we develop a readability model as a theoretic background of measuring software complexity of software systems with considering their readability.

3.2 Readability Model

Theoretically, for a computable algorithm α, its algorithmic volume $V(\alpha)$ and its algorithmic effort $E(\alpha)$ are constants. That is, the algorithmic volume $V(\alpha)$ and the algorithmic effort $E(\alpha)$ are independent of the implementation. We state in a more formal way as follows:

> **Proposition 1:** For any computable algorithm α, $V(\alpha)$ and $E(\alpha)$ are constant, where $V(\alpha)$ is its algorithmic volume and $E(\alpha)$ is its algorithmic effort.

Without loss of generality, we assume that the algorithmic volume $V(\alpha)$ and the algorithmic effort $E(\alpha)$ are measured on the algorithm α in its most readable format.

We denote as $P_x = I_x(\alpha)$ that a programmer x implements algorithm α and results in a program P_x. The developing effort of program P_x, which x spent on implementing α, is denoted as $E_d(P_x)$, and the developed volume of program P_x is denoted as $V_d(\alpha)$.

We are interested in measuring the effort $E_r(\alpha)$ required for reading algorithm α and the effort $E_r(P_x)$ required for reading the developed program P_x. However, it is not easy to measure $E_r(\alpha)$ and $E_r(P_x)$ directly. On the other hand, $E_r(\alpha)$ and $E_r(P_x)$ are usually related to $E(\alpha)$, and $E_d(P_x)$, respectively. We propose that the effort $E_r(\alpha)$ required for reading algorithm α depends linearly on its algorithmic effort $E(\alpha)$, and that, in a similar way, the effort $E_r(P_x)$ required for reading the developed program P_x depends linearly on its developing effort $E_d(P_x)$. We state as the following proposition:

> Proposition 2: (1) For a computable algorithm α, $E_r(\alpha) = c_rE(\alpha)$, where $E_r(\alpha)$ is the effort required for reading α, $E(\alpha)$ is the algorithmic effort of α, and c_r is called readability coefficient. And, (2) $\forall P_x = I_x(\alpha)$, $E_r(P_x) = c_rE_d(P_x)$, where $E_r(P_x)$ is the effort required for reading P_x, and $E_d(P_x)$ is the effort spent on developing P_x.

Now we define *Readability* as follows:

> Definition (Readability): The readability of algorithm α, $Read(\alpha)$, is defined as the effort required for reading a unit volume of α. That is, $Read(\alpha) = E_r(\alpha) \ / \ V(\alpha)$. And similarly, the readability of a program P_x, $Read(P_x)$, is defined as the effort required for reading a unit volume of P_x. That is, $Read(P_x) = E_r(P_x) \ / \ V_d(P_x)$.

This is the Readability model that we use to develop our new approach to Readability Metrics. There are a few measures in Readability model that are not easy for measuring, such as $E(\alpha)$, c_r, and so on. The following section describes our new methodology. We will show how we measure those difficult measures.

4. Simplified Readability Metrics

To maintain a software system, as we know, involves in dealing with a few jobs on the software source code [4], [7]. These maintaining jobs include deleting redundant code, adding new functions, correcting errors, and so on. Before any of the maintaining jobs gets started, there is one thing we must do, that we need to read through the source code. Hence, it is obvious that software readability is an important attribute for maintaining software systems.

One of our goals is to propose a family of metrics that measure the complexity of software systems by their readability in order to keep the software systems readable and maintainable. Furthermore, our goal is to have a series of metrics that can indicate the relative readability difference between implementations of the same algorithm by the difference in magnitudes of the metrics.

In the above section, we give the definition of *Readabiltity* as the effort required for reading unit volume of the algorithm. But, it is not clear how we may measure the effort and the volume of an algorithm. In this section, we introduce the hypothesis for Readability Metrics. We propose two preprocesses that make possible measuring algorithmic volume and algorithmic effort. We introduce Readability Metrics for measuring the readability of software systems. And, we include the application of Readability Metrics to MatrixChain1 and MatrixChain2. More experiment results are shown in the next section.

4.1 Hypothesis

According to the discussion of above section, we recognize that MatrixChain2 is more readable than MatrixChain1. There are two properties that make MatrixChain2 more readable than MatrixChain1, and we will show that they may be quantified and measured. The two properties are: using simple statements and giving simple variable definitions.

We further extend on these two properties to a couple of baselines for readable programs. The readability baselines are stated as follows:

1. *using statements as simple as possible, and*
2. *giving variable definitions as simple as possible.*

Thus, we make our hypothesis as:

A software system with following the readability baselines has a lower complexity; while a software system without following the readability baselines has a higher complexity.

4.2 Preprocesses

Looking at the software systems currently used in software industry, we can hardly find any of them exactly following the above baselines. We need to quantify the factors in the baselines and measure how far the software systems are away from the baselines and how much additional effort required for maintaining the less readability software systems.

Before measuring software complexity, we apply two kinds of program transformations to the source code. The program transformations are adapted with compiler front-end techniques, and are listed as follows:

1. Create temporary variables are created for storing the temporary values in compound statements.
2. Insert necessary assignment statements for assigning the temporary values to the created temporary variables.
3. Rewrite the compound statements as simple statements with the created temporary variables.

The algorithm that we use for the preprocesses is listed in Fig.3.

```
1. Parse program source code and locate the compound statements.
2. For each compound statement with n mathematical operators
3.    If compound expression is right-hand-side of assignment
4.       Create n-1 temporary variables for the temporary values
5.    else
6.       Create n temporary variables for the temporary values
7.    Insert an assignment statement for each temporary variable
8.    Rewrite the compound statement as a simple statement
```

Fig. 3: Preprocesses algorithm

The preprocessed subprogram MatrixChain3 is shown in Fig. 4. It is not difficult to see that the preprocesses algorithm transforms both MatrixChain1 and MatrixChain2 to MatrixChain3.

```
MatrixChain3 ()
{  int n = length;
   int i, j, k, l, q;
   int t1, t2, t3, t4, t5, t6, t7, t8;
   for (l=2; l<=n; l++) {
      t1 = n - l;
      t2 = t1 + 1;
      for (i=1; i<=t2; i++) {
         t3 = i + l;
         j  = t3 - 1;
         m[i][j] = 50000;
         for (k=i; k<j; k++) {
            t4 = k + 1;
            t5 = i - 1;
            t6 = p[t5] * p[k];
            t7 = t6 * p[j];
            t8 = m[i][k] + m[t4][j];
            q = t8 + t7;
            if (q < m[i][j]) {
               m[i][j] = q;
               s[i][j] = k;
}  }  }  }  }
```

Fig. 4: Preprocessed MatrixChain3 subprogram

measure	value
n_1	10
n_2	20
N_1	52
N_2	68
n	30
N	120
V	589
$\hat{E}$	9130

Table 4: The software science metrics applied to MatrixChain3

After we transform the program by the preprocesses algorithm, we apply the software science metrics to MatrixChain3. The results are listed in Table 4. Note that the metrics show that MatrixChain3 requires the most program volume and the least program effort among the three versions of implementation.

4.3 Readability Metrics

Recall one of our goals is to have a series of metrics that show the readability and the complexity of software systems with considering the readability difference in different versions of implementation. In particular, we need an index for the program readability indicating the additional program effort of less readable implementations with respect to the algorithmic effort. First, we need to estimate the algorithmic volume and the algorithmic effort.

In most of the cases, the Readability Metrics preprocesses transform the different versions of implementation on the same algorithm into the same program. For example, the Readability Metrics preprocesses transform both MatrixChain1 and MatrixChain2 into MatrixChain3. Therefore, we estimate the algorithmic volume and the algorithmic effort of the implemented algorithm by the program volume and the program effort of the preprocessed program.

Since the readability and readability coefficient defined in our Readability model are not easy to measure directly, we define algorithmic readability index *ARI* and program readability index *PRI* as follows:

$$\begin{aligned}\mathrm{ARI} &= \mathrm{Read}(\alpha) \;/\; c_r \\ &= (E_r(\alpha) / \; c_r) \;/\; V(\alpha) \\ &= E(\alpha) \;/\; V(\alpha)\end{aligned}$$

$$\begin{aligned}\mathrm{PRI} &= \mathrm{Read}(P_x) \;/\; c_r \\ &= (E_r(P_x) \;/\; c_r) \;/\; V_d(P_x) \\ &= E_d(P_x) \;/\; V_d(P_x)\end{aligned}$$

where *Read*(α) is the algorithmic readability, c_r is the readability coefficient, $E_r(\alpha)$ is the effort required for reading α, $E(\alpha)$ is the algorithmic effort, $V(\alpha)$ is the algorithmic volume, *Read*(P_x) is the readability of P_x, $E_r(P_x)$ is the effort required for reading P_x, $E_d(P_x)$ is the developing effort of P_x, and $V_d(P_x)$ is the developed volume of P_x.

According to Halstead's software science, we measure the program effort effort by its standard approximation. So we get approximated algorithmic readability index *ARI^* and approximated program readability index *PRI^* as follows:

$$\begin{aligned} ARI\hat{} &= E\hat{}(\alpha) \ / \ V(\alpha) \\ &= (n_1(\alpha) \times N_2(\alpha)) \ / \ (2 \times n_2(\alpha)) \end{aligned}$$

$$\begin{aligned} PRI\hat{} &= E\hat{}_d(P_x) \ / \ V_d(P_x) \\ &= (n_1(P_x) \times N_2(P_x)) \ / \ (2 \times n_2(P_x)) \end{aligned}$$

We define the normalized program readability index *NPRI* as follows:

$$NPRI = PRI\hat{} \ / \ ARI\hat{} \times 100$$

And, we interpret *NPRI* as follows:

The readability of the software system developed is NPRI percent of the readability of the implemented algorithm.

From this point of view, a program is perfectly readable if its *NPRI* is 100.

4.4 Application

The Readability Metrics applied to MatrixChain1 and MatrixChain2 are listed in Table 5. Please note that the smaller NPRI of MatrixChain2 shows its relative readability to MatrixChain1.

Readability Metrics	MatrixChain1	MatrixChain2
ARI^	15.50	15.50
PRI^	21.67	20.77
NPRI	140	134

Table 5: Readability Metrics applied to MatrixChain1 and MatrixChain2

As described in previous section, we may interpret the Readability Metrics listed in Table 5 as:

> *The readability of MatrixChain1 is 140% of the readability of the MatrixChain algorithm; while the readability of MatrixChain2 is 134% of the readability of the same algorithm.*

So that, Readability Metrics show that MatrixChain2 is a little more readable than MatrixChain1 by that MatrixChain2 has smaller *NPRI* than MatrixChain1 does. As we know, the improved readability is due to splitting the highly compound statement into simpler ones.

5. Experiments

We show the result of one of experiment sets in Table 6. In this set, we experiment on the implementation of three number theoretic algorithms. We measure the software complexity of each program by our Readability Metrics. The implementation with smaller *NPRI* is considered as more readable. Thus, we indicate the software readability of each implementation.

5.1 Experiment Results

This set of experiments include the implementation on three well-known number theoretic algorithms: extended Euclid's greatest common divisor, modular linear equation solver, and the Chinese remainder theorem. The details of the each algorithm can be found in many algorithm books [1], [10].

The extended Euclid's greatest common divisor algorithm solves the greatest common divisor problem with giving additional useful information. Specifically, the extended algorithm computes *(d,x,y)* such that

```
d = gcd(a,b) = ax + by
```

The modular linear equation solver algorithm solves the problem of finding solutions to the equation

```
ax ≡ b (mod n)
```

The Chinese remainder theorem provides a correspondence between a system of equations modulo a set of pairwise relatively prime moduli and an equation modulo their product.

Metric	ExtendedEuclid	MLES	ChineseRemainder
n_1	9	13	14
n_2	10	19	26
N_1	25	46	73
N_2	34	65	98
n	19	32	40
N	59	115	171
V	251	575	910
$\hat{E}$	3840	12786	24010
$\hat{ARI}$	12.81	19.05	22.70
$\hat{PRI}$	15.30	22.24	26.38
NPRI	119	117	116

Table 6: The result of an experiment set

By Readability Metrics, we indicate that the readability of ExtendedEuclid is 119% of the readability of the extended Euclid's greatest common divisor algorithm; that the readability of MLES is 117% of the readability of the modular linear equation solver algorithm; and that the readability of ChineseRemainder is 116% of the readability of the Chinese remainder theorem. So that, ChineseRemainder is considered as the most readable of the three programs in the experiment set. Recall that the readability is defined as the effort required for reading the unit volume of the program.

5.2 Experiment Summary

When applying Readability Metrics for measuring software complexity, we get serveral information from the magnitudes of the metrics:

1. With applying software science metrics, the program volume and the program effort are good in indicating the size of developed software systems and the effort spent on developing the software systems. By the Halstead's definitions [15] that the program effort is the mental activity required to implement a preconceived algorithm to an actual program in a programming language. However, the program effort measure does not show the effort required for maintaining the software systems.
2. When an algorithm is implemented by programmers with different programming styles, a few different versions of implementation will appear. Software science fails in showing the software complexity difference between the versions due to the difference in their readability. The Readability model describes the effort required for reading the source code of a software system, and reading the source code is one of the most mental activities in maintaining a software system. Readability Metrics indicate the readability of each version of the implementation.
3. To indicate the additional effort required for reading the software system due to the less readability of its the source code, we need to estimate the effort required for reading the algorithm that it implements. Readability preprocesses are proposed for measuring the readability of the algorithm.
4. The metric *NPRI* is proposed to indicate the readability of a software system. In the case that software systems are developed with following the proposed readability baselines, the *NPRI* metric to the systems is 100, which indicates reading the source code of the software system requires effort as much as the implemented algorithm requires; that is, no extra effort is required due to the less readable programming style.

6. Conclusion

A new approach to measuring the complexity of software systems with considering their readability is proposed. We develop a Readability model for describing the effort required for reading the source code since reading the source code is one of the most important activities of maintenance. The readability preprocesses are proposed for measuring the readability of the algorithm. Readability Metrics are proposed with Readability model for measuring the readability of software systems. The applications of Readability Metrics help in keeping the source code of software systems readable so that the software systems are maintainable in the later phases of the software development life-cycle.

One possible direction of our future researches is exploring more attributes that are not taken into consideration by the existing software complexity metrics. We are also interested in investigating more on applying our model to reverse engineering.

Acknowledgement

The author appreciates Professor Ajit Kambil for his kind support and encouragement on the progress of this research work. The author is grateful to Professor Edward Stohr for his help in making publishing this paper possible. The author thanks Professor Ellaine Weyuker for her precious comments on the earlier version of this paper.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Press, 1975.

{2] C. M. Chung, W. R. Edwards, and M. G. Yang, "A Software Environment Combining Metrics, Program Information, and Testing Methodologies," *Proceedings of International Computer Symposium 1988*, Taipei, Taiwan, Vol. 1, pp. 696 - 703.

[3] C. M. Chung, W. R. Edwards, and M. G. Yang, "Static and Dynamic Data Flow Metrics," *Policy and Information*, Vol. 13, No. 1, pp. 91-103, June 1989.

[4] C. M. Chung, and M. G. Yang, "A Software Maintainability Measurement," *Proceedings of The 1988 Science, Engineering and Technology Seminars*, Houston, Texas, pp. V4-12 - V4-16.

[5] C. M. Chung, and M. G. Yang, "A Software Metrics Based Software Environment for Coding, Testing and Maintenance," *Proceedings of The 1988 Science, Engineering and Technology Seminars*, Houston, Texas, pp. T3-13 - T3-17.

[6] C. M. Chung, M. G. Yang, J. H. Chou, and Y. H. Chou, "Algorithms for Finding the Most Complicated Loop-free and Loop-once Paths," *Proceedings of National Computer Symposium 1989*, Taipei, Taiwan, pp. 522-530.

[7] C. M. Chung, and C. Yung, "Measuring Software Complexity Considering Both Readability and Size," *Information and Communication*, Tamkang Univ., Taiwan.

[8] C. M. Chung, and C. Yung, "Readability Metrics," *The Proceedings of Mid-America Chinese Professional Annual Convention 1990*, Chicago, Illinois.

[9] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings Press, 1986.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1991.

[11]N. S. Coulter, "Software Science and Cognitive Psychology," Mar. 1983, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 2, pp. 166 - 171.

[12] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," Mar. 1979, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, pp. 96 - 104.

[13] H. E. Dunsmore, and H. D. Gannon, "Data Referencing: an Empirical Investigation," *IEEE Computer*, pp. 50-59, Dec. 1983.

[14] R. D. Gordon, "Measuring Improvements in Program Clarity," Mar. 1979, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, pp. 79 - 90.

[15] M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland Press, New York, 1977.

[16] M. H. Halstead, "Guest Editorial on Software Science," Mar. 1979, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, pp. 74 - 75.

[17] B. W. Kernighan, and D. M. Richie, The C Programming Language, 2nd Edition, Prentice-Hall Press, 1988.

[18] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 308-320, Dec. 1976.

[19] T. J. McCabe, and C. W. Butler, "Design Complexity Measurement and Testing," *Communications of the ACM*, Vol. 23, No. 12, pp. 1415 - 1425, Dec. 1989.

[20] E. Oviedo, "Control Flow, Data Flow and Program Complexity," *Proceedings of The IEEE COMPSAC 1980*, pp. 146-152.

[21] C. V. Ramamoothy, and W-T. Tsai, "Advances in Software Engineering," *IEEE Computer*, Vol. 29, No. 10, pp. 47-58, Oct. 1996.

[22] B. Ramamurthy, and A. Melton, "A Synthesis of Software Science Measures and the Cyclomatic Number," Aug. 1988, *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, pp. 1116 - 1121.

[23] W. H. Shaw, J. W. Howatt, R. S. Maness, and D. M. Miller, "A Software Science Model of Compile Time,", *IEEE Transactions on Software Engineering*, Vol. 15, No. 5, pp. 543 - 549, May 1989.

[24] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," Mar. 1983, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 2, pp. 155 - 165.

[25] J. Verner, and G. Tate, "A Software Size Model," IEEE Transactions on Software Engineering, Vol. 18, No. 4, pp. 265 - 278, Apr. 1992.

[26] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1357-1365, 1988.

[27] S. N. Woodfield, "An Experiment on Unit Increase in Problem Complexity," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, pp. 76-79.