Analysis of Weighted Composite Complexity Measure

D. I. De Silva
Faculty of Computing
Department of Software Engineering
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
dilshan.i@sliit.lk

Abstract— **Software metrics have become one of the essential elements in the software development life cycle due to their ability to identify and measure crucial parameters that affect the development of software. Out of different kinds of metrics that have been proposed, metrics introduced to measure the complexity of programs have managed to get an upper hand over the other metrics due to their ability to reduce the time, effort and cost associated with the testing and maintenance phases of the software development life cycle. Thus, numerous efforts have been taken to introduce new complexity metrics. Many of these complexity metrics have only considered a single factor to measure the complexity. As a result, they have not been able to provide the expected results for any given program. This issue can be addressed to a greater extent by introducing metrics that use a commonly accepted set of factors to measure complexity. Chhillar and Bhasins' weighted composite complexity measure is one of the very few metrics that has considered more than one factor to measure program complexity. However, there exist a number of factors that the weighted composite complexity measure has also not considered. This paper proposes such factors which the weighted composite complexity measure can consider to improve its accuracy. It also includes an application of the metric to a sample C++ program. In addition, it explains the importance of software metrics and complexity metrics.**

***Keywords*— Software metrics, software complexity, weighted composite complexity measure**

I. INTRODUCTION

Now it has come to an era where majority of the people living in all over the world are relying on their personal computers or tablets to do their day to day work. With each day to come the demand for new software applications continue to increase rapidly with the public searching for ways to make their day to day life much simpler and easier. This has caused to the development of large-scale, complex software. Development of these applications on time without exceeding the allocated budget with the expected quality has become a major problem for most of the Information Technology companies. All these issues can be addressed at least up to some extent by proper usage of software metrics.

This has made the path way to the introduction of various types of software metrics. Out of them, one of the mostly discussed metrics are the metrics proposed to measure the complexity associated with software. The main reason behind the popularity of these metrics is their ability to reduce the time, effort, and cost associated with the testing and maintenance phases of the software development life cycle.

Researchers and computer science practitioners have tried to measure the complexity associated with software during different phases of the development life cycle using the requirement engineering document, design, and source code. Majority of the complexity metrics have been proposed based the source code of a program. A common problem with many of these code complexity metrics is that they have only considered a single factor to measure the complexity of a program. As a result, they have not been able to provide the expected results when the main contributor of program complexity has been another factor.

Out of the proposed complexity metrics the weighted composite complexity measure which was introduced by Usha Chhillar and Shuchita Bhasin, is one of the very few metrics that has taken into account more than one factor to measure program complexity. However, there exist a number of factors the weighted composite complexity measure has also not considered. Thus, this paper attempts to propose such factors which the weighted composite complexity measure can consider to improve its accuracy.

This paper initially introduces software metrics including the definitions, classifications, and the importance of them. The readers' attention is then drawn towards a specific kind of a metric: software complexity metrics. Section III introduces the complexity metrics that has been proposed for both procedure-oriented and object-oriented (OO) programs. Section IV discusses about the weighted composite complexity measure. Section V discusses the additional factors that can be considered by the weighted composite complexity measure to further enhance its accuracy are presented. Finally section VI concludes the paper.

II. BACKGROUND

A. Software Metrics

With the progression of the software engineering field, software metrics have become a vital component of the software life cycle, in helping to better understand and evaluate software projects, products, processes, services, and resources against established standards and goals [1]. Software metrics are invaluable in presenting relevant information for technical and managerial decisions. They are critical to planning and controlling the resources and processes used to produce software, helping to lessen deviation between the actual emerging product and the theoretical development plan. Furthermore, software metrics enables measurement and analysis of various aspects of software such as the cost and effort required, quality of the final product, complexity, reusability, reliability, testability

978-1-5090-0082-1

etc., making the forecasting and maintenance much easier. They also help to predict attributes of future software entities.

Various definitions of software metrics have been advanced, including the following:

Definition 1 [2]: Software metrics can be used to quantity a software as well the process that is used to produce it.

Definition2[2]: Software metrics can be used to quantify attributes which are derived from development processes, products and supporting resources".

Definition 3[2]: Software metrics can be considered as process. The input for this process is data related to the software and the outcome of the process is a value which can possibly decide the way how an attribute would affect that software.

Out of the many available ways, one way to categorized software metrics is based on the following four types:

- **Procedure metrics:** used to control and manage the whole development process [2]. Development status of the High-level managers can be tracked using the procedure metrics [2]. They mainly focus on how long a procedure is going to last, the final cost, the techniques used, and the comparison results when one technique is compared with another alternative one [2].

- **Process metrics:** used to provide measures of the software development process [69]. They can be used to measure the overall development time, type of methodology, and the average programming experience of the employees [3].

- **Project metrics:** used to check whether the project meets its goals. These metrics are mainly used to modify the project, so that problems can be anticipated and avoided and to make the best use of the development plans [2].

- **Product metrics:** used to provide measures of the software product from the requirements gathering stage to deployment [3]. They measure documentation complexity, design complexity, and code complexity [3].

Apart from the above, software metrics can be classified in other ways as well [3], including objective or subjective metrics. Objective metrics such as the size of the program measured in lines of code (LOC) and development time, are independent of the person conducting the measurement [3]. Subjective metrics such as the level of programmer experience and the classification of software projects according to Boehm's COCOMO cost estimation model, depend on the individual performing the measurement [3].

Grady classifies software metrics as primitive metrics and computed metrics [3]. While primitive metrics are directly observable, computed metrics are computed in some manner from other metrics without being directly observed [3]. Program size in lines of code (LOC), number of defects, etc. are examples for primitive metrics [3]. On the other hand, Lines of code generated per person-month, number of defects per kilo lines of code, etc. are examples for computed metrics [3].

B. Software Complexity

Many experts in the software engineering discipline share different views on software complexity. Brooks defines software complexity as a crucial property, not as an unintentional one. Thus, explanations of a software entity which removes its complexity often removes its core values [6]. According to him, essential complexity is caused by the nature of the problem that is to be solved and the skill set that is required to understand the problem. The promising attacks on the essential complexity such as the use of off-the-shelf software (buy vs. build), the usage of rapid requirements refining and prototyping can be addressed by using the skills of great designers who could better tackle the complexity and abstract nature of the software engineering beast. On the other hand, accidental complexity is imposed by the tools and processes around the development of software – such as physical limitations of size and speed, low levels of abstraction in existing programming languages, problems in the specification or interpretation of requirements, and inefficient communication among developers. He further stated that most of the so-called silver bullets (at that time) such as high-level languages, time-sharing, unified programming environments, object-oriented programming, Artificial Intelligence (AI), expert systems and graphical/automatic programming addressed this accidental complexity.

Dattathreya and Singh define complexity of a software as a combination of reliability, availability, and maintainability (RAM) elements [8].

Measuring the complexity of a software provides several benefits for an organization:

- For start-ups in the industry, it allows to maintain capacities and plan for future expansions. This help the management to make their decisions much more easily.
- For existing organizations, measuring software complexity helps identify good programmers, reward their programming practices, and increase the profitability of the company, while backing its productivity.
- From a management perspective, it helps to predict the cost and the time line of forthcoming projects, assess the productivity effect of new tools and methods, establish productivity trends over time, improve the quality of software, predict future necessities of the employees, and foresee and decrease upcoming maintenance requirements [9].

III. COGNITIVE COMPLEXITY METRICS

Proposing cognitive metrics started in the early twenties. The first metric to propose was cognitive functional size (CFS) which was proposed by Shao and Wang in April 2003[7]. It measured the complexity that occurred due three factors: internal processing structures, inputs and outputs. Later in that year Klemola and Rilling proposed the kinds of lines of code identifier density (KLCID) complexity metric [11]. It captured the effect of understanding the set of unique kinds of lines of code of a program. In January 2006, Kushwaha and Misra introduced the cognitive information complexity measure (CICM) [12]. The same year, Sanjay Misra introduced the cognitive weight complexity measure (CWCM) [13]. In 2007, Sanjay Misra again proposed another metric: Class Complexity (CC) measure, which computed the complexity of a program by considering the internal structure of methods [14]. Furthermore, in 2008, Misra and Akman introduced weighted class complexity (WCC) which computed the structural and cognitive complexity of a class

978-1-5090-0082-1

[18]. Gupta and Chhabra suggested cognitive-spatial complexity measures to compute the complexity of functions and classes, in 2009 [16]. In 2011, Misra, Akman and Koyuncu proposed the cognitive code complexity (CCC) to evaluate the design of object-oriented code [17]. In July that same year, Chhillar and Bhasin suggested the weighted composite complexity measure by considering the type and the nesting level of control structures and the inheritance level [10]. In February 2012, Code comprehending measure (CCM) was proposed by Gurdev, Satinderjit, and Monika. It measured the complexity based on three aspects: data volume, structural complexity and data flow complexity [19]. Later in that year, Aloysius and Arockiam proposed their cognitive complexity metric which measured the complexity that occurred due to different types of coupling: Global Data Coupling (GDC), Internal Data Coupling (IDC), Lexical Content Coupling (LCC), Control Coupling (CC) and, Data Coupling (DC) [15].

IV. Weighted Composite Complexity Measure [10]

Chhillar and Bhasin believed that software complexity is a multidimensional attribute of software and thereby it cannot be measured by considering a single factor. With this in mind, they proposed weighted composite complexity measure. It was based on four significant factors that contribute to the complexity of software. The four factors are as follows:

Inheritance level of classes (W_i):

The degree of understanding a statement increases with the level of inheritance of classes. Taking this into account Chhillar and Bhasin assigned a weight of zero for executable statements in the bass class, a weight of one for the executable statements which are at the first derived class, 2 for the statements at the next derived class. Likewise the weight allocated for the statements increases by one for each derived class.

Type of control structures in classes (W_c):

Chhillar and Bhasin believed that the complexity added to a class/program by a control structure varies depending on its type. Thus, a weight of zero was assigned to sequential statements, one for conditional control structures such as if-else and if-else if conditions, two for iterative control structures such as for, while, and do-while loops, and n for switch-case statements with n cases.

Nesting level of control Structures (W_n):

The understandability of a program increases with the number of nesting levels of control structures. Consequently the complexity of that program will also increase as a result. Taking this into consideration Chhillar and Bhasin assigned a weight of zero for sequential statements, one for statements which are at the outer most level, two for statements which are at the next inner level of nesting and so on.

Size of a class in terms of token count:

With the belief that the complexity of a class or program increases along with the size of it, Chhillar and Bhasin considered the size of a class or program to be the final factor of their measure. The size of a particular executable statement was calculated in terms of the operators, operands, methods/functions, and strings in that statement.

Considering the above mentioned aspects, a weighted complexity measure for an object-oriented program P was suggested as:

$$C_w(P) = \sum_{j=1}^{n} (S_j) * (W_t)_j$$

Where

$C_w(P)$ = Weighted complexity measure of program P

S_j = Size of j^{th} executable statement in terms of tokens count

n = Total number of executable statements in program P

j = Index variable

$W_t = W_n + W_i + W_c$

A calculation of the weighted composite complexity measure for the program in fig. 1 can be found in Table 1.

V. Discussion

Chhillar and Bhasin have only considered about only one object oriented feature: inheritance level in their metric. However, this metric can be further enhanced by considering the coupling between objects.

Weighted composite complexity measure does not consider the complexities that occur due to variable declarations. Thus, line numbers 5 and 39 in the program that is shown in Fig. 1 will not lead to any complexity. However, the complexity associated with a program should change depending on the number of variable declarations in it.

The count of functions and the complexity of those functions is a clear indicator of the time and effort required to implement and maintain a program [9]. This has been addressed by the weighted composite complexity measure. In addition to that the complexities that occur due to method calling have also been considered up to some extent. However, according to the proposed measure the complexities that occur due to method calling would not differ depending on the content of the method that is called. For example, as shown in Table 1, line number 56 and 58 in Fig. 1 have the same weighted complexities. But the statement that calls the area method should be more complex than the statement that calls the perimeter method.

This metric does not consider the complexity added to inputs and outputs of a program. But the author believes that the complexity added due to inputs and outputs of a program should also be considered to accurately to compute the complexity of a software program.

Conditional statements causes a program to execute differently based on a given condition. They can be divided into two types: simple and compound. Conditional statements which has only one expression are known as simple conditional statements and conditional statements with two or more expressions are known as compound conditional statements. In compound conditional statements all the expressions are checked at once using &&, ||, and ! logical operators. Chhillar and Bhasin only allocated a weight of one for all conditional statements regardless of the number of expressions that were checked using them. However, it is

978-1-5090-0082-1

```
1  # include <iostream>
2  using namespace std;
3  class Polygon1 {
4    protected:
5    int w, ht;
6    public:
7      void setValues(int a, int b);
8  };
9    void Polygon1 :: setValues(int a, int b)
10   {
11       w=a;
12       ht=b;
13   }
14
15 class Output {
16   public:
17       void printOutput( );
18 };
19     void Output  :: printOutput( )
20      {cout << "Area is : "; }
21
22 class Rectangle1: public Polygon1 {
23   public:
24      int area( );
25      int perimeter( );
26 };
27     int Rectangle1 :: area( ){
28       Output out;
29       out.printOutput( );
30       if( ht >0 &&  w >0)
31           return w * ht;
32      else
33           return 0;
34   }
35    int Rectangle1 :: perimeter( )
36      { return (w + ht) * 2; }
37
38 class Cube {
39   int tw, tht, tlength;
40   public:
41   int volume( );
42 };
43    int Cube :: cubeVolume( ){
44     tw = 4;
45     tht = 5;
46     tlength = 3;
47     Output out;
48     out.printOutput( );
49     if( tw > 0 && tht > 0 && tlength > 0 )
50          return  tw * tht * tlength;
51   }
52 int main ( ){
53   Cube c;
54   Rectangle1 rec;
55   rec.setValues (4,5);
56   cout << rec.area( ) << '\n';
57   cout << c.volume( ) << '\n';
58   cout << rec.perimeter( )<< '\n';
59   return 0;
60 }
```

Fig. 1 A sample program P

TABLE I. CALCULATION OF WEIGHTED COMPOSITE COMPLEXITY MEASURE FOR PROGRAM P

Line No	S_j	W_n	W_i	W_c	W_t	$S_j*(W_t)_j$
9	4	0	1	0	1	4
11	3	0	1	0	1	3
12	3	0	1	0	1	3
19	4	0	1	0	1	4
20	3	0	1	0	1	3
27	4	0	2	0	2	8
29	3	0	2	0	2	6
30	8	1	2	1	4	32
31	4	1	2	0	3	12
33	2	1	2	0	3	6
35	4	0	2	0	2	8
36	6	0	2	0	2	12
43	4	0	1	0	1	4
44	3	0	1	0	1	3
45	3	0	1	0	1	3
46	3	0	1	0	1	3
48	3	0	1	0	1	3
49	12	1	1	1	3	36
50	6	1	1	0	2	12
52	2	0	0	0	0	0
55	6	0	1	0	1	6
56	7	0	2	0	2	14
57	7	0	1	0	1	7
58	7	0	2	0	2	14
59	2	0	0	0	0	0
$C_w(P)$						**206**

believed that the accuracy of the complexity calculation can be improved by allocating different weights for conditional statements depending on the number of expressions checked from them. In the program that is shown in Fig. 1 the weight assigned for both the compound conditional statements in line number 30 and 49 is the same. But the compound conditional statement at line number 49 is more complex than the compound conditional statement at line number 30.

VI. CONCLUSION

Although a number of complexity metrics have been proposed, a common issue with most of them is that they have only looked at the complexity from one aspect. Chhillar and Bhasins' weighted composite complexity measure is one metric that has gone against the norm and has looked at the complexity from different aspects. It considers the complexities that arise due to inheritance level of statements, type of control structures, nesting level of control structures, and size of the program. In addition to these four factors, this paper propose some other factors that can have an effect on complexity, to further improve the accuracy of the weighted composite complexity measure.

978-1-5090-0082-1

ACKNOWLEDGEMENT

My gratitude goes to Prof. Saluka Kodituwakku, Dr. Amalka Pinidiyaarachchi, and Mr. Nuwan Kodagoda for the valuable comments lent to me when writing the paper.

REFERENCES

[1] L.Westfall, "12 Steps to Useful Software Metrics", 23 April 2015. [Online]. Available:http://www.westfallteam.com/Papers/12_steps_paper.pdf.

[2] T. Honglei, S.Wei and Z.Yanan, "The research on software metrics and software complexity metrics", in Proc. 2009 International Forum on Computer Science-Technology and Applications, Chongqing, China 25-27 Dec. 2009, 1, pp.131-136.

[3] Carnegie Mellon University, Software Engineering Institute. Software Metrics - SEI Curriculum Module SEI-CM-12-1.1: prepared by Everald E. Mills. Seattle: Washington, 1988.

[4] T.J. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering, Dec. 1976, SE-2 (4), pp. 308-320.

[5] G. J. Myers, "An extension to the cyclomatic measure of program complexity", SIGPLAN Notices, October. 1977, pp.61-64.

[6] Fred P. Brooks, "No Silver Bullet - Essence and Accidents of Software Engineering", IEEE Computer, April 1987, 4 (20), pp.10 - 19.

[7] Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," Canadian Journal of Electrical and Computer Engineering, April 2003, 28.

[8] M. S. Dattathreya and H. Singh, "Army Vehicle Software Complexity Prediction Metric – Five Factors", (Preprint). No. Tardec-20723. Army Tank Automotive Research Development and Engineering Center Warren Mi, 2010.

[9] S. R. Chidamber and C.F. Kemerer, "Towards a metrics suite for Object Oriented Design", in Proc. OOPSLA '91 Conference proceedings on Object-oriented programming systems, languages, and applications, New York, USA, Nov. 1991, 26, pp.197- 211.

[10] U. Chhillar and S. Bhasin, "A new weighted composite complexity for object-oriented systems", International Journal of Information and Communication Technology Research, July 2011, 1(3), pp.101-108.

[11] T. Klemola and J. Rilling, A cognitive complexity measure based on category learning, The Second IEEE International Conference on Cognitive Informatics (ICCI), Aug. 2003, pp. 106-112.

[12] D.S. Kushwaha and A.K. Misra, A modified cognitive information complexity measure of software. ACM SIGSOFT Software Engineering Notes, 31(1), Jan. 2006, pp.1-4.

[13] S. Misra, "A Complexity Measure Based on Cognitive Weights," International Journal of Theoretical and Applied Computer Sciences, 2006, 1(1), pp. 1-10.

[14] S. Misra, "An Object Oriented Complexity Metric Based on Cognitive Weights", 6th IEEE International Conference on Cognitive Informatics (ICCI 07), 2007, pp. 134-139.

[15] A. Aloysius and L. Arockiam, "Coupling Complexity Metric: A Cognitive Approach", International Journal of Information Technology and Computer Science (IJITCS), 4(9), August 2012, pp. 29- 35.

[16] V. Gupta and J. K. Chhabra, "Object-oriented cognitive-spatial complexity measures", International Journal of Computer Science and Engineering, 3(6), 2009, pp.122-129.

[17] S. Misra, I. Akman, and M. Koyuncu, "An inheritance complexity metric for object-oriented code: A cognitive approach",Sadhana Academy Proceedings in Engineering Sciences, 36 (3), 2011, pp. 317 – 377.

[18] S. Misra and K. I. Akman, "Weighted Class Complexity: A Measure of Complexity for Object Oriented System", Journal of Information Science and Engineering, 24 (6), November 2008, pp. 1689-1708.

[19] G. Singh, S. Singh, and M. Monga, "Code Comprehending Measure", International Journal of Computers and Technology, 2 (1), February 2012, pp. 9 - 14.

978-1-5090-0082-1