A Model for Measuring Cognitive Complexity of Software

Sanjay Misra and Ibrahim Akman

Department of Computer Engineering, Faculty of Engineering
Atilim University, Ankara, Turkey
smisra@atilim.edu.tr, akman@atilim.edu.tr

Abstract. This paper proposes a model for calculating cognitive complexity of a code. This model considers all major factors responsible for (cognitive) complexity. The practical applicability of the measure is evaluated through experimentation, test cases and comparative study.

Keywords: Software complexity, metric, size, structure, cognitive complexity, understandability.

1 Introduction

Software metrics have always been important for software engineers to assure software quality because they provide approaches to the quantification of quality aspects of software. However, absolute measures are uncommon in software engineering [9]. Instead, software engineers attempt to derive a set of indirect measures that lead to metrics that provide an indication of quality of some representation of software. The quality objectives may be listed as performance, reliability, availability and maintainability [10] and are closely related to software complexity. Complexity is defined by IEEE [3] as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" Over the years, research on measuring the software complexity has been carried out to understand, what makes software products difficult to develop, maintain, or use. Major complexity measures of software that refer to effort, time and memory expended have been used in the form of different software metrics. Cyclomatic number [4], Halstead programming effort [2], data flow complexity measures [8], cognitive functional size measure [11], are examples to such metrics. Number of metrics can also be found at [7]. These metrics calculate the complexity of software from the code and measures only specific internal attributes like size, algorithm complexity, control flow structures etc. In all above mentioned complexity metrics, they attempt to quantify the primitives which make software difficult to understand. For many of them, the developer's claim that their complexity metric based on an internal attribute is the most accurate predictor of software quality. However, the authors realize that a single internal attribute is not sufficient for measuring the complexity of the code. For measuring the complexity of a code, one must consider most of the internal attributes responsible for complexity. Therefore, the purpose of this paper is to propose a new complexity metric which

I. Lovrek, R.J. Howlett, and L.C. Jain (Eds.): KES 2008, Part II, LNAI 5178, pp. 879–886, 2008.

calculates complexity of the program code by considering all factors responsible for complexity. For this, first we identified the factors which are responsible for the complexity and then established a metric to reflect a proper relationship between these factors. In our previous work, we presented a metric in ICCI, 2007, [6] which is based on input, output and basic control structures (based on cognitive informatics [12]). In the present work, we extended our previous work by including all the factors responsible for complexity of software.

In section 2, we identified the primitives responsible for the complexity and accordingly proposed a new measure. The metric is demonstrated in section 3. Experimentation and comparative study are given in section 4. The last section 5 includes the conclusions drawn.

2 Proposed Metric: Unified Complexity Measure (UCM)

Complexity of a code is directly dependent on the understandability of the code and relates to ease of comprehension. It is a cognitive process. All the factors that makes program difficult to understand are responsible for cognitive complexity. When we analyze a program code we find that that number of lines (size), total occurrence of operators and operands (size), numbers of control structures (control flow structuredness), function call (coupling) are the factors which directly affect the complexity. In general, these primitives are measured independently by different complexity measures and each one of these is assumed to represent overall complexity of the software. When we look at most of the known complexity measures, we can observe the close relation between number of lines, operator and operand counts, and basic control structures. Consequently, these primitives of software may constitute the components of a unified, comprehensive complexity measure.

In our opinion, the complexity of a software system depends on following factors:

1. Complexity of program depends on the size of the code. We suggest that the size of the code can be measured by total occurrence of operators and operands. Therefore, the complexity due to i^{th} line of the code can be calculated as

$$SOO_i = N_{i1} + N_{i2} \text{ Where}$$

N_{i1}: The total number of occurrences of operators at line i,
N_{i2}: The total number of occurrences of operands at line i,

2. Complexity of the program is directly proportional to the cognitive weights of Basic Control Structures (BSC). Cognitive weight of software [11] is the extent of difficulty or relative time and effort for comprehending given software modeled by a number of BCS's. BCS's, sequence, branch and iteration [11] are basic logic building blocks of any software and their weights are one, two and three respectively. These weights are assigned on the classification of cognitive phenomenon as discussed by Wang [11]. He proved and assigned the weights for sub conscious function, meta cognitive function and higher cognitive function as 1, 2 and 3 respectively. In fact, cognitive weights correspond to the number of executed instructions. The details of the weights for different BCS's are given in Table-1, see [11].

Table 1. Basic Control Structures and their weights

Category	Basic Control Structures	Cognitive Weight
Sequence	Sequence	1
Branch	If-Then-Else	2
	Case	3
Iteration	For-do	3
	Repeat-until	3
	While-do	3
Embedded Component	Function Call	2
	Recursion	3

As a result, the cognitive complexity due to ith line of the code, CW_i, can be weighted as in Table-1.

Using the above considerations, we propose the following model to establish a proper relationship among internal attributes of software.

$$UnifiedComplexityMeasure(UCM) = \sum_{i=1}^{n} \sum_{j=1}^{m_i} (SOO_{ij} * CW_{ij}) \qquad (1)$$

where complexity measure of the software code UCM is defined as the sum of complexity of its n modules and module i consists of m_i line of code.
It is important to note here that in this formula:

- number of lines (m_i), number of operators and operands correspond to size of software,
- total occurrence of basic control structures, operators and operands (SOO_{ij}) is related to algorithm complexity,
- basic control structures (CW_{ij}) are related to control flow structuredness, therefore corresponds to structural complexity,
- CW_{ij} also corresponds to cognitive complexity.
- number of modules (n) is related to modularity,
- function calls in terms of basic control structures are related to coupling between modules(in terms of CW_{ij}'s).

We believe that these are the major factors which are responsible for the program comprehension, therefore complexity of the software system.

In our context, the concept of cognitive weights is used as an integer multiplier. Therefore, the unit of the UCM (Unified Complexity Unit-UCU) is always a positive integer number. This implies achievement of scale compatibility of *SOO* and *CW*.

3 Demonstration of UCM

The proposed complexity metric given by equation 1 is demonstrated with the programming example given by the following Table 2.

Table 2. Calculated complexity values for the example program

Line No.	Sample Algorithm	Components		UCM_i
		SOO_i	CW_i	
Line 1	#include<stdio.h>	0	1	0
Line 2	#include<stdlib.h>	0	1	0
Line 3	#include<conio.h>	0	1	0
Line 4	int main (){	0	1	1
Line 5	long fact (int n);	3	1	3
Line 6	int isprime(int n);	3	1	3
Line 7	int n;	2	1	2
Line 8	long int temp;	2	1	2
Line 9	clrscr();	1	1	1
Line 10	printf("\n input the num-ber");	1	1	1
Line 11	scanf("%d",&n);	2	1	2
Line 12	temp=fact(n);	5	2	10
Line 13	{printf("\n is prime");}	1	1	1
Line 14	int flag1=isprime(n);	5	2	10
Line 15	if (flag1==1)	3	2	6
Line 16	else	0	1	0
Line 17	{printf("\n is not prime")};	1	1	1
Line 18	printf("\nfactorial(n)=%d", temp);	2	1	2
Line 19	getch();	1	1	1
Line 20	long fact(int n)	2	1	2
Line 21	{long int facto=1;	4	1	4
Line 22	if (n==0)	3	2	6
Line 23	facto=1;else	4	1	4
Line 24	facto=n*fact(n-1);	9	1	9
Line 25	return (facto); }	2	1	1
Line 26	int isprime(int n)	2	1	2
Line 27	{ int flag;	2	1	1
Line 28	if (n==2)	3	2	6
Line 29	flag=1;	4	1	4
Line 30	else	0	1	0
Line 31	for (int i=2;i<n;i++)	10	3	30
Line 32	{ if (n%i==0)	5	2	10
Line 33	{ flag=0;	4	1	4
Line 34	break; }	1	1	1
Line 35	else {	0	1	0
Line 36	flag=1 ;}}	4	1	4
Line 37	return (flag);}}	2	1	2
	TOTAL			136

This example consists of a simple source code, which contains a main program and two functions. The main program (lines 1-19) calls the function fact (lines 20-25) to calculate the factorial of the inputted positive integer and calls the function prime (lines 26-37) to check whether the inputted integer is a prime number or not. The last

three columns of table 2 show how the UCM is calculated for each line of code. It also demonstrates how complexity value varies from line to line depending on the architecture and size of the line. The highest complexity value is 30 for line number 31 since this line consists a loop and ten operators and operands. In other words, this line is most complex in its structure and size. On the contrary, complexity value is zero for lines 1, 2, 3, 16, 30, 35 since these lines have the simplest structure, which do not contain any operator or operand. Similarly, line 14 and 16 have function calls and therefore the complexity due to call is double in comparison to an ordinary program line (without any branching, iterations, or embedded systems).

4 Experimentation and Comparative Study

Empirical studies play an important role in the evaluation of software engineering discipline [1]. We have taken eight different 'C' programs from Misra and Mishra [5] for the analysis of the UCM approach. We calculated the Unified Complexity Measure (UCM) for each one of those programs (see Table-3). The complexity values for their components and UCM are also given in table 3. We observe from this table that the UCM values are high for programs whose program lines generally contain high value for any one of their components. Obviously, it is due the fact that UCM depends on the number of lines, operators, operands and cognitive weights.

We also used these sample programs to calculate the value of four different complexity measures, namely cognitive functional size complexity measure, effort measure, cyclomatic complexity and statement count, for comparative purposes (Table-4). Inspection of Table 4 states that the behavior of UCM is similar to the

Table 3. Calculated complexity values for UCM and its Components

No.	The Number of Lines (NL)	SOO	CW	UCM
1	12	20	4	50
2	17	35	3	57
3	18	52	3	71
4	37	58	16	136
5	23	25	10	79
6	15	20	6	57
7	11	10	6	43
8	11	17	9	73

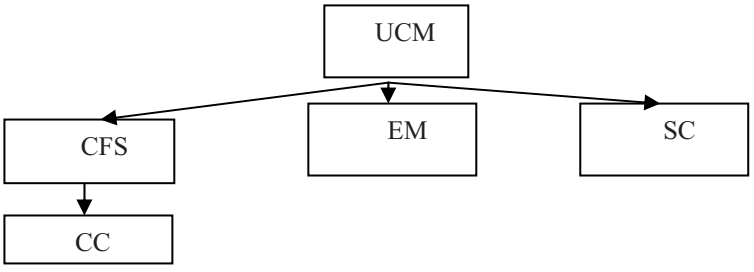

Fig. 1. UCM and other related complexity measures. CFS: Cognitive functional Size, EM: Effort Measure, SC: Statement Count; CC: Cyclomatic Complexity.

Table 4. Complexity values for different measures

Complexity Measures	Programs							
	Pgm.1	**Pgm.2**	**Pgm.3**	**Pgm.4**	**Pgm.5**	**Pgm.6**	**Pgm.7**	**Pgm. 8**
Statement Count	12	17	18	37	23	15	11	11
Cyclomatic Complexity	2	2	2	5	4	2	3	4
Effort Measure	1859	5191	6237	15556	5079	2869	1221	1039
Cognitive functional size	8	9	9	46	30	14	21	30
Unified Complexity Measure	50	57	71	136	79	57	43	73

other complexity measures. The higher values of UCM is due to the fact that the UCM includes most of the parameters of different measures. This means, the UCM can be assumed to be a superset (see fig 1.) of cognitive complexity, effort measure, cyclomatic complexity and statement count measures, which seems to be the most important advantage of UCM.

Interestingly, the inspection of Figure 2 states that the UCM and CFS show almost the same trend but the UCM has higher values. The relatively high values of UCM are because the UCM already includes the considerations of all cognitive aspects of CFS. Especially, **t**he highest value of UCM for the sample program 4 is due to the contribution of other factors i.e. larger size of the code, high cognitive complexity, high occurrences of operators and operands.

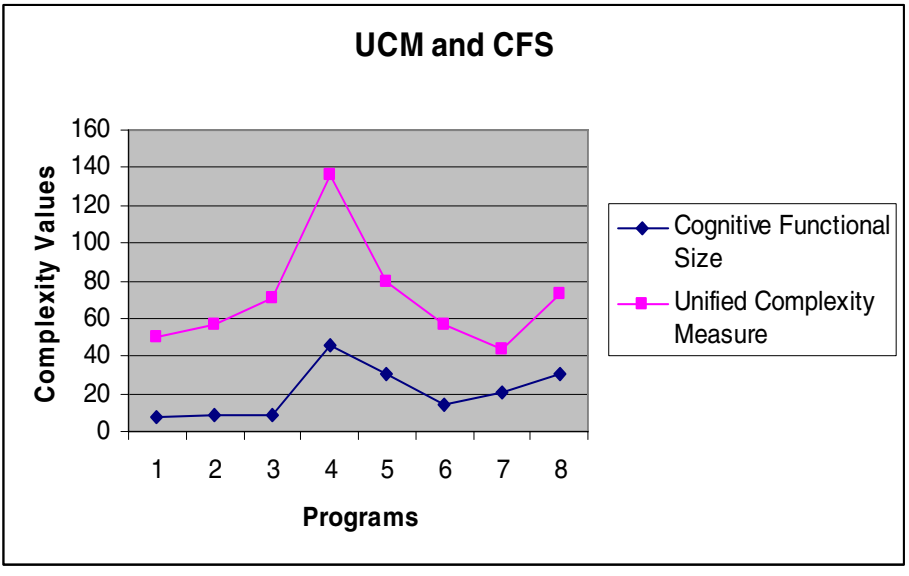

Fig. 2. Comparative Graph of UCM with CFS

5 Conclusion

In this paper, we proposed a metric by primarily considering all the internal attributes which directly affect the complexity. It uses number of lines (size), total occurrence of operators and operands (size), number of control structures (control flow structuredness) and function calls (coupling) as the internal attributes. The proposed metric also considers cognitive complexity since it is one of the important factors for increasing overall complexity and relates to comprehension. Understandability of software is the program comprehension and is a cognitive process. The cognitive complexity is used in terms of cognitive weights of basic control structures, which is also an indication of structural complexity. This means, the proposed metric is a unique model including all the factors responsible for increasing the complexity. The use of proposed metric is demonstrated by using a simple programming example. The practical applicability of the metric is evaluated by using eight different test cases which prove the soundness and robustness of the proposed measure. As a conclusion, we hope that the proposed metric, UCM, will aid the developers and practitioners in evaluating the complexity before and after coding.

References

1. Basili, V.: The Role of Controlled Experiments in Software Engineering Research. In: Basili, V.R., Rombach, H.D., Schneider, K., Kitchenham, B., Pfahl, D., Selby, R.W. (eds.) Empirical Software Engineering Issues. LNCS, vol. 4336, pp. 33–37. Springer, Heidelberg (2007)

2. Halstead, M.H.: Elements of Software Science. Elsevier North-Holland, New York (1997)
3. IEEE Computer Society: Standard for Software Quality Metrics Methodology. Revision IEEE Standard, 1061–1998 (1998)
4. McCabe, T.H.: A Complexity Measure. IEEE Transactions Software Engineering, 308–320 (1976)
5. Misra, S., Misra, A.K.: Evaluating Cognitive Complexity Measure with Weyuker's properties. In: Proc. of IEEE (ICCI 2004), pp. 103–108 (2004)
6. Misra, S.: Cognitive Program Complexity Measure. In: Proc. of IEEE (ICCI 2007), pp. 120–125 (2007)
7. Mills, E.: Software Metrics (2007), http://www.sei.UCMu.edu/publications/documents/UCMs/UCM.012.html
8. Oviedo, E.I.: Control flow, Data and Program Complexity. In: Proc. of IEEE COMPSAC, Chicago, IL, pp. 146–152 (1980)
9. Pressman, R.S.: Software Engineering: A Practitioner's approach, 5th edn. McGraw Hill, New York (2001)
10. Sommerville, I.: Software Engineering, 6th edn. Addison-Wesley, Reading (2001)
11. Wang, Y., Shao, J.: A New Measure of Software Complexity based on Cognitive Weights. Can. J. Elect. Comp. Eng. 28(2), 69–74 (2003)
12. Wang, Y.: The theoretical framework of cognitive informatics. International Journal of Cognitive Informatics and Natural Intelligence 1(1), 10–22 (2007)