International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

Underlying Cognitive Complexity Measure Computation with Combinatorial Rules

Benjapol Auprasert, and Yachai Limpiyakorn

***Abstract*—** Measuring the complexity of software has been an insoluble problem in software engineering. Complexity measures can be used to predict critical information about testability, reliability, and maintainability of software systems from automatic analysis of the source code. During the past few years, many complexity measures have been invented based on the emerging Cognitive Informatics discipline. These software complexity measures, including cognitive functional size, lend themselves to the approach of the total cognitive weights of basic control structures such as loops and branches. This paper shows that the current existing calculation method can generate different results that are algebraically equivalence. However, analysis of the combinatorial meanings of this calculation method shows significant flaw of the measure, which also explains why it does not satisfy Weyuker's properties. Based on the findings, improvement directions, such as measures fusion, and cumulative variable counting scheme are suggested to enhance the effectiveness of cognitive complexity measures.

***Keywords*—**Cognitive Complexity Measure, Cognitive Weight of Basic Control Structure, Counting Rules, Cumulative Variable Counting Scheme.

I. Introduction

Measuring the complexity of software has been an insoluble problem in software engineering. Many software complexity measures have been proposed and evolved for ages. The basis of all these measures lies on the discipline of counting, as the initial step of most measures is usually "counting some elements of the software". Combinatorics is a branch of pure mathematics concerning the study of counting. It is frequently used in computer sciences to obtain estimates of the number of elements of certain sets. As all complexity measures involve counting, they should be explainable with combinatorics proof.

Cognitive Informatics [13], [14], [15] is an area of studying the internal information processing mechanisms of the brain and the processes involved in perception and cognition. During the past few years, many researchers have been integrating Cognitive Informatics to derive software complexity metrics so called *cognitive complexity* measures [1], [2], [3]. The approach has been considered as one of the promising solutions to measure the complexity of software. Complexity measures can be used to predict critical information about testability, reliability, and maintainability of software systems from automatic analysis of the source code. Recently, one of the emerging research areas has focused on cognitive complexity metrics to reflect the software complexity. The key part of determining the cognitive complexity relies on the total cognitive of basic control structures (BCS's) e.g. sequence, if-else, switch-case, while-loop, for-loop, etc. The calculation of the total cognitive weights of the basic control structures of software seems to resemble the counting rules in Combinatorics, i.e. the rule of sum, and the rule of product. This research, thus, construes the calculation of total cognitive weights as combinatorial meanings.

In Combinatorics, when the two approaches to counting something are combinatorial equivalence, they should be explainable by the double counting (two-way counting) proof technique [7], [8], [9]. This research attempts are to find a new algebraically equivalence way to calculate the total cognitive weights of the BCS's, and to analyze the combinatorial meanings of the two ways in order to reveal some major problems inherent in the existing cognitive complexity measures. and make suggestions for future improvement.

The contents in this paper are organized as follows: section 2 briefly explains the counting rules in Combinatorics. The calculation of a cognitive complexity measure is then described in section 3, followed by proposing the alternative calculation of the total cognitive weights in section 4, and analyzing the corresponding combinatorial meanings in section 5. Suggestions for future improvement are provided in section 6. Section 7 finally concludes the work in this paper.

.

II. Counting Rules in Combinatorics

In Combinatorics, there are two basic counting rules:– the rule of sum, and the rule of product. Counting rules are the foundation of any matters involving counting.

The rule of product [10] states that "the number of ways to do a procedure that consists of two subtasks is the product of the number of ways to do the first task and the number of ways to do the second task after the first has been completed".

B. Auprasert is with the Department of Computer Engineering, Chulalongkorn University, Bangkok 10330, Thailand (corresponding author to provide e-mail: 51703650@student.netserv.chula.ac.th).

Y. Limpiyakorn is with the Department of Computer Engineering, Chulalongkorn University, Bangkok 10330, Thailand (e-mail: Yachai.L@chula.ac.th).

This research is funded by Software Industry Promotion Agency (SIPA) that has collaborated with Chulalongkorn University for the Software Quality Research and Development Project.

This rule indicates that "multiplication" is used when the two sets we are counting, are dependent on each other. Applying this rule to counting the cognitive complexity implies that the total cognitive complexity of two blocks of software code should be calculated from the product of the amount of the cognitive complexity of each block if and only if the understanding of a particular block of code requires the preceding comprehension of the other block.

The rule of sum [10] states that "the number of ways to do a task in one of the two ways is the sum of the number of ways to do these tasks if they cannot be done simultaneously".

This rule reflects a fact about set theory. It states that "addition" is used when the two sets we are counting, are disjoint. Applying this rule to counting the cognitive complexity implies that the total cognitive complexity of two blocks of software code should be computed from the sum of the amount of the cognitive complexity of each block if and only if to comprehend each block does not require the understanding of the other block at all.

III. Total Cognitive Weights of Basic Control Structures

In 2003, Wang and Shao [1] proposed cognitive functional size (CFS) as a software complexity measure based on W_c - the total cognitive weights of Basic Control Structures (BCS's) of software. W_c is defined as the total sum of cognitive weights of its q linear blocks composed in individual BCS's. Since each block may consist of 'm' layers of nesting BCS's, and each layer with 'n' linear BCS's,

$$W_c = \sum_{j=1}^{q} \left[\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right] \quad (1)$$

where weights W_c (j,k,i) of BCS's were initially proposed as in TABLE I.

TABLE I. Cognitive Weights (W_c) of BCS's

Category	BCS	W_i
Sequence	Sequence (SEQ)	1
Branch	If–Then-Else (ITE)	2
	Case (CASE)	3
Iteration	For-do (R_i)	3
	Repeat-until (R_1)	3
	While-do (R_0)	3
Embedded Component	Function Call (FC)	2
	Recursion (REC)	3
Concurrency	Parallel (PAR)	4
	Interrupt (INT)	4

W_c has been a remarkable breakthrough in software engineering that inspires tremendous new ideas for measuring the software because it is independent from implementation technologies, easy to calculate, and based on a lot of sound Cognitive Informatics principles. Many cognitive complexity measures have been proposed based on W_c , for example :

Wang's CFS [1] is defined as

$$CFS = (N_i + N_o) * W_c \quad (2)$$

Kushwaha and Misra's CICM [3] is defined as

$$CICM = WICS * W_c \quad (3)$$

where WICS is the weighted information count of the software derived from:

$$WICS = \sum_{k=1}^{LOCS} \{\#(\text{identifiers and operators in the } k^{th.} \text{ line}) / (LOCS-k)\} \quad (4)$$

Wang's modified $C_c(S)$ [2] is defines as

$$C_c(S) = f(\text{data objects}) * W_c \quad (5)$$

where f (data objects) is the function that counts the number of global and local data objects such as inputs, outputs, data structures, and internal variables.

IV. Algebraically Equivalence Total Cognitive Weights

Based on the calculation of the total cognitive weights of BCS's described in section 2, the Distributive Property in Algebra, i.e. "a (b + c) = ab + ac", can be used to re-arrange the terms into an alternative way to calculate the W_c .

```
Program A{
   Statement1;
   If (condition1){...} else {...};
   While (condition2){
      For {...};
      Statement2;
      Statement3;
      While (condition3){
         If (condition4) {...}
      }
   }
}
```

Fig. 1. BCS's structure of sample program

From the sample program's structure presented in Fig. 1, W_c from Wang's proposed method can be calculated as below:

$$W_c = W_{sequence} + W_{if} + [W_{while} * \{W_{for} + W_{sequence} + (W_{while} * W_{if})\}] \quad (6)$$

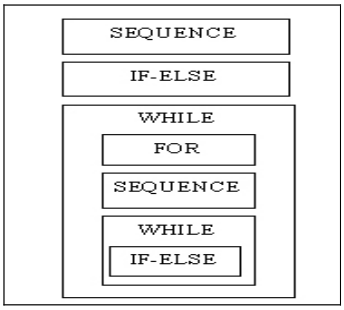

Fig. 2. The structure of BCS's of program in Fig. 1.

Applying the distributive property to the term $[W_{while} * \{W_{for} + W_{sequence} + (W_{while} * W_{if})\}]$ in Equation (6), an alternative way to calculate W_c results in:

$$W_c = W_{sequence} + W_{if} + W_{while} * W_{for} + W_{while} * W_{sequence} + W_{while} * W_{while} * W_{if} \quad (7)$$

In general, from Wang's proposed calculation method in Equation (1), the cognitive weights are summed up from the inner structures out, layer by layer, recursively. Whereas in our alternative way, W_c can be computed by finding the basic control structures within which do not contain any basic control structures, multiplying their weights by the weights of all their outer BCS's, then summing them altogether. This implies that algebraically, the program structure shown in Fig. 3 has the same total cognitive weights as that of the program structure in Fig. 2.

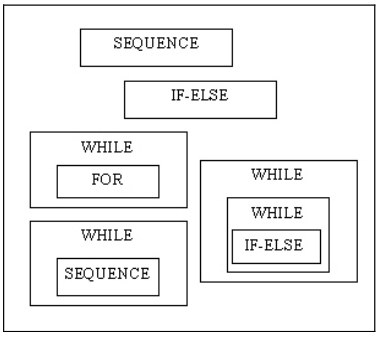

Fig. 3. Another structure with W_c algebraically equivalent to that of structure in Fig. 2

V. Combinatorial Meaning Analysis of Total Cognitive Weights Computation

Since the W_c of the structures in Fig. 2 and Fig. 3 are proved equivalence algebraically in section 4, the count of total cognitive weights should be explainable by the double counting technique in Combinatorics.

Fig. 4 shows the program of which the BCS's align with the structure depicted in Fig. 3. The BSC's structure of program in Fig. 4 is also equivalent to that of program in Fig. 1 as being derived from the calculation of the total cognitive weights. It is obvious that the program structures in Fig. 2 and Fig. 3 should not be equivalent in the cognitive complexity perspective, otherwise, the effort used to comprehend the programs in Fig. 1 and Fig. 4 are indifferent.

```
Program{

  Statement1;

  If (condition1){...} else {...};

  While (condition2){
     For {...}
  }

  While (condition3){
     Statement2;
     Statement3;
  }

  While (condition4){
     While(condition5){
        If (condition6) {...}
     }
  }

}
```

Fig. 4. BCS's structure of program derived from alternative total cognitive weights computation equivalent to that of Fig. 1.

To inspect if any mistakes exist in the calculation, we examined the combinatorial meanings underlying the calculation method. The weight of each BCS, as shown in TABLE I, is defined as "relative effort spent on comprehending the function and semantics of a BCS against that of the sequential BCS" [2], while the total cognitive weights is defined as "the extent of relative difficulty or effort spent in comprehending the software" [2].

When something can generate more possible different combinations, they need more effort for comprehension. To quantify the abstraction of counting the total cognitive weights, in this paper the "effort used to comprehend each BCS" is considered as the number of ways that BCS can generate some factors that make it difficult to comprehend. The cognitive weight of a BCS is measured as the number of ways that BCS can generate some factors that make it difficult to comprehend relative to the sequential BCS, of which the cognitive weight is '1'. The value of total cognitive weights of the software is measured as the number of relative ways that software can generate the combination of factors that make the function and semantics difficult to comprehend.

Based on these definitions, the rule of sum and the rule of product in Combinatorics are applied to counting the total cognitive weights. The use of 'multiplication' with the

weights of nested BCS's implies that to understand the whole nested BCS's structure, it is required to fully understand the whole contents in the inner BCS's first, then understand the outer ones surrounding it, layer by layer from inside out. This seems reasonable compared to the use of 'addition' with linear BCS's structures, which implies that the cognition of these BCS's are completely disjoint. In other words, BCS's in linear structure and the contents they contain within can be understood separately and simultaneously.

In fact, it is observed that even the linear BCS blocks are not necessarily understandable without others. Hence, they cannot be understood simultaneously. The counter example is shown in Fig. 5 when there are some variables in the previous blocks that have effect on the following blocks.

```
f (int a){

   int fac =1;

   while (a>0){
      fac = fac * a;
      a = a-1;
   }

   if (a>20000) return true;
   else return false;

}
```

Fig. 5. Counter example of linear BCS's against simultaneously comprehensible

It can be seen from Fig. 5 that the "while" block and "if" block are posed linearly. However, it is clearly that the intents of the two blocks cannot be understood simultaneously. The descending "if" block cannot be understood without the comprehension of the variable existing in the preceding "while" block. The variable *a* seems to transfer the complexity of one block to another and it disproves the assumption that the complexity of the blocks in linear structure are disjoint.

The finding reveals the major weakness of the total cognitive weights of software that it does not consider the possible data flow from one BCS's block to another. This incident could carry the complexity from one block to another, making them incomprehensible simultaneously as evident by the use of "addition" with the weights of BCS's in linear structures. Moreover, the existing method to compute W_c assigns the same complexity to each BCS's of the same kind. For example, the "while" blocks are always considered as equally difficult to understand no matter how many different numbers of variables contained within as long as they do not contain any BCS's. For these reasons, we can find no reasonable double counting proof to show the equivalence of the two calculation methods in section 4, because they are not really equivalent. The algebraically equivalence only happened by chance because the definition of W_c is based on the assumption that the complexity inside one block cannot be transferred to another block in linear structure.

A. Weyuker's properties and cognitive complexity measure

Weyuker's properties [4] consist of nine properties of syntactic software complexity measures widely used as criteria for evaluating software measures. However, many classical complexity measures, such as LOC, McCabe Cyclomatic number, Halstead's effort, fail to satisfy some of these properties as shown in TABLE II [6].

TABLE II. EVALUATION OF COMPLEXITY MEASURES AGAINST WEYUKER'S PROPERTIES

Property	LOC	McCabe's Cyclomatic	Halstead's Effort	Dataflow Complexity	CFS
1	/	/	/	/	/
2	/	X	/	X	/
3	/	/	/	/	/
4	/	/	/	/	/
5	/	/	X	X	/
6	X	X	/	/	X
7	X	X	X	/	/
8	/	/	/	/	/
9	X	X	/	/	/

As proved in [5], [6] and shown in TABLE II, the Cognitive Functional Size (CFS) [1], [2] brilliantly satisfies eight of these properties, that is:

Let P and Q be a program body.

Property 1. $(\exists\ P)\ (\exists\ Q)\ (\ |P| \neq |Q|\ )$

Property 2. Let c be a non negative number, then there are only finitely many programs of complexity c.

Property 3. There are distinct program P and Q such that $|P| = |Q|$

Property 4. $(\exists\ P)\ (\exists\ Q)\ (P \equiv Q\ \ \&\ \ |P| \neq |Q|)$

Property 5. $(\forall\ P)\ (\forall\ Q)\ (|P| \leq |P;Q|\ \&\ |Q| \leq |P;Q|)$

Property 7. There are some program bodies P and Q such that Q is formed by permuting the order of statements of P, and $|P| \neq |Q|$

Property 8. If P is renaming of Q, then $|P| = |Q|$

Property 9. $(\exists\ P)\ (\exists\ Q)\ ((\ |P| + |Q|\ ) <\ |P;Q|\ )$

However, CFS fails to satisfy *Property 6*, which states that:

Property 6a. $(\exists\ P)(\exists\ Q)(\exists\ R)(\ (|P|=|Q|)\ \&\ (\ |P;R| \neq |Q;R|)\ )$

Property 6b. $(\exists\ P)(\exists\ Q)(\exists\ R)(\ (|P|=|Q|)\ \&\ (\ |R;P| \neq |R;Q|)\ )$

The combinatorial analysis in the previous section clearly explains the reason why it fails to satisfy this property. This is because the calculation of the total cognitive weights implies that the complexity of blocks in linear structure can be assessed separately and simultaneously.

It is found that the measures, which satisfy this property, must take into account of the possibility of the complexity flowing between blocks of BCS's. That is, suppose program blocks P and R be in linear structure, when there are some

identifiers in R that depend on the processing in P, the complexity in understanding P; R should be regarded more than the sum of complexity of P and R individually (|P;R| > |P|+|R|). However, when P and R are completely independent, the complexity in understanding P; R should be the same as the complexity in understanding them simultaneously (|P;R| = |P|+|R|).

VI. Suggestions for Future Improvement

According to the analysis in section 5, this section proposes some improvements on the cognitive complexity measures to enable satisfying all nine Weyuker's properties.

A. Complexity Measures Fusion

From TABLE II, the two measures, i.e. Halstead's effort measure, and data flow complexity measure manage to satisfy Property 6, while CFS lacks. Therefore, integrating the principles of these measures into the calculation of W_c would result in the measure satisfying all nine properties. Here, the data flow complexity [11] seems to fit for improving CFS, since the method's concept is to measure the possibility for control to transfer from one program block to another by counting "the number of variable definitions which reach the block" as the complexity of that block. This is precisely what CFS fails to cover as being analyzed in the previous sections.

1) Oviedo's data flow complexity measure

Oveido's [11] is a software complexity measure based on the data flow characteristics of the program, defined as per below:

"A program can be uniquely decomposed into a set of disjoint blocks of ordered statements having the property that whenever the first statement of the block is executed, the other statements are executed in the given order. Furthermore, the first statement of the block is the only statement which can be executed directly after the execution of a statement in another block. Intuitively, a block is a chunk of code which is always executed as a unit."

A program flow graph is a directed graph in which each node corresponds to a block of the program and the edges correspond to the program branches. If the nodes ni and nj of the flow graph correspond to the program blocks ni and nj then there is an edge (ni , nj) from node ni to node nj if it is possible for control to transfer directly from block ni to block nj in the program.

A variable definition takes place in a PROGRAM statement or in an assignment statement. A variable reference takes place when the variable is used in an expression (i.e., in an assignment statement or predicate) or an OUTPUT statement.

A locally available variable definition for a program block is a definition of the variable in the block. A locally exposed variable reference in a block is a reference to a variable which is not preceded in the block by a definition of that variable.

A variable definition in block ni is said to reach block nk if the definition is locally available in block ni and there is a path from ni to nk (i.e., nk is a successor of ni) along which the variable is not locally available in any block on the path, i.e. the variable is not redefined along that path. A variable definition in a block overrides all other definitions of this variable that might otherwise reach the block.

Data flow complexity of block i is defined as "the sum of the numbers of available definition of variables whose references are locally exposed in block i" [4], [11].

Measuring the data flow complexity of each BCS block along with the cognitive weight of that BCS would help eliminate the sense that the measure does not consider the complexity of each block as disjoint.

2) Halstead's Software Metrics

Halstead [12] proposed a set of software metrics for measuring the algorithmic complexity by counting operators and operands from software codes. Let

n_1 = number of distinct operators,
n_2 = number of distinct operands,
N_1 = total number of operator occurrences, and
N_2 = total number of operand occurrences.

Based on the above notations, the definition of Halstead's measures can be summarized as displayed in TABLE III

TABLE 3. Definitions of Derived Measures of Halstead's Software Metrics [1], [12]

Measure	Symbol	Formula
Program length	N	$N = N_1 + N_2$
Program vocabulary	n	$n = n_1 + n_2$
Volume	V	$V = N*(\log_2 n)$
Estimated abstraction level	L	$L= (2\ n_2) / (n_1*N_2)$
Difficulty	D	$D= 1 / L$
Effort	E	$E = V * D$
Time	T	$T = E / 18$
Remaining bugs	B	$B = E^{2/3} / 3000$

Using Halstead's metrics to measure the operators and operands inside each BCS along with the cognitive weight of that BCS would help eliminate the sense that the measure considers all BCS's of the same kind as having the same cognitive complexity no matter what they contain inside.

B. Cumulative Variables Counting Scheme

Another possible resolution for the issue of complexity that may flow between BCS's blocks when calculating W_c is to

count the number of variables in the BCS along with the weight of BCS itself. Moreover, since the difficulty in comprehending the variables increases as they appear more and more in the program, we then not only count the number of times they appear in that BCS, but also cumulate the number of times they appear in the program up until that BCS block. In other words, when a variable appears later in the program, the value stored inside it is dependent on its preceding occurrences in the program, hence it cumulates the complexity of its preceding occurrences.

Based on this cumulative variables counting scheme, we can quantify the complexity of each BCS's block in linear structure separately and simultaneously, as the possibility of the transfer of complexity between blocks is already weighted by the counting scheme.

VII. Conclusion

As counting occurrences is one of the basis operations of measurement, the counting rules in Combinatorics are used in this work to reveal the flaws in the calculation of the total cognitive weights of basic control structures (BCS's), which is used in many current cognitive complexity measures of software. The mentioned calculation approach assumes that when two BCS's are located in linear structure, they can then be comprehended independently and simultaneously. This assumption causes the complexity of the programs that have different control structures and should not be always equivalent to become equivalent algebraically according to the means the total cognitive weights are calculated. This is also the reason why one of the well-known cognitive complexity measures, the Cognitive Functional Size, still misses one of the nine Weyuker's properties. It defeats some other complexity measures by satisfying eight of the Weyuker's properties, though.

It is obvious that the BCS blocks are not independent from each other, even though they are posed in linear structure. This is because the variables can carry the complexity from one block to another. Therefore, the complexity when try to understand the linear BCS's chunks cannot be evaluated separately and simultaneously as implied by the calculation of the total cognitive weights. Suggestions for the improvement would be to include the data flow complexity in the calculation, and the propose of cumulative variables counting scheme as the attempts to fortify the cognitive complexity measures to satisfy all nine Weyuker's criteria, and more precisely and rigorously reflect the effort spent to comprehend the software.

References

[1] Yingxu Wang and Jingqiu Shao, "Measurement of the Cognitive Functional Complexity of Software", Proceedings of the 2nd IEEE International Conference on Cognitive Informatics, p.67, August 18-20, 2003

[2] Yingxu Wang, "Cognitive Complexity of Software and its Measurement," *Cognitive Informatics, 2006. ICCI 2006. 5th IEEE International Conference on* , vol.1, no., pp.226-235, 17-19 July 2006

[3] Dharmender Singh Kushwaha , A. K. Misra, "A modified cognitive information complexity measure of software", ACM SIGSOFT Software Engineering Notes, v.31 n.1, January 2006

[4] E. J. Weyuker, "Evaluating Software Complexity Measures, IEEE Transactions on Software Engineering", v.14 n.9, p.1357-1365, September 1988

[5] Sanjay Misra , A. K. Misra, "Evaluation and comparison of cognitive complexity measure", ACM SIGSOFT Software Engineering Notes, v.32 n.2, March 2007

[6] Misra, S. and Misra, A.K "Evaluating cognitive complexity measure with Weyuker properties", Cognitive Informatics, 2004. Proceedings of the Third IEEE International Conference on, 16-17 Aug. 2004

[7] AT Benjamin, JJ Quinn, Proofs that Really Count: The Art of Combinatorial Proof, Washington, DC: Mathe-matical Association of America, 2003.

[8] Aigner, Martin; Ziegler, Günter. Proofs from THE BOOK. Berlin; New York: Springer, 2003.

[9] M. H. A. Newman, "On Theories with a Combinatorial Definition of "Equivalence"", The Annals of Mathematics, Second Series, Vol. 43, No. 2 (Apr., 1942), pp. 223-243

[10] Joe Sacada, "The Basic of Counting", Lecture Notes: CIS 2910: Discrete Structures in Computer Science II, University of Guelph

[11] E. I. Oviedo, "Control flow, data flow, and program complexity," Proceedings of COMPSAC, pp.146-152., 1980

[12] Maurice H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., New York, 1977.

[13] Wang, Y., "On Cognitive Informatics" , Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy, 4(2), pp.151-167, 2003.

[14] Yingxu Wang. "On the Cognitive Informatics Foundations of Software Engineering", Proceedings of the Third IEEE International Conference on Cognitive Informatics 2004

[15] Yingxu Wang , "On the informatics laws of software", Cognitive Informatics, 2002. Proceedings. First IEEE International Conference on, 2002.