An Advanced Cognitive Complexity Metric for Cohesion Factor

N. Vijayaraj
Assistant Professor
Department of Computer Science
Srimad Andavan Arts and Science College, India.
vijay_sjctni@yahoo.co.in

Dr. T.N. Ravi
Assistant Professor
Department of Computer Science
Periyar E.V.R. College, India.
proftrnri@gmail.com

Abstract—Software metrics plays a major role in assessing the quality of the software in testing process. Software metrics elucidates the complexity, reusability, maintainability and understandability of the software code. Cognitive complexity metrics are one of the emerging types of software metrics that focuses on the cognitive analysis of software in terms of understandability and maintainability. In other words, it can be rephrased as the effort taken to comprehend the program code for future enhancements. Cognitive complexity metrics has a direct impact with the analysis of complexity in software through an intrinsic study on the object oriented features. This paper proposes a novel Cognitive Complexity Metric for Cohesion Factor (CCMCF), to highlight the cognitive features with cohesion and weighs the complexity of a class.

Keywords: Cognitive complexity, software metric, cohesion, software maintainability, software reusability.

I. INTRODUCTION

Software metric has been used as an important criterion for assessing the quality of software product. The major types of software metrics are broadly classified into procedure and object oriented. The blooming features of object oriented programming such as modularity, maintainability, modifiability and understandability have propelled the development of OO software in the present decade. Thus, the proposal of newer software metrics for measuring the features of object oriented programming is the central focus on the software engineering research.

Modularity in OO software is highly attainable through cohesion factor. High cohesion increases high integrity of methods in module and vice versa. The analysis of cohesion factor in software modules is yet to be improved with cognitive weights for predicting the time taken for understanding the software. This paper assigns cognitive weights for shared variables between the methods. The more the weights of the variable increases, the more the cohesiveness of the class is.

The remaining section of the paper is organized as follows: Section II contains review of literature, section III describes the proposed methodology, Section IV consists of the calibration of proposed CCMCF, Section V explains the theoretical evaluation of CCMCF, Section VI describes the comparative analysis of CCMCF with traditional LCOM and final Section VII explains the summary of this research work.

II. REVIEW OF LITERATURE

Misra et al. [1] have proposed a three OO complexity metrics namely, Coupling Weight for a Class (CWC), Attribute Complexity (AC) and Weighted Class Complexity (WCC) Metric. CWC is defined with an assumption that two classes are coupled if and only if there exists a message call from one class to the other. CWC adds the weights of internal and external message calls to CWC, rather than counting the total number of calls. Here, the complexities due to message calls are assessed by summing the weights of call and the weights of called methods, which can be formulated as shown in equation 1.

\[ CWC = \sum_{i=1}^{n} (2 + MC_i) \]  \( (1) \)

Where, the number 2 is the weight of message to an external method and MC_i is the weight of called method. If the number of external calls is ‘n’, then CWC is computed as sum of weights of all message calls.

AC is designed with the principle that the complexity of a class is high, if it contains more number of attributes. The attributes that instantiates an object used in one method may not be used in other methods increases the complexity of a class. The weight of AC is the total number of attributes associated in the class which can be denoted as shown in equation 2.
\[ AC = \sum_{j=1}^{n} 1, \]  \hspace{1cm} (2)

Where \( 'n' \) is the total number of attributes in the class.

The structure of OO programming purely depends upon classes and objects whose elements are methods and attributes. The complexity of the class is measured by the number of attributes and the methods that exist in the class. Hence, the WCC of a class is the sum of attribute weights and method weights of the corresponding class. The formula for calculating the WCC is denoted as shown in equation 3.

\[ WCC = AC + \sum_{j=1}^{n} MC_j \]  \hspace{1cm} (3)

Where, WCC is the sum of attribute complexity and sum of all the method complexities of class.

Arockiam et al. [2] have proposed a complexity metric, Attribute Weighted Class Complexity (AWCC) that assesses the complexity of inherited classes. The complexity of a class with AWCC is calculated using the method and attribute complexities with the complexity of the inherited members. Supposing, if a class holds \( 'm' \) attributes and \( 'n' \) methods and the class is derived from \( 'O' \) number of classes then, the AWCC of the corresponding class can be calculated as shown in equation 4.

\[ AWCC = \sum_{x=1}^{m} AC_x + \sum_{y=1}^{n} MC_y + \sum_{z=1}^{n} ICC_x \]  \hspace{1cm} (4)

Where, AC is the attribute complexity

MC is the method complexity

ICC is the inherited class complexity

The attribute complexity of AWCC is the sum of multiplication of data type weights with the number of attributes belonging to the data type, which can be denoted using the formula denoted as equation 5.

\[ AC = (PDT \times W_d) + (DDT \times W_d) + (UDDT \times W_u) \]  \hspace{1cm} (5)

Where, PDT is the number of attributes belonging to primary data type

DDT is the number of attributes belonging to derived data type

UDDT is the number of attributes belonging to user-defined data type

\( W_d \) is the weight for PDT which is 1

\( W_d \) is the weight for DDT which is 2

\( W_u \) is the weight for UDDT which is 3

The MC is calculated as defined by Misra et al. and ICC

The formula for calculating the ICC is denoted in equation 6.

\[ ICC = (DIT \times C_L) \times \sum_{x=1}^{n} RMC_{e} + RNA \]  \hspace{1cm} (6)

Where DIT denotes the depth inheritance Tree metric

\( C_L \) is the cognitive Load of level L

\( S \) is the number of inherited methods

RNA is the total number of reused attributes

RMC is the reused method complexity

ICC is the inherited complexity

Arockiam et al. Cognitive Weighted Response For a Class (CWRFC) metric for measuring the complexity involved in message passing [3]. Supposing if a class holds \( 'n' \) number of response sets CWRFC calculates the complexity of the class using the response set complexity as shown in equation 7.

\[ CWRFC = \sum_{i=1}^{n} RSC_i \]  \hspace{1cm} (7)

Where RSC denotes the response set complexity, which is calculated by summing the set of all m methods in a class and set of R methods called by any of those methods.

\[ RSC = \forall f R_i + M \]  \hspace{1cm} (8)

As per message passing, the methods of the classes are segmented into two such as, Methods With Arguments (MWA) and methods without arguments (MOA). MOA is also referred as Default Function (DF). The arguments of MWA can either be passed through Pass By Value (PBV) or Pass By Reference (PBR). Hence, R can be computed using the formula shown in equation 9.

\[ R = DF \times (CW_{f} + WF_{f}) + PBV \times (CW_{f} + WF_{f}) + \]  \hspace{1cm} (9)

where, DF is the total number of default functions

PBV is the total number of Pass By Value Function Call Statements

PBR is the total number of Pass By Reference Function Call Statements

\( CW_f \) is the CWs of the Function Call Statement

\( WF_f \) is the Weighting Factor of the DFCS

\( WF_f \) is the Weighting Factor of the PBV statements

\( WF_f \) is the Weighting Factor of the PBR statements

Arockiam et al. proposed Cognitive Weighted Coupling Between Objects (CWCBO) metric to elucidate the complexity involved with coupling of classes by considering the different types of coupling such as control, data, interface and global couplings [4]. The unnecessary object coupling increases the complexity of the chances of system exploitation. CWCBO can be calculated using the equation 10.

\[ CWCBO = ((CC \times WFCC) + (GDC \times WFGDC) + (IDC \times WFDIC) + (DC \times WFDC) + (LCC \times WFLCC) \]  \hspace{1cm} (10)

Where

CC is the total number of modules that contains Control Coupling

WFCC is the Weighting Factor of Control Coupling

GDC is the count of Global Data Coupling

WFGDC is the Weighting Factor of Global Data Coupling and its weight is given as 1

IDC is the count of Internal Data Coupling

WFIDC is the Weighting Factor of Internal Data Coupling and its weight is given as 2

DC is the count of Data Coupling

WFDC is the Weighting Factor of Data Coupling and its weight is given as 3
III. METHODOLOGY

In object oriented programming cohesion refers to the degree of intra-relatedness of functions within a module, which means the functions of a module must be goal specific and should focus on a single well-defined task. A module is said to be fair, when all of its attributes are shared among the functions. Metrics in software cohesion measures the complexity of the program module in such a way that high cohesion represents low complexity and low cohesion refers to high complexity. Cognitive metrics in object oriented programming assigns numerical weights to OO features for highlighting their intrinsic specification or characteristics. This paper combines the advantages of cohesion and cognitive metrics to identify the cohesion type hidden in the module. A module with high cohesion is highly preferable than a module with low cohesion. Thus, the proposed CCMCF assess the existence of cohesion through cognitive weights. The CCMCF can be calculated as follows:

$$CCMCF = \frac{\sum_{i=1}^{n} \text{cognitive weights (V)} }{f^v}$$

Where, CCMCF is the Cognitive complexity metric for cohesion factor.

Cognitive of Weights (V) is the weight of variables 1...n, when the variables occurs for the first time in a method, the weight is one and gradually increases. If a variable is used multiple times in a method, only one copy of the variable is taken. f is the total number of functions in the module v is the total number of variables in the module

Illustration

Pseudo Code 1 – arithmetic.cpp

1. Create a module with the name arithmetic
2. Declare variables three variables a, b and c
3. Create object for module for arithmetic
4. Call input() method
4.1. Read inputs for a and b from user
5. Call add() method
5.1. Add variables a and b
5.2. Store the result in c
5.3. Print c
6. Call sub() method
6.1. Subtract variable b from a
6.2. Store the result in c
6.3. Print c
7. Call mul() method
7.1. Multiply variables a and b
7.2. Store the result in c
7.3. Print c
8. End
Pseudo Code 2 – arithmetic1.cpp
1. Create a module with the name arithmetic1
2. Declare nine variables namely a, b, c, d, e, f, g, h and i
3. Create object for arithmetic1
4. Call add () method
   4.1. Read inputs for the variables a and b
   4.2. Add the variables a and b
   4.3. Store the result in c
   4.4. Print c
5. Call sub () method
   5.1. Read inputs for the variables d and e
   5.2. Add the variables d and e
   5.3. Store the result in f
   5.4. Print f
6. Call mul () method
   6.1. Read inputs for the variables g and h
   6.2. Add the variables g and h
   6.3. Store the result in i
   6.4. Print i
7. End

IV. CALIBRATION OF CCMCF

Pseudo code 1-arithmetic.cpp
The weights of pseudo code 1 are computed as follows:
Since the variables a and b are used in four methods of
the arithmetic class, the weights of the variable is 4 for
each and the variable c is used twice in three methods,
but still the weight of c is taken as 1 for each three
methods, thus the weight of c is 3. To interpret the
weights in the proposed metric we get

\[ CCMCF = \frac{4 + 4 + 3}{3} = \frac{11}{3} = 0.91 \]

Program 1 depicts a good representation of cohesion
class since the CCMCF value is 0.91

Pseudo code 2-arithmetic1.cpp
The weights of pseudo code 2 are computed as follows:
The variables of program 2 are used is one of the
functions of a class. Thus, the weights are 9 and the
total number of functions in the class is 3. Thus, the
CCMCF metric value for program 2 be.

\[ CCMCF = \frac{1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1}{9 * 3} = \frac{9}{27} = 0.33 \]

Program 2 depicts a poor representation of cohesive
class, since the CCMCF value is only 0.33.

V. THEORETICAL EVALUATION OF CCMCF

Many inventions have suggested that the software
metric should satisfy certain properties for their real
time usability in software testing. Basili and Reiter [7]
suggested that software metrics should be sensitive to
external observable differences in development process,
and should correspond to intuitive notions about the
characteristic differences between the software artifacts
being measured. Weyuker has also proposed an
authorized list of properties for software metrics that
could be evaluated on the existing software metrics [8].
The notions of the Weyuker’s properties include
permutation, interaction, monotonicity, non-
coarseness. Many researchers have recommended various properties
uniqueness and so on. The challenge in this section is to
evaluate the proposed CCM against the nine properties
of Weyuker’s to prove its usefulness.

Though, several Weyuker’s properties are
considered to be most significant to classify the
complexity of a measure. Weyuker’s properties state

Property 1
Non-coarseness:
Not all class can have the same complexity. If there are
‘n’ numbers of modules in the software, CCMCF does
not rank all ‘n’ modules as equally complex.

Property 2
Granularity: Let ‘r’ be a non-negative number and there
could be only finite number of modules have the
complexity r. If the number of modules in large scale
system is finite, the complexity value of CCMCF is also
finite. Hence this property is satisfied.

Property 3
Non-uniqueness: This property implies that there may
be number of modules have the same complexity.
CCMCF abides this property, if the hierarchies of class
in the modules are similar.

Property 4
Design details are important:
The property affirms that though if two classes have the
same functionality, they may differ in terms of details of
implementation. If the design implementation of two
modules is different, CCMCF produces different
complexity values for each module.

Property 5
Monotonicity:
Let the concatenation of two modules R and S be R+S.
Hence, this property states that complexity value of the
combined class may be larger than the complexity of the
individual classes R or S. CCMCF abides this property
if there is a possibility of inheritance between the
modules R and S while concatenation.

Property 6
Non-equivalence of interaction:
This property states that if a new module is added to the
two existing modules R and S which has the same
module complexity, if a new module T is added with
both modules, the module complexities of the two new
combined modules may be different or the interaction
between R and T may be different than the interaction
between S and T resulting in different complexity
values for R + T and S + T. CCMCF for sure yields different complexity values for both modules R and S since T is dependent on the fitness of inheritance with the existing modules R and S.

Property 7
Permutation: There are program bodies I and J such that J is formed by permuting the order of the statements of I and (|I| = |J|). This property is not taken into the consideration of object oriented metrics.

Property 8
Renaming: If module R is renamed as S then |R| = |S|. This property requires that renaming a module should not affect the complexity of the module. CCMCF does not have any impact over the change of name of module, hence CCMCF satisfies property 8.

Property 9
Interaction increases complexity: The property says that the class complexity measure of a new class combined from two classes may be greater than the sum of two individual class complexity measures. This property is not satisfied with CCMCF as the complexity of combined modules could be possibly equal to the individual complexity but not greater. Summary of the CCMCF validation is described in Table 1.

Table 1. Evaluation of CCIM Metric with Weyuker’s Properties

<table>
<thead>
<tr>
<th>Metric</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>P9</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCIM</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

VI. COMPARATIVE ANALYSIS OF CCMCF WITH LCOM

The proposed CCMCF metric is compared with its base LCOM metric to highlight the usability of cognitive weights to measure the cohesion level in object oriented metrics. The traditional LCOM metric assess the cohesion level by computing the intersecting methods of the module [9]. It neither represents the degree of cohesion nor the level of cohesion. But, the proposed CCMCF metric highlights the degree of cohesion with high and low level. The comparative analysis of LCOM and CCMCF is denoted in Table 2.

Table 2. Comparative Analysis of CCMCF with LCOM

<table>
<thead>
<tr>
<th>Program</th>
<th>LCOM</th>
<th>CCMCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic.cpp</td>
<td>0.91</td>
<td>3</td>
</tr>
<tr>
<td>arithmetic1.cpp</td>
<td>0.33</td>
<td>3</td>
</tr>
</tbody>
</table>

The graphical notion of CCMCF comparison with LCOM is depicted in Figure 1. From the results is found that the traditional LCOM value gives value 0 for high cohesion and integer value for poor cohesion. Whereas, the proposed CCMCF is able to differentiate the high and low cohesion through the metric values.

VII. CONCLUSION

Cohesion plays a vital role in assessing the quality of software modules. Cohesion also reflects in the cognitive analysis of program since a module with high cohesion is easily interpretable and understandable than the module with low cohesion. This paper proposed a new software cohesion metric called CCMCF for assessing level of cohesion by assigning cognitive weights of variables inside a class. Moreover, the higher complexity in software leads to more cost expensive and less maintainability of software. The assurance of less complexity software is of been great interest to researchers since the early days of development. Hence, the proposed CCMCF metric will be helpful for the developers to identify the flaws in their program in the development stage itself.

REFERENCES


