Advanced Object-Oriented Code Metrics

Evelyn R. Jenkins, Julian A. Sawyer
Massachusetts Institute of Technology, Cambridge, USA

***Abstract*—**Inherited complexity is one of the difficult tasks in software engineering field. Further, it is said that there is no physical laws or standard guidelines suit for designing different types of software. Hence, to make the software engineering as a matured engineering discipline like others, it is necessary that it has its own theoretical frameworks and laws. Software designing and development is a human effort which takes a lot of time and considers various parameters for successful completion of the software. The cognitive informatics plays an important role for understanding the essential characteristics of the software. The aim of this work is to consider the fundamental characteristics of the source code of Object-Oriented software i.e. complexity and understandability. The complexity of the programs is analyzed with the help of extracted important attributes of the source code, which is further utilized to evaluate the understandability factor. The aforementioned characteristics are analyzed on the basis of 16 C++ programs by distributing them to forty MCA students. They all tried to understand the source code of the given program and mean time is taken as the actual time needed to understand the program. For validation of this work, Briand's framework is used and the presented metric is also evaluated comparatively with existing metric which proves its robustness.

***Keywords*—**Software metrics, object-oriented, complexity, cognitive weight, understandability, basic control structures.

I. Introduction

SOFTWARE engineering is an applied discipline of software science which acquires engineering approaches. These approaches are very helpful for the researchers to handle the software product, such as establishing methodologies, processes, measurement, tools, architectures, standards, organization and management methods, quality assurance, quality controllable activities, seeking to high productivity, low cost, measurable development time and schedule [1]-[5]. These aforementioned software measurement techniques are directly or indirectly related to the complexity of the software. Definition of complexity according to IEEE is "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [6]. Software complexity measures lead to attaining the accurate estimation of the milestones that further helps the researchers to improve the product quality. Software complexity measures are also an important and determinant factor for the successful

nature of the software or its failures and a higher risk involved whenever these measures are ignored. Over the many years, some research has been carried out to calculate the complexity of software [9]-[21]. Most of the measures focus on increasing the performance of prediction ability in many aspects like effort, cost, quality, time/schedule or all these factors. Generally, indirect measures help developers to understand the information of software development processes through some quantitative basis. These software measures are very important for the software development for estimating and enhancing the quality of the software. Quality defines the meaningful terms for the users and the quality attributes are reliability, maintainability, performance, and availability [7], which are closely associated with the software complexity.

Over two decades, Object-Oriented (OO) approaches dominate the software industry due to the maintainability of the OO software. Design quality helps the researchers to evaluate the maintainability of the software with the help of some software metrics on the basis of quantification means. Once the design has been finalized and implemented, then any change in the design reflects higher difficulty and higher costs at the end of the software development. To overcome the above-mentioned problems, it is necessary to analyze the design very carefully for finding its effectiveness before finalizing it [8]. Variety of software metrics are available in the literature to compute the complexity in various perspective. Some among of them are mentioned as Chidamber & Kermerer (CK) [9] metrics suite, metrics of OO design (MOOD) [10], Lorenz and Kidd metrics [11], modified CK metrics [12], product metrics for OO design [13], [14], weighted class complexity metric [15], cognitive code complexity of inheritance for OO software [16] and an OO cognitive complexity metric [17]. Aforementioned software metrics are related to the OO design software's, which indicates some quality attributes of software and these metrics have their own benefits and limitations. Moreover, introducing new software complexity measures or perhaps enhancing the performance of existing one's is always welcome to achieve the higher quality software.

The OO approach is characterized by its classes and objects and the class consists of data (attributes) and methods (operations), and the methods are only responsible to access the attributes of the class through objects. Thus, when the number of methods in a class increased that means the complexity of the class increases, which directly affect the understandability factor of the software. The software is nothing, but just a collection of information and the information is the function of operands and operators, which contributes to the complexity and also impose some difficulty to understand it. According to Cognitive Informatics (CI), it is observed that the functional complexity of a software system depends on three aspects: input, output and internal architecture [18].

Cognitive complexity refers to quantify the difficulties faced by the personnel while understanding the source code or the human efforts needed to perform a task. The cognitive complexity measures emphasis on all the above mentioned factors which makes difficult to comprehend the software. Cognitive Functional Size (CFS) was proposed by Wang in 2003 and satisfy the rules of CI up to some limit [19]. In this work, authors introduced a promising solution by assigning the cognitive weights (W_c) to the possible Basic Control Structures (BCSs) of the software. Wang

verified and assigned the cognitive weights for sub conscious function, metacognitive function and higher cognitive function as 1, 2 and 3, respectively. The most common possible BCSs that can be incorporated in a software system are sequential, branch, iteration, embedded and their respective W_c are one, two, three and two. W_c of the each BCS describes the psychological burden or the extent of difficulty imposed on the staff who deals with the source code, i.e. developers, testers and the maintainers. The higher cognitive weight of the BCS specifies that a higher level of human effort or relative time needed to comprehend the BCS and vice versa. However, the entire cognitive complexity of the software is not only contributed by the cognitive weight of BCSs but also along with some other important attributes of the source code like operands, operators, and their relationship. High cognitive complexity is not desirable due to fault proneness and maintainability of the software. A higher complexity value also indicates poor design, which is not easily manageable by the personnel's and also causes to increase the effort drastically at the maintenance phase [20].

In our previous work, the cognitive complexity of the software is computed on the behalf of operands, operators, cognitive weight and it was validated through Weyuker property [17]. As an extension of the previous paper [17], we extend our metric to calculate the understandability factor and validate it through Briand property. It is also compared with related existing work of Misra et al. to verify the outcome of the proposed metric [15]. An experiment has been conducted to understand the source code of the software. Forty MCA students participated in this experiment. The source code of 16 concerned OO programs of are distributed among all the students and they are asked to understand the source code and what problem area the program has addressed, so that they can modify the program very easily and effectively whenever required [24]. The time taken by the students is recorded and their mean time is considered an actual time needed to understand the code. Thereafter, we apply some method to estimate the time from the proposed complexity metric attributes to achieve the recorded time, so that the Mean Relative Error (MRE) is reduced and the estimation accuracy increased. A correlation is also calculated for verifying the actual and estimated time of all program and the result shows that a good relation exists between the results of the proposed metric and recorded data.

The rest of the paper is organized as follows: Section II deals with the related work. Section III provides the detail of the proposed metric and its validation with Briand framework. Section IV gives the detail of the experimental results and the comparative study. Conclusion and future work are discussed in Section V.

II. Related Work

This section consists of WCC metric and its evaluation to calculate the cognitive complexity of the classes for OO software's.

A. WCC

The WCC metric has been devised by Misra et al. in 2008 to measure the cognitive complexity of the OO software [15]. It uses the summation of the cognitive weight of all the BCSs to calculate the complexity of a method and the class complexity is calculated by summing the complexity values of all the methods and the number of attributes present in the individual class. The WCC metric first calculates the complexity of operations in the method by assigned cognitive weight to each BCS by Wang [19]. Then, complexity values of all the methods of the class are added to find the complexity of the particular class and the entire complexity of the software is calculated by addition of individual class complexity values, which are generated as described in (1)-(4).

The complexity of the individual method is defined as the sum of the cognitive weight *q* linear blocks composed of individual BCSs. Each block may consist of *m* layers of nested BCSs and each layer with *n* linear BCSs. The total complexity of a method of any class is calculated as in (1).

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

where, MC is the method complexity and W_c is the cognitive weight of the concerned BCS.

Equations (2) and (3) are used to calculate the complexity of each class of the software. Equation (1) provides the complexity of a single method, if there are many methods incorporated in a class, then the complexity of each class is calculated by the addition of MC value of all its methods. In addition to it, a total number of attributes of a class is also calculated and added to the result of (2). In (3) the total number of attributes are represented with N_a.

$$AMC = \sum_{p=1}^{s} MC_p \quad (2)$$

$$WCC = N_a \sum_{p=1}^{s} MC_p \quad (3)$$

Further, if there are many classes embedded in the OO code, then the overall complexity of the code is the summation of the complexity values of the individual classes.

$$TotalWeighedClassComplexity = \sum_{x=1}^{y} WCC_x \quad (4)$$

where, y indicates the number of classes present in the source code of OO software.

It is important to note that the WCC metric considers only the cognitive weight of the incorporated BCSs and the number of attributes of the class, but it excludes the other occurrences of attributes in the methods, i.e. the number of operators and accessing same attributes by many methods of the class. These parameters are also contributing and affect the complexity level

for both the man and the machine. Although software is observed as formally a described design of information and implementation of statements of computing applications which also mean that software is just a collection of information [21]. This equivalence between the software and the information leads to the understanding the level of difficulty in a software, it means that whenever a software contains higher information contents inside the source code then it is more difficult to understand. The software is a mathematical entity and represents the computational information. The entire information is represented by the software as a function of operands (that hold the information) and the operators (that carry out operations on operands). Whenever information is manipulated by the operators, this manipulated information is hard to handle and even harder to understand. So, operators cannot be disregarded and it should be included while measuring the cognitive complexity of the software that helps to find the difficulty to understand the code.

III. Evaluation of Proposed Metric and its Validation with Briand Property

A. Illustration of the Proposed Metric

The Object-Oriented software comprises of classes, subclasses and objects where attributes, methods and messages are their elements. Objects are the class instance, which cooperates through message exchanges. The complexity is defined as a function of the interaction between the set of properties and in this case attributes and the methods are the properties of the software [22]. These elements or properties are defined in the class declaration and contribute to software complexity. Among these elements, the methods play an essential role because they operate on the attributes or data in response to the messages. Even though the complexity of a method directly affects the understandability (known as program comprehension) of the code due to more information content is incorporated into the method.

Most of the OO software metrics do not consider the cognitive complexity of the software. Cognitive complexity defines the mental burden supplied to the personnel while dealing with the source code, i.e. developers, testers and maintainers. Now, we can make the relation more clearly and introduce a new complexity metric to calculate the complexity as well as the understandability of the software. Our new proposed metric calculates the complexity of the software on the basis of cognitive weight (W_c) and the number of operands, operators and the way to access the attributes by the methods of the class. The cognitive weight (W_c) is calculated as defined by Wang [19]. These cognitive weights of the BCSs are assigned according to the difficulty level to comprehend the given structure.

So, the understandability time of the program is calculated by conducting an experiment with 40 MCA students. Concerned programs are distributed among all the students and they are asked to understand the code and their time is recorded individually. The actual time is considered by calculating the average time of all the students.

The proposed metric first calculates the complexity of a method and the calculated complexity values of all the methods of a particular class are summed to get the class complexity. More formally, the class complexity (CC) is calculated as:

$$CC = \sum_{i=1}^{n} (Information \times W_c \times RASP)_i \quad (5)$$

where, CC denotes the cognitive complexity of a class and *n* denotes the number of methods resides in the class. The information represents the number of operands and operators available in a given method, and W_c is the cognitive weight of the BCSs (calculated as in (1)). The Ratio of Accessing Similar Parameters (RASP) is calculated by the intersection of the methods on the basis of used parameters of the concerned class. The same is applied to all the methods and the resulted sum is divided by the number of parameters of the class as described in (7).

$$ECCC = \sum_{s=1}^{x} CC_s \quad (6)$$

If there are *x* number of classes present in the software, then Entire Cognitive Code Complexity (ECCC) is calculated by summing the cognitive complexity weight of individual class as provided in (5).

$$RASP = \sum M_i \cap M_j / na \quad (7)$$

where, M denotes the method of a class, i, j represents the method numbers and na indicates the number of attributes present in the same class.

$$AMP = (\sum_{i=1}^{n} M_i)/ n \quad (8)$$

where, AMP represents Average Method Parameters, M represents the method and n represent the number of methods present in a class. AMP provides how each method accessed the parameters of the class on an average.

The entire complexity of the software includes not only the number of operands, operators and cognitive weight (W_c), but it should also include the complexity of the main program. When the CC value increases, then the entire program complexity of the software increases. For validation of the complexity metric results, an experiment is conducted and it is found that the attributes of the proposed metric have significance effect on the program comprehension. In this experiment, 16 C++ programs are distributed among the students and they are asked to analyze the source code of the program and tried to understand it and their time is recorded to achieve the comprehension factor. In addition

to it, we have also estimated the time to understand the code with the help of complexity values of the proposed metric. Equation (9) is used to find out the comprehension time by using some important parameters of the source code.

$$Unders\tan dablity\_time = \frac{(W_C \times 0.27),((RASP \times AMP)\times 0.02),}{(\inf oramtion \times 0.12)} / 1.65 \quad (9)$$

where, W_c indicates cognitive weights of all BCSs, RASP and AMP are stated above and information represents the combination of operands and operators of the entire program. The outcome of (9) gives a numeric value that specifies the required time in minutes to understand the program.

B. Evaluation of the Proposed Metric through Briand Property

- **Property Complexity 1: (Non-negative)**. The complexity of a system S = <E, R> is non-negative if complexity (S) ≥ 0.

Proof: As aforementioned that the proposed metric obtained the complexity value by using the non-negative weights of the BCSs, the number of operands and operators in the methods of a given class. Without these above-said attributes, software cannot do anything. So, this property is satisfied by the proposed metric.

- **Property Complexity 2: (Null value)**. The complexity of a system S = <E, R> is null if R is empty. This can be formulated as:

R = ∅ ⇒ Complexity (S) = 0.

Proof: If information attributes are not incorporated in the system, then the cognitive complexity by proposed metric will be null, means, operands, operators and BCSs are not present in the methods of a class, then naturally the complexity of the software system in terms of the cognitive weight is null.

- **Property Complexity 3**: (Symmetry). The complexity of a system S = <E, R> does not depend on the convention chosen to represent the relationships between its elements.

(Let S = < E, R > and S^{-1} = < E, R^{-1}) ⇒
Complexity (S) = Complexity (S^{-1}).

Proof: The proposed metric assigns the cognitive weight (W_c) to a control structure and it does not depend on the sequence order of their representation in the program. So, there will be no influence on the cognitive complexity value, when changing the sequence order or its representations. Hence, this property is also satisfied by the proposed metric.

- **Property Complexity: 4 (Module Monotonicity)**. The complexity of a system S = <E, R> is no less than the sum of the complexities of any two of its modules with no relationships in common.

(Let S = <E, R> and m_1 = <E_{m1}, R_{m1}> and m_2 = <E_{m2}, R_{m2}> and $m_1 \cup m_2 \subseteq S$ and $R_{m1} \cap R_{m2} = \emptyset$)
⇒ Complexity (S) ≥ Complexity (m1) + Complexity (m2).

Proof: Whenever, the class S is divided into two sub-modules (subclasses) m1 and m2 without modification implied to the sub-modules or we can say a class S is divided into two subclasses without any change, then the cognitive complexity of the partitioned classes or subclasses will never be greater than the complexity of the joined class. This property can be proved by taking the example of Appendix I in [15], and the cognitive complexity value of each class of the given program according to WCC [15] and proposed metric is provided in Table I. Five classes: Person-Employee-Student-FacultyAdministrative are available in the given example of Appendix I and these classes are partitioned into five subclasses Person, Employee (Emp.), Student, Faculty and Administrative. The cognitive complexity value of the entire program of Appendix I is 93 and the partitioned subclasses is provided in Table II. Therefore, the cognitive complexity of the entire code is equal to the summation of the complexity values of its subclasses. Hence, this property is also satisfied by the proposed metric.

TABLE I
CALCULATED WCC AND PROPOSED METRIC COMPLEXITY VALUES FOR SUBCLASSES OF APPENDIX I IN [15]

Class name	Person	Emp.	Student	Faculty	Administrative
WCC		4	8	3	
Proposed metric		8	24	5	

- **Property Complexity 5: (Disjoint Module Additivity)**. The complexity of a system S = <E, R> composed of two disjoint modules m1, m2, is equal to the sum of the complexities of the two modules.

(S = <E, R> and S = m1 ∪ m2, and m1 ∩ m2 = ∅) ⇒ Complexity (S) = Complexity (m1) + Complexity (m2).

Proof: The cognitive complexity of the module m1 and m2 is equal to the complexity of these concatenated modules into a single class. In other words, if the two autonomous classes are concatenated into a single program or class then the cognitive weight of the concatenated classes is just by summation of the individual class complexity values. Here, description of property 4 is considered to prove this complexity property is satisfied by the proposed metric. The cognitive complexity of the combined and independent modules according to the proposed metric is provided in Table I, which indicates that the complexity of the entire program is equal to the complexity of Person, Employee, Student, Faculty and Administrative classes, i.e. (93 = 46 + 8 + 24 + 5 + 10) respectively. Hence, this property is also satisfied by the proposed metric.

Though, the analysis of the complexity properties 1 to 5, it is found that all desired properties are satisfied by the proposed metric to calculate the complexity of the OO software.

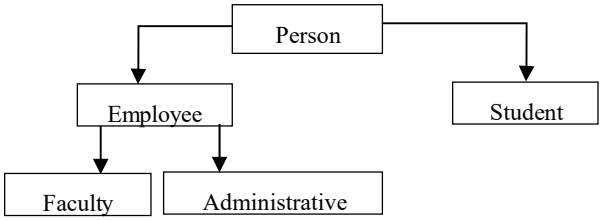

Fig. 1 Class hierarchy of given OO code of Appendix I in [15]

TABLE II
CALCULATED WCC AND PROPOSED METRIC COMPLEXITY VALUES FOR DIFFERENT COMBINATION OF OO CODE

Subclasses	WCC	Proposed metric
PERSON-STUDENT-EMPLOYEEFACULTY-ADMINISTRATIVE		
PERSON- EMPLOYEE-FACULTY-ADMINISTRATIVE		
PERSON-EMPLOYEE-ADMINISTRATIVE		
PERSON- EMPLOYEE-FACULTY		
PERSON-STUDENT		

Weyuker property can be used to validate any complexity metric theoretically. As described in our previous paper [17], seven out of nine properties are satisfied by the proposed metric. These properties are to be satisfied by any effective complexity metric, but this does not provide the adequate situation for overall validation. Since the practical success of any new measure also depends upon some other important issues like user understandability and the existing relation between measures and its attributes. The proposed metric is an indirect measure because it uses some important parameters of source code in a different sequence to calculate the complexity of OO programs. C. Karner provides a more suitable method to validate the new measures [23]. The description of this practical method of the proposed metric is provided here.

Measure's purpose: the main purpose of our metric is to calculate the cognitive complexity of the software and on the basis of calculated complexity values the developers can analyze the complexity of the software, whether it is legitimate or not. If they find an unpredictable behavior, then further action can be accommodated to overcome the problem before it becomes critical with respect to product design and quality.

Measure's scope: the metric can be used after the development of the source code of OO design software, but not at earlier stages of the software development life cycle. This metric is evaluated to estimate the comprehension time to understand the source code and can also be used to estimate the maintenance effort.

Identifying parameters to measure: this measure indicates the quality of the source code, which is implemented by the developers. The complexity of the source code indicates the difficulty level to make changes and to understand the program. Higher complexity value makes software less manageable and increases required effort and lesser complexity value indicates more skillful and manageable software.

Measure's instruments: complexity by the proposed metric can be calculated manually or by using some automated tools.

Instrument natural variability when measurement: the proposed metric calculates the complexity in a simple and straightforward way and it is very easy to understand. Thus, there will be no variability while measuring the attributes of the proposed complexity metric.

The relation between parameters and the metric value: a direct relation exists between the source code parameters and the proposed metric values because when the C2M value increases, it means the complexity of the software increases and the quality of the product decreases with respect to time and space. The proposed metric is a quality indicator for the OO designed software, but not unique.

The effect of the automated instrument: once an automated tool is developed for measuring the attributes of the proposed metric, then there will be no need for personnel to calculate the attributes of the proposed metric and only the automated tool cost will be imposed on the company.

IV. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, the applicability of the proposed metric has been analyzed by applying it to the 16 C++ programs of [24] and also to an OO example of Appendix I of [15] and its class hierarchy is provided in Fig. 1. The proposed metric focuses on some significant attributes of the source code and make some arrangement to find out the more accurate results. This metric considers the number of operands, operators, cognitive weight (W_c), RASP and AMP to calculate the cognitive complexity of the source code. In addition to it, the required time to understand the source code of the program is also calculated. The proposed metric is validated with the help of Briand property and a comparative study is also done with a similar metric WCC [15]. In WCC, only the cognitive weight of BCSs and the class attributes are taken into account to find out the complexity of the OO programs, but some other remaining attributes also contribute to the complexity of the program like other occurrences of information parameters, accessing of similar parameters by different methods and the complexity of the main program as well, for estimating the complexity of the entire software. So, these aforementioned parameters should be considered while calculating the complexity of the software. After that, an experiment has been conducted in which 40 MCA students took participate and they are asked to understand the problem statement of the source code. The time taken by the individual student is recorded and their average time is considered an actual time required to understand the source code and a formula is also developed to estimate the time to understand the code with the help of proposed complexity metric attributes and the formula is described in (9). The constant values in the formula of (9) are selected in such a way that the outcome reduces the error between actual and estimated time. In Table III, actual time to understand and estimated time to understand is denoted by ATU and ETU respectively. Calculated complexity values of all 16 C++ programs with ATU and ETU according to WCC and proposed metric is provided in Table III and a graph showed the ATU and ETU values in Fig. 3.

After measurement of ATU and ETU, the Error% is calculated by using the following formula of (10). In this equation,

$TIME_{calculated}$ and $TIME_{actual}$ specify the ETU and ATU respectively and error between calculated and actual time shows overestimation and underestimation.

$$Error(\%) = \frac{TIME_{calculated} - TIME_{actual}}{TIME_{actual}} \times 100 \quad (10)$$

Each program is analyzed in terms of a unit known as lines of code (LOC) and the unit of proposed and WCC metric is cognitive weight unit (CWU) as provided in Table III and shown in Fig. 2.

The LOCs of software or the length of the source code can be used as a predictor of program characteristics such as effort and difficulty in maintenance. However, it characterizes the software only in one specific aspect, i.e. static length or size because it takes no account of the functionality and the other limitations of this estimation are described in the previous version of this paper by using a coding efficiency (CE) method.

The value obtained by the proposed metric indicates that as the CWU increases that means, the program will become more complex to understand due to the inclusion of greater number of operands, operators and greater number of methods that increase the complexity of a class. Certain interesting observations are made from Tables I-III, and Figs. 2-4, which is as follows:

- Tables I and II and Fig. 1 provides the result of Appendix I of [15].
- Table III contains the actual cognitive complexity values for all classes. High complexity value indicates high complexity attributes involved in the program, as it involves greater number of operands, operators, BCSs weight and ratio of accessing similar parameters of the classes and vice-versa with low complexity value.
- From Table III, it is found that the trends for the proposed metric and WCC follow basically the same pattern. As the proposed metric value increases, so does the corresponding WCC complexity value. It is noteworthy that there are two points for which the proposed metric generates identical complexity value at WCC (see painted with red and bottle green row in Table III) i.e. 85/84 and 18 respectively. This indicates that these two programs have higher information, BCSs, RASP and AMP to implement the code as mentioned by BCSs weight W_c= 28/21, information = 49/51 with LOC = 48/60 and WCC = 18/18. And, other similar highlighted colors indicate the difference between the proposed metric and WCC metric calculated complexity values in measuring the complexity of the software. Somewhere smaller complexity value as at WCC metric indicates higher complexity by proposed metric due to many other important parameters embedded into the source code, which is ignored by the WCC.
- Table III also contains the results of ATU and ETU values. ATU is calculated by conducting a controlled experiment with the help of 40 students of our institute and all the 16 C++ programs are distributed among them and they are asked to understand the source code. Their analyzed time is recorded and the average time of all the students is considered an actual time required in understanding the program code. After that, a relation has been formed between the utilized parameters of the proposed metric to estimate the ETU as described in (9). Both the calculated results of each program are provided in Table III as corresponding to the program number.
- Another important result has been discovered of the proposed metric and WCC metric values is that a linear relationship exists between them and its result is provided in Table III and shown in Fig. 2. The proposed metric of this paper yield higher complexity value as a result than WCC metric because of many other significant parameters considered in the formation of our proposed metric formula described in (6).

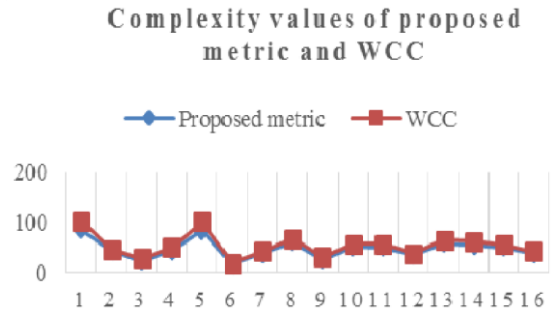

Fig. 2 Comparison chart of complexity values of proposed and WCC metric

- Fig. 3 shows the result of ATU and ETU of all the 16 programs and ETU is calculated as in (9). Higher time indicates that it is more difficult to understand the program due to the complex structure and may be more informative contents are present in the program body. Hence, the comprehension time depends on the complexity level of the program means higher complex program requires much more time to understand than a simple program.
- Fig. 4 shows the error% according to as (10) with its absolute values. It indicates the difference between ATU and ETU and the result shows that there are only two programs out of the 16 programs is near to 30% of error and approximately 50% of the programs are under 10% of error, which reveals the accuracy level of the estimated result. The correlation between the ATU and ETU also indicates the effectiveness of the proposed work of this paper, and its correlation factor is 0.97.

From all analysis of the above experiments, it is found that the proposed metric has a good capability to calculate the complexity as well as the comprehension time of OO programs and can qualify as a worthy complexity metric.

3) This work can also be extended to measure the effort required to test the software in testing and in the maintenance phase.
4) Boundaries can be defined for the calculated complexity values for the particular software designs.

REFERENCES

[1] F.L. Bauer, "Software Engineering", *Information Processing*, 1972.
[2] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976
[3] F.P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, vol. 20, no. 4, 1987, pp. 10-19.
[4] J.F. Peters and W. Pedrycz, *Software Engineering: An Engineering Approach*, John Wiley & Sons, Inc., NY, 1998.
[5] Y. Wang, "Software Engineering Foundations: A Trans disciplinary and Rigorous Perspective", *CRC Book Series in Software Engineering*, Vol. 2, 2006.
[6] IEEE CS, "IEEE Standard Glossary of Software Engineering Terminology", *IEEE Standard 610.12*, (1990).
[7] I. Sommerville, *Software Engineering*, 8th edition, Boston, MA: Addison-Wesley, 2007.
[8] R. Reißing, "Towards a model for object-oriented design measurement", *In 5th International ECOOP workshop on quantitative approaches in object-oriented software engineering*, 2001, pp. 71-84.
[9] S.R. Chidamber and C.F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, 1994, pp. 476-493.
[10] R. Harrison, S.J. Counsell and R.V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics", *IEEE Transactions on Software Engineering*, vol. 24, no. 6, 1998, pp. 491-496.
[11] M. Lorenz, J. Kidd, *Object-oriented software metrics*, Englewood Cliffs,
New Jersey: Prentice Hall, 1994.
[12] V.R. Basili, L.C. Briand, and W.L. Melo, "A validation of objectoriented design metrics as quality indicators", *IEEE Transactions on Software Engineering*, vol. 22, no. 10, 1996, pp. 751-761.
[13] S. Purao, and V. Vaishnavi, "Product metrics for object-oriented systems". *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 191-221.
[14] V.K. Vaishnavi, S. Purao, and J. Liegle, "Object-oriented product metrics: A generic framework", *Information Sciences*, vol. 177 no. 2, 2007, pp. 587-606.
[15] S. Misra and I. Akman, "Weighted class complexity: A measure of complexity for object –oriented system" *Journal of Information Science and Engineering*, vol. 24, 2008, pp. 1689-1708.
[16] S. Misra, I. Akman and M. Koyuncu, "An inheritance complexity metric for object-oriented code: A cognitive approach", *Sadhana*, vol. 36, no. 3, 2007, pp. 317-337.
[17] A.K. Jakhar and K. Rajnish, "Measure of Complexity for Object-Oriented Programs: A Cognitive Approach", *In Proc. of 3rd International Conf. on Advanced Computing, Networking and*
[18] Y. Wang, "On the Cognitive Informatics Foundations of Software Engineering", *Proc. of the 3rd IEEE International Conf. on Cognitive Informatics,* 2004, pp. 21-31.
[19] Y. Wang and J. Shao, "A new measure of software complexity based on cognitive Weights", *IEEE Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, 2003, pp. 69-74.
[20] L.C. Briand and J. Wüst, "Modeling development effort in objectoriented systems using design properties", *IEEE Transactions on Software Engineering*, vol. 27, no. 11, 2011, pp. 963-986. [21] Y. Wang and J. Shao, "Measurement of the Cognitive Functional Complexity of Software", in *Proc. of 2nd IEEE International Conf. on Cognitive Informatics*, 2003, pp. 69-74.
[22] D. Abbot, "A design complexity metric for object-oriented development", *Unpublished Master's Thesis*, *Department of Computer Science, Clemson University*, U.S.A, 1993.
[23] C. Kaner, "Software Engineering Metrics: What do they Measure and how do we know?", In *In Metrics 2004*, IEEE, CS.
[24] A. Kamthane, *Object-Oriented Programming with ANSI & Turbo C++,* Pearson Education, Fourth Edition, India, 2003.