Received November 29, 2017, accepted December 27, 2017, date of publication January 10, 2018, date of current version March 13, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2791344

A Suite of Object Oriented Cognitive Complexity Metrics

SANJAY MISRA[1,2], ADEWOLE ADEWUMI[1], LUIS FERNANDEZ-SANZ[3], AND ROBERTAS DAMASEVICIUS[4]
[1]Covenant University, Ota 1023, Nigeria
[2]Atilim University, 06830 Anakra, Turkey (on leave)
[3]Universidad de Alcala de Henares, 28871 Alcala de Henares, Spain
[4]Kauno Technologijos Universitetas, 51368 Kaunas, Lithuania

Corresponding author: Adewole Adewumi (wole.adewumi@covenantuniversity.edu.ng)

This work was supported by Covenant University through the Centre for Research, Innovation and Discovery.

ABSTRACT Object orientation has gained a wide adoption in the software development community. To this end, different metrics that can be utilized in measuring and improving the quality of object-oriented (OO) software have been proposed, by providing insight into the maintainability and reliability of the system. Some of these software metrics are based on cognitive weight and are referred to as cognitive complexity metrics. It is our objective in this paper to present a suite of cognitive complexity metrics that can be used to evaluate OO software projects. The present suite of metrics includes method complexity, message complexity, attribute complexity, weighted class complexity, and code complexity. The metrics suite was evaluated theoretically using measurement theory and Weyuker's properties, practically using Kaner's framework and empirically using thirty projects.

INDEX TERMS Cognitive complexity, cognitive weights, empirical validation, software metrics.

I. INTRODUCTION

Object orientation is now a widely adopted approach in software engineering because software built using this technique is usually easier to maintain. Object-oriented (OO) software development offers powerful features such as: dynamic binding, encapsulation, inheritance, interaction, polymorphism, and reusability. Quite a number of metrics have been put forth for evaluating OO software including: ''Chidamber and Kemerer (CK) metrics suite [1], MOOD metrics for OO Design [2], design metrics for testing [3], product metrics for object-oriented design [4], Lorenz and Kidd metrics [5], Henderson–Seller *et al.* metrics [6], (slightly) modified CK metrics [7], and size estimation of OO systems [8]''. These metrics often target specific phases of software development such as design, implementation and testing. In addition, they cover some features of OO languages as well as some quality attributes, among which are: efficiency, correctness, integrity, flexibility, maintainability, interoperability, reliability, portability, reusability, usability and testability [9]. Across the aforementioned attributes, maintainability is considered as the most essential attribute of software products [10]. In order to predict critical information about the reliability and maintainability of software systems from automatic analysis of source code, complexity metrics can be used [11].

Complexity metrics have a lot of potential uses which include: provision of feedback during a software project to help control the design activity, and provision of detailed information about software modules to help pinpoint areas of potential instability during testing and maintenance. Cyclomatic complexity is the most widely used complexity metric for computer software [12]. It is a software metric that provides a quantitative measure of the logical complexity of a program. The introduction of cognitive informatics to the software engineering domain through the work of Wang [13] has brought about the emergence of a new set of complexity metrics referred to as cognitive complexity metrics. These metrics introduce cognitive weights - which define the effort required, relative time or extent of difficulty in comprehending software. In cognitive informatics, the functional complexity of software in design and comprehension depends on three key elements namely: its input, internal processing and output [14]. Initially three basic control structures (BCS), branch, iteration and sequence were identified [15]. However, the work of Shao and Wang [14] modified these BCSs and introduced what obtains in Table 1. These BCSs are the fundamental logic building blocks of software.

The aim of this research is to validate the metrics suite presented in a conference [16]. This includes theoretical

TABLE 1. BCSs with their respective cognitive weights (W_c).

Category	BCSs	Weight W_c
Sequence	Sequence (SEQ)	1
Branch	If-then-else (ITE)	2
	Case (CASE)	3
Iteration	For-do (Ri)	3
	Repeat-until (R0)	3
	While-do (R1)	3
Embedded Component	Function/Method call (FC)	2
	Recursion (REC)	2
Concurrency	Parallel (PAR)	4
	Interrupt (INT)	4

and empirical validation as well as a robust comparative study. The rest of this paper is structured as follows: Section II presents existing cognitive complexity measures. In Section III, the proposed metrics suite was explained and demonstrated using an OO project. In Section IV, a comprehensive validation of the proposed metrics suite was conducted through theoretical and empirical validation. A comparative study was also conducted. Section V concludes the work while also pointing out areas of future research.

II. EXISTING COGNITIVE COMPLEXITY METRICS

This section reviews existing cognitive complexity metrics, which include:

A. COGNITIVE FUNCTIONAL SIZE (CFS) OF SOFTWARE

The first cognitive complexity metric proposed was by Shao and Wang [14] and is used to measure the cognitive functional size (CFS) of software. The functional size of software depends on its input, output and internal control flow and is denoted by the formula:

$$CFS = (N_i + N_0) \times W_c \tag{1}$$

Where N_i is the number of program inputs, N_o is the number of program outputs and W_c is the sum of the cognitive weight of all basic control structures (BCSs).

Implementation of CFS is easy and technology independent. It however excludes some essential details of cognitive complexity such as information that is contained in the identifiers and operators. It also does not consider some unique features of the object-oriented paradigm such as inheritance.

B. COGNITIVE INFORMATION COMPLEXITY MEASURE (CICM)

It was put forth in [17] and is defined as, "the product of weighted information count of software (WICS) and the cognitive weight (Wc) of the BCS in the software i.e.

$$CICM = WICS^{*} W_c \tag{2}$$

Where WICS is the sum of the weighted information count (WICL) of every line of code (LOCs) of a given software denoted as:

$$WICS = \sum_{k=1}^{LOCS} WICL_k \tag{3}$$

WICL of software is a function of the identifiers and operands per line of code as well as the number of lines of code in that software. It is denoted as:

$$WICL_k = ICS_k/[LOCs - k] \tag{4}$$

Where LOCs is the number of lines in software and

ICS_k is the information contained in a software program for the kth line".

Calculating this metric can be cumbersome especially where the software contains several lines of code. In addition, this metric does not take into consideration distinct features of OO paradigm such as inheritance.

C. MODIFIED COGNITIVE COMPLEXITY MEASURE (MCCM)

It was proposed in [18] and simplifies the complication associated with CICM by taking into consideration all operators and operands in a software program. The formula is given as:

$$MCCM = (N_{i1} + N_{i2}) \times W_c \tag{5}$$

Where N_{i1} is the total number of operators and

N_{i2} is the aggregate number of operands

W_c is the aggregate of the cognitive weight of all basic control structures (BCSs)

The drawback of this metric is that the values obtained from its computation tend to be very large.

D. COGNITIVE PROGRAM COMPLEXITY MEASURE (CPCM)

This metric was put forth on the premise that the cognitive complexity of software is strongly affected by the total number of occurrences of input and output variables [19]. It is denoted as:

$$CPCM = S_{IO} + W_c \tag{6}$$

$$S_{IO} = N_i + N_o \tag{7}$$

Where N_i is the total occurrence of input variables and

N_o is the total occurrence of output variables

W_c is the sum of the cognitive weight of all basic control structures (BCSs)

However, the process of counting the number of inputs and outputs has been criticized for being unclear and ambiguously interpreted [20].

E. NEW COGNITIVE COMPLEXITY OF PROGRAM (NCCoP)

This measure was put forth in [20] to measure the cognitive complexity of a program. It is based on data objects (consisting of input and output), internal behaviour of the software, the operands, and the individual weight of BCSs of every line of code. This measure does not consider operators in its computation, which makes it unlike CICM and MCCM. The measure is formulated as:

$$NCCoP = \sum_{k=1}^{LOCS} \sum_{v=1}^{LOCS} N_v \times W_c(k) \tag{8}$$

Where, the first summation is the line of code from 1 to the last LOC

N_v is the number of variables in a particular line of code

W_c is the BCS weight

The drawback of this measure is that the process of counting the number of variables per line of code is unclear and ambiguously interpreted in [20].

F. INHERITANCE COMPLEXITY METRIC

This metric proposed in [21] can be used in evaluating the design of OO code. According to the study, "it is based on inheritance, which happens to be an important feature of OO systems. The metrics is first interested in calculating the complexity of methods by considering the corresponding weights for each method of the class of the system denoted as:

$$MC = \sum_{j=1}^{q} \left[\prod_{k=1}^{m} W_c\,(j,k,l) \right] \tag{9}$$

Where W_c is the cognitive weight of the concerned BCS. Thus, the method complexity of a software component is defined as the sum of cognitive weights of its q linear blocks composed of individual BCSs, since each block may consist of m layers of nested BCSs and each layer with n linear BCSs. Where methods in an OO code include recursive method calls, each recursive method call is considered as a new call and taken into account during the calculation of method complexity.

The second stage of the metric calculates the complexity of each class. MC gives the complexity of a single method and so if there are several methods in a class then complexity of an individual class is calculated by the summation of the weights of all methods and is denoted as:

$$ClassComplexity(CC) = \sum_{p=1}^{s} MC_p \tag{10}$$

Where s is the number of methods in a class.

The third stage of this metric calculates the complexity of the entire code by identifying the existing relations between classes. For a system containing more than one class and the classes are in the same level then their weights are added. If however they are children of a class then their weights are multiplied due to the inheritance property. For a system where there are m levels of depth in the OO code and level j has n classes then the cognitive code complexity (CCC) of the system is given by:

$$CCC = \prod_{j=1}^{m} \left[\sum_{k=1}^{n} CC_{jk} \right] \tag{11}$$

If there are more than one class hierarchies in a project, then the CCCs of each hierarchy are added to calculate the complexity of the whole system. The class complexity unit (CCU) of a class is defined as the cognitive weight of the simplest software component (single class, single method and linear structure) and is used as the basic unit for complexity". Among the aforementioned existing metrics, this metric is the first to factor in inheritance in computing complexity of OO systems.

G. SOFTWARE METRIC FOR PYTHON (SMPy)

This metric was proposed in [22] and computes the complexity of python (or any other OO programming language) code. According to the study, "it is computed by summing up factors that affect complexity in OO code which include complexity due to inheritance, complexity of distinct classes, global complexity and complexity due to coupling between classes. This is denoted as:

$$SMPy = CIclass + CDclass + Cglobal + Ccoupling \tag{12}$$

Where CIclass is the complexity due to inheritance

CDclass is the complexity of distinct class

Cglobal is the global complexity

Ccoupling is the complexity due to coupling between classes

Before calculating CIclass and CDclass however, the complexity of a class (Cclass) is first determined and is formulated as:

$$\begin{aligned} Cclass = {} & weight(attributes) \\ & + weight(variables) \\ & + weight(structures) \\ & + weight(objects) - weight(cohesion) \end{aligned} \tag{13}$$

Where:

weight (attributes) and weight (variables) is defined as:

$$weight(attributes) = 4 \times AND + MND \tag{14}$$

$$weight(variables) = 4 \times AND + MND \tag{15}$$

AND is the number of Arbitrarily Named Distinct Variables/Attributes

MND is the number of Meaningfully Named Distinct Variables/Attributes

weight (structures) is defined as:

$$weight(structures) = weight(BCS) \tag{16}$$

weight of objects is defined as:

$$weight(objects) = 2 \tag{17}$$

This is because creating an object is similar to calling a function [22]

weight of cohesion is defined as:

$$weight(cohesion) = MA/AM \tag{18}$$

MA is the number of methods where attributes are used

AM is the number of attributes used inside methods

Cglobal is defined as:

$$\begin{aligned} Cglobal = {} & weight(variables) \\ & + weight(structures) + weight(objects) \end{aligned} \tag{19}$$

In computing CIclass, if the classes are in the same level in the class hierarchy then their weights are added. If they are children of a class then their weights are multiplied due to their inheritance property. If there are m levels of depth in the OO code and level j has n classes then the complexity of the system due to inheritance is given as:

$$CIclass = \prod_{j=1}^{m} \left[\sum\nolimits_{k=1}^{n} Cclass_{jk}\right] \tag{20}$$

CDclass is defined as:

$$CDclass = Cclass(x) + Cclass(y) + \ldots \tag{21}$$

Ccoupling is defined as

$$Ccoupling = 2^{c} \tag{22}$$

Where c is the number of connections made from one method to other method(s) in another class''.

This metric majorly considers inheritance and gives as result, a single value, which represents the complexity of OO code. With this metric also, it is possible to determine which factor contributes the most to an OO code's complexity.

Aside these aforementioned metrics, Crasso *et al.* [23] presented a set of software metrics that measure software cognitive complexity in Java-based OO projects. In the study, an automatic tool for gathering the metrics was implemented as an Eclipse plug-in. The proposed metrics were validated both theoretically and empirically using ten OO projects and the results showed that the metric could be considered in real development scenarios. Also, Sheela and Aloysius [24] presented a Cognitive Weighted Coupling on Attribute Reference (CWCoAR) complexity metric that measures not only the software structural complexity of aspect-oriented programs but also the cognitive complexity on the basis of type. In other words, the metric measures aspect level complexity. It was validated through a statistical analysis, case study and comparative study. Aside these, there are some other recent studies [25]–[29] which might not necessarily be classified as cognitive complexity measures. These were found to focus more on the cohesion feature of OO projects.

H. MOTIVATION AND DISCUSSION

The popular CK metric suite has not considered the complexity of the internal architecture of methods, which represents the cognitive complexity. All the metrics in CK metric suite are very simple and do not care for the complexity inside the program or called method. Several metrics have been proposed and are still being proposed for different object oriented features but:

1) No suite has been proposed after the CK metrics suite,
2) No metric suite using cognitive complexity is in existence.
3) The existing cognitive complexity metrics shown in previous sections only consider some specific attributes of object-oriented programming.

By considering all these pitfalls the aim of this research is as follows

1) To propose a suite of metrics which considers all-important features of OO.
2) To perform a complete validation on real projects which include
   a. Theoretical validation, which proves the scientific base of the proposed metrics
   b. Empirical validation, which proves the practical application of metric on real projects
   c. Comparative study, which proves the worth of the proposed metrics

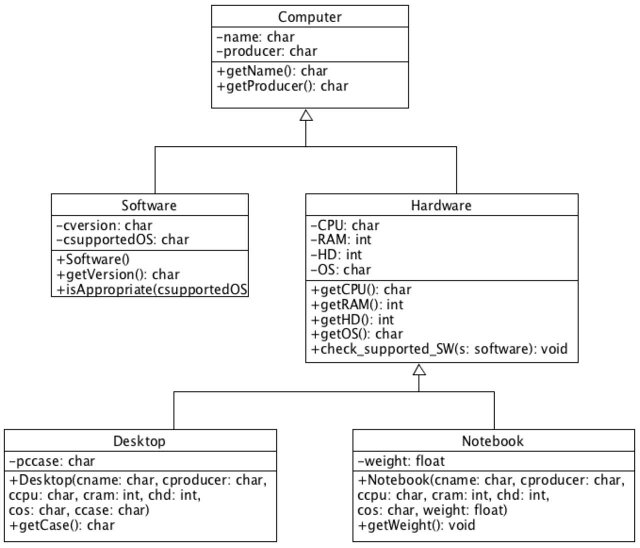

FIGURE 1. Class diagram of an object-oriented project.

III. PROPOSED METRICS SUITE AND DEMONSTRATION

From the aforementioned motivation, in this present study, a metric suite for evaluating the cognitive complexity of OO projects is presented. The proposed metric suite is an integration of metrics that cover the crucial features of OO. The metrics are adapted in a manner that each covers a distinct OO feature gleaned from existing literature. More specifically, we analysed the existing metrics, selected those with distinct features, suggested modifications where necessary and then presented in a suite. The suite consists of five metrics namely: Attribute Complexity (AC), Method Complexity (MC), Class Complexity (CLC), Message complexity also referred to as Coupling Weight for a Class (CWC) and Code Complexity (CC). In this section, we explain and demonstrate the computation of these metrics through an object-oriented project whose class diagram is given in Fig. 1. The full code is given in Appendix 1.

A. METHOD COMPLEXITY (MC)

The method complexity of a software component can be defined as, ''the sum of cognitive weights of its q linear blocks composed of individual BCSs, since each block may consist of m layers of nested BCSs, and each layer with n linear BCSs. This is calculated by considering corresponding cognitive weights of structures in a class' method(s)''. In other words, we associate a weight (number) with each method of a class and then sum the weights of all the methods. A formal representation of the complexity of a single method is given as follows:

$$MC = \sum_{j=1}^{q}\left[\prod_{k=1}^{m}\sum_{i=1}^{n} W_c(j,k,l)\right] \qquad (23)$$

W_c is the cognitive weight of the concerned Basic Control Structure (BCS) as given in Table 1. The unit of MC is taken as Code Complexity Unit (CCU). The method complexity for the sample project illustrated in Fig. 1 (full implementation is found in the Appendix), is calculated as follows:

$$\begin{aligned} MC_{getName} &= MC_{get\text{Producer}} \\ &= \sum_{j=1}^{q}\left[\prod_{k=1}^{m}\sum_{i=1}^{n} W_c(j,k,l)\right] = 1 \end{aligned}$$

So, the method complexity for class COMPUTER is:

$$\begin{aligned} \text{MC}_{\text{COMPUTER}} &= \text{MC}_{\text{getName}} + \text{MC}_{\text{getProducer}} \\ &= 1 + 1 = 2 \text{ CCU} \end{aligned}$$

The method check_supported_sw() of the class HARDWARE shows a more detailed example:

$$\text{MC}_{\text{check\_supported\_sw}} = 1 + (2 + 7) + 2 = 12,$$

Where 1 is for sequence and 2 is for call to method is Appropriate() in class SOFTWARE with an MC value of 7. The last 2 in the calculation is for the IF statement (see Appendix 1 for full source).

Here is the full computation of method complexity for all classes in Fig. 1:

$$\begin{aligned} \text{MC}_{\text{COMPUTER}} &= \text{MC}_{\text{getName}} + \text{MC}_{\text{getProducer}} \\ &= 1 + 1 = 2 \text{ CCU} \\ \text{MC}_{\text{HARDWARE}} &= \text{MC}_{\text{getCPU}} + \text{MC}_{\text{getRAM}} + \text{MC}_{\text{getHD}} \\ &\quad + \text{MC}_{\text{getOS}} + \text{MCcheck\_supported\_sw} \\ &= 1 + 1 + 1 + 1 + 12 = 16 \text{ CCU} \\ \text{MC}_{\text{SOFTWARE}} &= \text{MC}_{\text{Software}} + \text{MC}_{\text{getVersion}} \\ &\quad + \text{MC}_{\text{isAppropriate}} \\ &= 4 + 1 + 7 = 12 \text{ CCU} \\ \text{MC}_{\text{DESKTOP}} &= \text{MC}_{\text{desktop}} + \text{MC}_{\text{getCase}} = 1 + 1 = 2\text{CCU} \\ \text{MC}_{\text{NOTEBOOK}} &= \text{MC}_{\text{notebook}} + \text{MC}_{\text{getWeight}} \\ &= 1 + 1 = 2\text{CCU} \end{aligned}$$

B. MESSAGE COMPLEXITY (COUPLING WEIGHT FOR A CLASS (CWC))

Two classes are coupled when there is a message call in one class for the other class [16]. According to the study, ''if there are message calls for other classes, the total number of such messages as well as the weight of the called methods is added. In other words, complexities due to message calls are the sum of weights of call (which is 2 from Table 1) and the weight of called methods represented formally as:

$$CWC = \sum_{i=1}^{n} (2 + MC_i) \qquad (24)$$

Where, 2 is the weight of the message to an external method and MC_i is the weight of the called method. If there are n numbers of external calls, then the CWC is calculated as the sum of weights of all message calls.

In the class given in Fig. 1, the class (HARDWARE) includes one external message call to the SOFTWARE class through the isAppropriate() method. We can therefore calculate the coupling weight of the class HARDWARE as the weight of the called methods''. This gives:

$$CWC = \sum_{i=1}^{n} (2 + MC_i) = 2 + 7 = 9CCU$$

C. ATTRIBUTE COMPLEXITY (AC)

This reflects complexity due to data members (attributes). To compute it, ''the total number of attributes associated with a class is assigned as the complexity due to data members'' [16]. It can be represented formally as:

$$AC = \sum_{i=1}^{n} 1 \qquad (25)$$

Where n is total number of attributes. Therefore, the values of the AC metric for COMPUTER, HARDWARE, SOFTWARE, DESKTOP and NOTEBOOK are 2, 4, 2, 1, and 1 respectively.

D. CLASS COMPLEXITY (CLC)

The complexity of a class is a function of the data attributes and the methods. CLC is computed by summing the attribute complexity as well as the aggregate of all the method complexities of a class represented formally as:

$$CLC = AC + \sum_{p=1}^{n} MC_p \qquad (26)$$

Where, AC is the attribute complexity

MC is the method complexity

Method complexity has already been calculated in section III.A. Therefore, CLC values for the class in Fig. 1 is given thus:

$$\begin{aligned} \text{CLC}_{\text{COMPUTER}} &= 2 + 2 = 4 \text{ CCU} \\ \text{CLC}_{\text{HARDWARE}} &= 16 + 4 = 20 \text{ CCU} \\ \text{CLC}_{\text{SOFTWARE}} &= 12 + 2 = 14 \text{ CCU} \\ \text{CLC}_{\text{DESKTOP}} &= 2 + 1 = 3 \text{ CCU} \\ \text{CLC}_{\text{NOTEBOOK}} &= 2 + 1 = 3 \text{ CCU} \end{aligned}$$

E. CODE COMPLEXITY (CC)

In order to calculate the complexity of entire OO software, ''there is need to consider not only the complexity of all the classes, but also the relations among them. In this regard, emphasis is on the inheritance property because classes can either be parent or children classes of others. In the case of a child class, it inherits the features from the parent class.

Therefore in calculating code complexity of entire OO software program the following is proposed:

If the classes are of the same level then their weights are added

If they are subclasses or children of their parent then their weights are multiplied

If there are m levels of depth in the OO code and level j has n classes then the code complexity is represented formally as:

$$CC = \prod_{j=1}^{m} \left[\sum_{k=1}^{n} CLC_{jk} \right] \tag{27}$$

The unit of CC is taken as 1 Code Complexity Unit (CCU)".

The code complexity value of the class in Fig. 1 is thus:

$$\begin{aligned} CC &= \mathrm{CLC_{COMPUTER}}^{*}(\mathrm{CLC_{HARDWARE}}^{*}(\mathrm{CLC_{DESKTOP}} \\ &\quad + \mathrm{CLC_{NOTEBOOK}}) + \mathrm{CLC_{SOFTWARE}}) \\ &= 4^{*}(20^{*}(3+3)+14) \\ &= 536 \text{ CCU} \end{aligned}$$

Apart from the five metrics proposed in this section, we discuss and demonstrate five others that help to give additional information regarding the projects. The additional metrics are presented as follows:

F. AVERAGE METHOD COMPLEXITY (AMC)

As the name implies, this is used to calculate the average method complexity for any given class by summing the method complexities and dividing by the total number of methods in the class. It is represented formally as:

$$AMC = \sum_{p=1}^{n} MC_p/n \tag{28}$$

Where, MC is the method complexity

n is the total number of methods in a class.

Thus

$$\begin{aligned} \mathrm{AMC_{COMPUTER}} &= 2/2 = 1\mathrm{CCU} \\ \mathrm{AMC_{HARDWARE}} &= 16/5 = 3.2\mathrm{CCU} \\ \mathrm{AMC_{SOFTWARE}} &= 12/3 = 4\mathrm{CCU} \\ \mathrm{AMC_{DESKTOP}} &= 2/2 = 1\mathrm{CCU} \\ \mathrm{AMC_{NOTEBOOK}} &= 2/2 = 1\mathrm{CCU} \end{aligned}$$

G. AVERAGE METHOD COMPLEXITY PER CLASS (AMCC)

This gives us the overall idea of the method complexity of a project by summing the AMC values of all classes that make up a project and then dividing by the number of classes in the project. It is represented formally as:

$$AMCC = \sum_{p=1}^{m} AMC/m \tag{29}$$

Where m is total number of classes in a project

AMC is the average method complexity of a project

Thus AMCC for the example being considered is the summation of the average method complexity of class computer, hardware, software, desktop and notebook calculated as follows:

$$\mathrm{AMCC} = (1+3.2+4+1+1)/5 = 2.04 \text{ CCU}$$

H. AVERAGE CLASS COMPLEXITY (ACC)

This gives an idea of the class complexity of a project by summing its CLC value and dividing by total number of classes represented formally as:

$$ACC = \sum_{p=1}^{m} CLC/m \tag{30}$$

Where, "CLC is the complexity of a class and m is the total number of classes.

$$\mathrm{ACC} = (4+20+14+3+3)/5 = 8.8 \text{ CCU}$$

i.e. the average class complexity of the sample project is 8.8 CCU".

I. AVERAGE COUPLING FACTOR (ACF)

This computes the average coupling that takes place in an OO project and is represented formally as:

$$ACF = \sum_{i=1}^{m} CWC/k \tag{31}$$

Where, CWC is the Coupling Weight for a Class and k is the number of messages to other classes.

$$\mathrm{ACF} = 9/1 = 9$$

The average coupling factor for the project depicted in Fig. 1 is 9 given that there is only one method call to an external class Software from class Hardware.

J. AVERAGE ATTRIBUTES PER CLASS (AAC)

This is represented formally as:

$$AAC = \sum_{i=1}^{m} AC/m \tag{32}$$

Where, AC is the attribute complexity and m is the total number of classes.

$$\mathrm{AAC} = (2+4+2+1+1)/5 = 2$$

i.e. the average number of attributes per class is 2.

IV. VALIDATION OF THE PROPOSED METRICS SUITE

The objective of this section is to validate the proposed object oriented cognitive complexity metrics suite. Validation is a very crucial aspect of the metrics proposition process and as such should be structured in a manner that will be easy to follow by the reader. We have performed a rigorous validation process for our proposed metrics. To achieve this, we followed the guidelines for conducting and reporting case studies in software engineering conducted by Runeson and Höst [30]; we evaluated our metrics theoretically, which includes the validation through measurement theory. The research questions (RQ) that this case study answered are:

RQ1: Do the metrics in the proposed suite qualify as effective measures?

RQ2: What do the metrics mean when applied on real projects?

RQ3: Can the proposed metric suite be used in place of the CK metric suite?

A. THEORETICAL VALIDATION

In the field of theoretical validation, a number of researchers have proposed different criteria [31]–[41], to which proposed software measures should adhere. In this study, we evaluate our metrics against Weyuker's properties and measurement theory as suggested in the framework proposed by Misra *et al.* [42]. The goal of theoretical validation is to ensure that our proposed metrics fulfill some basic requirements to qualify as effective measures thereby answering RQ1.

1) THROUGH WEYUKER's PROPERTIES

Table 5 shows the result of applying our proposed metrics suite as well as the CK metrics suite to thirty object-oriented projects taken from the Web. The CK metrics suite consists of six metrics namely: Response for a Class (RFC), Weighted Methods per Class (WMC), Number of Children (NOC), Depth of Inheritance Tree (DIT), Coupling Between Object classes (CBO) and Lack of Cohesion in Methods (LCOM). Table 5 is used extensively in this section for the purpose of validating our proposed metrics suite.

Property 1: (∃ P) (∃ Q) (|P| ≠ |Q|), where P and Q are object-oriented projects. By observing Table 5, it is clear that among the projects evaluated, there exist a large number of projects that give different values based on each of our metric computation. Hence, this property holds for all our measures.

Property 2: Let c be a non-negative number then there are only finitely many projects of complexity c. All projects have finite number of classes. Cohesion and coupling also takes place between finite numbers of classes. In addition, classes inherit from a finite number of other classes. Thus, given that a project contains a finite number of classes (attributes and methods) with cohesion, coupling and inheritance taking place between these finite number of classes then there are only finitely many projects that will be equal to the measure c. Our metrics thus holds for this property.

Property 3: There are distinct projects P and Q such that |P| = |Q|. As can be observed in Table 5, Projects 18, 19 and 21 are distinct projects that have the same complexity value for MC, CWC, AC, CLC, and CC metrics and thus satisfy this property.

Property 4: (∃ P) (∃ Q) (P ≡ Q&|P| ≠ |Q|). Based on this property, it should be possible for two similar projects to have different complexity values based on MC, CWC, AC, CLC, and CC metrics. It can be observed from Table 5 that although project 6 and project 10 are similar, they possess different complexity values for each of the proposed metric. This can also be observed between project 2 and project 4 as well as between project 23 and project 24. Thus, this property holds for our metrics.

TABLE 2. Metrics values of classes hardware and software.

Class	MC	CWC	AC	CLC	CC
HARDWARE (P)	16	9	4	20	20
SOFTWARE (Q)	12	0	2	14	14
COMBINED (P;Q)	28	9	6	34	34

Property 5: (∀P) (∀Q) (|P| ≤ |P; Q| and |Q| ≤ |P; Q|). This property states that, "the complexity values of two classes P and Q should be less than or equal to the complexity of the composition of the two classes. Considering classes Hardware and Software in Appendix 1 representing P and Q respectively. Let P;Q be a combination of both classes. As can be observed from Table 2, all the five metrics satisfy this property".

Property 6: (∃ P) (∃ Q) (∃ R) (|P| = |Q|) & (|P; R| ≠ |Q; R|). This property asserts that, we can find two classes of equal MC, CWC, AC, CLC, and CC values which when separately combined with a third class yields a class of different MC, CWC, AC, CLC, and CC values. In validating our measures against property 5 of Weyuker's properties, we found out that for any two classes P and Q to have the same complexity values; they must have similar structures (that is, they must be composed in a similar manner). Therefore, combining each of these (P, Q) with another class (R) will also produce similar complexity values for each of our measures. Therefore, our measures do not satisfy this property.

Property 7: There are projects P and Q such that Q is formed by permuting the order of the statements of P, and (|P| ≠ |Q|). Changing the order of the statements (methods in particular) in a class without changing the functionality of the class will not change its complexity value. Therefore, our measures do not satisfy this property.

Property 8: If P is renaming of Q, then |P| = |Q|. Renaming of a project does not change the value of our metrics. As a consequence, this property is satisfied by all of our metrics.

Property 9: (∃ P) (∃ Q) (|P| + |Q| < |P; Q|).

TABLE 3. Summary of evaluation of MC, CWC, AC, CLC and CC metrics through Weyuker's properties.

Property	MC	CWC	AC	CLC	CC
1	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓
3	✓	✓	✓	✓	✓
4	✓	✓	✓	✓	✓
5	✓	✓	✓	✓	✓
6	✗	✗	✗	✗	✗
7	✗	✗	✗	✗	✗
8	✓	✓	✓	✓	✓
9	✓	✓	✓	✓	✓

From Table 2, it can be easily observed that our measures do not satisfy the original Weyuker's property. However, the modified version of this property: (∃ P) (∃ Q) (|P| + |Q| ≤ |P; Q|) is more valuable in evaluating the complexity metrics [38], [40]. Therefore if we refer to the same example used in Property 5 and the metrics values for the classes given in Table 2, it can be observed that all the metrics satisfy this property. Table 3 gives a summary of the evaluation process through Weyuker's properties. The satisfied properties are marked.

2) THROUGH MEASUREMENT THEORY

The research community has proposed diverse criteria to which newly proposed measures should adhere. Most of the propositions agree that newly proposed measures should at least conform to some crucial requirements hinged on the measurement theory perspective [43], [44]. Therefore, in this section, we validate our proposed measures against measurement theory using the Briand *et al.* [33] framework. This framework is reported to be more practical and often used by researchers [45]. We begin our assessment by first providing the basic definitions and desirable properties that make up the framework.

Definition (Representation of Systems and Modules): "A system S is represented as a pair <E, R>, where E represents the set of elements of S, and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between S's elements" [33].

For our proposed complexity metrics suite, we take the entities as classes – meaning E is a set of classes in S. The binary relation on classes is chosen to be greater than or equally complex.

Definition (Complexity): "The complexity of a system S is a function Complexity (S) that is characterized by the following properties namely: non-negativity, null value, symmetry, module monotonicity and disjoint module additive" [33]. We highlight these properties as follows in evaluating our proposed measures:

Property Complexity 1 (Non-Negative): "The complexity of a system S = <E, R> is non-negative if complexity (S) $\geq$ 0" [33].

Proof: All the values obtained from each of our proposed metrics are positive, this property is thus satisfied by all our measures.

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

$R = \emptyset$ ==> complexity (S) = 0" [33].

Proof for MC Metric: If an OO code does not contain any method, then it will have no complexity in terms of its method hence this property is satisfied by MC metric.

Proof for CWC Metric: If coupling does not exist in any given class the value of CWC will be zero as seen in Table 5 (value of metrics for PDB class) hence this property is satisfied by CWC metric.

Proof for AC Metric: If a given class contains no attributes then naturally it will have no complexity value in terms of its attribute hence this property is satisfied by AC metric.

Proof for CLC Metric: If a given class does not contain any method then naturally the complexity value in terms of weight (CLC) is zero and therefore this property is satisfied by CLC metric.

Proof for CC Metric: If a given class contains no method and attributes or inherits from a base class containing no method nor attribute the resulting value for CC will be zero hence this property is satisfied by CC metric.

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})" [33].

Proof: There is no effect on the complexity values of our proposed metrics by changing the order or representation because weights assigned to the method or class cannot depend on the order or way of representation. Therefore, this property is satisfied by all our complexity measures.

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 for all m1 = <E_{m1}, R_{m1}> and m2 = <E_{m2}, R_{m2}> and $m1 \cup m2 \subseteq S$ and $R_{m1} \cap R_{m2} = \emptyset$) ==> Complexity (S) $\geq$ Complexity (m_1) + Complexity (m_2)" [33].

Proof: For this property, if any class is partitioned into two classes, the sum of the complexity values of its partitioned classes will never be greater than the weights of the joined class. This can be observed in Table 2 for all the proposed measures. Therefore, this property holds for all our proposed metrics.

Property Complexity 5 (Disjoint Module Additivity): "The complexity of a system S = <E, R> composed of two modules.

(S = <E, R> and $S = m_1 \cup m_2$, and $m_1 \cap m_2 = \emptyset$) ==> Complexity (S) = Complexity (m_1) + Complexity (m_2)" [33].

Proof: For the metric suite presented in this research, it can be said that the complexity value of the class obtained by concatenating m_1 and m_2 is equal to the sum of their calculated complexity values. If two independent classes are combined into a single class then the weights of the individual classes will be combined. Therefore, this property is also satisfied by our proposed measures.

By fulfilling these properties, one may say that the proposed complexity metric suite is on the ratio scale, which is the most desirable property of complexity measures from the point of view of measurement theory.

3) PRACTICAL VALIDATION WITH KANER's FRAMEWORK

In addition to the theoretical validation using Weyukers' properties and measurement theory, the framework given by Kaner [31], Cherniavsky and Smith [43] can also be adopted for evaluation of our metrics. This approach to metric validation is more practical than the formal approach of Weyukers' properties and measurement theory. The framework is based on providing answers to the following points:

Purpose of the Measures: The purpose of the measures is to evaluate the complexity of object-oriented codes.

Scope of the Measure: Object-orientation is widely adopted in the development of software - ranging from open source to proprietary software. Our measures can be used within and across these projects.

Identified Attribute to Measure: The identified attributes that our measures address are understandability and maintainability. These attributes determine the complexity of an object-oriented code.

Natural Scale of the Attribute: The natural scales of the attributes cannot be defined, since it is subjective and requires the development of a common view about them.

Natural Variability of the Attribute: Natural variability of the attributes can also not be defined because of their subjective nature. It is possible that one can develop a sound approach to handle such attribute, but it may not be complete because other factors also exist that can affect the attribute's variability. In this respect, it is difficult to attain knowledge about variability of the attribute.

Definition of Metric: The metrics suite has been formally defined in Section III.

Measuring Instrument to Perform the Measurement: We computed all the metrics manually – by hand. Further, we aim at developing a tool /software for measuring the proposed suite of metrics.

Natural Scale for the Metrics: The measures are on the ratio scale as mentioned in Section IV, sub-section A and number 2.

Relationship Between the Attribute and the Metric Value: All the proposed metrics contribute to determining the overall complexity of object-oriented code.

Natural Foreseeable Side Effects of Using the Instrument: There are no side effects to using the instrument because once we develop the complexity calculator, it would be very easy to measure all the complexity measures without any extra effort and additional workload of manpower. The only cost will be the result of automation.

4) COMPARATIVE ANALYSIS AND CONCLUSION OF THEORETICAL VALIDATION

It was needful to carry out the theoretical validation of our proposed measures using more than one approach so that the strengths of each could be brought to bear and also complement each other. Weyuker's properties for instance offered us nine properties and our measures satisfied seven out of the nine properties. These results were not convincing enough so we turned to the measurement theory, which has been argued to be suitable replacement for Weyuker's properties [46]. Measurement theory has five properties all of which were satisfied by our metrics suite. This shows that our proposed measures are additive and on the ratio scale. Kaner's framework is a more practical approach to the validation of the measures that involved asking practical questions to prove the usefulness of the proposed measures.

B. EMPIRICAL VALIDATION

In the previous sections, we demonstrated how our metrics suite could be applied to a simple OO project as illustrated in Fig. 1. We have also conducted theoretical and practical validation. However, the true worth of a metrics suite is best proved when applied on real projects. The goal of this section therefore is to answer RQ2 that states thus: What do the metrics mean when applied on real projects?

1) CASE SELECTION

There are several real projects that are adopting object orientation. It is important however that selection be made from those whose source files can be easily accessed. Therefore, in this study, we chose thirty projects from the Web as depicted in Table 4.

2) DATA COLLECTION AND ANALYSIS PROCEDURE

Thirty projects were taken from the Web as earlier mentioned and the specific references are given in Table 4. Microsoft Excel was used to automate the analysis procedure and the results are as given in Table 5.

3) EMPIRICAL VALIDATION RESULTS AND INTERPRETATIONS

The results of evaluating the 30 projects using our cognitive complexity measures are shown in Table 5. From the columns labelled Classes and Methods in Table 5 we observe that it is possible to have projects with same number of classes and methods as seen in project 2, 4 , 6 and 10 as well as projects 18 - 21. It can also be observed that the CLC metric value for a project is equivalent to the sum of the MC and AC metric values.

Lower MC values means lower complexity and increased understandability of code. The factor that affects this is the use of conditional (e.g. if-then statements) and iterative statements (such as for or while statements) in methods. For instance, projects 18, 19 and 21 (Table 5) have the lowest MC value due to the fact that conditional statements and iterative statements are absent in the methods that make up the project. On the other hand, project 30, that has the highest MC value, uses conditional and iterative statements repeatedly in its methods. In addition, it can be observed that projects with up to eight methods will usually have high values for MC metric.

Lower values for CWC means there are less message calls in one class to another within a project, which reduces complexity and improves code understandability. From Table 5 we observe that project 7 has a CWC value of 0 meaning no message calls between classes and thus is easier to understand compared to project 8 that has a CWC value of 116 or project 27 with a CWC value of 366.

AC metric is simply a count of the number of attributes that can be found in a project. It however plays a vital role in helping to determine the value of CLC (i.e. the summation of AC + MC). Since CLC is the result of summing AC with MC, then it means that the higher the MC value of a project, the higher its CLC value as well (see Table 5).

CC metric gives the overall complexity of OO project by considering the inheritance property alongside the CLC value (refer to section III, sub-section E). Projects with lower CC

TABLE 4. Projects used and their references.

Project	Name	Reference
1	maze_runner.py	http://programarcadegames.com/python_examples/f.php?file=maze_runner.py
2	bounce_ball_with_paddle.py	http://ProgramArcadeGames.com/python_examples/en/python_examples.zip
3	breakout_simple.py	http://ProgramArcadeGames.com/python_examples/en/python_examples.zip
4	bullets.py	http://programarcadegames.com/python_examples/show_file.php?file=bullets.py
5	game_class_example.py	http://programarcadegames.com/python_examples/show_file.php?file=game_class_example.py
6	move_with_walls_example.py	http://programarcadegames.com/python_examples/f.php?file=move_with_walls_example.py
7	platform_jumper.py	http://programarcadegames.com/python_examples/f.php?file=platform_jumper.py
8	platform_moving.py	http://programarcadegames.com/python_examples/f.php?file=platform_moving.py
9	platform_scroller.py	http://programarcadegames.com/python_examples/f.php?file=platform_scroller.py
10	sprite_circle_movement.py	http://programarcadegames.com/python_examples/show_file.php?file=sprite_circle_movement.py
11	joystick_calls.py	http://programarcadegames.com/python_examples/show_file.php?file=joystick_calls.py
12	move_sprite_game_controller.py	http://programarcadegames.com/python_examples/show_file.php?file=move_sprite_game_controller.py
13	move_sprite_keyboard_jump.py	http://programarcadegames.com/python_examples/show_file.php?file=move_sprite_keyboard_jump.py
14	move_sprite_keyboard_smooth.py	http://programarcadegames.com/python_examples/show_file.php?file=move_sprite_keyboard_smooth.py
15	move_sprite_mouse.py	http://programarcadegames.com/python_examples/show_file.php?file=move_sprite_mouse.py
16	move_sprites_bounce.py	http://programarcadegames.com/python_examples/show_file.php?file=moving_sprites_bounce.py
17	Sprite_collect_blocks_levels.py	http://programarcadegames.com/python_examples/show_file.php?file=sprite_collect_blocks_levels.py
18	Sprite_collect_blocks.py	http://programarcadegames.com/python_examples/show_file.php?file=sprite_collect_blocks.py
19	Sprite_collect_circle.py	http://programarcadegames.com/python_examples/show_file.php?file=sprite_collect_circle.py
20	Sprite_collect_graphic.py	http://programarcadegames.com/python_examples/show_file.php?file=sprite_collect_graphic.py
21	Spritesheet_functions.py	http://ProgramArcadeGames.com/python_examples/en/python_examples.zip
22	Pong.py	http://programarcadegames.com/python_examples/show_file.php?file=pong.py
23	Snake.py	http://programarcadegames.com/python_examples/show_file.php?file=snake.py
24	tdemo_minimal_hanoi.py	http://svn.python.org/projects/python/trunk/Demo/turtle/tdemo_minimal_hanoi.py
25	bpnn.py	http://arctrix.com/nas/python/bpnn.py
26	tictactoe.py	https://gist.github.com/mawuli/5608979
27	player_skeleton2.py	http://www.blog.pythonlibrary.org/wp-content/uploads/2010/04/Music_Player.zip
28	player_skeleton.py	http://www.blog.pythonlibrary.org/wp-content/uploads/2010/04/Music_Player.zip
29	calc.py	https://www.dropbox.com/s/2ss74kds98orack/calc.py
30	battleship.py	https://code.google.com/p/cs188/source/browse/trunk/src/ScracthFolder/battleship.py?r=114&spec=svn114

values are less complex, easier to understand and maintain compared to those with higher values. Projects 18, 19 and 21 (Table 5) all have the lowest CC value of 2. This is influenced by several factors namely: low method complexity, low attribute complexity which both impact on its CLC. The depth of the inheritance tree is also small. On the other hand, project 8 that has the highest CC value has a high method complexity and CLC value.

C. COMPARATIVE STUDY

The goal of this section is to answer RQ3 that states thus: Can the proposed metric suite be used in place of the CK metric suite?

1) CASE SELECTION

Validation is not complete without being able to compare our proposed metrics suite with an existing one that is well known and adopted by the research community. For this reason, we selected CK metrics suite, a widely adopted and used one.

2) DATA COLLECTION AND ANALYSIS PROCEDURE

In order to perform proper comparison between our proposed metric suite and the CK metric suite, we applied the CK metrics suite in evaluating the projects we selected in section IV, sub-section B and number 1 as shown in Table 5.

3) COMPARATIVE STUDY RESULTS

The results of the comparison we carried out between our proposed metrics suite and the Chidamber & Kemerer (CK) metrics suite are discussed in this section. In Table 4, it is observed that WMC value derives from the number of methods within a project. This is the case for RFC as well and the greater the values of WMC and RFC for a project, the greater the complexity and fault-proneness of that project. It is observed from Table 5 that the highest value for DIT is 2, which is observed in only four of the projects evaluated (i.e. projects 1, 7, 8, and 17). Projects 25, 26, 29 and 30 have a value of 0 while the other projects have DIT values of 1. This means that overall the projects are well designed. The maximum NOC value in Table 5 is 3, which is observed in projects 1 and 8. Project 9 has a NOC value of 2 and Projects 7 and 17 have a value of 1. All the other projects have NOC values of 0 meaning that they are of the same complexity. This is not the case when compared to the results from our metric suite. The LCOM value for the projects in Table 5 gives 0 in all cases meaning all the projects are all of the same complexity, this is also not the case when compared with our proposed metrics. The highest CBO value in Table 5 is 3 and is found in projects 1 and 8. Only projects 7 and 17 have a value of 1 while the rest have a value of zero. The higher the CBO value of a project, the greater its fault-proneness. Table 6 summarizes the differences between the CK metric suite and our proposed metric suite.

TABLE 5. Values of the metrics on ten projects using proposed metric suite and CK metric suite.

			Proposed Metric Suite					CK Metric Suite					
Project	**Classes**	**Methods**	**MC**	**CWC**	**AC**	**CLC**	**CC**	**WMC**	**RFC**	**DIT**	**NOC**	**LCOM**	**CBO**
1	6	8	119	22	4	123	548	8	8	2	3	0	3
2	3	5	66	32	6	72	72	5	5	1	0	0	0
3	3	6	77	45	6	77	77	6	6	1	0	0	0
4	3	5	29	26	0	29	29	5	5	1	0	0	0
5	3	9	176	78	5	181	181	9	9	1	0	0	0
6	2	4	46	12	0	46	46	4	4	1	0	0	0
7	4	12	119	0	5	124	1041	12	12	2	1	0	1
8	6	15	261	116	15	274	4983	15	15	2	3	0	3
9	5	14	191	68	5	196	3030	14	14	1	2	0	0
10	2	4	32	22	0	32	32	4	4	1	0	0	0
11	1	1	11	8	0	11	11	1	1	1	0	0	0
12	1	5	19	9	0	19	19	2	5	1	0	0	0
13	1	2	34	26	2	36	36	2	2	1	0	0	0
14	1	2	14	8	0	14	14	2	2	1	0	0	0
15	1	3	18	8	2	20	20	3	3	1	0	0	0
16	1	2	16	8	2	18	18	2	2	1	0	0	0
17	2	3	25	10	6	31	108	3	3	2	1	0	1
18	1	1	9	8	0	9	9	1	1	1	0	0	0
19	1	1	9	8	0	9	9	1	1	1	0	0	0
20	1	1	13	12	0	13	13	1	1	1	0	0	0
21	1	1	9	8	0	9	9	1	1	1	0	0	0
22	1	2	10	8	1	11	11	2	2	1	0	0	0
23	2	6	89	54	6	95	95	6	6	1	0	0	0
24	2	6	56	34	0	56	56	6	6	1	0	0	0
25	1	10	293	48	0	293	293	10	10	0	0	0	0
26	1	16	359	48	0	359	359	16	16	0	0	0	0
27	2	16	414	366	0	414	414	16	16	1	0	0	0
28	2	9	270	263	1	271	271	9	9	1	0	0	0
29	1	9	123	75	6	129	129	9	9	0	0	0	0
30	6	25	427	109	10	437	437	25	25	0	0	0	0

Table 6 compares our proposed metric suite with the CK metric suite along seven criteria namely: ability to determine method complexity; ability to determine overall complexity of projects; inheritance; coupling; cohesion; metric interconnectedness with the suite and numeric size of metric values. Overall, our metrics are able to differentiate one project from another in terms of complexity compared to the CK metric suite. The proposed metric suite also gives valuable insight into the design quality of OO projects. High CC values indicate that understandability and maintainability of the code is weak. Ultimately, it helps the software developer for better design. For example, the developer, who can satisfy the user requirements through the usage of a lesser number of message calls to other classes, lesser number of inheritance classes, lesser number of branching and looping primitives, is assumed to be more skillful.

D. LIMITATIONS OF THE METRIC SUITE AND THREATS TO VALIDITY

In any proposal, a good metric suite should consider not only the outer structure of OO system (such as number of methods, classes, subclasses and relations between them), but also the internal structure of the method. Although we have compared our proposed metric suite with the CK metric suite, it is important to mention the drawbacks of the proposed metric suite as given below:

1) LIMITATIONS OF THE METRIC SUITE

(i) The proposed metric suite gives a general idea of the complexity of the OO projects thus making it impossible to identify the underlying source of complexity. For instance, in the CK metric suite it is easy to understand the inheritance levels using DIT but in the proposed metric suite it is considered in the calculation of the code complexity.

(ii) The proposed metric suite gives the complexity value in numerical terms, which are generally high for large programs in comparison to the CK metric suite and high complexity values are not desirable.

(iii) It is difficult to assign the upper and lower boundaries for the complexity values.

2) THREATS TO VALIDITY

The main threats to the validity of this study are discussed as follows:

Internal Validity: This aspect of validity is of concern when causal relations are examined and this is not the case in this study.

External Validity: This relates to our ability to generalize the results of this study to industry practice. Although the projects used in this study can be classified as small projects, we believe that the interpretation for these also holds for larger projects.

TABLE 6. Proposed metric suite vs. CK metric suite.

Criteria	Proposed Metric Suite	CK Metric Suite
Ability to determine method complexity	Uses MC metric which is mostly able to differentiate one project's complexity from the other	Uses WMC metric whose value is also able to differentiate the complexity of one project from the other
Ability to determine overall complexity of project	CC metric caters for this	No metric is defined for this
Inheritance	CC considers class inheritance when being calculated for a project and the value is also able to clearly differentiate one project's complexity from the other	DIT metric only shows the level of inheritance and is not able to show the difference in complexity of one class from the other due to inheritance. NOC metric is also unable to differentiate a number of projects from the other (as seen in Table IV) since it does not consider internal factors such as control flow complexity.
Coupling	CWC metric is able to differentiate most of the projects in Table IV in terms of complexity due to coupling	CBO metric shows similar values for a number of projects explored in Table IV. As a result, it is unable to clearly differentiate the complexities of such projects
Cohesion	No metric is defined for this	LCOM metric is defined to, "measure the number of pairs of member functions without shared instance variables minus number of pairs of member functions with shared instance variables". However, the values for all the projects in Table IV are all zero and so this metric is not able to differentiate the complexity of one project from the other.
Metric interconnectedness with the suite	Four of the metrics in this suite are interdependent. For instance, the value of CC is determined by CLC whose value is in turn determined by AC and MC metrics.	The metrics in CK metric suite are independent of each other
Numeric size of metric values	Values here tend to be large because the metrics here usually consider internal factors that affect code complexity such as control flow complexity	The values here tend to be smaller because internal factors such as control flow complexity are not considered.

Construct Validity: The complexity values measuring different object-oriented features of our project are represented numerically. To the uninformed this may seem unclear but higher numerical values indicate higher degree of complexity.

Reliability: This is the extent to which the projects studied and the analysis procedure is dependent on the authors of this paper. During the analysis of the projects, we analysed only the focus of the metrics, that is, the methods and attributes we did not analyse the main program. As a result of the representation of class and method complexities of a project with numbers by our complexity measures, we discovered from this study that although the numeric values obtained by our complexity measures are always positive, it is difficult to set upper boundaries for each of our complexity measures. This is especially challenging as the size of the project increases.

V. CONCLUSION AND FUTURE WORKS

The proposed metrics suite can be employed in determining the complexity of OO projects. This is important since increase in complexity tends to bring about corresponding increase in effort required to maintain such projects. We have not only demonstrated each metric but have also gone ahead to validate them using theoretical and empirical means. For theoretical validation, we began by evaluating each metric based on Weyuker's properties. The results show that our metrics satisfy seven out of the nine properties. It has been argued that it is not compulsory for a measure to satisfy all of these properties to qualify as a good measure [47]. Moving on we compared each against measurement theory using the framework provided by Briand *et al.* [33]. From here, we discovered that each of our proposed metric satisfied the five properties of the framework placing all the metrics in the suite on the ratio scale – which is desired in every good measure. To provide a practical perspective to the metrics suite, we have adopted Kaner's framework in validating the practicality of each of the measures. Our metrics suite thus offers the following features:

1) Each of the metrics is language independent as observed in our study. We used a C++ project to demonstrate how each metric work and we used Python projects to empirically validate the metrics suite.
2) It can be used to evaluate design efficiency of an OO project. "A low complexity value is an indication of better design that in turn impacts positively on maintainability efforts" [48].
3) CWC metric can be used as component level design metric.
4) CC metric calculates the cognitive complexity of OO programs. "Low cognitive complexity is an indicator of good design" [49].
5) The metrics suite measures the important concepts of OO programs such as inheritance and coupling while also evaluating the complexity of the structure of methods.
6) All the metrics in the suite are on the ratio scale, a fundamental requirement for a good measure from the perspective of measurement theory.

For future work, we aim to apply the metric suite for bigger projects from the industry and in different environments. Also, assignments of the upper and lower boundaries of the complexity values for our metrics will be done. Furthermore, the proposed metrics suite will be studied in the light of making improvements to the remaining features of object-oriented code. In addition, an automated tool will also be developed for computing the metrics in the suite.

APPENDIX

This appendix contains the implementation of the classes used in demonstration of the metrics.

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
/*************CLASS COMPUTER***************/
class computer
{
public:
   char * getName(){return name;};
   char * getProducer() {return producer;};
protected:
   char * name;
   char * producer;
};
/**************CLASS SOFTWARE***************/
class software : public computer
{
public:
   software::software(char * cname, char * cproducer, char
* cversion, char * csupportedOS[5]);
   char * getVersion(){return version;};
   short isAppropriate(char * COS);
protected:
   char * version;
   char * supportedOS[5];
};
//WS1= WS11+WS12=1+3=4 //WS2=1
//WS3= WS31+(WS32*WS33)=1+3*2=7
software::software(char * cname, char * cproducer, char *
cversion,
 char * csupportedOS[5]){
   name= cname;
   producer=cproducer;
   version=cversion;
   for (int i=0; i<5; i++)
      supportedOS[i]=csupportedOS[i];
   }
short software::isAppropriate(char * COS){
 for (int i=0;i<5;i++) {
   if (strcmp(COS,supportedOS[i])==0)
      return true;
   }
 return false;
 }
/*************CLASS HARDWARE***************/
class hardware : public computer
{
public:
   char * getCPU(){return CPU;};
   int getRAM(){return RAM;};
   int getHD(){return HD;}
    char * getOS(){return OS;};
   void check_supported_sw(software * s);
protected:
   char * CPU;
//WH1=1
//WH2=1
//WH3=1
//WH4=1
//WH5=WH51+WH52+WH53 =1+9+2=12
int RAM;
int HD;
char * OS; };
void hardware::check_supported_sw(software *
s){//WH51=1(sequence)
short result=s->isAppropriate(OS);
//WH52=2(call)+7(weight of Called method(WS3))
if (result) //WH53=2 (if statement)
cout<<"This is an appropriate software for this
computer"<< \n ;
   else
      cout<<"This is NOT an appropriate software for this
computer"<< \n ;
}
/**************CLASS DESKTOP***************/
class desktop : public hardware
{
public:
desktop(char * cname, char * cproducer, char * ccpu, int
cram, int chd, char * cos, char * ccase); //WD1=1
char * getCase(){return pccase;}; //WD1=1
protected:
   char * pccase;
};
desktop::desktop(char * cname, char * cproducer, char *
ccpu, int
cram, int chd, char * cos, char * ccase){
   name= cname;
   producer=cproducer;
   CPU=ccpu;
   RAM=cram;
   HD=chd;
   OS=cos;
   pccase=ccase; }
/*********CLASS NOTEBOOK*****************/
class notebook : public hardware
{
public:
notebook(char * cname, char * cproducer, char * ccpu, int
cram, int chd, char * cos, float weight); //WN1=1
   float getWeight(){return weight;}; //WN2=1
   protected:
      float weight;
   };
   notebook::notebook(char * cname, char * cproducer, char
* ccpu, int
   cram, int chd, char * cos, float cweight){
      name=cname;
      producer=cproducer;
      CPU=ccpu;
      RAM=cram;
      HD=chd;
      OS=cos;
      weight=cweight; }
   /*============Main Program========*/
   int main () {
      char * supportedOS[5];
      supportedOS[0]="MS Windows";
      supportedOS[1]="Linux";
      supportedOS[2]="";
      supportedOS[3]="";
      supportedOS[4]="";
      software * sw1 = new software ("Rational", "IBM" ,
"7.1",
   supportedOS);
      desktop * ds1 = new desktop("Vaio", "Sony", "Intel core
duo",
   1024, 120, "MS Vista","All-in-One");
      notebook * nb1 = new notebook ("Pavillion", "HP",
"Intel core
   duo", 1024, 120, "Linux",2.5);
      cout<<ds1->getCase()<< \n ;
      ds1->check_supported_sw(sw1);
      cout<<nb1->getHD()<< \n ;
   }
```

REFERENCES

[1] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[2] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 491–496, Jun. 1998.

[3] R. V. Binder, "Design for testability in object-oriented systems," *Comms. ACM*, vol. 37, no. 9, pp. 87–101, Sep. 1994.

[4] S. Purao and V. Vaishnavi, "Product metrics for object-oriented systems," *ACM Comput. Surveys*, vol. 35, no. 2, pp. 191–221, Jun. 2003.

[5] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*. Upper Saddle River, NJ, USA: Prentice-Hall, 1994.

[6] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)," *Object Oriented Syst.*, vol. 3, no. 3, pp. 143–158, Sep. 1996.

[7] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[8] G. Costagliola, F. Ferrucci, G. Tortora, and G. Vitiello, "Class point: An approach for the size estimation of object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 1, pp. 52–74, Jan. 2005.

[9] S. L. Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice*. Chennai, India: Pearson Ed., 1998.

[10] I. Sommerville, *Software Engineering*. Reading, MA, USA: Addison-Wesley, 2004.

[11] T. J. McCabe and A. H. Watson, "Software complexity," *Crosstalk*, vol. 7, no. 12, pp. 5–9, 1994.

[12] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. New York, NY, USA: McGraw-Hill, 2005, pp. 649–672.

[13] Y. Wang, "On the cognitive informatics foundations of software engineering," in *Proc. ICCI*, 2004, pp. 22–31.

[14] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," *Electr. Comput. Eng., Can. J.*, vol. 28, no. 2, pp. 69–74, Apr. 2003.

[15] C. A. R. Hoare *et al.*, "Laws of programming," *Comms ACM*, vol. 30, no. 8, pp. 672–686, Aug. 1987.

[16] S. Misra, M. Koyuncu, M. Crasso, C. Mateos, and A. Zunino, "A suite of cognitive complexity metrics," In *Computational Science and Its Applications*, Heidelberg, Germany: Springer, 2012, pp. 234–247.

[17] D. S. Kushwaha and A. K. Misra, "Robustness analysis of cognitive information complexity measure using Weyuker properties," *ACM SIG Soft. Eng. Notes*, vol. 31, no. 1, pp. 1–6, 2006.

[18] S. Misra, "Modified cognitive complexity measure," in *Computer and Information Sciences*. Heidelberg, Germany: Springer, 2006, pp. 1050–1059.

[19] S. Misra, "Cognitive program complexity measure," in *Proc. 6th IEEE Int. Conf. Cognit. Inform.*, Aug. 2007, pp. 120–125.

[20] A. K. Jakhar and K. Rajnish, "A new cognitive approach to measure the complexity of software's," *Int. J. Softw. Eng. Appl.*, vol. 8, no. 7, pp. 185–198, 2014.

[21] S. Misra, I. Akman, and M. Koyuncu, "An inheritance complexity metric for object-oriented code: A cognitive approach," *Sadhana*, vol. 36, no. 3, pp. 317–337, Jun. 2011.

[22] S. Misra and F. Cafer, "Estimating complexity of programs in Python language," *Tehnički Vjesnik*, vol. 18, no. 1, pp. 23–32, Mar. 2011.

[23] M. Crasso, C. Mateos, A. Zunino, S. Misra, and P. Polvorín, "Assessing cognitive complexity in java-based object-oriented systems: Metrics and tool support," *Comput. Inform.*, vol. 35, no. 3, pp. 497–527, 2016.

[24] G. A. S. Sheela and A. Aloysius, "Design and analysis of aspect oriented metric CWCoAR using cognitive approach," in *Proc. World Congr. Comput. Commun. Technol. (WCCCT)*, 2017, pp. 195–197.

[25] S. Husein and A. Oxley, "A coupling and cohesion metrics suite for object-oriented software," in *Proc. Int. Conf. Comput. Technol. Develop. (ICCTD)*, vol. 1. 2009, pp. 421–425.

[26] S. Akbarinasaji *et al.*, "A metric suite proposal for logical dependency," in *Proc. IEEE/ACM 7th Int. Workshop Emerg. Trends Softw. Metrics (WETSoM)*, May 2016, pp. 57–63.

[27] M. Kaya and J. W. Fawcett, "A new cohesion metric and restructuring technique for object oriented paradigm," in *Proc. IEEE 36th Annu. Comput. Softw. Appl. Conf. Workshops (COMPSACW)*, Jul. 2012, pp. 296–301.

[28] M. Alzahrani and A. Melton, "Defining and validating a client-based cohesion metric for object-oriented classes," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1. Jul. 2017, pp. 91–96.

[29] S. M. Ibrahim, S. A. Salem, M. A. Ismail, and M. Eladawy, "Novel sensitive object-oriented cohesion metric," in *Proc. 22nd Int. Conf. Comput. Theory Appl. (ICCTA)*, 2012, pp. 154–159.

[30] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Apr. 2009.

[31] C. Kaner, "Software engineering metrics: What do they measure and how do we know?" presented at the 10th Int. Softw. Metrics Symp., 2004. [Online]. Available: http://www.kaner.com/pdfs/metrics2004.pdf

[32] C. Kaner, "Rethinking software metrics: Evaluating measurement schemes," *Softw. Test. Quality Eng.*, vol. 2, no. 2, pp. 50–57, 2000.

[33] L. C. Briand, S. Morasca, and V. R. Basily, "Property based software engineering measurement," *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 68–86, Jan. 1996.

[34] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Softw. Eng.*, vol. SE-14, no. 9, pp. 1357–1365, Sep. 1988.

[35] N. Fenton, "New software quality metrics methodology standards fills measurement needs," *IEEE Comput.*, vol. 26, no. 4, pp. 105–106, Apr. 1993.

[36] N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Trans. Softw. Eng.*, vol. 20, no. 3, pp. 199–206, Mar. 1994.

[37] *IEEE Standard for Software Quality Metrics Methodology*, IEEE Standard 1061, 1998.

[38] B. Kitchenham, S. L. Pfleeger, and N. Fenton, "Towards a framework for software measurement validation," *IEEE Trans. Softw. Eng.*, vol. 21, no. 12, pp. 929–944, Dec. 1995.

[39] M. Haug, E. W. Olsen, L. Consolini, L. Bergman, Eds., "Software process improvement: Metrics, measurement, and process modelling," in *Software Best Practice*, 4th ed. Heidelberg, Germany: Springer, 2011.

[40] H. Zuse, *Software Complexity*, New York, NY, USA: Walter de Cruyter, 1991.

[41] H. Zuse, "Properties of software measures," *Softw. Quality J.*, vol. 1, no. 4, pp. 225–260, Dec. 1992.

[42] S. Misra, I. Akman, and R. Colomo-Palacios, "Framework for evaluation and validation of software complexity measures," *IET Softw.*, vol. 6, no. 4, pp. 323–334, Aug. 2012.

[43] J. C. Cherniavsky and C. H. Smith, "On Weyuker's axioms for software complexity measures," *IEEE Trans. Softw. Eng.*, vol. 17, no. 6, pp. 636–638, Jun. 1991.

[44] S. Misra and I. Akman, "Applicability of Weyuker's properties on OO metrics: Some misunderstandings," *Comput. Sci. Inf. Syst.*, vol. 5, no. 1, pp. 17–24, 2008.

[45] D. Abbot, "A design complexity metric for object-oriented development," M.S. thesis, Dept. Comput. Sci., Clemson Univ., Clemson, SC, USA, 1993.

[46] S. Misra, "Evaluation criteria for object-oriented metrics," *Acta Polytech. Hungarica*, vol. 8, no. 5, pp. 110–136, 2011.

[47] D. Baski and S. Misra, "Metrics suite for maintainability of eXtensible Markup Language Web services," *IET Softw.*, vol. 5, no. 3, pp. 320–341, Jun. 2011.

[48] C. Mateos, A. Zunino, S. Misra, D. Anabalon, and A. Flores, "Migration from COBOL to SOA: Measuring the impact on Web services interfaces complexity," in *Proc. Int. Conf. Inf. Softw. Technol.*, 2017, pp. 266–279.

[49] S. Misra and A. Adewumi, "Object-oriented cognitive complexity measures: An analysis," in *Proc. Handbook Res. Innov. Syst. Softw. Eng.*, 2014, pp. 150–169.

SANJAY MISRA has been a Full Professor of computer engineering with Covenant University, Nigeria, since 2010. He has 25 years of wide experience in academic administration and research in various universities in Asia, Europe, and Africa. He has authored over 300 papers and received several awards for outstanding publications. His current research covers the areas of (but are not limited to): software engineering, project management, quality assurance, HCI, AI, cognitive informatics, and web engineering. He is the most productive researcher in Nigeria by SciVal (Scopus Analysis from 2012 to 2017). He has been a founding chair of three annual international workshops: Software Engineering Process and Applications (Springer) since 2009, Tools and Techniques in Software Development Process (IEEE) since 2009, Software Quality (IEEE and LNCS) since 2009. He is a Series Editor of *Advances in IT Personnel and Project Management* (IGI Global), a Chief Editor of the *International Journal of Physical Sciences* (SCOPUS Indexed, last IF- .540), and *Covenant Journal of ICT*. He has delivered over 80 keynote/invited speeches at several international conferences and institutes in over 50 countries.

ADEWOLE ADEWUMI received the B.S., M.S., and Ph.D. degrees in computer science from Covenant University, Ota, Nigeria, in 2008, 2013, and 2017 respectively. He has over seven years experience in teaching and research. His research interests include software metrics, open source software evaluation, and selection using multi-criteria decision-making methods.

LUIS FERNANDEZ-SANZ received the degree in computing from the Polytechnic University of Madrid, in 1989, and the Ph.D. degree in computing with a special award from the University of the Basque Country, in 1997. He has held the position of Vice-President of Council of European Professional Informatics Societies from 2011 to 2013 and in 2016. He is currently an Associate Professor with the Department of Computer Science, University of Alcalá (UAH). With over 20 years of research and teaching experience with UPM, Universidad Europea de Madrid, and UAH, he has also been engaged in the management of the main Spanish computing professionals association (ATI) as a Vice-President and he is a Chairman of the ATI Software Quality Group. His general research interests are software quality and engineering, accessibility, e-learning, and ICT professionalism and education.

ROBERTAS DAMASEVICIUS received the Ph.D. in informatics engineering from Kaunas University of Technology (KTU), Lithuania, in 2005. He is currently a Professor and a Senior Researcher with the Faculty of Informatics, KTU. He has authored or co-authored over 100 papers in refereed international journals and conferences, and a monograph (Springer). His research interests include data mining and brain-computer interfaces. He is a member of the ACM, IEEE, and DAAAM. He is involved in pre- and post-grad level teaching of several courses, including bioinformatics, and human–computer interface design. He is the Editor-in-Chief of the *Information Technology and Control Journal*, and an Editorial Board member of several high-reputed journals, and organizes several conferences and workshops.