Paper No: SC-06 **Smart Computing**

A modified cognitive complexity metric to improve the readability of object-oriented software

Thilini Jayalath*
Faculty of Graduate Studies and Research,
Sri Lanka Institute of Information Technology, Sri Lanka
thilini.j@sliit.lk

Samantha Thelijjagoda
SLIIT Business School
Sri Lanka Institute of Information Technology, Sri Lanka
samantha.t@sliit.lk

***Abstract*: Complexity of software can be identified as a term which expresses the difficulty level of reading, understanding, maintaining and modifying the software. This helps to the quality improvement of the software and maintenance process of the software through a long time period without any obstacle. Therefore, software complexity metrics have been introduced to calculate the complexity of a software using numerical values. While there are number of metrics which calculate the complexity of object-oriented programs, they only consider one or two object-oriented concepts. As a result of that, there is no single metric which has the capability of measuring the complexity of a program based on multiple object-oriented concepts. This research aims to build a new metric to evaluate the complexity of an object-oriented program in order to improve the readability. The new metric has been built based on the influence of previous object-oriented metrics and some disregarded factors in calculating the complexity. In order to evaluate the new metric, Weyuker's properties and Briand's properties are used. The new metric acquires most of the object- oriented concepts in calculating the complexity and helps to improve the readability of the software as well. In fact, it makes it easy to handle the maintainability, reusability, portability and reliability of the software, when readability is high. This will result in increasing the overall software quality.**

***Keywords**: Basic Control Structures (BCS), Cognitive complexity, Object-Oriented Software*

I. INTRODUCTION

Many users frequently say that their applications are difficult to comprehend, hard to be maintained and complex. Thus, quality of the application and quality of the process of building the application were introduced. Then, controlling the quality of the software as well as the development process of the software became very important processes in software engineering. To control this product quality and development process quality, engineers needed a way of measuring the complexity of the software.

There are number of definitions introduced to describe the software complexity. Some of them are as follows.

- Harmeet and Gurvinder have described the software complexity as the primary factor of performance, reliability and cost of the software. Software complexity has a major effect on the required effort to identify the requirements, design, implement, test and maintain the system during the software life cycle [1].
- Madi, Zein and Kadry state that the software complexity can arise anywhere inside Software Development Life Cycle (SDLC) such as, analysis phase, requirement gathering phase, design phase and implementation phase. They refer the complexity of a software as an undesired property which makes the software tougher to comprehend, therefore harder to be modified [2].

A single definition cannot be used to describe the term software complexity since it is a multi-dimensional term in software. Different researchers have used different definitions and views about software complexity. Therefore, authors have decided to describe the software complexity of a system as the difficulty in understanding, changing and maintaining the system [3].

In order to measure the complexity and express the complexity using numeric values, software metrics have been introduced to the software industry. The beginning of publishing the research papers on software metrics commenced in 1968. R.J. Rubey and R.D. Hartwick published a paper titled "Quantitative Measurement of Program Quality" at the ACM National Conference, Las Vegas in August 1968. Thereupon the community of software industry always works on proposing new software measures and modifying existing ones to calculate the software complexity in a more accurate way. Program complexity depends on a large number of factors. Number of inputs and outputs are very critical in measuring program complexity as well as BCSs and cognitive weights. When a program has a large number of inputs and outputs, it makes the program more complex. In fact, a programmer or reader needs to pay more attention to remind those attributes and their I/O processes. Human effort which is needed to perform a task is also a very important when it comes to measuring the complexity. That is called cognitive complexity measures.

The challenge in programming can be defined in terms of the way to build the logic, not how to describe the data. But object-oriented programming paradigm is more concerned about managing the objects rather than the logic required to manage them. The first step in object-oriented programming is to determine each and every object that the developer wants to manage and identify the relationship among them. Object can be specified as an autonomous entity which contains both data and procedures to manage the data. A class can be called as a prototype from which objects are built and that explains the details of related object. A class comprises three things: the name, attributes and operations.

To endure a software/system in a long time it should be properly maintained. For that, the program code of the software/system should be understandable and readable. Then the program code can be changed according to the available and considerable reasons. In fact, the understandability and

readability features of the software are very important for the maintenance phase of the object oriented (object-oriented) software life cycle.

II. LITERATURE REVIEW

IEEE defines a metric as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute" [4].Quantitative measurements are very important in all sciences, computer science practitioners and theoreticians make a continuous effort to bring similar approaches to software development [5].

Object oriented programming is wrapped with three main factors called, objects, classes and methods (functions). In 1999 Buckley, Layzell and Douce introduced a set of metrics that help in calculating the complexity of a given system or program code based on the object-oriented concepts such as the object and class. All those metrics were based on the spatial abilities which measure the complexity by calculating the distances between the program elements in the code. Spatial abilities benefit in order to read, understand and remember the program. Therefore it is easy to maintain the software. Authors proposed new metrics set which includes [6];

- A metric to measure the complexity of a distinct function [Function Complexity-FC]
- With the guidance of FC, measure program complexity where a program consists of one or more than one function. [Program Complexity-PC]
- A metric to calculate the complexity of a recursive method [RFC]
- Three metrics to measure object-oriented concepts
  - Method Location Rating [MLR]: Here consider the count of lines in between the method definition and class declaration.
  - Class relation measure [CRM]: Here consider the number of lines between the parent class and child class.
  - Object relation measure [ORM]: Here consider the number of lines between the object declaration and its class declaration.

These three metrics require a simple calculation process and it helps to measure the understandability of the source code. Other than that, MLR, CRM and ORM benefit in estimating the cost and time needed. But there are some drawbacks in these metrics, such as;

- Depending on the language
- Having minor barriers when applying some of those metrics in few incidents like; CRM cannot apply where class definition is unavailable inside the code and ORM cannot apply where the definition of object is not inside the available code
- Requiring considerable theoretical basis and implementation technological knowledge
- Not supporting to all object-oriented programming concepts
- Not considering the complexity arisen due to attributes of the class

In 2004 Kumar, Singh and Aggarwal have accomplished to propose two metrics [7], which consider the object-oriented concepts, such as polymorphism and encapsulation that help to measure spatial abilities. They attempted to measure the spatial complexity based on two categories as shown in Fig. 1.

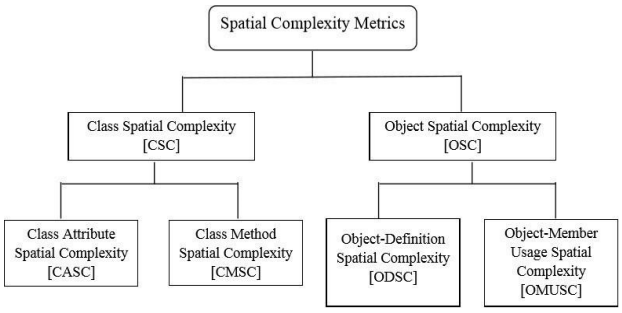

Fig. 1. Categories of spatial complexity metrics

These measures benefit in increasing program readability, measuring maintainability, evaluating the suitability of the class data members, improving and comparing the class cohesiveness and understandability. Based on previously mentioned metrics, these two also depend on the language and considerable theoretical basis and implementation technology knowledge. Since calculating CSC and OSC has a lengthy calculation process it is hard to calculate. Object oriented system can be a collection of objects, classes and methods.

In order to find the complexity of an object-oriented system, it is very essential to measure the complexity of entire system rather than measuring complexity of a particular object or class. Therefore, in 2007 Sanjay Misra [8] proposed a contemporary measure to calculate object-oriented system complexity. The following three steps can be used to calculate complexity using new metric proposed.

- Measure the complexity that arise due to operations upon each and every object
- Enumerate the complexity of individual object or class
- Lastly, calculate full code complexity (in entire object-oriented system)

In order to calculate the class complexity (CC), first calculate the complexity of individual classes based on cognitive weights of BCSs. Then calculate the total Class Complexity (CC) by adding the class complexities of all classes that exist inside the entire code (system) compared to CSC and OSC measures, calculating CC is little easy. In fact, it does not require much theoretical basis and implementation technological knowledge and accomplish the requirements that are needed for an acceptable and good metric. CC is language independent and a robust measure. It can be used to evaluate understandability and efficiency of the code. But when it comes to very lengthy source codes it can be somewhat hard to calculate. Most of the metrics proposed earlier have the following drawbacks.

- Absence in consideration of some object-oriented programming concepts and principles of measurements when designing the metrics
- Depending on technologies of implementation
- Not accommodating any convenient approach to measure the entire class/code complexity

By considering all those shortages Misra and Akman have introduced a new metric to measure the entire class/code complexity followed by an object-oriented approach in 2008 [9]. This measure is based on cognitive weights and the most important fact is this metric has considered both the complexity arisen as a result of the data members (attributes) and complexity arisen considering the function operations in the class/entire code. Calculating the total WCC does not depend on the language and it conjectures the effort of maintenance. WCC calculates the complexity of methods with regard to the messages and operations in those. It also acknowledges the method internal architecture which is called method complexity. While these being the advantages of WCC, disadvantages, on the other hand, are somewhat hard to calculate when it comes to lengthy codes and not supporting all object-oriented programming concepts. Since this method provides complexity values in a numerical format, it can be a large number for lengthy codes. High values of complexities are undesirable.

In 1999 Douce, Layzell and Buckley found set of spatial metrics like function complexity (FC), program complexity (PC), object relation measure (ORM) and class relation measure (CRM) to calculate the spatial abilities of object-oriented program [9]. But there was no consideration of architectural complexity (cognitive weights). Due to that in 2009 Gupta and Chhabra found new metrics to calculate the complexity of a program with regards to two aspects [10];

- Spatial aspect – using LOC
- Architectural aspect – using cognitive weights

Here Gupta and Chhabra have categorized the cognitive spatial complexity metrics of object-oriented software as Fig. 2.

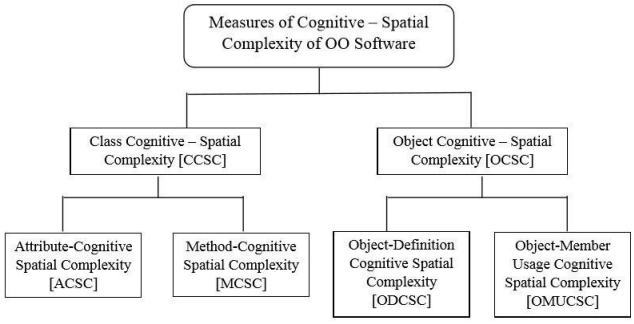

Fig. 2. Categories in measures of cognitive-spatial complexity of object- oriented software

This is a very good measure to indicate the cognitive effort needed to understand the program. Disadvantages of these metrics are; language dependency, difficulty that arises when calculating since it has a lengthy calculation process and it needs a considerable theoretical basis and technology knowledge.

In 2011, Koyuncu, Misra and Akman introduced a new measure to calculate the full code complexity considering the inheritance positions and the interior architecture of the code since the traditional metrics which tried to calculate the complexity of the object-oriented programming codes had the following issues [11

- Not acknowledging the interior architecture of the code and object-oriented concepts
- Not considering cognitive aspects
- Not being able to cover some theoretical and mathematical approaches
- Not being hugely based on implementation technology
- Not being able to measure the absolute complexity of a code acknowledging the inheritance positions and cognitive aspects in sync.

This measure has become prohibitive to appoint upper and lower bounds for complexity values. Since this method provides complexity values in the numerical format, it can be a large number for very lengthy codes. High values of complexities are undesirable. As a result of these issues Chhillar and Bhasin came up with a metric called new weighted complexity metric (WCM) in 2011 [12]. Chhillar and Bhasin acknowledged the following factors as Fig. 3, in order to propose a new metric.

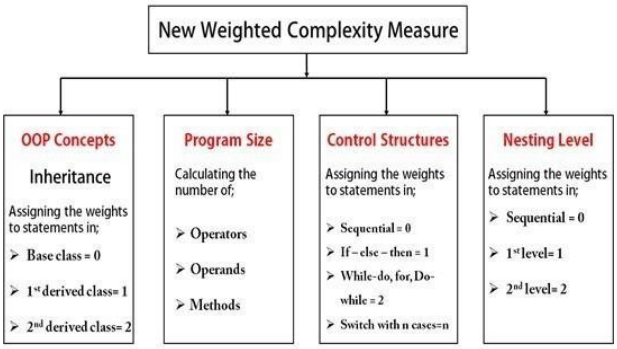

Fig. 3. Considered factors for WCM

In 2015, with confiding these circumstances Hussien, Jararweh, Shehab, Alandoli and Tashtoush came up with a new measure which concerns six elements that causes emergence of the program/system complexity [13]. Those six elements are indicated in Fig. 4. Out of these six, first two elements are extracted from the previously proposed approaches. In generating flow charts these authors followed the footsteps of Thomas J. McCabe in developing flow charts in Cyclomatic Complexity (CC) but assigned the weights as different from CC but same as Shao and Wang's measure. When calculating the operations in the program authors adopted the approach of Halstead volume. Other four factors are proposed by the authors in order to boost the accuracy of the complexity calculation.

A very strong advantage of the new measurement is that it depreciates the detriments of previous metrics and the disadvantages are that it does not consider the complexity that arises due to the objects created in the class and thus requires considerable theoretical basis and implementation technological knowledge.

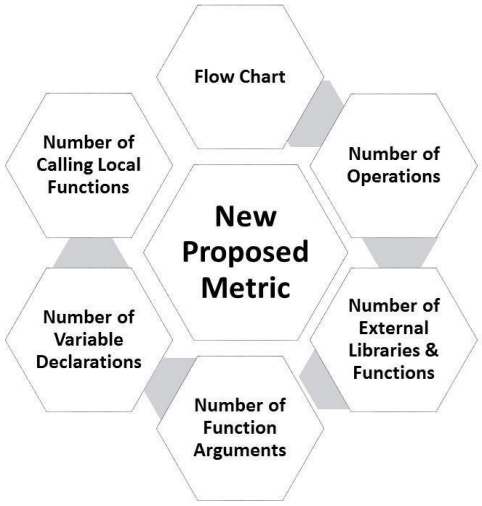

Fig. 4. Factors considered in the new metric

Sanjay Misra et al. (2018) published a paper on Object Oriented Cognitive Complexity Metrics and the authors have proposed a metric suite which covers some important features of object-oriented programing [14]. They have analyzed the literature and extracted some distinct features for the new metric. There are five essential metrics that exist within this suite;

- Class Complexity (CLC)
- Attribute Complexity (AC)
- Message Complexity (CWC)
- Method Complexity (MC)
- Code Complexity (CC)

This proposed metric suite considers the object-oriented concepts like inheritance, coupling and cohesion.

In 2019, Hussam, Hiba and Tharmer came up with a new model called The 2O2C model to calculate the complexity of object oriented programs [15] The authors have acquired some attributes from the literature including: NOC, DIT, COM and CBO. Other than these four, detailed class complexity and weighted class complexity have been considered by the proposed metric model. 2O2C metric model mainly focuses on attaining consistency, reusability, readability, extensibility, maintainability and understandability. Abstraction and encapsulation are the main two object-oriented concepts that were considered in this metric model.

As mentioned, the count of operands and operators, control structures, cognitive weights, nesting levels and object-oriented programing concepts like inheritance, polymorphism, and abstraction should be considered during the complexity calculation of an object-oriented program. Previously found metrics have not been able to consider all these parameters together in calculating complexity of an object-oriented program.

III. PROPOSED WORK

Even though there are number of metrics which calculate the complexity of object-oriented programs, they only consider one or two object-oriented concepts for the metric. As a result, there is not any single metric which has the capability of measuring the complexity of a program based on most of object-oriented concepts. This research is to build a new metric called Modified Cognitive Complexity Metric to calculate the complexity of a given object-oriented program considering more concepts of object-oriented programing such as,

- Counting ELOC for the size attributes other than operators, operands and strings
- Polymorphism and Abstraction
- Coupling and Cohesion
- Inheritance
- Encapsulation

A. Counting ELOC for the size attribute

Here, in order to calculate the size of any program, the authors have considered the calculation process of a size attribute of WCM [12]. Size is a very strong factor in measuring complexity in a system or module, because it is very simple to understand that the programs which are massive in size are more difficult to comprehend than the programs which are small. In fact, the size attribute plays a critical role in measuring complexity. There are number of ways to measure the size of a program/ module:

- By directly counting Lines of Code (LOC)
- By counting the number of operators, operands and strings in each program
- By counting the number of methods and classes in each module
- By measuring the distance between variable and method declarations and usages.

When it comes to counting the number of operators, operands and strings in each program, it requires the user to know the difference between operators, operands and strings. In addition, this approach requires the user to go through each element in the program which is highly time-consuming. Counting the number of methods and classes is not a very strong method in measuring the size of the program, because there can be modules which contain only one method, but that contains many LOC which is more complex. Measuring the distance between variable, method declarations and usages also have several disadvantages such as, the user is expected to remember the position of a declaration of a certain variable and the positions (calling statements) of using that variable or method. Thus, this method is also very tedious and time consuming. In comparison, directly counting the LOCs approach is very much simpler and less time consuming as the user is just required to count the number of lines of code in a certain block of code. Thus, the author decides to measure the size of a program by counting the LOC in it. But there are various ways of counting LOC of a program. Those can be described as follows.

- Number of physical lines – Counting the lines of the source code of program including comment lines

- Number of blank lines – Counting only the blank lines in the given program
- Number of logical lines – Counting only the lines which are executable statements
- Number of eLOC – Counting the lines of the source code of program excluding blank, comment lines and lines which include only parenthesis.
- Sum of lines of code – Getting the sum of physical lines and blank lines
- Number of executable physical – Getting the total lines of source code excluding the blank lines and comment lines
- Number of executable logical – Counting the number of executed statements inside the given program or module
- Number of comment lines – Counting the number of comment lines
- Number of comment words – Getting the count of comment words in the program

Since eLOC eliminates lines which contain only a bracket and count every other line excluding comments and blank lines, eLOC can be taken as the replacement for size attribute. Since counting eLOC is easier than counting operators, operands and strings, that can be proposed for the size attribute. In order to calculate the size of any given program, counting the number of eLOC can be considered in Modified Cognitive Complexity Metric (MCCM).

B. Polymorphism and abstraction

Polymorphism is a main concept of object. This feature can be called as when one method has multiple implementations, for a certain class of action. The most popular way in using polymorphism in object-oriented programing, where a child class object is referred by a parent class reference. Polymorphism can be further described by considering its two main parts:

- Static polymorphism [Overloading] - Static polymorphism also called early binding, compile time binding, because it happens during the compile time. This feature says that a class can have more than one method with the same method name, if their argument lists are different. When someone is referring to a code, this object-oriented feature makes it hard to understand the code for the person. Thus the person may get confused to identify the method that called.
- Dynamic polymorphism [Overriding] - Dynamic binding is also called late binding and runtime binding, since it happens during the run time of a program. Overriding of a program refers when implementing a method in subclass which is already present in the relevant parent class. A person who will read the code may get slightly confused with this situation.

A class can be called as an "abstract class", when it contains one or more abstract methods. When a method is declared without any implementation details, it is called as "an abstract method". Abstract classes may not be instantiated and require subclasses to provide implementations for the abstract methods. It is somewhat hard to understand an abstract method than a normal method. This is because in parent class there are no any implementations for abstract methods. There can be different implementations for different child classes. Taking these three problems into account, in order to measure these complexities, a new factor can be proposed, which is Wo. The above two situations can be considered by adding value of one to the Wo factor for a statement which calls an overloaded or overridden or abstract method. Other than those two situations, in overloading it should add value of one to the Wo factor for a statement which is an overloaded method declaration excluding the first method declaration.

C. Coupling and cohesion

Coupling and cohesion interact with the quality of an object- oriented design. Commonly, a good object-oriented design must be highly cohesive and loosely couple. This type of system is easy to develop, add new features, maintain and is less fragile.

Coupling can be described as the relationships between modules. A decrease in interconnections between classes (or modules) is therefore accomplished via a decrease in coupling. Coupling is categorized into many types according to the reasons that can arise between modules. Those can be listed as in TABLE I, in the order of complexity (Lower to highest) Cohesion is a measure that can be used to specify the degree to which a class has a well-focused or single purpose. Cohesion mainly emphasizes how a single class is designed. A better object-oriented design holds a high cohesion. There are 9 levels of cohesion as in TABLE II (Better to worst).

In order to acknowledge the complexities of above categories of coupling and cohesion, a new attribute, Wcc should be introduced, where the value of Wcc can be taken from the TABLE I and TABLE II.

TABLE I. SUGGESTED WEIGHTS ACCORDING TO COUPLING TYPES

Coupling Type	Weight [Wcc]
Data coupling	1
Stamp coupling	2
Control coupling	3
Common coupling	4
Content coupling	5

TABLE II. SUGGESTED WEIGHTS ACCORDING TO COHESION TYPES

Cohesion Type	Weight[Wcc]
Functional	1
Informational	2
Sequential	3
Communicational	4
Procedural	5
Temporal	5
Logical	5
Coincidental	5

D. Inheritance

Chhillar and Basin computed the complexity that arises due to different levels (nesting levels) of inheritance of classes

using Wi. But in object-oriented programming there are different types of inheritance (Fig. 5) which can be identified. That situation is not acknowledged in WCM. The different types of the inheritance can be listed as follows:

- Single inheritance: A class can extend only a single class.
- Hierarchical inheritance: One single base class can create more than one derived classes.
- Multilevel inheritance: One derived class is created from another derived class.
- Multiple inheritance: One class inherits from more than one base class.
- Hybrid inheritance: This is a mixture of any of the above inheritances (single, hierarchical and multilevel) which can be called as hybrid inheritance

Based on the complexity level of the inheritance types, weights are assigned (TABLE III). Here in the traditional calculation process of WCM, it allocates a weight for class inheritance level as; level 1 = 1, level 2 =2 and level 3 = 3, but it does not consider about these inheritance types. In order to acknowledge both these two factors, Wi attribute can be used as follows.

Wi = weight due to inheritance level * weight due to inheritance type

TABLE III. SUGGESTED WEIGHTS ACCORDING TO INHERITANCE TYPES

Inheritance Type	Weight [Wty]
Single	1
Multilevel	1
Hierarchical	2
Multiple	3
Hybrid	4

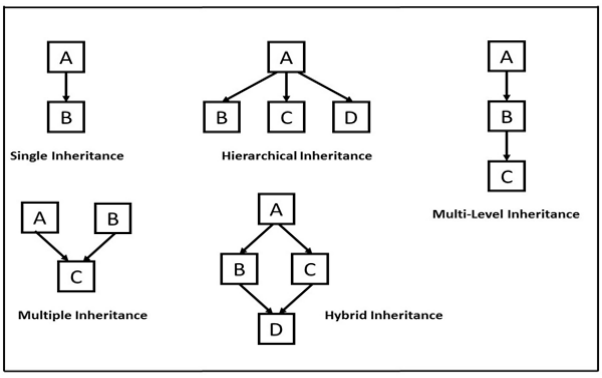

Fig. 5. Different types of inheritance

E. Encapsulation

The process of packing the variables (data) and code performing on the variables (functions) together as a single unit is called as encapsulation in object-oriented programming. In here the variables declared I a one class will be not exposed to any other classes and cannot be accessed by any other function which is not in the current class. Because of that this concept is also called data hiding. In order to achieve data hiding in object-oriented programing, following two ways can be used.

- Declaration of variables in a class should be private.
- Deliver public getter and setter functions to view and change the variable values.

To consider this concept in calculating complexity, authors have decided to introduce a new variable called We. The statements which declare the variables as private and the statements which contain the public getter and setter functions declarations should allocate a value of one for the We attribute.

Assuming these 5 factors total weight of a single line can be proposed as follows.

$$W = Wi + Wc + Wo + Wcc + We \qquad (1)$$

Wi - Weight due to the inheritance, Wc - Weight due to the type of BCS * Weight for nesting level of BCS, Wo - Weight due to abstraction and polymorphism, Wcc-Weight due to coupling and cohesion, We - Weight due to encapsulation.

Then considering the line by line calculation for the weights, complexity of the entire code can be taken by following formula;

$$MCCM = \log [(\sum_{n=1}^{P_n} W) * ELOC] \qquad (2)$$

Here it gets the log value, in order to eliminate receiving large numeric values for lengthy programs.

IV. RESULTS

Weyuker has established a formal list of properties (nine properties) in order to estimate the accuracy of software metrics. It has been used to evaluate numerous existing software metrics and it is not mandatory to that all nine properties should be satisfied by the metrics. This framework is used by many object-oriented metrics and is theoretically validated.

A. Property 1:(∃P)(∃Q)(|P | = |Q|)
Where P and Q are two disparate classes

This condition claims that a metric should not rank all programs as similarly complex. Since the considered two programs have totally different internal structures from each other and the formula giving two disparate values for MCCM, the condition is satisfied by this measure.

B. Property 2: Let c be a nonnegative value and then there are only finitely many programs of complexity c.

Every object-oriented programing (OOP) language contains only finite count of variables, methods, cognitive weights of basic control structures (BCS) and classes. MCCM depends on the weight of inheritance, type of BCSs, variables, encapsulation, abstraction, coupling, cohesion and eLOC; where all these factors are finite length program. Therefore, MCCM satisfy property 2.

C. Property 3: There are P and Q distinct programs such that (∃P)(∃Q)(|P | = |Q|)

Here condition 3 says that there can be classes with the equal complexity value. When considering MCCM there can be multiple classes containing the same MCCM. Hence this property is satisfied by MCCM.

D. Property 4: (∃P)(∃Q)(P ≡ Q and |P |=|Q|)

This 4th property states that even though two programs compute the same function, it is the details of the implementation that determine the program's complexity. Even though the functionalities of two programs are equal, the complexity of the programs is based on the implementation body of the program. Because of that MCCM holds the property 4.

E. Property 5: (∀P)(∀Q) (|P | ≤ |P ; Q| and |Q| ≤ |P ; Q|)

Modified cognitive complexity measure collaborates with weight of inheritance, type of BCSs, variables, encapsulation, abstraction, coupling, cohesion and eLOC which are always integers. Thus the sum of integers is always a non-negative integer. Additionally, it is true for all non-negative integers P and Q that: (Q ¡ P + Q) and (P ¡ P + Q). This confirms that raised elements of MCCM are comparable to Weyuker's 5th property. Since condition 5 is fulfilled by the MCCM measure.

F. Property 6:
6a: (∃P)(∃Q)(∃R)(|P | = |Q|) and |P ; R| ≠ |Q ; R|
6b: (∃P)(∃Q)(∃R)(|P | = |Q|) and |R; P | : |R ; Q|

These properties state that if there are two programs P and Q with same complexities and when they are combined with the same third program R, combined programs should be varied. In composite complexity measure the complexity of the program depends on the number ELOC and the cognitive weights which are not going to change due to the program combination. But it also considers the way of making the two programs interact which may differentiate the complexities of two combined programs. As an example, P program calls the R in a sequence statement (where; Sj * (Wt)j can be 0) and Q program can call the R program inside a for loop (where; Sj * (Wt)j can be 0) which may increase the complexity than the previous situation. Due to this reason MCCM satisfies this property of Weyuker's.

G. Property 7: There can be P and Q program bodies such that Q is designed by permuting the order of the statements of P, (|P | =|Q|)

In Object-oriented programming, changing the sequence of attribute, method declarations or order of statements do not influence the order of execution. As a result, the proposed measure does not satisfy this property.

H. Property 8: If P is renaming of Q, |P | = |Q|

This property states that the complexity of a class P does not change even it changes the class name. Changing the class name will not affect the token count or cognitive weights of the BCS s. Since it does not affect MCCM measure, this property is also satisfied.

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

This condition states that the addition of complexities of two separate classes is lower than the complexity of a class which is created by joining those two separate classes. Usually, two classes can have a finite number of unique functions with some cognitive weights. A combined class of the two individual classes would result in one class's version of the unique methods becoming redundant. Therefore, the complexity of the combined class in terms of cognitive weight reduces. Because of this situation this condition is not fulfilled by MCCM metric.

According to the above explanation, modified cognitive complexity measure satisfies all the properties of Weyuker's framework excluding 7th and 9th.

V. CONCLUSION AND FUTURE WORK

Object-oriented programming makes the software development easy and more convenient by having the following expediencies over other approaches.

- Providing a fair modular structure for programs
- Creating new objects with slight differences to actual ones, which makes easy to maintain and change the existing code
- Providing an excellent framework for code libraries.

Large number of OO complexity metrics have been introduced with the enlargement in the demanding of the object-oriented programming. These metrics can be used to measure the readability, accuracy and maintainability of the software. But there is not any single metric which addresses most of the object-oriented concepts. Thus, this research is conducted to build a new metric called modified cognitive complexity metric to improve the readability of the object-oriented software as follows.

$$\text{MCCM} = \log\left[\left(\sum_{n=1}^{P_n} W\right) * ELOC\right]$$

This new metric gives the number of advantages as follows;

- Considers most of the object-oriented concepts within a single metric
- Considers the size attribute (spatial quality) as well.
- Easy to calculate.
- Can be used to improve the readability, maintainability of the program by reducing the complexity value.

Software metrics can be applied to measure an application or define specifications in order to provide ease for readability, maintainability, analyzing and modifying the above areas. The data gained by the software metrics can be used in numerous ways in order to benefit the organizations such as budget planning, risk management, scheduling and software quality as a complex IT infrastructure expands or changes.

The information gained from the software metrics collection process helps organizations to improve scheduling, budget planning, cost estimation, software quality, and risk mitigation as a complex IT infrastructure enlargement or changes. Following points can be taken as the future developments of this research.

- Polish the formula in order to consider compound conditional statements
- Normal method calls and recursive method calls
- Automate the calculation processes

REFERENCES

[1] Uk.Ijeacs "Software complexity measurement: A critical review.",International Journal of Engineering and Applied Computer Science (IJEACS), 01:12–16, 12 2016.

[2] A. Madi, O.K. Zein, and S. Kadry. "On the improvement of cyclomatic complexity metric", International Journal of Software Engineering and its Applications, 7:67–82, 01 2013.

[3] Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics", IEEE Transactions on Software Engineering, SE-5(2):96–104, March 1979.

[4] "IEEE standard for a software quality metrics methodology", IEEE Std 1061-1998, pages i–, Dec 1998.

[5] Thaku. Software Metrics in Software Engineer- ing. http://ecomputernotes.com/software-engineering/software-metrics, [Accessed: 12-06-2019].

[6] Douce and P. Layzell. "Spatial measures of software complexity", 04 1999.

[7] K. Chhabra, K.K. Aggarwal, and Y. Singh. "Measurement of object-oriented software spatial complexity", Information and Software Technology, 46(10):689 – 699, 2004.

[8] Misra. "An object-oriented complexity metric based on cognitive weights", In 6th IEEE International Conference on Cognitive Informatics, pages 134–139, Aug 2007.

[9] S. Misra and I. Akman. Weighted class complexity: A measure of complexity for object-oriented system. J. Inf. Sci. Eng., 24:1689–1708, 11 2008.

[10] Gupta and J. Chhabra. Object-oriented cognitive-spatial complexity measures. 01 2009.

[11] S. Misra, I. Akman, and M. Koyuncu. An inheritance complexity metric for object-oriented code: A cognitive approach. c Indian Academy of Sciences, 36:317–337, 07 2011.

[12] U. Chhillar and S. Bhasin. A new weighted composite complexity measure for object-oriented systems. 2011.

[13] Mohammed, Shehab, M.Yahya, Tashtoush, Wegdan A. Hussien, N.Mohammed , Alandoli, Yaser, and Jararweh. An accumulated cognitive approach to measure software complexity. 2015.

[14] S. Misra, A.Adewumi, L. Fernandez-Sanz, and R. Damasevicius. A suite of object-oriented cognitive complexity metrics. IEEE Access, 6:8782–8796, 2018.

[15] Hourani, H. Wasmi, and T. Alrawashdeh. A code complexity model of object-oriented programming (oop). In 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), pages 560–564, April 2019

.