Improvements to a Complexity Metric: CB Measure

D. I. De Silva[1], N. Kodagoda[4]
Faculty of Computing
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
[1]dilshan.i@sliit.lk, [4]nuwan.k@sliit.lk

S. R. Kodituwakku[2], A. J. Pinidiyaarachchi[3]
Faculty of Science
University of Peradeniya
Peradeniya, Sri Lanka
[2]salukak@pdn.ac.lk, [3]ajp@pdn.ac.lk

Abstract— **Controlling the complexity of software applications is an essential part of the software development process as it directly affects maintenance activities such as reusability, understandability, modifiability and testability. However, as stated by Tom DeMarco "You cannot control what you cannot measure". Thus, over the years many complexity metrics have been proposed with the intention of controlling and minimizing the complexity associated with software. However, majority of these proposed complexity metrics are based on only one aspect of complexity. The CB measure introduced by Chhillar and Bhasin is one metric which relies on a number of complexity factors to decide on the complexity of a program. However, it also has some shortcomings and can be further improved. Thus, this paper attempts to propose some additional complexity factors that the CB measure has not considered, to further improve it. The paper also presents an extensive coverage about the software complexity metrics proposed in the literature.**

Keywords—CB measure; software complexity; software complexity metrics

I. INTRODUCTION

Controlling the complexity of software applications is an essential part of the software development process as it directly affects maintenance activities such as reusability, understandability, modifiability and testability [1]. However, as stated by Tom DeMarco "You cannot control what you cannot measure [2]."Thus measuring software complexity has been a widely distributed research area for many years.

As a result, over the past several years, various complexity metrics have been introduced based on various attributes of software, such as control flow [3], operator and operand count [7], information flow [11], data access[16], data flow [20], branching complexity [30], logical complexity [21], spatial complexity [22], [23], object orientation [12], [1], entropy [24], [25], category learning [10], and cognitive complexity [27], [28], [29] with the intention of controlling and minimizing the complexity associated with software.

Most of the proposed complexity metrics considered only one aspect of complexity. However, as correctly stated by Widheden, complexity is too abstract and multifaceted to be measured directly by considering a single aspect. Rather it can be measured as a combination of multiple aspects that would affect the complexity of a program. The weighted composite complexity measure (CB measure) introduced by Usha Chhillar and Shuchita Bhasin is one metric which relied on a number of complexity factors to decide on the complexity of a program. However, like with all metrics, this metric also has some shortcomings and can be further improved. Thus, this paper attempts to propose some additional complexity factors that the CB measure has not considered, to further improve it.

This paper initially presents an overview of software complexity. The readers' attention is then drawn towards software complexity metrics proposed in the literature. Then it discusses about Chhillar and Bhasins' CB measure. Later in the paper, some additional complexity factors that can be considered by the CB measure to improve its accuracy are proposed, followed by a section concluding the paper.

II. BACKGROUND

A. Software Complexity

An initial problem that arises when trying to understand software complexity is defining what it really means. According to the Oxford Advance Learner's Dictionary, complexity means "the state of being formed of many parts or the state of being difficult to understand" [15].

Meanwhile, Basili interprets software complexity as "a measure of the resources expended by a system while interacting with a piece of software to perform a given task. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software [4]."

According to Bill Curtis, there are two types of software complexity, psychological and algorithmic: "Psychological complexity affects the performance of programmers trying to comprehend or modify a class/module whereas algorithmic or computational complexity characterizes the run-time performance of an algorithm [5]."

Kearney et al describe software complexity as "the interaction between a program and a programmer working on some programming task [6]."

The Institute of Electrical and Electronic Engineers (IEEE) definition of software complexity is "the degree to which a

system or component has a design or implementation that is difficult to understand and verify [8]."

Zuse's definition of software complexity is "the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs [9]."

Measuring software complexity increases the ability to predict the effort and code efficiency, lowers the risk of introducing defects into production, preserves the quality of software and extends its lifetime, enables to estimate the right time to rewrite code, and lowers the galloping cost associated with software maintenance.

B. Software Complexity Metrics

Although the complexity associated with a piece of software cannot be eliminated completely, it can be controlled up to a certain level. Being able to accurately quantify the complexity is an essential prerequisite to controlling it. Hence, various complexity metrics have been introduced by considering different attributes of software.

Though there is no proper source of origin, the author considers lines of code (LOC) as the most primitive metric. Subsequently, complexity metrics for procedure-oriented programs emerged as an area of increasing interest: the first metric proposed, in 1976, was McCabe's cyclomatic complexity (CC) [3]. It measured the maximum number of linearly independent paths through a control flow graph. Also, it was used to measure the testability and maintainability of a program. Since CC was unable to differentiate the complexity of single and complex conditions in some conditional statements, in 1977, Myers extended CC (v (G)) to v' (G) by noting that compound 'IF' conditions are much harder than simple 'IF' conditions [17]. He suggested that complexity ought to be calculated as an interval rather than a single value.

Then, in 1977, Gilb introduced two logical complexity metrics [18]: absolute logical complexity (number of binary decisions instructions) and relative logical complexity (the relationship between the number of binary decision instructions and the number of all instructions). Also that year, Halstead proposed his software science metrics based on the operators and operands of a program, to estimate the work load or the time of work of programmers [7]. These complexity metrics implicitly measured the complexity of a given module as a separate entity.

As a result, several structural metrics were introduced by considering the interactions between system modules [19]: McClure's invocation complexity; Belady and Evangelistis' system partitioning measures; Henry, Kafura, and Harriss' complexity metric which based on the data flow between modules [30]; and Henry and Kafuras' information flow metrics [11].

In 1978, Hansen proposed a hybrid complexity measure, which was a combination of McCabe's cyclomatic complexity and a count of operators, similar to Halstead's unique operator count (n1) [13]. In January 1979, M.R. Woodward, M.A. Hennell, and D. Hedley proposed the 'knot measure' for control flow complexity in program text [32]. This was followed by S.H. Kan's structure complexity (based on stability measures) in 1980. That same year, E.I. Oviedo developed a model to calculate the control flow and data flow complexities using one measure [33].

In 1982, S.S. Iyengar, N. Parameswaran, and J. Fuller introduced a measure to compute the logical complexity of programs in terms of the variable dependency of the sequence of computations, inductive effort in writing loops, and complexity of data structures [34]. In 1984, K. C. Tai came up with a new approach to measure the software complexity of a program based on its data flow [20]. In 1990, Selig developed a hybrid complexity metric to incorporate the information-flow metric.

With the introduction of the object-oriented (OO) concept, researchers started proposing complexity metrics for object-oriented programs as well.

In 1989, Moreau and Dominck suggested three metrics for OO graphical information systems [36]. Pfleeger introduced a cost estimation model for OO development in 1990 [37]. One of the significant developments for OO metrics investigation was Chidamber and Kemerers' 1991 metric suite [12]. In 1994, Chidamber and Kemerer revised their six metrics using measurement theory and empirical data [38].

Sheetz, Tegarden and Monarchi also proposed a set of measures which measured the complexity of an OO system at the variable, method, object, and application levels in 1991[39]. In 1992, Lake and Cook proposed metrics to measure the inheritance of C++ programs [40]. In April 1993, Chen and Lui introduced eight metrics to measure the complexity and reusability of object-oriented software [41]. In October 1994, metrics for object-oriented design (MOOD) was proposed by Abreu et al. These MOOD metrics were able to measure OO mechanisms such as inheritance, encapsulation, polymorphism, and message passing.

In 1998, W. Li proposed an alternative suite of OO metrics to overcome some of the limitations that were present in Chidamber and Kemerers' metric suite [42]. Li's metric suite also consisted of six metrics. In January 1999, C. R. Douce, P. L. Layzell, and J. Buckley introduced a set of spatial measures of software complexity for both the procedural and object-oriented codes [22].

In 2002, Bansiya and Davis proposed a quality model for object oriented design (QMOOD). This QMOOD metrics were able to assess quality attributes such as reusability, functionality, effectiveness, understandability, extendibility and flexibility [43].

In 2007, Mishra proposed class complexity (CC) metric which measured the complexity of a class from a method level [35]. Yadav and Khans' 2009 metric calculated the overall complexity of design hierarchy by considering the inherited methods [31]. By taking into account the inheritance level of statements in classes, types of control structures, and nesting level of control structures, Chhillar and Bhasin proposed a weighted composite complexity measure for OO systems in 2011 [1].

With the beginning of the 20th century, complexity metrics based on Cognitive Informatics was considered as a promising

solution for measuring software complexity (given that it studies the internal information processing mechanisms of the human brain and the processes involved in perception and cognition [26]).

In 2003, Shao and Wang proposed cognitive functional size (CFS) based on cognitive weights and number of inputs and outputs [28]. The same year, Klemola and Rilling proposed the kinds of lines of code identifier density (KLCID) complexity metric [10]. It captured the effect of understanding the set of unique kinds of lines of code of a program. In January 2006, cognitive information complexity measure (CICM) was proposed by D. S. Kushwaha and A. K. Misra [29]; also in 2006, Sanjay Misra introduced the cognitive weight complexity measure (CWCM) based on cognitive weights of basic control structures [27]. In 2007, Misra developed the Class Complexity (CC) measure, which computes the complexity of a program by considering the internal structure of methods [35].

Furthermore, in 2008, Auprasert and Limpiyakorn determined a method to compute the cognitive complexity measure using combinatorial rules [26]. In 2009, Gupta and Chhabra proposed two categories of cognitive-spatial complexity measures to measure the cognitive-spatial complexity of a class and a method [23].

III. CB MEASURE

Usha Chhillar and Shuchita Bhasin believed that multiple factors affect the complexity of a program and hence it cannot be measured by considering just a single factor. Thus, based on the concept of weights they suggested the CB measure for OO programs. The CB measure mainly focuses on the following four factors:

Nesting level of control Structures (W_n): With the level of nesting of control structures the degree of understanding the statements which are inside them increases and consequently adds more complexity to the class or program. Thus, to consider the complexity added by nesting level of control structures Chhillar and Bhasin assigned a weight of zero for sequential statements, one for statements which are at the outer most level, two for statements which are at the next inner level of nesting and so on.

Type of control structures in class or program (W_c): Chhillar and Bhasin believed that the complexity added to a class/program by a control structure varies depending on its type. Thus, a weight of zero was assigned to sequential statements, one for conditional control structures such as if - else, if - else if conditions, two for iterative control structures such as for, while and do-while loops, and n for switch-case statements with n cases.

Inheritance level of statements in classes (W_i): The degree of understanding a statement increases with the level of inheritance of classes. Taking this into account Chhillar and Bhasin assigned a weight of zero for executable statements in the bass class, a weight of one for the executable statements which are at the first derived class, 2 for the statements at the next derived class. Likewise the weight allocated for the statements increases by one for each derived class.

Size of a class or program in terms of token count: With the belief that the complexity of a class or program increases along with the size of it, Chhillar and Bhasin considered the size of a class or program to be the final factor of their measure. The size of a particular executable statement was calculated in terms of the operators, operands, methods/functions, and strings in that statement.

By taking these four factors into consideration a complexity measure for an object-oriented program P was propose as follows:

$$C_w(P) = \sum_{j=1}^{n} (S_j) * (W_t)_j$$

Where

$C_w(P)$ = Proposed weighted complexity measure of program P

S_j = Size of j^{th} executable statement in terms of tokens count

n = Total number of executable statements in program P

j = Index variable

$W_t = W_n + W_c + W_i$

A calculation of the CB measure for the Polygon program in Fig. 1 can be found in Table I.

Unlike some metrics, the CB measure will always be a positive integer value, which makes it easy to compute and less prone to calculation errors. It also allows one to easily monitor which statements of the program add more complexity and which add less complexity.

IV. IMPROVEMENTS

Following are some factors that can be considered to improve the accuracy of the CB measure:

Pointers and references allow software developers to improve the efficiency of programs quite significantly through efficient management of memory. However, a program with these features is harder to understand and thus contributes more complexity. But this metric does not consider the complexity added by the pointers and references.

In addition to supporting single inheritance, object-oriented languages such as C++ support multiple inheritance, which is a very powerful technique that allows a class to inherit characteristics and features from more than one parent class. Problems which would otherwise be fairly difficult to be solved can be elegantly solved using multiple inheritance. However, as illustrated in fig. 2, a program with multiple inheritance can be ambiguous and can lead to lot of problems which can noticeably increase the complexity of it. Although the complexity that is added due to single inheritance is considered by CB measure, it does not consider the complexity that is added due to multiple inheritance.

```
1   # include <iostream>
2   using namespace std;
3   class Polygon {
4      protected: int width, height;
5      public: void setValues(int a, int b);
6   };
7      void Polygon::setValues(int a, int b){
8         width=a;
9         height=b; }
10  class Output {
11     public:
12     void outputArea( );
13     void outputPerimeter( );
14     void finalOutput( ); };
15     void Output  :: outputArea( )
16        {cout << "Area is : "; }
17     void Output  :: outputPerimeter( )
18        {cout << "Perimeter is : "; }
19     void Output  :: finalOutput( )
20        {cout << "End of program "; }
21  class Rectangle: public Polygon {
22     public:
23     int area( );
24     int perimeter( );
25  };
26     int Rectangle :: area( ){
27        Output out;
28        out.outputArea( );
29        if(width >0 && height >0)
30           return width * height;
31        else  return 0; }
32     int Rectangle :: perimeter( )
33        { return (width + height) * 2; }
34  class Triangle: public Polygon {
35     int ans;
36     public: int area( );
37   };
47     int Triangle :: area( ){
48        Output out;
49        out.outputArea( );
50        ans = (width * height)/2;
51        if(ans > 0)
52          return ans; }
53  int main ( ){
54     Triangle tri;
55     Rectangle rec;
56     Output out2;
57     Output out3;
58     rec.setValues (4,5);
59     tri.setValues (4,5);
60     cout << rec.area( ) << '\n';
61     cout << tri.area( ) << '\n';
62     out2.outputPerimeter ( );
63     cout << rec.perimeter( )<< '\n';
64     out3.finalOutput( );
65     return 0; }
```

Fig. 1. A sample C++ program - polygon

When calculating the complexity of a program statement, this metric considers the position of that statement in the inheritance hierarchy, its level of nesting, and size of it in terms of token count. However, it does not consider whether the statement accesses the memory statically or dynamically. But, when the memory is accessed dynamically it is harder to understand and thus contributes more complexity than when it is accessed statically.

A program that unnecessarily uses multiple instances of a class is harder to understand and thus should contribute more complexity than a program with just a single instance of a class. But the CB measure does not take this into account. As a result, for the program shown in fig. 1, the unnecessary creation of the 'out3' instance of the 'output' class to invoke the 'finalOutput' method (Line number 64 in Fig, 1) does not result in an increment in program complexity. Rather, its complexity is same as the complexity when the 'outputPerimeter' method is invoked by the first instance (out 2) of the 'output class'. This is clearly illustrated in table I using line number 62 and 64.

A program can consist of normal methods and recursive methods. Although the CB measure considers the complexity added due to the existence of methods it does not differentiate the complexity added from a normal method and recursive method. However, inclusion of a recursive method will increase the understandability of a program. Thus, the inclusion of a recursive method will contribute more complexity than the inclusion of a normal method.

Programs can be of two types: sequential and concurrent. As the names implies, a sequential program executes a single stream of operations and a concurrent program executes multiple streams of operations concurrently. The execution of each stream in a concurrent program happens in a similar manner to that of a sequential program except for the fact that the streams can communicate and interfere with each other. Concurrent programs allow programmers to do things which would not have been possible otherwise. However, the inclusion of multiple threads increases the understandability of programs and thus contributes more complexity. But the CB measure has not considered the complexity added by multiple threads.

For most software developers exceptions have become the go to mechanism for writing fault-tolerant code. They allow them to handle errors from any desired place. Also they facilitate easy structuring of program code by separating the error handling code from the rest of the code making the code more readable, robust and extensible [14]. On the other hand, exceptions break code structure by creating multiple invisible exit points making it hard to gain a thorough understanding about the code [14]. Thus, the inclusion of exceptions will increase the complexity of a program. However, the CB measure has not considered the complexity added by exceptions.

Conditional statements can be divided into two types: simple and compound. Chhillar and Bhasin only allocated a weight of one for all conditional statements (if-else and if-else if) without considering the type of it. But the accuracy of the complexity calculation can be further improved if the weight

allocation differs depending on the type of the conditional statement. In the program that is shown in fig. 1 the weight assigned for both the conditional statements in line number 29 and 51 is one. But those two statements should have two different complexities.

TABLE I. CALCULATION OF CB MEASURE FOR PROGRAM POLYGON

Line No	S_j	W_n	W_i	W_c	W_t	$S_j*(W_t)_j$
7	4	0	1	0	1	4
8	3	0	1	0	1	3
9	3	0	1	0	1	3
15	4	0	1	0	1	4
16	3	0	1	0	1	3
17	4	0	1	0	1	4
18	3	0	1	0	1	3
19	4	0	1	0	1	4
20	3	0	1	0	1	3
26	4	0	2	0	2	8
28	3	0	2	0	2	6
29	8	1	2	1	4	32
30	4	1	2	0	3	12
31	2	1	2	0	3	6
32	4	0	2	0	2	8
33	6	0	2	0	2	12
47	4	0	2	0	2	8
49	3	0	2	0	2	6
50	7	0	2	0	2	14
51	4	1	2	1	4	16
52	2	1	2	0	3	6
53	2	0	0	0	0	0
58	6	0	1	0	1	6
59	6	0	1	0	1	6
60	7	0	2	0	2	14
61	7	0	2	0	2	14
62	3	0	1	0	1	3
63	7	0	2	0	2	14
64	3	0	1	0	1	3
65	2	0	0	0	0	0
$C_w(P)$						**225**

The complexity associated with a statement that uses methods or attributes of its current class should be less than a statement that uses the objects of another class. But this has not been considered in the CB measure. As a result, for the program that is shown in fig. 1 the complexity of the statement in line number 15 is more than the complexity of the statement in line number 62. But in reality complexity of the statement in line number 62 should be more than the statement in line number 15.

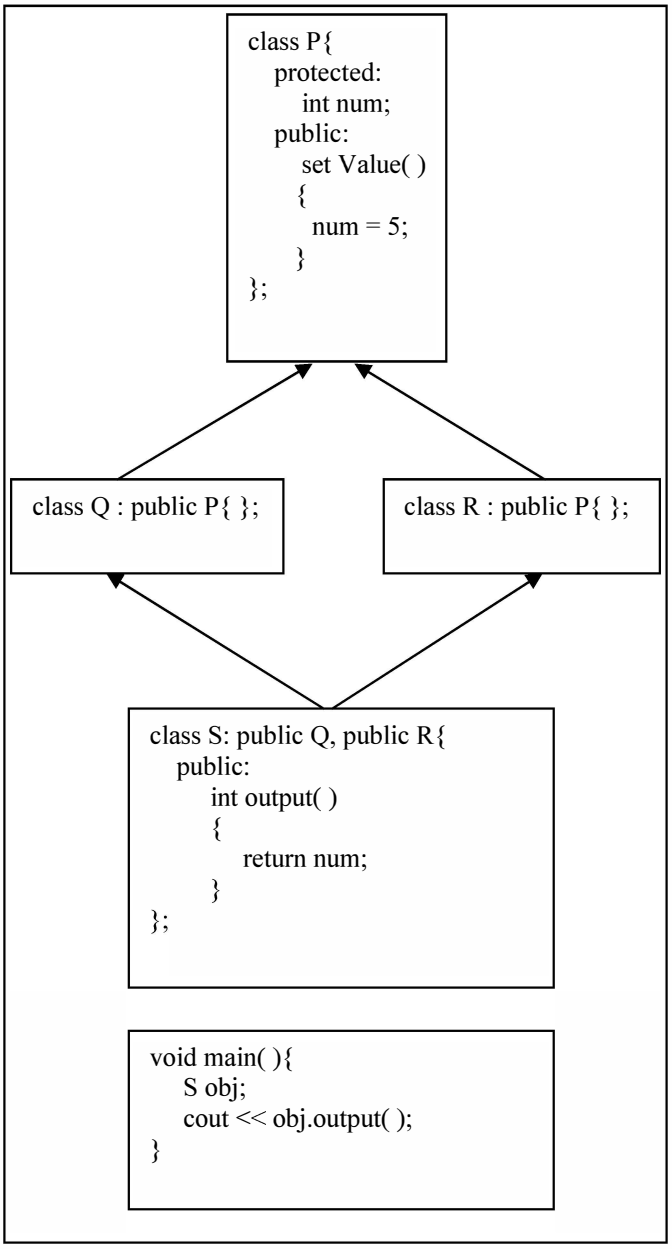

Fig. 2. Diamond problem of multiple inheritance

V. CONCLUSION

Although a number of complexity metrics have been proposed, a common issue with most of them is that they have only looked at the complexity from one aspect. CB measure is one metric that has gone against the norm and has looked at the complexity from different aspects. It considers the complexity that occur due to type of control structures, nesting level of control structures, inheritance level of statements, and size of

the program. In addition to these four factors, this paper draw the readers' attention to some other complexity factors that can have an effect on complexity, to further improve the accuracy of the CB measure.

REFERENCES

[1] U. Chhillar and S. Bhasin, "A new weighted composite complexity for object-oriented systems", International Journal of Information and Communication Technology Research, July 2011, 1(3), pp.101-108.

[2] T. DeMarco, Controlling software projects: management, measurement and estimation, 1st ed., Paperback - Facsimile, June 14, 1986.

[3] T.J. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering, Dec. 1976, SE-2 (4), pp. 308-320.

[4] V. R. Basili, "Qualitative Software Complexity Models: A Summary in Tutorial on Models and Methods for Software Management and Engineering", IEEE Computer Society Press, Los Alamitos,CA, 1980.

[5] B. Curtis, "Measurement and Experimentation in Software Engineering", IEEE Conference, Sep. 1980, 68(9), pp. 1144-1157

[6] J. K. Kearney et al, "Software Complexity Measurement", Communications of the ACM, Nov. 1986, 29 (11), pp.1044-1050.

[7] M.H. Halstead, Elements of Software Science, Elsevier North-Holland, New York, 1977.

[8] IEEE Computer Society: "IEEE Standard Glossary of Software Engineering Terminology." IEEE standard, 1990.

[9] H. Zuse, "Software Complexity Measures and Methods", W.de Gruyter, New York, 1991.

[10] T. Klemola and J. Rilling, A cognitive complexity measure based on category learning, The Second IEEE International Conference on Cognitive Informatics (ICCI), Aug. 2003, pp. 106-112.

[11] S. Henry, and D. Kafura, "Software structure metrics based on information flow," IEEE Transaction on Software Engineering, Sep. 1981, SE-7, pp. 510-518.

[12] S. R. Chidamber and C.F. Kemerer, "Towards a metrics suite for Object Oriented Design", in Proc. OOPSLA '91 Conference proceedings on Object-oriented programming systems, languages, and applications, New York, USA, Nov. 1991, 26, pp.197- 211.

[13] H. F. Li and W.K. Cheung, "An empirical study of software metrics," IEEE Transaction on Software Engineering, June 1987, SE-13, pp. 697-708.

[14] N. Trifunovic. (2009) C++ Exceptions: Pros and Cons. [Online]. Available: http://www.codeproject.com/Articles/38449/C-Exceptions-Pros-and-Cons

[15] (2015) Oxford Advance Learner's Dictionary. [Online]. Available: http://www.oxfordlearnersdictionaries.com/definition/english/complexity

[16] C. L. McClure, "A model for program complexity analysis", in Proc. 3rd International Conference on Software Engineering, Atlanta, Georgia, United States, May 10-12, 1978, pp.149.

[17] E. E. Mills, Software Engineering Institute. "Software Metrics - SEI Curriculum Module SEI-CM-12-1.1," Carnegie Mellon University, Seattle: Washington, 1988.

[18] T. Gilb. Software Metrics. Student literature, Lund, Sweden, 1976.

[19] H. H. Ammar, T. Nikzadeh, and J. Dugan, "Risk assessment of software-system specifications," IEEE transactions on reliability, June 2001, 50, pp.171- 183.

[20] K.C. Tai, "A program complexity metric based on data flow information in control graphs," in Proc. ICSE 84 Proceedings of the 7th international conference on software engineering, NJ, USA, 1984.

[21] S.S. Iyengar, N. Parameswaran, and J. Fuller, "A measure of logical complexity of programs," Computer Languages, 1982, 7, pp 147-160.

[22] C. R. Douce, P. J. Layzell, and J. Buckley, "Spatial measures of software complexity," in Proc. 11th Meeting of Psychology of Programming Interest Group, Leeds. Limerick, Ireland, Jan. 1999, pp 1-8.

[23] V.Gupta and J. K. Chhabra, "Object-oriented cognitive-spatial complexity measures," International Journal of Computer Science and Engineering, 3(6), 2009, pp.122-129.

[24] K. Kapsu, Y. Shin and W.Chisu, "Complexity measures for object-oriented program based on the entropy," in Proc. Second Asia-Pacific Software Engineering Conference (APSEC'95), Brisbane, Australia, December 06 - Dec. 09, pp.127 - 136.

[25] W. Harrison, An Entropy-based Measure of Software Complexity, IEEE Transactions on Software Engineering, 18(11), 1992, pp. 1025-1029.

[26] B. Auprasert, Y. Limpiyakorn, "Underlying cognitive complexity measure computation with combinatorial rules," in Proc. World Academy of Science, Engineering and Technology, Nov. 2008, 35, pp. 432 - 437.

[27] S. Misra, "A Complexity Measure Based on Cognitive Weights," International Journal of Theoretical and Applied Computer Sciences, 2006, 1(1), pp. 1-10.

[28] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," Canadian Journal of Electrical and Computer Engineering, April 2003, 28.

[29] D.S. Kushwaha and A.K. Misra, A modified cognitive information complexity measure of software. ACM SIGSOFT Software Engineering Notes, 31(1), Jan. 2006, pp.1-4.

[30] S. Henry, D. Kafura, and K. Harris, "On the relationships among three software metrics," in Proc. 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, New York, USA, March 1981, pp. 3-10.

[31] A. Yadav, R. A. Khan, "Measuring Design Complexity - An Inherited Method Perspective," SIGSOFT Software Engineering Notes, 34(4), July 2009.

[32] M.R. Woodward, M.A. Hennell, and D.Hedley. "A Measure of Control Flow Complexity in Program Text," IEEE Transactions on Software Engineering, Jan. 1979, SE-5, pp.45-50.

[33] E. I. Oviedo, "Control flow, data flow, and program complexity," in Proc. COMPSAC, 1980, Chicago, 1980, pp.146-152.

[34] S.S. Iyengar, N. Parameswaran, and J. Fuller, "A measure of logical complexity of programs," Computer Languages, 1982, 7, pp 147-160.

[35] S. Misra, "An Object Oriented Complexity Metric Based on Cognitive Weights," 6th IEEE International Conference on Cognitive Informatics (ICCI 07), 2007, pp. 134-139.

[36] D. R. Moreau, W. D. Dominick, "Object-oriented graphical information systems: Research plan and evaluation metrics," J. Syst. and Software, 10, 1989, pp.23-28.

[37] S. L. Pfleeger and J. D. Palmer, "Software estimation for object_oriented systems," in 1990 Int. Function Point Users Group Fall Conf., San Antonia, TX, 1990, pp. 181-196

[38] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, 20(6), June 1994, pp. 476-493.

[39] S. D. Sheetzq, D.P. Tegarden and D.E. Monarchi, "Measuring object-oriented system complexity,Working Paper," Univ. of Colorado, 1992.

[40] A. Lake and C. Cook, "A software complexity metric for C++," Tech. Rep. 92-60-03, Oregon State Univ., 1992.

[41] Y. J. Chen, J. F. Lui, "A new metric for object-oriented design, Infromation of Software Technology," 35(4), April 1993 pp.232-240.

[42] W. Li, "Another metric suite for object-oriented programming," The Journal of System and Software," 44 (2) , December 1998, pp. 155-162.

[43] J. Bansiya, C. G. Davis, "A hierarchical model for object-oriented design quality assessment," IEEE Transactions on Software Engineering, 2002, pp. 4-17.