Code Cognitive Complexity: A New Measure

Jitender Kumar Chhabra

Abstract- **There are different facets of software complexity, some of which have been computed using widely accepted metrics like cyclomatic complexity, data/information flow metrics, but very less attempts have been made to measure the cognitive aspect of the complexity. The human mind's efforts needed for the comprehension of the source code reflect a different dimension of complexity, which is being measured in this paper. There are two aspects of the readability of the source code. One of these is spatial aspect and the other is architectural aspect. This paper is an attempt to measure the cognitive complexity of the source code, by integrating the spatial distances, impact of control statements, and effect of input & output parameters. The proposed metric is evaluated against 5 different programs and also compared with standalone metrics to prove its usefulness.**

Index Terms-: **Code cognitive complexity, code spatial complexity, understandability, psychological complexity, cognitive weights, software metrics.**

1. INTRODUCTION

Since the inception of software engineering, complexity measurement has been always a point of focus for the researchers. Starting from well-addressed control flow based McCabe's cyclomatic complexity [1] and operator/operand based Halstead's science measures [2], researchers have targeted to measure complexity using design aspects, entropy[3], code's comprehension, live members and program weakness [4], [5] etc. and recently some researchers have started exploring the cognitive aspect of complexity.

Concept of spatial complexity was initiated by Douce et al [6], which was based on the theory of working memory and was reported to affect understandability of source code [7]. Spatial ability is a term that is used to refer to an individual's cognitive abilities relating to orientation, the location of objects in space, and the processing of location related visual information. Spatial ability has been correlated with the selection of problem solving strategy, and has played an important role in the formulation of an influential model of working memory. Program comprehension and software maintenance are considered to substantially use programmers' spatial abilities and proper understanding of source code helps in effective debugging and maintenance of the software. This concept of spatial ability was further extended and strengthened by the authors in [8] in form of code and data spatial complexity, and both of these measures were found to be strongly correlated with the perfective maintenance activities.

Dr Jitender Kumar Chhabra is associate professor with Department of Computer Engineering at National Institute of Technology, Kurukshetra-136119 (Institution of National Importance) INDIA (email: jitenderchhabra@gmail.com)

Another measure of cognitive complexity was proposed by Shao & Wang as Code Functional Size (CFS) in terms of cognitive weights [9]. This measure was based on the internal structure of the source code and assigned different weights to Basic Control Structures (BCS) depending on their psychological complexity. This idea was further extended by also incorporating the effect of operators and operands [10]. Both of these proposed metrics were based on architectural aspect of genitive informatics.

Thus, there are two different dimensions of cognitive complexity reported in the literature: spatial complexity and cognitive-weight-based complexity. Each of these two metrics is measuring a different cognitive aspect of the software. Spatial complexity is based on the theory of working memory and cognitive weights are based on the architectural structure of the source code. Spatial complexity treats equally all types of statements whether sequential or iterative or recursive calls, which is not acceptable from cognitive viewpoint [11]. Similarly cognitive weights neither consider at all the individual's spatial abilities of orientation, location and processing of objects in the working memory, and nor differentiate between complex/structured data types from elementary data types. So none of these two metric is alone sufficient to measure the cognitive complexity in totality. Obviously it is desirable to have a new metric of cognitive complexity, which should reflect spatial as well architectural complexity of the source code. This paper proposes a new metric named as Code Cognitive Complexity (CCC) which is an attempt to combine code's spatial complexity with the architectural complexity of control statements as well as data types.

II. CONCEPT OF SPATIAL COMPLEXITY AND COGNITIVE WEIGHTS

The concept of the Code spatial complexity (CSC) was introduced for the first time in the literature by the authors in [8]. This type of cognitive complexity was based on the spatial distance between the call of various modules from their respective definitions. The basis of this measurement was that the working of source code can be understood by comprehending the purpose of every module, which needs to recall its definition during its every use. The greater the distance in lines of code between the definition and use of the module, more is the cognitive effort required to comprehend the connections of those modules in the software [8]. The understanding will be easy for the module-calls which are made immediately after its definition, because the reader's working memory contains the details about the definition, and the call can be easily correlated with its definition. On the other hand, if a module is called after 500 lines of its definition, lot of searching/thinking has to be done, as many modules details appearing in those 500 lines will get their place in the working memory of the human mind, and recalling the details of a module read 500 lines earlier may not be easy. Based on these observations, concept of code-spatial complexity of a module (MCSC) was defined by [8] as average of distances (in terms of lines of code) between the use and definition of the module i.e.

$$MCSC = \frac{\sum_{i=1}^{n} Distance_i}{n} \qquad (1)$$

where n represents count of calls/uses of that module and $Distance_i$ is equal to the absolute difference in number of lines between the module definition and the corresponding call/use.
Total code-spatial complexity of a software was defined as average of code-spatial complexity of all modules, as shown below

$$CSC = \frac{\sum_{i=1}^{m} MCSCi}{m} \qquad (2)$$

where m is count of number of modules in the software.

Another dimension of cognitive complexity is the kind of control structure as well as data [9]-[12]. The architectural aspect of the control statements was reflected in cognitive complexity with help of using weights of various types of Basic Control Structures (BCS) (architectural differences of data were not addressed at all). The modules' calls were used as a multiplying factor for the number of inputs and outputs. But it is well established fact that code plays more important role than data in complexity of procedure-oriented software. Hence the computation needs to be code oriented and data members' impact needs to be integrated it with that. This paper attempts to compute such code cognitive complexity using the spatial complexity of the module integrated with the cognitive weights of data as well as code.

III. CODE COGNITIVE COMPLEXITY

Impact of call of a module on the working logic of program can be understood through the parameters passed to the module, then understanding the processing being done on these passed data (inside the definition of module) and then identifying the value(s) being returned by the module, if any. Hence the cognitive complexity of the source code depends on the type of control statements, various modules & their parameters and return values. So a new measure of cognitive complexity is proposed here for the first time in literature which takes into consideration all of these defined aspects. The cognitive weights are defined now not only for the control statements, but also for the type of parameters. It is obvious that a parameter passed through pointer require more efforts for comprehension than a simple integer variable. Similarly arrays and structures based parameters/return-value are more complex than atomic data. Hence a new refined and expanded table is given below which consists of a more comprehensive list of all members whose cognitive weights need to be considered while measuring the cognitive complexity. The table covers all different constructs used in programming such as iteration, selection, sequence and different types of data such as atomic variables, arrays, structures, points and more complex combinations of these.

TABLE 1: Cognitive Weights of All Members needing Integration with Spatial Distance

Category	BCS	Weight
Sequence	Sequence	1
Branch	if then else	2
	case	3
Iteration	for – do, while, do-while	3
	nested control statements	4
Constant Data	Constant Values	1
	Enumerations & defined constants	1
Variables	atomic & elementary	1
	array (1-d)& structure	2
	multi-dimensional array & pointer based indirection(single)	3
	multiple indirection, pointer to structure, etc.	4

In order to compute the code cognitive complexity, cognitive complexity of every module call needs to be computed. Corresponding to a module-call, the Module Cognitive Complexity MCC is now defined as:

$$MCC = W_c{*}Distance + \sum_{i=1}^{N_{ip}+N_{op}} W_{P_i} \qquad (3)$$

Where W_c represents the cognitive weight of the control statement from which the module call has been made. Distance represents the spatial distance of module call from its definition as defined in equation (1) above in section 2. N_{ip} & N_{op} represent number of input and output parameters respectively of the module. W_{P_i} represents the cognitive weight of parameter Pi. The formula of equation (3) is proposed to integrate the effect of spatial ability as well as cognitive weights. MCC is being computed as addition of two components- first component represents the cognitive complexity due to spatial distance and second component is impact of input and output parameters of the module. The basis of the computation of first component is that the sequential call of a module is most easy to understand, and hence uses the multiplying factor of 1 (cognitive weight of sequence is 1). On the other hand, if a module is being called from inside of nested loops & other control statements, this call is likely to be largely influenced by various conditions and computations done during each iteration. Hence comprehension of such module calls needs to contribute much more towards computation of cognitive complexity as compared to a simple call. This is ensured by use of multiplying factor as 4 (cognitive weight of nested control structures is 4). Calculation of second component of MCC is to reflect the impact of type of the parameter being passed to the module and returned by the module. If the input/output parameter is of primitive data type (int, char, float, etc.) or a constant, then it is relatively easier to be understood than a variable of type pointer to pointer or pointer to structure and accordingly cognitive weights have been defined in Table 1 above.

While computing the distance between the module call and definition, the authors of [8] did not take into consideration the possibility of searching from multiple-files for the module's definition. The distance for a particular call can be easily computed, if definition was present in the same file, where call was made. But for module-calls made from file not containing the definition, distance will depend on many other files which reader needs to search, if he has no idea about where it is present. The distance for such usage can be defined as

Thus, the distance for that particular call of the module can be computed as:

Distance = (Distance of call of the module from top of current file)

+ (Distance of definition of the module from top of file containing definition)

+ (0.1 * (total lines of code of remaining files)/2)

This formula is on basis of our experience that in more than 90 percent of the cases, the definition of the module is found in either the file, where it is being used, or in the next file, which the programmers looks into. But in remaining cases, the programmer has to keep on searching in the other source files, till he/she does not get the definition. For those cases, we have taken the average distance of remaining files and that has been multiplied by the worst-case probability i.e. 0.1 corresponding to remaining 10 percent cases [13].

The value of code cognitive complexity can be now computed by averaging the MCC values for all calls. So CCC is proposed as

$$\text{CCC} = \frac{\sum_{j=1}^{m} MCC(MC_j)}{m} \qquad (4)$$

$$= \frac{\sum_{j=1}^{m} (W_c * \text{Distance}(MC_j) + \sum_{i=1}^{N_{ip}+N_{op}} W_{P_i}(MC_j))}{m}$$

where m is count of all module calls in the software and MC_j represents the jth call.

IV. COMPUTATION OF CCC

In order to demonstrate the usefulness of CCC over CSC and CFS, author has computed the value of CCC on some programs of reasonable bigger size, having a mix of various control statements and different data structures. The initial study clearly indicates supremacy of this new proposed metric over CSC and CCC. The following figure 1 shows the value of CCC for 5 programs having increasing order of complexity as judged by experienced programmers.

The program 1 was judged as easiest and program 5 as most difficult to understand. The figure also shows the values of lines of code (LOC) for these 5 programs. It can be observed from the figure that the CCC values show a consistent increase for more difficult programs, but LOC does not.

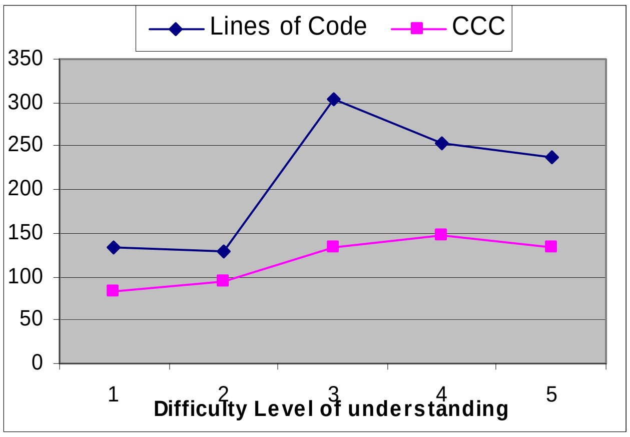

Fig 1: LOC & CCC for 5 programs (programs in order of increasing complexity)

The correlation of CCC with difficulty level of understandability comes out to be 0.87, but correlation of LOC is quite low (0.68). The following bar chart shows a comparison of LOC with CCC, from which it can be concluded, that CCC does not vary with LOC, but it definitely varies according to difficulty level of understandability.

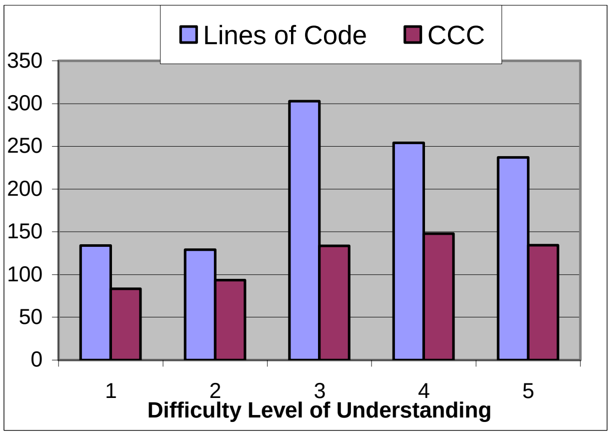

Fig 2: Correlation of CCC with LOC and Understanding Level

V. COMPARISON OF CCC WITH CSC & CFS

The values of CCC metric do not differ significantly from CSC and CFS for trivial programs like factorial, average etc. But that is easily understandable as those programs have very less cognitive complexity, and definition of new proposed metric is supposed to handle higher levels of cognitive complexity especially when cognitive weights and spatial distance values will be higher. Showing those experiments are beyond the scope of this paper, due to page limits. Hence a sample program is written below, which uses some of the constructs affecting the cognitive complexity and this program is used to demonstrate the effectiveness of proposed metric over the existing ones.

```
#include<stdio.h>
float div2int(int a, int b)
{
    /* 5 lines of this modules having one if*/
}
char *insert_dll(struct node **start,float input[][30])
{
        /* 10 sequential lines of having one if */
    }
    void main()
    {
      .../*say 5 lines of initialization*/
       s=div2int(m1,m2);
       for (i=1;i<=m;i++)
       {   ...
           while (x->val<=y->val)
           { ...
              t1=insert_dll(st1,inpmatrix);
           }
       }
    }
```

This program uses two modules namely div2int() and insert_dll(). The first module div2int() is very simple module and has two integers as input and one output parameter. On the other hand insert_dll() module is passed two parameters including of double indirection, and returns one value of pointer type. It is evident without any doubt that comprehension of module insert_dll() is much more than div2int() module. Hence the contribution of insert_dll() should be more towards overall cognitive complexity of the program. Further the calling of insert_dll() is from a nested loop, which further increases the complexity of main(). When we compute the MCC of both of these modules using equation (3), it will be

MCC (div2int)=1*20+3=23

MCC (insert_dll)=4*12+10=58

Based on these two values, the CCC will be as per equation (4)

CCC=(23+58)/2=40.5

On contrary to the proposed metric, if the already existing metrics are computed for the above program, then CSC will reflect only spatial aspect, and CFS will reflect the architectural aspect only. Let us have a look.

MCSC(div2int)=20

MCSC(insert_dll)=12 */*lesser than div2int*/*

CSC=(20+12)/2=16

Similarly CFS is calculated as per [9], [14]. Here the source code has assumed that internal structure of both modules use one if statement. So

CFS(div2int)= (2+1)*2=5

CFS(insert_dll)=(2+1)*2=5 /**same as of div2int**/

CFS(main)=depends on number of i/p and o/ps (but structure & type of input/output does not make any difference)

The above sample code has already highlighted the usefulness of proposed CCC metric. The CCC metric also takes into account the cognitive complexity due to the language. The CFS measure proposed by Shao et al. [9], [14] has been reported to be same for the three different languages (Pascal, C, Java). These results are contradictory to the earlier studies of many researchers where all of them have clearly mentioned that different languages have different levels and language level has been reported to affect cognitive efforts of understanding [2], [15]-[17]. Implementation of one algorithm in different languages can have same algorithmic complexity, but not cognitive complexity. Human comprehension level of the algorithm is definitely dependent on type of the language as well as the source code. Similarly the CSC measure is also not appropriate, as it comes out to be lesser for a complex module. Although CSC discriminates between different languages, but its values computed for div2int() and insert_dll() are not acceptable. But the proposed CCC measure is a better indicator of the cognitive complexity as it differentiates between complex and simple modules, and also takes care of language level.

VI. FUTURE WORKS

The future work of this measure requires a detailed empirical evaluation over large programs to find its suitability. The cognitive weights defined here in this paper are on basis of logical thinking and intuition of developers. However it will be better to compute these weights using a more scientific and statistical methods. There is also a possibility of proposing a suite of metrics for cognitive complexity instead of single measure and then conducting a comparative study among these two approaches.

VII. CONCLUSION

An attempt to measure source code's cognitive complexity has been done in this paper. The proposed CCC metric has been computed in a way that it takes into the consideration of the spatial aspect of modules, architectural differences of control statements and structural differences of data. The theory of working of human memory towards orientation and processing of data has been used to measure the spatial complexity of the modules. The structural complexity of various input/output parameters passed to modules as well as the architectural complexity of control statements originating the module-calls, have been accounted using cognitive weights as proposed in Table 1 above. While defining the metric, possibility of developing software using multiple source code files has also been taken into consideration. The proposed measure has been found to be closely related to the difficulty of understanding of the 5 programs considered by the author. The paper also demonstrates the better performance of CCC metric over CSC and CFS metrics.

REFERENCES

[1] McCabe TJ 1976, A Complexity Measure, *IEEE Transactions on Software Engineering*, vol SE-2, no 4, pp. 308-319.

[2] Halstead MH 1977, *Elements of Software Science*, North Holland, New York.

[3] Harrison W 1992, An Entropy based Measure of Software Complexity, *IEEE Transactions on Software Engineering*, vol 18, no 11, pp. 1025-29.

[4] Conte SD, Dunsmore HE, Shen VY 1986, *Software Engineering Metrics and Models*, Cummings Pub. Coi. Inc. USA.

[5] Aggarwal KK, Singh Y, Chhabra JK 2002, Computing Program Weakness using Module Coupling, *ACM SIGSOFT*, vol 27, no. 1, pp. 63-66.

[6] Douce CR, Layzell PJ, Buckley J 1999, *Spatial Measures of Software Complexity*, Technical Report, Information Technology Research Institute, University of Brighton, UK.

[7] Baddeley A 1997, *Human Memory: Theory and Practice*, Revised Edition, Hove Psychology Press.

[8] Chhabra JK, Aggarwal KK, Singh Y 2003, Code & Data Spatial Complexity: Two Important Software Understandability Measures, *Information and Software Technology*, vol 45, no 8, pp. 539-546.

[9] Shao J, Wang J 2003, A New Measure of Software Complexity based on Cognitive Weights, *Canadian Journal of Electrical & Computer Engineering*, vol 28, no 2, pp. 69-74.

[10] Mishra S 2006a, Modified Cognitive Complexity Measure, *Proceedings of 21st ISCIS'06 Lecture Notes in Computer Science*, 4263, pp.1050-59.

[11] Gold NE, Layzell PJ 2005, Spatial Complexity Metrics: An Investigation of Utility, *IEEE Transactions on Software Engineering*, vol 3, no 1, pp. 203-212.

[12] Mishra S 2006b, A Complexity Measure based on Cognitive Weights, *International Journal of Theoretical and Applied Computer Science*, vol 1, no 1, pp. 1-10.

[13] Chhabra JK, Aggarwal KK, Singh Y 2004, Measurement of Object Oriented Software Spatial Complexity, *Information and Software Technology*, vol 46, no 10, pp. 689-99.

[14] Wang Y, Shao J 2003, Measurement of the Cognitive Functional Complexity of Software, *Proceedings of IEEE International Conference on Cognitive Informatics, ICCI'03*, 2003, pp. 67-74.

[15] Singh Y 1995, Metrics and Design Techniques for Reliable Software, PhD Thesis, Kurukshetra University, Kurukshetra.

[16] Shen VY, Conte SD, Dunsmore HE 1983, Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support, *IEEE Transactions on Software Engineering*, vol SE-9, no 2, pp. 155-165.

[17] Kari Laitnen, "Estimating Understandability of Software Documents", ACM SIGSOFT, vol 21, July 1996, pp 81-92.

Author's Biography

Dr Jitender Kumar Chhabra, Associate Professor, Dept of Computer Engg, National Institute of Technology, Kurukshetra, has been always topper throughput his studies. He did his B Tech Computer Engg from Regional Engg College Kurukshetra & M Tech in Computer Engg from Regional Engg College Kurukshetra (now National Institute of Technology) as GOLD MEDALIST. He did his PhD in Software Metrics from Delhi.

He has published more than 60 papers in various International & National Conferences & Journals including of IEEE, ACM & Elsevier. He worked as Software Engineer in 1993 in Softek, Delhi, but later joined in R.E.C. Kurukshetra in 1994. He is coauthor of Byron S Gottfried for Schaum Series International book from MC Graw Hill on "Programming With C" and his another book for Conceptual & Tricky Problems of Programming for IT-Industry has been published recently from Tata McGraw Hill. He has also reviewed 5 books on Software Engineering & Object Oriented Programming from various reputed International Publishers. He has delivered more than 20 expert Talks and chaired many Technical Sessions in many national & International Conferences of repute including of IEEE in USA. He has visited many countries and presented his research work in USA, Spain, France, Turkey, and Thailand. His area of interest is Software Engineering, Soft Computing & object-Oriented Systems.

He is Reviewer of IEEE, Elsevier, Springer & Wiley Journals. One of his research papers has been published as a Chapter in an International Book. He has been organizing Secretary of two International Conferences and member of steering Committee of many International Conferences. He has worked in collaboration with multi national IT companies Hewelett- Packard (HP) and Tata Consultancy Services (TCS) in the area of Software Engineering. He has been awarded with many prizes & awards like- International Educator of year 2005, 2007, International Professional of the year 2008, International Scientist of 2009, Best Presenter, Best Project Award, National Scholarship, President's Scout etc. He has already guided one PhD candidate and 3 PhD students are presently continuing their research work in the area of software engineering.