Improved Cognitive Information Complexity Measure: A Metric that Establishes Program Comprehension Effort

Dharmender Singh Kushwaha and A.K.Misra
Department of Computer Science and Engineering
Moti Lal Nehru National Institute Of Technology
Allahabad, India.
Email:dharkush@yahoo.com,arun_kmisra@hotmail.com

Abstract
Understanding the software system is known as program comprehension and is a cognitive process. This cognitive process is the driving force behind creation of software that is easier to understand i.e. has lower cognitive complexity, because essentially it is the natural intelligence of human brain that describes the comprehensibility of software. The research area carrying out this study is cognitive informatics. This work has developed an improved cognitive information complexity measure (CICM) that is based on the amount of information contained in the software and encompasses all the major parameters that have a bearing on the difficulty of comprehension or cognitive complexity of software. It is also able to establish the relationship between cognitive complexity of a program and time taken to understand the program, thus mapping closely to the comprehension strategy of a person.

Keywords: Cognitive Information Complexity Measure (CICM), Cognitive Informatics, Information Unit, Information Contained in Software (ICS), Weighted Information Count (WIC.

1. Introduction

Software complexity measures are useful means for monitoring progress, attaining more accurate estimation of milestones and developing software system that contains minimal faults thus improving its quality. Measures are necessary to identify weaknesses of the development process [7]. It also prompt the necessary corrective activities and enable us to monitor the results which enables these measures to act as feedback mechanism, that plays a vital role in the improvement of the software development process. Since complexity measures are a significant and determinant factor of a systems success or failure, there is always a higher risk involved when the complexity measurement is ignored.

Software metrics have been used for over three decades to assess or predict properties of software systems but success has been limited by factors such as lack of sound models, the difficulty of conducting controlled repeatable experiments in educational or commercial context and the inability to compare data obtained by different researchers.

Over the years, research on measuring the software complexity has been carried out to understand what makes computer programs difficult to comprehend. Few measures have shown concern to propose the complexity measures whose computation itself is not complex. A major force behind these efforts is to increase the ability to predict the effort, quality, cost or all of these factors. Major Complexity measures of software’s that refer to cognitive effort have been used in the form of KLCID Complexity Metric [8] and Cognitive functional complexity [11].

IEEE [1] defines software complexity as “the degree to which a system or component has a design or implementation that is difficult to understand and verify. The notion of “difficult to understand” relates to ease of comprehension. Comprehension relates to cognition. The notion of “verify” relates to empirical verification by an objective value of the fact that why programs written in one programming paradigm requires lesser effort to understand i.e. has lower complexity than the same written in other programming paradigm. This finds its roots in cognitive informatics, which is the interdisciplinary study of cognition [10].

Cognitive informatics focuses on understanding the fundamental characteristics of the software. It studies the internal information processing mechanism and natural intelligence of the human brain and its application in computing. Information processing mechanism can only be studied after we have a measure of the information contained in the software. This measure of information in software is used to compute its cognitive complexity.

Cognitive complexity refers to the human effort needed to perform a task or the difficulty experienced in understanding the code or the information packed in it. In cognitive informatics it is established that functional complexity of software is dependent on internal architectural control flows and its inputs and outputs [11]. Cognitive informatics also takes into account the information contained in software [2, 6]. A fundamental research of cognitive informatics is the difficulty encountered in understanding the software. This work begins with cognitive information complexity measure [2, 4, 5] in the next section.

2. Cognitive Information Complexity Measure (CICM)

Various theories have been put forward in establishing a clear relationship between a program and its information content. Once we have a measure of information contained in the software, study of information processing mechanism becomes easier. It has already been established that software obeys the following laws of Informatics and Cognitive Sciences. In establishment / application to these laws, following assertions have been made [9, 13]:

- Software represents computational information.
- Software is a mathematical entity.
- Software is the coded solution to a given program.
- Software is a set of behavioral instructions to computer.

The perception of software as formally described design information and implementation instructions of computing applications leads that the software is a piece of information which is useful to both man and machine and this can be represented as:

Software ≈ Information

This equivalence between software and information further leads to:

Difficulty in understanding the software ≈ Difficulty in understanding the information

As such to ensure that software complexity is affected by software comprehension, cognitive complexity of the software should be based on the measure that takes into account the amount of information contained in the software.

Computational information in any software are constrained in identifiers, may it be variable name, function name, procedure name or any other declared name and the operators performing the various operations to compute the end result. Further, software represents computational information and is a mathematical entity, the amount of information contained in software is represented as a function of identifiers that hold the information and the operators that perform operations on the information, and is represented as:

Information = f (Identifiers, Operators)

Where 'f' is a function of number of identifiers and operators.

Identifiers are variable names, defined constants and other labels in software. Users select identifiers as per programming language rules. But as there is no constraint on the type of identifier names used by the practitioner in any programming language, use of meaningful identifier names add to easier understanding of code as compared with the use of arbitrarily selected cryptic identifier names. In order to assess this impact, the following experiment was carried out.

Experiment

This controlled experiment was performed on two identical groups of 30 students each. None of the student selected had undergone a formal course in programming languages but had done a course titled "Introduction to Computers". All the students were in final year diploma at Government Girls Polytechnic, Allahabad, India. The students participated on a voluntary basis and all subsequent analysis was on anonymous basis.

Each group was given 5 sample programs as enclosed in Appendix I. One set of programs had meaningful identifier names belonging to problem domain and the other used arbitrarily selected identifier name. Every student was asked to comprehend and state, what problem area the program addressed. The time taken by the students to do so was recorded.

This experiment was repeated on three other groups. The result showed that programs with arbitrarily selected identifier names required about 4 times the time to comprehend the programs as compared to programs with meaningful identifier names. Based on result of this controlled experiment, weights are associated with the two different kinds of identifier names as illustrated in table 1.

Sr.No.	Type of Identifier	Associated Weight
01.	Identifier Name belonging to Problem Domain	1
02.	Arbitrarily selected Identifier Name	4

Table 1: Weights associated with the Identifiers

Based on the above representation and experiment, information contained in software is formally defined as follows:

Definition 1: Information contained in one line of code is the sum of number of all operators and operands in that line of code. Thus in k_{th} line of code, information contained is:

$$\mathbf{I_k = (Identifiers + Operators)_k = (ID_k + OP_k)}$$

Where ID_k = Total number of identifiers in the k_{th} LOC of software and is a positive integer,
OP_k = Total number of operators in the k_{th} LOC of software and is a positive integer.

Weights are associated with each identifier depending on the type of naming convention used as depicted in table 1. For meaningful identifier names, the multiplying factor is 1 else it is 4.

Definition 2: Total **Information** contained in software (ICS) is the sum of information contained in all lines of code i.e.

$$\mathbf{ICS = \sum_{k=1}^{LOCS} I_k}$$

Where I_k = Information contained in k_{th} line of code and
LOCS = Total lines of code in the software.

Higher the information contained in the software, greater is the time required to comprehend it hence, it is the information contained in the identifiers and the necessary operations carried out by the operators in achieving the desired goal of the software that makes the software difficult to understand. This answers the first question.

In order to assess the impact of the information contained in a line of code on the number of lines remaining in a program, weighted information count of a line of code is defined next.

Definition 3: The **Weighted Information Count of a line of code (WICL)** of a software is a function of identifiers, operands and LOC and is defined as:

$$\mathbf{WICL_k = I_k / [LOCS - k]}$$

Where I_k = Information contained in a software for the k_{th} line.

Therefore **Weighted Information Count of the Software (WICS)** is defined as:

$$\mathbf{WICS} = \sum_{k=1}^{LOCS} \mathbf{WICL}_k$$

In order to be a complete and robust measure, measure of complexity should also consider the internal control structures of software. These basic control structures of any language are the only way to logically structure the program control flow. These have also been considered as Newton's law in software engineering [90]. These are a set of fundamental and essential flow control mechanisms that are used for building logical architectures of software.

Definition 4: The **sum of the cognitive weights of basic control structures (SBCS)** is defined as under:

Let W_1, W_2 ….. W_n be the cognitive weights of the basic control structures. The definition of basic control structures and their equivalent cognitive weights are as described in [2,11].

$$\text{Then } \mathbf{SBCS} = \sum_{i=1}^{n} (W_i)$$

Note: The weights of basic control structures get multiplied in case of nesting since nesting of basic control structures further increases the comprehension effort. Hence if there are four BCS (BCS_1, BCS_2, BCS_3 & BCS_4) and BCS_3 is nested in BCS_2, then SBCS is calculated as:

$SBCS = BCS_1 + (BCS_2 * BCS_3) + BCS_4$.

Using the above properties, cognitive information complexity measure is proposed in definition 5.

Definition 5: **Cognitive Information Complexity Measure (CICM)** is defined as the product of weighted information count of software (WICS) and sum of the cognitive weights of basic control structures (SBCS) of software. This is because the difficulty of understanding a program gets more and more complex as the number of BCS and their nesting increases.

CICM = WICS * SBCS

This complexity measure encompasses major parameters that have bearing on the difficulty in comprehending software or the cognitive complexity of the software. It introduces a method to measure the amount of information contained in software thus enabling us to calculate coding efficiency (E_I) as in definition 6.

Definition 6: Information Coding Efficiency (E_I) of software [38, 41] is defined as:

(E_I) = ICS / LOCS.

The cognitive information complexity is higher for programs, which have higher information coding efficiency [38, 39]. All the above definitions are illustrated with the help of an example below.

.*Example 1*: An algorithm to calculate the average of a set of numbers as shown below is used to illustrate the application of CICM to measure the complexity.

```
# define N  10
main( )
{
int count;
float sum, average, number;
sum = 0;
count = 0;
while (count < N)
{
   scanf("%f", &number);
   sum = sum + number;
   count = count + 1;
}
average = sum/N;
Printf("N=%d sum = %f"',N,sum);
Printf("average = %f"'average);
}
```

Fig. 1: An algorithm to calculate the average of a set of 'n'numbers

Calculation of KLCID complexity measure

Total no. of identifiers in the above program = 18
Total no. of lines of code = 17
ID = 18/17 = 1.05
No. of unique lines containing identifier = 9
No. of identifiers in the set of unique lines = 11
KLCID = 11 / 9 = 1.22

Calculation of CFS

Number of inputs N_i = 1
Number of outputs N_o = 3
BCS(sequence) W1 = 1
BCS(while) W2 = 3
$W_c = W1 + W2 = 1+3 = 4$
$CFS = (N_i + N_o) * W_c = (1+3)*4 = 16$

Calculation of CICM

LOC = 17
Total no. of identifiers = 18
Total no. of operators = 4
BCS(sequence) W_1 = 1
BCS(while) W_2 = 3
$SBCS = W_1 + W_2 = 1+3 = 4$
WICS = [1/16 + 1/13 + 3/12 + 1/11 + 1/10 + 3/9 + 1/7 + 4/6 + 3/5 + 4/3] = 3.63
(The weight associated with identifier names is 1).

CICM = WICS * SBCS = 3.63 * 4 = 14.53

Information Coding Efficiency (E_I) of the above program = 22/17 = 1.29.

3. Result

This section analyses the result of applying CICM on 15 'C' programs selected from E Balagurusamy's book "Programming in ANSI C" [3].

3.1 Validity of the result

In order to analyze the validity of results, the CICM for different programs is calculated and compared with CFS [83]. Table 3 illustrates both the complexities. It also illustrates the average time required to understand a given program. This time is based on the experiment carried out on a group of 25 students of Diploma in Interior Design. All the students participating in the study had undergone a course titled "Introduction to Computers" and each student was given 15 programs mentioned in table below. They were asked to read the program and come out with what problem the program addressed. Time required to understand was recorded. Lastly the average value was computed in seconds as shown in table 2.

No.	CICM	CFS	Average Comprehension Time (in seconds)	Ref. Of source code
1	0.91	1	176	pp. 4
2	0.99	1	190	pp. 35
3	2.53	3	213	pp. 38
4	14.52	16	251	pp. 42
5	16.68	8	268	pp. 43
6	22.2	18	284	pp. 70
7	28.73	13	292	pp. 96
8	28.49	21	289	pp. 106
9	36.69	36	307	pp. 113
10	39.2	42	317	pp. 122
11	39.98	20	323	pp. 135
12	52.08	70	364	pp. 141
13	53.4	91	390	pp. 220
14	59.8	91	414	pp. 224
15	75.01	91	483	pp. 226

Table 2: Complexity measure of CICM and CFS

Based on the values shown in table 3, a comparative plot for CICM and CFS is shown in figure 2.

Some interesting patterns evolve from comparison of the two cognitive complexity measures. Program 13,14 and 15 solve the same problem using different ways of calling functions. Now CFS complexity measure ranks all the three programs as equally complex since number of inputs, number of outputs and number of functions are same in all three programs, whereas CICM is different for the three programs depending on their complexities. Also, based on values in table 3, as we advance through the programs in the text, we notice increasing cognitive information complexity measure. This augurs well with our experience that when any subject is introduced, contents are very lucid initially and starts getting complex as we advance through the text.

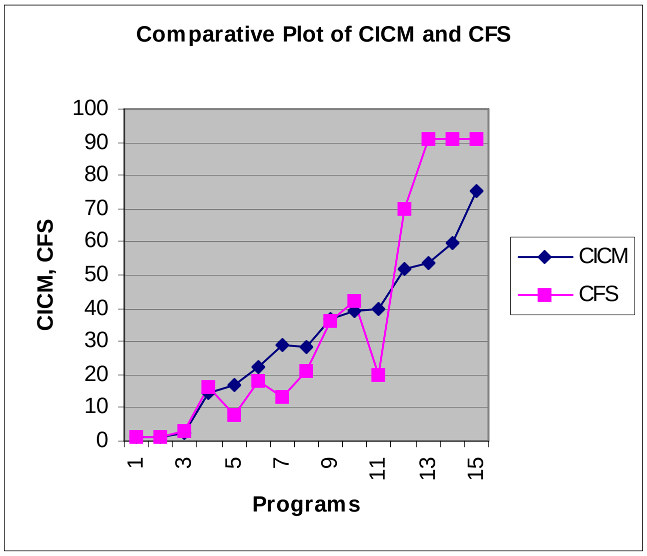

Fig. 2: Comparative Plot of CICM and CFS

Another important result is the linear relationship between the time required to understand a program and its cognitive complexity. As cognitive information complexity increases, time taken to understand the program also increased. Hence it models the comprehension strategy of users. A plot showing this relationship is shown in figure 3.

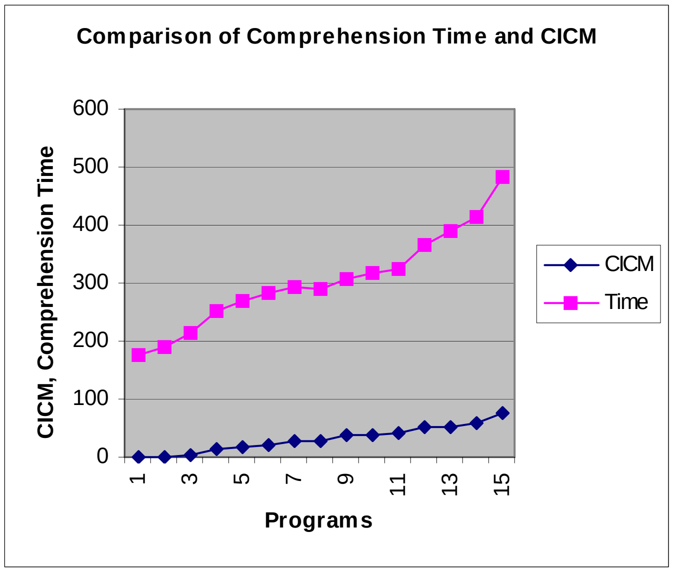

Fig.3: Relationship between Comprehension Time and CICM

4. Conclusion

This work has developed an improved cognitive information complexity measure (CICM) that is based on the amount of information contained in the software and encompasses all the major parameters that have a bearing on the difficulty of comprehension or cognitive complexity of software. This measure is computationally simple and will aid the developers and practitioners in evaluating the software complexity, which serves both as an analyzer and a predicator in quantitative software engineering. CICM satisfies all the nine Weyuker properties [3, 4] . This complexity measure maps very closely to real world situations and establishes that when a person starts reading a programming language text, initially he can flip through easily but as more concepts are introduced, it starts getting complex. Hence it is able to model the behavior of comprehension.

It is also able to establish the relationship between cognitive complexity of a program and time taken to understand the program, thus mapping closely to the comprehension strategy of a person.

References

[1] IEEE Computer Society : IEEE Standard Glossary of Software Engineering Terminology, *IEEE Standard* 610.12 – 1990, IEEE.

[2] Kushwaha D.S.and.Misra A.K "A Modified Cognitive Information Complexity Measure of Software", *ACM SIGSOFT*, Vol. 31, No. 1, January 2006.

[3] Kushwaha D.S.and.Misra A.K "Robustness Analysis of Cognitive Information Complexity Measure using Weyuker Properties", *ACM SIGSOFT*, Vol. 31, No. 1, January 2006.

[4] Kushwaha D.S.and.Misra A.K "Evaluating Cognitive Information Complexity Measure", *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, March 2006.

[5] Kushwaha D.S.and Misra, A.K. "A Complexity Measure Based on Information Contained in the Software", *5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS 2006)*, Madrid, Spain, Feb. 2006.

[6] Kushwaha D.S.and.Misra A.K, "Cognitive Information Complexity Measure: A Metric Based on Information Contained in the Software", *WSEAS Transactions on Computers,* 2006.

[7] Stiglic, B., Hericko, M., and Rozman, I., "How to Evaluate Object-Oriented Software Development", *ACM SIGPLAN Notices*, Vol. 30, No. 5, May 1999.

[8] Tuomas Klemola and Juergen Rilling, A Cognitive Complexity Metric Based on Category Learning, *IEEE International Conference on Cognitive Informatics,* 2003.

[9] Wang, Y., The Real-Time Process Algebra (RTPA), *Annals of Software Engineering: An International Journal*, Vol. 14, USA, 2002, pp. 235 - 274.

[10] Wang,Y., On Cognitive Informatics, Keynote Lecture, *Proceedings of IEEE International Conference on Cognitive Informatics*, 2002, pp. 34 – 42.

[11] Wang, Y., and Shao, J., Measurement Of The Cognitive Functional Complexity of Software, *IEEE International Conference on Cognitive Informatics,* 2003.

[12] Wang,Y., On The Cognitive Informatics Foundations of Software Engineering, *IEEE International Conference on Cognitive Informatics*, 2004.

[13] Wang,Y., On The Informatics Laws of Software, *IEEE International Conference on Cognitive Informatics*, 2004.

Appendix I

Program Set I (Identifiers with meaning Identifier Names)

```
main()
{
int year, period;
float amount, interestrate, value;
print(“input amount, interest rate, and period\n\n”);
scanf(“%f%f%d”, &amount, &interestrate, &period);
print(“\n”);
year = 1;
while(year <= period)
{
  value = amount + interestrate * amount;
  print (“%2d Rs %8.2f\n”, year, value);
  amount = value;
  year – year + 1;
}
}
```

Fig.1: Source code to calculate investments and return

```
# define N 10
main( )
{
int count
float, sum,average,number;
sum = count =0;
while (count < N )
  {
  scanf (“ %f”,& number);
  sum = sum+ number;
  count = count+1;
  }
average = sum / N;
printf (“Average =%f”,average);
}
```

Fig. 2: Source code to calculate the average of a set of N numbers.

```
#define FAHRENHEIT_LOW 0
#define FAHRENHEIT_MAX 255
#define STEP 25

main()
{
typedef float REAL;
REAL fahrenheit, celsius;
fahrenheit= FAHRENHEIT_LOW;
print(“Fahrenheit                Celsius\n\n);
while(fahrenheit<= FAHRENHEIT_MAX)
{
  celsius = (fahrenheit – 32.0) / 1.8;
  printf(“%5.1f       %7.2f\n”, fahrenheit, Celsius);
  fahrenheit = fahrenheit + STEP;
}
}
```

Fig. 3: Source Code for Temperature Conversion

```
main()
{
int months, days;
printf(“Enter days\n”);
scanf(“%d”, &days);
months = days / 30;
days = days % 30;
printf(“months = %d   days = %d”, months, days);
}
```

Fig. 4: Source Code to convert Days to Months and Days

```
#define BASE_SAlARY       5000.00
#define BONUS_RATE         600.00
#define COMMISSION            0.05

main()
{
int quantity_sold;
float gross_salary, price;
float bonus, commission;
printf(“input quantity sold and price\n”);
scanf(“%d   %f”, &quantity, &price);
bonus     = BONUS_RATE * quantity_sold;
commission        = COMMISSION * quantity_sold * price;
gross_salary      = BASE_SALARY + bonus + commission;
printf(“\n”);
printf(“bonus       = %6.2f\n”, bonus);
printf(“commission           = %6.2f\n”, commission);
printf(“gross_salary         = %6.2f\n”, gross_salary);
}
```

Fig. 5: Source Code to Calculate Salary of Salesman

Program Set II **(With arbitrarily selected Identifier Names)**

```
main()
{
int t, x;
float y, r, z;
print(“input y, r, and x\n\n”);
scanf(“%f%f%d”, &y, &r, &x);
print(“\n”);
t = 1;
while(t <= x)
{
 z = y + r * y;
  print (“%2d Rs %8.2f\n”, t, z);
  y = z;
  t – t + 1;
}
}
```

Fig.1: Source code to calculate investments and return

```
# define N 10
main( )
{
int c
float, s,a,n;
s = c =0;
while (c < N )
   {
   scanf (" %f",& n);
   s = s+ n;
   c = c+1;
   }
a = s / N;
printf ("A =%f",a);
}
```

Fig. 2: Source code to calculate the average of a set of N numbers.

```
#define FL 0
#define FM 255
#define STEP 25

main()
{
typedef float REAL;
REAL i, j;
i= FL;
print("I                J\n\n);
while(i<= FM)
{
   j = (i – 32.0) / 1.8;
   printf("%5.1f       %7.2f\n", i, J);
   i = i + STEP;
}
}
```

Fig. 3: Source Code for Temperature Conversion

```
main()
{
int a, b;
printf("Enter b\n");
scanf("%d", &b);
a = b / 30;
b = b % 30;
printf("a = %d   b = %d", a, b);
}
```

Fig. 4: Source Code to convert Days to Months and Days

```
#define B       5000.00
#define C         600.00
#define D             0.05

main()
{
int a;
float x, y;
float z, d;
printf("input quantity sold and y\n");
scanf("%d   %f", &a, &y);
z                    = C * a;
d                    = D * a * y;
x                    = B + z + d;
printf("\n");
printf("z = %6.2f\n", z);
printf("d = %6.2f\n", d);
printf("x = %6.2f\n", x);
}
```

Fig. 5: Source Code to Calculate Salary of Salesman