A Quantitative Cohesion Complexity Measure to Enhancing Software Quality

Pimvard Charoenporn and Peraphon Sophatsathit
Advanced Virtual and Intelligent Computing (AVIC) Center
Department of Mathematics and Computer Science
Faculty of Science, Chulalongkorn University
Bangkok, Thailand
pimvard@gmail.com

***Abstract*—This paper proposes a quantitative approach to measure module cohesion. The relatedness of elements within a module is quantified in the form of cohesion complexity. We first identify variable relatedness using variable dependence graph. Cohesion complexity is then analyzed and mathematically formulated in accordance with standard definitions. Variable relatedness being analyzed are data, selection, and loop. As such, traditional ordinal measure can be objectively clarified to distinguish the differences of design cohesion classification, reflecting the desired software quality. The result so obtained will help developers achieve better cohesive design of software.**

Keywords—cohesion; cohesion complexity; software quality; design cohesion

I. INTRODUCTION

High cohesion provides several desirable characteristics in software quality such as maintainability, flexibility, portability, code readability, reusability, etc. The notion of module cohesion was originally defined by Stevens, et al. [1] that it was the strength of functional relatedness among the processing elements within a module. The processing elements can be defined as many things like statements or output variables. Module cohesion is a measurement in ordinal scale, ranked into seven levels, namely, functional, sequential, communicational, procedural, temporal, logical, and coincidental cohesion, where functional is the highest (good) and coincidental is the lowest (bad). Any module can be defined in one of these seven levels. We can use several methods to measure level of a module. If there are modules classified in the same level, we may not be able to tell the differences between them. On the other hand, if they are in close levels, we may not assure that the higher cohesion is better. For example, if two modules are classified as communicational and procedural cohesion, we may say that the former tends to be better in quality since communicational is higher ranked than procedural. However, there are many factors that affect the quality of software such as number of variables, loops, and selections. Consequently, being classified at a particular level is not good enough to determine the design quality of software.

This paper introduces a quantitative measurement in software quality based on cohesion principle. It provides the same objectives as cohesion with quantifiable measurement to differentiate levels of module relatedness. The results of this proposed measurement will help developers decide whether the designated module should be further decomposed to improve the design.

There are five sections in this paper. The next section describes main related works of the proposed method by Stevens, et al., A. Lakhotia, and J. Nandigam. Section 3 presents the proposed method. The experiment is described in Section 4. Some final conclusions are given in Section 5.

II. RELATED WORK

Stevens, et al., defines module cohesion (*SMC* cohesion) as the strength of functional relatedness among the processing elements within a module [1]. The processing elements can be a statement, a group of statements, a data definition, or a procedure call. There are seven levels of cohesion as shown in Table I. The best or the strongest is functional and the worst or weakest is coincidental cohesion.

TABLE I. ASSOCIATIVE PRINCIPLE BETWEEN TWO PROCESSING ELEMENTS

Cohesion	Associative principles
Coincidental	Little or no meaningful relationship among the processing elements
Logical	Processing elements of a module perform a set of related functions, one of which is selected by the calling module at the time of the invocation
Temporal	Processing elements of a module are executed within the same limited period of time
Procedural	Processing elements share a common procedural unit. The common procedural unit may be a loop or a decision structure.
Communicational	Processing elements reference the same input data and/or produce the same output data
Sequential	Processing elements are sequentially cohesive when the output data or results from one processing element serve as input data for the other processing element.
Functional	Processing elements of a module contribute to the computation of a single specific result

978-1-4799-4963-2/14/$31.00 ©2014 IEEE

To consider if a given module will fit any of the above associative principles, Page-Jones has provided a decision tree that helps determine the cohesion level [5] as shown in Fig. 1.

In *SMC*, the concept of cohesion is emphasized on design-level rather than coding. On the other hand, Lakhotia defines term of processing elements in a more specific way which gives a suitable programming practice. In Lakhotia's work, output variables are considered as processing elements [3]. Output variables in a module are interpreted in a directed graph called Variable Dependence Graph (VDG) which is used to determine the level of cohesion. Nandigam [4] constructed a set of associative rules for each level of cohesion as shown in Table II.

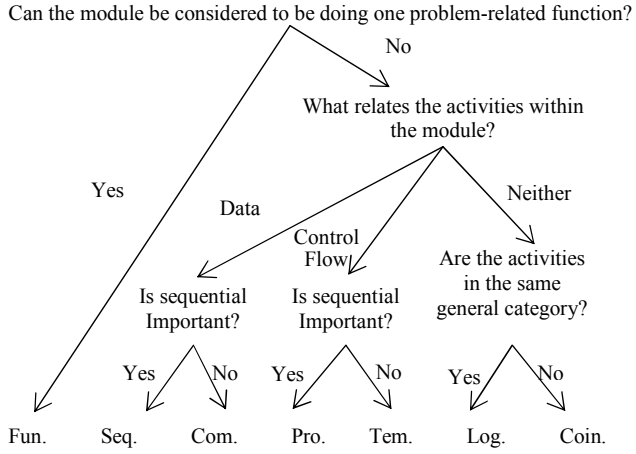

Fig. 1. Decision tree for determining module cohesion.

TABLE II. ASSOCIATIVE RULES BETWEEN TWO PROCESSING ELEMENTS

i	**Cohesion**	**Associative rules** $AR_i: Var \times Var \rightarrow Boolean$
1	Coincidental	$\neg\left(\forall_{i,i\in\{2\ldots5\}} AR_i(x,y)\right)$
2	Logical	$\exists z\left(z \xrightarrow{S(*,*)} x \;\; z \xrightarrow{S(*,*)} y\right)$
3	Procedural	$\exists z,n,k\left(z \xrightarrow{L(n)} xz \xrightarrow{L(n)} y\right)\left(z \xrightarrow{S(n,k)} xz \xrightarrow{S(n,k)} y\right)$
4	Communicational	$\exists z\left(z \xrightarrow{D} x \;\; z \xrightarrow{D} y\right)\left(x \xrightarrow{D} z \;\; y \xrightarrow{D} z\right)$
5	Sequential	$x \rightarrow y \;\; y \rightarrow x$

In this table, x and y represent output variables, z is a common variable, n is the line number of loop or selection statements in the module, and k is the selected branch. For functional cohesion, a module is considered to be functional if there is only one output variable in the module. In this research, temporal cohesion is omitted because static analysis of code cannot handle time-dependent relationships among processing elements. Details on associative rules will be further elaborated in Section III(A). The algorithm for determining the cohesion level is shown in Fig. 2.

Algorithm-1 *Compute-Module-Cohesion*
Input: *VDG* of module M
Output: *Cohesion* of module M
begin
$X \leftarrow$ {output variables in M};
if $|X| = 0$ **then** $Cohesion \leftarrow 'undefined'$
else if $|X| = 1$ **then** $Cohesion \leftarrow 'functional'$
else begin
cohesion_between_pairs ← {};
for all x **and** y in X **and** $x \neq y$ **do begin**
cohesion_between_pairs ← cohesion_between_pairs ∪
max$\{C_i | \; i \in \{1 \ldots 5\}\; AR_i(x,y)\}$;
end for;
if $(\forall_i \; i \in$ cohesion_between_pairs $\; i = coincidental)$
then $Cohesion \leftarrow coincidental$;
else
$Cohesion \leftarrow$ min(cohesion_between_pairs $-$ $\{coincidental\}$);
end;
end;
return *cohesion*
end *Compute-Module-Cohesion*

Fig. 2. Algorithm for determining module cohesion.

In Algorithm-1, a module will be considered as *undefined* cohesion if there is no output variable in the module. If there is only one output, the module will be considered as *functional* cohesion. A module will only be considered as *coincidental* cohesion if all pairs of processing elements are coincidentally combined. For others level of cohesion, we will select the minimum *cohesion_between_pairs* that does not include coincidental cohesion to define the whole module.

Three quantitative measures based on data-slice called Functional Cohesion (FC), namely, Weak Functional Cohesion (WFC), Strong Functional Cohesion (SFC), and Adhesiveness (A) were introduced by Bieman and Ott [9]. These measures give the ratio of glue or superglue tokens to the total number of data tokens in the range of [0, 1].

III. PROPOSED METHOD

In the proposed method, a module will be considered in terms of VDG whose output variables are considered as processing elements. Common variables and output variables are extracted from a module and dependencies are added to form a directed graph. This VDG will be passed along Algorithm-1 to determine the level of cohesion, which in turn will be used to compute cohesion complexity of the module. We define cohesion complexity as the summation of dependency of each variable, some of which are assigned proper weight to indicate their dependencies. This process will be elucidated in the sections that follow.

A. Variable Dependence Graph

According to Nandigam [4], common variables and output variables are represented as nodes, while their dependencies are represented as edges. Dependencies are classified into two types, namely, data dependency and control dependency. Control dependency is further classified into two sub-types,

namely, loop-control and data-control. The dependencies come from data and control flow analysis of the module [6][7]. The following definitions define the dependencies used in this paper.

Let x and y be variables in a module, n_1 and n_2 be statements that define x and y. y has data dependency on x, denoted by $x \xrightarrow{D} y$ if there exists a path in control flow graph from n_1 to n_2. For control dependence, n is a statement with a predicate that uses x on which y is dependent. x and y are associated by selection dependence if n is a selection statement. For example, in an *if* or *case* statement $x \xrightarrow{S(n,k)} y$, k designates the selection condition. If the statement is a loop, n will designate iteration condition such as *for* or *while*, denoted by $x \xrightarrow{L(n)} y$. If data and control dependencies exist between the same two variables, control dependency will be chosen as it dictates the execution flow of the module. Examples of VDG are shown in Fig. 3.

```
1: Procedure Sum1_or_Sum2 (n1, n2, flag : integer;
   arr1, arr2 : int_array; var sum1, sum2 : integer);
2: var i : integer;
3: begin
4:    sum1:= 0;
5:    sum2:= 0;
6:       if flag = 1
7:          for i:= 1 to n1 do
8:             sum1:= sum1 + arr1[i];
9:       else
10:         for i:= 1 to n2 do
11:            sum2:= sum2 + arr2[i];
12: end;
```

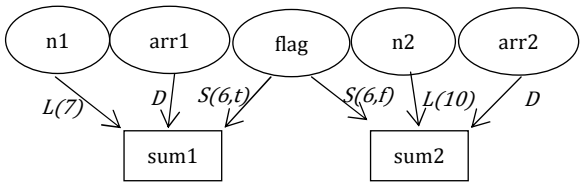

Fig. 3. Procedure and variable dependence graph of module Sum1_or_Sum2.

B. Cohesion Complexity

In computation of cohesion complexity, dependency of each variable will be considered. Complexity of a variable will be assigned the value 1 if the variable depends on nothing. Otherwise, it will be assigned to sum of the number of dependencies involved with the variable. Weights are also added to each type of dependency to balance the complexity. The variable complexity is shown in (1).

$$c = w_d(n) + w_s(n) + w_l(n) \quad (1)$$

where c denotes variable complexity, n denotes the number of dependencies associated with the variables, w_d, w_s, and w_l denote weights for data, selection, and loop dependency, respectively. From our preliminary experiment, w_d holds the minimum value while w_l holds the maximum value. It was found that choosing prime factor to be the weight values yielded better discriminating power than any arbitrary values. Thus, total variable complexity *(tc)* can be determined by (2), where N denotes the number of variables in the module.

$$tc = \sum_i^N c_i \quad (2)$$

Cohesion complexity *(Cc)* is the value of total variable complexity bounded with cohesion level as shown in (3)

$$Cc = \sqrt[a]{tc} \quad (3)$$

where a denotes cohesion level. The algorithm for computing cohesion complexity is shown in Fig. 4.

Algorithm-2 *Compute-Cohesion-Complexity*
Input: *VDG* and *Cohesion* of Module *M*
Output: *Cohesion_complexity* of Module *M*
begin
$CohesionArray \leftarrow \{coincidental, logical, temporal, procedural, communicational, sequential, functional\}$;
$tc = 0$;
for $i \leftarrow 1$ **to** 7 **do begin**
  if $(Cohesion = CohesionArray_i)$ **then**
    $a \leftarrow i$;
    break;
end for;
$N \leftarrow |\vartheta(V_M)|$;
for $j \leftarrow 1$ **to** N **do begin**
  if $(deg^-(\vartheta_i) = 0)$ **then**
    $tc \leftarrow tc + 1$;
  else
    $tc \leftarrow tc + \left(w_d(deg^-(\vartheta_i)) + w_s(deg^-(\vartheta_i)) + w_l(deg^-(\vartheta_i))\right)$;
end for;
$Cohesion\_complexity \leftarrow \sqrt[a]{tc}$;
return $Cohesion\_complexity$;
end;

Fig. 4. Algorithm for determining cohesion complexity.

The cohesion complexity based on sample code in Fig. 3 is described as follows. Module *Sum1_or_Sum2* in Fig. 3 has five common variables and two output variables, the relationship among processing elements matches $\exists z \left(z \xrightarrow{S(*,*)} x \;\; z \xrightarrow{S(*,*)} y\right)$ which is *logical* cohesion. Note that z denotes *flag*, x denotes *sum1* and y denotes *sum2*. The relationships among z, x and z, y are *S(6, t)* and *S(6, f)*, respectively. If a variable associates with a particular type of dependency, the value of w_d, w_s, and w_l will be set to the smallest prime factors 3, 5, and 7 for data, selection, and loop dependencies, respectively. Otherwise, they are set to 0.

There is no such in-degree of nodes *n1, arr1, flag, n2*, and *arr2* in the graph shown in Fig. 3, so variable complexity of each of these variables is 1. There are three in-degrees of *sum1* node and three in-degrees of *sum2* node, so n in (1) for *sum1* and *sum2* is 3. Hence, tc = 1 + 1 + 1 + 1 + 1 + (3(3) + 5(3) + 7(3)) + (3(3) + 5(3) + 7(3)) = 95. Since module *Sum1_or_Sum2* is considered *logical* cohesion, the value of a

in (3) is 2, so cohesion complexity for module *Sum1_or_Sum2* is $\sqrt[2]{95} = 9.7468$

To prove how the proposed cohesion complexity yields different *Cc* values for the same two modules having different cohesion levels, we selected *Sum_and_Prod* procedure [8] and modified it to use different variable sets, hereafter referred to as the original and modified procedures shown in Fig 5. The variables participate in cohesion classification consideration are as follows: *sum, prod,* and *avg* designate output variables or processing elements, and *n, arr, arr1,* and *arr2* designate common variables.

Original procedure	Modified procedure
1. **Procedure** *Sum_and_Prod*	1. **Procedure** *Sum_and_Prod*
$(n: integer;\ arr: int\_array;$	$(n: integer;\ arr1, arr2: int\_array;$
$var\ sum, prod: integer;$	$var\ sum, prod: integer;$
$var\ avg: float)$	$var\ avg: float)$
2. **begin**	2. **begin**
3. $sum := 0;$	3. $sum := 0;$
4. $prod := 1;$	4. $prod := 1;$
5. **for** $i := 1$ **to** n **do begin**	5. **for** $i := 1$ **to** n **do begin**
6. $sum := sum + arr[i];$	6 $prod := prod * arr1[i];$
7 $prod := prod * arr[i];$	7. $sum := sum + arr2[i];$
8. **end**;	8. **end**;
9. $avg := \frac{sum}{n};$	9. $avg := \frac{sum}{n};$
10. **end**;	10. **end**;

Fig. 5. Procedure of module Sum_and_Prod.

TABLE III. DEPENDENCIES OF MODULE SUM_AND_PROD

Dependency D_i	Original procedure	Modified procedure
D_1	$n \xrightarrow{L(5)} sum$	$n \xrightarrow{L(5)} sum$
D_2	$n \xrightarrow{L(5)} prod$	$n \xrightarrow{L(5)} prod$
D_3	$n \xrightarrow{D} avg$	$n \xrightarrow{D} avg$
D_4	$sum \xrightarrow{D} avg$	$sum \xrightarrow{D} avg$
D_5	$arr \xrightarrow{D} sum$	$arr2 \xrightarrow{D} sum$
D_6	$arr \xrightarrow{D} prod$	$arr1 \xrightarrow{D} prod$

Table III lists the dependencies of *Sum_and_Prod* original and modified procedures. In both procedures, they cannot be considered as $functional$ cohesion because the number of processing elements is more than one. Using the association rules in Table II and Algorithm-1, D_1 and D_2 of the original procedure match associative rule 3 $\left(n \xrightarrow{L(5)} sum\ n \xrightarrow{L(5)} prod\right)$, while D_5 and D_6 match associative rule 4 $\left(arr \xrightarrow{D} sum\ arr \xrightarrow{D} prod\right)$. There are two cohesion levels, namely, *procedural* and *communicational* between the same *Sum_and_Prod* procedure, hence *communicational* is selected since it is the higher level. D_4 matches associative rule 5 $\left(sum \xrightarrow{D} avg\right)$. D_3 does not participate in Algorithm-1 and is not considered. The overall assessment of the original module is therefore *communicational* cohesion since it is lower than *sequential* cohesion of D_4. Similarly, D_1 and D_2 of the modified procedure match associative rule 3 $\left(n \xrightarrow{L(5)} sum\ n \xrightarrow{L(5)} prod\right)$, and D_4 matches associative rule 5 $\left(sum \xrightarrow{D} avg\right)$. So the modified procedure is determined as *procedural* cohesion.

TABLE IV. DEPENDENCIES OF MODULE SUM_AND_PROD

Variable complexity (c)	
Original procedure	**Modified procedure**
$c_n = 0$	$c_n = 0$
$c_{arr} = 0$	$c_{arr1} = 0$
$c_{sum} = w_d(n_1)+w_l(n_1)$	$c_{arr2} = 0$
$c_{prod} = w_d(n_2)+w_l(n_2)$	$c_{sum} = w_d(n_1)+w_l(n_1)$
$c_{avg} = w_d(n_3)$	$c_{prod} = w_d(n_2)+w_l(n_2)$
	$c_{avg} = w_d(n_3)$
Total variable complexity (tc)	
$c_{sum} + c_{prod} + c_{avg}$	$c_{sum} + c_{prod} + c_{avg}$

In Table IV, the values of variable complexity (c) in both procedures are the same, so are total variable complexity (tc). Thus, the values of a in the original and modified modules are a_1 and a_2, respectively, where $a_1 > a_2$ (*communicational* > *procedural*). So, $\sqrt[a_1]{tc} < \sqrt[a_2]{tc}$.

C. Module decomposition process

In case the number of members in *cohesion_between_pairs* is more than one which means there is more than one type of cohesion involved, the lowest level will be selected. Higher cohesion is still hidden inside the module. From the above original *Sum_and_Prod* procedure which is classified as *communicational* cohesion, it can be further decomposed to improve for higher cohesion construct. Such an explicit decomposition is illustrated in Fig. 6.

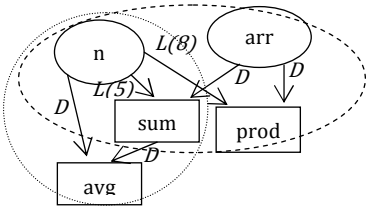

Fig. 6. Variable dependence graph of module Sum_and_Prod.

There are two *cohesion_between_pairs* in the original *Sum_and_Prod* procedure, i.e., *sequential* and *communicational* cohesion as shown earlier. We further decompose this module into two blocks. The first block is composed of *n, arr, sum,* and *avg* as the two output variables form *sequential* cohesion. The other one is composed of *n, arr, sum* and *prod* that form *communicational* cohesion as they refer to the same input *arr*. Cohesion complexity of this module before decomposition is 2.1689 and after decomposition for both blocks are 1.5552 and 2.1118. Thus, the modules are classified to be *sequential* and *communicational* cohesion. Note that the lower the value, the higher the cohesion level. In principle, modules are

decomposed as finer grained as the number of output variables found.

IV. EXPERIMENT

We tested two programs written in C from [9] and [10] and nine modules from [8] and [11]. The first program is a game called "Tic Tac Toe" and the second one is a phone service called "PHONEV2A." The former contains six modules and the latter contains thirteen modules. Table V shows the results of independent module cohesion level. The value of cohesion complexity indicates the degree by which developers can objectively discriminate their design cohesion through the proposed quantitative technique. Table VI and VII depict the results of all test program (whose name appears in column one) cohesion complexity with help of our CCM (Cohesion Complexity Measurement) tool. The second column shows all types of cohesion found in the module. The third column shows the resulting cohesion level of the module under investigation based on Algorithm-1. The fourth column shows the resulting *Cc* value which has been demonstrated using *Sum_and_Prod* in Section III (B). For *Sum_and_Prod* example, there were three types of cohesion found, namely, *coincidental, communicational, and sequential*, the resulting cohesion using Algorithm-1 turned out to be *communicational*, having *Cc* = 2.1689 by (3).

TABLE V. RESULTS OF MODULE COHESION LEVEL AND CORRESPONDING CC VALUE

Name	Cohesion Found	Module Cohesion	Cohesion Complexity
Sum1_and_Sum2	Coincidental	Coincidental	44
Sum1_or_Sum2	Logical	Logical	9.7468
Prod1_and_Prod2	Procedural	Procedural	2.5607
Sum_and_Prod	Coincidental Communicational Sequential	Communicational	2.1689
Fibo_Avg	Sequential	Sequential	1.6035
Sum	Functional	Functional	1.5552
Avg_or_Range	Logical	Logical	12.6491
Avg_and_SD	Communicational	Communicational	2.2974
SD_and_Var	Sequential	Sequential	1.8644

TABLE VI. RESULTS OF TIC TAC TOE MODULE AND CC ASSESSMENT

Name	Cohesion Found	Module Cohesion	Cohesion Complexity
Showframe	Coincidental	Coincidental	11.0000
Showbox	Undefined	Undefined	-
Putintobox	Functional	Functional	1.5112
Gotobox	Undefined	Undefined	-
Navigate	Functional	Functional	1.3459
Checkforwin	Functional	Functional	1.2917

TABLE VII. RESULTS OF PHONEV2A MODULE AND CC ASSESSMENT

Name	Cohesion Found	Module Cohesion	Cohesion Complexity
menu	Functional	Functional	1.000
chkstrdig	Undefined	Undefined	-
DeleteEntry	Coincidental Procedural Sequential	Procedural	4.4238
FindPhone	Procedural	Procedural	3.6109
	Sequential		
FindRoom	Procedural Sequential	Procedural	3.6109
GeTotalEntries	Functional	Functional	1.0000
ListAll	Sequential	Sequential	1.6189
SortAllEntries	Coincidental Procedural Sequential	Procedural	3.4879
AddEntry	coincidental	coincidental	9.0000
drawscreen	undefined	undefined	-
exitmenu	Procedural Sequential	Procedural	3.1137
LoadDB	Coincidental Procedural	Procedural	3.6002
refreshscreen	undefined	undefined	-

From the experiment, *coincidental* cohesion gives the highest result and *functional* cohesion yields the lowest value. This is in concert with standard classification. Notice that the same cohesion level can have different values in cohesion complexity. This is because more complex programming modules have higher values than the simple ones, despite the same cohesion classification. In the program "PHONEV2A", cohesion complexity of *FindPhone* and *FindRoom* module are the same because the code are identical, but variable names are different which result in more variables involved. Fig. 7 shows the variable dependency matrix and the resulting cohesion complexity value of module *FindPhone* computed by CCM tool. However, cohesion complexities of some modules do not exist because we cannot classify the level of module cohesion since they have no output variable, i.e., processing element. All modules in Table V were also tested against the FC measure as shown in Table VIII.

TABLE VIII. RESULTS OF PHONEV2A MODULE AND CC ASSESSMENT

Name	SMC Cohesion	CcMeasure	FC Measure	
Sum1_and_Sum2	Coincidental	44	WFC	0.28
			SFC	0.28
			A	0.28
Sum1_or_Sum2	Logical	9.7468	WFC	0.3846
			SFC	0.3846
			A	0.3846
Prod1_and_Prod2	Procedural	2.5607	WFC	0.2380
			SFC	0.2380
			A	0.2380
Sum_and_Prod	Communicational	2.1689	WFC	0.6957
			SFC	0.2174
			A	0.5362
Fibo_Avg	Sequential	1.6035	WFC	1
			SFC	1
			A	1
Sum	Functional	1.5552	WFC	0
			SFC	1
			A	0
Avg_or_Range	Logical	12.6491	WFC	0.3333
			SFC	0.3333
			A	0.3333
Avg_and_SD	Communicational	2.2974	WFC	0.3214
			SFC	0.3214
			A	0.3214
SD_and_Var	Sequential	1.8644	WFC	1
			SFC	1
			A	1

VDM	p	count	k	flag	phone	found	room
p	-	-	S9true	S9true	S9true	S9true	S9true
count	-	-	L4	L4	L4	L4	L4
k	-	-	-	-	-	-	-
flag	-	-	-	-	-	-	-
phone	-	-	S9true	S9true	-	S9true	S9true
found	-	-	-	-	-	-	-
room	-	-	-	-	-	-	-
Cohesion Between Pairs							
procedural, sequential							
Module Cohesion							
procedural							
Common Variable(s)							
p, count							
Processing Elements							
k, flag, phone, found, room							
Dependency							
0 Data(s), 9 Selection(s), 5 Loop(s)							
Cohesion Complexity:= 3.6109							

Fig. 7. Screen capture of CCM on FindPhone.

V. Discussion And Conclusion

A module should encapsulate some well-defined, coherent piece of functionality so that it is easy to maintain, reuse, and portable. We have followed *SMC* cohesion by adopting association rules, variable dependence graph, and using output variables as processing elements [3] to determine the level of cohesion. Such a quantification help distinguish finer grained of measure for the same level of cohesion in accordance with the *de facto* cohesion standard [2]. Case in point, as Cc method operates at design stage, developers can decide to rectify modular flaw well in advance rather than prolonging the problem till coding stage. Another benefit is that the FC measure could yield the same value for different design characteristics and complexity. For example, in Table VIII, procedure *Fibo_Avg* and *SD_and_Var* have the same result value for both *SMC* cohesion and *FC* measure, but the *Cc* values discern that *SD_and_Var* is more complex than *Fibo_Avg*.

We envision that more comprehensive quantification schemes can be derived with the help of elaborate VDG construct and realized as a programming tool. The benefits of cohesion complexity measure are several folds. First and foremost, quantitative analysis infers more objective design level of software than traditional subjective ordinal analysis. Software developers and maintainers can pinpoint the module in question and make proper redesign, improvement, or corrective adjustment to enhance software quality. Second, performance of software maintenance is efficient and effective since the job can be carried out easier and better understanding. Third, production of software can keep pace with rapid technological innovation. As a case in the third point, various modifications, feature enhancement, and bug fixes of *facebook* [12] that have undergone world-wide test and used over the years could have been performed with fewer effort and more objective design decision. All in all, well design modules having less cohesion complexity ease software development and maintenance effort which in turn will be conducive toward software quality.

References

[1] W. P. Stevens , G. J. Myers and L. L. Constantine "Structured design", *IBM Systems Journal*, vol. 13, no. 2, pp.115 -139 1974.

[2] E. Yourdon and L. Constantine. “Structured Design”. Yourdon Press, 1978

[3] A. Lakhotia "Rule-based approach to computing module cohesion", *Proc. 15th Int. Conf. Software Eng. (ICSE-15)*, pp.35 -44 1993.

[4] J. Nandigam, “A Measure for Module Cohesion”, The University of Southwestern Louisiana, 1995

[5] Page-Jones, M., The Practical Guide to Structured Systems Design, 2nd Edition, Yourdon Press Computing Series, 1988.

[6] Aho, A. V., Sethi, R. and Ullman, J. D., Compilers Principles, Techniques, and Tools, Addison-Wesley, 1986.

[7] Hecht, M. S., Flow Analysis of Computer Programs, North-Holland, Inc.,1977

[8] J.M. Bieman, B-K. Kang, “Measuring Design-Level Cohesion”, *IEEE Transactions on Software Engineering,* 24(2), 1998. pp. 111-123.

[9] J.M. Bieman and L. Ott. “Measuring Functional Cohesion”, *IEEE Transactions on Software Engineering,* 20(8), pp.644–657, August 1994

[10] http://www.codeproject.com/Articles/447332/Game-Programming-in-C-For-Beginners (access on April 1, 2014)

[11] http://www.cprogramming.com/source/phone.zip?action=Jump&LID=44 (access on April 1, 2014)

[12] https://www.dropbox.com/s/y8902tb6or6s8j9/3_procedures.txt (access on April 1, 2014)

[13] https://www.facebook.com/notes/facebook-android-beta/facebook-for-android-beta-app-change-history/190854467764586 (access on April 1, 2014)