QUANTITATIVE COHESION COMPLEXITY MEASURE TO ENHANCING SOFTWARE QUALITY

Miss Pimvard Charoenporn

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science Program in Computer Science and Information Technology
Department of Mathematics and Computer Science
Faculty of Science
Chulalongkorn University
Academic Year 2014

การวัดความซับซ้อนของการทำงานร่วมกันเชิงปริมาณภายในมอดูลเพื่อเพิ่มคุณภาพซอฟต์แวร์

นางสาวพิมวาสน์ เจริญพร

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2557

Thesis Title	QUANTITATIVE COHESION COMPLEXITY MEASURE TO ENHANCING SOFTWARE QUALITY
By	Miss Pimvard Charoenporn
Field of Study	Computer Science and Information Technology
Thesis Advisor	Associate Professor Peraphon Sophatsathit, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

....Dean of the Faculty of Science

(Professor Supot Hannongbua, Dr.rer.nat.)

THESIS COMMITTEE

....Chairman

(Assistant Professor Saranya Maneeroj, Ph.D.)

....Thesis Advisor

(Associate Professor Peraphon Sophatsathit, Ph.D.)

....External Examiner

(Associate Professor Arit Thammano, Ph.D.)

พิมวาสน์ เจริญพร : การวัดความซับซ้อนของการทำงานร่วมกันเชิงปริมาณภายในมอดูลเพื่อเพิ่มคุณภาพซอฟต์แวร์ (QUANTITATIVE COHESION COMPLEXITY MEASURE TO ENHANCING SOFTWARE QUALITY) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: รศ. ดร.พีระพนธ์ โสพัศสถิตย์, 64 หน้า.

วิทยานิพนธ์ฉบับนี้นำเสนอวิธีการวัดการทำงานร่วมกันเชิงปริมาณภายในมอดูล สัมพันธ์ขององค์ประกอบภายในมอดูลจะถูกวัดในรูปแบบของความซับซ้อนของการทำงานร่วมกัน ประการแรกระบุสัมพันธ์ของตัวแปรโดยใช้กราฟพึ่งพาตัวแปร ความซับซ้อนของการทำงานร่วมกันถูกนำมาวิเคราะห์และกำหนดเป็นสูตรคณิตศาสตร์ที่สอดคล้องกับคำนิยามมาตราฐาน ความสัมพันธ์ของตัวแปรที่นำวิเคราะห์ได้แก่ ตัวข้อมูล การเลือก และการวนซ้ำ ทั้งนี้การวัดลำดับแบบดั้งเดิมสามารถชี้แจงวัตถุประสงค์ในการแยกแยะความแตกต่างของการจัดหมวดหมู่การทำงานร่วมกันการออกแบบสะท้อนให้เห็นถึงคุณภาพของซอฟต์แวร์ที่ต้องการ ผลลัพธ์ที่ได้จะช่วยให้นักพัฒนาออกแบบการทำงานร่วมกันภายในมอดูลที่ดีขึ้น

ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	ลายมือชื่อนิสิต	....
		ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
สาขาวิชา	วิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ		
ปีการศึกษา	2557		

5672604923 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
KEYWORDS: COHESION / COHESION COMPLEXITY / SOFTWARE QUALITY / DESIGN COHESION

PIMVARD CHAROENPORN: QUANTITATIVE COHESION COMPLEXITY MEASURE TO ENHANCING SOFTWARE QUALITY. ADVISOR: ASSOC. PROF. PERAPHON SOPHATSATHIT, Ph.D., 64 pp.

This dissertation proposes a quantitative approach to measure module cohesion. The relatedness of elements within a module is quantified in the form of cohesion complexity. Firstly 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

Department:	Mathematics and Computer Science	Student's Signature	....
		Advisor's Signature	....
Field of Study:	Computer Science and Information Technology		
Academic Year:	2014		

ACKNOWLEDGEMENTS

Though the following dissertation is an individual work but I would never have been able to finish without the help, support, guidance and efforts from my advisor, friends, and my family.

Firstly, I would like to thank my advisor, Dr. Peraphon Sophatsathit for his excellent guidance, caring, patience, and providing me with an excellent atmosphere. Without your pearls of wisdom, it would have been nearly impossible to finish this dissertation. It was lucky that I had the chance of being one of your students.

I would like to thank my committee members Dr. Saranya Maneeroj and Dr. Arit Thammano for taking the time and effort to read and examine my dissertation and for providing me with your inventive and enriching comments.

My very special thanks go to my parents whom I owe everything I am today, my father and my mother, Supot and Pornthip Charoenporn. There is no single day that you would never support and encourage me with your best wishes and your love. You both are the most wonderful things that ever happened in my life.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Introduction

High cohesion provides several desirable characteristics in software quality such as maintainability, flexibility, portability, code readability, reusability, etc. Hence, constructing a program concerns a number of aspects such as functional, behavioral, and structural aspects. It is the last aspect encompassing the modular construct that leads to module cohesion and module coupling of processing elements. 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 such as 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) module properties. Any module can be defined in one of these seven levels. Several methods can be used to measure cohesion level of a module. Unfortunately, the sheer cohesion measures will not suffice to yield any discernable characteristics of similar or closely classified modules. Traditional module cohesion measure may not be able to tell the differences between two modules if they are classified in the same level. On the other hand, if they are in close levels, saying that the higher cohesion is better may not be so sure. For example, if two modules are classified as communicational and procedural cohesion, saying that the former tends to be better in quality since it is higher ranked than the latter is not accurate. This issue is the main consideration of this work and will be subsequently elaborated.

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. What decides a distinguishable characteristic of software design quality is module complexity. The issue of complexity involves many program design perspectives, for instance, algorithm, data, model, and various intrinsic/extrinsic attributes, etc. At present, the state-of-the-practice cannot cope with such involving issues, but merely offers a limited framework for software designers to follow. The final decision still remains the human call. Some research efforts are underway to improve such measures and will be recounted in the next chapter.

This research introduces a quantitative measurement in software design quality based on cohesion principle. It provides the same objectives as cohesion with quantifiable measurement to differentiate levels of module relatedness. The proposed method uses dependence relationships of all variables in the module to understand and determine the best classification. The results of this proposed measurement will help developers decide whether the designated module should be further decomposed to improve the module design.

1.2 Problem statements

This research attempts to work out the following questions:

1. How can traditional model cohesion measure be improved to arrive at a quantitative yardstick?
2. How can each level of cohesion classification be objectively distinguished from one another?

1.3 Scope of the research

This research will confine the scope of investigation within the following limits:

1. apply only to C language construct.
2. use small sample module having the size less than 50 LOC.

1.4 Contributions

Some of the benefits precipitate from this work are as follows:

1. quantitative measure in numeric values to permit a discernable distinction for individual module at a specific level of cohesion classification.
2. Distinguishable differences between levels of cohesion classification.

1.5 Document organization

This research is organized as follows. Chapter 2 recounts some of the relevant related prior work. Chapter 3 describes the proposed method, along with algorithm derivations of the supporting theorems. Chapter 4 describes the experiment pertinent to the proposed approach. Some evaluation, benefits, final thoughts, and future work are given in Chapter 5.

CHAPTER 2 LITERATURE REVIEW

Stevens et al., defines module cohesion (SMC cohesion) as the strength of functional relatedness among the processing elements within a module [1][2]. 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 2.1. The best or the strongest is functional and the worst or weakest is coincidental cohesion.

Table 2.1 Associative principles of processing elements based on SMC cohesion

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

The following pseudocode samples are some designed modules that represent level of cohesion measure.

Example2.1: Coincidental cohesion

```
$\textbf{Procedure}: \textbf{Compute\_A\_B\_C}(\textbf{int } \boldsymbol{m, n, o})$
$A \coloneqq m * 2;$
$for\ i = 1\ to\ n$
    $B \coloneqq B + B;$
$if\ \mathrm{o}\%3 = 0$
    $C \coloneqq o/3$
```

Compute_A_B_C procedure is considered as *coincidental* cohesion. Notice that there is no relationship among A or B or C. This procedure is a highly undesirable design of the module having the lowest cohesion level.

Example2.2: Logical cohesion

```
$\textbf{Procedure}: \textbf{Cut\_Paste\_Copy}(\textbf{int } \boldsymbol{flag}, \textbf{String } \boldsymbol{S})$
$if\ \mathrm{flag} = 1$
    $cut(S);$
$else\ if\ \mathrm{flag} = 2$
    $paste(S);$
$else\ if\ \mathrm{flag} = 3$
    $copy(S);$
```

Cut_Paste_Copy procedure is considered as *logical* cohesion. The processing elements in this procedure (cut, paste, copy) are in the same group of operation, in this case is edit text operation. Only one of these operations will be invoked for each operation call, depending on the value of $flag$ variable. This is also an undesirable module cohesion.

Example2.3: Temporal cohesion

```
$\textbf{Procedure}: \textbf{Reset}()$
$int\ A, B, C;$
$A \coloneqq 0;$
$B \coloneqq 1;$
$C \coloneqq 2;$
```

Reset procedure is considered as *temporal* cohesion. This procedure is similar to *Compute_A_B_C* procedure that the elements in the module actually do not have relationships among one another. They are merely put together under one condition that they have to execute at the same time. This module still has low cohesion level.

Example2.4: Procedural cohesion

```
$\textbf{Procedure}: \textbf{Compute\_P\_Q}(\textbf{int}\ \boldsymbol{n})$
 $int\ P, Q;$
 $for\ i = 1\ to\ n$
    $P := P + i;$
   $Q := Q * i;$
$end\ for;$
```

Compute_P_Q procedure is considered as *procedural* cohesion. Processing elements are executed in the same procedural unit, in this case is the *for* loop. *Procedural* cohesion is a moderate cohesion level which yields acceptable design.

Example2.5: Communicational cohesion

```
$\textbf{Procedure}: \textbf{Random\_Sort}(\textbf{int}\ []\ \boldsymbol{arr})$
 $int\ R;$
 $R := random(\mathrm{arr});$
$int[]S\_array;$
$S\_array := sort(arr);$
```

Random_Sort procedure is considered as *communicational* cohesion. A module will be considered as *communicational* cohesion when processing elements of the module use same data or produce the same data. In this case, *arr* is used to compute *R* and *S_array*. *Communicational* cohesion level is also an acceptable design of the module.

Example2.6: Sequential cohesion

```
$\boldsymbol{Procedure}: \boldsymbol{Sort\_Range}(\boldsymbol{int\ n}, \boldsymbol{int}\ []\ \boldsymbol{arr})$
$int[]S\_array;$
$S\_array \coloneqq sort(arr);$
$int\ Range;$
$Range \coloneqq S\_array[n] - S\_array[1];$
```

$Sort\_Range$ procedure is considered as $\boldsymbol{sequential}$ cohesion. One element uses another element to compute itself. Referring to this procedure, Range uses S_array to compute its value. This module cohesion is an acceptable design.

Example2.7: Functional cohesion

```
$\boldsymbol{Procedure}: \boldsymbol{Median}(\boldsymbol{int\ n}, \boldsymbol{int}\ []\ \boldsymbol{S\_array})$
$int\ Median;$
$if\,(n\%2 = 0)$
    $Median \coloneqq S\_array[ceil({}^{n}/_{2})];$
$else$
    $Median \coloneqq \frac{(S\_array[{}^{n}/_{2}] + S\_array[({}^{n}/_{2}) + 1])}{2};$
```

$Median$ procedure is considered as $functinal$ cohesion. This is the ideal module cohesion or the most desirable cohesion level. The module is designed to compute just only one problem.

To decide 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 [3] as shown in Fig. 2.1

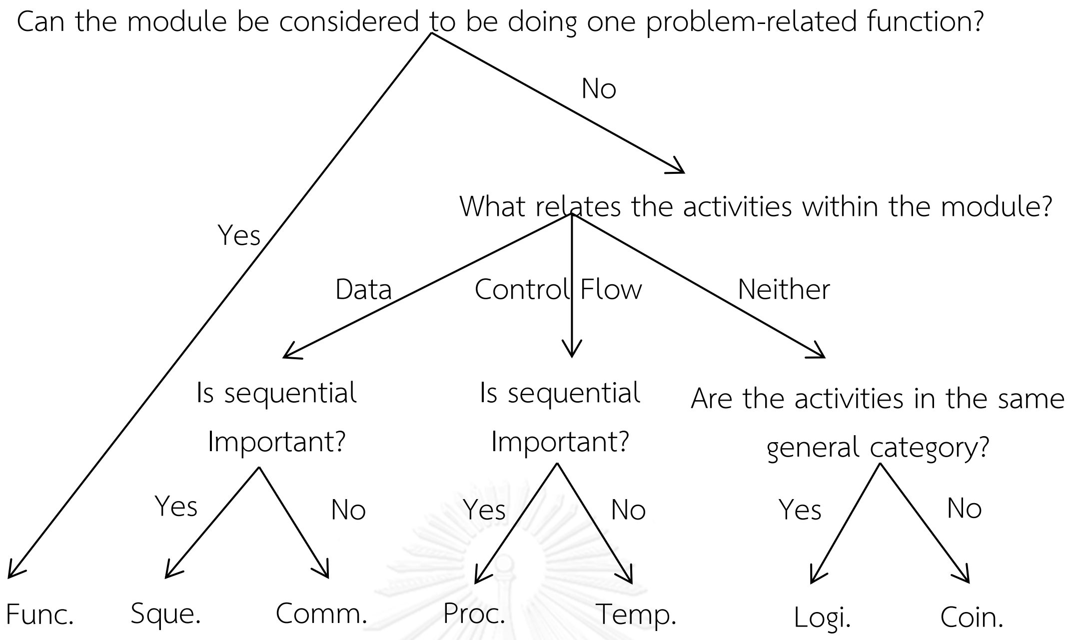


Figure 2.1 Decision tree for determining module cohesion

In SMC, the concept of cohesion is emphasized at design-level rather than coding, while Lakhotia defines terms of processing elements in a more specific way yet suitable for programming practice. In Lakhotia's work [4], output variables of a module are treated as processing elements expressed in a directed graph called Variable Dependence Graph (VDG). The VDG is subsequently used as a basis to determine the level of cohesion.

The example of Figure 2.2 illustrates a designed module and its corresponding VDG. Nodes represent variables and edges represent dependencies. The details of VDG will be further explained in the next chapter.

Example2.8: VDG of Sum1_and_Sum2 procedure

```
$\textbf{Procedure}: \textbf{Sum1\_and\_Sum2}(n1, n2: integer;\ arr1, arr2: int\_array;$
$var\ sum1, sum2: integer);$
$var\ i: integer;$
$begin$
    $sum1 := 0;$
    $sum2 := 0;$
```

```
6.  $for\ i := 1\ to\ n1\ do$
7.      $sum1 := sum1 + arr[i];$
8.  $for\ i := 1\ to\ n2\ do$
9.      $sum2 := sum2 + arr[i];$
10. $end;$
```

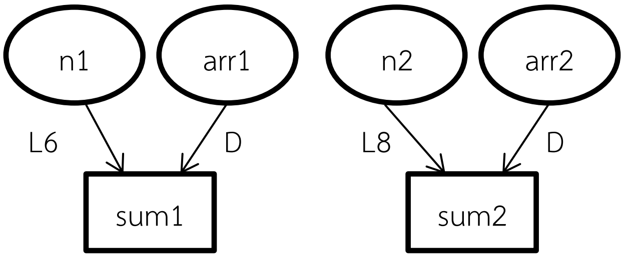


Figure 2.2 VDG of Sum1_and_Sum2 procedure

Nandigam [5] constructed a set of associative rules to describe each level of cohesion as shown in Table 2.2.

Table 2.2 Associative rules between two processing elements

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

In this Table, x and y represent output variables, z is a common variable, n is the line number of loop or a selection statement in the module, and k is a 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 accommodate 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. 2.3.

```
**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{$C_i | \; i \in \{1 \ldots 5\} \wedge AR_i(x, y)$ };
       end for;
       if ($\forall_i \; i \in$ cohesion_between_pairs $\wedge \; i = coincidental$)
         then $Cohesion \leftarrow coincidental$;
       else
         $Cohesion \leftarrow$ min(cohesion_between_pairs − {$coincidental$});
       end;
     end;
   return $cohesion$
end Compute-Module-Cohesion
```

Figure 2.3 Algorithm for determining module cohesion

In this algorithm, 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 levels of cohesion, the minimum **cohesion_between_pairs** within processing elements of the module will be used in the above algorithm, excluding coincidental cohesion.

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 [6] These measures give the ratio of glue or superglue tokens to the total number of data tokens in the range of [0, 1]. The data-slices are obtained from the data tokens like variables, constant definitions, and references. Data tokens that are common to more than one data-slice will be called glue tokens while data tokens that are common to every data-slice are called superglue tokens. WFC can be computed by using the ratio glue tokens to the total data tokens and SFC is the ratio of superglue tokens to total data tokens in the module. The adhesiveness or A is the ratio of the amount of adhesiveness to the total possible adhesiveness.

Example2.9: Computation of FC Measure

$1.\boldsymbol{Procedure}:\boldsymbol{Sum\_and\_Prod}($	sum	prod	avg
$n: integer;$	1	1	1
$arr: int\_array;$	1	1	1
$var\ sum,$	1		1
$prod: integer;$		1	
$var\ avg: float$	1		1
$2. begin$			
$3.\quad sum := 0$	2		2
$4.\quad prod := 1;$		2	
$5.\quad for\ i := 1\ to\ n\ do\ begin$	3	3	3
$6.\qquad sum := sum + arr[i];$	4		4
$7\qquad prod := prod * arr[i];$		4	
$8.\quad end;$			
$9.\quad avg \coloneqq \frac{sum}{n}$	3		3
$10. end;$			

In the above exmaple2.9, glue tokens are highlighted in light and dark grey representing the data tokens that are common to more than one data slice. The glue tokens in the procedure is equal to 16. The superglue tokens have been highlighted in dark grey which is 5. The measures of this module using FC Measure are shown below.

$$WFC = \frac{16}{23} = 0.6957$$

$$SFC = \frac{5}{23} = 0.2174$$

$$A = \frac{(11 * 2) + (5 * 3)}{23 * 3} = 0.5362$$

The measurements on FC measure of the same design could yield different values depending on the implementation by the developers. For example, consider the statements $result = result++$ and $result = result + 1$, data tokens on the first and second statement are 2 and 3, respectively.

CHAPTER 3 PROPOSED METHOD

This chapter will describe the proposed method in detail. In the conventional cohesion classification cannot differentiate the subtleties from the same or close cohesion levels. In some cases, modules having the same cohesion level exhibit different degree of complexity. In particular, the real effort of lowering cohesion for design improvement may be higher than as-is situation since the module size is different. This can be illustrated by the following sample pseudocode modules are classified to be the same level of cohesion which are totally different implementation and complexity.

Example3.1 Procedure: Sum_and_Prod

```
$\boldsymbol{Procedure}$: $\boldsymbol{Sum\_and\_Prod}(n: integer;$
            $arr: int\_array;\ var\ sum, prod: integer;\ var\ avg: float)$
$begin$
  $sum := 0$
  $prod := 1;$
  $for\ i := 1\ to\ n\ do\ begin$
      $sum := sum + arr[i];$
      $prod := prod * arr[i];$
  $end;$
  $avg := \frac{sum}{n}$
$end;$
```

The module *Sum_and_Prod* computes the value of summation, average, and product of the given inputs, which is classified as communicational cohesion based on SMC classification method.

Example3.2 Procedure: Sum_and_Prod

```
$\boldsymbol{Procedure}$: $\boldsymbol{Avg\_and\_Sd}(n, arr);$
$begin$
  $sum := 0;$
  $for\ i := 1\ to\ n\ do\ begin$
      $sum := sum + arr[i];$
```

```
end;
$avg := \frac{sum}{n}$;
$sumDiffSqr = 0$;
for $i := 1$ to $n$ do begin
  $sumDiffSquare = sumDiffSquare + ((arr[i] - avg) * (arr[i] - avg))$;
end;
  $sd = sqr\left(\frac{sumDiffSquare}{n}\right)$;
end;
```

The module Avg_and_Sd computes the average and standard deviation of the given inputs, which is also classified as communicational cohesion. Apparently, they are of different sizes and complexities.

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. Cohesion complexity is defined 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.

3.1 Variable Dependence Graph

According to Lakhotia [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 [7][8]. The following definitions are the original dependency definitions used in this paper.

Definition 1: The control flow graph, or simply a flow graph, of a program is a directed graph where the nodes correspond to the basic blocks of the program and the edges represent potential transfer of control between two basic blocks [7][8]. **Definition 2**: A basic block is a group of statements such that no transfer occurs into a group except to the first statement in that group, and once the first statement is executed, all statements in the group are executed sequentially [8]. **Definition 3**: A definition-use chain of variable x is of the form $< x, n_1, n_2 >$, where statement n_1 defines the variable x and statement n_2 uses the variable x, and there exists a path in the flow graph from n_1 to n_2 which does not contain another definition of x.

Definition 4: A variable y has data dependence on variable x, denoted $x \overset{D}{\rightarrow} y$, if statement n_1 defines x and statement n_2 defines y and there is a definition-use chain with respect to x from n_1 to n_2.

Definition 5: A variable y has control dependence on variable x due to statement n_1, denoted $x \xrightarrow{C(n)} y$, if statement n contains a predicate that uses x and the execution of the statement that defines y is dependent on the value of the predicate in n.

Definition 6: A VDG contains a data dependence edge from node x to node y labeled "D" if $x \overset{D}{\rightarrow} y$

Definition 7: A VDG contains a loop-control dependence edge from node x to node y, labeled "$L(n)$" if $x \xrightarrow{C(n)} y$, and n is a loop statement such as a while or for statement.

Definition 8: A VDG contains a selection-control dependence edge from node x to node y of the form "$S(n,k)$", if $x \xrightarrow{C(n)} y$, and n is an if or case statement and y is defined in the k^{th} branch.

VDG of module M donated as (V_M), $\vartheta(V_M)$ and $\varepsilon(V_M)$ denote vertices and edges of (V_M), respectively. In principle, the vertices and edges are graphical forms of the set of variables in module M, i.e., $\vartheta(V_M)$ set of variable in module M.

$$\varepsilon(V_M) = \left\{ e \,\middle|\, e = \left(x \xrightarrow{D} y \lor x \xrightarrow{L(n)} y \lor x \xrightarrow{S(n,k)} y \right) \land x \neq y \right\}$$

These conventions of representation for each type of cohesion can be exemplified by the following examples to demonstrate module design and their corresponding VDGs construct as follows.

Example3.3: VDG of Sum1_or_Sum2 procedure

```
1. **Procedure**: **Sum1_or_Sum2**($n1, n2, flag$: $integer$; $arr1, arr2$: $int\_array$;
   $var\ sum1, sum2$: $integer$);
2. $var\ I$ : $interger$;
3. $begin$
4.    $sum1 := 0$;
5.    $sum2 := 0$;
6.    $if\ flag = 1$
7.       $for\ i := 1\ to\ n1\ do$
8.          $sum1 := sum1 + arr1$;
9.    $else$
10.      $for\ i := 1\ to\ n2\ do$
11.         $sum2 := sum2 + arr2$;
12. $end$
```

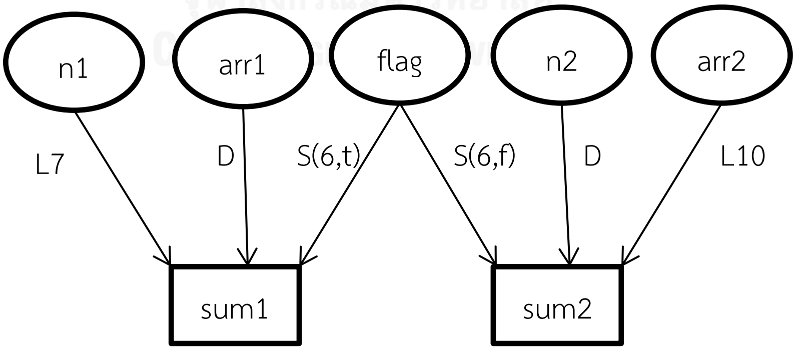


Figure 3.1 VDG of Sum1_or_Sum2 procedure

Example3.4: VDG of Prod1_and_Prod2 procedure

```
1. $\textbf{Procedure}: \textbf{Prod1\_and\_Prod2}(n: integer;\ arr1, arr2: int\_array;$
$var\ prod1, prod2: integer);$
2. $var\ i: integer;$
3. $begin$
4.    $prod1 := 1;$
5.    $prod2 := 1;$
6.    $for\ i := 1\ to\ n\ do\ begin$
7.       $prod1 := prod1 * arr1[i];$
8.       $prod2 := prod2 * arr2[i];$
9.    $end;$
10. $end;$
```

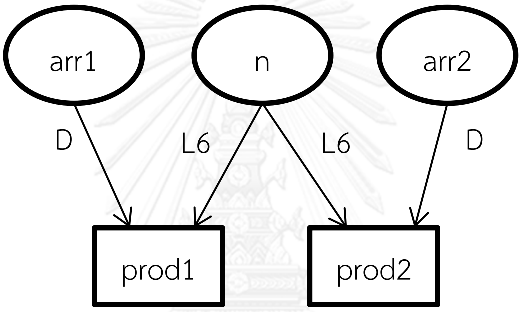


Figure 3.2 VDG of Prod1_and_Prod2 procedure

Example3.5: VDG of Sum_and_Prod procedure

```
1. $\textbf{Procedure}: \textbf{Sum\_and\_Prod}(n: integer;\ arr: int\_array;$
                    $var\ sum, prod: integer;\ var\ avg: float$
2. $begin$
3.    $sum := 0$
4.    $prod := 1;$
5.    $for\ i := 1\ to\ n\ do\ begin$
6.       $sum := sum + arr[i];$
7        $prod := prod * arr[i];$
8.    $end;$
9.    $avg := \frac{sum}{n}$
10. $end;$
```

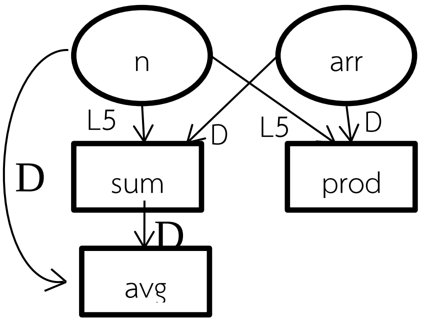


Figure 3.3 VDG of Sum_and_Prod procedure

Example3.6: VDG of Fibo_Avg procedure

```
1. $\boldsymbol{Procedure}: \boldsymbol{Fibo\_Avg}(n: integer;\ var\ fib\_arr: int\_array;\ var\ avg: float);$
2. $var\ sum: integer;$
3. $i: integer;$
4. $begin$
5.    $fib\_arr[1] := 1;$
6.    $fib\_arr[2] := 2;$
7.    $for\ i := 3\ to\ n$
8.       $fib\_arr[i] := fib\_arr[i-1] + fib\_arr[i-2];$
9.    $Sum(n, fib\_arr, sum);$
10.   $avg := sum/n;$
11. $end;$
```

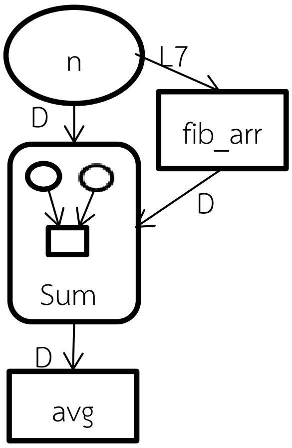


Figure 3.4 VDG of Fibo_Avg procedure

Example3.7: VDG of Sum procedure

```
1. $\boldsymbol{Procedure}$: $\boldsymbol{Sum}(n: integer;\ arr: int\_array;\ var\ sum: integer);$
2. $begin$
3.   $sum := 0;$
4.   $for\ i := 1\ to\ n\ do$
5.       $sum := sum + arr[i];$
6. $end;$
```

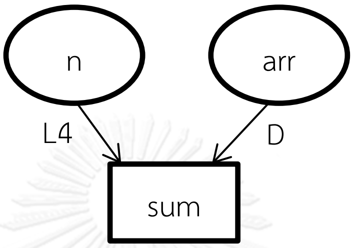


Figure 3.5 VDG of Sum procedure

3.2 Cohesion Complexity

In computations 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 variables. Weights are also added to each type of dependency to balance the complexity. The variable complexity is shown in (3.1).

$$c = w_d(n) + w_s(n) + w_l(n) \tag{3.1}$$

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 the 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 (3.2), where N denotes the number of variables in the module.

$$tc = \sum_i^N c_i \tag{3.2}$$

Cohesion complexity *(Cc)* is the value of total variable complexity bounded with cohesion level as shown in (3.3)

$$Cc = \sqrt[a]{tc} \tag{3.3}$$

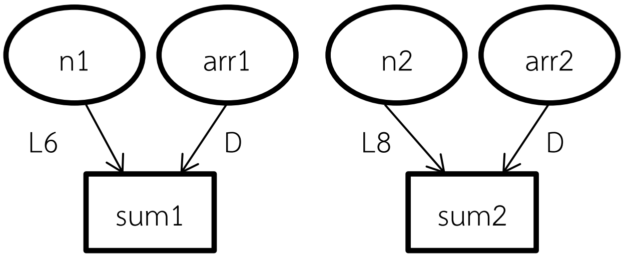
where a denotes the cohesion level. The algorithm for computing cohesion complexity is shown in Fig. 3.6.

```
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
      $tc \leftarrow tc + \left(w_d\left(deg^-(\vartheta_i)\right) + w_s\left(deg^-(\vartheta_i)\right) + w_l\left(deg^-(\vartheta_i)\right)\right);$
end for;
$Cohesion\_complexity \leftarrow \sqrt[a]{tc};$
return $Cohesion\_complexity;$
end;
```

Figure 3.6 Algorithm for determining cohesion complexity

The following examples demonstrate Cc computation measure of each cohesion level.

Example3.8: Cc computation for coincidental cohesion



Module cohesion: Coincidental (a=1)

$c_{n1} = 1$

$c_{arr1} = 1$

$c_{sum1} = w_d(n_{sum1}) + w_s(n_{sum1}) + w_l(n_{sum1})$

$= 7(2) + 0(2) + 3(2)$

$= 20$

$c_{n2} = 1$

$c_{arr2} = 1$

$c_{sum2} = w_d(n_{sum2}) + w_s(n_{sum2}) + w_l(n_{sum2})$

$= 7(2) + 0(2) + 3(2)$

$= 20$

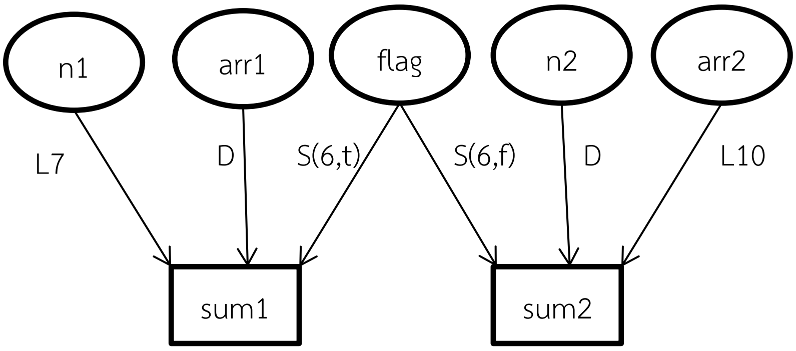
$tc = c_{n1} + c_{arr1} + c_{sum1} + c_{n2} + c_{arr2} + c_{sum2}$

$= 1 + 1 + 20 + 1 + 1 + 20$

$= 44$

$Cc = \sqrt[1]{44} = 44$

Example3.9: Cc computation for logical cohesion



Module cohesion: Logical (a=2)

$c_{n1} = 1$

$c_{arr1} = 1$

$c_{flag} = 1$

$$
\begin{aligned}
c_{sum1} &= w_d(n_{sum1}) + w_s(n_{sum1}) + w_l(n_{sum1}) \\
&= 7(3) + 5(3) + 3(3) \\
&= 45
\end{aligned}
$$

$c_{n2} = 1$

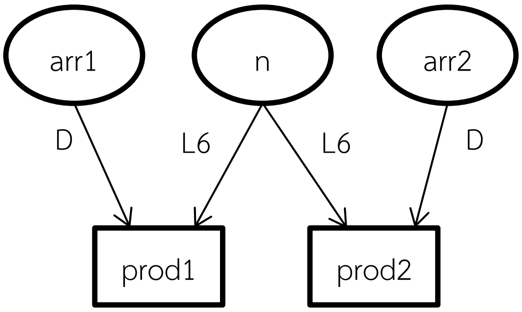
$c_{arr2} = 1$

$$
\begin{aligned}
c_{sum2} &= w_d(n_{sum2}) + w_s(n_{sum2}) + w_l(n_{sum2}) \\
&= 7(3) + 5(3) + 3(3) \\
&= 45
\end{aligned}
$$

$$
\begin{aligned}
tc &= c_{n1} + c_{arr1} + c_{flag} + c_{sum1} + c_{n2} + c_{arr2} + c_{sum2} \\
&= 1 + 1 + 1 + 45 + 1 + 1 + 45 \\
&= 95
\end{aligned}
$$

$Cc = \sqrt[2]{95} = 9.7468$

Example3.10: Cc computation for procedural cohesion



Module cohesion: Procedural (a=4)

$c_{arr1} = 1$

$c_n = 1$

$c_{arr2} = 1$

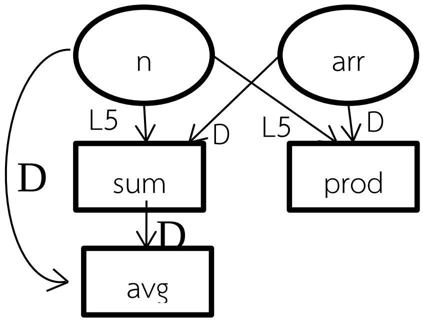
$$c_{prod1} = w_d(n_{prod1}) + w_s(n_{prod1}) + w_l(n_{prod1})$$
$$= 7(2) + 0(2) + 3(2)$$
$$= 20$$

$$c_{prod2} = w_d(n_{prod2}) + w_s(n_{prod2}) + w_l(n_{prod2})$$
$$= 7(2) + 0(2) + 3(2)$$
$$= 20$$

$$tc = c_{arr1} + c_n + c_{arr2} + c_{prod1} + c_{prod2}$$
$$= 1 + 1 + 1 + 20 + 20$$
$$= 43$$

$$Cc = \sqrt[4]{43} = 2.5607$$

Example3.11: Cc computation for communicational cohesion



Module cohesion: Communicational (a=5)

$c_n = 1$

$c_{arr1} = 1$

$c_{sum} = w_d(n_{sum}) + w_s(n_{sum}) + w_l(n_{sum})$

$= 7(2) + 0(5) + 3(2)$

$= 20$

$c_{prod} = w_d\left(n_{prod}\right) + w_s\left(n_{prod}\right) + w_l\left(n_{prod}\right)$

$= 7(2) + 0(5) + 3(2)$

$= 20$

$c_{avg} = w_d\left(n_{avg}\right) + w_s\left(n_{avg}\right) + w_l\left(n_{avg}\right)$

$= 0(2) + 0(2) + 3(2)$

$= 6$

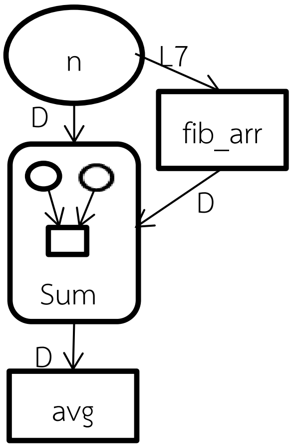
$tc = c_n + c_{arr} + c_{sum} + c_{prod} + c_{avg}$

$= 1 + 1 + 20 + 20 + 6$

$= 48$

$Cc = \sqrt[5]{48} = 2.1689$

Example3.12: Cc computation for sequential cohesion



Module cohesion: Sequential (a=6)

$c_n = 1$

$c_{fib\_arr} = w_d(n_{fib\_arr}) + w_s(n_{fib\_arr}) + w_l(n_{fib\_arr})$

$= 0(1) + 0(1) + 7(1)$

$= 7$

$c_{sum} = w_d(n_{sum}) + w_s(n_{sum}) + w_l(n_{sum})$

$= 3(2) + 0(2) + 0(2)$

$= 6$

$c_{avg} = w_d(n_{avg}) + w_s(n_{avg}) + w_l(n_{avg})$

$= 3(1) + 0(2) + 0(2)$

$= 3$

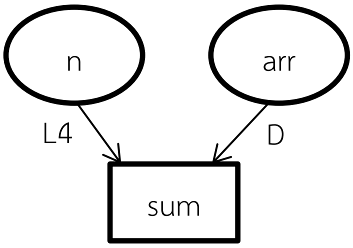
$tc = c_n + c_{fib\_arr} + c_{sum} + c_{avg}$

$= 1 + 7 + 6 + 3$

$= 17$

$Cc = \sqrt[6]{17} = 1.6035$

Example3.13: Cc computation for functional cohesion



Module cohesion: Functional (a=7)

$$c_n = 1$$

$$c_{arr} = 1$$

$$c_{sum} = w_d(n_{sum}) + w_s(n_{sum}) + w_l(n_{sum})$$

$$= 7(2) + 0(2) + 3(2)$$

$$= 20$$

$$tc = c_n + c_{arr} + c_{sum}$$

$$= 1 + 1 + 20$$

$$= 22$$

$$Cc = \sqrt[7]{22} = 1.5552$$

The cohesion complexity based on example 3.9 can be explained as follows. VDG of $Sum1\_or\_Sum2$ contains five common variables and two output variables, the relationship among processing elements matches the associative rule $\exists z\left(z \xrightarrow{S(*,*)} x \wedge z \xrightarrow{S(*,*)} y\right)$ in Table 2.1 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. Since there is no in-degree of nodes $n1$, $arr1$, $flag$, $n2$, and $arr2$ in the graph of example 3.9, each variable complexity of these variables is 1. However, there are three in-degrees of $sum1$ node and three in-degrees of $sum2$ node, so n

in (3.1) for both $sum1$ 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, the cohesion complexity of 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, Sum_and_Prod procedure in example 3.11 is selected and modified to use different variable sets, hereafter referred to as the original and modified procedures as shown in Fig 3.7. 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

```
$\boldsymbol{Procedure}\ Sum\_and\_Prod$
$(n: integer;\ arr: int\_array;$
$var\ sum, prod: integer;$
$var\ avg: float)$
$\boldsymbol{begin}$
    $sum := 0;$
    $prod := 1;$
    $\boldsymbol{for}\ i := 1\ \boldsymbol{to}\ n\ \boldsymbol{do}\ \boldsymbol{begin}$
        $sum := sum + arr[i];$
        $prod := prod * arr[i];$
    $\boldsymbol{end};$
    $avg := \frac{sum}{n};$
$\boldsymbol{end};$
```

Modified procedure

```
$\boldsymbol{Procedure}\ Sum\_and\_Prod$
$(n: integer;\ arr1, arr2: int\_array;$
$var\ sum, prod: integer;$
$var\ avg: float)$
$\boldsymbol{begin}$
    $sum := 0;$
    $prod := 1;$
    $\boldsymbol{for}\ i := 1\ \boldsymbol{to}\ n\ \boldsymbol{do}\ \boldsymbol{begin}$
        $prod := prod * arr1[i];$
        $sum := sum + arr2[i];$
    $\boldsymbol{end};$
    $avg := \frac{sum}{n};$
$\boldsymbol{end};$
```

Figure 3.7 Procedure of module Sum_and_Prod.

Table 3.1 Dependencies of module Sum_and_Prod

Dependency $\boldsymbol{D_i}$	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	$n \overset{D}{\rightarrow} avg$	$n \overset{D}{\rightarrow} avg$
D_4	$sum \overset{D}{\rightarrow} avg$	$sum \overset{D}{\rightarrow} avg$
D_5	$arr \overset{D}{\rightarrow} sum$	$arr2 \overset{D}{\rightarrow} sum$
D_6	$arr \overset{D}{\rightarrow} prod$	$arr1 \overset{D}{\rightarrow} prod$

Table 3.1 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 2.1 and Algorithm-1, D_1 and D_2 of the original procedure match associative rule 3 $\left(n \xrightarrow{L(5)} sum \wedge n \xrightarrow{L(5)} prod\right)$, while D_5 and D_6 match associative rule 4 $\left(arr \overset{D}{\rightarrow} sum \wedge arr \overset{D}{\rightarrow} prod\right)$. There are two qualified cohesion levels, namely, $procedural$ and $communicational$ for Sum_and_Prod procedure. Hence $communicational$ is selected since it is the higher level. D_4 matches associative rule 5 $\left(sum \overset{D}{\rightarrow} avg\right)$. 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 $\left(n \xrightarrow{L(5)} sum \wedge n \xrightarrow{L(5)} prod\right)$, and D_4 matches associative rule 5 $\left(sum \overset{D}{\rightarrow} avg\right)$. So the modified procedure is determined as $procedural$ cohesion.

Table 3.2 Variable and total complexity 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 3.2, 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 $a_1 > a_2$ $(communicational > procedural)$. This yields $\sqrt[a_1]{tc} < \sqrt[a_2]{tc}$.

3.3 Modified Cohesion Complexity

The range of Cc values in the previous section 3.2 is quite high among the low cohesion levels as the illustrating examples are somewhat contrasting. For example, Cc value for the next-to of low cohesion levels like coincidental and logical module cohesion are 44 and 9.7468, respectively, which gives the range of 34.2532. As such, its applicability could be limited. An alternative approach is also proposed to reduce the ranges between levels. The new modified Cc measure is describes below.

If a variable does not depend on any variables, complexity of the variable is 0.1, otherwise the complexity of the variable is the summation of dependence complexity on other variables. That is,

$$c = \sum_{i=1}^{3} dependency\ complexity_i \tag{3.4}$$

and dependency complexity becomes

$$dependency\ complexity_i = adt_i \times td \times dw_i \tag{3.5}$$

where i denotes dependency type $i = 1$ or data dependence, $i = 2$ or selection dependence, and $i = 3$ or loop dependence. adt_i denotes number of associated dependency type i of the variable. td denotes number of total variables on which the variable depends. dw denotes the weight for each type of dependency. Thus, the total variable complexity tc is the summation of all variable complexity plus 1, that is

$$tc = 1 + \sum_{n}^{N} variable\ complexity_n \tag{3.6}$$

where n denotes variables in module M. The Cc can be computed as

$$Cc = tc^{0.a} \tag{3.7}$$

where a denotes cohesion level (functional = 1, sequential = 2, ..., coincidental = 7).

Many trial-and-error runs were tested to determine an appropriate scale for the weight parameters in (3.7). Table 3.3 shows the results of Cc values and 0.1 yielded the best range spreading.

Table 3.3 Computation of Cc value using various scales

Procedure	**0.001**	**0.05**	**0.1**	**0.5**	**1**	**2**
Sum1_and_Sum2	1.0334	2.3552	3.4229	9.5183	15.0269	24.4112
Percentage difference	1.8147	14.8487	13.4998	3.6726	1.6524	8.2380
Sum1_or_Sum2	1.0525	2.7659	3.9571	9.8812	14.7786	22.4002
Percentage difference	3.2304	41.3609	49.3038	63.6208	68.4348	72.2886
Prod1_and_Prod2	1.0185	1.6219	2.0061	3.5947	4.6649	6.2074
Percentage difference	0.2553	8.7058	13.0053	24.4053	29.0617	34.3670
Sum_and_Prod	1.0159	1.4807	1.7452	2.7174	3.3092	4.0741
Percentage difference	1.1714	22.4218	28.6214	40.5535	44.9837	48.6684
Fibo_Avg	1.0040	1.1487	1.2457	1.6154	1.8206	2.0913
Percentage difference	0.1594	5.8066	9.2719	19.9950	24.5249	29.5797
Sum	1.0024	1.0820	1.1302	1.2924	1.3741	1.4727

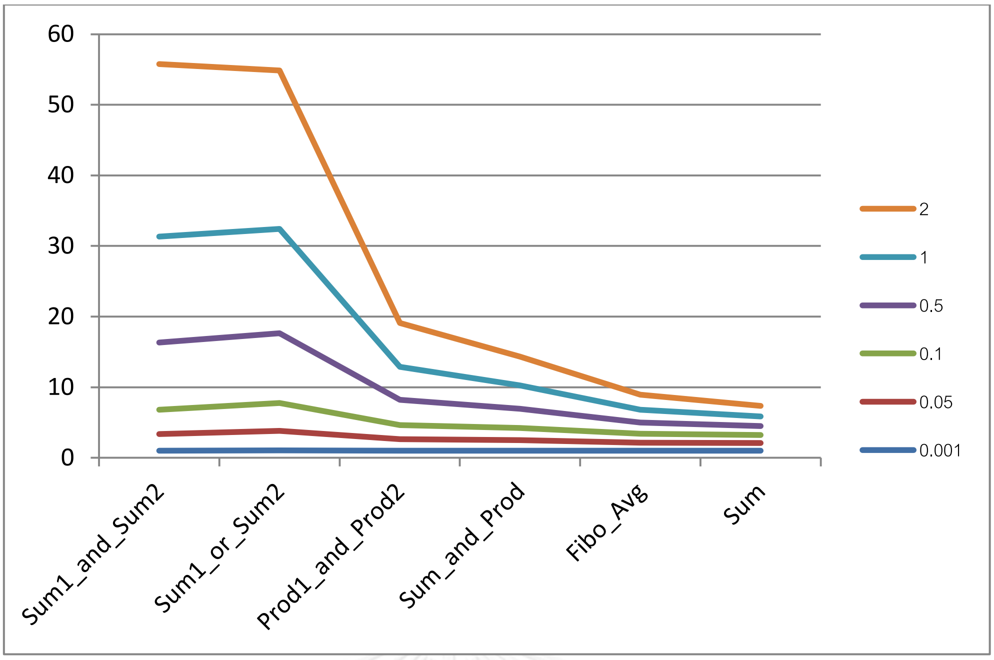


Figure 3.8 Graph of Cc computation using various scales

Fig 3.8. depicts Cc plots of 6 sample modules arranged from coincidental cohesion to functional cohesion. The graph shows that the scales at 2, 1, and 0.5 give very high ranges in low cohesion levels while the scales 0.05 and 0.001 give undiscernable differentiation between the levels. It is apparent that, 0.1 gives the best distinguishable the differences between the levels. The scales at 0.001, 0.05, 0.1, and 0.5 make the results of $logical$ cohesion greater than $coincidental$ cohesion, this is because the examples are very small in term of size and the two modules are very similar to each other. $Sum1\_or\_Sum2$ which determine as $logical$ cohesion has more variable than $Sum1\_and\_Sum2$ which determine as $coincidental$ cohesion and there are more relations among the variables in $Sum1\_or\_Sum2$ module, these are the reasons that cause $coincidental$ results greater complexity than $logical$ cohesion. The total complexity (tc) of the scales at 1 and 2 are higher compares to the rests and when powered by a, Cc of $coincidental$ cohesion (a = 0.7) spreads a lot faster than $logical$ cohesion (a = 0.6) eventhough tc for $logical$ cohesion is higher. However, this case rarely occurs in real world implementation as shown in the next chapter 4 experiment, the results are distinguishable and spread evenly.

Therefore, a standard score for each variable complexity is 0.1. To further elaborate the expressiveness of the measures, additional terms are added to compute variable complexity. adt tells exactly how many instances of dependency involved with each variable, while td is the n value in the previous method. As for weight factor, data dependence still has the smallest value, selection dependence holds intermediate values, and loop dependence has the highest value. The rationale has been described in Section 3.2.

The criteria for determining these weights are as follows. For a plain data dependence where a variable does not depend on any variable, its weight is 0.1. If the variable depends on other variables, the weight becomes 0.2. For selection dependence, there must be at least one condition check. Thus, the weight is set to 0.3. For loop dependence, at least three condition checks are required. There are initialization, termination, and increment-decrement. The weight is equal to $3 \times selection\ dependence$ or 0.9. As tc is bounded by the new values raising to the power of $0.a$, the range of Cc between low cohesion level becomes closer. This is because the previous method tc is bounded ath root or in the other word tc is bounded by the power of $\frac{1}{a}$ which is hard to control the value since $\frac{1}{a}$ is not linear. A constant 1 is added to prevent the result of tc to the power of $0.a$ which could yield the value less than 1.

```
**Algorithm-2** *Compute-Cohesion-Complexity*
**Input:** *VDG* and *Cohesion* of Module $M$
**Output:** *Cohesion_complexity* of Module $M$
**begin**
$CohesionArray \leftarrow \{coincidental, logical, temporal, procedural,$
                        $communicational, sequential, functional\};$
$tc = 0;$
$\textbf{for}\ i \leftarrow 1\ \textbf{to}\ 7\ \textbf{do begin}$
   $\textbf{if}\ (Cohesion = CohesionArray_i)\ \textbf{then}$
      $a \leftarrow i;$
      **break;**
**end for**;
$N \leftarrow |\vartheta(V_M)\,|;$
```

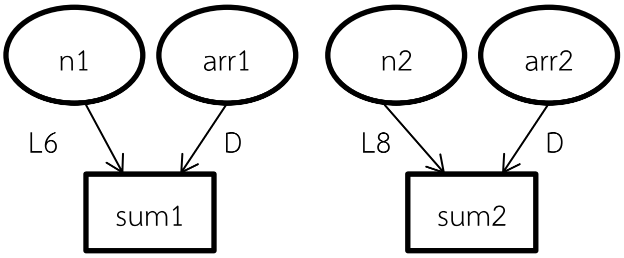
```
$\textbf{for}\ j \leftarrow 1\ \textbf{to}\ N\ \textbf{do begin}$
    $\textbf{if}\ (deg^{-}(\vartheta_i) = 0)\ \textbf{then}$
        $tc \leftarrow tc + 0.1;$
    $\textbf{else}$
        $tc \leftarrow tc + \sum_{i=1}^{3} dependency\ complexity_i\ ;$
$\textbf{end for};$
$Cohesion\_complexity \leftarrow \sqrt[a]{tc};$
$\textbf{return}\ Cohesion\_complexity;$
$\textbf{end};$
```

Figure 3.9 Algorithm for determining cohesion complexity

The following examples demonstrate modified Cc measure of each cohesion level.

Example3.14: Modified Cc computation for coincidental cohesion

Module cohesion: Coincidental ($a = 0.7$)



$c_{n1} = 0.1$

$c_{arr1} = 0.1$

$$c_{sum1} = [data\ complexity\,] + [selection\ complexity] + [loop\ complexity]$$

$$= [(adt_d)\,(td)(w_d)] + [(adt_s)\,(td)(w_s)] + [(adt_l)\,(td)(w_l)]$$

$$= [(1)\,(2)(0.2)] + [(0)\,(0)(0.3)] + [(1)\,(2)(0.9)]$$

$$= 2.2$$

$c_{n2} = 0.1$

$c_{arr2} = 0.1$

$$c_{sum2} = [data\ complexity\,] + [selection\ complexity] + [loop\ complexity]$$

$$= [(adt_d)\,(td)(w_d)] + [(adt_s)\,(td)(w_s)] + [(adt_l)\,(td)(w_l)]$$

$$= [(1)\,(2)(0.2)] + [(0)\,(0)(0.3)] + [(1)\,(2)(0.9)]$$

$$= 2.2$$

$$tc = 1 + (c_{n1} + c_{arr1} + c_{sum1} + c_{n2} + c_{arr2} + c_{sum2})$$

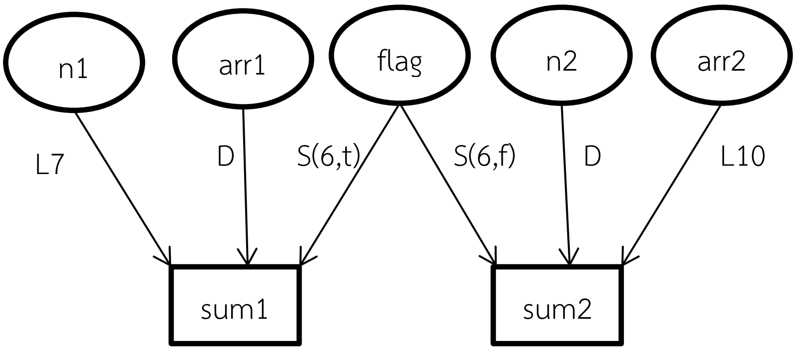
$$= 1 + (0.1 + 0.1 + 2.2 + 0.1 + 0.1 + 2.2)$$

$$= 5.8$$

$$Cc = 5.8^{0.7} = 3.4229$$

Example3.15: Modified Cc computation for logical cohesion

Module cohesion: Logical ($a = 0.6$)



$c_{n1} = 0.1$

$c_{arr1} = 0.1$

$c_{flag} = 0.1$

$$\begin{aligned} c_{sum1} &= [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity] \\ &= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)] \\ &= [(1)\ (3)(0.2)] + [(1)\ (3)(0.3)] + [(1)\ (3)(0.9)] \\ &= 4.2 \end{aligned}$$

$c_{n2} = 0.1$

$c_{arr2} = 0.1$

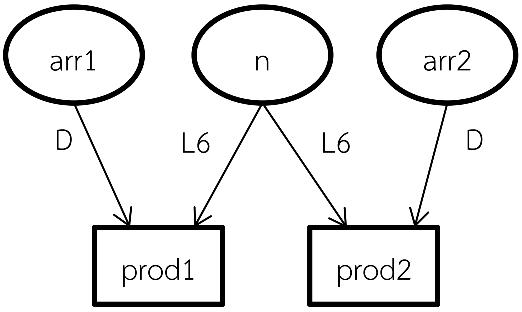
$$\begin{aligned} c_{sum2} &= [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity] \\ &= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)] \\ &= [(1)\ (3)(0.2)] + [(1)\ (3)(0.3)] + [(1)\ (3)(0.9)] \\ &= 4.2 \end{aligned}$$

$$\begin{aligned} tc &= 1 + \left(c_{n1} + c_{arr1} + c_{flag} + c_{sum1} + c_{n2} + c_{arr2} + c_{sum2}\right) \\ &= 1 + (0.1 + 0.1 + 0.1 + 4.2 + 0.1 + 0.1 + 4.2) \\ &= 9.9 \end{aligned}$$

$Cc = 9.9^{0.6} = 3.9571$

Example3.16: Modified Cc computation for procedural cohesion

Module cohesion: Procedural ($a = 0.4$)



$c_{arr1} = 0.1$

$c_n = 0.1$

$c_{arr2} = 0.1$

$$\begin{aligned} c_{prod1} &= [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity] \\ &= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)] \\ &= [(1)\ (2)(0.2)] + [(0)\ (0)(0.3)] + [(1)\ (2)(0.9)] \\ &= 2.2 \end{aligned}$$

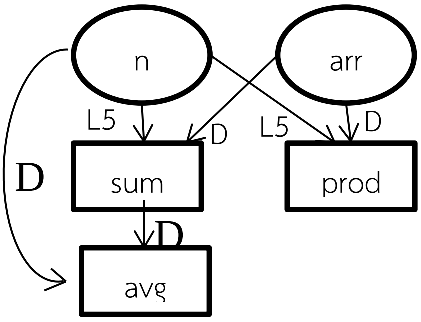
$$\begin{aligned} c_{prod2} &= [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity] \\ &= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)] \\ &= [(1)\ (2)(0.2)] + [(0)\ (0)(0.3)] + [(1)\ (2)(0.9)] \\ &= 2.2 \end{aligned}$$

$$\begin{aligned} tc &= 1 + \left(c_{arr1} + c_n + c_{arr2} + c_{prod1} + c_{prod2}\right) \\ &= 1 + (0.1 + 0.1 + 0.1 + 2.2 + 2.2) \\ &= 5.7 \end{aligned}$$

$Cc = 5.7^{0.4} = 2.0061$

Example3.17: Modified Cc computation for communicational cohesion

Module cohesion: Communicational ($a = 0.3$)



$c_n = 0.1$

$c_{arr1} = 0.1$

$c_{sum} = [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity]$

$= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)]$

$= [(1)\ (2)(0.2)] + [(0)\ (0)(0.3)] + [(1)\ (2)(0.9)]$

$= 2.2$

$c_{prod} = [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity]$

$= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)]$

$= [(1)\ (2)(0.2)] + [(0)\ (0)(0.3)] + [(1)\ (2)(0.9)]$

$= 2.2$

$c_{avg} = [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity]$

$= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)]$

$= [(2)\ (2)(0.2)] + [(0)\ (0)(0.3)] + [(0)\ (0)(0.9)]$

$= 0.8$

$tc = 1 + \left(c_n + c_{arr} + c_{sum} + c_{prod} + c_{avg}\right)$

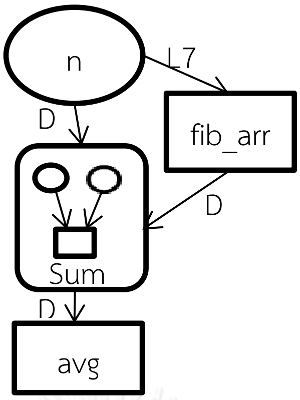
$= 1 + (0.1 + 0.1 + 2.2 + 2.2 + 0.8)$

$= 6.4$

$Cc = 6.4^{0.3} = 1.7452$

Example3.18: Modified Cc computation for sequential cohesion

Module cohesion: Sequential ($a = 0.2$)



$c_n = 0.1$

$c_{fib\_arr} = [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity]$

$= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)]$

$= [(0)\ (0)(0.2)] + [(0)\ (0)(0.3)] + [(1)\ (1)(0.9)]$

$= 0.9$

$c_{sum} = [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity]$

$= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)]$

$= [(2)\ (2)(0.2)] + [(0)\ (0)(0.3)] + [(0)\ (0)(0.9)]$

$= 0.8$

$c_{avg} = [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity]$

$= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)]$

$= [(1)\ (1)(0.2)] + [(0)\ (0)(0.3)] + [(0)\ (0)(0.9)]$

$= 0.2$

$tc = 1 + (c_n + c_{arr} + c_{sum} + c_{prod} + c_{avg})$

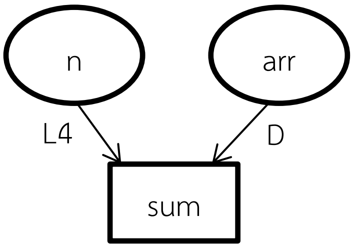
$= 1 + (0.1 + 0.9 + 0.8 + 0.2)$

$= 3$

$Cc = 3^{0.2} = 1.2457$

Example3.19: Modified Cc computation for functional cohesion

Module cohesion: Functional ($a = 0.1$)



$c_n = 0.1$

$c_{arr} = 0.1$

$$\begin{aligned} c_{sum} &= [data\ complexity\ ] + [selection\ complexity] + [loop\ complexity] \\ &= [(adt_d)\ (td)(w_d)] + [(adt_s)\ (td)(w_s)] + [(adt_l)\ (td)(w_l)] \\ &= [(1)\ (2)(0.2)] + [(0)\ (0)(0.3)] + [(1)\ (2)(0.9)] \\ &= 2.2 \end{aligned}$$

$$\begin{aligned} tc &= 1 + (c_n + c_{arr} + c_{sum}) \\ &= 1 + (0.1 + 0.1 + 2.2) \\ &= 3.4 \end{aligned}$$

$Cc = 3.4^{0.1} = 1.1302$

Table 3.4 shows the ranges between original Cc measures and modified Cc measures.

Table 3.4 Ranges of Cc measure between cohesion levels

Procedure	Cc measure	Modified Cc measure
Sum1_and_Sum2	44	3.4229
Range	34.2532	0.5342
Sum1_or_Sum2	9.7468	3.9571
Range	7.1861	1.951
Prod1_and_Prod2	2.5607	2.0061
Range	0.3918	0.2609
Sum_and_Prod	2.1689	1.7452
Range	0.5654	0.4995
Fibo_Avg	1.6035	1.2457
Range	0.0483	0.1155
Sum	1.5552	1.1302

3.4 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 [9]. Such an explicit decomposition is illustrated in Fig. 3.10.

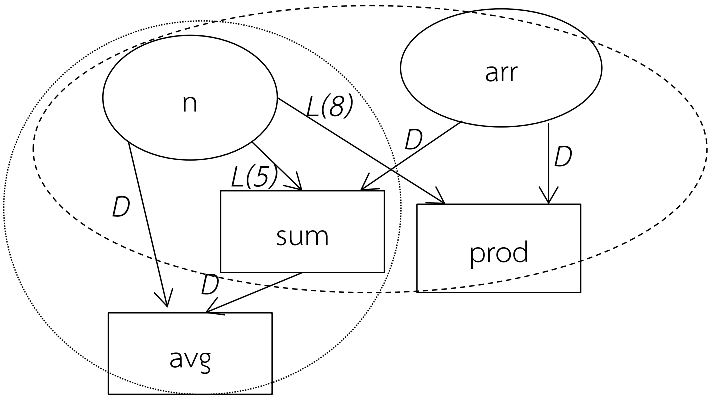


Figure 3.10 Variable dependence graph of module Sum_and_Prod

There are two $cohesion\_between\_pairs$ in the original Sum_and_Prod procedure, i.e., $sequential$ and $communicational$ cohesions as shown earlier. The module is decomposed into two blocks. The first block is composed of n, arr, sum, and avg, the two output variables form $sequential$ cohesion. The other module 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 1.7452 and after decomposition for both blocks are 1.2287 and 1.6767. 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.

CHAPTER 4 EXPERIMENT

In the experiment, both Cc measure and modified method were tested with module designs and real programs. Programs and module designs were translated into VDG before inputted to a Cohesion Complexity Measure tool or CCM. CCM automatically computes Cc value, cohesion between pairs, and module cohesion. Results of the experiment are described in the sections that follow.

4.1 Experiments on Cc Measure

Two programs written in C from [10] and [11] and nine modules from [12] and [13] were used. The first program is a "Tic Tac Toe" game and the second one is a phone service called "PHONEV2A." The former contains six modules and the latter contains thirteen modules. Table 4.1 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 4.2 and 4.3 depict the results of all test programs (whose name appears in column one) cohesion complexity with help of the CCM tool. The second column shows type 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 3.2. 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 Eq (3.3).

Table 4.1 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 4.2 Results of modules in Tic Tac Toe 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
Boxesleft	Functional	Functional	1.2917

Table 4.3 Results of modules in PHONEV2A 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 Sequential	Procedural	3.6109
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	-

4.2 Experiments on Modified Cc Measure

In modified Cc measure method, some algorithms in [11] which are written in C, all previous input programs, and designed module are tested. The results of Cc value are shown as follows:

Table 4.4 Results of module cohesion level and corresponding Cc value

Name	Cohesion Found	Module Cohesion	Cohesion Complexity
Sum1_and_Sum2	Coincidental	Coincidental	3.4229
Sum1_or_Sum2	Logical	Logical	3.9571
Prod1_and_Prod2	Procedural	Procedural	2.0061
Sum_and_Prod	Coincidental Communicational Sequential	Communicational	1.7452
Fibo_Avg	Sequential	Sequential	1.2457
Sum	Functional	Functional	1.1302
Avg_or_Range	Logical	Logical	5.7957
Avg_and_SD	Communicational	Communicational	1.9267
SD_and_Var	Sequential	Sequential	1.4404

Table 4.5 Results of modules in Tic Tac Toe and Cc assessment

Name	Cohesion Found	Module Cohesion	Cohesion Complexity
Showframe	Coincidental	Coincidental	1.5672
Showbox	Undefined	Undefined	-
Putintobox	Functional	Functional	1.0820
Gotobox	Undefined	Undefined	-
Navigate	Functional	Functional	1.0481
Checkforwin	Functional	Functional	1.0342
Boxesleft	Functional	Functional	1.0342

Table 4.6 Results of modules in PHONEV2A and Cc assessment

Name	Cohesion Found	Module Cohesion	Cohesion Complexity
menu	Functional	Functional	1.0184
chkstrdig	Undefined	Undefined	-
DeleteEntry	Coincidental Procedural Sequential	Procedural	5.9013
FindPhone	Procedural Sequential	Procedural	3.4181
FindRoom	Procedural Sequential	Procedural	3.4181
GeTotalEntries	Functional	Functional	1.0096
ListAll	Sequential	Sequential	1.2106
SortAllEntries	Coincidental Procedural Sequential	Procedural	3.4307
AddEntry	coincidental	coincidental	1.5090
drawscreen	undefined	undefined	-
exitmenu	Procedural Sequential	Procedural	1.2697
LoadDB	Coincidental Procedural	Procedural	3.0009
refreshscreen	undefined	undefined	-

Table 4.7 Results of Algorithm modules and Cc assessment

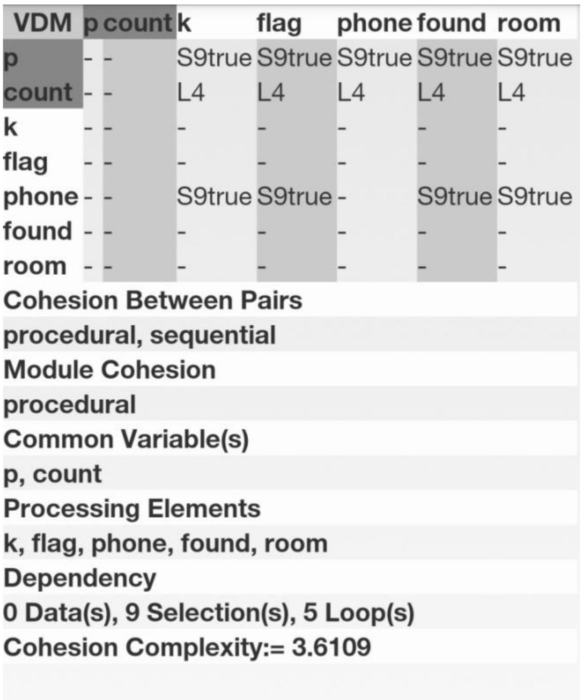
Name	Cohesion Found	Module Cohesion	Cohesion Complexity
inputa	Functional	Functional	1.0718
outputa	Functional	Functional	1.1792
swap	Sequential	Sequential	1.0986
dosearch	Undefined	Undefined	-
getyear	Functional	Functional	1.0096
get_day_code	Functional	Functional	1.1722
get_leap_year	Undefined	Undefined	-
print_calendar	Functional	Functional	1.3057

From the experiments of both methods, coincidental cohesion gives the highest value 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. 4.1 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 They cannot be classified the level of module cohesion since they have no output variable, i.e., processing element. All modules in Table 4.8 were also tested against the FC measure.

Table 4.8 Results of Cc and Fc assessments

Name	SMC Cohesion	Cc Measure	Modified Cc Measure	FC Measure	
Sum1_and_Sum2	Coincidental	44	3.4229	WFC	0.28
				SFC	0.28
				A	0.28
Sum1_or_Sum2	Logical	9.7468	3.9571	WFC	0.3846
				SFC	0.3846
				A	0.3846
Prod1_and_Prod2	Procedural	2.5607	2.0061	WFC	0.2380
				SFC	0.2380
				A	0.2380
Sum_and_Prod	Communicational	2.1689	1.7452	WFC	0.6957
				SFC	0.2174
				A	0.5362

Fibo_Avg	Sequential	1.6035	1.2457	WFC	1
				SFC	1
				A	1
Sum	Functional	1.5552	1.1302	WFC	0
				SFC	1
				A	0
Avg_or_Range	Logical	12.6491	5.5887	WFC	0.3333
				SFC	0.3333
				A	0.3333
Avg_and_SD	Communicational	2.2974	1.9267	WFC	0.3214
				SFC	0.3214
				A	0.3214
SD_and_Var	Sequential	1.8644	3.4229	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

Figure 4.1 Screen capture of CCM on FindPhone program

4.3 Cohesion Complexity Measure Tool

Cohesion Complexity Measure (CCM) is a tool that computes Cc value, module cohesion, and cohesion between pairs. The tool is written in JAVA language running on android platform. This application can also run on Windows by means of emulators such as BlueStacks. CCM is designed to use VDG as its input. Users can input the VDG by using two alternative methods. First input the graph manually and second input the graph via text file in a designated format.

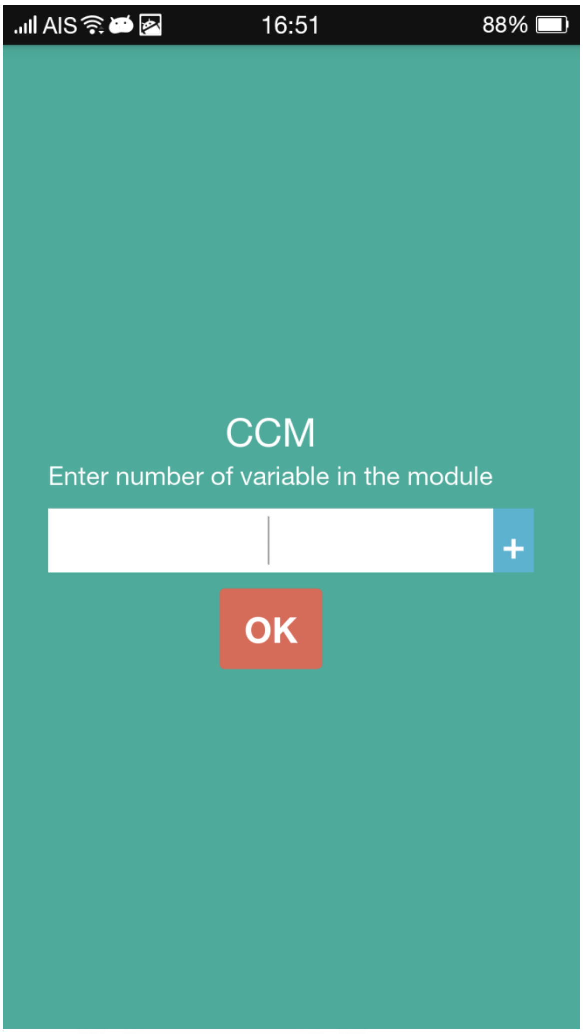
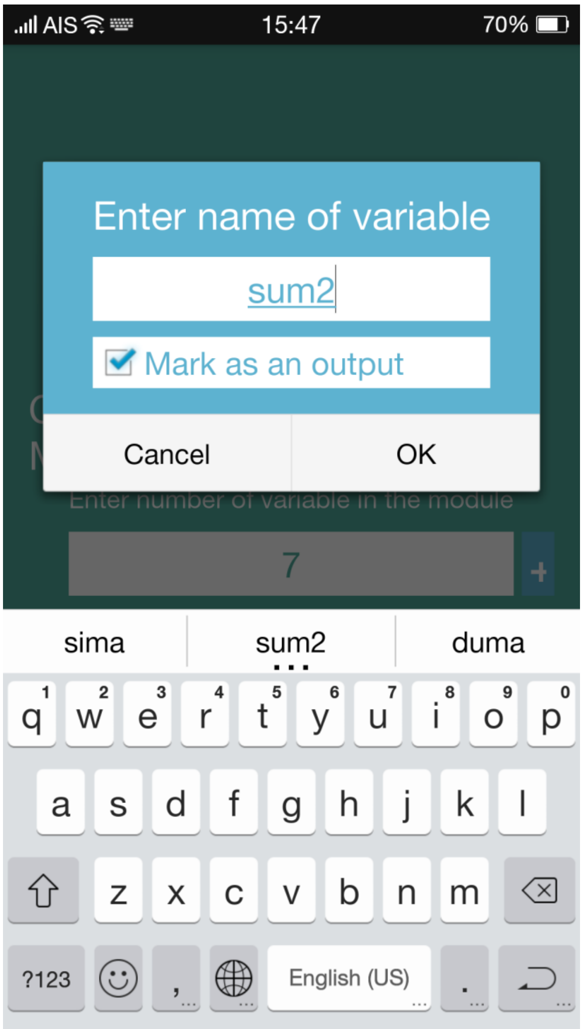


Figure 4.2 Home screen

Fig 4.2 illustrates the home screen of CCM tool where user can enter the number of variables appears in the module. Then click the OK button, A window will pop up with textbox to permit variable name input. If the variable is an output, click the checkbox, otherwise, leave it blank. A sample input screen is shown in Fig 4.3. The process repeats until all input variable names are entered. Click the Next button to go to the variable list screen as shown in Fig 4.4. This screen allows the user to make a final change to the variable list.



Enter name of variable

sum2

☑ Mark as an output

Enter number of variable in the module

7

Figure 4.3 Input screen Of CCM

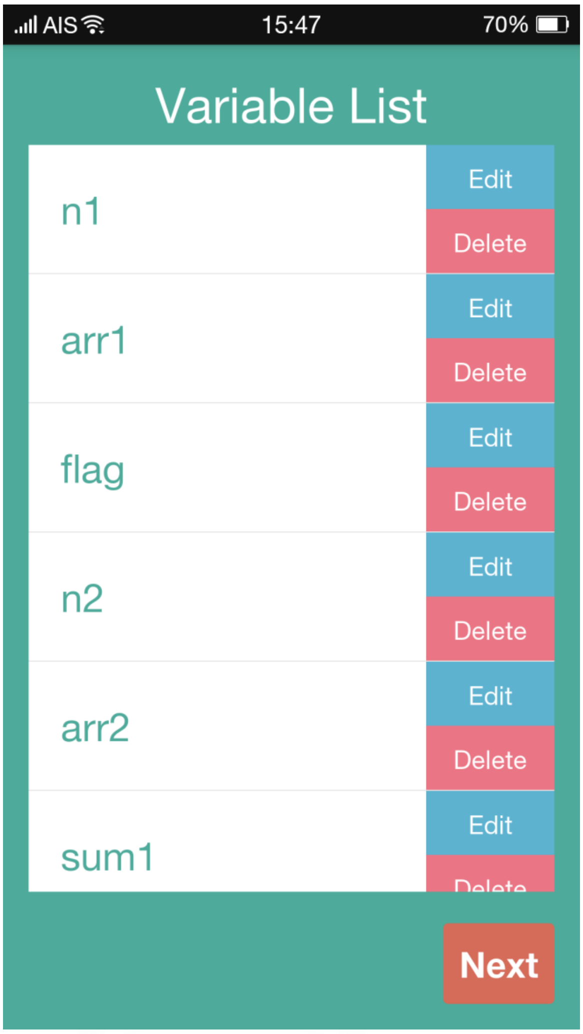
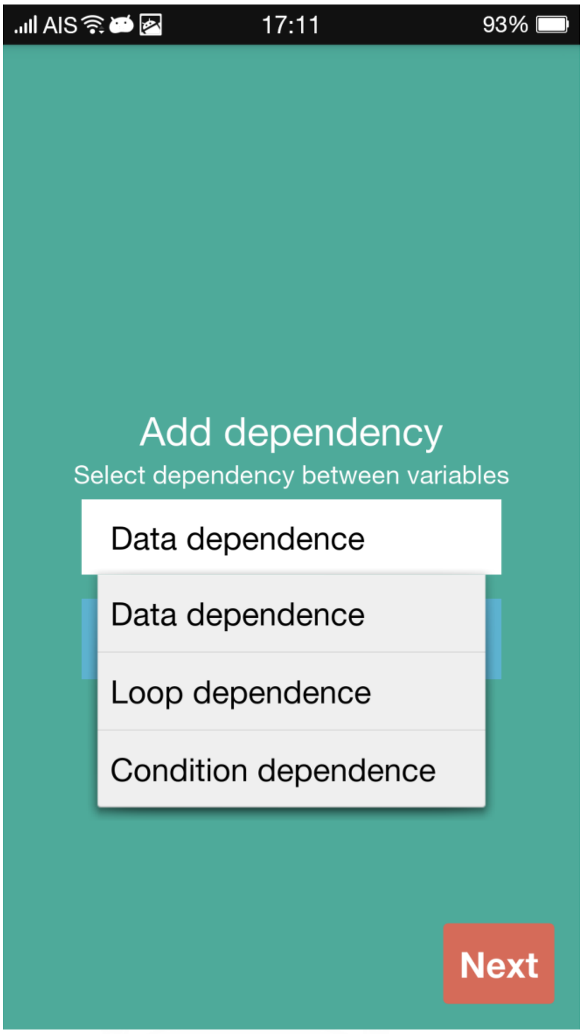


Figure 4.4 Variable list screen



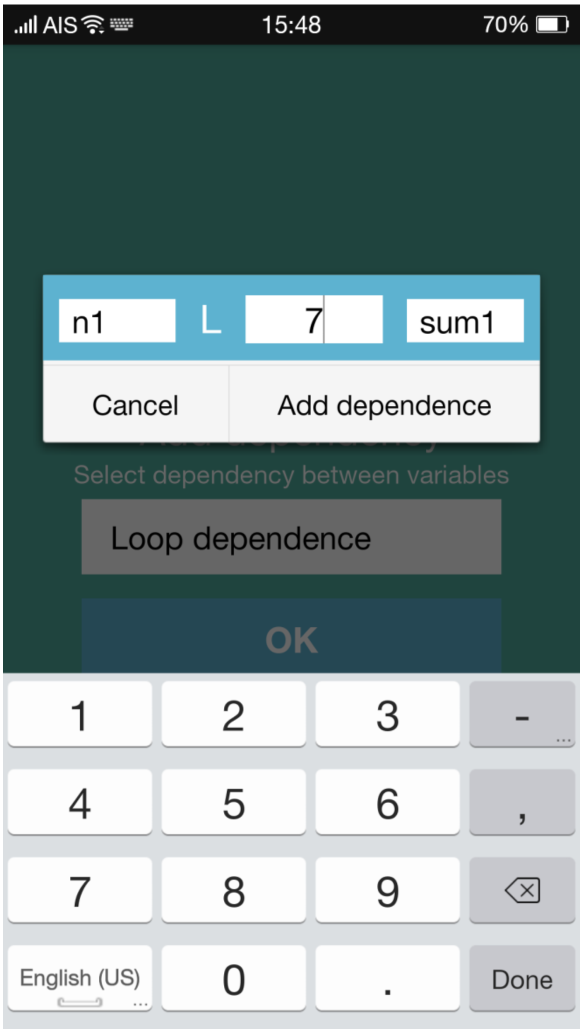
Add dependency

Select dependency between variables

Data dependence

Figure 4.5 Dependence screen

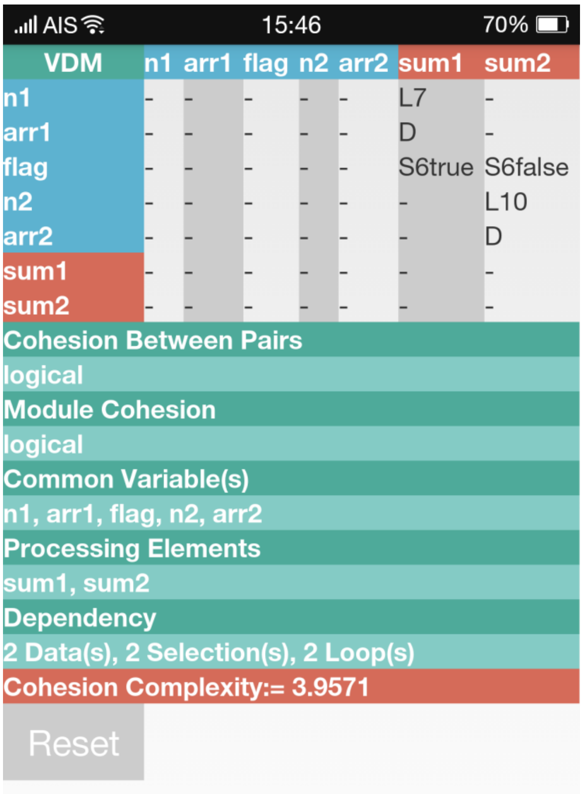
The next screen is the dependence screen to enter dependencies between variables. There are 3 types of dependency, data, loop, and, selection dependence.



n1 L 7 sum1

Figure 4.6 Dependence window

After selecting the type of variable dependence, a window pops up to add more details relating to the dependence pair. In Fig 4.6, variable sum1 depends on n1 by loop dependence at statement 7.



VDM	n1	arr1	flag	n2	arr2	sum1	sum2
n1	-	-	-	-	-	L7	-
arr1	-	-	-	-	-	D	-
flag	-	-	-	-	-	S6true	S6false
n2	-	-	-	-	-	-	L10
arr2	-	-	-	-	-	-	D
sum1	-	-	-	-	-	-	-
sum2	-	-	-	-	-	-	-

Cohesion Between Pairs
logical
Module Cohesion
logical
Common Variable(s)
n1, arr1, flag, n2, arr2
Processing Elements
sum1, sum2
Dependency
2 Data(s), 2 Selection(s), 2 Loop(s)
Cohesion Complexity:= 3.9571

Figure 4.7 Result screen

When all the dependencies are added, CCM will compute the result of module analysis as shown in Fig 4.7. The result screen depicts dependence matrix between variables, where output variables are highlighted in red. Other statistics such as cohesion between pairs, module cohesion, common variables, processing elements, and Cc value are also shown.

The alternative option to input the VDG is to insert a text file that contains information about the graph in a designated format. This text file is created at the time the application is launched. The text file ccm_text is formatted as follows:

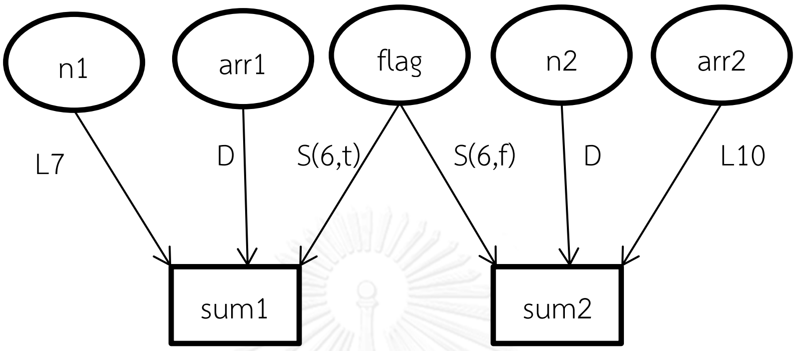
variable$_1$, variable$_2$, variable$_3$,…, variable$_n$;

variable$_{X1}$ -dpd-> variable$_{Y1}$, variable$_{X2}$ -dpd-> variable$_{Y2}$,
variable$_{X3}$ -dpd-> variable$_{Y3}$, … ,variable$_{Xm}$ -dpd-> variable$_{Ym}$;

The `dpd` is dependency for each dependence type, i.e. data, loop, and selection labeled by `D`, `L`, and `S`, respectively. Loop and selection dependence must be followed by the number of statements that the loop or selection occurs in the source code.

The first statement contains all the variable, each of which is delimited by comma (`,`). Output variables must be followed by an asterisk (`*`). The second statement holds the dependence statement delimited by comma. Both statements terminated by semicolons (`;`). Example 4.1 depicts the VDG file format.

Example 4.1: VDG format



```
n1, arr1, flag, n2, arr2, sum1*, sum2*; n1 -L7-> sum1,
arr1 -D-> sum1, flag -S6true-> sum1, flag -S6false->
sum2, n2 -L10-> sum2, arr2 -D-> sum2;
```

4.4 Application of the CCM Tool

The implementation of Cc computation and CCM tool helps developers determine whether any modules should be decomposed or not. For example, the result of Avg_or_Range module in Table 4.4 is quite prominent compares with the rests of module in the same program. The module gets 5.7957 score while the others get less than 3.9 and mostly just above 1.0. Thus, module Avg_or_Range is the first candidate that deserves developers' attention. Since the cohesion of this module is $logical$ cohesion and also only $logical$ cohesion found by cohesion between pairs, no better cohesion type can be selected. Nevertheless, developers can use the techique provided in Section 3.4. Since, module Avg_or_Range has no other cohesion between pairs, the only way to decompose this module by separating the output variables as shown in Fig 4.8

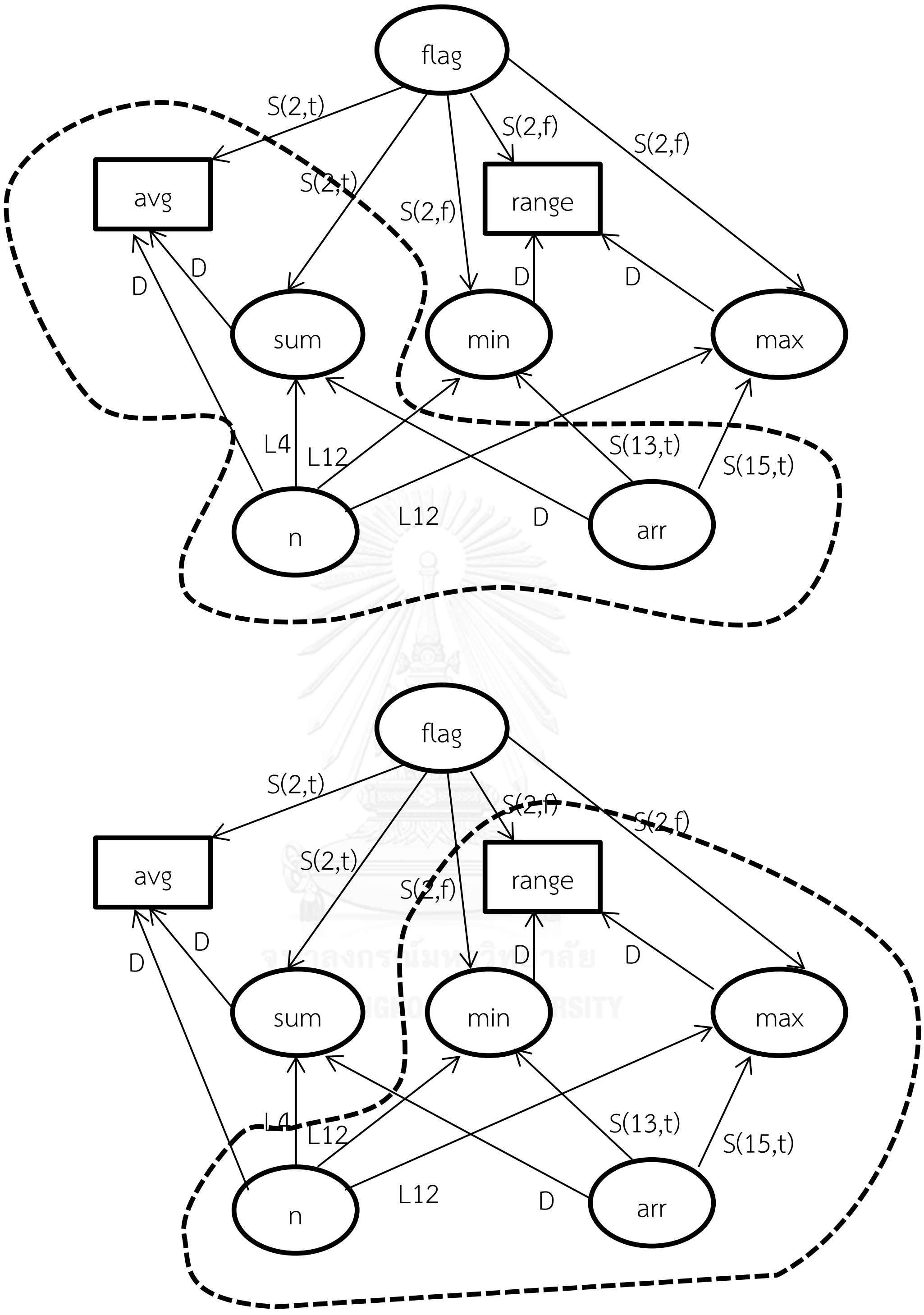


Figure 4.8 Module decomposition for Avg_or_Range

After decomposition, the original module is divided into two modules having $functional$ cohesion with the Cc values to be 1.1543 and 1.2113, respectively.

CHAPTER 5 DISCUSSION AND CONCLUSION

A module should encapsulate some well-defined, coherent piece of functionality so that it is easy to maintain, reuse, and portable. This proposed method has followed SMC cohesion by adopting association rules, variable dependence graph, and using output variables as processing elements [4] to determine the level of cohesion. Such a quantification helps 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 flaws 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 $Fibo\_Avg$.

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 better understood and carried out easier and. Third, production of software can keep pace with the rapid technological innovation. As a case in point, various modifications, feature enhancement, and bug fixes of facebook [14] that have undergone world-wide test and used over the years could have been performed with fewer efforts and more objective design decisions. 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.

In this research, every VDGs were manually constructed. Thus, all sample modules were confined to small programs. For medium to large programs, it would be expedient if there is a supporting to convert source code tool to the VDG. Thereby the output VDG can be further processed by the proposed CCM tool.

Alternatively, one can transform source module into VDG directly using some refine language [15] having predefine rules and pattern matching.

REFERENCES

1. Stevens, W.P., G.J. Myers, and L.L. Constantine, *Structured design.* IBM Systems Journal, 1974. **13**(2): p. 115-139.
2. Yourdon, E. and L.L. Constantine, *Structured Design*. 1978: Yourdon Press.
3. Page-Jones, M., *The practical guide to structured systems design: 2nd edition*. 1988: Yourdon Press. 368.
4. Lakhotia, A., *Rule-based approach to computing module cohesion*, in *Proceedings of the 15th international conference on Software Engineering*. 1993, IEEE Computer Society Press: Baltimore, Maryland, USA. p. 35-44.
5. Nandigam, J., *A measure for module cohesion*. 1995, University of Southwestern Louisiana.
6. Bieman, J.M. and L.M. Ott, *Measuring Functional Cohesion.* IEEE Trans. Softw. Eng., 1994. **20**(8): p. 644-657.
7. Aho, A.V., R. Sethi, and J.D. Ullman, *Compilers: principles, techniques, and tools*. 1986: Addison-Wesley Longman Publishing Co., Inc. 796.
8. Hecht, M.S., *Flow Analysis of Computer Programs*. 1977: Elsevier Science Inc. 232.
9. Lakhotia, A. and J.C. Deprez. *Restructuring functions with low cohesion*. in *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*. 1999.
10. *Game Programming in C - For Beginners*. Available from: http://www.codeproject.com/Articles/447332/Game-Programming-in-C-For-Beginners.
11. *Source code - Projects*. April 1, 2014]; Available from: http://www.cprogramming.com/source/phone.zip?action=Jump&LID=44.
12. Bieman, J.M. and B.-K. Kang, *Measuring Design-Level Cohesion.* IEEE Trans. Softw. Eng., 1998. **24**(2): p. 111-124.
13. *Designed Modules*. April 1, 2014]; Available from: https://www.dropbox.com/s/y8902tb6or6s8j9/3_procedures.txt.

14. *Facebook for Android Beta App Change History*. April 1, 2014]; Available from: https://www.facebook.com/notes/facebook-android-beta/facebook-for-android-beta-app-change-history/190854467764586.

15. Kotik, G. and L. Markosian. *Application of REFINE Language Tools to software quality assurance*. in *Knowledge-Based Software Engineering Conference, 1994. Proceedings., Ninth*. 1994.

APPENDIX

VITA

Pimvard Charoenporn was born in Bangkok, Thailand on 14 October 1990. In 2012 received the B.S. in Computer Science from Chulalongkorn University. Currently, studying for a M.S. in Computer Science, Chulalongkorn University. The areas of interest are software engineering, data structures, and design and analysis of algorithms.