ASSESSING SOFTWARE QUALITY THROUGH VISUALIZED COHESION METRICS

Timothy K. Shih* Ming-Chi Lee** Teh-Sheng Huang* Lawrence Y. Deng*

*Dept. of Computer Science and Information Engineering Tamkang University
Taiwan, R.O.C
E-mail: g8190070@tkgis.tku.edu.tw

**Dept. of Business Administration National PingTung Institute of Commerce
Taiwan, R.O.C

ABSTRACT

Cohesion is one of the most important factors for software quality as well as maintainability, reliability and reusability. Module cohesion is defined as a quality attribute that seeks for measuring the singleness of the purpose of a module. The module of poor quality can be a serious obstacle to the system quality. In order to design a good software quality, software managers and engineers need to introduce cohesion metrics to measure and produce desirable software. A highly cohesion software is thought to be a desirable constructing. In this paper, we propose a function-oriented cohesion metrics based on the analysis of live variables, live span and the visualization of processing element dependency graph. We give six typical cohesion examples to be measured as our experiments and justification. Therefore, a well-defined, well-normalized, well-visualized and well-experimented cohesion metrics is proposed to indicate and thus enhance software cohesion strength. Furthermore, this cohesion metrics can be easily incorporated with software CASE tool to help software engineers to improve software quality.

Keywords: Software metrics, Cohesion, Software Quality, Live Variables

INTRODUCTION

The quantitative measurement of software has been defined as one of the functions of engineering disciplines. The establishment and the use of software measurement still make a great deal of extensive discussion. Software production is often out of control due to the lack of measurement [DeMarco, T. (1982)]. Without software metrics, software would be error prone, expensive and with a poor quality. Therefore, suitable software measures are essential for software development. Modularization of software represents a significant software engineering activity, which continues to receive considerable research attention.

Module cohesion is defined as a quality attribute, which seeks for measuring the singleness of purpose of a module. Cohesiveness is a measure of an individual module's internal strength, which is the strength of the interrelationship of its internal element [Martin, J. McClure, C. (1983)]. Module cohesion indicates how closely a module's internal components are related to others. Cohesion is a critical assessment of software design quality. A software module with low cohesion will be error prone. Fifty percent of the highly cohesive procedures are fault free, whereas only eighteen percent of procedures with low cohesion are fault free [Fifty_Percent, 1998]. Cohesion is an attribute that can predict properties of implementations such as "ease of debugging, ease of maintenance, and ease of modification" [Yourdon E. and Constantine L.L. (1979)]. Cohesiveness is one of the possible measurements for program modifiability [Stevens, W., Myers, G. and Constantine, L. (1974)]. A modifiable program is made up of modules that have high cohesiveness. Therefore, well-defined and well-proved cohesion measures can support software engineer to design a higher cohesion module and enhance software quality, maintainability and reusability. This implies that the software quality can be improving by maximizing the degree of module cohesion. However, Stevens, Myers and Contantine first define the notion of cohesion (SMC Cohesion), on ordinal order of seven levels that describes the degree of modules. The factors are *functional*, *sequential*, *communicational*, *procedural*, *temporal*, *logical* and *coincidental* [Stevens, W., Myers, G. and Constantine, L. (1974)][Yourdon E. and Constantine L.L. (1979)]. The cohesion strength of a specific procedure is increasing from *coincidental* to *functional*. A highly cohesive software module has one basic function and is indivisible [Fenton, N. E. (1991)]. And, the *coincidental* is the least desirable but the *functional* is the most desirable cohesion level.

In the past decade, a few researchers work on cohesion measures. Biemen's cohesion measures is based on program slicing. A program slice is a portion of program text that affects specified program variables [Bieman, J. M. and Ott, L. M. (1994)]. The program slice based measures depends on data flow and control flow of a procedure [Bieman, J. E. and Kang, B.K. (1998)]. For complex program functionality, the program slicing method can produce slices that are either too large to understand, or too simple but not significant to comprehend [Lucia, A.D and Fasolino, A.R. (1996)]. On the other hand, the mechanism does not consider the global data variables effect and the analyzed scope is not a formal module scope. Lakhotia developed a logical analysis method to investigate cohesion types based on logical analysis [Lakhotia, A. (1993)]. Although, the result of the logical analysis meets the SMC Cohesion levels logically. But, it does not provide numerical system to compute cohesion strength. Ott and Thuss's approach is based on slice profile and processing element flow graph to express four types of module cohesion [Ott, L. M. and Thuss, J. J. (1989)]. But, this approach does not consistent

with the SMC Cohesion and the Fenton's cohesion spectrum. Also, this method does not express a numerical system how to compute module cohesion. Fortunately, the well-known Henry-Kafura measurement measures the relationship between procedures based information flow [Henry, S. and Kafura, D. (1981)]. But, it fails to measure the cohesion attribute of intra-procedure.

Live variables describe the extent of variables to be refereed within a module, while variable span captures the range of variable to be used in a module [Conte, S. D., Dunsmore H. E. and Shen, V.Y. (1986)]. Generally, instance variable is used to derive higher-level inference on cohesion [Metr_dis, 1998]. Module cohesion aims to measure how tight between output processing elements [Ott, L. M. and Thuss, J. J. (1989)][Bieman, J. E. and Kang, B.K. (1998)][Bieman, J. M. and Ott, L. M. (1994)]. In this paper, we propose a function-oriented cohesion metrics, which is based on an analysis model of live variable semantics. We examine the function-oriented cohesion metrics (FOCM) for each output function, the most tightest function-oriented cohesion metrics (MTFOCM), the least tightest function-oriented cohesion metrics (LTFOCM), and the average tight function-oriented cohesion metrics (ATFOCM) of a specific procedure.

In general, we always strive for a high cohesion. Although the mid-range of the cohesion spectrum is often acceptable [Pressman, R.S. (1988)]. And, a module may exhibit more than one type of cohesion. However, we will show that the proposed metrics do not only closely match SMC Cohesion and Fenton's cohesion strength spectrum [Fenton, N. E. (1991)], but also meet nonlinear cohesion scale of cohesion that was stressed by Pressman and Sommerville in [Pressman, R.S. (1988)][Sommerville, I. (1996)]. The main contributions of this paper are that we propose a formal and visualize analysis model, scientific basis, well-normalized, well-experimented, and well-evaluated function-oriented cohesion measures to improve software quality.

The rest of the paper is organized as follows. We address basic works about live variable semantics and formal definitions in section 2. We examine visualized analysis model and the proposed cohesion metrics in section 3. Section 4 gives six typical cohesion procedure examples to make empirical study. To discuss and analysis our experimental results is expressed in section 5. Finally, in section 6, the conclusion and future research are presented and discussed.

BASIC WORKS AND DEFINITIONS

A module is a contiguous sequence of program statements that are bounded by boundary elements, and has an aggregate identifier [Yourdon E. and Constantine L.L. (1979)]. A module is usually defined as a segment of codes that is compilable independently [Conte, S. D., Dunsmore H. E. and Shen, V.Y. (1986)]. However, many researchers define a module to be a compilation unit of code, a view of program, or a procedure. By functionality, the processing elements in a procedure could be categorized into the following four categories:

- Input variable (IV): a set composed of the variables that are the input arguments to a procedure.
- Internal variable (INV): a set of local variables of a procedure.
- Global variable (GV): a set of global variables.
- Output variable (OV): a set of variables that are the output functions of a procedure.

In general, variables in an individual module can be in assignment statement, initial statement, decision predicate, output function and iteration constructor, Boolean semantics which could construct the main body of the whole procedure. A module will contain specific function semantics. In practice, it will be the key issue to develop a suitable model to construct the overall reference scenarios of the function in the procedure scope. Live variables of a statement in module are the processing elements that are referenced in the statement. For sake of explaining the concept of live variables, a procedure example is given as follows:

```
1 Procedure SumAndProd(N:integer; Var sum; prod:Integer)
2 Var I:Integer
3 Begin
4  sum:=0;
5  prod:=1;
6   For I:=1 to N do begin
7    sum:=sum+I;
8    prod:=prod*I;
9  end
10 end
```

Figure 1. A procedure example

As *SumAndProd* listed in Figure 1, *sum* in line 4 is a live variable. Both *I* and *N* are live variables in line 6. On the other hand, *sum* is only referred at two statements in the procedure. Should software engineers concern *sum* in other statements other than line 4 and 7? The answer is "yes". However, software engineers constantly have to keep in mind where *sum* may be referred between statements 4 and 7. Software engineers always realize that in each iteration *sum* has a different status in the for_loop, *sum* will be incremented, although the statements do not refer to *sum* in line 5 and 6. Finally, the summation operation will exit. Similarly, *prod* has the same scenario like *sum*. And, Comprehending software artifacts is an important software engineering activities. A significant amount of time of a software engineer is spent in looking at the source code to discover information during testing, review and code inspection [Canfora G. and Lakhotia A.(1999)]. We also know a programmer must be aware of the status of a number of data items during the programming process [Conte, S. D., Dunsmore H. E. and Shen, V.Y. (1986)]. Thus, the more data items what a programmer must keep track of when constructing a procedure, the more difficult it is to construct. Based on the above issues, live variables of a statement are not limited to the number of variable reference in that statement. In the past years, there are several possible scopes of live variables to be defined inevitably [Curtis B. (1981)][Conte, S. D., Dunsmore H. E. and Shen, V.Y. (1986)]. The first is the live variable computes from the beginning of a procedure to the end of the procedure. According to this definition, the computation is simple, but this violates the live variable abstract. The second is at a particular statement, only if variable is referenced a contain number of statements before or after that statements. Depend on the second definition, we may calculate lifespan. But there is no agreement on what "certain number of statements" should be and no successful use has been reported [Conte, S. D., Dunsmore H. E. and Shen, V.Y. (1986)]. The third one is from its first reference to its last reference within a procedure. However, the third definition not only meets the principles of live variable abstraction, but can be counted algorithmically as well. We believe that the third is a suitable domain for computing module cohesion. Therefore, we will adopt the third as the definition of live variables in this paper. For precise definition and helping software engineer to understand software easily, we use mathematics to define the live variables and use live span to visualize a live variable scope. Therefore, we denoted the lifespan of a variable to be the reference domain that begins at the first occurrence and extends through the last occurrence.

Definition 1: The lifespan of a variable in a procedure is a set, denoted by LS(*var_nam*). The elements of the set are the name of the variable from the first (*i*-th line) to the last (*j*-th line) referenced, LS(*var_nam*)={var_nam_i,var_nam_{i+1}.....,var_nam_j}. And, the size of lifespan, | LS(*var_nam*) | = j-i+1.

As an example in Figure 1, we investigate the live span of variable *sum*. The set LS(*sum*)={sum_4,sum_5,sum_6,sum_7}, with a size of 4. The LS set also represents a mapping from the references of *sum* to different locations (i.e. lines 4,5,6 and 7).

Definition 2: The live variables (LV) of a specific procedure (*sp*) is the set union of the lifespan of each variables which belongs to IV, INV, GV or OV. We denote LV(*sp*)= $\cup$ LS(var_nam_i) where $var\_nam_i \in$ IV $\cup$ INV $\cup$ GV $\cup$ OV and $i \in$ N.

In Figure 1, we could construct LV(*SumAndProd*). First, we know the processing elements are IV={*N*}, OV={*sum*,*prod*},GV=$\varnothing$, and INV={*I*} respectively. Next, LS(*inv*)={I_6,I_7,I_8} where $ivn \in$ IVN, LS(*ov*)={sum_4,sum_5,sum_6,sum_7,$prod_5$,$prod_6$,$pord_7$,$prod_8$} where $ov \in$ OV, LS(*gv*)= $\varnothing$ where $gv \in$ GV, LS(*iv*)={N_6} where $in \in$ IV. Finally, LV(*SumAndProcd*)=LS(*iv*) $\cup$ LS(*ov*) $\cup$ LS(*gv*) $\cup$ LS(*inv*).
For simplicity, we can describe the live variables and live span of the procedure as Figure 2.

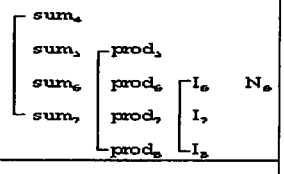
Line	LV(*SumAndProd*)	Count	Live Span
4	sum_4	1	
5	sum_5,$prod_5$	2	
6	sum_6,$prod_6$,I_6,N_6	4	
7	sum_7,$prod_7$,I_7	3	
8	$prod_8$,I_8	2	
		12	

Figure 2. Live variables and live span of *SumAndProd*

The live span and live variables of Figure 1 are illustrated in Figure 2. For instance, *sum* is firstly referenced in line 4, the last reference is in line 7. According to Definition 1, the software engineer have to keep the reference of *sum* from line 4 and line 7, although *sum* does not appear in statement 5 and statement 6. Therefore, line 4 to line 7 are the live span of variable *sum* and we use a symbolic "[" to describe the live span of a variable. Together, a live span could clearly be viewed as a visualized scope of live variables in a specific procedure.

VISUALIZATION AND COHESION METRICS

In this section, we propose function-oriented cohesion measures that are based on the analysis of live variables and live span abstraction. According to the previous section, we know that not all variables involve the computing of output functions in a specific procedure. Logically, a module is the function that processes input, internal and global variables to produce a result. The output functions in OV(*sp*) are on behalf of the result of the module. In fact, not all elements of live span of a variable contribute a result of the output function of the procedure. For instance, the LS(*prod*) consists of $prod_5$, $prod_6$, $prod_7$, and $prod_8$, in which no element influences the result of output *sum*. But both $prod_5$ and $prod_8$ may involve the value of the output function *prod* through the assignment statement. As of the result, $prod_5$ and $prod_8$ are the function-oriented live variables of output function *prod*. Similarly, variables I_6, I_8, N_6 are the function-oriented live variables of *prod*. Both I_6 and N_6 may change the value of the output function and compute the multiplication via I_8 indirectly. Therefore, function-oriented live variables of *prod* is a set, denoted $\{prod_5, I_6, N_6, prod_8, I_8\}$. Therefore, the FOLV and FOLS of the procedure *SumAndProd* are shown in Figure 3.

However, it is not enough formal that we express function-oriented live variables and live span just rely on the above description. Therefore, firstly we will define direct variable and indirect variable of an output function before giving the definitions of function-oriented live variables.

Definition 3. *Direct variable* of an output function is an element of LV(*sp*), which may influence the result of the output function in the same statement.

For instance, variable I_7 contributes the value of the sum_7 directly. Thus, I_7 is the direct variable of the output function sum_7. According to Definition 3, output variable surely is the direct variable by itself.

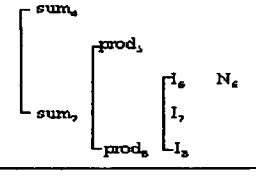
Line	FOLV(*SumAndProd*)	Count	FOLS
4	sum_4	1	
5	$prod_5$	1	
6	I_6,N_6	2	
7	sum_7,I_7	2	
8	$prod_8$,I_8	2	
		8	

Figure 3. The FOLV and function-oriented live span of *SumAndProd*

Definition 4. *Indirect variable* of an output function is an element of LV(*sp*), which may decide the execution of the output function or perform the result of the output function via direct variables.

For instance, variables I_6 and N_6 of iteration statement in line 6 contribute the value of the sum_7 indirectly. So I_6 and N_6 are the indirect variable of the output function sum_7.

Definition 5. The *function-oriented live variables* of an output function in the specific procedure is a set, denotes

$$FOLV(ov_i)=\{lv \in LV(sp) \mid lv \text{ is direct or indirect variable of } ov_i;\ ov_i \in OV(sp)\}$$

Definition 6. The *function-oriented live span* of an output function in the specific procedure is a set, denotes

$$FOLS(ov_i)=\{lv \in LS(sp) \mid lv \text{ is direct or indirect variable of } ov_i;\ ov_i \in OV(sp)\}$$

According to Definition 5 and 6, the function-oriented live variables and function-oriented live span of *sum* and *prod* are depicted in Figure 4.

Variables	FOLV(*sum*)	FOLV(*sum*)∩FLOV(*prod*)	FOLV(*prod*)
sum	sum_4, sum_7		
I	I_6, I_7	I_6	I_6, I_8
N	N_6	N_6	N_6
prod			$prod_5$,$prod_8$
	5	2	5

Figure 4. The FOLV of *sum*, *prod* and their intersection

In Figure 4, the FOLV of the output function *sum* is the set $\{sum_4, I_6, N_6, sum_7, I_7\}$. The size of the FOLV(*sum*) is 5. On the other hand, the FOLV of the output function *prod* is the set of $\{prod_5, I_6, N_6, prod_7, I_8\}$, the size of the FOLV(*sum*) is 5. We know each element in FOLV(*SumAndProd*) will appear either in FOLV(*sum*) or in FOLV(*prod*). which is defined by FOLV(*sum*) ∪ FOLV(*prod*) as shown in Figure 3. More importantly, the function-oriented live span could help software engineer to friendly visualize the scope of function oriented live variables in the specific procedure *SumAndProd*.

In general, the size of the FOLV(ov_i) is the number of elements which belongs to LV(*sp*), and influence the value of ov_i directly or indirectly. Then, the size of FOLV(ov_i) implies the number of live variables in the specific procedure, shall contribute to the result of the output function *ov*. More precisely speaking, the proportion of |FOLV(ov_i)| in |LV(*sp*)| can be viewed as the cohesion strength which just restricts on ov_i in a module. Therefore, the *function-oriented cohesion measure* restricts on ov_i is defined as follows:

$$FOCM(ov_i) = |\,FOLV(ov_i)| \,/\, |LV(sp)| \quad \text{for each } ov_i \in OV(sp)$$

From the perspective of the cohesion, the common shared function-oriented live variables of *sum* and *prod* are the most critical elements. Module cohesion aims to measure how tight between output procession elements. And, the relation levels of output function pairs determine a cohesion level of a module [Bieman, J. E. and Kang, B.K. (1998)]. We define the *most tight function-oriented cohesion measure* of the specific procedure (*sp*) as follows:

$$MTFOCM(sp) = |\cap FOLV(ov_i)| / |LV(sp)| \quad \text{for each } ov_i \in OV(sp)$$

In a specific procedure, some elements of live variables really may not involve in influencing the result of output functions. This implies that not all of live variables have function-oriented relation with each *ov*. Obviously, the live variable VL(*sp*) excluding the function-oriented live variable FOLV(*sp*) is a subset of live variables, which is defined as VL(*sp*)-FOLV(*sp*). Therefore, we define the *least tight function-oriented cohesion measurement* as:

$$LTFOCM(sp) = |\,LV(sp) - \cup\, FOLV(ov_i)| \,/\, |LV(sp)| \quad \text{for each } ov_i \in OV(sp)$$

However, it is also an important reference point for software measures that the average strength of all live variables involves the results of output functions. We propose the *average tight function-oriented cohesion measurement* of the specific procedure as follows:

$$ATFOCM(sp) = |\cup FOLV(ov_i)| \,/\, |LV(sp)| \quad \text{for each } ov_i \in OV(sp)$$

Consequently, We proposed four function-oriented cohesion measures in a specific procedure. The values of the four proposed cohesion measures are in the range between 0 and 1. Therefore, the numerical systems of the function-oriented cohesion measures is a well-normalized cohesion.

Practically, it is necessary to express the FOLV using a dependency graph to visualize the function-oriented relationships between processing elements for helping software engineer to easily comprehend the FOLV.

Definition 7. The *function-oriented live variables dependency graph* of an output function *ov* in a specific procedure is a directed graph, denoted as $FOLVDG_{ov}(V,E)$, where V is a finite set of elements called vertices which consist of FOLV(*ov*), E is a finite set of elements called edge which is defined as (*lv*, *ov*), where *lv* is direct or indirect variables of *ov*, $ov \in OV(sp)$ and $lv \in FOLV(ov)$. We use directed arrow with '*id*' and '*d*' to describe the indirect or direct function-oriented live variables of an output function, respectively.

Furthermore, a function-oriented dependency graph of a specific procedure may express the semantics of direct

or indirect dependency between function oriented live variables and output functions.

Definition 8. The *function-oriented live variables dependency graph* of a specific procedure is a directed graph, denoted as $FOLVDG_{sp}(N,E)= \cup FOLVDG_{ov}(N,E)$, for all $ov \in OV(sp)$.

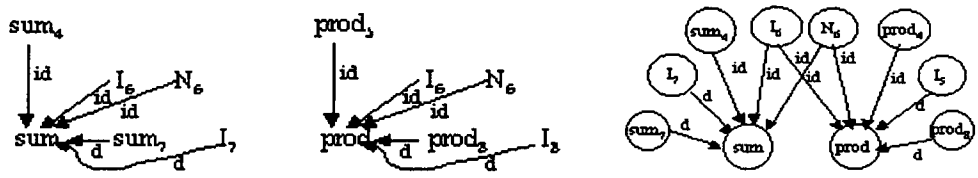


Figure 5. The FOLVDGs of *sum*, *prod* and *SumAndProd*

FOLVDGs of *sum*, *prod* and *SumAndProd* in Figure 5 are the visualizations of FOLV of *sum*, *prod* and *SumAndProd* respectively. It shall be very useful that the visualizations of FOLV may help software engineer to investigate the dependency relations between FOLV and output functions.

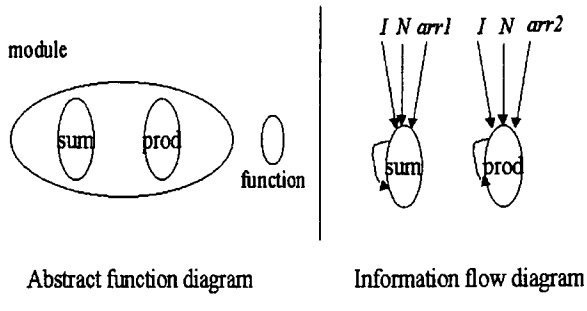
TYPICAL EXAMPLES AND EXPERIMENTS

In this section, there are six typical cohesion procedure implementations to be given to make an empirical study using our proposed cohesion measures. The purpose of measuring the distinctive cohesion procedure examples is to check the results of experiment of the proposed function-oriented cohesion measurements and to estimate whether they are compatible with the cohesion strength spectrum. Six specific procedure implementations are *coincidental, logical, procedural, communicational, sequential,* and *functional* cohesion examples. We believe that the experiments will not affect the completeness of the empirical study although the six implementations will not include the *temporal* one. A system initialization module can be considered as a typical *temporal cohesion.*

Cohesion is a measure of the strength of the interrelatedness of statements within a module. In general, there are many definitions of the seven levels of cohesion in [Fenton, N. E. (1991)][Modularity 1999][Cohesion 1999][Intramodule Cohesion 1999]. A functional cohesion is the strongest cohesion in the cohesion strength spectrum. Therefore, the functional cohesion is the most desirable.

Coincidental cohesion is to estimate whether a module performs more than one function, and whether there are unrelated [Fenton, N. E. (1991)]. In the *coincidental cohesion* example, there is no significant relationship between the output elements. It is only by coincidence that they are together. Similarly, it is hard to describe module purpose. However, a typical *coincidental cohesion* example and its abstract function diagram are given in Figure 6.

```
01    Procedure SumAndProd(N: integer; var sum, prod: integer; arr1,arr2:int_array);
02  var I: integer;
03  begin
04   sum:=0;
05   prod:=0;
06   for I:=1 to N do
07    sum:=sum+ arr1[I];
08   for I:=1 to N do
09         prod:=prod* arr2[I];
10  end
```



Abstract function diagram Information flow diagram

Figure 6. A typical *coincidental cohesion* implementation

By definition 1 and definition 2, we know $LV(SumAndProd)=\{sum_4,sum_5,prod_5,sum_6,prod_6,I_6,N_6,sum_7,prod_7,arr1_7,I_7,prod_8,I_8,N_8,prod_9,arr2_9,I_9\}$, and $|LV(SumAndProd)|=17$. We express LV and LS of the *coincidental cohesion* example in Figure 7. On the other hand, $FOLV(sum)=\{sum_4,I_6,N_6,sum_7,arr1_7,I_7\}$, $FOLV(prod)=\{prod_5,I_8,N_8,prod_9,arr2_9,I_9\}$ and $\cap FOLV(ov)=\{lv \in FOLV(sum)$ and $lv \in FOLV(prod) | lv \in LV\}=\varnothing$. They are shown in Figure 8. In Figure 9, the FOLV of *sum*, *prod* and their intersection are illustrated. FOLDGs of output functions and *coincidental cohesion* example are depicted in Figure 10.

line	LV(*SumAndProd*)	Count	LS
4	sum_4	1	
5	sum_5,$prod_5$	2	
6	um_6,$prod_6$,I_6,N_6	4	
7	sum_7,$prod_7$,$arr1_7$,I_7	4	
8	$prod_8$,I_8,N_8	3	
9	$prod_9$,$arr2_9$,I_9	3	
		17	

Figure 7. The LV, LS of a *coincidental cohesion* example

line	FOLV(*SumAndProd*)	Count	FOLS
4	sum_4	1	
5	$prod_5$	1	
6	I_6,N_6	2	
7	sum_7,$arr1_7$,I_7	3	
8	I_8,N_8	2	
9	$prod_9$,$arr2_9$,I_9	3	
		12	

Figure 8. The FOLV and FOLS of a *coincidental cohesion* example

Variables	FOLV(*sum*)	FOLV(*sum*) ∩FOLV(*prod*)	FOLV(*prod*)
Sum	sum_4, sum_7		
I	I_6,I_7		I_8,I_9
N	N_6		N_8
Arr1	$arr1_7$		
Arr2			$arr2_9$
Prod			$prod_5$,$prod_9$
	6	0	6

Figure 9. The FOLV of *sum*, *prod* and their intersection

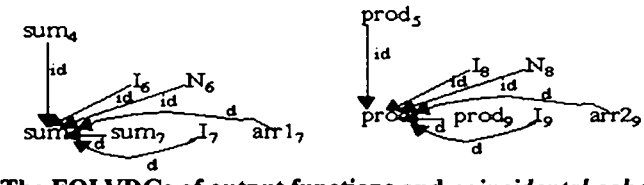


Figure 10. The FOLVDGs of output functions and *coincidental cohesion* example

According to the function-oriented cohesion measures presented in section 3, the values of function-oriented cohesion measures of *coincidental cohesion* example are given as follows:

FOCM(*sum*)=| FOLV(*sum*) | / | LV(*SumAndProd*) |=0.353
FOCM(*prod*)= | FOLV(*prod*) | / | LV(*SumAndProd*) |=0.353
MTFOCM(*SumAndProd*)=|∩FOLV(*ov*)|/|LV(*SumAndProd*) |=0
LTFOCM(*SumAndProd*)= |LV(*SumAndProd*) – ∪FOLV(*ov*) | / | LV(*SumAndProd*) |=0.29
AVFOCM(*SumAndProd*)=|∪FOLV(*ov*) | / | LV(*SumAndProd*) |=0.71

In this example, there are no common processing elements used to produce both output functions. And, the value

of MTFOCM of the coincidental cohesion is equal to zero. The *Coincidental cohesion* type is the lowest cohesion level.

Logical cohesion is to check whether the module performs more than one function, and whether there are related logically [Fenton, N. E. (1991)]. In a *logical cohesion* example, the module performs some related functions. One or more of them are selected by calling module. In other words, there is a dynamic function selection in the module. However, a typical *logical cohesion* example and its abstract function diagram are shown as follow.

```
01 Procedure SumAndProd(N: integer; var sum, prod: integer;arr1,arr2:int_array);
02 var I, J,flag: integer;
03 begin
04  sum:=0;
05  prod:=1;
06  if(flag>1)
07   for I:=1 to N do begin
08   sum:=sum+ arr1[I];
09  else
10   for I:=1 to N do begin
11   prod:=prod* arr2[I];
12 end
```

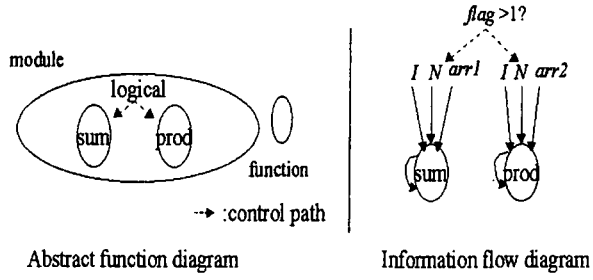


Figure 11. A typical *logical cohesion* implementation

By definition 1 and definition 2, LV(*SumAndProd*)={sum_4,sum_5,$prod_5$,sum_6,$prod_6$,$flag_6$,sum_7,$prod_7$,I_7,N_7,sum_8,$prod_8$,$arr1_8$,I_8,N_8,$prod_9$,I_9,N_9,$prod_{10}$,I_{10},N_{10},$prod_{11}$,$arr2_{11}$,I_{11} }, and |LV(*SumAndProd*)|=24, which are diagramed in Figure 12. In other way, we may have FOLV(*sum*)={sum_4,$flag_6$,I_7,N_7,sum_8,$arr1_8$,I_8}, and |FOLV(*sum*)|=7, FOLV(*prod*)={$prod_5$,$flag_6$,I_9,N_9,$prod_{10}$,$arr1_{10}$,I_{10}} and |FOLV(*prod*)|=7, and ∩FOLV(*ov*)={lv ∈ FOLV(*sum*) and lv ∈ FOLV(*prod*)| lv ∈ LV}={$flag_7$} and |∩FOLV(*ov*)|=1, and they are shown in Figure 13. In Figure 14, the FOLV of *sum*, *prod* and their intersection are illustrated. FOLDGs of output functions and *logical cohesion* example are illustrated in Figure 15.

Line	LV(*SumAndProd*)	Count	LS
4	sum_4	1	
5	sum_5,$prod_5$	2	
6	sum_6,$prod_6$,$flag_6$	3	
7	sum_7,$prod_7$,I_7,N_7	4	
8	sum_8,$prod_8$,$arr1_8$,I_8,N_8	5	
9	$prod_9$, I_9,N_9	3	
10	$prod_{10}$,I_{10},N_{10}	3	
11	$prod_{11}$,$arr2_{11}$,I_{11}	3	
		24	

Figure 12. The LV and LS of a *logical cohesion* example

Line	FOLV(*SumAndProd*)	Count	FOLS
5	sum_4	1	
6	$prod_5$	1	
7	$Flag_6$	1	
8	I_7,N_7	2	
9	sum_8,$arr1_8$,I_8	3	
10		0	
11	I_{10},N_{10}	2	
12	$prod_{11}$,$arr2_{11}$,I_{11}	3	
		13	

Figure 13. The FOLV and FOLS of a *logical cohesion* example

Variables	FOLV(*sum*)	FOLV(*Sum*)∩FOLV(*Prod*)	FOLV(*prod*)
Sum	sum_4,sum_8		
I	I_7, I_8		I_{10}, I_{11}
N	N_7		N_{10}
Flag	$flag_6$	$flag_6$	$Flag_6$
Arr1	$arr1_8$		
Arr2			$arr2_{11}$
Prod			$prod_5$, $prod_{11}$
	7	1	7

Figure 14. The FOLV of *sum*, *prod* and their intersection

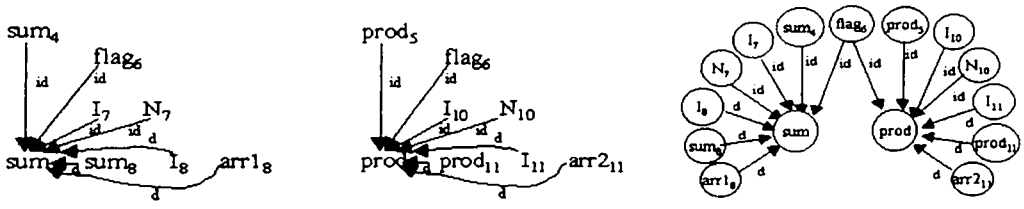


Figure 15. The FOLVDGs of output functions and *logical cohesion* example

According to the definitions of function-oriented cohesion measures, the values of function-oriented cohesion measures of *logical cohesion* example are given as follows:

FOCM(*sum*)=| FOLV(*sum*) | / | LV(*SumAndProd*) |=0.29
FOCM(*prod*)= | FOLV(*prod*) | / | LV(*SumAndProd*) |=0.29
MTFOCM(*SumAndProd*)=|∩FOLV(*ov*) | / | LV(*SumAndProd*) |=0.041
LTFOCM(*SumAndProd*)=|LV – ∪FOLV(*ov*)| / | LV(*SumAndProd*) |=0.46
ATFOCM(*SumAndProd*)=|∪FOLV(*ov*) | / | LV(*SumAndProd*) |=0.64

In this example, there are some common elements that are used to decide which function to be requested logically. The value of MTFOCM of the logical cohesion is equal to 0.041. From the viewpoint of cohesion strength, this *logical cohesion* is just little stronger than *coincidental cohesion*. However, the *logical cohesion* is still a low-end cohesion.

Procedural cohesion is to evaluate whether the module that performs more than one function, and whether the module is related to a general procedural effected by the software [Fenton, N. E. (1991)]. This means that if each function in the module needs to execute following a specific order, then it has the strong procedure cohesion. Explicitly, Both functions are elements of some iteration or decision operations [Lakhotia, A. (1993)]. In the *procedural cohesion* example, the module performs a series of functions related by a sequence of steps. In other words, a module consists of functions related to the procedural processes in a repetition or selection constructs. However, A typical *procedural cohesion* example and its abstract function diagram are given as follows:

```
01 Procedure SumAndProd(N: integer; var sum, prod: integer;arr1,arr2:int_array);
02 var I: integer;
03 begin
04    sum:=0;
05    prod:=1;
06 for I:=1 to N do
07  sum:=sum+arr1[I];
08  prod:=prod*arr2[I];
09 end
```

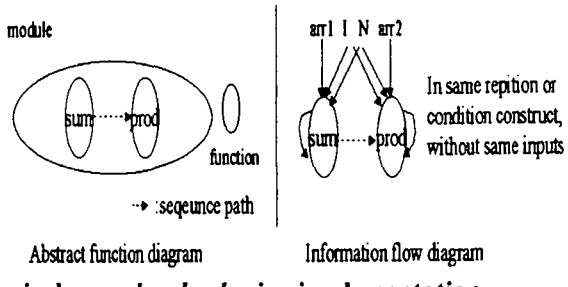


Figure 16. A typical *procedural cohesion* implementation

By definition 1 and definition 2, we know LV(*SumAndProd*)={sum_4,sum_5,$prod_5$,sum_6,$prod_6$,I_6,N_6,sum_7,$arr1_7$,$prod_7$,I_7,$prod_8$,$arr2_8$,I_8} and |LV(*SumAndProd*)|=14. We express LV and LS of the *procedural cohesion* example in Figure 17. And, we may have FOLV(*sum*)={sum_4,I_6,N_6,sum_7,$arr1_7$,I_7} with size of 6, FOLV(*prod*)={$prod_5$,I_6,N_6,$prod_8$,$arr2_8$,I_8} with size of 6, and ∩FOLV(*ov*)={lv ∈ FOLV(*sum*) and lv ∈ FOLV(*prod*)| lv ∈ LV }={I_6,N_6} with size of 2, and it's

visualized FOLS is displayed in Figure 18. In Figure 19, the FOLV of *sum*, *prod* and their intersection are presented. FOLDGs of output functions and *logical cohesion* example are shown in Figure 20.

Line	LV(*SumAndProd*)	Count	LS
4	sum_4	1	
5	sum_5,$prod_5$	2	
6	sum_6,$prod_6$,I_6 ,N_6	4	
7	sum_7,$prod_7$,$arr1_7$,I_7	4	
8	$prod_8$,$arr2_8$,I_8	3	
		14	

Figure 17. The LV and LS of a *procedural cohesion* example

Line	FOLV(*SumAndProd*)	Count	FOLS
4	sum_4	1	
5	$prod_5$	1	
6	I_6 ,N_6	2	
7	sum_7,$arr1_7$,I_7	3	
8	$prod_8$,$arr2_8$,I_8	3	
		10	

Figure 18. The FOLV and FOLS of a *procedural cohesion* example

Variables	FOLV(*sum*)	FOLV(*Sum*) ∩FOLV(*Prod*)	FOLV(*prod*)
sum	sum_4,sum_7		
I	I_6,I_6	I_6	I_6,I_8
N	N_6	N_6	N_6
arr1	$arr1_7$		
arr2			$arr2_8$
prod			$prod_5$,$prod_8$
	6	2	6

Figure 19. The FOLV of *sum*, *prod* and their intersection

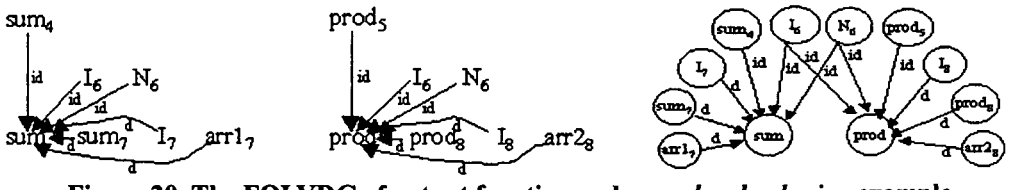


Figure 20. The FOLVDG of output function and *procedural cohesion* example

By the definition of function-oriented cohesion measures, we can derive the values of function-oriented cohesion measures of *coincidental cohesion* example as follows:

FOCM(*sum*)=| FOLV(*sum*) | / | LV(*SumAndProd*) |=0.428
FOCM(*prod*)= | FOLV(*prod*) | / | LV(*SumAndProd*) |=0.428
MTFOCM(*SumAndProd*)=|∩FOLV(*ov*) | / | LV(*SumAndProd*) |=0.142
LTFOCM(*SumAndProd*)=|LV – ∪FOLV(*ov*)| / | LV(*SumAndProd*) |=0.288
ATFOCM(*SumAndProd*)=|∪FOLV(*ov*)| / | LV(*SumAndProd*) |=0.712
In this example, a number of elements are involved in different activities. But the activities are sequential. The value of MTFOCM of the procedural cohesion is equal to 0.142. This value is larger than the value of *logical cohesion*. From the cohesion strength perspective, *procedural cohesion* is a little stronger than *logical cohesion*.

Communicational cohesion is to indicate whether the module performs more than one function, and whether the module is on the same data [Fenton, N. E. (1991)]. This means that if every function in the module operates on

the same data. In the *communicational cohesion* example, the module performs a series of functions related by the same data. In other words, the output functions are used to operate on the same data. However, a typical *communicational cohesion* example and its abstract function diagram are given as follows:

```
01 Procedure SumAndProd(N:integer; var sum, prod: integer;arr1:int_array);
02 var I:integer;
03 begin
04 sum:=0;
05 prod:=1;
06  for I:=1 to N do begin
07  sum:=sum+arr1[I];
08  prod:=prod*arr1[I];
09  end
10 end
```

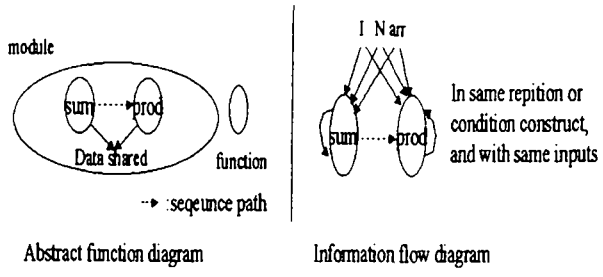


Figure 21. A typical *communicational cohesion* implementation

According to definition 1 and definition 2, we know LV(*SumAndProd*)={sum_4,sum_5,$prod_5$,sum_6,$prod_6$,I_6,N_6,sum_7,$prod_7$,$arr1_7$,I_7,$prod_8$,$arr1_8$,I_8} and |LV(*SumAndProd*)|=14, which are depicted in Figure 22. Again, FOLV(*sum*)={sum_4,I_6,N_6,sum_7,$arr1_7$,I_7} with size of 6, FOLV(*prod*)={$prod_5$,I_6,N_6,$prod_8$,$arr1_8$,I_8} with size of 6 and ∩FOLV(*ov*)={lv ∈ FOLV(*sum*) and lv ∈ FOLV(*prod*)| lv ∈ LV}={I_6,N_6} with size of 2, and it's visualized FOLS is sketched in Figure 23. In Figure 24, the FOLV of *sum*, *prod* and their intersection are illustrated. FOLDGs of output functions and *logical cohesion* example are shown in Figure 25.

Line	LV(*SumAndProd*)	Count	LS
4	sum_4	1	
5	sum_5,$prod_5$	2	
6	sum_6,$prod_6$,I_6,N_6	4	
7	sum_7,$prod_7$,$arr1_7$,I_7	4	
8	$prod_8$,$arr1_8$,I_8	3	
		14	

Figure 22. The LV and LS of a *communicational cohesion* example

Line	FOLV(*SumAndProd*)	Count	FOLS
4	sum_4	0	
5	$prod_5$	1	
6	I_6,N_6	2	
7	sum_7,$arr1_7$,I_7	0	
8	$prod_8$,$arr1_8$,I_8	3	
		6	

Figure 23. The FOLV and FOLS of *communicational cohesion* example

Variables	FOLV(*sum*)	FOLV(*Sum*)∩FOLV(*Prod*)	FOLV(*prod*)
sum	sum_4,sum_7		
I	I_6,I_7	I_6	I_6,I_8
N	N_6	N_6	N_6
arr1	$arr1_7$		$arr1_8$
prod			$prod_5$,$prod_8$
	6	2	6

Figure 24. The FOLV of *sum*, *prod* and their intersection

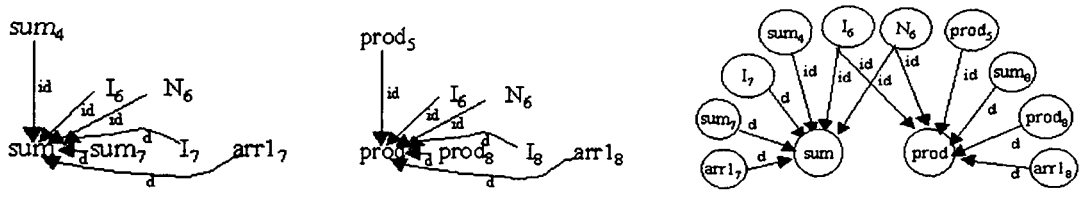


Figure 25. The FOLVDGs of output functions and *communicational cohesion* example

However, the values of function-oriented cohesion measures of *coincidental cohesion* example can be presented as follows:

FOCM(*sum*)=| FOLV(*sum*) | / | LV(*SumAndProd*) |=0.428
FOCM(*prod*)=| FOLV(*prod*) | / | LV(*SumAndProd*) |=0.428
MTFOCM(*SumAndProd*)=|∩FOLV(*ov*)| / | LV(*SumAndProd*) |=0.142
LTFOCM(*SumAndProd*)=|LV − ∪FOLV(*ov*)| / | LV(*SumAndProd*) |=0.288
AVFOCM(*SumAndProd*)=|∪FOLV(*ov*)| / | LV(*SumAndProd*) |=0.712

In this example, a number of elements are involved in different activities, but the activities are sequential. The value of MTFOCM of *communicational cohesion* is equal to 0.142. The value seems to be the same as procedural cohesion. We have *communicational cohesion*, which is not significant stronger cohesion strength than *procedural cohesion*.

Sequential cohesion is to estimate whether the module performs more than one function. Function dependency occurs in an order, which is described in the specification [Fenton, N. E. (1991)]. Generally, the output data from a function is the input for the next function in a module. In the *sequential cohesion* example, the module performs a series of functions related to I/O data. In other words, the functions of a module are related to perform different parts of a sequence of operations, where the output of one function is the input to the next. However, a typical *sequential cohesion* example and its abstract function diagram are given as follows:

```
01 Procedure SumAndProd(N:integer; var sum, prod: integer;arr:int_array);
02   var I: integer;
03   begin
04   sum:=0;
05   prod:=1;
06   for I:=1 to N do begin
07    sum:=sum+arr[I];
08    prod:=prod*sum;
09   end
```

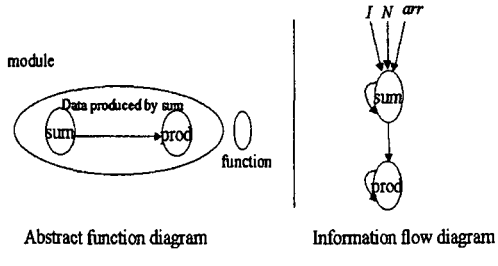


Figure 26. A typical *sequential cohesion* implementation

By the definition of live variables and function oriented live variables, we have LV(*SumAndProd*)={sum_4,sum_5,$prod_5$,sum_6,$prod_6$,I_6,N_6,sum_7,$prod_7$,arr_7,I_7,$prod_8$,sum_8} and |LV(*SumAndProd*)|=13, We express LV and LS of the *sequential cohesion* example in Figure 27. FOLV(*sum*)={sum_4,I_6,N_6,sum_7,arr_7,I_7} with size of 6, FOLV(*prod*)={sum_4,$prod_5$,I_6,N_6,sum_7,I_7,$prod_8$,sum_8} with size of 9 and ∩FOLV(*ov*)={lv ∈ FOLV(*prod*) and lv ∈ FOLV(*prod*)| lv ∈ LV }={sum_4,I_6,N_6,sum_7,arr_7,I_7} and | ∩FOLV(*ov*)|=6, and their visualized FOLS are depicted in Figure 28. In Figure 29, the FOLV of *sum*, *prod* and their intersection are illustrated. FOLDGs of output functions and *logical cohesion* example are detailed in Figure 30.

Line	LV(*SumAndProd*)	Count	LS
4	sum_4	1	
5	sum_5,$prod_5$	2	
6	sum_6,$prod_6$,I_6,N_6	4	
7	sum_7,$prod_7$,arr_7,I_7	4	
8	$prod_8$,sum_8	2	
		13	

Figure 27. The LV and LS of a *sequential cohesion* example

Line	FOLV(*SumAndProd*)	Count	FOLS
4	sum_4	1	
5	$prod_5$	1	
6	I_6,N_6	2	
7	sum_7,arr_7,I_7	3	
8	$prod_8$,sum_8	2	
		9	

Figure 28. The LV and LS of *sequential cohesion* example

Variables	FOLV(*sum*)	FOLV(*Sum*)∩FOLV(*Prod*)	FOLV(*prod*)
sum	sum_4,sum_7	sum_4,sum_7	sum_4,sum_7,sum_8
I	I_6,I_7	I_6,I_7	I_6,I_7
N	N_6	N_6	N_6
arr	arr_7	arr_7	Arr_7
prod			$prod_5$,$prod_8$
	6	6	9

Figure 29. The FOLV of *sum*, *prod* and their intersection

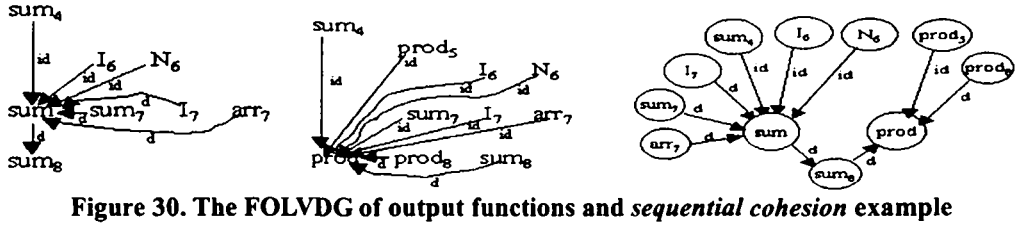


Figure 30. The FOLVDG of output functions and *sequential cohesion* example

FOCM(*sum*)=| FOLV(*sum*) | / | LV(*SumAndProd*) |=0.46
FOCM(*prod*)=| FOLV(*prod*) | / | LV(SumAndProd) |=0.69
MTFOCM(*SumAndProd*)=|∩FOLV(*ov*)| / | LV(*SumAndProd*) |=0.46
LTFOCM(*SumAndProd*)=|LV – ∪FOLV(*ov*)| / | LV(*SumAndProd*) |=0.31
AVFOCM(*SumAndProd*)=|∪FOLV(*ov*)| / | LV(*SumAndProd*) |=0.69

In this example, some elements involved in different activities. But the activities are sequential. The value of MTFOCM of the *sequential cohesion* is equal to 0.46. The values show that *sequential cohesion* is a little stronger than *communicational cohesion*.

Functional cohesion is to check whether the module performs on a single function [Fenton, N. E. (1991)]. The module is the one on which all of the elements contribute to exactly one function. In the *functional cohesion* example, the module achieves exactly one goal. However, a typical *functional cohesion* example and its abstract function diagram are given as follows:

```
01 procedure Sum(N:integer; var sum: integer; arr:int_array);
02 var I : integer;
03 begin
04 sum:=0;
05 for I:=1 to N do begin
06 sum:=sum + arr[I];
07    end
```

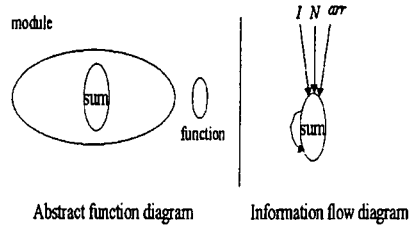


Figure 31. A typical *functional cohesion* implementation

Rely on definition 1 and definition 2, It is easy to know that LV(*sum*)={sum_4,sum_5,I_5,N_6,sum_6,arr_6,I_6} with size of 7 and FOLV(*sum*)={sum_4,I_5,N_5,sum_6,arr_6,I_6} with size of 6. Their visualized LS and FOLS are developed in Figure 32 and Figure 33 respectively. However, there is a single output function in the *functional cohesion* implementation. The live variables, live span, function-oriented live variables and function-oriented live span of the output function is the behalf of the *functional cohesion*. In Figure 34, the FOLV of the single output function *sum* are depicted. FOLDGs of *functional cohesion* example are diagramed in Figure 35.

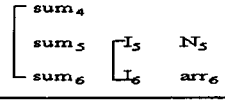
Line	LV(*SumAndProd*)	Count	LS
4	sum_4	1	
5	sum_5,I_5,N_5	3	
6	sum_6,arr_6,I_6	3	
		7	

Figure 32 The LV of a *functional cohesion*

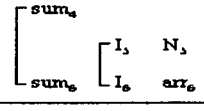
Line	FOLV(*SumAndProd*)	Count	FOLS
4	sum_4	1	
5	I_5,N_5	2	
6	sum_6,arr_6,I_6	3	
		6	

Figure 33. The FOLV of *functional cohesion*

Variables	FOLV(*sum*)
sum	sum_4, sum_6
I	I_5,I_6
N	N_5
arr	arr_6
	6

Figure 34. The FOLV of an output function *sum*

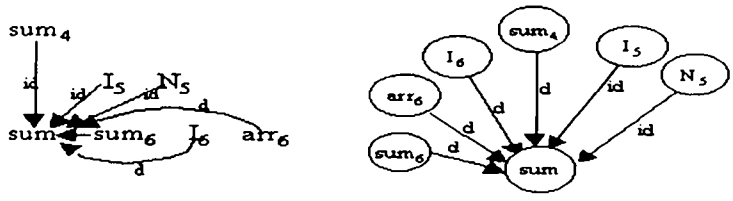


Figure 35. The FOLVDGs of output function and *functional cohesion*

However, the values of function-oriented cohesion measures of *functional cohesion* example are given as follows:

FOCM(*sum*)=| FOLV(sum) | / | LV(sum) |=0.86
MTFOCM(*sum*)=|∩FOLV(*ov*)| / | LV(*sum*) |= 0.86
LTFOCM(*sum*)=|LV(*sum*) – ∪FOLV(*ov*)| / | LV(*sum*) |=0.143
ATFOCM(*sum*)=|∪FOLV(*ov*)| / | LV(*sum*) |=0.857

In this example, all elements involved in a single activity. The value of MTFOCM of the *functional cohesion* is 0.86. *Functional cohesion* is a stronger cohesion strength than *sequential cohesion*. The *Functional cohesion* is the highest cohesion level.

DISCUSSIONS AND RESULTS

Cohesion of a module measures the strength of the relationship between the elements within the module: good modules should be highly bound [Modularity (1999)]. Cohesion can be evaluated on a one-dimensional scale [Stevens, W.P. (1981)][Fenton, N. E. (1991)]. A scale of several points can identify different types of cohesion and giving their associated strength [Modularity (1999)]. According to SMC Cohesion level and Fenton's cohesion spectrum, we know that the cohesion strength spectrum can be illustrated as follows:
In the previous section, we know the values of the cohesion of all experiments are in [0,1]. This also implies that the proposed cohesion measures are well-normalized. In particular, the values of MTFOCM on the six typical cohesion examples from coincidental to functional are increasing between 0 and 1. In the other words, the cohesion strength of coincidental is the weakest type, but the functional cohesion strength is the strongest type. According to

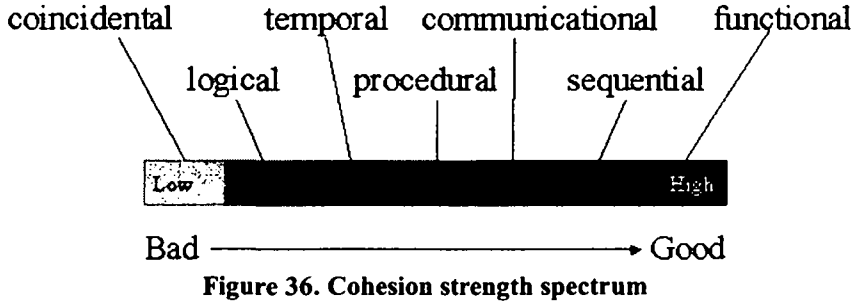


Figure 36. Cohesion strength spectrum

the above experiments, it is not significant difference that the cohesion strengths of *procedure* cohesion and *communication* cohesion. More importantly, MTFOCM does match the SMC Cohesion and Fenton's cohesion strength spectrum. The quantitative values of the MTFOCM experiments the levels of cohesion are sketched in Figure 37.

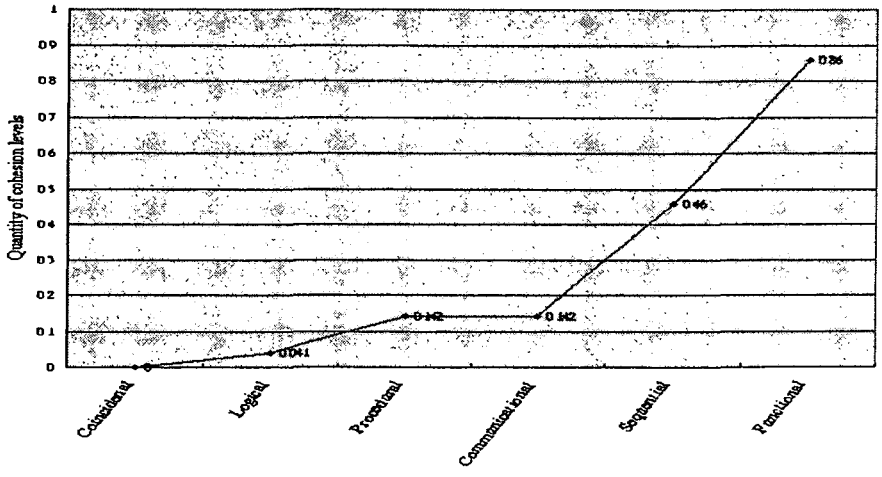


Figure 37. The non-linear curve of cohesion levels

Strictly speaking, coincidental cohesion performs completely unrelated actions, which is the worst level of cohesion. In other words, a coincidental cohesion is an undesirable type of cohesion. Logical cohesion performs function dynamically, which relies on other module calling by passing input arguments. The strength of logical cohesion is a little more than coincidental cohesion. So logical cohesion is still a lower level cohesion. Procedural cohesion is also better than coincidental and logical cohesion because the procedural has more than one function being performed, which is related to one another in order to achieve the goal. But this type of cohesion is still on the lower level of cohesion. In our experiments, the cohesion strength of communication cohesion is not remarkable distinction to procedural cohesion. The cohesion strength of sequential cohesion is better than communicational cohesion because the output from one function is the input to the other functions, which is medium cohesion strength. Functional cohesion performs exactly one action, which is a high-end cohesion. In other words, it is the most desirable type of cohesion.

However, the curve of the quantitative values of MTFCOM in Figure 37 shows that low-end cohesiveness is more worsen than middle-range. The value of MTFOCM of logical cohesion is just 0.041. From coincidental to logical cohesion, the cohesion strength is increasing very slowly. As a result, the difference of cohesion strength between coincidental cohesion and logical cohesion is not too significant. This implies that both coincidental and logical cohesions may be undesirable cohesion levels. The value of MTFOCM of procedural or communication cohesion is 0.142. And, the curve of cohesion strength rises slowly from logical type to procedural or communication type. This means that those procedural and communication cohesion are low-end cohesion types. The value of MTFOCM of sequential cohesion is 27.9% (more than communication). The curve of cohesion strength climbs up sharply at a turn around point of procedural or communication cohesion. This means that sequential cohesion is a high-end cohesion type. The value of MTFOCM of functional cohesion is 27.4% more than sequential. The cohesion strength soars from sequential to functional. Therefore, the functional cohesion should be a desirable cohesion type. Furthermore, The nonlinear scale from coincidental cohesion to functional cohesion consists with the assertions of Pressman and Somerville in their software engineering articles.

According to the empirical experiments, the proposed cohesion measures have some paradigmatic characteristics which are: (1)MTFOCM(sp) ≤FOCM(ov_i) ≤ATFOCM(sp) for each ov_i (2)LTFOCM(sp)+ ATFOCM(sp)=1.0.

From the viewpoint of software quality, reusability, and maintainability, coincidental and logical cohesion are hard to describe the module purpose. But, on reviewing software in test maintenance stage coincidental and logical cohesion may be easy to break into some stronger cohesion modules because it performs multiple unrelated actions. Logical cohesion has more than one action to be intertwined logically, which is not only difficult to understand the interface but also difficult to maintain and reuse. Procedural cohesion performs a series of sequential actions with weak connection. Communicational cohesion is alike procedural cohesion with performing on the same data, which may be little better than procedural cohesion for software quality. The cohesion strength of sequential cohesion is much stronger than communicational or procedural cohesion. Therefore, a sequential cohesion module has a higher software quality better than procedural and communicational cohesion. However, functional cohesion accomplishes a single specific action, which can more readily be reused in a variety of situations. This type of cohesion is easier to understand and maintain. In practice, software engineer may not concern the cohesion categories in a specific procedure. Rather, the cohesion concepts should be realized and low-end cohesion should be avoided when software is designed.

CONCLUSIONS

In this paper, we proposed function-oriented cohesion measures based on the visualized analysis model of function-oriented live variables and live span. We gave a well-designed experiment to examine our proposed function-oriented cohesion measures. The experimental result remarkably meets SMC Cohesion and the Fenton's cohesion spectrum, and matches a nonlinear scale of cohesion. The major contributions of this paper are that we proposed a formal definitions, scientific basis and well-normalized, well-experimented and easy measure algorithmically function-oriented cohesion metrics to improve software quality. Furthermore, It is note worthy that this visualized function-oriented cohesion metrics can be easily incorporated with software CASE tool to help software engineers to enhance software development and quality. However, it may be inevitable to check theoretically the proposed metrics against cohesion properties to make them more precise. For the sake of space limitation, we leave it to future discussion and research.

REFERENCES

Bieman, J. E. and Kang, B.K. (1998), "Measureing Design-Level Cohesion", **IEEE Transactions on Software Engineering**, Feb, Vol. 24. No. 2.

Bieman, J. M. and Ott, L. M. (1994), "Measuring Functional Cohesion", **IEEE Transactions on Software Engineering**, August, Vol. 20. No. 8.

Conte, S. D., Dunsmore H. E. and Shen, V.Y. (1986), **Software Engineering Metrics and Models**, TheBenjamin/Cummings Publishing Company, Inc.

Canfora G. and Lakhotia A.(1999), "Program Comprehension", **Journal of Systems and Software**, 44, pp.79-80.

Curtis B. (1981), **Human Factors in Software Development**, pp. 170-179, Silver Spring, MD: Computer Society Press.

Cohesion, 1999, **WWW** http://www.cs.albany.edu/~mhc/csi518/design/design.html (accessed Aug. 1, 1999)

DeMarco, T. (1982), **Controlling Software Projects**, Prentice Hall, New York.

Fenton, N. E. (1991), **Software Metrics, a Rigorous Approach**, Chapman& Hall, London.

Fifty_Percent, 1998, WWW, http//osiris.sunderland.ac.uk/~cs0hed/campbell/chap~7.html, (accessed May. 10, 1998)

Henry, S. and Kafura, D. (1981), "Software structure metrics based on information flow", **IEEE Transactions on Software Engineering**, Feb, SE-7(5),pp. 510-518.

Intramodule Cohesion, 1999, **WWW** http://www.cs.unc.edu/~stotts/145/cohesion.html (accessed Sep. 7, 1999)

Lucia, A.D and Fasolino, A.R. (1996), "Understanding Function Behaviors through Program Slicing", **Proceedings of 4th IEEE Workshop on Program Comprehension**, Berlin, Germany, March, IEEE Comp. Soc. Press, pp. 9-18.

Lakhotia, A. (1993), "Rule-based approach to computing module cohesion", **in Proceedings of 15th International Conference Software Engineering (ICSE-15)**.

Martin, J. McClure, C. (1983), **Software Maintenance the problem and its solutions**, Prentice-Hall, INC., Englewood Cliffs, New Jersey 07632.

Metr_dis, 1998, WWW http://www.hatteras.com/metr_dis.html (accessed Aug. 10,1999)

Modularity, (1999), WWW http://www.dis.unimelb.edu.au/staff/jacob/lectures/csdl/modularity/tsld013.html (accessed Sep. 7, 1999)

Ott, L. M. and Thuss, J. J. (1989), "The relationship between slices and modules ohesion", **In Processdings of the 12th International Conference on Software Engineering (ICSE-12)**.

Pressman, R.S. (1988), **Software Engineering: A Practitioners Approach**, McGraw-Hill International Editions.

Sommerville, I. (1996), **Software Engineering**, 5th ed. Addison-Wesley, 1996.

Stevens, W., Myers, G. and Constantine, L. (1974), "Structured Design", **IBM Systems Journal**, Vol. 13, No 2, pp.115-139.

Yourdon E. and Constantine L.L. (1979), **Structured Design**, Prentice Hall.