Rule-based Approach to Computing Module Cohesion

Arun Lakhotia
The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
arun@cacs.usl.edu
(318) 231-6766, -5791 (Fax)

Abstract

Stevens, Myers, and Constantine introduced the notion of cohesion, an ordinal scale of seven levels that describes the degree to which the actions performed by a module contribute to a unified function [13]. They provided rules, termed as 'associative principles' to examine the relationships between 'processing elements' of a module and designate a cohesion level to it. Stevens et. al., however, did not give a precise definition for the term 'processing element', thereby leaving it open for interpretations.

This paper interprets the 'output variables' (not statements) of a module as its processing elements. Stevens et. al.'s associative principles are transformed to relate the output variables based on their 'data' and 'control dependence' relationships. What results is a rule-based approach to computing cohesion. Experimental results show that, but for temporal cohesion, the cohesion associated to a module under our reinterpretation and that due to the original definitions are identical for all examples.

1. Introduction

The conceptual complexity of developing a large program is reduced by decomposing the program into smaller modules and developing the modules independently. A program can be decomposed into modules in several ways of which one is chosen during the design process. The choice of the decomposition has critical effect on software quality attributes such as maintainability, reliability, modifiability, testability, etc. of the finished product. Stevens, Myers, and Constantine have proposed that *cohesion* - the functional relatedness of actions performed by individual modules - is a good indicator of the quality attributes of a software system [13]. Their proposition has been accepted by the software community without experimental validation.

One reason why the relation between cohesion and properties of a software product has not been investigated so far is that Stevens et. al. defined cohesion in a subjective manner. As per their definitions, the cohesion of a module is measured by *inspecting* the *association* between all pairs of its *processing elements.* The term processing element was defined as an action performed by a module – such as a statement, procedure call, or "something which must be done in a module but which has not yet been reduced to code" [15]. The informal definition of the cohesion was intentional since Stevens et. al. intended cohesion as a measure to *predict* the properties of modules that would be created from a given design. The measure was to be used by designers as a guide to evaluate various decompositions of a task.

This paper introduces an objective method to compute cohesion of a completely implemented module. The measure, it is hoped, will enable investigations on the effects of cohesion on the quality attributes of a software product. Such investigations are beyond the scope of this paper. One such work in progress is that of [9] aimed at studying the effects of cohesion on the modifiability of a program. This is different from the work presented in [10] investigating the effects of program modifications on a module's cohesion.

The rest of the paper is organized as follows. The next section gives Stevens et. al.'s informal definition of cohesion and the motivations behind our approach. Section 3 defines variable dependence graph, a graph that summarizes the relationships between variables of a module. Section 4 presents our approach to computing module cohesion. Section 5 provides an explanation of our rules for designating cohesion levels and a laboratory validation for the measure. Section 6 describes two other methods of computing cohesion and compares the results of classifying programs using these two approaches, our approach, and Stevens et. al.'s definitions. The conclusions are presented in Section 7.

2. Motivations

In early 70s Constantine attempted to learn why designers associated things into modules. He found that they

Table 1 Associative principle between two processing elements and the corresponding cohesion in increasing order of levels.

Cohesion	*Associative principles*
Coincidental	None of the following associations hold.
Logical	At each invocation, one of them is executed.
Temporal	Both are executed within the same limited period of time during the execution of the system.
Procedural	Both are elements of some iteration or decision operation.
Communicational	Both reference the same input data set and/or produce the same output data set.
Sequential	The output of one serves as input for the other.
Functional	Both contribute to a single specific function.

used certain relationships between a set of actions to determine whether or not they should be performed by the same module. He termed these relationships *associative principles* - principles (properties or characteristics) used by designers to associate actions to be placed in a module. He classified three such principles and arranged them in a linear order (or levels) reflecting the preference of most designers for one principle over another. The ordinal scale defined by the set of associative principles along with their order he termed *cohesion*. Stevens, Myers, Yourdon, and Constantine expanded the list of associative principles to seven, which has now become the *de facto* standard [13, 15].

Table 1 enumerates their list of seven associative principles and the corresponding cohesion level. The associative principles give the cohesion between pairs of processing elements. The cohesion of a module on the whole, as per Stevens et. al., is defined as the lowest of the cohesion between all pairs of processing elements. Figure 1 gives an algorithm to compute cohesion using Stevens et. al.'s method. The informal definition of processing elements and the associative principles make cohesion a subjective metric. The algorithmic description of the steps for computing cohesion suggests that if the definitions of processing elements and the associative principles were formalized, this measure could be made objective. This paper does precisely that. Under our interpretation, the *output variables* of a module are its processing elements. The associative principles of Table 1 are also reinterpreted and formalized using two relations: *control* and *data dependence between variables*, derived from control and data flow analysis.

```
Algorithm: SMC's-compute-module-cohesion
Input: A module or its narrative description
Output: Cohesion of the module
    Identify the set of processing elements of the module
    For every pair of processing elements do
      • identify the set of associative principles in
        Table 1 that suitably define the association
        between the pair
      • the highest level of cohesion corresponding
        to these principles is the cohesion for the
        pair
    The cohesion of a module is the lowest cohesion
    of all pairs of processing elements of the module.
```

Figure 1 The basic steps for computing module cohesion according to Stevens et. al. [13]

The individual levels of cohesion, their ordering, and their relationship to software quality attributes have not yet been validated. It is quite likely that empirical investigation may lead to a reordering, redefinition, addition, or deletion of levels. It is, therefore, desirable that the method for computing cohesion preserve the intent of associative principles as well as be modifiable to accomodate the results of new findings.

In our approach each level of cohesion is defined by translating Stevens et. al.'s associative priniciples into rules of logic. This requires us to associate interpretations to certain terms that were left ambiguous by Stevens et. al. That our rules preserve the intent of the associative principles can be verified by questioning our interpretations and the translation. Similarly, the levels of cohesion correspond to the order of the rules (which themselves are position independent). The levels may be changed simply by reordering the rules.

We are familiar with two other efforts on computing cohesion, first by Ott & Thuss [11] and second by Emerson [5]. The three works (including ours) take totally different approaches; described in detail in Section 6. In the approaches proposed by Ott, Thuss, and Emerson one first computes a number (usually in the range of 0 to 1) using a procedure's text and then assigns a symbolic level based on some threshold values. There is no obvious relationship between the computation procedures of these methods and Stevens et. al.'s definitions. While their measures may be indicative of the strengths of internal bindings of a function, that they measure "cohesion" (a term that so far has taken the definition given by Stevens et. al.) has to be experimentally verified. Similarly, the ordering of the levels in these approach is fixed by the ordering of the threshold values. These methods are, therefore, not conducive to change.

3. Variable dependence graph

We consider modules[1] written in a simple procedural language. The reader is referred to [1] for definitions of terms pertaining to data flow analysis.

A variable dependence graph abstracts the control and data dependences between module variables used in the associative principles for computing cohesion. The control dependence between variables is of two types - *true* and *false*, and is defined in terms of an *if* or a *while* vertex that controls the dependence. Thus a variable y may have *true-* (or *false)* control dependence on a variable x due to a statement n, denoted $x \rightarrow_{c(n,t)} y$ (or $x \rightarrow_{c(n,f)} y$). These two dependences are defined below.

Definition: (*Control dependence*: $x \rightarrow_c (n_1) y$). A variable, say y, has a control dependence on another variable, say x, due to some statement, say n_1, when statement n_1 contains a predicate that uses x and there exists a statement, say n_2, that defines y and the execution of n_2 may be controlled due to the success or failure of predicate in n_2.

This dependence is of type *true* or *false* depending upon which branch of n_1 is n_2 in.

Definition: (*True-control dependence*: $x \rightarrow_{c(n_1,t)} y$). $x \rightarrow_{c(n_1)} y$ and n_1 is either a *while* vertex or an *if* statement with n_2 in its *then* part.

Definition: (*False-control dependence*: $x \rightarrow_{c(n_1,f)} y$). $x \rightarrow_{c(n_1)} y$ and n_1 is an *if* statement with n_2 in its *else* part.

Definition: (*Data dependence: $x \rightarrow_d y$*). Let n_1 and n_2 be statements defining variables x and y, respectively, $x \neq y$. The variable y has *data dependence* on variable x if there is a def-use chain [1] from n_1 to n_2.

Definition: $x \rightarrow y \equiv x \rightarrow_d y \vee (\exists n k . x \rightarrow_{c(n,k)} y)$.

Definition: A *variable dependence graph* (VDG) of a module M, denoted V_M, is the directed graph[2] with typed edges defined as follows:

$\nu(V_M) = Var(M)$ (The set of variables of module M)

$\epsilon(V_M) = \{e$ such that
$e = x \rightarrow_d y$ where $x, y \in \nu(V_M)$ and y has data dependence on x in M, or
$e = x \rightarrow_{c(n,k)} y$ where $x, y \in \nu(V_M)$ and y has control dependence of type k on x due to n in the module $M\}$.

The modules used as examples in the next section are shown with their VDGs.

Canonical naming of module variables

A variable is termed *output* variable for a given procedure if it is modified within it and is either a reference parameter or declared outside the scope of that procedure. Our rules for computing cohesion are based on interdependencies between pairs of output variables. To draw an analogy between variables and actions so as to correctly compute module cohesion it is important that in a module *all definitions of a variable be related to a single purpose.* The meaning of this statement can be stated by the following module which violates it.

```
1: procedure example2;
2:    x := g1(a)
3:    y := f1(x)
4:    x := g2(b)
5:    z := f2(x)
6: end(y,z)
```

If the occurrences of the variable x in statements 2 & 3 are replaced with m, the meaning of the module will not change. We say that the two assignment statements defining the variable x 'do not have the same purpose'.

Definition: A variable has a *single purpose* if all its definitions *reach* the end of the module.

Definition: A module has *canonically named variables,* or is canonically named, if every variable defined in it has a single purpose.

In [8] we give a polynomial time algorithm to translate a module into an 'equivalent' module that is canonically named by selectively renaming some occurrences of its variables. Henceforth we assume that all modules are canonically named.

4. Rules for computing cohesion

In this section we present Stevens et. al.'s definitions (identified as "SMC") of terms [13] along with our interpretations (identified as "AL"). The next section validates the measures resulting from our interpretation against that due to Stevens et. al.'s.

Processing element

SMC: An action performed by a module – such as a statement, procedure call, or "something which must

[1] Our work relies on previous efforts in the areas of flow analysis of programs and computing module cohesion. Terms such as procedure, program, modules, and statements are used in these areas sometimes with different meanings. We call *module* what in PASCAL is termed as a *procedure.*

[2] A directed graph G is a tuple (V, E) such that $E \subseteq V \times V$. We use the notations $\nu(G)$ and $\epsilon(G)$ to refer to V and E elements of the tuple.

Table 2 Rules for computing cohesion between processing elements. Here x, y denote output variables and $a \rightarrow_{tag} b$ is a short hand for $a \rightarrow_{tag} b \in \epsilon(V_M)$.

i	*Cohesion* C_i	*Associative principles or Rules* $rule_i : Var \times Var \rightarrow Boolean$
1.	Coincidental	$rule_1(x,y) = \neg(\bigvee_{i \in \{2 \ldots 5\}} rule_i(x,y))$
2.	Logical	$rule_2(x,y) = \exists znk \forall l.\ z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg k)} y \wedge \neg(z \rightarrow_{c(n,l)} x \wedge z \rightarrow_{c(n,l)} y)$
3.	Procedural	$rule_3(x,y) = \exists znk.\ z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,k)} y$
4.	Communica-tional	$rule_4(x,y) = \exists z. \forall nkl.\ \neg(z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg k)} y) \wedge \neg(z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,k)} y) \wedge ((z \rightarrow x \wedge z \rightarrow y) \vee (x \rightarrow z \wedge y \rightarrow z))$
5.	Sequential	$rule_5(x,y) = x \rightarrow y \vee y \rightarrow x$

be done in a module but which has not been reduced to code".

AL: An *output variable* of a module.

Our choice of output variables (instead of statements) stems from the observation that the functionality of a module is typically defined in terms of its inputs and outputs. This is true for both formal specifications involving pre- and post- conditions and informal documentation provided as headers of each function.

Module cohesion may now be defined as:

SMC: The lowest cohesion of all pairs of its processing elements as determined from rules in Table 1.

Notice that the above definition from Stevens et. al. does not define cohesion for a module that has only 0 or 1 processing elements. We rectify this as follows.

AL: The cohesion of a module is **functional** if it has only 1 output variable; it is **undefined** if it has no output variables; else it is the lowest cohesion of all pairs of the output variables of the module.

The above interpretation of the definition of module cohesion preserves the intent of Stevens et. al.'s definition of functional cohesion: namely, a module has functional cohesion if all its processing elements contribute to a single specific function.

```
Algorithm AL-compute-module-cohesion
Input: a canonically named module P
Output: Module cohesion of P or 'undefined'

  Construct the PDG of P
  Construct the VDG $V_P$
  Let $X$ be the set of all the output variables of P
  If | $X$ |= 0 then cohesion := 'undefined'
      else if | $X$ |= 1 then cohesion := 'functional'
          else begin
              - - initialize to highest value
              cohesion := 'sequential'
              for $x$ in $X$ and $y$ in $Y$ and $x \neq y$ do

                  - - maximum of all cohesions for
                  a variable pair
                  $VC := MAX(\{C_i \mid i \in \{1..5\} \wedge rule_i(x,y)\})$
                  - - minimum of all pairs
                  cohesion := min(cohesion, VC)

              end-for
          end
  return cohesion
  end
```

Figure 2 Algorithm for computing module cohesion using our interpretation of terms. C_i and $rule_i$ refer to the entry in the corresponding columns of the i_{th} row of Table 2.

Our rules for designating cohesion between pairs of output variables are stated in Table 2 as logical expressions $rule_i$, i = 1..5. Given a pair of output variables a and b and the module M if $rule_i(a,b)$ evaluates to *true* then the variables a and b are said to have the corresponding cohesion, C_i. Since the rules are symmetric the order of the variables in the pair is not significant. In the logical expressions, $a \rightarrow_{tag} b$ is a short hand for $a \rightarrow_{tag} b \in \epsilon(V_M)$ where M is the module under consideration.

Figure 2 gives our algorithm for computing module cohesion after rectifying the problem with Stevens et. al.'s algorithm (Figure 1) and introducing our interpretation of processing elements, their associations, and the associative principles.

5. Explanation and validation of rules of Table 2

This section gives a narrative of the intuition behind our rules in Table 2 for computing cohesion between pairs of output variables. Also provided is a validation of the measure: i.e. our rules indeed compute cohesion as defined by Stevens et. al. [13]. Notice that this does not validate the relationship if any between any quality

```
 1 procedure sum(n: integer;
             var result: integer);
 2 var i: integer;
 3 begin
 4   i := 1;
 5   result := 0;
 6   while i < n do begin
 7     i := i + 1;
 8     result := result + i
 9   end
10 end
```

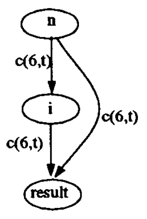

Figure 3 A module computing the sum of first n numbers and its variable dependence graph. The module has functional cohesion.

```
 1 procedure sum_or_product(m,n,flag: integer;
             var sum,prod: integer);
 2 var i,j: integer;
 3 begin
 4   if flag = 1 then begin
 5     i := 1;
 6     sum := 0;
 7     while i <= m do begin
 8       sum := sum + i;
 9       i := i + 1
10     end
11   end
12   else begin
13     j := 1;
14     prod := 1;
15     while j <= n do begin
16       prod := prod * j;
17       j := j + 1
18     end
19   end
20 end
```

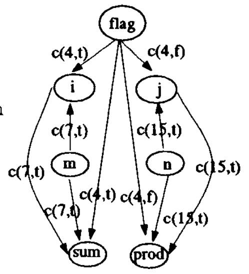

Figure 4 A module computing sum or product of first n numbers and its variable dependence graph. The module has logical cohesion.

attribute of a software system and its cohesion; as stated earlier such a validation is beyond the scope of this paper.

Our validation approach consists of handcrafting a set of programs with "known" cohesion according to Stevens et. al.'s definition [13], computing their cohesion with our approach, and comparing against the "known" value. Similar validation method has been employed by Ott and Thuss to validate their measure for cohesion [12]. The reader is referred to [2] on details of approaches to validating software metric.

Figure 3 contains a module with a single output variable hence expressing functional cohesion. Figures 4 to 8 contain modules with 2 output variables and expressing logical, communicational, procedural, sequential, and coincidental cohesion. Figure 9 contains a module with 3 output variables. Its cohesion is the *minimum* of the cohesions between the pairs of these variables.

```
 1 procedure sum_and_product3(n: integer;
             var sum,prod: integer);
 2 var i: integer;
 3 begin
 4   i := 1;
 5   sum := 0;
 6   prod := 1;
 7   while i <= n do begin
 8     sum := sum + i;
 9     prod := prod * i;
10     i := i + 1
11   end
12 end
```

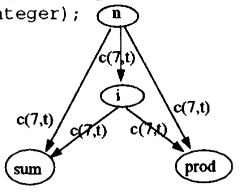

Figure 5 A module computing the sum and product of first n numbers using a single loop. It demonstrates procedural cohesion.

Logical cohesion

SMC: Two processing elements have logical cohesion if at each invocation of the module only one of them is invoked.

AL: Two variables have logical cohesion if they have different type of control dependence on the same variable due to the same node

$rule_2(x,y) = \exists znk.\forall l.$

$$z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg k)} y$$
$$\wedge \neg(z \rightarrow_{c(n,l)} x \wedge z \rightarrow_{c(n,l)} y)$$

The above expression evaluates to true if there exists a variable z such that a) x and y have control dependence on it due to the same vertex n and of different type, k and $\neg k$, i.e. $z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg k)} y$ and b) they do not have a control dependence of the same type on this variable due to the same vertex n. The first condition will hold iff n is an *if* statement and one variable is defined in one branch of this statement and the other variable is defined in its other branch. The condition $\neg(z \rightarrow_{c(n,l)} x \wedge z \rightarrow_{c(n,l)} y$ ensures that both the variables are not defined in the same branch. This ensures the exclusion condition required for logical cohesion, i.e. only one variable be assigned to during each invocation.

The module of Figure 4 computes:

the sum of first 'm' integers if the value of 'flag' is 1 else it computes the product of first 'n' integers.

It has *logical cohesion* since at each invocation only one of two functions are performed. The choice of the function is controlled by the variable *flag*. The module's VDG shows that the output variables *sum* and *prod* have a control dependence on variable *flag* due to statement 4. The dependence is of different type: *true* for *sum* and *false* for *prod*.

Thus $rule_2(sum, prod) = true$, implying that the module's cohesion is $C_2 = logical$.

```
1 procedure sum_and_product2(n: integer;
           var sum,prod: integer);
2 var i,j: integer;
3 begin
4   i := 1;
5   sum := 0;
6   while i <= n do begin
7     sum := sum + i;
8     i := i + 1
9   end;
10  j := 1;
11  prod := 1;
12  while j <= n do begin
13    prod := prod * j;
14    j := j + 1
15  end
16 end
```

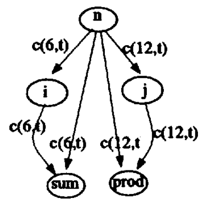

Figure 6 A module computing the sum and product of first n numbers using two loops and its variable dependence graph. The module demonstrates communicational cohesion.

Procedural cohesion

SMC: Two processing elements have procedural cohesion if they belong to the same iteration or decision operation.

AL: Two variables have procedural cohesion if they have control dependence of the same type on the same variable due to the same node.

$rule_3(x,y) = (\exists znk.z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,k)} y)$

The condition will succeed iff there are definitions of both the variables subordinate to the same *while* statement or to the same branch of the *if* statement.

The module in Figure 5 computes:

the sum and product of the first 'n' integers. The two functions are related in that they are performed simultaneously in the same loop.

The module therefore has *procedural cohesion.* Since *sum* and *prod* are computed in the same loop they have control dependence on all variables in the loop predicate. The dependence is due to the same statement and is of the same type, in this case *true.*

Thus $rule_3(sum, prod) = true$ and the module cohesion is C_3 which is *procedural.*

Communicational cohesion

SMC: Two processing elements have communicational cohesion if they reference the same input data and/or produce the same output data.

AL: Two variables have communicational cohesion if

- they have data dependence on the same variable, or
- the same variable has data dependence on them, or
- one variable has control dependence and the other has data dependence on the same variable
- the two variables have control dependence on the same variable but due to different vertices.

In other words, two variables have communicational cohesion if they have relation with each other or a common variable and this relation is not captured by logical or procedural cohesion.

$$\begin{aligned} rule_4(x,y) &= \exists z.\forall nkl. \\ &\quad \neg(z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg k)} y) \\ &\quad \wedge\neg(z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,k)} y) \\ &\quad \wedge(\ (z \rightarrow x \wedge z \rightarrow y) \\ &\quad\quad \vee(x \rightarrow z \wedge y \rightarrow z)) \end{aligned}$$

The first two negations above ensure that x and y do not depend on z such that it may lead to logical or procedural cohesion. The next condition then ensures that x and y have some relation to the common variable z.

The module of Figure 6 computes:

the sum and product of first 'n' integers. The two functions are computed independently.

The two functions are related in that they both depend on the value of the variable n. Hence the module has *communicational cohesion.* In the modules VDG the variables *sum* and *prod* have control dependence on n, but the dependence is not due to the same statement. Thus $rule_4(sum, prod) = true$ and the module cohesion is $C_4 = communicational.$

```
1 procedure sum_and_average(n:integer;
           var sum,average: integer);
2 var i: integer;
3 begin
4   i := 1;
5   sum := 0;
6   while i <= n do begin
7     sum := sum + i;
8     i := i + 1
9   end;
10  average := sum / n
11 end
```

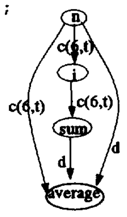

Figure 7 A module computing the sum and average of first n numbers. It demonstrates sequential cohesion.

Sequential cohesion

SMC: Two processing elements have sequential cohesion if the output of one serves as an input to the other.

AL': Two variables have sequential cohesion if one has data dependence on the other.

$rule_5(x,y) = (x \rightarrow_d y \vee y \rightarrow_d x)$

```
1 procedure sum_and_product1(m,n: integer;
          var sum,prod: integer);
2 var i,j: integer;
3 begin
4   i := 1;
5   sum := 0;
6   while i <= m do begin
7     sum := sum + i;
8     i := i + 1
9   end;
10  j := 1;
11  prod := 1;
12  while j <= n do begin
13    prod := prod * j;
14    j := j + 1
15  end
16 end
```

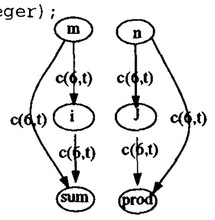

Figure 8 **A module computing the sum of first m numbers and product of first n numbers. It demonstrates coincidental cohesion.**

```
1 procedure sum_sumsquares_product2(
          m,n,flag: integer;
          var sum,sumsquares,prod: integer);
2 var i,j: integer;
3 begin
4   if flag = 1 then begin
5     i := 1;
6     sum := 0;
7     while i <= m do begin
8       sum := sum + i;
9       i := i + 1
10    end;
11  end
12  else begin
13    j := 1;
14    sumsquares := 0;
15    prod := 1;
16    while j <= n do begin
17      sumsquares := sumsquares + j * j;
18      prod := prod • j;
19      j := j + 1
20    end;
21  end
22 end
```

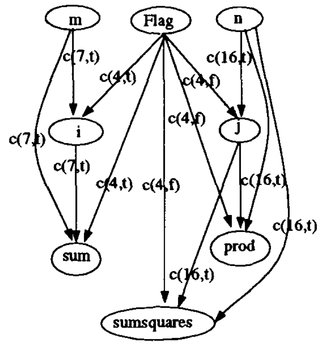

Figure 9 A module containing three output variables with different types of cohesions between pairs of these variables.

The above expression captures the intent of Stevens et. al.'s definition of sequential cohesion. Notice that, just as in communicational cohesion, this definition does not state anything about x having control dependence on y or *vice versa.* Using argument similar to that in communicational cohesion we believe that this expression should also be generalized as follows:

$$rule_5(x, y) = (x \rightarrow y \vee y \rightarrow x)$$

The module of Figure 7 computes:

sum and average of the first n integers.

Since the computation of average uses the result from computing sum, the two functions have *sequential cohesion.* The dependence between the variables, *sum* and *average*, corresponding to these functions exhibits this. The variable *average* has a data dependence on the variable *sum* hence $rule_5(sum, average)$ is *true* and the module cohesion is *sequential.*

Coincidental cohesion

SMC: Two processing elements that do not have logical, temporal, procedural, communicational, sequential, or functional cohesion have coincidental cohesion.

AL: Two variables that do not have logical, procedural, communicational, or sequential cohesion have coincidental cohesion.

$$rule_1(x, y) = \neg(\bigvee_{\forall i, i \in \{2\ \ldots\ 5\}} rule_i xy)$$

The module of Figure 8 demonstrates *coincidental cohesion.* It computes:

the sum of all numbers between 1 *and* m *and the product of all numbers between* 1 *and* n.

The two functions it performs are independent and it has two independent loops performing the tasks. This is reflected in its VDG. The graph consists of two disjoint subgraphs. The two output variables *sum* and *prod* do not depend on each other, nor do they depend on the same variable. Thus $rule_1(sum, prod)$ is *true* and the module has *coincidental* cohesion.

The examples so far, except for functional cohesion, had two output variables. Our next example looks at a program with three output variables.

Consider the module in Figure 9. It takes input $flag$ and n. If the value of $flag$ is n it computes the sum of first n positive integers otherwise it simultaneously computes the sum of squares of the first n integers and also their product. The module performs three functions: compute sum, compute sum of squares, compute product. It performs only the first function on certain invocation and both the remaining functions in other invocations. It clearly has *logical* cohesion between the processing elements of the first function and the other two functions. The processing

Table 3 Comparison of cohesions associated to Modules 1 through 7 by the different approaches discussed in this paper. Emerson and Ott & Thuss reclassifiy the cohesions as subsets of the original categories. The reclassification is as follows: Type I ={functional, sequential, communicational}; Type II = {procedural, temporal}; Type III = {logical, coincidental}; Low = {coincidental, temporal}, Control = {procedural, logical} High = {sequential, functional}. A (*) indicates that the cohesion assigned to that module differs with that of Stevens et. al.'s assignment.
A ? indicates that that the classification for that module is ambiguous or not well defined.

Approach → Module ↓	Stevens et. al.'s	Lakhotia & Nandigam's	Emerson's	Ott & Thuss'
Module 1	functional	functional	Type I	High
Module 2	logical	logical	Type III	Control
Module 3	communicational	communicational	Type II (*)	Low (*)
Module 4	procedural	procedural	Type I (*)	Control
Module 5	sequential	sequential	Type II (*)	High
Module 6	coincidental	coincidental	Type II (*)	Low
Module 7	logical	logical	Type III	?

elements of the latter two functions have *procedural* cohesion, because they are performed simultaneously. The cohesion of the module is the smaller of *logical* and *procedural*, i.e. *logical*.

This is computed using our rules as follows.

$$rule_2(sum, sumsquares) = true,$$
$$rule_2(sum, prod) = true, \text{ and}$$
$$rule_3(sumsquares, prod) = true.$$

Hence the set of cohesions between the pairs of output variables is $\{logical, procedural\}$. The minimum element of this set is $logical$.

6. Comparison with related works

Section 2 eluded to Ott & Thuss' [11] and Emerson's [5] methods of computing module cohesion. It stated the "philosophical" difference between the other methods and ours. In this section we describe these methods and compare the classifications done by them. We believe that Emerson's approach is flawed in so far as computation of cohesion as defined by Stevens et. al. [13] is concerned. The reasons are explained below.

The result of classifying the modules from the previous section using Stevens et. al's, Ott & Thuss', Emerson's and our approach are shown in Table 3. In order to compare the classifications done by different methods, the seven levels of cohesion of Stevens et. al. are mapped to four levels of Ott & Thuss and three levels of Emerson. A (*) in Table 3 indicates those modules whose cohesion differs from that of Stevens et. al.'s after such reclassification. The cohesion assigned to the various modules using our method is consistent with that of Stevens et. al.'s assignment. The same is not true for the other two methods.

In the following subsections we give the essence of Ott & Thuss' and Emerson's approaches and reason why the cohesion assigned by these methods differ from Stevens et. al.'s definitions.

Ott and Thuss's Approach Ott & Thuss reclassify the original seven levels of cohesion into four categories: *low = {coincidental, temporal}*, *control = {logical, procedural}*, *data = {communicational}* and *high = {sequential, functional}*. They determine the "relationships between processing elements" of a program by examining statements in the intersection of its *end-slices* of output variables. An end-slice of a variable is a slice[3] performed at the last statement with respect to that variable. An *output variable* of a module, as per their definition, is a variable declared as reference parameter or a variable defined by the operating system.

The associative principles of Ott & Thuss 'relate the sets of statements in the end-slices of a pair of output variables'. These principles and their corresponding cohesions are summarized in Table 4. The associative principles relate to the set of statements in the intersection of the end-slices. Ott & Thuss restrict that only "variant referent executable statements" - executable statements that refer to variables be considered when comparing slices.

Comparison of the cohesion assignment due to Ott & Thuss' approach with that due to Stevens et. al. brings

[3] A *slice* of a program at a statement i with respect a variable v consists of all statements of the program that may effect the value of v at statement i. For more details see Weiser [14]. We omit details for the sake of brevity. The term *end-slice* is our term. It is not used by Ott & Thuss.

Table 4 Summary of Ott & Thuss's associative principles for computing module cohesion. The principles compare the intersection of end-slices of output variables. Only "variable referent executable statements" are considered as part of the end-slices.

Cohesion C_i	*Ott & Thuss' Associative principles* AP_i
low	the intersection is empty
control	the intersection primarily contains control statements and definitions for the control variables
data	the intersection contains non-control variable data definitions
high	one slice is totally contained in another

out its weakness. Modules that perform different functions using the same input data, as in module of Figure 5, are classified by Ott & Thuss as *low* because the intersection of the end-slices of their output variables is empty. Stevens et. al. assign communicational cohesion to such programs (which is reclassified by Ott & Thuss as *data* cohesion). Thus Ott & Thuss' assignment is not consistent with Stevens et. al.

Emerson's Approach Emerson reclassifies the seven levels of cohesion into three: *Type I = {functional, sequential, communicational}, Type II = {procedural, temporal},* and *Type III = {logical, coincidental}.* He represents a program as a flow graph [6][4] and constructs a reference set for each variable - the set of vertices that refer to that variable in the flow graph. If R_i is the set of vertices in a flow graph F that reference variable i then he defines a metric $\kappa(R_i, F)$ - cohesion of R_i in flow graph F as follows:

$$k(R_i, F) = \frac{|R_i| dim R_i}{|\nu(F)| dim\ \nu(F)}$$

where *dim A* is the number of "maximal linearly independent paths" [4] passing through the vertex set A in the flow graph F.

He then defines the cohesion of a module with flow graph F, $\kappa(F)$, as the arithmetic mean of $\kappa(R_i, F)$ for all variables i. We call this measure *graph cohesion.* Emerson uses the value of $\kappa(F)$ as a *discriminant,* that is he associates a range of values of the metric for each of the three types of cohesion:

1. Type I - $0 \leq \kappa(F) \leq (q/c)/s$
2. Type II - $(q/c)/s \leq \kappa(F) \leq 1/s^2$
3. Type III - $1/s^2 \leq \kappa(F) \leq 1$

[4] Actually Emerson operates on reduced flow graph – a graph derived after performing some transformation on a programs flow graph. The details of the representation are not relevant for our discussion.

where s is the number of executable statements that refer to variables, q is average number of variable references per executable statement, and c is the ratio of number of variables in a module and number of executable statements. The constants q and c are language specific parameters supposed to be computed from a domain of sample programs. A module is classified to have a type of cohesion if the value of the cohesion metric falls in the associated range.

Table 3 shows that the cohesion associated to a module using Emerson's approach is not consistent with that associated using Stevens et. al.'s definition (after augmenting it to reclassify the seven levels of cohesion into Emerson's Type I, II, and III). The source of the problem can be traced to the assumptions that Emerson makes to derive the above ranges.

In order to derive cut-off ranges Emerson models the flow graph of Type II and III modules using graph constructs. He models a Type II module as those modules whose flowgraph may be constructed by a 'sequence' of simpler flow graphs. Similarly Type III modules are modules whose flowgraph may be modelled as a set of simpler flowgraphs connected in parallel such that only one of the them may be selected for execution. Although these models capture the structure of all modules with Type II and Type III cohesions, respectively, not all modules modelled by these constructs have Type II or Type III cohesions. This leads to the incorrect classifications. For example the module in Figure 8 with coincidental cohesion is classified as Type II because its flow graph can be modelled as a sequence of two simpler flow graphs. Similarly a module in which all simpler flow graphs controlled by a selection compute values for the same variable will be incorrectly classified as Type III.

7. Conclusions

Module cohesion was introduced by Stevens, Myers, and Constantine [13] as a property that describes the degree to which actions performed by a module contribute to a unified function. This is an ordinal metric with seven levels measured in terms of the *type of associations* between pairs of *processing elements* of a module. Stevens et. al. gave descriptive definitions of the notion of processing elements and the rules for designating a cohesion level to a module. This left the definitions open to interpretations and made cohesion a subjective measure [3].

This paper gives a formal definition for the term "processing elements" and the rules for measuring the type of associations. A processing element as per Stevens et. al.'s intent is a functionality provided by a module. While the statements of a module implement a modules behavior, in the domain of procedural programs, the actions performed by a module, except those related over time, are reflected in the change of its variables' state and/or the state of its input and output streams. This leads us to define the output variables of a module as its processing elements. The association between the output variables is defined in terms of control and data dependence between variables which in turn are derived from similar dependences between statements. The rules for designating a cohesion level are defined such that they preserve the intent of the original definitions in the context of the new definition of processing elements.

We have validated our measure for cohesion in two parts. In the first part we created sample modules that depicted a particular cohesion according to Stevens et. al.'s definitions and compared their cohesions to that assigned by our method. The examples in this paper show a subset of these modules. As is shown in Table 3, but for temporal cohesion, the cohesion associated to a module using our interpretation and that due the Stevens et. al.'s definitions are indentical in *all* cases.

In the second part of our experiment we took sample programs from books, such as Kernighan and Plaugher [7], and computed their cohesion using the two approaches. It turns out that textbook programs tend to be have functional cohesion since they are written with the intention of teaching good programming style. These programs therefore do not provide a good sample set for testing algorithms for measuring program quality. Similar inference has previously been made by Emerson [5].

Our objective definition of computing cohesion should enable investigations of the effect of module cohesion on various software quality attributes. Work is in progress to study the effect of module cohesion on software modifiablility [9].

Acknowledgments: The author collaborated with Jagadeesh Nandigam on the earlier stages of this work [8] and thanks him for his comments and contribution. This work was supported in part by the grant LEQSF (1991–92) ENH-98 from the Louisiana Board of Regents.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty. A philosophy for software measurement. *J Systems Software,* 12:277–281, 1990.

[3] V. R. Basili. Evaluating software development characteristics: Assessment of software measures in the software engineering laboratory. In *Proceedings, 6th Software Engineering Workshop, NASA/Goddard Space Flight Center,* 1977.

[4] N. Biggs. *Algebraic Graph Theory.* Cambridge University Press, 1974.

[5] T. J. Emerson. A discriminant metric for module cohesion. In *Proceedings of the 7th International Conference on Software Engineering,* Mar. 1984.

[6] M. S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, New York, 1977.

[7] B. W. Kernighan and P. Plaugher. *Software Tools.* Addison-Wesley Publishing Company, 1976.

[8] A. Lakhotia and J. Nandigam. Computing module cohesion. Technical Report CACS-TR-91-5-5, University of Southwestern Louisiana, July 1991.

[9] J. Nandigam. An empirical study of the effects of module cohesion on program modifiability. University of Southwestern Louisiana, 1993.

[10] L. M. Ott and J. M. Bieman. Effects of software changes on module cohesion. In *Proceedings of the Conference on Software Maintenance,* pages 345–353. IEEE Computer Society Press, 1992.

[11] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 12th International Conference on Software Engineering,* May 1989.

[12] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 12th International Conference on Software Engineering,* May 1989.

[13] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal,* 13(2):115–139, 1974.

[14] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.,* 10(4):352–357, 1984.

[15] E. Yourdon and L. L. Constantine. *Structured Design.* Yourdon Press, 1978.