Computer Science
Technical Report

Colorado State University

Using Design Cohesion to Visualize, Quantify, and Restructure Software

Byung-Kyoo Kang
kang@cs.colostate.edu

James M. Bieman
bieman@cs.colostate.edu

January 22, 1996
Submitted for Publication

Technical Report CS-96-103

Research partially supported by NASA Langley Research Center grant NAG1-1461.

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Using Design Cohesion to Visualize, Quantify, and Restructure Software

Byung-Kyoo Kang James M. Bieman

January 22, 1996
Technical Report CS-96-103
Submitted for Publication

Abstract

During design or maintenance, software developers often use intuition, rather than an objective set of criteria, to determine or recapture the design structure of a software system. A decision process based on intuition alone can miss alternative design options that are easier to implement, test, maintain, and reuse. The concept of design-level cohesion can provide both visual and quantitative guidance for comparing alternative software designs. The visual support can supplement human intuition; an ordinal design-level cohesion measure provides objective criteria for comparing alternative design structures. The process for visualizing and quantifying design-level cohesion can be readily automated and can be used to re-engineer software.

Index terms — cohesion, software design, software maintenance, software visualization, software measurement and metrics, software restructuring and re-engineering, software reuse, measurement theory.

1 Introduction

Poorly structured software designs can result in systems that are difficult to test, upgrade, maintain, and reuse, and are unreliable. Thus, the life cycle costs of poorly designed software systems can be much higher than that of well designed systems. An inferior design can be due to inadequate choices during the initial design of a system, or can be a natural result of software evolution.

Objective criteria for evaluating design alternatives are needed. Many existing criteria are applicable to implementations, not designs. Examples of objective criteria for evaluating code structure include principles of structured programming, the cyclomatic number [11], functional cohesion [4], and many others. The principles of information hiding and data abstraction provide guidance for structuring a design, but do not give objective means for comparing alternative structures. Function points are used to predict the expected size of an implementation rather than to evaluate design structure [1]. The object-oriented design measures proposed by Chidamber and Kemerer provide a mechanism to gather quantitative information about classes in object-oriented software, but they do not provide guidance to help evaluate design alternatives [5]. Gamma et al describe a set of structural design patterns for object-oriented software and objective, but not quantitative, criteria for choosing a particular pattern [8].

Visual displays of software designs and ordinal measures of design attributes are potential tools to identify and evaluate design alternatives. A visual display of a design structure will increase the accuracy of decisions

Research partially supported by NASA Langley Research Center grant NAG1-1461.

Address correspondence to: J. Bieman, Computer Science Dept, Colorado State Univ., bieman@cs.colostate.edu, Fort Collins, CO 80523. (970)491-7096, Fax: (970) 491-2466.

based on intuition. Measures that provide objective, quantitative characterizations of a design add further insight, and can potentially be used in an automated structuring system.

Design visualization and measurement tools can help in developing an initial design, and they can be used to re-engineer existing software. The most difficult software to re-engineer is legacy software, which often has no available design documentation. To re-engineer such software we need to recapture the design structure from the implementation. Software visualization tools can certainly help here. After the design is recaptured, the system can be restructured.

Our objective is to create design visualization and measurement tools that can be applied to design-level entities. These tools should support the visualization and quantitative evaluation of design structure, and be useful in restructuring a software design.

In the remainder of this paper we show that the concept of design-level cohesion can be used to visualize, quantify, and restructure software. The term "software cohesion," which was introduced more than 20 years ago [12], refers to the relatedness of module components. A highly cohesive software module is a module whose components are tightly coupled. Cohesive modules are difficult to split into separate components. Thus, the degree of cohesiveness should be an attribute that is useful for evaluating the structure of modules.

A clear understanding of an attribute like design-level cohesion is required before the attribute can be measured in a meaningful way [6, 7]. A model that captures the essence of the attribute is also needed [2]. A design model that can help make design attributes visible can be exceptionally valuable.

2 A Model for Visualizing Software Designs

An input-output dependence graph (IODG) can model a design-level view of a module. The model is based on the data and control dependence relationships between input and output components of a module.

Input components of a module include in-parameters and referenced global variables. Output components include out-parameters, modified global variables, and 'function return' values. An in-out-parameter becomes two components, an input component and an output component. The term 'component' refers to a static entity. An array, a linked list, a record, or a file is one component rather than a group of components. We define the data and control dependence informally; their formal definitions are given in compiler texts, for example, see reference [14].

Definition: A variable y has a *data dependence* on another variable x ($x \xrightarrow{d} y$) if x 'reaches' y through a path consisting of a 'definition-use' and 'use-definition' chain. (Here by data dependence, we mean the 'true dependence' determined by examining the data flow of the static components.) A typical case of data dependence between two variables is that a variable is used to compute the other through a sequence of assignment statements.

Definition: A variable y has a *control dependence* on another variable x if the value of x determines whether or not the statement containing y will be performed.

Definition: A variable y is *dependent* on another variable x ($x \rightarrow y$) when there is a path from x to y through a sequence of data or control dependence. We call the path a *dependence path*.

Definition: A variable y has *condition-control dependence* on another variable x ($x \xrightarrow{cc} y$) if y has control dependence on x, and x is used in the predicate of a decision (i.e., if-than-else) structure. For example, all variables in the 'then' and 'else' bodies of an 'if' statement are condition-control dependent on variables used in the predicate of the decision.

Definition: A variable y has *iteration-control dependence* on another variable x ($x \xrightarrow{ic} y$) if y has control dependence on x, and x is used in the predicate of an iteration structure. For example, all variables in a 'while' body are iteration-control dependent on variables used in the loop predicate.

Definition: A variable y has *c-control dependence* on another variable x ($x \xrightarrow{c} y$) if the dependence path between x and y contains a decision-control dependence. For example, for (1) $x \xrightarrow{cc} y$, (2) $x \xrightarrow{d} a \xrightarrow{cc} b \xrightarrow{d} y$, and (3) $x \xrightarrow{cc} a \xrightarrow{ic} b \xrightarrow{d} y$, y has c-control dependence on x. The c-control dependence between an input and an output variable means that the output value is controlled by the input value through decision structure.

Definition: A variable y has *i-control dependence* on another variable x ($x \xrightarrow{i} y$) if the dependence path between x and y contains an iteration-control dependence but no condition-control dependence. For example,

for (1) $x \xrightarrow{ic} y$ and (2) $x \xrightarrow{d} a \xrightarrow{ic} b \xrightarrow{d} y$, y has i-control dependence on x. When an output has i-control dependence on an input, the output value is affected by the execution of a iteration process whose execution count is affected directly or indirectly by the input.

In our model, a dependence between an input and an output of module is either data, c-control, or i-control dependence.

IODG Definition. The *input-output dependence graph* (IODG) of a module M is a directed graph, $G_M = (V, E)$ where V is a set of input-output components of M, and E is a set of edges labeled with dependence types such that $E = \{(x, y) \in V \times V \mid$ y has data, c-control, and/or i-control dependence on x$\}$

The graph contains the information how input-output components are related. Each input contributes to one or more outputs; they are used to compute output(s), as input data, decision invariant, and/or loop invariant. The dependence between components can be determined by data flow analysis using a compiler-like tool when an implementation is available. Without an implementation, a designer must specify the dependencies between input and output components. Such a specification is a key component of a detailed design. An IODG can be readily displayed visually as shown in Figure 1.

The caller-callee relationship is represented by including the input-output dependence relationship of the callee in the corresponding place of the I/O dependence diagram of the caller. In such a digram, an input is represented by a circle, and an output by a square. The texts in each circle and square are the names of input and output variables. Each arrow indicates the dependence between two components.

Figure 1 shows two IODG's, one for procedure *Asum_Hsum* and another for procedure *Fibo_Amean_Hmean*. *Fibo_Amean_Hmean* generates an array of n Fibonacci numbers and computes the arithmetic mean and harmonic mean of the numbers by calling procedure *Asum_Hsum*.

The IODG of *Fibo_Amean_Hmean* shows the caller-callee relationship: *Asum_Hsum* is called by procedure *Fibo_Amean_Hmean*. The call relationship is represented by the callee's IODG within the rounded square in the IODG of the caller. Each dependence relation between the caller and the callee is represented by an arrow with a dependence type(s). If a callee contains a function call, the dependence information of the callee are included in the caller. So, the IODG of the callee must be determined before generating the caller's IODG.

The extended IODG contains the complete dependence paths between inputs and outputs of a module. Thus, we can determine exact dependence relationships between input/output components. For example, consider input n and output *amean* of the IODG of *Fibo_Amean_Hmean*. We find three dependence paths between them: (1) $n \xrightarrow{d}$ *amean*, (2) $n \xrightarrow{d}$ input_parameter $\xrightarrow{i}$ output_parameter $\xrightarrow{d}$ *amean*, and (3) $n \xrightarrow{i}$ *fib_arr* $\xrightarrow{d}$ input_parameter $\xrightarrow{d}$ output_parameter $\xrightarrow{d}$ *amean*. According to our dependence definitions, *amean* has data and i-control dependencies on n.

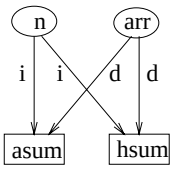
To simplify the representation, the arrow in the IODG indicates only a direct dependence between input and output. An indirect dependence is implied through a sequence of direct dependences. The IODG of *Fibo_Amean_Hmean* shows that the direct and indirect dependence relationship between input n and output *amean* and *hmean*.

The IODG shows the relationship between input and output components of a module. In its graphical form, the IODG visually displays the functional structure of the module. This representation is used to define a design-level cohesion measure and is applied to the problem of restructuring software at design and maintenance stages.

3 Measuring Design Cohesion

Software cohesion, as described by Stevens, Myers, and Constantine (SMC Cohesion) [12], provides an intuitive mechanism for assessing the relatedness of the components in an individual module. It can be used to determine whether the components of a module actually belong together. After describing SMC Cohesion, we show that SMC Cohesion can be applied directly to the IODG representation of a module to evaluate the design-level cohesiveness of the module. We use the ordering imparted by SMC Cohesion on the set of all IODG's as an empirical relation system to show that our own automatable design-level cohesion measure

```
procedure Asum_Hsum
   ( n : integer;
     arr : int_array;
     var asum : integer;
     var hsum : float );
var i : integer;
begin
   asum := 0;
   hsum := 0;
   for i := 1 to n do begin
      asum := asum + arr[i];
      hsum := hsum + 1.0/arr[i];
   end;
end;
```




```
procedure Fibo_Amean_Hmean
   ( n : integer;
     var fib_arr : int_array;
     var amean : integer;
     var hmean : float );
var asum, hsum : int_array;
     i : integer;
begin
   fib_arr[1] := 1;
   fib_arr[2] := 2;
   for i := 3 to n
      fib_arr[i] := fib_arr[i-1] + fib_arr[i-2];
   Asum_Hsum(n, fib_arr, asum, hsum);
   amean := asum / n;
   hmean := n / hsum;
end;
```

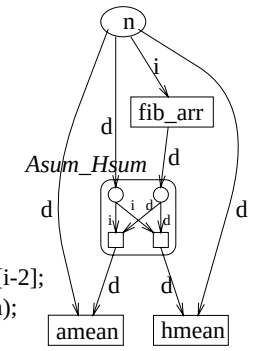


Figure 1: Input-output dependence graph representation for *Asum_Hsum* and *Fibo_Amean_Hmean*.

(DLC) satisfies the representation condition of measurement [6, 7]. That is, we show that the DLC measure is consistent with the intuition provided by SMC Cohesion.

3.1 SMC Cohesion as an Empirical Relation System

Stevens, Myers and Constantine defined seven levels of cohesion on an ordinal scale [12]. The SMC Cohesion of a module is determined by inspecting the association between all pairs of its processing elements. The purpose of SMC Cohesion is to predict properties of implementations that will be created from a given design, so a *processing element* is a module behavior that may not yet be reduced to code. SMC Cohesion is based on seven distinct associative principles between each pair of processing elements in a module. These seven levels are listed in order of increasing strength of association:

1. Coincidental association: there is no relationship between the processing elements.
2. Logical association: both processing elements belong to the same logical class of related functions.
3. Temporal association: each occurrence of both processing elements occurs within the same limited period of time during execution.
4. Procedural association: both processing elements are elements of a common procedural unit which is an iteration or decision process.
5. Communicational association: both processing elements operate upon the same input data set and/or produce the same output data.
6. Sequential association: the output data from one processing element is input to the other processing element.
7. Functional association: both processing elements are essential to the performance of a single function.

When a pair of processing elements exhibit more than one cohesion level, the cohesion for the pair is their highest association level. When a module contains more than one pair of processing elements, the module's cohesion is the lowest association level of all pairs.

Because of its intuitive nature, the assessment of SMC Cohesion requires the judgment of human raters. As a result, SMC Cohesion cannot be readily applied to measure cohesion in practice [13].

Though not a measure, SMC Cohesion defines an intuitive notion of the cohesion attribute of design components. Since SMC Cohesion also imparts an ordering on design components, we can use it as an empirical relation system to help us to define a quantitative cohesion measure that can be readily automated.

3.2 A Design-Level Cohesion (DLC) Measure

The DLC measure is derived from the design-level view of module, modeled by the IODG. In deriving the DLC measure, we follow the approach used to develop SMC Cohesion. We define six relations between a pair of output components based on the IODG representation. The corresponding cohesion level is based on six relations:

1. **Coincidental relation (R_1):**
   $R_1(o_1, o_2) = \neg(o_1 \rightarrow o_2) \wedge \neg(o_2 \rightarrow o_1) \wedge \neg\exists x\, [(x \rightarrow o_1) \wedge (x \rightarrow o_2)]$
   Two outputs o_1 and o_2 of a module have neither dependence relationship with each other, nor dependence on a common input.

2. **Conditional relation (R_2):**
   $R_1(o_1, o_2) = \exists x\, [((x \xrightarrow{c} o_1) \wedge (x \xrightarrow{c} o_2)) \vee ((x \xrightarrow{c} o_1) \wedge (x \xrightarrow{i} o_2)) \vee ((x \xrightarrow{i} o_1) \wedge (x \xrightarrow{c} o_2))]$
   Two outputs are c-control dependent on a common input, or one of two outputs has c-control dependence on the input and the other has i-control dependence on the input.

3. **Iterative relation (R_3):**
   $R_1(o_1, o_2) = \exists x\, [(x \xrightarrow{i} o_1) \wedge (x \xrightarrow{i} o_2)]$
   Two outputs are i-control dependent on a common input.

4. **Communicational relation (R_4):**
   $R_1(o_1, o_2) = \exists x\, [((x \xrightarrow{d} o_1) \wedge (x \rightarrow o_2)) \vee ((x \xrightarrow{d} o_1) \wedge (x \rightarrow o_2))]$
   Two outputs are dependent on a common input. One of two outputs has data dependence on the input and the other can have a control or a data dependence.

5. **Sequential relation (R_5):**
   $R_1(o_1, o_2) = (o_1 \rightarrow o_2) \wedge (o_2 \rightarrow o_1)$
   One output is dependent on the other output.

6. **Functional relation (R_6):**
   $R_1(o_1, o_2) = o_1 \equiv o_2$
   There is only one output in a module.

These six relations are in an ordinal scale; cohesion strength increases from R_1 to R_6. These six relations correspond to six association principles (temporal cohesion is not included) of SMC Cohesion with some degree of overlap. (The correspondence is shown in section 3.3.) The DLC measure is defined based on the six relations.

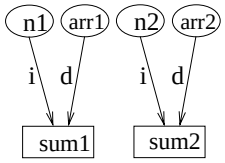
DLC Measure Definition. The cohesion level of a module is determined by the relation levels of output pairs. For each pair of outputs, the strongest relation for that pair is used. The cohesion level of the module is the weakest (lowest level) of all of the pairs. That is, the output pair with the weakest cohesion determines the cohesion of the module.

Consider the IODG's of Figure 1. Outputs *hsum* and *asum* of module *Asum_Hsum* have iterative and communicational relations. Since the communicational relation is stronger than the iterative relation, the cohesion level of module *Asum_Hsum* is communicational cohesion. Module *Fibo_Amean_Hmean* has three pairs of outputs. The output pair *fib_arr* and *amean* has three relations, iterative, communicational, and sequential. Since the sequential relation is the strongest, the pair has a sequential relation. Similarly, the output pair *fib_arr* and *hmean* has a sequential relation, and the output pair *amean* and *hmean* has a communicational relation. Since the communicational relation is the weakest among the relations of all pairs, the entire module exhibits a communicational cohesion.

Figure 2 shows six cohesion levels for six simple modules. The figure visually displays the intuition behind each DLC cohesion level.

(a) procedure Sum1_and_Sum2

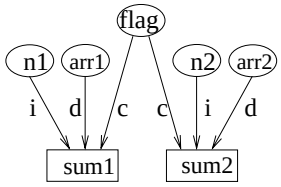
```
( n1, n2 : integer;
  arr1, arr2 : int_array;
  var sum1,
      sum2 : integer );
var i : integer;
begin
   sum1 := 0;
   sum2 := 0;
     for i := 1 to n1 do
        sum1 := sum1 + arr1[i];
     for i := 1 to n2 do
        sum2 := sum2 + arr2[i];
end;
```



Coincidental cohesion

(b) procedure Sum1_or_Sum2

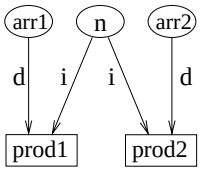
```
( n1, n2, flag : integer;
  arr1, arr2 : int_array;
  var sum1,
      sum2 : integer );
var i : integer;
begin
   sum1 := 0;
   sum2 := 0;
   if flag = 1
     for i := 1 to n1 do
        sum1 := sum1 + arr1[i];
    else
     for i := 1 to n2 do
        sum2 := sum2 + arr2[i];
end;
```



Conditional cohesion

(c) procedure Prod1_and_Prod2

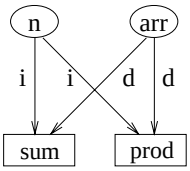
```
( n : integer;
  arr1, arr2 : int_array;
  var prod1,
      prod2 : integer );
var i : integer;
begin
   prod1 := 1;
   prod2 := 1;
   for i := 1 to n do begin
      prod1 := prod1 * arr1[i];
      prod2 := prod2 * arr2[i];
   end;
end;
```



Iterative cohesion

(d) procedure Sum_and_Prod

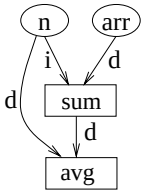
```
( n : integer;
  arr : int_array;
  var sum,
      prod : integer );
var i : integer;
begin
   sum := 0;
   prod := 1;
   for i := 1 to n do begin
      sum := sum + arr[i];
      prod := prod * arr[i];
   end;
end;
```



Communicational cohesion

(e) procedure Sum_and_Avg

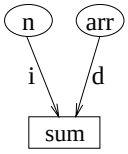
```
( n : integer;
  arr : int_array;
  var sum : integer;
  var avg : float );
var i : integer;
begin
   sum := 0;
   for i := 1 to n do
      sum := sum + arr[i];
   avg := sum / n;
end;
```



Sequential cohesion

(f) procedure Sum

```
( n : integer;
  arr : int_array;
  var sum : integer );
var i : integer;
begin
   sum := 0;
   for i := 1 to n do
      sum := sum + arr[i];
end;
```



Functional cohesion

Figure 2: IODG's and DLC levels for six simple procedures.

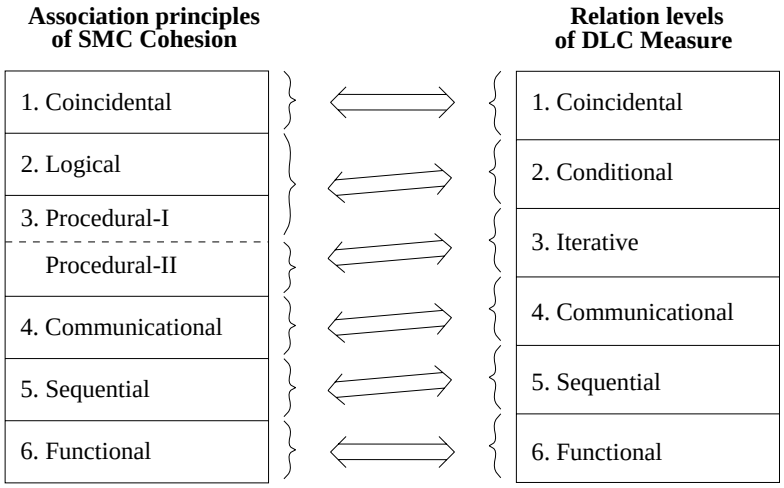


Figure 3: The relationship between the six association principles of SMC Cohesion and the six relation levels of DLC measure.

3.3 The Relationship between the DLC Measure and SMC Cohesion

SMC Cohesion is intended to be used to predict the quality attributes of modules that would be created from a given design. It is defined in terms of "processing elements", which is processing that must be done in a module but may not yet be reduced to code. The DLC measure treats each output component as part of a module's functionality, which is how functional cohesion measures [4] and Lakhotia's rules to compute module cohesion [10] have been defined. The "processing element" of SMC Cohesion, therefore, corresponds to the output component of the DLC measure.

Figure 3 shows the relationship between the DLC measure and SMC Cohesion. We omit the temporal association of SMC Cohesion because the DLC measure can not indicate temporal cohesion.

For SMC Cohesion, procedurally associated processing elements are elements of the same procedural unit. The common procedural units are an iteration and decision process. We separate procedural association into two categories, the decision unit (procedural-I) and the iteration unit (procedural-II). We now examine the relationship between the relation levels of DLC measure and the association principles of SMC Cohesion.

1. **Coincidental relation vs coincidental association:** When a pair of output components has a coincidental relation, the data tokens having dependence on one output do not have any connection with the data tokens corresponding to the other output – there are two independent functions. Thus, the coincidental relation matches the intuition of the coincidental association of SMC Cohesion.

2. **Conditional relation vs logical/procedural-I associations :** Consider two cases: (1) a pair of outputs are c-control dependent on a common input and (2) one output is c-control dependent on an input and the other output is i-control dependent on the same input. Case (1) includes both the logical and procedural-I associations of SMC Cohesion since processing elements of both associations always share a common decision unit. Case (2) does not match any of SMC's association principles. We include case (2) in the DLC conditional relation because it is, intuitively, clearly stronger than the coincidental relation and weaker than the iterative relation. The conditional relation includes both logical and procedural-I associations but not other associations. Thus, we can reasonably match the conditional relation with both logical and procedural-I associations.

3. **Iterative relation vs procedural-II association:** Since processing elements of procedure-II association share a common iteration unit, the iterative relation includes procedure-II association. It cannot include other associations. (The iterative relation includes some rare cases of communicational association. See the following discussion of communicational relation.) The iterative relation reasonably matches procedural-II association.

4. **Communicational relation vs communicational association:** Processing elements in a communicational association operate upon the same input and/or produce the same output. An example is a pair of components that have data dependence on one input. A processing element can operate upon input data without causing data dependence between them. For example, the summation of numbers from 1 to n can be implemented by direct computation ($n(n+1)/2$) or by iteration. When using iteration, the sum is not data dependent on the input n; it is only control dependent. However, such cases are rare and are not included in our analysis.

5. **Sequential relation vs sequential association:** In a sequential association, the output data from one processing element serve as input to the next processing element. This is clearly represented by the dependence of the data flow graph between the two processing elements. The sequential relation matches the intuition of sequential association.

6. **Functional relation vs functional association:** When a module contains only one output, the module has functional relation. Thus, the functional relation matches the function association of SMC cohesion.

Since the six association principles of SMC Cohesion are on ordinal scale, we claim the following relationship between the six relation levels of the DLC measure:

$$Coincidental < Conditional \leq Iterative < Communicational < Sequential < Functional$$

The DLC measure is on an ordinal scale as long as we accept the ordering implied by the association principles of SMC Cohesion.

4 Restructuring Software Designs

The DLC cohesion level can be used as a criterion to determine whether or not a given module should be redesigned or restructured. An IODG provides visual help to determine how to perform the restructuring. The restructuring process is a sequence of restructuring operations.

4.1 Restructuring Operations

Figure 4 shows eight basic restructuring operations using the IODG. Figure 4 (a) shows the decomposition of a module that exhibits coincidental cohesion. Since each group of data tokens corresponding to each output does not have any dependence relation on the other group, the decomposition simply requires the separation of the groups.

Figure 4 (b) shows the decomposition of a module with conditional, iterative, or communicational cohesion. The decomposition process copies all common and non-common data tokens in a dependence relationship with the each output into the resulting module.

Figure 4 (c) shows two operations: (1) the decomposition of a module with sequential cohesion and (2) the composition of two modules with a sequential relationship. The output of a module (producer module) is used as the input of the other (user module). In case (1), a module with sequential cohesion becomes two modules that have a sequential relationship. The producer module includes all data tokens on which the first output depends. The user module includes all data tokens on which the second output depends without the data tokens on which the first output has dependence. The operation of case (2) is the inverse of case (1).

Figure 4 (d) shows another way of decomposing a module with sequential cohesion. An output component is replaced by a module call and is factored out into a separate module (callee). The callee includes the output and all data tokens that the output depends on. The output and data tokens of the callee are removed from the caller and replaced by a module call statement.

Figure 4 (e) shows the composition of two module with a caller/callee relationship. The call statement is replaced by the tokens of the callee. The composition may be appropriate when the callee is called only by the caller. The composition process can reduce unnecessary coupling.

Figure 4 (f) contains two operations, 'hide' and 'reveal'. Using hide, H(*M1*:*O1*), output *O1* of module *M1* is hidden by changing the output into a local variable. This operation removes an unnecessarily exposed output; the output is not used outside of its module.

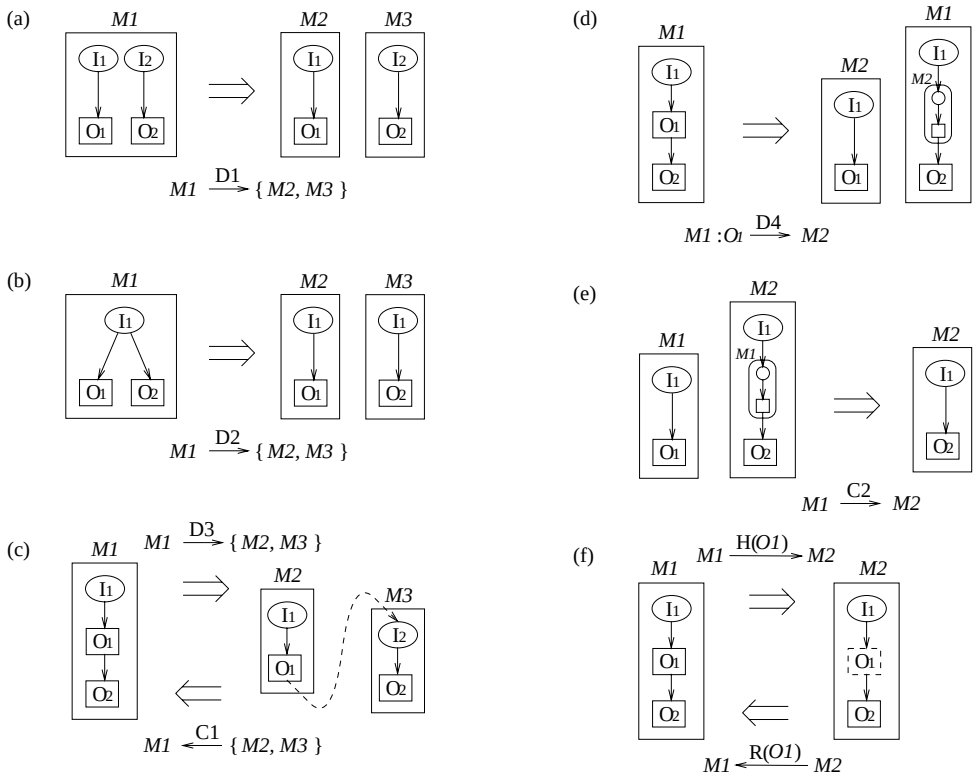


Figure 4: Eight basic operations for module restructuring.

'Reveal' is the inverse of hide. Using reveal, R(*M1*:*O1*), a local variable *O1* of *M1* is revealed by changing the local variable into an output variable. The operation reveals a hidden function and exports it. Reveal can be used to separate a hidden function from a large module. We simply reveal the local variable corresponding to the hidden function and apply the appropriate decomposition operation.

Existing software can potentially be restructured automatically by applying the restructuring operations. The data dependences, IODG's, and the DLC measure can be generated using practical code analysis technique.

4.2 A Restructuring Process

The following restructuring process consisting of a sequence of restructuring operations, is applied to improve the design structure of software system:

1. Generate IODG's of the modules of interest. If the software is at the design stage, the designer draws the IODG for each module. (Ideally, a software tool would construct an IODG from from a design.) During maintenance, the input-output dependence information can be automatically generated using a tool (a DLC tool) and an IODG can be drawn based on the information.

2. Compute the DLC level from each IODG. It is straightforward to compute DLC level from the IODG information. This step can also be automated by the DLC tool.

3. Locate the modules with low DLC levels and determine the poorly-designed modules among them. Modules with multiple independent functions will be identified. The optimal DLC level will depend on the application, the required reusability, readability, and maintainability of software. Managers need to specify expected marginal DLC levels of modules.

4. Decompose the IODG of each module that has been identified as poorly-designed one. This step includes two sub-steps:

(a) Partition the output components of the IODG so that when decomposed according to the partition, each resulting IODG has higher DLC level. The IODG and DLC measure guides the partitioning process. The partitioning process can be automated by computing DLC values for all possible partitions. The number of output components of a module is generally limited to a tractable number.

(b) Decompose each IODG according to its partition. Each resulting IODG includes input-output components that have dependence relation with the partitioned outputs. The dependence type (i.e., data, i-control, or c-control dependence) between components is also copied.

To decompose two IODG's with a caller-callee relationship, the callee is examined first. The corresponding invocation in the caller is changed to reflect the callee's decomposition, and then the decomposition is applied to the caller.

This step is repeated until the DLC level of each resulting IODG is acceptable.

5. Locate unnecessarily decomposed (i.e., overmodularized) modules and compose them. When a system is overmodularized, the overall interaction between modules is unnecessarily increased, i.e., the coupling of the system is high. To locate overmodularized modules, a practitioner can use other quality measures such as coupling, size, and/or reuse measures. The IODG can help an engineer visualize the module structure to help identify candidates for composition.

6. Generate module code. If the software being restructured is an existing product, the final step is generating module code corresponding to each IODG. The process of code generation can also be automated by the DLC tool. The tool uses the data tokens and the dependence information that was obtained from the initial modules during step 1.

4.3 Restructuring Examples

Example 1. Figure 5 shows the restructuring process applied to procedure *Sum_Prod_Avg*, which computes the sum and average of the values in one array and the product of the values in another array. If development is in the design stage, only IODG's are available. During maintenance, the corresponding program code is available.

Assume that the code of procedure *Sum_Prod_Avg* exists and is considered for restructuring. First the IODG information of *Sum_Prod_Avg* is generated, and the corresponding DLC level is computed. The graph shows that *sum* and *prod* have an iterative relation, *sum* and *avg* have sequential and communicational relations, and *avg* and *prod* have iterative and communicational relations.

Since *sum* and *avg* have two relations and sequential relation is ranked higher than communicational relation, sequential relation is chosen for the output pair. Since *avg* and *prod* have also two relations and communicational relation has higher rank than iterative relation, communicational relation is chosen for the output pair. Because among three pairs of outputs *sum* and *prod* have lowest rank of relation, iterative relation, the corresponding cohesion level of the procedure is therefore iterative cohesion.

We want to decompose the procedure into two procedures with higher cohesion levels. The optimal partition of output components of the procedure among all possibilities is one partition for *sum* and *avg*, and another partition for *prod*. Decomposed IODG's *Sum_Avg* and *Prod* corresponding to each partition are generated. The cohesion level of procedure *Sum_Avg* is sequential cohesion and that of *Prod* is functional cohesion. We do not decompose the procedures further and generate program code corresponding to the IODG's.

Example 2. Figure 6 shows the restructuring process of modules *Asum_Hsum* and *Fibo_Amean _Hmean* of Figure 1. The restructuring involves the caller-callee relationship between the two procedures and several restructuring operations. The resulting restructured modules are given in Figure 7. At the start of the restructuring process, both modules exhibit communicational cohesion. The modules are restructured into three modules that exhibit functional cohesion, the strongest cohesion level. The restructured modules should be easier to understand, maintain, and reuse.

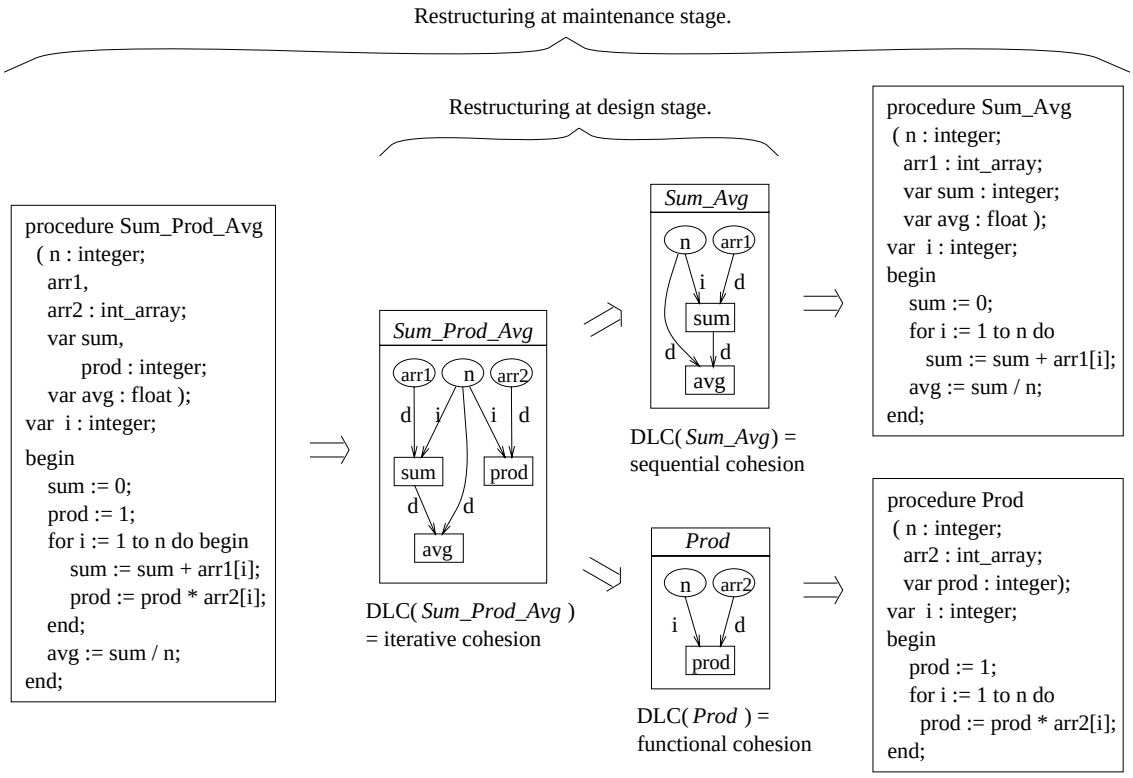


Figure 5: Example 1: restructuring procedure *Sum_Prod_Avg*.

Example 3: Figure 8 shows the restructuring process of modules with the sequential relationship, i.e., an output of a module is used as an input of another module. Assume that procedure *BasicSalary* is called only by procedure *Salary_Bonus* and the functionality 'salary' of procedure *Salary_Bonus* is sometimes used independently from 'bonus' of the module. Their design structures viewed from IODG representation is not desirable since the functionality 'basic_salary' is exposed unnecessarily and the functionality 'salary' needs to be in an independent module. The given modules, *BasicSalary* and *Salary_Bonus* are the examples of poorly-designed software even though they have relatively high DLC cohesion levels, sequential cohesion and functional cohesion. In this example, the IODG representation plays a more important role than the DLC measure. The resulting modules, after the restructuring process, are *Salary* and *Bonus* whose DLC cohesion level are both functional cohesion.

5 Related Work

Closely related work has focused on code-level cohesion measures and restructuring based on code-level cohesion. Lakhotia uses the output variables of a module as the processing elements of SMC Cohesion and defines rules for designating a cohesion level which preserve the intent of the SMC Cohesion [10]. The associative principles of SMC Cohesion are transformed to relate the output variables based on their data dependence relationships. A 'variable dependence graph' models the control and data dependences between module variables. The rules for designating a cohesion level are defined using a strict interpretation of the association principles of SMC Cohesion. The rules to determine cohesion levels are formal. Thus, a tool can automatically perform the classification. However, the technique can be applied only after the coding stage since they are defined upon the implementation details.

Bieman and Ott develop cohesion measures that indicate the extent to which a module approaches the ideal of functional cohesion [4]. They introduce three measures of functional cohesion based on "data slices" of a procedure. Bieman and Ott show that the measures satisfy the requirements of an ordinal scale. The functional cohesion measures are formally defined, and cohesion measurement tools have been built.

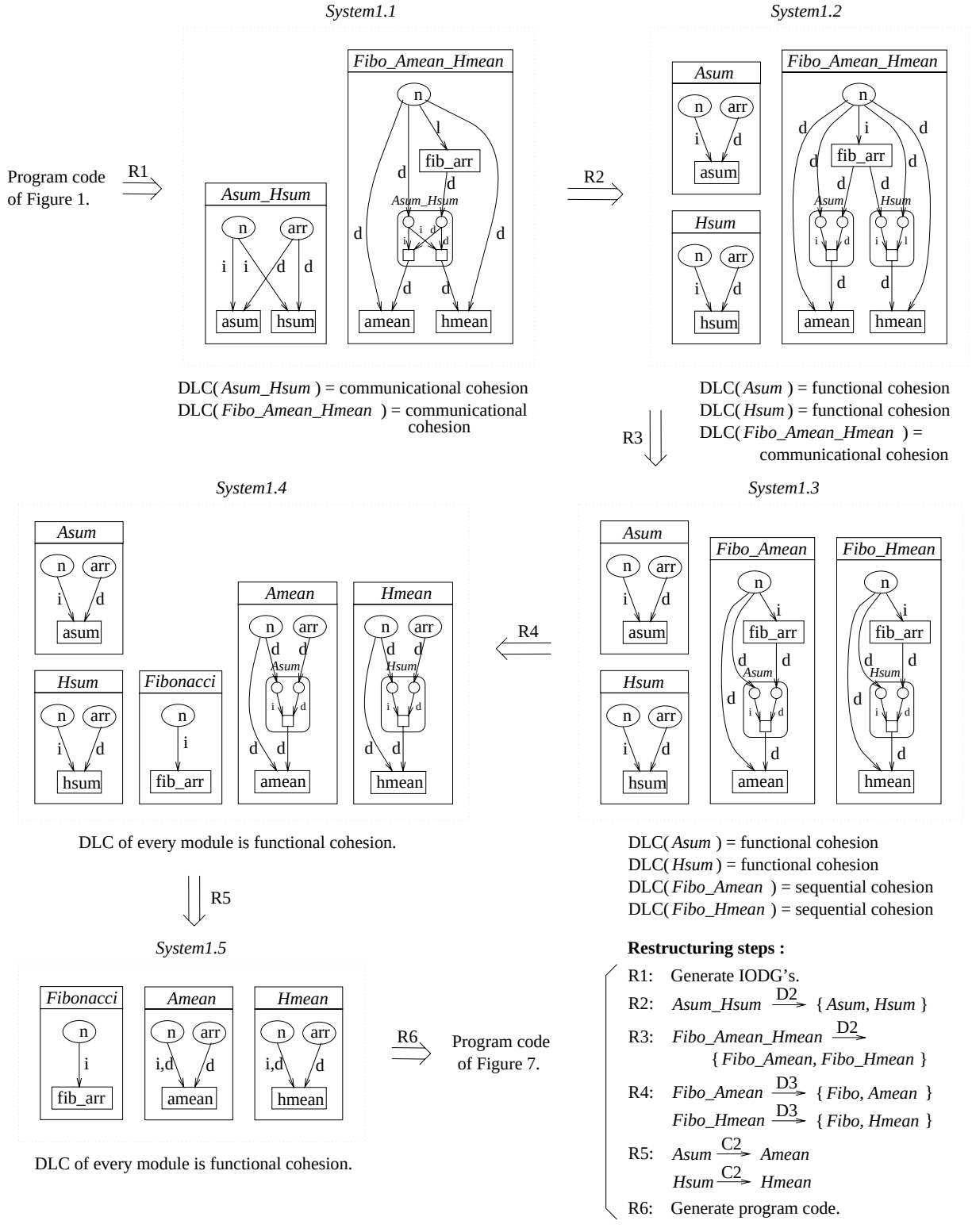


Figure 6: Example 2: restructuring procedures *Asum_Hsum* and *Fibo_Amean_Hmean* of Figure 1.

```
procedure Fibonacci
   ( n : integer;
     var fib_arr : int_array );
var i : integer;
begin
   fib_arr[1] := 1;
   fib_arr[2] := 2;
   for i := 3 to n
      fib_arr[i] := fib_arr[i-1]
                  + fib_arr[i-2];
end;
```

```
procedure Amean
   ( n : integer;
     arr : int_array;
     var amean : float );
var i, asum : integer;
begin
   asum := 0;
   for i := 1 to n do begin
      asum := asum + arr[i];
   end;
   amean := asum / n;
end;
```

```
procedure Hmean
   ( n : integer;
     arr : int_array;
     var hmean : float );
var i, hsum : integer;
begin
   hsum := 0;
   for i := 1 to n do begin
      hsum := hsum + 1.0/arr[i];
   end;
   hmean := n / hsum;
end;
```

Figure 7: Procedures produced after restructuring the procedures of Figure 1.

However, the measure also depends on the implementation details of module and can be applied only after coding stage of software development process.

Kim, Kwon, and Chung introduce restructuring methods where module strength (cohesion) is used as a criterion to restructure modules [9]. They define *processing blocks* which are similar to the 'data slices' of Bieman and Ott. A processing block is a group of data tokens with data or control dependence relationship with an output variable. A rule recognizes 'logically associated' module functions that are dependent together on an output. Each of these logically associated functions are also considered as a processing block. Unfortunately, these logically associated functions cannot always be automatically detected by analyzing program code. An examination of dependencies alone cannot determine whether a predicate variable is used to select a function or to compute a function.

Module strength is defined in terms of *data sharing*, *control sharing*, and *level of sharing*. Depending on its module strength, a module is restructured by either 'separating' or 'grouping'. A module with low module strength is split into new modules, while other modules are decomposed and the resulting components are grouped into a package. The decision to group processing blocks into a package cannot be made using only module strength. The process of making a package requires an understanding of both module functions and design decisions.

Like our approach, module strength is used as a criterion for software restructuring. However, Kim et al define cohesion based only on the code implementation. The attributes that are actually quantified by the measure are not specified. For restructuring, the measure computes the average of the relatedness between processing blocks rather than finding the most weakly connected blocks.

Our approach is unique in that we use only design-level information to determine the cohesion and restructuring options. Our design-level cohesion measure quantifies well-defined attributes in a consistent fashion. Finally, our cohesion measure, cohesion model and restructuring process can be automated.

6 Conclusions

The choice of a good software design structure should be made in the most objective fashion possible. We apply the notion of design cohesion to the problem of visualizing, quantifying, and restructuring a software system. Our method is based on the notion of cohesion developed by Stevens, Myers, and Constantine [12]. In this paper, we report the following progress towards improving the ability to make objective software design decisions:

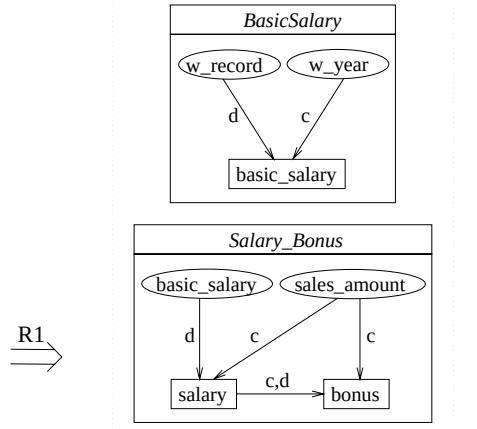
1. We define the IODG, a graph model that represents a design-level view of a module. The IODG is a formal model based on the dependency relationships between inputs and outputs of a module. It can be used to graphically visualize the design structure of a module. The IODG with its formal basis and graphic presentation can surely help one to understand the functionalities of a module.

2. We derive a design-level cohesion (DLC) measure based on the IODG representation of module, and we show that DLC is consistent with the intuition provided by SMC Cohesion. The DLC measure

```
procedure BasicSalary
  (w_record : int_array;
   w_year : integer;
   var basic_salary : integer );
var i, w_sum : integer;
begin
    w_sum := 0;
    for i := 1 to 31 begin
        w_sum := w_sum + w_record[i];
    if w_year < 5
        basic_salary := w_sum * 10;
    else
        basic_salary := w_sum * 15;
end;
```

```
procedure Salary_Bonus
  (basic_salary,
   sales_amount : integer;
   var salary,
        bonus : integer );
var credit : integer;
begin
    if sales_amount < 1000
       salary := basic_salary + 100;
  else salary := basic_salary + 200;
    crdit := sales_amount /1000
             + salary/100;
    if credit < 100
        bonus := salary / 20;
    else if credit < 500
        bonus := salary / 10;
    else bonus := salary / 5;
end;
```

System2.1

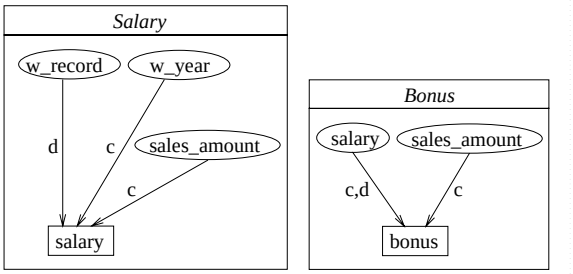


R1

DLC(*BasicSalary*) = functional coh.
DLC(*Salary_Bonus*) = sequential coh.

R2

System2.3

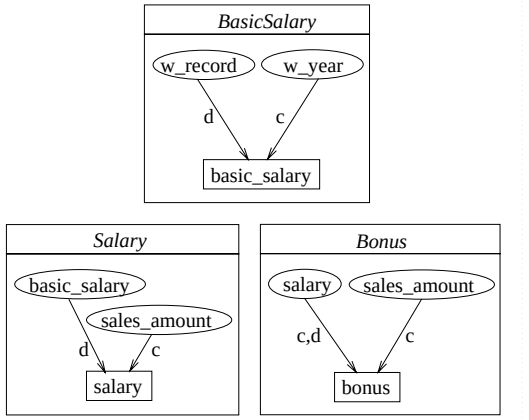


R3

DLC(*Salary*) = functional coh.
DLC(*Bonus*) = functional coh.

R4

System2.2



DLC(*BasicSalary*) = functional coh.
DLC(*Salary*) = functional coh.
DLC(*Bonus*) = functional coh.

```
procedure Salary
  (w_record : int_array;
   w_year : integer;
   sales_amount : integer;
   var salary : integer );
var i, w_sum, basic_salary : integer;
begin
    w_sum := 0;
    for i := 1 to 31 begin
        w_sum := w_sum + w_record[i];
    if w_year < 5
        basic_salary := w_sum * 10;
    else basic_salary := w_sum * 15;
    if sales_amount < 1000
        salary := basic_salary + 100;
  else salary := basic_salary + 200;
end;
```

```
procedure Bonus
  (salary,
   sales_amount : integer;
   var bonus : integer );
var credit : integer;
begin
    crdit := sales_amount /1000
             + salary/100;
    if credit < 100
        bonus := salary / 20;
    else if credit < 500
        bonus := salary / 10;
    else
        bonus := salary / 5;
end;
```

Restructuring steps :

R1: Generate IODG's.

R2: *Salary_Bonus* $\xrightarrow{D3}$ { *Salary*, *Bonus* }

R3: { *BasicSalary*, *Salary* } $\xrightarrow{C1}$ *BasicSalary_Salary*

BasicSalary_Salary $\xrightarrow{H(\,BasicSalary\,)}$ *Salary*

R4: Generate program code.

Figure 8: Example 3: Restructuring procedures with sequential relationship.

provides an objective criteria for evaluating and comparing alternative design structures.

3. We define eight basic restructuring operations based on the IODG representation and the DLC measure. We describe a process for applying the restructuring operations to improve design of system modules. We show that the restructuring process can improve the design-level cohesion in three examples.

The IODG representation, the DLC measure, and the restructuring process can be applied during software design or maintenance. During design, IODG's can be constructed from design information. Implementation details are not needed. During maintenance, IODG's can be readily generated using a compiler-like code analysis tool. Such a tool can be used to recapture designs from existing, possibly legacy, systems. The DLC measure can be easily computed once an IODG is generated either from a design or an implementation.

We are now implementing tools to generate IODG's from software designs and implementations, to graphically display IODG's, and to support the restructuring process. We have already developed class cohesion measures and measurement tools for object-oriented software [3]. We also plan to evaluate the effects of restructuring on external quality attributes such as testability, reusability, reliability, and maintainability.

References

[1] A. Albrecht and J. Gaffney. Software function, source lines of code, and development effort prediction. *IEEE Trans. Software Engineering*, SE-9(6):639–648, June 1983.

[2] A. Baker, J. Bieman, N. Fenton, D. Gustafson, A. Melton, and R. Whitty. A philosophy for software measurement. *J. Systems and Software*, 12(3):277–281, July 1990.

[3] J. Bieman and B-K Kang. Cohesion and reuse in an object-oriented system. *Proc. ACM Symp. Software Reusability. (SSR'94)*, pp. 259–262, April 1995. Reprinted in *ACM Software Engineering Notes*, Aug. 1995.

[4] J. Bieman and L. Ott. Measuring functional cohesion. *IEEE Trans. Software Engineering*, 20(8):644–657, Aug. 1994.

[5] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[6] N. Fenton. *Software Metrics - A Rigorous Approach.* Chapman and Hall, London, 1991.

[7] N. Fenton. Software measurement: a necessary scientific basis. *IEEE Trans. Software Engineering*, 20(3):199–206, 1994.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[9] H-S Kim, Y-R Kwon, and I-S Chung. Restructuring programs through program slicing. *Int. J. Software Engineering and Knowledge Engineering*, 4(3):349–368, Sept. 1994.

[10] A. Lakhotia. Rule-based approach to computing module cohsion. *Proc. 15th Int. Conf. Software Eng.*, pp. 35–44, 1993.

[11] T. McCabe. A complexity measure. *IEEE Trans. Software Engineering*, SE-2(4):308–320, 1976.

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

[13] M. Woodward. Difficulties using cohesion and coupling as quality indicators. *Software Quality J.*, 2(2):109–127, June 1993.

[14] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.