ELSEVIER

The Journal of Systems and Software 42 (1998) 175–187

The Journal of Systems and Software

Using design abstractions to visualize, quantify, and restructure software [1]

Byung-Kyoo Kang [a], James M. Bieman [b,*]

[a] *Electronics and Telecommunications Research Institute, South Korea*

[b] *Department of Computer Science, Colorado State University, Fort Collins, CO 80523, USA*

Received 10 December 1996; received in revised form 10 November 1997

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. © 1998 Elsevier Science Inc. All rights reserved.

Keywords: 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 [1], functional cohesion [2], 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 [3]. The object-oriented design measures proposed by Chidamber and Kemerer [4] provide a mechanism to gather quantitative information about classes in object-oriented software, but they do not provide guidance to help evaluate design alternatives. Gamma et al. [5] describe a set of structural design patterns for object-oriented software and objective, but not quantitative, criteria for choosing a particular pattern.

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 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

[*] Corresponding author. Tel.: +1 970 491 7096; fax: +1 970 491 2466; e-mail: bieman@cs.colostate.edu.

[1] A short version of this paper was published in *Proc. 8th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE '96)*. (Research partially supported by NASA Langley Research Center grant NAG1-1461.)

0164-1212/98/$19.00 © 1998 Elsevier Science Inc. All rights reserved.
PII: S0164-1212(98)10008-0

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 [6], 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 [7,8]. A model that captures the essence of the attribute is also needed [9]. A design model that can help make design attributes visible can be exceptionally valuable.

2. A model for visualizing software designs

A number of program representations for analyzing and maintaining procedural programs have been proposed in the literature, and are surveyed by Cimitile et al. [10]. Lakhotia [11] uses a variable dependence graph to model control and data flow in a program. We adapt Lakhotia's model to develop an input–output dependence graph (IODG) which models a design-level view of a module. The IODG 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 using the notation of Lakhotia; more formal definitions are given in compiler texts, for example, see [12]. The first two of the following relations are from Lokhotia [11].

Definition 1. 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 2. 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 3. 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 4. 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 5. 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 6. 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 condition-control dependence but no iteration-control dependence. For example, for (1) $x \xrightarrow{cc} y$ and (2) $x \xrightarrow{d} a \xrightarrow{cc} b \xrightarrow{d} y$, 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 a decision structure.

Definition 7. 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. For example, for (1) $x \xrightarrow{ic} y$ and (2) $x \xrightarrow{d} a \xrightarrow{cc} b \xrightarrow{ic} c \xrightarrow{d} y$, 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 $E = \{(x,y) \in V \times V \mid y \text{ has data, } c\text{-control, and/ or } i\text{-control dependence on } x\}$.

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 Fig. 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.

Fig. 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.

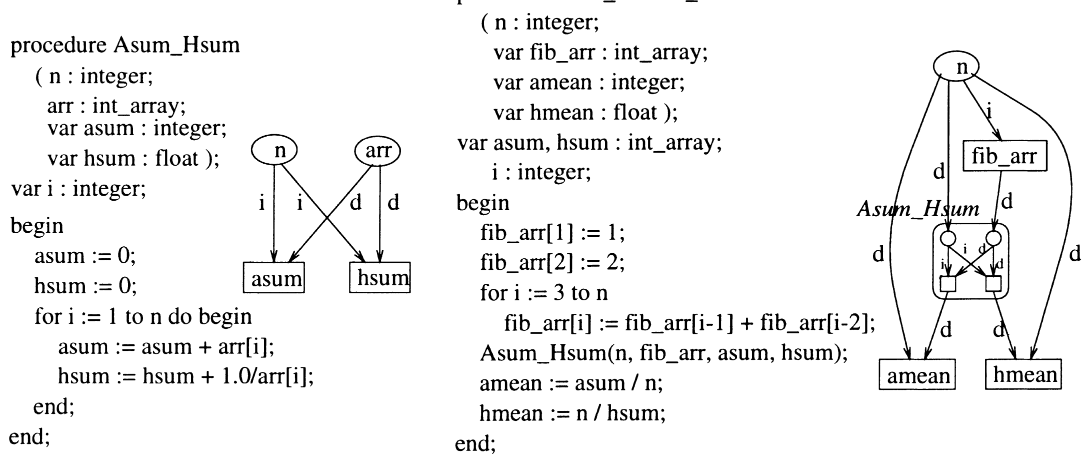
The extended IODG contains the complete dependence paths between inputs and outputs of a module. Thus, we can determine exact dependence relationships between input/output components. For example, consider input *n* and output *amean* of the IODG of *Fibo_Amean_Hmean*. We find three dependence paths 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. The visualization of an IODG is not necessary to take the cohesion measurements, but visualization is an aid in the restructuring process.

3. Measuring design cohesion

Software cohesion, as described by Stevens, Myers, and Constantine (SMC Cohesion) [6], 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



```
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;
```

Fig. 1. Input–output dependence graph representation for *Asum_Hsum* and *Fibo_Amean_Hmean*.

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 (DLC) satisfies the representation condition of measurement [7,8]. 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 [6]. 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 and by Lakhotia [11]. 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):

   $R_2(o_1, o_2) = \exists x\,[(x \xrightarrow{c} o_1) \wedge (x \xrightarrow{c} o_2)]$.

   Two outputs are *c*-control dependent on a common input.
3. *Iterative relation* (R_3):

   $R_3(o_1, o_2) = \exists x\,[(x \xrightarrow{i} o_1) \wedge (x \xrightarrow{i} o_2)]$.

   Two outputs are *i*-control dependent on a common input.
4. *Communicational relation* (R_4):$R_4(o_1, o_2) = \exists x\,[((x \xrightarrow{d} o_1) \wedge (x \xrightarrow{d} o_2)) \vee ((x \xrightarrow{p} o_1) \wedge (x \xrightarrow{q} o_2))]$, where $p, q \in \{d, c, i\}$, and $p \neq q$. Two outputs are dependent on a common input. An input is used to compute both outputs, but not as a condition flag to select one of two outputs, nor as a loop invariant to compute both outputs.
5. *Sequential relation* (R_5):

   $R_5(o_1, o_2) = (o_1 \rightarrow o_2) \vee (o_2 \rightarrow o_1)$.

   One output is dependent on the other output.
6. *Functional relation* (R_6): There is only one output in a module.

These six relations are on 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.) Relations 1, 4, and 5 are from corresponding relations of Lakhotia [11]. The DLC measure is defined in terms of the six relations following the approach of SMC Cohesion and Lakhotia.

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 Fig. 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.

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

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 [2] and Lakhotia's rules to compute module cohesion [11] have been defined. The "processing element" of SMC Cohesion, therefore, corresponds to the output component of the DLC measure.

Fig. 3 shows the relationship between the DLC measure and SMC Cohesion. We omit the temporal association of SMC Cohesion because the DLC measure cannot 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.

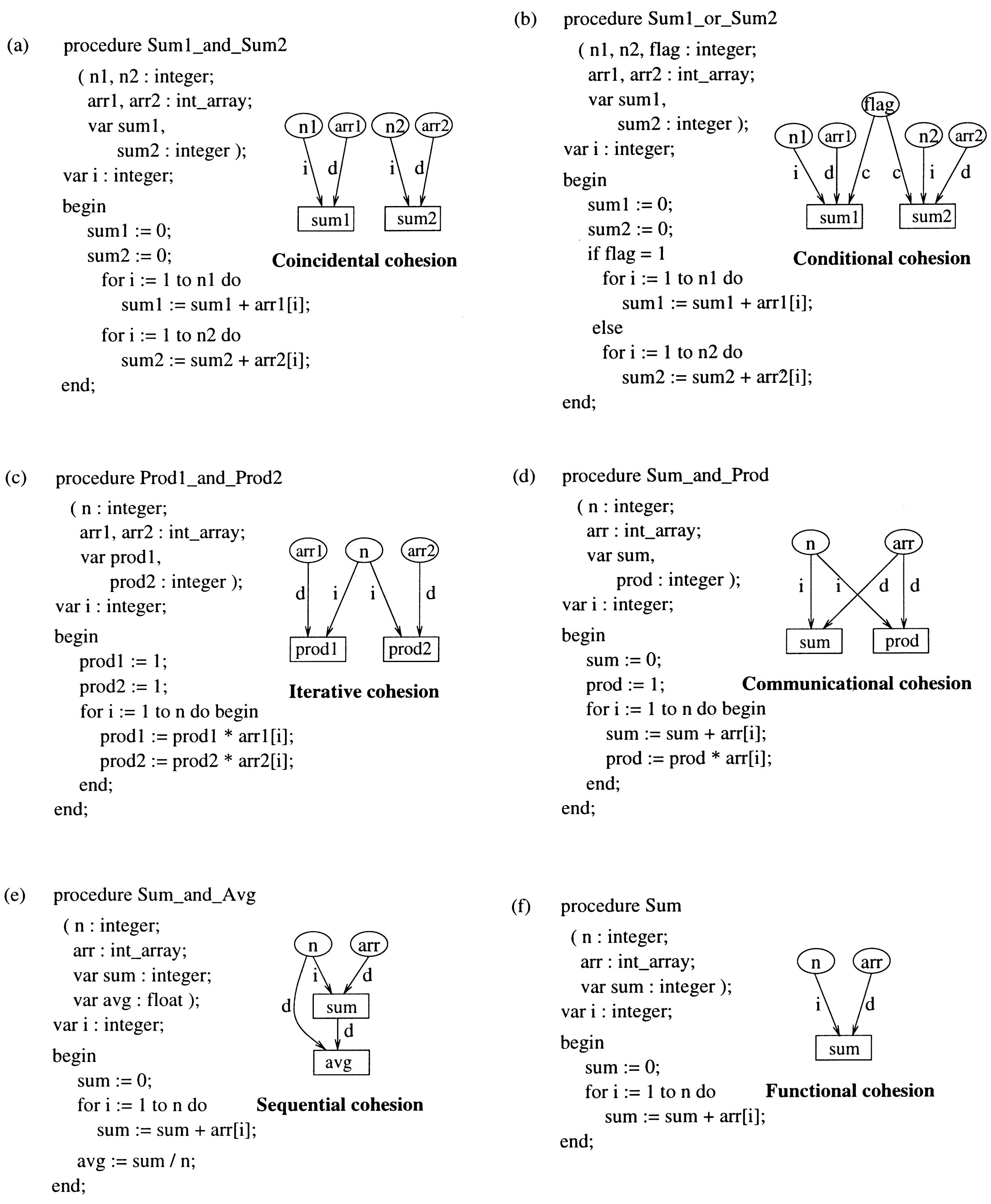
5. *Sequential relation vs. sequential association*: In a sequential association, the output data from one processing element serve as input to the next processing element. This is clearly represented by the dependence of the data flow graph between the two processing elements. The sequential relation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

$$\mathit{Coincidental} < \mathit{Conditional} \leqslant \mathit{Iterative}$$
$$< \mathit{Communicational} < \mathit{Sequential} < \mathit{Functional}$$

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



(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

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

3.4. Measurement tool

We have developed a prototype tool, FUNCO, that automatically generate the DLC measure. The tool processes C programs in a UNIX workstation environment. It has been installed and tested for SUN SPARC-stations running SUN-OS and IBM RS6000 systems running AIX. The beta version of Funco is available at URL `http://www.cs.colostate.edu/bieman/funco.html`.

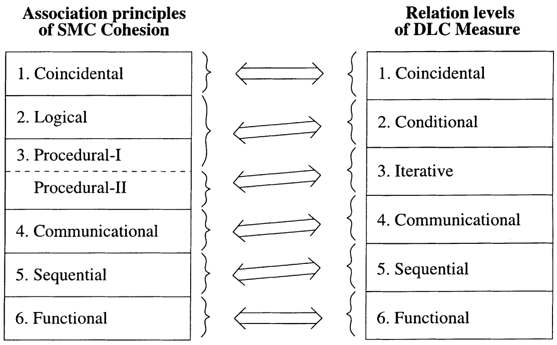


Fig. 3. The relationship between the six association principles of SMC Cohesion and the six relation levels of DLC measure.

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

Fig. 4 shows eight basic restructuring operations using the IODG. Fig. 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.

Fig. 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.

Fig. 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

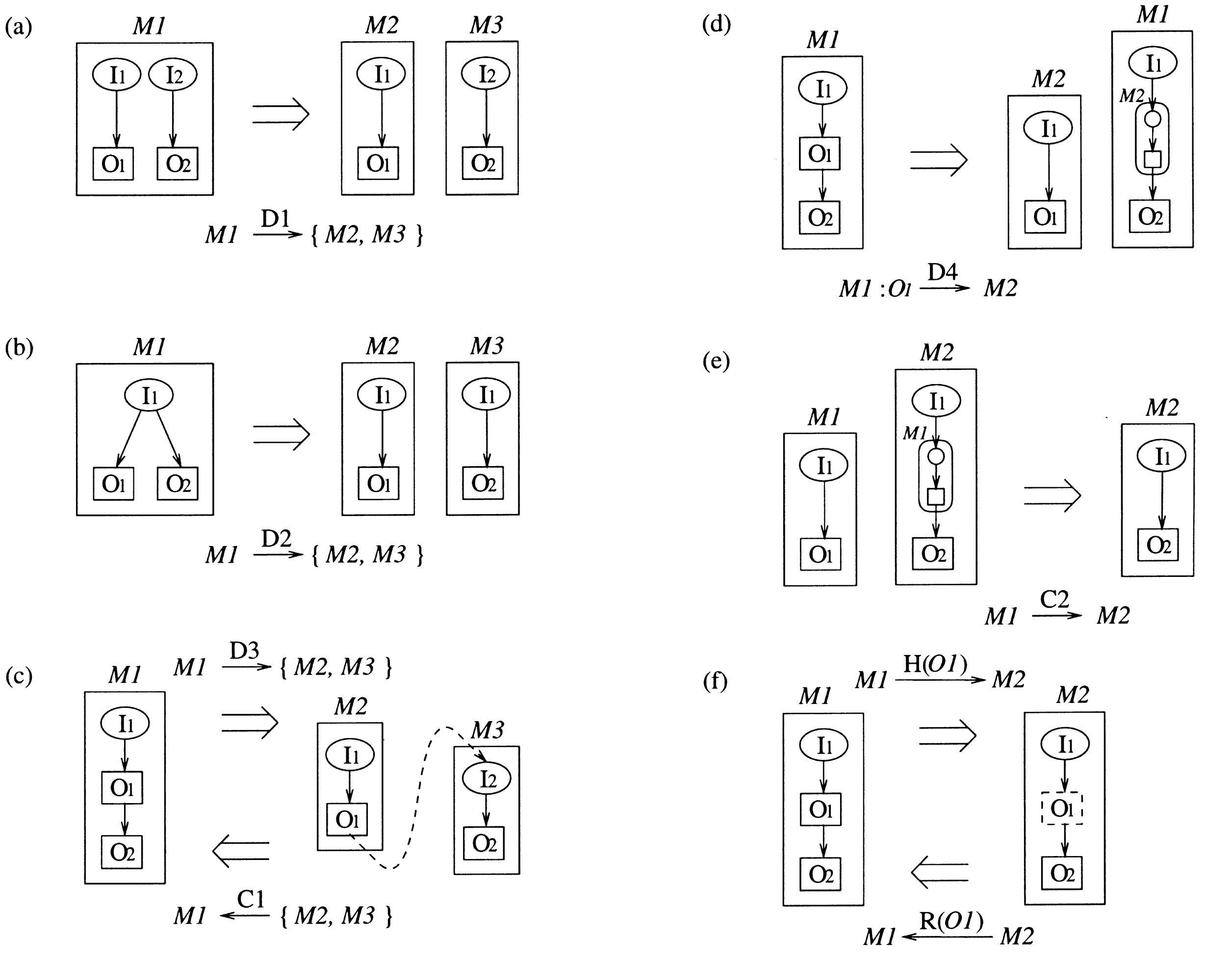

(a)
M1
I1
I2
O1
O2
M2
M3
$M1 \xrightarrow{D1} \{M2, M3\}$
(b)
M1
M2
M3
$M1 \xrightarrow{D2} \{M2, M3\}$
(c)
M1
$M1 \xrightarrow{D3} \{M2, M3\}$
M2
M3
$M1 \xleftarrow{C1} \{M2, M3\}$
(d)
M1
M2
M1
M2
$M1 : O1 \xrightarrow{D4} M2$
(e)
M2
M1
M1
M2
$M1 \xrightarrow{C2} M2$
(f)
M1
$M1 \xrightarrow{H(O1)} M2$
M2
$M1 \xleftarrow{R(O1)} M2$

Fig. 4. Eight basic operations for module restructuring.

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).

Fig. 4(d) shows another way of decomposing a module with sequential cohesion. An output component denoted by $M1$:$O1$ 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.

Fig. 4(e) shows the composition of two modules with a caller/callee relationship, the reverse of the operation in Fig. 4(d). 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.

Fig. 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.

'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. A system based on graph rewriting rules can be implemented for IODG's. We can follow the example of Engels et al. [14] who use attributed grammars to model, specify and manipulate software documents.

4.2. A restructuring process

There are many possible processes that can be used to restructure a system based on the IODG representation. We propose the following restructuring process, which consists of a sequence of restructuring operations, 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 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. Appropriate tokens can be identified from mappings through the original abstract syntax tree and new code can be generated to match restructured IODG's. Our process relies on data and control flow analysis rather than other formalisms, for example category theory as used by Maggiolo–Schettini et al. [15].

4.3. Restructuring examples

Example 1. Fig. 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.

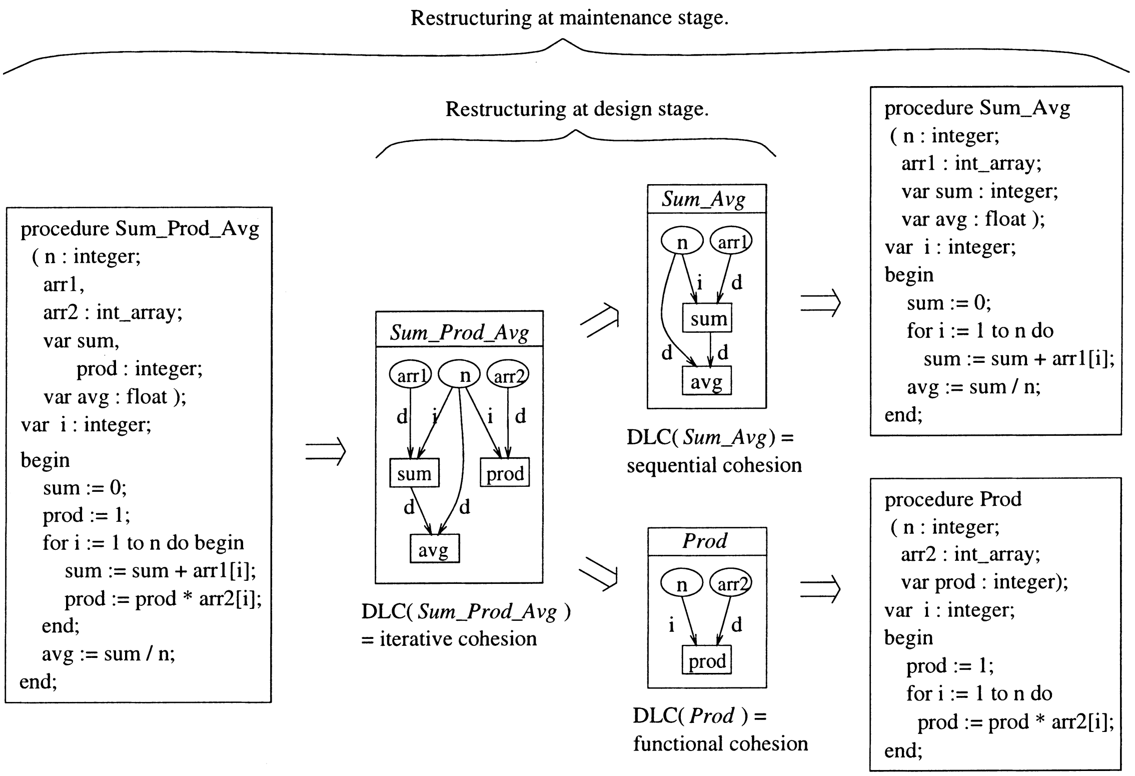
Assume that the code of procedure *Sum_Prod_Avg* exists and is considered for restructuring. First the IODG information of *Sum_Prod_Avg* is generated, and the corresponding DLC level is computed. The graph shows that *sum* and *prod* have an iterative relation, *sum* and *avg* have sequential and communicational 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. Fig. 6 shows the restructuring process of modules *Asum_Hsum* and *Fibo_Amean_Hmean* of Fig. 1. The restructuring involves the caller–callee relationship between the two procedures and several restructuring operations. The resulting restructured modules are given in Fig. 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.

Example 3. Fig. 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

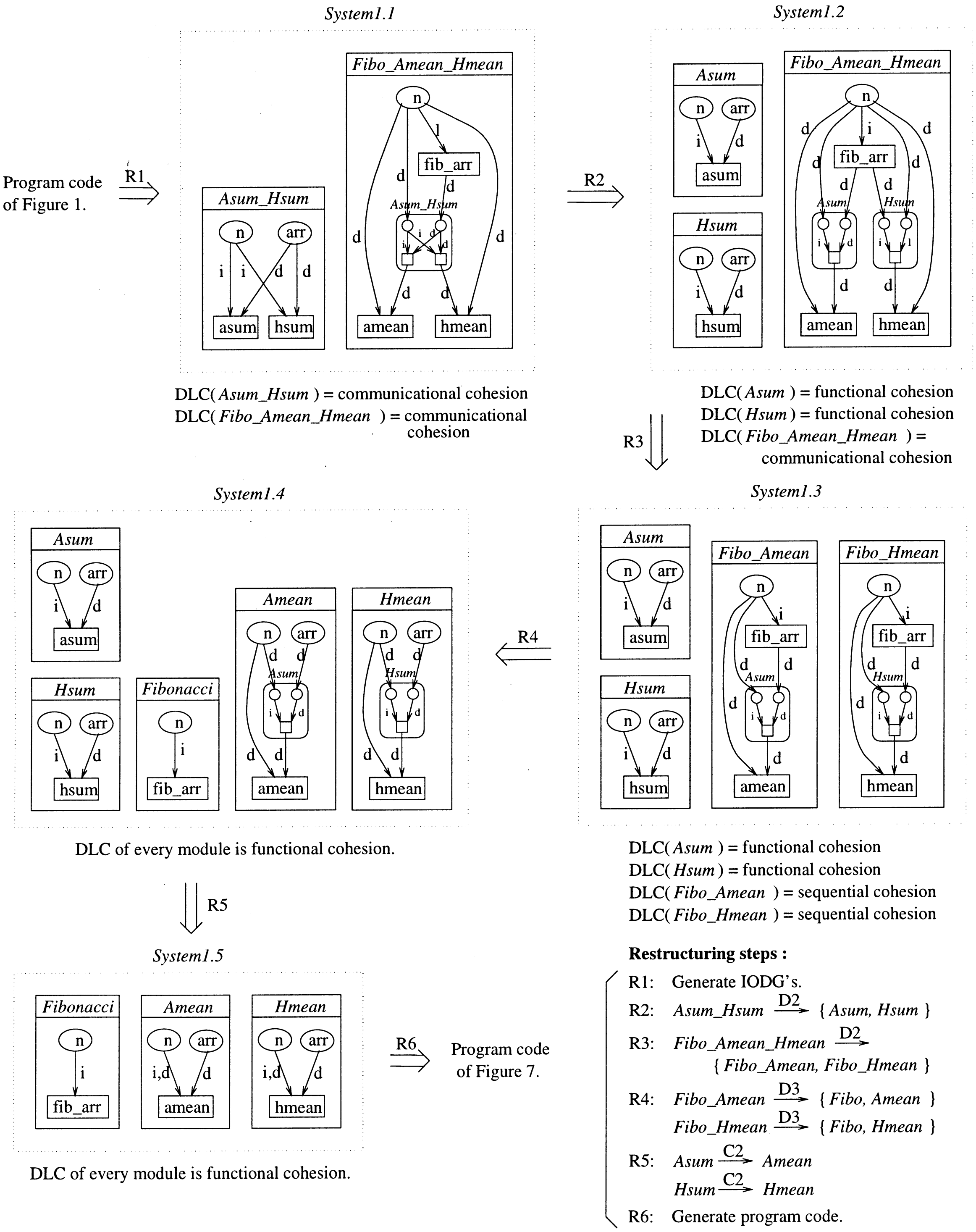

Restructuring at maintenance stage.
Restructuring at design stage.
Sum_Prod_Avg
arr1
n
arr2
d
i
i
d
sum
prod
d
d
avg
DLC(Sum_Prod_Avg) = iterative cohesion
Sum_Avg
n
arr1
i
d
sum
d
d
avg
DLC(Sum_Avg) = sequential cohesion
Prod
n
arr2
i
d
prod
DLC(Prod) = functional cohesion

```
procedure Sum_Prod_Avg
  ( n : integer;
    arr1,
    arr2 : int_array;
    var sum,
        prod : integer;
    var avg : float );
var  i : integer;
begin
   sum := 0;
   prod := 1;
   for i := 1 to n do begin
      sum := sum + arr1[i];
      prod := prod * arr2[i];
   end;
   avg := sum / n;
end;
```

```
procedure Sum_Avg
 ( n : integer;
   arr1 : int_array;
   var sum : integer;
   var avg : float );
var  i : integer;
begin
   sum := 0;
   for i := 1 to n do
      sum := sum + arr1[i];
   avg := sum / n;
end;
```

```
procedure Prod
 ( n : integer;
   arr2 : int_array;
   var prod : integer);
var  i : integer;
begin
   prod := 1;
   for i := 1 to n do
     prod := prod * arr2[i];
end;
```

Fig. 5. Example 1: Restructuring procedure *Sum_Prod_Avg*.



DLC(*Asum_Hsum*) = communicational cohesion
DLC(*Fibo_Amean_Hmean*) = communicational cohesion

DLC(*Asum*) = functional cohesion
DLC(*Hsum*) = functional cohesion
DLC(*Fibo_Amean_Hmean*) = communicational cohesion

DLC of every module is functional cohesion.

DLC(*Asum*) = functional cohesion
DLC(*Hsum*) = functional cohesion
DLC(*Fibo_Amean*) = sequential cohesion
DLC(*Fibo_Hmean*) = sequential cohesion

DLC of every module is functional cohesion.

Restructuring steps :

R1: Generate IODG's.
R2: *Asum_Hsum* $\xrightarrow{D2}$ { *Asum, Hsum* }
R3: *Fibo_Amean_Hmean* $\xrightarrow{D2}$ { *Fibo_Amean, Fibo_Hmean* }
R4: *Fibo_Amean* $\xrightarrow{D3}$ { *Fibo, Amean* }
Fibo_Hmean $\xrightarrow{D3}$ { *Fibo, Hmean* }
R5: *Asum* $\xrightarrow{C2}$ *Amean*
Hsum $\xrightarrow{C2}$ *Hmean*
R6: Generate program code.

Fig. 6. Example 2: Restructuring procedures *Asum_Hsum* and *Fibo_Amean_Hmean* of Fig. 1.

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, *Basic-Salary* 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.

```
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;
```

Fig. 7. Procedures produced after restructuring the procedures of Fig. 1.

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 [11]. 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. The technique, as originally described, can be applied only after the coding stage since it is defined upon the implementation details.

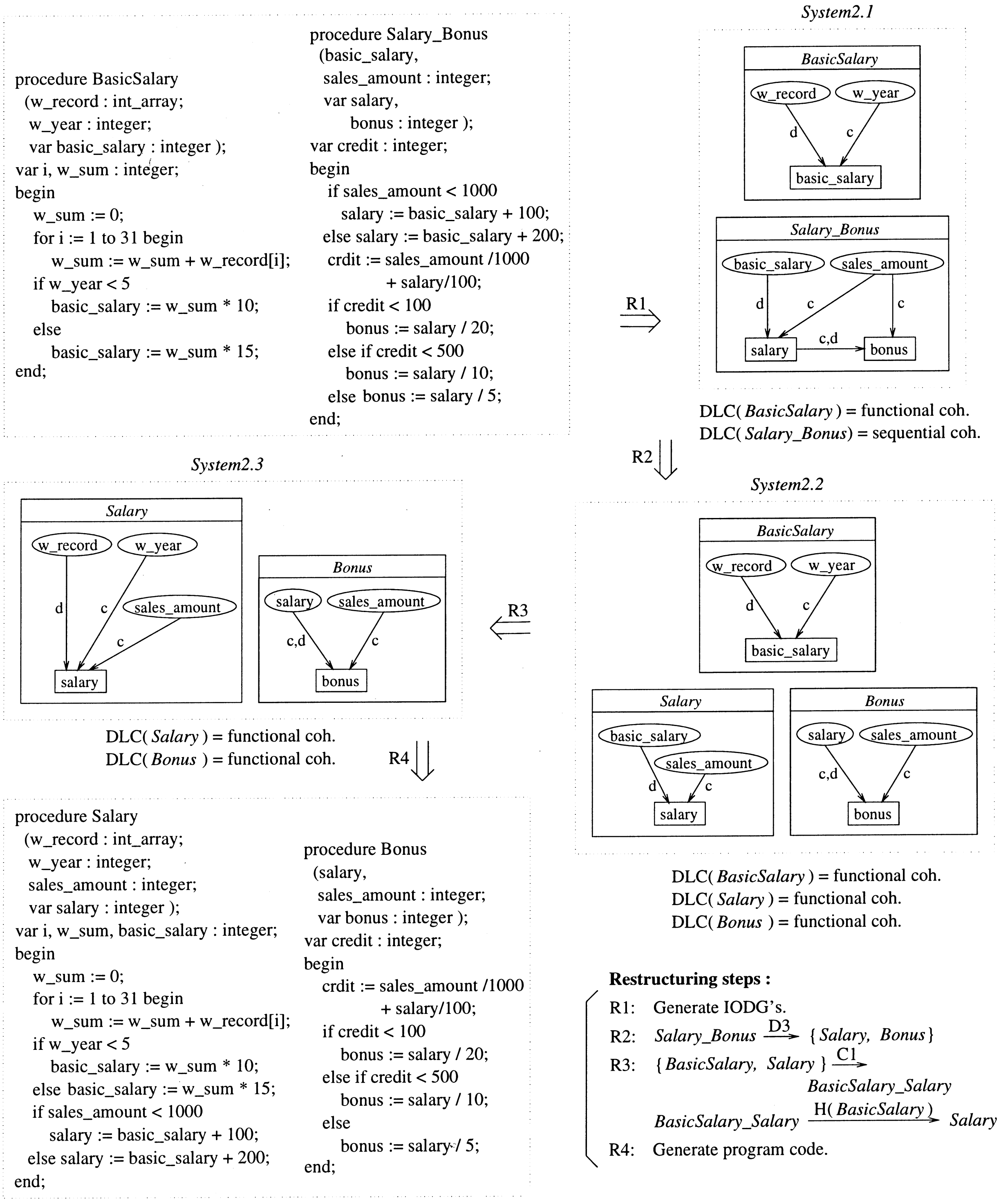
Bieman and Ott [2] develop cohesion measures that indicate the extent to which a module approaches the ideal of functional cohesion. They introduce three measures of functional cohesion based on "data slices" of a procedure. Bieman and Ott show that the measures satisfy the requirements of an ordinal scale. The functional cohesion measures are formally defined, and cohesion measurement tools have been built. However, the measures also depend on the implementation details of module and can be applied only after coding stage of software development process.

Bieman and Kang define 'class cohesion' measures for object-oriented programs [16]. Class cohesion is an attribute of a class that refers to the "relatedness" or the "connectivity" among components of the class. The components of a class are the instance variables and methods defined in the class plus those that are inherited. A method and an instance variable are related by the way that an instance variable is used by the method. Two methods are related (connected) through instance variable(s) if both methods use the instance variable(s). Class cohesion measures are defined in terms of the relative number of directly and/or indirectly connected methods in the class. These measures can be applied to object-oriented and object-based software.

Kim et al. [17] introduce restructuring methods where module strength (cohesion) is used as a criterion to restructure modules. 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.



```
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;
```

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

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

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

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 et al. [6]. 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, adapted from Lakhotia's variable dependence graph [11], models 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. Following the approach introduced by Lakhotia [11], 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 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, DLC measure, and 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 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.

Our prototype FUNCO toolkit, available at URL `http://www.cs.colostate.edu/bieman/funco.html`, can generate IODG's and DLC measures from code. We plan to evaluate the effects of restructuring on external quality attributes such as testability, reusability, reliability, and maintainability.

References

[1] T. McCabe, A complexity measure, IEEE Trans. Software Eng. 2 (4) (1976) 308–320.

[2] J. Bieman, L. Ott, Measuring functional cohesion, IEEE Trans. Software Eng. 20 (8) (1994) 644–657.

[3] A. Albrecht, J. Gaffney, Software function, source lines of code, and development effort prediction, IEEE Trans. Software Eng. 9 (6) (1983) 639–648.

[4] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Trans. Software Eng. 20 (6) (1994) 476–493.

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

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

[7] N. Fenton, Software Metrics – A Rigorous Approach, Chapman & Hall, London, 1991.

[8] N. Fenton, Software measurement: A necessary scientific basis, IEEE Trans. Software Eng. 20 (3) (1994) 199–206.

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

[10] A. Cimitile, A. De Lucia, M. Munro, A specification driven slicing process for identifying resuable functions, Software Maintenance: Research and Practice 8 (3) (1996) 145–178.

[11] A. Lakhotia, Rule-based approach to computing module cohesion, Proceedings of the 15th International Conference on Software Engineering, 1993, pp. 35–44.

[12] H. Zima, B. Chapman, Supercompilers for Parallel and Vector Computers, Addison-Wesley, Reading, MA, 1991.

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

[14] G. Engles, C. Lewerentz, M. Nagl, W. Schafer, A. Schurr, Building integrated software development environments part I: Tool specification, ACM Trans. Software Eng. and Meth. 1 (2) (1992) 135–167.

[15] A. Maggiolo, M. Napoli, G. Tortora, Web structures: A tool for representing and manipulating programs, IEEE Trans. Software Eng. 14 (11) (1988) 1621–1639.

[16] J. Bieman, B-K. Kang, Cohesion and reuse in an object-oriented system, Proceedings of the ACM Symposium Software Reusability (SSR'94), 1995, pp. 259–262, Reprinted in ACM Software Engineering Notes, 1995.

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

Byung-Kyoo Kang is a Senior Member of Technical Staff in the Software Engineering division of the Electronics and Telecommunications Research Institute (ETRI) in Korea. He received a Ph.D. in Computer Science from Colorado State University in 1997 where he developed a quantitative framework for restructuring software. He also implemented the Funco cohesion measurement tool. He also received an M.S. in Computer Science from Washington University in St. Louis, and a B.S. in Computer Science from Western Illinois University.

James M. Bieman is an Associate Professor in the Computer Science Department at Colorado State University. Dr. Bieman's research is focused on software measurement and automated software testing. He is developing methods to quantify software reuse, functional cohesion, and attributes of object oriented software. He is evaluating the use of executable specifications and software probes as practical software testing oracles. Bieman has conducted collaborative research projects with engineers at Reliable Software Technologies, Storage Technology, Hewlett-Packard, Martin Marietta, CTA, Micro-Motion, and other companies. He is a senior member of the IEEE, a member of the ACM, and is currently the Chair of the IEEE-CS TCSE Subcommittee on Quantitative Methods, Chair of the Steering Committee for the IEEE-CS International Symposium of Software Metrics, and General Chair of Metrics'97.