Using Design Cohesion to Visualize, Quantify, and Restructure Software*

Byung-Kyoo Kang and James M. Bieman

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
kang@cs.colostate.edu, bieman@cs.colostate.edu

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.

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. 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 [10], functional cohesion [3], 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 [4]. 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 [7].

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 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 visu-

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

alize, quantify, and restructure software. The term "software cohesion," which was introduced more than 20 years ago [11], 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.

First, we need a model that captures the essence of the attribute is needed [2].

2 A Model for Visualizing 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 module input and output components.

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. 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 [13].

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. 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, $G_M = (V, E)$. 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. 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 pro-

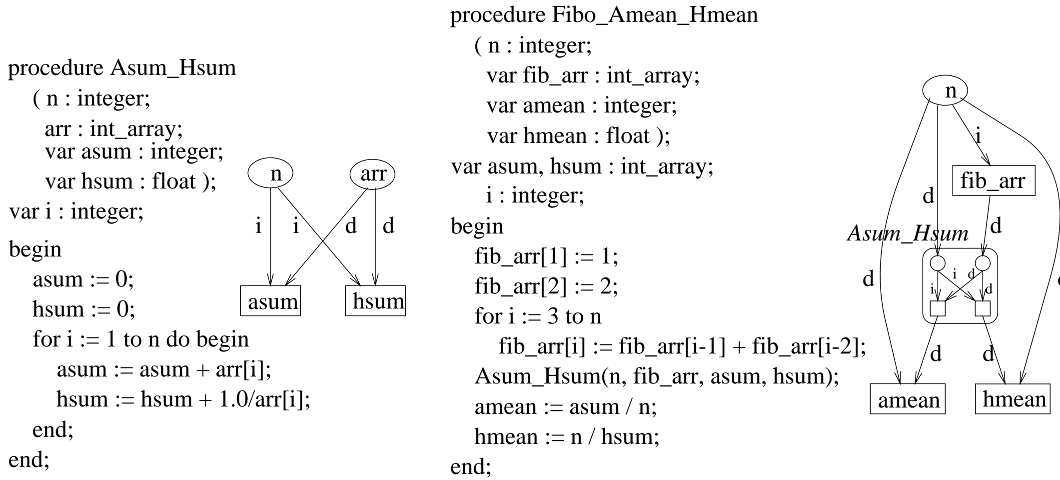

Figure 1: Input-output dependence graph representation for *Asum_Hsum* and *Fibo_Amean_Hmean*.

cedure *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*.

We can extend the IODG to include caller/callee relationships, and then 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) $n \stackrel{d}{\rightarrow} amean$, (2) $n \stackrel{d}{\rightarrow}$ input_parameter $\stackrel{i}{\rightarrow}$ output_parameter $\stackrel{d}{\rightarrow}$ $amean$, and (3) $n \stackrel{i}{\rightarrow} fib_arr \stackrel{d}{\rightarrow}$ input_parameter $\stackrel{d}{\rightarrow}$ output_parameter $\stackrel{d}{\rightarrow} amean$. According to our dependence definitions, *amean* has data and i-control dependencies on n.

3 Measuring Design Cohesion

Software cohesion, as described by Stevens, Myers, and Constantine (SMC Cohesion) [11], provides an intuitive mechanism for assessing the relatedness of the components in an individual module. 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 [5, 6]. 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 [11]. 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, (2) logical association, (3) temporal association, (4) procedural association, (5) communicational association, (6) sequential association, and (7) functional association.

When a pair of processing elements exhibit more than one association 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.

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 [12]. However, SMC Cohesion defines an intuitive notion of the cohesion 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

We define six relations between a pair of output components based on the IODG:

1. **Coincidental (R_1):** $R_1(o_1, o_2) = \neg(o_1 \rightarrow o_2) \wedge \neg(o_2 \rightarrow o_1) \wedge \neg\exists x\, [(x \rightarrow o_1) \wedge (x \rightarrow o_2)]$. Module outputs o_1 and o_2 do not have dependence relationship with each other, nor dependence on a common input.

2. **Conditional (R_2):** $R_1(o_1, o_2) = \exists x\, [((x \stackrel{c}{\rightarrow} o_1) \wedge (x \stackrel{c}{\rightarrow} o_2)) \vee ((x \stackrel{c}{\rightarrow} o_1) \wedge (x \stackrel{i}{\rightarrow} o_2)) \vee ((x \stackrel{i}{\rightarrow} o_1) \wedge (x \stackrel{c}{\rightarrow} o_2))]$. 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 (R_3):** $R_1(o_1, o_2) = \exists x\, [(x \stackrel{i}{\rightarrow} o_1) \wedge (x \stackrel{i}{\rightarrow} o_2)]$. Two outputs are i-control dependent on a common input.

4. **Communicational (R_4):** $R_1(o_1, o_2) = \exists x\, [((x \stackrel{d}{\rightarrow} o_1) \wedge (x \rightarrow o_2)) \vee ((x \stackrel{d}{\rightarrow} o_1) \wedge (x \rightarrow o_2))]$. 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 overlap.

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.

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 2 shows eight basic restructuring operations using the IODG. Figure 2 (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 2 (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 2 (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 2 (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

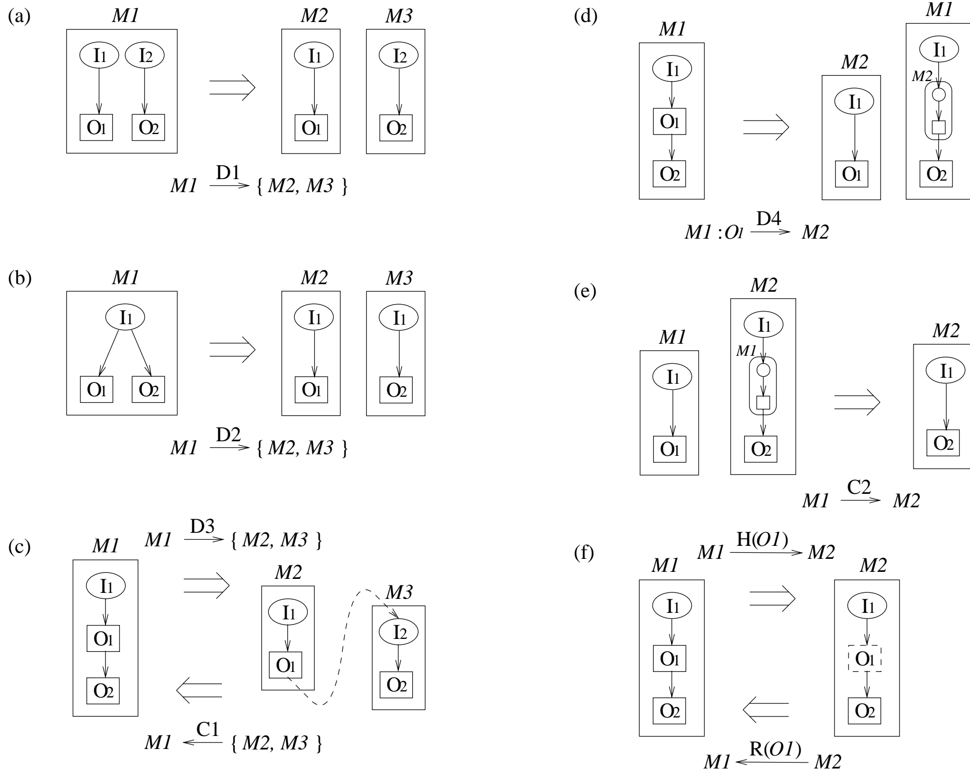

Figure 2: Eight basic operations for module restructuring.

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 2 (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 2 (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.

4.2 A Restructuring Process

A sequence of restructuring operations can be applied to improve the design structure of software system:

1. Generate IODG's of the modules of interest.
2. Compute the DLC level from each IODG.
3. Locate the modules with low DLC levels and determine the poorly-designed modules among them. Modules with multiple independent functions will be identified.
4. Decompose the IODG of each poorly-designed module as follows:
   (a) Partition the output components of the IODG so that when decomposed according to the partition, each resulting IODG has a higher DLC level. The IODG and DLC measure guides the partitioning process. The

DLC measure indicates the weakest connection among outputs.

(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.

Step 4 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 for each IODG.

4.3 Restructuring Example

Figure 3 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 4. 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.

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 formal rules for designating a cohesion level which preserve the intent of the SMC Cohesion [9]. The associative principles of SMC Cohesion are transformed to relate the output variables based on their data dependence relationships. A tool can automatically perform the classification. However, the technique can be applied only after the coding stage since it is defined upon the implementation details.

Bieman and Ott develop cohesion measures that indicate how close a module approaches the ideal of functional cohesion [3]. Three measures of functional cohesion are based on "data slices" of a procedure and satisfy the requirements of an ordinal scale. The functional cohesion measures are formally defined, and cohesion measurement tools have been built. These measures also depend on the implementation details of a module.

Kim, Kwon, and Chung introduce restructuring methods where module strength (cohesion) is used as a criterion to restructure modules [8]. A *processing blocks* is a group of data tokens with data or control dependence relationships with an output variable. A rule recognizes 'logically associated' module functions which together depend on an output, and are considered a processing block. Unfortunately, logically associated functions cannot always be automatically detected by analyzing program code.

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 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. 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

We report the following progress towards improving the ability to make objective software design decisions:

1. We define the IODG model that represents a design-level view of a module. The IODG is based on the dependency relationships between module

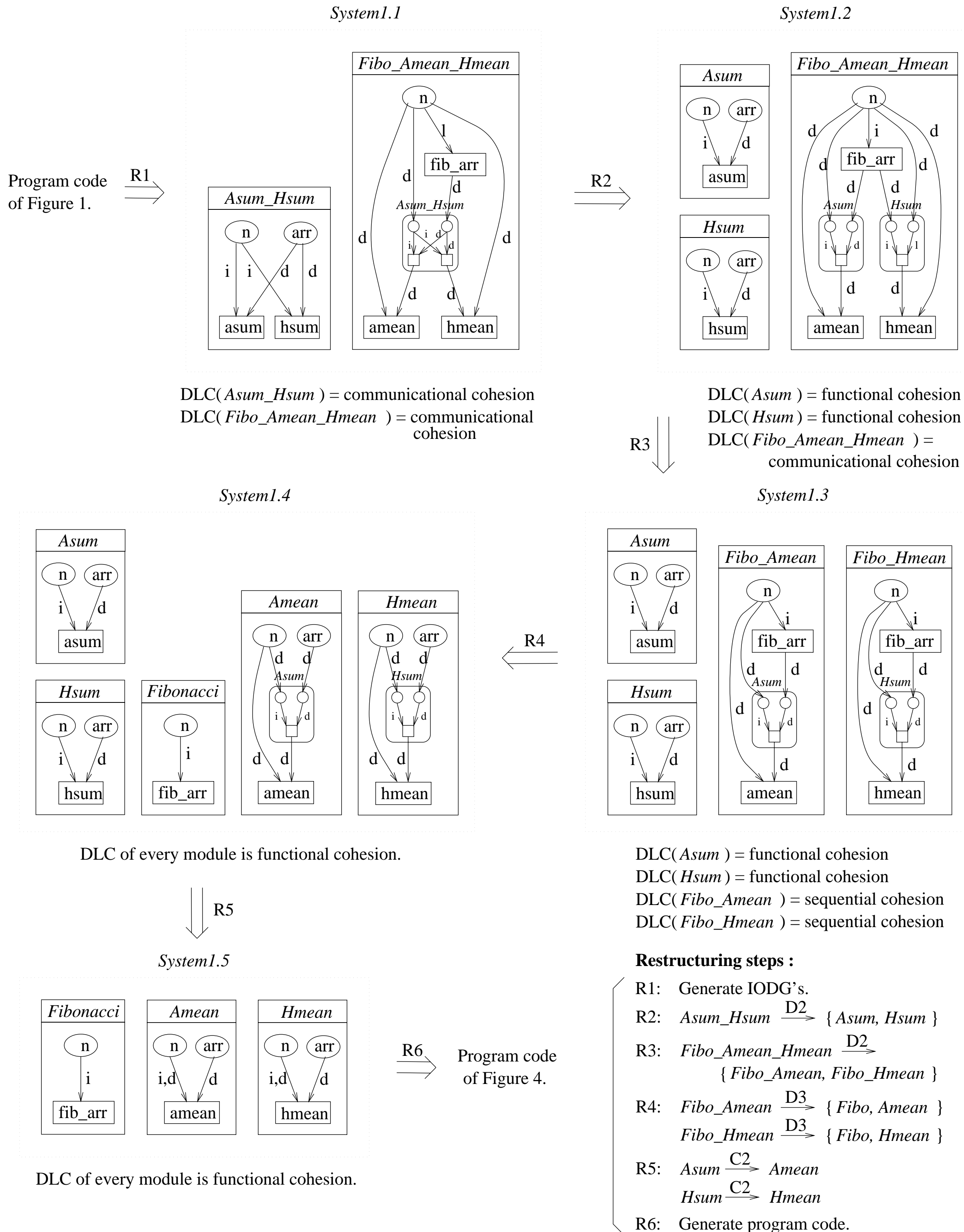

Figure 3: Restructuring procedures *Asum_Hsum* and *Fibo_Amean_Hmean* of Figure 1.

```
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 4: Procedures produced after restructuring the procedures of Figure 1.

inputs and outputs. It can be used to graphically visualize the design structure 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 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.

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.

References

[1] A. Albrecht and J. Gaffney. Software function, source lines of code, and development effort prediction. *IEEE Trans. Software Eng.*, 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 & Software*, 12(3):277–281, July 1990.

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

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

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

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

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

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

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

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

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

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

[13] H. Zima & B. Chapman. *Supercompilers for Parallel & Vector Computers.* Addison-Wesley 1991.