Research

A Quantitative Framework for Software Restructuring

BYUNG-KYOO KANG[1] and JAMES M. BIEMAN[2]*
[1] *Technology Deployment International, Inc., Santa Clara CA 95054 U.S.A.*
[2] *Department of Computer Science, Colorado State University, Fort Collins CO 80523, U.S.A.*

SUMMARY
Many existing software systems can benefit from restructuring to reduce maintenance cost and improve reusability. Yet, intuition-based, *ad hoc* restructuring can be difficult and expensive, and can even make software structure worse. We introduce a quantitative framework for software restructuring. In the framework, restructuring decisions are guided by visualized design information and objective criteria. The design information can be extracted directly from code to restructure existing or legacy software. Criteria for comparing alternative design structures include measures of design-level cohesion and coupling. Restructuring is accomplished through a series of decomposition and composition operations which increase the cohesion and/or decrease the coupling of individual system components. An example and a case study demonstrate the framework. The framework insures that restructuring results in measurable improvements in design quality.

J. Softw. Maint., **11**(0), 000–000, (1999)

No. of Figures: 12. No. of Tables: 3. No. of References: 50.

KEY WORDS: software restructuring; software re-engineering; cohesion; coupling; software measurement and metrics; software design

Contract/grant sponsor: NASA Langley Research Center; Contract/grant number: NAG1-1461.

*Correspondence to: Dr. James M. Bieman, Department of Computer Science, Colorado State University, Fort Collins CO 80523, U. S. A. Email: bieman@cs.colostate.edu

1. INTRODUCTION

Software restructuring is the process of re-organizing the logical structure of existing software systems in order to improve particular quality attributes of software products (Arnold, 1989). Some examples of software restructuring are improving coding style, editing documentation, transforming program components (renaming variables, moving expressions, abstracting functions, so on), and enhancing functional structures (relocating functional components into other or new modules).

Poor software structure results from an immature process, improper prototyping, overly optimistic deadlines, etc. (Arnold, 1989). Even the structure of well designed software tends to deteriorate as it is maintained over time (Lehman, 1980). The restructuring of old and new software systems can potentially make them easier to understand, change, and reuse. Restructured software can reduce costs to the user community by containing fewer bugs, and by allowing for quick responses to user requests for system changes. Other savings include reduced maintenance costs, increased component reuse, and extended software lifetimes (Arnold, 1989).

Commercial software systems are generally far too large and complex to be effectively restructured on an *ad hoc* basis. Analysts need rigorous techniques and tools to restructure large software systems. Ideally, restructuring should be done during design so that design alternatives can be evaluated before coding. However, existing systems also can be restructured to recover lost design integrity. Thus, analysts can make use of restructuring tools and techniques that can be applied to both designs and implementations.

Griswold and Notkin demonstrate one approach to help analysts restructure existing software (Griswold and Notkin, 1993 and 1995). After an analyst makes a local change to a program, a tool makes the necessary non-local changes. The tool requires implementation information, and does not help an analyst decide initially what to modify or what change to make. Kim, Kwok, and Chung (1994), Choi and Scacchi (1990), and Tesch and Klein (1991) develop heuristics to help analysts make restructuring decisions. Our objective is to develop restructuring criteria and operations that are rigorously defined, can be applied to software designs, and can be readily automated. Structural measures can predict future maintenance costs (Ferneley, 1999). Thus, we use measures of design structure to steer the restructuring process.

In preliminary work, we show how a model of intramodular dependencies can be used to visually display module structure and to quantify design-level cohesion (Kang and Bieman, 1996 and 1998). We also propose a set of restructuring operations which are defined in terms of the model. In Bieman and Kang (1998), we evaluate these and other design-level cohesion measures and show that they can predict code level attributes.

This paper presents a quantitative framework for software restructuring. The framework has three key elements: models of software designs, measurement based restructuring criteria, and a process for restructuring. The model, measures, and operations defined in our prior papers (Bieman and Kang, 1998; Kang and Bieman, 1996 and 1998) fit the requirements of the framework. The framework allows other models, measures, and operations. Here, we introduce a model and measure of intermodule connections into the framework.

Framework restructuring models will generally use graphs to represent module components and the connections between modules. These graphs can be readily displayed in a visual form. To apply the restructuring framework to existing code, design information is extracted from the code.

Since our focus is on structure, we use two commonly referenced structural attributes as restructuring criteria: *cohesion* and *coupling*. Software *cohesion* or *strength* refers to the relatedness of a module's components. Cohesive modules are difficult to split into separate components. Software *coupling* refers to the connectedness of modules. A module with high coupling has many connections with other modules. A commonly used software design heuristic is to design modules with high cohesion and low coupling (Stevens, Myers and Constantine, 1974). Thus, one restructuring objective is to increase cohesion and reduce coupling. We use a focused set of cohesion and coupling measures as our restructuring criteria. These measures are defined in terms of the restructuring models, and their definitions follow the requirements of measurement theory (Baker *et al.*, 1990; Fenton and Pfleeger, 1997; Fenton, 1994).

The restructuring process consists of a sequence of decomposition and composition operations; the measures are criteria to determine whether or not a given module should be restructured. A set of tools support the restructuring framework which includes a tool to extract design information from program code, cohesion and coupling measurement tools, and a program-slicing tool. An example and a case study demonstrate the restructuring process.

Our current restructuring framework supports the restructuring of procedural software, such as software written in C, COBOL, FORTRAN, or Pascal. The support tools are designed to help restructure programs implemented in C. We do not attempt to convert procedural software into object-based or object-oriented software. Rather, the restructuring presented here aims to improve design characteristics without changing the design paradigm. In this work, we use the term 'module' to refer to individual procedures or functions, which are the logical units in C programs.

The rest of this paper is organized as follows. Section 2 outlines the framework for software restructuring. Section 3 defines the models used to represent program units and their connections. We use the models to derive and validate design-level cohesion and coupling measures in Section 4. In Section 5, we describe a set of restructuring operations and a restructuring process. Section 6 describes our restructuring support tools. In Section 7, we demonstrate the process through its application on a software system. Section 8 reviews related work, and our conclusion are provided in Section 9.

2. PROFILE OF A SOFTWARE RESTRUCTURING FRAMEWORK

Our restructuring framework is based on four principles:

1. A practitioner should be provided only with the information needed for restructuring; unnecessary details should be hidden.

2. Restructuring should be based on objective criteria for comparing alternative structures.

3. The restructured programs should be 'functionally' equivalent to the original programs, because restructuring should not change, delete, or add functions.

4. The restructuring process should be automated, because it should be possible to build a tool to perform low-level activities corresponding to a user's high-level decisions.

The restructuring framework provides a practitioner with abstracted design information derived from program code—the module interface information needed for restructuring.

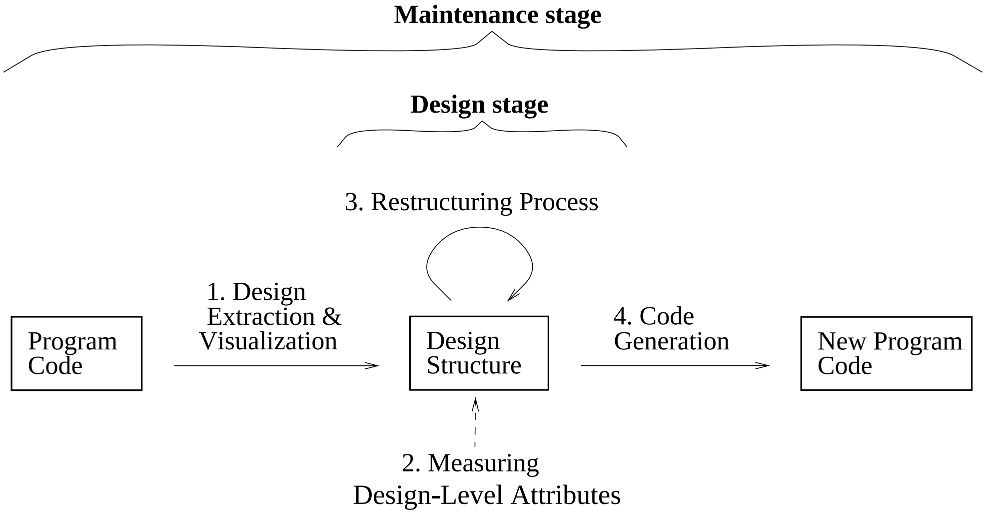


Figure 1. A quantitative restructuring framework

Other details are hidden. Restructuring choices are guided by quantitative measures and visualized representations. The restructuring process consists of a series of semantic preserving decompositions and compositions of 'processing elements'. Thus, the functions of a system are preserved during the process. Program code and its abstracted information are represented by control and data dependence relations between program components. Dependencies between components can be determined by data flow analysis using a compiler-like tool when an implementation is available.

Figure 1 shows such a framework for software restructuring. The framework includes four major activities: (1) Design information is extracted from program code and represented in a visualized graph form, (2) the design structure is evaluated objectively by design-level measures together with the visualized design information, (3) the design is restructured based on the evaluation, and (4) restructured code is generated from the restructured design. Step (2) and (3) are repeated until the evaluation of the resulting design structure is accepted. The design structure is represented by module interface information which is available during design. The software restructuring framework satisfies the following requirements; all are covered in detail in the later sections of this paper.

1. The restructuring process can be applied during design, development, and maintenance.

2. Module design information can be captured and extracted from module code. We can automate the extraction of information.

3. The restructuring process manipulates only design information rather than implementation details. We restructure design structures by decomposing and/or composing them.

4. The design information (design structure) is represented in a visual form. We can thus visualize the whole restructuring process to help us understand the design structure

and evaluate alternative designs. A software tool can construct a visual representation from a design.

5. Restructured code can be generated from the restructured design.

6. The restructuring decisions are based on objective quantitative criteria for comparing alternative design structures, as well as human intuition guided by graphical representation of modules.

7. The restructuring process can be automated. We can automate design extraction, design visualization, design-level measures, design decomposition/composition, and code generation.

3. SOFTWARE DESIGN ABSTRACTIONS

3.1. A Model to Represent Modules

Techniques that help analysts extract design and structure information can support software reuse (Choi and Scacchi, 1990; Lano and Haughton, 1992; Müller *et al.*, 1993) by helping to locate reusable software components from existing (or legacy) systems (Calliss and Cornelius, 1989; Esteva and Reynolds, 1991; Ning, Engberts and Kozaczynski., 1993). Abstract models represent modules and relationships between modules, and can serve as a basis for measurement (Baker *et al.*, 1990; Gustafson, Tan and Weaver, 1993).

An input-output dependence graph (IODG) models data and control dependence relationships between module interface components (Bieman and Kang, 1998; Kang and Bieman, 1996 and 1998). The IODG is adapted from the *variable dependence graph* introduced by Lakhotia (Lakhotia, 1993). Since in this work, a module is a procedure or function, the interface components are procedure/function inputs and outputs. Input components are values imported either as parameters or referenced global entities. Output components are values exported either through reference parameters, values returned by functions, and modified globals. A composite component, such as an array, linked list, or record, is treated as a single component.

We make use of *data dependence* and two types of control dependence: *c-control dependence* and *i-control dependence*, which are derived from the notion of control dependence used in program dependence graphs (PDG's) (Ottenstein and Ottenstein, 1984). The following definition assumes the existence of a PDG.

Definition 1 Dependence Classifications

Data Dependence: Output y *has a* data dependence *on input or output* x $(x \xrightarrow{d} y)$ *if* x *'reaches'* y *through a dependence path consisting only of data dependence edges in a PDG.*

C-Control Dependence: Output y *has* c-control dependence *on input or output* x $(x \xrightarrow{c} y)$ *if a dependence path between* x *and* y *in a PDG contains a control dependence edge, but no control dependence edges where the source variable* *is used in a predicate of a loop structure. A c-control dependence between an input and an output indicates that the output value is controlled by the input value through a decision structure.*

I-Control Dependence: Output y *has* i-control dependence *on input or output* x *($x \xrightarrow{i} y$) if a dependence path in a PDG between* x *and* y *contains a control dependence edge which includes a variable used in a predicate of a loop structure. 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 determined directly or indirectly by the input.*

I/O Direct Dependence: Output y *has data, c-control, and/or i-control* I/O direct dependence *on input or output x if y has a data dependence on x, and there is no other input or output on the dependence path between x and y.*

Definition 1 is simplified from our prior work (Kang and Bieman, 1996 and 1998; Bieman and Kang, 1998). Compiler texts provide more formal definitions of the notion of dependence (Zima and Chapman, 1991, pp. 112–172). The IODG is defined in terms of the input output components and their dependencies.

Definition 2 Input-Output Dependence Graph (IODG).

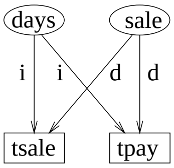
The IODG of module M *is a directed graph, G_M =* (V, E) *where* V *is a set of input-output components of* M*, and* E *is a set of edges labeled with dependence types such that $E = \{(x, y) \in V \times V \mid$* y *has data, c-control, and/or i-control I/O direct dependence on* x$\}$*. Each vertex in an IODG is labeled with one of five types: in-parameter, out-parameter, in-global, out-global, and function-return.*

The IODG is an abstraction of a PDG. It is a PDG without nodes and edges representing internal dependencies. In addition, the IODG classifies control dependencies as i-control or c-control. Figure 2 depicts two IODG's, one for procedure `Tsale_Tpay` and another for procedure `Sale_Pay_Profit`. `Sale_Pay_Profit` gets a sale record from a file and computes the pay and profit from the sale record by calling procedure `Tsale_Tpay`.

The figure shows that a graphical display of the IODG can make a module's interface easy to visualize. We extend the IODG to represent a caller-callee relationship by including the input-output dependence relationship of the callee inside the IODG of the caller. A circle represents a parameter input, and a square represents an output and function-return. A dashed circle indicates a global input and a dashed square represents an output. The texts in each circle and square are the names of input and output variables. Each arrow indicates the dependence between two components.

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 `days` and output `pay` of the IODG of `Sale_Pay_Profit`. We find three dependence paths between them: (1) `days` $\xrightarrow{d}$ `pay`, (2) `days` $\xrightarrow{d}$ `input_parameter` $\xrightarrow{i}$ `output_parameter` $\xrightarrow{d}$ `pay`, and (3) `days` $\xrightarrow{i}$ `sale` $\xrightarrow{d}$ `input_parameter` $\xrightarrow{d}$ `output_parameter` $\xrightarrow{d}$ `pay`. According to our dependence definitions,

```
procedure Tsale_Tpay
   ( days : integer;
     sale : int_array;
     var tsale : integer;
     var tpay : float );
var i : integer;
begin
   tsail := 0;
   tpay := 0;
   for i := 1 to days do begin
      tsale := tsale + sale[i];
      tpay := tpay + 0.1 * sale[i];
      if sale[i] > 1000 then
         tpay := tpay + 50;
   end;
end;
```




```
procedure Sale_Pay_Profit
   ( days: integer;  file: text;
     cost: float;
     var sale : int_array;
     var pay : float;
     var profit : float );
var i, tsale : int;
     tpay : float;
begin
   reset(file);  i := 0;
   while i > days do begin
      i := i + 1;
      readln(file, sale[i])
   end;
   Tsale_Tpay(days, sale, tsale, tpay);
   pay := tpay / days + 100;
   profit := 0.9 * tsale - cost;
end;
```

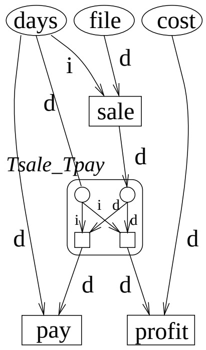


Figure 2. Input-output dependence graph representation for `Tsale_Tpay` *and* `Sale_Pay_Profit`

`pay` has data and i-control dependencies on `days`. Indirect dependences are implied by a sequence of direct dependences. The IODG of `Sale_Pay_Profit` in Figure 2 shows that the direct and indirect dependence relationship between three inputs and two outputs.

3.2. A Model to Represent Relationships between Modules

Several models capture the relationship between modules. A well known model is a *call-graph* which represents a call relationship between modules. Weiser defines a concept of *interprocedural dependences* using control and data dependencies in the context of producing *slices* of programs (Weiser, 1994). The interprocedural slices do not include the call relationship between procedures. Loyall and Mathisen define a model that captures inter-procedural control and data dependences (Loyall and Mathisen, 1993). They use the model to perform dependence analysis for determining effects of software modifications and support automated test and coverage analysis.

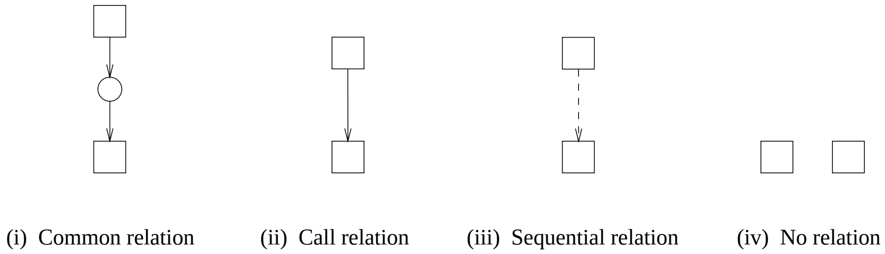
We model the relationship between modules in a system with a *module interconnection graph (MIG)*. The model is based on the control, data flow relationship, and call relationship between modules of a system. In the model, there are four types of relationships between two modules:

1. **common relation:** a module sends information to another through a global component; a module writes a variable, a compound variable, or data file, which in turn is read by another module,

2. **call relation:** a module imports another module's computation to execute its functions; a module calls another module,

3. **sequential relation:** an output of a module is passed to another module as an input; an output of a module is used as an input of another module, and

(a)



(b)

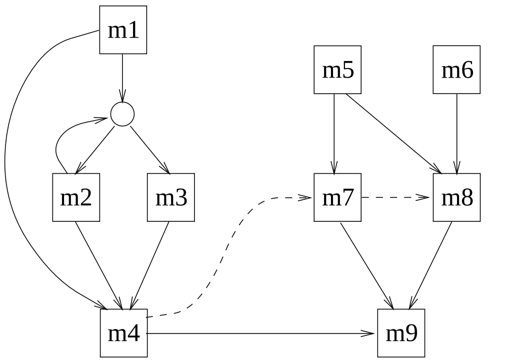


Figure 3. (a) Four types of module relationships; (b) Example module interconnection graph (MIG)

4. **no relation:** two modules do not have any of above relations.

Figure 3(a) shows graphical representations of these relation types between modules, where a module is represented by a square and a global component by a circle. Thus, in Figure 3(a),(i) an arrow from a square to a circle indicates that a module corresponding to the square writes to a global component, and an arrow from a circle to a square indicates that a module corresponding to the square reads from a global component, (ii) a solid arrow from square A to square B indicates that module A is called by module B, (iii) a dashed arrow from square A to square B indicates that an output of module A is used as an input of module B, and (iv) no connection between squares indicates that there is no relation between modules.

Multiple parameters connecting two modules are treated as one connection since additional parameters do not affect decisions concerning module composition. For the same reason, passing multiple global components between two modules is treated as one connection. Multiple function calls between two modules are also represented as one connection since the additional function calls do not increase the need to compose the modules.

The MIG of a module is a directed graph, where a node represents a module or a global component, and an edge between two nodes represents one of three relations: common, call, or sequential relation.

Definition 3 Module Interconnection Graph (MIG).

The MIG of a system S *is a directed graph,* G_S = (V, E) *where* $V = V_1 \cup V_2$, $E = E_1 \cup E_2 \cup E_3 \cup E_4$ *such that*

$V_1 =$ *a set of modules of* S,
$V_2 =$ *a set of global components of* S,
$E_1 = \{(x, y) \in V_1 \times V_1 \mid x$ *is invoked by* y *through a function call (call relation)*$\}$,
$E_2 = \{(x, y) \in V_1 \times V_1 \mid$ *an input of* y *has a dependence on an output of* x *(sequential relation)*$\}$,
$E_3 = \{(x, y) \in V_1 \times V_2 \mid x$ *writes to* $y\}$, *and*
$E_4 = \{(x, y) \in V_2 \times V_1 \mid y$ *reads* $x\}$.

When a MIG is displayed graphically, an edge E_2 (sequential relation) is represented by a dashed line and the other edges by solid lines. The MIG can be generated automatically from program code or can be determined during the design stage. When program code is available, the relation types can be determined by dependence analysis using a compiler-like tool. During the design stage, the information about global components, function-calls, and caller-callee relationships should be available.

Figure 3(b) shows an example of a MIG representation of a system. The MIG shows the relationship between modules of a system. In its graphical form, the MIG visually displays the system's organization. This representation is used to define design-level coupling measures and is applied to software restructuring.

Analysts can use the visualized IODG and MIG representations to help them understand the design structure of program. The IODG shows input-output components of a module. It also shows how the input-output components are related through dependence information. The MIG shows system components, modules and global structures, and the relationship between the components. The IODG and MIG representations can be used to recapture designs from existing, possibly legacy, systems.

When program size is large or program structure is complex, program understanding by reading code is a painful process. It is not easy to glean program functionalities by reading program code. The IODG and MIG information is much more compact than program code. IODG size is not proportional to program size; it is related to the size of a procedure's interface, not its body.

4. DESIGN MEASURES

4.1. Measuring Design Cohesion

Design-level cohesion and coupling measures guide restructuring in the proposed framework. Measures identify modules that may be restructured by decomposition—splitting into two or more modules—or by composition—merging with other modules. We develop a set of design-level cohesion and coupling measures based on the IODG and MIG design-level module abstractions. The *representation condition* of measurement theory requires that, in deriving a measure of an attribute, the empirical relations of the attribute should be preserved in the numerical relation system (Fenton, 1994). We first define intuitive relations and derive measures in terms of those relations. This approach is consistent with the model-order-mapping paradigm for specifying software measures (Gustafson, Tan and Weaver, 1993).

Stevens, Myers, and Constantine (1974) define software cohesion ("SMC Cohesion") as a property that can be used to help determine if the components of a module actually belong together. Based on the properties of the associations between each pair of processing elements in a module, SMC Cohesion defines seven cohesion levels on an ordinal scale: *coincidental, temporal, procedural, communicational, sequential,* and *function cohesion.* Experts use their intuition to determine the SMC Cohesion of a component. Because of this dependence on intuitive judgement, SMC Cohesion cannot be readily applied to measure cohesion in practice (Nandigam, Lakhotia and Čech 1999; Woodward, 1993).

In prior work, we applied SMC Cohesion directly to the IODG of a module to derive a design-level cohesion (DLC) measure (Kang and Bieman, 1996 and 1998) satisfies the representation condition of measurement (Fenton and Pfleeger, 1997, pp. 23–72). The DLC measure derivation is based on the IODG; we use the approach used to define SMC Cohesion and by Lakhotia to define module cohesion (Lakhotia, 1993). Here, we briefly define the DLC measure. Detailed derivations of DLC are found in prior papers (Bieman and Kang, 1998; Kang and Bieman, 1998).

The DLC measure is ordinal; a module exhibits one of six cohesion "levels." These levels are based on six relations between module output pairs. The levels and relations are designed to be consistent with the difficulty of splitting a module.

Definition 4 Module Output Relations.

1. **Coincidental relation (R_1):** *Two outputs have no dependence relationship with each other and no dependence relation on a common input (Lakhotia, 1993).*
2. **Conditional relation (R_2):** *Two outputs are c-control dependent on the same input.*
3. **Iterative relation (R_3):** *Two outputs are i-control dependent on the same input.*
4. **Communicational relation (R_4):** *Two outputs are dependent on a common input. This input is used to compute both outputs, but the two outputs are not c-control or i-control dependent on this input (Lakhotia, 1993).*
5. **Sequential relation (R_5):** *One output is dependent on the other output (Lakhotia, 1993).*
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. Like Lakhotia's module cohesion measure (Lakhotia, 1993), DLC measurement is based on the pair of outputs with the weakest association.

Definition 5 Design-Level Cohesion (DLC) Measure Calculation.

Determine the strongest relation for each pair of outputs. The DLC value of the module is the weakest (lowest level) relation of the set of strongest relations for each pair.

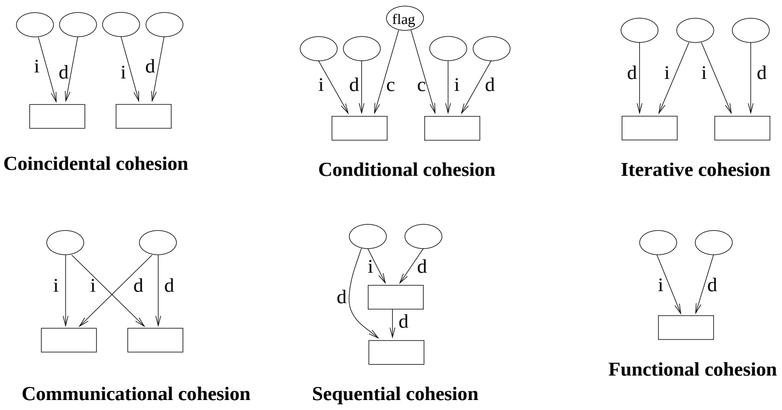


Figure 4. Sample IODG's for each DLC level

We apply the DLC Measure to the IODG's of Figure 2. Outputs `tsale` and `tpay` of module `Tsale_Tpay` have iterative and communicational relations. Since the communicational relation is stronger than the iterative relation, the cohesion level of module `Tsale_Tpay` is communicational cohesion. Module `Sale_Pay_Profit` has three pairs of outputs. The output pair `sale` and `pay` has three relations: iterative, communicational, and sequential. Since the sequential relation is the strongest, the pair has a sequential relation. Similarly, the output pair `sale` and `profit` has a sequential relation, and the output pair `pay` and `profit` has a communicational relation. Since the communicational relation is the weakest among the relations of all pairs, the entire module exhibits communicational cohesion.

Figure 4 graphically depicts six IODG's, one for each of the six cohesion levels. Edges are labeled to indicate the dependence relations. The figure demonstrates the relationship between the ordering of the cohesion levels and the relative "splittability" of a module.

4.2. Measuring Coupling

4.2.1. Myers' Coupling Relations

Coupling is defined as the degree of "interaction" or "interdependence" between modules (Myers, 1978, p. 41), (Pressman, 1997, p. 359). A common design principle is that coupling should be kept low to reduce the system complexity (Stevens, Myers and Constantine, 1974).

To perform the restructuring process systematically, we want to solve local problems first and then global ones. Thus, in our restructuring framework, restructuring is applied first to individual modules and then applied to an increasingly larger set of modules. To fit this "bottom-up" restructuring approach, we want to measure coupling between pairs of modules, rather than the coupling of entire systems of modules. We derive a coupling measure using the intuition provided by Myers (Myers, 1978, pp. 41–56), who defines six levels of coupling. Our coupling measure is defined in terms of the IODG abstraction.

Using Myers' approach, coupling levels are determined by inspecting the type of interaction between two modules. The levels of interaction range from the strongest coupling to the weakest:

1. Content Coupling: One module references the contents of the other, i.e., one branches into a label of the other, modifies a statement of the other, or changes local data of the other.

2. Common Coupling: Two modules have access to the same global data.

3. Control Coupling: One module passes a flag as parameter to control the logic of the other.

4. Stamp Coupling: Two modules use a common data structure but may operate on different portions of the data structure.

5. Data Coupling: One module passes simple data or a data structure as a parameter to the other. All components of the data structure are used by the called module.

6. No Coupling: There is no communication between two modules.

A general design goal is to avoid the strongest forms of coupling. Myers' coupling, like SMC cohesion, is difficult to determine because of its subjective nature (Woodward, 1993). We use Myers' coupling levels as an empirical relations system to validate our design-level coupling measure.

4.2.2. A Design-Level Coupling (DCP) Measure

Our DCP measure is defined by examining coupling exhibited by the MIG and IODG representations of modules following a similar approach to that used by Myers. The coupling between two modules is classified by five relations which are ordered from strongest to weakest:

1. **Common Coupling:** In the MIG representation, two modules have a common relation when one module writes to a global data and the other reads from the global data. In the IODG representation, common coupling exists when two IODG's contain the same global data and one IODG has the global data as an input component and the other IODG has it as an output component.

2. **Conditional Coupling:** Conditional coupling occurs when, in the MIG representation, two modules have a call relation and in the IODG representation, one module passes a parameter to another module, and the parameter has c-control dependence on an output component. Conditional coupling includes Myers' control coupling since a flag is implemented as an 'if-then-else' predicate component and a flag in an IODG representation always has 'c-control dependence' on an output. However, since a component with c-control dependence is not always a flag, DCP conditional coupling includes some cases of Myers' stamp, and data coupling.

3. **Computational Coupling:** Computational coupling occurs when, in the MIG representation, two modules have a call relation and in the IODG representation, one module passes a parameter to another module, and the parameter has i-control or data dependence on an output component. Computational coupling corresponds to both Myers' stamp and data coupling, since whenever data or a data structure that is

not used as a flag is passed through parameter, it has i-control or data dependence. We cannot differentiate between a data structure and a simple data input, because in an IODG an input-output component is represented by its name but not by its contents.

4. **Sequential Coupling:** Two modules have a sequential coupling when, in the MIG representation, outputs of one module are used as inputs of the other module (sequential relation). In the IODG representation, the output components of an IODG are input components of the other IODG. Sequential coupling is classified as no-coupling by Myers since there is no connection through parameter passing or global data. Thus, the connection strength is weaker than Myers' data coupling but stronger than or the same as Myers' no coupling.

5. **No Coupling:** Two modules have no coupling when there are no connections between the MIG representation of two modules. There is no coupling when, in the IODG representation, there is no global data sharing and no function-calls between two modules.

A pair of modules may have several connections. Since we want to apply the coupling measure to software restructuring, we represent multiple connections between modules by the corresponding multiple coupling levels of the modules. Fenton and Melton merge both the connection kind and the number of connections between a pair of modules in their definition of a coupling measure (Fenton and Melton, 1990). They choose the strongest (least desirable) level of coupling when there are multiple connections between two modules.

We cannot apply Myers' content coupling in our framework since the IODG cannot represent direct references to the contents of another module (content coupling primarily occurs in assembly language programming). We introduce a new level of coupling, sequential coupling, where the output of one module is used as input of the other. Though the two modules are independent when inspecting them in isolation, they have a caller-callee relationship from the perspective of the control module.

We can always determine the DCP measure based on the MIG and IODG information. When program code is available, the MIG and IODG can be generated automatically from the analysis of control and data dependence between program components using a compiler-like tool. Without code, a designer must specify the dependencies between input and output components of IODG and the relationship between modules of MIG.

Since the derivation of the DCP measure is consistent with the intuition used to define Myers' Coupling, we assume the following ordering of the five DCP relation levels:

$$Common \; > \; Conditional \geq \; Computational \; > \; Sequential \; \geq \; No\,Coupling \qquad (1)$$

The DCP measure is on an ordinal scale as long as we accept the ordering implied by the relations of Myers' Coupling.

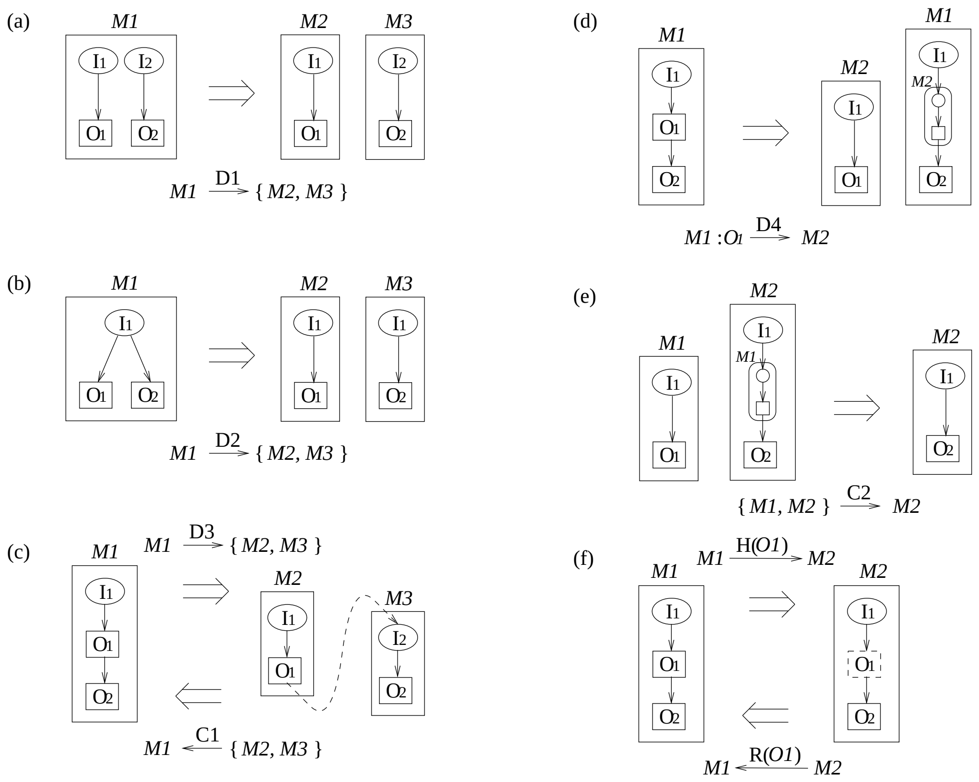


Figure 5. Restructuring Operations (from Kang and Bieman (1996)); edge labels are not included

5. SOFTWARE RESTRUCTURING

5.1. Basic Restructuring Operations

Restructuring is completed by applying restructuring operations. The restructuring process is guided by the DLC and DCP measures and the intuition provided by the graphically displayed IODG and MIG models.

Basic restructuring operations can either compose (merge), decompose (split), or reorganize modules. Individual decomposition operations can be applied to modules with specific DLC cohesion levels. Conversely, the application of composition operations depends on the DCP coupling of module pairs. Modules can also be restructured to hide components with greater visibility than necessary. We use eight basic operations, shown in Figure 5, to restructure modules (Kang and Bieman, 1996):

Coincidental Decomposition (D1): A module exhibiting coincidental cohesion has disjoint components—one or more groups of data tokens linked to individual outputs without any dependence relations on another group. These modules can be easily split by separating the disjoint groups as shown in Figure 5(a).

CIC Decomposition (D2): Modules with conditional, iterative, or communicational cohesion have one or more inputs tied to all of the outputs. These modules can be decomposed by copying all data tokens linked to more than one output as shown in Figure 5(b).

Sequential Decomposition A (D3): In modules with sequential cohesion one or more outputs depend on other outputs. Such modules can be decomposed by splitting the modules into two or more modules with sequential coupling, which is one of the most desired forms of coupling. The output of a module (producer module) is used as the input of the other (user module) as shown in Figure 5(c).

Sequential Composition (C1): The inverse of sequential decomposition is to compose two modules with sequential coupling (also shown in Figure 5(c)).

Sequential Decomposition B(D4): An alternative to operation D3, Sequential Decomposition A, is to replace an output component with a module call, and move the output component and the components that it depends on into a separate callee module as shown in Figure 5(d).

Caller/Callee Composition (C2): Two modules can be composed if they have a call relation, exhibit either conditional or computational coupling, and the callee has only one caller. The call statement is replaced by the tokens of the callee, and unnecessary coupling is reduced. Figure 5(e) shows operation C2.

Hide (H): The *hide* operation, shown in Figure 5(f), converts an exported output into a hidden local variable, when the exported output is not actually used externally.

Reveal (R): *Reveal*, also shown in Figure 5(f), is the inverse of hide. Using reveal, R(*M1*:*O1*), a local variable *O1* of *M1* is exported by changing the local variable into an output variable. Reveal can be used to separate a hidden function from a large module.

5.2. Restructuring Process

A software restructuring process provides guidance for determining when to apply individual restructuring operations. Systematic restructuring process can improve cohesion and coupling attributes of a system. An appropriate restructuring will: (1) locate the modules with low cohesion levels and determine which are the poorly-designed modules among them, (2) decompose the modules identified as a poorly-designed ones, and (3) locate overly decomposed (i.e., overmodularized) modules that increase the coupling of the system, and recompose them.

During restructuring, decomposition should precede composition because of the potential existence of *partially-used* modules—modules where each output component is used by different modules. The decomposition process converts a partially-used module into a fully-used module. Composition of fully-connected modules does not decrease module cohesion. To perform restructuring systematically, we solve local problems first and then global ones: restructuring is applied first to each module and then expanded to a larger set of modules.

An improvement in one software attribute can deteriorate other attributes. The optimal cohesion and coupling levels will depend on the application, the required reusability, readability, and maintainability of software. Restructuring should be performed objectively so that both cohesion and coupling measures of software are improved, and at least both measures do not become worse.

The six step restructuring process that we proposed in (Kang and Bieman, 1996 and 1998) can be applied to procedures `Tsale_Tpay` and `Sale_Pay_Profit` in Figure 2:

1. Generate IODG's of candidate modules. IODG's can be constructed solely from design information. If a formal design notation is used, they can be generated automatically by a tool. If the code for modules already exists, the information necessary to construct the IODG's can be extracted from the code using a compiler-like tool. Figure 2 shows the IODG's of procedures `Tsale_Tpay` and `Sale_Pay_Profit`. System 1.1 in Figure 6 also depicts these IODG's.

2. Compute DLC levels. The IODG's for procedures `Tsale_Tpay` and `Sale_Pay_Profit` in System 1.1 in Figure 6 both exhibit communicational cohesion.

3. Locate modules with low DLC levels and identify restructuring subjects. Low DLC levels indicate modules with multiple independent functions. If the policy is to aim for functional cohesion, restructure both `Tsale_Tpay` and `Sale_Pay_Profit` of System 1.1 in Figure 6. Both modules perform multiple functions.

4. Decompose the IODG of each identified restructuring subject using the following procedure:

   (a) Partition the output components of the IODG so each resulting IODG has a higher DLC level. An optimal partition can be identified by computing DLC values for all possible partitions, since the number of module output components is generally limited to a tractable number.

   (b) Decompose each IODG according to the partition of IODG outputs. Resulting IODG's include input-output components with dependence relations with the partitioned outputs.

   Use a bottom-up approach to decompose two IODG's with a caller-callee relationship—examine the callee first. Change the corresponding invocation in the caller to reflect the callee's decomposition, and then apply the decomposition to the caller. Thus, we decompose `Tsale_Tpay` before we decompose `Sale_Pay_Profit`.

   Repeat Step 4 until the DLC level of each resulting IODG is acceptable, or there are no further candidates for decomposition.

   In the example shown in Figure 6, `Tsale_Tpay` is called by `Sale_Pay_Profit`. Thus, the decomposition of procedure `Tsale_Tpay` affects the code of its caller `Sale_Pay_Profit`; a modified IODG of `Sale_Pay_Profit` reflects this change. We first decompose `Tsale_Tpay` using restructuring operation D2, CIC Decomposition, producing two modules, `Tsale` and `Tpay`. We also modify the caller, `Sale_Pay_Profit`, to invoke the modified callee. This restructuring step is shown as transition *R2* in Figure 6, producing System 1.2. Procedures `Tsale` and `Tpay` both exhibit functional cohesion, while the cohesion level of `Sale_Pay_Profit` remains unchanged.

   We apply restructuring operation D2 again, as shown in Figure 6 as transition *R3*. This time we split `Sale_Pay_Profit` into `Sale_Pay` and `Sale_Profit` producing *System 1.3*. Both new modules exhibit sequential cohesion.

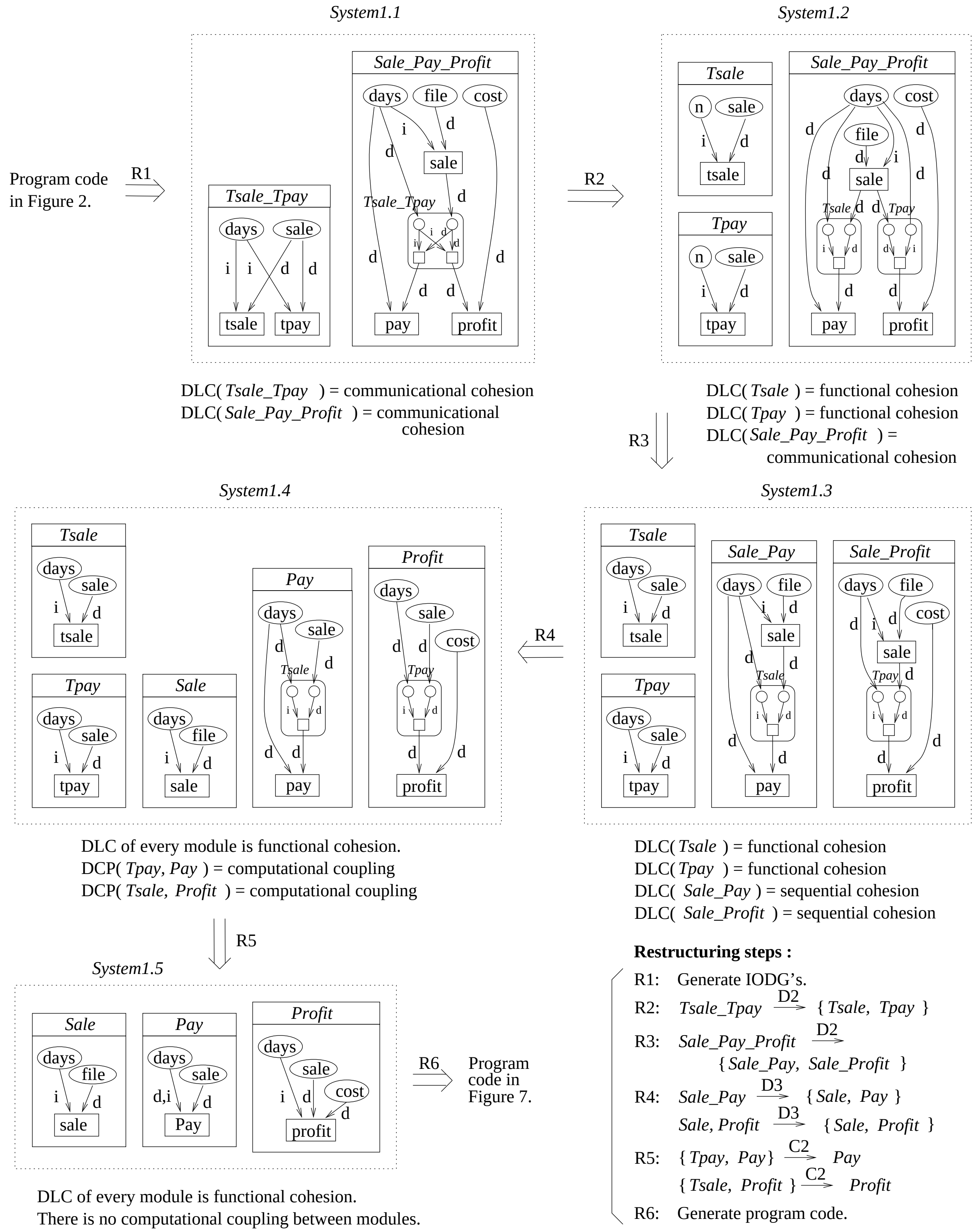


Figure 6. Restructuring procedures `Tsale_Tpay` *and* `Sale_Pay_Profit` *of Figure 2*

We repeat the process using operation D3 (transition *R4*) on both `Sale_Pay` and `Sale_Profit` producing modules `Sale`, `Pay`, and `Profit` in *System 1.4*. After completing transitions *R2, R3,* and *R4*, all of the modules exhibit functional cohesion. We now examine coupling to consider candidates for composition.

5. Locate unnecessarily decomposed modules and compose them. After repeated decompositions, a system can become overmodularized, resulting in too much interaction between modules and an increase in coupling. The DCP between each pair of modules can identify overmodularized modules. Apply composition operations to overmodularized modules only when composition does not decrease their cohesion levels and does not affect connections with other modules.

   We apply the DCP measure to the module pairs in *System 1.4* of Figure 6 and find computational coupling between procedure `Tsale` and `Profit`, and `Tpay` and `Pay`, and sequential coupling between procedure `Sale` and `Pay`, and between `Sale` and `Profit`. The computational coupling is removed by applying composition operator C2, Caller/Callee Composition, as shown in transition *R5* producing *System 1.5*. All of the modules in *System 1.5* exhibit functional cohesion, and there is no computational coupling.

6. Generate module code (if the original IODG's were generated from source code). A tool can automate code generation by using the data tokens and the dependence information obtained from the initial modules during step 1.

   Transition *R6* in Figure 6 represents the generation of new code for the example modules. The resulting restructured modules are given in Figure 7. At the start of the restructuring process, both modules exhibit communicational cohesion and computational coupling. The modules are restructured into three modules that exhibit functional cohesion, the strongest cohesion level and sequential coupling, the weakest coupling. The restructured modules should be easier to understand, maintain, and reuse.

Coupling plays a limited role in our framework since a composition of modules must consider other factors in addition to coupling. For example, future reuse: if a module is likely to be reused, we should not hide the module by composing into other modules. We use coupling as a supplementary measure and cohesion as a primary one.

At the start of the restructuring process, both modules in Figure 2 exhibit communicational cohesion and computational coupling. These two modules are restructured into three new modules. The restructured modules, shown in Figure 7, exhibit functional cohesion, the strongest cohesion level and sequential coupling, the weakest coupling. The restructured modules should be easier to understand, maintain, and reuse.

6. RESTRUCTURING TOOLS

Our software restructuring framework includes four major processes: (1) the design information of modules is extracted from module code and represented in a visualized graph form;

```
procedure Sale
   ( days: integer;
     file: text;
     var sale : int_array );
var i : integer;
begin
   reset(file);  i := 0;
   while i < days do begin
      i := i +1;
      readln(file, sale[i])
   end;
end;
```

```
procedure Pay
   ( days : integer;
     sale : int_array;
     var pay : float );
var i, tpay : integer;
begin
   tpay := 0;
   for i := 1 to days do begin
      tpay := pay + 0.1 * sale[i];
      if sale[i] > 1000 then
         tpay := tpay + 50;
   end;
   pay := tpay / days + 100;
end;
```

```
procedure Profit
   ( days : integer;
     cost: float;
     sale : int_array;
     var profit : float );
var i, tpay : integer;
begin
   tsale := 0;
   for i := 1 to days do begin
      tsale := sale + sale[i];
   end;
   profit := 0.9 * tsale - cost;
end;
```

Figure 7. Procedures produced after restructuring the procedures of Figure 2

(2) the design structure is evaluated objectively by design-level measures; (3) the design is restructured based on the measures; and (4) the restructured code is generated from the restructured design.

We have developed tools to support the restructuring process using `lex` and `yacc` from the gcc compiler. The tools process C programs in a UNIX workstation environment. They have been installed and tested for SUN SPARCstatations running SUN-OS and IBM RS6000 systems running AIX. Key components of the restructuring tool include the following:

1. *Control Flow Graph Generator:* Vertexes of this graph are module statements. C output variables are identified, including modified global variables, reference arguments, and function return values.

2. *Data Token Identifier:* Identifies all defined and used data tokens of each statement in a module, and all local variables and output variables.

3. *Dependency Analyzer:* Computes data dependencies and control dependencies for data tokens in module statements, and identifies control dependencies involving loop predicates.

4. *F-Slicer:* Generates program (also data) slice information for a module. A *program slice* is the portion of program text that affects a specified program variable (Weiser, 1994): Bieman and Ott (1994) defined a *data slice* for a data token as the sequence of all data tokens in the statements that comprise the “backward” and “forward” slices of the data token.

5. *IODG tool:* Generates IODG information with only input and output components using program slice information generated by the *F-Slicer*.

6. *DLC tool:* Computes the DLC measure using the IODG information.

7. *Decomposition Tool:* Performs decomposition using the slice information and IODG.

8. *Coupling Analyzer:* MIG information and a DCP tool can be easily constructed from call graph and data flow information. To build a composition tool, in addition to the call graph and data flow information, we need to consider the details of program constructs, such as declarations, globals, and variable names.

Our *FUNCO* tool set is a metrics analyzer for C programs that integrates the above tool components. Prototypes for components 1–6 are operational. We use information generated by the *F-Slicer* for component 7. However, the F-slicer is not designed to automatically generate source code slices, so the process is labor intensive. Existing slicing tools can potentially automate component 7 (Gallagher and Lyle, 1991; Jackson and Ladd, 1994). The current version of Unravel, a C-slicer, is capable of producing compilable slices (Lyle *et al.*, 1995), so it is a good candidate for component 7. A coupling analyzer is now under construction. FUNCO also includes an implementation of the functional cohesion measure defined by Bieman and Ott (Bieman and Ott, 1994). FUNCO has been installed and tested for SUN SPARC workstations running SUN-OS and IBM RS6000 systems running AIX. It is available for general use on the World Wide Web (Kang and Bieman, 1997).

7. RESTRUCTURING CASE STUDY

7.1. Limitations

We restructure a small software system to demonstrate the restructuring process and to evaluate how effectively the process can be applied to real software. The IODG tool generated IODG information from program code and the DLC tool measured DLC levels from the IODG information. We generated visual IODG representations manually from the IODG information to provide further intuitive information.

When this study was conducted, the available version of Unravel could not generate compilable slices. Thus, for this case study, we used the F-Slicer tool to help perform the decomposition operation. The F-Slicer tool provides program slice information from program code. Decomposition decisions were based on the IODG representation and DLC measurement. We generated the restructured program code by selecting a program slice or slices that correspond to one or more output components of each restructured IODG.

7.2. Applying the Restructuring Process

The data for the case study was the JASMIN software system (Bieman and Zhao, 1995). JASMIN is a research tool that collects data concerning the use of inheritance in C++ software. JASMIN measures the depth of the inheritance, number of classes with/without private sections, number of classes derived from classes with/without private sections, and number of private, public, and multiple inheritances. JASMIN contains 39 C functions and 4540 lines of source code. On average, each function has 4.69 inputs, 2.26 outputs, and 66.95 data tokens.

We applied the restructuring process to the JASMIN system following the steps given in Section :

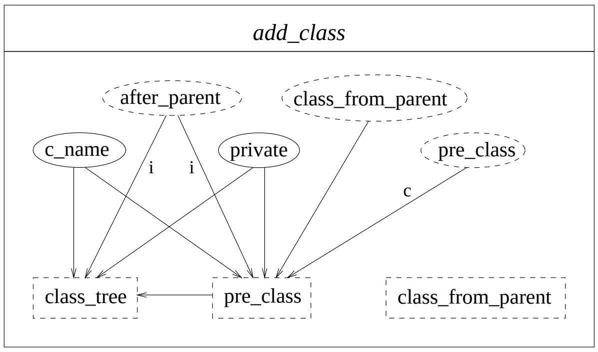


Figure 8. An IODG representation of function `add_class`

1. Generate IODG's: The IODG tool generated IODG information from JASMIN source code. Figure 8, Figure 9, and Figure 10 show example IODG representations for functions `add_class`, `print_info`, and `print_summary` respectively. The Appendix includes the program code for function `print_info`.

2. Compute DLC levels: The DLC tool computed DLC levels for each generated IODG. We found 22 functions with functional cohesion, 3 functions with sequential cohesion, 11 functions with communicational cohesion, and 3 functions with coincidental cohesion. No functions exhibit conditional or iterative cohesion. Actually, many JASMIN output components exhibit conditional or iterative *associations*. However, those associations are hidden by other stronger associations, such as communicational association.

3. Determine restructuring candidates among functions with low DLC levels: Among the three coincidental functions, one function initializes a tree data structure, and related global variables. This function actually exhibits *temporal cohesion* according to the original SMC Cohesion criteria (Stevens, Myers and Constantine, 1974), which is a cohesion category not supported by the DLC measure. Our examination eliminates this function from the restructuring process. Both of the two remaining coincidental functions, `add_class` and `tg_add_class`, contain two related outputs and an isolated output that is used for computation control purpose. We restructure these two coincidental functions since they unnecessarily include a control variable.

   Among the 11 functions exhibiting communicational cohesion, two functions, `print_info` and `print_summary`, include loosely connected outputs—they generate output files that are not tightly related to each other. We restructure these functions because of the weak relationship between their output files. Figure 8, 9, and 10 show IODG's for three of the restructuring candidates.

4. Decompose the IODG of each module identified as a poorly-designed one: We apply appropriate decomposition operations:

$$R1 : \texttt{add\_class} \xrightarrow{D1} \{\texttt{add\_class}, \texttt{temp}\}$$
$$R2 : \texttt{tg\_add\_class} \xrightarrow{D2} \{\texttt{tg\_add\_class}, \texttt{tg\_temp}\}$$
$$R3 : \texttt{print\_info} \xrightarrow{D3} \{\texttt{print\_inh\_info}, \texttt{print\_output}\}$$

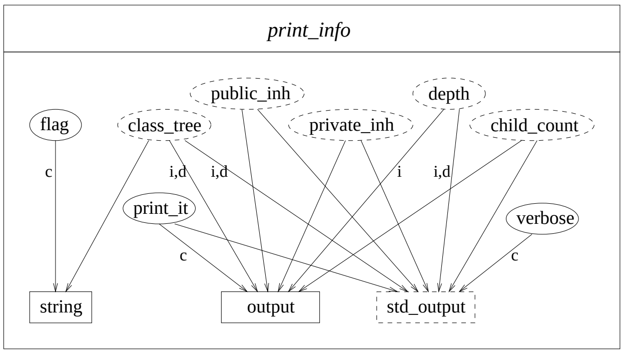


Figure 9. An IODG representation of function `print_info`

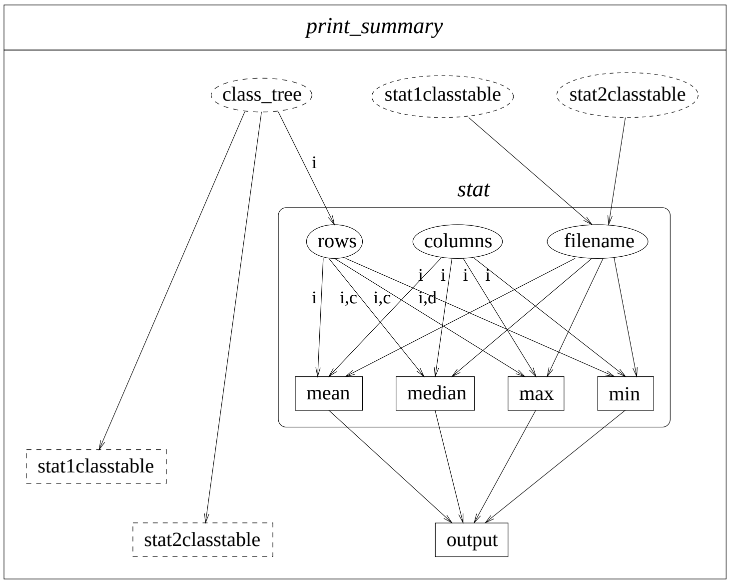


Figure 10. An IODG representation of function `print_summary`

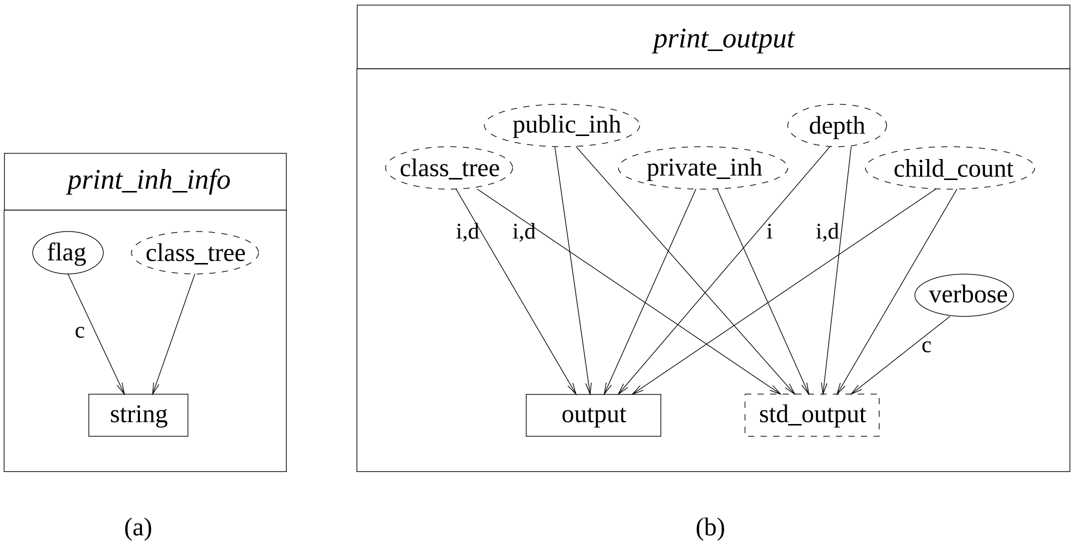


Figure 11. (a) shows IODG `print_inh_info` *and (b) shows IODG* `print_output`*; they are IODG's decomposed from IODG* `print_info`

$$R4 : \texttt{print\_summary} \xrightarrow{D3} \{\texttt{print\_stat1}, \texttt{print\_stat2}, \texttt{print\_out\_summary}\}$$

We decompose IODG `add_class` into two IODG's. One, also named `add_class`, includes two output components, `class_tree` and `pre_class`. The other IODG, `temp`, includes only one output component, `class_from_parent`. In the new `add_class` function, `class_from_parent` is assigned a constant value. We will later compose `temp` with its caller(s). The new `add_class` IODG exhibits sequential cohesion and IODG `temp` exhibits functional cohesion. Following a similar procedure, IODG `tg_add_class` is decomposed into IODG `tg_add_class` and IODG `tg_temp`.

IODG `print_info` has two tightly connected output components, `output` and `std_output`. Another output, `string`, is loosely connected with the other outputs. We decompose IODG `print_info` into IODG `print_inh_info` and IODG `print_output`, as shown in Figure 11. IODG `print_inh_info` exhibits functional cohesion and IODG `print_output` exhibits communicational cohesion.

IODG `print_summary` contains three output components that are loosely connected. Two outputs, `stat1classtable` and `stat2classtable`, could be placed into the same IODG or separated into individual IODG's. This decision depends on the detailed and conceptual relationship between two outputs and the software application. We decompose IODG `print_summary` into three new IODG's, `print_stat1`, `print_stat2`, and `print_out_summary`. Figure 12 displays these IODG's. The IODG of `print_out_summary` omits an invocation of `stat`. All of the new IODG's in the Figure 12 exhibit functional cohesion.

In applying the decompositions, we change the invocations in caller IODG's to reflect the callee's decomposition. For example, function `add_class` is called by function `add_parent`, and thus, an invocation of `add_class` in `add_parent` is replaced by two

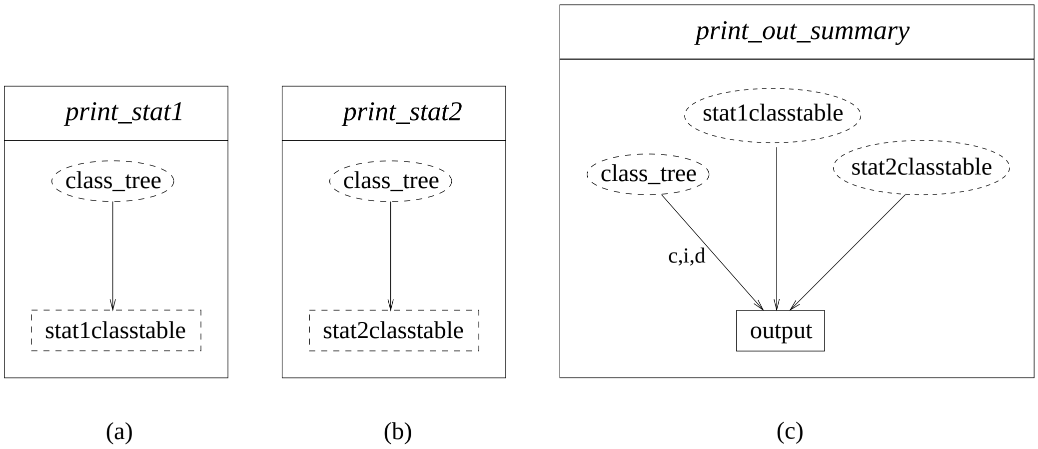


Figure 12. (a) shows IODG `print_stat1`*, (b) IODG* `print_stat2`*, and (c) IODG* `print_out_summary`*; they are IODG's decomposed from IODG* `print_summary`

invocations of `add_class` and `temp`.

5. Locate unnecessarily decomposed modules and compose them: We apply composition operations:

$$R5 : \{\texttt{add\_parent}, \texttt{temp}\} \xrightarrow{C1} \texttt{add\_parent}$$
$$R6 : \{\texttt{main}, \texttt{tg\_temp}\} \xrightarrow{C2} \texttt{main}$$

IODG `temp` is called from three functions (call-couplings), but used only in IODG `add_parent`. Note that `temp` includes a global variable assigned with a constant value. Thus, we combine `temp` into IODG `add_parent` and remove the invocations in the other functions. IODG `tg_temp` is called and used only from IODG `main`, and thus, we also combine `tg_temp` into IODG main.

6. Generate program code from the restructured IODG's: Program code is generated from each corresponding IODG. The Appendix includes program code corresponding to IODG `print_inh_info` and IODG `print_output`.

Through restructuring, two functions exhibiting coincidental cohesion have been converted into functions with sequential cohesion, one function exhibiting communicational cohesion was decomposed into a function with communicational cohesion and a function with functional cohesion, and another function with communicational cohesion has been decomposed into three functions with functional cohesion.

7.3. Restructuring Results

At the end of the process, we had restructured six functions, creating nine new functions. The restructuring process included five decompositions and two compositions. All restructuring decision were based on the IODG's and DLC/DCP values.

Table 1. Properties of the Restructured Functions in the JASMIN System, Before Restructuring.

Original Functions	No. of inputs	No. of outputs	No. of lines	No. of nodes	No. of edges	V(g)	DLC
1. tg_add_class	5	3	44	12	19	9	coincidental
2. add_class	5	3	40	12	17	7	coincidental
3. print_info	8	3	88	16	25	11	communicational
4. print_summary	3	3	111	22	30	10	communicational
5. add_parent	5	4	37	9	13	6	communicational
6. main	3	4	285	53	72	21	communicational
Totals	29	20	605	124	176	$*$ 64	$**$
Mean	4.83	3.33	100.83	20.67	29.33	10.67	$**$

$*$ This is the V(g) count for the aggregate of the set of functions rather than the total V(g).
$**$ The sum and average of DLC levels are not defined.

Table 1 shows properties of the six functions before restructuring, and Table 2 shows these same properties of the nine resulting functions. The process converted two functions with coincidental cohesion and four functions with communicational cohesion into three functions with functional cohesion, two functions with sequential cohesion, and three with communicational cohesion. The set of restructured functions have 20 more lines of code than the original functions, which represent an increase of about 5%. However, the average size of the restructured functions is 30% smaller than the original functions.

Just decreasing the size of functions does not demonstrate improvement, and we should expect that the restructured functions would show an improvement in cohesion levels, since cohesion is a key factor driving restructuring. We do see that the cyclomatic complexity, $V(g)$, in the restructured functions is much lower than the original ones: the $V(g)$ of restructured functions average 7.67, while the original functions have an average $V(g)$ of 10.67. Keeping $V(g)$ below 10 is common advice, first proposed by McCabe (McCabe, 1976), and all but one of the restructured functions have $V(G) < 10$, while three of the original functions have $V(G) \geq 10$.

Table 3 summarizes the properties of both the entire system and the modified functions, before and after restructuring. The restructured JASMIN has 42 functions: 26 functions have functional cohesion, 3 functions have sequential cohesion, 12 functions have communicational cohesion, and one functions has coincidental cohesion. The one coincidental function actually exhibits *temporal* SMC cohesion. The restructured JASMIN has three more call-couplings (function calls) than the original JASMIN. Clearly, restructuring improved module cohesion, with a small increase in overall system coupling. We also see that the average number of lines of code in a function decreased by 7%, even though the total number of lines increased by 30 lines or less than 1%.

Four poorly designed functions were greatly improved at a cost of a small increase in call-coupling. DCP call-coupling corresponds to the 'data coupling' of Myers' coupling hierarchy, which is the best type of coupling when communication between modules is necessary (Myers, 1978, pp. 41–56).

Table 2. Properties of the Restructured Functions in the JASMIN System, After Restructuring.

Original Functions	No. of inputs	No. of outputs	No. of lines	No. of nodes	No. of edges	V(g)	DLC
1. tg_add_class	5	2	41	11	16	6	sequential
2. add_class	5	2	38	11	15	6	sequential
3a. print_inh_info	2	1	25	7	10	5	functional
3b. print_output	7	2	69	12	19	9	communicational
4a. print_stat1	1	1	19	4	6	4	functional
4b. print_stat2	1	1	23	5	8	5	functional
4c. print_out_summary	3	1	89	17	22	7	functional
5. add_parent	5	4	41	11	15	6	communicational
6. main	3	4	290	55	74	21	communicational
Totals	32	18	635	133	186	* 69	**
Mean	3.56	2.00	70.56	14.78	20.67	7.67	**

* This is the V(g) count for the aggregate of the set of functions rather than the total V(g).
** The sum and average of DLC levels are not defined.

Table 3. Properties Before and After Restructuring for the Entire JASMIN System and Changed Functions.

	Entire System		Changed Functions	
Attributes	Before	After	Before	After
No. Functions	39	42	6	9
No. Lines	4540	4570	605	635
Lines per Function	116.41	108.81	100.83	70.56
No. Functions at each DLC level:				
Functional	22	26	0	4
Sequential	3	5	0	2
Communicational	11	10	4	3
Iterative	0	0	0	0
Conditional	0	0	0	0
Coincidental	3	1	2	0

The restructuring framework includes only eight basic operations, which are described in Section and in Figure 5. It might seem that this limited number of transformation would apply to too few modules to be useful. Yet, in the case study, we could restructure six out of 39 modules, which represent 15% of the system's components. An improvement of 15% of the modules is significant—the restructured system shows measurable improvements over the original system.

7.4. Further Information Needs

Overall, our restructuring process was very effective. However, we found that, to make restructuring decisions, we sometimes needed more detailed information about a program:

1. Simple dependence information between input-output components cannot depict some relationships. For example, if two outputs of a function are computed by sequentially reading different parts of a common file or data structure, it is difficult to decompose the program slices corresponding to the two outputs into two separated functions. The IODG and DLC measure cannot be used to recognize such an input-output relationship, because the IODG representation does not distinguish between a file or a data structure and a simple scalar. We could extend the IODG representation by decorating its nodes and edges with additional information. For example, each node can be labeled with a node type such as a file, a data structure, or a simple variable, and each edge is with connection information such as a single, multiple, or specific number of connections.

2. Some functions with low DLC levels have outputs sharing many inputs. These functions have tightly-connected input-output components although they have low DLC levels. The DLC measure is useful for software restructuring because it always finds the weakest connection among input-output components. However, it does not indicate how tightly bound other components are. Consider two functions A and B; A has two outputs that have data dependence on one common input and B has two outputs that shares five inputs. The DLC level for both functions is communicational cohesion while the outputs of B is more tightly connected than those of A.

   Another class of cohesion measures, Design Functional Cohesion (DFC) measures (Bieman and Kang, 1998), detects the tightness difference between input-output components. These measures can complement the DLC measure. We could first find a set of procedures with weak cohesion levels using the DLC measure, and then identify a subset with loosely connected input-output components using the DFC measure.

   Even though two outputs of a program access multiple inputs and share only one input, the two outputs might share large segments of program code. The IODG representation and DLC/DFC measures do not indicate how much program code is shared by output components. A function with outputs sharing significant amount of code should not be decomposed. If we decompose outputs sharing program code, the shared code part can be copied, and the execution of the different programs result in the repeated execution of identical code. In order to represent the amount of code sharing, we can extend the IODG representation by decorating each node with the amount of the shared code. At design stage, to include the size information, a designer would need to estimate the amount of shared code. If program code is available, the information can be obtained from dependence analysis. We can also apply Functional Cohesion (FC) measures (Bieman and Ott, 1994) on program code to have the code-level information. The FC measures indicate how much code is shared by output components. DFC and FC measures can be used with the DLC measure to provide more specific information.

3. The restructuring process allows us to do effective decomposition and composition of design components and program code. However, the process does not support low-level restructuring. We find frequent use of global components for communication between program components in JASMIN. Some of those global components can be implemented by parameter-passing mechanisms which minimizes the overall system coupling. Our restructuring process can reveal the existence of global components, but cannot effectively restructure this code.

Until now, we have only addressed the restructuring of procedural programs such as those written in the C language. The principal restructuring processes are (1) extracting and visualizing design information from program code, (2) applying design-level measures on the design, (3) restructuring the design based on the visualized design information and design-level measures, and (4) generating restructured program code from the restructured design. We have not yet addressed restructuring aimed towards converting procedural software into modular-style programs—programs with sets of procedures that access regional variables. Such conversions may require additional semantic information beyond that available from our graph-based design representations. The basic approach in our restructuring framework can be applied to object-oriented software. We plan to extend our work to the restructuring of object-oriented software. We have already defined a model to represent object-oriented programs and two class cohesion measures (Bieman and Kang, 1995).

8. RELATED WORK

8.1. Cohesion and Coupling Measurement

Closely related work includes the development of cohesion and coupling measures, and restructuring methods based on design or code-level analyses.

Emerson's Cohesion Measurement. Emerson developed a cohesion measure in terms of a control flow graph model (Emerson, 1984). Flow subgraphs are generated for each variable referenced in a module. The precise cohesion value depends upon the ratio of the size and cyclomatic complexity of the subgraphs to the size of the complete flowgraph. Unfortunately, multiplying the reference set size by the cyclomatic complexity masks the view of cohesion. Cyclomatic complexity is a control flow measure, and combining the measures of different attributes weakens the discriminating power of a measure (Melton *et al.*, 1990). Emerson's cohesion measure can only be applied to implementations.

Rules for SMC 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 SMC Cohesion (Lakhotia, 1993). 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. Because the rules are formal, a tool can automatically perform the classification. The technique, as defined by Lakhotia (1993), can be applied only after the coding stage since it is defined in terms of implementation details. However, if designers specify variable dependence graphs for inputs and outputs, this method could be applied to designs in a manner similar to our DLC measure.

Functional Cohesion Measures. Bieman and Ott develop cohesion measures that indicate the extent to which a module approaches the ideal of functional cohesion (Bieman and Ott, 1994). 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. These measures also depends on the implementation details of module and can be applied to program code. This development of measures of functional cohesion was based on earlier work by Ott and Thuss (1989), which introduced the notion of defining cohesion in terms of program slices.

Coupling Measure from Empirical Relations. Fenton and Melton also use Myers' Coupling (Myers, 1978, pp. 41–56) as an empirical relation system to determine coupling of a pair of modules and global coupling (Fenton and Melton, 1990). The relations are ranked with a type number; type 5 (content coupling) is least desirable and type 0 (no coupling) is most desirable. The authors define a coupling measure between module x and y on an ordinal scale by considering the relation type between x and y and the number of connections of the relation type between x and y:

$$M(x, y) = i + \frac{n}{n+1} \tag{2}$$

where i is the greatest coupling type number between x and y, and n is the number of interconnections of the relation type between x and y. Global coupling $C(S)$ of a system is defined as the overall level of connectivity in the system: $C(S)$ is the median value of the pairwise coupling values for all modules in a system. The coupling measures are defined rigorously based on the measurement theory (Fenton and Pfleeger, 1997). However, the empirical relations are not defined quantitatively and cannot determined objectively.

Cohesion and Coupling Measures for Object-Oriented Software Our focus in this paper is on procedural software. However, there is active work on developing cohesion and coupling measures for object-oriented and object-based software. Bieman and Kang define 'class cohesion' measures based on dependencies between methods through their references to instance variables (Bieman and Kang, 1995). Chidamber and Kemerer define another class cohesion measure, lack of cohesion between methods (LCOM), which is also based on method interconnections through instance variable references (Chidamber and Kemerer, 1994). Hitz and Montazeri (1995) define coupling measures based on the distinction between "dynamic dependency" and "static dependency". Briand *et al.* introduce cohesion and coupling measures for object-based systems based on interactions between declarations (Briand, Morasca and Basili, 1993). In related work, Briand *et al.* develop a framework for developing and analyzing both cohesion and coupling in object-oriented systems (Briand, Daly and Wüst, 1997 and 1978; Briand, Devanbu and Melo, 1997).

8.2. Software Restructuring

Slicing and Restructuring. The restructuring method of Kim, Kwon, and Chung (1994) makes use of a notion of module strength (cohesion). They define *processing blocks* which are similar to the 'data slices' of Bieman and Ott (1994). 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. They treat each of these logically associated functions as a processing block. Unfortunately, an analysis of program code cannot always automatically detect these logically associated functions. 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. Information on module strength alone is not sufficient to determine how to group components into packages. An understanding of module functions and underlying design decisions is needed. Like our approach, module strength is used as a criterion for software restructuring. However, unlike our approach, module strength is based only on the code implementation. The attributes that are actually quantified by the measure are not specified. The module strength measure computes the average of the relatedness between processing blocks rather than finding the most weakly connected blocks.

A Restructuring Assistant. Griswold and Notkin (1993 and 1995) provide support for analysts in making a specific change. They do not help guide the analyst in making his/her initial decision. A tool performs nonlocal structural changes in behalf of an analyst:

1. When the analyst moves an expression of a program, the tool checks to ensure that the change preserves meaning.

2. When the analyst renames a variable, the tool renames all its uses and makes sure that the new name does not conflict with any existing names.

3. When the analyst wants to replace the uses of a variable definition with the defining expression, he/she selects the assignment to be inlined and the tool handles finding and inlining the uses.

4. When the analyst turns a sequence of expressions into a function, the tool replaces the abstracted statements with a call on the function.

The software analyst manipulates the program code itself rather than an abstracted representation of the program. After the analyst makes a local change in the program to improve structure, the tool automatically selects the global changes. Restructuring depends on the analyst's initial intuitive decision about what kinds of change in the program need to be made. Thus, it is difficult to show that the restructured program has been improved over the original program.

Finding Hierarchical Relationships. Choi and Scacchi (1990) propose a restructuring process to change the relationship between modules from a resource-exchange relationship to a hierarchical relationship. Here, source files are treated as modules. The resource-exchange relationships are represented by a resource-flow diagram (RFD) where resources exchanged between modules include data types, procedures, and variables. A hierarchical relationship shows the control connections between modules and a control module, and is represented by a resource-structure diagram (RSD).

The main goal is to map from an RFD to an RSD. The restructuring algorithm minimizes "module coupling" and "alteration distance", which are defined informally. Module coupling is the number of modules in the system or subsystem. Alteration distance between modules is the length of the path between the altered and affected module. The alteration distance of a system or subsystem is the sum of the alteration distances of all the modules of a system or subsystem.

The measures used as criteria to map from an RFD to an RSD are not defined formally. Thus, it is not clear how the values of the measures are improved through the restructuring algorithm. There is no support to help determine which program code should be changed or which additional code is needed during restructuring. The hierarchical relationship among modules is built by adding a subsystem node to a set of modules or other subsystems. However, the contents of the subsystem node are not specified. This restructuring method provides a hierarchical design corresponding to a module relationship (i.e., RFD). It does not help perform the actual restructuring of the original software.

A Documentation-based Approach. Tesch and Klein (1991) propose a design optimization method based on cohesion and coupling measurements of existing documentation. The goal is to develop a complete hierarchy chart of organized processes in a top-down, optimal fashion. A composite measure is formulated from the cohesion and coupling measures. The composite measure is used to decompose the processes in a system into successive groupings of modules on a hierarchy chart by grouping modules that are closely related and separating modules that perform different functions.

They determine SMC cohesion by computing three relational matrices: *precedence matrix*, *matrix of timing relationship*, and *incidence matrix*. For each cohesion level, a value between 0 and 1 is assigned. Thus, the resulting cohesion measure is on ordinal scale.

Coupling is defined as the percentage of data items used within groupings of modules such that:

$$C_{i,j} = \frac{\text{number of shared data items for } i \text{ and } j}{\text{number of data items for all processes}} \tag{3}$$

As $C_{i,j}$ increases, the objective is to group process i with process j to lower the coupling.

These cohesion and coupling measures are used to derive a single composite measure for evaluating a design such that:

$$P_{i,j} = w_1 W_{i,j} + w_2(1 - C_{i,j}) \tag{4}$$

where the $W_{i,j}$ is the cohesion level of a module including process i and j, $C_{i,j}$ is the coupling value between two processes i and j, w_i's represent importance weights on the criteria. Then, the design optimization process use the composite measure to decompose the process graph into a hierarchy chart by trying to minimize the measure.

The module clustering method is developed to help construct an optimal design for a system from existing documentation. A composite measure accomplishes this goal, thus, the validation of the clustering process depends on the measure. Unfortunately, the composite measure combines the ordinal-scale cohesion measure and the ratio-scale coupling measure using addition operation. Applying the addition operation to an ordinal-scale measure is not a meaningful transformation.

Converting Procedural Programs into Object-Module Programs. Zimmer (1990) demonstrates the conversion of a procedural program into an *object-module* program. Both the original and restructured program are implemented in FORTRAN. The resulting *object-module* program makes use of regional variables that simulate the module-scoped variables used in an object-based language, or the attributes, fields, or instance variables used in an object-oriented language. Restructuring requires semantic knowledge of the program including a fairly formal specification including state invariants, an understanding the "coherent purpose" of code segments, and an understanding of code details. For example, data flow analysis determines dependencies between variables, called *cobwebs*.

The restructuring of an example 100 line FORTRAN program is effective—data dependencies are reduced. However, the restructuring required an intuitive understanding of many code details and a significant amount of human labor. The restructured program barely resembles the original; it has a very different design structure. For such a method to scale up, much of the process needs to be automated. On a large system, the code, design documents, integration and regression test plans and data, etc. would have to be adapted to match the new implementation. Quantitative criteria might potentially identify potential object-modules, and allow this kind of restructuring to fit into a quantitative restructuring framework. However, quantifying such semantic information is an open problem.

9. CONCLUSIONS

Software restructuring should be easy, efficient, and semantic preserving. We provide a software restructuring framework where design structures can be extracted from code, visualized, quantified, and restructured. Software analysts can directly apply the presented models, measures, restructuring operations, and restructuring process to re-engineer legacy systems that were developed using the procedural paradigm. The measures can quickly identify components that are amenable to restructuring. Visualized design-level models provide further intuition to help select restructuring candidates, without including all of the details in the code. The restructuring process and operations gives a step-by-step procedure for optimizing improvements in cohesion and coupling. Again, the visualized models help guide the process.

In the case study, restructuring increased cohesion and decreased module size and cyclomatic complexity. Such improvements suggest that future maintenance and testing will be easier. Also, the process of developing design-level models can improve an analyst's understanding of system structure.

These results show that restructuring can demonstrate improvements, while preserving the original development paradigm. A system developed using the procedural paradigm, remains a procedural system, with measurable improvements in its structure. Restructuring, without changing development paradigms, will be simpler than the restructuring required to move from the procedural to an object-based or object-oriented development paradigm. Restructuring will involve fewer and simpler changes, and the architecture of the resulting system will still resemble the original system. Thus, the original development life-cycle products, such as design documents and regression test data, will require less revision. An effective restructuring process that is simple should be practical. A simple process, with automated support, is especially important when restructuring large industrial systems.

The quantitative restructuring framework uses two design-level models, the IODG and MIG. The IODG is an abstraction of a module interface; it models dependency relationships between module inputs and outputs. The MIG models the connections between modules. Design-level cohesion (DLC) and design-level coupling (DCP) measures can be computed in terms of these models. These measures guide the restructuring process. Our restructuring approach provides several benefits:

1. Since the restructuring operations are applied to the IODG's rather than program code itself, the whole restructuring process can be simplified and visualized.

2. Since users manipulate the IODG's rather than code, unnecessary implementation details can be hidden from users.

3. The IODG and MIG information can be generated from system code. Thus, analysts can recapture designs from existing, possibly legacy, systems.

4. Since restructuring operations are applied to design-level rather than more detailed code-level information, performance of the restructuring process can be improved greatly.

5. DLC and DCP measures provide objective criteria for restructuring while graphical displays of IODG's and MIG's provide visual guidance.

6. Although the restructuring framework is based on the DLC and DCP measures, additional design-level measures based on the IODG and MIG models can be defined and applied within the framework.

7. The IODG and MIG representations, the DLC and DCP measures, and the restructuring process can be applied during software design as well as maintenance.

8. The restructuring process consists of a series of semantic preserving decompositions and compositions of 'processing elements'. Thus, the semantics of a system are preserved during the process.

9. The restructuring process can be easily automated. IODG's and MIG's can be readily generated using a compiler-like code analysis tool. The DLC and DCP measures can be easily computed once IODG's and MIG's are generated either from a design or implementation. A tool to perform the decomposition and composition operations can be built using a concept of program slicing (Weiser, 1994). Several slicing tools are available (Jackson and Ladd, 1994; Lyle *et al.*, 1995).

We have developed a tool to generate IODG information from program code, tools to compute DFC and DLC measures from the generated IODG information, and a program slicing tool to support the restructuring process. These tools have been integrated as part of FUNCO, which is available for general use on the World Wide Web(Kang and Bieman, 1997). Planned developments include a tool to graphically display IODG information and an integrated user-interface to allow easy user access to the tools for software restructuring. We are investigating other design measures which can be used as restructuring criteria. For example, much more work needs to be done to develop quantitative means to identify objects in large procedural systems and restructure the code in an appropriate manner. We also plan to evaluate the effects of restructuring on external quality attributes such as testability, reusability, reliability, and maintainability.

Acknowledgements

This research was partially supported by NASA Langley Research Center grant NAG1-1461. We thank the Journal's anonymous reviewers for their comments, which greatly improved both the content and presentation.

APPENDIX. PROGRAM CODE RESTRUCTURING EXAMPLE

In this appendix, we provide an original source program, `print_info`, of the JASMIN system and its restructured program code `print_inh_info` and `print_output`.

A.1. Original Code: `print_info`

```
/**********************************************************************************
 print_info writes given class-inheritance information in three files, a file named
 by an out-parameter 'string', a file named by another out-parameter 'output', and
 a standard output file.
**********************************************************************************/
void print_info(char *string, char *flag, int print_it, char *output, int verbose) {

save_class *cl;
children   *buffe;
parent     *temp;
float      childof_private_pct = 0.0,
           childof_non_private_pct = 0.0,
           private_pct = 0,
```

```
           non_private_pct = 0;
FILE       *fp,
           *fd;
int        CTG = 0,
           parent_ct = 0,
           private_ct = 0,
           non_private_ct = 0,
           num_class = 0,
           num_undefined = 0,
           childof_private = 0,
           childof_non_private = 0,
           multiple_inheritance = 0,
           i;

  fp = fopen(string, flag);
  for(cl = class_tree; cl; cl = cl->next) {
     if (cl != class_tree)
        num_class++;
     fprintf(fp,"class: %s\tprivate: %d\tdepth: %d\tchild: %d\tparents: %d\n",
       cl->name, cl->private, cl->inh_depth, cl->child_count, cl->parent_count);
     fprintf(fp,"defined: %d\n" , cl->defined);
     temp = cl->parent;
     while (temp) {
        fprintf(fp, "parent: %s \t %d\n", temp->name, temp->scope);
        temp = temp->next;
     }
     fprintf(fp, "parent: 0 \t %d\n", 0);
     buffe = cl->child;
     while (buffe) {
        fprintf(fp, "child: %s\n", buffe->name);
        buffe = buffe->next;
     }
     fprintf(fp, "child: 0\n");
     if ((cl != class_tree) && cl->not_printed == 1) {
        if(cl->private==1) {
           childof_private += cl->child_count;
           private_ct++;
        }
        else {
           childof_non_private += cl->child_count;
           non_private_ct++;
        }
        cl->not_printed = 0;
        if (cl->parent_count >= 2)
           multiple_inheritance++;
        if (cl->defined == 0)
           num_undefined++;
     }
  }
  fclose(fp);

  if (print_it) {
     fd = fopen(output, "a");
     fprintf(fd, "\n\n");
```

```
    private_pct = 100 *((float) private_ct) / ((float) num_class);
    non_private_pct = 100 *((float) non_private_ct) / ((float) num_class);
    childof_private_pct = 100 * ((float) childof_private) /
       ((float)(childof_private + childof_non_private));
    childof_non_private_pct = 100 * ((float) childof_non_private) /
       ((float)(childof_private + childof_non_private));

    fprintf(fd, "   Total     |         %5d\n", num_class);
    fprintf(fd, "  Undefined  |         %5d\n", num_undefined);
    fprintf(fd, "\n\n");
    fprintf(fd, "   Private   |         %5d\n", private_inh);
    fprintf(fd, " Non_private |         %5d\n", public_inh);
    fprintf(fd, "  Multiple   |         %5d\n", multiple_inheritance);
    fprintf(fd, "\n\n");
    fprintf(fd, "  Private    ||  %5d      %4.2f%%   |  %5d        %4.2f%%\n",
    private_ct, private_pct, childof_private, childof_private_pct);
    fprintf(fd, " Non-Private ||  %5d      %4.2f%%   |  %5d        %4.2f%%\n",
       non_private_ct, non_private_pct, childof_non_private,
       childof_non_private_pct);
    fprintf(fd, "\n\n");
    fprintf(fd,   "Nodes ");
    for (i = 0; i <= depth; i++)
      fprintf(fd, "%5d", child_count[i]);
    fprintf(fd, "\n\n\n");
    fclose(fd);

    if (verbose) {
       printf("\n\n");
       printf("Total number of classes:         %5d\n", num_class);
       printf("Classes with private section:    %5d \t %3.2f%%\n",
          private_ct, private_pct);
       printf("Classes without private section: %5d \t %3.2f%%\n",
          non_private_ct, non_private_pct);
       printf("Num of classes derived from classes with private sections:\n\
          %5d \t %3.2f%%\n", childof_private, childof_private_pct);
       printf("Num of classes derived from classes without private sections:\n\
          %5d \t %3.2f%% \n",childof_non_private,childof_non_private_pct);
       printf("Private inheritance:     %5d \n", private_inh);
       printf("Non-private inheritance: %5d \n", public_inh);
       printf("Multiple inheritances:   %5d \n", multiple_inheritance);
       printf("Undefined classes:       %5d \n", num_undefined);
       printf("Tree depth:              %5d \n", depth + 1);
       for (i = 0; i <= depth; i++)
          printf("Num of nodes in tree level %d: %d\n", i, child_count[i]);
    }
  }
} /* print_info */
```

A.2. Restructured Code: print_inh_info and print_output

```
/*****************************************************************************
 print_inh_info writes given class-inheritance information in a file named by
 an out-parameter 'string'. Another input 'flag' indicates write or append
 operation of the file.
```

```
******************************************************************************/
void print_inh_info(char *string, char *flag) {

save_class *cl;
children   *buffe;
parent     *temp;
FILE       *fp;

  fp = fopen(string, flag);
  for(cl = class_tree; cl; cl = cl->next) {
     fprintf(fp,"class: %s\tprivate: %d\tdepth: %d\tchild: %d\tparents: %d\n",
       cl->name, cl->private, cl->inh_depth, cl->child_count, cl->parent_count);
     fprintf(fp,"defined: %d\n", cl->defined);
     temp = cl->parent;
     while (temp) {
        fprintf(fp, "parent: %s \t %d\n", temp->name, temp->scope);
        temp = temp->next;
     }
     fprintf(fp, "parent: 0 \t %d\n", 0);
     buffe = cl->child;
     while (buffe) {
        fprintf(fp, "child: %s\n", buffe->name);
        buffe = buffe->next;
     }
     fprintf(fp, "child: 0\n");
   }
   fclose(fp);
} /* print_inh_info */



/******************************************************************************
 print_output writes given class-inheritance information in a file named by an
 out-parameter 'output' and standard output file based on a flag 'verbose'.
******************************************************************************/
void print_output(char *output, int verbose) {

save_class *cl;
float      childof_private_pct = 0.0,
           childof_non_private_pct = 0.0,
           private_pct = 0,
           non_private_pct = 0;
FILE       *fd;
int        private_ct = 0,
           non_private_ct = 0,
           num_class = 0,
           num_undefined = 0,
           childof_private = 0,
           childof_non_private = 0,
           multiple_inheritance = 0,
           i;

  for(cl = class_tree; cl; cl = cl->next) {
     if (cl != class_tree)
```

```
      num_class++;
   if ((cl != class_tree) && cl->not_printed == 1) {
      if(cl->private==1) {
         childof_private += cl->child_count;
         private_ct++;
      }
      else {
         childof_non_private += cl->child_count;
         non_private_ct++;
      }
      cl->not_printed = 0;
      if (cl->parent_count >= 2)
         multiple_inheritance++;
      if (cl->defined == 0)
         num_undefined++;
   }
}

fd = fopen(output, "a");
fprintf(fd, "\n\n");

private_pct = 100 *((float) private_ct) / ((float) num_class);
non_private_pct = 100 *((float) non_private_ct) / ((float) num_class);
childof_private_pct = 100 * ((float) childof_private) /
   ((float)(childof_private + childof_non_private));
childof_non_private_pct = 100 * ((float) childof_non_private) /
   ((float)(childof_private + childof_non_private));

fprintf(fd, "   Total     |         %5d\n", num_class);
fprintf(fd, "  Undefined  |         %5d\n", num_undefined);
fprintf(fd, "\n\n");
fprintf(fd, "   Private   |         %5d\n", private_inh);
fprintf(fd, " Non_private |         %5d\n", public_inh);
fprintf(fd, "  Multiple   |         %5d\n", multiple_inheritance);
fprintf(fd, "\n\n");
fprintf(fd, "  Private    ||  %5d         %4.2f%%    |  %5d        %4.2f%%\n",
   private_ct, private_pct, childof_private, childof_private_pct);
fprintf(fd, " Non-Private ||  %5d         %4.2f%%    |  %5d        %4.2f%%\n",
   non_private_ct, non_private_pct, childof_non_private,
   childof_non_private_pct);
fprintf(fd, "\n\n");
fprintf(fd,   "Nodes ");
for (i = 0; i <= depth; i++)
   fprintf(fd, "%5d", child_count[i]);
fprintf(fd, "\n\n\n");
fclose(fd);

if (verbose) {
   printf("\n\n");
   printf("Total number of classes:         %5d\n", num_class);
   printf("Classes with private section:    %5d \t %3.2f%%\n",
      private_ct, private_pct);
   printf("Classes without private section: %5d \t %3.2f%%\n",
      non_private_ct, non_private_pct);
```

```
    printf("Num of classes derived from classes with private sections:\n\
       %5d \t %3.2f%%\n", childof_private, childof_private_pct);
    printf("Num of classes derived from classes without private sections:\n\
       %5d \t %3.2f%% \n",childof_non_private,childof_non_private_pct);
    printf("Private inheritance:     %5d \n", private_inh);
    printf("Non-private inheritance: %5d \n", public_inh);
    printf("Multiple inheritances:   %5d \n", multiple_inheritance);
    printf("Undefined classes:       %5d \n", num_undefined);
    printf("Tree depth:              %5d \n", depth + 1);
    for (i = 0; i <= depth; i++)
       printf("Num of nodes in tree level %d: %d\n", i, child_count[i]);
  }

} /* print_output */
```

References

Arnold, R. (1989) 'Software restructuring', *Proceedings of the IEEE*, **77**(4), 607–617.

Baker,A.; Bieman, J.; Fenton, N.; Gustafson, D.; Melton, A. and Whitty, R. (1990) 'A philosophy for software measurement', *The Journal of Systems and Software*, **12**(3), 277–281.

Bieman, J. and Kang, B.-K. (1995) 'Cohesion and reuse in an object-oriented system', *SIGSOFT Software Engineering Notes*, **20**(5), 259–262.

Bieman, J. and Kang, B.-K. (1998) 'Measuring design-level cohesion,' *IEEE Transactions on Software Engineering*, **24**(2), 111–124.

Bieman, J. and Ott, L. (1994) 'Measuring functional cohesion', *IEEE Transactions on Software Engineering*, **20**(8), 644–657.

Bieman, J. and Zhao, J. X. (1995) 'Reuse through inheritance: A quantitative study of C++ software', *SIGSOFT Software Engineering Notes*, **20**(5), 47–52.

Briand, L.; Daly, J. and Wüst, J. (1997) 'A unified framework for cohesion measurement in object-oriented systems', in *Proceedings of the Fourth International Software Metrics Symposium*, IEEE Computer Society Press, Los Alamitos CA, pp. 43–53.

Briand, L.; Daly, J. and Wüst, J. (1998), 'A unified framework for cohesion measurement in object-oriented systems', *Empirical Software Engineering*, **3**(1), 65–117.

Briand, L.; Daly, J. and Wüst, J. (1999) 'A unified framework for coupling measurement in object-oriented systems', *IEEE Transactions on Software Engineering*, **25**(1), 91–121.

Briand, L.; Devanbu, P. and Melo, W. (1997) 'An investigation into coupling measures for C++', in *Proceedings 19th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 412–421.

Briand, L.; Morasca, S. and Basili, V. (1993) 'Measuring and assessing maintainability at the end of high level design', in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 88–95.

Calliss, F and Cornelius, B. (1989) 'Two module factoring techniques', *Journal of Software Maintenance: Research and Practice*, **1**(2), 81–89.

Chidamber, S. and Kemerer, C. (1994) 'A metrics suite for object-oriented design', *IEEE Transactions on Software Engineering*, **20**(6), 476–493.

Choi, S. and Scacchi, W. (1990) 'Extracting and restructuring the design of large systems', *IEEE Software*, **7**(1), 66–71.

Emerson, T. (1984) 'A discriminant metric for module cohesion', in *Proceedings of the 7th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 294–303.

Esteva, J. and Reynolds, R. (1991) 'Identifying reusable software components by induction', *International Journal of Software Engineering and Knowledge Engineering*, **1**(3), 271–292.

Fenton, N. (1994) 'Software measurement: a necessary scientific basis', *IEEE Transactions on Software Engineering*, **20**(3), 199–206.

Fenton, N. and Melton, A. (1990) 'Deriving structurally based software measures', *The Journal of Systems Software*, **12**(3), 177–187.

Fenton, N. and Pfleeger, S.L. (1997) *Software Metrics—A Rigorous and Practical Approach*, International Thompson Computer Press, London, 638 pp.

Ferneley, E. (1999), 'Design metrics as an aid to software maintenance: an empirical study', *Journal of Software Maintenance: Research and Practice*, **11**(1), 55–72.

Gallagher, K. and Lyle, L. (1991) 'Using program slicing in software maintenance', *IEEE Transactions on Software Engineering*, **17**(8), 751–761.

Griswold, W. and Notkin, D. (1993) 'Automated assistance for program restructuring', *ACM Transactions on Software Engineering and Methodology*, **2**(3), 228–269.

Griswold, W. and Notkin, D. (1995) 'Architectural tradeoffs for a meaning-preserving program restructuring tool', *IEEE Transactions on Software Engineering*, **21**(4), 275–287.

Gustafson, D.; Tan, J. and Weaver, P. (1993) 'Software measure specification', in *Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM Press, New York NY, pp. 163–168.

Hitz, M. and Montazeri, B. (1995) 'Measuring coupling and cohesion in object-oriented systems', in *Proceedings of the International Symposium on Applied Corporate Computing*, XXXXX, pp. 75–76, 197, and 78–84.

Jackson, D. and Ladd, D. (1994) 'Semantic Diff: A tool for summarizing the effects of modifications', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 243–252.

Kang, B.-K. and Bieman, J. (1996) 'Using design cohesion to visualize, quantify, and restructure software', in *The 8th International Conference on Software Engineering and Knowledge Engineering*, Knowledge Systems Institute, Skokie IL, pp. 222–229.

Kang, B.-K. and Bieman, J. (1997) *Funco—A Functional Cohesion Software Measurement Tool for C Programs*, Computer Science Department, Colorado State University, Fort Collins CO, URL http://www.cs.colostate.edu/~bieman/funco.html

Kang, B.-K. and Bieman, J. (1998) 'Using design abstractions to visualize, quantify, and restructure software', *The Journal of Systems and Software*, **42**(2), 175–187.

Kim, H.-S.; Kwon, Y.-R. and Chung I.-S. (1994) 'Restructuring programs through program slicing', *International Journal of Software Engineering and Knowledge Engineering*, **4**(3), 349–368.

Lakhotia, A. (1993) 'Rule-based approach to computing module cohesion', in *Proceedings 15th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 35–44.

Lano, K. and Haughton, J. (1992) 'Extracting design and functionality from code', in *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 74–82.

Lehman, M. (1980) 'Programs, lifecycles, and laws of software evolution', *Proceedings of the IEEE*, **68**(9), 1060–1076.

Loyall, J. and Mathisen S. (1993) 'Using dependence analysis to support the software maintenance process', in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 282–291.

Lyle, J.; Wallace, D.; Graham, J.; Gallagher, K.; Poole, J. and Binkley, D. (1995) *Unravel: A CASE Tool to Assist Evaluation of High Integrity Software*, Technical Report NIST IR 5691, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg MD, 124 pp.

McCabe, T. (1976) 'A complexity measure', *IEEE Transactions on Software Engineering*, **SE-2**(4), 308–320.

Melton, A.; Gustafson, D.; Bieman, J. and Baker, A. (1990) 'A mathematical perspective for software measures research', *Software Engineering Journal*, **5**(5), 246–254.

Müller, H.; Orgun, M.; Tilley, S. and Uhl, J. (1993) 'A reverse-engineering approach to subsystem structure identification', *Journal of Software Maintenance: Research and Practice*, **5**(4), 181–204.

Myers, G.J. (1978) *Composite/Structural Design*, Van Nostrand Reinhold, New York NY, 1978, 174 pp.

Nandigam, J.; Lakhotia, A. and Čech, G. (1999) 'Experimental evaluation of agreement among programmers in applying the rules of cohesion', *Journal of Software Maintenance: Research and Practice*, **11**(1), 35–53.

Ning, J.; Engberts, A. and Kozaczynski, W. (1993) 'Recovering reusable components from legacy systems by program segmentation,' in *Proceedings of the Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 64–72.

Ott, L. and Thuss J. (1989) 'The relationship between slices and module cohesion', in *Proceedings 11th International Conference on Software Engineering*, IEEE Computer Society Press, Washington DC, pp. 198–204.

Ottenstein, K. and Ottenstein [Ott], L. (1984) 'The program dependence graph in a software development environment', *SIGPLAN Notices*, **19**(5), 177–184.

Pressman, R. (1997) *Software Engineering: a Practitioner's Approach Fourth Edition*, McGraw Hill, New York NY, 852 pp.

Stevens, W.; Myers, G. and Constantine, L. (1974) 'Structured design', *IBM Systems Journal*, **13**(2), 115–139.

Tesch, D. and Klein, G. (1991) 'Optimal module clustering in program organization', in *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, Volume 2, IEEE Computer Society Press, Washington DC, pp. 238–245.

Weiser, M. (1984) 'Program slicing', *IEEE Transactions on Software Engineering*, **10**(4), 352–357.

Woodward, M. (1993) 'Difficulties using cohesion and coupling as quality indicators', *Software Quality Journal*, **2**(2), 109–127.

Zima, H. and Chapman, B. (1991) *Supercompilers for Parallel and Vector Computers*, ACM Press, New York NY, 376 pp.

Zimmer, J. (1990) 'Restructuring for style', *Software Practice and Experience*, **20**(4), 365–389.

Authors' biographies:

Byung-Kyoo Kang is a Senior Member of Technical Staff at Technology Deployment International, Inc. in Santa Clara California. He was a Senior Member of Technical Staff in the Software Engineering division of the Electronics and Telecommunications Research Institute (ETRI) in Korea in 1997 and 1998. He received a Ph.D. in Computer Science from Colorado State University in 1997. 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. His email address is kang@tdiinc.com

James M. Bieman is an Associate Professor in the Computer Science Department at Colorado State University. His research is focused on software design evaluation and improvement, and automated software testing. Jim is currently the Chair of the IEEE-CS TCSE Subcommittee on Quantitative Methods, and Chair of the Steering Committee for the IEEE-CS International Symposium on Software Metrics. He was General Chair of Metrics'93 and Metrics'97. He received a Ph.D. and an M.S. in Computer Science from the University of Southwestern Louisiana, a Master of Public Policy from the University of Michigan, and a B.S. in Chemical Engineering from Wayne State University. His email address is bieman@cs.colostate.edu