Extracting and Restructuring the Design of Large Systems

Song C. Choi and ***Walt Scacchi***, *University of Southern California*

This approach to reverse engineering first maps the resource exchange among modules, then derives a hierarchical design description using a system-restructuring algorithm.

Reverse engineering involves first generating a design description from an implementation description and then a specification description from the design description. It requires that you abstract four properties of a system:

• Structural: A system's structural properties are described by the resources exchanged among modules and subsystems through interconnected interfaces.

• Functional: A system's functional properties are described by the semantics of the exchanged resources. For example, for operational resources (those that perform an operation), you abstract precondition and postcondition assertions. For nonoperational resources (those that store a value), you abstract a type definition.

• Dynamic: A system's dynamic properties are described by the procedural algorithms that transform imported resources into exported resources. Dynamic properties are intramodular.

• Behavioral: A system's behavioral properties are described by the behavior of its objects (modules) in terms of the relations among objects, their attributes, and the actions that manipulate them. Behavioral properties are mainly intermodular.

The focus of this article is on extracting the structural and, to a lesser degree, functional and dynamic properties of large systems — systems composed of modules and subsystems. This process is equivalent to reverse-engineering a system-level design description.

The medium for our design description is a module interconnection language, NuMIL, which is described briefly in the box on p. 68. Once an initial design description is generated, our system-restructuring algorithm constructs a hierarchy of the system's modules and subsystems.[1]

0740-7459/90/0100/0066/$01.00 © 1990 IEEE

Design description

In system design, you specify the architectural structure and the procedural algorithms that manipulate exchanged objects (resources). A design specification is a network of operational modules that progressively transform required objects into resources provided.

Module interconnection languages are well-suited to describe such interconnected networks of module interfaces. An MIL provides formal constructs for specifying the various module interconnections necessary to assemble a complete system. You can describe the procedural algorithms in many ways, using either a program-design language or a functional specification language.

To generate a design description from an implementation description, you must analyze the source code written in an implementation language and generate its corresponding design description in an MIL. In NuMIL, you must generate the operational resources' precondition and postcondition assertions and restructure the configuration of modules into a hierarchy of subsystems and modules.

To do this, you must identify six characteristics of the system: its modules, the resources exchanged among modules, its modules' structural properties, its subsystems, its modules' dynamic properties, and its modules' functional properties. This article describes the first four steps, and Choi describes the other steps elsewhere.[2]

Module identification. There are two ways to recognize a module, which can be a single function (a term we use interchangeably with "procedure") or a collection of functions. One way is to consider each source-code file as a module. Another way is to perform a one-to-one transformation of a function in the source language to a module in NuMIL.

In the first method, we assume that the functions in a file are semantically related and form a cohesive logical module.[3] Because it is usually true that programmers group related functions into a file, although with varying degrees of functional cohesion,[3] it is reasonable to consider each file to be a module. Also, most compilers use the file as the compilation unit, and other tools such as editors, debuggers, and preprocessors and postprocessors use the file as their I/O medium. Finally, when you choose to consider every source file as a module, you can readily query for other information, including its owner, latest author, latest revision, access conditions, file-path directory, and network file server, all of which can provide useful configuration-management information.[4,5] We chose to use this method.

To generate a design description from an implementation description, you must analyze the source code written in an implementation language and generate its corresponding design description in an MIL.

The second way to recognize a module uses a one-to-one transformation. However, this results in modules that consist of a single function, with fixed (high) cohesion. This may result in an unnecessarily complex and misleading system design structure.

Resource exchange. Resources exchanged among modules include data types, procedures, and variables.

It is easy to identify variables and data types because they are declared either locally or globally. It is also easy to identify functions and procedures because their syntactic declarations are recognizable.

However, in languages such as C you must also analyze preprocessor extensions (include files, definitions and replacements, and conditional-compilation commands) and data-type conversions to determine what resources can be exchanged between communicating modules.

Structural properties. To identify the modules' structural properties, you use static analysis to divide the exchanged resources into those that are provided and those that are required. For example, if a module uses a function call, it requires the called function.

You can examine the syntax of exchanged parameters to determine if a variable provides or requires a value. Also, you can determine required and provided resources in used variables and declared data types by examining their syntax.

Subsystem identification. In NuMIL, a system is composed of subsystems, and a subsystem is composed of other subsystems or modules. Therefore, a system is a hierarchy of subsystems and modules, where subsystems correspond to interior nodes and modules correspond to leaf nodes in the hierarchy.

Because implementation descriptions are flat (they give no explicit construction of modules and subsystems), it is not obvious how to construct a system hierarchy. As we describe later, we have developed an algorithm to do this structuring.

Dynamic properties. To identify the modules' dynamic properties, you con-

Specifying design with NuMIL

Recently, module interconnection languages have been used to support configuration management,[1] reusable-component composition,[2] and incremental system construction and verification,[3] in ways independent of the source-code language.

A module interconnection language describes a system as a collection of families of several quasi-autonomous modules and subsystems, where each subsystem is an aggregation of one or more modules or subsystems. In turn, families represent the set of module or subsystem versions that conform to a specified interface.

Each module provides a set of resources — entities such as data types, functions, procedures, and variables — that can be declared in the host language. Therefore, each module has a set of resources that other modules may use, and in turn it may require a set of resources that is provided by other modules.

The exchange of these resources leads to interdependencies among the modules. It is common practice to isolate resource exchange to a well-defined interface: an imaginary port through which the resources are exchanged.

NuMIL[1] views the dependencies between modules as having a functional and structural character that can be verified through static analysis. In NuMIL, the resources' functional properties are divided into two classes: operational and nonoperational. Operational resources perform an operation, such as a procedure, function, or subroutine, and their functional properties are specified with preconditions and postcondition assertions. Nonoperational resources, such as data types or variables, do not perform a particular operation but instead store values, and their functional properties are defined with type definitions.

You can specify functional properties many ways, such as with logical predicates or some other formal specification technique.

Figure A shows an example NuMIL module template with its structural properties (operational properties are not shown) and its corresponding C source code. This example uses the logical predicates to specify the functional properties.[1]

A subsystem family is similar to a module family. Each subsystem family has a specification to be satisfied by all of its members, and each member of the subsystem family is essentially a composition of modules or other subsystems.

```
subsystem parser is
      provides parse_tree, ...;
      configurations
            parser1 = {lexer, ....}
end parser

module lex is
   provide lexer, token, token_str, flag;
   require input_char, state;
   implementation version { ldeg :
                              realization
                                 "/usr/castor3/Song/Files/imp.lex.c",
                              owner
                                 "sfmaster (System Factory Project)",
                              system
                                 "pollux.usc.edu (128.125.1.16)",
                              revision date
                                 " 12/08/1988 16:35:26 PST"
                              }
end
```
(a)

```
#include                    "ldeg.var"

lexer(ch, token, token_str, found)
int             ch;
char            *token, *token_str;
int             *found;
{     int      space,letter,digit;
      int      long_op = (0);
      char     literal = "";
      int      number;
      char     *charp;
      extern  int    lex_state;
      int      index;

      { *found = false;
         switch ( lex_state ) {
...
      }
}
```
(b)

Figure A. (a) Template specification for a NuMIL module and **(b)** its source code.

References

1. K. Narayanaswamy and W. Scacchi, "Maintaining Configurations of Evolving Software Systems," *IEEE Trans. Software Eng.*, March 1987, pp. 324-334.
2. G.E. Kaiser and D. Garlan, "Melding Software Systems from Reusable Building Blocks," *IEEE Software*, July 1987, pp. 17-24.
3. D.E. Perry, "Software Interconnection Models," *Proc. Ninth Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., 1987, pp. 61-69.

vert the procedural algorithm written in an implementation language into a language that specifies procedural algorithms — a program-design language, for example. This translation from one language into a more abstract language should be straightforward, since many program-design languages use generalized constructs found in common languages.

Functional properites. To identify the modules' functional properties, you must find the assertions that characterize the resources. For operational resources, you must identify the precondition and postcondition assertions; for nonoperational resources, you must identify the static assertion of a type, for example.

It is easier to identify the nonoperational resources' assertions because they involve only static semantics. It is much harder to find preconditions and postcondition assertions because this involves abstracting the dynamic semantics of an operational resource. To do this, you must analyze and understand how the implementation description of the operational resource is manipulating input. There is no easy solution to this problem for large systems.[2]

Subsystems' structural properties

Because an implementation description typically does not give an explicit structure for modules and subsystems, we need an algorithm that can construct a hierarchical structure from an implementation description. Deriving such an algorithm first requires an understanding of how to map resource exchange among modules.

Mapping. A resource-flow diagram shows the relationships among modules in terms of the resources they exchange. Given source code, you can derive the resource flow automatically by analyzing the functions, procedure calls, and data items that are input to a module. Alex Wolf and colleagues have designed an RFD using directed graphs: If module *A* provides one or more resources to module *B*, there is an arc from *A* to *B*.[6]

Our representation of an RFD is an undirected graph because in our scheme it is sufficient to know that resources are exchanged; you need not know the direction of the exchange. For example, in the RFD in Figure 1a, module 5 exchanges resources with modules 2, 4, and 6.

A resource-structure diagram shows the control relationship between modules and a control module, commonly called a subsystem: You configure a set of modules so they are controlled by a subsystem. A system consists of subsystems and modules, and a subsystem consists of other subsystems or modules. An RSD represents a system's architectural design.[3] Figure 1b shows an example RSD for the modules in Figure 1a.

To generate a design description, you use our algorithm to map from an RFD to an RSD, a process we call *system restructuring*.

The term "restructuring" has been used to refer to the imposition of a clear control structure within source-code modules. For example, you restructure old, unreadable code to bring it into compliance with structured-programming concepts. Or you restructure a specific construct, such as a loop, so it is more efficient and better understood. To us, the current understanding of restructuring is as an intramodule process: restructuring-in-the-small.

But our use of the term is quite different. Our restructuring process changes the relationship between modules from a resource-exchange relationship to a hierarchical relationship. Because it occurs at the module and system level, our restructuring is an intermodule process: restructuring-in-the-large.

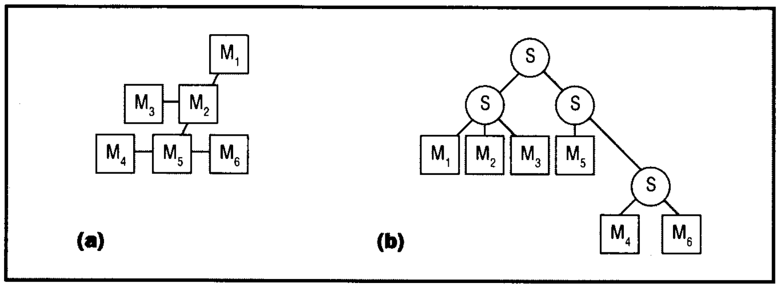


Figure 1. (a) An example resource-flow diagram and **(b)** resource-structure diagram. "M" stands for module; "S" for subsystem.

Definitions. There are many ways to map an RFD to an RSD. A simple RSD is flat, with all system modules belonging to one subsystem. But any number of modules can be controlled by a subsystem, so there are an arbitrary number of choices of RSDs. We need some criteria to assign modules to subsystem. We based our restructuring criteria on minimizing module coupling[3] and minimizing alteration distance.

To explain these concepts, we must first define some terms:

- A subsystem's *control visibility* is all the modules attached to it (which are controlled by it). The control visibility of subsystem *S* is the set of modules attached to *S*.
- A subsystem's *alteration visibility* is all the modules attached to it that are affected by an alteration to a module. The alteration visibility of subsystem *S* is the set of modules affected by an alteration to a module attached to *S*.
- A module and a subsystem are *coupled*. Coupling is a measure of the strength of association established by a connection between modules.[3] Strong coupling complicates the system because a module is harder to understand and modify when it is strongly related to other modules. Systems designed with weak coupling are less complex because they minimize the paths along which changes and errors can be propagated into other parts of the system. W.P. Stevens and colleagues have identified seven types of coupling,[3] the most desirable being no direct coupling between modules. We define module coupling as the number of modules under the same control visibility. Furthermore, we define the coupling of a system or subsystem as the sum of all the couplings in the system or subsystem.

It is desirable to minimize coupling in a system. You can do this by reducing the fan-out factor of a subsystem by requiring the number of modules attached to a subsystem (its control visibility) be kept low.

- The *alteration distance* between modules is a measure of the distance between an altered module and the affected module. The alteration distance is zero if one subsystem controls both the altered and affected module. Otherwise, the distance 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.

In an RSD, it is desirable that the control visibility and the alteration visibility be the same — the alteration distance should be zero. This means that we want module alterations to be as local as possible. It follows from this that the higher the control

```
/* A graph G, which consists of nodes and edges, will be divided into
/* subgraphs. The mechanism for breaking the graph into subgraphs is
/* finding the biconnected components of the graph. The articulation points
/* are the connecting nodes between the components. If there are no
/* articulation points, the graph cannot be divided into subgraphs.
step 1: Find the articulation points A(j) of the Graph G.
        If there is no articulation point then
          it is done and make each node of the graph
          a component of the subsystem.
        Endif
step 2: Find the biconnected components of the Graph G.
step 3: Divide the Graph G into subgraphs G(i) with
          1 <= i <= number of biconnected components.

/* Each articulation point is member of at least 2 different
/* subgraphs. Remove all articulation points from the subgraphs.
step 4: For all articulation points A(j) do
          remove A(j) from from G(i) thereby building G'(i) = (V', E'),
             an induced graph of G(i).
        Endfor

/* Build a subsystem S for each articulation point and
/* the articulation point becomes a component of the subsystem S.
/* We also create as many subsystems as there are subgraphs with that
/* articulation point and these subsystems are attached to S.
step 5: For all articulation points A(j) do
          create a subsystem S and make A(j) a component of the subsystem S.
          For each G(i) that contains A(j) do
             if V' is not empty then
               create a subsystem S' and attach it to the subsystem S.
               attach vertices in V' that are adjacent to A(j) to S'
               build an induced graph G"(i) by removing
                 the adjacent vertices of A(j) from G'(i).
               assign G"(i) to G and goto step 1.
             endif
          Endfor
        Endfor

/* Make sure that there is no subsystem with a single node.
step 6: If a subsystem consists of one node then
          Merge the node with a higher level subsystem.
        Endif
```

Figure 2. System-restructuring algorithm.

visibility, the better the alteration's localization.

An extreme case is when all modules are controlled by one subsystem. The two visibilities are then clearly the same. However, this structure contradicts the minimum-coupling objective. Therefore, if we want to accomodate both objectives, we cannot set the control visibility equal to the alteration visibility. In other words, we want to *minimize* the alteration distance by keeping all the affected modules that are not part of the control visibility of the altered module as close as possible.

Algorithm. Our restructuring algorithm accommodates both the minimum-coupling and minimum-alteration-distance objectives.

Figure 2 shows our algorithm. We begin with a graph in which each module is a node and each path for resource exchange between modules is an edge. The initial system is an undirected graph, which we assume is connected. In other words, if some modules are not exchanging resources then they are neither part of nor useful to the system.

The first two steps of the algorithm apply a biconnectivity algorithm, which divides the into subgraphs connected only by articulation points, the nodes, which are found in the second step. These two steps are detailed elsewhere;[7] the remaining steps are unique to our approach.

Steps 3 through 6 illustrate the analysis done on one edge of an RFD to map it to an RSD. Figure 3 shows an RFD (a line indicates a resource exchange between modules). Figure 4 shows three

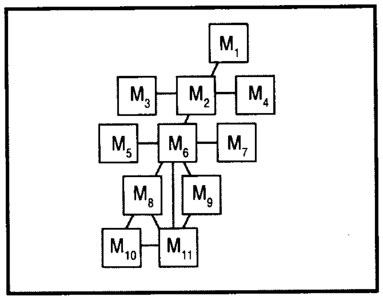


Figure 3. A resource-flow diagram of modules.

structured diagrams: Figure 4a shows restructuring based on maximum control visibility (minimum alteration distance); Figure 4b illustrates minimum coupling; Figure 4c shows the structure diagram after we applied the restructuring algorithm.

Performance

The performance of our algorithm makes it clearly practical. It requires linear time in the number of edges in the RFD graph to find the biconnected components and articulation points.[7] Because the maximum number of articulation points is $O(n)$, steps 4 and 5 require a similar number of operations. Step 5 is iterated by the number of articulation points times the number of biconnected components. Because both numbers are $O(n)$ in worst cases, the iteration is an $O(n^2)$ algorithm. Finally, because the number of edges is always less than n^2, the performance is proportional to $O(n^2)$ in each iteration.

Previously, we identified modules by considering each source-code file to be a module. However, you may prefer to construct an RFD with individual functions. In this case, you would cluster a collection of functions as a module. To determine what should be in this cluster, you apply the algorithm to the RFD associated with individual source-code functions rather than modules: functions in each subsystem in step 5 subtracted by all the functions in the induced graphs $G"(i)$. Such a restructuring of functions into logical modules and subsystems might then suggest a basis for modifying the source-code files to be consistent with the implicit system design.

Once you have extracted a system's structural properties into NuMIL, you can visualize its configuration.[5] You can also use the NuMIL processing enviroment[3-4] to manage the configuration of multiversion systems. You can also perform incremental analysis of modifications represented as generated NuMIL descriptions to minimize recompilation and retesting.[1] Other tools in our software-engineering environment let you apply integrity checks to determine inconsistencies, incompleteness, or intraceability,[1,5] thus supporting the evolution of large software systems in both forward and reverse directions.

By using our approach to extract and restructure system designs, you can apply advanced software-engineering tools and techniques to large systems that previously were difficult to understand and modify. ❖

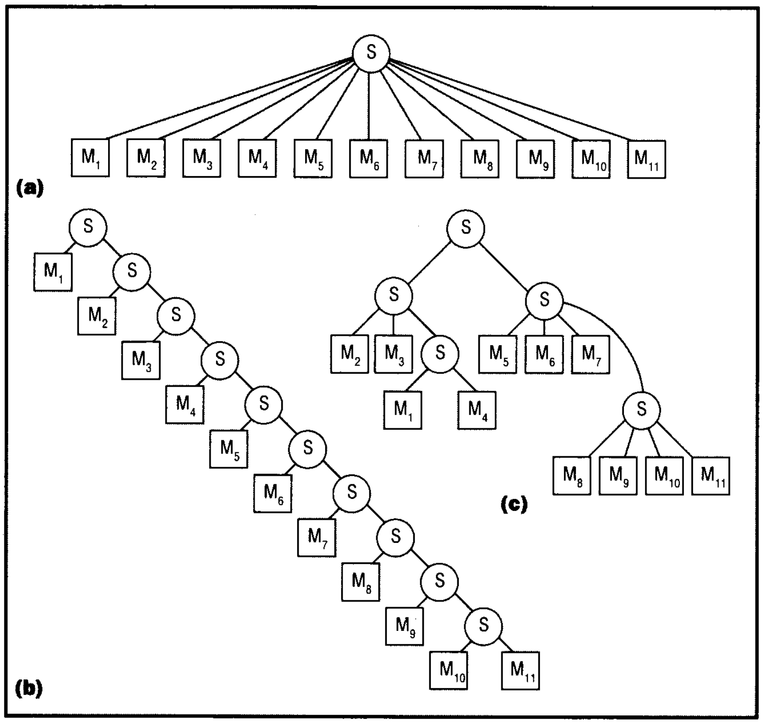
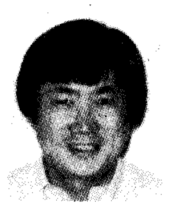


Figure 4. A resource-structure diagram based on **(a)** maximum control; **(b)** minimum coupling; and **(c)** using our restructuring algorithm.

References

1. S.C. Choi and W. Scacchi, "Assuring Correctness of Configured Software Descriptions," *ACM Software-Eng. Notes*, Nov. 1989, pp. 65-75.
2. S.C. Choi, *Softman: An Environment Supporting the Engineering and Reverse Engineering of Large Software Systems*, PhD dissertation, Univ. of Southern California, Los Angeles, 1989.
3. W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured Design," *IBM Systems J.*, Jan. 1974, pp. 115-139.
4. K. Narayanaswamy and W. Scacchi, "A Database Foundation to Support Software System Evolution," *J. Systems and Software*, Jan. 1987, pp. 37-48.
5. P. Garg and W. Scacchi, "A Software Hypertext Environment for Configured Software Descriptions," *Proc. Int'l Workshop Software Version and Configuration Control*, Tuebner, Stuttgart, West Germany, 1988, pp. 326-343.
6. A. Wolf, L. Clarke, and J. Wileden, "A Model of Visibility Control," *IEEE Trans. Software Eng.*, April 1988, pp. 512-520.
7. A. Aho, J. Hopcraft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974, pp. 179-187.



Song C. Choi is a research associate in computer science at the University of Southern California. His research interests are forward and reverse software engineering and language-directed editing environments.

Choi received a Diplom Physiken from the University of Karlsruhe and an MS and PhD in computer science from the University of Southern California.

Walt Scacchi is a research associate in computer science at the University of Southern California. His research interests are large-scale software engineering, software factories, and social analysis of computing.

Scacchi received a PhD in computer science from the University of California at Irvine. He is a member of the IEEE Computer Society, ACM, AAAI, CPSR, and the Society for the History of Technology.

Address questions about this article to the authors at Computer Science Dept., University of Southern California, Los Angeles, CA 90089-7424; Internet scacchi@pollux.usc.edu