# On Controlling the Interactions of Software Components: An Evolutionary Perspective


Spiros Mancoridis
Dept. of Math. and Computer Science
Drexel University
3141 Chestnut Street
Philadelphia, PA, USA 19104
e-mail: smancori@mcs.drexel.edu
Phone: (215) 895-6824
FAX: (215) 895-1582

Richard C. Holt
Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto, ON, Canada M5S 3G4
e-mail: holt@turing.toronto.edu
Phone: (416) 978-8726
FAX: (416) 971-2326



**Abstract**

Being able to control the interactions among software components is essential for the successful initial development and subsequent maintenance of large and complex software systems. The control of these interactions occurs at various levels of a software specification. In this paper we describe how notations are used to specify constraints on the interactions of software components at the architectural, module, and block structure level. Our presentation gives an evolutionary perspective by showing that scoping rules for constraining architectural interactions have evolved from rules for constraining module interactions, which in turn have evolved from scoping rules for controlling the interactions that occur between components in block structured language.


## 1 Introduction

Large software systems consist of many components (resources) that interact in intricate ways. Being able to control these interactions is critical for both the initial development and long-term maintenance of a software system. Early programming languages, such as ALGOL-60 [12], recognized the importance of limiting the interactions of fine-grained resources such as variables, constants, and types by encapsulating them into block structures and using scoping rules to dictate how these resources were able to interact.

As software systems grew in size, block structures were supplemented by constructs such as modules and classes. Many modern programming languages, such as Modula-3 [1], support the encapsulation of a variety of resources, such as procedures and functions, into modules. In these languages, block structure scoping is still used to control the interactions within procedure blocks. However, it is supplemented by module scoping rules, which are used to restrict the visibility and, hence, interactions of the resources that are encapsulated inside a module. Modules permit programmers to list the resources of a module that are accessible to other modules. This list is called the *interface* of the module. Module scoping rules determine the visibility of encapsulated resources by means of *import* and *export* directives.

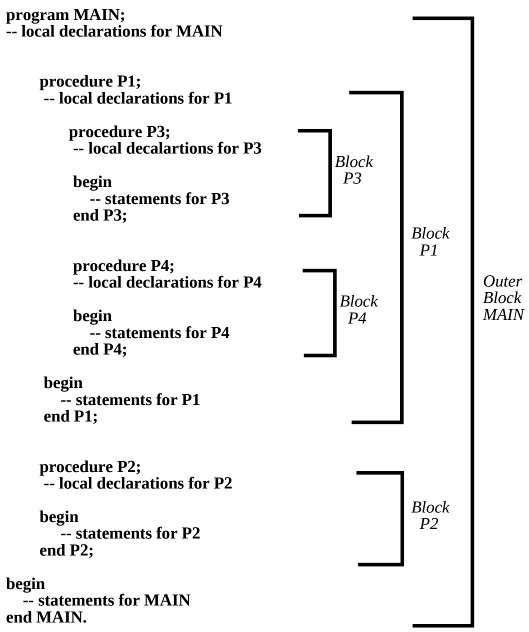


Figure 1: **Static Block Structure of a Program**

Architectural notations, such as the Software Landscape [13], were developed in order to encapsulate coarse-grained resources, such as entire modules, into subsystems. These subsystems have interfaces to, and dependencies with, other subsystems within the same system. The scoping rules at this level are used to control the dependencies among the various subsystems in a subsystem hierarchy. To a large extent, these scoping concepts have evolved from concepts found in programming languages that support modules and classes.

Our presentation of scoping rules gives an evolutionary perspective. In Section 2, we begin by describing block structure scoping using ALGOL-60 [12] as an example. In Section 3, we describe module scoping using Modula-3 [1] as an example. These two kinds of scoping pre-date subsystem scoping, which is described in Section 4. We describe subsystem scoping using the Software Landscape [13] architectural notation as an example.

## 2 Block Structure Scoping

One of the earliest programming language mechanisms for controlling the scope of resources is the block structure scoping of ALGOL-60 [12]. Because of its elegance and efficient implementation, block structure scoping has been adopted by a wide range of programming languages. In a typical block structured program (Figure 1), each procedure contains a set of nested blocks. The declarations of each block define the local environment of a block. If a resource is referenced within the body of a block and no local declaration for that resource exists, then the reference is considered a reference to a declaration within the block that immediately encloses the first block.

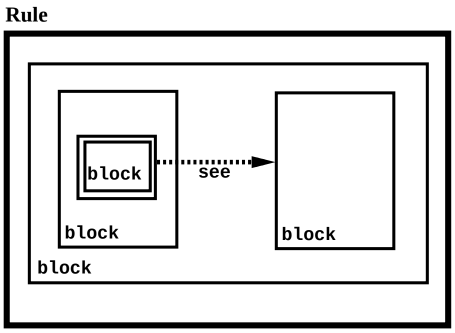


Figure 2: **ALGOL-60 Block Scoping**

If no declaration exists there, then it refers to a declaration in the block that immediately encloses the first block, and so on. If a block contains another block, then any local declarations within the inner block are completely hidden from the outer block.

We use the ASF (Architectural Style Formalism) [8], (see Figure 2) to define the *see* relation, which specifies the visibility among ALGOL-60 blocks. In ASF, a box with a double frame represents a block that is transitively contained within another block. This transitivity is reflexive, hence the box with a double frame can be the same as the box that contains it, or it can be its child, or its grandchild, and so on. Hence the rule states that sibling blocks see (dashed arrow) each other. Note that the siblings we are referring to need not be proper siblings because there is no relation that indicates they are distinct. Hence, in the rule of Figure 2, the two nested blocks within the outermost block may be the same block. This implies that each block also sees itself. The rule also states that nested blocks see outwards (*i.e.,* see their uncles), but blocks cannot see within other blocks. A simplifying assumption made here, and throughout this paper, is that the *declaration before use* scoping rule has been ignored.

The hiding of local block declarations was an important early step towards developing more powerful language mechanisms for controlling the scope of resources. Another major step was the evolution of data types, which led to module and class scoping.

# 3 Module and Class Scoping

The first data types were primitives for representing variables of type character, integer, real, and so on. Primitive data types became the basis of *composite homogeneous data types* such as arrays, and *composite heterogeneous data types* such as records. In terms of scoping, these composite data types have traditionally had an open scope. This means that access to a record or array variable usually implies access to its constituent fields or elements.

The generalization of records to modules and classes is based on the concept of *Abstract Data Types (ADTs).* An ADT is defined as a set of operations that may be performed on a set of entities. Modules and classes are implementations of ADTs, where data entities and operations on those data entities are encapsulated so that clients of the modules or classes cannot access the data except through an interface. An interface consists of the signatures of the set of visible module or class operations. The implementations of these operations, and any private operations whose signature is not part of the interface, are not visible to clients. Since clients can only access modules

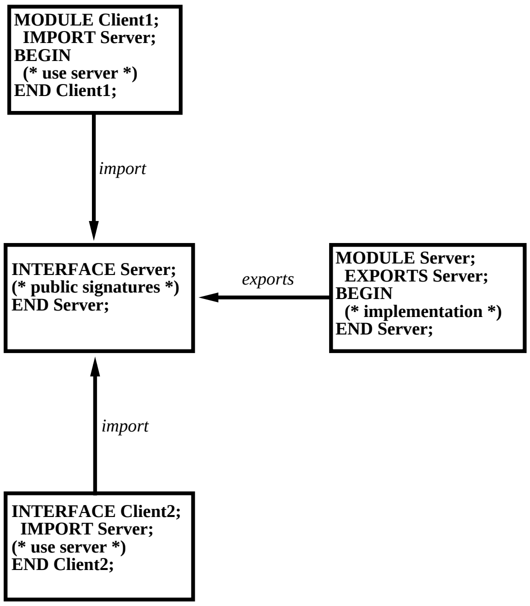


Figure 3: **Modula-3 Example**

and classes through their interfaces, the implementations of their operations may be modified or replaced without changing the clients, as long as the signatures (and the corresponding semantics of their implementations) of the interface remain unchanged. Early programming languages that support modules or classes include Simula [3], Ada [16], Euclid [7, 6], and Modula-2 [19]. More recent languages include Eiffel [10], Modula-3 [1], Turing [4], and C++ [18].

Figure 3 shows an example of the architecture of a Modula-3 program. We have used boxes and edges to make the architectural dependencies in the code obvious. In Modula-3, a resource (*e.g.,* variable, constant, type, procedure) is visible to a module if it is declared in the module or in an interface that is imported or exported by the module. An interface (*e.g., Server* and *Client2*) is a group of declarations that specify only the signature, not the implementation. In Figure 3, module *Server* exports an interface *Server* and supplies implementations for the procedures declared in the interface. Client module *Client1* and client interface *Client2* each import interface *Server* to make the resources declared in *Server* visible to them.

In Figure 4, we define Modula-3 scoping by defining the *see* relation using four rules.

- **Rule (1):** Within a module or an interface (m_i), sibling resources (res) see each other and themselves.

- **Rule (2):** If a module (m) exports an interface (i), the resources of the module see the resources of the interface.

- **Rule (3):** If a module or an interface (m_i) imports an interface (i), the resources of the importing module or interface see the resources of the imported interface.

- **Rule (4):** If a module or interface (m_i) imports an interface resource, the resources of the importing module or interface see the imported resource.

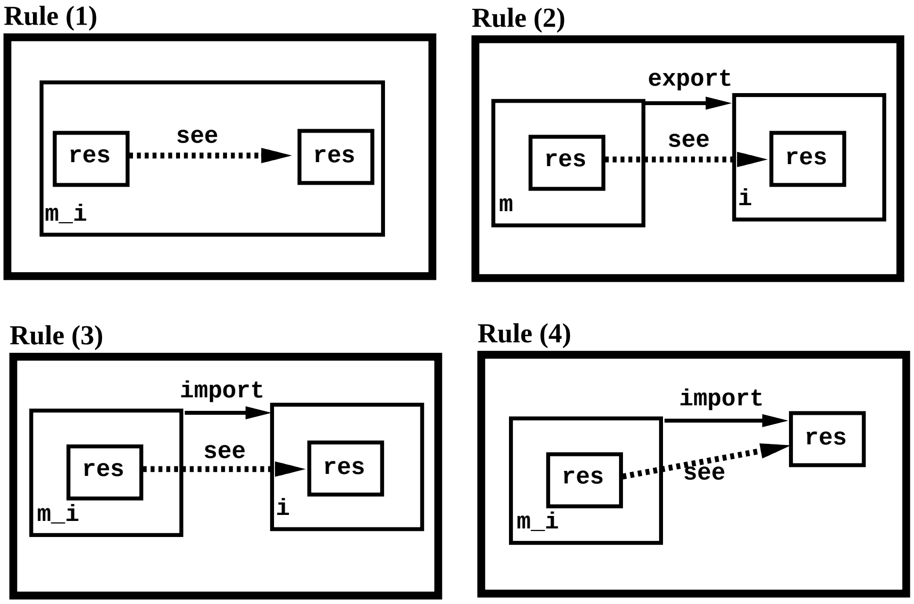


Figure 4: **Modula-3 Scoping Rules**

This concludes our discussion of module and class scoping. Next, we describe subsystem scoping as a generalization of module and class scoping.

# 4 Subsystem Scoping

The demand for high-level architectural abstractions led to the development of textual, and more recently, visual notations for specifying software architectures, called MILs (Module Interconnection Languages) [15]. MILs are used for specifying static architectural dependencies among software components (resources or subsystems). Subsystems may contain modules and classes as well as other subsystems to form a containment hierarchy which models the system decomposition structure. The granularity of resources varies with the notation. The Cooprider MIL [2], for example, assumes that resources are procedures, variables, types, and so on. However, the Software Landscape MIL [13, 5, 9] uses more coarsely-grained resources, such as modules and classes, as the "currency of exchange" between subsystems. The emergence of formal methods for programming has resulted in MILs, like Inscape [14] and NuMIL [11], that also feature pre- and post-conditions as resources.

The dependencies among the components in a MIL specification are not arbitrary; they must conform to a set of scoping rules that is specific to the MIL. The Software Landscape is a visual MIL that has adopted several ideas from an earlier visual MIL called ARCH [17]. Figure 5 is an example of a Software Landscape taken from Penny's Ph.D. dissertation [13]. The containment relation is shown by drawing a box representing the contained entity inside of the box representing the containing entity. The "S" in the top-right corner of some boxes (*e.g., S1*) indicates that these are structure objects (subsystems). The "I" in the top-right corner of the *S2* box, within box *S1*,

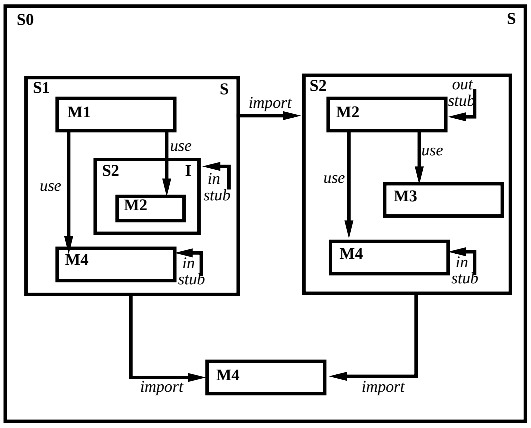


Figure 5: **Software Landscape Example**

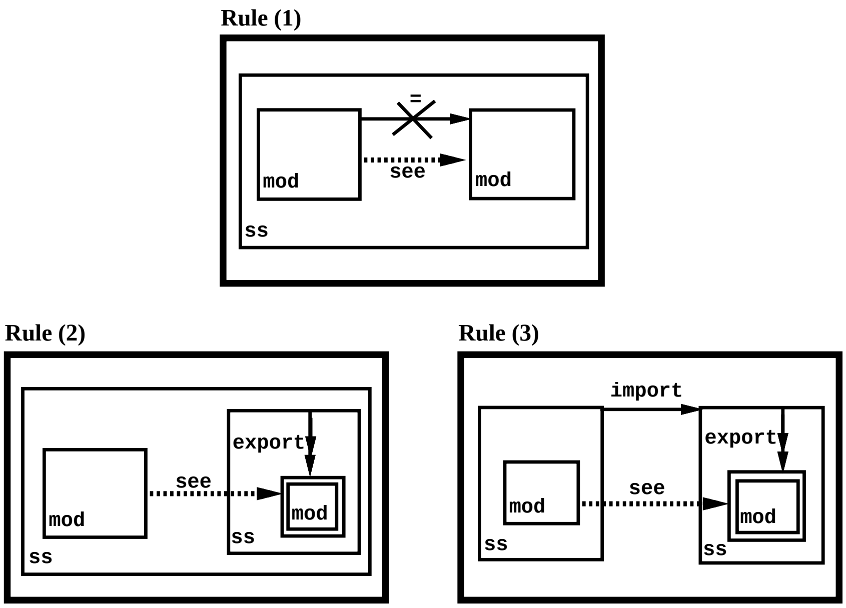


Figure 6: **Software Landscape Scoping Rules**

indicates that only the exported contents of a structure object box are shown. The boxes without letters in their top-right corner are *meaning* objects (modules or classes). To import entities into the scope of a subsystem, a replica of the imported entity must be drawn inside the subsystem that is importing. The replicas are indicated using the *in-stub* visual symbol. For example, in Figure 5, module *M4* is imported into both subsystems *S1* and *S2* by replicating the module into each subsystem and annotating it with the *in-stub* symbol. If the imported entity is a subsystem, such as *S2* in the example, then only its exported contents (those with an *out-stub*) are imported.

Figure 6 specifies the visibility among modules in a Software Landscape by defining the *see* relation using three rules.

- **Rule (1):** Proper sibling modules (mod), that is two distinct modules that are contained in the same subsystem (ss), see each other.
- **Rule (2):** Modules see the modules that are transitively exported (double arrow) by their sibling subsystems.
- **Rule (3):** If a subsystem imports another subsystem, any modules in the first subsystem see the modules that are transitively exported by the second subsystem.

There are additional rules, which we will not give, that specify when a subsystem can import or export a subsystem or module.

Scoping rules at the level of Software Landscapes are used to control the structure of the overall architecture of large software systems. The details of these rules vary from MIL to MIL. These details are not important; what is important is that such rules can be a powerful aid in maintaining good structure of large-scale software.

# 5 Conclusions

MILs can be used to control the interactions among the subsystems of large software systems. This control is affected by rules that determine which interactions are legal. In this paper, we point out that these rules are an evolution of rules that determine the scope of imported and exported components at the level of modules and classes. In turn, module and class scoping rules are an evolution of earlier rules such as ALGOL-60's block structure scoping rules. We used the ASF notation to provide a simple way of specifying and visualizing scoping rules at various levels, from blocks to modules and classes to subsystems.

## References


[1] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 Language Definition. *ACM SIGPLAN Notices 27*, 8 (August 1992), 15–42.

[2] Cooprider, L. W. The Representation of Families of Software Systems. Tech. Rep. CMU–CS–79–116, Computer Science Department Carnegie-Mellon University, April 1979.

[3] Dahl, O. J., and Nygaard, K. SIMULA – An Algol-based Simulation Language. *Communications of the ACM 9* (1966), 671–678.

[4] Holt, R. C., and Cordy, J. R. The Turing Programming Language. *Communications of the ACM 31*, 12 (December 1988), 1410–1423.

[5] Holt, R. C., Penny, D. A., and Mancoridis, S. Multicolour Programming and Metamorphic Programming: Object Oriented Programming–in–the–Large. In *Proceedings of the 1992 IBM CASCON Conference* (November 1992), pp. 43–58.

[6] Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. J. Report on the Programming Language Euclid. *ACM SIGPLAN Notices 12*, 2 (February 1977), 1–79.

[7] Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. J. Report on the Programming Language Euclid. Tech. Rep. CSl-81-2, Xerox Palo Alto Research Centers, October 1981.

[8] Mancoridis, S. *Controlling the Interactions of Architectural Design Components Using Scoping Rules.* PhD thesis, Department of Computer Science, University of Toronto, 1996.

[9] Mancoridis, S., Holt, R. C., and Penny, D. A. A Conceptual Framework for Software Development. In *Proceedings of the Twenty–First ACM Computer Science Conference* (February 1993), pp. 74–80.

[10] Meyer, B. *Object-Oriented Software Construction.* Prentice Hall International, Englewood Cliffs, New Jersey, 1988.

[11] Narayanaswami, K., and Scacchi, W. Maintaining Configurations of Evolving Software Systems. *IEEE Transactions on Software Engineering 13*, 3 (March 1987), 324–334.

[12] Naur, P. Revised Report of the Algorithmic Language Algol 60. *Communications of the ACM 6*, 1 (1963), 1–17.

[13] Penny, D. A. *The Software Landscape: A Visual Formalism for Programming–in–the–Large.* PhD thesis, Department of Computer Science, University of Toronto, 1992.

[14] Perry, D. E. The Inscape Environment. In *Proceedings of the 11th IEEE International Conference on Software Engineering* (Pittsburgh, Pennsylvania, 1989), pp. 2–12.

[15] Prieto-Diaz, R., and Neighbors, J. M. Module Interconnection Languages. *The Journal of Systems and Software 6* (1986), 307–334.

[16] Rogers, R. W. *Ada: Language, Compilers, and Bibliography.* Cambridge, 1984.

[17] Schwanke, R. W., Altucher, R. Z., and Platoff, M. A. Discovering, Visualizing, and Controlling Software Structure. In *Proceedings of the Fifth International Workshop on Software Specification and Design* (Pittsburgh, Pennsylvania, May 1989), pp. 147–150.

[18] Stroustrup, B. *The C++ Programming Language.* Addison–Wesley Publishing Company, Reading, Massachusetts, 1986.

[19] Wirth, N. *Programming in Modula-2.* Springer Verlag (2nd ed.), 1983.