Introduction to
A Unified Treatment of
Interface Control and Program Structure

Lori A. Clarke
Jack C. Wileden
Alexander L. Wolf

COINS Technical Report 82-32
December 1982

Computer and Information Science Department
University of Massachusetts, Amherst
Amherst, Massachusetts 01003

This work was supported in part by the National Aeronautics and Space Administration under NASA grant NAG1-115.

Abstract

Since its introduction in Algol60, nesting has been the predominant mechanism for achieving interface control and program structuring in modern programming languages. But nesting is inadequate for achieving some desirable software properties, such as information hiding, and is antithetical to others, such as program readability, incremental development and separate compilation. This paper introduces a small, integrated set of programming language concepts that comprises a unified mechanism for interface control and program structuring. There are two aspects to our proposed mechanism. First, we propose a program structure that supports information hiding, separate compilation and managerial control. Second, our proposal includes language contructs that, in conjunction with this program structure, provide precise interface control.

1. Introduction

Even in small and fairly simple computer programs, the issues of interface control and program structure are more important than has generally been acknowledged. As software systems become larger and more complex, however, these "programming-in-the-large" [DERE76] issues assume an even greater prominence. Especially in such software systems there is a real need for mechanisms that:

- facilitate readability,
- support information hiding,
- encourage logical system organizations,
- support separate compilation, incremental development, and maintenance, and
- provide managerial control capabilities.

Yet no existing programming language mechanisms address these needs in a completely acceptable fashion.

Since its introduction in Algol60, nesting has been the predominant mechanism for achieving interface control and program structuring in modern programming languages. But nesting is inadequate for achieving some desirable software properties, such as information hiding, and is antithetical to others, such as program readability, incremental development and separate compilation [CLAR80, HANS81]. Thus, particularly for large complex software systems, more versatile and powerful mechanisms for interface control and program structuring are required.

An additional impetus for the introduction of improved interface control and program structuring mechanisms is the emergence and increasing acceptance of automated software development support environments. The advent of such environments allows us to consider a programming language not as an isolated tool but rather as one component in an integrated suite of tools. This view is clearly reflected in the design of Ada and its environment [DOD82, BUXT80].

The prospect of having tools for constructing, viewing, manipulating and analyzing programs opens a wide range of possibilities for programming language organization and constructs. For example, the availability of tools to assist in the construction of a program [MEDI81, TEIT81] effectively counteracts the traditional resistance to redundancy in programming languages.

This paper introduces a small, integrated set of programming language concepts that comprises a unified mechanism for interface control and program structuring. These concepts provide means for controlling both components of visibility -- that is, the accessibility of external entities from within a given program unit and the availability of a unit and the entities comprising it to other program units in the software system. Our mechanism is illustrated using language constructs whose syntax is Ada-like. This syntax is merely a vehicle for conveying our ideas, however, and should not be construed as a concrete proposal for adding specific constructs to Ada or any other existing language.

In section 2 we discuss existing mechanisms for interface control and program structuring. Section 3 details our proposed mechanism, while section 4 illustrates it with an example. Future directions are mentioned in section 5.

2. Related Work

This section briefly compares and relates our proposed mechanism to many of the mechanisms for interface control and program structuring developed in previous language design efforts. The first mechanism examined is the primitive notion of nesting as defined in Algol60. Encapsulation and import/export concepts are then considered. In particular, we consider Ada [DOD82], Alphard [SHAW81], CLU [LISK79], Euclid [POPE77], Gypsy [AMBL77], Mesa [MITC79], Modula [WIRT77] and Protel [CASH81]. Finally, module

interconnection languages are examined.

In nested languages, such as Algol60, interface control is described by a simplistic tree structure. For each program unit, the accessibility of external entities from within the program unit as well as the availability of that unit to other program units is determined by its position in the tree structure of the program. Such a tree structure is not general enough to describe the wide range of possible interface relationships among program units, which are more appropriately represented by a directed graph. Therefore, nesting usually provides weaker interface control than desired. For example, a subprogram's "local" entities are unavoidably available to other program units nested within that subprogram. Furthermore, nesting forces a program to take the form of a single, monolithic unit, which makes separate compilation as well as managerial control of interfaces extremely cumbersome. We have argued elsewhere [CLAR80] that for these reasons nesting is an inappropriate interface control and program structuring mechanism. In our proposal, as well as in CLU and Gypsy, the nesting of program units is not supported. While nesting is supported in Ada, Alphard, Euclid, Mesa, Modula, and Protel, its use can, with some effort, be avoided in these languages (see, for example, [CLAR80]).

Since the development of Algol60, a variety of attempts have been made to compensate for the inadequacies of nesting. The languages considered here have relied, to a greater or lesser degree, on the concepts of encapsulation and import/export to describe which entities are accessible and available in a program. In its most general form, although not the one used in all of these languages, an encapsulation serves to group related program units, objects, and types. Examples of encapsulation constructs include the Ada package, Alphard form, CLU cluster, and Modula module. Our proposal combines encapsulation and augmented import/export concepts to create a mechanism

capable of describing the desired accessibility and availability of entities in a program more precisely than is possible in any of the other languages.

Selective accessibility is the ability to import arbitrary subsets of the available entities. This capability is provided in our proposal as well as in Modula, Euclid and Mesa. In other languages, only all or none of the available entities from an encapsulation can be imported, which frequently results in some entities being made accessible unnecessarily. Selective availability, which is the corresponding ability to make specific entities available to specific program units, is provided in Gypsy, through its use of "access lists". Since we believe that selective availability is an important capability it is also provided in our proposed mechanism.

A module interconnection language [DERE76, MITC79, TICH79] is a separate language, used in conjunction with a programming language, for specifying and controlling the interfaces among program units. One of these languages ([TICH79]) also provides configuration control for multiple versions of systems. The principal benefit derived from a physically separate specification of interfaces is the possibility of independent managerial control. While selective accessibility is possible in all these languages, none supports selective availability. In many cases, the use of a separate description causes a large amount of duplication. The mechanism proposed here incorporates the functionality and independence of the interface control provided by module interconnection languages without the duplication. We believe that version control is best left outside of the programming language and thus have not incorporated it into our mechanism.

In summary, while many existing programming and module interconnection languages provide some of the desired capabilities, none provides all. In the next section we describe how our mechanism, using fairly straightfoward constructs, achieves this goal.

3. Constructs of the Proposed Mechanism

There are two aspects to our proposed mechanism. First, we propose a program structure that supports information hiding, separate compilation and managerial control. Second, our proposal includes language contructs that, in conjunction with this program structure, provide precise interface control. Although our mechanism is quite simple, it results in a more uniform, complete, flexible and precise treatment of interface control than any of the approaches previously proposed.

Under our proposal a software system consists of a (nest-free) collection of program units, where a program unit is either a subprogram (procedure, function or task) or a package (i.e., an encapsulation). To support information hiding, separate compilation and managerial control, we propose having three distinct kinds of subunits associated with each program unit: an interface specification, a user specification and a body. An interface specification subunit completely describes a program unit's interface. It specifies the accessibility of external entities from within the program unit. It also specifies the availability of the program unit, and (in the case of a package) the entities comprising it, to other program units. A user specification contains the minimum amount of information necessary for use of the program unit by a particular user. A program unit's body contains the actual code section(s) implementing the unit's specification.

A major benefit of the program structuring mechanism proposed here is that the physical separation of a program unit's interface from its body fosters managerial control over both the accessibility and availability of that program unit. In cases where such control is not desired, however, implementors of program units can assume the role of project leader and construct their own interface specifications. The physical separation of user specifications from the interface specification also fosters information

hiding and allows different views of the same program unit. Furthermore, the separation of a program unit's various specifications from its body facilitates separate compilation, both of its users and of its implementation [ICHB76]. Taken together, this separation of concerns aids in the incremental development of large software systems by reducing the interdependency of the components of those systems.

The second aspect of our proposal concerns the constructs used in a program unit's interface specification to achieve precise interface control. We propose two constructs, a with clause for defining accessibility and a restrict clause for defining availability. The basic function of the with clause, which is similar to that of the Ada with clause, is to serve as an import list for program units. We have enhanced the Ada with clause in order to achieve selective accessibility. In our proposal the with clause can import not only packages and unpackaged subprograms but also specific entities (subprograms, objects and types) encapsulated within packages. Thus, if only some of the entities provided by a package are needed, then the balance of the entities provided by that package do not also have to be imported. For example

```
subprogram EXAMPLE ( ... )
    with REALS, SUB1, STACK.ST
```

indicates that the subprogram imports the entire package REALS, the unpackaged subprogram SUB1 and just the type ST from the package STACK.

As a starting point for controlling availability, our proposal includes constructs that distinguish a package's provided entities from its hidden ones. Both the provided and hidden entities are available to all other entities in the defining package, but only the provided entities are available outside of the package. Several existing languages employ similar constructs.

In those languages, however, availability is controlled on an all-or-nothing basis; either an entity is made available to every program unit, or it is made available to no program unit other than the defining package. To remedy this shortcoming, our proposal also includes a restrict clause, which has its roots in Gypsy's "access list". The restrict clause may be appended to any of a package's provided entities in order to selectively limit their availability to other program units when such control is desired. For example

```
OBJ : INTEGER
  restricted to PROGUNIT
```

indicates that object OBJ is only available to program unit PROGUNIT. For convenience, a single restrict clause may be appended onto the package itself to indicate that all the entities provided by the package are limited in the same way. The restrict clause also applies to an unpackaged subprogram. An appended restrict clause for such a subprogram limits its availablity and avoids the need to create a superfluous package to encapsulate the subprogram and control its availability.

An important aspect of information hiding is the distinction between the availability of the name of a type and the availability of the representation of that type. Hence, a type provided by a package may be associated with two restrict clauses, one referring to the availability of the name and the other referring to the availablity of the representation. Access to the name of the type is, of course, necessary for any sort of use of the type. Therefore, limitations on the availability of the representation are a subset of those on the name and so a restrict clause associated with the representation serves as a further restriction on the representation beyond that inherited from the name. Six basic levels of control result (Figure 1). Associating two restrict clauses with a type definition allows abstract data types to be

```
(1)  type A is B
          -- name:            no restriction
          -- representation:  no restriction
          --   name and representation are available to all


(2)  type A is B
       restricted to X
          -- name:            no restriction
          -- representation:  restriction
          --   name available to all; representation
          --   available only to X and defining package


(3)  type A is B
       restricted
          -- name:            no restriction
          -- representation:  complete restriction
          --   name available to all; representation
          --   available only to defining package


(4)  type A restricted to X
       is B
          -- name:            restriction
          -- representation:  same restriction as name
          --   name and representation available only
          --   to X and defining package


(5)  type A restricted to X, Y
       is B restricted to X
          -- name:            restriction
          -- representation:  restriction
          --   name available only to X, Y and defining
          --   package; representation available only to
          --   X and defining package


(6)  type A restricted to Y
       is B restricted
          -- name:            restriction
          -- representation:  complete restriction
          --   name available only to Y and defining
          --   package; representation available only to
          --   defining package
```

Basic Levels of Control
Over Provided Packaged Type Definitions

Figure 1.

easily defined and also solves the problem of sharing the representation of an abstract type among different program units [KOST76].

Although not discussed further here, our proposal also allows control to be applied to the operations associated with particular objects and types. This is similar to the access constraint mechanism proposed for objects by Jones and Liskov [JONE78].

It should be pointed out that our proposal provides little control within a program unit over the accessibility and availability of entities declared in that program unit. This lack of control, which is based on the presumption that entities are declared together because they share some logical relationship, may be viewed as a syntactic shorthand for a commonly occurring situation. If the control is desired, however, then it can be achieved through appropriate use of our proposed mechanism.

A program is said to be interface-correct if there is consistency among the interfaces of all the program's packages and subprograms. For each program unit, this specifically means that all imported entities must be available to it. An appealing property of the with and restrict clauses as defined here is that the interface-correctness of a program can be guaranteed before execution of the program.

4. Example

To illustrate our proposed mechanism, consider one possible scenario of its use. The project leader of a software project recognizes the need for a package that defines a stack abstract type (what else!). The project leader therefore creates an interface specification subunit for the stack package that describes what types and operations it provides, how the abstract type is to be represented, and to which other program units the abstract type is available (Figure 2). In particular, the stack data type ST and stack element

```
package STACK interface specification
  with LINKED_LIST.LIST_ELEMENT, LINKED_LIST.LIST

provides
  type EL is new LINKED_LIST.LIST_ELEMENT
      restricted to CONVERT

  type ST is new LINKED_LIST.LIST
      restricted to CONVERT

  subprogram POP ( S : ST; OLDTOPEL : EL )
    with LINKED_LIST.DELETE
    restricted to USERA

  subprogram PUSH ( S : ST; NEWTOPEL : EL )
    with LINKED_LIST.INSERT

  subprogram TOP ( S : ST; TOPEL : EL )
    with LINKED_LIST.EXAMINE

end package STACK
```

Interface Specification Subunit

Figure 2.

data type EL have three operations PUSH, POP and TOP associated with them. Further, the project leader has specified that the representation of the abstract type should be based on entities provided by a previously defined, although not necessarily implemented, linked-list package. Any program unit may create objects of the types ST and EL, but only program unit CONVERT shares the representation of the types with the stack package. Moreover, only program unit USERA may use operation POP.

Even though the stack operations have not yet been implemented, the project leader would like to initiate development of the program units that will use the package. The project leader therefore creates a user specification subunit for each user of the package (Figure 3). The resulting user specifications present views of the package that are relevant to the different users. Clearly, the software development environment could provide a tool to derive appropriate user specifications from an interface specification.

Independent of the development of the program units using it, the stack package is eventually implemented. As it turns out, an additional object as well as an additional subprogram, both of which should be unavailable outside of the package, are needed in the implementation. Hence the project leader updates the interface specification to include these two new entities, TOTAL_STACKS and UTIL, still retaining control over their interface even though they are part of the implementation (Figure 4). As part of the updating procedure, the new interface specification is checked for consistency with the old interface specification using a tool provided in the environment. The actual bodies of the subprograms defined in the package appear in a body subunit (Figure 5). Notice that the implementor is constrained by the interface specification to use the entities provided by the linked-list package in implementing these subprograms.

```
package STACK user specification
  type EL

  type ST

  subprogram PUSH ( S : ST; NEWTOPEL : EL )

  subprogram TOP ( S : ST; TOPEL : EL )

end package STACK
```

(a)

```
package STACK user specification
  type EL

  type ST

  subprogram POP ( S : ST; OLDTOPEL : EL )

  subprogram PUSH ( S : ST; NEWTOPEL : EL )

  subprogram TOP ( S : ST; TOPEL : EL )

end package STACK
```

(b)

```
package STACK user specification
  type EL is new LINKED_LIST.LIST_ELEMENT

  type ST is new LINKED_LIST.LIST

  subprogram PUSH ( S : ST; NEWTOPEL : EL )

  subprogram TOP ( S : ST; TOPEL : EL )

end package STACK
```

(c)

User Specification Subunits for General Users (a),
Program Unit USERA (b), and Program Unit CONVERT (c)

Figure 3.

```
package STACK interface specification
  with LINKED_LIST.LIST_ELEMENT, LINKED_LIST.LIST,
       LINKED_LIST.LIST_COUNTER

provides
  type EL is new LINKED_LIST.LIST_ELEMENT
      restricted to CONVERT

  type ST is new LINKED_LIST.LIST
      restricted to CONVERT

  subprogram POP ( S : ST; OLDTOPEL : EL )
    with LINKED_LIST.DELETE
    restricted to USERA

  subprogram PUSH ( S : ST; NEWTOPEL : EL )
    with LINKED_LIST.INSERT

  subprogram TOP ( S : ST; TOPEL : EL )
    with LINKED_LIST.EXAMINE

hides
  TOTAL_STACKS : LINKED_LIST.COUNTER := 0

  subprogram UTIL ( E : EL )
    with LINKED_LIST.STATISTICS

end package STACK
```

Updated Interface Specification Subunit

Figure 4.

```
package STACK body
  subprogram UTIL ( ... ) is ... end subprogram UTIL

  subprogram POP ( ... ) is ... end subprogram POP

  subprogram PUSH ( ... ) is ... end subprogram PUSH

  subprogram TOP ( ... ) is ... end subprogram TOP

end package STACK
```

Body Subunit

Figure 5.

A number of other scenarios exist that exploit our interface control and program structuring mechanism in interesting ways. For instance, a methodology could have been employed that would have called for the definition of the various user specifications before the definition of the interface specification. A tool could conceivably be used to synthesize the interface specification from these user specifications and then check its consistency.

5. Future Directions

Several facets of the mechanism proposed here are presently being studied. In particular, we are currently developing a formal model of visibility, investigating the types of analytic techniques and tools that should be provided to support the proposed mechanism, and examining the usefulness of this mechanism during the pre-implementation phases of the software development process.

The formal model of visibility that we are developing describes entity accessibility and availability in a general, comprehensive manner. This model will be used to describe the interface control provided by existing mechanisms as well as by our proposed mechanism. We expect to use this model to rigorously compare and evaluate these mechanisms. Moreover, this formalism will be used to help define analytic techniques that can be applied to systems using our proposed mechanism.

As alluded to throughout this paper, there are a number of tools that should be developed to support interface control. Some of these tools are quite straightforward to develop, such as a tool to derive a user specification from an interface specification. Other tools, such as tools to determine interface consistency or to determine what recompilation is necessary after an interface specification has been modified, require additional investigation.

Finally, the application of these concepts to the pre-implementation phases of software development is being explored. The mechanism has been designed to support incremental development. The major system program units and their interfaces could be described in a functional specification using the proposed mechanism. During the design phase, the program units and their interfaces could be further refined. We expect that many of the tools mentioned above could be applied to high level, partial solutions, thus providing precise, consistent interface control early and throughout the software development process.

References

[AMBL77] A.L. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch, and R.E. Wells, "GYPSY: A Language for Specification and Implementation of Verifiable Programs," Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Noticies, vol. 12, no. 3, pp.1-10, March 1977.

[BUXT80] J.N. Buxton, Requirements for Ada Program Support Environments ("Stoneman"), United States Department of Defense, Washington, D.C., February 1980.

[CASH81] P.M. Cashin, M.L. Joliat, R.F. Kamel, and D.M. Lasker, "Experience with a Modular Typed Language: Protel," Proceedings of the Fifth International Conference on Software Engineering, San Diego, California, pp.136-143, March 1981.

[CLAR80] L.A. Clarke, J.C. Wileden and A.L. Wolf, "Nesting in Ada Programs is for the Birds," Proceedings of an ACM-SIGPLAN Symposium on the Ada Progamming Language, SIGPLAN Notices, vol. 15, no. 11, pp.139-145, November 1980.

[DERE76] F. DeRemer and H.H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Transactions on Software Engineering, SE-2, no. 2., pp.80-86, June 1976.

[DOD82] Reference Manual for the Ada Programming Language, United States Department of Defense, Washington, D.C., July 1982.

[HANS81] D.R. Hanson, "Is Block Structure Necessary?," Software - Practice and Experience, vol. 11, no. 8, pp.853-866, August 1981.

[ICHB76] J.D. Ichbiah and G. Ferran, "Separate Definition and Compilation in LIS and its Implementation," Lecture Notes in Computer Science, no. 54, Springer-Verlag, Berlin, pp.288-297, 1977.

[JONE78] A.K. Jones and B.H. Liskov, "A Language Extension for Expressing Constraints on Data Access," CACM, vol. 21, no. 5, pp.358-367, May 1978.

[KOST76] C.H.A. Koster, "Visibility and Types," Proceedings of a Conference on Data, SIGPLAN Notices, vol 11, no. 2, pp.179-190, February 1976.

[LISK79] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Schiefler, and A. Snyder, "CLU Reference Manual," Technical Report TR-225, MIT Laboratory for Computer Science, October 1979.

[MEDI81] R. Medina-Mora and P.H. Feiler, "An Incremental Programming Environment," IEEE Transactions on Software Engineering, vol. SE-7, no. 5, pp.472-482, September 1981.

[MITC79] J.G. Mitchell, W. Maybury and R. Sweet, "Mesa Language Manual Version 5.0," Technical Report CSL-79-3, Xerox PARC, Palo Alto, California, April 1979.

[POPE77] G.J. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell, and R.L. London, "Notes on the Design of Euclid," Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, vol. 12, no. 3, pp.11-18, March 1977.

[SHAW81] M. Shaw (ed.), ALPHARD: Form and Content, Springer-Verlag, New York, 1981.

[TEIT81] T.Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," CACM, vol. 24, no. 9, pp.563-573, September 1981.

[TICH79] W.F. Tichy, "Software Development Control Based on Module Interconnection," Proceedings of the Fourth International Conference on Software Engineering, Munich, West Germany, pp.29-41, September 1979.

[WIRT77] N. Wirth, "Modula: A Language for Modular Multiprogramming," Software - Practice and Experience, vol. 7, no. 1, pp.3-35, January-February 1977.