Precise Interface Control: System Structure, Language Constructs, and Support Environment

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

COINS Technical Report 83-26
August 1983

Working Paper

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

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

ABSTRACT

Although the importance of *interface control* in the production and maintenance of large, complex software systems has been repeatedly recognized, there has been no systematic treatment of interface control and its role in the software development process. Existing languages provide incomplete and imprecise mechanisms for indicating which entities in a software system can access other entities and how they are allowed to manipulate those entities. Moreover, these weaknesses undermine the effectiveness of any *analysis* that can be done to assure consistency among interfaces and proper use of shared entities.

This paper presents an approach for achieving precise interface control which has three interrelated aspects. First, it employs a system structure that, in addition to facilitating interface control, fosters information hiding, managerial control, separate compilation and incremental analysis. Second, it provides a small set of language constructs that, in conjunction with this system structure, provide a framework for a consistent set of abstractions for representing module decomposition and entity interaction throughout the software development process. Finally, the approach relies upon a support environment that includes an integrated set of tools for constructing, viewing, manipulating, and analyzing the interface control aspects of a software system during all phases of its development. A realistic example demonstrating use of the approach during design of a software system is presented.

1. Introduction

The description of the entities contained in a software system along with the relationships among those entities is of primary importance during every phase of the software lifecycle. For example, generating such a description is one of the first activities undertaken during the early phases of software development, while insuring that entity relationships remain correct and consistent is a major consideration during maintenance. In most software languages, *entities* are those language elements that are given names, such as procedures, functions, tasks, data objects and data types. Unfortunately, software languages, including the recently introduced Ada® programming language [DOD83] and its PDL/Ada design language derivatives [KERN83], continue to be deficient in their ability to describe the relationships among the entities contained in a software system. In particular, the mechanisms provided in existing languages for indicating which entities can access other entities and how they are allowed to manipulate those entities are generally incomplete and imprecise. As software systems become larger and more complex, these *interface control* considerations become increasingly important.

In this paper, we present an approach to achieving precise interface control that is based upon a hierarchical, but nest-free, system structure, a small number of language constructs, and an integrated set of tools in a support environment. In addition to facilitating interface control, the system structure is designed to foster information hiding, managerial control, separate compilation and incremental analysis. The entire approach has been carefully tailored to support incremental development and to be applicable across the phases in the software lifecycle. Specifically, the system structure and language constructs provide a uniform framework for representing module decomposition and entity interaction throughout the software development process. The support environment enhances this capability by providing an integrated set of

®Ada is a registered trademark of the US Government (Ada Joint Program Office).

tools for constructing, viewing, manipulating, and analyzing the interface control aspects of a software system. As a result, the system structure, language constructs, and associated tools provide a genuinely integrated approach to interface control applicable throughout the software development process.

In the next section we present the background and motivation for this approach to interface control. Section 3 describes the system structure and language constructs. Section 4 describes the support environment, with particular emphasis on the analysis required to incrementally maintain interface consistency during the design, coding, and maintenance of a software system. A realistic example illustrating use of the approach during design of a software system appears in Section 5. The final section reports on the status of a prototype implementation of these ideas and comments on directions for future research.

2. Background and Motivation

Traditionally, interface control has been associated with *visibility control*, where visibility is usually described in terms of *declaration*, *scope*, and *binding*. A declaration introduces an entity and associates an identifier (name) with that entity. The scope of a declaration is the region of program text over which that declaration is in effect. Many languages allow a single identifier to be associated with more than one declaration and the scopes of those declarations to overlap. Binding relates the use of an identifier, at a given point in a program, to a particular declaration. A description of a visibility control mechanism is essentially a description of how that mechanism controls scope. *Interface control* is that facet of visibility control that deals with the visibility relationships among the units of a system, where a unit is, for example, a subprogram (procedure, function, or task) or an encapsulation (package, module, or common block).

The traditional approach to visibility control is exemplified by scope rules based on nesting, the predominant visibility control mechanism in modern programming languages. In nested languages, from Algol60 to Ada, visibility depends upon the location of entities within a program's nested structure. This approach is inadequate for precisely describing the wide range of possible interface relationships among the entities contained in a software system since it usually results in weaker interface control than desired. For example, a subprogram's "local" entities are unavoidably visible to other subprograms nested within that subprogram. Furthermore, nesting forces a software system to take the form of a single, monolithic unit.[1] For these reasons, it has been argued that nesting is inadequate for achieving some desirable software properties, such as information hiding, and is antithetical to others, such as readability, incremental development and separate compilation [WULF73, CLAR80, HANS81]. Thus, particularly for large, complex software systems, more versatile and powerful mechanisms for interface control are required.

A more general view of interface control that is the basis for the approach presented here arises from an important distinction between two aspects of scope: *accessibility* and *provision*. An entity's accessibility refers to the entities that can be (potentially) accessed by that entity. In general, the accessible entities for a unit include the unit itself and any locally declared entities, as well as any non-local entities imported (implicitly or explicitly) into that unit. An entity's provision, on the other hand, refers to those entities that it makes available for access by other entities in a software system. The entities provided by a unit may include the unit itself and any locally declared entities exported (implicitly or explicitly) from that unit.

[1] Numerous attempts have been made to alleviate this problem while still retaining a nested structure. The Ada *subunit* facility, for example, permits the textual separation of a nested unit from its parent in order to allow the nested unit to be separately compiled. The fact remains, however, that for purposes of determining and understanding the visibility context of the textually separate unit, the unit is still considered to be logically nested; i.e., it is treated as being located at some particular point in the monolithic tree structure of the software system.

Thus, accessibility and provision present two different, yet complementary, points of view on visibility control—accessibility describing what can be (potentially) used by some unit and provision describing what is made available by some unit to other units in a software system.

A variety of languages, such as Ada [DOD83], Alphard [SHAW81], CLU [LISK79], Euclid [POPE77], Gypsy [AMBL77a], Mesa [MITC79], Modula [WIRT77], and Protel [CASH81], have attempted to compensate for the inadequacies of nesting by offering alternative mechanisms for interface control. These languages have relied, to a greater or lesser degree, on the concepts of *encapsulation* and *explicit import/export control* to describe which entities are accessible and provided in a unit of a software system. In its most general form, which is not exactly the way it is used in all of these languages, an encapsulation serves to group related units, objects, and types. Examples of encapsulation constructs include the Ada *package*, Alphard *form*, CLU *cluster*, and Modula *module*. Explicit import/export control provides the means by which external entities are made accessible from within a unit and, in the case of encapsulations, the means by which internally defined entities are provided outside the unit. Many languages incorporating these concepts retain nesting and use import/export to further constrain the visibility resulting from a nested structure.[2] Not one of the languages mentioned above, however, supports the precise description of both what an entity accesses and what an entity provides. The approach advocated in this paper shuns nesting altogether and builds upon existing encapsulation and import/export concepts to create a mechanism capable of describing the desired accessibility and provision of entities in a program more precisely and flexibly than is possible with the mechanisms provided in any one of these previous languages.

[2] 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]).

The system structure and language constructs embodying this approach are quite simple. Without appropriate automated support, however, development of the proper interface relationships for large software systems remains a complex task. This task could be greatly facilitated by the application of processing tools and sophisticated analysis techniques to assure the consistency of the software system. Therefore, an important aspect of the approach is the inclusion of an integrated set of tools constituting part of a software development environment. These tools would provide an integrated set of facilities applicable throughout the software development process and would support *analysis* of a software system at any point during its development, even when the description of the system is only partially complete. In particular, the system structure and language constructs embodying the approach are suitable for describing the modular decomposition and entity relationships of a software system from the earliest points in design through maintenance. The associated set of tools can be applied to descriptions at all of these stages to provide analysis information about a particular class of properties of the software system being developed.

3. System Structure and Language Constructs

The approach to interface control that we are advocating is based upon the adoption of a general hierarchical system structure that is not restricted to a tree-like format. Unlike nesting, this approach makes no attempt to capture the (two-dimensional) hierarchical structure of a system by the (one-dimensional) textual location of the units. Instead, a software system created using our approach consists of a nest-free collection of units where the hierarchical relationships are explicitly declared.

A *unit* in the approach described here is either a subprogram (procedure, function or task) or a package (i.e., an encapsulation). To support information hiding, managerial control, separate compilation and incremental analysis, there are three distinct kinds of *subunits* that may be associated with each unit: an *interface specification*, a *body*, and an *interface stub*. In describing each subunit and the benefits of the resulting system structure, we will employ an Ada-like notation. This notation is merely a vehicle for conveying our ideas, however, and could easily be modified to demonstrate how these ideas could be incorporated into many other languages.

An interface specification subunit completely describes a unit's interface. It specifies both the accessibility of external entities from within a unit and the provision of entities by that unit to other units. In Ada, an import statement, called the *with clause*, is used to control a (non-nested) unit's accessibility. A *with clause* can import the names of packages and unpackaged subprograms. Importation of a package affords access to all the package's provided entities, while importation of an unpackaged subprogram permits invocation of that subprogram. Our notation uses an Ada-like *with clause* in the interface specification subunits to specify importation. In our notation, however, *with clauses* can be attached to packaged entities as well as to packages themselves, in order to provide more flexible control over accessibility. When attached to the package itself, a *with clause* governs accessibility for all the entities of the package, while *with clauses* attached to individual packaged entities control the accessibility for just the individual entities. Further flexibility is achieved in our notation through *selective importation*, the ability to import arbitrary subsets of the entities provided by an encapsulation. This capability, although not available in Ada, has been provided in Modula, Euclid, and Mesa. To achieve selective importation, we have enhanced the Ada *with clause* by permitting individual packaged entities (subprograms, objects, and types) as well as entire packages and unpackaged subprograms to be specified. Therefore, 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 made accessible. In the example in Figure 1a, procedure Sub1 imports the entire

```
package Pac1 interface
    type Typ1 is ...

    Obj1 : Integer
        provided to Pac3.Sub4

    procedure Sub1 ( ... )
        with Pac2, SubA, Pac3.Obj2

    procedure Sub2 ( ... )
        with Pac2
        provided to Pac3.Sub4

private
    procedure Sub3 ( ... )
        with Pac2.Sub1

end Pac1
```

(a)

```
package Pac1 body

   procedure Sub1 ( ... )
       ...
   end Sub1

   procedure Sub2 ( ... )
       ...
   end Sub2

   procedure Sub3 ( ... )
       ...
   end Sub3

end Pac1
```

(b)

Figure 1: Example Interface Specification Subunit (a) and Body Subunit (b).

package Pac2, the unpackaged subprogram SubA, and just the object Obj2 from the package Pac3. Notice that the interface specification subunit of a package may contain the interface specifications of encapsulated subprograms. (The keyword *interface* is dropped from a packaged subprogram's interface specification for brevity.)

The interface specification subunit also defines what is provided by a unit to other units in a system.[3] In Ada, provision of packaged entities is controlled through constructs that textually separate a package's provided entities from its hidden entities. 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. In the approach advocated here, the provided entities reside in a section of the interface specification subunit referred to as the *provides part* (and in Ada as the *visible part*) while the hidden entities reside in a section referred to as the *private part* (as in Ada). In Ada, however, provision is controlled on an all-or-nothing basis; either an entity is provided to every unit, or it is provided to no unit, and so is hidden. While these two extremes are useful (for instance, in describing the global provision of a library unit such as a package of trigonometric functions or the hiding of a low-level utility subprogram within the package needed to implement the trigonometric functions), the intended provision of a particular entity often lies somewhere in between [MINS83]. Therefore, our approach extends Ada by including a *provides clause*, which has its roots in Gypsy's *access list* [AMBL77a]. The *provides clause* may be appended to any of a package's provided entities in order to selectively limit their provision to external units. The absence of a *provides clause* on a provided entity is interpretted to mean that the entity is provided to "all." The *provides clause* can also be applied to an unpackaged subprogram. An appended *provides clause* for such

[3] Unlike the visibility rules of a nested structure, the system structure presented here does not permit subprograms to provide their internally defined entities to other units. In fact, subprograms can provide nothing but themselves; packages are the only units that can provide their internally defined entities. As a result, entities that are shared among subprograms would not be (artifically) placed within one of those subprograms.

a subprogram limits its provision to other units and avoids the need to create a superfluous package to encapsulate the subprogram and control its availability. In the package in Figure 1a, object Obj1 and subprogram Sub2 are only provided to subprogram Sub4 in package Pac3, while type Typ1 and subprogram Sub1 are provided to all units in the system. Finally, procedure Sub3 is not provided to any external unit since it appears in the *private part* of package Pac1.

An important aspect of the approach we are endorsing is that it can be used to distinguish between the provision of the name of a type and the provision of the representation of that type. Hence, in this approach, as in Gypsy, a provided type may be associated with two *provides clauses*, one referring to the provision of the name and the other referring to the provision of the representation. Access to the name of the type is, of course, necessary for any sort of use of the type. Therefore, a *provides 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 2), permitting a high degree of flexibility in controlling the use of a type definition. By contrast, Ada only provides a few of these levels. Associating two *provides clauses* with a type definition allows abstract data types to be easily defined and also solves the problem of sharing the representation of an abstract type among different units [KOST76].

A unit's body subunit contains the actual code sections implementing the unit's interface specification. Figure 1b shows the skeleton of the body associated with the interface specification subunit given in 1a. Notice that a package's body subunit contains the bodies associated with the interface specifications of subprograms mentioned in the package's interface specification. (The keyword *body* is dropped from a packaged subprogram's body for brevity.) When our approach to interface control is used in the pre-implementation phases of the development process, the body would contain a high-level description of its procedures. In the

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


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


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


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


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


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

Figure 2: Basic Levels of Control Over Provided Packaged Type Definitions.

example presented in Section 5, a design is presented and the body of procedure InterfaceCheck is given in a PDL (Figure 6). Both there and in the examples of Figure 1 we make liberal use of the *incompleteness construct*, which is denoted using an ellipsis. This construct is useful in all phases of development prior to final implementation for explicitly indicating where details that will be supplied later have been omitted from a description.

Interacting components of large software systems are often developed independently —perhaps even at different times. When a group of subunits desires access to entities from a unit for which no interface specification subunit is yet available, an interface stub subunit can be provided. An interface stub usually only contains some of the information described in the interface specification. In particular, an interface stub of a unit need not contain any information about what that unit is accessing but only needs to describe what is being provided by the unit to the subunits in the accessing group. A number of different interface stubs of a unit may exist to accomodate the development of a number of different groups. The interface-stub mechanism provides a means for the various groups of users of a unit to document their particular intended uses, or *views*, of that unit before the unit is available. When a unit's interface specification is available (in a library) or completely known, then it could be used for processing instead of the interface stub. As described in the next section, the environment provides tools to assure consistency among the interface stubs, to generate an accumulated view, and to check that the interface specification, when submitted, is consistent with any existing interface stubs of that unit. Use of interface stubs is illustrated in the example presented in Section 5.

It should be pointed out that the system structure and language constructs discussed above provide little control within a unit over the accessibility and provision of entities declared in that unit. This lack of control, which is based on the presumption that entities are declared together because they are strongly interrelated, may be viewed as a notational shorthand for a commonly occurring situation. If more control is desired, then it can be achieved through the creation of additional units to hold the appropriate entities.

There are a number of benefits associated with the system structure that results from the approach to interface control outlined above. Since what is accessible and provided is explicitly and clearly stated, the resulting software is more readable than it would be using a nested structure. Nested structures also have the disadvantage that units must often be physically moved within the text of a program or design to accomodate changes to accessibility and provision. Since the approach presented here results in a linear, order-independent collection of units, no such movement is required. We contend that such a system is easier to change and therefore easier to develop and maintain. Moreover, the structure facilitates managerial control, separate compilation and incremental analysis, and information hiding. The remainder of this section elaborates on these points.

The system structure associated with this approach requires that a unit's interface control information be completely separate from its body. Such a separation results in a structure that is similar to a *module interconnection language* or MIL [DERE76, MITC79, TICH79]. In fact, it becomes only a conceptual issue whether interfaces are viewed as being specified in a separate language or not. A major benefit of a MIL, and thus of the approach described here as well, is that *managerial control* over both the accessibility and provision of a unit can be supported.[4] To achieve such control, a project leader would be the only person permitted to create or modify an interface specification or interface stub subunit. While in languages such as Ada, Mesa, and Protel there is support for "specifications" of units separate from their bodies, these specifications do not completely define the interfaces to the units. In Ada, for instance, a body

[4] Tichy's MIL [TICH79] also supports *version control*. While this important capability is not addressed here, our approach's system structure is certainly amenable to inclusion of such a feature.

may itself import entities. In our approach the *with clauses*, which can only appear in an interface specification subunit, completely constrain the external entities accessible from a unit's body. Similarly, the provision of a unit is completely defined in the interface specification subunit. Thus, by giving only the project leader the ability to create and modify the interface specification subunit, a method of enforcing interface control is obtained. Of course, in cases where such managerial control is not desired, implementors of units can assume the role of project leader and construct their own interface specifications.

The separation of a unit's interface specification from its body facilitates *separate compilation* [ICHB76] and *incremental analysis*, while minimizing the need for recompilation and reanalysis. For instance, an interface specification or interface stub subunit may be developed first and entered into a system library. Other units that access the provided entities of that unit may be compiled (or analyzed) using the library description, which is independent of any particular implementation of that unit. Later, if the body changes but the interface does not, these units do not have to be recompiled (reanalyzed). Of course, a unit may not actually make use of all the entities to which it is given access. The support environment provides a tool to detect such a situation in order to avoid needless recompilation (reanalysis) of a unit's subunits when a change is made to an accessible entity that is in fact not accessed by that unit. Finally, the separation of a unit's interface specification and body also allows the provision of a unit's entities, as specified in the interface specification subunit, to be changed without the need to recompile (reanalyze) the body subunit, although subunits accessing the entities may require recompilation (reanalysis). The incremental analysis capability supported by this approach is particularly important since it permits interface consistency analysis to be performed early and as often as desired during the software lifecycle.

Another beneficial feature of this system structure is that the textual separation of concerns fosters *information hiding*. As noted above, the system structure separates a unit's interface from its body. Different implementations can be given without changing the interface to a unit or the units that access that unit. Moreover, the environment can provide tailored views of a unit, much like schemas in databases. A software developer working on a unit that will access entities from another unit only needs to see the specifications of the provided entities of the accessed unit and each such specification only needs to contain information relevant to the accessing unit. The *provides clauses* in an interface specification actually define the different views particular units have of a package's provided entities.

In sum, while many existing specification, design, programming, and module interconnection languages provide some of the desired capabilities, none provides all. The preliminary framework outlined here incorporates features distilled from many of these previous attempts. The resulting language framework is relatively simple and straightforward, yet surpasses these previous attempts by providing precise interface control as well as the comprehensive collection of benefits outlined above. Moreover, this framework provides a uniform approach to interface control that can be exploited throughout the software development and maintenance process.

4. Support Environment

The system structure and language constructs described above are most useful when accompanied by an automated support environment. Such an environment should help a software developer to create, store, modify, and analyze information about a system's structure and interfaces. Manipulation and analysis of these aspects of a system could be carried out at any stage in the development of a software system using the environment's tools.

We are currently developing a prototype environment that will comprise a set of tools applicable throughout the software development process. The capabilities of this prototype will be invoked using an extensible command language and associated command language interpreter, modelled on those provided in Toolpack [OSTE82]. Through this command language, users of the environment will have access to three classes of tools: *library management tools*, *processing tools*, and *analysis tools*. In the remainder of this section we briefly describe these tools.

Library Management Tools. Along with the submitted subunits themselves, the system library holds information about those subunits, such as completeness and interface-consistency status. These tools maintain the integrity of this information. Since the major emphasis of our environment is on controlling interface interaction among units, the library management tools provide protection mechanisms to control modification of the interface specification and interface stub subunits, preventing arbitrary changes by unauthorized users. In addition, these tools support various management disciplines regarding the analyses performed on subunits, to assure that checks are done thoroughly and in appropriate sequences. Finally, these tools support the use of multiple libraries and of sublibraries within a library.

Processing Tools. These tools are used to create new subunits, modify existing subunits, and produce reports regarding the current status of the software development project. They include an editor, interface and view generators, update processor, translator, and report generator. The editor is a typical syntax-directed editor that recognizes the interface control and other constructs of the design and programming languages. The update processor is used to replace one subunit with another. In so doing, it performs an update analysis (described below) and modifies the status indicators of any subunits in the library that are (potentially) affected by the replacement. The translator is a preprocessor that turns subunits containing the constructs used to achieve precise interface control into the target language (which in our prototype is Ada). The report generator produces reports on the current status of the software development project, as reflected in the current status of the library, and on the results of the various analyses that have been performed on the library's contents.

Perhaps the most interesting of the processing tools are the interface and view generators. Given a set of interface stubs of a unit, the interface generator will generate a single, accumulated interface stub of that unit. Since a collection of interface stubs of a unit is not likely to specify a consistent interface for the unit, this generator also reports any inconsistencies that exist among the interface stubs. Note that the generator can also produce a template for the interface specification subunit that is a minimal, and possibly incomplete, representation of the interface. Conversely, given an interface specification of a unit, the user view generator will generate consistent user views of that unit as it would appear to other (user) units. The initial version of our environment will produce these user views in the form of interface stubs. Later versions will produce other forms for these descriptions, including graphical representations of the descriptions.

Analysis Tools. These tools perform various kinds of analysis on individual subunits and on the relationships among subunits. The analysis tools can be invoked through the command language or, in some cases, by a processing tool. The results of these analyses may be stored as new library information, attached either to a particular library or to the subunit itself, depending on the type of analysis. Results may also go directly to the report generator or to another processing tool, such as the update processor. The types of analysis that can be performed include local syntax and semantic checking, visibility and semantic analysis, subunit interaction checking, and update analysis.

Local syntax and semantic checking constitutes the analysis that can be performed on an individual subunit without reference to any other subunits. Correct syntax is checked for, as much semantic consistency of the subunit as possible is ascertained and incompleteness of the subunit, as indicated by the appearance of PDL constructs such as ellipses, is noted.

Visibility and semantic analysis tools check the consistency of interrelated subunits. One set of these tools compares two descriptions of the same unit, such as an interface specification and the corresponding body, or an interface specification and a stub for that interface specification. Others of these tools compare subunits corresponding to different but related units. Instances of these include comparisons of the interface specifications of two units that reference one another or comparison of the body of one unit with the interface specification of another unit that it references. In all cases, the analysis is intended to reveal whether units and the entities that they contain are being used in the appropriate manner. The use of these visibility and semantic analysis tools is illustrated in the example presented in the next section.

Subunit interaction checking determines what kinds of interactions a given subunit has with other subunits in a particular library. The analysis can be performed with respect to the entities that a subunit provides or with respect to entities that it accesses. The analysis determines whether particular entities are available or not and whether those entities that are available are actually referenced.

Update analysis determines what impact the proposed replacement of one subunit by another will have on other subunits in the library. In particular, it is possible to discover whether a subunit that references entities provided by the replaced subunit must be rechecked or revised. Correspondingly, it is possible to discover whether a subunit that provides entities used by the replaced subunit must be rechecked or revised.

5. Example

To illustrate the capabilities provided by the system structure, language constructs, and support environment described above, this section presents a simple, yet realistic, example showing the specification and analysis of an evolving system's units during the high-level and low-level design phases. The example is drawn from the development of the software for our prototype implementation of the support environment.

In this example, two units are being designed: a package LowLevelAnalysisTools, providing the low-level visibility and inter-subunit semantic analysis tools, and a package ProcessingTools, providing the general subunit processing tools. Both sets of tools are to operate on subunits through an abstract internal representation (attributed trees) realized in a third package, InternalRepresentation. For the purposes of this example, it is assumed that package InternalRepresentation is undergoing parallel development at a separate site and has not yet been delivered. (This is in fact the situation encountered in the development of compilers for Ada: Tartan Laboratories developed Diana [EVAN83], the internal representation for Ada programs, at the same time that compilers using Diana were being built at Intermetrics [TAFT82] and Softech [BABI82].)

To allow development of the two tool packages to proceed while still gaining a certain degree of confidence in the interface consistency of the system, a stub is created for the interface of package InternalRepresentation (Figure 3). The interface stub indicates (a subset of) the entities the package is expected to provide. In particular, it defines a data type Tree for representing entities and a function MakeTree for initializing such representations. The remaining entities defined in the stub are used to handle the attributes associated with entity representations. Type AttributeKind is an enumeration of the different kinds of attributes that can be used to describe entities, while type Attribute is the definition for a variant structure

```
package InternalRepresentation interface stub
  used by LowLevelAnalysisTools, ProcessingTools

  -- Tree manipulation entities

  type Tree
     is ... private

  function MakeTree ( ... ) return Tree
     provided to ProcessingTools

  ...

  -- Attribute manipulation entities

  type AttributeKind                          -- enumeration of attribute kinds
     is ( ArrayDeclaration, RecordDeclaration, ...,
          IfStatement, CaseStatement, ...
          ImportedEntities, ... )

  type Attribute ( AK : AttributeKind )       -- attribute value representations
     is record
          case AK is
            when ArrayDeclaration  => ...
            when RecordDeclaration => ...
            ...
            when IfStatement       => ...
            when CaseStatement     => ...
            ...
            when ImportedEntities  => ...
            ...
          end case
        end record

  procedure AddAttribute ( T : in out Tree; A : in Attribute )
     provided to ProcessingTools

  function GetAttribute ( T : Tree; WhichAttribute : AttributeKind )
                                                            return Attribute

  ...

end InternalRepresentation
```

Figure 3: Interface Stub Subunit of Internal Representation Package.

representing actual attribute values; the structure of an object of the latter type is discriminated by a value of the former. Finally, subprograms AddAttribute and GetAttribute are used to store an attribute value and retrieve an attribute value, respectively. Notice that specifications of the entities are given at various levels of detail. For instance, the descriptions of parameters to function MakeTree and the elements of type AttributeKind are deferred through the use of the *incompleteness construct* (ellipsis), while the parameters to subprograms AddAttribute and GetAttribute are fully described. Notice also that although the implementation of type Tree is not yet known, the presence of the keyword *private* following the (incomplete) definition indicates that users will not be able to operate on Tree's representation. Moreover, the fact that only entities in package ProcessingTools can invoke the subprograms that create or update objects of type Tree is specified by restricting the relevant subprograms to that package.

The first subunit to be submitted for checking with the interface stub of package InternalRepresentation is the interface specification subunit of package LowLevelAnalysisTools (Figure 4). Appearing in this subunit are interface specifications for procedures realizing six of the basic analyses underlying the analysis capabilities described in Section 4. The three functions EntitiesOf, Unavailable, and SemanticConflict, are utility subprograms employed by the low-level analysis tools and hidden within the package. At the top of the package is a common *with clause* that imports a number of entities from package InternalRepresentation. The list of imported entities and the parameter lists for the six procedures are only partially specified as indicated by the ellipses. Invoking the environment's visibility-analysis tools at this point reveals that package LowLevelAnalysisTools attempts to import an entity that is not available. Specifically, the common *with clause* contains the entity AddAttribute defined in package InternalRepresentation which has been restricted to package ProcessingTools (see Figure 3).

```
package LowLevelAnalysisTools interface
    with
      InternalRepresentation.( Tree, AttributeKind, Attribute, GetAttribute,
                                           AddAttribute, ... )

  procedure InterfaceCheck ( InterfaceSubunit1, InterfaceSubunit2 :
                                                  in  InternalRepresentation.Tree;
                                                          ... )

  procedure IntraUnitBodyCheck ( BodySubunit, InterfaceSubunit :
                                                  in  InternalRepresentation.Tree;
                                                          ... )

 -procedure InterUnitBodyCheck ( BodySubunit, InterfaceSubunit :
                                                  in  InternalRepresentation.Tree;
                                                          ... )

  procedure WeakInterfaceCheck ( InterfaceSubunit, InterfaceStub :
                                                  in  InternalRepresentation.Tree;
                                                          ... )

  procedure WeakInterUnitBodyCheck ( BodySubunit, InterfaceStub :
                                                  in  InternalRepresentation.Tree;
                                                          ... )

  procedure StubConsistencyCheck ( InterfaceStub1, InterfaceStub2 :
                                                  in  InternalRepresentation.Tree;
                                                          ... )

private
  function EntitiesOf ( Subunit : InternalRepresentation.Tree ) return ...

  function Unavailable ( ImportedEntity, Entity :
                                           InternalRepresentation.Tree ) return Boolean

  function SemanticConflict ( ImportedEntity, Entity :
                                           InternalRepresentation.Tree ) return Boolean

  ...    -- Other hidden utility entities

end LowLevelAnalysisTools
```

Figure 4: Interface Specification Subunit of Low Level Analysis Tools Package.

Assuming the error lies with the interface of package LowLevelAnalysisTools, the inconsistency can be rectified with appropriate editing of the interface specification subunit and invocation of the update processor to recheck and replace the subunit.

The next subunit submitted is the interface specification for package ProcessingTools (Figure 5). The subprograms defined in this subunit perform most of the basic processing functions described in Section 4. Note that in addition to the entities imported from package InternalRepresentation by the common *with clause* at the top of the package, certain other entities defined in package InternalRepresentation that are used to create and update internal representations are imported by a few of the packaged subprograms. The effect is to limit, within package ProcessingTools, those subprograms that can alter an internal representation. In particular, only subprograms Recognize, Edit, and GenerateInterface can perform such operations. Invocation of the visibility-analysis tools at this stage of development would not reveal any inconsistencies between the interface specification of ProcessingTools and the interface stub of InternalRepresentation.

Independent of the development of the rest of the system, low-level design of the body subunit of package LowLevelAnalysisTools can begin. Figure 6 shows this subunit at a stage in which the basic algorithm of procedure InterfaceCheck has been specified using PDL constructs. This algorithm involves checking, for each entity E defined in the first interface specification subunit, whether the entities defined in the second interface specification subunit referenced by E are both provided to E and accessed by E in semantically consistent ways.

With the corresponding interface specification subunit of the package and the interface stub subunit of package InternalRepresentation already present, a substantial amount of consistency checking can be performed on the body subunit of package LowLevelAnalysisTools even at this early stage. Invoking the visibility-analysis tools to analyze the consistency between

```
package ProcessingTools interface
    with
      InternalRepresentation.( Tree, AttributeKind, Attribute, GetAttribute, ... )

  procedure Recognize ( SourceCode : in   ...;
                                  Subunit : out InternalRepresentation.Tree;
                                                      ... )

     with InternalRepresentation.( MakeTree, AddAttribute,
                                              ...  -- Other tree alteration entities
                                                      )


  procedure Edit

     with InternalRepresentation.( MakeTree, AddAttribute,
                                              ...  -- Other tree alteration entities
                                                      )


  procedure Translate ( Subunit : in  InternalRepresentation.Tree; TargetCode : out ... )


  procedure ProcessUpdate ( OldSubunit : in  InternalRepresentation.Tree;
                                      NewSubunit : in  InternalRepresentation.Tree;
                                                      ... )


  procedure GenerateInterface ( InterfaceStubs : in  InternalRepresentation....;
                                          InterfaceSubunit : out InternalRepresentation.Tree;
                                                      ... )

     with InternalRepresentation.( MakeTree, AddAttribute,
                                              ...  -- Other tree alteration entities
                                                      )


  function GenerateView ( OfUnit : InternalRepresentation.Tree;
                                    ForUnitName : InternalRepresentation.Attribute ) return ...

end ProcessingTools
```

Figure 5: Interface Specification Subunit of Processing Tools Package.

```
package LowLevelAnalysisTools body

  function EntitiesOf ( Subunit : InternalRepresentation.Tree ) return ...
      begin ... end EntitiesOf

  function Unavailable ( ImportedEntity, Entity :
                                  InternalRepresentation.Tree ) return Boolean
      begin ... end Unavailable

  function SemanticConflict ( ImportedEntity, Entity :
                                        InternalRepresentation.Tree ) return Boolean
      begin ... end SemanticConflict

  ...    -- Other utility entities (e.g., RecordInterfaceCheckError)

  procedure InterfaceCheck ( InterfaceSubunit1, InterfaceSubunit2 :
                                          in  InternalRepresentation.Tree; ... )

      Entity            : InternalRepresentation.Tree
      WithList          : InternalRepresentation.Attribute
      ImportedEntity    : ...
      ...        -- Declarations of other local objects and types

      use InternalRepresentation

    begin
      foreach Entity in EntitiesOf ( InterfaceSubunit1 ) loop
        WithList := GetAttribute ( ImportedEntities, Entity )

        foreach ImportedEntity in WithList loop
            if ( ImportedEntity.Parent = InterfaceSubunit2 ) then
                if ( Unavailable ( ImportedEntity, Entity ) ) then
                    RecordInterfaceCheckError ( ... )
                elsif ( SemanticConflict ( ImportedEntity, Entity ) ) then
                    RecordSemanticInconsistencyError ( ... )
                end if
            end if
        end loop
      end loop

  end InterfaceCheck

  ...  -- Bodies of other low level analysis procedures

end LowLevelAnalysisTools
```

Figure 6: Body Subunit of Low Level Analysis Tools Package.

the interface specification subunit of package LowLevelAnalysisTools and its body subunit does not reveal any errors. On the other hand, invoking these tools to analyze the consistency between the body subunit and the interface stub of package InternalRepresentation reveals that function GetAttribute is being used improperly; the parameters to the function are reversed. A decision must then be made as to which subunit is in error. Let us assume it is decided that the body subunit is incorrect. We will then assume further that the parameter list is appropriately edited, and finally that the subunit is resubmitted through the update processor and is now found to be consistent.

Eventually, an official version of package InternalRepresentation is delivered. In general, the interface specification subunit of a utility package such as InternalRepresentation (e.g., Diana) is delivered in a "virgin" state; application specific interface restrictions are left unspecified. In order to tailor the package to the particular application under development and foster a high degree of interface control, the interface specification subunit must be augmented to include any desired restrictions on its provided entities. Significantly, such augmentation does not affect the implementation of the package since restrictions on provision exclusively involve the unit's interface. Returning to the example, the appearance of the (augmented) official interface specification subunit of package InternalRepresentation makes the interface stub obsolete. All checking can now be performed—with greater confidence—against the true interface subunit. As noted in Section 4, this checking can be expedited by using the already-checked stub, rather than the subunit bodies, as a basis for most of the checking of the newly-introduced interface specification.

6. Future Enhancements and Current Status

The approach to precise interface control described here improves upon the capabilities provided in the existing approaches. Nevertheless, there are a few additional interface control capabilities that might be desirable and hence we are investigating a number of enhancements to the design of the mechanism we have presented. Most notably we are examining even more stringent controls over provision. In particular, we are considering run-time constraints for providing dynamic control over the provision of an entity [MINS83]. In addition, we are considering finer control on the operations that can be performed on objects [JONE78] and the operations that can be associated with types. We suspect that these enhancements, while semantically quite powerful, can be added to our current framework with only relatively simple syntactic modifications.

We are also exploring how a general approach to precise interface control may provide protection capabilities [SALT75, AMBL77b]. Section 3 outlined how the system structure that we have adopted will support stringent managerial control. This concept, combined with the enhancements described above, provides the basis for extensive protection capabilities. We are investigating the strengths and limitations of such a mechanism, but suspect that with minor extensions it can provide language level support for both object-oriented and capability-based protection, which could be useful in database and operating system applications. Another extension being explored is interface control and system structure for concurrent systems. We suspect that in many cases the constructs that we adopt for sequential programs will also suffice for concurrent systems. In other instances, particularly if a system can spawn processes dynamically, additional constructs may be required.

To evaluate our ideas and to demonstrate the system structure, language constructs, and support environment, we are currently developing a prototype implementation. As the example in Section 5 indicates, we are using the system structure and language constructs described in this paper in designing the prototype. Although pre-implementation and programming languages embodying this approach to precise interface control could be based on almost any modern language, we are basing our prototype's languages on Ada. The prototype tools will be provided as part of a preprocessor that performs the necessary recognition, analysis, and eventual translation into ANSI-standard Ada. We believe that this prototype will be very important in demonstrating the added power provided by our approach to precise interface control as well as showing its applicability throughout the software development process.

REFERENCES

AMBL77a 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**, appearing in **SIGPLAN Notices**, Vol. 12, No. 3, pp.1-10, March 1977.

AMBL77b A.L. Ambler and C.G. Hoch, *A Study of Protection in Programming Languages*, **Proceedings of an ACM Conference on Language Design for Reliable Software**, appearing in **SIGPLAN Notices**, Vol. 12, No. 3, pp.25-40, March 1977.

BABI82 W. Babich, L. Weissman, and M. Wolfe, *Design Considerations in Language Processing Tools for Ada*, **Proceedings of the Sixth International Conference on Software Engineering**, Tokyo, Japan, pp.40-47, September 1982.

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 Programming Language**, appearing in **SIGPLAN Notices**, Vol. 15, No. 11, pp.139-145, November 1980.

DERE76 F. DeRemer and 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.

DOD83 **Reference Manual for the Ada Programming Language** (ANSI/MIL-STD-1815A), United States Department of Defense, Washington, D.C., January 1983.

EVAN83 A. Evans Jr. and K. J. Butler (eds.), **Diana Reference Manual (Revision 3)**, TL 83-4, Tartan Laboratories Inc., Pittsburgh, Pennsylvania, February 1983.

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*, **Communications of the ACM**, Vol. 21, No. 5, pp.358-367, May 1978.

KERN83 J. S. Kerner, *Design Methodology Subcommittee Chairperson's Letter and Matrix*, **Ada Letters**, Vol. 2, No. 6, pp.110-115, May-June 1983.

KOST76 C.H.A. Koster, *Visibility and Types*, **Proceedings of a Conference on Data: Abstraction, Definition and Structure**, appearing in **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*, **Lecture Notes in Computer Science**, Vol. 114, Springer-Verlag, New York, 1981.

MINS83 N.H. Minsky, *Locality in Software Systems*, **Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages**, Austin, Texas, pp.299-312, January 1983.

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.

OSTE82 L.J. Osterweil, *Toolpack - An Experimental Software Development Environment Research Project*, **Proceedings of the Sixth International Conference on Software Engineering**, Tokyo, Japan, pp.166-175, September 1982.

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**, appearing in **SIGPLAN Notices**, Vol. 12, No. 3, pp.11-18, March 1977.

SALT75 J.H. Saltzer and M.D. Schroeder, *The Protection of Information in Computer Systems*, **Proceedings of the IEEE**, Vol. 63, No. 9, pp.1278-1308, September 1975.

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

TAFT82 T. Taft, *Diana as an Internal Representation in an Ada-in-Ada Compiler*, **Proceedings of the AdaTEC Conference on Ada**, Arlington, Virginia, pp.261-265, October 1982.

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.

WULF73 W.A. Wulf and M. Shaw, *Global Variable Considered Harmful*, **SIGPLAN Notices**, Vol. 8, No. 2, pp.28-34, February 1973.