SUPPORT FOR PROGRAMMING-IN-THE-LARGE WITH ADA®

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

COINS Technical Report 84–25
November 1984
(Revised December 1984)

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

A subsequent version of this report is to appear in IEEE Software, Vol. 2, No. 2, March 1985, under the title "Ada-Based Support for Programming-in-the-Large".

This work supported in part by the NSF under grant DCR–84–04143.

ABSTRACT

Developers and maintainers of large systems need extensive support for describing, analyzing, organizing, and managing the numerous modules in those systems—that is, they need an environment for "programming-in-the-large". We are developing such an environment, based upon Ada®, that relies upon a small number of specialized language features and an integrated set of tools. The language features facilitate flexible and precise descriptions of interface control and also complement the modularization capabilities already found in Ada. The tools support incremental development, analysis, and management throughout the software lifecycle. The focus of this paper is on the environment's language features. A brief overview of the support to be provided by the environment's tools is also given. A realistic example demonstrating use of the language features and analysis tools during design of a software system is presented.

1. Introduction

The Ada® programming language is intended for the implementation of large and complex software systems. Such systems often exceed a half-million lines of code. If their developers adhere to the software engineering maxim that no module should contain more than fifty lines of code, then the number of modules in such systems will exceed ten thousand! As DeRemer and Kron point out, dealing with "a *large collection of modules to form* a *'system' is an essentially distinct and different intellectual activity from that of constructing the individual modules*" ([5], pg. 82). Thus, developers and maintainers of large Ada systems will require tools beyond those for "programming-in-the-small" (e.g., syntax-directed editors, compilers, and debuggers) [9,11,2]; they will need extensive support for describing, analyzing, organizing, and managing the modules in a system—that is, an environment for "programming-in-the-large".

In essence, programming-in-the-large involves the two complementary activities of *modularization* and *interface control*. Modularization is the identification of the major system modules and the entities contained in those modules, where *entities* are language elements that are given names, such as subprograms, data objects, and types. Interface control is the specification and control of the interactions among entities in different modules.

To provide proper support for modularization and interface control, an environment for programming-in-the-large should address a number of software development concerns. These include:

- *Lifecycle Support.* Support for modularization and interface control is of primary importance during every phase of the software lifecycle. For example, generating a description of the major modules and their interactions is one of the first activities undertaken during the early phases of software development, while insuring that those

interactions remain correct and consistent is a primary consideration during implementation and maintenance. Providing such support during the pre-implementation stages of development requires that the environment deal explicitly with incompleteness in representations.

- *Precise Interface Control.* Modules are, essentially, producers and consumers of resources. As producers they make themselves, and perhaps their internally defined entities, available for use by entities in other modules. As consumers they, or their internally defined entities, utilize entities made available by other modules. The environment should provide means for precisely specifying this relationship among modules. Ideally, such specifications should be from the points of view of both producers and consumers.

- *Analysis Support.* For large systems, the ability to specify relationships among modules is of limited value without tools to aid in analyzing that information. Feedback about the consistency of the interactions among modules must be automatically and readily available. Furthermore, it should be possible to begin performing analyses during the earliest stages of development and continue through maintenance.

- *Version Control.* Systems must often be configured for a number of different operating environments (e.g., operating systems, machines, peripherals) and developers often must contend with maintaining running versions of a system while developing new, extended versions. Thus, an environment for programming-in-the-large must facilitate the description and configuration of these system families.

- *Managerial Support.* Large software systems are usually produced by teams of individuals, and different modules are typically developed by different team members. Thus, programming-in-the-large is a management activity, involving matters such as organization and interaction. The environment should provide project leaders means for controlling the modularization and interface-control activities, while supporting a variety of managerial disciplines.

- *Method Independence.* Various methods have been proposed for guiding the modularization process [6]. The environment should be general enough and powerful enough to be employed with any one of these methods, since none can realistically be expected to be appropriate for all applications.

We are developing an environment for programming-in-the-large, based upon Ada, that provides capabilities meeting the criteria outlined above. This environment relies upon a small number of specialized language features and an integrated set of tools that have been carefully tailored to support incremental development and to be applicable across the phases

in the software lifecycle. We believe that Ada already provides substantial support for flexibly representing module decomposition, but is very limited in its support for interface control. Therefore, our Ada-based environment's language features consist primarily of constructs for precisely describing entity interaction. We refer to this environment as "PIC" since *p*recise *i*nterface *c*ontrol is the central concern being addressed. When the PIC language features are combined with modularization capabilities such as those found in Ada, the result is a uniform framework facilitating programming-in-the-large. This framework supports many of the major design methods that have been proposed, including those most closely associated with Ada. Moreover, it supports both graphical and textual representation of the architectural structure of a system, as well as easy movement between these two forms. The environment enhances the descriptive capabilities of the language features by providing an integrated set of tools for analyzing and managing the interface control aspects of a software system. Furthermore, while the environment currently does not contain specialized support for version control, it is compatible with a number of recent proposals for version control mechanisms (e.g., [26,14]).

The PIC environment is intended to significantly extend the capabilities of Ada development systems (e.g., [11,2,24]). All modularization and interface-control decisions made throughout the development and maintenance process are to be recorded using languages that incorporate the PIC language features and then organized, managed, and analyzed using the support tools. This implies that during pre-implementation phases the language features and support tools are to be used in conjunction with specification and design languages, while during implementation and maintenance they will be used in conjunction with the Ada programming language. Figure 1 depicts this organization, showing various kinds

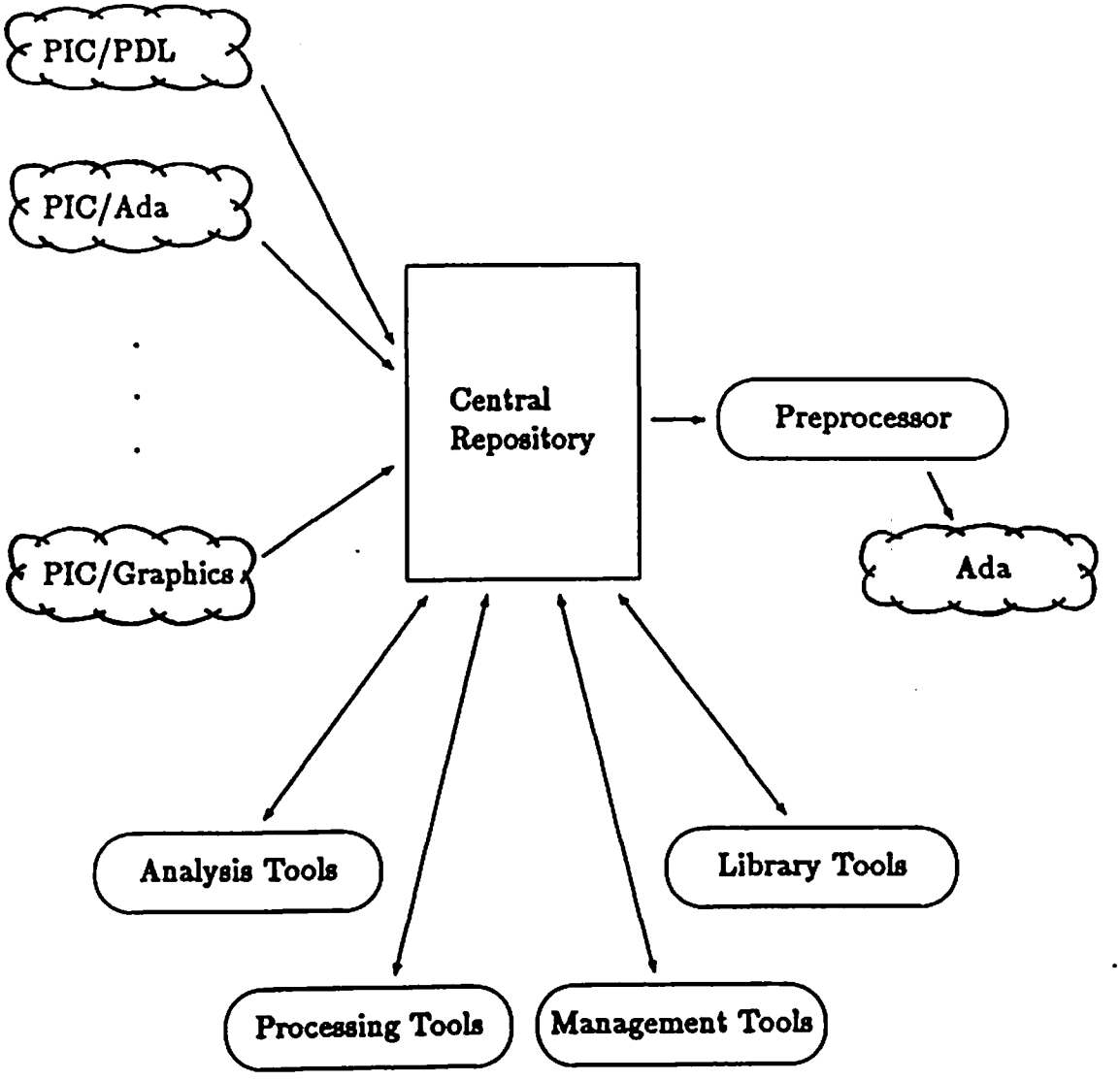

Figure 1: Conceptual Organization of the PIC Environment.

of support tools applied to a variety of PIC-oriented languages, such as an Ada-like textual pre-implementation language (PIC/PDL), a textual implementation language that is an enhanced version of Ada (PIC/Ada), and a graphical language (PIC/Graphics). Compilation of PIC/Ada will be a two-step process in which the interface control information is first incorporated, as far as possible, into standard Ada code (essentially a preprocessing step) and then a normal Ada compilation is carried out. Since the design of the language features is in harmony with the design philosophy underlying Ada, much of the incorporation process is straightforward. Interface control information that cannot be directly captured in Ada can still be enforced through the environment's analysis tools prior to the preprocessing step. Any changes that are made to the Ada implementation, including those resulting from maintenance activities, will be performed using the PIC-oriented languages and will be processed by the support tools. As a result, the environment, with all its descriptive, analytical, organizational, and managerial capabilities, will be actively involved in all phases of development and maintenance and thus represents a genuinely integrated approach to programming-in-the-large.

The focus of this paper is on the PIC language features. The next section presents the basic concepts underlying our approach and describes related work. Section 3 introduces the language features, illustrating their use in PIC/PDL. A brief overview of the support to be provided by the environment's tools is given in Section 4. Section 5 discusses possible extensions to the environment. Finally, the appendix contains a realistic example of the use of the language features and analysis tools during design.

2. Background

A general view of interface control, offering a richer conceptual foundation than views based solely on traditional visibility concepts of declaration, scope, and binding, arises from an important distinction between two aspects of visibility: *requisition* of access and *provision* of access. *Access* to an entity is the *right* to make reference to, or use of, that entity in declarations and statements. Requisition of access occurs when an entity (implicitly or explicitly) requests the right to potentially refer to some set of entities. Thus, in most programming languages a subprogram would typically request access to itself and any locally declared entities, as well as certain non-local entities. Provision of access occurs when an entity (implicitly or explicitly) offers, to some set of entities, the right to potentially refer to that entity. Again, in most programming languages a subprogram typically provides access (i.e., the right to potentially invoke that subprogram) to itself and, in languages supporting nesting, the subprogram's parent, siblings, and descendents. Under this view, an actual reference by an entity E_i to an entity E_j is only possible if E_i requests access to E_j and E_j provides access to E_i.[1]

An interface control mechanism is the means for specifying requisition and provision. Programming languages have historically taken different approaches to this specification. In languages such as ALGOL60 and Pascal, the *nesting* interface control mechanism results in requisition and provision that are essentially mirror images; access that is requested by an entity is always also provided to that entity and *vice versa*. In the designs of more

[1] In the remainder of this paper, when the intended meaning is clear, the word "access" is dropped from certain phrases involving the terms "requisition" and "provision". Thus, a "requested entity" is one to which access is requested, and the "requisition of an entity" refers to the requisition of access to the entity. Similarly, a "provided entity" is one for which access is provided, and the "provision of an entity" refers to the provision of access to the entity.

recent languages, particularly languages intended for the construction of large and complex software systems such as the implementation languages Ada [7], GTEL Pascal [22], Mesa [18], and MODULA-2 [27], the program specification language SPECIAL [21], various program design languages based on Ada [23,20], and the module interconnection languages MIL75 [5], C/Mesa [18], and INTERCOL [26], the desire for greater control over interfaces has resulted in mechanisms that address requisition and provision in separate, but often unequal, ways.

These languages have relied, to a greater or lesser degree, on the concepts of *encapsulation* and *explicit import/export control* to describe both the accesses that are requested and the accesses that are provided by the entities in a module 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 subprograms, objects, types, and other encapsulations. Explicit import/export control furnishes the means by which a module requests and provides access to external entities for its constituent entities. In Ada, the encapsulation construct is the *package* and import/export control may be effected through a combination of features including *with clauses*, *visible* and *private parts*, and *nesting*. Unfortunately, not one of the languages mentioned above, including Ada, supports precise and flexible control over both what accesses an entity can request and what accesses an entity can provide [3,28]. For instance, Ada's *with clause* only permits the requisition of access to either no entities or all entities in the visible part of a package and Ada's *private/visible* mechanism only permits the provision of access to either no modules or all modules in the scope of a package. These and other shortcomings of modularization and interface control in Ada are discussed in greater detail in the following section.

It is our contention that precise interface control mechanisms are of great potential value

to developers and maintainers of large software systems [4]. Such precision would permit the requisition and provision of exactly those accesses desired in a system while disallowing others. In addition, support for both precise requisition and precise provision serves to encourage redundancy, which, in turn, can facilitate more rigorous analysis of the interface relationships of a system's components. For example, based on this view it is possible to formulate complementary descriptions of exactly how two modules are intended to interact, giving one description from the perspective of each of the modules, and then to analyze those interactions by checking the two descriptions for consistency.

The language features and support tools discussed in the next two sections exploit the concepts outlined here to build upon Ada's basic encapsulation and import/export concepts, thereby providing mechanisms capable of describing, analyzing, and managing the interface control aspects of large systems precisely and flexibly. While this paper focuses on support for Ada, the basic language features and tools could be applied to most modern programming languages.

3. Language Features

There are two aspects to the language features of the PIC environment. First, they provide a system structure that imposes a strict separation of the interface control information from the algorithmic details of how that information is used locally by a module. Second, they include constructs that, in conjunction with this system structure, provide for precise interface control. Thus, the language features in effect constitute a *module interconnection language* for Ada systems, where the interface control component of a system can be viewed as a description in this language.

The environment recognizes two kinds of modules: *packages* and *subprograms*, which correspond to their Ada namesakes.[2] To realize the separation of interface control information from algorithmic detail, a module always consists of two physically distinct parts: a *specification submodule* and a *body submodule*. A package's specification submodule describes the entities encapsulated by the package, while a subprogram's specification submodule simply describes the information needed for an invocation of the subprogram. In the case of both packages and subprograms, a specification submodule also completely describes a module's requisition of access, through one or more *request clauses*, and provision of access, through one or more *provide clauses*. The body submodule for both packages and subprograms contains the actual code sections realizing the module. During the pre-implementation phases, the body would take the form of an Ada-based PDL description, while in later phases it would consist of standard Ada code.

Figure 2 presents a simple example illustrating several aspects of the language features as they appear in PIC/PDL. The example shows the specification submodule of a print queue package implemented using linked lists. The package provides a type for print jobs, **Job**, a type for print queues, **Queue**, and two subprograms, **Enqueue** and **Dequeue**, realizing the abstract operations of adding and removing a job from a print queue. A more realistic example is given in the appendix, showing the specification and analysis of a system during the high-level and low-level design phases. That example is drawn from our initial work on the prototype PIC environment that we are currently developing.

A specification submodule in this notation is essentially an Ada program unit specification together with a small number of additional, and powerful, features for enhancing interface

[2]To simplify the presentation, Ada tasks and generics are not considered in this paper.

```
package PrintQueue is
      request LinkedList.( ListElement, List );

   type Job
      is new LinkedList.ListElement  provide to Reorder;

   type Queue
      is new LinkedList.List  provide to Reorder;

   procedure Enqueue ( J : in Job; Q : in out Queue )
         request LinkedList.Append, ...;

   procedure Dequeue ( J : out Job; Q : in out Queue )
         request LinkedList.Delete
         provide to Printer;

private
   procedure Util ( ... )
         request LinkedList.Statistics, ...;

   ...;

end PrintQueue;
```

Figure 2: Specification Submodule of a Print Queue Package.

global provision of a library module 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 [17]. Therefore, our notation extends Ada by including the *provide clause*, which may be appended to any of a package's provided entities in order to selectively limit their provision to external modules. The absence of a *provide clause* on a provided entity is interpreted to mean that access to the entity is provided to "all".[5] For example, access to procedure **Enqueue** is provided to all, but the *provide clause* attached to procedure **Dequeue** indicates that it is provided only to module **Printer**. Thus, while any module in the system would be allowed to add a job to a print queue, **Printer** is the only module permitted to remove a job. The *provide clause* can also be applied to an unpackaged subprogram. An appended *provide clause* for such a subprogram limits its provision to other modules and avoids the need to create a superfluous package to encapsulate the subprogram and control its availability.

Another aspect of the approach we are proposing 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, a provided type may be associated with two *provide 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 *provide clause* associated with the representation serves as a further restriction on the representation beyond that inherited from the name.[6] As shown in Figure 3, six

[5] This choice was made in an effort to keep within the spirit of Ada. The alternative, which is to interpret an absent *provide clause* as meaning provided to "none", might be preferable for some languages.

[6] As in Ada, the case of a type's representation being completely hidden within the defining package is denoted simply using the keyword *private* in place of the representation; the representation of that type is then given in the package's *private part*. It might be preferable in some languages, particularly those that do not already textually separate provided and hidden entities, to instead attach the clause "*provide to none*" to

control. The *request clause* appearing at the top of the submodule in Figure 2, for instance, indicates that access to the entities **ListElement** and **List**, defined in package **LinkedList**, is requested by all of the entities in package **PrintQueue**. The entity **Append**, also defined in package **LinkedList**, is only to be referred to by procedure **Enqueue**, since **Enqueue** is the only entity in **PrintQueue** that has an attached *request clause* mentioning **Append**. Similarly, the *request clauses* attached to procedures **Dequeue** and **Util** indicate that only these subprograms will refer to the entities **Delete** and **Statistics**, respectively, defined in package **LinkedList**.[3] Notice that the *request clause* in this notation is more flexible than its counterparts in most other languages, including Ada's *with clause*, in at least two ways. First, it does not necessarily import all the provided entities of a package but can import subsets of those entities. Second, a *request clause* can be attached to an individual packaged entity, as well as to the package itself, so that requisition by the entities within a package can be differentiated.

Figure 2 also illustrates the language features' support for provision. In Ada, provision of packaged entities is controlled through constructs that textually separate a package's provided entities from its hidden entities.[4] 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 Ada, provision is controlled on an all-or-nothing basis; either access to an entity is provided to every module (in a given scope), or it is provided to no module, and so the entity is hidden. While these two extremes are useful (for instance, in describing the

[3]Although not shown in this paper, a complete subprogram header, which includes the subprogram's name and formal parameters, can be given in a *request clause* or a *provide clause* to disambiguate requested and provided access to an overloaded subprogram.

[4]The language features presented here do not permit subprograms to provide their internally defined entities to other modules, as do nested languages. In fact, subprograms can provide nothing but themselves; packages are the only modules that can provide access to their internally defined entities.

basic levels of control result, permitting a high degree of flexibility in controlling the use of a type definition. In contrast, Ada provides the first and third levels shown in the figure.[7] Associating two *provide 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 modules [12]. This sharing is illustrated in the print queue example where access to the names of the types **Job** and **Queue** are provided without restriction, but provision of access to their representations is limited to the defining package **PrintQueue** and module **Reorder**.

Another aspect of the language features suggested in Figure 2 is their applicability to high-level, incomplete descriptions of a system's components and their interaction. The *incompleteness construct*, denoted using an ellipsis, is useful for explicitly indicating where details that will be supplied later have been omitted from a description. It complements other constructs, not illustrated in this example, that facilitate the formulation of abstract, pre-implementation descriptions, such as notations to formally specify a module's external behavior or to describe intended algorithms. When used in conjunction with such constructs, the language features are well suited for expressing modularization and interface properties during early stages of a system's development.

In addition to specification and body submodules, the language features include a third kind of submodule referred to as a *specification stub*. This kind of submodule is supplied in response to the fact that interacting modules of large software systems are often developed independently—perhaps even at different times. If, at some point before development is com-

---

the representation in the type definition.

[7] Although in some cases the other levels can be approximated with careful nesting of packages, nesting is not general enough to capture these levels in every situation.

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

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

(3) type A
      is private;
      -- name:           no restriction
      -- representation: complete restriction
      --   name provided to all; representation not provided

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

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

(6) type A provide to Y
      is private;
      -- name:           restriction
      -- representation: complete restriction
      --   name provided only to Y; representation not provided
```

Figure 3: Basic Levels of Control Over Provided Packaged Type Definitions.

plete, a group of modules requires access to entities from a module for which no specification submodule is yet available, a specification stub submodule can be constructed. A specification stub usually only contains some of the information that would eventually be described in the specification submodule. In particular, the specification stub need not contain any information about the module's requisitions but only needs to describe what is being provided by that module to the modules in the requesting group. A number of different specification stub submodules of a module may exist to accommodate various intended uses of that module resulting from the development activities of a number of different groups. The specification stub mechanism provides a means for the various groups of users of a module to document these *views* of the module before the module is available. Two examples of specification stub submodules of a linked list package are shown in Figure 4. The two submodules partially describe the two, slightly different, views of package `LinkedList` that have been defined by the developers of the print queue package of Figure 2 and of a stack package, respectively. The *used-by clauses*, appearing at the top of the specification stub submodules, indicate the intended users of those stubs. When a module's specification submodule is available (in a library) or completely known, then it could be used for processing instead of the stub. As described in the next section, the environment provides tools to assure consistency among the stubs, to generate an accumulated view, and to check that the specification submodule, when submitted, is consistent with any existing stubs of that module. Use of specification stub submodules is further illustrated in the example presented in the appendix.

There are a number of benefits associated with the PIC language features. One of these is improved readability. Since the language features explicitly and clearly state what accesses can be requested and provided, we contend that the readability of software with these features

```
package stub LinkedList is
      used by PrintQueue;

   type ListElement;
   type List;

   procedure Append ( ... );
   procedure Delete ( ... );
   procedure Statistics ( ... );

   ...;

end LinkedList;
```

```
package stub LinkedList is
      used by Stack;

   type ListElement;
   type List;

   procedure Insert ( ... );
   procedure Delete ( ... );

   ...;

end LinkedList;
```

Figure 4: Two Specification Stub Submodules of a Linked List Package.

is enhanced, making it easier to discern the relationships among the modules. This in turn makes systems easier to change and therefore easier to develop and maintain.

Additional benefits accrue from consolidating interface control information into the specification submodules and separating a module's interface specification from its body. While in languages such as Ada, Mesa, and MODULA-2 there is support for "specifications" of modules separate from their bodies, these specifications do not completely define the interfaces to modules. In Ada, for instance, a body may itself import entities using an attached *with clause*. In our approach the *request clauses*, which can only appear in a specification submodule, completely constrain the external entities that can be referred to by a module's body. Because of this complete separation of concerns, the language features constitute a genuine module interconnection language, and specification and specification stub submodules written in this language fully describe the interface control component of a system. Moreover, by enforcing a complete separation of concerns, the language features facilitate incremental development, managerial control, and information hiding.

As Ada and other modern languages have demonstrated, the separation of specification and implementation concerns fosters incremental development (i.e., incremental analysis and separate compilation) of large software systems. By refining this separation and adding precise interface control, the module interconnection language based on the PIC language features enhances incremental development by permitting more meaningful and detailed interface consistency analysis to be performed as well as allowing this analysis to be done early and throughout the software lifecycle. In particular, the interface control component of a system can be created and analyzed separately from the bodies (i.e., implementations) of the modules in that system and, in fact, only needs to be combined with the bodies to facilitate

further analysis or to support separate compilation. Moreover, the specification submodule of a given module could be associated with more than one body submodule of that module, where the choice of the actual implementation of the module is delayed until perhaps as late as link time. Support for incremental development is further enhanced by the language features' explicit treatment of incompleteness, which, among other things, permits module development to proceed in any desired order. This contrasts strongly with Ada's rigid method for incremental development of library units, where primarily for code generation reasons a bottom-up development process is enforced.

Managerial control over modularization and module interfaces can be achieved by permitting only a project leader to create or modify a specification or specification stub submodule. Alternatively, under a more decentralized discipline, implementors can construct specification stub submodules of the modules they expect to use in their implementations; the stubs of a given module can then be collected together by a project leader responsible for deciding (or negotiating!) on the final form of the specification submodule. Finally, the language features also support an autonomous discipline in cases where managerial control is not desired, since implementors of modules can assume the role of project leader and construct their own specification submodules.

Finally, information hiding is also facilitated by the separation of concerns supported by the module interconnection language since a developer working on a module that will refer to entities from another module only needs to see the specifications of the provided entities of the referred module and each such specification only needs to contain information relevant to the referring module. The *provide clauses* in a specification submodule actually define the different views particular external modules have of a package's provided entities.

In sum, while many existing specification, design, programming, and module interconnection languages support some of the desired capabilities, none supports all. The language features outlined here incorporate capabilities distilled from many of these previous attempts. The resulting language framework is relatively simple and straightforward, yet surpasses these previous attempts by supporting precise interface control as well as the comprehensive collection of benefits outlined above.

4. Support Tools

Despite the simplicity of the language features, development of the proper interface relationships for large software systems remains a complex task. This task could be aided by an integrated toolset consisting of analysis, library, management, and general processing tools to facilitate development and assure consistency of the software system. In this section, we give a brief overview of the support to be provided by the PIC environment's tools.

4.1 Analysis Tools

The various features provided in a language influence not only the precision possible in interface description but also the complexity of analyzing interface relationships. For instance, analysis techniques developed for systems described in nested languages such as ALGOL60 and Pascal are quite straightforward, largely because those systems are monolithic and the controls provided in the languages are quite limited. More recently-designed nested languages, such as MODULA-2 and Euclid [13], furnish additional interface control features in an attempt to compensate for the inadequate controls of nesting. Still other features are supplied in some new nested languages, such as Ada and GTEL Pascal, to support

incremental development. Unfortunately, the combination of nesting and these additional interface control and incremental development features complicates the analysis techniques applicable to those languages (e.g., [10,19]). The absence of nesting in languages such as Gypsy [1] and CLU [15], which also support incremental development, allows for simpler, but no less powerful, analysis techniques.

Our approach to interface control has its own interesting ramifications for the design of analysis techniques. First, the language features' added expressive power makes possible the precise description of intended interface relationships and hence raises the prospect of more revealing analyses. Second, the language features' explicit support for incompleteness causes a reexamination of the traditional meaning of *consistency* in interface relationships. Third, the desire for the approach to be applicable and integrated across the phases in the software lifecycle dictates that the analysis tools must be flexible enough to permit their operation upon distinctly different forms of representation. Each of these areas is discussed in turn below.

We have identified a basic set of analyses that exploits the expressive power of the language features. Viewing analyses as comparisons between pairs of submodules, two distinct classes arise; *intra-module analyses* focus on the interface relationships between two submodules of the same module, while *inter-module analyses* focus on the interface relationships between two submodules of different modules. An example of an analysis belonging to the first class is one that compares a specification submodule to its corresponding body submodule to check, among other things, that the body only refers to those external entities requested by the specification. An example of an inter-module analysis is one that compares two specification submodules of different modules to check, among other things, that the entities defined in

the second module which the first module requests access to are in fact specified as provided by the second to the first. By composing various basic analyses from these two classes, a rigorous evaluation of the consistency of the interface relationships results.

Analysis techniques for the early stages of the software lifecycle must be able to deal with *incompleteness*. Existing analysis techniques for even those languages that permit the explicit expression of incompleteness do not appear to provide this support. When a software system under analysis is complete (or assumed to be complete for the sake of analysis), it is essentially straightforward to define what it means for two submodules to be consistent. If, however, one or more submodules are incomplete (e.g., contain incompleteness constructs), then questions arise as to how consistency should be defined and what information the analysis tools in the support environment should provide to the user. To address these issues we define consistency between two submodules as a state in which the interface relationship cannot be shown incorrect. Consider the example in Figure 5. A package `Pac` provides two objects, `Obj1` and `Obj2`. Access to `Obj1` is provided without restriction, while access to `Obj2` is limited through an attached *provide clause*. Appearing in that *provide clause*, along with the name of an entity `Proc1`, is an ellipsis indicating a place where the definition of the package is incomplete.[8] Thus, it can be assumed that there might be entities in addition to `Proc1` to which `Obj2` is provided. The *request clause* in the specification submodule of procedure `Proc1` indicates that `Proc1` requests access to entities `Obj1` and `Obj2` of package `Pac`. This is certainly consistent with the specification for `Pac` since `Obj1` is provided to all entities and `Proc1` appears in the *provide clause* of `Obj2`. As is the case for procedure `Proc1`, the

[8] Of course, no submodule in a developing system can ever really be considered complete, whether or not it contains any PDL constructs (such as ellipses), since it may always be updated at any time. The presence of PDL constructs, however, gives the support tools explicit information regarding incompleteness that they can exploit.

```
package Pac is
   Obj1 : Typ1;

   Obj2 : Typ2
      provide to Proc1, ...;
end Pac;


procedure Proc1
   request Pac.( Obj1, Obj2 );


procedure Proc2
   request Pac.( Obj1, Obj2 );
```

Figure 5: Consistency in the Presence of Incompleteness.

specification submodule of procedure **Proc2** indicates that **Proc2** requests access to entities **Obj1** and **Obj2** of package **Pac**. But unlike that case, access to **Obj2** is not explicitly provided to **Proc2** by **Pac** since **Proc2** does not appear in the *provide clause* attached to **Obj2**. Under our definition, however, **Proc2**'s interface is still considered to be consistent with the interface of **Pac** because the presence of the ellipsis in the *provide clause* allows the possibility that **Obj2** will at some time be provided to **Proc2** and therefore no inconsistency can be shown to exist between the interfaces.

Clearly, there is a qualitative difference between the consistency of **Pac** and **Proc1** and the consistency of **Pac** and **Proc2**. This difference is captured by recognizing that consistency between two submodules only depends upon the consistency of those portions of the submodules that actually interact. From this, two levels of consistency between submodules

can be defined. Thus, we say a pair of submodules is *consistent* if 1) the relationship between the two submodules cannot be shown incorrect and 2) the portions of the submodules that are relevant to their interaction are complete. Two submodules are said to be *conditionally consistent* if (1) holds but (2) does not. In the example above, it can be seen that **Pac** and **Proc1** are consistent but **Pac** and **Proc2** are only conditionally consistent.

Our goal of providing analysis support that is applicable and integrated across a range of software lifecycle phases is closely related to support for incompleteness. Indeed, once the handling of incompleteness is incorporated into the basic analysis techniques then the uniform application of these techniques to descriptions for different software lifecycle phases hinges on the development of a consistent internal representation of those descriptions. The internal representation we have developed is founded on a formal model of interface control, which is also used for describing and evaluating interface control mechanisms [28]. Briefly, the formal model is based on a directed graph model of module interfaces, which is used to uniquely represent a particular set of interface relationships. The nodes of the graph correspond to entities, while two separate sets of arcs are used, respectively, to denote the requisition and provision relationships among those entities. The interface control aspects of the various languages for specification, design, and implementation are translated into this internal representation. The analysis tools are then applied to that representation.

4.2 Library Tools

It is common practice for large software systems to be broken down into more manageable *libraries*, each consisting of several logically related modules. Historically, these libraries were viewed simply as repositories for compiled code—the end product of the development effort.

The linker was given the job of checking the interfaces among the modules in the library and hence it was only at link time that the opportunity arose for discovering interface errors. More recently, the concept of the *program library* has emerged (e.g., in [7,15]). Through the program library, the compiler is able to incrementally perform the interface checking formerly done *en masse* by the linker. It does this by saving within the program library, in addition to the compiled code, certain pieces of relevant information discovered during the compilation process. The compiler can then use the information gained from previous compilations to check the interface consistency of subsequently compiled modules. Within a single program library, therefore, consistency can be maintained.

For the development of large software systems involving numerous developers, the program library as simply a repository for the compilation information of an entire program is not an adequate tool. Incremental development involves more than just incremental compilation; it involves incremental analysis during all phases of the software system's development. Moreover, projects involving many members necessitate separate work areas, both for individuals and for various working groups. We have found that what is truely required to support incremental development in such a setting is a synthesis of the capabilities of a program library with those of an operating system's file manager. We call the result of this synthesis a *development library*.

The development library resembles a program library in that it maintains information about the submodules it contains. More than just holding compiled code and interface information, the development library may store various forms of the source code. More importantly, the development library maintains information concerning the results of analyses among its submodules. As a further generalization of the program library concept, a devel-

opment library may even contain other development libraries; this is where the capabilities of a file manager come in to play. File systems, such as the one in the UNIX™ operating system, provide a simple mechanism for partitioning a work area. In addition, they provide the means for sharing among work areas; a UNIX file or directory may be a member of more than one directory. By incorporating this file-system model into the structure of the development library, it becomes possible to conceive of using multiple libraries in the development of a system and to share information, particularly interface and analysis information, among those libraries in a fairly general way.

The cost of sharing information among development libraries, however, is added complexity in the designs of the analysis and library tools. Indeed, sharing information among development libraries is much more complicated than simply sharing a pointer, as is done for files in operating systems. For instance, activities in one development library, such as changes to module interfaces, may affect other libraries comprising the system. The analysis and library tools are thus responsible for deciding where and how to propogate those effects. This added complexity is nonetheless offset by the fact that sharing of interface and analysis information facilitates sharing and reuse of software in ways that are not possible when the sharing is only at the level of source code files.

4.3 Management Tools

One of the necessary capabilities of an environment for programming-in-the-large is support for managerial control. Thus, there must be management tools that provide mechanisms to control the establishment and modification of a system's interface relationships. Since those relationships are completely determined by the specification and specification stub submod-

ules in PIC, the management tools can operate by controlling programmer access to these submodules. The management tools, therefore, must work in close cooperation with the library tools. Furthermore, they should enforce whichever managerial discipline is chosen for the project. In particular, if the project leader is given sole responsibility for determining interface relationships, then only that person should be permitted to enter or replace specification and specification stub submodules in the development library. Under a more decentralized discipline, implementors should be permitted to enter or replace specification stub submodules, but only the project leader should be permitted to enter or replace the "official" specification submodules. Finally, an autonomous managerial discipline would permit any project member to enter or replace specification submodules for the modules they are developing or maintaining in the development library.

4.4 Processing Tools

A number of general processing tools must be available in the PIC environment to perform such tasks as creating and modifying submodules, generating specification submodules from sets of specification stub submodules, generating views of modules from their interfaces, reporting on the effects of updates on interface relationships, and reporting on submodule interactions from both the requisition and provision perspectives. As do the analysis tools, these tools are designed to use the consistent internal representation and handle incompleteness appropriately so that they too can be uniformly applied throughout the software development and maintenance process. The common internal representation additionally facilitates easy movement from the graphical representations of a system to corresponding textual representations and *vice versa*, permitting further development of the system to be

recorded through refinements in either or both representations.

One very important processing tool is the PIC/Ada preprocessor. The design of the language features makes the translation of PIC/Ada into Ada a relatively straightforward operation. For instance, the specification submodule, as mentioned in Section 3, closely resembles an Ada program unit specification. Certain situations, however, do require some special care during translation. For example, the fact that under our approach requisition of access can only be described in specification submodules leads to a conflict, in cases of mutual recursion among subprograms, with Ada's elaboration rules. We have developed techniques to deal with this and other such situations (e.g., detecting cycles in subprograms' interface relationships and "breaking" them with appropriate use of *with clauses* attached to program unit bodies).

The information in PIC/Ada that cannot be so readily translated into Ada is primarily the precise specification of the interface relationships among modules that is made possible by the *request* and *provide clauses*. Nevertheless, the translated Ada implementation, while not as precise as the PIC/Ada version, can at least be made to allow the desired references and, in some cases, deny the undesired ones. An Ada *with clause*, for example, can be created for a program unit specification derived from some module's specification submodule by gathering just the module names (i.e., names of other program units) found in the *request clauses* of that module's specification submodule. The resulting *with clause* then approximates the effect of those *request clauses*. The information that is sacrificed, of course, is the identification of the specific subset of entities within those external modules to which access is desired. A similar circumstance arises with Ada's concepts of *library unit* and *visible part*, which are refined by the environment's *provide clause*. In all of these situations, the environment's

analysis tools, operating on pre-translation descriptions, can be used to check the correctness of the relationships, and hence the imprecision resulting from the translation process does not actually diminish the value of the PIC language features.

5. Conclusion

The environment for programming-in-the-large described in this paper successfully addresses the software development concerns outlined in Section 1 through the careful design of its language features and support tools. In particular, the basic interface control, incompleteness, and physical-separation concepts underlying the language features are general enough and powerful enough to be applicable in every phase of the software lifecycle without presupposing any particular managerial, modularization, or version-control method. Moreover, the support tools can be used to provide rigorous analysis throughout the software development and maintenance process. As a result, the language features and associated tools provide a genuinely integrated approach to programming-in-the-large applicable throughout the software lifecycle.

While our Ada-based environment provides the fundamental capabilities necessary for programming-in-the-large, various extensions could be considered that might enrich its capabilities or facilitate their use by software developers. One such extension would be language features and tools supporting higher-level or more convenient descriptions of the relationships among modules. It would, for instance, be possible to provide shorthand notations for identifying groups of modules and/or entities when describing interface control relationships. These shorthand notations might be based on a facility for giving names to groups, or for identifying groups through a common attribute such as the name of a programmer [17], or

even for giving a more abstract semantic description such as input/output behavior [16]. Another extension would be additional analysis tools that could evaluate properties other than requisition and provision relationships, such as patterns of actual usage of entities in a software system. Those tools could, for example, check that a provided data object is assigned a value before it is ever read or determine that shared entities intended for mutually exclusive use are never referenced simultaneouly by multiple users. A final possible extension would be support for dynamic interface control mechanisms, whereby the requisition and/or provision relationships within a software system might change during the system's execution. In keeping with the static nature of Ada's declaration and visibility rules, the language features in PIC provide an entirely static interface control mechanism. The underlying framework of requisition and provision could, however, be extended to encompass a dynamic mechanism. Indeed, it is our belief that all of the extensions mentioned above are compatible with the basic approach taken in the PIC environment and that that environment comprises the primitive capabilities required for programming-in-the-large.

To evaluate our ideas and to demonstrate the language features and support tools, we are currently building a prototype implementation of the PIC environment. We believe that this prototype will be very important in demonstrating the added power furnished by our approach to modularization and interface control as well as showing its applicability throughout the software development and maintenance process. The prototype is being designed and built incrementally to give us an opportunity to use the tools constructed in earlier versions to help in the development of tools for later versions. Thus, developing the software for the prototype is serving as a significant and realistic initial test case for our ideas and is providing us with useful feedback. As the example in the appendix indicates, we have used the language features

described in this paper in designing the prototype.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] L.A. Clarke, J.C. Wileden, and A.L. Wolf, *Precise Interface Control: System Structure, Language Constructs, and Support Environment*, **Technical Report 83–26**, COINS Department, University of Massachusetts, Amherst, Massachusetts, August 1983.

[5] 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.

[6] **Ada Methodologies: Concepts and Requirements**, United States Department of Defense (Ada Joint Program Office), Washington, D.C., November 1982, Appearing in **Software Engineering Notes**, Vol. 8, No. 1, pp.33–50, January 1983.

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

[8] A. Evans, Jr. and K. J. Butler (eds.), *Diana Reference Manual (Revision 3)*, **Technical Report TL 83–4**, Tartan Laboratories Inc., Pittsburgh, Pennsylvania, February 1983.

[9] A.N. Habermann, *The Gandalf Research Project*, **Computer Science Research Review 1978–1979**, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1979.

[10] R.C. Holt and D.B. Wortman, *A Model For Implementing Euclid Modules and Prototypes*, **ACM Transactions on Programming Languages and Systems**, Vol. 4, No. 4, pp.552–562, October 1982.

[11] **Ada System Specification for Integrated Environment Type A**, Intermetrics, Inc., March 1981.

[12] 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.

[13] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, *Report on the Programming Language Euclid*, **Technical Report CSL–81–12**, Xerox PARC, Palo Alto, California, October 1981.

[14] B.W. Lampson and E.E. Schmidt, *Organizing Software in a Distributed Environment*, **Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems**, appearing in **SIGPLAN Notices**, Vol. 18, No. 6, pp.1–13, June 1983.

[15] 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.

[16] D.C. Luckham and F.W. von Henke, *An Overview of Anna, a Specification Language for Ada*, **Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments**, St. Paul, Minnesota, pp.116–127, October 1984.

[17] 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.

[18] 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.

[19] B.G. Moore and M. Chandrasekharan, *Tools for Maintaining Consistency in Large Programs Compiled in Parts*, **Proceedings of the International Telecommunications Conference**, July 1983.

[20] J.P. Privitera, *Ada Design Language for the Structured Design Methodology*, **Proceedings of the AdaTEC Conference on Ada**, Arlington, Virginia, pp.76–90, October, 1982.

[21] L. Robinson and O. Roubine, *SPECIAL—A Specification and Assertion Language*, **Stanford Research Institute Technical Report CSL–46**, January 1977.

[22] A. Rudmik and B.G. Moore, *An Efficient Separate Compilation Strategy for Very Large Programs*, **Proceedings of the SIGPLAN '82 Symposium on Compiler Construction**, appearing in **SIGPLAN Notices**, Vol. 17, No. 6, pp.301–307, June 1982.

[23] J.E. Sammet, D.W. Waugh, and R.W. Reiter, Jr., *PDL/Ada—A Design Language Based on Ada*, **Proceedings of ACM '81**, appearing in **Ada Letters**, Vol. 2, No. 3, pp.19–31, November–December, 1982.

[24] T.A. Standish and R.N. Taylor, *Arcturus: A Prototype Advanced Ada Programming Environment*, **Proceedings of the ACM/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**, appearing in **Software Engineering Notes**, Vol. 9, No. 3, pp.57–64, April 1984.

[25] 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.

[26] 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.

[27] N. Wirth, **Programming in MODULA-2** (second edition), Springer-Verlag, New York, 1983.

[28] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *A Formalism for Describing and Evaluating Visibility Control Mechanisms*, **Technical Report 83–34**, COINS Department, University of Massachusetts, Amherst, Massachusetts, October 1983.

A. Example

To illustrate the capabilities provided by the language features and support tools of the PIC environment, this appendix presents a simple, yet realistic, example showing the specification and analysis of an evolving system's modules during the high-level and low-level design phases. The example is drawn from actual development work on the prototype implementation of the environment.

In this example, two modules are being designed: a package providing low-level interface-analysis tools, **LowLevelAnalysisTools**, and a package providing general submodule processing tools, **ProcessingTools**. Both sets of tools are to operate on submodules through an abstract internal representation (attributed graphs) 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 [8], the internal representation for Ada programs, at the same time that compilers using Diana were being built at Intermetrics [25] and Softech [2].)

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 submodule is created for the specification of package **InternalRepresentation**. The specification stub submodule is initially created in graphical form (Figure 6). The developer can choose to perform further work on this submodule using the graphical representation or automatically translate it into a corresponding textual representation. Figure 7 shows a refined version of the submodule in textual form. The *used-by clause* appearing at the top of the submodule indicates the two

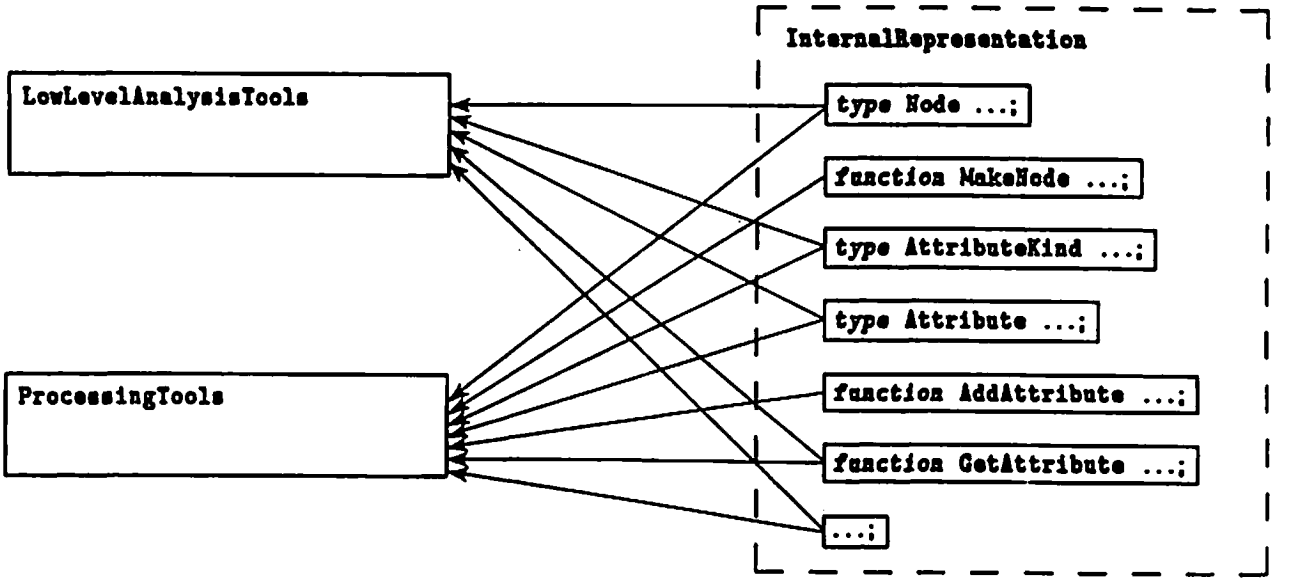

Figure 6: Initial Specification Stub Submodule of Internal Representation Package (Graphical Form).

users of the stub. The specification stub submodule indicates (a subset of) the entities the package is expected to provide. In particular, it defines a data type **Node** for representing entities and a function **MakeNode** for initializing such representations. The remaining entities defined in the stub submodule 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 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 **MakeNode** and the elements of type **AttributeKind**

```
package stub InternalRepresentation is
      used by LowLevelAnalysisTools, ProcessingTools;

   type Node is private;

   function MakeNode ( ... ) return Node
      provide to ProcessingTools;

   ...;

   type AttributeKind is ( NodeKind, ...,
          RequestedEntities, ProvidedEntities, ... );

   type Attribute ( AK : AttributeKind )
      is record
           case AK is
              when NodeKind           => ...;
              ...;
              when RequestedEntities => ...;
              when ProvidedEntities  => ...;
              ...;
           end case;
      end record;

   procedure AddAttribute ( N : in out Node; A : in Attribute )
      provide to ProcessingTools;

   function GetAttribute ( N : Node; AK : AttributeKind ) return Attribute;

   ...;

end InternalRepresentation;
```

Figure 7: Refined Specification Stub Submodule of Internal Representation Package (Textual Form).

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 **Node** is not yet known, the presence of the keyword **private** indicates that users will not be able to operate on **Node**'s representation. Moreover, the fact that only entities in package **ProcessingTools** can invoke the subprograms that create or update objects of type **Node** is specified by restricting the relevant subprograms to that package.

The first submodule to be submitted for checking with the specification stub submodule of package **InternalRepresentation** is the specification submodule of package **LowLevelAnalysisTools** (Figure 8). Appearing in this submodule are specifications for procedures realizing six basic analyses. 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 *request clause* that imports a number of entities from package **InternalRepresentation**. The list of requested entities and the parameter lists for the six procedures are only partially specified as indicated by the ellipses. Invoking the interface analysis tools at this point reveals that package **LowLevelAnalysisTools** requests an entity that is not available. Specifically, the common *request clause* contains the entity **AddAttribute** defined in package **InternalRepresentation** which has been restricted to package **ProcessingTools** (see figures 6 and 7). This is immediately evident from a (zoomed) graphical representation of the interface relationship between the two submodules (Figure 9), which when retrieved from the development library shows that there is a requisition arc (dotted arrow) without a matching provision arc (solid arrow). Assuming the error lies with the interface of **LowLevelAnalysisTools**, the inconsistency can

```
package LowLevelAnalysisTools is
      request InternalRepresentation.( Node, Attribute,
                     AttributeKind, GetAttribute, AddAttribute, ... );

   procedure InterfaceCheck ( SpecSubmodule1, SpecSubmodule2 :
                              in InternalRepresentation.Node; ... );

   procedure IntraModuleBodyCheck ( ... );

   procedure InterModuleBodyCheck ( ... );

   procedure WeakInterfaceCheck ( ... );

   procedure WeakInterModuleBodyCheck ( ... );

   procedure StubConsistencyCheck ( ... );

private
   function EntitiesOf ( Submodule :
                     InternalRepresentation.Node ) return ...;

   function Unavailable ( EntityRequested, Entity :
                     InternalRepresentation.Node ) return Boolean;

   function SemanticConflict ( EntityRequested, Entity :
                     InternalRepresentation.Node ) return Boolean;

   ...;  -- Other hidden utility entities

end LowLevelAnalysisTools;
```

Figure 8: Specification Submodule of Low-Level Analysis Tools Package.

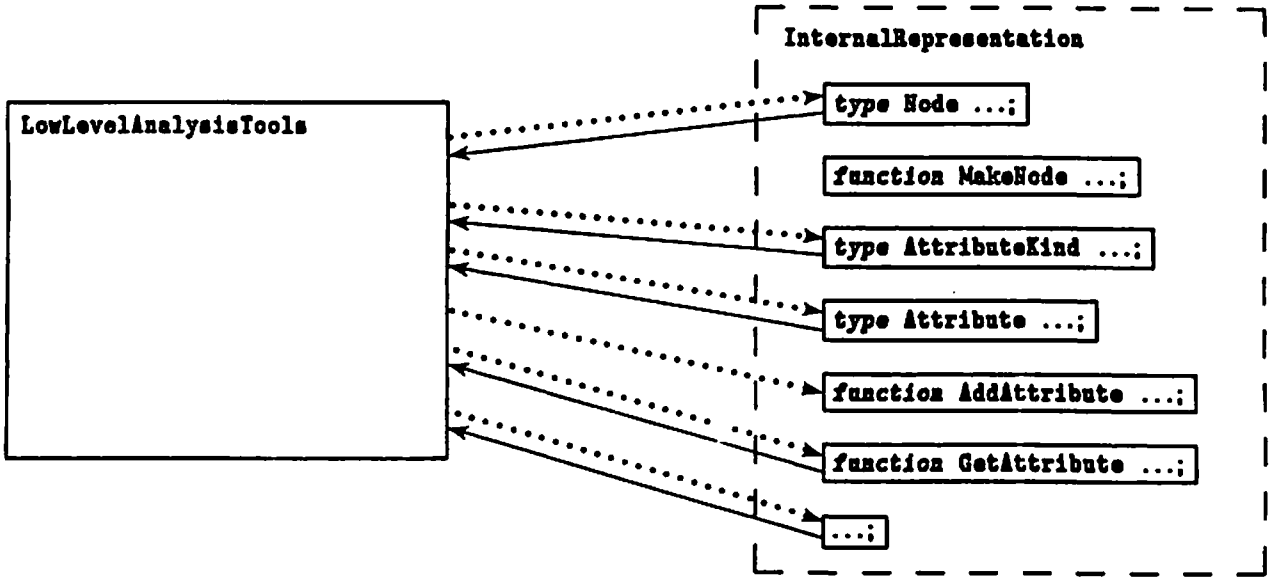

Figure 9: Erroneous Interface Relationship Between Internal Representation and Low-Level Analysis Tools Packages.

be rectified by appropriately editing either the graphical or textual representation of the specification submodule and then rechecking and replacing that submodule.

The next submodule submitted is the specification submodule for package **ProcessingTools** (Figure 10). Note that in addition to the entities imported from **InternalRepresentation** by the common *request clause* at the top of the package, certain other entities defined in **InternalRepresentation** that are used to create and update internal representations are requested 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 **GenerateSpec** can perform such operations. Invocation of the interface analysis tools at this stage of development would not reveal any inconsistencies between the specification submodule of **ProcessingTools** and the

```
package ProcessingTools is
      request InternalRepresentation.( Node, Attribute,
                     AttributeKind, GetAttribute, ... );

   procedure Recognize ( ... )
      request InternalRepresentation.( MakeNode,   -- node-update
                              AddAttribute, ... );  --   entities

   procedure Edit ( ... )
      request InternalRepresentation.( MakeNode,   -- node-update
                              AddAttribute, ... );  --   entities

   procedure Translate ( ... );

   procedure ProcessUpdate ( ... );

   procedure GenerateSpec ( ... )
      request InternalRepresentation.( MakeNode,   -- node update
                              AddAttribute, ... );  --   entities

   function GenerateView ( ... ) return ...;

end ProcessingTools;
```

Figure 10: Specification Submodule of Processing Tools Package.

specification stub submodule of **InternalRepresentation**.

Independent of the development of the rest of the system, low-level design of the body submodule of package **LowLevelAnalysisTools** can begin. Figure 11 shows this submodule 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 specification submodule, whether the entities defined in the second specification submodule referenced by E are both provided to E and requested by E in semantically consistent ways.

With the corresponding specification submodule of the package and the specification stub submodule of package **InternalRepresentation** already present, a substantial amount of consistency checking can be performed on the body submodule of package **LowLevelAnalysisTools** even at this early stage. Invoking the interface analysis tools to analyze the consistency between the specification submodule of package **LowLevelAnalysisTools** and its body submodule does not reveal any errors. On the other hand, invoking these tools to analyze the consistency between the body submodule and the specification stub submodule 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 submodule is in error. Let us assume it is decided that the body submodule is incorrect. We will then assume further that the parameter list is appropriately edited, and finally that the submodule is resubmitted and is now found to be consistent.

Eventually, an official version of package **InternalRepresentation** is delivered. In general, the specification submodule 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

```
package body LowLevelAnalysisTools is
   function EntitiesOf ( Submodule :
         InternalRepresentation.Node ) return ... is
      begin ... end EntitiesOf;
   function Unavailable ( EntityRequested, Entity :
         InternalRepresentation.Node ) return Boolean is
      begin ... end Unavailable;
   function SemanticConflict ( EntityRequested, Entity :
         InternalRepresentation.Node ) return Boolean is
      begin ... end SemanticConflict;

   ...;  -- Other utilities (e.g., RecordInterfaceError)

   procedure InterfaceCheck ( SpecSubmodule1, SpecSubmodule2 :
                  in InternalRepresentation.Node; ... ) is
         EntityRequested : InternalRepresentation.Node;
         RequestList     : ...;
         Entity          : InternalRepresentation.Node;
         ...;  -- Other local objects and types
         use InternalRepresentation;
   begin
    foreach Entity in EntitiesOf ( SpecSubmodule1 ) loop
     RequestList := GetAttribute ( RequestedEntities, Entity );
     foreach EntityRequested in RequestList loop
      if ( EntityRequested.Parent = SpecSubmodule2 ) then
       if ( Unavailable ( EntityRequested, Entity ) ) then
          RecordInterfaceError ( ... );
       elsif ( SemanticConflict ( EntityRequested, Entity ) ) then
          RecordSemanticError ( ... );
       end if;
      end if;
     end loop;
    end loop;
   end InterfaceCheck;

  ...;  -- Bodies of other low-level analysis procedures
end LowLevelAnalysisTools;
```

Figure 11: Body Submodule of Low-Level Analysis Tools Package

foster a high degree of interface control, the specification submodule must be augmented to include any desired restrictions on its provided entities. Significantly, such augmentation, which can be done in a straightforward manner using either the graphical or textual representation, does not affect the implementation of the package since restrictions on provision exclusively involve the module's interface. Returning to the example, the appearance of the (augmented) official specification submodule of package `InternalRepresentation` makes the specification stub submodule obsolete. All checking can now be performed—with greater confidence—against the true specification submodule. Such checking can be expedited by using the already-checked stub submodule, rather than the other submodules, as a basis for most of the checking of the newly introduced specification submodule.