# ADA®-BASED SUPPORT FOR PROGRAMMING-IN-THE-LARGE

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

COINS Technical Report 85-32
October 1985

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

*Appeared in IEEE Software, Vol. 2, No. 2, March 1985, pp.58-71.*

*A preliminary version appeared in the Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments, St. Paul, Minnesota, October 15-18, 1984.*

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.

# Ada-Based Support for Programming-in-the-Large

**Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden**
**University of Massachusetts**

**Specialized language features and an integrated set of tools can help developers of a large software system describe, analyze, organize, and manage its many modules.**

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 50 lines of code, then the number of modules in such systems will exceed 10,000! 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."[1] Thus, developers and maintainers of large Ada systems will require tools beyond the syntax-directed editors, compilers, debuggers and so on needed for "programming-in-the-small."[2-4] 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."

## 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 those modules contain, 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 properly support modularization and interface control, an environment for programming-in-the-large should address a number of software development concerns. These include:

- Life-cycle support. Support for modularization and interface control is of primary importance during every phase of the software life cycle. 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 ensuring that those interactions remain correct and consistent is primary 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 use entities made available by other modules. The environment should provide means for precisely specifying these relationships among modules. Ideally, such specifications should include both the producer and consumer points of view.
- Analysis support. For large systems, the ability to specify relationships among modules is of limited value without tools to aid



0740-7459/85/0100/0058$01.00 © 1985 IEEE

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 analysis during the earliest stages of development and continue through maintenance.

- Version control. Many systems must be configured for a number of different operating environments (for example, operating systems, machines, and peripherals) and developers often must maintain running versions of a system while developing new, extended versions. Thus, an environment for programming-in-the-large must facilitate description and configuration of these system families.
- Managerial support. Usually, teams of individuals produce large software systems; typically, different team members develop different modules. Thus, programming-in-the-large is a management activity involving such matters 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.[5] The environment should be general enough and powerful enough to work with any of these methods, since none can realistically be expected to be appropriate for all applications.

We are developing an environment, based upon Ada, for programming-in-the-large. The environment is to provide capabilities meeting the requirements outlined above. This environment relies upon a small number of specialized language features and an integrated set of tools; both have been carefully tailored to support incremental development and apply to all phases in the software life cycle.

We believe that Ada already substantially supports flexible representation of module decomposition, but is very limited in its support of 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 *precise interface control* is the central concern. The PIC language features, combined with modularization capabilities such as those in Ada, result in a uniform framework facilitating programming-in-the-large. This framework supports many of the major proposed design methods, 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. Though the environment currently does not contain specialized support for version control, it is compatible with a number of recent proposals[6,7] for version-control mechanisms.

**This framework supports many of the major proposed design methods, including those most closely associated with Ada.**

The PIC environment is intended to significantly extend the capabilities of Ada development systems.[3,4,8] All modularization and interface-control decisions made throughout the development and maintenance process are to be recorded in languages that incorporate the PIC language features. These decisions will then be organized, managed, and analyzed with the aid of the support tools. This implies that during pre-implementation phases the language features and support tools will 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 of support tools applied to a variety of PIC-oriented languages: PIC/PDL, an Ada-like textual pre-implementation language; PIC/Ada, a textual implementation language that is an enhanced version of Ada; and PIC/Graphics, a graphical language.

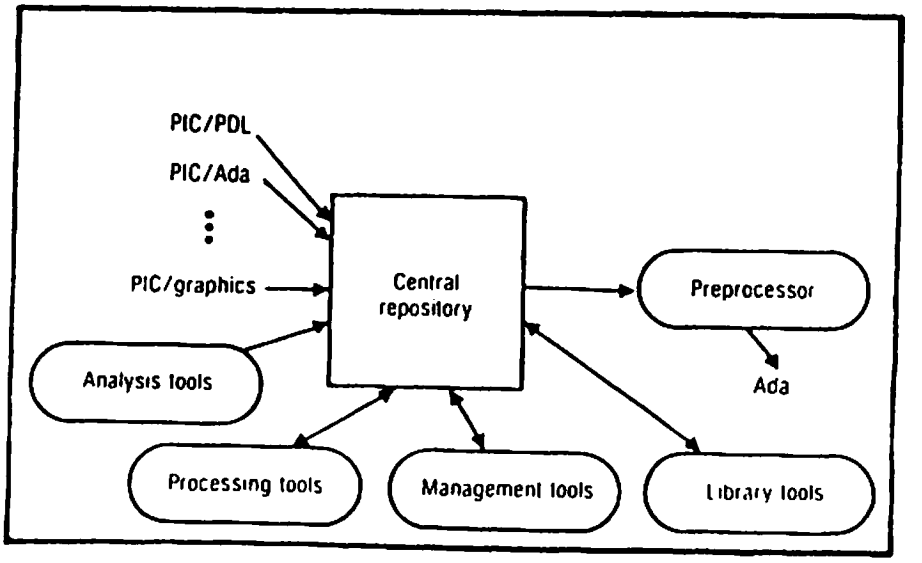


**Figure 1. Conceptual organization of the PIC environment.**

Compilation of PIC/Ada will be a two-step process. The first is essentially a preprocessing step, in which the interface control information is incorporated, as far as possible, into standard Ada code. The second is a normal Ada compilation. 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 made to the Ada implementation, including those resulting from maintenance activities, will be performed in the PIC-oriented languages and 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. Thus, it represents a genuinely integrated approach to programming-in-the-large.

This article focuses on the PIC language features—the basic concepts underlying them and examples of their use—but also briefly describes the support the tools will provide and possible extensions to the environment.

## Background

A general view of interface control—which offers a richer conceptual foundation than views based solely on the 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 refer to or use that entity in declarations and statements. Requisition of access occurs when an entity implicitly or explicitly requests the right to refer to some set of entities. Thus, in most programming languages a subprogram typically requests access to itself and any locally declared entities, as well as to certain nonlocal entities. Provision of access occurs when an entity implicitly or explicitly offers some set of entities the right to refer to that entity. Again, in most programming languages a subprogram typically provides access (that is, the right to invoke that subprogram) to itself and, in languages that support nesting, to 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 $E_i$. (In the remainder of this article, the word "access" is dropped from certain phrases involving the terms "requisition" and "provision" when the meaning is clear. 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.)

**Precise interface control would permit requisition and provision of exactly those accesses desired in a system and disallow others.**

**Specifying requisition and provision; typical strategies.** An interface control mechanism is the means for specifying requisition and provision. Programming languages have, historically, differed in their 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 requested by an entity is always also provided to that entity and vice versa. In the designs of more recent languages, particularly those intended for the construction of large and complex software systems (such as the implementation languages Ada,[9] GTEL Pascal,[10] Mesa,[11] and Modula-2,[12] the program specification language Special,[13] various program design languages based on Ada,[14,15] and the module interconnection languages MIL75,[1] C/Mesa,[11] and Intercol[6]), 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 greater or lesser degrees, on the concepts of *encapsulation* and *explicit import/export control* both to describe 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 groups 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.

*Ada.* In Ada, the encapsulation construct is the *package* and import/export control can be effected through a combination of features, including *with clauses*, *visible* and *private parts*, and *nesting*. Unfortunately, neither Ada nor any of the languages mentioned above supports precise and flexible control over both the accesses an entity can request and the accesses an entity can provide.[16,17] For instance, Ada's *with clause* only permits 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 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 our treatment of language features below.

**The value of precision.** We contend that precise interface control mechanisms are of great potential value to developers and maintainers of large

software systems.[18] Such precision would permit requisition and provision of exactly those accesses desired in a system and disallow others. In addition, support for both precise requisition and precise provision encourages redundancy, which in turn can facilitate more rigorous analysis of the interface relationships of a system's components. For example, with this approach it is possible to formulate complementary descriptions of exactly how two modules are supposed to interact, giving one description from the perspective of each of the modules, and then analyze those interactions by checking the two descriptions for consistency.

The language features and support tools discussed below exploit the concepts outlined here to build upon Ada's basic encapsulation and import/export concepts. Thereby, they provide mechanisms capable of describing, analyzing, and managing the interface control aspects of large systems precisely and flexibly. While this article focuses on use of the mechanisms in conjunction with Ada, the basic language features and tools could be used with most modern programming languages.

## Language features

The language features of the PIC environment have two aspects. First, they provide a system structure that strictly separates the interface control information from the algorithmic details of how a module uses that information locally. 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; the interface control component of a system can be viewed as a description in this language.

**Packages and subprograms.** The environment recognizes two kinds of modules: *packages* and *subprograms*, which correspond to their Ada namesakes. (To simplify the presentation, Ada tasks and generics are not considered in this article.) 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 the package encapsulates; a subprogram's specification submodule simply describes the information needed to invoke the subprogram. In 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 that realize the module. During the pre-implementation phases, the body takes the form of an Ada-based PDL description; in later phases, it consists 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 with linked lists. The package provides a type for print jobs, Job, a type for print queues, Queue, and two subprograms, Enqueue and Dequeue. Enqueue and Dequeue realize the abstract operations of adding and removing a job from a print queue. A more realistic example, showing the specification and analysis of a system during the high-level and low-level design phases, appears in the appendix. That example arises from our initial work on the prototype PIC environment we are developing.

**The specification submodule.** In this notation, a specification submodule is essentially an Ada program unit specification plus a small number of powerful features for enhancing interface control. The *request clause* at the top of the submodule in Figure 2, for instance, indicates that all the entities in package PrintQueue request access to the entities ListElement and List, defined in package LinkedList. Only procedure Enqueue can refer to the entity Append, also defined in package LinkedList, since Enqueue is the only entity in PrintQueue with an attached *request clause* mentioning Append. Similarly, the *request clauses* attached to procedures Dequeue and Util indicate that only these subprograms may refer to the entities Delete and Statistics, respectively, defined in package LinkedList. (Although it is not shown in this article, a complete subprogram header, including the subprogram's name and formal parameters, can be given in a *request clause* or a *provide clause* to

```
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.**

clarify requested and provided access to an overloaded subprogram.)

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 requisitions 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. (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.) Both the provided and hidden entities are available to all other entities in the defining package, but only the provided entities are available outside the package. In Ada, provision is controlled on an all-or-nothing basis; access to an entity is provided to either every module (in a given scope) or to no module, and so the entity is hidden. While these two extremes are useful (as in describing the 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.[19]

Therefore, our notation extends Ada by including the *provide clause*, which can be appended to any of a package's provided entities 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." (This choice was made 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.) 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 is 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.

**Distinction between name and representation.** Another aspect of our approach 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 can be associated with two *provide clauses*, one referring to the provision of the name, the other to the provision of the representation. Access to the name of the type is, of course, necessary for any use of the type. Therefore, a *provide clause* associated with the representation is a restriction on the representation beyond the restrictions inherited from the name. As in Ada, the keyword private, in place of the representation, denotes the case of a type's representation being completely hidden within the defining package. 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

```
(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 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.**

entities, to instead attach the clause provide to none to the representation in the type definition.

As shown in Figure 3, six basic levels of control result from distinguishing between the names and representations of types. This allows a high degree of flexibility in controlling the use of a type definition. In contrast, Ada provides the first and third levels in Figure 3. (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.) Associating two *provide clauses* with a type definition allows abstract data types to be easily defined and solves the problem of sharing the representation of an abstract type among different modules.[20] The print queue example illustrates this sharing; here, access to the names of the types Job and Queue are provided without restriction, but provision of access to their representation is limited to the defining package PrintQueue and module Reorder.

**The incompleteness construct.** Figure 2 suggests another aspect of the language features, their applicability to high-level, incomplete descriptions of a system's components and their interactions. The *incompleteness construct*, denoted with an ellipsis, is useful for explicitly indicating where details to 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. Used in conjunction with such constructs, the language features are well suited to expressing modularization and interface properties during early stages of a system's development.

**The specification stub submodule.** In addition to specification and body submodules, the language features include a third kind of submodule, the *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 complete, 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 contains only 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. It only needs to describe what that module is providing to the modules in the requesting group. A number of different specification stub submodules of a module can exist, to accommodate various uses different development groups might have for that module. 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.

Figure 4 shows two examples of specification stub submodules of a linked list package. 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. 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, it can be used for processing instead of the stub.

```
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.**

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. The appendix further illustrates use of specification stub submodules.

**Benefits of the PIC language features.** The PIC language features have a number of benefits; one is improved readability. Language features explicitly and clearly state which accesses can be requested and provided; we contend that these features enhance the readability of software and thus make it easier to discern the relationships among the modules. This in turn makes systems easier to change and therefore easier to develop and maintain.

Other benefits accrue from consolidating interface control information into the specification submodules and separating a module's interface specification from its body. While languages such as Ada, Mesa, and Modula-2 support "specifications" of modules separate from their bodies, these specifications do not completely define the interfaces to modules. In Ada, for instance, a body might import entities by using an attached *with clause*. In our approach, the *request clauses*, which can only appear in a specification submodule, completely constrain the external entities to which a module's body can refer. Because of this complete separation of concerns, the language features constitute a genuine module interconnection language; specification and specification stub submodules written in this language fully describe the interface control component of a system. By en-

forcing 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 incremental development (that is, 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—it permits more meaningful and detailed interface consistency analysis to be performed and allows this analysis to be done early and throughout the software life cycle. In particular, the interface control component of a system can be created and analyzed separately from the bodies (implementations) of the modules in that system. In fact, interface control components need be combined with the bodies only to facilitate further analysis or to support separate compilation. Moreover, the specification submodule of a given module can be associated with more than one version of the body submodule of that module; the actual implementation of the module can be chosen as late as link time. Support for incremental development is further enhanced by the language features' 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. That method enforces bottom-up development, primarily for code-generation reasons.

With PIC's separate interface control component, managerial control over modularization and module interfaces reduces to control over creation and modification of specification and specification stub submodules. Centralized control can be achieved by permitting only a project leader to create or modify these submodules. Under a more decentralized discipline, implementors can construct specification stub submodules of the modules they expect to use. A project leader can then decide (or negotiate!) the final form of the specification submodule after reviewing those stubs. The language features also support an autonomous discipline for cases in which managerial control is not desired, since implementors of modules can assume the role of project leader and construct their own specification submodules.

**The language incorporates capabilities distilled from many previous attempts. The resulting framework is relatively simple and straightforward, yet surpasses previous attempts.**

Finally, the separation of concerns supported by the module interconnection language facilitates information hiding, because a developer working on a module that refers to entities from another module only needs to see the specifications of the provided entities of the referred module, and because 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 some, many existing specification, design, programming, and module interconnection languages support some of the desired capabilities, but 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 previous attempts by supporting precise interface control as well as the comprehensive collection of benefits outlined above.

## Support tools

Despite the simplicity of the language features, development of the proper interface relationships for large software systems remains a complex task. An integrated toolset, consisting of analysis, library, management, and general processing tools would facilitate such development and assure consistency of the software system.

**Analysis tools.** The features a language provides 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 relatively straightforward, largely because those systems are monolithic and the controls they provide are quite limited. More recent nested languages, such as Modula-2 and Euclid,[21] furnish additional interface control features in an attempt to compensate for the inadequate controls of nesting. Some new nested languages, such as Ada and GTEL Pascal, supply still other features 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.[22,23] The absence of nesting in languages such as Gypsy[24] and CLU,[25] which also support incremental development, allows for simpler, but no less powerful, analysis techniques.

Our approach to interface control has its own 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 re-examination of the traditional meaning of *consistency* in interface relationships. Third, we want the approach to be applicable and integrated across the phases in the software life cycle; there-

fore, the analysis tools must be flexible enough to permit their operation upon distinctly different forms of representation. We now consider each of these three points in more detail.

*Expressive power.* We have identified a basic set of analyses that exploits the expressive power of the language features. If analysis is viewed as pair-wise comparison of submodules, two distinct classes of analysis arise: *Intramodule analyses* focus on the interface relationships between two submodules of the same module; *intermodule analyses* focus on the interface relationships between two submodules of different modules. An example of an intramodule analysis is one that compares a specification submodule to its corresponding body submodule, in order to check, among other things, that the body refers only to those external entities the specification requested. An example of an intermodule analysis is one that compares two specification submodules of different modules, in order to check, among other things, that the entities defined in the second module (to which the first module requests access) are in fact specified as provided by the second to the first. A rigorous evaluation of the consistency of the interface relationships results from composing various basic analyses in these two classes.

*Incompleteness.* Analysis techniques for the early stages of the software life cycle 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, a submodule is incomplete (for example, contains 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 to be incorrect.

Consider the example in Figure 5. A package Pac provides objects Obj1 and Obj2. Access to Obj1 is provided without restriction, while provision of access to Obj2 is limited through an attached *provide clause.* 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. (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 can be updated at any time. The presence of PDL constructs, however, gives the support tools explicit information, which they can exploit, regarding incompleteness.) Thus, we can assume Obj2 might be provided to entities in addition to Proc1. 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 specification submodule of procedure Proc2 indicates that Proc2 requests access to entities Obj1 and Obj2 of package Pac. But unlike that case, Pac does not explicitly provide Proc2 with access to Obj2, since Proc2 does not appear in the provide clause attached to Obj2. Under our definition, however, Proc2's interface is still considered 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 between the interfaces can be shown to exist.

```
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.**

Clearly, the consistency of Pac and Proc1 and the consistency of Pac and Proc2 differ qualitatively. The fact that consistency between two submodules depends only upon the consistency of those portions of the submodules that actually interact leads to the definition of two levels of consistency between submodules. 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 relevant to their interaction are complete. Two submodules are said to be *conditionally consistent* if (1) holds but (2) does not. In the example in Figure 5, Pac and Proc1 are consistent, but Pac and Proc2 are only conditionally consistent.

*Applicable and integrated.* Our goal of providing analysis support that is applicable and integrated across a range of software life-cycle phases is closely related to support for incompleteness. Indeed, once the handling of incompleteness is incorporated into the basic analysis techniques, the uniform application of these techniques to descriptions for different software life-cycle 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.[17] 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 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.

**Library tools.** It is common to break large software systems 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; hence, the opportunity for discovering interface errors arose at link time.

More recently, the concept of the *program library* has emerged.[9,25] Through the program library, the compiler can incrementally perform the interface checking the linker formerly did en masse. It does this by saving, within the program library and 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 large software systems involving numerous developers, the program library as a simple repository for the compilation information of an entire program is not an adequate tool. Incremental development involves more than incremental compilation; it involves incremental analysis during all phases of the software system's development. Moreover, projects involving many people require separate work areas, for individuals and for various working groups. We have found that what is truly required to support incremental development in such a setting is a synthesis of the capabilities of a program library and those of an operating system's file manager. We call this synthesis a *development library*.

*The development library.* The development library resembles a program library, in that it maintains information about the submodules it contains. Beyond compiled code and interface information, the development library can store various forms of the source code. More important, the development library maintains the results of analyses of its submodules. As a further generalization of the program library concept, a development library can even contain other development libraries; here, 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 can be a member of more than one directory. Incorporating this file-system model into the structure of the development library makes it possible to conceive of using multiple libraries in the development of a system and sharing information, particularly interface and analysis information, among those libraries in a fairly general way.

**The cost of sharing information among development libraries is added complexity in the designs of the analysis and library tools.**

The cost of sharing information among development libraries 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, can affect other libraries that compose the system; the analysis and library tools are thus responsible for deciding where and how to propagate those effects. However, this sharing of interface and analysis information facilitates sharing and reuse of software in ways not possible when the sharing is only at the level of source code files; this offsets the added complexity.

**Management tools.** One of the necessary capabilities of an environment for programming-in-the-large is support for managerial control. It must have management tools with mechanisms to control establishment and modification of a system's interface relationships. Since in PIC the specification and specification stub submodules completely determine those relationships, the management tools can operate by controlling programmer access to these submodules. The management tools, therefore, cooperate closely with the library tools.

These tools should enforce any managerial discipline 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 submodules, but only the project leader should be permitted to enter or replace the "official" specification submodules. An autonomous managerial discipline would permit all project members to enter or replace specification submodules for the modules they are developing or maintaining in the development library.

**Processing tools.** A number of general processing tools must be available in the PIC environment for 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. Like the analysis tools, these tools are designed to use the consistent internal representation and handle incompleteness appropriately. Thus, they too can be uniformly applied throughout the software development and maintenance process. The common internal

representation also facilitates easy movement from the graphical representations of a system to corresponding textual representations and vice versa. It permits further development of the system to be recorded through refinements in either or both representations.

*PIC/Ada preprocessor.* One very important processing tool is the PIC/Ada preprocessor. The design of the language features makes the translation of PIC/Ada into Ada relatively straightforward. For instance the specification submodule closely resembles an Ada program unit specification.

Certain translation situations, however, require special care. For example, under our approach, requisition of access can only be described in specification submodules. In cases of mutual recursion among subprograms, this leads to a conflict with Ada's elaboration rules. We have developed techniques to deal with this and other such situations—such as cases in which cycles are detected in subprograms' interface relationships and are "broken" with appropriate use of *with clauses* attached to program unit bodies.

The information in PIC/Ada not so readily translated into Ada is, primarily, the precise specification, made possible by *request* and *provide clauses*, of the interface relationships among modules. 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 a program unit specification derived from some module's specification submodule, for example, can be created by gathering just the module names (that is, names of other program units) found in the *request clauses* of that module's specification submodule. The resulting *with clause* approximates the effect of those *request clauses*. The information 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 the environment's *provide clause* refines.

In all these situations, the environment's analysis tools, operating on pre-translation descriptions, can be used to check the correctness of the relationships, so the imprecision resulting from the translation process does not actually diminish the value of the PIC language features.

**The basic language features are general enough, powerful enough, and integrated enough to apply to every phase of the software life cycle.**

The PIC environment for programming-in-the-large successfully addresses the software development concerns outlined in the beginning of this article. It does so through the careful design of its language features and support tools. The basic interface control, incompleteness, and physical-separation concepts underlying the language features are general enough, powerful enough, and integrated enough to apply to every phase of the software life cycle without presupposing any particular management, modularization, or version-control method. Moreover, the support tools can be used to provide rigorous analysis throughout the software development and maintenance process.

**Possible extensions.** While our Ada-based environment provides the fundamental capabilities necessary for programming-in-the-large, various extensions might enrich its capabilities or facilitate their use.

Among them are language features and tools supporting higher-level or more convenient descriptions of the relationships among modules. These would make it possible, for example, 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 naming groups, for identifying groups through a common attribute (such as the name of a programmer[19]), or even for giving a more abstract semantic description (such as input/output behavior[26]).

Additional analysis tools could evaluate properties other than requisition and provision relationships, such as patterns of actual use of entities in a software system. These tools could, for example, check to see that a provided data object is assigned a value before it is ever read, or ensure that multiple users never simultaneously reference shared entities intended for mutually exclusive use.

Another possible extension is support for dynamic interface control mechanisms. This would allow requisition and/or provision relationships within a software system to change during the system's execution. In keeping with the static nature of Ada's declaration and visibility rules, PIC language features provide an entirely static interface control mechanism. The underlying framework of requisition and provision could, however, be extended to encompass a dynamic mechanism. Indeed, we believe that all the extensions mentioned above are compatible with the basic approach taken in the PIC environment, and that the environment comprises the primitive capabilities required for programming-in-the-large.

**Prototype.** To evaluate our ideas and to demonstrate the language features and support tools, we are currently building a prototype implementation of the PIC environment. This prototype will be very important in demonstrating the power our approach adds to modularization and interface control and in showing its applicability throughout software development and maintenance. The prototype is being designed and built incrementally, to give us an opportunity to use the tools constructed in earlier versions and to help in the development of tools for later versions. Thus, development of the software for the prototype is a significant and realistic initial test case for our ideas and is pro-

viding useful feedback. As the example in the appendix indicates, we have used the language features described in this article in designing the prototype. □

## Acknowledgment

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

## References

1. F. DeRemer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Trans. Software Engineering*, Vol. SE-2, No. 2, June 1976, pp. 80-86.
2. A. N. Habermann, "The Gandalf Research Project," *Computer Science Research Review 1978-1979*, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pa., 1979.
3. "Ada System Specification for Integrated Environment Type A," Intermetrics, Inc., Cambridge, Mass., Mar. 1981.
4. W. Babich, L. Weissman, and M. Wolfe, "Design Considerations in Language Processing Tools for Ada," *Proc. 6th Int'l Conf. Software Engineering*, Tokyo, Japan, Sept. 1982, pp. 40-47.
5. "Ada Methodologies: Concepts and Requirements," US Dept. of Defense Ada Joint Program Office, Washington, D.C., Nov. 1982, in *Software Engineering Notes*, Vol. 8, No. 1, Jan. 1983, pp. 33-50.
6. W. F. Tichy, "Software Development Control Based on Module Interconnection," *Proc. 4th Int'l Conf. Software Engineering*, Munich, Sept. 1979, pp. 29-41.
7. B. W. Lampson and E. E. Schmidt, "Organizing Software in a Distributed Environment," *Proc. Sigplan 83 Symp. Programming Language Issues in Software Systems*, in *Sigplan Notices*, Vol. 18, No. 6, June 1983, pp. 1-13.
8. T. A. Standish and R. N. Taylor, "Arcturus: A Prototype Advanced Ada Programming Environment," *Proc. ACM/Sigplan Software Engineering Symp. Practical Software Development Environments*, in *Software Engineering Notes*, Vol. 9, No. 3, Apr 1984, pp. 57-64.
9. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, US Dept. of Defense, Washington, D.C., Jan. 1983.
10. A. Rudmik and B. G. Moore, "An Efficient Separate Compilation Strategy for Very Large Programs," *Proc. Sigplan 82 Symp. Compiler Construction*, in *Sigplan Notices*, Vol. 17, No. 6, June 1982, pp. 301-307.
11. J. G. Mitchell, W. Maybury, and R. Sweet, "Mesa Language Manual Version 5.0," technical report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif., Apr. 1979.
12. N. Wirth, *Programming in MODULA-2*, 2nd edition, Springer-Verlag, New York, 1983.
13. L. Robinson and O. Roubine, "SPECIAL—A Specification and Assertion Language," Stanford Research Institute technical report CSL-46, Palo Alto, Calif., Jan. 1977.
14. J. E. Sammet, D. W. Waugh, and R. W. Reiter, Jr., "PDL/Ada—A Design Language Based on Ada," *Proc. ACM 81*, in *Ada Letters*, Vol. 2, No. 3, Nov.-Dec. 1982, pp. 19-31.
15. J. P. Privitera, "Ada Design Language for the Structured Design Methodology," *Proc. AdaTEC Conf. Ada*, Oct. 1982, pp. 76-90.
16. L. A. Clarke, J. C. Wileden, and A. L. Wolf, "Nesting in Ada Programs is for the Birds," *Proc. ACM-Sigplan Symp. Ada Programming Language*, in *Sigplan Notices*, Vol. 15, No. 11, Nov. 1980, pp. 139-145.
17. A. L. Wolf, L. A. Clarke, and J. C. Wileden, "A Formalism for Describing and Evaluating Visibility Control Mechanisms," technical report 83-34, COINS Dept., University of Massachusetts, Amherst, Mass. Oct. 1983.
18. 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 Dept., University of Massachusetts, Amherst, Mass., Aug. 1983.
19. N. H. Minsky, "Locality in Software Systems," *Conf. Record 10th Ann. ACM Symp. Principles Programming Languages*, Jan. 1983, pp. 299-312.
20. C. H. A. Koster, "Visibility and Types," *Proc. Conf. Data: Abstraction, Definition and Structure*, in *Sigplan Notices*, Vol. 11, No. 2, Feb. 1976, pp. 179-190.
21. B. W. Lampson et al., "Report on the Programming Language Euclid," technical report CSL-81-12, Xerox Palo Alto Research Center, Palo Alto, Calif., Oct. 1981.
22. R. C. Holt and D. B. Wortman, "A Model For Implementing Euclid Modules and Prototypes," *ACM Trans. Programming Languages and Systems*, Vol. 4, No. 4, Oct. 1982, pp. 552-562.
23. B. G. Moore and M. Chandrasekharan, "Tools for Maintaining Consistency in Large Programs Compiled in Parts," *Proc. Int'l Telecommunications Conf.*, July 1983.
24. A. L. Ambler et al., "GYPSY: A Language for Specification and Implementation of Verifiable Programs," *Proc. ACM Conf. Language Design for Reliable Software*, in *Sigplan Notices*, Vol. 12, No. 3, Mar. 1977, pp. 1-10.
25. B. Liskov et al., "CLU Reference Manual," *Lecture Notes in Computer Science*, Vol. 114, Springer-Verlag, New York, 1981.
26. D. C. Luckham and F. W. von Henke, "An Overview of Anna, a Specification Language for Ada," *Proc. IEEE Computer Soc. 1984 Conf. Ada Applications and Environments*, Oct. 1984, pp. 116-127.
27. A. Evans, Jr., and K. J. Butler, eds., *Diana Reference Manual (Revision 3)*, technical report TL 83-4, Tartan Laboratories, Inc., Pittsburgh, Pa., Feb. 1983.
28. T. Taft, "Diana as an Internal Representation in an Ada-in-Ada Compiler," *Proc. AdaTEC Conf. Ada*, Oct. 1982, pp. 261-265.

## Appendix: An example specification and analysis of an evolving system's modules

To illustrate the capabilities of the PIC environment's language features and support tools, this appendix presents a simple, yet realistic, example of 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: LowLevelAnalysisTools, a package of low-level interface-analysis tools, and ProcessingTools, a package of general submodule processing tools. Both sets of tools are to operate on submodules through an abstract internal representation (attributed graphs) realized in a third package, InternalRepresentation. For present purposes, assume that package InternalRepresentation is undergoing parallel development at a separate site and has not yet been delivered. (This was, in fact, the situation in the development of compilers for Ada. Tartan Laboratories was developing

DIANA,[27] an internal representation for Ada programs, at the same time Intermetrics[28] and Softech[4] were building compilers that use DIANA.)

To allow development of the two tool packages to proceed while still gaining a degree of confidence in the interface consistency of the system, a specification stub submodule for package InternalRepresentation is created in graphical form (Figure A1). The developer can choose to work further on this submodule by using the graphical representation or can automatically translate it into a corresponding textual representation. Figure A2 shows a refined version of the submodule in textual form. The *used-by clause* at the top of the submodule names the two 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 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; type Attribute defines a variant structure representing actual attribute values. A value of the former type of object is a discriminant for the structure of an object of the latter type. Finally, subprograms PutAttribute and GetAttribute are used to store an attribute value and retrieve an attribute value, respectively.

Note 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 are deferred through use of the *incompleteness construct* (ellipsis), while the parameters to subprograms PutAttribute and GetAttribute are fully described. Note 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. Finally, note that only entities in package ProcessingTools can invoke the subprograms that create or update objects of type Node; this 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 A3). Specifications for procedures realizing six basic analyses appear in this submodule. 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*, which imports a number of entities

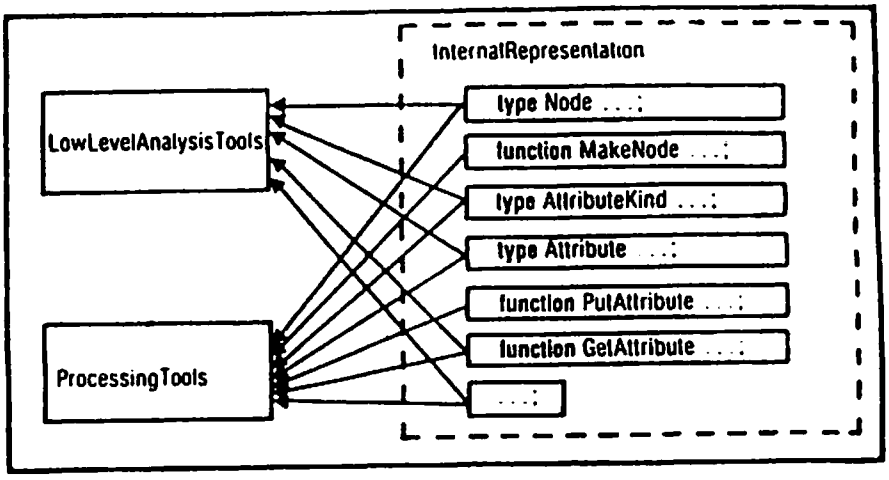


**Figure A1. Initial specification stub submodule of internal representation package (graphical form).**

```
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 PutAttribute ( N : in out Node; A : in Attribute )
      provide to ProcessingTools;
   function GetAttribute ( N : Node; AK : AttributedKind ) return Attribute;
   ...;
end InternalRepresentation;
```

**Figure A2. Refined specification stub submodule of internal representation package (textual form).**

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 PutAttribute defined in package InternalRepresentation, which has been restricted to package ProcessingTools (see Figures A1 and A2). This is immediately evident from Figure A4, a zoomed graphical representation of the interface relationship between the two submodules. When retrieved from the development library, this representation shows a requisition arc (dotted arrow) without a matching provision arc (solid arrow). Assuming the error lies with the interface of LowLevelAnalysisTools, the inconsistency can be rectified by appropriately editing either the graphic 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 A5). In addition to the

```
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 A3. Specification submodule of low-level analysis tools package.**

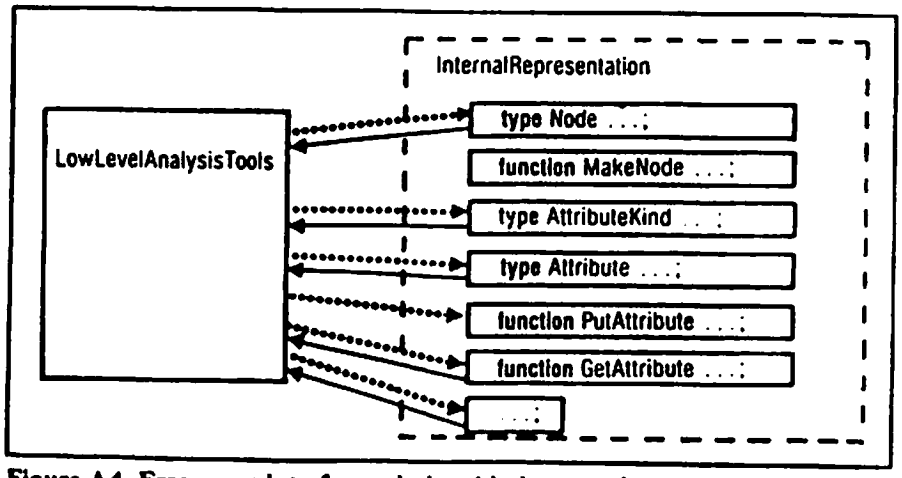


**Figure A4. Erroneous interface relationship between internal representation and low-level analysis tools packages.**

```
package ProcessingTools is
    request InternalRepresentation.( Node, Attribute,
                AttributeKind, GetAttribute, ... );
    procedure Recognize ( ... )
        request InternalRepresentation.( MakeNode, -- node-update
                PutAttribute, ... ); --    entities
    procedure Edit ( ... )
        request InternalRepresentation.( MakeNode, -- node-update
                PutAttribute, ... ); --    entities
    procedure Translate ( ... );
    procedure ProcessUpdate ( ... );
    procedure GenerateSpec ( ... )
        request InternalRepresentation.( MakeNode, -- node-update
                PutAttribute, ... ); --    entities
    function GenerateView ( ... ) return ...;
end ProcessingTools;
```

**Figure A5. Specification submodule of processing tools package.**

entities the common *request clause* at the top of the package imports from InternalRepresentation, a few of the packaged subprograms request certain other entities defined in InternalRepresentation. These are used to create and update internal representations. The effect is to limit those subprograms, within package ProcessingTools, that can alter an internal representation; only subprograms Recognize, Edit, and GenerateSpec can perform such operations. Invocation of the interface analysis tools at this stage of development would reveal no inconsistencies between the specification submodule of ProcessingTools and the specification stub submodule of InternalRepresentation.

Low-level design of the body submodule of package LowLevelAnalysisTools could begin at anytime. Figure A6 shows this submodule at a stage in which the basic algorithm of procedure InterfaceCheck has been specified by 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. Invocation of the interface analysis tools in order to analyze the consistency between the specification submodule of package LowLevelAnalysisTools and its body submodule reveals no errors. On the other hand, invocation of 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. The developer must then decide which submodule is in error. Assume it is decided that the body submodule is incorrect. Further assume that the parameter list is appropriately edited and that the submodule is resubmitted and is found consistent.

Eventually, an official version of package InternalRepresentation is delivered. In general, the specification submodule of a utility package such as InternalRepresentation (for example, 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 specification submodule must be augmented to include any desired restrictions on its provided entities. Significantly, such aug-

```
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 A6. Body submodule of low-level analysis tools package.**

mentation, which can be done straightforwardly by means of either the graphical or the textual representation, does not affect the implementation of the package, since restrictions on provision involve the module's interface exclusively.

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 previously checked stub submodule, rather than the other submodules, as a basis for most of the checking of the newly introduced specification submodule.

**Alexander L. Wolf** is a research assistant completing his PhD dissertation in the software development laboratory of the Computer and Information Science Department at the University of Massachusetts, Amherst. His primary research interests include software development environments, software engineering, programming languages, and distributed systems.

Wolf received a BA degree in computer science and geology from Queens College, City University of New York, in 1979 and an MS degree in computer science from the University of Massachusetts in 1982. Wolf is a member of the ACM and the IEEE Computer Society.

**Lori A. Clarke** is an asssociate professor in the Department of Computer and Information Science at the University of Massachusetts and director of the software development laboratory. Prior to joining the University of Massachusetts, she worked as a programmer for the University of Rochester School of Medicine and for the National Center for Atmospheric Research. Her primary research interests are in integrated software development environments, especially software testing and validation tools. She is a former IEEE Distinguished Visitor and ACM National Lecturer.

Clarke received the BA degree in mathematics from the University of Rochester and the PhD degree in computer science from the University of Colorado.

**Jack C. Wileden** is an associate professor in the Department of Computer and Information Science at the University of Massachusetts and associate director of the software development laboratory. His research interests center on integrated software development environments, especially development tools and techniques applicable to concurrent software systems.

Wileden received an AB degree in mathematics and MS and PhD degrees in computer and communications sciences, all from the University of Michigan. He is a member of ACM, IEEE, and Sigma Xi, and has served as an ACM National Lecturer.

The authors' address is Software Development Laboratory, Computer and Information Science Department, University of Massachusetts, Amherst, MA 01003.