# The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process

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

COINS Technical Report 86-51
September 1986
(Revised December 1986)

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

*A version of this report to appear in*
*IEEE Transactions on Software Engineering.*

---

This work supported in part by the NSF under grants DCR-83-18776 and DCR-84-08143.

## ABSTRACT

Despite the importance of describing and analyzing the relationships among a software system's components, most languages and development environments do not provide suitable support for these activities. While Ada and the various existing Ada environments offer some assistance, the capabilities they offer are inadequate for use in truly large and complex software development projects. To address these shortcomings, we are developing the AdaPIC toolset, which we envision as an important component of an Ada software development environment. The AdaPIC toolset is one particular instantiation, specifically adapted for use with Ada, of the more general collection of language features and analysis capabilities that constitute the PIC approach to describing and analyzing relationships among software system components. This toolset is being tailored to support an incremental approach to the interface control aspects of the software development process. Following a discussion of the interface control and incremental development concepts, this paper describes the AdaPIC toolset, concentrating on its analysis tools and support for incremental development and demonstrating how it contributes to the technology for developing large Ada software systems.

## 1. Introduction

In relatively small computer programs, the relationships among the program's parts may be few and simple enough that an experienced programmer can understand them with little help. While Ada [1] could certainly be used to build such programs, its primary use is intended to be in the construction of large, complex software systems. Throughout the development of software of this kind, the ability to accurately describe and analyze the relationships among system components is of crucial importance. For example, describing the major modules and their interactions is the primary concern of architectural or high-level design, while maintaining correct and consistent interfaces is an overriding concern during the implementation and maintenance of a software system. In most large systems, controlling component relationships is so difficult and demanding that it cannot realistically be performed without substantial automated aid.

Despite the importance of describing and analyzing component relationships, most languages and development environments do not provide suitable support for these activities. For example, Ada permits the relationships among a software system's components to be described with only limited accuracy [2,3], and Ada environments seldom provide tools capable of performing a thorough analysis of those relationships. Moreover, existing approaches to software system analysis, including the Ada model of compilation ordering, require either that the analysis of the relationships be delayed until the entire system is completed or that the system be developed and analyzed in a restricted fashion (e.g., strictly bottom-up).

To address these shortcomings, we have been investigating a range of issues concerning the activities of describing, enforcing, and analyzing the relationships among system components. We refer to these activities as *interface control* and, because we are particularly concerned with providing a framework that supports *precise* interface control, we refer to our approach as PIC.

The AdaPIC toolset, which we are currently developing, is one particular instantiation of the more general collection of language features and analysis capabilities that constitute the PIC approach. It is specifically adapted for use with Ada and, therefore, consists of Ada-specific versions of language features for precisely specifying interface control relationships and a collection of tools for analyzing and managing information about those relationships. The toolset is also being tailored to support an incremental approach to the interface control activities of the software development pro-

cess. We envision the AdaPIC toolset to be an important component of Ada software development environments.

Following a discussion of the interface control and incremental development concepts, this paper describes the AdaPIC toolset, concentrating on its analysis tools and support for incremental development and demonstrating how it contributes to the technology for developing large Ada software systems. Specifically, Section 3 provides an overview of the PIC language features and their use in an Ada-like design language, Section 4 outlines the analyses that can be performed on systems described using the language features, and Section 5 presents an example that illustrates how the PIC approach to interface control supports incremental development. The conclusion discusses the current status of a prototype AdaPIC toolset.

## 2. Fundamental PIC Concepts

One important aspect of the PIC approach is the ability to precisely describe the relationships among a software system's components. This ability, which is relevant throughout the lifetime of a software system, is provided by a small set of specialized language features. Appropriate dialects of these language features can be developed to make them compatible with a variety of languages, such as design or programming languages. In the sequel, our examples are given in terms of an Ada-based design language, which we refer to as PIC/ADL. Also included in the AdaPIC toolset are a PIC-enhanced dialect of Ada itself, known as PIC/Ada, and a graphical form of the PIC language features, suitable for use during either design or coding, called PIC/Graphics.

Another important aspect of the PIC approach is the ability to rigorously analyze relationships among a software system's components. Hence, the AdaPIC toolset includes a set of analysis capabilities applicable to descriptions phrased in any of the AdaPIC notations. This applicability of analysis capabilities throughout the development and maintenance process, including the ability to carry out analyses on arbitrarily incomplete descriptions, supports a highly interactive style of software development that we call incremental development.

Before proceeding to describe the PIC language features and analyses, we devote the remainder of this section to a discussion of the interface control and incremental development concepts that are fundamental to the PIC approach.

**Interface Control.** Interface control is concerned with describing and

limiting the interactions that can occur between the entities in different modules of a software system. Entities are named language elements such as objects, types, and subprograms. A module is either a subprogram unit, such as an Ada *procedure* or *function*, or an encapsulation unit, such as an Ada *package*. An encapsulation serves to group together entities such as objects, types, and subprograms. The interface control mechanism of a language is used to specify what (and sometimes how) entities within one module can be used by another module.

There have been a number of different interface control mechanisms proposed over the years. FORTRAN primarily used labelled and blank *common*. ALGOL60 introduced nested declarations. More recent languages, such as Ada, Clu [4], MODULA-2 [5], and Mesa [6], have predominantly used different variations of import/export lists, sometimes combined with the use of nested declarations; the various module interconnection languages, such as MIL75 [7], C/Mesa [6], or INTERCOL [8], have relied on essentially these same concepts. The particular mechanisms found in Ada include *with clauses*, for listing imported entities, and *package visible parts*, for listing exported entities. Ada also supports the use of *nesting* to control module interfaces.

We have demonstrated that the interface control mechanisms of these languages do not adequately describe all the interface relationships that need to be expressed [2,3] and have observed that, partially for this reason, they do not permit thorough interface control analysis.

The conceptual foundation for the PIC mechanism is provided by a general view of interface control that is richer than views based solely on traditional visibility concepts of declaration, scope, and binding. This view distinguishes 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 or statements. Requisition of access occurs when an entity (implicitly or explicitly) requests the right to refer to some set of entities. Provision of access occurs when an entity (implicitly or explicitly) offers, to some set of entities, the right to refer to that entity.[1] Given this view, an

[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".

interface control mechanism is simply a means for specifying requisition and provision.

The PIC language features improve upon the precision found in current mechanisms, including Ada's, and allow complementary, albeit redundant, descriptions of the interface control relationships among modules. In particular, the language features provide support for the explicit specification of both requisition and provision, and thus constitute a precise interface control mechanism. The analysis tools exploit this redundancy and precision by offering more detailed and revealing assessments of a system's interface relationships than has previously been possible, as we demonstrate later in this paper. These tools improve the software development process by providing information that can lead to the early detection and correction of errors, thereby reducing development costs and improving system reliability.

**Incremental Development.** Our work on PIC has been strongly influenced by a belief that software development environments must support incremental development. That is, environments should provide both languages and tools that facilitate the step-by-step manner in which large, complex software systems are most effectively developed. Such environments would allow developers to successively focus on particular aspects of the system, record their decisions about each aspect in the appropriate pre-implementation or implementation language, and then assess that step using suitable analysis tools. We have found that, at least with respect to interface control, support for incremental development implies support for:

- *Consistent abstractions.* The languages used throughout development should be based upon a consistent set of abstractions [9]. Although the syntax may vary greatly (e.g., from graphical icons to text) the basic underlying model should remain the same, thereby facilitating movement from one level of description to another and permitting the same or similar tools to be applicable.

- *Incremental analysis.* Developers should be able to perform meaningful analysis as they create the system. In the context of interface control, this means that as soon as interface control aspects of a module are specified, it should be possible to analyze whether that module is internally consistent as well as whether it is consistent with the already existing modules in that system.

- *Order-independent development.* Developers should be able to create modules and enter them into the system for analysis in any desired

order. In particular, the languages and tools in a software development environment should support top-down development, since this is generally recognized as a desirable development model. Approaches other than top-down should not be excluded, however, and thus it is important that an arbitrary submission order be adequately handled.

Both incremental analysis and order-independent development, in turn, depend heavily upon support for:

- *Incompleteness.* The interface control mechanism must make explicit provision for incomplete descriptions and the analysis tools should be capable of generating as much feedback as possible based on the provided information. These capabilities are essential for permitting analysis to be done as soon as developers start to formulate a description of the system, since at early stages in development many of the modules will not be specified and many of the specified modules will be incomplete.

The PIC approach has been specifically designed to facilitate incremental development. The language features to describe interface relationships have been based on the consistent abstractions *requisition* and *provision* of access, described above. Support for incomplete descriptions of modules is also provided. In particular, the PIC language features include a construct for explicitly indicating within a module's description that additional information is to be provided later. Moreover, the language features provide constructs for describing the pertinent interface control aspects of missing modules. These characteristics of the language features are complemented by a toolset design that permits flexible composition of analyses. Together, therefore, the PIC language features and analyses strongly support incremental analysis and order-independent development.

## 3. Overview of the PIC Language Features

The PIC language features take on specific, and sometimes different, forms in each of the AdaPIC language dialects. Within each dialect the features can be employed in conjunction with all the module kinds found in Ada or in an Ada-like notation, such as an Ada-based design language. Here, however, we present only the form of the PIC language features found in the PIC/ADL dialect. Moreover, we describe only those aspects of PIC/ADL applicable to one kind of module, namely Ada's encapsulation unit, the

package. A more detailed treatment of the PIC/ADL dialect of the PIC language features can be found in [10].

## 3.1 Basic Features

There are three aspects to the PIC language features. First, the features provide a system structure that results in systems that are collections of nest-free modules. Second, the requisition and provision of each entity in a system can be precisely specified. The language features used to capture these two aspects of entity visibility are the *request clause*, for specifying requisition, and the *provide clause*, for specifying provision. Third, the features provide a module structure that imposes a strict separation of interface control information from algorithmic detail. To realize this separation, a module consists of two distinct parts: a *specification submodule*, which contains a specification of the module's contents and all of the module's request and provide clauses, and a *body submodule*, which contains the actual code sections of the module. For pre-implementation languages, such as PIC/ADL, the body takes the form of a design-language description, while for implementation languages it consists of implementation-language code.

We illustrate the basic PIC language features via an example developed in PIC/ADL. Figure 1 shows the specification and body submodules of a package AutomaticTeller, which is one component of a hypothetical automatic bank-teller system. The subprograms in this package realize several customer-oriented and maintenance-oriented operations, including depositing and withdrawing funds and reporting on the cash available for withdrawal from the machine. Other modules in this system include Customer, ATMaintenance, and Officer, which use subprograms provided by AutomaticTeller in realizing three classes of user-interface capabilities. Also part of the system are the modules AccountManager, which provides facilities for manipulating customer accounts, PINManager, which provides facilities for manipulating the "personal identification numbers" that serve as passwords for customer accounts, and CommonCodes, which provides a collection of generally used error- and status-code entities.

A module's specification submodule completely determines both the entities requested by each entity in the module and the entities provided by the module. In PIC/ADL, the request clause is a list of entity names beginning with the reserved word **request**. The request clause appearing at the top of the specification submodule in Figure 1 indicates that access to the entities provided by package CommonCodes is requested for all the entities

```
package AutomaticTeller is
    request CommonCodes;

  procedure BeginSession ( ... )
    request PINManager.( PIN, Verify ),
            AccountManager.( Account, Verify );

  procedure Deposit ( ... )
    provide to Customer
    request AccountManager.Credit;

  procedure Withdraw ( ... )
    provide to Customer
    request AccountManager...;

  function RemainingCash ( ... ) return ...
    provide to ATMaintenance, Officer;

  function DepositsMade ( ... ) return ...
    provide to ATMaintenance, ...;

  ...;

end AutomaticTeller;


package body AutomaticTeller is
  procedure BeginSession ( ... ) is ... end BeginSession;
  procedure Deposit ( ... ) is ... end Deposit;
  procedure Withdraw ( ... ) is ... end Withdraw;
  function RemainingCash ( ... ) return ... is ... end RemainingCash;
  function DepositsMade ( ... ) return ... is ... end DepositsMade;

  ...;

end AutomaticTeller;
```

Figure 1: PIC/ADL Specification and Body Submodules of Package **AutomaticTeller.**

in package AutomaticTeller. The request clause attached to BeginSession, on the other hand, indicates that access to the entities PIN and Verify, defined in package PINManager, and to the entities Account and Verify, defined in package AccountManager, is requested only for BeginSession, since BeginSession is the only entity in AutomaticTeller that has an attached request clause mentioning those entities. Similarly, procedure Deposit is the only entity in the package that can refer to Credit, which is another entity defined in package AccountManager. The provide clause in PIC/ADL is a list of entity names beginning with the reserved words **provide to**. The provide clause appended to procedure Deposit indicates that Deposit is only provided to the entities of Customer, whereas the provide clause appended to function RemainingCash indicates that RemainingCash is only provided to the entities of ATMaintenance and Officer. The absence of a provide clause is interpreted to mean that access to the entity is provided to "all", which is the case for BeginSession.

The request clause is more precise and flexible than its counterparts in most other languages, including Ada's requisition construct, the *with clause*, in at least two ways. First, it does not necessarily import all the provided entities of a module but can import subsets of those entities. Second, a request clause can be attached to an individual entity of a module so that requisition by the entities within that module can be differentiated.

The provide clause is also more precise and flexible than its counterparts in most other languages. In Ada, for example, provision is controlled on an all-or-none basis; either access to an entity is provided to every module (in a given scope) by placing the entity in a package visible part, or it is provided to no module, and so the entity is hidden, by placing the entity in a package private part or body. While these two extremes are useful (for instance, 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 module needed to implement the trigonometric functions) and indeed specified the same way in PIC/ADL as in Ada, the intended provision of a particular entity often lies somewhere in between [11]. Hence, the PIC language features' provide clause may be appended to any of a package's (visible) entities in order to *selectively* limit their provision to specific non-local entities.

## 3.2 Features for Incomplete Descriptions

The PIC approach provides support for incomplete descriptions of a system's modules and their interactions through two features: the *incompleteness feature* and the *specification stub submodule*. They are intended to complement other features, which are not part of the PIC language framework, 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 those other features, the PIC language features are well suited for expressing modularization and interface properties during early stages of a system's development.

The incompleteness feature is used in place of names, declarations, or statements to explicitly indicate where details of a module that will be supplied later have been omitted from a description. In PIC/ADL, as in some other design languages, the incompleteness feature is denoted by an ellipsis. Ellipses appear in several places in the example of Figure 1, including the declaration lists of the specification and body submodules, the parameter and statement lists of the subprograms, and various request and provide clauses.

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 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 to document the required access and so facilitate analysis. 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. As a result of separate development activities, several different specification stub submodules of a module may exist to accommodate various intended uses of that module. The specification stub mechanism provides a means for the various groups of clients of a module to document these *views* of the module before the module is available.

Two examples of PIC/ADL specification stub submodules of module

PINManager are shown in figures 2 and 3. The two submodules partially describe the two, slightly different, views of PINManager that have been defined by the developers of AutomaticTeller and the developers of Officer, respectively. The *used-by clause* appearing in a PIC/ADL specification stub submodule is a readability aid intended to succinctly indicate the intended clients of that specification stub; the provide clauses still determine provision. Thus, for example, if the specification stub in Figure 2 were shared by another module, then that module's name would also appear in the used-by clause. A provide clause in a specification stub submodule that includes the keyword **only** indicates that the associated entity is provided *exclusively* to the listed modules; this special feature records the intention that no other specification stub of that module should provide the entity and that the specification submodule (when it becomes available) should provide the entity only to the listed modules.

As shown in Section 5, the information contained in specification stub submodules can be exploited by the analyses to provide early feedback about the system. When a module's specification submodule is available, then it could be used for processing instead of the stubs. A certain class of analyses, described in Section 4.2, is used 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 specification stub submodules of that module.

Why is the specification-stub-submodule feature provided when a specification submodule with strategically placed incompleteness features could conceivably be used instead? The answer is that while only one specification submodule of a module is permitted to exist, several specification stub submodules of that module can populate a developing system to account for the activities of several different development groups. More generally, we feel that a common situation arises in large software projects in which the client modules of a shared module are developed separately—both from each other and from the shared module. The role of specification stub submodules, then, is to represent the (possibly) different views of the shared module held by the various clients. The role of the specification submodule, on the other hand, is to represent, and in fact distinguish, the one, "official" specification of the shared module. The consistency of the various views, as well as their compliance with the "official" specification, can be determined by analyses described in Section 4.2.

```
package stub PINManager is
    used by AutomaticTeller
    provide to AutomaticTeller;

  type PIN is private;
  function Verify ( ThePIN : PIN; ... ) return Boolean;

  ...;

end PINManager;
```

Figure 2: PIC/ADL Specification Stub Submodule of Package PINManager Used by AutomaticTeller.

```
package stub PINManager is
    used by Officer;

  type PIN provide to Officer
    is private;
  procedure Issue ( ...; ThePIN : out PIN )
    provide only to Officer.SetupAccount;
  MasterPIN : constant PIN
    provide only to Officer;

  ...;

end PINManager;
```

Figure 3: PIC/ADL Specification Stub Submodule of Package PINManager Used by Officer.

## 3.3 A Note on Types

Older programming languages, such as FORTRAN and ALGOL60, provide only a limited set of types with which developers can describe systems. In contrast, Ada, like many modern languages, including Clu and Mesa, allows developers to build new types. In essence, the definition of a new type involves the declaration of a name for the type and the identification of a representation for the type in terms of predefined or previously defined types. Additionally, the definition of a new type involves the association of a set of operations with the type. A type together with its operations define a *data abstraction*. An operation of a new type can be an inherited operation, which is an operation originally associated with (one of) the representation types but made applicable to the new type, or it can be a newly defined operation. Thus, new types not only can inherit operations from existing types, they can have their inherited set of operations augmented by the definition of additional operations applicable to objects of the new type.

The interface control that can be exercised over types and their operations is limited in existing languages. In Clu, for example, the only operations of a new type that can be provided by the module declaring the type are those operations that are newly defined for the type; operations inherited from representation types are hidden within the module. Ada is somewhat more flexible than Clu. In Ada, the inherited operations can be provided, if desired, along with any newly defined operations. Unfortunately, both Clu and Ada suffer from the all-or-none syndrome. In particular, all newly defined operations in Clu and Ada are provided to all, no inherited operations in Clu can be provided, and either all inherited operations in Ada are provided to all or they are provided to none. The situation in Mesa is only slightly better than in Ada. In Mesa, the inherited operations of a new type can be requested by other modules. Here again, however, either all or none of the operations are requested and by all or none of the entities in a module.

The PIC language features support a more general and precise approach to interface control for types and their operations. Using the PIC language features, access to the name of a type can be distinguished from access to an operation associated with that type. Moreover, access to each operation, whether that operation is inherited or newly defined, can be individually controlled. Access to the name of a type affords the right to declare objects of the type and to use the type as a representation type in a type declaration. Access to an operation of a type affords the right to invoke the operation

and to inherit the operation as part of a new type declaration.

The request and provide clauses are used to control access to a type's name and operations. Basing control on request and provide clauses permits the specification of multiple views of an abstraction—much like the schema mechanism available in databases [12]—by permitting different entities access to different sets of operations. Note that a view of an abstraction applies not only to the use of the operations, but also to the further inheritance of those operations when the associated type is used in the representation of another type.

As an example, consider an abstraction of a bank account that includes operations to open and close an account, verify an account number, credit and debit funds, and report the balance of an account. Such an abstraction might be supplied by package AccountManager of the automatic bank-teller example. Entities representing actions performed by customers should perhaps only have a view of the abstraction that includes the verify, credit, debit, and report-balance operations, whereas entities representing actions performed by bank employees might have the full view of the abstraction. The PIC language features allow these two views to be described, and therefore enforced, by supporting a precise specification of which entities are given access to particular operations of the abstraction. Notice that the features, because they support both precise requisition and precise provision, allow such views to be specified from either or both the requisition and provision perspectives.

As another example, this one illustrating control over operation inheritance, consider an abstraction of a special kind of bank account, the "holiday club". A holiday-club account differs from the standard account described in the previous example in that customers are not permitted to make withdrawals; entities representing actions performed by customers should only be allowed to invoke the verify, credit, and report-balance operations on holiday-club accounts. The abstraction for holiday-club accounts can (and perhaps should) be described as a derivative of the abstraction for standard accounts, with appropriate adjustments made to the provision and requisition of the associated set of operations. For instance, this can be expressed from the provision perspective, in the syntax of PIC/ADL, as follows.

```
type HolidayAccount is new AccountManager.Account
  provide to Officer,
             Customer with < Verify, Credit, Balance >
```

This specifies that all the operations inherited from Account are provided

to Officer, but that only the inherited operations Verify, Credit, and Balance are provided to Customer.

A more complete discussion of control over types and associated operations is given in [10].

### 3.4 Practical Considerations

The benefits of using the AdaPIC toolset will have to significantly outweigh any extra effort that it requires if developers are to accept the toolset as a practical aid for developing Ada software. In subsequent sections we describe the enhanced analysis possibilities afforded by the toolset and also discuss the incremental development style that it supports. In our opinion, these represent major benefits that justify the limited amount of additional effort entailed in using the toolset. In the remainder of this section we mention some other considerations involved in making the benefits of the AdaPIC toolset outweigh any associated increase in developer effort.

Most of the additional effort involved in using the AdaPIC toolset, as in any approach that is based on explicit import and export constructs, is associated with creating and maintaining the lists of entities appearing in request and provide clauses. The designers of Ada were evidently so concerned about the *"danger of long name lists"* and the concomitant tendency for programmers to *"use standard import lists in fear of omitting something (as is often done for* FORTRAN *common lists)"* ([13], p. 9-5) that they chose to eschew explicit import and export in favor of nesting and all-or-none visibility constructs, with their associated imprecision. Since we believe that extremely precise interface control is crucially important throughout the creation and maintenance of large software systems, we have opted for explicit import and export and have sought ways to minimize the name lists and corresponding effort required of software developers.

Our main approach to addressing this concern has been through careful design of the language constructs, especially their default values. The AdaPIC language dialects have been designed according to the general philosophy, which we believe to be in keeping with the spirit of Ada, that additional effort should only be required when added control over module interfaces is desired. In particular, the defaults for requisition and provision in the dialects—that is, the results when a developer supplies the minimal amount of interface control information through use of the constructs—have been chosen to mimic the control available in Ada itself. For instance, as noted earlier, the absence of a provide clause on a provided entity is inter-

preted to mean that access to the entity is provided to "all". A request clause, like an Ada with clause, is needed to import non-local entities, but requisition of all the provided entities of a package need only entail the listing of that package's name in a request clause, not mentioning each individual requested entity. Achieving finer control over either requisition or provision involves expanded use of the language constructs, as the philosophy suggests. Thus the AdaPIC dialects offer high levels of control, but impose additional effort only on those wishing to take advantage of that increased control.

Another feature that would limit the amount of additional effort entailed by the AdaPIC language dialects is support for higher-level or more convenient descriptions of the relationships among modules. Where our current versions of the constructs require the explicit listing of all modules and entities involved in interface control relationships, this feature would make it possible, for example, to provide shorthand notations for identifying groups of modules and/or entities when describing such 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 or manager [14]), or even for giving a more abstract semantic description (such as input/output behavior [15,16]). We believe that this feature could significantly reduce the additional effort involved in using the AdaPIC toolset, thereby further increasing the extent to which that additional effort is outweighed by the toolset's benefits. We have not yet devoted much attention to this feature, however, preferring to focus on maximizing the benefits that the toolset offers before taking further steps to reduce the associated effort.

While enhanced interface control analysis and support for incremental development are the two principal sources of benefits, the PIC approach, and the AdaPIC toolset in particular, offers other benefits as well. These, spelled out more fully in [17], include improved readability, which contributes to easier modification and maintenance, and careful separation of the concerns of interface control specification from module implementation, which facilitates management control and information hiding. These benefits, coupled with those described in the remainder of this paper, should suffice to make the AdaPIC toolset a practical aid to Ada software developers, well worth whatever little additional effort its use may entail.

## 4. The PIC Analyses

The precision and redundancy of the language features outlined above are of limited value without the ability to obtain feedback about the consistency of the interface relationships specified using those features. This capability is supported in the AdaPIC toolset as an integrated collection of analyses, each of which concentrates on some particular aspect of interface control. By distilling out analysis from the compilation mechanism, which is where it has been historically confined, the PIC approach makes feedback available throughout the development and maintenance process. Moreover, by fashioning analyses from individual *tool fragments* [18], the AdaPIC toolset allows particular analyses, or combinations of analyses, to be flexibly applied as desired.

The analyses can be classified into three major kinds: *basic interface analyses*, *stub analyses*, and *update analyses*. This section elaborates on each of the analysis classes in turn, describes how they handle incompleteness, and then briefly discusses how they are provided as actual tools.

### 4.1 Basic Interface Analyses

There are six basic interface analyses. They provide information on interface consistency within and among modules and are distinguished by the kind of submodules upon which they operate, as depicted in Figure 4. In this figure, each line connecting two submodules corresponds to the analysis applicable to those submodules. (The significance of the dashed line appearing in the figure is explained below.) Note that, at least conceptually, the basic interface analyses make no distinction between specification and specification stub submodules, since specification stubs contain a subset of the information contained in specifications.

While the majority of the basic interface analyses involve pair-wise comparisons of submodules, there are two analyses that can provide meaningful information by simply examining a single submodule in isolation. This is indicated in Figure 4 by the two lines, numbered 1 and 2, originating and terminating at the same submodule. The ability to analyze isolated submodules is important, since, in general, the submodules of an incrementally developed system come into existence one at a time. We are accounting here for the possibility that even specification stub submodules may not be available. Indeed, one (secondary) result of analysis can be a template for a specification stub submodule.

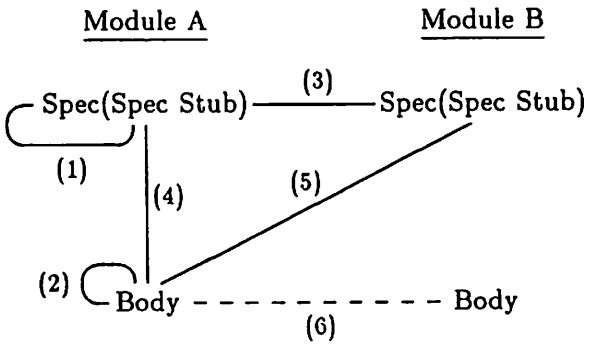


**Figure 4: Basic Interface Analyses.**

The basic interface analyses seek to uncover *errors* and *anomalies* in interface relationships. Anomalies are so named because their detection does not, in and of itself, indicate a definite error, but rather a possible problem that may deserve further attention. One would anticipate that numerous anomalies would be found when incomplete modules are analyzed, whereas anomalies discovered in completed modules might indicate an unacceptable programming style. The decision to classify an interface relationship as erroneous or anomalous, in some cases, can depend upon such factors as the application domain, the language in use, the development method in use, or even the managerial discipline in force. Ideally, tools performing the analyses should be flexible in what they report to the developer, so that they can accommodate a variety of applications, languages, methods, and disciplines. This issue is discussed in Section 4.5.

The two kinds of information that a basic interface analysis makes use of are the available *type* and *requisition/provision* information found in the submodule(s) under examination. PIC/ADL, because it is based on Ada, has some rather sophisticated features that complicate analysis of type information. For example, declarations can be initialized by expressions, which can include function-subprogram invocations. Subprograms can be overloaded; that is, two or more subprograms with the same name can coexist as long as their parameter/return profiles differ. Analysis of type information is further complicated by the fact that the declaration of an entity may not be available during analysis of the use of that entity. For example, a body submodule may make reference to an entity defined in some

other module whose specification is not yet available. As long as there are at least two such references, however, a comparison can be performed that determines the consistency of those references. This is done by *inferring* the type of the entity using techniques similar to the one described in [19]. In some cases, an inconsistency will indicate the presence of a definite error, although which reference (if any) is the correct one cannot be determined without the declaration. In other cases, the analysis can only reveal an anomaly. This is particularly true when comparing subprogram calls, since a perceived inconsistency in the parameterization of those calls may simply be due to overloading. Because the type analysis in PIC/ADL is what would generally be found in a supportive Ada compiler, analysis of type information is not discussed here further.

The requisition/provision information analyzed by the basic interface analyses is contained in specifications of requisition, specifications of provision, and actual references to non-local entities; requisition/provision problems arise from incongruities among these three aspects of module interaction. Table 1 summarizes the problems, showing the error/anomaly classification for PIC/ADL. There are three things to notice about the entries in this table. First, the entries are concerned only with problems associated with module interactions; not listed in the table are errors or anomalies that are exclusively local concerns of a module, such as references to non-provided, local entities whose declarations are missing. Second, the precision and redundancy of the PIC language features can result in the detection of a more revealing set of errors and anomalies than is possible when other languages are used. Finally, while the entries at *b–f* would be considered errors under any circumstance, the error/anomaly classification of the entries at *a* and *g–i* is completely flexible. The given classification reflects just one possible choice; that choice is specifically intended to keep PIC/ADL within the spirit of Ada. For example, Ada is designed with the expectation that many systems will be built from libraries of general-purpose modules, some of whose elements may or may not get used. Therefore, situations where entities are provided but not requested might be common, and so the entry at *g* is considered an anomaly rather than an error. (Control over whether such an anomaly would even be reported by an analysis tool is discussed in Section 4.5.) As another example, consider the entry at *a*. Tradition dictates that requisition of an entity that is not provided be considered an error, irrespective of whether there is an actual reference to the entity. Conceivably, such an interface relationship could instead be considered an anomaly until it is determined that an erroneous reference does or does not exist (cf., *c*).

| ERRORS | |
|---|---|
| *a.* | entity $e_1$ requested by entity $e_2$, but $e_1$ not provided to $e_2$ |
| *b.* | (non-local) entity $e_1$ referred to by entity $e_2$, but $e_1$ not provided to $e_2$ |
| *c.* | (non-local) entity $e_1$ referred to by entity $e_2$, but $e_1$ not requested by $e_2$ |
| *d.* | subprogram provided by a package, but subprogram's body not defined in that package's body submodule |
| *e.* | subprogram, type, or object provided to an entity, but subprogram's parameter/return type, type's discriminant/component type, or object's type, if defined in the same spec or spec-stub, not provided to that entity |
| *f.* | (non-local) packaged subprogram referred to by an entity, but subprogram's body not defined in the package's body submodule |
| ANOMALIES | |
| *g.* | entity $e_1$ provided to entity $e_2$, but $e_1$ not requested by $e_2$ |
| *h.* | entity $e_1$ requested by entity $e_2$, but $e_1$ not referred to by $e_2$ |
| *i.* | entity defined in a module, but not provided nor referred to by that module |

**Table 1: PIC/ADL Requisition/Provision Errors and Anomalies Detected by Basic Interface Analyses.**

| BASIC INTERFACE ANALYSES | SUBMODULE(S) INVOLVED | | | | REQUI./PROVI. ERRORS & ANOMALIES |
|---|---|---|---|---|---|
| | Spec or Spec-Stub A | Body A | Spec or Spec-Stub B | Body B | |
| 1 | √ | | | | *e* |
| 2 | | √ | | | |
| 3 | √ | | √ | | *a(b)*†, *g* |
| 4 | √ | √ | | | *c, d, h, i* |
| 5 | | √ | √ | | *b, f* |
| 6 | | √ | | √ | *f* |

†In PIC/ADL, a reference to a non-local entity in a spec or spec-stub implicitly causes a request for that entity.

**Table 2: Correspondence Between Basic Interface Analyses and PIC/ADL Requisition/Provision Errors and Anomalies.**

The correspondence between the six basic interface analyses and the requisition/provision errors and anomalies is given in Table 2. (Although not indicated in that table, the actual implementations of the analyses also involve the other sorts of checks that would be possible on the submodules—specifically, the analysis of type information and analysis of concerns local to a module, which are mentioned above.) The table is arranged so that those errors and anomalies that can be uncovered by examining a single submodule are listed with analyses 1 and 2 and those found by examining a pair of submodules are listed with analyses 3 through 6. Thus, while any analysis involving a specification or specification stub submodule could detect *e*, that problem is only listed with the first analysis.

The first two analyses examine single submodules. The fact that analysis of a body in isolation cannot reveal any requisition/provision errors or anomalies (although it may reveal type errors or anomalies) is consistent with the fact that bodies are not involved in interface control per se. The third analysis is an inter-module analysis that compares the specification (specification stub) submodule of a module B to the specification (specification stub) submodule of another module A, checking the entities requested by A against the entities provided by B. The fourth analysis is an intra-module analysis that checks the entities requested in the specification (specification stub) submodule against the entities actually referred to in

the corresponding body submodule. In addition, this analysis checks that a subprogram provided in the specification (specification stub) submodule has a body defined in the body submodule. The fifth basic interface analysis is an inter-module analysis that checks the entities provided by a module, through a specification (specification stub) submodule, against the entities referred to in the body submodule of a second module. This analysis also checks the subprogram references in the specification (specification stub) submodule against the subprogram bodies defined in the body submodule. Finally, the sixth analysis is an inter-module analysis that checks the subprogram bodies defined in the body submodule of a module B against the references in the body submodule of a module A. This last analysis could be disregarded (hence the dashed line in Figure 4) if the reasonable assumption is made that a body submodule would not be analyzed with respect to any other module until that body's corresponding specification (specification stub) submodule is present and intra-module analysis 4 performed. If this assumption is made, then no new information about the interface consistency of the modules is gained by the sixth analysis, since an error in the interface relationship would always be revealed as one or both of *c* or *d*.

## 4.2 Stub Analyses

As described in Section 3, a specification stub submodule represents the view some set of modules has of a given module. There are two stub analyses and these provide information on the consistency of such a view. Like the basic interface analyses, they seek to uncover interface problems by examining the available type and requisition/provision information.

The first stub analysis is used to check the consistency of one view of a module with respect to another view of that same module. The two primary functions of this analysis are: (1) to identify the entities occurring in one specification stub submodule and not the other (i.e., a "difference" analysis); and (2) to identify the entities occurring in both specification stubs, such as common declarations or common references, and to check the consistency of those common occurrences. For the first function, when an entity occurs in only one of the views, the situation is not considered an inconsistency, since it is the express purpose of specification stubs to accommodate differences during development. For the second function, the checking of common occurrences mostly involves analysis of type information. In fact, the only PIC/ADL requisition/provision problem that can be detected is when two specification stubs provide the same entity, but one of the specification stubs

| ANOMALIES | |
|---|---|
| *aa.* | entity $e_1$ provided to entity $e_2$ in the specification, but $e_1$ not provided to $e_2$ in the specification stub |
| *bb.* | entity $e_1$ provided to entity $e_2$ in the specification stub, but $e_1$ not provided to $e_2$ in the specification |
| *cc.* | entity $e_1$ provided *exclusively* to entity $e_2$ in the specification stub, but $e_1$ not provided *exclusively* to $e_2$ in the specification |
| *dd.* | entity requested in the specification stub, but not requested in the specification |
| *ee.* | entity requested in the specification, but not requested in the specification stub |

**Table 3: PIC/ADL Requisition/Provision Anomalies Detected by Spec/Stub Stub Analysis.**

attempts to provide the entity *exclusively* to some module (see Section 3.2).

The second stub analysis is used to check the consistency of each view of a module with respect to the "official" specification submodule of that module. The information contained in a specification stub submodule must be some subset of the information contained in the specification submodule and that subset must be type and requisition/provision consistent. Table 3 summarizes the requisition/provision problems that can be detected by this analysis. (Again, type errors and anomalies are not discussed here.) In PIC/ADL, the entries in the table are all considered anomalies, since they do not necessarily indicate the existence of an error. For example, consider entry *bb*. If the entity is not actually requested by the module using the specification stub, then the fact that the entity is not provided to the module by the specification leaves the relationship consistent.

It is important to point out that the first stub analysis is useful for more than just monitoring the development of specification stub submodules. In particular, if the views are found to be consistent, then the results of that analysis can be used by a specialized processing tool to generate a *template* for a specification submodule. This template would be a consistent

amalgamation of the information given in the specification stubs. The developer could then use this template as the basis for creating the "official" specification submodule, supplying missing information such as additional requisition specifications.

## 4.3 Update Analyses

Change is an intrinsic characteristic of any software development process. Particularly in the development of a large system, it is important to know not only *what* has changed but what *effect* a change has had on the system, since those effects can be far reaching and perhaps unanticipated.

The PIC approach supports three update analyses, which correspond to each of the three kinds of submodules. To provide information on changes to interface relationships, each analysis involves a comparison between two versions of the same submodule and looks for changes in declarations, requisition and provision specifications, or references to non-local entities.

What it means for one submodule to be a version or *update* of another submodule depends upon the kind of submodule involved. In the case of specification submodules, the two submodules must simply be specification submodules of the same module. Likewise for body submodules, the two submodules must be body submodules of the same module. For specification stub submodules, the two submodules must be specification stubs of the same module and—because multiple specification stubs of a module, corresponding to different clients of that module, can coexist in a system—they must provide one or more entities to the same module. (Syntactically, versions of specification stubs are easily recognized in PIC/ADL by their having common elements in their used-by clauses; for example, they begin with the header statements **package stub** A **is used by** B and **package stub** A **is used by** B, C.)

The greater precision with which a developer can specify interface relationships using the PIC language features allows the update analyses to supply more revealing and meaningful information about changes than is possible with other languages. For example, if the developer of a module has specified exactly which other modules an entity is provided to, then a change to that provision, such as no longer providing that entity to one of the modules, is detectable. This is in contrast to the situation in Ada, where changes of this sort are masked by the imprecision of the visible part of packages.

While the update analyses do not directly assess interface consistency,

which is one of the primary purposes of the other two classes of analyses, they are important to interface consistency analysis in that they reveal the relationships that must be subjected to *reanalysis* as a result of a change. Moreover, knowledge of exactly what is, and what is not, affected by a change can help reduce the sheer amount of that reanalysis.[2]

## 4.4 Consistency and Incompleteness

The analyses described above account for incompleteness by treating *consistency* as a state in which interface relationships cannot be shown incorrect. To illustrate, consider the relationships between three of the submodules involved in the automatic bank-teller example: the specification submodule of AutomaticTeller (Figure 1) and the specification submodules of ATMaintenance and Officer (Figure 5).

AutomaticTeller provides, among other entities, functions RemainingCash and DepositsMade. Access to these functions is limited through attached provide clauses; access to RemainingCash is provided to ATMaintenance and Officer, while access to DepositsMade is provided to ATMaintenance and, as indicated by the ellipsis in the provide clause, some as yet undetermined other submodule(s). The request clause in the specification submodule of ATMaintenance indicates that access is requested to RemainingCash and DepositsMade. This is certainly consistent with the provision specified in AutomaticTeller since ATMaintenance appears in the provide clauses of both RemainingCash and DepositsMade. As is the case for ATMaintenance, the specification submodule of Officer indicates that access is requested to RemainingCash and DepositsMade. But unlike that case, access to DepositsMade is not explicitly provided to Officer by AutomaticTeller since Officer does not appear in the provide clause attached to DepositsMade. Under our definition, however, Officer's interface is still considered to be consistent with the interface of AutomaticTeller because the presence of the ellipsis in the provide clause allows the possibility that DepositsMade will at some time be provided to Officer and therefore no inconsistency can be shown to exist between the interfaces.

When the application of a basic-interface or stub analysis does not reveal any errors or anomalies, then the two submodules involved are said to be consistent with respect to that analysis. Confidence in that consistency must be tempered if there is incompleteness in the submodules, since consistency

[2]Tichy [20] discusses this in the restricted context of recompilation savings.

```
package ATMaintenance is
        request AutomaticTeller.( RemainingCash, DepositsMade );

    ...;

end ATMaintenance;


package Officer is
        request AutomaticTeller.( RemainingCash, DepositsMade );

    ...;

end Officer;
```

**Figure 5: Specification Submodules of Packages AT-Maintenance and Officer.**

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, the basic-interface and stub analyses recognize a pair of submodules as *consistent* only 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 AutomaticTeller and ATMaintenance are consistent but AutomaticTeller and Officer are only conditionally consistent.

## 4.5 From Analyses to Analysis Tools

The analyses described above represent the primitive feedback capabilities made possible by the precision, redundancy, and explicit treatment of incompleteness in the PIC language features. Although it is conceivable to think of these analyses as separate tools in an environment, they are probably best thought of as composeable tool fragments. For example, the detection of an anomaly by a basic-interface or stub analysis often implies the need to perform further checking to determine if an error actually exists. An update analysis that reports a change in provision or requisition is an example of where one analysis can lead to or "trigger" the application of

other analyses.

The way in which the analyses are presented as actual tools to the user depends upon which sequences of analyses, if any, are to be enforced by the environment. Specifying sequences of analyses could be left totally to the discretion of the developer. This provides the developer with the most freedom and flexibility, since combinations of different analyses would be applied as desired. Such a lack of control, however, could be costly. For example, the rechecking of interface relationships that should follow update analysis could be forgotten by the developer. At the other extreme, the environment could rigidly enforce predefined sequences of analyses. A better alternative is to make the sequences of analyses a "programmable" aspect of the environment. This compromise would allow flexibility at the same time that it enforces development discipline.

An example of a high-level algorithm that specifies a sequence of analyses is given in Figure 6. In this example, spec/stub stub analysis has uncovered an anomaly of type *aa*. This means that in a specification submodule of a module A, some entity E1 has been provided to an entity B.E2, but the specification stub submodule of A used by B did not provide that entity. This anomaly raises the question: Is the specification submodule of A correcting an oversight in the specification stub submodule or is the specification submodule inaccurately adding superfluous information? As this example illustrates, there are a number of different interrelations that exist among the analyses. To reap the full benefits of the AdaPIC toolset, these relationships must be captured so that sequences of analyses can be automatically activated as errors and anomalies are uncovered. Our work on environment architecture is investigating ways in which the environment infrastructure can support the algorithmic specification and application of sequences of tool activations [21,22].

In addition to deciding which analyses are to be applied when, another concern is deciding what information the developer is actually given as a result of a particular application of an analysis or sequence of analyses. As previously pointed out, reports of numerous anomalies would be anticipated when analyzing incomplete modules; the developer could easily be overwhelmed by all the "revealing and meaningful information" produced! Thus, developers should be able to turn on and off particular types of reports provided by the analyses. As described in Section 6, the AdaPIC toolset provides a separate "reporter" tool fragment that supports this capability. Another possibility is to have the tool activation algorithms screen sequences of invocations and delay error reporting. This is demonstrated

```
procedure Anomaly_aa_Found ( ... ) is
  -- Specification submodule of A provides E1 to B.E2, but
  -- specification stub submodule of A used by B does not

  -- Check existing interface consistency data
  if Error_a ( A.E1, B.E2 ) or Error_b ( A.E1, B.E2 ) then
    -- Specification submodule of A corrects specification stub submodule's
    -- omitted provision of E1 to B.E2
    Update_Interface_Consistency_Data ( ... );

  else
    -- Undetected problem
    -- Check if newly provided entity A.E1 is requested or referenced by B.E2

    -- First check if requested
    Apply_Analysis_3 ( A, B, ...);

    -- Then check if referenced
    Apply_Analysis_4 ( B, ... );

    if Anomaly_g ( A.E1, B.E2 ) then
      -- A.E1 not requested; is it referenced?

      if Error_c ( A.E1, B.E2 ) then
        -- Record "A.E1 provided to and referenced by B.E2 but not requested"
        Update_Interface_Consistency_Data ( ... );

      else
        -- Record "A.E1 provided to B.E2 but not requested nor referenced"
        Update_Interface_Consistency_Data ( ... );

      end if;

    else
      -- A.E1 requested
      if Anomaly_h ( A.E1, B.E2 ) then
        -- Record "A.E1 provided to and requested by B.E2 but not referenced"
        Update_Interface_Consistency_Data ( ... );

      end if;
    end if;
  end if;

end Anomaly_aa_Found;
```

**Figure 6: Skeleton of an Algorithm that Specifies a Sequence of Analyses.**

in the example above where anomaly *aa* is not reported directly. Instead, other analyses are applied so that more information is known before a report is generated.

## 5. Incremental Development in PIC

It is commonly held that languages such as Ada, Mesa, and MODULA-2 can, through their facilities for separate compilation, support the incremental development of large software systems. Unfortunately, that belief is not wholly justified, since these languages can in fact only support a restricted form of incremental development. That form is governed by the following rule for submitting submodules for analysis (i.e., compilation) in Ada, Mesa, and MODULA-2.

> *A module's specification submodule must be analyzed and "accepted" before its body submodule is submitted and before a body or specification submodule (of some other module) that uses it may be submitted.*

On the one hand, this rule means that body submodules can be developed in any order, as long as the appropriate specification submodules have already been analyzed and "accepted". On the other hand, it means that specification submodules must be developed in a very particular order, namely one that is strictly bottom up.[3] This restriction has some rather severe methodological implications. In particular, it forces the programming of the lowest-level modules to begin before any analysis can be done at higher levels. Moreover, if one considers specification submodules to represent design decisions concerning the modularization and interface relationships of a system, then those decisions—if they are to be subject to incremental analysis—can only be made from the bottom up.

The restriction these languages impose on the development of specification submodules stems from a desire to perform code generation at the same time as incremental interface analysis. In Ada, for example, the representation of a private type must appear in the specification part of a package even though that representation is logically hidden from the users

[3]The Ada *subunit* facility that allows the body of a unit to be compiled separately from its nested declaration was specifically devised to support top-down program development ([13], p. 10-5). In fact, it only supports top-down development of the bodies of nested modules. The specifications of those nested modules are still unavoidably limited to bottom-up development.

of that abstraction; its presence in the specification is solely to provide code-generation information. The only way to perform both code generation and incremental interface analysis at once in languages such as Ada, Mesa, and MODULA-2 is to insist on a bottom-up development process for specification submodules. When such programming languages serve as models for specification and design languages, then this restriction is carried into pre-implementation phases, where it causes even worse problems. This is the case for many Ada-based design languages (e.g., [23]), which because of this restriction impede top-down system development.

While substantial information is indeed necessary to perform code generation (and, especially, code optimization), meaningful interface analyses can be performed with much less information. We would argue, therefore, that the concerns of code generation, while extremely important, should be addressed separately from those of interface analysis. Such a separation would, for instance, facilitate the top-down development of specification submodules by allowing those of high-level modules to be analyzed early in development. Actual code generation would occur, as before, once sufficient information was present, yet subsequent to the interface analyses. Thus, to support incremental development, the standard monolithic compiler structure becomes inadequate. Syntactic analysis, semantic analysis, and code generation, for example, are now all analysis components that may be invoked at very different times in the development process.

The primary characteristics of the PIC approach that permit it to fully support incremental development are (1) the formulation of interface analyses as specialized tools that are distinct from other activities (such as code generation), and (2) the availability of the incompleteness feature and the specification stub submodule for explicitly deferring design decisions by representing, in an analyzable form, incomplete information about a module's interface. The power of these two capabilities is illustrated in the example below, which is an elaboration on the automatic bank-teller example. In Section 3.1, a package AutomaticTeller is described, which is at an intermediate level in the hierarchical structure of the system; the package's position in the hierarchy is evident from the fact that it both provides and requests entities. Here we show several steps in the top-down development of that portion of the automatic bank-teller system rooted at AutomaticTeller.

As discussed in Section 3.1, AutomaticTeller contains requests for entities from two packages: AccountManager and PINManager. If this system were being developed in pure Ada, then the specification submodules of both those packages would have to be created, analyzed, and "accepted" before

any analysis on the submodules of AutomaticTeller could be performed. Furthermore, if either of those specification submodules contained a reference to another, lower-level module (which, as shown below, they in fact do), then the specification of that lower-level module would also have to be created, analyzed, and "accepted" before any analysis involving the submodules of AccountManager, PINManager, or AutomaticTeller could be performed, and so on. The result, therefore, would be a bottom-up development of the specification submodules in the automatic bank-teller system.

With just the specification and body submodules of AutomaticTeller available, an analysis (number 4 of Figure 4) can be performed using the AdaPIC toolset that provides, among other information, feedback about whether there are references in the body submodule that exceed the requests in the specification submodule. To begin inter-module analysis, nothing more needs to be done in the way of development than to supply a specification stub submodule of either AccountManager or PINManager that can be used by AutomaticTeller. Figure 2 shows such a submodule of PINManager. This submodule indicates that PINManager is expected to provide AutomaticTeller with a type PIN, whose representation is not available, and a function Verify, whose parameters have not been completely determined. Two additional analyses can now be performed, one checking requests in the specification submodule of AutomaticTeller (number 3 of Figure 4) and the other checking references in the body submodule of AutomaticTeller (number 5 of Figure 4).

Eventually, the "official" specification submodule of PINManager is made available (Figure 7). Once again, if the system were being developed in pure Ada, then the specification submodule of the lower-level module ESManager, which is referred to in PINManager, would have to be developed before any analysis involving PINManager could be performed. Instead, an analysis can already be performed with the AdaPIC toolset that checks the consistency between AutomaticTeller and PINManager. (Consistency can also be checked at this point between Officer and PINManager.) This analysis would involve the specification stub submodule of PINManager used by AutomaticTeller (spec/stub analysis of Table 3) or, alternatively, the specification and body submodules of AutomaticTeller directly (analyses 3 and 5 of Figure 4). The choice depends mainly upon whether the specification stub submodule sufficiently represents the use of PINManager by AutomaticTeller. There are a number of ways to automatically determine the best choice, based on previous analyses, and to limit the amount of unnecessary (re)analysis. For example, if the specification stub had in fact been mechanically generated

```
package PINManager is
      request ESManager;

  type PIN is private;

  function Verify ( ThePIN : PIN; ... ) return Boolean
    provide to AutomaticTeller;

  procedure Issue ( ... )
    provide to Officer.SetupAccount;
  MasterPIN : constant PIN
    provide to Officer;

  ...;

private
  type PIN is new ESManager.EncryptedStringType;
  MasterPIN : constant PIN := ...;

end PINManager;
```

**Figure 7: Specification Submodule of Package PINManager.**

from AutomaticTeller, then that specification stub is guaranteed to fully represent AutomaticTeller's view of PINManager. These and other approaches are being explored in our prototype of the AdaPIC toolset.

The automatic bank-teller system is now at a similar point in its top-down development as when AutomaticTeller was about to undergo inter-module analysis; a specification stub submodule of a lower-level module (ESManager) needs to be supplied for a higher-level module (PINManager). Thus, development would proceed from here in a manner similar to that discussed above. The major advantage of the PIC approach to incremental development is that developers can be confident that, even though the system is incomplete, the current description of the system is consistent.

## 6. Concluding Remarks

We are currently constructing a prototype of the AdaPIC toolset within the larger context of the Arcadia software development environment project

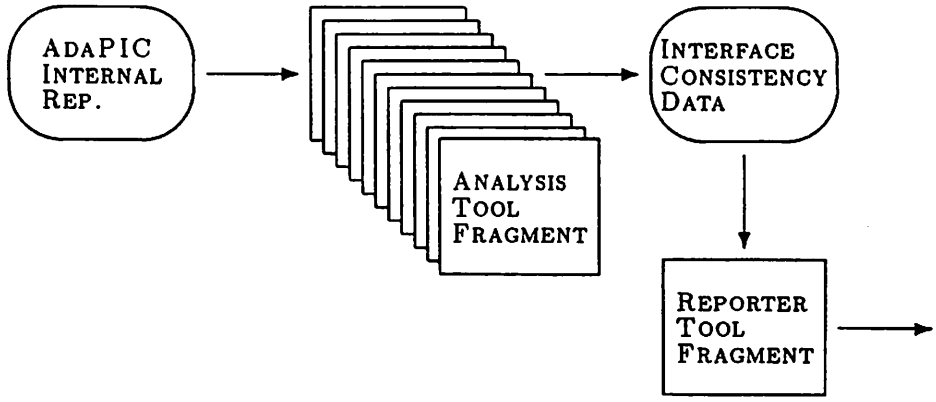


**Figure 8: Conceptual Organization of the AdaPIC Prototype's Analysis Component.**

[21,22]. The AdaPIC toolset's analysis capabilities are being implemented as small, self-contained tool fragments, so as to permit experimentation with our approach to incremental analysis and order-independent development. Figure 8 shows the conceptual organization of this analysis component. The various analysis tool fragments correspond to the eleven *basic*, *stub*, and *update* analyses discussed in Section 4. They operate upon an internal representation derived from a description expressed in any one of the AdaPIC dialects, such as PIC/ADL. Each fragment records interface consistency data, including indications of any interface problems encountered. The reporter tool fragment is charged with deciding which problems to report and whether those problems are reported as errors or as anomalies. A specific analysis tool is therefore a combination of the reporter tool fragment with some subset of the analysis tool fragments. Arcadia provides facilities for flexibly integrating these fragments and thus provides a means to compose analyses as described in Section 4.5.

Implementation of the prototype AdaPIC toolset is itself being carried out incrementally, using the PIC language features and analysis capabilities to facilitate a top-down incremental development. Based on this preliminary use of the approach, we are encouraged about the contributions that the completed toolset will make to the technology for developing large Ada software systems.

## Acknowledgements

We wish to thank Walter Kopp for contributions to the design and implementation of the AdaPIC prototype.

## REFERENCES


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

[2] 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, November 1980, pp. 139–145.

[3] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *A Formal Model for Describing and Evaluating Visibility Control Mechanisms*, **Proceedings of the IEEE Computer Society 1986 International Conference on Computer Languages**, Miami Beach, Florida, October 1986, pp. 182–189.

[4] 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.

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

[6] 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.

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

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

[9] J.C. Wileden and L.A. Clarke, *Feedback-Directed Development of Complex Software Systems*, **Proceedings of Software Process Workshop**, Egham, Surrey, England, February 1984, pp. 89–93.

[10] A.L. Wolf, *Language and Tool Support for Precise Interface Control* (Ph.D. Dissertation), **COINS Technical Report 85–23**, COINS Department, University of Massachusetts, Amherst, Massachusetts, September 1985.

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

[12] J.D. Ullman, **Principles of Database Systems**, Addison-Wesley, Reading, Massachusetts, 1980.

[13] J.D. Ichbiah, et al., **Rationale for the Design of the Ada Programming Language**, appearing in **SIGPLAN Notices**, Vol. 14, No. 6, June 1979.

[14] N.H. Minsky and A. Borgida, *The Darwin Software-Evolution Environment*, **Proceedings of the ACM-SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**, appearing in **SIGPLAN Notices**, Vol. 19, No. 5, May 1984, pp. 89–95.

[15] D. Luckham and F.W. von Henke, *An Overview of Anna, a Specification Language for Ada*, **IEEE Software**, Vol. 2, No. 2, March 1985, pp. 9–22.

[16] D.E. Perry, *Inscape Reports*, **Technical Report**, Computer Technology Research Laboratory, AT&T Bell Laboratories, Murray Hill, New Jersey, April 1986.

[17] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *Ada-Based Support for Programming-in-the-Large*, **IEEE Software**, Vol. 2, No. 2, March 1985, pp. 58–71.

[18] L.J. Osterweil, *Toolpack—An Experimental Software Development Environment Research Project*, **IEEE Transactions on Software Engineering**, Vol. SE-9, No. 6, November 1983, pp. 673–685.

[19] M.R. Levy, *Type Checking, Separate Compilation and Reusability*, **Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction**, appearing in **SIGPLAN Notices**, Vol. 19, No. 6, June 1984, pp. 285–289.

[20] W.F. Tichy, *Smart Recompilation*, **ACM Transactions on Programming Languages and Systems**, Vol. 8, No. 3, July 1986, pp.273–291.

[21] R.N. Taylor, L.A. Clarke, L.J. Osterweil, J.C. Wileden, and M. Young, *Arcadia: A Software Development Environment Research Project*, **Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments**, Miami Beach, Florida, April 1986, pp. 137–149.

[22] A.L. Wolf, *An Overview of Arcadia*, **Proceedings of the ACM-SIGAda Future APSE '86 Workshop**, Saratoga Springs, New York, September 1986, to appear in **Ada Letters**, 1987.

[23] P. Baker and C. Youngblut, *Survey of Ada-based PDLs*, Naval Avionics Center, Indianapolis, Indiana, January 1985.