INTERFACE CONTROL AND INCREMENTAL DEVELOPMENT IN THE PIC ENVIRONMENT

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

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

ABSTRACT

The PIC environment is designed to provide support for interface control that facilitates incremental development of a software system. Interface control, the description and analysis of relationships among system components, is important from the earliest stages of the software development process right through to the implementation and maintenance stages. Incremental development, wherein a software system is produced through a sequence of relatively small steps and progress may be rigorously and thoroughly assessed after each step, must be accommodated by any realistic model of the software development process. This paper focuses on the analysis component of the PIC environment and demonstrates how it contributes to precise interface control capabilities while supporting an incremental software development process.

1. Introduction

The ability to accurately describe and analyze the relationships among components of a software system plays a key role throughout the software development process. 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 [1].

Despite the importance of these activities, most languages and development environments do not provide adequate support for them. Existing languages typically permit the relationships among a software system's components to be described with only limited accuracy and environments seldom provide tools capable of performing a thorough analysis of those relationships. Moreover, existing approaches 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 are developing the PIC (Precise Interface Control) environment. The PIC environment will consist of language features for precisely specifying interface relationships [12] and an extensive set of tools for analyzing and managing interface control information. It is being tailored to support an incremental approach to the interface control aspects of the software development process. This paper focuses on the PIC environment's tools for analysis of interface control information and its support for incremental development.

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

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 simple subprogram unit, such as a subroutine, function, or task, or an encapsulation unit, such as an Ada® *package* or MODULA-2 *module*. An encapsulation serves to group together 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 throughout the years. FORTRAN primarily used labelled and blank *common*. ALGOL60 introduced nested declarations. More recent languages, such as Ada, CLU, MODULA-2, and Mesa, 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 [3], C/Mesa [6], or INTERCOL [8], have relied on essentially these same concepts. We have demonstrated that these interface control mechanisms do not adequately describe all the interface relationships that need to be expressed [2,11] and have observed that, partially for this reason, they do not permit thorough interface control analysis.

The PIC language features improve upon the precision found in current mechanisms and allow complementary, albeit redundant, descriptions of the interface control relationships between modules. 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 our 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:

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

- *Consistent abstractions.* The languages used throughout development should be based upon a consistent set of abstractions [10]. 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 PIC context, 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 the environment adequately handle an arbitrary submission order.

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 environment has been specifically designed to facilitate incremental development. The language features to describe interface relationships have been based on the consistent abstractions, termed *requisition* and *provision* of access, described in Section 2. Support for incomplete descriptions of modules is provided in two ways. First, the PIC language features include a construct for explicitly indicating within a module's description that additional information is to be provided later. Second, 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.

The remainder of this paper describes the PIC environment and demonstrates how it supports incremental development. The next section briefly describes the language features and their use in a language based on Ada. Section 3 outlines the analyses that can be performed on systems described using the language features. Section 4 presents an example that illustrates how the PIC approach to interface control supports incremental development. The conclusion discusses the current status of the PIC environment.

2. Overview of the PIC Language Features

The conceptual foundation for the PIC language features 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 interface control mechanism is simply a means for specifying requisition and provision.

The PIC language features provide support for the explicit specification of both requisition and provision, and thus constitute a precise interface control mechanism. In addition, they provide a system structure that imposes a strict separation of interface control information from the algorithmic details (if any) of how that information is used locally by a module. This separation facilitates information hiding and managerial control. It also supports incremental analysis by allowing interface control information to be created, modified, and (re)analyzed independently of a system's (detailed) algorithm development.

The capabilities provided by the language features are relevant throughout the lifetime of a software system, and appropriate dialects of the 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-flavored dialect, which we refer to as PIC/Ada, suitable for use in conjunction with an Ada-based PDL or the Ada language itself. Here, we describe only those aspects of PIC/Ada applicable to one kind of module, namely Ada's encapsulation unit, the package. A more detailed treatment of this Ada dialect of the PIC language features can be found in [12].

To realize the separation of interface control information from algorithmic detail, a module consists of two physically distinct parts: a *specification submodule* and a *body submodule*. A package's specification submodule describes the entities encapsulated by the package. It also completely describes the package's requisition, through one or more *request clauses*, and provision, through one or more *provide clauses*. The body submodule for a package contains the actual code sections realizing the module. During the pre-implementation phases, the body might take the form of a PDL description, while in later phases it would consist of standard implementation-language code.

Figure 1 presents an example illustrating several aspects of the language features. The example shows the specification and body submodules of a package AutomaticTeller, which is one module in 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 CustomerInterface, ATMaintenanceInterface, and OfficerInterface, which use subprograms provided by AutomaticTeller in realizing three classes of user-interface capabilities. Also part

[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".

```
package AutomaticTeller is
  procedure BeginSession ( ... )
    provide to CustomerInterface
    request PINManager.( PINType, Verify ),
            AccountManager.( AccountType, Verify );
  procedure Deposit ( ... )
    provide to CustomerInterface
    request AccountManager.Credit;
  procedure Withdraw ( ... )
    provide to CustomerInterface;
    request AccountManager...;
  ...;
  function RemainingCash ( ... ) return ...
    provide to ATMaintenanceInterface, OfficerInterface;
  function DepositsMade ( ... ) return ...
    provide to ATMaintenanceInterface, ...;
  ...;
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: Specification and Body Submodules of Package AutomaticTeller.

of the hypothetical system are the modules AccountManager, which provides facilities for manipulating customer accounts, and PINManager, which provides facilities for manipulating the "personal identification numbers" that serve as passwords for customer accounts.

A specification submodule in this PIC/Ada notation is essentially an Ada unit specification together with a small number of additional, and powerful, features for enhancing interface control. The *request clause* is used to specify exactly the entities that a given module, or packaged entity, wishes to have the right to access.[2] In Figure 1, procedure BeginSession requests access, through a *request clause*, to two entities in PINManager as well as access to two entities in AccountManager. The *provide clause* may be appended to any of a package's visible entities in order to selectively limit their provision to external modules. The *provide clause* appended to procedure BeginSession indicates that it is only provided to CustomerInterface, whereas the *provide clause* appended to function RemainingCash indicates that it is only provided to ATMaintenanceInterface and OfficerInterface.

The *request clause* is more flexible than its counterparts in most other languages, including Ada's *with clause*, in at least two ways. First, it does not necessarily import all the provided entities of a package but can import subsets of those entities. Second, a *request clause* can be attached to an individual packaged entity, as well as to the package itself, so that requisition by the entities within a package can be differentiated. The *provide clause* is similarly more flexible than provision facilities of other languages, including Ada's mechanism, which is based on constructs that textually separate a package's visible (i.e., provided) entities from its hidden entities. Under the Ada mechanism, provision is controlled on an all-or-nothing basis; either access to an entity is provided to every module (in a given scope), or it is provided to no module, and so the entity is hidden. The PIC *provide clause*, however, supports *selective* provision.

Another aspect of the PIC language features is their applicability to high-level, incomplete descriptions of a system's components and their interaction. The *incompleteness construct*, denoted by an ellipsis in PIC/Ada and appearing, for example, in the parameter lists of the subprograms in Figure 1, is useful for explicitly indicating where details that will be supplied later have been omitted from a description. It complements other constructs, not illustrated here, that facilitate the formulation of abstract, pre-implementation descriptions, such as notations to formally specify a module's external behavior or to describe intended algorithms. When used in conjunction with such constructs, the language features are well suited for expressing modularization and interface properties during early stages of a system's development and hence supportive of incremental development.

In addition to specification and body submodules, the PIC 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. 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. As shown in Section 4, these views can be exploited by the analysis tools to provide early feedback about the system.

Two examples of 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 OfficerInterface, respectively. The *used-by clause* appearing in a specification stub submodule indicates the intended clients of that stub. 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 should provide the entity only to the listed modules.

3. Analyses

The precision and redundancy of the language features outlined above is of limited value without the ability to obtain feed-

[2] As a notational shorthand in PIC/Ada, an actual reference to a non-local entity that occurs within a specification submodule, such as the declaration of an object whose type is not locally defined, *implicitly* causes the requisition of that entity. The *request clause* is primarily used, therefore, to request entities that are to be referred to in the body submodule.

```
package stub PINManager is
    used by AutomaticTeller
    provide to AutomaticTeller;
  type PINType;
  function Verify ( PIN : PINType; ... ) return Boolean;
  ...;
end PINManager;
```

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

```
package stub PINManager is
    used by OfficerInterface;
  type PINType
    provide to OfficerInterface;
  procedure Issue ( ...; PIN : out PINType )
    provide only to OfficerInterface;
  MasterPIN : constant PINType
    provide only to OfficerInterface;
  ...;
end PINManager;
```

Figure 3: Specification Stub Submodule of Package PINManager Used by OfficerInterface.

back about the consistency of the interface relationships specified using those features. This capability is provided in PIC 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 environment makes feedback available throughout the development and maintenance process. Moreover, by fashioning analyses from individual *tool fragments* [7], the environment 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 and then briefly discusses how they can be provided as actual tools in the environment. A discussion of how the analyses handle incompleteness appears in [12].

3.1 Basic Interface Analyses

The six basic interface analyses provide information on interface consistency within and among modules. They are distinguished by the kind of submodules upon which they operate, as depicted in Figure 4. 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 important, since, in general, the submodules of an incrementally developed system come into existence one at a time. Hence, PIC can provide support for incremental interface analysis even when submodules are developed before the submodules they interact with are developed. Note that 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 such a submodule.

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, at least in some cases, can depend upon such factors as the development method in use or the managerial discipline in force. Ideally, tools performing the analyses should be flexible in what they report to the developer as an error or as an anomaly in such cases.

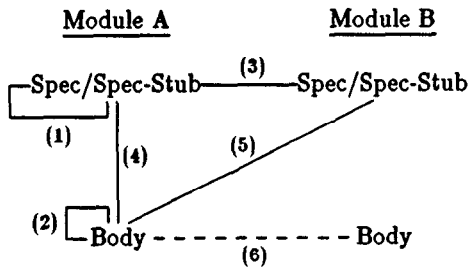

Figure 4: Basic Interface Analyses.

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/Ada, 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 [5]. 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. For the sake of brevity, and because the type analysis in PIC is what is generally found in a supportive 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 nonlocal entities; requisition/provision errors and anomalies arise from incongruities among these three aspects of module interaction. Table 1 summarizes the detectable errors and anomalies. There are three things to notice about the entries in that 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. Of course, a complete analysis tool would seek to detect those problems as well. Second, the precision and

ERRORS	
E1.	entity requested in a module, but not provided to that module
E2.	(non-local) entity referred to in a module, but not provided to that module
E3.	(non-local) entity referred to in a module, but not requested in that module
E4.	subprogram provided by a package, but subprogram's body not defined in that package's body submodule
E5.	subprogram or object provided to a module, but subprogram's parameter/return type or object's type (if defined in the same spec or spec-stub) not provided to that module
E6.	(non-local) packaged subprogram referred to in a module, but subprogram's body not defined in the package's body submodule
ANOMALIES	
A1.	entity provided to a module, but not requested in that module
A2.	entity requested in a module, but not referred to in that module
A3.	entity defined in a module, but not provided nor referred to by that module

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

redundancy available in PIC/Ada 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 E2–E6 would be considered errors under any circumstance, the error/anomaly classification of the entries at E1 and A1–A3 is completely flexible. The given classification reflects just one possible choice; that choice is specifically intended to keep PIC/Ada 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 A1 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 3.4.) As another example, consider the entry at E1. 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., E3).

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 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 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 E5, that error 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 (stub) submodule of a module B to the 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 (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 (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 (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 (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 (stub) submodule is present and intra-module analysis number 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 E3 or E4.

	SUBMODULE(S) INVOLVED				ERRORS & ANOMALIES
	Spec or Spec-Stub A	Body A	Spec or Spec-Stub B	Body B	
1	√				E5
2		√			
3	√		√		E1(E2)†, A1
4	√	√			E3, E4, A2, A3
5		√	√		E2, E6
6		√		√	E6

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

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

3.2 Stub Analyses

As described in Section 2, a specification stub submodule represents the view some set of modules has of a given module. The two stub analyses provide information on the consistency of such a view, seeking to uncover interface problems, as do the basic interface analyses, by examining the available type and requisition/provision information. Thus, while a number of specification stub submodules of a module may be independently constructed, their development, and the development of the modules using them, can be monitored using the information supplied by the two stub analyses.

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

ANOMALIES	
A1.	entity provided to a module by specification, but not provided to that module by specification stub
A2.	entity provided to a module by specification stub, but not provided to that module by specification
A3.	entity provided *exclusively* to a module by specification stub, but not provided *exclusively* to that module by specification
A4.	entity requested in specification stub, but not requested in specification
A5.	entity requested in specification, but not requested in specification stub

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

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 checking 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 requisition/provision problem that can be detected is when two specification stubs provide the same entity, but one of the specification stubs attempts to provide the entity *exclusively* to some module (see Section 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 are not discussed here.) In PIC/Ada, the entries in the table are considered anomalies, since they do not necessarily indicate the existence of an error. For example, consider entry A2. 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.

3.3 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 on the system, since those effects can be far reaching and perhaps unanticipated.

The PIC environment provides 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. The greater precision with which a developer can specify interface relationships using the PIC language features allows the analyses to supply more revealing and meaningful information about changes. 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 library units.

While the update analyses do not directly assess interface consistency, which is the primary purpose 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.[3]

3.4 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 the environment, they are probably best thought of as composed of 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 tools to the user of the environment, therefore, depends upon whether, and how, the designers of that environment wish to enforce combinations and/or sequences of analyses.

The combinations and sequences of analyses in different development processes could be enforced by developer discipline. This would allow flexibility (e.g., combinations of different software processes, or even discovery of new ones) at the expense of a lack of managerial control. Such a lack of control could be costly when, for example, rechecking of interface relationships that should follow update analysis is forgotten by the developer. At the other extreme, the environment could rigidly enforce predefined combinations and sequences of analyses. A better alternative is to make the combinations and sequences of analyses a "programmable" aspect of the environment. This compromise would allow flexibility at the same time that it allows managerial control.

Another concern is deciding what information the developer is actually given as a result of a particular application of an analysis. As pointed out above, 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 generated by the analyses.

4. 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. Stated in the terminology of PIC, that form is governed by the following rule for submitting submodules for analysis (i.e., compilation).

[3] Tichy and Baker [9] discuss this in the restricted context of recompilation savings.

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.[4] 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 an abstract (i.e., *"private"*) type must appear in the specification part of a package even though that representation is logically hidden from the users 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 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-like design languages, which because of this restriction impede hierarchical 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 incremental 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 were present, yet subsequent to the analyses. Thus, to support incremental development, the standard view of compilation 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 PIC that permit it to fully support incremental development are (1) the formulation of analyses as specialized tools in the environment that are distinct from other activities (such as code generation), and (2) the availability of the *incompleteness construct* 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 given in Section 2. In that section, 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 2, **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 in PIC 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** to 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 **PINType**, 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).

At this point the question arises as to why a specification stub submodule is used and not simply a specification submodule with strategically placed *incompleteness constructs*. The answer is that while only one specification submodule of a module is permitted to exist (discounting multiple versions), 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 3. In terms of the automatic bank-teller example, there are at least two

[4] 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 ([4], 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.

```
package PINManager is
      request ESManager;
  type PINType is private;
  function Verify ( PIN : PINType; ... ) return Boolean
    provide to AutomaticTeller;
  procedure Issue ( ... )
    provide to OfficerInterface;
  MasterPIN : constant PINType
    provide to OfficerInterface;
  ...;
private
  type PINType is new ESManager.EncryptedStringType;
  MasterPIN : constant PINType := ...;
end PINManager;
```

Figure 5: Specification Submodule of Package PINManager.

modules that share PINManager: AutomaticTeller, whose view is shown in Figure 2, and a module that embodies the interface to bank officers, OfficerInterface, whose view of PINManager is shown in Figure 3. The consistency of these two specification stub submodules can be established using the stub/stub analysis.

Eventually, the "official" specification submodule of PINManager is made available (Figure 5). 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 in PIC that checks the consistency between AutomaticTeller 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 (analysis numbers 3 and 5 of Figure 4). The choice depends mainly upon whether the specification stub submodule sufficiently represents the use of PINManager by AutomaticTeller. Although not discussed here, 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.

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.

5. Concluding Remarks

The PIC environment is an experimental investigation of interface control and incremental development and their role in the software development process. Although our current version of these capabilities is oriented toward Ada and a somewhat traditional view of the software development process, we believe that interface control and incremental development are important no matter what particular model of that process one might adopt.

We are currently constructing a prototype version of the Ada-based version of the PIC environment. The environment's analysis capabilities are being implemented as small, self-contained analysis tools, so as to permit incremental analysis and order-independent development. The Odin component of the Toolpack system [7] provides a starting point for integrating these tool fragments to achieve higher-level analyses, such as those described in Section 3.

Implementation of the prototype PIC environment 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 environment will make to improving the software development process.

REFERENCES

[1] *Special Section on Computing in Space*, **Communications of the ACM**, Vol. 27, No. 9, September 1984.

[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] 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.

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

[5] 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.

[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] 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.

[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] W.F. Tichy and M.C. Baker, *Smart Recompilation*, **Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages**, New Orleans, Louisiana, January 1985.

[10] 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.

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

[12] 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.