Language and Tool Support for Precise Interface Control

Alexander L. Wolf

COINS Technical Report 85-23
September 1985

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

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1985

Computer and Information Science

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

© Alexander Lee Wolf 1985
All Rights Reserved

ACKNOWLEDGEMENTS

The problem with acknowledgement pages is that they are one dimensional, while personal and professional relationships are not. Be that as it may...

I thank my mentors, advisors, colleagues, and friends Lori A. Clarke and Jack C. Wileden for seeing me through my graduate career. I was fortunate in having two such able and willing people to learn from and work with. They complemented each other in the best possible ways: when one was a stickler for detail, the other kept an eye on the overall picture; when one was too busy, the other took up the slack. I can only hope that I have fulfilled the promise they saw in me.

I thank the other members of my committee, Robert M. Graham and George S. Avrunin, for carefully reading drafts of this dissertation.

I thank Laura K. Dillon for being my exemplary graduate student. Laurie started one month before me, earned her M.S. degree four months before me, and earned her Ph.D. degree one year before me. But then again, I didn't have two children during that time, as she did!

I thank my parents, Henry W. and Ilse R. Wolf, for teaching me—by their example and with their love—that although "hard work ain't easy", it is made *easier* with self-respect, a positive attitude toward life, and someone special to

share the pleasure and the pain.

I thank that someone special, Rachel M. Ball, my *sine qua non*. Rachel offered unfailing support at all the right times ... and delightfully irresistible distractions at all the wrong times! In the end, I could not have accomplished this without her.

Abstract

Language and Tool Support
for
Precise Interface Control

September 1985

Alexander Lee Wolf

B.A., Queens College, City University of New York

M.S., Ph.D., University of Massachusetts

Directed by: Associate Professor Lori A. Clarke
Associate Professor Jack C. Wileden

As software systems become larger and more complex, *interface control* becomes an increasingly important aspect of software development. Interface control is control over the interactions among entities in different software modules, where *entities* are those language elements that are given names, such as subprograms, data objects, and types. Unfortunately, existing languages typically permit the relationships among a software system's components to be described with only limited accuracy and tools capable of performing a thorough analysis of those relationships are seldom available. Even in relatively small programs, *precise* interface control could help to eliminate errors and greatly simplify maintenance.

In large software systems, stringent and accurate control over the interactions among entities in different modules is critical.

This dissertation undertakes a systematic treatment of the issues involved in providing improved support for interface control in large software systems. A formal model is defined to help characterize the concerns of interface control and provide a means for rigorously describing and evaluating interface control mechanisms. A new approach to interface control is developed that consists of both language features and analysis tools tailored to support a genuinely incremental view of the software development process. The language features constitute a framework from which precise interface control mechanisms can be derived. The framework provides a module structure that imposes a strict separation of interface control information from algorithmic detail and includes features that, in conjunction with this module structure, provide for precise interface control. While the syntax would certainly vary, the language features could be employed in most any language of the software lifecycle, from a graphical specification language to a textual implementation language. The descriptive capabilities of the language features are exploited by the analyses, which provide developers with feedback about the interface relationships of a software system. To illustrate this new approach to interface control, the dissertation presents an Ada-like design language that incorporates the language features and presents the detailed designs for the basic analyses appropriate to that language.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT v
LIST OF TABLES xi
LIST OF FIGURES xiii

CHAPTER

I. INTRODUCTION 1

§1. Principal Themes 3

§1.1 Generalization of "Visibility" 3

§1.2 Incremental Development 5

§2. Related Work 7

§2.1 Formal Models 8

§2.2 Interface Control Mechanisms 10

§2.3 Analysis Tools 13

§3. Plan of the Dissertation 15

II. PIC LANGUAGE FRAMEWORK 19

§1. Basic Features 20

§1.1 A Note on Types 24

§2. Features for Incomplete Descriptions 27

§3. Discussion 30

III. OVERVIEW OF INTERFACE CONTROL IN PIC/ADL 35

§1. Modules and Submodules 37

§2. Specifying Module Interfaces 38

§2.1 Provide Clause 39

§2.2 Request Clause 40
§2.3 Common Provide and Request Clauses 41
§2.4 Use Clause 41
§2.5 Clause Items 42
§3. Control Over Provided Types 44
§3.1 Basic Concepts 44
§3.2 Requisition Control 48
§3.3 Provision Control 49
§3.4 Operation Clause 52
§3.5 Discussion 55
§4. Incompleteness 57
§4.1 Specification Stub Submodule 57
§4.2 Incompleteness Construct 58
§5. Discussion 59
IV. FORMAL MODEL OF VISIBILITY 63
§1. Basic Definitions 64
§2. Describing Interface Control Mechanisms 67
§3. Evaluating Interface Control Mechanisms 72
§4. Discussion 77
V. NESTING: AN ANACHRONISM 79
§1. Practical Evaluation of Nesting's Control of Entity Visibility 80
§1.1 Common Programming Scenarios 82
§1.2 A Second(ary) Alternative 92
§2. Further Arguments Against Nesting 99
§2.1 Software Engineering Considerations 100
§2.2 Dynamic Memory Management 111
§2.3 Implementation Considerations 113
§2.4 Discussion 117

§3. Attempts to Enhance Nesting 118
§3.1 Interface Control Enhancements 119
§3.2 Readability Enhancements 125
VI. INTERFACE ANALYSIS 131
§1. Basic Interface Analyses 132
§2. Stub Analyses 138
§3. Update Analyses 141
§4. Consistency and Incompleteness 142
§5. Incremental Development in PIC 144
VII. CONCLUSION 151
§1. Summary 151
§2. Future Directions 153
§2.1 A Complete PIC Environment 153
§2.2 Extensions to the Language Framework and Analyses 161
BIBLIOGRAPHY 163
APPENDIX
A. PIC/ADL SYNTAX SUMMARY 173
B. REQUISITION/PROVISION PROBLEM-DETECTION ALGORITHMS FOR BASIC INTERFACE ANALYSES 187
§1. Overview 187
§2. The Algorithms 191
§2.1 Package BasicInterfaceProblems 191
§2.2 Package InternalRepUtilities 200
§2.3 Package AccessClauseUtilities 205

List of Tables

1. PIC/ADL Requisition/Provision Errors and Anomalies Detected by Basic Interface Analyses 135

2. Correspondence Between Basic Interface Analyses and PIC/ADL Requisition/Provision Errors and Anomalies 137

3. PIC/ADL Requisition/Provision Anomalies Detected by Spec/Stub Stub Analysis 140

4. Dictionary of Functions in Package InternalRepUtilities 189

LIST OF FIGURES

1. PIC/ADL Specification and Body Submodules of Package AutomaticTeller 22
2. PIC/ADL Specification Stub Submodule of Package PINManager Used by AutomaticTeller 29
3. PIC/ADL Specification Stub Submodule of Package PINManager Used by OfficerInterface 29
4. Specification Submodule of a Linked-list Package 47
5. Basic Levels of Provision Control Over Type Declarations 51
6. Specification Submodule of a Receipt-queue Package 52
7. A Visibility Graph 65
8. A Nested Program and its Representation as a Nesting Graph 70
9. Visibility Graph of a Common Programming Situation 75
10. Flawed Local Visibility Using Nesting 83
11. Local Visibility Using Nest-free PIC/ADL 85
12. Flawed Global Visibility Using Nesting 86
13. Global Visibility Using Nest-free PIC/ADL 88
14. Flawed Selective Visibility Using Nesting 90
15. Selective Visibility Using Nest-free PIC/ADL 91
16. Using the Parameter Method to Share an Object: Direct Caller/Callee Relationship 93
17. Using the Parameter Method to Share an Object: Indirect Caller/Callee Relationship 94

18. Using the Parameter Method to Share an Object: Independent Relationship . . . 95

19. Mutually Recursive Subprograms Sharing an Object Using Nesting and the Parameter Method . . . 98

20. A Tree-structured Call Graph, Which Also Serves as a Nesting Graph 101

21. Skeleton Textual Representation for Nesting Graph of Figure 20 102

22. Potential-call Graph for Nesting Graph of Figure 20 . . . 103

23. Modified Call Graph of Figure 20 . . . 104

24. Another Modified Call Graph of Figure 20 . . . 107

25. A Nesting Graph Realizing Calling Structure of Figure 24 . . . 107

26. Separate Compilation in a Nested Setting . . . 110

27. Separate Compilation in a Nest-free Setting . . . 110

28. Closed Scopes in Euclid . . . 120

29. Package Nesting in Ada . . . 123

30. Weakness in Package Nesting . . . 124

31. Textual Layout of Nested Program in Standard Pascal and Extended Pascal . . . 127

32. Ada's Subunit Facility Applied to Example of Figure 31 . . . 129

33. Basic Interface Analyses . . . 133

34. Specification Submodules of Packages ATMaintenanceInterface and OfficerInterface . . . 143

35. Specification Submodule of Package PINManager . . . 149

36. Conceptual Organisation of the Ada-based, Prototype PIC Environment . . . 154

37. A PIC/ADL Request Clause and its Internal Representation . . . 190

CHAPTER I

INTRODUCTION

As software systems become larger and more complex, *interface control* becomes an increasingly important aspect of software development. Interface control is control over the interactions among entities in different software modules, where *entities* are those language elements that are given names, such as subprograms, data objects, and types.

Even in relatively small programs, *precise* interface control could help to eliminate errors and greatly simplify maintenance. In large software systems, stringent and accurate control over the interactions among entities in different modules is critical. Indeed, specifying and analyzing the relationships among those entities is of primary importance throughout the software development process. For example, describing the major modules and their interactions is one of the first and most important activities undertaken during architectural or high-level design, while maintaining correct and consistent interfaces is an overriding concern during implementation and maintenance. An additional consideration is the fact that large software systems are often produced by teams of individuals, and different, interacting modules are typically developed by different teams or team members.

Under these conditions, insuring that only the intended interactions are actually effected is a formidable task.

Although the unique concerns related to producing large software systems have been repeatedly recognized, and some partial measures have been proposed, no systematic study of interface control and its role in the software development process has been carried out. The consequences have been that existing languages typically permit the relationships among a software system's components to be described with only limited accuracy and tools capable of performing a thorough analysis of those relationships are seldom available. 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, such as strictly bottom up.

This dissertation undertakes a systematic treatment of the issues involved in providing improved support for interface control in large software systems. A formal model is defined to help characterize the concerns of interface control and provide a means for rigorously describing and evaluating interface control mechanisms. A new approach to interface control is developed that consists of both language features and analysis tools tailored to support a genuinely incremental view of the software development process. The language features constitute a framework from which precise interface control mechanisms can be derived. The framework provides a module structure that imposes a strict separation of interface control information from algorithmic detail and includes features that, in conjunction with this module structure, provide for precise interface control.

While the syntax would certainly vary, the language features could be employed in most any language of the software lifecycle, from a graphical specification language to a textual implementation language. The descriptive capabilities of the language features are exploited by the analyses, which provide developers with feedback about the interface relationships of a software system. To illustrate this new approach to interface control, which we refer to as "PIC", since *p*recise *i*nterface *c*ontrol is the central concern being addressed, the dissertation presents PIC/ADL, an Ada-like design language that incorporates the language features, and presents the detailed designs for the basic analyses appropriate to that language.

§1. Principal Themes

The work presented here is guided by two principal themes: a generalization of the concept of visibility and broad support for incremental development. To a great extent, adherence to these themes is what distinguishes this work from the related work of others. Each theme is described in detail below.

§*1.1* *Generalization of "Visibility"*

Traditionally, interface control has been associated with the concept of *visibility*, which is defined in terms of *declaration*, *scope*, and *binding* (cf., [Schwanke, 1978]). A declaration introduces an entity and associates an identifier (name) with that entity. The scope of a declaration is the region of program

text over which that declaration is visible and has effect. Many languages allow a single identifier to be associated with more than one declaration and the scopes of those declarations to overlap. Binding relates the use of an identifier, at a given point in a program, to a particular declaration.

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

An interface control mechanism is the means for specifying requisition and

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

provision. The distinction between requisition and provision reflects the differences in the overall approaches to controlling entity visibility taken in different languages. In languages such as ALGOL60 and Pascal, requisition and provision are essentially mirror images; those entities requested by an entity are always also provided to that entity and vice versa. In the designs of more recent languages, particularly languages intended for the construction of large and complex software systems, the desire for greater control over entity visibility has resulted in mechanisms that address requisition and provision in separate, and often unequal, ways.

It is our contention that precise interface control mechanisms are of great potential value to developers and maintainers of large software systems. Such precision would permit the requisition and provision of exactly those accesses desired in a system while disallowing others. In addition, support for both precise requisition and precise provision can result in a redundancy that facilitates more rigorous analysis of the interface relationships of a system's components. For example, based on this view it is possible to formulate complementary descriptions of exactly how two modules are intended to interact, giving one description from the perspective of each of the modules, and then to analyze those interactions by checking the two descriptions for consistency.

§1.2 *Incremental Development*

The work described in this dissertation has been strongly influenced by a belief that software development environments must support incremental devel-

opment. That is, both languages and tools should be provided that facilitate the step-by-step manner in which large, complex software systems are most effectively developed. This 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 [Wileden & Clarke, 1984]. 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 interface control context, this means that as soon as interface control aspects of a module are specified, it should be possible to analyse 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 recognised 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.

§2. Related Work

Existing formal models of visibility, interface control mechanisms, and tools for analysing interface relationships suffer from a narrow view of interface control that does not account for both the requisition and provision aspects of entity visibility. Furthermore, existing interface control mechanisms and analysis tools do not provide capabilities that are adequate for use throughout the lifetime of a software system. This section reviews several of those existing formal models,

interface control mechanisms, and analysis tools. More detailed comparisons of the work presented in this dissertation to the related work of others appear within the subsequent chapters.

§2.1 *Formal Models*

The primary purpose of the formal model presented in this dissertation is to provide an effective means of describing interface control mechanisms so that one can reason about and evaluate those mechanisms. Existing informal and formal descriptive methods have proven inadequate. The Pascal Report [Jensen & Wirth, 1974], for example, causes many problems due to the ambiguity of its prose description of entity visibility [Welsh, Sneeringer, & Hoare, 1977; Brinch Hansen, 1981]. The few formal approaches to describing interface control mechanisms are operational in nature. Thus, they appear to be more suitable for the implementor and verifier than for the language designer or system developer. Such formal descriptions of interface control mechanisms have appeared primarily in operational and denotational semantic specifications, where a mechanism is typically described by the manipulation of an (identifier) environment component. The technique of employing an environment component in a formal description is unsatisfactory because the method for describing manipulation of that environment component is essentially algorithmic (despite the use of a "functional" notation; see, for example, [Honeywell, 1980]). Moreover, the information in the environment component only describes entity requisition. Employing such a description makes it difficult to understand the ramifications of using a mech-

anism. With nesting, for example, a subprogram's so-called "local" entities are unavoidably made visible to other subprograms nested within that subprogram, but this fact is only implicitly stated in existing formal descriptions of nesting.

The formal model presented here is based on a graph model of entity visibility. Graphs have been used elsewhere to describe concepts related to visibility. For instance, graphs are used informally for describing nesting's effect on data and control flow in Ada systems [Clarke, Wileden, & Wolf, 1980]. Thomas [Thomas, 1976] uses graphs more formally to analyze "resource information flow". The usefulness of Thomas's approach is restricted by its strong orientation to the particular module interconnection language developed in [Thomas, 1976]; it was never intended as a general, descriptive formalism. Moreover, it lacks the useful concept of provision. Lipton and Snyder [Lipton & Snyder, 1977] use a graph model to study a particular protection mechanism, the *take and grant* system, in which arcs in a graph are labeled with the access rights one node has to another. Although oriented toward control of access, the purpose of this model is to understand the effect of rewrite rules that dynamically add and delete nodes and arcs, and thus addresses a different problem domain. Ossher [Ossher, 1984] presents an extremely complicated, albeit general, graph model for describing entity relationships at multiple levels of abstraction, which was developed for specifying VLSI fabrication processes. While it would certainly be possible to recast that model to describe requisition and provision relationships, the complexity inherent in the model hinders its usefulness for our current purposes.

§2.2 *Interface Control Mechanisms*

Historically, the first non-trivial system structure is the one found in a language like FORTRAN. FORTRAN systems consist of independent, textually distinct subprogram modules. FORTRAN subprograms interact by invoking one another and by having access to common data areas called *common blocks*. Unfortunately, the interface controls in FORTRAN are very weak. In particular, all subprograms are (implicitly) requested by and provided to all other subprograms in a system except themselves, which means that any subprogram can potentially refer to (i.e., invoke) any other subprogram in that system. The situation is slightly better for common blocks. Each subprogram that desires access to an object in a common block must (explicitly) request access to that block—by naming and describing the block in its declaration part—before it can refer to the object. On the other hand, common blocks are (implicitly) provided to all subprograms in a system.

In an effort to improve upon FORTRAN's weak interface controls and primitive system structure, ALGOL60 introduced *nesting*, now the predominant interface control and system structuring mechanism in modern languages. Nested software systems consist of modules organized in a tree structure. The tree structure is represented by textually enclosing, or nesting, modules at lower levels in the tree structure within modules at higher levels.[2] In languages supporting nesting,

[2]While the term "nesting" is also commonly used to describe the embedding of statements within statements, such as nested if statements or nested loops, nesting of this sort does not concern us. Rather our concern is with the embedding of declarations that can result when modules are nested, and hence we use the term nesting only in this sense in the remainder of the dissertation.

from ALGOL60 to Ada, visibility depends upon the location of entities within a systems's nested textual structure. This approach is inadequate, however, for precisely describing the wide range of possible interface relationships among entities contained in a software system, since it usually results in weaker interface control than desired. Furthermore, nesting essentially forces a software system to take the form of a single, monolithic unit. For these and other reasons, nesting is inadequate for achieving some desirable software properties, such as information hiding, and is antithetical to others, such as readability and incremental development; chapters IV and V present detailed arguments against nesting. Thus, particularly for large, complex software systems, more versatile and powerful mechanisms for interface control are required.

A variety of languages have attempted to compensate for the inadequacies of nesting by offering supplemental or alternative mechanisms for interface control and system structuring. Such languages include the implementation languages Ada, Alphard, CLU, Euclid, Gypsy, LIS, Mesa, MODULA-2, NIL, and Protel [DoD, 1983; Shaw, 1981; Liskov et al., 1981; Lampson et al., 1981; Ambler et al., 1977; Ichbiah & Ferran, 1977; Mitchell, Maybury, & Sweet, 1979; Wirth, 1983; Strom & Yemini, 1983; Cashin et al., 1981], the specification languages OBJ and SPECIAL [Goguen & Tardo, 1979; Robinson & Roubine, 1977], the design languages EDL, PDL/Ada, and SDM/Ada [Rudmik, Casey, & Cohen, 1982; Sammet, Waugh, & Reiter, 1982; Privitera, 1982], and the module interconnection languages C/Mesa, INTERCOL, MIL75, and SML [Mitchell, Maybury, & Sweet, 1979; Tichy, 1979; DeRemer & Kron, 1976; Schmidt, 1982].

Most of these languages have relied, to a greater or lesser degree, on the concepts of *encapsulation module* and *explicit import/export control* to describe both the accesses that are requested and the accesses that are provided by the entities in a module of a software system. In its most general form, which is not exactly the way it is used in all of these languages, an encapsulation module serves to group related subprograms, objects, types, and even other encapsulations. Explicit import/export control furnishes the means by which a module requests and provides access to external entities for its constituent entities. In Ada, for example, the encapsulation module is the *package* and import/export control may be effected through a combination of features including *with clauses*, *visible* and *private parts*, and *nesting*. Unfortunately, not one of the languages mentioned above supports precise and flexible control over both what accesses an entity can request and what accesses an entity can provide; the formal model presented in Chapter IV is used to expose these weaknesses. For instance, Ada's with clause only permits the requisition of access to either no entities or all entities in the visible part of a package and Ada's private/visible mechanism only permits the provision of access to either no entities or all entities in the scope of a package.

Gypsy and NIL are notable exceptions to the approach taken by most modern languages. While there is no nesting in Gypsy, there is also no concept of encapsulation module; entities can "exist" (i.e., be declared and compiled) independently. Access to entities in Gypsy is controlled exclusively through so-called *access lists*, which are attached to the entities and which precisely specify the provision of those entities; there is, however, no control over requisition. NIL is

representative of the *dynamic* approach to configuring software systems. This approach is mostly used in programming distributed and concurrent systems, where the primary encapsulation module is the sequential "process" and the concept of explicit import/export control is replaced by some sort of "capability" mechanism. Using NIL, the interconnections among processes can be made to change as a system executes. The language features presented in this dissertation are based upon the static approach to controlling module interactions, building upon the encapsulation module and import/export concepts of languages like Ada.

§2.3 *Analysis Tools*

Cross-reference information, which is essentially a catalogue of entity declarations and references, has been available to system developers since the early FORTRAN compilers. Typically, however, cross-reference tools are only designed to extract this information "after the fact"—that is, once a system has been developed. The result is that the information cannot be easily fed back into the development process. The main reason for this limitation is that language processors (e.g., compilers for implementation languages) are monolithic; cross-referencing is bound up with other language processing activities, such as semantic analysis and code generation. The work described in this dissertation attempts a distillation of interface analysis from other language processing activities by fashioning analyses from individual *tool fragments* [Osterweil, 1983].

The Masterscope tool of the Interlisp environment [Teitelman, 1978] successfully integrates a separate cross-reference tool with a text editor and database.

The result is the ability to change a module—that is, a function subprogram—in a system (through the editor), automatically reanalyze the changed module (through the cross-reference tool), automatically determine what other modules are affected by the change (through the database), and then perhaps change the module or affected modules based on the information provided by the analysis. Modules affected by a change are automatically marked as requiring reanalysis. Therefore, Masterscope appears to solve the problem of feeding cross-reference information back into the development process. Much of the apparent power of Masterscope, however, is derived from the simple fact that it is operating on an interpreted, functional language that does not have the concept of encapsulation module and in which there are only a small number of possible kinds of module interactions. Moreover, Interlisp provides very limited control over those few kinds of interactions.

As illustrated by Interlisp's Masterscope tool, the various features provided in a language influence not only the precision possible in interface description but also the complexity of analyzing interface relationships. For instance, analysis techniques developed for systems described in nested languages such as ALGOL60 and Pascal are quite straightforward, largely because those systems are monolithic and the controls provided in the languages are quite limited. As mentioned above, more recently-designed nested languages, such as MODULA-2 and Euclid, furnish additional interface control features in an attempt to compensate for the inadequate controls of nesting. Still other features are supplied in some new nested languages, such as Ada and GTE's CHILL [Rudmick & Moore, 1982],

to support aspects of incremental development. The combination of nesting and these additional interface control and incremental development features complicates the analysis techniques applicable to those languages (see, for example, [Holt & Wortman, 1982; Moore & Chandrasekharan, 1983]). The absence of nesting in languages such as Gypsy and CLU, which also support incremental development, allows for simpler, but no less powerful, analysis techniques. Our approach to interface control has its own unique, and interesting, ramifications for the design of analysis techniques. In particular, the language framework's added expressive power makes possible the precise description of intended interface relationships and hence raises the prospect of more revealing analyses. Furthermore, the language framework's explicit support for incompleteness causes a reexamination of the traditional meaning of *consistency* in interface relationships. Finally, attempting to fashion analyses from individual tool fragments introduces questions of how to integrate and organize those fragments so that particular analyses, or combinations of analyses, can be flexibly applied as desired.

§3. Plan of the Dissertation

This dissertation reports on a systematic study of the issues surrounding interface control. Chapters II and III respectively present and then illustrate the PIC language framework. In Chapter II, the basic features for separating interface control information from algorithmic detail and precisely specifying requisition and provision are discussed. The features for describing incomplete systems are

also given. As a concrete example of a language that incorporates the features of the framework, the design language PIC/ADL is summarized in Chapter III. The complete syntax for PIC/ADL is given in Appendix A. PIC/ADL is used throughout the dissertation to depict various aspects of the PIC approach. Chapter III and Appendix A should therefore be used as the primary references when questions concerning the syntax or semantics of PIC/ADL examples arise.

Chapter IV introduces a formal model for describing and evaluating interface control mechanisms that captures the generalized visibility concepts of requisition and provision. As an example of the utility of the model, a portion of the nesting mechanism found in ALGOL60 is formally described. Four mechanisms, including ALGOL60's nesting and the approach advocated in this dissertation, are then evaluated with respect to several formally defined properties.

Chapter V complements the formal evaluation of nesting presented in Chapter IV with a pragmatic look at the mechanism. It is argued that given the more flexible and powerful concepts of encapsulation module and explicit import/export control, nesting is an anachronism. The chapter contains a number of practical demonstrations of nesting's weaknesses and dispells several myths concerning its supposed indispensability. Nesting is given such a close examination in this dissertation because of its historical position as the foremost interface control and system structuring mechanism in modern languages.

Chapter VI describes the analysis component of the PIC approach to interface control. The chapter begins with the presentation of three classes of interface analyses; Appendix B gives algorithms, described in PIC/ADL, used by the most

important of these three classes. The manner in which the analyses account for incompleteness is then discussed. Finally, PIC's support for incremental development is demonstrated. The analyses are illustrated in this chapter by showing how they would be applied to systems described in PIC/ADL.

Chapter VII summarizes the results of the dissertation and discusses possible enhancements to, and future applications of, the formal model, language features, and analyses.

CHAPTER II

PIC LANGUAGE FRAMEWORK

This chapter introduces a small set of language features for describing the interface relationships of a software system's modules. The features constitute a *framework* from which precise interface control mechanisms can be derived. This framework contributes to the goal of supporting incremental development by providing a consistent abstraction of interface control and by providing features that account for missing and incomplete modules. While the syntax would certainly vary, the language features could be employed in most any language of the software lifecycle, from a graphical specification language to a textual implementation language. The graphical language used in [Wolf, Clarke, & Wileden, 1985a], for example, provides the language features as labeled boxes and arrows. The Ada-like design language PIC/ADL, which is summarized in Chapter III, provides textual constructs corresponding to the language features.[1]

The next section introduces the basic features of the PIC language framework. Features for describing incomplete systems are then presented. The chapter concludes with a discussion of the benefits of the PIC language framework.

[1] Throughout this dissertation, the terms "feature" and "construct" are used distinctly; "feature" refers to a language concept and "construct" refers to a concrete realization of a language concept in a particular language.

Throughout this chapter, the language features are illustrated using constructs of PIC/ADL that are detailed in the following chapter. It is important to remember that those constructs, whose forms were chosen to keep within the spirit of Ada, are particular realizations of the language features; a different set of constructs derived from the framework might be appropriate for use in other languages.

§1. Basic Features

There are three aspects to the PIC language framework. First, the framework provides a system structure that results in systems that are collections of nest-free modules. (Arguments against the use of nesting are given in chapters IV and V.) 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 framework provides 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 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, the body takes the form of a design-language description, while for implementation languages it consists of implementation-language code.

Figure 1 presents an example developed in PIC/ADL that illustrates the ba-

sic features of the PIC language framework. The example shows the specification and body submodules of an encapsulation module AutomaticTeller, which is one component of a hypothetical automatic bank-teller system. An encapsulation module in PIC/ADL is called a *package* after its Ada namesake. 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 of the hypothetical 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 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

```
package AutomaticTeller is
    request CommonCodes;

  procedure BeginSession ( ... )
    provide to CustomerInterface
    request PINManager.( PIN, Verify ),
            AccountManager.( Account, 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: PIC/ADL Specification and Body Submodules of Package AutomaticTeller.

BeginSession, since BeginSession is the only entity in AutomaticTeller that has an attached request clause mentioning those entities. Similarly, the entity Credit, also defined in package AccountManager, is only to be referred to by procedure Deposit. 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 BeginSession indicates that BeginSession is only provided to the entities of CustomerInterface, whereas the provide clause appended to function RemainingCash indicates that RemainingCash is only provided to the entities of ATMaintenanceInterface and OfficerInterface.

The request clause is more precise and flexible than its counterparts in most other languages, such as 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 other languages. In Ada, for example, provision is controlled on an all-or-nothing basis; either access to an entity is provided to every module (in a given scope), or it is provided to no module, and so the entity is hidden. While these two extremes are useful (for instance, in describing the 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), the intended provision of a particular entity often lies somewhere in

between [Minsky, 1983]. The PIC language framework's provide clause may be appended to any of a module's entities in order to selectively limit their provision to specific non-local entities.

§1.1 *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, many modern languages, including Ada, CLU, and Mesa, allow 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 and type constructors. Additionally, the definition of a new type involves the association of a set of operations with the type.[2] 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 properties 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

[2]An "operation" of a type is a subprogram having a parameter and/or return type that is of the type. Languages typically provide special, non-subprogram-like syntax, however, for invoking certain basic operations, such as assignment or record-field selection. Although the invocations of those operations do not look like ordinary subprogram invocations, the definitions of the operations can still be treated as subprogram declarations.

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 framework supports a more general and precise approach to interface control for types and their operations than is found in existing languages. In the PIC language framework, access to the name of a type is 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 [Ullman, 1980]—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, deposit and withdraw 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 deposit, withdraw, and report-balance operations, whereas entities representing actions performed by bank employees might have the full view of the abstraction. The PIC language framework allows 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 framework, because it supports both precise requisition and precise provision, allows such views to be specified from either or both the requisition and provision perspectives. More complete examples of control over types and associated operations are given in Chapter III.

§2. Features for Incomplete Descriptions

The PIC language framework provides extensive support for incomplete descriptions of a system's modules and their interactions through two features: the *incompleteness feature* and the *specification stub submodule*. These features 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 missing 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, 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 re-

quires 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 OfficerInterface, respectively.[3] As shown in Chapter VI, 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 Chapter VI, is used to assure consistency among the stubs, to generate an accumulated view, and to

[3]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.

```
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 PIN-Manager Used by AutomaticTeller.

```
package stub PINManager is
      used by OfficerInterface;

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

   ...;

end PINManager;
```

Figure 3: PIC/ADL Specification Stub Submodule of Package PIN-Manager Used by OfficerInterface.

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 in the PIC language framework 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 (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 Chapter VI.

§3. Discussion

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

readability of software with these features is enhanced, making it easier to discern the relationships among the modules. This in turn makes systems easier to change and therefore easier to develop and maintain.

Additional benefits accrue from consolidating interface control information into the specification submodules and separating a module's interface specification from its body. While in languages such as Ada, Mesa, and MODULA-2 there is support for "specifications" of modules separate from their bodies, these specifications do not completely define the interfaces to modules. In Ada, for instance, a body may itself import entities using an attached with clause. In the PIC language framework, the request clauses, which can appear in a specification submodule but not in a body submodule, completely constrain the non-local entities that can be referred to by a module's body. This separation of concerns effectively results in a *module interconnection language*—one that is in fact more precise than existing ones—where the interface control component of a system (i.e., the specification and specification stub submodules) can be viewed as a description in this language. Moreover, by enforcing a complete separation of concerns, the language features facilitate incremental development, managerial control, and information hiding.[4]

As Ada and other modern languages have demonstrated, the separation of specification and implementation concerns fosters incremental development (i.e.,

[4]Some module interconnection languages, such as INTERCOL and SML, also provide support for *version control*, which is another complementary concern to interface control. While this important capability is not currently supported in the PIC approach, the approach is certainly amenable to its inclusion.

incremental analysis and separate compilation) of large software systems. By refining this separation and adding features for precise interface control and incomplete descriptions, the PIC language framework supports incremental development by permitting more meaningful and detailed interface consistency analysis to be performed as well as allowing this analysis to be done early and throughout the software lifecycle. In particular, the interface control component of a system can be created, modified, and analysed separately from the bodies (i.e., implementations) of the modules in that system and, in fact, only needs to be combined with the bodies to facilitate further analysis or to support separate compilation. Moreover, the language features' explicit treatment of incompleteness permits module development to proceed in any desired order. This contrasts strongly with, for example, Ada's rigid method for incremental development of library units, where primarily for code generation reasons a bottom-up development process is enforced (see Chapter VI).

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

can assume the role of project leader and construct their own specification submodules.

Finally, information hiding is also facilitated by the separation of concerns supported by the PIC language framework since a developer working on a module that will refer to entities from another module only needs to see the specifications of the provided entities of the referred module and each such specification only needs to contain information relevant to the referring module. The request and provide clauses actually define the different views particular external modules have of a package's provided entities. Conceivably, a processing tool could be developed to automatically extract such views and display them to developers.

In sum, while many existing specification, design, programming, and module interconnection languages support some of the desired capabilities for expressing interface relationships, none supports all. The language features outlined here incorporate capabilities distilled from many of these previous attempts. The resulting language framework is relatively simple and straightforward, yet surpasses these previous attempts by supporting precise interface control as well as the comprehensive collection of benefits outlined above. The following chapter describes a concrete realization of the language features, demonstrating how the framework would be incorporated into a complete language.

It should be pointed out that the design of the PIC language framework was conceived in the context of sequential, procedural languages providing interface control mechanisms that statically determine interface relationships. While this class of languages clearly is still the predominant one used for the construction of

large software systems (witness FORTRAN and COBOL), recent developments portend shifts in several other directions. One is toward the inclusion of constructs for expressing distributed and/or concurrent execution. We anticipate that in many cases the framework we have developed for sequential systems will also suffice for distributed and/or concurrent ones. In other instances, particularly if a system can spawn processes dynamically, additional features may be required. Another direction is toward the use of functional languages, which until recently were employed only for building research prototypes but are now beginning to be used in the development of true software products. The primary difference between functional and procedural languages is in the control structures they offer [Ralston, 1983].[5] Therefore, we suspect that the language framework can easily be made to accommodate its use with functional languages, since the framework is concerned with the module-interface level of systems and not with the algorithmic-control level. Detailed investigation of the framework's applicability to non-sequential and non-procedural languages is an important, and future, topic of study.

[5]Until recently there was another important difference between functional and procedural languages: functional languages historically provided dynamic interface control mechanisms, whereas procedural languages relied on static ones. The designs of new functional languages, particularly those intended for production use, such as the proposed standard Common LISP [Steele, 1984], now provide static interface control mechanisms.

CHAPTER III

OVERVIEW OF INTERFACE CONTROL IN PIC/ADL

PIC/ADL is an Ada-like design language, incorporating constructs that correspond to the language features of the PIC framework. Nearly every feature of Ada is present in the language, including Ada's lexical, type, expression, and statement structures, as well as its overloading and exception-handling mechanisms. Not found in PIC/ADL is Ada's nesting structure, which we feel is an anachronism (see Chapter V). In addition, tasking, generics, and representation specifications are not currently supported; inclusion of these features (especially that of tasking) is planned as future work.

Overall, the design of PIC/ADL was guided by a desire to keep the language in harmony with Ada. This task was relatively simple in the areas of the design where there is a clear mapping from the features of the PIC language framework to the constructs of Ada. For example, Ada already provides the basis for a suitable module structure, with its separate specification and body parts, that requires only a moderate amount of refinement. The task of keeping the design in harmony with Ada was more difficult in other areas, particuarly interface control, where Ada is limited in its support. Of course, compromises in the realization of

some language features had to be made that might not be considered appropriate, or even necessary, for other languages. These compromises are pointed out below.

This chapter describes those aspects of PIC/ADL that are relevant to interface control. The purpose of the chapter is twofold: (1) to illustrate how the PIC language framework can be concretely realized in a full language, and (2) to serve as a reference for examples used in this dissertation. Therefore, the chapter straddles the line between a design rationale and a reference manual, providing both motivational examples and detailed syntax rules. Explanations of the underlying concepts, which are presented in Chapter II, are not repeated. Because of PIC/ADL's similarity to Ada, the discussion below utilizes a substantial amount of terminology that was established for Ada in the Ada Language Reference Manual [DoD, 1983] and includes many comparisons with constructs found in Ada; the reader is assumed to be at least somewhat familiar with Ada's basic constructs and terminology.

The first section describes the overall system and module structures of PIC/ADL. The primary constructs for specifying module interfaces are presented in the next section. Following that, the special constructs for controlling provided types are discussed. The constructs for describing incomplete systems are then presented. The chapter concludes with a discussion of several general characteristics of PIC/ADL. Relevant portions of the PIC/ADL syntax are given at the beginnings of (sub)sections introducing particular constructs. The complete syntax appears in Appendix A, which also details the modified Backus-Naur Form used here as the syntax description language.

§1. Modules and Submodules

```
submission ::= {submodule}

submodule ::=
      package_specification          | subprogram_specification
    | package_specification_stub     | subprogram_specification_stub
    | package_body                   | subprogram_body

package_specification ::=
    package identifier is
      [ common_access_clauses [use_clause]; ]
      { provided_declarative_item }
  [ private
      { hidden_declarative_item } ]
    end package_simple_name;

subprogram_specification ::= subprogram_heading [access_clauses];

subprogram_heading ::=
      procedure identifier [formal_part]
    | function designator [formal_part] return type_mark

designator ::= identifier | operator_symbol
```

A PIC/ADL software system is a collection of nest-free modules, of which there are two kinds: *packages* and non-packaged *subprograms*. A package encapsulates declarations of subprograms (functions or procedures), objects, and types. A non-packaged subprogram is a syntactic shorthand for a package containing only a single subprogram and avoids the need to create a superfluous package to hold that subprogram.

A module can be associated with three kinds of submodules: a *specification submodule*, a *body submodule*, and one or more *specification stub submodules*. The submodule is the (logical) unit of submission to a PIC/ADL processor. A specification submodule of a package describes the entities encapsulated by the

package, while a specification submodule of a subprogram simply describes the information needed for an invocation of the subprogram. The body submodule for both packages and subprograms contains the actual code sections realizing the module. The specification stub submodule is a special kind of specification submodule that is used in place of a (missing) specification submodule. While there can only be one specification submodule of a module (discounting multiple versions), there may be several specification stub submodules. In the rest of this chapter, details of specification and specification stub submodules are given in terms of specification submodules alone. Where there are differences between the two kinds of submodules, those differences are pointed out.

§2. Specifying Module Interfaces

A package specification is divided into three parts. The optional first part may contain common request and provide clauses, as well as a use clause, as explained below. The second and third parts, following Ada, textually separate provided entities, in the *visible part*, from hidden entities, in the optional *private part*. The private part begins with the reserved word **private**. Both the provided and hidden entities are available to all other entities in the defining package, but only the provided entities are available outside of the package.

§2.1 *Provide Clause*

---

provide_clause ::= **provide** [**only**] **to** **provide_item** {, **provide_item**}

The textual separation of provided and hidden entities in a package specification is a first approximation of control over provision. Precise control is achieved by attaching a *provide clause* to an entity in the visible part; the provide clause lists the entities to which that entity is provided. A non-packaged subprogram is considered a provided entity and so the provide clause can also be applied to such a subprogram. In the absence of a provide clause, a provided entity is by default provided to "all". (The hidden entities can be thought of as having attached provide clauses specifying provision to "none".) A special form of provide clause is used with provided type declarations (see below).

A provide clause that includes the reserved word **only** indicates that the associated entity is provided *exclusively* to the listed entities. For specification submodules, this is redundant information; since there can only be one specification submodule (disregarding multiple versions), there can only be one submodule's provide clauses that apply to the entity. Conversely, since there can be several specification stub submodules of a module, it is possible that there may be provide clauses in different submodules that apply to the same provided entity (e.g., the entity PIN in the two specification stubs of PINManager in figures 2 and 3). The presence of the reserved word **only** in a provide clause of a specification stub submodule 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.

§2.2 *Request Clause*

```
request_clause ::= request request_item {, request_item}
```

The *request clause*, which is used to control requisition, can be attached to entities in both the visible and private parts, as well as attached to non-packaged subprograms. A request clause lists the entities requested by the entity to which it is attached. Requisition of an entity that is a provided type has a special form (see below).

As a notational shorthand in PIC/ADL, 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 unless there is a request clause already attached to that declaration, in which case the (explicit) request takes precedence. (Note that the mention of an entity in a provide, request, or use clause is not considered a "reference" to that entity.) For example, the object declaration

```
X : SomePac.SomeType;
```

appearing in a specification submodule is equivalent to the declaration

```
X : SomePac.SomeType
        request SomePac.SomeType;
```

§2.3 *Common Provide and Request Clauses*

---

access_clauses ::= request_clause [provide_clause] | provide_clause [request_clause]

Requests for entities that are common to all the entities in a package can be collected together into a request clause appearing in the first part of a package's specification submodule (e.g., the request for the entities of package CommonCodes in Figure 1). Similarly, if the provided entities of a package are all provided to some common set of entities, then that set of entities can be listed in a provide clause appearing in the first part of the package's specification submodule (e.g., the provision of entities to AutomaticTeller in Figure 2). The entities listed in these *common* clauses are logically considered to be (additional) items of individual clauses attached to the entities of a package. Note, however, that such clauses do not apply to any *implicitly* declared entities. In particular, common request and provide clauses do not apply to the inherited operations of a provided type declaration (see below).

§2.4 *Use Clause*

---

use_clause ::= **use** request_item {, request_item}

The *use clause* allows entities requested from a package to be referred to by their simple names—that is, without having to prefix the name of their defining package using the "dot" notation. Paralleling the precision of the request clause,

individual packaged entities can be listed in a use clause and then only those listed entities of the package can be referred to by their simple names.

The use clause can appear in the first part of a package specification submodule, as the first item in the declarative part of a body submodule, or as the first item in the declarative part of a packaged subprogram's body. The effect of a use clause only extends over the submodule or packaged subprogram body in which it appears. This differs from Ada, where a use clause appearing in a specification submodule also has meaning for the corresponding body submodule. The motivation for this difference comes about from our desire to support incremental development. In particular, we make no *a priori* assumption (or restriction) about the order in which the submodules of a system are analyzed; a module's body submodule may be analyzed independently from its specification submodule (see Chapter VI). To gain the most information from an analysis that involves a body submodule, however, we need to be able to identify references to non-local entities. This identification must be possible, therefore, without depending upon information contained in the specification submodule.

§2.5 *Clause Items*

```
provide_item ::=
        identifier          | package_identifier...  | ...
    |   ...simple_item      | package_identifier.simple_item
    |   ...(simple_item_or_ellipsis {, simple_item_or_ellipsis})
    |   package_identifier.(simple_item_or_ellipsis {, simple_item_or_ellipsis})

simple_item_or_ellipsis ::= simple_item | ...

simple_item ::= operator_symbol | subprogram_signature | identifier
```

```
request_item ::=
        identifier                  |  package_identifier... [private] [operation_clause]  |  ...
    |   ...request_simple_item      |  package_identifier.request_simple_item
    |   ...(request_simple_item_or_ellipsis {, request_simple_item_or_ellipsis})
    |   package_identifier.(request_simple_item_or_ellipsis {, request_simple_item_or_ellipsis})

request_simple_item_or_ellipsis ::= request_simple_item  |  ... [private] [operation_clause]

request_simple_item ::=
    operator_symbol  |  subprogram_signature  |  identifier [private] [operation_clause]
```

The items that can be listed in a provide, request, or use clause include the names of packages, individual packaged entities, and non-packaged subprograms. Moreover, a subprogram's complete signature, which includes a subprogram's name, formal parameters (if any), and return type (if any), can be given to disambiguate among overloaded subprogram names. (In the syntax specifications given above and in Appendix A, the signature of a parameterless procedure is not recognised as subprogram_signature, but rather as identifier.) Finally, an item in a provide, request, or use clause can be an ellipsis ("..."), which is used to indicate incomplete information (see below). Notice that when an ellipsis immediately precedes or follows the selection operator ".", it is shortened to "..".

Mentioning a package name, without detailing the particular entities within that package, is a notational shorthand. For provide clauses, mention of a package name implies provision to all the package's entities. For request clauses, mention of a package name implies a request for all the provided entities of the package. For use clauses, mention of a package name implies all the entities that are requested from that package.

§3. Control Over Provided Types

```
provided_full_type_declaration ::=
        type identifier [discriminant_part] [common_operation_clause] [type_provide_clause]
            is type_representation
                [common_operation_clause] [type_provide_clause] [request_clause]
    |   type identifier [discriminant_part] [common_operation_clause] [type_provide_clause]
            is private
```

A *provided type* is a type declared in the visible part of a package's specification submodule. Special facilities are available for controlling provided types and associated operations in order to support the definition and control of data abstractions. These facilities expand upon and generalize those already found in Ada. The conceptualization of provided types, associated operations, and access to types and operations given in [DoD, 1983] is inadequate for describing PIC/ADL's more general facilities. Therefore, a refined conceptualization is given below, followed by details of the relevant PIC/ADL constructs.

§3.1 *Basic Concepts*

A provided type declaration consists of two parts: a name and a representation (called a *definition* in Ada). The representation part, which begins with the reserved word **is**, indicates the type from which the newly declared type's values and operations are to be inherited. An inherited operation is an operation originally associated with the representation type but made applicable to the new type. To be consistent with Ada, inherited operations are considered to be implicitly declared immediately after the type declaration. (Recall that common

provide clauses do not apply to such implicit declarations.) Additional operations associated with the type can be created by declaring—within the visible part containing the type declaration—subprograms that have parameters and/or return types of the type.[1] If the signature of such a newly defined operation is the same as that of an inherited operation, then the inherited operation is hidden by the new operation.

Operations

Five classes of operations are defined in Ada (see [DoD, 1983, Section 3.3.3]).

- *Basic Operations.* These operations include membership tests, attributes, conversions, component selections, etc. Different kinds of types may have different basic operations. For example, structured types have component selection operations, whereas scalar types do not.

- *Predefined Operations.* These include relational, logical, numeric operations, etc. Different kinds of types may have different predefined operations. For example, numeric types have numeric operations, whereas boolean types have logical operations. Most of the predefined operators can be overloaded by explicitly defined operations (see [DoD, 1983, Section 4.5]).

- *Enumeration Literals.* These are parameterless functions that return a value of an enumeration type.

[1]In Ada, subprograms declared in places other than the same visible part that have parameters and/or return types of the type are also considered operations of the type. Such operations can be ignored in this discussion.

- *Explicitly Defined Operations.* These are subprograms, declared within the visible part containing the declaration of a type, having parameters and/or return types of the type.

- *Derived Operations.* These are the explicitly defined operations inherited from a representation type and (transitively) the derived operations inherited from that representation type.

Note that the operations collectively referred to above as the *inherited operations* are those operations in classes other than the class of explicitly defined operations.

The classification of operations in PIC/ADL differs from the Ada classification in one important respect: Assignment is not considered a member of the basic operation class. This is because assignment is treated differently from the other basic operations, as explained below. Assignment can be considered the only member of a special, sixth class of operation.

The operations associated with a type are characterized by the specific part of the type declaration to which they are associated. In particular, assignment, the equality/inequality and membership tests, component selection for discriminants, explicit conversions, certain attributes,[2] and the explicitly defined operations are all associated with the name part. The remaining basic and predefined operations, the derived operations, and the enumeration literals are all associated with the representation part. This characterization of operations is based on the following criterion: If an operation is involved in manipulating an object as though it were

[2]Base, Constrained, etc. (see [DoD, 1983, Section 7.4.2])

```
package LinkedList is
   type List
      is record ... end record;

   procedure Insert ( ...; L : in out List );
   procedure Remove ( ...; L : in out List );
   procedure SetVal ( ...; L : in out List );

   ...;

end LinkedList;
```

Figure 4: Specification Submodule of a Linked-list Package.

an object of the representation type, then it is associated with the representation part, otherwise the operation is associated with the name part. The purpose of segregating the operations is to mimic the control over access to operations that is available in Ada (see below).

As an example, consider the package LinkedList, which contains declarations for a provided type, List, and three provided subprograms, Insert, Remove, and SetVal (Figure 4). The operations associated with the name part of List are assignment, the membership and equality/inequality tests, explicit conversions, various attributes, and the explicitly defined operations Insert, Remove, and SetVal. (Component selection is not defined for the name part of List, since the type declaration does not contain any discriminant components.) The operations associated with the representation are those remaining operations appropriate to record types, such as component selection (see [DoD, 1983, Section 3.7.4]).

Access to Types and Operations

Access to an operation of a provided type affords the right to invoke the operation and to inherit the operation as part of a new type declaration. Access to the name of a type affords the right to create objects of the type and to use the type as a representation type in a type declaration. To be consistent with the control available in Ada, access to the name of a type also affords access to the inherited operations associated with the name part and access to the representation of a type affords access to the operations associated with the representation part. Access to the explicitly defined operations associated with the name part can be controlled separately using the normal request and provide clause mechanism (i.e., through provide clauses attached to the operations' declarations and request clauses mentioning those operations). An entity that has access to the name, but not the representation, of a type views the type as a so-called *private type*.

§3.2 *Requisition Control*

Requisition of an entity that is a type is a request for access to the name of the type. Additionally, requisition of a type is a request for all the inherited operations associated with the type unless the reserved word **private** follows the type's name in the request, in which case the request is only for the inherited operations associated with the name part of the type. Requests for explicitly defined operations associated with the type can be made independently of the

request for the type and inherited operations. For example, the request clause

request LinkedList.(List **private**, Insert, Remove, SetVal)

specifies a request for provided type List, implies requests for the inherited operations associated with the name part of List, such as assignment, and requests three explicitly defined operations. Thus, the entity requesting List views List as a private type, since it is not requesting access to the operations associated with the representation part.

§3.3 *Provision Control*

```
type_provide_clause  ::=  provide [only] to type_provide_item {, type_provide_item}

type_provide_item  ::=
        identifier [operation_clause]   |  package_identifier... [operation_clause]
     |  ... [operation_clause]          |  ...simple_item [operation_clause]
     |  package_identifier.simple_item [operation_clause]
     |  ...(simple_item_or_ellipsis [operation_clause]
                                        {, simple_item_or_ellipsis [operation_clause]})
     |  package_identifier.(simple_item_or_ellipsis [operation_clause]
                                        {, simple_item_or_ellipsis [operation_clause]})
```

Provision of an entity that is a type involves the separately controlled provisions of the name, representation, and explicitly defined operations. Provision of explicitly defined operations is simply controlled through provide clauses attached to those subprograms. To control provision of the name and representation, two provide clauses can be associated with a provided type declaration, one referring to the provision of the name and the other referring to the provision of the representation. A common provide clause, which can appear in the first part of a

package's specification submodule, applies only to the name of a type. Access to the name of a type is, of course, necessary for any sort of use of the type. Therefore, a provide clause associated with the representation serves as a further restriction on the representation beyond that inherited from the name; the entities to which the representation is provided must be a subset of the entities to which the name is provided. As in Ada, the case of a type's representation being completely hidden within the defining package—that is, when no non-local entity is provided access to the representation—is denoted simply using the reserved word **private** in place of the representation; the representation of that type is then given in the package's private part. As shown in Figure 5, six basic levels of control result. Note that Ada directly supports only the first and third levels shown in the figure. Although in some cases the other levels can be approximated in Ada with careful nesting of packages, nesting is not general enough to capture these levels in every situation.

As an example of the provision control over types available in PIC/ADL, consider a package ReceiptQueue, which queues items for printing on the receipt that is generated for a customer by an automatic teller. The specification submodule of this package is shown in Figure 6. The package, which is implemented using the linked-list package of Figure 4, provides a type for receipt queues, Queue, that is derived from type List. The package also provides two subprograms, Enqueue and Dequeue, that realize the abstract operations of adding and removing an item from a receipt queue. The name of type Queue is provided without restriction, but provision of access to its representation is limited to module Reorder, which

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

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

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

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

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

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

Figure 5: Basic Levels of Provision Control Over Type Declarations.

```
package ReceiptQueue is
  type Queue
    is new LinkedList.List provide to Reorder
      request LinkedList.( List private, Insert,  Remove,  SetVal );

  procedure Enqueue ( Item : in ...; Q : in out Queue );
  procedure Dequeue ( Item : out ...; Q : in out Queue );

  ...;

end ReceiptQueue;
```

Figure 6: Specification Submodule of a Receipt-queue Package.

can reorder the items on a queue.[3] From the point of view of most clients of ReceiptQueue, therefore, type Queue is a private type that essentially has two operations associated with it. From the (privileged) point of view of Reorder, however, a receipt queue can also be manipulated using linked-list-like operations inherited from List.

§3.4 *Operation Clause*

---

operation_clause ::= **with** <[simple_item_or_ellipsis {, simple_item_or_ellipsis}]>

As described so far, access to a type's name or representation implies access to *all* the inherited operations associated with that name or representation. The *operation clause* is used to precisely control access to those operations, specify-

[3]Recall that entities declared in a package are available to all other entities declared in that package. Hence, the inherited linked-list operations, which are implicitly declared in package ReceiptQueue and which are defined to operate on objects of type Queue, are also available to procedures Enqueue and Dequeue.

ing the individual inherited operations that are requested or provided. The only exceptions are access to the basic operations and enumeration literals. In particular, an entity is given access to either all the basic operations and enumeration literals that are associated with the name or representation part of a type or access is given to none of those operations. The version of PIC/ADL described here does not currently support individual control over basic operations and enumeration literals because they reside at such a low, intrinsic level of the language. Nevertheless, support for such control is a topic for future study.

The operation clause can appear in four basic situations.

1. *In the name part of a provided type declaration.* The operations that can be specified are assignment and the equality/inequality tests.[4]

2. *In the representation part of a provided type declaration.* The operations that can be specified are any of the inherited operations associated with the representation part (except the basic operations and enumeration literals).

3. *In a request for a private type.* The operations that can be specified are assignment and the equality/inequality tests.

4. *In a request for a non-private type.* The operations that can be specified are any of the inherited operations associated with the representation part (except the basic operations and enumeration literals).

---

[4]Operations, such as assignment, whose syntactic notations involve operator symbols are specified using string literals. String literals are any sequence of characters embedded in double quotation marks (see Appendix A). For example, assignment is expressed in an operation clause as the string literal ":=".

Situations (1) and (2) require use of the operation clause, since the relevant operations are only implicitly declared and so are not controllable using the normal provide clause mechanism. Situations (3) and (4), on the other hand, are notational conveniences, since the specified items could as well be controlled using the normal request clause mechanism. For example, the request clause

```
request LinkedList.( List private, ":=", Insert, Remove, SetVal )
```

is equivalent to the request clause

```
request LinkedList.( List private with < ":=" >, Insert, Remove, SetVal )
```

In fact, PIC/ADL allows any operation associated with a requested type, including explicitly defined operations, to be mentioned in an operation clause in order to permit those operations to be treated uniformily from the requisition perspective. Hence, the request clause

```
request LinkedList.List private with < ":=", Insert, Remove, SetVal >
```

is equivalent to both of the previously shown request clauses. While it is conceivable to also allow such a notation for the provision perspective (i.e., to allow explicitly defined operations to be mentioned in operation clauses attached to provided types), PIC/ADL does not do so. This is to minimize the number of places in a specification submodule that a reader would have to examine when trying to understand the provision of a particular subprogram (explicitly) declared in that submodule.

A *common operation clause*, which can appear in either or both the name and representation parts of a provided type declaration, applies to all the entities to which that part of the type declaration is provided. An empty operation clause (i.e., **with** <>) is used to indicate requisition or provision of none of the relevant operations, while the complete absence of operation clauses (i.e., the default case) is used to indicate requisition or provision of all the operations.[5]

To illustrate the utility of operation clauses, consider the following refinement to the receipt-queue example of Figure 6. According to the description of ReceiptQueue given in that figure, Reorder is provided all of the inherited operations associated with Queue, since there is no applicable operation clause. To prevent Reorder from using SetVal to change the value of a receipt-queue element, the declaration of Queue can be replaced by the following.

```
type Queue
  is new LinkedList.List provide to Reorder with < Insert, Remove >
    request LinkedList.( List private, Insert,  Remove,  SetVal );
```

This declaration of Queue specifies that the only inherited operations provided to Reorder are Insert and Remove.

§3.5 *Discussion*

The control over provided types supported by PIC/ADL expands upon and generalizes the control found in Ada in two ways. First, the PIC/ADL constructs

[5]For an entity that views a type as private, an absent operation clause in the name part of a provided type declaration results in the equivalent of an Ada *private type*, while an empty operation clause in the name part of a provided type declaration results in the equivalent of an Ada *limited private type*.

allow more precise control over access to the operations associated with a provided type. Except for enumeration literals and basic operations (which can only be controlled to the extent that a basic operation is associated with either the name part or the representation part), access to each operation can be individually specified. This allows the description of multiple views of a data abstraction. Second, because access can be specified from either the requisition or provision perspectives, different views of a data abstraction can be defined by both the provider of the abstraction and the clients of the abstraction. In Ada, a view of a data abstraction can only be defined by the provider of the data abstraction.

The operation clause is similar in some respects to a construct found in Euclid and a construct proposed by Jones and Liskov [Jones & Liskov, 1978]. The PIC/ADL operation clause differs from the Euclid *with clause* in two ways. First, the Euclid construct can only be used for controlling the provision of operations. Second, the only operations that can be specified in the Euclid construct are the basic operations, enumeration literals, assignment, and the equality test; while the Euclid construct allows for more precise control over the basic operations and enumeration literals than does PIC/ADL, it cannot be used to control the provision of other kinds of operations. Jones and Liskov propose a language extension for limiting the operations that can be applied to particular objects. That construct is therefore complementary to the PIC/ADL operation clause. We intend to investigate the appropriateness of such a mechanism for inclusion in the PIC language framework.

§4. Incompleteness

PIC/ADL provides two constructs for describing incomplete systems: the *specification stub submodule* and the *incompleteness construct.*

§4.1 *Specification Stub Submodule*

```
package_specification_stub ::=
      package stub identifier is
         used_by_clause
         [ common_access_clauses [use_clause]; ]
         { provided_declarative_item }
    [ private
         { hidden_declarative_item } ]
      end package_simple_name;

subprogram_specification_stub ::=
    subprogram_stub_heading used_by_clause [access_clauses];

subprogram_stub_heading ::=
        procedure stub identifier [formal_part]
    |   function stub designator [formal_part] return type_mark
```

A *specification stub submodule* is used in place of a missing specification submodule. The syntactic differences between a specification stub submodule and a specification submodule are the reserved word **stub** and the used-by clause. The semantic difference between the two submodules is that there can be several, coexisting specification stub submodules of a module, but only one specification submodule (discounting multiple versions). However, no two specification stub submodules of a module may be used by the same module (discounting versions of a specification stub submodule).

Used-by Clause

```
used_by_clause ::= used by used_by_item {, used_by_item}
used_by_item ::= identifier | ...
```

The *used-by clause* appearing in a specification stub submodule lists the module names of the intended clients of that specification stub. There are two rules governing the module names that can be mentioned in a used-by clause.

1. No module name may appear in the used-by clauses of more than one specification stub submodule of the same module (discounting versions of a specification stub submodule).

2. The set of module names listed in a specification stub submodule's provide clauses should all appear in the set of module names listed in the submodule's used-by clause.

§4.2 *Incompleteness Construct*

The *incompleteness construct* explicitly indicates a place within a submodule where missing, additional information is to be provided later. The incompleteness construct is denoted using an ellipsis ("..."). Ellipses can appear as

- elements of declaration, parameter, and statement lists;
- items in request, provide, use, operation, and used-by clauses;

- type representations, subtype indications, or type marks;
- indices in array type definitions;
- discriminants in type declarations;
- components in record type definitions;
- ranges or (simple) expressions; and
- enumeration literals.

As mentioned above, an ellipsis immediately preceding or following the selection operator "." is shortened to "..". For example, "Pac..." denotes an item in a package Pac.

§5. Discussion

PIC/ADL, like Ada, provides little control within a module over the visibility of entities declared in that module. This lack of control, which is based on the presumption that entities are declared together because they are strongly interrelated, can be viewed as a notational shorthand for a commonly occurring situation. If more control is desired, then it can be achieved through the creation of additional modules to hold the appropriate entities. This limitation in PIC/ADL is not one that is inherent in the PIC language features. For example, it would be feasible to extend the semantics of request clauses and provide clauses to support intra-module control. Doing so in PIC/ADL, however, would not be

in keeping with the spirit of Ada; a level of control such as that might be more appropriate for a language based, for example, on Euclid.

The defaults for requisition and provision in PIC/ADL—that is, the minimum amounts of interface control information that must be explicitly supplied by a developer—are designed to mimic the control available in Ada. For instance, the absence of a provide clause on a provided entity is interpreted to mean that access to the entity is provided to "all". A request clause, like an Ada with clause, is used 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 each individual requested entity. In general, the PIC/ADL design philosophy, which we believe is in keeping with the spirit of Ada, is that the more control over module interfaces desired by a developer, the more information the developer must specify. For PIC dialects based on languages other than Ada, defaults of tighter control or requirements of more complete information may be appropriate. Thus, for example, it may be preferable in some languages to interpret an absent provide clause as meaning provide to "none", rather than provide to "all".

In terms of the formal model of visibility discussed in the next chapter, PIC/ADL permits the description of arbitrary graph structures of requisition and provision relationships, except for those involving basic operations and enumeration literals. This is true despite the fact that it provides little control within a module over the requisition and provision of entities declared in that module; in the extreme, one package can be created for each of the entities in a system and the requisition and provision of those entities controlled individually and pre-

cisely using request clauses to determine requisition arcs and provide clauses to determine provision arcs (cf., proof of Theorem 2 in Chapter IV).

CHAPTER IV

FORMAL MODEL OF VISIBILITY

A variety of mechanisms have been designed for controlling entity visibility. As with most language concepts in computer science, these interface control mechanisms have been developed in an essentially *ad hoc* fashion with no clear indication given by their designers as to how one proposed mechanism relates to another. This chapter presents a formal model for describing and evaluating interface control mechanisms. The formal model reflects our generalized view of visibility, which distinguishes the concepts of *requisition of access* and *provision of access*, thereby providing a powerful means for characterizing and reasoning about the various properties of interface control mechanisms. In particular, the notion of *preciseness* can be rigorously defined. The next section presents the basic features of the formal model. The use of the formal model for describing interface control mechanisms is then discussed. The chapter concludes with an illustration of the utility of the formal model in evaluating interface control mechanisms; theorems are presented that serve to characterize the relative strengths and weaknesses of the interface control mechanisms found in ALGOL60, Ada, Gypsy, and the approach presented in this dissertation, PIC.

§1. Basic Definitions

The formal model centers on the construction and manipulation of a representation of entity visibility relationships. This representation takes the form of a graph called the *visibility graph*.

Definition. A *visibility graph* $vg = (N, A_r, A_p)$ is a directed graph where

N is a finite set of nodes labeled by the (unique) names of entities;

A_r is a finite set of ordered pairs of nodes (n_i, n_j) denoting the requisition relationship n_i "requests access to" n_j;

A_p is a finite set of ordered pairs of nodes (n_i, n_j) denoting the provision relationship n_i "is provided to" n_j.

The ordered pairs in A_r and A_p determine the arcs in the graph. Any pair of nodes in a visibility graph may be connected by multiple arcs resulting from the requisition and provision relationships. A visibility graph may also contain loops (arcs whose origin and terminus are the same node) and cycles, both resulting from recursive requisition and provision relationships. Figure 7 depicts an example visibility graph.

A visibility graph uniquely represents a particular set of visibility relationships among a set of entities. The visibility relationships of any entity e are defined by the arcs from a node n_e to that node's nearest neighbors in the graph (i.e., adjacent nodes). The absence of an arc between two nodes indicates that no visibility relationship exists between the corresponding entities.

To consider requisition and provision separately, we refer to two spanning subgraphs of a visibility graph. One represents only the requisition relationships

$N = \{Q,R,S\}$
$A_r = \{(R,Q),(S,R)\}$
$A_p = \{(Q,Q),(Q,R),(Q,S)\}$

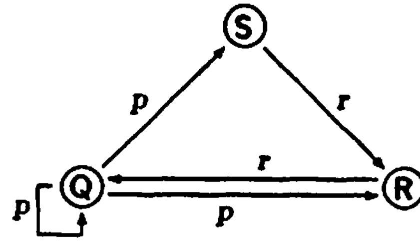


Figure 7: A Visibility Graph.

of the entities in the visibility graph, while the other represents only the provision relationships.

> *Definition.* A *requisition graph* $rg = (N, A_r, A_p)$ is a visibility graph where $A_p = \emptyset$.

> *Definition.* A *provision graph* $pg = (N, A_r, A_p)$ is a visibility graph where $A_r = \emptyset$.

Two functions on visibility graphs are defined that produce these subgraphs.

> *Definition.* The *requisition extraction function* $re : VG \rightarrow VG$, is defined by mapping the visibility graph $vg = (N, A_r, A_p)$ to the requisition graph $rg = (N, A_r, \emptyset)$.

> *Definition.* The *provision extraction function* $pe : VG \rightarrow VG$, is defined by mapping the visibility graph $vg = (N, A_r, A_p)$ to the provision graph $pg = (N, \emptyset, A_p)$.

Using these functions, two useful relationships between visibility graphs can be defined.

> *Definition.* A visibility graph vg_i *request-satisfies* a visibility graph vg_j iff $re(vg_j) \subseteq re(vg_i)$.

Definition. A visibility graph vg_i *provide-satisfies* a visibility graph vg_j iff $pe(vg_j) \subseteq pe(vg_i)$.

where we say $vg_1 \subseteq vg_2$ if $N_1 \subseteq N_2$, $A_{r,1} \subseteq A_{r,2}$, and $A_{p,1} \subseteq A_{p,2}$. Informally stated, the desire for a set of entities s_j to be requested by (provided to) some entity e is satisfied by e requesting (being provided) any set of entities s_i of which s_j is a subset.

A visibility graph can be derived from some *representation* of a program by applying the rules of a particular interface control mechanism, or combination of mechanisms, to the entities in the representation. More formally, we denote the collection of program representations by PR and define a function that performs this mapping as follows:

Definition. A *visibility function* $vf : PR \rightarrow VG$, is a function that maps a program representation pr to a visibility graph vg.

A set of visibility functions $VF = \{vf_a, vf_b, \ldots\}$ can be defined where vf_m is the visibility function implementing the interface control mechanism m.

The formal model uses the visibility graph to record requisition and provision relationships without insisting on a particular interpretation of the consistency/ inconsistency of those relationships. For instance, an Ada interpretation of the graph in Figure 7 would view node Q as representing a *library entity*—an entity, such as a sine function, provided to all other entities—even though not all those other entities request it. Thus, the Q–R and Q–S relationships, for example, would be interpreted as consistent. The R–S relationship, on the other hand, would be interpreted as inconsistent. For Ada, a minimum condition for the

consistency of a set of entity visibility relationships is that the entities that each entity requests are in fact provided. In terms of visibility graphs, this corresponds to the following property:

> *Definition.* A visibility graph $vg = (N, A_r, A_p)$ is *well-formed for Ada* iff $\forall (n_i, n_j) \in A_r, (n_j, n_i) \in A_p$.

Of course, other consistency/inconsistency interpretations of the graph in Figure 7 are also reasonable. For example, node Q might represent some sort of "authorisation" module; the fact that S does not request access to Q might then indicate a problem in the system. In general, the appropriateness of an interpretation can depend upon the language, the application domain, the development method, or even the managerial discipline in force.

§2. Describing Interface Control Mechanisms

An interface control mechanism m is described by its corresponding visibility function vf_m. Each such function has two components that *explicitly* address the requisition and provision aspects of entity visibility. The *requisition function* rf_m describes requisition by mapping a program representation to a requisition graph while the *provision function* pf_m describes provision by mapping a program representation to a provision graph. Thus,

$$vf_m(pr) = rf_m(pr) \cup pf_m(pr)$$

where pr is some program representation and the union of two visibility graphs vg_1 and vg_2 is the visibility graph $\{N_1 \cup N_2, A_{r,1} \cup A_{r,2}, A_{p,1} \cup A_{p,2}\}$. Component

functions rf and pf can be further broken down into functions operating on individual *kinds* of entities, such as subprograms and objects, as follows:

$$\begin{aligned} rf_m(pr) &= rf_{m,ek_1}(pr) \cup \cdots \cup rf_{m,ek_n}(pr) \\ pf_m(pr) &= pf_{m,ek_1}(pr) \cup \cdots \cup pf_{m,ek_n}(pr) \end{aligned}$$

where ek_i denotes the entity kind upon which the requisition or provision function operates. Hence, for each entity kind that is of interest, there is a function that describes requisition and a function that describes provision. Requisition functions are similar in nature to the "binding" functions of [Hennessy & Kieburtz, 1981]. Provision functions, however, appear to have no counterpart in previous formalisms.

To illustrate the use of the descriptive method, descriptions of the requisition and provision of subprograms as they are controlled by the nesting mechanism of ALGOL60 are given here. This entails the definition of the requisition function $rf_{ALGOL60,subprograms}$ and the provision function $pf_{ALGOL60,subprograms}$. The discussion here is simplified by only considering the visibility of subprograms to subprograms.

Requisition and provision functions, as mentioned above, operate on a representation of a program. One suitable representation for ALGOL60 is a graph we call the *nesting graph*.

Definition. A *nesting graph* $ng = \{N_{ng}, A_{ng}\}$ is a labeled digraph where

N_{ng} is a finite set of nodes labeled by the (unique) names of entities;

A_{ng} is a finite set of ordered pairs of nodes (n_i, n_j) denoting the relationship n_i "parent of" n_j, which means n_j is directly nested in n_i.

Notice that for this example N_{ng} consists only of nodes whose entities are subprograms. Figure 8 shows the skeleton of a nested ALGOL60 program and its representation as a nesting graph.

In the subsequent definition of the requisition and provision functions, we make use of three auxiliary functions, each of which maps N_{ng} to the powerset of N_{ng} as follows:

$$(1)\quad Parent(n_i) = \{\, n_j \in N_{ng} \mid (n_j, n_i) \in A_{ng} \,\}$$

$$(2)\quad Siblings(n_i) = \left\{ n_j \in N_{ng} \,\middle|\, \begin{array}{l} \exists n_k \in N_{ng} \text{ such that } (n_k, n_i) \in A_{ng} \\ \text{and } (n_k, n_j) \in A_{ng} \text{ and } i \neq j \end{array} \right\}$$

$$(3)\quad Ancestors(n_i) = \left\{ n_j \in N_{ng} \,\middle|\, \begin{array}{l} n_j = Parent(n_i) \\ \text{or } n_j \in Siblings(Parent(n_i)) \\ \text{or } n_j \in Ancestors(Ancestors(n_i)) \end{array} \right\}$$

For any $n_i \in N_{ng}$, $Parent(n_i)$ will always be a singleton since an entity can be directly nested in only one other entity.

The requisition function is now defined to transform ng, a graph representation of the nested structure of subprograms, into a requisition graph, which explicitly describes the effect of ALGOL60's nesting mechanism on subprograms' requisition.

Definition. $rf_{ALGOL60,subprograms}(ng) = \{N, A_r, A_p\}$ where

$$N = N_{ng}$$

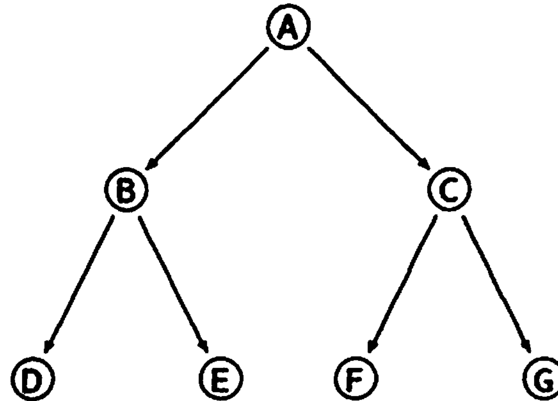
$$A_r = \left\{ (n_i, n_j) \,\middle|\, \begin{array}{l} i = j \text{ or } n_i = Parent(n_j) \text{ or } n_j \in Siblings(n_i) \\ \text{or } n_j \in Ancestors(n_i) \end{array} \right\}$$

$$A_p = \emptyset$$

From this description it can be easily seen that (1) a subprogram is requested by itself, (2) a subprogram is requested by the subprogram in which it is directly

```
begin
   procedure A;
      procedure B;
         procedure D;
         begin ... end D;
         procedure E;
         begin ... end E
      begin ... end B;
      procedure C;
         procedure F;
         begin ... end F;
         procedure G;
         begin ... end G
      begin ... end C
   begin ... end A
end
```

(a)



(b)

Figure 8: A Nested Program (a) and its Representation as a Nesting Graph (b).

nested, (3) a subprogram is requested by those subprograms with the same parent, and (4) a subprogram is requested by those subprograms nested within it as well as requested by those subprograms nested within its siblings.

For ALGOL60, the provision function is quite similar to the requisition function.

Definition. $pf_{ALGOL60,subprograms}(ng) = \{N, A_r, A_p\}$ where

$$
\begin{aligned}
N &= N_{ng} \\
A_r &= \emptyset \\
A_p &= \left\{ (n_i, n_j) \middle| \begin{array}{l} i = j \text{ or } n_j = Parent(n_i) \text{ or } n_i \in Siblings(n_j) \\ \text{or } n_i \in Ancestors(n_j) \end{array} \right\}
\end{aligned}
$$

These descriptions reveal the fact that in ALGOL60 requisition and provision are essentially mirror-image counterparts. In particular, the expression defining the set of tuples (n_i, n_j) in A_p of the provision function is the same as the expression defining the set of tuples (n_i, n_j) in A_r of the requisition function, except that the i's and j's are reversed. This similarity, however, is certainly not true of all mechanisms.

We contend that requisition and provision functions of the formal model presented here are easier to comprehend than the manipulation of a dynamic environment component found in other formal models. As mentioned in Chapter I, for instance, those other models make it difficult to recognise that with nesting, a subprogram's so-called "local" entities are unavoidably made visible to other subprograms nested within that subprogram. This problem is clearly exposed using the formal model presented here; by simply looking at the requisition and provision functions for subprograms it is immediately evident that a subprogram's

supposedly "local" child subprogram is in fact visible to any other subprograms nested within that subprogram.

§3. Evaluating Interface Control Mechanisms

Interface control mechanisms can be characterized in a number of ways and these characterizations can then provide a basis for evaluating the strengths and weaknesses of different mechanisms. This section presents one such characterization which is possible within the framework of our formal model. Specifically, the notion of *preciseness* is defined for an interface control mechanism in terms of the mechanism's accuracy in capturing desired requisition and provision relationships.

It can easily be argued that a language's interface control mechanism(s) m should be such that $\forall vg \in VG$, $\exists pr \in PR$ such that $vf_m(pr)$ request-satisfies and provide-satisfies vg. In other words, it should be possible to find a program representation that realizes any requisition and provision relationships that a developer might wish to devise, although additional requisition and provision may be allowed as well. It is not surprising that the interface control mechanisms of all modern languages that we have examined satisfy this minimum requirement.[1] Stronger properties for evaluating mechanisms are needed, however, which leads to the following definitions.

[1] Many pre-ALGOL60 languages do not satisfy this requirement since they do not support recursion. For example, the FORTRAN standard [ANSI, 1978] excludes recursion from the language, and therefore no pr can be found such that $vf_{FORTRAN}(pr)$ request-satisfies or provide-satisfies a vg containing a loop in either its A_r or A_p.

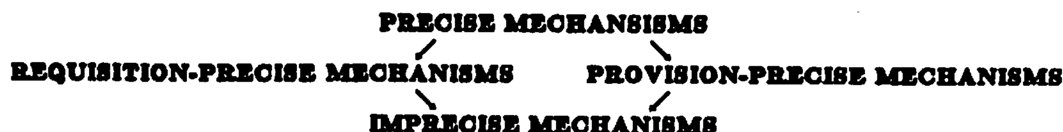
Definition. An interface control mechanism m is *requisition-precise* iff $\forall vg \in VG$, $\exists pr \in PR$ such that $re(vf_m(pr)) = re(vg)$.

Definition. An interface control mechanism m is *provision-precise* iff $\forall vg \in VG$, $\exists pr \in PR$ such that $pe(vf_m(pr)) = pe(vg)$.

Definition. An interface control mechanism m is *precise* iff it is both requisition-precise and provision-precise.

Definition. An interface control mechanism m is *imprecise* iff it is neither requisition-precise nor provision-precise.

Intuitively, the definitions state that if for each possible visibility graph, a program representation can be found with the property that the visibility relationships among its entities are exactly those specified in the visibility graph, then the mechanism is indeed precise. A mechanism is less than precise if the requisition relationships or provision relationships cannot be exactly realized. This suggests the following hierarchy of interface control mechanisms based on preciseness:



If we disregard self-recursive visibility, which in most languages cannot be fully controlled, then entries in this preciseness-characterisation hierarchy are exemplified by the mechanisms found in ALGOL60, Ada, Gypsy, and our approach to interface control, PIC. The following theorems position these mechanisms in the hierarchy.

THEOREM 1. ALGOL60 is imprecise.

PROOF. For a mechanism to be imprecise, a visibility graph must exist for which a program representation cannot be found that results in exactly the desired requisition and, similarly, a visibility graph must exist for which a program representation cannot be found that results in exactly the desired provision. One such graph, which reflects a very common situation in programming, conveniently exhibits both these properties. This graph, depicted in Figure 9, represents two subprograms A and B, each not callable by the other, sharing exclusive use of a third, utility subprogram C. From the definition of $rf_{ALGOL60,subprograms}$ and $pf_{ALGOL60,subprograms}$ it can be seen that for the two subprograms to be hidden from each other, and so not callable by each other, one cannot be nested (directly or indirectly) in the other nor can they be siblings. The utility subprogram must then be an ancestor (other than a parent) so that it is callable by both subprograms. This being the case, the utility subprogram must unavoidably be requested by, and provided to, other ancestors of the subprograms, thus violating the desired visibility relationships. □

THEOREM 2. Ada is requisition-precise but not provision-precise.

PROOF. Ada supports a nesting mechanism similar to ALGOL60's, but in addition offers alternatives that can be used to avoid many of nesting's shortcomings [Clarke, Wileden, & Wolf, 1980]. These alternatives are the *private/visible* mechanism of Ada's encapsulation construct, the package, and the *with clause.* The first can be used in combination with nesting to achieve a greater degree

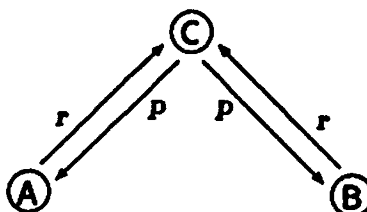


Figure 9: Visibility Graph of a Common Programming Situation.

of provision control than is possible in ALGOL60. In particular, it can be used to selectively hide nested entities that would otherwise be undesirably provided. That selection, however, is on an all-or-nothing basis; either an entity is provided to all entities in the scope of the package or it is provided to no entity. Therefore, Ada's version of nesting is still not provision-precise. This shortcoming with respect to provision extends to nest-free packages, where the "scope" of a package is then the entire program. Entities provided by a package (in Ada's terminology, the *visible* packaged entities) are unavoidably provided to all other entities in the program and hence their corresponding nodes in provision graphs have arcs to every other node. Thus, Ada is shown not to be provision-precise. To show Ada is requisition-precise, first observe that Ada programs can be constructed exclusively from nest-free packages. Each such package employs the second alternative mentioned above, the with clause, to specify the entities requested by its contents. The with clause allows the realization of any arbitrary set of requisition relationships since, in the extreme, one package can be created for each of the entities in the program.[2] In terms of visibility graphs and program rep-

[2]Of course, purely local entities need not be packaged, but can be left, for instance, in the subprograms in which they are used. Recursive subprograms referencing shared entities intro-

resentations, this means that if only with clauses are used to induce requisition arcs, then for any given visibility graph a program representation can be found that results in exactly the desired requisition graph. Thus, Ada is shown to be requisition-precise. □

THEOREM 3. Gypsy is provision-precise but not requisition-precise.

PROOF. Gypsy does not support any degree of nesting. To control provision, Gypsy employs a construct called an *access list* which specifies for an entity just those other entities to which it is provided. In terms of visibility graphs and program representations, this feature solely determines provision arcs. Hence, for any given visibility graph a program representation can be found that results in exactly the desired provision graph with the consequence that Gypsy is provision-precise. Gypsy does not, however, have Ada's concept of the with clause. Indeed, there is no way to control requisition in Gypsy; all provided entities are unavoidably requested. Therefore, aside from visibility graphs having pairs of nodes connected by both a provision arc and a requisition arc, desired requisition relationships cannot be realised. Thus, Gypsy is not requisition-precise. □

THEOREM 4. The PIC language framework is precise.

PROOF. The PIC language framework contains the *request clause* for specifying requisition and the *provide clause* for specifying provision. By definition,

---

duce some minor complications, but this can be handled by appropriate use of parameters and packages (see Chapter V). Finally, types that are mutually dependent cannot be separately packaged. It can be argued, however, that while this special case involves two or more syntactically separate type declarations, only one *genuine* type is being defined; it makes no sense to request (or provide) access to one component of the definition without requesting (or providing) access to the others.

these clauses can be used to completely determine the requisition and provision of individual entities and, therefore, can be used to realise the requisition and provision of any desired visibility graph. The PIC language framework is thus shown to be a precise interface control mechanism. □

Notice that these theorems demonstrate that the design of the PIC language framework results in a precise interface control mechanism and that the mechanism is *more* precise than several other such mechanisms. Indeed, no other interface control mechanism we are aware of resides at the same position in the preciseness-characterisation hierarchy occupied by the PIC language framework.

§4. Discussion

There are two limitations to the formal model that should be pointed out. First, the formal model presented here is concerned only with *static* interface control mechanisms, that is, those mechanisms that determine bindings before execution. Second, this formal model only deals with *direct* visibility; the fact that an entity can use another entity through a third entity is not considered (for instance, if e_i and e_j are subprograms, e_k is an object, and e_i can invoke e_j to operate on e_k, this does not necessarily imply that e_k is visible to e_i.) Relaxing these restrictions is a topic for future study.

We recognise that there are other considerations that affect how an interface control mechanism is used. For instance, the package in Ada, besides being used in the control of entity visibility, is used as a primary modularisation tool; there are

practical situations in which modularity and interface control constraints conflict. For example, the proof of Theorem 2 suggests that precision of requisition in Ada can be achieved by placing entities in separate packages. Such a separation may interfere with the colocation of entities that, while perhaps not requested in the same way, are otherwise logically related. In light of this, the formal model should perhaps be extended to explicitly address the sources of such conflicts.

In summary, we have defined a formal model that can be used both to describe interface control mechanisms and to reason about those mechanisms in order to provide characterizations of their strengths and weaknesses. In this chapter, we have shown how the formal model can be used to characterise the preciseness of interface control mechanisms. In so doing, we have pointed up the very different approaches to controlling entity visibility employed in four such mechanisms. Based on examples such as these, we believe this formal model can be a valuable aid to developers and language designers as they try to decide on the suitability of a mechanism.

CHAPTER V

NESTING: AN ANACHRONISM

"The choice of what is to be omitted from a new language is in practice much more critical than the choice of what is to be included. The decision to omit a feature requires not only familiarity with this feature (and knowledge of how to live without it) but the courage to face the inevitable criticism of its absence in the new language in spite of its presence in another existing language."

—attributed to N. Wirth by Ledgard and Singer in [Ledgard & Singer, 1982]

Nesting has been a controversial language feature ever since its introduction in ALGOL60. The programming language and software engineering literature is riddled with remarks alluding to the supposed advantages or disadvantages of a nested program structure. There is certainly no consensus on this issue, as is evident in the designs of languages that have succeeded ALGOL60; whereas languages such as C, CLU, and Gypsy have eschewed nesting, the designs of languages such as Pascal, Euclid, and most recently Ada have included nesting as an integral part. Indeed, some language designers are apparently willing to

include nesting *despite* its recognized shortcomings. To quote Brinch Hansen [Brinch Hansen, 1981, page 366]:

> *"I find nested procedures as hard to read as prose that makes frequent use of paranthetic* (sic) *remarks...*[yet] *general nesting was allowed merely as a means of simplifying the language description."*

This chapter provides a thorough discussion of the issues associated with nesting, seeking once and for all to show that nesting is an anachronism when compared to mechanisms that rely on encapsulation and explicit import/export control. The discussion complements the formal treatment of nesting given in the previous chapter by offering a pragmatic demonstration of nesting's inadequacies and by examining its role in other aspects of a language's design and a software system's development. The chapter concludes with a discussion of several attempted "fixes" to the basic nesting mechanism. Although our fundamental conclusion essentially corroborates our earlier view [Clarke, Wileden, & Wolf, 1980], as well as the views of Wulf, Shaw, Hanson [Wulf & Shaw, 1973; Hanson, 1981] and others, we believe that the formal foundation, pragmatic justification, and comprehensive scope of our arguments makes this the most thorough and convincing critique of nesting yet to appear.

§1. Practical Evaluation of Nesting's Control of Entity Visibility

From a theoretical standpoint, the arguments in Chapter IV constitute an interesting, formal demonstration of the inadequacy of nesting as an interface

control mechanism. In the utilitarian world of programming, however, there is a need to address pragmatic concerns. In particular, a developer might like to know if the counterexample given in the proof of Theorem 1 in Chapter IV is pathological, and nesting in fact acceptable for other programming problems, or whether there are further practical situations in which nesting fails. Additionally, a developer might like to know if there are alternative, nest-free language mechanisms that circumvent nesting's shortcomings in those situations.

This section discusses several basic programming scenarios in an effort to demonstrate, from a practical standpoint, the inadequacies of nesting. Unlike the previous chapter, no theorems are given to support the arguments. Rather, the arguments rely upon the reader's intuition and experience as a basis for recognizing the comprehensiveness of the scenarios and the viability of the solutions. Nesting in ALGOL60 is once again used as the prototypical nesting mechanism. The encapsulation and import/export concepts of PIC/ADL are used to exemplify nest-free alternatives, although less sophisticated approaches, such as the nest-free style for Ada described in [Clarke, Wileden, & Wolf, 1980], would suffice. The existence of these alternatives, which offer solutions that are usually better but certainly no worse than nesting, shows that nesting can indeed be avoided as an interface control mechanism. Note that while the specific examples illustrating the scenarios and their solutions discussed in this section exclusively involve objects and subprograms, the reasoning is nonetheless applicable to other kinds of entities.

§1.1 *Common Programming Scenarios*

In essence, the two principal goals of interface control are the hiding of an entity from another entity, and the sharing of an entity among entities. Clearly, these two goals are complementary; when an entity is to be shared among some subset of entities in a program, then that entity should at the same time be hidden from the entities not in the subset. This view of interface control covers the entire spectrum of possible entity visibility relationships, ranging from *local visibility*, in which an entity is visible only to a single other entity and hidden from all the rest, to *global visibility*, in which an entity is visible to (and thus shared among) all the entities and hidden from none. At intermediate points within this spectrum an entity may be shared among some entities while at same time hidden from others; this is referred to here as *selective visibility*.

Local Visibility

The concept of "locality", which is the explicit expression of an object's, type's, or subprogram's confinement to a particular subprogram, is one of the supposed advantages of nesting [Wirth, 1983]. Nesting does in fact protect an entity from access by certain other subprograms, in particular, those subprograms declared at any point not within the subtree rooted at that entity's parent. The subprograms declared within the subtree, however, do have access to the entity and hence may violate the entity's intended local visibility. Consider the example in Figure 10, where an object X and a procedure C are declared within a procedure A. Despite the fact that only procedure A should manipulate X and invoke C,

```
procedure A;
    ... X;                declaration of supposedly local object X in A

    procedure C;          declaration of supposedly local subprogram C in A
        ...
    begin
        ...
    end C;
    ...
    procedure B;
        ...
    begin
        X := ...;         mistaken, but legal, use of X in B
        ...
        C;                mistaken, but legal, use of C in B
        ...
    end B;
    ...
begin
    X := ...;             use of X in A
    ...
    C;                    use of C in A
    ...
end A
```

Figure 10: Flawed Local Visibility Using Nesting.

other subprograms declared within A, such as procedure B, have access to X and C; procedure B may, for example, mistakenly read or even update X.

The expression of an object's or type's local visibility is immediate in the absence of nesting, since an object or type declared within a subprogram is not visible to the other, necessarily external subprograms. This is shown for object X in Figure 11. The expression of a subprogram's local visibility can be made using a provide clause. In Figure 11, the fact that procedure C has a provide clause mentioning only procedure A means that only A can legitimately invoke C; any other invoker of C, such as procedure B, is in error.

Global Visibility

In ALGOL60, and similarly in most nested languages, a program is a single, monolithic subprogram or block. The global visibility of an object, type, or subprogram is achieved by declaring that entity directly in this outermost subprogram or block. The global visibility of an entity can be interfered with, however, by the (perhaps inadvertent) introduction of another entity having the same name. In Figure 12, for example, there is a declaration of an object X in the outermost procedure A as well as in an intermediate procedure B. The result is to hide the supposedly global declaration from a portion of the program, effectively rendering that entity non-global.

Using the nest-free alternative supported by PIC/ADL, an object or type can be globally provided by placing its declaration in the visible part of a package. A subprogram can be globally provided by declaring it in a package's visible

```
procedure C                   declaration of subprogram C local to A
   provide to A;

procedure C is
   ...

begin
   ...

end C;

procedure A
   request C;

procedure A is
   X : ...;                   declaration of local object X in A
   ...

begin
   X := ...;                  use of X in A
   ...
   C;                         use of C in A
   ...

end A;

...

procedure B is
   ...

begin
   X := ...;                  DETECTABLE ERROR: mistaken, illegal use of X in B
   ...
   C;                         DETECTABLE ERROR: mistaken, illegal use of C in B
   ...

end B;
```

Figure 11: Local Visibility Using Nest-free PIC/ADL.

```
procedure A;
   ... X;               declaration of supposedly global object X in A
   ...

   procedure B;
      ... X;            (re)declaration of object X; "global" declaration
                        of X in A not visible in subtree rooted at B
      ...

      procedure C;
         ...

      begin
         ...            C cannot refer to X declared in A

      end C;

      ...

   begin
      X := ...;         use of X declared in B
      ...

   end B;

   ...

begin
   X := ...;            use of X declared in A
   ...

end A
```

Figure 12: Flawed Global Visibility Using Nesting.

part or by declaring it as a non-packaged subprogram. The global visibility of such declarations cannot, by definition, be interfered with by declarations having the same name appearing in other parts of a program. This is illustrated in Figure 13, where a nest-free solution to the problem of Figure 12 is shown. It should be pointed out that Ada offers a special mechanism, which is a refinement to the basic nesting mechanism, that also solves this problem. In Ada, an entity can be denoted by the unambiguous path name in the nesting graph, rendering an otherwise hidden entity visible. For instance, in the example of Figure 12, object X can be referred to from procedure C using the notation "A.X".

Selective Visibility

Several common situations arise in the course of programming when it is convenient or even necessary to share objects, types, or subprograms among some subset of a program's modules. This is the basis, for example, of the notion of *data abstraction*; only the subprograms realizing the operations of an abstraction should be able to directly manipulate, and thus share access to, the entities involved in the implementation of that abstraction. The selective visibility that can be achieved using nesting is limited to a form in which an entity is shared among all the subprograms in the nesting-structure subtree rooted at that entity's parent.[1] This makes it practically impossible, for instance, to use nesting

[1] Of course, if the identifier associated with that entity is redeclared within the subtree, then the entity is hidden from that point in the tree on downward (cf., Figure 12). Such redeclarations could conceivably be used in this manner to achieve a certain degree of increased interface control, but at best this seems to be an awkward method.

```
package Pac is
   X : ...;            declaration of global object X in Pac

end Pac;

procedure A
   request Pac;
procedure A is
   ...

begin
   Pac.X := ...;       A can refer to X declared in Pac
   ...

end A;

procedure B
   request Pac;
procedure B is
   X : ...;            (re)declaration of object X in B
   ...

begin
   Pac.X := ...;       B can refer to X declared in Pac
   ...
   X := ...;           B can refer to X declared in B
   ...

end B;

procedure C
   request Pac;
procedure C is
   ...

begin
   Pac.X := ...;       C can refer to X declared in Pac
   ...

end C;
```

Figure 13: Global Visibility Using Nest-free PIC/ADL.

to enforce the visibility relationships of data abstraction. As an example [Wulf & Shaw, 1973], consider the attempt at programming a stack abstraction shown in Figure 14. The objects Stack and StackPointer, which are shared by the subprograms (i.e., abstract operations) Initialize, Push, Pop, and Empty, are unavoidably made accessible to the users of those subprograms, such as StackUser. The counterexample given in the proof of Theorem 1 in Chapter IV is another illustration of this phenomenon. In general, an entity that is shared must be located at a level in the nesting structure at least as high as the level of its highest-level user, making it available to all subprograms at the same or lower levels.

The concepts of encapsulation and import/export are particularly well suited for describing selective visibility. They can be used to achieve any desired selective-visibility relationship by placing the shared entities into a package and, via the request and provide clauses, making the entities in that package visible only to those subprograms sharing the entities. Figure 15 gives one possible nest-free solution to the data abstraction problem of Figure 14.

```
begin

    integer array Stack [ 1 : 100 ];
    integer StackPointer;

    procedure Initialize;
    begin
        StackPointer := 0

    end Initialize;

    procedure Push ( Item ); integer Item;
    begin
        StackPointer := StackPointer + 1;
        Stack[ StackPointer ] := Item

    end Push;

    integer procedure Pop;
    begin
        Pop := Stack[ StackPointer ];
        StackPointer := StackPointer - 1

    end Pop;

    boolean procedure Empty;
    begin
        if StackPointer = 0 then Empty := true
        else Empty := false
    end Empty;

    procedure StackUser;
    begin
        ...     StackUser has undesirable access to Stack and StackPointer

    end StackUser

end
```

Figure 14: Flawed Selective Visibility Using Nesting.

```
package StackData is
      provide to StackOps;

   Stack : array ( 1 .. 100 ) of Integer;
   StackPointer : Integer;

end StackData;

package StackOps is
      request StackData;

   procedure Initialize;
   procedure Push ( Item : in Integer );
   function Pop return Integer;
   function Empty return Boolean;

end StackOps;

package body StackOps is
   procedure Initialize is
   begin ... end Initialize;

   procedure Push ( Item : in Integer ) is
   begin ... end Push;

   function Pop return Integer is
   begin ... end Pop;

   function Empty return Boolean is
   begin ... end Empty;

end StackOps;

procedure StackUser
   request StackOps;

procedure StackUser is
begin
   ...     StackUser does not have access to Stack and StackPointer

end StackUser;
```

Figure 15: Selective Visibility Using Nest-free PIC/ADL.

§1.2 *A Second(ary) Alternative*

It is interesting to consider a somewhat different nest-free approach to sharing, one involving the well-understood technique of passing an object or subprogram as a *parameter* during subprogram invocation.[2] In contrast to nesting, where sharing is determined by the nesting structure of a program, and explicit interconnection, where sharing is determined by import/export relationships, the parameter method depends upon the *calling structure* to determine sharing. Given two procedures A and B sharing an object or subprogram X, there are three situations to consider. If A calls B, then X can be made available to A and shared with B as part of A's invocation of B (Figure 16). On the other hand, if A does not call B directly but does call a procedure C that calls B—that is, A indirectly calls B—then X can be made available to A and passed to B through C (Figure 17). Finally, if A and B are independent—that is, they do not exhibit either a direct or an indirect caller/callee relationship—then a subprogram D that (directly or indirectly) calls both A and B can always be found,[3] X made available to D, and X passed to A and B (Figure 18).

The parameter method works well in the first situation, where an object or subprogram is shared between a caller and direct callee; the caller shares, as part of the invocation of the callee, specific objects at specific points in its execution.

---

[2] Some languages, such as Ada, allow for the creation of *generic* modules that are type parameterized and so can achieve sharing of types, in addition to objects and subprograms, through a parameter-passing-like mechanism. This form of "parameter passing" is not considered here.

[3] Two nodes in a calling structure always have at least one common ancestor. At worst, this ancestor is the root node of the call graph.

```
procedure A;
    ... X;
    ...
begin
    ...
    X := ...;             use of X
    ...
    B ( X );
    ...
end A;

procedure B ( XB ); ... XB;
    ...
begin
    ...
    XB := ...;            use of X through XB
    ...
end B;
```

Figure 16: Using the Parameter Method to Share an Object: Direct Caller/Callee Relationship.

```
procedure A;
    ... X;
    ...

begin
    ...
    X := ...;          use of X
    ...
    C ( X );
    ...

end A;

procedure C ( XC ); ... XC;
    ...

begin
    ...
    B ( XC );
    ...

end C;

procedure B ( XB ); ... XB;
    ...

begin
    ...
    XB := ...;         use of X through XB
    ...

end B;
```

Figure 17: Using the Parameter Method to Share an Object: Indirect Caller/Callee Relationship.

```
procedure D;
   X : ...;
   ...
begin
   ...
   A ( X );
   ...
   B ( X );
   ...
end D;

procedure A; ( XA ) ... XA;
   ...
begin
   ...
   XA := ...;          use of X through XA
   ...
end A;

procedure B ( XB ); ... XB;
   ...
begin
   ...
   XB := ...;          use of X through XB
   ...
end B;
```

Figure 18: Using the Parameter Method to Share an Object: Independent Relationship.

This precision is not surprising since this is the particular situation for which the mechanism is designed! There is certainly no reason to believe, however, that the calling structure serves any better than the nesting structure as a general framework upon which to base entity visibility. Indeed, the parameter method suffers in the other two situations from a shortcoming similar to that found for nesting; a shared object or subprogram is undesirably opened to extraneous access by any intermediately called subprograms. In those cases, the method involving encapsulation and explicit import/export control is the preferred choice. Moreover, proponents of nesting argue that the use of parameters may lead to long and repetitious parameter lists, a situation they point out can be avoided with nesting. There are, however, nest-free ways to reduce the number of parameters. One is to group parameters into aggregate data structures, such as records, that can be passed as single parameters. An alternative is to collect the erstwhile parameters into a package and then make them available using request and provide clauses. This alternative is akin to nesting because it places the parameters in a commonly accessible location but it does not suffer from nesting's weaknesses in controlling that accessiblity.

Given encapsulation and explicit import/export control, the parameter method can be considered a redundant, sometimes imprecise means for avoiding nesting in many common programming situations. It is, nonetheless, absolutely necessary in situations involving *recursion* because the parameter method, like nesting but unlike encapsulation and explicit import/export control, can be used to describe certain dynamic visibility relationships.

The goal of recursion is to create new instantiations of subprogram-local objects while at the same time retaining the old instantiations. When using recursion, the placement within a program of objects that need recursive instantiations is critical. Specifically, if a new instantiation of an object is desired as part of a recursive call to a subprogram, the object must be local to that subprogram. This concern has significant ramifications for the solutions to the scenarios discussed above. In particular, if there is a recursive control dependency among the subprograms sharing an object, and therefore there exists either a direct or an indirect caller/callee relationship, then encapsulation and explicit import/export control cannot be used since separate packaging of an object makes that object non-local. Fortunately, the parameter method provides a perfectly adequate, nest-free solution. As an example, consider Figure 19 which contains a selective-visibility problem suggested by P. Abrahams. In this figure, procedures A and B are mutually recursive and share an object X. Procedure B requires a new instantiation of object X each time it recursively calls procedure A; this last fact dictates that object X must be local to procedure A. Figure 19a presents a nested solution to the problem. Figure 19b presents an equivalent, nest-free solution based on the parameter method. In general, the parameter method can be used to recover the dynamic qualities of nesting that may be lost if encapsulation and explicit import/export control are only used.

In sum, nesting fails to provide precise control over entity visibility in a number of very basic programming scenarios. The simple nest-free methods described here, when appropriately applied, result in solutions that are generally superior

```
procedure A;
    ... X;

    procedure B;
        ...

    begin
        ...
        X := ...;
        ...
        A;   recursive invocation of A
        ...

    end B;

begin
    ...
    X := ...;
    ...
    B;
    ...

end A;
```

(a)

```
procedure A;
    ... X;

    begin
    ...
    X := ...
    ...
    B ( X );
    ...

end A;

procedure B ( XB ); ... XB;
    ...

begin
    ...
    XB := ...; use of X through XB
    ...
    A;          recursive invocation of A
    ...

end B;
```

(b)

Figure 19: Mutually Recursive Subprograms Sharing an Object Using Nesting (a) and the Parameter Method (b).

to those possible with nesting. Further, the existence of these methods obviates nesting's inclusion into a language as the interface control mechanism.

§2. Further Arguments Against Nesting

Although the interface control aspects of nesting have the most profound impact on the nature of nested languages, there are several other areas in which nesting has an effect. In particular, the rigid structure imposed by nesting bears upon the software engineering concerns of readability, modifiability, and incremental analysis. Moreover, it is often suggested that nesting is needed to perform dynamic memory management. Finally, relevant to the consideration of any language feature are issues concerning the manner in which the feature is implemented and the effect the presence of the feature has on other language features. This section explores each of these areas, pointing out a number of important ways in which nesting interferes with readability, modifiability, and incremental analysis, demonstrating that dynamic memory management can in fact be realized independently of nesting, and showing that an implementation of a language that does not support nesting should be no worse (and perhaps better), in terms of the implementation's complexity and run-time efficiency, than an implementation of a language that supports nesting.

§2.1 *Software Engineering Considerations*

Readability

Advocates of nesting argue that a program's nesting structure can be used to reflect the logical structure of that program and, hence, lead to improved readability. This is particularly true, they believe, for logical structures that are hierarchical, since a program's nesting structure, a tree, is itself hierarchical. There are, however, three problems with these notions. First, many logical structures are not hierarchical simply due to the presence of recursion. Second, many logical structures that are hierarchical do not take the form of trees. Third, many logical structures that are tree hierarchies cannot be captured using the nesting structure.

Following Parnas [Parnas, 1974], the term *structure* is used here to refer to a partial description of a program showing it as a collection of parts and showing some relation among the parts. A structure is termed *hierarchical* if the relation defines a partial ordering between pairs of parts. One logical structure commonly found in programs is the *calling structure*, mentioned in the previous section, where the parts are *subprograms* and the relation is *invokes*. This structure forms the basis for a number of important software design methods including "abstract machines" [Shaw, 1974].

Figure 20 shows an example of a tree calling structure in the form of a *call graph*, a graphical representation of the subprogram invocations found within a program. Since this call graph is a tree, it should also serve well as the *nesting*

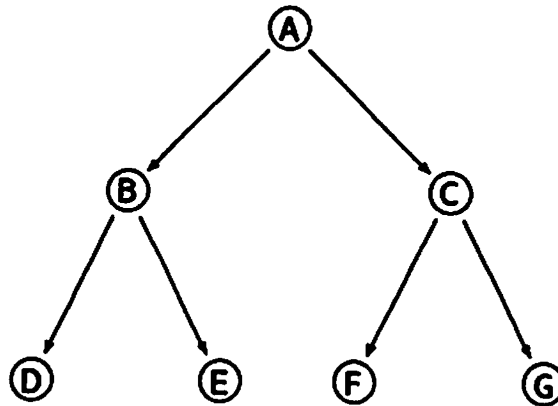


Figure 20: A Tree-structured Call Graph, Which Also Serves as a Nesting Graph.

graph, depicting the nesting structure of a program (see Chapter IV). Figure 21 shows the skeleton of a textual representation of the structure given by this graph. Although the nesting and logical structures of the program are identical, the nesting structure allows for the possibility of numerous other calling structures besides the desired one. This is a problem directly attributable to nesting's inadequate interface controls that, in this case, has a detrimental effect on readability. A subgraph of a program's visibility graph (see Chapter IV), the *potential-call graph*, shows all possible subprogram invocations permitted by a particular nesting graph. The potential-call graph for the nesting graph of Figure 20 is shown in Figure 22.[4]

As illustrated by the potential-call graph, the nesting structure of the skeleton in Figure 21 realizes not only the desired calling structure, but many others as well. In particular, any program whose calling structure is a subgraph of the

[4] For simplicity, cycles of length one, i.e., self-recursive subprogram calls, have not been shown.

```
procedure A;

   procedure B;

      procedure D;
      begin
         ...
      end D;

      procedure E;
      begin
         ...
      end E

   begin
      ...
   end B;

   procedure C;

      procedure F;
      begin
         ...
      end F;

      procedure G;
      begin
         ...
      end G

   begin
      ...
   end C

begin
   ...
end A
```

Figure 21: Skeleton Textual Representation for Nesting Graph of Figure 20.

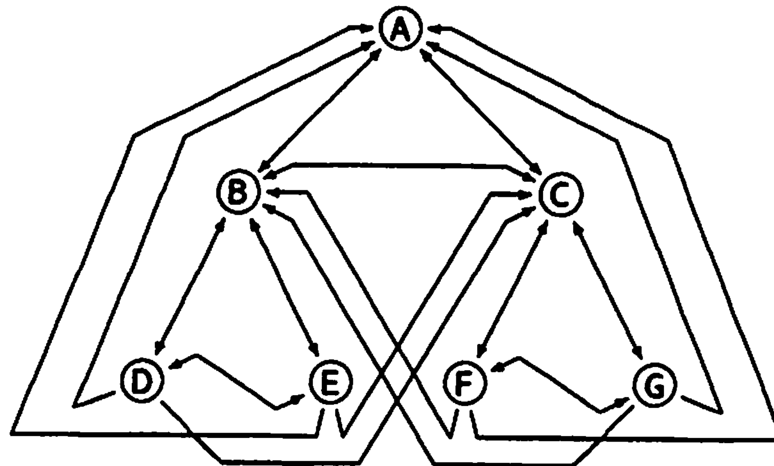


Figure 22: Potential-call Graph for Nesting Graph of Figure 20.

potential-call graph in Figure 22 may be represented using the skeletal text of Figure 21. For example, suppose that the calling structure depicted in Figure 20 is modified so that procedure B is invoked by procedure F as well as procedure A, as shown in Figure 23. This logical structure, while still hierarchical, is no longer a tree. Nevertheless, the nesting structure of Figure 21 supports the calling structure of Figure 23; the potential-call graph derived from this nesting structure (Figure 22) subsumes the call graph of Figure 23. In general, a given calling structure may be realised by several different nesting structures and a given nesting structure may permit numerous distinct calling structures. Hence, the gross nesting structure of a program may easily give a misleading perspective on the logical structure of that program, even one whose logical structure is a tree. To discern the true logical structure, detailed examination of the text must be made, effectively subverting the supposed benefits of using the nesting structure.

Further readability problems arise from the need to represent the two-

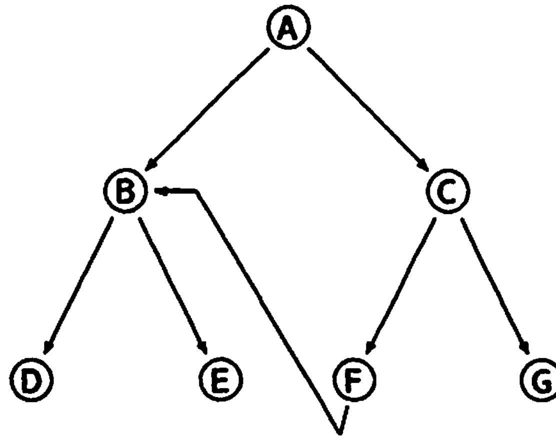


Figure 23: Modified Call Graph of Figure 20.

dimensional nesting structure of a program in the form of a one-dimensional string of text in a program listing. The traditional approach taken for nested languages is to interpret the program text as a post-order traversal of the nesting graph; the text of a nested module appears at a point before the code section of its parent. This extremely strict layout leaves very little room for the developer to present the modules of the program as desired, even if that presentation is as simple as showing high-level modules before lower-level modules. This is illustrated in the program skeleton of Figure 21 where, for example, the full texts of the presumably low-level procedures D, E, F, and G are encountered before the presumably higher-level procedure C.

Another feature of the texts of nested programs that detracts from their readability is the physical separation of a subprogram's heading and local declarations from its code section due to intervening headings, local declarations, and code sections of nested modules. This separation, which comes about from the use of

the nesting structure to specify the non-local visibility of a nested module's local declarations, makes it difficult to understand where and how an entity is used. The code of procedure C in Figure 21, for example, is separated from its heading and local declarations by the entirety of nested procedures F and G. A more concrete example is the CDC 6000 Pascal compiler, itself a Pascal program, in which there are places where over ten pages of text separate procedure headings from corresponding bodies [Hanson, 1981].

Finally, the use of blocks to declare objects and types at intermediate locations within a subprogram is generally considered a poor programming practice that hinders readability [Lecarme & Desjardins, 1974]. Confining all declarations of variables to the declarative part of a subprogram precisely establishes the entities in use within the subprogram and provides a common point of reference for the subprogram's name space.

Modifiability

For a variety of reasons, such as the discovery of errors or a desire for enhanced functionality, software systems typically undergo a significant amount of change during their lifetimes. An important concern, therefore, is that the software be amenable to modifications. Nesting, by virtue of its rigid structure and weak interface controls, interferes with the modification process. Suppose, for example, that the calling structure shown in Figure 20 is modified so that procedure E invokes procedure F. The resulting call graph is presented in Figure 24. Since this call graph is not a tree, it cannot be used as a nesting graph. Therefore, con-

structing a nested program to realize the calling structure requires the additional effort of finding a suitable nesting structure.

In general there are several such trees. One possible nesting structure that supports the pattern of invocations shown in Figure 24 is given as the nesting graph of Figure 25. A potential-call graph can be derived from this nesting graph to show that it does indeed subsume the call graph of Figure 24. In general, however, the translation from an intended calling structure to a suitable nesting structure is not a particularly natural operation. In practice, developers usually discover a suitable nesting structure by moving subprograms invoked by many other subprograms to successively higher nesting levels in their programs. For example, the program skeleton given in Figure 21 can be modified to support the call graph of Figure 24 by moving procedure F up to the point just ahead of procedure B, which results in the nesting graph of Figure 25. Thus, one consequence of nesting is that large programs frequently begin with a long list of low-level utility subprograms, objects, and types.

The problem of finding a suitable nesting structure is even more complicated when two entities have the same identifier, since hiding can then lead to unexpected results. For instance, suppose that procedures A and C each declare an object called X and procedure F contains a reference to X. The original nesting graph of Figure 20 results in F making reference to the object X declared in C. With the modification to the calling structure shown in Figure 24 resulting in the nesting graph of Figure 25, F refers to the different object X declared in A. Subtle, yet onerous, errors such as these are often very difficult to uncover.

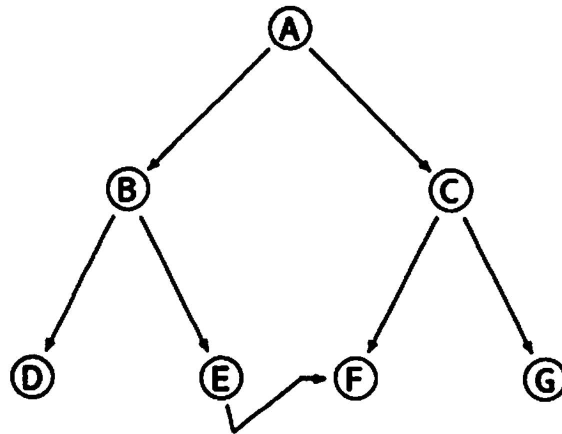


Figure 24: Another Modified Call Graph of Figure 20.

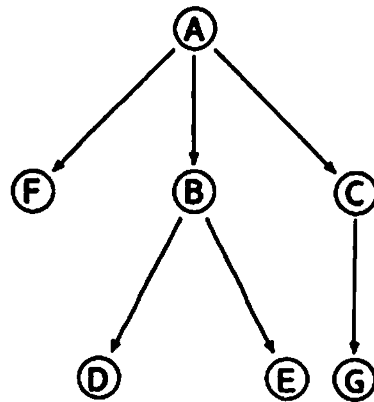


Figure 25: A Nesting Graph Realising Calling Structure of Figure 24.

Incremental Analysis

As discussed in Chapter I, incremental analysis is an important aid to the development of software systems. It allows groups of one or more modules to be treated in isolation while at the same time permitting the consistency of those modules' interfaces to be checked. Unfortunately, nesting makes incremental analysis both complicated and expensive. To incrementally analyze a nested module, that module's entire enclosing environment must be known, and thus saved. This environment may in fact be larger than required if the module does not actually need all of the entities it is provided access to by virtue of its position in the nesting structure. Furthermore, when a module is reanalyzed all modules nested within it are reanalyzed, even when it may be unnecessary to do so. Figure 26 contains an example in Ada illustrating these points. (In Ada, incremental analysis occurs at compilation time and is called *separate compilation.*) In the figure, procedure A has declared within it a local object X and a body stub (indicated by the reserved words **is separate**) for another procedure B that, although it is impossible to tell from looking at procedure A, does not use object X. Hence, despite the fact that procedure B does not need the declaration of object X for its compilation, the declaration will be saved as a (potential) A/B interface item. In addition, if a change to object X is made, then procedure B must be unnecessarily recompiled. Many researchers have tried to accommodate separate compilation in nested languages (e.g., [LeBlanc & Fischer, 1979; Celentano et al., 1980]). But, as Hanson astutely observes [Hanson, 1981], incorporation of separate compilation into nested lan-

guages tends to reduce and even inhibit the role of nesting. This observation is substantiated by evidence collected by De Prycker [De Prycker, 1982b].

By eliminating nesting and substituting for it a mechanism that allows the precise description of the interfaces among modules, incremental analysis with strong interface checking is more practical. With such a mechanism, only the actual interfaces between modules need to be saved between analyses of modules and, more importantly, only those modules that need to be reanalyzed are reanalyzed. Figure 27 presents an alternative solution, in a nest-free setting feasible within Ada, to the problem of Figure 26. (Horizontal lines demarcate separate compilation units.) The first compilation unit is a specification of procedure B's interface. This is used in the second compilation unit, procedure A, to check procedure A's use of procedure B. The third compilation unit, containing local declarations and the code section, is the body of procedure B. (In Ada, subprogram headings are repeated in bodies as a syntactic aid to readability.) Now, with the interface between procedures A and B exactly specified, procedure A can be altered and recompiled without forcing recompilation of procedure B. Only if a change is made to the specification of procedure B will procedure A and (the body of) procedure B both need recompilation.

```
procedure A is
   X : ...;

   procedure B ( ... ) is separate;

begin
   ...        use of X and B

end A;
```

Figure 26: Separate Compilation in a Nested Setting.

```
procedure B ( ... );
__________________________

   with B;
procedure A is
   X : ...;

begin
   ...      use of X and B

end A;
__________________________

procedure B ( ... ) is
   ...

begin
   ...

end B;
```

Figure 27: Separate Compilation in a Nest-free Setting.

§2.2 *Dynamic Memory Management*

Dynamic memory management is the execution-time control of the memory used by a program. Its purpose is two-fold. First, it allows a program to reduce the amount of memory it requires for execution by releasing memory it no longer needs. Such a capability is useful when a program must occupy more total memory than is available from the system.[5] Second, dynamic memory management allows memory to be requested and received as it is needed. This capability is necessary when employing dynamic data structures or recursion since, in general, the total memory needed by the program cannot be predetermined.

Dynamic memory management can be achieved in a program either explicitly or implicitly. Explicit dynamic memory management is based on the use of executable language statements that allocate and deallocate portions of memory. With these statements a developer can control the period during which an instantiation of an object is available, usually referred to as the "lifetime" of the object. Alternatively, memory can be managed implicitly by exploiting the fact that an object's lifetime is often coincident with the activation of a particular region of program text. The local objects of a subprogram or block, for instance, can be made to be available only during the activation of the subprogram or block. Rather than having the developer explicitly allocate memory for these objects at the start of a subprogram's or block's execution and then explicitly deallocate that same memory at the end of the subprogram's or block's execution, the

[5]Of course, this may become a moot point for programming languages, given the growing use of the memory management technique of *virtual memory*.

memory can be managed implicitly by the underlying language system.

Dynamic memory management, whether explicit or implicit, is feasible in either a nested or a nest-free setting. In particular, explicit dynamic memory management depends on the execution sequence of a program's statements and not on the structure of its declarations [Ichbiah et al., 1979]. Similarly, subprogram-level implicit dynamic memory management depends on the calling structure of the program, and not on the program's nesting structure, since memory for objects declared within a subprogram is only allocated when the subprogram is invoked and deallocated when the subprogram returns. The FORTRAN77 Standard, for example, does not permit the nesting of subprograms, but does allow subprogram-level dynamic memory management [Katsan, 1978]. Finally, block-level implicit dynamic memory management can be viewed in a manner analogous to that of its subprogram-level counterpart if a block is treated as a parameterless subprogram executed "in-line".

One consequence of nesting is that certain dynamically created objects can be shared. For instance, if a subprogram s declares a local object o and nested subprograms s_1 through s_n, then at each invocation of s the implictly dynamic object o can be shared among subprograms s_1 through s_n. At first glance, this situation may seem to pose a problem in a nest-free setting; if subprograms s_1 through s_n are removed from within subprogram s, and object o is also removed and placed in a package provided to the subprograms sharing o, then an invocation of s cannot (implicitly) create a new instantiation of the object. There is, however, a suitable nest-free solution, which involves the use of the parameter method discussed in

the previous section. Instead of removing object *o* from within subprogram *s* and sharing it through explicit import/export, object *o* can be passed as a parameter from subprogram *s* to any subprograms sharing it. Thus, object *o* is local to subprogram *s*, implicitly and dynamically created upon invocation of *s*, and shared among a number of subprograms—all without nesting. As discussed in the previous section, the use of parameters to share objects does have drawbacks in certain situations. In particular, an entity may be visible to more subprograms than necessary. Nevertheless, this problem is no worse than when nesting is used.

§2.3 *Implementation Considerations*

ALGOL60 ushered in an era of dramatic advances in language implementation technology. The techniques developed for ALGOL60's implementation have become so important that they constitute the de facto standard example in most texts on compiler design! However well understood the techniques may be, support for nesting makes the implementation of a language unavoidably complex and expensive, from manipulation of the symbol table during compilation [Aho & Ullman, 1977, page 327], to allocation of memory during execution [Hanson, 1981]. This is especially true for languages that attempt to add features, such as separate compilation or more stringent interface controls, to the basic nesting mechanism. An excellent example of this complexity can be found in the implementation of Euclid described in [Holt & Wortman, 1982].

It can be argued that the development of the compilation portion of a language system is a one-time cost and, with the advent of "compiler-compilers"

[Meijer & Nijholt, 1982] a potentially insignificant cost at that. Moreover, it can be argued that the cost of compiling a piece of software does not compare with the costs of the rest of the software development process. Therefore, the assessment given here of nesting's effect on language implementation does not further consider compilation costs. The difference between the run-time efficiency of programs written in a nested language and a nest-free language, however, must be addressed and can be illustrated most clearly by examining subprogram-level dynamic memory management and recursion.

Subprogram-level Dynamic Memory Management

Typically, subprogram-level dynamic memory management is accomplished through the use of a (conceptual) run-time stack of memory [Aho & Ullman, 1977]. To hold entities such as parameters and local objects, a sequence of stack locations, called the *activation record*, is allocated upon invocation of a subprogram and deallocated upon termination. The location of the first element in a sequence is called the *base*, and a *base register*[6] is designated to maintain the base of the currently active subprogram. The activation record entries for this subprogram are then referred to as offsets from the contents of its base register.

The complication that nesting introduces into subprogram-level dynamic memory management is the possibility of reference to objects in enclosing subprograms. The enclosing subprograms, whose activation records are based elsewhere

[6]Although the term "register" implies a hardware implementation, the functionality of the base register could, of course, be realized in software.

in the stack, must be distinguished from other, non-enclosing (and so inaccessible) subprograms. Doing so usually involves employing either the display or static-chain method to maintain the bases of the enclosing subprograms [Aho & Ullman, 1977]. Since the nesting environment of a subprogram is static, determination of the appropriate display or chain for a particular subprogram only complicates the compilation process and does not affect the run-time efficiency of the program. What does potentially have an impact on run-time efficiency is the manner in which the display or chain is referred to during run-time address calculation. If the display or chain is maintained in registers so that the base of an enclosing subprogram is as easily (i.e., inexpensively) accessible as the base of the subprogram itself, then the costs of references to local and non-local objects are essentially equivalent. If, on the other hand, the display or chain is kept in memory, then reference to a non-local object requires at least one additional memory access (and probably more in the static-chain method) to fetch the base of the enclosing subprogram. Because no architecture (not even one designed specifically to support nested languages, such as the Burroughs B6700 ALGOL machine [Organick, 1973]) can provide enough registers to hold an arbitrarily large display or chain, this latter scenario is the most realistic. Some implementations of nested languages attempt a compromise by storing only the bases of the outermost subprogram and the currently active subprogram in dedicated registers [De Prycker, 1982a]. Such a compromise tends to downplay the role of nesting. Moreover, it effectively penalizes programs that refer to objects in other than the currently active or outermost subprograms by making those references relatively

more expensive.

In an implementation of a nest-free language, there is no need for a display or chain and the associated manipulation overhead because there are no enclosing subprograms. The static objects associated with packages, which are the only non-local objects that a subprogram can access, can be kept in distinguished locations in memory, such as at the bottom of the run-time stack. These stack addresses can be easily determined at compile time. Therefore, an implementation of a nest-free language should be no less, and perhaps even more efficient, than an implementation of a nested language that must incorporate a display or chain.

Recursion

Recursion is often thought to be an especially inefficient language feature and so is shunned by many developers [Wulf, 1980]. In fact, this belief in recursion's supposed inefficiency arises, not from some intrinsic property of recursion, but from its coincidental presence in most (if not all) nested languages. The programming language C is an example of a language that supports subprogram recursion but not subprogram nesting.

The stack used to implement subprogram-level dynamic memory management is also typically used to implement subprogram recursion [Pratt, 1975], since recursion is itself a dynamic language feature. The implementation of recursion can be viewed as a form of subprogram-level dynamic memory management: When a subprogram is recursively invoked, a distinct copy of its activation record is placed on the stack; upon termination, the activation record is removed. Thus, a

recursive instantiation of a subprogram is handled as any other instantiation of a subprogram, making the costs of "normal" subprogram invocations equivalent to those of recursive subprogram invocations. In the case of nested languages, however, these costs are inflated by the necessary use of a display or chain, as described above. The inefficiency often attributed to recursion, therefore, is actually the inefficient interaction of subprogram-level dynamic memory management with nesting.

§2.4 *Discussion*

Looked at individually, the problems of nesting pointed out in this section can be characterized best as *annoyances* that make software development that much more difficult and costly. When taken togther, however, these problems add up to a strong argument against the use of nesting in software development and, by extension, the inclusion of nesting in a language. To summarise:

- It is difficult to accurately express, as well as to discern, the logical structure of a nested system.
- It is difficult to modify a nested system.
- It is costly to incrementally analyse a nested system.
- It is costly to support subprogram-level dynamic memory management and recursion in a nested language.

Despite these problems, nesting is still included in the designs of many new languages. We suspect that this is because language designers rarely consider the

cumulative effects of the problems. Indeed, rather than attacking the problems at their root by simply eliminating nesting, many language designers attempt to "fix" the basic nesting mechanism. Several of those fixes are discussed in the next section.

§3. Attempts to Enhance Nesting

Since the development of ALGOL60, numerous attempts have been made to compensate for the inadequacies of nesting. These efforts have essentially taken two directions. The first shuns nesting altogether and instead employs alternative mechanisms, such as encapsulation and explicit import/export control, for supporting desired language capabilities. Notable language designs that adopt this nest-free approach include C, CLU, and Gypsy. The other direction retains nesting but enhances the basic features of the mechanism. This second direction is taken in the designs of Ada, Alphard, Euclid, and MODULA-2, as well as in various implementations of (pseudo) Pascal (e.g., [Celentano et al., 1980; Young, 1981; Rudmick & Moore, 1982]). Certain of these enhancements, such as those for incorporating separate compilation into nested languages, are discussed above. This section concentrates on enhancements to nesting in two other areas: interface control and program readability. Although only a few languages incorporating these enhancements are mentioned below, they are representative of the significant ideas in the two areas.

§3.1 *Interface Control Enhancements*

As discussed above, one of the major problems associated with nesting is that the location of an entity's declaration in the nesting structure of a program gives that entity broad access to other entities at the same or higher levels in the structure. To eliminate this weakness in interface control, some languages allow nested subprograms to be *closed* to non-local declarations; that is, a nested subprogram is not permitted to refer to entities at the same or higher levels in the nesting structure. This is the case, for example, in Euclid. Thus, in the skeleton Euclid program of Figure 28a, procedures B and C cannot refer to objects X, Y, and Z. To allow access from nested subprograms, entities declared in enclosing scopes must be explicitly *imported* from the immediately enclosing scope,[7] as shown in Figure 28b. The idea, therefore, is to combine nesting with stringent control over requisition. In fact, Euclid can be shown to have a requisition-precise interface control mechanism like Ada's. This mechanism, while no more precise than Ada's, is in some sense more flexible than Ada's because it can be used in combination with nesting; Ada's mechanism is requisition-precise only in the absence of nesting (see Theorem 2 in Chapter IV).

Clearly, this enhancement to the basic nesting mechanism can be used to reduce the extent to which a subprogram can be referred to by other entities within a program. For instance, in some cases this enhancement allows the accurate ex-

[7]In Euclid, a declaration can be specified as *pervasive*. This indicates that the entity is to be treated using the standard rules of nesting and so accessible from subprograms even if they do not explicitly import the entity. This is simply a syntactic shorthand that obviates the need to mention the pervasive entity in import lists attached to all the relevant modules.

```
procedure A =
begin
  var X, Y, Z ...

  procedure B =

  begin

    procedure C =

    begin
      Y := ...   ERROR: use of Y in C

    end C

    X := ...     ERROR: use of X in B

  end B

  X := ...       use of X in A
  Y := ...       use of Y in A
  Z := ...       use of Z in A

end A
```

(a)

```
procedure A =
begin
  var X, Y, Z ...

  procedure B =
    imports ( var X, var Y )
  begin

    procedure C =
      imports ( var Y )
    begin
      Y := ...   use of Y in C

    end C

    X := ...     use of X in B

  end B

  X := ...       use of X in A
  Y := ...       use of Y in A
  Z := ...       use of Z in A

end A
```

(b)

Figure 28: Closed Scopes in Euclid.

pression of the locality of a declaration, such as entity Z in Figure 28. In general, however, such precise requisition can only be achieved by avoiding the nesting component of the enhanced mechanism—that is, by placing all subprograms directly within a single, flat, "dummy" scope and using the requisition controls to define the desired visibility (cf., Theorem 2 in Chapter IV). This is because the enhancement can only *refine* the overall visibility dictated by the nesting structure. This is illustrated in the Euclid example of Figure 28b where procedure B must import Y, and thus gain access to the entity, solely to give procedure C the opportunity to import Y.[8]

Another interface control enhancement found in nested languages approaches the problem from the point of view of provision. It supports the selection of specific entities declared within some scope to be provided outside that scope. This represents a significant departure from the basic nesting mechanism in that it permits reference to entities lower in the nesting structure than just those defined at the local level. Cormack [Cormack, 1983] describes an extremely general version of such an enhanced nesting mechanism in which an entity declared within any scope-defining construct, such as blocks and subprograms, can be specified as provided to any enclosing scope. A more typical version is found in Ada. There, the enhancement simply takes the form of nested packages. A nested package defines a special scope having a specification part that details which entities de-

[8]The only enhanced nesting mechanism we are aware of that permits importation of an entity from an enclosing scope that is not the immediately enclosing scope, appears in an early version of Ada [Ichbiah et al., 1979] and is called the *restricted unit* facility. This feature was dropped from later versions of the language, probably because of its extreme complexity.

clared within that package are provided to the scope containing the declaration of the package. Once provided to an enclosing scope, an entity is then essentially as accessible as if it were actually declared in that scope. Figure 29 shows a nested-package solution to the stack abstraction example of Figure 14 (cf., Figure 15). (Note that packaged subprograms in Ada are not closed scopes.) Figure 29 illustrates the primary use of the enhanced mechanism; unlike pure nesting, the enhanced mechanism allows subordinate entities (Stack and StackPointer) to be shared among certain other entities (Initialize, Push, Pop, and Empty) without those subordinate entities also being provided to the user (StackUser) of the sharing entities.

As the example in Figure 29 demonstrates, this second enhancement to nesting does indeed provide a solution in an important interface control situation; it succeeds in restricting the possible referents of a set of shared entities. This solution, however, does not generalise. Consider the skeleton Ada program in Figure 30. Three procedures, A, B, and C are provided from a package Pac. Procedures A and B share object X, while procedures B and C share object Y. Nonetheless, A has access to Y and C has access to X. No use of nested packages, not even ones nested inside of Pac, will result in the desired visibility relationships among the entities in Pac. Moreover, this enhancement, although aimed toward improving the provision component of the basic nesting mechanism, is not sufficient to render it provision-precise since the provided entities are made available to an entire scope in the nesting structure and not particular entities within that scope (cf., Theorem 2 in Chapter IV).

```
procedure Outer is
   ...

   package StackPac is
      procedure Initialize;
      procedure Push ( Item : in Element );
      function Pop return Integer;
      function Empty return Boolean;

   end StackPac;

   package body StackPac is
      Stack : array ( 1 .. 100 ) of Integer;
      StackPointer : Integer;

      procedure Initialize is
      begin ... end Initialize;

      procedure Push ( Item : in Element ) is
      begin ... end Push;

      function Pop return Integer is
      begin ... end Pop;

      function Empty return Boolean is
      begin ... end Empty;

   end StackPac;

   procedure StackUser is
   begin
      ...      StackUser does not have access to Stack and StackPointer

   end StackUser;

begin ... end Outer;
```

Figure 29: Package Nesting in Ada.

```
procedure Outer is
   ...

   package Pac is
      procedure A;
      procedure B;
      procedure C;

   end Pac;

   package body Pac is
      X : ...;
      Y : ...;

      procedure A is
         ...

      begin
         ...   desires access to X; has access to X and Y

      end A;

      procedure B is
         ...

      begin
         ...   desires access to X and Y; has access to X and Y

      end B;

      procedure C is
         ...

      begin
         ...   desires access to Y; has access to X and Y

      end C;

   end Pac;

begin ... end Outer;
```

Figure 30: Weakness in Package Nesting.

Euclid offers both enhancements to the basic nesting mechanism: stringent requisition controls and nested packages (called *modules* in Euclid). By combining these two enhancements, a greater variety of visibility relationships built upon nested program structures can be accurately expressed. Together, they solve the problem encountered in the Ada example of Figure 30. Specifically, an import list can be associated with each of the procedures in Pac detailing exactly the entities desired. From a formal perspective, however, the combination of enhancements in Euclid does nothing to increase the general precision of its interface control mechanism since, although the requisition controls can be used to achieve precise requisition (often at the exclusion of nesting), the module facility does not lead to precise provision.

§3.2 *Readability Enhancements*

An argument often used against nesting, and discussed in the previous section, is that it imposes a rigid and often unnatural layout of the program text, making the reading and understanding of the program difficult. Even with the use of so-called "prettyprinting" techniques [Oppen, 1980], the fact remains that in a typical nested program the heading and local declarations of subprograms can be widely separated from their code sections due to the intervening headings, local declarations, and code sections of nested modules. Furthermore, the layout of the text is dictated more by interface control concerns than those of readability, often complicating the logical presentation of the program. Two representative approaches to solving these problems are examined in the remainder of this sec-

tion: a program structuring feature of an extended Pascal and the Ada *subunit* facility.

Tennent [Tennent, 1982] argues that the shortcomings associated with nesting result, not from weaknesses inherent in the mechanism's basic concepts, but rather from poor realizations of those concepts in popular languages such as ALGOL60 and Pascal. In defense of the readability of nested programs, Tennent discusses an alternative to nesting's familiar textual layout. As it appears in an extended Pascal, this alternative allows the placement of a nested subprogram's body, which contains the local declaration and code sections, *after* the body of the subprogram in which that subprogram is defined. All that remains within the text of the enclosing subprogram is the heading of the nested subprogram. The heading is used to define the position of the subprogram in the overall nesting structure and to act as a *forward declaration* in the Pascal terminology or a *body stub* in the Ada terminology. Figure 31 contains an example illustrating the difference between the two layouts. Headings corresponding to nested subprograms are followed by the reserved word **forward**. Notice that the extended-Pascal version avoids the problem of physical separation. This layout has a major drawback, however, which it shares with nesting's more familiar textual layout; it does not allow a developer to organise the text as desired. This becomes evident when the (one-dimensional) text of a program is viewed as a traversal of the (two-dimensional) nesting structure of that program. In particular, the textual layout of the extended Pascal advocated by Tennent simply substitutes what amounts to a pre-order traversal of the nesting tree for standard Pascal's post-order traversal.

```
procedure A ( ... );
   ...
   procedure B ( ... );
      ...
      procedure D ( ... );
         ...
      begin ... end;
      ...
   begin ... end;
   ...
   procedure C ( ... );
      ...
   begin ... end;
   ...
begin ... end;
```

(a)

```
procedure A ( ... );
   ...
   procedure B ( ... ); forward;
   ...
   procedure C ( ... ); forward;
   ...
begin
   ...
where
   ...
   procedure B;
      ...
      procedure D ( ... ); forward;
      ...
   begin
      ...
   where
      ...
      procedure D;
         ...
      begin ... end;
      ...
   end;
   ...
   procedure C;
      ...
   begin ... end;
   ...
end;
```

(b)

Figure 31: Textual Layout of Nested Program in Standard Pascal (a) and Extended (b) Pascal.

Thus, the bodies of low-level utilities, for instance, often appear early in the program text. This is the case in Figure 31 where the body of procedure D, a local utility of procedure B, unavoidably comes before the higher-level body of procedure C.

The inflexibility of the two structuring methods compared above is significantly reduced with use of Ada's subunit facility. Although designed to allow the separate compilation of nested module bodies, this facility can be used as a method of structuring the program text. Figure 32 shows the use of Ada subunits to solve both the physical-separation and text-structuring problems encountered in the example of Figure 31. The subunit facility appears to overcome the major drawbacks of a nested program structure. For example, the text of programs developed in a top-down fashion can be organised in a top-to-bottom manner, as illustrated in the Ada reference manual [DoD, 1983, pages 10-8–10-9], with only the body stubs of referenced subunits appearing prior to the actual reference. Since the subunit facility preserves the basic nesting mechanism, however, it exhibits some of nesting's other shortcomings with respect to readability. Specifically, the textually separate subunit body is still considered to be logically located at the point where the body stub appears—that is, nested within the declarative part of another block or module. It is the location of this body stub that determines the context (i.e., the visibility of other entities) within which this subunit is to be understood. Since the subunit is now textually distinct from the body stub whose location determines its logical context, this can make both writing and understanding the subunit extremely difficult.

```
procedure A ( ... );
   ...
   procedure B ( ... ) is separate;
   ...
   procedure C ( ... ) is separate;
   ...
begin ... end A;

   separate ( A );
procedure B ( ... ) is
   ...
   procedure D ( ... ) is separate;
   ...
begin ... end B;

   separate ( A );
procedure C ( ... ) is
   ...
begin ... end C;

   separate ( A.B );
procedure D ( ... ) is
   ...
begin ... end D;
```

Figure 32: Ada's Subunit Facility Applied to Example of Figure 31.

CHAPTER VI

INTERFACE ANALYSIS

The precision and redundancy of the language features presented in Chapter II 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 PIC approach as an integrated collection of analyses, each of which concentrates on some particular aspect of interface control. The analyses exploit the language features by offering more detailed and revealing assessments of a system's interface relationships than has previously been possible. They 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. 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* [Osterweil, 1983], the approach 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 chapter elaborates on each of

the analysis classes in turn, describes how they handle incompleteness, and then demonstrates how the PIC approach to interface control supports incremental development. To give the discussion concreteness, these topics are presented in the context of the semantics of PIC/ADL, which is described in Chapter III.

§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 33. (The significance of the dashed line appearing in this figure is explained below.) 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. 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

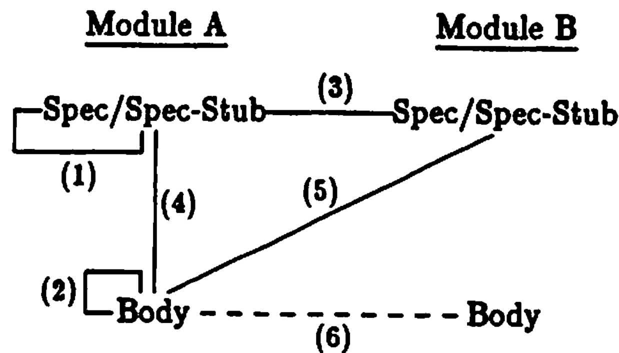


Figure 33: Basic Interface Analyses.

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 as an error or as an anomaly in such cases so that they can accommodate a variety of applications, languages, methods, and disciplines.

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 [Levy, 1984]. 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 parameterisation of those calls may simply be due to overloading. Because the type analysis in PIC/ADL is what would generally 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; algorithms for detecting the problems are given in Appendix B. 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. Of course, a complete

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.

analysis tool would seek to detect those problems as well. Second, the precision and redundancy of constructs derived from the PIC language framework 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. 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*).

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, an actual implementation of the analyses would 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

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.

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 re-

ferred 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 33) if the reasonable assumption is made that a body submodule would not be analysed 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*.

§2. Stub Analyses

As described in Chapter II, 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.

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 attempts to provide the entity *exclusively* to some module (see Chapter III).

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

ANOMALIES	
a.	entity e_1 provided to entity e_2 in the specification, but e_1 not provided to e_2 in the specification stub
b.	entity e_1 provided to entity e_2 in the specification stub, but e_1 not provided to e_2 in the specification
c.	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
d.	entity requested in the specification stub, but not requested in the specification
e.	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.

error. For example, consider entry *b*. 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 could be used by a processing tool in a development environment 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.

§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[1] 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 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 library units.

[1]The manner in which one submodule is recognized as a version or *update* of another submodule depends upon the kind of submodule involved. In the case of specification submodules, it is simply the fact that the two submodules are both specification submodules of the same module (e.g., for a PIC/ADL package A, the submodules both begin with the statement **package A is**). Likewise for body submodules (e.g., they both begin with the statement **package body A is**). For specification stub submodules, versions are recognized by their being specification stubs of the same module and—because multiple specification stub submodules of a module, corresponding to different users of that module, can coexist in a system—by their having common elements in their used-by clauses (e.g., they begin with the statements **package stub A is used by B** and **package stub A is used by B, C**).

While the update analyses do not directly assess interface consistency, which is one of 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.[2]

§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 begun in Chapter II: the specification submodule of AutomaticTeller (Figure 1 on page 22) and the specification submodules of ATMaintenanceInterface and OfficerInterface (Figure 34).

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 ATMaintenanceInterface and OfficerInterface, while access to DepositsMade is provided to ATMaintenanceInterface 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

[2] Tichy and Baker [Tichy & Baker, 1985] discuss this in the restricted context of recompilation savings.

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

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

Figure 34: Specification Submodules of Packages ATMaintenanceInterface and OfficerInterface.

ATMaintenanceInterface indicates that access is requested to RemainingCash and DepositsMade. This is certainly consistent with the provision specified in AutomaticTeller since ATMaintenanceInterface appears in the provide clauses of both RemainingCash and DepositsMade. As is the case for ATMaintenanceInterface, the specification submodule of OfficerInterface indicates that access is requested to RemainingCash and DepositsMade. But unlike that case, access to DepositsMade is not explicitly provided to OfficerInterface by AutomaticTeller since OfficerInterface does not appear in the provide clause attached to DepositsMade. Under our definition, however, OfficerInterface'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 OfficerInterface 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 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 ATMaintenanceInterface are consistent but AutomaticTeller and OfficerInterface are only conditionally consistent.

§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 analysed 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 analysed 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 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

[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 ([Ichbiah et al., 1979, 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.

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 (e.g., [Baker & Youngblut, 1985]), 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 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 analysed 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 the PIC approach that permit it to fully support incremental development are (1) the formulation of 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 analysable 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 Chapter II, 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 Chapter II, 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 33) can be performed in the PIC approach 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 devel-

opment than to supply a specification stub submodule of either AccountManager or PINManager that can be used by AutomaticTeller. Figure 2 on page 29 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 33) and the other checking references in the body submodule of AutomaticTeller (number 5 of Figure 33).

Eventually, the "official" specification submodule of PINManager is made available (Figure 35). 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 the PIC approach that checks the consistency between AutomaticTeller and PINManager. (Consistency can also be checked at this point between OfficerInterface 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 33). The choice depends mainly upon whether the specification stub submodule sufficiently represents the use of PINManager by AutomaticTeller. There appear to be a number of ways to automatically determine the best choice, based on previous analyses, and to limit the amount of

```
package PINManager is
      request ESManager;

   type PIN is private;

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

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

   ...;

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

end PINManager;
```

Figure 35: Specification Submodule of Package PINManager.

unnecessary (re)analysis. Discussion of this, however, is beyond the scope of this dissertation.

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.

CHAPTER VII

CONCLUSION

§1. Summary

This dissertation presents an improved approach to supporting the development and control of the module interfaces of large software systems. The approach, which we call PIC, consists of a framework of language features and a set of analyses. It is distinguished from previous efforts in that it facilitates precise interface control and addresses the needs of pre-implementation lifecycle phases in addition to those of the implementation and post-implementation phases. The preciseness of the control comes about from our taking a new perspective on entity visibility. This perspective recognizes two complementary concerns of interface control, namely requisition of access and provision of access. Early lifecycle support is achieved through language features that facilitate the explicit expression of incompleteness and analyses that account for that incompleteness. Moreover, the analyses are built from tool fragments that can be flexibly composed and applied.

Rather than simply proposing a new interface control mechanism, a systematic approach to the design of the mechanism was taken. First, several existing

mechanisms were evaluated, foremost of which is nesting. In the course of this evaluation, a formal model for rigorously describing and evaluating interface control mechanisms was defined to help characterize and reason about interface control concerns. Next, the new mechanism was designed and evaluated using this formal model. It was shown to be a precise mechanism, and one that is in fact more precise than existing mechanisms. Analysis techniques were then developed to exploit the preciseness of the mechanism. Finally, a practical evaluation of this entire approach to interface control was begun. The practical evaluation involves the design of a prototype implementation of the PIC approach.[1] This prototype, which is being based on Ada, consists (initially) of an Ada-like design language called PIC/ADL and analyses appropriate to that language. The implementation of the prototype is being carried out incrementally, using the PIC language features and analysis techniques (in the guise of PIC/ADL) to facilitate a top-down incremental development. Based on this preliminary use of the approach, which is serving as a significant and realistic initial test case for our ideas and providing us with useful feedback, we are encouraged about the contributions that the completed implementation will make to improving the software development process.

[1]At least one industrial organization has also begun to realize the approach proposed here [Feblowitz & Moore, 1984]. That organization is retrofitting the language CHILL.

§2. Future Directions

§2.1 *A Complete PIC Environment*

The PIC approach to interface control is intended to provide additional, significant capabilities to software development environments, including those for constructing Ada systems [Intermetrics, 1981; Babich, Weissman, & Wolfe, 1982; Standish & Taylor, 1984]. It is envisioned that such an environment would incorporate the PIC language features into various lifecycle languages, resulting in several PIC "dialects". In addition, the PIC analyses would be incorporated into the basic suite of support tools. All modularisation and interface-control decisions made throughout the development and maintenance process would be recorded using the dialects and then organised, managed, and analysed using the support tools. Figure 36 depicts this organisation for the PIC prototype's Ada-based environment, showing various kinds of support tools applied to a variety of PIC dialects of Ada, such as a textual pre-implementation language (PIC/ADL), a textual implementation language (PIC/Ada), and a graphical language (PIC/Graphics). As a result, the PIC approach, with all its descriptive, analytical, and managerial capabilities, would be actively involved in all phases of development and maintenance.

The uniform application of the support tools to descriptions for different software lifecycle phases hinges on the development of a consistent internal representation of those descriptions. Such a representation would emphasise the common semantic content of those descriptions while suppressing (irrelevant) syntactic dif-

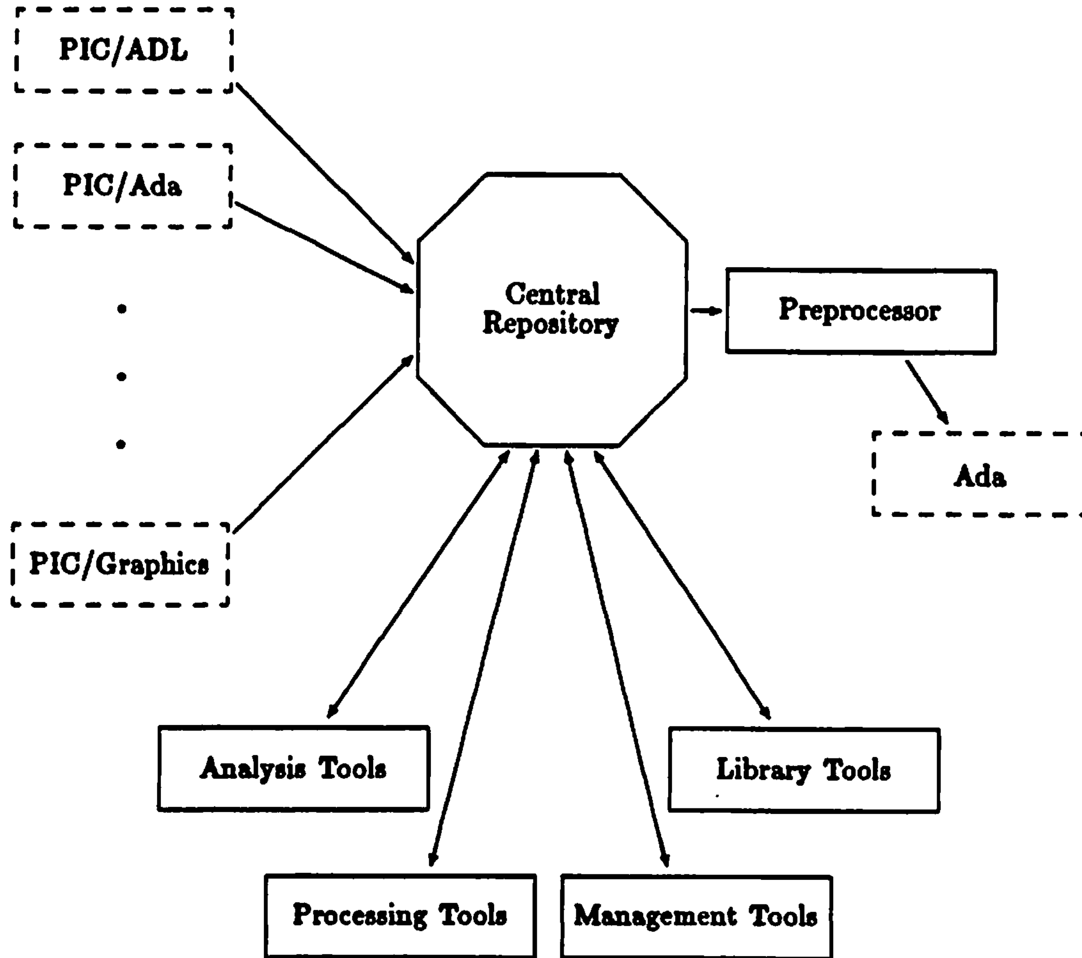


Figure 36: Conceptual Organisation of the Ada-based, Prototype PIC Environment.

ferences. The interface control aspects of the various languages for specification, design, and implementation would be translated into this internal representation. The support tools could then be applied to that representation. We have begun to develop such an internal representation that is based on the formal model of visibility presented in Chapter IV.

The remainder of this section further discusses each of the various kinds of support tools depicted in Figure 36, highlighting open issues in their designs. Of those tools, only the analysis tools have already received extensive treatment in this dissertation. Detailed investigation into the designs of the other kinds of tools shown in the figure is beyond the scope of the dissertation, but is a natural outgrowth of this work.

From Analyses to Analysis Tools

The analyses described in Chapter VI represent the primitive feedback capabilities made possible by the precision, redundancy, and explicit treatment of incompleteness in the PIC language framework. Although it is conceivable to think of these analyses as separate tools in a software development environment, they are probably best thought of as representative compositions 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 an environment, therefore, depends upon whether, and how, the designers or managers 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 underlying software development 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 an 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 in Chapter VI, reports of numerous anomalies would be anticipated when analysing 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.

Library Tools

It is common practice for large software systems to be broken down into more manageable *libraries*, each consisting of several logically related modules.

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

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

The development library resembles a program library in that it maintains in-

formation about the submodules it contains. More than just holding compiled code and interface information, the development library may store various forms of the source code. More importantly, the development library maintains information concerning the results of analyses among its submodules. As a further generalization of the program library concept, a development library may even contain other development libraries; this is where the capabilities of a file manager come in to play. File systems, such as the one in the UNIX operating system, provide a simple mechanism for partitioning a work area. In addition, they provide the means for sharing among work areas; a UNIX file or directory may be a member of more than one directory. By incorporating this file-system model into the structure of the development library, it becomes possible to conceive of using multiple libraries in the development of a system and to share information, particularly interface and analysis information, among those libraries in a fairly general way.

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

when the sharing is only at the level of source code files.

Management Tools

One of the necessary capabilities of a development environment for large software systems is support for managerial control. Thus, there must be management tools that provide mechanisms to control the establishment and modification of a system's interface relationships. Since those relationships are completely determined by the specification and specification stub submodules in the PIC language framework, the management tools can operate by controlling programmer access to these submodules. The management tools, therefore, must work in close cooperation with the library tools. Furthermore, they should enforce whichever managerial discipline, or range of disciplines, is chosen for the project. In particular, if the project leader is given sole responsibility for determining interface relationships, then only that person should be permitted to enter or replace specification and specification stub submodules in the development library. Under a more decentralized discipline, implementors would be permitted to enter or replace specification stub submodules, but only the project leader should be permitted to enter or replace the "official" specification submodules. Finally, an autonomous managerial discipline would permit any project member to enter or replace specification submodules for the modules they are developing or maintaining in the development library.

Processing Tools

A number of general processing tools must be available in a complete PIC environment to perform such tasks as creating and modifying submodules, generating specification submodules from sets of specification stub submodules, and generating views of modules from their interfaces. As are the analysis tools, these tools should be designed to use the consistent internal representation and handle incompleteness appropriately so that they too can be uniformly applied throughout the software development and maintenance process. The common internal representation could additionally facilitate easy movement from graphical representations of a system to corresponding textual representations and *vice versa*, permitting further development of the system to be recorded through refinements in either or both representations.

One very important processing tool in the PIC prototype is the *preprocessor*, which serves to translate PIC/Ada into standard Ada code. Compilation of PIC/Ada is therefore a two-step process in which the interface control information is first incorporated, as far as possible, into Ada (the preprocessing step) and then a normal Ada compilation is carried out.[2] Interface control information that cannot be directly captured in Ada can still be enforced through the environment's analysis tools prior to the preprocessing step. Any changes that are made to the Ada implementation, including those resulting from maintenance activities, would be performed using the PIC dialect languages and would be processed

[2]The reason for using a preprocessor to an Ada compiler is to save development time in constructing a prototype. A complete software development environment would presumably not contain such a tool.

by the support tools.

§2.2 *Extensions to the Language Framework and Analyses*

Various extensions could be considered that might enrich the fundamental capabilities of the PIC approach or facilitate its use by software developers. One class of such extensions would be language features and analyses supporting higher-level or more convenient descriptions of the relationships among modules. It would, for instance, be possible to provide shorthand notations for identifying groups of modules and/or entities when describing interface control relationships. These shorthand notations might be based on a facility for giving names to groups, or for identifying groups through a common attribute such as the name of a programmer [Minsky, 1983], or even for giving a more abstract semantic description such as input/output behavior [Luckham & von Henke, 1985].

Until recently, systems built using functional languages did not require sophisticated lifecycle support since they were generally used only as research prototypes. With the growing interest in knowledge-based and expert systems, however, builders of these systems will have to come to grips with the problems of producing production quality software having extended development and maintenance lifetimes. Because the interface control issues we are addressing are relevant to any large software system, they should apply naturally to those built using functional languages. One of our goals is to see how the PIC approach can be enhanced to accommodate its inclusion in such languages.

Finally, another possible extension would be support for dynamic inter-

face control mechanisms, whereby the requisition and/or provision relationships within a software system might change during the system's execution. Currently, the PIC language features provide an entirely static interface control mechanism. The underlying concepts of requisition and provision could, however, be extended to encompass a dynamic mechanism. Indeed, it is our belief that all of the extensions mentioned above are compatible with the basic PIC approach to interface control.

BIBLIOGRAPHY

[Aho & Ullman, 1977]

A.V. Aho and J.D. Ullman, **Principles of Compiler Design**, Addison-Wesley, Reading, Massachusetts, 1977.

[Ambler et al., 1977]

A.L. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch, and R.E. Wells, *Gypsy: A Language for Specification and Implementation of Verifiable Programs*, **Proceedings of an ACM Conference on Language Design for Reliable Software**, appearing in **SIGPLAN Notices**, Vol. 12, No. 3, March 1977, pp. 1–10.

[ANSI, 1978]

ANSI X3.9–1978 (American National Standard Programming Language FORTRAN).

[Babich, Weissman, & Wolfe, 1982]

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

[Baker & Youngblut, 1985]

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

[Brinch Hansen, 1981]

P. Brinch Hansen, *The Design of Edison*, **Software–Practice and Experience**, Vol. 11, No. 4, April 1981, pp. 363–396.

[Cashin et al., 1981]

P.M. Cashin, M.L. Joliat, R.F. Kamel and D.M. Lasker, *Experience with a Modular Typed Language: Protel*, **Proceedings of the Fifth International Conference on Software Engineering**, San Diego, California, March 1981, pp. 136–143.

[Celentano et al., 1980]

A. Celentano, P. Della Vigna, C. Ghezzi, and D. Mandrioli, *Separate Compilation and Partial Specification in Pascal*, **IEEE Transactions on Software Engineering**, SE-6, No. 4., July 1980, pp. 320–328.

[Clarke, Wileden, & Wolf, 1980]

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.

[Cormack, 1983]

G.V. Cormack, *Extensions to Static Scoping*, **Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems**, appearing in **SIGPLAN Notices**, Vol. 18, No. 6, June 1983, pp. 187–191.

[De Prycker, 1982a]

M. De Prycker, *A Performance Analysis of the Implementation of Addressing Methods in Block-Structured Languages*, **IEEE Transactions on Computers**, Vol. C-31, No. 2, February 1982, pp. 155–163.

[De Prycker, 1982b]

M. De Prycker, *On the Development of a Measurement System for High Level Language Program Statistics*, **IEEE Transactions on Computers**, Vol. C-31, No. 9, September 1982, pp. 883–891.

[DeRemer & Kron, 1976]

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.

[DoD, 1983]

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

[Feblowitz & Moore, 1984]

M.D. Feblowitz and B.G. Moore, *Towards Precise Interface Control in CHILL*, **Proceedings of the Third CHILL Conference**, Cambridge University, Cambridge, England, September 1984.

[Goguen & Tardo, 1979]

J.A. Goguen and J.J. Tardo, *An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications*, **Proceedings of an IEEE Computer Society Conference on the Specification of Reliable Software**, April 1979, pp. 170–189.

[Honeywell, 1980]

Formal Definition of the Ada Programming Language, Honeywell, Inc., Minneapolis, MN, November 1980.

[Habermann, 1979]

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

[Hanson, 1981]

D.R. Hanson, *Is Block Structure Necessary?*, **Software–Practice and Experience**, Vol. 11, No. 8, August 1981, pp. 853–866.

[Hennessy & Kieburtz, 1981]

J.L. Hennessy and R.B. Kieburtz, *The Formal Definition of a Real-Time Language*, **Acta Informatica**, Vol. 16, 1981, pp. 309–345.

[Holt & Wortman, 1982]

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

[Ichbiah & Ferran, 1977]

J.D. Ichbiah and G. Ferran, *Separate Definition and Compilation in LIS and its Implementation*, **Lecture Notes in Computer Science**, No. 54, Springer-Verlag, Berlin, 1977, pp. 288–297.

[Ichbiah et al., 1979]

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

[Intermetrics, 1981]

Ada System Specification for Integrated Environment Type A, Intermetrics, Inc., March 1981.

[Jensen & Wirth, 1974]

K. Jenson and N. Wirth, *Pascal—User Manual and Report*, **Lecture Notes in Computer Science**, Vol. 18, Springer–Verlag, New York, 1974.

[Jones & Liskov, 1978]

A.K. Jones and B.H. Liskov, *A Language Extension for Expressing Constraints on Data Access*, **Communications of the ACM**, Vol. 21, No. 5, May 1978, pp. 358–367.

[Katzan, 1978]

H. Katzan, **FORTRAN 77**, Van Nostrand Reinhold, New York, 1978.

[Koster, 1976]

C.H.A. Koster, *Visibility and Types*, **Proceedings of a Conference on Data: Abstraction, Definition and Structure**, appearing in **SIGPLAN Notices**, Vol. 11, No. 2, February 1976, pp. 179–190.

[Meijer & Nijholt, 1982]

H. Meijer and A. Nijholt, *Translator Writing Tools Since 1970: A Selective Bibliography (June 1982)*, **SIGPLAN Notices**, Vol. 17, No. 10, October 1982, pp. 62–72.

[Lampson et al., 1981]

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

[Lampson & Schmidt, 1983]

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

[LeBlanc & Fischer, 1979]

R.J. LeBlanc and C.N. Fischer, *On Implementing Separate Compilation in Block-Structured Languages*, **Proceedings of the SIGPLAN Symposium on Compiler Construction**, appearing in **SIGPLAN Notices**, Vol. 14, No. 6, August 1979, pp. 139–143.

[Lecarme & Desjardins, 1974]

O. Lecarme and P. Desjardins, *More Comments on the Programming Language Pascal*, **Acta Informatica**, Vol. 4, No. 3, 1974, pp. 231–243.

[Ledgard & Singer, 1982]

H.F. Ledgard and A. Singer, *Scaling Down Ada (Or Towards a Standard Ada Subset)*, **Communications of the ACM**, Vol. 25, No. 2, February 1982, pp. 121–125.

[Levy, 1984]

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.

[Lipton & Snyder, 1977]

R.J. Lipton and L. Snyder, *A Linear Time Algorithm for Deciding Subject Security*, **Journal of the ACM**, Vol. 24, No. 3, July 1977, pp. 455–464.

[Liskov et al., 1981]

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.

[Luckham & von Henke, 1985]

D.C. 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.

[Minsky, 1983]

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.

[Mitchell, Maybury, & Sweet, 1979]

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.

[Moore & Chandrasekharan, 1983]

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

[Ossher, 1984]

H.L. Ossher, *Grids: A New Program Structuring Mechanism Based on Layered Graphs*, **Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages**, Salt Lake City, Utah, January 1984, pp. 11–22.

[Oppen, 1980]

D.C. Oppen, *Prettyprinting*, **ACM Transactions on Programming Languages and Systems**, Vol. 2, No. 4, October 1980, pp. 465–483.

[Organick, 1973]

E.I. Organick, **Computer System Organisation—The B5700/B6700 Series**, Academic Press, New York, 1973.

[Osterweil, 1983]

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.

[Parnas, 1974]

D. Parnas, *On a "Bussword": Hierarchical Structure*, **Information Processing 74, Proceedings of IFIP Congress 74**, J.L. Rosenfeld (ed.), North Holland Press, Amsterdam, 1974.

[Pratt, 1975]

T.W. Pratt, **Programming Languages: Design and Implementation**, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Privitera, 1982]

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

[Ralston, 1983]

A. Ralston (ed.), **Encyclopedia of Computer Science and Engineering, Second Edition**, Van Nostrand Reinhold, New York, 1983.

[Robinson & Roubine, 1977]

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

[Rudmick & Moore, 1982]

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

[Rudmik, Casey, & Cohen, 1982]

A. Rudmik, B.E. Casey, and H. Cohen, *Consistency Checking within Embedded Design Languages*, **Proceedings of the Sixth International Conference on Software Engineering**, Tokyo, Japan, September 1982, pp. 236–245.

[Sammet, Waugh, & Reiter, 1982]

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

[Schmidt, 1982]

E.E. Schmidt, *Controlling Large Software Development in a Distributed Environment* **Technical Report CSL–82–7**, Xerox PARC, Palo Alto, California, December 1982.

[Schwanke, 1978]

B. Schwanke, *Survey of Scope Issues in Programming Languages*, **Technical Report CMU-CS-78-131**, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1978.

[Shaw, 1974]

A.C. Shaw, **The Logical Design of Operating Systems**, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[Shaw, 1981]

M. Shaw (ed.), **Alphard: Form and Content**, Springer–Verlag, New York, 1981.

[Standish & Taylor, 1984]

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

[Steele, 1984]

G.L. Steele, Jr., **Common LISP, The Language**, Digital Press, 1984.

[Strom & Yemini, 1983]

R.E. Strom and S. Yemini, *NIL: An Integrated Language and System for Distributed Programming*, **Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems**, appearing in **SIGPLAN Notices**, Vol. 18, No. 6, June 1983, pp. 73–82.

[Teitelman, 1978]

W. Teitalman, **The Interlisp Reference Manual**, Xerox PARC, Palo Alto, California, October 1978.

[Tennent, 1982]

R.D. Tennent, *Two Examples of Block Structuring*, **Software–Practice and Experience**, Vol. 12, No. 4, April 1982, pp. 385–392.

[Thomas, 1976]

J.W. Thomas, *Module Interconnection in Programming Systems Supporting Abstraction*, **Technical Report CS–16**, Computer Science Program, Brown University, April 1976.

[Tichy, 1979]

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.

[Tichy & Baker, 1985]

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.

[Ullman, 1980]

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

[Welsh, Sneeringer, & Hoare, 1977]

J. Welsh, W.J. Sneeringer and C.A.R. Hoare, *Ambiguities and Insecurities in Pascal*, **Software–Practice and Experience**, Vol. 7, No. 6, November/December 1977, pp. 685–696.

[Wileden & Clarke, 1984]

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

[Wirth, 1983]

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

[Wolf, Clarke, & Wileden, 1983]

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.

[Wolf, Clarke, & Wileden, 1985a]

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.

[Wolf, Clarke, & Wileden, 1985b]

A.L. Wolf, L.A. Clarke, and J.C. Wileden, *Interface Control and Incremental Development in the PIC Environment*, **Proceedings of the Eighth International Conference on Software Engineering**, London, England, August 1985.

[Wulf, 1980]

W.A. Wulf, *Trends in the Design and Implementation of Programming Languages*, **Computer**, Vol. 13, No. 1, January 1980, pp. 14–25.

[Wulf & Shaw, 1973]

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

[Young, 1981]

S.J. Young, *Improving the Structure of Large Pascal Programs*, **Software–Practice and Experience**, Vol. 11, No. 9, September 1981, pp. 913–927.

APPENDIX A

PIC/ADL SYNTAX SUMMARY

This appendix presents the syntax of the Ada-like design language PIC/ADL. The notation used to describe the syntax is the same modified Backus-Naur Form used in [DoD, 1983]:

- Lower case words, some containing embedded underlines, are used to denote syntactic categories.
- Boldface words are used to denote reserved words.
- Square brackets enclose optional items.
- Braces enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule.
- A vertical bar separates alternative items unless it occurs immediately after an opening brace, in which case it stands for itself.
- If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part; the italicized part is intended to convey some semantic information.

Except for one additional compound delimiter and several additional reserved words, the lexical rules of PIC/ADL follow those of Ada (see Chapter 2 of [DoD, 1983]). The additional compound delimiter is the ellipsis ("..."), which denotes incompleteness. The additional reserved words are listed below.

by	**each**	**none**	**only**	**provide**
request	**stub**	**to**	**used**	

Certain reserved words of Ada are not used in PIC/ADL but are nevertheless reserved in PIC/ADL. Primarily, this is to "reserve" those words for future use when an Ada feature not currently supported in PIC/ADL, such as tasking, is incorporated into the language. The Ada reserved words **limited** and **separate** are superfluous; they are reserved simply to disallow any non-Ada identifiers from appearing in a PIC/ADL design that can conflict with an Ada implementation.

```
graphic_character ::=
   basic_graphic_character | lower_case_letter | other_special_character

basic_graphic_character ::=
      upper_case_letter | digit
   | special_character | space_character

basic_character ::= basic_graphic_character | format_effector

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

numeric_literal ::= decimal_literal | based_literal
```

decimal_literal ::= integer [.integer] [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | E - integer

based_literal ::= base # based_integer [.based_integer] # [exponent]

base ::= integer

based_integer ::= extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter

character_literal ::= 'graphic_character'

string_literal ::= "{graphic_character}"

pragma ::= **pragma** identifier [(argument_association {, argument_association})]

argument_association ::=
    [*argument_*identifier =>] name
  | [*argument_*identifier =>] expression

simple_declaration ::=
    object_declaration | number_declaration
  | exception_declaration | subtype_declaration
  | simple_type_declaration | ...

object_declaration ::=
    identifier_list : [**constant**] subtype_indication [:= expression]
  | identifier_list : [**constant**] constrained_array_definition [:= expression]

number_declaration ::=
  identifier_list : **constant** := *universal_static_*expression

identifier_list ::= identifier {, identifier}

simple_type_declaration ::=
  partial_type_declaration | full_simple_type_declaration

partial_type_declaration ::= **type** identifier [discriminant_part]

```
full_simple_type_declaration  ::=
    type identifier [discriminant_part] is simple_type_representation

simple_type_representation  ::=
        enumeration_type_definition   |  integer_type_definition
    |   real_type_definition          |  array_type_definition
    |   record_type_definition        |  access_type_defintion
    |   ...

type_representation  ::=  simple_type_representation  |  derived_type_definition

subtype_declaration  ::=  subtype identifier is subtype_indication

subtype_indication  ::=  type_mark [constraint]

type_mark  ::=  type_name  |  subtype_name  |  ...

constraint  ::=
        range_constraint          |  floating_point_constraint
    |   fixed_point_constraint    |  index_constraint
    |   discriminant_constraint

derived_type_declaration  ::=
    type identifier [discriminant_part] is derived_type_definition

derived_type_definition  ::=  new subtype_indication

range_constraint  ::=  range range  |  range ...

range  ::=
        range_attribute
    |   simple_expression .. simple_expression

enumeration_type_definition  ::=
    (enumeration_literal_specification {, enumeration_literal_specification})

enumeration_literal_specification  ::=  enumeration_literal

enumeration_literal  ::=  identifier  |  character_literal  |  ...

integer_type_definition  ::=  range_constraint

real_type_definition  ::=  floating_point_constraint  |  fixed_point_constraint
```

```
floating_point_constraint ::= floating_accuracy_defintion [range_constraint]

fixed_accuracy_defintion ::= delta static_simple_expression_or_ellipsis

fixed_point_constraint ::= fixed_accuracy_defintion [range_constraint]

fixed_accuracy_defintion ::= digits static_simple_expression_or_ellipsis

array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
    array (index_subtype_definition {, index_subtype_definition})
      of component_subtype_indication

constrained_array_definition ::=
    array index_constraint of component_subtype_indication

index_subtype_definition ::= type_mark range <>

index_constraint ::= (discrete_range {, discrete_range})

discrete_range ::= discrete_subtype_indication | range

record_type_definition ::=
    record
      component_list
    end record

component_list ::=
      component_declaration {component_declaration}
    | {component_declaration} variant_part
    | null;

component_declaration ::=
      identifier_list : component_subtype_definition [:= expression];
    | ...;

component_subtype_definition ::= subtype_indication

discriminant_part ::= (discriminant_specification {; discriminant_specification})

discriminant_specification ::= identifier_list : type_mark [:= expression] | ...
```

discriminant_constraint ::= (discriminant_association {, discriminant_association})

discriminant_association ::=
    [*discriminant*_simple_name {| *discriminant*_simple_name} =>] expression

```
variant_part ::=
    case discriminant_simple_name is
      variant {variant}
    end case;
```

variant ::= **when** choice {| choice} => component_list

```
choice ::=
        simple_expression         | discrete_range
    |   component_simple_name     | others
```

access_type_declaration ::= **access** subtype_indication

```
name ::=
        simple_name        | character_literal
    |   operator_symbol    | indexed_component
    |   slice              | selected_component
    |   attribute
```

simple_name ::= identifier

prefix ::= name | function_call

indexed_component ::= prefix(expression {, expression})

slice ::= prefix(discrete_range)

selected_component ::= prefix.selector

```
selector ::=
        simple_name        | character_literal
    |   operator_symbol    | all
```

attribute ::= prefix'attribute_designator

attribute_designator ::= simple_name [(*universal_static*_expression)]

aggregate ::= (component_association {, component_association})

```
component_association ::= [choice {| choice} =>] expression

expression ::=
        relation {and relation}    | relation {and then relation}
      | relation {or relation}     | relation {or else relation}
      | relation {xor relation}

relation ::=
        simple_expression_or_ellipsis [relational_operator simple_expression_or_ellipsis]
      | simple_expression_or_ellipsis [not] in range
      | simple_expression_or_ellipsis [not] in type_mark

simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}

simple_expression_or_ellipsis ::= simple_expression | ...

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=
        numeric_literal | null            | aggregate
      | string_literal  | name            | allocator
      | function_call   | type_conversion | qualified_expression
      | (expression)

logical_operator ::= and | or | xor

relational_operator ::= = | /= | < | <= | > | >=

binary_adding_operator ::= + | - | &

unary_adding_operator ::= + | -

multiplying_operator ::= * | / | mod | rem

highest_precedence_operator ::= ** | abs | not

type_conversion ::= type_mark(expression)

qualified_expression ::= type_mark'(expression) | type_mark'aggregate

allocator ::= new subtype_indication | new qualified_expression
```

```
sequence_of_statements ::= statement {statement}

statement ::= {label} simple_statement | {label} compound_statement

simple_statement ::=
       null_statement              | assignment_statement
     | procedure_call_statement    | exit_statement
     | return_statement            | goto_statement
     | raise_statement             | ...

compound_statement ::=
       if_statement     | case_statement
     | loop_statement   | block_statement

label ::= <<label_simple_name>>

null_statement ::= null;

assignment_statement ::= variable_name := expression;

if_statement ::=
      if condition then
         sequence_of_statements
    { elsif condition then
         sequence_of_statements }
    [ else
         sequence_of_statements ]
      end if;

condition ::= boolean_expression

case_statement ::=
    case expression is
       case_statement_alternative {case_statement_alternative}
    end case;

case_statement_alternative ::=
    when choice {| choice} => sequence_of_statements

loop_statement ::=
    [loop_simple_name:]
       [iteration_scheme] loop
          sequence_of_statements
       end loop [loop_simple_name];
```

```
iteration_scheme ::= while condition | for loop_parameter_specification

loop_parameter_specification ::=
        identifier in [reverse] discrete_range
    |   each identifier in name

block_statement ::=
    [block_simple_name:]
        begin
          sequence_of_statements
      [ exception
          exception_handler {exception_handler} ]
        end [block_simple_name];

exit_statement ::= exit [loop_name] [when condition];

return_statement ::= return [expression];

goto_statement ::= goto label_name;


exception_declaration ::= identifier_list : exception

exception_handler ::=
    when exception_choice {| exception_choice} =>
      sequence_of_statement

exception_choice ::= exception_name | others

raise_statement ::= raise [exception_name];


subprogram_specification ::= subprogram_heading [access_clauses];

subprogram_heading ::=
        procedure identifier [formal_part]
    |   function designator [formal_part] return type_mark

subprogram_specification_stub ::=
    subprogram_stub_heading used_by_clause [access_clauses];
```

```
subprogram_stub_heading ::=
       procedure stub identifier [formal_part]
    |  function stub designator [formal_part] return type_mark

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal

formal_part ::= (parameter_specification {; parameter_specification})

parameter_specification ::= identifier_list : mode type_mark [:= expression] | ...

mode ::= [in] | in out | out

subprogram_body ::=
      subprogram_heading is
        [ use_clause; ]
        { simple_declaration; }
      begin
          sequence_of_statements
    [ exception
          exception_handler {exception_handler} ]
      end [designator];

procedure_call_statement ::= procedure_name [actual_parameter_part];

function_call ::= function_name [actual_parameter_part];

actual_parameter_part ::= (parameter_association {, parameter_association})

parameter_association ::= [formal_parameter =>] actual_parameter

formal_parameter ::= parameter_simple_name | ...

actual_parameter ::= expression | variable_name | type_mark(variable_name)

used_by_clause ::= used by used_by_item {, used_by_item}

used_by_item ::= identifier | ...
```

```
package_specification  ::=
      package identifier is
        [ common_access_clauses [use_clause]; ]
        { provided_declarative_item }
    [ private
        { hidden_declarative_item } ]
      end [package_simple_name];

package_specification_stub  ::=
      package stub identifier is
         used_by_clause
        [ common_access_clauses [use_clause]; ]
        { provided_declarative_item }
    [ private
        { hidden_declarative_item } ]
      end [package_simple_name];

provided_declarative_item  ::=
    |  subprogram_specification              |  number_declaration [provide_clause];
    |  object_declaration [access_clauses];  |  exception_declaration [provide_clause];
    |  provided_subtype_declaration;         |  provided_type_declaration;
    |  deferred_constant_declaration [provide_clause];
    |  ... [access_clauses];

provided_subtype_declaration  ::=
    subtype identifier [provide_clause] is subtype_indication [request_clause]

provided_type_declaration  ::=
    partial_type_declaration [access_clauses]   |  provided_full_type_declaration

provided_full_type_declaration  ::=
      type identifier [discriminant_part] [common_operation_clause] [type_provide_clause]
          is type_representation
               [common_operation_clause] [type_provide_clause] [request_clause]
    | type identifier [discriminant_part] [common_operation_clause] [type_provide_clause]
          is private

deferred_constant_declaration  ::=  identifier_list : constant type_mark
```

```
hidden_declarative_item ::=
        subprogram_heading [request_clause];
    |   simple_declaration [request_clause];
    |   derived_type_declaration [request_clause];

package_body ::=
      package body package_simple_name is
        [ use_clause; ]
        { package_body_declarative_item }
    [ begin
          sequence_of_statements ]
    [ exception
          exception_handler {exception_handler} ]
      end [package_simple_name];

package_body_declarative_item ::= {basic_declarative_item} {later_declarative_item}

basic_declarative_item ::= simple_declaration; | subprogram_heading;

later_declarative_item ::= subprogram_heading; | subprogram_body | ...


provide_clause ::= provide [only] to provide_item {, provide_item}

request_clause ::= request request_item {, request_item}

access_clauses ::= request_clause [provide_clause] | provide_clause [request_clause]

use_clause ::= use request_item {, request_item}

provide_item ::=
        identifier        | package_identifier... | ...
    |   ...simple_item    | package_identifier.simple_item
    |   ...(simple_item_or_ellipsis {, simple_item_or_ellipsis})
    |   package_identifier.(simple_item_or_ellipsis {, simple_item_or_ellipsis})

simple_item_or_ellipsis ::= simple_item | ...

simple_item ::= operator_symbol | subprogram_signature | identifier
```

```
subprogram_signature ::=
      designator [formal_part] return type_mark
   |  identifier formal_part

request_item ::=
      identifier                 |  package_identifier... [private] [operation_clause]  |  ...
   |  ...request_simple_item     |  package_identifier.request_simple_item
   |  ...(request_simple_item_or_ellipsis {, request_simple_item_or_ellipsis})
   |  package_identifier.(request_simple_item_or_ellipsis
                                 {, request_simple_item_or_ellipsis})

request_simple_item_or_ellipsis ::=
    request_simple_item  |  ... [private] [operation_clause]

request_simple_item ::=
    operator_symbol  |  subprogram_signature  |  identifier [private] [operation_clause]

type_provide_clause ::=  provide [only] to type_provide_item {, type_provide_item}

type_provide_item ::=
      identifier [operation_clause]  |  package_identifier... [operation_clause]
   |  ... [operation_clause]         |  ...simple_item [operation_clause]
   |  package_identifier.simple_item [operation_clause]
   |  ...(simple_item_or_ellipsis [operation_clause]
                                 {, simple_item_or_ellipsis [operation_clause]})
   |  package_identifier.(simple_item_or_ellipsis [operation_clause]
                                 {, simple_item_or_ellipsis [operation_clause]})

operation_clause ::=  with <[simple_item_or_ellipsis {, simple_item_or_ellipsis}]>


submission ::=  {submodule}

submodule ::=
      package_specification        |  subprogram_specification
   |  package_specification_stub   |  subprogram_specification_stub
   |  package_body                 |  subprogram_body
```

APPENDIX B

REQUISITION/PROVISION PROBLEM-DETECTION ALGORITHMS FOR BASIC INTERFACE ANALYSES

This appendix gives algorithms for finding the nine problems that are detected by the basic interface analyses presented in Chapter VI. The problems are summarized in Table 1 on page 135, where a particular error/anomaly classification is given for them. Such a classification is independent of the algorithms themselves; the classification is significant only for interpreting the results. The algorithms are described in the design language PIC/ADL outlined in Chapter III.

§1. Overview

The algorithms are modularised into three packages. The first package, BasicInterfaceProblems, contains nine function subprograms that correspond to the nine entries of Table 1. The parameters to these functions are determined by the entities and submodules available to the basic interface analyses that detect the problems (see Table 2 on page 137). The result returned by each function is a three valued flag: the value ConditionallyConsistent indiciates that the problem does not exist but incompleteness is involved; the value Consistent indicates

that the problem does not exist and no incompleteness is involved; and the value Inconsistent indicates that the problem does exist. The second package, InternalRepUtilities, contains type definitions and function subprograms for interrogating the internal representation(s) of the submodule(s) involved in an analysis. Table 4 summarizes the roles played by the functions in that package. Only the bodies of functions Interaction and IsTypeRepProvidedTo are given below, since these are the functions in InternalRepUtilities most directly related to requisition and provision. The third package, AccessClauseUtilities, contains type definitions used to internally represent request and provide clauses, and a function subprogram, LookupInClause, to interrogate that representation.

Each of the three packages resides at a different conceptual level, with BasicInterfaceProblems at the highest level shown and AccessClauseUtilities at the lowest. InternalRepUtilities is at an intermediate level, serving to shield details of the internal representation from the nine problem-detection functions. It is assumed that any processing needed to identify the entities and modules involved in an analysis (e.g., interaction with the user and locating "pointers" to internal representations held in some library structure) is performed at the level(s) above BasicInterfaceProblems. Moreover, any interpretation of the results of an analysis, particuarly classification of detected problems as errors or as anomalies, is also assumed to occur above BasicInterfaceProblems.

The data structure used to internally represent request clauses is the same data structure used to internally represent provide clauses. That data structure is a

FUNCTION	ROLE PLAYED BY FUNCTION
AnyReferenceTo	determines if any reference to a given entity occurs within a given submodule
FindDeclaration	determines if a given entity is declared in a given submodule, and if so, retrieves a "pointer" to the declaration; if no declaration exists, but the declaration list is incomplete, retrieves the collection of "pointers" to ellipses in the list
GetClause	retrieves the "pointer" to a request or provide clause attached to a given entity
GetComponentTypes	retrieves the collection of "pointers" to the component types of a given object or type
GetDiscrimTypes	retrieves the collection of "pointers" to the discriminant types of a given type
GetName	retrieves a given entity's name
GetRep	retrieves the "pointer" to a given object's or type's representation
GetSignatureTypes	retrieves the collection of "pointers" to a given subprogram's parameter/return types
Interaction	determines either (1) if a given entity of a module A is provided to a given entity of a module B, or (2) if a given entity of a module B is requested by a given entity of a module A
IsBodyDefined	determines if a given subprogram's body is defined in a given body submodule
IsReferredTo	determines if a reference to a given entity is made by a given entity
IsSubprogram	determines if a given entity is a subprogram
IsType	determines if a given entity is a type
IsTypeRepProvidedTo	determines if a given type representation is provided to a given entity
SameParent	determines if two given entities are declared in the same submodule

Table 4: Dictionary of Functions in Package InternalRepUtilities.

request A, B.(W, X, ...), B.Y, ...(Z, X), ...

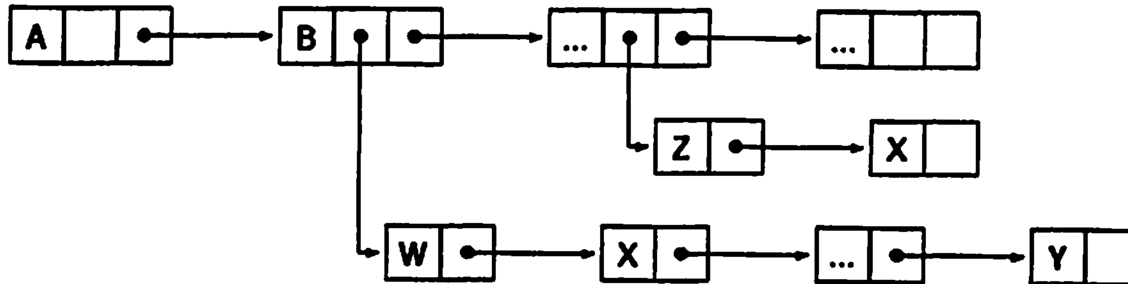


Figure 37: A PIC/ADL Request Clause and its Internal Representation.

rather straightforward (linked) list of (linked) lists.[1] Figure 37 shows a PIC/ADL request clause and depicts that clause's internal representation. The main list is a collection of module names. Rooted at each element in this list is a subsidiary list of entities requested from the main list element's associated module. Special module- and entity-list elements are used to represent incompleteness items, which are denoted by ellipses. Any redundancies in a source clause are collapsed in the internal representation of that clause (e.g., there is only one main list element for module B in Figure 37). Function LookupInClause of package AccessClauseUtilities cycles through the representation of a request or provide clause, seeking to find a match for a given module and/or entity. (*Note:* The items in common request and provide clauses of PIC/ADL are internally represented by melding them into the internal representations of clauses associated with individual entities.)

[1] The data structure discussed here is used primarily for expository purposes. Experience has not yet shown, however, that a different data structure, such as a hash table, would be more appropriate for implementation.

§2. The Algorithms

§2.1 *Package* BasicInterfaceProblems

```
package BasicInterfaceProblems is
    request ...(InternalRep, Name), ...SearchResult
    use     ...(InternalRep, Name);

  type ConsistencyStatus is (Consistent, ConditionallyConsistent, Inconsistent);

  function LookForA (RequestingSubmodule : InternalRep;
                     RequestingEntity    : Name;
                     RequestedSubmodule  : InternalRep;
                     RequestedEntity     : Name) return ConsistencyStatus
    request InternalRepUtilities.(GetName, Interaction, InteractionKinds);
      -- Determines if RequestedEntity, declared in RequestedSubmodule,
      -- is provided to RequestingEntity, declared in RequestingSubmodule

  function LookForB (ReferringSubmodule : InternalRep;
                     ReferringEntity    : Name;
                     ReferredSubmodule  : InternalRep;
                     ReferredEntity     : Name) return ConsistencyStatus
    request InternalRepUtilities.(GetName, Interaction, InteractionKinds);
      -- Determines if ReferredEntity, declared in ReferredSubmodule,
      -- is provided to ReferringEntity, declared in ReferringSubmodule

  function LookForC (RequestingSubmodule : InternalRep;
                     ReferringEntity     : Name;
                     ReferredModule      : Name;
                     ReferredEntity      : Name) return ConsistencyStatus
    request InternalRepUtilities.(Interaction, InteractionKinds);
      -- Determines if ReferredEntity of ReferredModule
      -- is requested by ReferringEntity in RequestingSubmodule

  function LookForD (ProvidedSubprogram : Name;
                     BodySubmodule      : InternalRep) return ConsistencyStatus
    request InternalRepUtilities.IsBodyDefined;
      -- Determines if body of ProvidedSubprogram is declared in BodySubmodule

  function LookForE (ProvidingSubmodule : InternalRep;
                     ProvidedEntity     : InternalRep;
                     ToModule           : Name;
                     ToEntity           : Name) return ConsistencyStatus
    request ...(CollectionOfInternalRep, ConcatCollection),
            InternalRepUtilities.(GetComponentTypes, GetDiscrimTypes, GetName,
```

```
                                        GetRep, GetSignatureTypes,
                                        Interaction, IsSubprogram, IsType,
                                        IsTypeRepProvidedTo, SameParent);
      -- determines if the types associated and declared with ProvidedEntity in
      -- ProvidingSubmodule, are provided to entity ToEntity of module ToModule;
      -- result returned will be InConsistent if any type is not provided,
      -- ConditionallyConsistent if any incompleteness, or Consistent if all types
      -- are provided and there is no incompleteness

  function LookForF (ReferredSubprogram : Name;
                     BodySubmodule      : InternalRep) return ConsistencyStatus
    request InternalRepUtilities.IsBodyDefined;
      -- Determines if body of ReferredSubprogram is declared in BodySubmodule

  function LookForG (RequestingSubmodule : InternalRep;
                     RequestingEntity    : Name;
                     ProvidedSubmodule   : InternalRep;
                     ProvidedEntity      : Name) return ConsistencyStatus
    request InternalRepUtilities.(GetName, Interaction, InteractionKinds);
      -- Determines if ProvidedEntity, declared in ProvidedSubmodule,
      -- is requested by RequestingEntity in RequestingSubmodule

  function LookForH (RequestedModule    : Name;
                     RequestedEntity    : Name;
                     ReferringSubmodule : InternalRep;
                     ReferringEntity    : Name) return ConsistencyStatus
    request InternalRepUtilities.IsReferredTo;
      -- Determines if RequestedEntity of RequestedModule
      -- is referred to by ReferringEntity in ReferringSubmodule

  function LookForI (SpecOrSpecStubSubmodule : InternalRep;
                     BodySubmodule           : InternalRep;
                     DefinedEntity           : Name) return ConsistencyStatus
    request ...CollectionOfInternalRep,
            InternalRepUtilities.(AnyReferenceTo, DeclarationSearchResult,
                                  FindDeclaration, VisibilityKinds);
      -- Determines if DefinedEntity is provided by SpecOrSpecStubSubmodule
      -- or referred to in either SpecOrSpecStubSubmodule or BodySubmodule

end BasicInterfaceProblems;


package body BasicInterfaceProblems is
  use ...(CollectionOfInternalRep, ConcatCollection, InternalRep, Name), ...SearchResult,
      InternalRepUtilities.(AnyReferenceTo, FindDeclaration, GetClause,
                            GetComponentTypes, GetDiscrimTypes, GetName,
```

```
                        GetRep, GetSignatureTypes,
                        Interaction, InteractionKinds, IsBodyDefined,
                        IsReferredTo, IsSubprogram, IsType,
                        IsTypeRepProvidedTo, SameParent, VisibilityKinds);

function LookForA (RequestingSubmodule : InternalRep;
                   RequestingEntity    : Name;
                   RequestedSubmodule  : InternalRep;
                   RequestedEntity     : Name) return ConsistencyStatus is
   -- Determines if RequestedEntity, declared in RequestedSubmodule,
   -- is provided to RequestingEntity, declared in RequestingSubmodule

begin
  -- Check provision relationship between RequestedEntity and RequestingEntity
  case Interaction(Provision, RequestedSubmodule, RequestedEntity,
                   GetName(RequestingSubmodule), RequestingEntity) is
    when FoundComplete =>
      -- RequestedEntity is explicitly provided to RequestingEntity
      return Consistent;
    when FoundIncomplete =>
      -- Possibility that RequestedEntity is provided to RequestingEntity
      -- because incompleteness involved
      return ConditionallyConsistent;
    when NotFound =>
      -- No possibility that RequestedEntity is provided to RequestingEntity,
      -- so problem detected
      return Inconsistent;
  end case;
end LookForA;


function LookForB (ReferringSubmodule : InternalRep;
                   ReferringEntity    : Name;
                   ReferredSubmodule  : InternalRep;
                   ReferredEntity     : Name) return ConsistencyStatus is
   -- Determines if ReferredEntity, declared in ReferredSubmodule,
   -- is provided to ReferringEntity, declared in ReferringSubmodule

begin
  -- Check provision relationship between ReferredEntity and ReferringEntity
  case Interaction(Provision, ReferredSubmodule, ReferredEntity,
                   GetName(ReferringSubmodule), ReferringEntity) is
    when FoundComplete =>
      -- ReferredEntity is explicitly provided to ReferringEntity
      return Consistent;
```

```
      when FoundIncomplete =>
        -- Possibility that ReferredEntity is provided to ReferringEntity
        -- because incompleteness involved
        return ConditionallyConsistent;
      when NotFound =>
        -- No possibility that ReferredEntity is provided to ReferringEntity,
        -- so problem detected
        return Inconsistent;
    end case;
  end LookForB;

  function LookForC (RequestingSubmodule : InternalRep;
                     ReferringEntity     : Name;
                     ReferredModule      : Name;
                     ReferredEntity      : Name) return ConsistencyStatus is
      -- Determines if ReferredEntity of ReferredModule
      -- is requested by ReferringEntity in RequestingSubmodule

  begin
    -- Check requisition relationship between ReferredEntity and ReferringEntity
    case Interaction(Requisition, RequestingSubmodule, ReferringEntity,
                     ReferredModule, ReferredEntity) is
      when FoundComplete =>
        -- ReferredEntity is explicitly requested by ReferringEntity
        return Consistent;
      when FoundIncomplete =>
        -- Possibility that ReferredEntity is requested by ReferringEntity
        -- because incompleteness involved
        return ConditionallyConsistent;
      when NotFound =>
        -- No possibility that ReferredEntity is requested by ReferringEntity,
        -- so problem detected
        return Inconsistent;
    end case;
  end LookForC;

  function LookForD (ProvidedSubprogram : Name;
                     BodySubmodule      : InternalRep) return ConsistencyStatus is
      -- Determines if body of ProvidedSubprogram is declared in BodySubmodule

  begin
    -- Check for declaration of ProvidedSubprogram in BodySubmodule
    case IsBodyDefined(ProvidedSubprogram, BodySubmodule) is
      when FoundComplete =>
```

```
        -- Declaration found
        return Consistent;
      when FoundIncomplete =>
        -- No declaration found, but declaration list is incomplete
        return ConditionallyConsistent;
      when NotFound =>
        -- No declaration found and declaration list is complete
        return Inconsistent;
    end case;
  end LookForD;


  function LookForE (ProvidingSubmodule : InternalRep;
                     ProvidedEntity     : InternalRep;
                     ToModule           : Name;
                     ToEntity           : Name) return ConsistencyStatus is
      -- determines if the types associated and declared with ProvidedEntity in
      -- ProvidingSubmodule, are provided to entity ToEntity of module ToModule;
      -- result returned will be InConsistent if any type is not provided,
      -- ConditionallyConsistent if any incompleteness, or Consistent if all types
      -- are provided and there is no incompleteness

    ReturnStatus : ConsistencyStatus := Consistent;
      -- Flag indicating lowest level of confidence in consistency
      -- that was detected during examination
    TypeCollection : CollectionOfInternalRep;
      -- Collection of "pointers" to types associated with ProvidedEntity
    TypeRep : InternalRep;
      -- Temporary to hold "pointer" to representation of a type

  begin
    -- Determine what kind of entity ProvidedEntity is
    -- and collect appropriate "pointers" to associated types
    if IsSubprogram(ProvidedEntity) then
      -- ProvidedEntity is a subprogram, so collect each of its signature types,
      -- which include the types of its parameters and return type (if any)
      TypeCollection := GetSignatureTypes(ProvidedEntity);
    elsif IsType(ProvidedEntity) then
      -- ProvidedEntity is a type, so collect types of discriminants (if any)
      -- and component types of representation if that representation is also
      -- provided to ToEntity
      TypeRep := GetRep(ProvidedEntity);
      if IsTypeRepProvidedTo(TypeRep, ToModule, ToEntity) /= NotFound then
        -- Representation (possibly) provided, so collect each component type
```

```
      -- in addition to any discriminant types
      TypeCollection := ConcatCollection(GetDiscrimTypes(TypeRep),
                                         GetComponentTypes(TypeRep));
    else
      -- Representation not provided, so just collect any discriminant types
      TypeCollection := GetDiscrimTypes(TypeRep);
    end if;
  else
    -- ProvidedEntity is an object, so collect each component type
    TypeCollection := GetComponentTypes(GetRep(ProvidedEntity));
  end if;
  -- Examine each element in collection
  for each Element in TypeCollection loop
    -- Only a type declared in the same package as ProvidedEntity is relevant
    if SameParent(Element, ProvidedEntity) then
      -- Check provision relationship between Element and ToEntity
      case Interaction(Provision, ProvidingSubmodule, GetName(Element),
                       ToModule, ToEntity) is
        when FoundComplete =>
          -- Element is explicitly provided to ToEntity;
          -- continue to next element
          null;
        when FoundIncomplete =>
          -- Possibility that Element is provided to ToEntity
          -- because incompleteness involved; set ReturnStatus
          -- to reflect incompleteness and continue to next element
          ReturnStatus := ConditionallyConsistent;
        when NotFound =>
          -- No possibility that Element is provided to ToEntity,
          -- so problem detected
          return Inconsistent;
      end case;
    end if;
  end loop;
  -- Fell through loop, so no problem detected; return lowest level
  -- of confidence in consistency that was detected during examination
  return ReturnStatus;
end LookForE;
```

```
function LookForF (ReferredSubprogram : Name;
                   BodySubmodule      : InternalRep) return ConsistencyStatus is
    -- Determines if body of ReferredSubprogram is declared in BodySubmodule

begin
  -- Check for declaration of ReferredSubprogram in BodySubmodule
  case IsBodyDefined(ReferredSubprogram, BodySubmodule) is
    when FoundComplete =>
      -- Declaration found
      return Consistent;
    when FoundIncomplete =>
      -- No declaration found, but declaration list is incomplete
      return ConditionallyConsistent;
    when NotFound =>
      -- No declaration found and declaration list is complete
      return Inconsistent;
  end case;
end LookForF;



function LookForG (RequestingSubmodule : InternalRep;
                   RequestingEntity    : Name;
                   ProvidedSubmodule   : InternalRep;
                   ProvidedEntity      : Name) return ConsistencyStatus is
    -- Determines if ProvidedEntity, declared in ProvidedSubmodule,
    -- is requested by RequestingEntity in RequestingSubmodule

begin
  -- Check requisition relationship between ProvidedEntity and RequestingEntity
  case Interaction(Requisition, RequestingSubmodule, RequestingEntity,
                 GetName(ProvidedSubmodule), ProvidedEntity) is
    when FoundComplete =>
      -- ProvidedEntity is explicitly requested by RequestingEntity
      return Consistent;
    when FoundIncomplete =>
      -- Possibility that ProvidedEntity is requested by RequestingEntity
      -- because incompleteness involved
      return ConditionallyConsistent;
    when NotFound =>
      -- No possibility that ProvidedEntity is requested by RequestingEntity,
      -- so problem detected
      return Inconsistent;
  end case;
end LookForG;
```

```
function LookForH (RequestedModule    : Name;
                   RequestedEntity    : Name;
                   ReferringSubmodule : InternalRep;
                   ReferringEntity    : Name) return ConsistencyStatus is
    -- Determines if RequestedEntity of RequestedModule
    -- is referred to by ReferringEntity in ReferringSubmodule

begin
  -- Check reference relationship between RequestedEntity and ReferringEntity
  case IsReferredTo(RequestedModule, RequestedEntity,
                    ReferringSubmodule, ReferringEntity) is
    when FoundComplete =>
      -- RequestedEntity is explicitly referred to by ReferringEntity
      return Consistent;
    when FoundIncomplete =>
      -- Possibility that RequestedEntity is referred to by ReferringEntity
      -- because incompleteness involved
      return ConditionallyConsistent;
    when NotFound =>
      -- No possibility that RequestedEntity is referred to by ReferringEntity,
      -- so problem detected
      return Inconsistent;
  end case;
end LookForH;


function LookForI (SpecOrSpecStubSubmodule : InternalRep;
                   BodySubmodule           : InternalRep;
                   DefinedEntity           : Name) return ConsistencyStatus is
    -- Determines if DefinedEntity is provided by SpecOrSpecStubSubmodule
    -- or referred to in either SpecOrSpecStubSubmodule or BodySubmodule

  BodyReferSearchResult : SearchResult;
    -- Temporary to hold result of search for reference in BodySubmodule
  ProvideSearchResult : SearchResult;
    -- Temporary to hold result of search for provided declaration
    -- in SpecOrSpecStubSubmodule
  SpecReferSearchResult : SearchResult;
    -- Temporary to hold result of search for reference in SpecOrSpecStubSubmodule

begin
  -- Check whether declaration of DefinedEntity is provided (i.e., "visible")
  ProvideSearchResult
    := FindDeclaration(SpecOrSpecStubSubmodule, DefinedEntity, Visible).Result;
  -- Check whether there is a reference to DefinedEntity in specification or
```

```
    -- specification stub submodule
    SpecReferSearchResult
      := AnyReferenceTo(SpecOrSpecStubSubmodule, DefinedEntity);
    -- Check whether there is a reference to DefinedEntity in body submodule
    BodyReferSearchResult
      := AnyReferenceTo(BodySubmodule, DefinedEntity);
    if (     ProvideSearchResult    = NotFound
        and SpecReferSearchResult  = NotFound
        and BodyReferSearchResult  = NotFound) then
      -- DefinedEntity is not provided nor referred to, so problem detected
      return Inconsistent;
    elsif (  ProvideSearchResult     = FoundIncomplete
          or SpecReferSearchResult  = FoundIncomplete
          or BodyReferSearchResult  = FoundIncomplete) then
      -- Possibility that DefinedEntity is either provided or referred to
      -- because incompleteness involved
      return ConditionallyConsistent;
    else
      -- DefinedEntity is explicitly provided or referred to
      return Consistent;
    end if;
  end LookForI;

end BasicInterfaceProblems;
```

§2.2 *Package* InternalRepUtilities

```
package InternalRepUtilities is
    request ...(CollectionOfInternalRep, InternalRep, Name), ...SearchResult,
            ...
    use     ...(CollectionOfInternalRep, InternalRep, Name), ...SearchResult;

  type ClauseKinds is (ProvideClause, RequestClause);
  type DeclarationSearchResult (Result : SearchResult) is
    record
      case Result is
        when FoundComplete =>
          -- There can be only one actual declaration
          EntityRep : InternalRep;
        when FoundIncomplete =>
          -- Declaration list may have several incompleteness entries
          EllipsesList : CollectionOfInternalRep;
        when NotFound =>
          -- No declaration and declaration list complete
          null;
      end case;
    end record;
  type InteractionKinds is (Requisition, Provision);
  type VisibilityKinds is (Visible, Hidden, Either);

  ...;

  function AnyReferenceTo (Submodule : InternalRep;
                           Entity    : Name) return SearchResult;
      -- Determines if any reference to Entity occurs within Submodule

  function FindDeclaration (Submodule      : InternalRep;
                            Entity         : Name;
                            VisibilityKind : VisibilityKinds := Either)
                                             return DeclarationSearchResult;
      -- Determines if Entity is declared in Submodule and if so
      -- retrieves a "pointer" to the declaration; if no declaration exists,
      -- but the declaration list is incomplete, retrieves the collection of
      -- "pointers" to ellipses in the list

  function GetClause (Entity     : InternalRep;
                      ClauseKind : ClauseKinds) return AccessClauseUtilities.Clause
    request AccessClauseUtilities;
      -- Retrieves the "pointer" to a request or provide clause attached to Entity
```

function GetComponentTypes (ObjectOrTypeRep : InternalRep)
return CollectionOfInternalRep;
-- *Retrieves the collection of "pointers" to the component types*
-- *of* ObjectOrTypeRep

function GetDiscrimTypes (TheType : InternalRep) **return** CollectionOfInternalRep;
-- *Retrieves the collection of "pointers" to the discriminant types of* TheType

function GetName (Entity : InternalRep) **return** Name;
-- *Retrieves the name of* Entity

function GetRep (ObjectOrType : InternalRep) **return** InternalRep;
-- *Retrieves the "pointer" to the representation of* ObjectOrType

function GetSignatureTypes (Subprogram : InternalRep)
return CollectionOfInternalRep;
-- *retrieves the collection of "pointers" the parameter*
-- *and return types of* Subprogram

function Interaction (InteractionKind : InteractionKinds;
SubmoduleOfA : InternalRep;
EntityOfA : Name;
ModuleB : Name;
EntityOfB : Name) **return** SearchResult
request AccessClauseUtilities;
-- *For* InteractionKind = Requisition, *determines if* EntityOfB *of* ModuleB
-- *is requested by* EntityOfA, *whose declaration is in* SubmoduleOfA;
-- *for* InteractionKind = Provision, *determines if* EntityOfA, *whose declaration*
-- *is in* SubmoduleOfA, *is provided to* EntityOfB *of* ModuleB;
-- *if* SubmoduleOfA *is a submodule of a (non-packaged) subprogram,*
-- *then* EntityOfA *is ignored*

function IsBodyDefined (Subprogram : Name;
BodySubmodule : InternalRep) **return** SearchResult;
-- *Determines if* Subprogram *is defined in* BodySubmodule

function IsReferredTo (ReferredModule : Name;
ReferredEntity : Name;
BySubmodule : InternalRep;
ByEntity : Name) **return** SearchResult;

-- *Determines if a reference to* RefferedEntity *of* ReferredModule *is made by*
-- ByEntity, *whose declaration appears in* BySubmodule

function IsSubprogram (Entity : InternalRep) **return** Boolean;
-- *Determines if* Entity *is a subprogram*

```
function IsType (Entity : InternalRep) return Boolean;
    -- Determines if Entity is a type

function IsTypeRepProvidedTo (TypeRep : InternalRep;
                              Module  : Name;
                              Entity  : Name) return SearchResult
  request AccessClauseUtilities;
    -- Determines if type representation TypeRep is provided to
    -- entity Entity of module Module

function SameParent (Entity1 : InternalRep;
                     Entity2 : InternalRep) return Boolean;
    -- Determines if Entity1 and Entity2 are declared in the same submodule

...;

end InternalRepUtilities;


package body InternalRepUtilities is
  use ...(CollectionOfInternalRep, InternalRep, Name), ...SearchResult;

...;

function Interaction (InteractionKind : InteractionKinds;
                      SubmoduleOfA    : InternalRep;
                      EntityOfA       : Name;
                      ModuleB         : Name;
                      EntityOfB       : Name) return SearchResult is
    -- For InteractionKind = Requisition, determines if EntityOfB of ModuleB
    -- is requested by EntityOfA, whose declaration is in SubmoduleOfA;
    -- for InteractionKind = Provision, determines if EntityOfA, whose declaration
    -- is in SubmoduleOfA, is provided to EntityOfB of ModuleB;
    -- if SubmoduleOfA is a submodule of a (non-packaged) subprogram,
    -- then EntityOfA is ignored

  use AccessClauseUtilities.(LookupInClause);

  ClauseKind : ClauseKinds;
    -- Flag indicating kind of clause implied by kind of interaction being examined
  DeclarationSearch : DeclarationSearchResult;
    -- Temporary to hold result of search for declaration of EntityOfA

begin
  -- Establish flag for calls to GetClause
  case InteractionKind is
    when Requisition =>
```

```
        ClauseKind := RequestClause;
      when Provision =>
        ClauseKind := ProvideClause;
    end case;
    if IsSubprogram(SubmoduleOfA) then
      -- SubmoduleOfA is simply a non-packaged subprogram,
      -- so just examine its clause
      return LookupInClause(GetClause(SubmoduleOfA, ClauseKind),
                            ModuleB, EntityOfB);
    else
      -- Find the declaration of EntityOfA in SubmoduleOfA
      DeclarationSearch := FindDeclaration(SubmoduleOfA, EntityOfA);
      case DeclarationSearch.Result is
        when FoundComplete =>
          -- Declaration found; examine associated clause
          return LookupInClause(GetClause(DeclarationSearch.EntityRep, ClauseKind),
                                ModuleB, EntityOfB);
        when FoundIncomplete =>
          -- No declaration found, but declaration list is incomplete;
          -- examine clause associated with each incompleteness item
          for each Element in DeclarationSearch.EllipsesList loop
            case LookupInClause(GetClause(Element, ClauseKind),
                                ModuleB, EntityOfB) is
              when FoundComplete | FoundIncomplete =>
                -- EntityOfB or incompleteness found, but attached to
                -- an incompleteness item
                return FoundIncomplete;
              when NotFound =>
                -- EntityOfB or incompleteness not yet found;
                -- continue to next element
                null;
            end case;
          end loop;
          -- Fell through loop without finding EntityOfB or incompleteness
          return NotFound;
        when NotFound =>
          -- No declaration found and declaration list complete
          return NotFound;
      end case;
    end if;
end Interaction;

...;
```

```
function IsTypeRepProvidedTo (TypeRep : InternalRep;
                              Module  : Name;
                              Entity  : Name) return SearchResult is
    -- Determines if type representation TypeRep is provided to
    -- entity Entity of module Module

  use AccessClauseUtilities.(LookupInClause);

begin
  -- Examine the clause associated with TypeRep
  return LookupInClause(GetClause(TypeRep, ProvideClause), ModuleA, EntityOfA);
end IsTypeRepProvidedTo;

...;

end InternalRepUtilities;
```

§2.3 *Package* AccessClauseUtilities

```
package AccessClauseUtilities is
    request    ...Name, ...SearchResult
    provide to InternalRepUtilities.(GetClause, Interaction, IsTypeRepProvidedTo), ...
    use        ...Name, ...SearchResult;

  type Clause is private;

  function LookupInClause (TheClause           : Clause;
                           Module              : Name;
                           Entity              : Name;
                           LookForModuleOnly : Boolean := False) return SearchResult;
      -- Determines if entity Entity of module Module is mentioned in
      -- request or provide clause TheClause; if LookForModuleOnly = True,
      -- then only checks for mention of Module

  ...;

private
  type CompletenessStatus is (Complete, Incomplete);
  type ModuleListElement (Completeness : CompletenessStatus);
  type EntityListElement (Completeness : CompletenessStatus);

  type ModuleListPointer is access ModuleListElement;
  type EntityListPointer is access EntityListElement;

  type Clause is access ModuleListElement;

  type ModuleListElement (Completeness : CompletenessStatus) is
    record
      EntityList    : EntityListPointer;
      NextElement : ModuleListPointer;
      case Completeness is
        when Complete =>
          ModuleName : Name;
          ...;
        when Incomplete =>
          null;
      end case;
    end record;

  type EntityListElement (Completeness : CompletenessStatus) is
    record
      NextElement : EntityListPointer;
      case Completeness is
```

```
        when Complete =>
          EntityName : Name;
          ...;
        when Incomplete =>
          null;
      end case;
    end record;

end AccessClauseUtilities;


package body AccessClauseUtilities is
    use ...Name, ...SearchStatus;

  function LookupInClause (TheClause            : Clause;
                           Module               : Name;
                           Entity               : Name;
                           LookForModuleOnly : Boolean := False)
                                                 return SearchResult is
      -- Determines if entity Entity of module Module is mentioned in
      -- request or provide clause TheClause; if LookForModuleOnly = True,
      -- then only checks for mention of Module

    EntityPointer : EntityListPointer;
      -- Temporary to hold pointer to entity list element
    FoundEntityIncompleteness : Boolean;
      -- Flag indicating incompleteness item in entity list found
    IncompletenessMatch : Boolean := False;
      -- Flag indicating candidate incompleteness match for entity and module found
    ModulePointer : ModuleListPointer := TheClause;
      -- Temporary to hold pointer to module list element

  begin
    -- Main loop traverses the module list
    while ModulePointer /= null loop
      if ModulePointer.Completeness = Complete then
        -- Not an incompleteness item
        if ModulePointer.ModuleName = Module then
          -- Found the module name
          if LookForModuleOnly then
            -- Examination completed
            return FoundComplete;
          end if;
          -- Examine the entity list
          EntityPointer := ModulePointer.EntityList;
          if EntityPointer = null then
```

```
            -- No entity list, so implicit mention of all entities
            return FoundComplete;
         end if;
         -- Traverse entity list to look for Entity
         FoundEntityIncompleteness := False;
         while EntityPointer /= null loop
            if EntityPointer.Completeness = Complete then
               -- Not an incompleteness item
               if EntityPointer.EntityName = Entity then
                  -- Found the entity name
                  return FoundComplete;
               end if;
               -- Continue to next element of entity list
            else
               -- Found incompleteness item; set flag to reflect incompleteness
               -- and continue to next element of entity list
               FoundEntityIncompleteness := True;
            end if;
            -- Move pointer to next entity list element
            EntityPointer := EntityPointer.NextElement;
         end loop;
         -- Fell through loop, so no exact match found;
         -- see if any incompleteness encountered
         if FoundEntityIncompleteness then
            -- Found candidate incompleteness match; set flag to reflect
            -- incompleteness and continue to next element of module list
            IncompletenessMatch := True;
         end if;
      end if;
   else
      -- Incompleteness item in module list
      if LookForModuleOnly then
         -- Found candidate incompleteness match; set flag to reflect
         -- incompleteness and continue to next element of module list
         IncompletenessMatch := True;
      else
         -- Examine the entity list
         EntityPointer := ModulePointer.EntityList;
         if EntityPointer = null then
            -- No entity list, so implicit mention of all entities;
            -- found candidate incompleteness match; set flag to reflect
            -- incompleteness and continue to next element of module list
            IncompletenessMatch := True;
         else
```

```
          -- Traverse entity list to look for Entity;
          -- because entity list is attached to incompleteness item,
          -- result cannot be better than "candidate incompleteness match", so
          -- halt traversal if incompleteness item found first
          while EntityPointer /= null loop
            if EntityPointer.Completeness = Complete then
              -- Not an incompleteness item
              if EntityPointer.EntityName = Entity then
                -- Found the entity name; candidate incompleteness match,
                -- so set flag to reflect incompleteness and
                -- continue to next element of module list
                IncompletenessMatch := True;
                exit;
              end if;
            else
              -- Found incompleteness item; set flag to reflect incompleteness
              -- and continue to next element of module list
              IncompletenessMatch := True;
              exit;
            end if;
            -- Move pointer to next entity list element
            EntityPointer := EntityPointer.NextElement;
          end loop;
        end if;
      end if;
    end if;
    -- Move pointer to next module list element
    ModulePointer := ModulePointer.NextElement;
  end loop;
  -- Fell through main loop, so no exact match found;
  -- check for incompleteness match
  if IncompletenessMatch then
    -- Found incompleteness match
    return FoundIncomplete;
  else
    -- No possible match found
    return NotFound;
  end if;
end LookupInClause;

...;

end AccessClauseUtilities;
```