The Theory of Interface Slicing

Jon Beck
West Virginia University Research Corporation

May 1, 1993

JOHNSON
GRANT
IN-61-CR
167280
p. 48

Cooperative Agreement NCC 9-16
Research Activity No. RB.10:
RBSE: Component Classification Support

NASA Technology Utilization Program
NASA Headquarters

ricis

Research Institute for Computing and Information Systems
University of Houston-Clear Lake

WHITE PAPER

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Jon Beck of West Virginia University. Dr. E. T. Dickerson served as RICIS research coordinator.

Funding was provided by the NASA Technology Utilization Program, NASA Headquarters, Code C, through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Ernest M. Fridge III, Deputy Chief of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

The Theory of Interface Slicing

Jon Beck
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
beck@cs.wvu.edu

1 May, 1993

1 Introduction

1.1 Statement of the Problem

Interface slicing is a new tool which has been developed to facilitate reuse-based software engineering, by addressing the following problems, needs and issues.

1.1.1 Size of Systems Incorporating Reused Modules

The reuse of tested, robust artifacts from previous software development efforts can result in savings in the current development effort due to the time saved both from the creative effort proper and also from the reduced maintenance needed for the reuse of proven, tested components [9]. There is a problem with reusing some artifacts. Typically any particular software system will require only a subset of the functionality of a given repository component. This is especially likely to be true in three cases. The first is the case of a component written to be a general component in a non-domain-specific repository. Such a component which was written for reuse will contain all possible anticipated functionality, the better to accommodate all possible anticipated applications. The second case in which only a subset of a component would typically be desired is that of a component in a domain-specific repository which was originally written for a specific system. Such a component will typically have custom functionalities tailored for its original target system which will not be needed when it is used in a new application. The third case is that of a component which has been used and reused many times, each time having a bit more functionality added to it. This is exemplified by the creeping featurism of Unix programs. Long-lived components can accrete numerous operations and functionality over their lifetimes, in what is called the kitchen sink syndrome.

In most software development environments, when a reusable module is incorporated into a software system under development, the entire module is imported into the software system. This includes all visible and hidden variables, subprograms, and types. This is easily demonstrated by a megabyte-sized "Hello, world" Ada program. Standard size optimizers and dead-code detectors cannot address this size problem because modules are separately compiled without any knowledge of how the module will be used. Only at system composition time is there sufficient knowledge to determine what part of an imported module is needed, and what part is useless in this specific system.

1.1.2 Knowledge Requirements for Program Modification

If a reusable module provides desired functionality in a current development effort, but results in a very large system due to unused extra functionality, then a solution is to remove the unnecessary functionality and retain only that which is desired. However, "removing the unnecessary functionality" means modifying a module, an act which opens a Pandora's box of difficulties. Modifying a module requires extensive knowledge about the module. According to Kozaczynski, et al. [17], program modification is a knowledge-intensive activity, far more than simple editing. When maintainers modify a program, it is insufficient for them to narrowly understand only the syntax and semantics of the code; rather, the maintainer must gain an general understanding of its functionality — what it is supposed to do — before maintenance can commence. The effort involved in understanding a reusable module sufficiently well to modify it nullifies much of the savings attributable to its reuse in the first place. Another portion of the savings of reuse is consumed in the necessity of retesting the modified module. The danger of introducing new bugs into code as a result of modification is famous in the software engineering community to the point of anecdote. Furthermore, modifying a component for the purpose of compositional reuse runs counter to the concept of verbatim reuse. This concept discourages modifications to distributed copies of repository components with the attendant difficulties of maintenance and consistency, and instead calls for modifications to be made only to the base repository component for perfective or corrective reasons.

Interface slicing does not address the entire problem of software modification. Rather, it is narrowly focused on providing the ability to modify a reusable module by removing unnecessary functionality without the attendant necessity of having extensive knowledge of the module before the modification takes place and without having to extensively test the module after the modification has taken place. This action does not violate verbatim reuse, because the modification is an automatic transformation at the time of system composition, in the same spirit as compilation, and does not alter the logical design or structure of the reused component.

1.1.3 Program Understanding for Reverse Engineering

Many modifications do not fall into the above category. There are times when it is necessary to thoroughly understand an existing module so that its manual modification can be undertaken. The discovery or recovery of that knowledge falls within the scope of reverse engineering. While many tools and techniques exist for the acquisition of knowledge about existing systems, there are gaps in the capabilities or efficiencies of those tools and techniques. In particular, there is an unexploited strategy which reduces the amount of knowledge needed for modification of a module by reducing its size and complexity, as opposed to the conventional strategy of employing a more powerful tool to gain more knowledge about the module. This new strategy entails the need for a tool or technique which reduces the burden of module comprehension for modification by reducing the size and complexity of the module to be modified.

There are situations in which it is not possible to reduce the size or complexity of a module, but in which there is still a necessity for discovering knowledge about the module. Again, there are existing techniques for this, but their coverage is not perfect for all situations. Therefore, there is a general need for a tool which aids in the acquisition of certain types of knowledge about an existing module, distinct from the technique which reduces the module size.

1.1.4 Module Granularity and Domain Management

The DoD Software Reuse Vision and Strategy defines the concept of centrally managed reuse within a domain [11]. According to this plan, reuse-based software engineering is to be administered from a domain management office which will perform the domain engineering tasks of domain analysis, architecture development, and the creation and recovery of reusable components. In addition to other tasks, the domain manager will assist application engineers by disseminating modified versions of repository components for the maintenance of systems which were developed from the domain model using repository components. When maintenance is performed on a repository component by a domain engineer, the domain manager is responsible for updating clients of that component [21].

In a reuse repository environment, there are conflicting benefits and drawbacks due to component size or granularity. On one hand, there are specific benefits from fine component granularity. A system created by composition of reusable components can be viewed as a pyramid of components [24] in which the lower levels consist of small, non-domain-specific components which are easily understood and which contain little design or architectural knowledge relating to the complete system. These small components are easiest for the application engineer to reuse; reuse at this level results in a large number of components being reused. Also, since these small components are relatively non-domain-specific, each component is applicable to many domains, resulting in more efficient creation and recovery of reusable components for the domain manager and engineer. If the repository mechanisms include some facility for localizing the retesting and revalidation of modified components, then the domain manager's post-modification task of client update is easier if the modification is to a small component rather than to a large component. This is meant in the sense that if some functionality x is supplied to clients, and x is modified, then fewer clients are affected by the change if x is contained in a small component than if it is contained in a large component.

On the other hand, a very strong benefit of coarse component granularity is a greater payoff for the reuse of each component, as more domain and design knowledge is captured in a large component than in a small, resulting in a smaller input of application knowledge by the application engineer.

These conflicting benefits of fine and coarse component granularity indicate a need for a tool which can balance the two and provide the benefits of both in a single reuse repository scenario.

1.1.5 Time and Space Complexity of Conventional Slicing

A final area which calls for the services of a new tool is that of conventional program slicing [25]. As evidenced by the current interest in conventional program slicing for many applications (e.g., [2,5,13]), slicing is a worthwhile technology. Unfortunately, conventional program slicing can be an expensive operation, varying in time complexity from $O(n^2)$ to $O(n^3)$, depending upon the slicing technique employed, where n is the number of program statements, expressions, or quadruples. Dependence graph representations of programs, employed by many conventional slicing techniques, can occupy several times as much space as occupied by more traditional representation forms, especially when interprocedural data and control flow is included in the representation. Because of this, it may be difficult to apply conventional program slicing to large, real systems [22]. Interface slicing, however, can be performed in at most two passes over the source code, giving a time requirement of $O(n)$, where n is the number of program elements. A piece of code which is too large or complex for conventional program slicing may be sufficiently reduced in size by a preliminary pass of an interface slicer to enable subsequent conventional slicing. In such a case, a wedding of the two slicing technologies would produce tangible benefits. Even when conventional slicing is possible, however, it may not be necessary. Conventional slicing is at the level of statements and expressions, which is a very detailed level. In programs composed of very many small subprograms, slicing at the subprogram level may yield slices small enough for immediate comprehension, obviating the need for a more detailed, difficult and expensive slicing at the statement level.

1.2 Statement of the Solution

This paper is based on the thesis that interface slicing is a way to approach all of the problems, needs, and concerns listed above with a single technique. Specifically, we assert that:

- Interface slicing theory exists and has a mathematical basis which can be demonstrated. Certain structures of existing programs can be represented as interface slices. There is a mapping between the manipulations of the mathematical representation of interface slices and the semantics of the program structures being represented, such that inferences made or conclusions reached based on the mathematical representation imply corresponding inferences or conclusions in program semantics.
- Interface slicing shares a common intuitive and philosophical background with conventional slicing.
- Interface slicing differs from conventional slicing both mathematically and procedurally.
- The problems in language theory and software engineering for which conventional and interface slicing are appropriate intersect but neither is a subset of the other.

The specific contribution which this paper makes is the definition of a new form of static program analysis called interface slicing.

2 Definitions and Representation Issues

2.1 Terminology and Definitions

In this paper, we will employ the following conventions unless we explicitly note otherwise. A *subprogram* is a unit of code; the term is intended to denote regular procedures and functions, including "main" programs, and where appropriate, more exotic code units such as Ada tasks. If the language under discussion permits, a subprogram may be specification, body, or both. If subprogram *A* contains subprogram *B* nested within it, a reference to *A* will in general *not* include any reference to *B* unless *B* is explicitly referenced. A *package* is a collection of subprograms and implies at least the possibility of separate compilation, with allowances made for language systems which do not have the capacity of separate compilation. Package includes the Ada notion of package but is not limited to Ada, as it also may be used to mean a set of units in a standard object library. *Module* is used as a general term to include both subprogram and package as described above, when specifying either would be too restrictive. *Component* specifically refers to a module potentially residing in a reuse repository. A component is thus a code asset of a repository, either before or after it has been reused by incorporation into a software system. We use the term *element* to mean a named programmatic entity. Types, structures, variables, subprograms, tasks, and exceptions are all included in this term, but statements, even labeled statements, are specifically excluded.

We describe certain characteristics of a program element by using the terms visible, hidden, unprotected, and protected. A program element is *visible* if it is visible and available at least for examination by non-privileged portions of the software system. We use the term *hidden* to refer to a program element which is not visible outside the scope of its module. An element is *unprotected* if there is no language mechanism applied to it which prevents non-privileged access to its internal structure, while an element is *protected* if some form of language-based mechanism, not including simple scope, is used to limit access to its internal structure by non-privileged portions of the system. Thus, visible and hidden refer to access to the element's name, while unprotected and protected refer to an element's internal structure. For example, from the standpoint of a main Pascal program, a local variable within a subprogram is visible and unprotected, as only the scoping conventions of the language make the variable inaccessible to the main program. As another example, the variable *MyVariable* declared on line 3 of the Ada package specification shown in Figure 1 is visible and protected. It is visible because it is available by name to any part of the system which *with*s this package, and protected because its internal structure is not available outside the package. Finally, type *MyType2* on line 6 is hidden and protected, as neither its name nor its structure are visible outside the package. The reason for making a point of using this terminology is twofold. First, these concepts are language-independent, even though they have been implemented to various extents in different languages. However, as the implementations are generally not pure, we do not wish to use terms of a specific language, in order to avoid the implication that we are referring to a specific language's implementation of one of these concepts. Second, in discussing the mechanisms of interface slicing, there are important considerations based upon a program element's visibility and protection status before and after the slicing transformation. Since these considerations are language-independent, it is important that colorations from existing language implementations not creep into the discussion.

2.2 General Theoretic Concepts

2.2.1 Sets and Graphs

For subset[1] notation, we use $A \subseteq B$ to indicate that set A is a subset of, and possibly the same as, set B, and $A \subset B$ to indicate that A is a proper subset of B so that $A \neq B$. The cardinality of A is denoted by $|A|$. A graph[2] is a pair (N,A) where N is a finite nonempty set of *nodes* or *vertices*, and $A \subseteq N \times N$ is a set of directed *edges* or *arcs*

```
1 package MyPackage is
2   type MyType1 is private;
3   MyVariable: MyType1;
4   ...
5 private
6   type MyType2 is ...;
7   ...
8 end MyPackage;
```

Figure 1 Example for visibility and protection

1 Unless specified otherwise, all sets herein are finite.

2 Unless specified otherwise, all graphs herein are directed graphs.

between nodes. The edge denoted (x,y) or $x \rightarrow y$ leaves the tail or source node x and enters the head or target node y, making x a *predecessor* of y, and y a *successor* of x. The number of predecessors of a node is its *in-degree*, and the number of successors is its *out-degree*. Since the edges are members of a set, a graph may have at most one edge from a given node x to another node y. A structure which allows multiple edges from one node to another is a *multigraph*. In other words, a graph consists of a set of nodes and a set of edges, while a multigraph consists of a set of nodes and a bag of edges. A *path* from x_1 to x_k is a sequence of length k of vertices $(x_1, x_2, \ldots, x_k)$ with $x_i \in N$, $1 \leq i < k\text{-}1$ such that each pair $x_i \rightarrow x_{i+1} \in A$. Two graphs G_1 and G_2 are *isomorphic*, denoted $G_1 = G_2$ iff there exists a one-to-one correspondence between their sets of nodes and a one-to-one correspondence between the sets of edges such that the corresponding edges also agree on the corresponding source and target nodes.

2.2.2 Partial Orderings and Lattices

A reflexive, antisymmetric, transitive relation on a set S is a *partial ordering*, denoted by $\sqsubseteq$. The pair $(S,\sqsubseteq)$ is a partially ordered set, or *poset*. For a given poset $(S,\sqsubseteq)$, $\sqsubset$ denotes the reflexive reduction where $\sqsubset = \sqsubseteq - \{(x,x) | x \in S\}$. Given a poset $(S,\sqsubseteq)$ with $a,b \in S$, then a *join* or *least upper bound* of a and b is an element c:

$$c \in S \mid a \sqsubseteq c \wedge b \sqsubseteq c \wedge \neg\exists x(x \in S \wedge a \sqsubseteq x \sqsubset c \wedge b \sqsubseteq x \sqsubset c)$$

Similarly, a *meet* or *greatest lower bound* of a and b is an element c such that:

$$c \in S \mid c \sqsubseteq a \wedge c \sqsubseteq b \wedge \neg\exists x(x \in S \wedge c \sqsubset x \sqsubseteq a \wedge c \sqsubset x \sqsubseteq b)$$

If a and b have a unique join, it is denoted $a \sqcup b$; a unique meet, $a \sqcap b$. A set of pairwise incomparable elements of a poset is called a *cochain*.

A *lattice* L is a poset, every pair of elements of which have a unique join and meet; the lattice is denoted by the triple $L = (S,\sqcup,\sqcap)$. An element a of a lattice L is a *minimal element* if there does not exist an element b of L such that $b \sqsubset a$. A minimal element a is also a *least element* if $a \sqsubseteq b$ for every b in L. If L has a least element, it is unique. Similarly, a is a *maximal element* if there does not exist a b in L such that $a \sqsubset b$, and a unique maximal element a is a *greatest element* if $b \sqsubseteq a$ for every b in L. Each element of the poset is said to be contained in a node of the lattice. Sometimes the node and the element it contains are used interchangeably.

The *power set* of a set S, denoted 2^S, is the set of all subsets of S, i.e., $2^S = \{T \mid T \subseteq S\}$. A particular lattice structure of interest in this paper is the following. Given a finite set S and the usual set union and intersection operations denoted by $\cup$ and $\cap$, the poset $(2^S,\subseteq)$ is the basis for the lattice $(2^S,\cup,\cap)$. Each node of this lattice contains a unique subset of the elements of 2^S. We will often refer to this structure as a subset inclusion lattice[3]. This lattice is of interest here because if S is the set of all statements of a program then 2^S is the set of all subsets of the program statements. Since a slice is a subset of program

3 $(2^S,\cup,\cap)$ is also known as a Boolean algebra, and is characterized by being a lattice with distributivity, existence of greatest and least elements, and complements.

statements, then every slice corresponds to an element of 2^S, and thus to a node in the lattice $(2^S, \cup, \cap)$. This lattice is therefore a convenient structure for discussing slices of a program and their relationships.

This lattice may be depicted using a Hasse diagram as a graph in which the greatest element, the set S, is a node of in-degree 0, and the least element, $\varnothing$, is a node of out-degree 0. An edge $a \rightarrow b$ is drawn in the diagram iff $b \subsetneq a$ and there does not exist a node c such that $b \subsetneq c \subsetneq a$. In this case, a is considered the parent of b, and b the child of a. Notice that this excludes the possibility that two separate nodes in the lattice contain the same element. For example, given the set $S = \{1,2,3\}$, the lattice $L = (2^S, \cup, \cap)$ is shown in Figure 2.

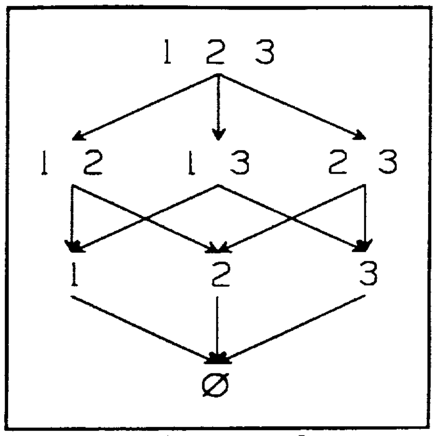

Figure 2 Power set lattice for {1,2,3}

Given a poset $(A, \sqsubset)$, the relation $\sqsubset$ defines a set of ordered pairs of elements of A. Given a set $B \subseteq A$, then some of the ordered pairs of $\sqsubset$ may also be ordered pairs of elements of B. The set of those elements of $\sqsubset$ which consist of ordered pairs of elements of B is called the *restriction* of $\sqsubset$ to B, and is a partial ordering of B. If the poset $(A, \sqsubset)$ is a lattice, the new poset $(B, \sqsubset)$ is not necessarily a lattice. However, if B includes at least the greatest and least elements of A, then B must also be a lattice. For example, let A be 2^S as in the example above, so that the poset $(A, \subseteq)$ is shown in Figure 2, and let $B = \{\{1,2\},\{2,3\}\}$. Then $(B, \subseteq)$ is a poset, but is not a lattice. However, if we consider B' as B augmented with the greatest and least elements of A, so that $B' = \{\varnothing,\{1,2\},\{2,3\},\{1,2,3\}\}$, then $(B', \subseteq)$ is a lattice.

2.2.3 Dependences

According to Podgurski and Clarke [19], *dependences*[4] are relationships among program statements and are of two types, control and data flow (or simply data) dependences. In a program, two types of situations create dependences between two statements, or between a statement and a predicate. In Figure 3, a *control dependence* exists between the predicate A on line 2 and the statement B of line 3; the execution of B is control-dependent on the value of A because the value of A immediately controls the execution of B.

```
1 begin
2   if A then
3     B;
4   end if;
5 end;
```

Figure 3 Control dependence

In Figure 4, the assignment statement on line 3 is *data dependent* on the assignment statement on line 2, because the correctness of C's value in line 3 depends upon the prior execution of the statement on line 2. Thus a data dependence exists between two statements when a variable in one may have an incorrect value if the order of execution of the two statements is reversed. Another way of stating this is

```
1  begin
2     C := f(D);
3     E := C;
4  end;
```

Figure 4 Data dependence

4 Some authors use *dependency*, singular, and *dependencies*, plural.

that one statement is data dependent upon another if data can potentially flow from the latter to the former in a sequence of assignment statements.

2.3 Issues of Program Representation

In one strict view, only a set of machine instructions in a computer's memory can be termed a "computer program". But this strict interpretation is usually relaxed so that various program representations are spoken of as being, or being equivalent to, computer programs. Common program representation schemes include high-level source code, pseudocode, and flow charts; the purpose of these various representation forms depends upon the context and may include human readability, annotation for verifiability, and transformation for application to a different platform such as a parallel multiprocessor. In the context of program slicing, program representations are used to facilitate the automation of slicing. For a very simple program, a slice can be prepared by hand. But with increasing size and complexity of the program, there is increasing need to employ the assistance of automation. Current automated slicing techniques require that information gleaned from a source code form of the program to be sliced be transformed into some different program representation during the slicing process. Various program representation schemes have resulted from the search for ever more complete and efficient slicing techniques. In the discussions of program representations which follow, it is important to remember that there is no single correct way of building, say, a dependence graph program representation, nor is there a single exact set of information which must be available to enable slicing.

A program which is to be modeled with one of these representations is written in some language. While this discussion concentrates on executable languages, we do not wish to exclude the possibility of including non-executable forms such as pseudocode or formal specifications. Each language has its own peculiarities which affect the way it can be represented, and the form of the representation. In the explanations of the different representation mechanisms below, it is useful to keep in mind the differences in languages. For example, C has a switch statement structure which allows multiple exits, but C has no nested subprograms; Pascal has a regular, partitioning, single-exit case statement, but also has nested subprograms; FORTRAN has an equivalence statement parameter passing mechanism which allows variable aliasing by array overlap; Ada has various synchronization mechanisms for tasking. There probably is no perfect universal program representation scheme because each of these language features may call for a somewhat different representation mechanism. Conversely, a program representation may well serve to bridge the gap between disparate languages.

It is common to represent programs as graphs and lattices pictorially with closed shapes standing for nodes and directed lines representing edges. For simplicity, in fact, the picture is often spoken of as "being" the graph or lattice, or the graph as "being" the program, but it is important to keep in mind that the picture or graph drawing is only a representation of an abstract mathematical or programmatic entity. The model may be imbued with a set of desired semantics, with the nodes and arcs drawn in various shapes and given various labels, provided that there is an unambiguous and consistent mapping between the semantics and the model such that the mathematical or syntactic integrity of the model is maintained. In this case, results derived from mathematical proofs and manipulations on the model give strong credence to the corresponding semantics.

3 Interface Slicing

3.1 Introduction

To date, several major forms of program slicing have been developed. Listed briefly, the major forms and some common techniques of performing them are:

- Static Slicing
    - Incremental flow analysis [25]
    - Dependence graph reachability [15]
    - Information-flow relation equations [8]

- Dynamic Slicing
    - Incremental flow analysis [16]
    - Dependence graph reachability [3]

These forms will be referred to collectively in this paper as *conventional* slicing, in contrast to the new form of slicing which is presented here. In this paper,[5] we present and develop an entirely new form of program slicing, *interface slicing*. This form of slicing is also discussed by Beck and Eichmann [6,7,12].

Intuitively, an *interface slice* may be viewed as a subset of the behavior of a module, similarly to the original notion of the conventional static slice. However, while a conventional slice seeks to isolate the behavior of a specified set of program variables, even across module boundaries, an interface slice seeks to isolate specified functionalities which a given module exports to its containing software system.

The purpose for which interface slicing was developed is very different from that for which conventional slicing was developed. While conventional slicing was originally designed primarily for debugging and comprehension, interface slicing was primarily investigated as a tool for use in a reuse repository environment to 1) enhance the reusability of components in the repository and 2) improve the quality of the system which results from a software development-with-reuse effort. But just as the role of conventional slicing has expanded to embrace many areas of both forward and reverse software engineering, so we see a broad applicability of interface slicing to software engineering efforts in general, including all phases of new system development, as well as comprehension, maintenance, redocumentation, and reengineering of legacy systems.

3.2 A Simple Example

We present here a simple example designed to give the flavor of interface slicing. The example illustrates one application of interface slicing, in which it is used to project a subset of an Ada package's functionality.

Consider a simple abstract data type (ADT) implemented as an Ada package which exports the operations necessary to implement a boolean toggle and which maintains the

5 Much of the material in this section appeared in a condensed form in [7].

```
 1  package toggle1 is
 2    function on return boolean;
 3    function off return boolean;
 4    procedure set;
 5    procedure reset;
 6  end toggle1;
 7  package body toggle1 is
 8    value: boolean := false;
 9    function on return boolean is
10      begin
11        return value = true;
12      end on;
13    function off return boolean is
14      begin
15        return value = false;
16      end off;
17    procedure set is
18      begin
19        value := true;
20      end set;
21    procedure reset is
22      begin
23        value := false;
24      end reset;
25  end toggle1;
```

Figure 5 A boolean toggle package

Consider a simple abstract data type (ADT) implemented as an Ada package which exports the operations necessary to implement a boolean toggle and which maintains the state of the toggle. An example of such an ADT is given in Figure 5. This package exports the operations *on, off, set,* and *reset*. *On* and *off* are query operations which examine the state of the toggle, while *set* and *reset* are operations which modify the state of the toggle. The actual state of the toggle is maintained in the hidden variable *value*. Suppose that a program under development needs the functionality that this toggle ADT provides. In a standard software development scenario in which this package is available in the repository, the specification of the package in lines 1 - 6 would be available for inspection. After being selected from the repository as the appropriate component, the package would be incorporated into the software system. *Toggle1* would be *with*ed in the appropriate scope of the system under development which needed the toggle functionality, and the system would then have all the functionality, all four operations, of the toggle package available to it.

However, suppose that in the course of developing a system we find that we need, not all, but only some of the functionality of the *toggle1* package. For this example, suppose that we have need of only the *on, set,* and *reset* operations, but do not need the *off* operation. In a standard development scenario, we have two options, neither of which is ideal. The first option is to incorporate the complete toggle package *in toto*, exactly as described above. The disadvantage of this option is that in the finished software system, the *off* function becomes "dead" code in the sense that it is never called or executed. This is the kitchen sink syndrome which characterizes development in languages such as Ada. The system is larger than necessary in both source and executable forms, taking extra time and space to compile and link. Furthermore, the dead code remains for the life of the system (which may span decades), constantly serving as a source of extra time and confusion for the software engineer charged with maintaining the system. The confusion caused by dead code is due to the natural tendency to assume that each line of code in a system actually does something. This assumption is violated by dead code.

Alternatively, the second option is to manually edit the source code of *toggle1* and delete the *off* operation from the body and specification of the package. The disadvantage of this option is that the editing operation requires full code comprehension of the *toggle1* package and involves the very real danger of introducing logical bugs into the package due to hidden linkages and dependences and introducing syntactic bugs due to typing or editing errors. In addition, stating the option of manually editing the source code assumes that the developer has access to the source code. This is not necessarily the case, especially in the context of a reuse repository which contains proprietary software which has been licensed for (re)use, but not for copying, reverse engineering, or modification. It is easy to propose a repository structure which gives full source code access to an automated interface slice tool, while restricting human access to just the specification, thus preserving the integrity of proprietary software rights.

Interface slicing provides a third alternative which does not have the disadvantages of the two options above. We wish to use a subset of the behavior of, or a subset of the functionality provided by, a component. All of the functionality exported by an encapsulated module is, by definition, described in the *interface* of that module, but we are interested in a subset of that functionality. In the case of an Ada package, the interface is the package specification. In effect, we wish to remove, i.e. slice away, the unneeded functionality, as in manual editing of the source code, but without the attendant problems of editing. By examining only the specification of the module we know that the module contains some functionality that we want in our system which is under development, but we also know that it contains more functionality than we want.

We thus invoke the notion of an interface slicing tool which takes as input 1) a complete module consisting of both an interface specification and a code body, and 2) a *list* which enumerates the subset of the module functionality which we desire. This list is the *interface slicing criterion.*

Definition 1 Interface Slicing Criterion. A possibly empty list of module elements which the module makes visible and available at least for reference to the surrounding system. □

The tool produces as output a slice, a new module which is a subset of the original, but which contains all and only the code necessary to support the functionality specified in the slicing criterion of desired operations. In the example above, we desired the functionality of the operations *on*, *set*, and *reset* in the *toggle1* package, but not that of *off*. A slice of *toggle1* on the slicing criterion ⟨*on*, *set*, *reset*⟩ is shown in Figure 6. Note that in

```
package toggle1 is
  function on return boolean;
  procedure set;
  procedure reset;
end toggle1;
package body toggle1 is
  value: boolean := false;
  function on return boolean is
    begin
      return value = true;
    end on;
  procedure set is
    begin
      value := true;
    end set;
  procedure reset is
    begin
      value := false;
    end reset;
end toggle1;
```

Figure 6 *toggle1* sliced on ⟨*on,set,reset*⟩

this simplest example, the slice consists merely of the original package without the unwanted function or its specification, just as would have been produced by manually deleting the *off* procedure from the package specification and body. This simple example does not have any linkages or dependences among its operations; we will now discuss a case which does.

As a second example, consider the slightly more sophisticated boolean toggle which is implemented by the package shown in Figure 7. In addition to the operations of *toggle1*, this package also exports the operation *swap* which reverses the current value of the toggle. Suppose that we wish to include in our software system just the functionalities of the operations *on* and *swap*, corresponding to a slicing criterion of ⟨*on, swap*⟩. In this situation, a naive editing of the *toggle2* package to remove *off, set,* and *reset* will no longer suffice, because *swap* has dependences on *on, set,* and *reset*. In order to include *on* and *swap*, we must also include *set* and *reset*. The result of interface slicing *toggle2* on ⟨*on, swap*⟩ is shown in Figure 8. Note that while *set* and *reset* no longer appear in the interface, they do appear in the body of the sliced package.

```
package toggle2 is
  function on return boolean;
  function off return boolean;
  procedure set;
  procedure reset;
  procedure swap;
end toggle2;
package body toggle2 is
  value: boolean := false;
  function on return boolean is
    begin
      return value = true;
    end on;
  function off return boolean is
    begin
      return value = false;
    end off;
  procedure set is
    begin
      value := true;
    end set;
  procedure reset is
    begin
      value := false;
    end reset;
  procedure swap is
    begin
      if on then reset;
      else set;
      end if;
    end swap;
end toggle2;
```

Figure 7 A larger boolean toggle package

It is important to realize that the specific dependences among *swap, on, set,* and *reset* in this example arise due to the specific code implementation of the package. They are not due to the design of the surrounding software system, nor to the requirements or specification of the toggle package. It is easy to envision a toggle implementation in which *swap* depends upon *off* rather than upon *on*, or even one in which *swap* depends upon neither. This comprehension of and knowledge about the package is required for manual editing; interface slicing is designed in part to obviate this comprehension requirement. With an interface slicing tool available, we do not have to know anything about the internal dependences of the toggle package, as the slicing tool does the dependence analysis during its operation.

Definition 2 Interface Slice. Given a syntactically correct module M and an interface slicing criterion $C = \langle e_1, e_2, \ldots, e_k \rangle$, where the e_i are visible elements exported by M, an *interface slice* $S = M(C)$ is a syntactically correct module with the following properties:

1 $M(C)$ can be formed by deleting zero or more declarations or subprograms from M;

2 $M(C)$ exports a set of elements which are syntactically and semantically equivalent to the elements in C, and no other elements. □

Note that this definition only allows elements which are visible before slicing to appear in the criterion, and only allows elements which are contained in the criterion to remain visible after the slicing. (See also Section 3.6, Page 16.)

```
package toggle2 is
  function on return boolean;
  procedure swap;
end toggle2;
package body toggle2 is
  value: boolean := false;
  function on return boolean is
    begin
      return value = true;
    end on;
  procedure set is
    begin
      value := true;
    end set;
  procedure reset is
    begin
      value := false;
    end reset;
  procedure swap is
    begin
      if on then reset;
      else set;
      end if;
    end swap;
end toggle2;
```

Figure 8 *toggle2* sliced on ⟨*on,swap*⟩

3.3 An Interface Slicing Mechanism

The previous examples were based on interface slicing being used to project a subset of the functionality exported by a package. The examples illustrate the usefulness of interface slicing but do not indicate how it can be accomplished. In this section, we demonstrate a method for generating the interface slices of the previous section.

As stated above, the interface slicing tool has as input a syntactically correct module and an interface slicing criterion which is a list of desired visible subprograms, types, and variables. This list is supplied without knowledge of the module implementation. The problem at hand for the slicer is to determine from a static analysis of the module which portions of the module it should retain in order to support the items in the slicing criterion, and which portions can be safely deleted. In addition, it must determine which retained portions should reside in the specification, and which should be removed from the specification and be retained only in the body. The solution to the problem is to perform a reachability analysis [14], based on the slicing criterion, of a dependence graph of the package.

3.4 The Interface Dependence Graph

In the following discussion, we make the simplifying assumption of a language without procedure or block nesting. We make this assumption because we wish to concentrate on the ideas of interface slicing and avoid getting bogged down in laborious cases of the different variations of nesting and scope in the various languages, none of which is new and all of which is covered in the literature. For example, C++ allows a declaration to appear anywhere a statement can, while Ada declarations must be confined to a declarative part; Ada allows unlimited nesting of blocks and subprograms, while C++ does not. Each of these variations requires different handling of scope and visibility issues, and requires more language-specific treatment which is more properly treated in an implementation report. This does not, however, detract from the applicability or power of interface slicing theory, as scope and nesting are well-understood areas of language systems. Therefore, we assume that all variable declarations and procedure definitions are global to the module. However, we do assume that some declarations and definitions are visible, and others hidden. Also, we often employ Ada-centric terminology, but only out of a desire to use consistent and familiar terminology in order to evoke a clear mental image.

The specific dependence graph required for the analysis of a package we term the package's *interface dependence graph*. The package interface dependence graph can be constructed with at most two passes[6] over the source code of the package's specification and body in the following manner. Each node of the graph corresponds to a statement which defines: any type (including subtypes, subranges, generics, etc.), any global variable (including constants and generic formal parameters), or any subprogram (including procedures, functions, tasks, and exceptions) — in short, any unique, named, global program entity.[7] We here make the simplifying assumption for the purposes of discussion, but without loss of generality, that there is at most one definition per statement and per line of source code. Every node is labeled with the name of the defined program element to which the node corresponds, and every node is annotated with various information necessary for an unambiguous determination of the program element to which the node corresponds. Depending upon the language under consideration, this information may include the node's corresponding line number in the source code, the type signature of the corresponding program element, and, in the case of nesting, scoping and name over-riding information. Since we are not considering nesting here, we will not attempt to keep track of the scope of definitions in the following examples: every subprogram is global to the package, and only global variables and type definitions are of interest.

As the name implies, the edges of the interface dependence graph are dependence edges. The edges are constructed as follows. There is an edge from node x, corresponding to program element X, to node y, corresponding to program element Y, if X contains a definition- or use-reference to Y. These are static reference edges. There must also be dynamic reference edges: if X contains a statement which has a pointer, there must be an edge from X to every program element which contains a potential target of the pointer. Self-edges, indicating direct recursion, are not necessary and are therefore excluded, but indirect recursion is certainly possible, and will appear as a cycle in the interface de-

6 If there is no nesting and all definitions are placed before uses, then only a single pass is required.

7 This does not include a renaming of an already-existing program entity. In the case of a renaming, the slicer must keep track of the name, but not create another node for the renamed element.

pendence graph. The interface dependence graph thus combines the information of a standard call graph with a data dependence graph for global (global to the package) type definitions and variables. Interface dependence graphs for the *toggle1* and *toggle2* packages described above are shown in Figure 9 (with annotations omitted for clarity). A more formal definition of the interface dependence graph follows.

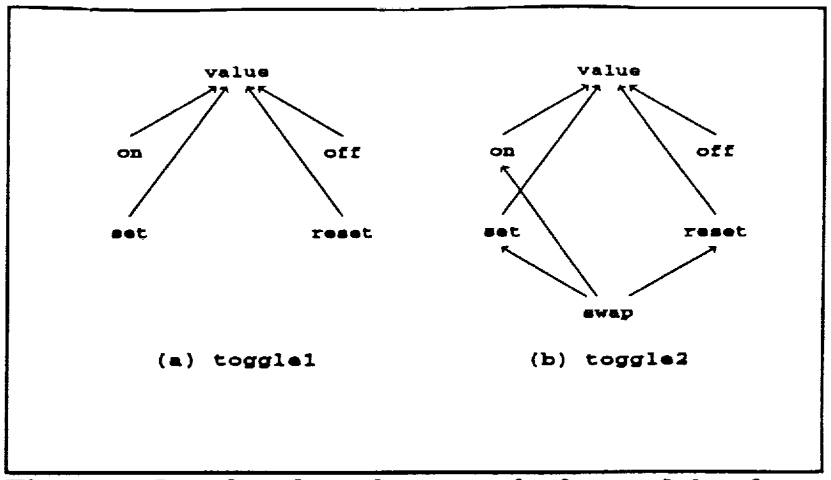

Figure 9 Interface dependence graphs for *toggle1* and *toggle2*

Definition 3 Interface Reference. Given a module M, containing uniquely named program entities X and Y, there is an *interface reference* from X to Y if $X \neq Y$ and either

1 X contains a statement or definition which references Y, or

2 X contains a pointer which has Y or some portion of Y as a potential target.

The interface references of X form a set denoted by *iref(X)*. □

Definition 4 Interface Dependence Graph. Given a module M, the *interface dependence graph* for M is an annotated graph $G = (N,A)$ where N is the set of nodes $x,y,\ldots$ which correspond to the uniquely named programmatic entities $X,Y,\ldots$, and $A \in N \times N = \{(x \rightarrow y) \mid Y \in iref(X)\}$. Each node in N is annotated with the line number of the source statement in which the name of corresponding entity is first encountered in M and sufficient name, typemark, and scoping information to unambiguously identify the entity to which N corresponds. □

In general, a package will have some visible program elements, and some hidden ones. Hidden program elements are not available to be used in an interface slicing criterion; only exported elements can be in the slicing criterion. However, hidden program elements must be included in the interface dependence graph, as the transitive closure of a visible element may flow to a hidden element. Because of this, an element's visibility status is a part of the definition of the slicing criterion, but not of the interface dependence graph. Thus, with the exception that hidden elements may not appear in the slicing criterion list, hidden elements are treated identically to visible ones during the interface slicing process. The slicer, being a pre-compilation code transformation tool with privileges similar to a compiler, has complete access to all portions of the source code, and thus is not hindered by these language mechanisms.

3.5 Slicing the Interface Dependence Graph

Once a package's interface dependence graph has been constructed as described above, an interface slice based on a given slicing criterion of desired functionality can be generated with its aid. Starting with the nodes in the graph which correspond to the named items in the slicing criterion, generate the transitive closure of those nodes by following the dependence edges. The interface slice consists of the definitions and subprograms which correspond to the transitive closure, plus any needed syntactic sugar (see Section 3.10) required for the package structure.

For instance, consider the interface slice which this process generates for the *toggle1* example discussed above. That example discussed an interface slice for the *toggle1* package, whose interface dependence graph is shown in Figure 9(a), based on the slicing criterion ⟨*on, set, reset*⟩. The transitive closure of this criterion in Figure 9(a) consists of the nodes *on, set, reset,* and *value.* This means that the slice should consist of the subprograms *on, set,* and *reset,* and the definition of *value.* This was the same conclusion reached in the intuitive consideration above, as shown in Figure 6. Similarly, the transitive closure of the slicing criterion ⟨*on, swap*⟩ in Figure 9(b) consists of the nodes *on, set, reset, swap,* and *value,* corresponding to the subprograms and variable by those names in the package *toggle2.* This also matches the conclusion reached above, as shown in Figure 8.

3.6 Position of Retained Elements

After the program elements to be retained are identified by the dependence analysis of the interface dependence graph, their final locations in the slice — the new package — must be determined. There are three rules which determine the final location of an element, depending upon the visibility of the element in the original package. The rules are:

1 *Elements which were hidden remain hidden.* This ensures that no previously hidden elements becomes visible, a situation which would violate the information hiding design of the original package.

2 *Elements named in the slicing criterion remain visible.* Recall that in order to be in the slicing criterion in the first place, an element must be visible in the original package. The definition of the interface slice requires that it is exactly the named elements which the slice must export. Therefore the named elements must remain visible in the slice.

3 *Elements not in the slicing criterion which were visible become hidden.* This allows the slice to conform to one principle of good modular design, which is that only the elements which a surrounding system uses should be exported by a module. It is this principle which is so easily violated in standard reuse-based software development. If our software system does not use *toggle2's reset* operation, then the package should not export the operation. If the operation is exported, then a future maintainer will search in vain for its use in the system, because no such use exists. This wastes time and hampers comprehension. Instead, this rule ensures that the *reset* operation becomes hidden, so that the maintainer will know with a glance that the operation is not used elsewhere in the system.

3.7 An Extended Example

The examples above illustrate the general concept of interface slicing, but leave out some important details. To fill in some of these details, we will next discuss a pair of generic Ada packages which originated in the public domain.[8] These packages were explicitly written to be used as building blocks for Ada programs, similarly to the components of Booch [10] or Uhl and Schmid [23]. The first of the packages implements the a *set* abstraction in the Ada package *SetPkgTemplate*. The package is instantiated by supplying it with two parameters, the first being the type of element which the set is to contain, and the second being a comparison function to determine the equality of two elements of this type. The package provides all the operations necessary to create, manipulate, query, and destroy sets. The specification of the set package is listed in Appendix A.

SetPkgTemplate is written to use a *list* abstraction as the underlying structure upon which it builds the set ADT, and so the set package requires the second of the two generic packages discussed here, which implements the list ADT as the Ada package *ListPkgTemplate*. This happens to be a singly-linked dynamic list implementation, although nothing in *SetPkgTemplate* requires this to be the case. *ListPkgTemplate* exports all the operations necessary to create, manipulate, query, and destroy lists. This package requires three generic parameters. The first two are similar to the generic parameters of the set package, namely, the type of element in the list and the equality function. The third generic parameter is a *copy* function which gives the list package the ability to copy a list element, to provide for one-level-deep copying of the list. The specification for the list package is listed in Appendix B.

3.8 A Single Level of Interface Slicing

Suppose that we wish to make use of sets and set operations in a program we are writing, but we have need for only a few of the set operations, namely, in this example, *create, insert,* and *equal*. In the repository we have available the source code for *SetPkgTemplate*; by examining its specification we see that it does provide a set type and operations for that type, and that in order to use *SetPkgTemplate*, we must supply the type *ElemType*. Therefore, we wish to slice *SetPkgTemplate* using the interface slicing criterion ⟨*Create, Insert, Equal, Set, ElemType*⟩. We want to include all the code necessary to allow me to use these three operations and two types, but would like to have only the necessary code, and no more. In order to slice the set package, we must examine the interface dependence graph for the set package, which is shown in Figure 10. The three operations *Create, Insert,* and *Equal*, and the types *Set* and *ElemType* are defined on lines 9, 11, 21, 6, and 3, respectively, of Appendix A. The transitive closure of these five nodes, corresponding to the desired slice of the set package, is shown in Figure 11.

Out of the total of 16 subprograms, one of which is a generic parameter, on 151 lines of code in the original package, the slice contains 8 subprograms on 84 lines of code. Thus interface slicing has reduced the number of subprograms and the number of lines of code by a factor of 2 in this example. A comparison of Figure 10 vs. Figure 11 shows visually the reduction in interface size of the sliced set package (see also Table 1, Page 24).

8 These packages were extracted and modified from the ASR repository on SIMTEL 20 and were written by B. Altus and R. Kownacki of Intermetrics.

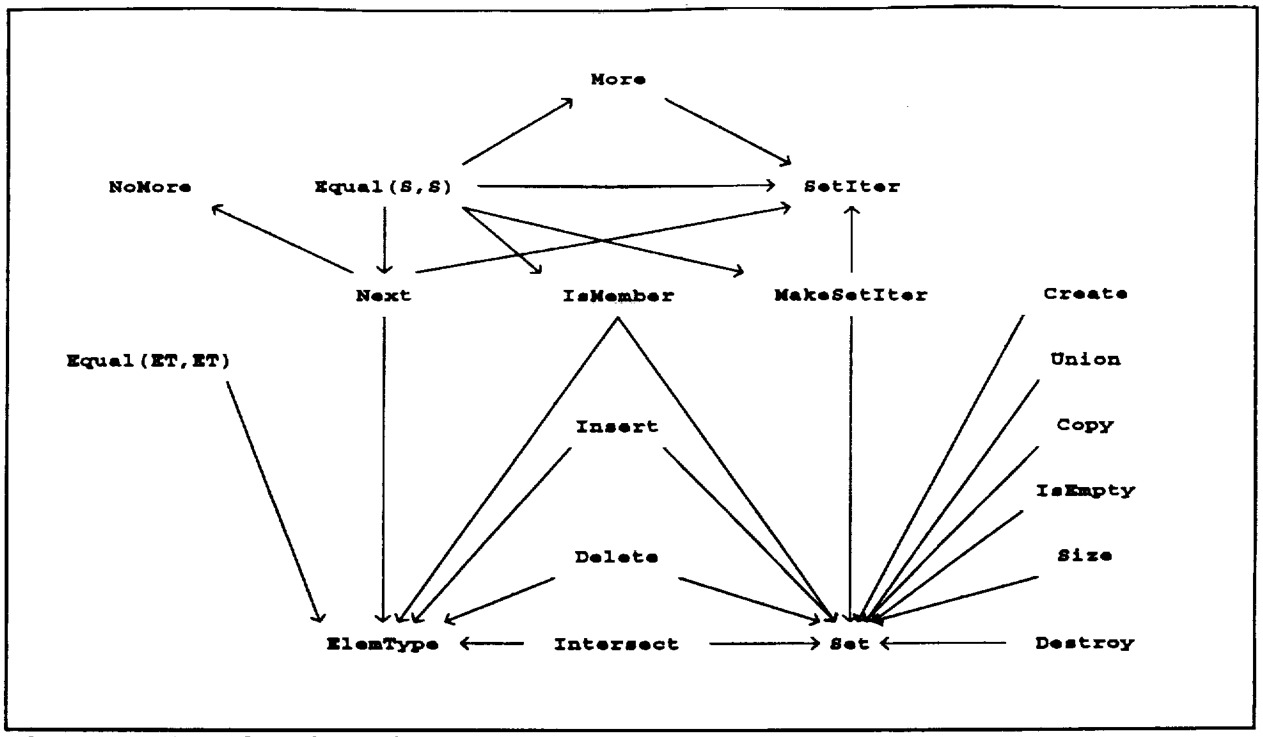

Figure 10 Interface dependence graph for *SetPkgTemplate*

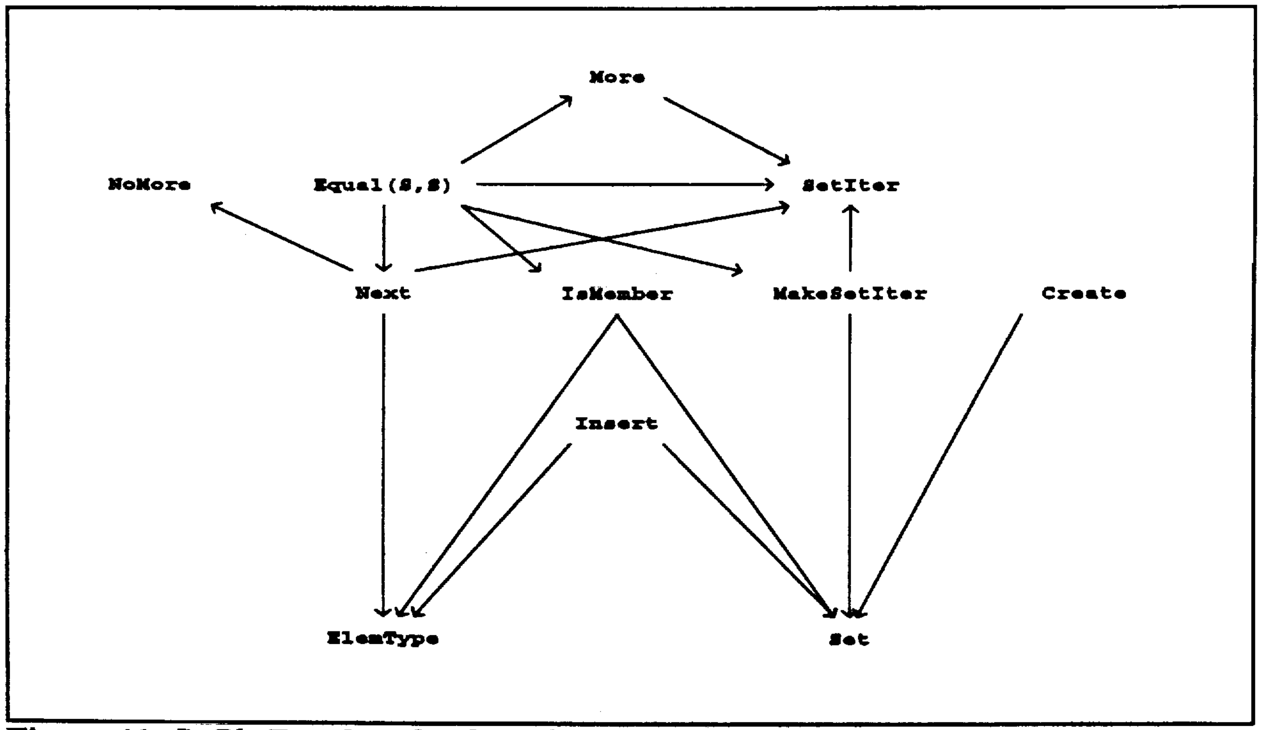

Figure 11 *SetPkgTemplate* sliced on ⟨*Create, Insert, Equal, Set, ElemType*⟩

3.9 Name Overloading

In the previous section, to slice the Ada package *SetPkgTemplate,* we used the slicing criterion ⟨*Create, Insert, Equal, Set, ElemType*⟩. Giving this exact slicing criterion to an automatic slicing tool won't work, however, because the name *Equal* is overloaded in the package. Referring to the package specification in Appendix A, *Equal* appears on line 4 with the type signature

$$\textit{function: ElemType} \times \textit{ElemType} \rightarrow \textit{boolean},$$

while on line 21 it appears with the type signature

$$\textit{function: Set} \times \textit{Set} \rightarrow \textit{boolean}.$$

Note that the interface dependence graph in Figure 10 has two separate nodes for the two occurrences of the overloaded name *Equal* (in the figure, *Set* and *ElemType* as parameter typemarks are abbreviated as *S* and *ET*, respectively). Since *Equal* is overloaded, the slicing criterion listed above is ambiguous, and needs amplification with the argument type signature of *Equal*. An unambiguous slicing criterion might appear as:

⟨*Create():Set, Insert(Set,ElemType), Equal(Set,Set):boolean, type Set, type ElemType*⟩

Henceforth we will use the shortest form which is unambiguous, e.g.:

⟨*Create, Insert, Equal(Set,Set), Set, ElemType*⟩

3.10 Syntactic Sugar

Up to this point in the discussion of interface slicing, none of the concepts presented have been language-specific. While we are using Ada as the language of the examples, the concepts are applicable to any language such as C++ or ML which includes such features as separate compilation and specifications with type signatures. But while the concepts are language-independent, a working interface slicing tool cannot be. This is because each language has its own structure and syntax which must be respected, else the output of the slicer will be syntactically incorrect and therefore useless.

Figure 5 (Page 10) lists a complete Ada package, while Figure 6 (Page 11) lists a slice of that package. Lines 1, 6, 7, and 25 of Figure 5 appear in Figure 6, even though they are not part of the transitive closure of the interface dependence graph of *toggle1* upon which the slice of Figure 6 is based. As with conventional slicers discussed in previous sections, the interface slicer must keep track of the language syntax when generating the slice.

A slightly more difficult example of this occurs in *SetPkgTemplate*. Line 38 of that specification (in Appendix A) is:

```
package ListPkg is new ListPkgTemplate(ElemType, Equal);
```

The purpose of this line is to instantiate the generic package *ListPkgTemplate* by supplying it with the identifier *ListPkg* and the two parameters *ElemType* and *Equal*. As we have described it so far, the interface slicer is a pre-compilation text transforming tool. As such, it does not have the information of the required typemark for the object indicated by *Equal* in this context. By examining the specification of *ListPkgTemplate*, in Appendix B, we can see that the proper generic parameter must be the function *Equal* which is defined on line 4 of *SetPkgTemplate*. But the interface slicer does not know this, as it is only considering *SetPkgTemplate* and does not have access to *ListPkgTemplate*. (We will discuss *ListPkgTemplate* in the following sections.) Since the name *Equal* is overloaded, the slicer doesn't know to which *Equal* this line refers.

The interface slicer must include line 38 of *SetPkgTemplate* in the slice, as this is syntactic sugar which it has no reason to slice out. But if it includes this line, which has the name *Equal* in it, the slicer must also include the definition of *Equal*. Since *Equal* here is ambiguous, the slicer must therefore include all definitions of *Equal*, to be sure of including the correct one. The two definitions of *Equal* occur on lines 4 and 21 of *SetPkgTemplate*.

Figure 11 (Page 18) shows the results of the reachability analysis of the interface dependence graph of *SetPkgTemplate*. By this analysis, the interface slicer will include the definition of *Equal* which occurs on line 21, as indicated by the appearance of the node *Equal (Set, Set)* in the graph. However, node *Equal (ElemType, ElemType)* corresponding to the definition of *Equal* on line 4, does not appear in the graph. This is because the interface slicing algorithm gives no reason to include that node, as nothing in the slicing criterion depends upon *Equal (ElemType, ElemType)* in *SetPkgTemplate*. In summary, line 21 is included in the slice because of the transitive closure of the interface dependence graph, and line 4, while not included in the sliced graph, is included in the final text of the slice because of syntactic analysis. This difficulty is due to the fact that we are describing the slicer as though it were considering *SetPkgTemplate* in isolation. In fact, it is most reasonable to consider that a production interface slicer would be implemented as a module of an integrated development environment, with full access to the program databases and libraries of that environment. This would considerably reduce some of the difficulties described here. To give the flavor of this, Section 3.12 will extend consideration to *ListPkgTemplate* proper.

3.11 Number of Generic Parameters

A procedure which instantiates *SetPkgTemplate* has to supply an element type and an equality function. In the previous examples, the number of these parameters did not change due to interface slicing. However, it is easy to produce an example in which interface slicing eliminates all references to a generic parameter and renders it unnecessary. The elimination of unnecessary parameters increases the usefulness of interface slicing in reducing size and complexity of reused packages.

Consider, for example, a procedure which instantiates *SetPkgTemplate* and slices it on ⟨*Create, Insert, Intersect, Size, Set, ElemType*⟩. The interface dependence graph of the set package sliced on this criterion is shown in Figure 12 (cf. Figure 10 on Page 18, and Figure 11). Notice that the generic function parameter *Equal (ElemType, ElemType)*, node S4, does not appear in the slice. This is because none of the operations in the slicing criterion either directly or transitively reference the equality function. This means that based strictly on a consideration of *SetPkgTemplate* sliced with this criterion, the generic parameter *Equal* is unnecessary and can be omitted.

A generic parameter cannot simply be omitted from a standard non-defaulted Ada package instantiation, however. For a procedure to instantiate *SetPkgTemplate*, the compiler expects a statement such as:

```
package SetPkg is new SetPkg-
Template (MyElementType, My-
Equal);
```

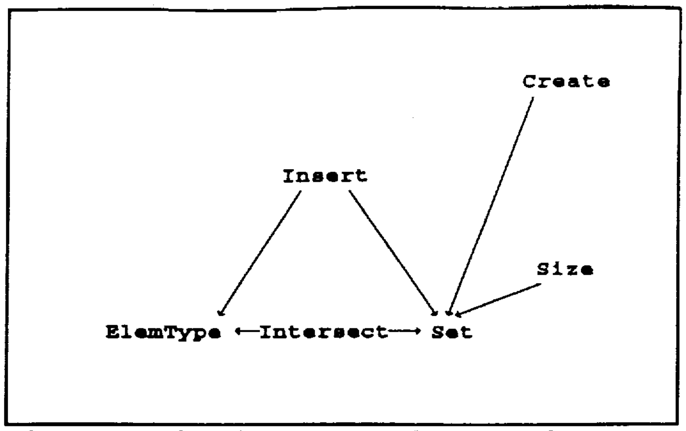

Figure 12 *SetPkgTemplate* sliced on ⟨*Create, Insert, Intersect, Size, Set, ElemType*⟩

If the *MyEqual* is dropped from the line above, the compiler will generate an error message complaining about a missing generic subprogram argument. While defaulting generic parameters will help in some situations, in other cases there is no appropriate default and attempting to include an inappropriate default parameter merely to enable this feature of interface slicing would lead to serious programmatic as well as stylistic errors.

In general, given a generic package the instantiation of which originally took the form:

```
package APkg is new APkgTemplate (p1,...,pn);
```

in which $p_1,...,p_n$ represent generic parameters, after interface slicing we wish to instantiate the package with some number of parameters removed by the interface slicing process.

One approach to reconciling mismatched parameters has been advanced by Purtilo and Atlee [20], who have developed the module interconnection language *Nimble*. Nimble was designed to automatically adapt module interfaces which have large discrepancies in the types and sizes of their parameters; merely reconciling their number, as required here, is an easy job for Nimble. We therefore assume an automatic interface slicer would be implemented with some mechanism perhaps similar to Nimble for reconciling mismatched numbers of package parameters.

3.12 A Second Level of Slicing

While the slice in the example above represents a considerable reduction in the size of the set package, a much greater overall savings can be realized if the slicing process is extended to the list package upon which the set package is based. In addition, if the language in use allows a direct named reference to program elements in other modules, as was the case with the ambiguous reference to *Equal* discussed above, then the most precise slice of one module may require information from other modules.

Just as the main program in the example above used functionality provided by the set package, so the set package needs to use functionality provided by the list package. But just as the main program above did not need all of the functionality of the set package but only a subset, so too does the set package need only a subset of the functionality of the list package. That subset, or slice, is based, as above, on the transitive closure of the slicing criterion of visible program elements exported by the list package which the set package directly references. It does not matter which program elements the original unsliced set package references. All that matters is which program elements the set package references *after* being sliced.

The interface dependence graph of *ListPkgTemplate* is shown in Figure 13. In this figure, for legibility, program element names are replaced with the line numbers of the package as listed in Appendix B. In the case of *SetPkgTemplate* sliced on ⟨*Create, Insert, Equal (Set, Set), Set, ElemType*⟩, the references from the new, sliced set package to the list package consist of: *List, EmptyList, Attach (ItemType, List): List, Create, DeleteItems, FirstValue, IsEmpty,* and *IsInList*. These appear on lines 5, 8, 31, 47, 56, 63, 66, and 69 of Appendix B, and are represented by the nodes of the same numbers in Figure 13. These list program elements therefore exactly constitute the slicing criterion on which to slice the list package, based on the original desire to employ the set package elements *Create, Insert, Equal(Set,Set), Set,* and *ElemType*. The elements in *ListPkgTemplate* which are referenced from *SetPkgTemplate* must be visible elements, appropriate for inclusion in a slicing criterion, lest they could not be referenced by *SetPkgTemplate* in the first place.

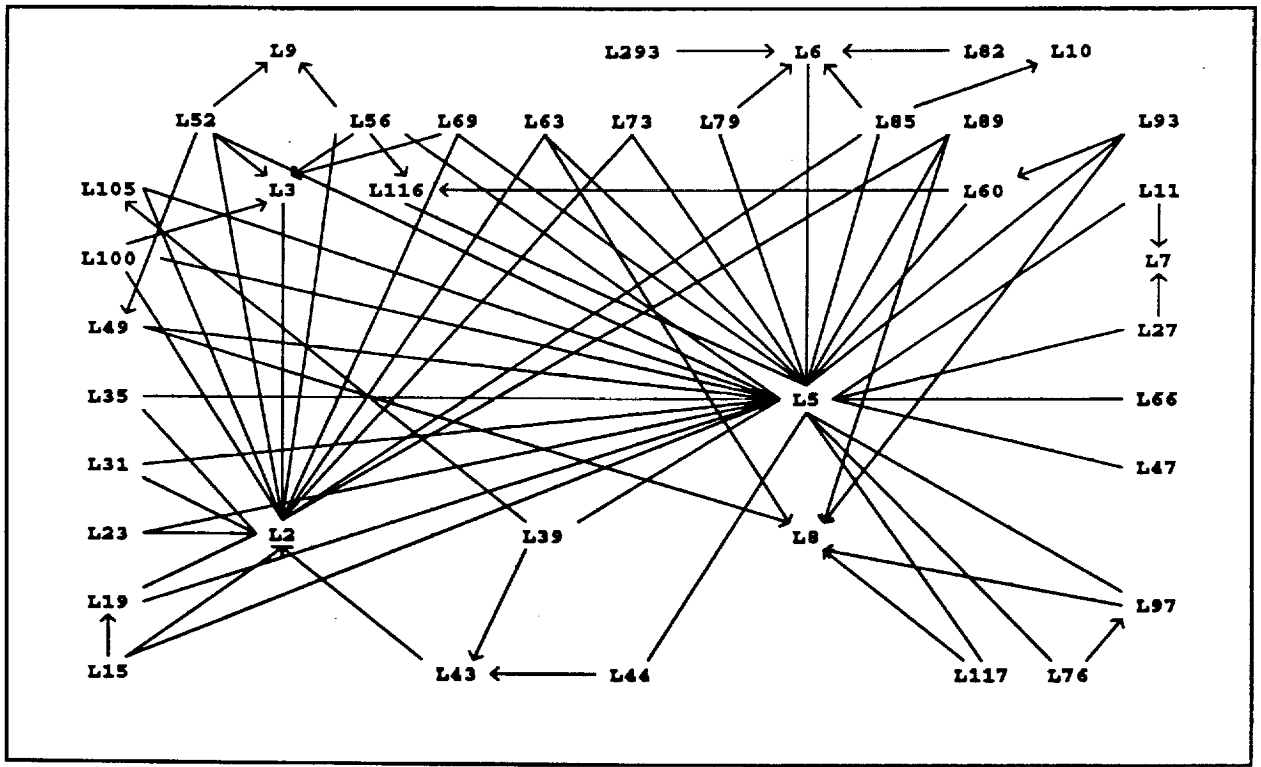

Figure 13 Interface dependence graph for *ListPkgTemplate*

The interface dependence graph of the original *ListPkgTemplate* is shown in Figure 13. This package is large and complex enough to make manual editing a decidedly non-trivial task requiring a major comprehension effort. However, slicing it using the criterion ⟨*List, EmptyList, Attach (ItemType, List): List, Create, DeleteItems, FirstValue, IsEmpty, IsInList*⟩ produces the much smaller graph shown in Figure 14. In this case, in fact, not only is the resulting list package much smaller, but it also has a less complex interface dependence graph. There is a correspondingly large reduction in overall size and complexity of the source code which the slicer produces not only as output for the compiler but equally importantly for the software engineer charged with development or maintenance.

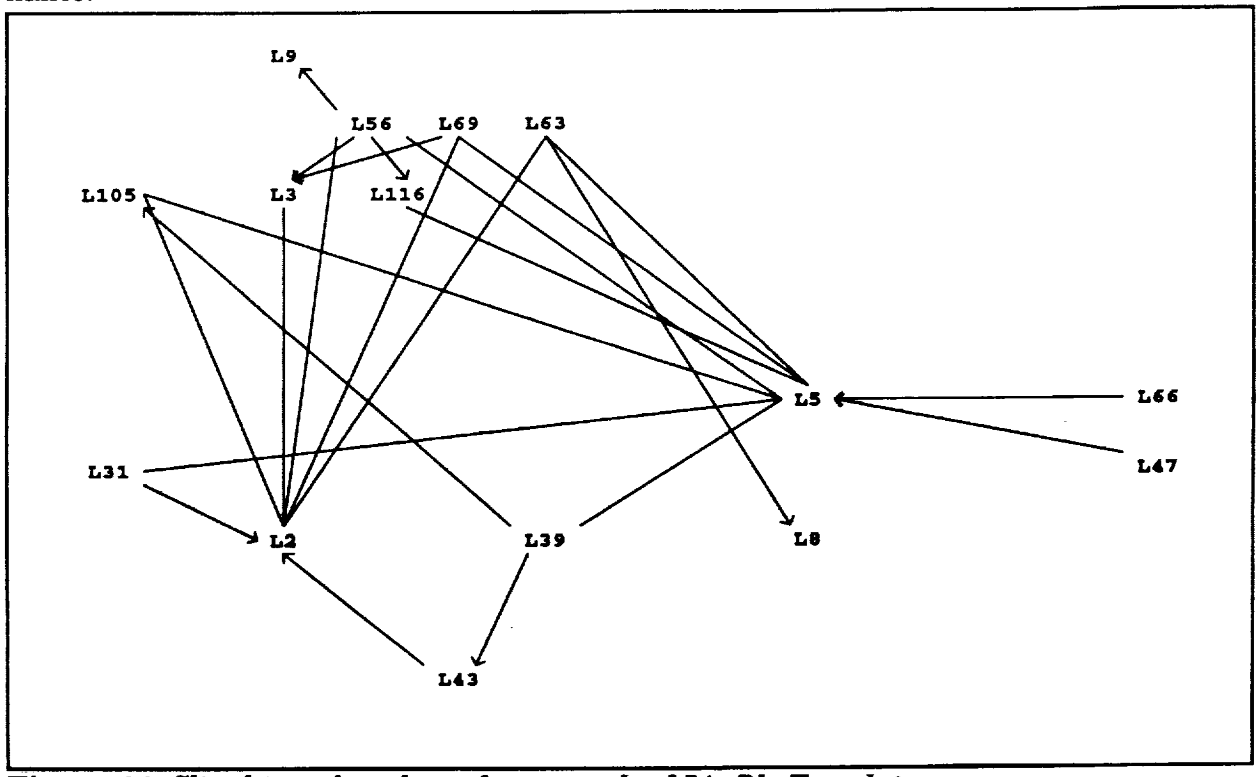

Figure 14 Sliced interface dependence graph of *ListPkgTemplate*

Slicing need not stop at two levels, of course, but can be continued to the maximum number of levels of packages in a given software system. Weide, et al., imply that in a mature reusable-component development environment, the number of levels in the component composition hierarchy may be quite large [24]. While a comparison of Figure 13 with Figure 14 gives an indication of the effect of interface slicing, we list in Table 1 the actual change in size of the interface dependence graphs and of the source and two examples of executable code for the original and sliced set and list packages in the example above. The driver program was minimal in size while still making references to every *Set* entity in the slicing criterion. We generated an executable on several different platforms, and list two representative ones in the table: VAX Ada for VMS 5.5-2, and Meridian Ada 4.1.3 for Sun-4 Unix. The numbers in the table indicate that slicing reduced the size of the set and list component source code by more than half, and reduces the size of a test driver program's executable by up to 45%. In other words, at least 17%, and up to 45%, of the executable for the simple unsliced example program produced by standard commercial compilers is dead code. While numerical results from a larger sample size of larger programs will have to await the completion of an automatic interface slicer implementation, based on this example of source code and dead code reduction alone, the interface slicer can help reduce the size and complexity of systems, and thus to ease the comprehension problems in software maintenance.

	# IDG nodes	# IDG edges	# source lines	executable bytes VMS	executable bytes Unix
Full Set+List	19+39=58	29+69=98	151+393=544	20000	49152
Sliced Set+List	11+15=26	18+22=40	84+117=201	11000	40960
% Reduction	55%	59%	63%	45%	17%

Table 1 Size reductions in IDG and code of *SetPkgTemplate* and *ListPkgTemplate*

Because conventional slicing analyzes programs at the statement level, we term conventional slicing a microanalysis technique. Current conventional slicing techniques depend upon program dependence graph representations, the generation of which can be expensive in time and space requirements. Thus there is some question about the practicality of applying conventional slicing to very large programs. In contrast, we consider interface slicing to be a macroanalysis technique in that it deals with programs at the package level with no larger time or space requirements than a compiler requires, and so are appropriate for application to even very large programs. Indeed, it is possible that the size reduction of source code of a package after interface slicing will be sufficient to allow conventional slicing to be performed on that package.

3.13 Dynamic Interface Slicing

So far, we have presented interface slicing only from the perspective of a static analysis, but it is possible to imagine dynamic interface slicing. Consider the system in Figure 15. A strict static analysis, as described so far, would conclude that package *M* references both packages *A* and *B*, and that any system which includes *M* must also include both *A* and *B*. But if the value of the predicate *X* were known, using the techniques of dynamic slicing, then it would be possible to conclude that either *A* or *B*, but not both, were necessary for inclusion. However, given the

```
package A is                 with A, B;
  ...                        package M is
  procedure F;                 ...
  ...                        begin
end A;                         ...
                               if X then
package B is                     A.F;
  ...                          else
  procedure G;                   B.G;
  ...                          end if;
end B;                       end M;
```

Figure 15 Example for dynamic interface slicing

applications currently envisioned for interface slicing, and the techniques currently used to generate interface slicing as presented here, we do not foresee substantial practical use for dynamic interface slicing. Indeed, it is not clear that many systems even contain constructs such as shown in Figure 15, which would be amenable to dynamic interface slicing.

4 Posets and Lattices of Slices

4.1 Decomposition Slice Poset

Gallagher and Lyle [13] use conventional static slicing as the basis of a total program decomposition. The units of this program decomposition they term decomposition slices, and they arrange these decomposition slices into a poset, the elements of which each consist of a subset of the program statements.[9] Their intent is to ascertain the limits past which a specific program modification cannot reach, enabling software maintainers to focus on the region of the program which their change does affect, secure in the knowledge that the change can have no linkages to — and thus can introduce no bugs in — other parts of the program. Gallagher and Lyle accomplish this by using the poset of decomposition slices to establish dependence and independence among program statements.

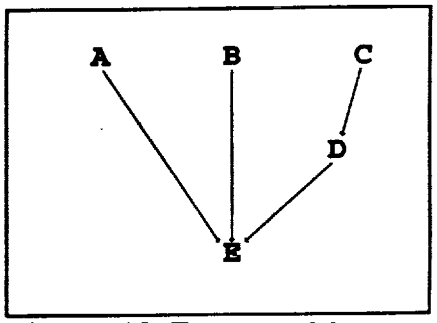

Figure 16 Decomposition slice poset

Figure 16 shows an example decomposition slice poset. The meaning of this figure is that the five nodes represent the decomposition slices of a program, each using one of the program's five variables as the slicing criterion. The set of statements in the slice *E* is wholly contained in each of the other three slices; the set of statements in *D* is wholly contained in *C*; *D* contains at least one statement not in *E*. If the poset included a greatest element (which would correspond to the original undecomposed program), it would be a lattice; rather, it has a set of maximal elements and is thus a poset. Since the poset represents a total program decomposition, the union of the three maximal slices yields the original program (i.e, every program statement is in at least one maximal slice).[10] Using the structured relationship of maximal and interior slices, Gallagher and Lyle proceed to

9 In the referenced paper, Gallagher and Lyle term this poset a lattice, but it is not a lattice as presented. There is no upper bound of any pair of the maximal elements in their structure, and thus the structure is not a lattice. Their conclusions about the relationships among the elements of the poset remain valid, however.

10 Gallagher and Lyle fail to point out that if the poset produced by their decomposition is in fact a lattice, that is, if there is a single maximal element, then there are no independent slices in the program. In this case, their procedure results in no improvement for the maintenance process. A determination of how frequently this is the case must await a real slicer.

characterize every program statement and variable as dependent or independent. With the statements and variables so characterized, Gallagher and Lyle give precise rules which state the kinds of program modifications which can be performed while still guaranteeing that the modifications have no effect outside the maximal slice where the modifications take place. The specific rules which they present are not of interest in this discussion. What is of interest is their concept of total program decomposition yielding a poset arrangement of slices, the relationship of which allows specific characterizations of program elements, and thus new conclusions about those program elements.

4.2 A Lattice Construction Algorithm

Before we discuss interface slice lattices in the next section, we first take the time here to discuss the construction of lattices in general. Because of the paucity of published lattice algorithms, and of the desirability to experiment with a number of lattices during the course of this investigation, we developed a lattice construction algorithm. While it was not our intent to necessarily develop the most efficient algorithm possible, we believe that, for an unordered set of nodes about which no prior knowledge is available, no algorithm can substantially improve upon this one for time requirements. In a production implementation, however, more space-efficient techniques such as those of Aït-Kaci, et al., which use bit-vector representations of sets and which thereby accomplish very space-efficient lattice representations might be more appropriate [1]. Although the researchers of that paper do not present a lattice construction algorithm, once the lattice is built using our algorithm, their techniques could then be used for representation and manipulation purposes. Rather, our intent was to develop an algorithm which clearly and explicitly built the lattice structure in order to 1) aid in visualizing the relationships among the lattice elements, 2) provide a basis for reasoning about the program semantics inferred from those relationships, and 3) provide a vehicle for experimenting with those relationships. With appropriate small modifications, the same algorithm can be used to build a graphical representation of a poset or a semilattice as well as a lattice.

The algorithm which we developed for constructing the lattice of interface slices is listed in Figure 17.[11] Recall that each lattice element is a set of program statements, and that each element has associated with it a set of parents and a set of children which are other lattice elements. This algorithm inserts a new set into the lattice by identifying the set of parents and the set of children which the new set will have, and by removing parent and child relations which would become transitive ancestor and descendant relations once the new set is inserted. The algorithm listing here assumes that the set element type is a code statement and that the poset partial ordering relation is subset. The algorithm assumes an implementation in which each node's set of parents and children is available

11 We implemented this algorithm in approximately 600 sloc of ANSI C. It is interesting to note that an almost identical algorithm was implemented by Atkins [4] in the Opal programming language of Servio Corporation's GemStone Object-Oriented Database Management System using only 25 sloc.

```
/* Find the proper place for NewSet in the lattice */

/* Find the set of parents for the new node */

SetOfParentsForNewSet ← ∅
SetOfPotentialParents ← {Univ}
while SetOfPotentialParents ≠ ∅
   Anode ← a node removed from SetOfPotentialParents
   foreach Achild in Anode's set of children do
     if NewSet = Achild
       return (NewSet is already in the lattice)
     if NewSet ⊆ Achild
       add Achild to SetOfPotentialParents
   if no Achild was added to SetOfPotentialParents
     add Anode to SetOfParentsForNewSet

/* Similarly find the set of children for the new node */

SetOfChildrenForNewSet ← ∅
SetOfPotentialChildren ← {Void}
while SetOfPotentialChildren ≠ ∅
   Anode ← a node removed from SetOfPotentialChildren
   foreach Aparent in Anode's set of parents do
     if Aparent ⊆ NewSet
       add Aparent to SetOfPotentialChildren
   if no Aparent was added to SetOfPotentialChildren
     add Anode to SetOfChildrenForNewSet

/* Remove would-be transitive links between all nodes  */
/* in SetOfParentsForNewSet and SetOfChildrenForNewSet */

foreach Pnode in SetOfParentsForNewSet do
   foreach Cnode in SetOfChildrenForNewSet do
     foreach Child in Pnode's set of children do
       if Child = Cnode
         remove Pnode from Cnode's set of parents
         remove Cnode from Pnode's set of children

/* Insert the NewSet into the lattice */

NewSet's children ← SetOfChildrenForNewSet
NewSet's parents ← SetOfParentsForNewSet
```

Figure 17 A lattice construction algorithm

for inspection and modification.[12] Further, we stipulate the original existence of two lattice nodes, a greatest node *Univ* which contains the set of all program statements, and a least node *Void* which contains the empty set. Initially, *Univ*'s only child is *Void*, and *Void*'s only parent is *Univ*. *Univ*'s set of parents and *Void*'s set of children are both invariantly empty.

In the discussion of the algorithm which follows, we consider *Univ* to be at the "top" of the lattice and *Void* to be at the "bottom," and so all directed edges in the diagrams point downwards. We use the terms *node* and *set* synonymously to refer to a lattice element. The algorithm mainly consists of a pair of walks of the lattice, one starting at the top and working down (lines 3-15) and the other starting at the bottom and working up (lines 17-27). While this algorithm cannot be implemented with a strictly depth-first search strategy, whether the walks are strictly breadth-first or of a hybrid nature depends upon whether the sets *SetOfPotentialParents* and *SetOfPotentialChildren* in lines 6 and 20 are implemented as queues or stacks respectively. We will describe the top-down walk in detail; the bottom-up walk is essentially the same with the roles of parents and children reversed.

The purpose of the top-down walk in the first section of the algorithm is to identify the nodes which are to become parents of the new set to be inserted into the lattice. The desired result of the walk is to identify all nodes a such that

1 $a \subseteq NewSet$, and

2 there does not exist a node c such that $a \subseteq c \subseteq NewSet$.

At the start of the walk, line 6, *Univ* is the only element in *SetOfPotentialParents*, a set of nodes which always satisfies 1, and for which 2 is true to the extent that the algorithm has determined so far. The walk is implemented as a *while* loop, lines 7-15, which continues as long as there are still nodes in *SetOfPotentialParents* to consider. The body of the loop consists of removing a node from the *SetOfPotentialParents* and examining each of its children in relation to the *NewSet*. If any of the children is the same as *NewSet* (lines 10-11), meaning that *NewSet* is already in the lattice, the algorithm terminates immediately, as the definition of the lattice states that no two nodes are the same. However, if *NewSet* is a subset of a child, that child is added to the *SetOfPotentialParents*, as *NewSet* by definition must be a closest subset of each of its parents. At the end of each iteration of the *while* loop (lines 14-15), all of the children of the node have been considered. If none of them was a closer superset of *NewSet* than the node itself, then the node by definition must be a parent of *NewSet*, and so is added to the *SetOfParentsForNewSet*. This discussion assumes that *SetOfPotentialParents* and *SetOfParentsForNewSet* are implemented as sets at least in the sense that duplicate elements are excluded. Otherwise, the same

12 The sets of parents and children in the lattice contain redundant information: if x is a child of y, then y is a parent of x. Maintaining both sets makes the logic of the algorithm much easier to follow, and makes the representation more explicit, at the expense of the space and effort to store redundant information.

element could appear multiple times in the *SetOfParentsForNewSet*, which violates the assumption that if the edge $a \rightarrow b$ exists in the lattice, it is unique.

In the second section, the bottom-up walk in lines 19-27 uses the same principles to find the *SetOfChildrenForNewSet*. The only notable difference is that the test for the prior existence of *NewSet* in the lattice in lines 10-11 is not repeated in the bottom-up walk, as the top-down walk is guaranteed to have found all such duplicates in the lattice.

At the conclusion of the two walks, all necessary information for inserting the *NewSet* into its proper place in the lattice has been obtained and is stored in the sets *SetOfParentsForNewSet* and *SetOfChildrenForNewSet*. This insertion is performed in the fourth section of the algorithm, lines 39-42. The third section of the algorithm, lines 29-37, is necessary to maintain the consistency of the lattice structure. While transitivity is an inherent feature of the lattice (i.e., if $a \subseteq b \subseteq c$ then $a \subseteq c$), transitive relations are not explicitly represented in the lattice. In fact, the definition of the lattice forbids the representation of transitive relations. If *Univ* and *Void* are distinguished nodes whose contents are never modified (as in this algorithm), then every node added to the lattice would result in an inconsistent set of children for every parent of the new node, and an inconsistent set of parents for every child of the new node. To see this, consider that every pair of elements in the lattice $a \subseteq b$ such that inserting node c results in $a \subseteq c \subseteq b$, a must no longer have b as one of its children, and b can no longer have a as one of its parents. Rather, b becomes a descendant of a, but the descendant relationship is not explicitly represented in the lattice structure. This is illustrated in Figure 18.

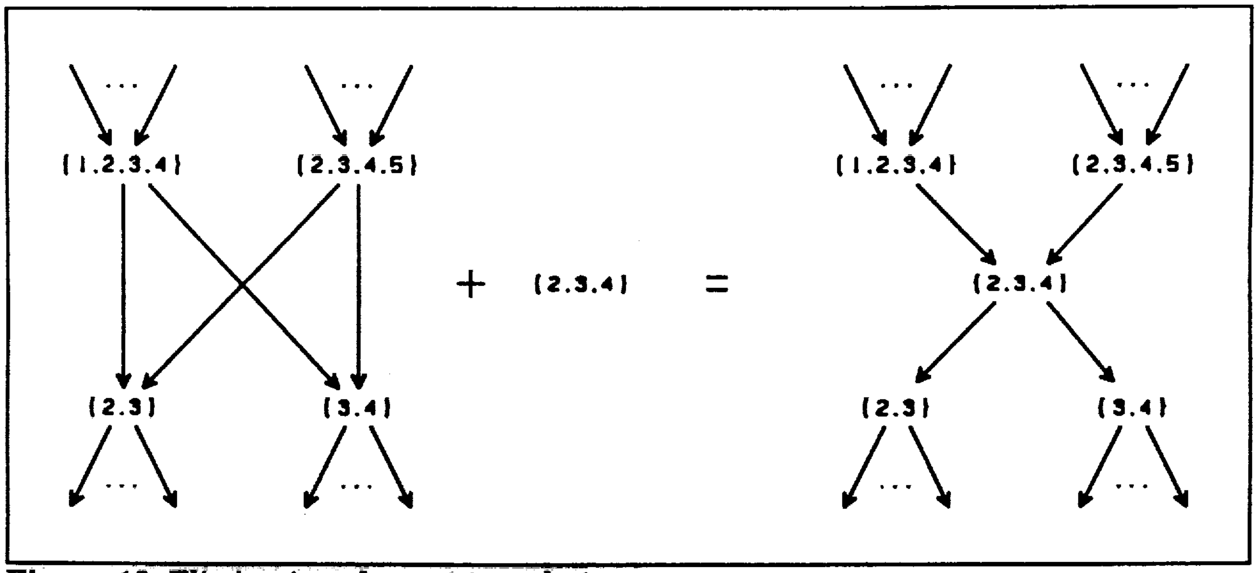

Figure 18 Elimination of transitive relations

Therefore, in the third section of the algorithm, it is necessary to do an exhaustive pairwise comparison between every element in the *SetOfParentsForNewSet* and the *SetOfChildrenForNewSet*. If any child of a member of the former is a member of the latter, that parent-child relationship is dissolved (lines 36-37).

4.3 Interface Slice Lattice

Interface slices by themselves are interesting and useful, but when an interface slice is presented as a standalone artifact it is difficult to understand its relationships to other interface slices and to the original package. Also, as presented, each interface slice must be computed afresh from the interface dependence graph as a transitive closure graph operation. Might there be a less expensive method of generating interface slices? In Section 3 we specifically mentioned two slices of the generic Ada package *SetPkgTemplate*. These two slices were generated by the slicing criteria ⟨*Create, Insert, Equal (Set, Set), Set, ElemType*⟩ and ⟨*Create, Insert, Intersect, Size, Set, ElemType*⟩, and are shown in Figure 11 (Page 18) and Figure 12 (Page 21), respectively. The two slices are patently different. Are they related? If so, what is their relationship? Do other interface slices of the same package exist? If so, how many are there, and what can be said about their relationships to other slices and to the original program?

In an attempt to answer these questions, we will discuss the arrangement of interface slices into a lattice built using the technique of the previous section. Since all interface slices, like all program slices generally, consist solely of subsets of the original set of statements, they can be arranged into a lattice based on the poset of the original program elements and the subset operation. More formally, however, we make the following observation.

Proposition 1 Interface Slices Form a Lattice. Given a set S of program elements of a module and the subset relation $\subseteq$, and the set $T \subseteq 2^S$ which is the set of interface slices of the module, then the poset $(T,\subseteq)$ is a lattice.
Proof: By definition, the restriction of $\subseteq$ to T is a partial ordering. By definition, the set of interface slices of a module includes S, corresponding to the unsliced program, and $\varnothing$, corresponding to the slice generated by the empty slicing criterion. But those two elements are exactly the greatest and least elements of the lattice $\{2^S,\subset,\cap\}$. Therefore, $(T,\subseteq)$ is a lattice. □

SetPkgTemplate has 19 visible elements. They are listed in Table 2, along with their line number designations and their respective transitive closures. The second and third columns of the table, taken together, constitute a bag of sets, one set per row. Each of these sets can be considered an interface slice of the package whose slicing criterion is the singleton listed in the second column of the table. The sets in this table form a poset, from which a lattice can be constructed using the algorithm described above. The algorithm adds the empty and universe sets, thus guaranteeing that the poset is a lattice. A Hasse diagram of the lattice (with the universe set omitted for clarity) is shown in Figure 19. (The arrows on the directed edges in this figure are also omitted for clarity; all of the edges point downward.) The figure shows that nodes corresponding to certain package elements (e.g., *ElemType*) appear with cardinality 1. This corresponds to the fact that the Ada type *ElemType* can be exported by *SetPkgTemplate* alone, without any dependence on other package elements. Other package elements do not appear as singletons in the graph (e.g., *Intersect* appears only in combination with *ElementType* and *Set*). This corresponds to the fact that it is not possible to manually edit *SetPkgTemplate* to retain the procedure *Intersect*, while eliminating *ElementType* and *Set*, and still have a compilable package. A slice on the criterion ⟨*Intersect*⟩ will include the code defining *ElementType* and *Set*.

Name	Line	Transitive Closure
ElemType	S3	
Equal(ElemType,ElemType)	S4	S3
Set	S6	
NoMore	S7	
SetIter	S8	
Create	S9	S6
Insert	S11	S3,S6
Delete	S13	S3,S6
Intersect	S15	S3,S6
Union	S17	S6
Copy	S19	S6
Equal(Set,Set)	S21	S3,S6,S7,S8,S25,S30,S32,S34
IsEmpty	S23	S6
IsMember	S25	S3,S6
Size	S28	S6
MakeSetIter	S30	S6,S8
More	S32	S8
Next	S34	S3,S7
Destroy	S36	S6

Table 2 Visible elements of *SetPkgTemplate*

This is not to claim that slicing *SetPkgTemplate* on ⟨*ElemType*⟩ necessarily produces a useful slice. In general, the minimum functionality set for a useful ADT must include at least one constructor function and one observer function. An interface slicer could easily flag a slicing criterion, either manually or automatically generated, which fails to meet minimum "usefulness" standards, or other measures of orthodoxy, indicating likely design flaws. Figure 19 contains a number of slices which do contain at least a constructor and an observer, and which provide quite useful collections of functionalities.

Proposition 2 Smaller Subsets Do Not Exist. No non-empty interface slice can exist which is a subset of the parents of ∅ in the interface slice lattice.
Proof: The construction of the slices in columns 2 and 3 of Table 2 originated with singleton visible package elements and included only their transitive closures. Since the transitive closure of a directed graph node is unique and the interface slice construction algorithm is deterministic, no smaller interface slice based on singleton elements can

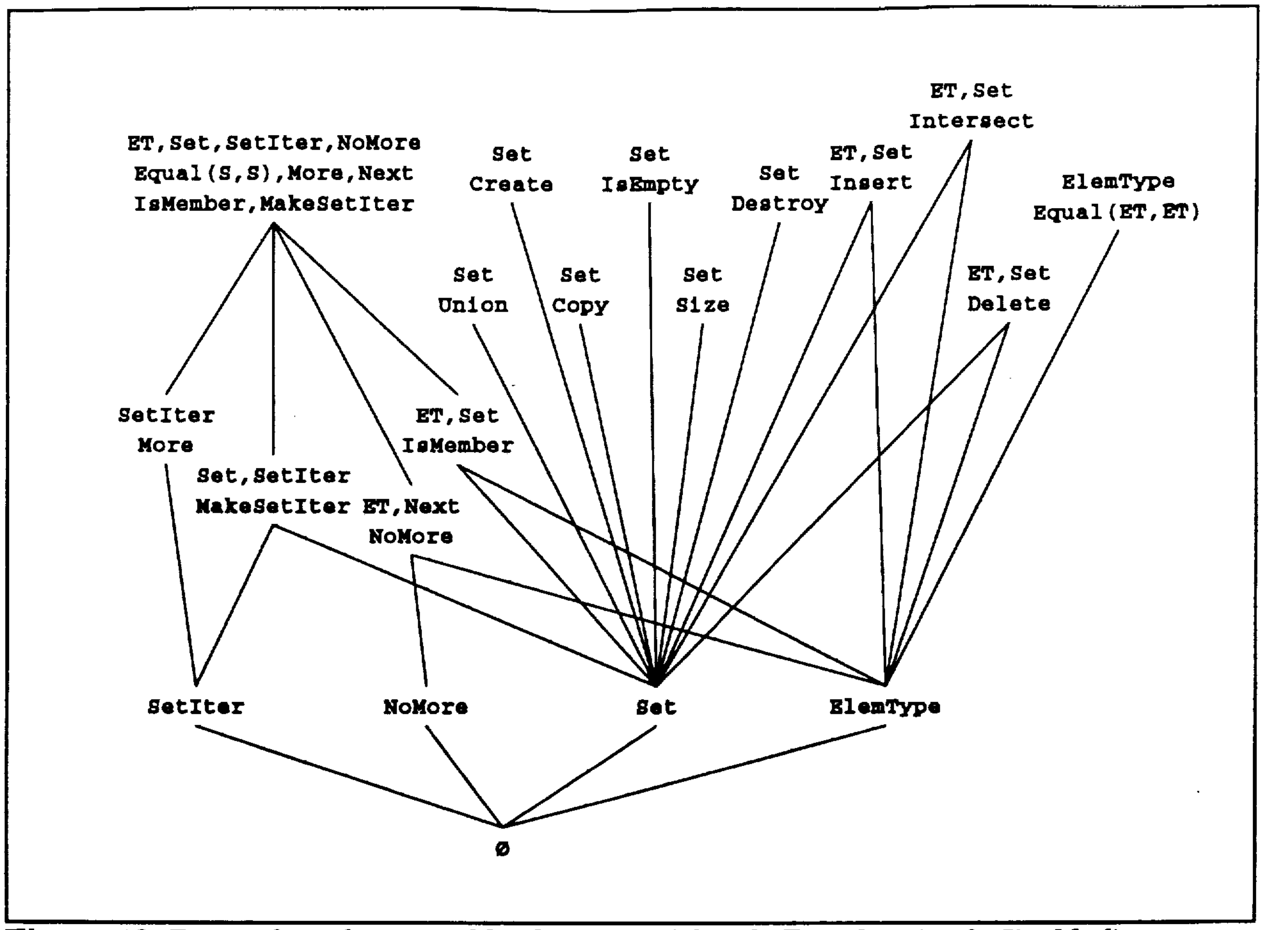

Figure 19 Poset of singleton visible elements of *SetPkgTemplate* (with Ø added)

exist. Since all non-singleton interface slices are equivalent to the set union of the singleton slices, and set union is an operation of monotonically increasing cardinality, no smaller slice can be generated. □

Definition 5 Strict Independence. For the purposes of interface slicing, a module element is strictly independent if it appears in a slice which has Ø as its only child. A module element is strictly dependent otherwise. □

In lattice terms, the only descendant of the node containing *ElemType* is Ø and so *ElemType* is independent, implying in package terms that *ElemType* does not depend on any other package element. The element *Intersect* appears only in nodes having non-Ø nodes as children, implying that *Intersect* depends on other package elements. A generalization of this observation leads to the following:

Proposition 3 Nodes Without Children. If the only child of a slice lattice node *A* is Ø, that is, *A* is independent, then every element which appears in *A* has no dependences upon any element not in *A*.

Proof: Let A have an element i. Assume that i is dependent upon an element j in node B. Then, by the construction of the interface dependence graph, there will be an edge from A to B, i.e., B is a child of A. But this violates the assumption that the only child of A is $\varnothing$□

Proposition 4 Nodes With Children. At least one package element which appears in a node with a non-$\varnothing$ child is a dependent element.
Proof: This is the converse of the previous Proposition. Let node A have elements $i,...,k$. Assume that no element in $i,...,k$ is dependent. Then, by the construction of the interface dependence graph, the only child of A is $\varnothing$. But this violates the assumption that A has a non-$\varnothing$ child. □

Another observation which can be made from Figure 19 is that certain pairs of nodes have a non-$\varnothing$ greatest lower bound (e.g., the g.l.b. of nodes {*Set, Copy*} and {*Set, IsEmpty*} is {*Set*}) while for others the g.l.b. is $\varnothing$ (e.g., the nodes {*Set, Copy*} and {*SetIter, More*}. This corresponds to the fact that the Ada code for the slices ⟨*Set, Copy*⟩ and ⟨*Set, IsEmpty*⟩ have package elements in common, while the code for the slices ⟨*Set, Copy*⟩ and ⟨*SetIter, More*⟩ is disjoint. Recall, however, that the unique g.l.b. for a pair of nodes is not necessarily the same as the set of common lower bounds for the pair of nodes. For example, consider the two nodes {*ElementType, Set, IsMember*} and {*ElementType, Set, Intersect*}. These two nodes have a g.l.b. of $\varnothing$, but they are clearly not disjoint, having *ElementType* and *Set* in common. This is reflected by the fact that the two nodes have a non-empty set of common maximal lower bounds, namely {*ElementType, Set*}; by definition, this set is a cochain. This leads to the following:

Definition 6 Mutual Independence. Two nodes which have a non-empty set of common maximal lower bounds are mutually dependent. Two nodes which have $\varnothing$ as their set of common maximal lower bounds are mutually independent.

These definitions are distinct from the definitions of strict independence and strict dependence given previously. These definitions can be directly extended from pairs to sets of mutually dependent and independent nodes; they will be used below for further results derived from the lattice arrangement.

4.4 Generating All Possible Interface Slices

We turn now to a consideration of all possible interface slices of a module. While it turns out that actually generating all possible interface slices is impractical for a production development environment, as will be shown below, it is instructive to consider the process. After a discussion of a technique for generating all possible interface slices, a process which is very expensive in time and space requirements, we will describe a technique for generating any desired interface slice in linear time.

Proposition 5 Interface Slices Are a Finite Set. Given a set S of program elements of a module, the set of interface slices of that module is finite. The set of interface slices is a subset of a finite set which can be generated.
Proof: Every interface slice of the module is some combination of the finite members of S. Therefore the set of interface slices of the module is finite. The lattice $\{2^S, \subset, \cap\}$ contains all possible combinations of the members of S. Therefore the set of interface slices is a subset of the nodes in the lattice, and the lattice contains all interface slices. Since the

lattice $\{2^S, \subset, \cap\}$ can be generated by a combinatorial brute-force algorithm, it is possible to generate all possible interface slices of the module. □

4.4.1 A Generation Technique

Unlike Weiser slicing in which it is undecidable to generate all possible slices, and unlike Gallagher and Lyle's decomposition slices which vary depending on the slicing technique employed in their construction, this proposition guarantees that it is possible to generate all possible interface slices. Unfortunately, the proposition does not indicate a method for recognizing which of the lattice elements are in fact interface slices, and so does not provide a useful generation algorithm. In addition, the proposition indicates a process which is combinatorial in the number of program elements in the module.

All the information necessary to specifically generate just the interface slices is contained in the interface dependence graph as discussed in Section 3. The graph for *SetPkgTemplate* is shown in Figure 10 (Page 18). Recall that an interface slice can be found by taking the transitive closure of the slicing criterion which is a set of visible package elements in the interface dependence graph. It is obvious that generating all possible interface slices can be accomplished simply by taking the transitive closure of all combinations of visible package elements. Recall that taking the transitive closure of the desired combination of package elements was how we generated the interface slices in Section 3. Unfortunately, however, using this brute-force technique to generate all possible interface slices requires time which is combinatorial in the number of visible elements. Even by taking advantage of all the information in the lattice structure, it is not possible to generate these slices in time less than combinatorial in the number of singleton interface slices.

In fact, in the worst case this method based on lattice construction degrades to the time requirement of the brute-force approach, even though this is improbable in practice. In the worst case, for n visible package elements, there are 2^n interface slices. For this to occur, however, each element in the package would have to be strictly independent of all others. But the whole philosophy of package modularization is to group program elements which are related, and which thus tend strongly to have dependences among them. The more dependences within the set of package elements, the fewer distinct interface slices there are, and the better the time requirements of generating all possible interface slices.

While the combinatorial time required to actually generate all possible interface slices makes the process impractical in a production system, the fact that it is possible to generate all possible interface slices of a package allows us to use the process to aid in determining relationships among the slices and the original package.

The key to generating these interface slices is to consider the poset of Figure 19 (plus the empty set added to the bottom) as the "truncated" lower portion of the lattice of all interface slices. We have already shown that the set of interface slices must form a lattice. The task at hand is to generate the "missing" upper portion of the lattice. More precisely, the poset of Figure 19 is a subset of the complete lattice which we desire to construct, or at least to visualize. Again, in a production system we would expect an efficient bit-vector representation to be used. We have used the explicit lattice representation to aid in the visualization of the process and its results. Before presenting the process, a statement about the union of interface slices is necessary.

Proposition 6 Union of Interface Slices. Let M be a module, $C_1 = \langle e_1, e_2, \ldots, e_j \rangle$ and $C_2 = \langle f_1, f_2, \ldots, f_k \rangle$ be two interface slice criteria, and $S_1 = M(C_1)$ and $S_2 = M(C_2)$, be the two respective interface slices. Also let $C_U = C_1 \cup C_2$ be the union of the two criteria. Then $S_1 \cup S_2 = M(C_U)$.
Proof: By definition,

$$S_1 \cup S_2 = M(C_1) \cup M(C_2)$$
$$S_1 \cup S_2 = \left(M(e_1) \cup M(e_2) \cup \ldots \cup M(e_j)\right) \cup \left(M(f_1) \cup M(f_2) \cup \ldots \cup M(f_k)\right)$$

Because of the commutativity of set union, we have

$$S_1 \cup S_2 = M(e_1) \cup M(e_2) \cup \ldots \cup M(e_j) \cup M(f_1) \cup M(f_2) \cup \ldots \cup M(f_k)$$
$$S_1 \cup S_2 = M(C_U) \qquad \square$$

In words, Proposition 6 states that the union of two interface slices based on two slicing criteria is identical to the interface slice based on the union of the two slicing criteria. The proof states that the interface slice based on C_1 is defined as the union of the transitive closure of each of the individual elements e_i. Since union is commutative, the order in which the transitive closure of each element and its subsequent union into the interface slice is inconsequential, and similarly for the elements f_i for the slice based on C_2. Because of the nature of interface slice construction, therefore, the two expressions $S_1 \cup S_2$ and $M(C_U)$ are computationally equivalent, and thus are identical.

To generate all interface slices, one can use an iterative process which continues until no more slices are generated. The first iteration consists of a pairwise union of every set originally produced by the transitive closure of the singleton visible package elements, the result of each union compared with all existing sets to eliminate duplicate sets. Each subsequent iteration consists of performing a pairwise union of all sets produced in the previous iteration with the original sets, again comparing the results to the existing sets to eliminate duplicates. The new sets which are not duplicates are the product of the current iteration. Each iteration produces a new set of sets, each of which is inserted into the growing lattice of interface slices. This insertion can use any appropriate lattice construction algorithm, such as the one of the previous section. The result of this process is a lattice, the least element of which is the empty set which corresponds to the empty slicing criterion $\langle \rangle$, and the greatest element of which is the set of every package element, which corresponds to the slicing criterion which lists every visible package element, or equivalently, to the unsliced original package. Between these two nodes is every possible interface slice.

4.4.2 An Example Lattice Completion

To illustrate the process of generating all interface slices, we shall use a contrived example which is considerably smaller than *SetPkgTemplate*, which in the very first iteration of the process grows to a size impossible to show in a figure. We shall return to *SetPkgTemplate* in the next section.

Consider the poset shown in Figure 20(a), in which the lowercase letters stand for program elements. (Again in this figure, all the directed edges point down.) The poset contains 5 sets; the least element of the empty set has been added to the diagram. The pairwise union of these 5 sets results in the generation of 7 new sets which are shown with the original 5 in Figure 20(b). The pairwise union of the 7 new sets with the original 5 produces 2 additional sets, as shown in the lattice of Figure 20(c). The pairwise union of these last 2 with the original 5 produces no new sets, and so the iteration process ends.

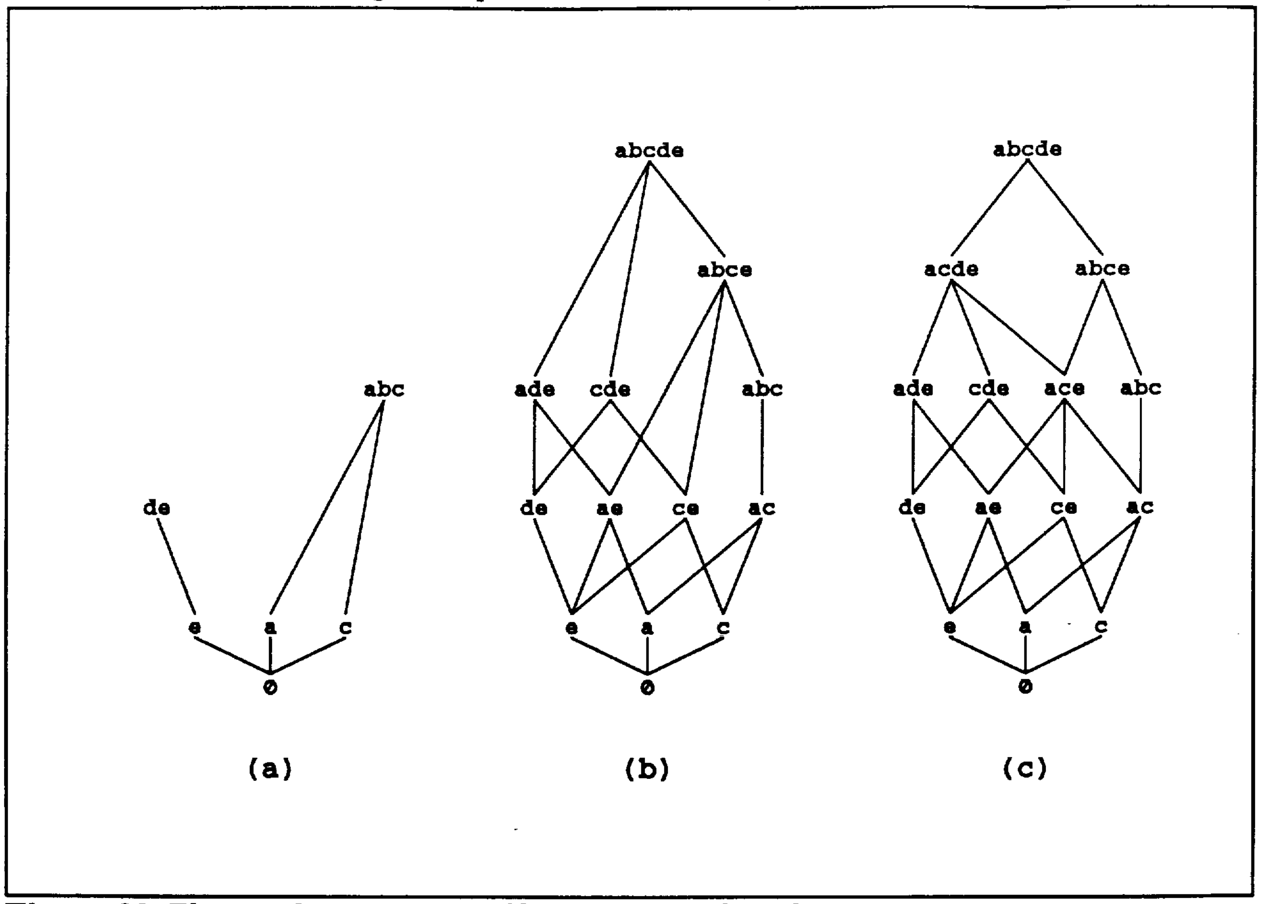

Figure 20 Phases of construction of lattice of interface slices

This process can of course be applied to the poset of *SetPkgTemplate* which is shown in Figure 19, but the results are too large to be displayed here, and dramatically illustrate the impracticality of this generation technique in a real system. The process requires only 14 iterations (compared to the upper bound of 18) and produces 32,407 sets, i.e., the package has 32,407 different interface slices. Thus, even though this number is much smaller than the upper bound of 524,287, it is obvious that generating and storing all possible slices for all packages is an impractical operation for a production development environment.

4.4.3 Counting the Interface Slices

Even though generating and storing all the interface slices of a module may be impractical, it is possible to count the number of interface slices without actually generating them. A dependence between two program elements can be phrased as a *restriction* which eliminates some combinations of program elements from being a valid interface slice. In a module with no dependences, and thus no restrictions on valid combinations of program elements, every possible combination of program elements forms an interface slice. In this case, the number of interface slices is simply $2^{|S|}$, which forms an upper bound on the number of interface slices of a module. (This number includes the original unsliced module and the empty set.) Dependences among program elements in a module reduce the actual number of interface slices from this upper bound.

A dependence can be phrased as: "program element *d* depends on program element *e*," with the meaning that in an interface slice, *d* cannot appear unless *e* also appears. This situation is illustrated in Figure 20(a). This dependence, stated as a restriction, can take the form: "restriction R_1 is a combination of elements including *d* but not including *e*," or more compactly, $R_1 := d \wedge \neg e$. No combination of program elements which satisfies the condition stated in the restriction is an interface slice. Let the set of combinations of program elements which do satisfy R_1 be denoted by A_1. In this case, $A_1 = \{d, ad, bd, cd, abd, acd, bcd, abcd\}$, and $|A_1| = 8$. Note that none of these combinations appear in Figure 20. If this were the only restriction in the module, then the total number of interface slices of the module would be $2^{|S|} - |A_1|$.

In the example in Figure 20, there is a second restriction $R_2 := b \wedge \neg a \wedge \neg c$, corresponding to which is $A_2 = \{b, ab, bc, bd, be, abd, abe, bcd, bce, bde, abde, bcde\}$. Since R_1 and R_2 are the only two restrictions for this module, the set of all interface slices is therefore 2^S - A_1 - A_2. According to the inclusion-exclusion principle of counting [18], the size of this set is $2^{|S|} - |A_1| - |A_2| + |A_1 \cap A_2|$. The Venn diagram in Figure 21 illustrates this number of interface slices. More generally, given module *M*, with *n* program elements, and dependences among those elements expressed in the form of restrictions $R_1 \ldots R_n$, which correspond to excluded combinations of elements $A_1 \ldots A_n$, the number of interface slices is:

$$2^{|S|} - \sum_{i} |A_i| + \sum_{i \neq j} |A_i \cap A_j| - \ldots + (-1)^n \sum_{i \neq j \neq \ldots \neq n} |A_i \cap A_j \cap \ldots \cap A_n|$$

For this example, the value of this expression is $2^5 - 8 - 12 + 3 = 15$, which is exactly the number of nodes shown in Figure 20.

For the interface slices of *SetPkgTemplate*, the restrictions can be read directly from Table 2 (Page 31). The second line of that table would produce the restriction "a combination which includes the element *Equal (ElemType, ElemType)* but does not include the element *ElemType*," or, $S4 \wedge \neg S3$. These restrictions denote sets of elements which do not constitute interface slices, and which allow an expression of the actual number of interface slices to be formulated. We must note, however, that evaluating the expression thus formulated is in general a non-trivial task, and may in some cases be as computationally expensive as forming the interface slices directly, and then counting them, as described in the previous section.

4.5 Any Desired Interface Slice

The fact that there is a proof that every interface slice can be generated guarantees that specific interface slices of interest can also be generated. Generating all possible slices is unnecessary, as any particular interface slice can be generated as needed. In Section 3, we discussed a technique for generating interface slices which was linear in the number of source statements. However, it is possible, after a linear-time scan, and the computation of some sets, to generate any desired interface slice in time which is linear in the number of elements in the slicing criterion. Each slice so generated can be viewed in relation to any other using the lattice as a structuring mechanism.

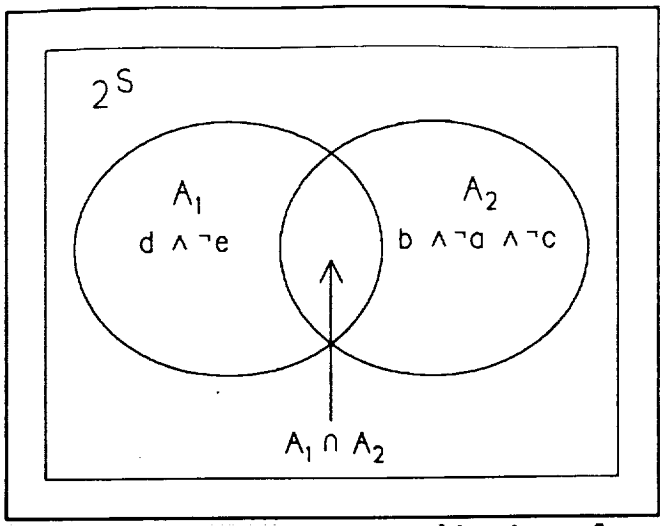

Figure 21 Restrictions on combinations of program elements

The result of Proposition 6 leads to a less expensive technique for generating interface slices than computing the transitive closure of the elements in the slicing criterion. The first step of the technique is to generate and store the transitive closures of all singleton visible elements of the module. This is exactly the information shown in Table 2 (Page 31). Once this information is stored, all that is required to generate an interface slice is to form the union of the elements in the interface criterion and their respective transitive closures as shown in the table. For example, in Section 3, we generated the interface slice of *SetPkgTemplate* on ⟨*Create, Insert, Equal (Set, Set), Set, ElemType*⟩ by taking the transitive closure of the set of elements in the slicing criterion. Depending upon the transitive closure algorithm used, this requires time from $O(n^3)$ to $O(ne)$. Proposition 6 asserts that instead, we can generate the interface slice by taking the union of the transitive closures of each of the elements in the slicing criterion. If these have already been pre-computed for each visible element in the module, then the union can be computed in time linear in the number of elements in the slicing criterion. In Section 3, the transitive closure method generated the interface slice {*Set, ElemType, Create, Insert, Equal (Set, Set), NoMore, SetIter, IsMember, MakeSetIter, More, Next*}, and by consulting Table 2, it is clear that this is exactly the interface slice generated by the union method as well.

In Section 3, we discussed two specific slices of interest of *SetPkgTemplate*. The mechanisms are now in place to insert these two slices into the lattice and examine their import and relationships. Figure 22 shows the lattice with these two slices in place. (In the figure, *ET* is sometimes used for *ElemType*, to save space.)

4.6 Modifications in Interface Slices

Now that there is a process to generate interface slices of interest and a process to arrange them in a lattice based on subset inclusion, what conclusions can be drawn about the interface slices so generated and arranged? What information can be gleaned from this arrangement? Of what use is this representation? Some answers to these questions appear in this section, some are more properly discussed in the context of specific applications and so are deferred to a future report, and some will no doubt become apparent during the course of further development of the theory and technology of interface slicing.

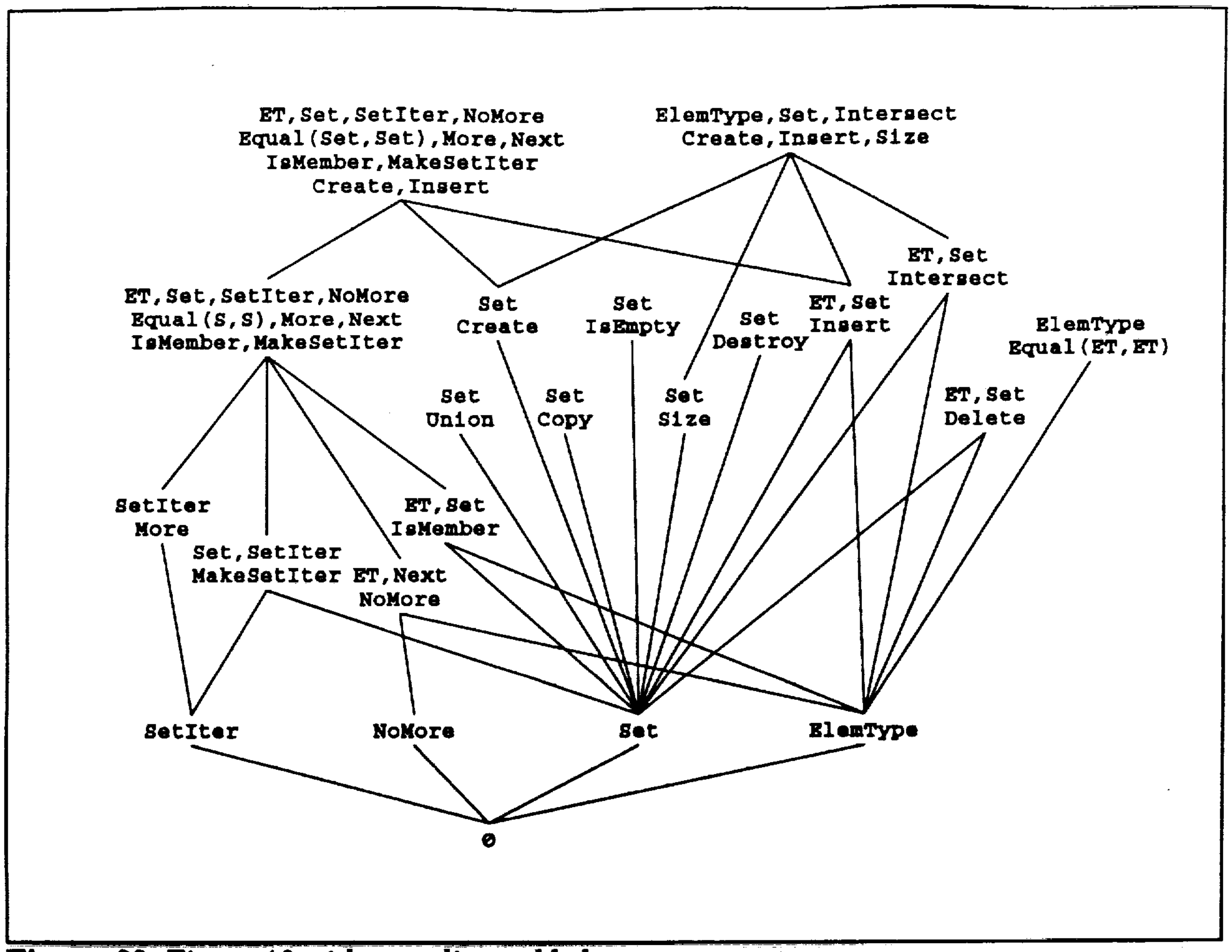

Figure 22 Figure 19 with two slices added

Proposition 7 Modifications in Mutually Independent Slices. Given two mutually independent interface slices S_1 and S_2 of module M, a modification to S_1 which does not add a reference to an element in S_2 has no effect on S_2.
Proof: Since S_1 and S_2 are mutually independent, by definition they have $\varnothing$ as their only common maximal lower bound. By the construction algorithm, this means that for all elements x in S_1 and for all elements y in S_2, $y \notin iref(x)$. Therefore a modification in S_1 cannot affect an element in S_2, provided the modification does not add a new reference in S_1 to an element to S_2. □

This result can be extended directly to sets of mutually independent slices, so that given a set $S_1, S_2, \ldots, S_n$ of mutually independent interface slices, a modification to S_1 which does not add a reference to any element in $S_2, \ldots, S_n$ has no effect on any of the $S_2, \ldots, S_n$.

The converse of this result is that given a pair of mutually dependent interface slices S_1 and S_2, a modification to S_1 may have an effect on S_2, and thus call for an examination of that effect. The arrangement of the slices into a lattice allows the determination of mutual dependence by inspecting the set of common lower bounds of the nodes in question.

References

[1] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. "Efficient implementation of lattice operations," *ACM Transactions on Programming Languages and Systems*, vol 11, no 1, pp 115-146, January, 1989.

[2] H. Agrawal, R. DeMillo, and E. Spafford. "Dynamic slicing in the presence of unconstrained pointers," Technical Report SERC-TR-93-P, Software Engineering Research Center, Purdue University, West Layfayette, Indiana, July 1991.

[3] H. Agrawal and J. Horgan. "Dynamic program slicing," Technical Report SERC-TR-56-P, Software Engineering Research Center, Purdue University, West Layfayette, Indiana, December 1989.

[4] J. Atkins. Personal communication, January 1993.

[5] L. Badger and M. Weiser. "Minimizing communication for synchronizing parallel dataflow programs," *Proceedings of the International Conference on Parallel Processing*, (St. Charles), pp 122-126, 1988.

[6] J. Beck. "Interface slicing: a static program analysis tool for software engineering," Ph.D. Dissertation, Department of Statistics and Computer Science, West Virginia University, Morgantown, WV, May 1993.

[7] J. Beck and D. Eichmann. "Program and interface slicing for reverse engineering," *Proceedings of the Working Conference on Reverse Engineering*, (Baltimore, 21-23 May), and *Proceedings of the 15th International Conference on Software Engineering*, (Baltimore, 17-21 May), 1993.

[8] J. Bergeretti and B. Carré. "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems*, vol 7, no 1, pp 37-61, January 1985.

[9] T. Biggerstaff and A. Perlis. *Software Reusability*, Vol I, Addison-Wesley, Reading, MA, 1989.

[10] G. Booch. *Software Engineering with Ada*, Benjamin-Cummings, Menlo Park, CA, 1983.

[11] Department of Defense. "DoD Software Reuse Vision and Strategy," *CrossTalk*, no 37, pp 2-8, October 1992.

[12] D. Eichmann and J. Beck. "Balancing generality and specificity in component-based reuse," submitted for publication to *International Journal of Software Engineering and Knowledge Engineering*, May 1992, under revision.

[13] K. Gallagher and J. Lyle. "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol 17, no 8, pp 751-761, August 1991.

[14] M. Hecht. *Flow Analysis of Computer Programs*, Elsevier North-Holland, New York, 1977.

[15] S. Horwitz, T. Reps, and D. Binkley. "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol 12, no 1, pp 26-60, January 1990.

[16] B. Korel and J. Laski. "Dynamic program slicing," *Information Processing Letters*, vol 29, no 3, pp 155-163, October 1988.

[17] W. Kozaczynski, J. Ning, and A. Engberts. "Program concept recognition and transformation," *IEEE Transactions on Software Engineering*, vol 18, no 12, December 1992.

[18] C. Liu. *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

[19] A. Podgurski and L. Clarke. "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering*, vol 16, no 9, pp 965-979, September, 1990.

[20] J. Purtilo and J. Atlee. "Module reuse by interface adaptation," *Software - Practice and Experience*, vol 21, no 6, pp 539-556, June 1991.

[21] K. Smith. *Application Engineering With Domain-Specific Reuse*, Course Description, Central Archive for Reusable Defense Software (CARDS), STARS-AC-04102B/001/00, 6 March 1993.

[22] M. Soffa. "A framework for generalized slicing," an address to the Computer Science Department, West Virginia University, Morgantown, WV, 2 February 1993.

[23] J. Uhl and H. Schmid. "A systematic catalogue of reusable abstract data types," *Lecture Notes in Computer Science* vol 460, Goos and Hartmanis, eds., Springer-Verlag, Berlin, 1990.

[24] B. Weide, W. Ogden, and S. Zweben. "Reusable software components," in M. Yovits (ed.), *Advances in Computers*, Academic Press, Orlando, 1991.

[25] M. Weiser. "Program slicing," *IEEE Transactions on Software Engineering*, vol SE-10, no 4, pp 352-357, July 1984.

Appendix A: Ada Specification of *SetPkgTemplate*

```
with ListPkgTemplate;
generic
    type ElemType is private;
    with function Equal(e1, e2: ElemType) return boolean;
package SetPkgTemplate is
    type Set is private;
    NoMore: exception;
    type SetIter is private;
    function Create
        return Set;
    procedure Insert(s: in out Set;
                     e: in     ElemType);
    procedure Delete(s: in out Set;
                     e: in     ElemType);
    function Intersect(s1, s2: Set)
     return Set;
    function Union(s1, s2: Set)
     return Set;
    function Copy(s: Set)
     return Set;
    function Equal(s1, s2: Set)
        return boolean;
    function IsEmpty(s: Set)
        return boolean;
    function IsMember(s: Set;
                       e: ElemType)
        return boolean;
    function Size(s: Set)
        return natural;
    function MakeSetIter(s: Set)
        return SetIter;
    function More(iter: SetIter)
        return boolean;
    procedure Next(iter: in out SetIter;
                   e:    out    ElemType);
    procedure Destroy(s: in out Set);
private
    package ListPkg is new ListPkgTemplate(ElemType, Equal);
    use ListPkg;
    type Set is new List;
    type SetIter is new List;
end SetPkgTemplate;
```

Appendix B: Ada Specification of *ListPkgTemplate*

```
generic
      type ItemType is private;
      with function Equal ( X,Y: in ItemType) return boolean;
package ListPkgTemplate is
          type List       is private;
          type ListIter   is private;
    CircularList     :exception;
    EmptyList            :exception;
    ItemNotPresent       :exception;
    NoMore               :exception;
procedure Attach(
          List1:     in out List;
          List2:     in     List
);
function Attach(
         Element1: in ItemType;
         Element2: in ItemType
) return List;
procedure Attach(
         L:       in out List;
         Element: in     ItemType
);
procedure Attach(
         Element: in      ItemType;
         L:       in  out List
);
function Attach (
         List1: in     List;
         List2: in     List
) return List;
function Attach (
         Element: in    ItemType;
         L:       in    List
) return List;
function Attach (
         L: in          List;
         Element: in    ItemType
) return List;
function Copy(
       L: in List
) return List;
generic
        with function Copy(I: in ItemType) return ItemType;
function CopyDeep(
         L: in     List
) return List;
function Create
return List;
procedure DeleteHead(
          L: in out List
);
procedure DeleteItem(
      L:        in out List;
      Element: in     ItemType
);
procedure DeleteItems(
      L:        in out List;
      Element: in     ItemType
);
procedure Destroy(
          L: in out List
);
function FirstValue(
         L: in List
) return ItemType;
function IsEmpty(
         L: in     List
) return boolean;
function IsInList(
         L:       in     List;
         Element: in     ItemType
) return boolean;
function LastValue(
         L: in List
) return ItemType;
function Length(
         L: in List
) return integer;
function MakeListIter(
         L: in List
) return ListIter;
function More(
         L: in ListIter
) return boolean;
procedure Next(
    Place:    in out ListIter;
    Info:        out ItemType
);
procedure ReplaceHead(
     L:    in out List;
     Info: in     ItemType
);
procedure ReplaceTail(
          L:       in out List;
          NewTail: in     List
);
function Tail(
         L: in List
) return List;
function Equal(
         List1: in List;
         List2: in List
 )  return boolean;
private
    type Cell;
    type List is access Cell;
    type Cell is
         record
              Info: ItemType;
              Next: List;
         end record;
    type ListIter is new List;
end ListPkgTemplate;
```