Program and Interface Slicing for Reverse Engineering*

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

David Eichmann
Department of Computer Science
University of Houston - Clear Lake
2700 Bay Area Boulevard, Box 113
Houston, TX 77058
eichmann@cs.wvu.wvnet.edu

Abstract

Reverse engineering involves a great deal of effort in comprehension of the current implementation of a software system and the ways in which it differs from the original design. Automated support tools are critical to the success of such efforts. We show how program slicing techniques can be employed to assist in the comprehension of large software systems, through traditional slicing techniques at the statement level, and through a new technique, interface slicing, at the module level.

1 Introduction

Reverse engineering is partly the result of inadequate design capture in software development and partly the result of inexorable advance of software science. Inadequate design capture results in code that is difficult to support, maintain, and reuse. The continuous advancement in software science can result in even today's good code seeming archaic tomorrow. Both situations demand the tools and techniques of reverse engineering. Rekoff defined reverse engineering as the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system [21] - typically by someone other than the original designer. The applicability to software has been demonstrated by work such as that described in [17] and the conferences on Software Maintenance. Few systems have the same maintainers as they had designers, and few designers possess the ability to remember for long the multitude of design choices made during a software development effort. Furthermore, the complexity of the source code for large systems makes comprehending design choices embedded within the code extremely difficult without support tools.

Chikofsky and Cross [4] characterize *forward engineering* as the traditional process of moving from high-level abstractions and logical designs to the physical implementation of a system; *reverse engineering* as the process of analyzing a system to identify the its components and their interrelationships, and creating representations of the system in another form or at a higher level of abstraction; and *reengineering* as the examination and alteration of a system to reconstitute it in a new form and the subsequent implementation of the new form. Hence, forward engineering drives the creation of implementation from design representations, reverse engineering drives the recreation of the design representations from the implementation, and reengineering drives the creation of a new implementation from an old implementation. Intermediate representations can play implicit or explicit roles in these transformations.

Reverse engineering and reengineering transformations involve the reabstraction of an existing implementation. However, reabstraction techniques frequently suffer from noise induced in the implementation by maintenance efforts. Program slicing provides a toolset to provide both semantic and syntactic reabstraction, without regard to additional enhancements or patches done to the implementation. Further, decomposition slice lattices [8] provide a visualization mechanism for the relationships between the syntatically and semantically correct subsets of program behavior. In particular, the lattice of slices for a current implementation differs from the lattice of slices for the original implementation in ways specifically related to the maintenance done on the system. Reabstraction, therefore, can be concentrated on the differences between the original and current lattices.

We take a rather broad perspective as to the nature of reverse engineering, decomposing it into two phases, algorithm recognition and design recognition. Algorithm recognition involves extraction of behavior from the implementation. Design recognition involves extraction of rationale for behavior from the algorithm. Many traditional techniques employed in reverse engineering (e.g., structure charts and call graphs) are inadequate for the recognition

*This work was supported in part by NASA as part of the RBSE project, cooperative agreement NCC-9-16, and in part by a grant from MountainNet, Inc.

of algorithms, and certainly for the recognition of design. Slicing mechanisms provide an excellent foundation on which to construct recognition tools.

Our perspective is colored by our interest in slicing in the contexts of repositories and reusability. Long-lived components frequently accrete much functionality over their lifetimes (the kitchen sink syndrome), making the comprehension required for modification or reabstraction increasingly difficult. We introduce a new form of program slicing, called *interface slicing*, to support one of the goals of reverse engineering, providing knowledge about a component to support its modification. Interface slicing eliminates the need for some manual reverse engineering efforts, as it automates specific kinds of modifications.

Repository use involves many of the tools and techniques of reverse engineering to first select and then modify components for a new system. Interface slicing reduces the burden of comprehension through a reduction of the size and complexity of subsystem interfaces.

In this paper we present a case for the use of conventional and interface slicing as enabling mechanisms for numerous reverse engineering and reengineering tasks. We first discuss the applicability of conventional slicing to algorithm extraction and design recovery at statement-level granularity. We then present interface slicing and show how it provides similar capabilities at module-level granularity.

We employ the following terms. *Module* is a general term for a collection of subprograms, possibly with information hiding mechanisms; it includes but is not limited to Ada packages. *Component* refers to a module in a reuse repository. A component is thus a code asset of a repository, possibly also incorporated into a program.

We will use Ada for our examples, as Ada's features facilitate the types of transformations which we invoke, but our concepts are not confined to any particular language.

2 Conventional program slicing

In this section, we describe how program slicing can be applied to some of the goals of reverse engineering. While program slicing of any type is insufficient by itself to completely accomplish the goals of reverse engineering, program slicing can be used as a powerful support tool to complement other techniques employed by the software engineer attempting to understand, redocument, and modify existing software systems.

Weiser introduced the concept of program slicing while studying the abstraction mechanisms used by programmers in analyzing existing programs while debugging code [25, 26,27]. Before Weiser's work, all abstraction mechanisms to date had decomposed programs into "units" by grouping *sequential* program elements. At the lowest level, a raw dump of an executable program consists of a very large number of homogeneous units, bytes, far too many for the mind to grasp. The process of understanding a program consists of organizing this huge number of units into fewer units, allowing the program to be viewed at a higher level of abstraction. For example, a single assembly language statement replaces many sequential bytes of the dump, while a compiler allows the replacement of many assembly language statements with one high-level statement. Further, a hierarchical organization can be imposed on statements by grouping many sequential statements into subprograms, subprograms into modules, and so on.

After experiments in which he studied the behavior of programmers who were attempting to comprehend and debug programs, Weiser concluded that while the grouping of sequential sets of statements did indeed serve as an aid to program comprehension, programmers who were attempting to debug a program used a different mental abstraction mechanism for grouping program statements. Specifically, they used a mental mechanism which grouped generally *non-sequential* sets of statements. Weiser concluded that the statements grouped in this way were those which applied to "units of data components," that is, variables. He found that he could understand the mental abstraction of the debugging process by examining data flow diagrams of a program. He called his theory of data behavior abstraction *slicing*, and the units of abstraction, *slices*. The slices abstract a program based on the behavior of the program with respect to a specified set of variables rather than with respect to sequential statement listing. A slice is a complete program which contains a subset of the statements of the original program, and which performs a subset of the computations performed by the original program. The slice is obtained by removing statements from the original program which do not affect the specified behavior of interest. The slicing algorithm must ensure that the behavior of the slice is also a subset of the original program behavior.

A number of researchers (e.g., [7,13]) have worked to improve the techniques of program slicing. The techniques employed are covered by these authors, we wish only to note that the consensus technique for slicing involves first building a program representation with some form of dependence graph using data- and control-flow analysis, then generating actual slices graph operations. These forms of slicing we will henceforth term *conventional* slicing, in distinction from the new form, *interface slicing*, which we present below.

2.1 Slicing in software engineering

In addition to the original focus on program comprehension and debugging, slicing has been used to address various software engineering issues including program maintenance and testing [15,8], integrated development environments [14], module cohesion metrics [18], program variant merging [12], repository component generation [8], parallelization [26], and software portability [16]. As has been noted many times (e.g., [4]), software (forward) engineering and reverse engineering, far from being separate, often involve complementary and even overlapping tools and techniques. It is not surprising, therefore, that program slicing, a technique which has seen wide application to the problems of traditional forward software engineering, would have strong applicability to reverse engineering tasks as well. We wish to suggest one or two applications in which slicing can be applied to the goals of reverse engineering in ways which have previously not been considered.

2.2 Redocumentation for maintenance

Gallagher and Lyle [8] have developed the idea of totally decomposing a program by slicing and arranging the decomposition slices into a lattice based on the partial ordering of statement set inclusion. They use this decomposition lattice as the basis for guaranteeing whether a change made to a program will or will not have any effect outside a specific set of program statements. Because the decomposition is total, the locus of a desired statement change is within at least one decomposition slice; the rest of the program is the complement. Based on the partial ordering of the lattice, that locus is either independent of the complement, or not. If independent, the change can be made with the guarantee that the complement, which is the rest of the program, will not be affected by the change in any way; a program modified by such a change must only be retested and revalidated within the decomposition slice. If the locus of change is not independent, then the complement *is* affected by the change, and thus an investigation of that effect is necessary. The arrangement of the program into a total decomposition slice lattice provides a redocumentation of the program which makes clear to the maintainer the relationships of statements which are dependent upon one another, and those which are independent.

This lattice arrangement of slices has far more potential for reverse engineering than has yet been suggested, however. Consider the example program used by Gallagher and Lyle [8], a version of the Unix *wc* program, which counts occurrences of characters, words, and lines. The root slice, involving only the character variable, actually provides a cliché (see Section 2.4) for correct character-level I/O in a Unix system.

The incremental changes mentioned above will cause the slice lattice for the new version of the system to differ at particular vertices from the lattice for the original system. Examining the slice lattice for an existing system and how it differs from the lattice for the original system, which corresponds to the original design, specifically identifies those portions of the current system in need of redocumentation.

2.3 Design recovery

Source-to-source transformation approaches to reengineering are typically based on some form of pattern matching. Engberts, et al., [6] and Platoff, et al. [19] use pattern matching techniques to transform source code to program concepts (both syntactic and semantic) and to (application specific) domain concepts. However, their concept recognition mechanisms are limited by their implementation language (COBOL and C, respectively). Block structured languages can smear concepts across multiple procedures in multiple scopes, requiring a slicing-like mechanism to extract them in a form that a pattern-matcher can recognize.

2.4 Related work

The notion of organizing a program in ways other than the traditional hierarchy of units of increasing abstraction is not unique to slicing. Soloway and Ehrlich [23] developed the theory that programming knowledge consists in part of *programming plans*. A programming plan is an abstract structure which a programmer uses as a template or link between a goal and a specific program fragment instance. A programmer might use, for example, a *data guard plan* to help accomplish the goal of preventing division by zero. In the program, the data guard plan is manifested in the test predicate and control structure necessary to prevent division by zero, while allowing division by appropriate values. While a plan may be a single abstract entity, it is manifested in a program by statements which are, in general, non-sequential. Indeed, in many cases, an appropriate choice of slicing criterion applied to a program segment is sufficient to recover an intact plan as a slice. Rich and Wills [22] have developed a prototype module of the Programmer's Apprentice called the *Recognizer* which automatically recognizes *clichés*, which essentially are the manifestation of plans in programs. Similarly to slicers, the Recognizer stores program information in the form of a flow graph; the clichés are then found by graph analysis.

3 Interface slicing

Intuitively, an *interface slice* may be viewed as a subset of the behavior of a module, just like the original notion of a conventional 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 behaviors which a given module exports to its containing software system.

While conventional slicing was originally designed primarily for debugging and comprehension, interface slicing was primarily developed 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 code which results from the reuse effort. But just as the role of conventional slicing has expanded to many areas of forward and reverse engineering, so we see a broad applicability of interface slicing to comprehension, maintenance, redocumentation, and reengineering.

3.1 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.

```
package toggle1 is
   function on return boolean;
   function off 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;
   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;
 end toggle1;
```

Figure 1 A boolean toggle package

Consider a simple 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 1. 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. 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 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.

However, suppose that in developing a system we find that we need, not all, but only some of the functionality of the *toggle1* package. Specifically, suppose that we have need of only the *on, set,* and *reset* operations, but not of 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. A 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. Alternatively, the second option is to manually edit the source code of *toggle1* and delete the *off* operation from both body and specification of the package. A disadvantage of this option is that manual editing requires full code-level 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 errors.

Indeed, the option of manually editing the source code assumes that we have access to it. But this is not necessarily the case, especially in a commercial reuse repository context. If an interface slicer is available, it is easy to propose a repository structure which gives full code access to the slicer, allowing automated modification, while restricting human access to 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 discussed above. In the scenario described above, we wish to use 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 the module; we are interested in a subset of that functionality. In effect, we wish to remove, i.e. slice away, the unneeded functionality, as in manual editing, 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 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 interface specification and code body, and 2) a *list* consisting of the subset of the operations which we desire. This

```
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 2 *toggle1* sliced on *<on, set, reset>*

```
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 3 A larger boolean toggle package

list is the *interface slicing criterion*. 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 2. Note that in this simplest example, the slice consists merely of the original package without the specification or body of the unwanted function *off,* just as would have been produced by manually deleting the *off* procedure from the package specification and body. This simple example has no linkages or dependences among its operations; we will now discuss these.

As a second example, consider the more sophisticated boolean toggle which is implemented by the package shown in Figure 3. In addition to the operations of *toggle1*, this package also exports the operation *swap* which reverses the value of the toggle. Suppose that we wish to include in the software system just the functionalities of the operations *on* and *swap,* with *toggle2* available in the repository. 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 *swap,* we must also include *set* and *reset.* The result of interface slicing *toggle2* on *<on, swap>* is shown in Figure 4. 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;
   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 4 *toggle2* sliced on *<on, swap>*

The specific dependences among *swap, on, set,* and *reset* 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. We could easily envision an implementation in which *swap* depends upon *off* rather than upon *on*. This comprehension of the package is required for manual editing; interface slicing obviates this comprehension requirement. Using an interface slicing tool, we do not have to know anything about the internal dependences of the toggle package for this modification, as the slicer does the dependence analysis during its operation.

3.2 An interface slicing mechanism

The previous examples illustrate the usefulness of interface slicing but do not indicate how it can be accomplished. Here, we demonstrate a method for generating the interface slices of the previous section.

When a package is *with*ed in a standard Ada environment, the entire package is imported into the software system. This includes all public and private variables, subprograms, and types. But as illustrated in the examples above, we make the generalization that typically any particular software system will need only a subset of the functionality of a given repository component. This is particularly likely to be true in three common cases. The first is the case of a component written to be a general component in a non-domain-specific repository, which was written for reuse, and which thus will contain all possible anticipated functionality, the better to accommodate all possible anticipated uses. 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 as a general component. The third case is that of a component which has been reused many times, each time having a bit more functionality accreted to it. This is exemplified by the creeping featurism of Unix programs.

As stated above, the interface slicing tool has as input the original complete package and a slicing criterion which is a list of desired public subprograms, types, and variables. This list is supplied without knowledge of the package implementation. The problem at hand for the slicer is to determine from a static analysis of the package what portion of the package is necessary to support the items in the slicing criterion. The solution to the problem is to perform a reachability analysis [9], based on the slicing criterion, of a dependence graph of the package.

3.3 The interface dependence graph

In this discussion, for simplicity, we assume no nesting. That is, all the subprograms defined in the package are at the top level of the package. The specific dependence graph required for this analysis we term the package's *interface dependence graph* (IDG). The IDG of a package can be constructed with a single pass 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), any global variable (including constants and generic formal parameters), or any subprogram (including tasks and exceptions). Every node is labeled with the name of the defined program element to which the node corresponds, and nodes are annotated with source code line numbers and program element type signatures. Since we are not considering nesting in this example, it is not necessary to keep track of the scope of definitions: every definition is global to the package.

The edges of the IDG are dependence edges, 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*. If *X* contains a pointer, there must be an edge from *X* to every possible target of the pointer. Self-edges, indicating direct recursion, are not necessary and are omitted. IDGs for the *toggle1* and *toggle2* packages described above are shown in Figure 5.

Once the IDG has been constructed, generating an interface slice based on a slicing criterion of desired functionality is a straightforward process. 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 program elements which correspond to the transitive closure, plus any needed syntactic sugar (see Section 3.7) required for the package structure.

For instance, consider the interface slice which this process generates for the *toggle1* example discussed earlier. The example discussed an interface slice for the *toggle1* package, whose IDG is shown in Figure 5(a), based on the slicing criterion *<on,set,reset>*. The transitive closure of this criterion 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 we reached by informal consideration, as shown in Figure 2. Similarly, the transitive closure of the criterion *<on,swap>* for Figure 5(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 4.

3.4 An extended example

The examples above illustrate the general concept of

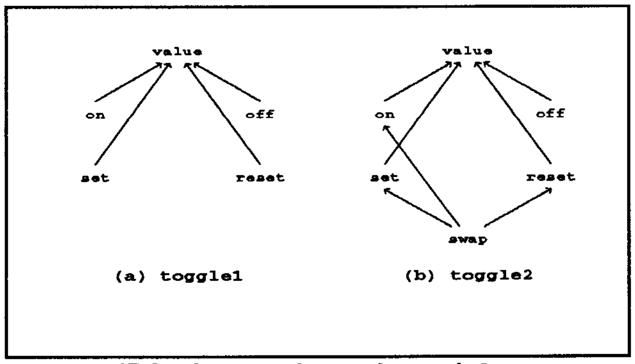

Figure 5 IDGs for *toggle1* and *toggle2*

interface slicing, but leave out some important details. To fill in some of these details, we will next examine a pair of generic Ada packages which are in the public domain. (They were extracted and modified from the ASR repository on SIMTEL 20 and were originally written by B. Altus and R. Kownacki of Intermetrics.) These packages were written to be used as building blocks for Ada programs, similarly to the components of Booch [3] or Uhl and Schmid [24]. The first of the packages implements the ADT *set* in the 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 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.

SetPkgTemplate happens to use a *list* ADT 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 package *ListPkgTemplate.* This is a singly-linked dynamic list implementation which 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.

In the particular list and set packages we used for this example, there are some private subprograms and types. Private program elements are not available to be used in an interface slicing criterion; only the exported subprograms, variables, and types in the specification can be in the slicing criterion. However, private program elements must be included in the IDG, as the transitive closure of a public element may flow to a private element. Thus, with the exception that private elements may not appear in the slicing criterion, private elements are treated identically to public ones during the interface slicing process. The slicer is a privileged code transformation tool and has complete access to all portions of the source code.

3.5 A single level of slicing

Suppose that we wish to use the set package 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 addition, to use *SetPkgTemplate* at all, we must also use the type *Set* and we must supply the set element type *ElemType.* Therefore we wish to slice *SetPkgTemplate* on the slicing criterion *<Create,Insert, Equal,Set,ElemType>.* We would like to include all the code necessary to allow us 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 IDG for the set package, which is shown in Figure 6. In the interest of legibility in that and subsequent figures, the annotation and labeling of the nodes are abbreviated. The transitive closure of the five nodes listed in the slicing criterion, corresponding to the desired slice of the set package, is shown in Figure 7.

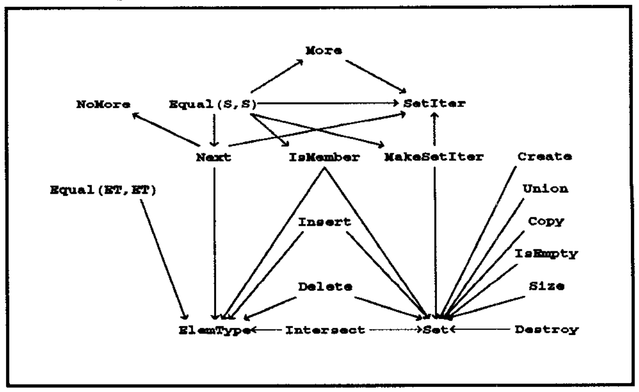

Figure 6 IDG for *SetPkgTemplate*

Out of the total of 3 global types and 16 subprograms, one each of which is a generic parameter, on 151 lines of code in the original package, the slice contains 3 global types and 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 6 vs. Figure 7 shows the reduction in interface size and complexity of the sliced set package (see also Table 1).

3.6 Name overloading

To slice *SetPkgTemplate,* we used the slicing criterion *<Create, Insert, Equal, Set, ElemType>.* Giving this exact slicing criterion to an automatic slicing tool won't work,

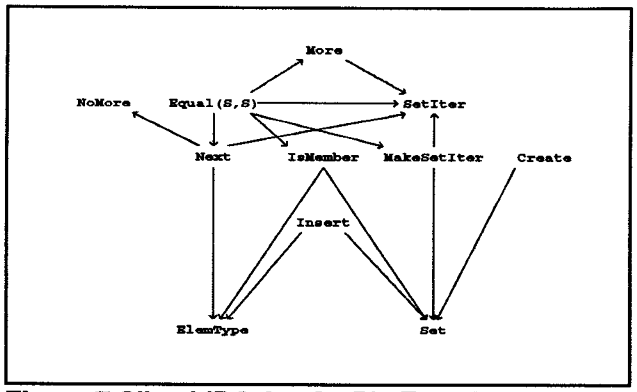

Figure 7 Sliced IDG for *SetPkgTemplate*

however, because the name *Equal* is overloaded in the package. *Equal* appears once in the package with the type signature *function: ElemType* × *ElemType* → *boolean,* while it appears again with the type signature *function: Set* × *Set* → *boolean.* Note that the IDG in Figure 6 has separate nodes for the two occurrences of the overloaded name *Equal.* Since *Equal* is overloaded, the slicing criterion above is ambiguous, and needs amplification with the argument type signature of *Equal,* i.e., *Equal(Set,Set)* (abbreviated in the figure as Equal(S,S).

3.7 Syntactic sugar

Up to this point, none of the concepts presented have been language-specific. But while the concepts are language-independent, an interface slice tool cannot be. Each language has its own structure and syntax which must be respected, else the output of the slicer will be syntactically incorrect. The interface slicer must keep track of the language syntax when generating the slice.

An example of this occurs in *SetPkgTemplate,* whose specification includes the line: `package ListPkg is new ListPkgTemplate(ElemType,Equal);`. As we have described it, the interface slicer is a pre-compilation text transforming tool which does not know to which of the two *Equals* in *SetPkgTemplate* this line refers: *Equal* is overloaded. By examining the specification of *ListPkgTemplate,* we can see that the proper generic parameter must be the function *Equal(ElemType,ElemType),* but the interface slicer does not know this.

The interface slicer must include the generic instantiation of *ListPkgTemplate* in the slice, as this is required syntax which it has no reason to slice out. If the slicer includes this line, which has the name *Equal* in it, it must also include the definition of *Equal.* Since *Equal* is ambiguous, the slicer must therefore include all definitions of *Equal,* to be sure of including the correct one.

3.8 Generic parameter number and other issues

A procedure which instantiates *SetPkgTemplate* has to supply an element type and an equality function. In the previous examples, the number (2) 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.

We cannot simply omit a generic parameter 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 SetPkgTemplate(ElementType, EqualFunc)`. If we simply drop the *EqualFunc* from the statement, we will get a compiler error complaining of a missing generic subprogram argument.

In general, with interface slicing we wish to be able to instantiate each generic package with some number of the parameters removed by the interface slicing process. One approach to reconciling mismatched parameters has been advanced by Purtilo and Atlee in the module interconnection language *Nimble,* which was designed to automatically adapt module interfaces which have large discrepancies in their parameters. Merely reconciling their number is easy for Nimble [20]. We therefore assume an interface slicer would be implemented with some mechanism for reconciling mismatched numbers of package parameters.

We have not covered here some advanced topics in interface slicing such as fully nestable structures, definitions *with*ed from other packages, issues related to the slicing of tasks, and late binding. Also, we have presented the slicer as though it were a standalone pre-compilation code transformer; in fact, it should be implemented as a portion of an integrated development environment, with full access to the libraries and databases of the environment. Some of these issues are covered in [1] and [5]; others will be in the report on the prototype under construction. The complete Ada specifications of the *set* and *list* packages are in [2].

3.9 A second level of slicing

While slicing *SetPkgTemplate* results in size and complexity improvements, 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. Just as the main program in the example above used functionality provided by the set package, so the set package requires functionality provided by the list package. But just as the main program did not need all of the functionality of the set package, so too does the set package need only a subset of the list package. That subset, or slice, is based, as above, on the slicing criterion of public variables, types, and subprograms exported by the list package which the set package directly references. It does not matter which elements the original unsliced set package referenced. All that matters is which elements the sliced set package references. In the case of the set package sliced on *<Create,Insert,Equal(Set,Set),Set,ElemType>*, the references to the list package consist of: *List, EmptyList, Attach(ItemType,List):List, Create, DeleteItems, FirstValue, IsEmpty,* and *IsInList*. This is therefore exactly 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*.

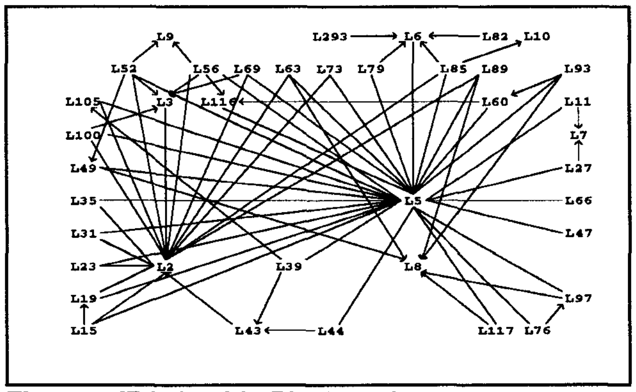

Figure 8 IDG for *ListPkgTemplate*

The IDG of the original *ListPkgTemplate* is shown in Figure 8. (In this figure, for legibility, line numbers are used rather than identifiers.) This package is large and complex enough to make manual editing a decidedly nontrivial task. However, slicing it using the criterion above, which corresponds to nodes L5, L8, L31, L47, L56, L63, L66, and L69, produces the much smaller and simpler graph shown in Figure 9, with 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 even more importantly for the software engineer charged with maintenance.

We list in Table 1 the actual change in size of the IDGs and packages of the set and list packages in the example above. We instantiated the packages in a driver program which was minimal in size while still using every *Set* entity in the slicing criterion; the executable was generated by Meridian Ada 4.1.3 for Sun-4 unix. The numbers in the table indicate that slicing reduces size of the set and list component source code by more than half, reduces the size of a test driver program's executable by a sixth. In other words, 17% of the executable for our simple unsliced example program produced by a commercial compiler is dead code. (We don't wish unfairly to pick on Meridian here. Similar results were obtained with a variety of other compilers and platforms.) While numerical results from a larger sample of larger programs will have to await the completion of our interface slicer prototype, based on this example of size reduction alone, the interface slicer can help to ease the size and comprehension problems in software maintenance.

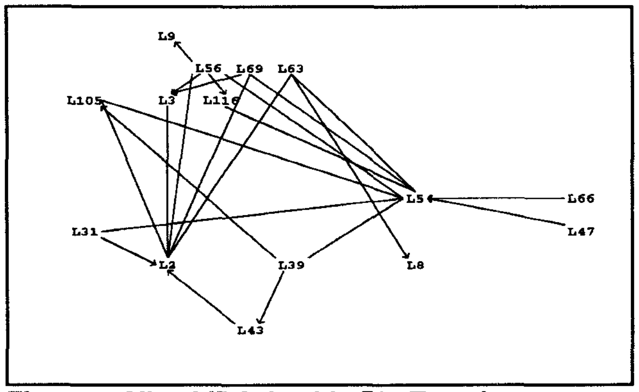

Figure 9 Sliced IDG for *ListPkgTemplate*

Because of the package structure and emphasis on composition-based modular reuse, Ada is very amenable to interface slicing. Because of the very large installed base of Ada code, representing substantial development investment and intellectual capital of both industry and government, the potential benefit for efficient reengineering, adaptation, and reuse of Ada packages is enormous. Interface slicing is a technique which can substantially reduce the comprehension required for a package's reuse, and substantially reduce the size of the software system, in both source and object forms, resulting from reuse.

4 Future directions

Our future plans include both abstract and practical enhancement of this work. We will be implementing an Ada interface slicer this summer as part of our work with the Repository Based Software Engineering project. This slicer will be used to generate the (program and interface) lattice of slices for those components in our repository that were written in Ada. We expect this work to allow us to

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

Table 1 Size improvements of IDG and package for Set and List

infer relationships between the complexity of the slice lattices and the reusability and maintainability of the corresponding components.

We also intend to apply our techniques to object-oriented languages, particularly focusing on the possibility of interface slicing class definitions for both the derivation of new parent classes as part of the reengineering process and for the purpose of negotiating reuse of existing class definitions through composition. Recent work in contracts [10,11] appears relevant here. We are also considering our techniques in the context of reengineering legacy code into new object-oriented systems.

5 Conclusion

Reverse engineering will always be with us. While organizations that adopt mature, process-oriented development models will find less need to reverse engineer or reengineer since they will have captured the transformations as they occur and their rationales as they are generated, organizations at earlier stages of the maturity continuum will not have this ability. The only way in which they will be able to leverage the intellectual capital buried in their legacy code is through reverse engineering. Software engineering environments must provide a balanced set of forward and reverse engineering tools to support both mature organizations in their normal operations and less mature organizations in their transition to maturity. We have presented here a portion of the foundation to build such a toolset.

References

[1] J. Beck, "Interface slicing: a static program analysis tool for software engineering," PhD diss., Dept. Stat. & Computer Sci., West Virginia Univ., under preparation, 1993.

[2] J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering," Comp. Sci. Tech. Rep. TR-93-3, West Virginia Univ., 1993.

[3] G. Booch. *Software Components with Ada*, Benjamin-Cummings, 1987.

[4] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Softw.*, 7(1), pp 13-17, Jan. 1990.

[5] D. Eichmann and J. Beck, "Balancing generality and specificity in component-based reuse," submitted for publication, 1992.

[6] A. Engberts, W. Kozaczynski and J. Ning, "Concept recognition-based program transformation," *Proc. Conf. Softw. Maintenance*, pp 73-82, Sorrento, Italy, 15-17 Oct. 1991.

[7] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programming Lang. and Syst.*, 9(3), pp 319-349, Jul. 1987.

[8] K. Gallagher and J. Lyle, "Using program slicing in software maintenance," *IEEE Trans. Softw. Eng.*, 17(8), pp 751-761, Aug. 1991.

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

[10] A. Helm, I. Holland and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," *Proc. OOPSLA/ECOOP 90*, Ottawa, Canada, pp 169-180, 21-25 Oct. 1990.

[11] I. Holland, "Specifying Reusable Components Using Contracts," *Proc. ECOOP 92*, Utrecht, The Netherlands, pp 287-308, 29 Jun. - 3 Jul. 1992.

[12] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," in *15th Ann. ACM SIGACT-SIGPLAN Symp. Principles Prog. Lang.*, pp 133-145, (San Diego, 13-15 Jan.), ACM Press, 1988.

[13] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Prog. Lang. and Syst.*, 12(1), pp 26-60, Jan 1990.

[14] P. Livadas, S. Croll, and P. Roy, "Towards an integrated software maintenance environment," in *Proc. 1st Softw. Eng. Research Forum*, (Tampa, Florida, 7-9 Nov.), 1991.

[15] J. Lyle and K. Gallagher, "A program decomposition scheme with applications to software modification and testing," *Proc. 22nd Hawaii Intl. Conf. Syst. Sci.*, pp 479-485, 1989.

[16] J. Mooney and M. Sitaraman, pers. comm., Oct. 1992.

[17] W. Osborne and E. Chikofsky (eds.). Special Issue on Maintenance, Reverse Engineering and Design Recovery, *IEEE Softw.*, 7(1), Jan. 1990.

[18] L. Ott and J. Thuss, "The relationship between slices and module cohesion," *Proc. 11th Intl. Conf. Softw. Eng.*, pp 198-204, May 1989.

[19] M. Platoff, M. Wagner and J. Camaratta, "An integrated program representation and toolkit for the maintenance of C programs," *Proc. Conf. Softw. Maint.*, pp 129-137, Sorrento, Italy, 15-17 Oct. 1991.

[20] J. Purtilo and J. Atlee, "Module reuse by interface adaptation," *Softw.-Practice and Exper.*, 21(6), pp 539-556, Jun. 1991.

[21] M. Rekoff, "On reverse engineering," *IEEE Trans. Systems, Man, and Cybernetics*, pp 244-252, Mar.-Apr. 1985.

[22] C. Rich and L. Wills, "Recognizing a program's design: a graph-parsing approach," *IEEE Softw.*, 7(1), pp 82-89, Jan. 1990.

[23] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Trans. Softw. Eng.*, SE-10(5), pp 595-609, Sep. 1984.

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

[25] M. Weiser, "Program slicing," *Proc. 5th Intl. Conf. Softw. Eng.*, pp 439-449, May 1981.

[26] M. Weiser, "Programmers use slices when debugging," *Comm. ACM*, 25(7), pp 446-452, Jul. 1982.

[27] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, SE-10(4), pp 352-357, Jul. 1984.