Balancing Generality and Specificity in Component–Based Reuse*†

David Eichmann and Jon Beck

Software Reuse Repository Lab
Dept. of Statistics and Computer Science
West Virginia University

Send correspondence to:
David Eichmann
SoRReL
Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506

Email: eichmann@cs.wvu.wvnet.edu

---

[*] Submitted to *The International Journal of Software Engineering and Knowledge Engineering*.

[†] This work was supported in part by NASA as part of the Repository Based Software Engineering project, cooperative agreement NCC-9-16, project no. RICIS SE.43, subcontract no. 089 and in part by a grant from MountainNet Inc.

Abstract

For a component industry to be successful, we must move beyond the current techniques of black box reuse and genericity to a more flexible framework supporting customization of components as well as instantiation and composition of components. Customization of components strikes a balance between creating dozens of variations of a base component and requiring the overhead of unnecessary features of an "everything but the kitchen sink" component. We argue that design and instantiation of reusable components have competing criteria – design-for-reuse strives for generality, design-with-reuse strives for specificity – and that providing mechanisms for each can be complementary rather than antagonistic. In particular, we demonstrate how program slicing techniques can be applied to customization of reusable components.

1 – Introduction

The impediments to a successful reuse infrastructure in the software engineering community have typically been separated into social and technological issues [26]. Furthermore, the social issues (e.g., comprehension, trust, and investiture) often are characterized as being the more critical, as there is a perception that all of the technical issues (e.g., environments, repositories, and linguistic support) have been solved [27]. We do not agree with this assessment (see [8] for our arguments regarding repositories and environments), and furthermore believe that appropriate application of technology can alleviate certain of the social issues just mentioned.

This paper addresses two reuse impediments – component comprehension by a reuser [14] and the fitness of a component for a given application – and how technical support, in this case language features and program slicing, alleviate these impediments. These two impediments drive the consumer side of reuse repository design, for without comprehensibility users will not select artifacts from the repository, and without adequate conformance to requirements users will not incorporate artifacts into systems even if they do select them. These two impediments also drive the design process for reusable components, since components perceived as ill-suited for reusers' application domains (and hence not incorporated into the resulting systems) have not met the requirements of a design-for-reuse effort.

We begin in section 2 by characterizing the inherent conflict between the design goals for design-for-reuse and design-with-reuse. We then review mechanisms that support particular structural and behavioral aspects of component design in section 3. The mechanisms described support flexibility in the ***design*** of a component. We consider mechanisms in section 4 to constrain an implementation, supporting specificity in the ***instantiation*** of a component, and show in section 5 how to employ program slicing as one such mechanism. Section 6 demonstrates the application of our technique to a moderate-sized example.

2 – Design-For-Reuse versus Design-With-Reuse

Design for reuse focuses on the potential reusability of the artifacts of a design process. *Design with reuse*, on the other hand, focuses on employing existing artifacts wherever possible in the design process. The intent of the two approaches, and hence the various criteria that each of them employ, is then quite distinct. In particular, design for reuse strives for generality, even to the point of additional cost to the current project, and design with reuse strives to reduce cost to the current project, even to the point of adapting non-critical project requirements to achieve conformance with existing artifacts.

Garnett and Mariani proposed the following attributes for reusable software [10]:

- environmental independence – no dependence on the original development environment;
- high cohesion – implementing a single operation or a set of related operations;
- loose coupling – minimal links to other components;
- adaptability – easy customization to a variety of situations;
- understandability;
- reliability; and
- portability.

These attributes clearly reflect goals that should apply to all products of a design-for-reuse effort, and some of these attributes (particularly understandability and reliability) apply to all software development efforts. Not so clear is whether these attributes reflect the goals of design-with-reuse efforts.

We contend that there is an inherent conflict between design-for-reuse and design-with-reuse that centers upon adaptability. Design-for-reuse strives to create artifacts that are as generally applicable as possible, in the worst case creating "everything-but-the-kitchen-sink" artifacts, loading a component with features in an effort to ensure applicability in all situations. Design-with-reuse strives to identify that artifact which most specifically matches a given requirement. Anything less requires additional effort, both in comprehension and coding. Anything more carries with it the penalty of excess resource consumption and increased comprehension effort.

The specificity that we seek in design-with-reuse takes two forms – the first is that of avoiding additional functionality in a simple component; the second is that of avoiding additional functionality in an abstraction, implemented as a package/module. Specificity becomes increasingly critical when considering scale. The additional storage consumed and increased comprehension effort posed by a simple abstract data type quickly become the multi-megabyte "hello world" applications of today's user interface management systems, and threaten intractability in the domain of megaprogramming [4, 19].

3 – Language Mechanisms Supporting Design–For–Reuse

Designing a software component for reuse involves a number of issues, including analysis of the intended target domain [21, 22], the coverage that this component should provide for the domain [22], and the nature and level of parameterization of the component [7, 28, 29]. A number of developments in programming language design directly bear upon these issues. We focus here upon those we see as most beneficial.

3.1 – Procedural and Modular Abstraction

The obvious advantages that functions and procedures provide in comprehension and reuse of portions of a program (even if the reuse is only at a different location in the same program) are so well recognized, that no contemporary language proposal is taken seriously without them. The package (or module) concept, with separate specification and implementation of a collection of data and procedural definitions, has arguably reached the same level of acceptance. Sommerville's list of classes of reusable components (functions, procedures, declaration packages, objects, abstract data types, and subsystems) [25] indicates the depth of this acceptance – virtually every class listed is directly implementable using one of the two mechanisms (objects being the only non-obvious fit).

3.2 – Parameterization and Genericity

The utility of a function or procedure is severely limited without the ability to provide information customizing the effect of a specific invocation. Parameters comprise the explicit contract between a function and its invocations, and are generally accepted as far preferable to the implicit contract provided by shared global state. Genericity, or more formally, parametric polymorphism [6], involves the parameterization of program units (both functions/procedures and packages/modules) with types, variables, and operations (functions, procedures, tasks, and exceptions). Parameters effectively support families of *invocations*. Genericity extends this support to families of *instantiations*, each with its own family of invocations, providing increased adaptability and portability [28].

3.3 – Inheritance

Inheritance involves the creation of generalization/specialization structures, a tree in the case of single inheritance, a lattice in the case of multiple inheritance. These generalizations/specializations may be structural (in the case of subtypes [6]) or behavioral (in the case of classes [11]). Whatever the structuring mechanism, inheritance supports the creation of variations of a base component, each with its own interface [15], as well as instances of those variations. Inheritance thus is a very useful mechanism for the creation of certain classes of software artifacts. Note, however, that using inheritance as a reuse-enabling mechanism is not without its own hazards, most notably scalability and the violation of information hiding [23, 24].

4 – Language Mechanisms Supporting Design-With-Reuse

The previous section primarily addressed the *creation* of program structure. Our primary interest in this section involves not the creation of new reusable components, but rather their natural involvement in the development process. This corresponds to the responsibilities of Basili's project organization [3].

4.1 – Procedural and Modular Abstraction

Much of today's reuse takes place at the level of procedures and packages, either as source or object code. The linguistic and environmental mechanisms for this, including source and object libraries and separate compilation, provide little over what a simple text editor with cut and paste commands provides. The onus of comprehension and adaptation is placed upon the reuser, particularly if the reuser is interested in increasing the specificity of the component (which may even be proscribed by the social infrastructure, i.e. management). The consequence of design-with-reuse in this context is thus *monolithic* reuse, an all or nothing acceptance of an entire component.

4.2 – Genericity

Genericity readily supports the creation of specializations of the generic artifact through instantiation. However, genericity as defined in languages such as Ada provides little beyond complete instantiation of a generic component into a completely concrete instance. Further, partial instantiation does little in terms of additional flexibility, as every successive partial instantiation makes the resulting generic more concrete. Hence genericity provides the same form of monolithic reuse as that described in the previous section, with the option of customizing the instances.

4.3 – Inheritance

Inheritance performs as readily in support of a reuser as in support of a developer of components. The reuser can both instantiate new instances of the component and derive new component classes from the original. This second issue is a particularly beneficial one, as it allows for the development of unanticipated refinements to the program model without requiring adaptation of existing code. However, inheritance exhibits the same specificity limitations as abstraction and genericity, supporting only monolithic reuse, in the case of instantiation, or incremental monolithic reuse, in the case of class refinement.

5 – Program Slicing

The mechanisms discussed in sections 3 and 4 *add* structure and/or complexity to a program. Parameterization and genericity increase the interface complexity of a program unit. Packages and inheritance increase either the number of program units or the structural complexity of those units. Hence, current languages do not have explicit mechanisms that address the conflicting goals of design-for-reuse and design-with-reuse. We therefore propose a new mechanism for reconciling the two approaches (by increasing component structural specificity) which works in conjunction with the facilities provided in Ada – a new form of program slicing. We use Ada for our examples, as it is a language whose built-in features facilitate the types of transformations which we invoke. However, the concepts we present are not confined to any particular language.

In his thesis [30], Weiser introduced the concept of program slicing. In this form of slicing, called *static slicing*, a *slice* of a program is an executable subset of the source statements which make up program. A slice is specified by a variable and a statement number, and consists of all statements which contribute to the value of that variable at the end of execution of that statement, together with any statements needed to form a properly executing wrapper around the slice proper.

Dynamic slicing, [1, 2, 17] is a second form of slicing which is determined at runtime and is dependent on input data. A dynamic slice is the trace of all statements executed during a program run using a particular input data set, refined by specifying only those executed statements which reference a specified set of variables. Dynamic slicing was specifically designed as an aid in debugging, and is used to help in the search for offending statements in finding a program error.

By definition, static slicing is a pre-compilation operation, while dynamic slicing is a run-time analysis. Our interface slicing belongs in the category of static slicing, as it is a data-independent pre-compilation code transformation. Since our interest here is only with static slices, henceforth we will use *slicing* to mean static slicing, and we will not again discuss dynamic slicing.

```
 1  procedure wc (theFile : in string; nl, nw, nc : out natural := 0) is
 2  inword : boolean := FALSE;
 3  theCharacter : character;
 4  file : file_type;
 5  begin
 6      open(file, IN_FILE, theFile);
 7      while not end_of_file(file) loop
 8          get(file, theCharacter);
 9          nc := nc + 1;
10          if theCharacter = LF then
11              nl = nl + 1;
12          end if;
13          if theCharacter = ' '
14           or theCharacter = LF
15           or theCharacter = HT then
16              inWord = FALSE;
17          else if not inWord then
18              inWord = TRUE;
19              nw = nw + 1;
20          end if;
21      end loop;
22      close(file);
23  end wc;
```

Figure 1: wc, a procedure to count text

5.1 – Previous Work in Slicing

In his thesis [30] and subsequent work [31, 32, 33], Weiser used slicing to address various issues primarily concerned with program semantics and parallelism. Gallagher and Lyle more recently employed a variation of slicing in limiting the scope of testing required during program maintenance [20].

Program slicing has been proposed for such uses as debugging and program comprehension [32], parallelization [5], merging [12, 18], maintenance, and repository module generation [9].

As an example of program slicing, we present the following example, adapted from Gallagher & Lyle [9]. The procedure `wc`, presented in Figure 1, computes the count of lines, words, and characters in a file.* Figure 2 gives the results of slicing wc on the variable `nc` at the last line of the procedure. Since the variables `nl`, nw, and `inword` do not contribute to the value of `nc`, they do not appear in the slice. Also, the statements on lines 10 through 20 of the original procedure do not

---

* This procedure is not entirely correct, since the Ada *get* procedure skips over line terminators, unlike the C *getchar* function. We adapted wc in this way to clarify its actions and retain the flavor of the original function.

```
procedure wc (theFile : in string; nc : out natural := 0) is
theCharacter : character;
file : file_type;
begin
    open(file, IN_FILE, theFile);
    while not end_of_file(file) loop
        get(file, theCharacter);
        nc := nc + 1;
    end loop;
    close(file);
end wc;
```

Figure 2: wc sliced on nc

appear in the slice. While this slice follows the spirit of a classic slice, and will serve to illustrate classic slicing, it also differs in several important ways, as described below.

5.2 – Interface Slicing

We propose a new form of slicing, *interface slicing,* which is performed not on a program but on a component. Similar to previous work in static slicing, our interface slice consists of a compilable subset of the statements of the original program. The interface slice is defined such that the behavior of the statements and the values of the variables in the slice is identical to their behavior and values in the original program.

However, while previous slicing efforts have attempted to isolate the behavior of a set of variables, even across procedural boundaries, our slice seeks rather to isolate portions of a component which export the behavior we desire. In the following discussion, we assume for simplicity that a package implements a single ADT, and we use package and ADT interchangeably.

Unlike standard slicing techniques which are usually applied to an entire program, interface slicing is done on a fragment of a program – a *component* – since our goal is to employ the necessary and sufficient semantics of a component for use in the target system. Interface slicing is at the level of procedures, functions, and task types. If a procedure is invoked at all, the entire procedure must be included, as we have no way of knowing *a priori* what portion of the procedure will be needed.* However, if an ADT is incorporated into a system, not necessarily all of its operations are invoked. The interface slicing process determines which operations are to be included, and which can be eliminated. Because interface slicing treats procedures atomically, the complex program dependence graph analysis of standard slicing [13] is not necessary. A single pass of the call graph of an ADT's operations is sufficient to determine the slice. We use "operation" as a general term to encompass procedures, functions, and exceptions, and include tasks with procedures in that a task is another way of encapsulating a subprogram unit.

We will illustrate the concept of interface slicing first by examining a simple example, a toggle ADT. First consider package `toggle1`, in Figure 3. This package exports the public operations `on`, `off`, `set`, and `reset`. `On` and `off` are examination operations which query the state of the toggle, while `set` and `reset` are operations which modify the state of the toggle. Now suppose that we wish to have a toggle in a program which we are writing, but we have a need for only three of the four operations, namely `on`, `set`, and `reset`. In standard Ada, we have two choices. We can include the package as is, and have the wasted space of the off operation included in our program. This is the kitchen sink syndrome. Alternatively, we can edit the source code manually (assuming we have access to it) and remove the `off` operation, thereby saving space, but requiring a large amount of code comprehension and introducing the danger of bugs due to hidden linkages and dependencies. In both these cases, we see the generality of design-for-reuse competing with the desired specificity of design-with-reuse.

Instead, we propose the invocation of an interface slicing tool to which we give the `toggle1` package together with the list of operations we wish to include in our program. The tool then automatically slices the entire package based on the call graph of its operations, generating a slice containing only those operations (and local variables) needed for our desired operations. The slice of `toggle1` which contains only the three operations is shown in Figure 4.

---

* In other words, an interface slice is orthogonal to a standard static slice. The use of one technique neither requires nor inhibits the use of the other. We are not discussing the technique of standard static slicing here, other than to contrast it with our interface slice, and so we do *not* assume that an interprocedural slicer is operating at the same time as our interface slicer.

```
package toggle1 is

    function on return boolean;

    function off return boolean;

    procedure set;

    procedure reset;

end toggle1;

package body toggle1 is

    theValue : boolean := FALSE;

    function on return boolean is
    begin
        return theValue = TRUE;
    end on;

    function off return boolean is
    begin
        return theValue = FALSE;
    end off;

    procedure set is
    begin
        theValue := TRUE;
    end set;

    procedure reset is
    begin
        theValue := FALSE;
    end reset;

end toggle1;
```

Figure 3: A toggle package

As another example, consider the package `toggle2`, which in addition to the operations of toggle1 includes the operation `swap`. This package is shown in Figure 5. Suppose we wish to write a program which needs a toggle ADT and the operations `on` and `swap`. The interface slicing tool finds that the operation `on` has no dependencies, but the operation `swap` needs `on`, `set`, and `reset`, and so the desired slice of toggle2 which is produced for our program is contains the four operations, `on`, `set`, `reset`, and `swap`, and does not contain `off`. This slice is shown in Figure 6.

One of the differences between interface slices and standard slices is the way that interface slices are defined. While a standard slice is defined by a slicing criterion consisting of a program, a statement and a set of variables, an interface slice is defined by a package and a set of operations

```
package toggle1 is

    function on return boolean;

    procedure set;

    procedure reset;

end toggle1;

package body toggle1 is

    theValue : boolean := FALSE;

    function on return boolean is
    begin
        return theValue = TRUE;
    end on;

    procedure set is
    begin
        theValue := TRUE;
    end set;

    procedure reset is
    begin
        theValue := FALSE;
    end reset;

end toggle1;
```

Figure 4: The toggle package sliced by on, set and reset

in its interface. The package is an example of design-for-reuse and implements a full ADT, complete with every operation needed to legally set and query all possible states of the ADT. The interface slicer is an aid to design-with-reuse and prunes the full ADT down to the minimal set of operations necessary to the task at hand. The interface slicer does not add functionality to the ADT, as the ADT contains full functionality to start with. Rather, the slicer eliminates unneeded functionality, resulting in a smaller, less complex source file for both compiler and reuser to deal with, and smaller object files following compilation.

6 – 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 examine a pair of generic packages in the public domain. These packages were explicitly written to be used as building blocks for Ada

```
package toggle2 is

    function on return boolean;

    function off return boolean;

    procedure set;

    procedure reset;

    procedure swap;

end toggle2;

package body toggle2 is

    theValue : boolean := FALSE;

    function on return boolean is
    begin
        return theValue = TRUE;
    end on;

    function off return boolean is
    begin
        return theValue = FALSE;
    end off;

    procedure set is
    begin
        theValue := TRUE;
    end set;

    procedure reset is
    begin
        theValue := FALSE;
    end reset;

    procedure swap is
    begin
        if on then
            reset;
        else
            set;
        end if;
    end swap;

end toggle2;
```

Figure 5: Version 2 of the toggle package

programs. The first is a generic package which provides the ADT *set*. 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 members of this type. The package provides all the operations necessary to create, manipulate, query, and destroy sets. The full interface specification of the set is given in Appendix A.

```
package toggle2 is

    function on return boolean;

    procedure swap;

end toggle2;

package body toggle2 is

    theValue : boolean := FALSE;

    function on return boolean is
    begin
        return theValue = TRUE;
    end on;

    procedure set is
    begin
        theValue := TRUE;
    end set;

    procedure reset is
    begin
        theValue := FALSE;
    end reset;

    procedure swap is
    begin
        if on then
            reset;
        else
            set;
        end if;
    end swap;

end toggle2;
```

Figure 6: Version 2 of toggle sliced by on and swap

This *set* package happens to use a *list* as the underlying representation upon which it builds the set ADT, and so requires the second generic package which supplies the *list* ADT. This happens to be a singly-linked list implementation which exports all the operations necessary to create, manipulate, query, and destroy lists. This package also requires two generic parameters, the same ones which *set* requires. The specification for the list package is given in Appendix B.

In the particular list and set packages we used for our example, there were no private operations. Private operations are not available to be used in an interface slicing criterion; only the exported operations in the interface can be in the slicing criterion. In general, however, private operations are treated identically to exported ones during the slicing process. The slicer, being a

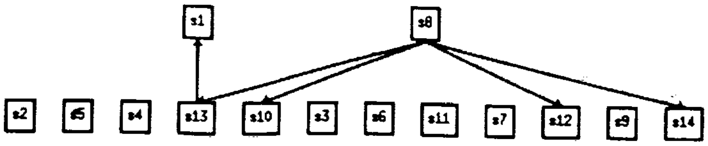

Figure 7: The call graph for set

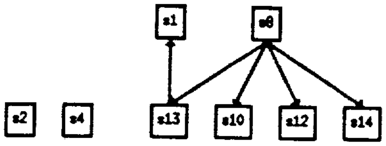

Figure 8: The sliced set

privileged pre-compilation code transformer, does not respect privacy.

6.1 – A Single Level of Slicing

Now suppose we wish to use the set package in a program we are writing, but we have a need for only a few of the set operations, specifically, in this example, *create*, *insert*, and *equal*. We would like to include all the code necessary to accomplish these operations, but would like to have *only* the necessary code, and no more.

In order to slice the set package, we must examine the call graph of operations in the set package for the transitive closure of the three desired operations. Figure 7 shows the complete call graph of the set package, and figure 8, shows the transitive closure of *create*, *insert*, and *equal* (nodes s2, s4 and s8, respectively).* Figure 8 shows the slice corresponding to these three operations. Out of the total of 14 operations exported by the original package, the slice based on *create*, *insert*, and *equal* contains only 8 operations, with a considerable reduction in total size of code, although the complexity of the call graph remains the same.

Notice that in this example, the sliced set package needs the same number and type of generic parameters as did the original package. This will not always be the case, however. In Figure 1, the

---

* The call graph node labels correspond to the comments associated with each operation for the package specifications appearing in the appendices.

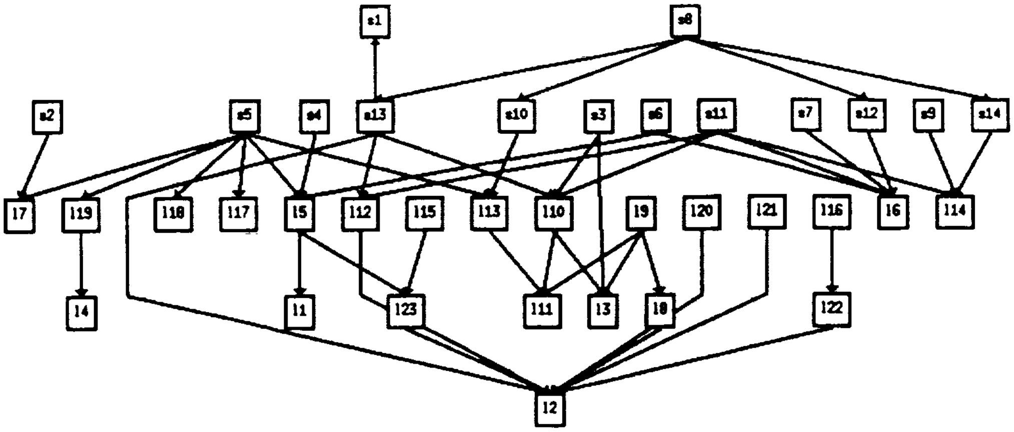

Figure 9: The combined set and list call graph

original wc procedure needed 4 parameters, but the slice based on nc shown in Figure 2 needed only 2 parameters. In general, out of all the local variables in a component, including both variables bound to parameters and those declared within the component's scope, a slice will include a subset of these local variables.

6.2 – A Second Level of Slicing

While the 8 operations represent an improvement over the original 14, we can go further, and examine not only the set package, but also the list package as well. If we examine the transitive closure of the three desired operations in the call graph of all the operations of both the set and list packages, we can accomplish a much more dramatic improvement in the size and complexity of the resulting slice. Figure 9 shows the full call graph of the set and list packages. In standard Ada usage, all of this would be included in a program were the generic set and list packages instantiated in a program. Figure 10 shows the call graph which is exactly the transitive closure of the set operations *create*, *insert*, and *equal*, as would be produced by interface slicing. The size and complexity of this call graph are obviously much less than that of the full graph. Table 1 gives some statistics on the relative sizes of the packages and their call graphs.

None of the examples above involved overloaded names. Interface slicing in the presence of overloading is somewhat more complicated. Assuming that the resolution can be accomplished

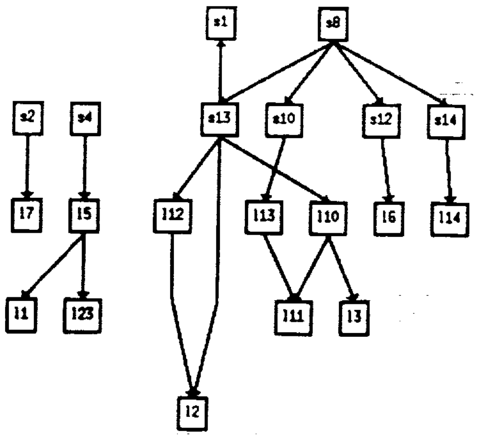

Figure 10: The sliced set and list

Table 1: Package Statistics

	# of nodes	# of edges	# of statements
Full Set	14	5	95
Sliced Set	8	5	57
% reduction	36	0	40
Full Set and List	37	46	345
Sliced Set and list	20	19	200
% reduction	46	59	42

completely at compile time, there are two options. The first is a simple, naive approach in which all versions of an overloaded operation are included. The second is to perform the type checking for parameters and return value (if any) to determine which of the overloaded versions are actually called. For example, assume that list's operation *attach* is a quadruply overloaded procedure which can be called with two elements, an element and a list, a list and an element, or two lists. Resolution of the overloading may, in a particular situation, allow three of the four procedures to be sliced away, resulting in improved reduction of size and complexity.

If the overloading cannot be resolved at compile time, but must wait until runtime, we have no option but to include the code for all possible operations which may be called. A static slice can only blindly assume worst-case in the presence of run-time binding of overloaded procedure names. Although our example extends to only two levels, the slicing can extend to as many levels as exist in the compilation dependency graph of the packages included in the program.

7 – Conclusion: Balancing Genericity and Specificity

We have discussed two main reuse-oriented paradigms in software engineering, namely design-for-reuse and design-with-reuse, and how the goals of these two paradigms have in the past been viewed as being antagonistic, with the former striving for generality and the latter striving for specificity. We have shown that with the proper language mechanisms and development techniques, the goals are in fact complementary. The specific mechanism we use by way of example is a new form of static program slicing which we call interface slicing. Using interface slicing, a complete and generic component can be adapted to the specific needs of the program at hand, increasing comprehension and reducing complexity, without sacrificing the generality of the base component. Thus a developer designing a component for reuse can be completely unfettered of all size constraints and strive for total generality, knowing that a reuser of the components can effortlessly have all unneeded functionality sliced away in a pre-compilation step.

The artifacts produced by an interface slicer should not be considered as new components, any more than instantiations of a generic are viewed as new components. Rather, we want to emphasize the retention of the derivation *specification*, avoiding additional maintenance problems though the life-cycle of what would then be custom components. We should keep the desired interface specification, and alter that when we need to change the way in which we bind through the interface to the base component. Just as we don't associate any cost per se with the instantiation of a generic, we should not associate a cost with specialization through interface slicing, since it can be completely handled by the development environment.

Our approach addresses indirectly a critical social aspect of reuse, the trust that reusers place in the components extracted from the repository [16]. Deriving a family of interface slices from a base component implies that if the base component is correct (or at least certified), then all of the slices must necessarily be correct (or at least certified) also.

References

1 H. Agrawal and J. Horgan, *Dynamic Program Slicing,* Technical Report SERC-TR-56-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, December 1989.

2 H. Agrawal, R. DeMillo, and E. Spafford, *Efficient Debugging with Slicing and Backtracking,* Technical Report SERC-TR-80-P, Software Engineering Research Center, Purdue University West Layfayette, Indiana, October 1990.

3 V. R. Basili, G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory," *ACM Transactions on Software Engineering and Methodology,* 1(1), January 1992, p. 53-80.

4 D. Batory, "On the Differences Between Very Large Scale Reuse and Large Scale Reuse," *Proc. 4th Annual Workshop on Software Reuse,* Reston VA, November 18-22 1991.

5 W. Baxter and H. R. Bauer, "The Program Dependence Graph and Vectorization," *Proc. Principles of Programming Languages: 16th Annual ACM Symposium,* Austin, TX, January 11-13, 1989, p. 1-11.

6 L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys,* 17(4), December 1985, p. 471-522.

7 S. H. Edwards, *An Approach for Constructing Reusable Software Components in Ada,* IDA Paper P-2378, Institute for Defense Analyses, Alexandria VA, Sept. 1990.

8 D. Eichmann, "A Repository Architecture Supporting Both Intra-Organizational and Inter-Organizational Reuse," to be submitted.

9 K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering,* 17(8), August 1991, p. 751-761.

10 E. S. Garnett and J. A. Mariani, "Software Reclamation," *Software Engineering Journal,* (5)3, May 1990, p. 185-191.

11 A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley, 1983.

12 S. Horwitz, J. Prins, and T. Reps, "Integrating Non-interfering Versions of Programs," *Proc. Fifteenth Annual ACM Symposium on Principles of Programming Languages*, New York, January 13-15, 1988, p. 133-145.

13 S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, (12)1, p. 26-60, January 1990.

14 K. E. Huff, R. Thomson, and J. W. Gish, "The Role of Understanding and Adaptation in Software Reuse Scenarios," *Proc. 4th Annual Workshop on Software Reuse*, Reston VA, November 18-22 1991.

15 R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, 1(2), June/July 1988, p. 22-35. Also appears in [22].

16 J. C. Knight, "Issues in the Certification of Reusable Parts," *Proc. 4th Annual Workshop on Software Reuse*, Reston VA, November 18-22 1991.

17 B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, (29)3, p. 155-163, October 1988.

18 A. Lakhotia, *Graph Theoretic Foundations of Program Slicing and Integration*, Technical Report CACS-TR-91-5-5, Center for Advanced Computer Studies, University of Southwestern Louisiana Lafayette, LA, December 2, 1991.

19 H. Li and J. van Katwijk, "A Model to Reuse-in-the-Large," *Proc. 4th Annual Workshop on Software Reuse*, Reston VA, November 18-22 1991.

20 J. R. Lyle and K. B. Gallagher, "A Program Decomposition Scheme with Applications to Software Modification and Testing," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, vol. 2, January 1989, p. 479-485.

21 R. Prieto-Díaz, "Domain Analysis for Reusability," *Proceedings of COMPSAC '87*, p. 24-29. Also appears in [22].

22 R. Prieto-Díaz and G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

23 R. K. Raj and H. M. Levy, "A Compositional Model for Software Reuse," *The Computer Journal*, (32)4, 1989, p. 312-322.

24 A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *Proc. OOPSLA'86*, Portland OR, September 29 - October 2 1986, p. 38-45.

25 I. Sommerville, *Software Reuse*, ISF Study Paper ISF/UL/WP/IS-3.1, University of Lancaster, UK, January 1988.

26 W. Tracz, "Software Reuse: Motivators and Inhibitors," *Proc. of COMPCON '87*, 1987, p. 358-363.

27 W. Tracz, "Software Reuse Myths," *ACM SIGSOFT Software Engineering Notes*, (13)1, January 1988, p. 17-21.

28 W. Tracz, "Parameterization: A Case Study," *Ada Letters*, (IX)4, May/June 1989, p. 92-102.

29 B. W. Weide, W. F. Odgen, and S. H. Zweben, "Reusable Software Components," in *Advances in Computers*, v. 33, M. C. Yovits (ed.), Academic Press, 1991, p. 1-65.

30 M. Weiser, *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*, PhD Thesis, University of Michigan, Ann Arbor, Michigan, 1979

31 M. Weiser, "Program slicing," *Proceedings of 5th International Conference on Software Engineering*, p. 439-449, May 1981.

32 M. Weiser, "Programmers use slicing when debugging," *Communications of the ACM*, 25(7), July 1982, p. 446-452.

33 M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, SE-10, July 1984, p. 352-357.

Appendix A – The Package Specification for Set

Note: the comments in the right margin refer to the node labels in the call graphs of Figures 7, 8, 9, and 10.

```
generic
    type elemType is private;
    with function equal(e1, e2: elemType) return boolean is "=";
package setPkg is

    type set is private;
    type iterator is private;

    noMore: exception;                                              -- s1

    function create return set;                                     -- s2

    procedure delete(s: in out set; e: in elemType);                -- s3

    procedure insert(s: in out set; e: in elemType);                -- s4

    function intersection(s1, s2: set) return set;                  -- s5

    function union(s1, s2: set) return set;                         -- s6

    function copy(s: set) return set;                               -- s7

    function equal(s1, s2: set) return boolean;                     -- s8

    function isEmpty(s: set) return boolean;                        -- s9

    function isMember(s: set; e: elemType) return boolean;          -- s10

    function size(s: set) return natural;                           -- s11

    function makeIterator(s: set) return iterator;                  -- s12

    procedure next(iter: in out iterator; e: out elemType);         -- s13

    function more(iter: iterator) return boolean;                   -- s14

end setPkg;
```

Appendix B – The Package Specification for List

Note: the comments in the right margin refer to the node labels in the call graphs of Figures 9 and 10.

```
generic
    type elemType is private;
    with function equal(e1, e2: elemType) return boolean is "=";
package listPkg is

    type list is private;
    type iterator is private;

    circularList: exception;                                    -- 11
    emptyList: exception;                                       -- 12
    itemNotPresent: exception;                                  -- 13
    noMore: exception;                                          -- 14

    procedure attach(l1: in out list; l2 in list);              -- 15

    function copy(l: list) return list;                         -- 16

    function create return list;                                -- 17

    procedure deleteHead(l: in out list);                       -- 18

    procedure deleteItem(l: in out list; e: in itemType);       -- 19

    procedure deleteItems(l: in out list; e: in itemType);      -- 110

    function equal(l1, l2: list) return boolean;                -- 111

    function firstValue(l: list) return itemType;               -- 112

    function isInList(l: list; e: itemType) return boolean;     -- 113

    function isEmpty(l: list) return boolean;                   -- 114

    function lastValue(l: list) return itemType;                -- 115

    function length(l: list) return integer;                    -- 116

    function makeIterator(l: list) return iterator;             -- 117

    function more(l: iterator) return boolean;                  -- 118

    procedure next(iter: in out iterator; e: itemType);         -- 119

    procedure replaceHead(l: in out list; e: itemType);         -- 120

    procedure replaceTail(l: in out list; newTail: in list);    -- 121

    function tail(l: list) return list;                         -- 122

    function last(l: list) return list;                         -- 123

end listPkg;
```