NASA-CR-192761

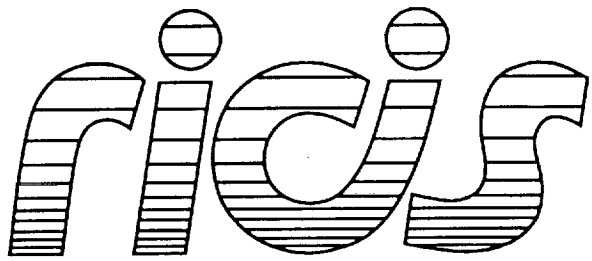
Integrating Interface Slicing Into Software Engineering Processes

Jon Beck
West Virginia University Research Corporation

August 15, 1993

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

NASA Technology Utilization Program
NASA Headquarters



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.

Integrating Interface Slicing Into Software Engineering Processes

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

15 August, 1993

1 Introduction

Interface slicing is a tool which has been developed to facilitate software engineering. As previously presented [5], it has been described in terms of its techniques and mechanisms. In this paper, we consider the integration of interface slicing into specific software engineering activities by discussing a number of potential applications of interface slicing. The applications we discuss specifically address the problems, issues, or concerns raised in [3]. Because a complete interface slicer is still under development, these applications must be phrased in future tenses. Nonetheless, the interface slicing techniques which have been presented can be implemented using current compiler and static analysis technology. Whether implemented as a standalone tool or as a module in an integrated development or reverse engineering environment, they require analysis no more complex than that required for current system development environments. By contrast, conventional slicing is a methodology which, while showing much promise and intuitive appeal, has yet to be fully implemented in a production language environment despite 12 years of development.

2 Code and Executable Size Reduction

Interface slicing has the ability to substantially reduce the size of both source code and executable programs by eliminating unused code. Conventional compiler technology spends considerable effort in performing optimizations to improve the speed and reduce the size of programs [1]. However, these efforts have previously been almost exclusively confined to very local transformations such as the use of constant propagation to remove one of the clauses of a branch statement. Interface slicing can be incorporated into present compilers to perform size optimization in a way which has not previously been performed.

Beyond compiler optimization, however, a reduction in actual size of reused components without any reduction in available functionality has significant implications for the practicality of repository-based component reuse. Domain engineers can develop components which have the full functionality deemed proper, and even redundant functionality accomplished by different techniques, without any concern about the resulting component size. Subsequent interface slicing will ensure that only the functionality specifically needed in each system will be incorporated into each system. It is well established that smaller component size facilitates component reuse, but small component size also captures less domain and architecture knowledge [6]. Interface slicing allows large components with much functionality and captured domain knowledge in the repository, while also allowing a system which reuses them to be as small as if it were composed of custom-designed components.

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.

3 Domain Management and Update Granularity

In the domain-specific, centrally managed reuse repository scenario of software development advanced by DoD, a responsibility of the domain manager is to keep track of the clients of each reused artifact, so that the client can be updated when changes to the artifact occur [10]. Interface slicing eases the burden of accomplishing this for the domain manager by providing a smaller granularity of components for the repository clients.

To see this, consider the situation in which clients *A* and *B* both use module *M* from the repository; *M* consists of elements *x*, *y*, and *z*; *z* is mutually independent of *x* and *y*, while *x* and *y* are mutually dependent. Assume that *A* uses the functionality of *x* and *z*, while *B* uses the functionality *y* and *z*. This situation is illustrated in Figure 1(a). Without interface slicing, any modification to *x*, *y*, or *z* requires that both clients *A* and *B* be updated by the domain manager. But with interface slicing, the domain manager can more precisely know when each client must be updated. In this example, if *x* is modified, only *A* need be updated, and if *y* is modified, only *B* need be updated. A modification to *z* requires updates to both *A* and *B*. The situation with interface slicing is pictured in Figure 1(b). For the software development enterprise as a whole, this finer effective granularity of the reused components results in far less time spent in re-incorporating modified components, with the attendant retest-

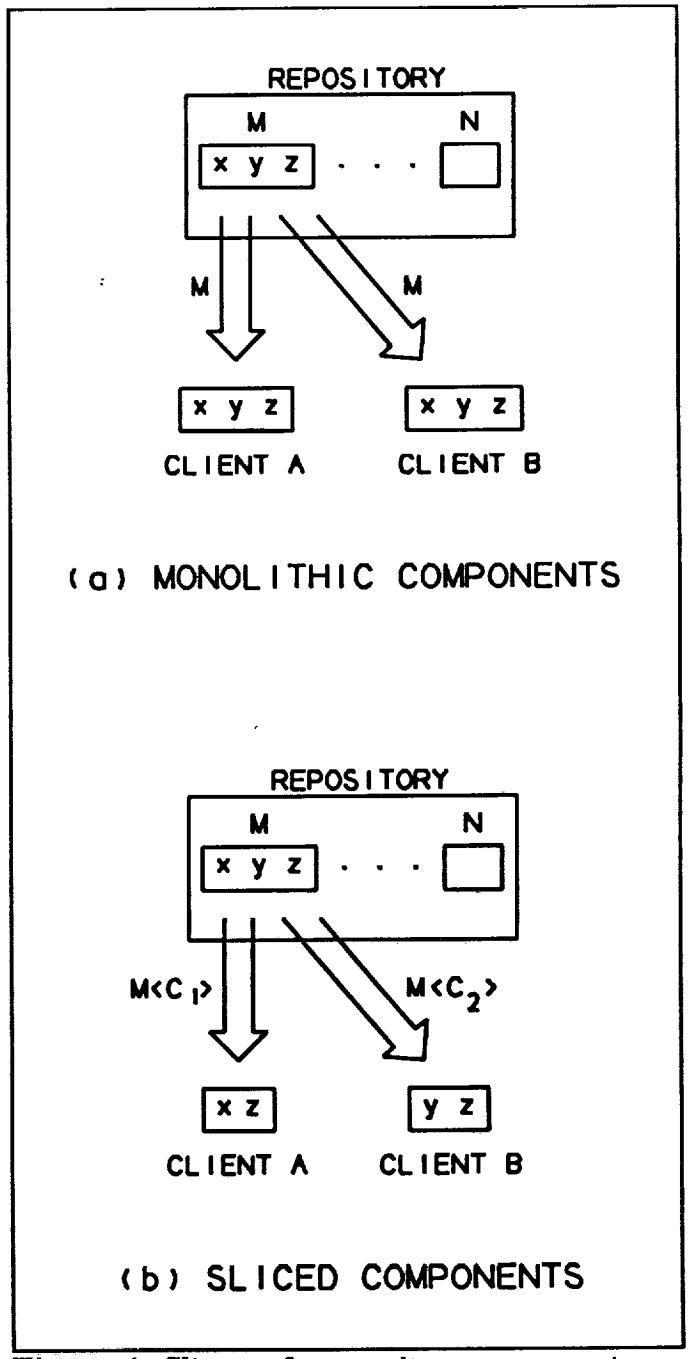


Figure 1 Clients of a repository component

ing, revalidating, and recertification of the client software systems.

4 Verbatim Reuse vs Customized Components

The benefits of using centrally managed and engineered components strongly favor verbatim reuse, that is, reuse of components as supplied by the domain engineer without local modification by the application engineer. This protects the purity of the domain model and greatly reduces the possibility of modification-induced error, as well as simplifying the process of future modification. Unfortunately, this tends to preclude the use of customized components. Interface slicing provides a mechanism for supplying customized components by customizing them automatically at system composition time, without manual changes, in effect combining the benefits of both verbatim reuse and customized components.

This can be viewed as analogous to the way that high-level source code maintains logical program design, while the optimizing pass of the compiler is allowed to violate the logical design by performing all sorts of macro expansions, loop unrollings, and common sub-expression eliminations, combining the benefits of high-level logical design and optimized code. But the logical design is never actually violated, because the base source code, which the programmer sees, is never altered. In the same way, interface slicing does not alter the base component which is supplied by the domain engineer, but rather performs its function at compile time, just as does the compiler's optimizer.

5 Reverse Engineering

In the previous section we mentioned program modification. Maintenance modifications are an integral part of software engineering; making modifications requires a large amount of program knowledge and comprehension to be successful. Interface slicing can aid in program knowledge and comprehension in two specific ways. First, to the extent that interface slicing can be used to reduce the overall size of a program, it is useful for program comprehension, as in general a smaller program is easier to comprehend than a large one, other variables being held constant. But more importantly, especially for the reverse engineering of legacy code which was originally created without the benefit of an interface slicer, interface slicing can be incorporated into a tool which generates a report detailing 1) the elements of one module which another module references, both directly and transitively, and 2) which other elements of a module can possibly be influenced by modifications to each specific module. This latter functionality is also provided by Gallagher and Lyle's poset of static slices, but at a different level of granularity.

In a more complex example of the use of slicing for reverse engineering, consider the common desire to update and modernize a legacy system by a transformation from an old strictly modularized and hierarchical design implemented in an old language lacking information hiding mechanisms to a system characterized by an object-oriented design and implementation. Object-oriented design is characterized by very different design methodologies and criteria than were used for legacy hierarchical designs. In particular, the behavior of an object which is encapsulated within the object tends to cut across the

boundaries of traditional modularization in a manner very reminiscent of the way that slicing cuts across the same boundaries. Eichmann [11] has suggested that in fact this resemblance is not coincidence and that slicing can be used specifically to reengineer the designs of legacy systems into object-oriented designs.

To see this, consider a traditional hierarchical payroll accounting system. Code which affects an employee is scattered throughout the system, spread widely into different packages and subprograms. One subprogram calculates the employee's current paycheck, another subprogram keeps track of the employee's hours allocated to different projects, while a third is used to update the employee's home address. In a reengineering of this system into an object-oriented design, *Employee* might be a class, with each employee an instance of that class. In this case, all the system's functions which affect the employee might be encapsulated within the *employee* objects. Manually searching the old system for all the code which affects employees is an arduous task. Slicing can help automate this process. Giving a slicing criterion consisting of all the functionality which affects an employee, the slicer can project from the system just that functionality, with its supporting code, ready to be encapsulated into the new *employee* object.

6 Issues of Program Design

As explained in [4], the implementation of a module determines the interface slices of that module. Specifically, the pattern of dependences among the module's elements determines how many interface slices exist, and the dependence relationships among those slices. This leads directly to two different ideas relating interface slices to program design. The first is to use interface slicing as a metric of existing program design. Possible module characteristics which could be evaluated include reliability, maintainability, portability, and reusability. The second is to consider how our notions of "good" module design may change with the knowledge that interface slicing is available in the development environment. For example, redundant functionality in a module, might change from being considered an undesirable characteristic to a desirable one were an interface slicer known to be available.

Many researchers have considered the problem of designing "good" reusable components (e.g., [8,12,13]), as well as the problem of what constitutes a "good" reusability metric (e.g., [2,7,9]). However, all of this work has been done in the context of assuming that the entire reusable module will be incorporated into a software system under development, without consideration of a tool such as interface slicing. Adding interface slicing to the development environment adds a new variable, potentially causing some assumptions and conditions to change, and thus potentially requiring a re-evaluation of what is considered to be "good" program design.

7 Extensions of Interface Slicing

In the discussions of interface slicing in [5] the terminology and examples implied an Ada environment. Clearly, the ideas have a much wider reach than one language, but more work is needed to delineate the core language-independent features, and those which are

more language-dependent. For example, the extent to which interface slicing can be profitably applied to C object libraries and FORTRAN subprograms has yet to be explored.

It is well established that the more abstract the artifact being reused, the greater the payoff from that artifact's reuse in a software development effort. This leads to the idea that applying the concepts of interface slicing to formal specifications or formal requirements would yield greater benefits than claimed here for interface slicing of code artifacts. This applies as much to conventional slicing as to interface slicing.

We point out that we have phrased this discussion of the interface slicer as though it were a standalone pre-compilation code transformer; in fact, for effectiveness it should be implemented as a portion of an integrated development environment, or as one member of a suite of an integrated toolset, with full access to the libraries and databases of the environment.

8 Integration of Conventional and Interface Slices

Dynamic and static program slicing have been presented in the literature as competing technologies. By contrast, interface and static slicing are complementary technologies. They work at different levels of granularity; the programmatic entities upon which they operate are different. In an earlier paper [5], we presented experimental evidence that an interface slicing pass can reduce code size by over 60%. Given the concerns that static slicing may be difficult to perform on large modules, the possibility exists that a first pass of interface slicing will sufficiently reduce the size and complexity of the module to facilitate easier static slicing. However, it is possible that a marriage of the two techniques in an integrated environment would provide a greater than additive benefit in terms of system comprehension, regardless of the impetus for that comprehension, than would be expected by a simplistic first-pass - second-pass concatenation of interface and static slicing. This is because the two forms of slicing operate at different levels of program structure granularity, and therefore provide two different views of the program structure. These two views are not orthogonal, rather they may be likened to two different levels of magnification along the same axis. Conventional slicing provides a view focused at the statement level, while interface slicing provides a view focused at the level of calls and references among subprograms and global variables.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[2] ASSET. *Criteria and Implementation Procedures for Evaluation of Reusable Software Engineering Assets*, The National Software Technology Repository, IBM, March 1992.

[3] J. Beck. "A survey of program slicing for software engineering," White Paper, NASA Cooperative Agreement NCC-9-16, project RICIS RB.10, subcontract 118, April 1993.

[4] 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, West Virginia, 1993.

[5] J. Beck. "The theory of interface slicing," White Paper, NASA Cooperative Agreement NCC-9-16, project RICIS RB.10, subcontract 118, May, 1993.

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

[7] G. Boetticher, K. Srinivas, and D. Eichmann. "A neural net-based approach to software metrics," *Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering*, (San Francisco), 16-18 June 1993.

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

[9] CARDS. *Library Development Handbook*, Central Archive for Reusable Defense Software, 7 October 1992.

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

[11] D. Eichmann. Personal communication, 1992.

[12] J. Hollingsworth. "Software component design-for-reuse: a language-independent discipline applied to Ada," Ph.D. Dissertation, The Ohio State University, 1992.

[13] S. Litvintchouk and A. Matsumoto. "Design of Ada systems yielding reusable components: an approach using structured algebraic specification," *IEEE Transactions on Software Engineering*, vol SE-10, no 5, September 1984.