ACHIEVING SOFTWARE QUALITY THROUGH REUSE

John C. Knight
Aaron G. Cass

Computer Science Report CS-95-40
September 1995

ACHIEVING SOFTWARE QUALITY THROUGH REUSE†

John C. Knight and Aaron G. Cass

(knight | agc3u)@virginia.edu

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Contact Author:

John C. Knight
Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903

(804)982-2216
knight@Virginia.EDU

†. Supported in part by the National Science Foundation under grant number CCR-9213427, in part by NASA under grant number NAG1-1123-FDP, and in part by Motorola.

ABSTRACT

Software reuse is advocated primarily as a technique to improve programmer productivity. Reuse permits various artifacts of software development to be used on more than one project in order to amortize their development costs. Productivity is not the only advantage of reuse although it is the most widely publicized. By incorporating reusable parts into a new product, the parts bring with them whatever qualities they possess, and these can contribute to the quality of the new product. This suggests that reuse might be exploited for improving dependability as an entirely separate goal from improving productivity. If useful properties pertaining to dependability could be shown to be present in products as a direct result of software development based on reuse, this might be a cost-effective way of achieving those qualities irrespective of the productivity advantages.

In this paper, we address the issue of certifying reusable parts and exploiting certification to establish properties of systems. We advocate the development of software by reuse with the specific intent of establishing as many of the required properties in the final product as possible by depending upon properties present in the reusable parts. For this goal to succeed, a precise definition of certification of reusable parts is required together with a detailed mechanism for the exploitation of certification. We present a precise definition of certification and a development framework for its exploitation. The benefits of the definition and the way in which it supports the goal are explored.

Finally, we illustrate the concept of certification with some examples from a case study. We show how the certification and exploitation processes can be used to demonstrate that a complex system possesses several properties including freedom from memory leaks. This latter property is defined formally as an invariant on program variables and demonstrated using static analysis techniques.

INTRODUCTION

Software reuse is advocated primarily as a technique to improve programmer productivity. Reuse permits various artifacts of software development to be used on more than one project in order to amortize their development costs.

Productivity is not the only advantage of reuse although it is the most widely publicized. By employing reusable parts during product development, they become part of a new product thereby improving productivity as intended. However, the parts also bring with them whatever qualities they possess, and, at least in principle, these contribute to the quality of the new product. Thus, extensive effort expended to establish desirable properties of the reusable parts might permit establishment of the same or similar properties in the product with substantially less effort than would otherwise be required.

This scenario suggests that reuse might be exploited for improving dependability as an entirely separate goal from improving productivity. If useful qualities pertaining to dependability could be shown to be present in products as a direct result of software development based on reuse, this might be a cost-effective way of achieving those qualities irrespective of the productivity advantages. In the limit, there might be qualities that for all practical purposes can only be achieved this way because the cost of establishing the qualities by analysis of the product alone might be prohibitive. In practice, this is similar to the path taken by some theorem proving systems in which libraries of proofs of lemmas and theorems are preserved for use in establishing proofs of new theorems. In many cases establishing proofs directly from the axioms requires infeasible levels of effort. Reusing theorems is a very limited application of reuse, and is not tied directly to the software development method as modern software reuse is.

The adjective certified has been used in a variety of ways in the reuse literature. An early view was that the term should be used to describe parts that have been tested in some way prior to entry into a library (e.g., [11]). Testing parts prior to their insertion into a reuse library is often claimed to be a productivity advantage. There is the vague expectation that building software from tested parts will somehow make testing simpler or less resource intensive, and that products will be of higher quality [2, 7, 11]. For example,

Horowitz and Munson [6] give the potential productivity improvement through reuse for the entire lifecycle. The various aspects of testing are listed, and a potential reduction in cost resulting from reuse is shown for each.

A different view of certification is that it relates to the measured reliability of components and the use of this data to predict system reliability based on how the parts are used within the system [14]. A third view is that certification should be associated with formal verification and that proofs of system properties should be facilitated by formal specifications of reusable assets.

Levelled certification schemes have been introduced to try to provide a broader framework. In a levelled scheme, a small number of quality levels are defined and parts are assigned to the different levels based on their meeting the prescribed qualities associated with each level.

In this paper, we discuss the development of software by reuse with the specific intent of establishing as many of the required quality properties in the final product as possible by depending upon properties present in the reusable parts. For this goal to succeed, a different definition of certification of reusable parts from those in the literature is required because of the emphasis on supporting system-level qualities. Such a definition is presented. The benefits of the definition and the way in which it supports the goal are explored. The key to success of the ideas that we present is the notion of *exploitation*, i.e., using certified reusable parts with the explicit goal of establishing qualities in the systems constructed via reuse. Finally, we give examples of how certification can be applied and exploited to demonstrate properties of systems.

SOFTWARE REUSE

The modern concept of software reuse, sometimes referred to as *systematic reuse*, expands on notions such as the conventional subprogram library by attempting to apply reuse outside of the traditional framework [5, 12]. Reuse under more general circumstances still has economic benefits since costs might be reduced and hence productivity improved even if a part is reused only once or just a few times. If large collections of reus-

able parts were available and a substantial fraction of a new application could be prepared from those parts, there is an obvious financial advantage. The economics of reuse are in fact quite complex. Several models have been produced that attempt to make cost predictions [1, 6] but the intuitive argument that reusing an existing part is likely to be cheaper than building it from scratch is clear.

In software development that incorporates parts-based reuse†, a new software product is constructed to as great an extent as possible by reusing parts (also known as *components*) that have been prepared previously and stored in reuse libraries with the specific goal of their being available for reuse. Parts may be large or small, may be skeleton systems, skeleton subsystems, complete subsystems, complete low-level subprograms, or any other structure that has the potential for being reused. Parts are obtained either by deliberately tailoring them from the outset specifically for reuse or by scavenging them from existing software. A scavenged part might require some refinement in order to increase its potential for reuse before being placed into a reuse library. This will depend to a large extent on whether or not the author of the part planned for reuse when the part was constructed.

When building a new application, suitable parts are located from reuse libraries and combined with custom code in ways prescribed by the reuse process employed. In many cases, parts will have been parameterized by their developers and will require configuration information to be supplied before use. This is referred to as *anticipated adaptation*. In some cases, parts have to modified in ways that the developer did not expect. This is referred to as *unanticipated adaptation*. The techniques to be used will depend on the parts being reused and this information is usually contained in the library with the components. Figure 1 outlines the overall process of building systems using reusable parts.

DEFINING CERTIFICATION

To permit the exploitation of reuse in support of dependability, it is essential that a pre-

†. Parts-based reuse is just one approach to reuse but it is in common use. The other two major approaches are application generation and special-purpose languages.

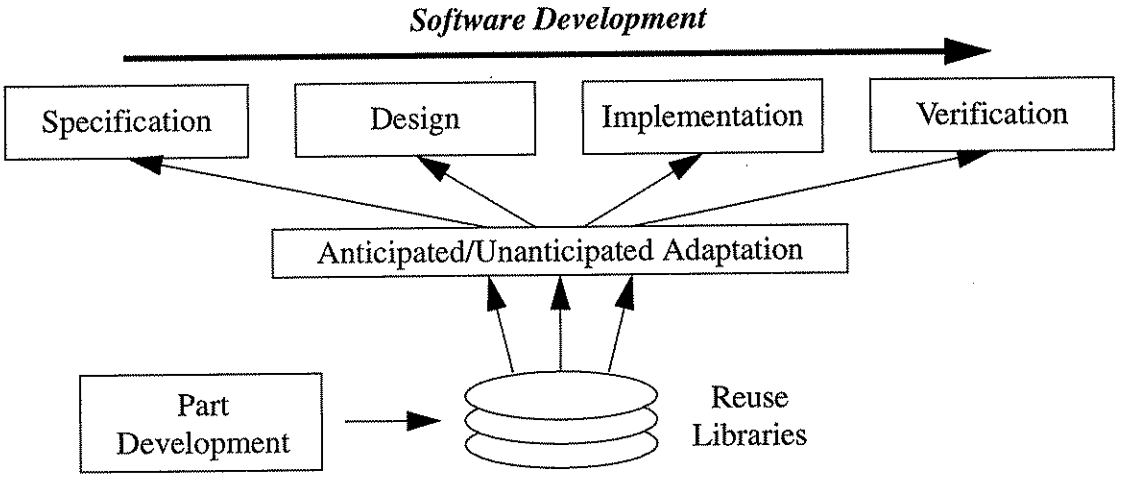

Figure 1. Overview of Reuse

cise definition of certification be available. With no definition, there can be no assurance that parts retrieved from a reuse library possess useful properties nor that different parts possess the same properties. Given the previous notions of certification that have appeared, it is tempting to think that a definition of certification to which reuse libraries in general would comply should be in terms of some test metric or similar. For example, certification might mean that a part has been tested to achieve some particular value of a coverage metric or has a failure probability below some critical threshold.

The major difficulty with this approach and other existing approaches, no matter how carefully applied, is that any single definition that is offered cannot possibly meet the needs of all interested parties. In practice, it will meet the needs of none. Knowing that parts in a reuse library have failure probabilities lower than some specific value is of no substantial merit if the target application requires an even lower value. A second difficulty with using test coverage metrics is that they do not provide the information required to have confidence in the component. A test metric does not inform reusers what situations are being tested for, what assumptions are made, or in what situations the component can be expected to work as desired. A test metric alone does not eliminate the need to investigate the part carefully to determine its usefulness for a particular application. A third difficulty with certification defined in terms of testing is that it does not apply to non-executable parts. Reusing specifications, for example, has a high potential payoff and certification should address the goal of certifying specification parts. Finally, we note that a

definition based on testing omits other important aspects of quality. It is useful in many cases, for example, for parts to possess properties related to execution time. An executable part that has been "tested" might be very poorly engineered or it might provide only mediocre execution-time performance.

With these difficulties in mind, it is clear that a different approach to certification is required. Our goal of exploitation also dictates that certification has to be undertaken with a view to using whatever properties are held by reusable parts to support the assurance that the final product has desired system-level properties. The following are proposed as definitions for use in the context of reuse and are used throughout the remainder of this paper:

Certification Instance — A certification instance is a set of properties that can be possessed by the type of part that will be certified according to that instance.

Certified Part — A part is certified according to a given certification instance if it possess the set of properties prescribed by that instance.

Certification — Certification is the process by which it is established that a part is certified.

Informally, these definitions define certification in terms of a set of properties that parts in a library are expected to have. Any properties can be included to meet the needs of the organization that is building the library, doing the certification, and exploiting the associated properties.

In establishing a certified reuse library, the associated certification instance has to be defined and the process by which these properties are demonstrated has to be created. When developing a part for placement in the library, it is the developer's responsibility to show that the part has the properties required for that library. When using a part, it is the user's responsibility to inquire about the precise set of properties that the part has and to ensure that they meet his or her needs for exploitation.

Initially, these definitions appear to be of only marginal value because no prescribed properties are included. However, it is precisely this aspect that makes the definitions useful. The definitions have three very valuable characteristics:

- *Flexibility* — As many different certification instances can be defined as are required, and different organizations can establish different sets of properties to meet their needs. Although the ability to create different sets of properties is essential, the communication that a single set facilitates within a single organization or project is also essential. Within an organization, that organization's precise and unambiguous instance of certification is tailored to its needs and provides the required assurance of quality in its libraries of certified parts.

- *Generality* — Nothing is assumed about the type of part to which the definitions apply. There are important and useful properties for parts other than source code. For example, a precise meaning for certification of specification parts could be developed. This would permit the requirements specification for a new product to be prepared from certified parts with the resulting specification possessing useful properties, at least in part. Useful properties in this case might be certain aspects of completeness or, for natural language specifications, simple (but useful) properties such as compliance with rules of grammar and style.

- *Precision* — Once the prescribed property list in the certification instance is established, there is no doubt about the meaning of certification. The property list is not limited in size nor restricted in precision. Thus certification can be made as broad and as deep as needed to support the exploitation goals of the organization.

The properties included in a *specific* certification instance can be anything relevant to the organization expecting to use the certified parts. For example, the following is a list of properties, stated informally, that might be used for reusable source-code parts. The mechanisms by which some of these properties would be exploited are discussed below:

- Compliance with a set of prescribed programming guidelines [10].

- Subjected to detailed formal inspection [4].

- Tested to some standard such as achieving a certain level of a coverage metric.

- Compliance with certain performance standards such as efficient processor and memory utilization or achieving some level of numeric accuracy.

INSTANTIATING CERTIFICATION

The definition of certification presented in the previous section provides the various advantages cited, but, since no specific properties are mentioned, it offers no guidance on what a particular instance of certification should be. This raises the issue of exactly which properties should be included by an organization in the instance of certification for its own reuse library or libraries.

The key to the selection of properties is exploitation. Many properties come to mind as being desirable including the examples listed in the previous section. However, since preparation of reusable parts is a major capital undertaking, it is inappropriate to include properties that are not essential. Consider, for example, a required certification property being the existence of a formal proof that a part has some specific quality. This means that each part in a certified library must be accompanied by such a proof. This is likely to raise the cost of developing those parts considerably. Unless the existence of the proofs can be exploited routinely to establish characteristics of systems built using those parts, the proofs are of marginal value at best.

The opposite circumstance is also a factor. If establishing a necessary system quality is facilitated by the reusable parts that the system includes having a certain property, then that property had better be included in the certification instance. The exploitation of certification properties in this manner is critical.

The process of defining a certification instance therefore is to include any and all of those properties that will contribute to establishing qualities of systems built from the certified parts - and no others. The only justification for the inclusion of a particular property in a specific certification instance is that possession of that property by parts in a library contributes to the establishment of useful qualities in systems built from those parts. The certification instance for a specific reuse library is developed from the qualities desired in systems that will be build from that library. Of course, this does not preclude the possibility of a common instance being used for many libraries or "standard" instances being developed for entire domains or classes of application.

This approach to defining a certification instance appears to shift the problem rather than solve it. The original problem was the selection of properties in an instance of certification. The new problem is the determination of desired system qualities from which the instance of certification is derived. However, the system qualities are the ones of real concern, and they are likely, in practice, to be defined by the requirements specification for the system being built. The way in which reuse, certification, and exploitation interrelate in the approach we advocate is shown in Figure 2.

In summary, the parts in a reuse library are certified according to a certification instance for the benefit of the users of the parts. The certification instantiation for a given library, therefore, has to be developed to maximize the value of the reusable parts to the library user. Thus, the properties included in a certification instantiation are determined by the desired qualities of the systems built from the reuse library. The requirement is that establishing the desired *system* properties should be facilitated to the greatest extent possible by the existence of the *part* properties resulting from certification.

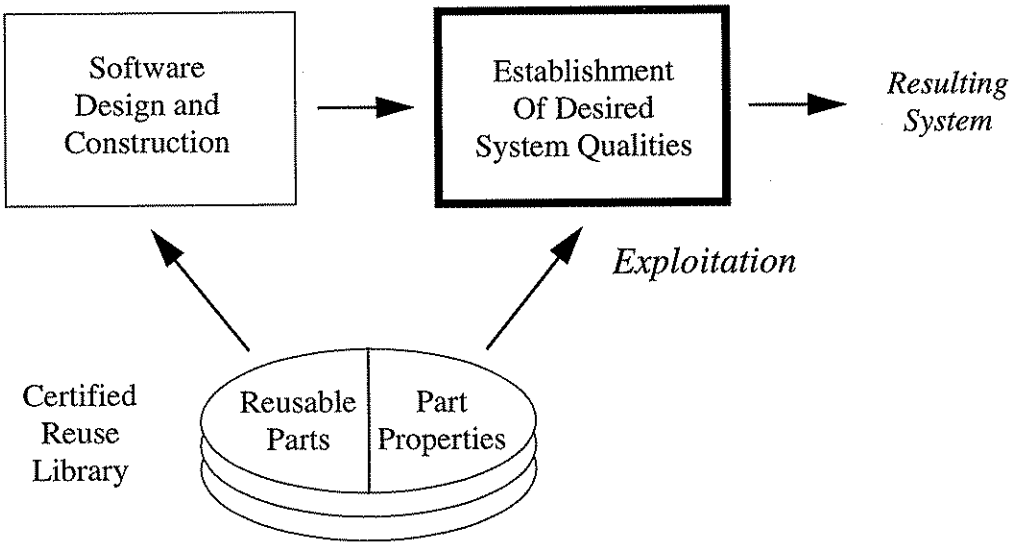

Figure 2. Exploiting Certification in Reuse-based Software Development

EXAMPLE CERTIFICATION INSTANCES

In order to evaluate the approach to achieving quality through reuse that we have described, we are undertaking a demonstration case study. For purposes of illustration, we describe three example properties from the case study that are typical of what would be found in a certification instance that was designed for executable components. The properties address a simple coding standard, real-time performance and memory management, and, as such, span the range from simple to complex. For each system quality, we define the quality itself, define the associated component properties, and describe the exploitation of the component properties in demonstrating the system quality. The majority of the discussion focuses on the most complex quality because it provides the most comprehensive example of the application of certification.

Coding Standards

The first system quality is from a general class of qualities dealing with coding standards. Programmers make many common mistakes that can be avoided by restricting their use of the programming language with which they are building software. It is important (essential in some cases) that a software system comply with some coding standards and assuring that a complete system complies can be facilitated by exploiting a simple certification property.

We use as an example the common pitfall in C and C++ of trying to test for equality but performing an assignment instead. Note that this is merely an example for demonstration. The property could be checked easily in practice by static analysis.

Desired System Property (From Certification Instance)

Conditional expressions do not contain assignment operations.

Associated Component Property (Established For All Components)

Same as system property.

Certification Exploitation

Immediate. That fraction of the final system that is derived from reusable parts is guaranteed to have the desired property if the parts themselves do.

Although both the property and its exploitation seems trivial here, it must be kept in mind that infringements of such rules can lead to subtle faults. Repairing violations of this standard, even if they are intentional, is a tedious and time-consuming process. Exploiting certified parts is a cost-effective approach to ensuring compliance with such standards.

Real-Time Performance

A more complex property that might be required of systems in a number of domains is real-time performance. Determining that a system meets required real-time deadlines is a difficult task that can be facilitated considerably by exploiting properties of certified reusable parts.

In this example, we assume that a system will be built from an available reusable part that supplies the basic real-time architecture of a synchronous system. Such a part, sometimes referred to as a *canonical design*, would contain the necessary timer interrupt handler, for example, and be parameterized to permit any desired basic frame rate. Much of the required additional application functionality would be provided by other reusable parts such as device drivers, function generators, etc., that would be integrated into the system's software architecture. Required functionality that could not be achieved by tailoring reusable parts would be developed as custom software.

Desired System Property

> *The total time for the main loop of the program to complete must never exceed T milliseconds when compiled with the specified target hardware, compiler, and operating system.*

Associated Component Property

> *Guaranteed upper bound on the execution time of component for the specified target hardware, compiler, and operating system.*

Certification Exploitation

> *Compute total loop time based on known system architecture from canonical design, known bounds of functions obtained from reusable parts, and explicit measurements made of custom elements of the system.*

In this example, extensive reuse is being made of the performance data that is devel-

oped once (very carefully) for the reusable parts. For purposes of explanation the description here is simplistic, but, in principle, performance bounds can be computed in this manner provided appropriate, precise certification criteria are included in the certification instance. Other performance bounds that can be computed similarly include average execution times, bounds on maximum size of execution-time data structures, and so on.

Correct Memory Management

The third example property that we discuss concerns memory "leaks" or the apparent loss of dynamically allocated storage. Memory leaks often manifests themselves as progressively worse performance or exhaustion of memory after extended execution times. Both situations are potentially very serious.

Clearly, memory leaks should be prevented, and a system-level quality in which we might be interested could be described informally as "freedom from memory leaks". In order to tackle the problem using the certification and reuse process that we advocate, it is necessary to have a precise definition but achieving precision in this case is surprisingly difficult. Our goal with the process we describe is to show significant properties in a rigorous way through reuse. This example illustrates how complicated this can be.

Informally, many people have the following definition in mind when they speak of a memory leak:

> *A memory leak is a piece of data that the program dynamically allocates but forgets to deallocate.*

This definition, however, is not satisfactory because "forget" is undefined. When can an observer determine that the program has "forgotten" to deallocate the space? If a piece of memory is inaccessible, it is clearly "forgotten" but a piece of memory that is still accessible might also have been forgotten.

The definition used by a tool designed to detect memory leaks [9] is:

> *"Memory leaks are memory chunks that cannot be accessed or freed."*

This definition has the advantage of being somewhat more precise but is also unsatisfactory because it omits storage that is actually accessible but no longer needed.

We introduce a definition that is precise and captures what we think is the essence of the problem. Before proceeding, we note that the definition we use and that we explore in the context of certification does not capture all of the circumstances that are typically associated with the informal notion of a memory leak. The reason is that the informal notion actually includes several distinct ideas and they need to be, and can be, dealt with separately. For purposes of illustration here, we deal with what we consider to be the central issue.

Our definition of a memory leak is that it is a violation of an invariant on the program state. Invariants can be used to describe any relationship between variables that is expected to hold at specific points during program execution. For the purpose of dealing with memory leaks, correct operation of a program includes correct allocation and deallocation of dynamic storage. Storage that is allocated correctly will be used within the various data structures maintained by the program, and, as execution proceeds, these data structures will be manipulated appropriately. Any error in the manipulation of the data structures will result in a discrepancy between state information such as the actual size of a data structure and the size the program "expects" the data structure to be. This is the basis for the definition of the invariant and the associated violation in the event of an error is a memory leak.

As an example of this definition, consider a simple, hypothetical telephone switching system that monitors telephone lines and responds to calls connecting and disconnecting (Figure 3). From the design, we know that the details of current calls are kept in a linked list with one element per call. Thus, the amount of allocated storage, *s*, at the beginning

```
forever do
   for each telephone line do
      check for call
      if new call
         increment call count
         add to list
      if hang-up
         decrement call count
         delete from list
```

Figure 3. A Simple Telephone Call Processing System

and end of each iteration of the loop, given the number of current calls, c, is:

$$s = K \,.\, c$$

where K is a constant equal to the amount of storage required by one element in the linked list.

In practice, this invariant is not what is needed. The problem is that it depends upon knowledge of the actual amount of storage allocated, s, and, in general, there is no way to predict this. Even if there were, the exact value of K will not be known and is subject to change. The invariant that is required is a statement of the required consistency between the different views of memory which are obtained from different parts of the program's state. In this case, one view is the size of the linked list and the other is the actions that cause list elements to be manipulated. The latter is the number of active calls.

The required invariant is a relationship between what we refer to as the *domain variables* of the program and the *implementation variables*. By domain variables we mean parts of the program state that derive directly from the specification. By implementation variables, we mean parts of the program state that derive solely from the specific algorithms used to implement the specification. With this in mind, the required invariant for our simple example can be stated as:

$$\begin{aligned} c &= l \\ &= \textit{number of list insertions - number of list deletions} \end{aligned}$$

where c is the total number of active telephone calls and l is the total number of elements in the list describing the calls. With the invariant stated this way, the effect of the program's various operations should be such that the invariant is true at the top of the loop.

Truth of invariant is only equivalent to the real goal of detecting incorrect use of dynamic storage if the operations used to manipulate the data structure are implemented correctly. But this is precisely what we expect to assume with a reuse-based development process. If the various operations that manipulate the data structure are taken from a reuse library and assumed to be correctly implemented, then the issue at the level of the entire system, is whether or not these operations were actually used correctly.

The invariants with which we are concerned are likely to be quite complex for programs of any size, and one might ask how they will be developed since the processing of data structures in most programs is quite involved. The invariant reflects the developers' understanding of his or her software and, as such, is something that should be fairly straightforward (if time consuming) to develop. If it is not, then there is probably more wrong with the program than mere memory leaks.

Recall that the purpose of this discussion of the meaning of memory leaks was to permit a precise definition for the system quality that would be sought for systems developed with a reuse process based on certified reuse libraries. We are in a position now, therefore, to define the system property of interest:

Desired System Property

> *At predefined points during execution, for each data structure created by a program, the size computed from the current state of the data structure should be equal to the size computed from the sequence of operations that were used to create the data structure.*

Associated Component Property

> *Correct implementation of functional specification.*
> *The component only modifies local variables or its parameters.*

In order to demonstrate the property, the effect that the component has on the invariant will be required. Therefore, the component property for each component is a description of what aspects of the state it modifies and how it does so. This will usually be in the form of a specification.

We now turn to the issue of exploitation for this property. What is required is to show the truth of the invariant from knowledge of the included parts. In principle, this can be done with standard verification techniques, but, for programs of any size, this approach is almost certainly infeasible. The invariant likely to be required will be a complex expression consisting probably of a number of conjuncts, one per data structure. As if this were not bad enough, it is also likely to be the case that the invariant will only be true at one or a small number of points in a program and that these points will be located high in the call graph. Thus establishing the invariant will probably require analysis of the entire program.

The key observation about this situation is that it is precisely this complexity that reuse and the associated certification is designed to mitigate. If the major design aspects of the program were obtained from a canonical design (as was the case in the discussion dealing with real-time performance) then the canonical design will have been designed with the establishment of this property in mind. Thus, for example, the manipulation of the data structures will have been carefully hidden.

In addition, we note that dealing with the required proof of the invariant is facilitated in practice by the fact that the invariant, though perhaps complex, will usually involve very little of the program's state. We need to be concerned only with those operations that manipulate dynamic storage and proving the invariant requires analysis of those program paths that invoke these operations. Thus the problem can be reduced substantially by *slicing* [13] the program (but *not* the lower-level reusable parts) at the location of the invariant on the variables in the invariant. This will yield precisely that part of the program that affects the invariant and any proof techniques can be limited to dealing with this slice.

Slicing can be used to simplify the problem further if it can be assumed that data structures are independent. The conjuncts that constitute the invariant can be viewed as separate invariants in that case and can be used as separate slicing criteria. Slicing and the associated proof can then be undertaken for each conjunct separately.

Even with this degree of simplification, we have found that establishing proofs of the various invariants for a few sample programs that we have studied is still difficult. Despite the removal of significant fractions of a program by slicing and depending on high-level information about the reusable parts, the fraction of the program that remains is still sufficiently large that establishing a formal proof in most cases remains challenging.

In this situation, the exploitation goal remains the same but to make the process practical, we turn to non-formal approaches based on inspection. Convincing arguments can be made quite effectively about the truth of invariants by human inspection of the slices that the invariant produces.

Certification Exploitation

> *Slice program at the location of the invariant on the variables in the invariant. Verify the invariant either using formal verification or rigorous argument based on inspection.*

Our experience that has led us to the conclusions described above includes the development of a simple prototype slicing tool (based on the tool architecture used by Dunn and Knight [3]) designed to slice C++ programs developed with reusable parts. Using this tool in part and human effort for the remainder, an implementation of the telephone switch example has been sliced and its memory-use invariant verified. The verification was undertaken with PVS [8] although this was quite an involved process.

To evaluate the feasibility of using slicing and inspections to verify the required invariant, we are exploring the techniques with a second program. This second program is approximately 6,000 lines long with several user-interface views of the state, and was written by graduate students as part of a class project. Very little of this program is involved with dynamic memory management, and, the total size of the slices needed to verify the invariant are expected to be only a few hundred lines. Establishing rigorous arguments of the truth of the invariant by inspection in that case seems imminently feasible at this point.

CONCLUSIONS

Software reuse can be exploited to improve dependability entirely separately from the highly publicized goal of using it to improve programmer productivity. The reuse of parts that have been shown to possess desirable properties has the potential for conveying those properties to the product in which the parts are used. This information can then be exploited to help establish desirable qualities in the final product.

To do this effectively requires a precise framework for dealing with part properties, a topic typically referred to in the reuse literature as certification. Such a framework has been presented and explored. A by-product of the use of this framework is that it provides a means of documenting the properties possessed by reusable parts. Within a development organization this permits users of reusable parts to have confidence in the parts, confi-

dence that is usually missing. This is expected to facilitate systematic reuse considerably.

It is not uncommon for software systems to be sufficiently complex that it is difficult to ensure the correct management of dynamic memory, and errors in this management can lead to significant performance problems. We have demonstrated how the framework for part properties can be used and exploited to help establish that systems manage memory correctly. In exploring this property we have developed a general framework from properties based on invariants between domain variables and implementation variables. These invariants are of concern because it is violations of this type of invariant that affect the visible operation of the system.

Finally, we conclude that the general goal of achieving dependability through reuse is imminently possible and cost effective, and a significant supplement to the traditional goal of reuse, namely productivity.

ACKNOWLEDGMENTS

We gratefully acknowledge the support we have received for this work from the National Science Foundation under grant number CCR-9213427, from NASA under grant number NAG1-1123-FDP, and from Motorola.

REFERENCES

1. Barnes, B., T. Durek, J. Gaffney, A. Pyster, Framework and Economic Foundation for Software Reuse. Proceedings the Workshop Software Reusability and Maintainability, National Institute of Software Quality and Productivity, October, 1987.
2. Bassett, P.G., Frame-Based Software Engineering IEEE Software July, 1987.
3. Dunn, M. and J. Knight. Automating the Detection of Reusable Parts in Existing Software. *Proceedings of the 15th International Conference on Software Engineering*. Baltimore Maryland. 17-21 May 1993. Pages 381-390.
4. Fagan, M.E., Advances Software Inspections IEEE Transactions Software Engineering Vol. SE-12, No. 7, July 1986.
5. Freeman, P., (editor), Software Reuse: Emerging Technology IEEE Computer Society Press, 1988.
6. Horowitz, E. and J.B. Munson, Expansive View Reusable Software IEEE Transactions Software Engineering Vol. SE-10, No. 5, September 1984.
7. Lenz, M., H.A. Schmid, and P.F. Wolf, Software Reuse Through Building Blocks IEEE Software July, 1987.
8. Owre, S., N. Shankar, and J. M. Rushby. User Guide for the PVS Specification and Verification System. SRI International, Menlo Park, CA, 1993.
9. Pure Software. *Purify User's Guide*. Pure Software, Inc., Sunnyvale, CA, 1994.
10. Software Productivity Consortium, Ada Quality and Style: Guidelines for Professional Programmers, Van Nostrand Reinhold, 1989.
11. Tracz, W., Software Reuse: Motivators and Inhibitors Proceedings COMPCON 1987.
12. Tracz, W., (editor), Software Reuse: Emerging Technology IEEE Computer Society Press, 1988.
13. Weiser, M. Program Slicing. *IEEE Transactions on Software Engineering*. Volume SE-10, Number 4, July 1984. Pages 352-357.
14. Wohlin, C and P. Runeson, Certification of Software Components *IEEE Transactions on Software Engineering*. Volume 20, Number 6, June 1994. Pages 494-499.