The Certification of Reusable Components

John C. Knight

Computer Science Report No. TR-91-09
April 5, 1991

THE CERTIFICATION OF REUSABLE COMPONENTS

John C. Knight

Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903
(804) 924-7605
knight@virginia.edu

ABSTRACT

Software reuse is being pursued in an attempt to improve programmer productivity. The concept of reuse is to permit 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 components into a new product, the components 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.

The adjective *certified* is sometimes used to describe components that have been tested in some way prior to entry into a library but the term certified is not formally defined in the reuse literature. In this paper, we address the issue of certifying reusable components. 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 components. For this goal to succeed, a precise definition of certification of reusable components is required and such a definition is presented. The benefits of the definition and the way in which it supports the goal are explored.

Keywords and Phrases: software reuse, reusable components, component certification.

1. INTRODUCTION

Software reuse is an emerging technology that is being pursued in an attempt to improve programmer productivity [8, 23]. The primary goal of reuse is to permit various artifacts of software development, such as partial designs and sections of source code, to be used on more than one project in order to amortize their development costs. Many social, managerial, and technical difficulties have to be overcome if reuse is to become an accepted part of routine software development [21, 22]. However, given current software development costs and the general lack of improvement in productivity that has been achieved by other means, reuse remains an important candidate technology for productivity improvement.

A substantial difficulty that appears to be limiting reuse is a lack of perceived quality in the artifacts being reused. Although a software engineer may have available a library of useful artifacts or *reusable components*, there is frequently a reluctance to use them because of concerns about quality. Essentially, the engineer feels that without a lot of knowledge of the component, he or she would be better off rebuilding it. The argument that is often made to justify rebuilding is that it is quicker to rebuild than to try to understand the function and interface of the available component. That lack of perceived quality is a detractor from reuse is an observation based only on anecdotal evidence but appears to be the software-reuse manifestation of the ''not-invented-here'' syndrome.

Productivity is not the only advantage of reuse although it is the most widely publicized. By employing reusable components during product development, the components become part of a new product thereby improving productivity as intended. However, the components 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 components 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 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 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* is sometimes used to describe components that have been tested in some way prior to entry into a library (e.g., [21]). Testing components 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 components will somehow make testing simpler or less resource intensive, and that products will be of higher quality [3, 12, 21]. For example, Horowitz and Munson [10] 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. Despite these various discussions of testing and reuse, the term certified is not formally defined in the reuse literature[†].

In this paper, we address the issue of certifying reusable components. 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 components. For this goal to succeed, a precise definition of certification of reusable components is required and such a definition is presented. The benefits of the definition and the way in which it supports the goal are explored. An important byproduct of a precise definition of certification is that it provides a mechanism for *communication* about component quality between the developer of a component and users of the component. Users no longer have to question the quality of components - certification describes for the prospective user exactly what can be expected of a component. This eliminates the ''not-invented-here'' difficulty mentioned above and facilitating higher reuse levels.

2. SOFTWARE REUSE

The modern concept of software reuse, sometimes referred to also as *systematic* reuse, expands on notions such as the conventional subprogram library by attempting to exploit reuse outside of the traditional framework. Reuse under more general circumstances still has economic benefits since costs might be reduced and hence productivity improved even if a component is reused only once or just a few times. If large collections of reusable components were available and a substantial fraction of a new application could be prepared from those components, 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 [2, 10] but the intuitive argument that reusing an existing component is likely to be cheaper than building it from scratch is clear. A second goal with generalized reuse is to extend reuse to include components other than traditional source code. An overview software development based on reuse is shown in Figure 1.

In software development that incorporates reuse, a new software product is constructed to as great an extent as possible by reusing components (also known as *parts*) that have been prepared previously and stored in *reuse libraries* with the specific goal of their being available for reuse. Components 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. Components are obtained either by deliberately *tailoring* components from the outset specifically for reuse or by *scavenging* them from existing software. A scavenged component 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 component planned for reuse when the component was constructed.

When building a new application, suitable components have to be located from reuse libraries in what amounts to a database search process. Present approaches to cataloging and searching rely to a large extent on natural language specifications for components, although there is frequently a domain-specific structure imposed on the perceived library organization to facilitate searching.

No matter what its origin, a component might be suitable for use in a new application immediately upon location or might need to be modified in a process referred to as *adaptation*.

[†]It is important to note that the informal use of the term certification in the context of reuse is entirely separate from other uses in software engineering, for example as a synonym for formal verification [25].

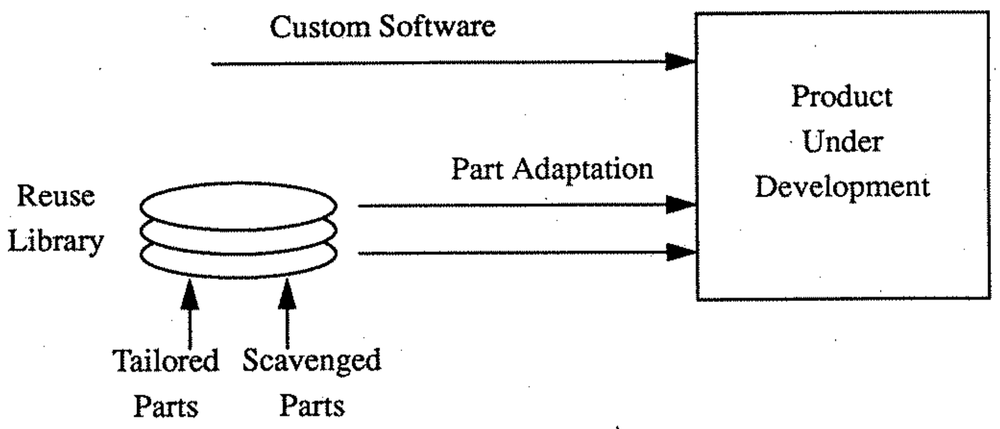

Figure 1 - Software Development With Reuse

In some cases, provision for change is included when a component is written. Adaptation might be as simple as setting a parameter, but could also involve making a substantial modification. For example, before it can be used, a component that implements a desired sort algorithm might require that details of the records to be sorted be defined, including denoting the field to be used for comparison. However, a user of the component might also wish to adapt the component by redefining the order relationship to be used when sorting. This might involve rewriting substantial portions of the part.

Adaptation has been recognized as a necessity for generalized reuse to the extent that provision for it is finding its way into programming languages. Generic program units are present in Ada [24], for example, to support adaptation. The designer of an Ada generic part can parameterize sections of the code and allow the user of the part to specify the details when instantiating it. Symbolic constants and conditional compilation also provide facilities for modifying source text at compile time according to the needs of a particular use.

Adaptation is likely to be extensive with generalized reuse, and ease of adaptation is an important factor in the success of generalized reuse. By contrast, adaptation is almost nonexistent in traditional reuse except when porting. More importantly from the perspective of certification, adaptation of a certified component is likely to affect the certification substantially even with informal notions of certification based on testing.

Finally, even when reuse is consistently and extensively practiced, custom software has to be built for those elements of the application that could not be constructed by reusing components from a library. That custom software might itself be a source of new components for inclusion in a reuse library.

3. CERTIFYING REUSABLE PARTS

Although no formal definition of certification exists in the context of reuse, it is essential that such a definition be available to permit users to trust reusable components and to permit the exploitation of reuse in support of dependability. With no definition, there can be no *assurance* that components retrieved from a reuse library possess useful properties nor that different components possess the same properties. Given the informal notions of certification that have appeared, it is tempting to think that a definition of certification should be in terms of some test metric or similar. For example, certification might mean that the 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, 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 components 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 is that by focusing on a testing-based definition, other important aspects of quality are omitted from consideration. It is useful in many cases, for example, for components to possess properties related to efficient execution.

With these difficulties in mind, it is clear that a different approach to certification is required. The following are proposed as definitions for use in the context of reuse and are used throughout the remainder of this paper:

Certified Part - A part is certified if it possess a prescribed set of properties.

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

In establishing any specific reuse library, the prescribed set of properties has to be defined and the process by which these properties are demonstrated has to be created. When developing a component 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 component, it is the user's responsibility to enquire about the precise set of properties that the component has and ensure that they meet his or her needs.

These definitions appear to be of only marginal value because the prescribed properties are not included. However, it is precisely this aspect that makes the definitions useful. The definitions have three very valuable characteristics:

(1) *Flexibility.*
As many different sets of prescribed properties 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 definition of certification is tailored to its needs and provides the required assurance of quality in its libraries of certified parts.

(2) *Generality.*
Nothing is assumed about the *type* of component to which the definitions apply. There are important and useful properties for components other than source code. For example, a precise meaning for certification of *requirements* components could be developed. This would permit the requirements specification for a new product to be prepared from certified

components 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.

(3) *Precision.*
Once the prescribed property list 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 goals of the organization.

The properties included in a specific definition of certification can be anything relevant to the organization expecting to use the certified parts. As an example, the following are informal statements of properties that might be used for a reuse library of source code components:

- Compliance with a detailed set of programming guidelines such as those prepared for Ada [20].
- Subjected to detailed formal inspection [7].
- 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.

4. TESTING ASPECTS OF CERTIFICATION

Although the approach to certification allows the inclusion of any desired properties, clearly those associated with testing will frequently be present. As with all certification properties, the goal with properties related to testing is to establish them prior to placing components into a reuse library and to use knowledge of these properties to simplify or reduce testing of the resulting product.

Software development based on reuse raises many new testing issues in both the testing of components and the exploitation of tested components. In the context of preparing components for entry into a library, some of the testing issues raised are:

(1) *Component quality.*
By definition, a component that is entered into a reuse library is being offered for use by others and has to be prepared for *every* possible use [18]. This is very different from the normal development situation in which a piece of software is intended for a *single* use and is usually tested with that in mind.

Testing is also complicated by the differing component software structures. Very few reusable components will be subprograms. Other components will be skeleton systems, essentially canonical designs, in which the overall structure of the program is present but the bulk of the detail is missing since it is application specific. Such components are not immediately amenable to traditional techniques of unit testing.

(2) *Adaptable components.*
Adaptable components must be tested before being placed into a reuse library just as every other component must be tested. Concepts such as Ada's generic program units present

significant challenges for testing. The parameters used with Ada generic units are not merely for numeric or symbolic substitution. Subprograms can be used as parameters thereby allowing different instantiations to function entirely differently. This raises the question of exactly how, or even if, generic program units can be tested in any useful way [6].

Once prepared and placed into a reuse library, taking advantage of the testing properties of parts also raises issues, for example:

(1) *Component adaptation.*
Once a component is changed, the results of any testing that took place prior to placing the component in the library cannot necessarily be trusted. That testing might have been extensive and be expensive to repeat in its entirety. This is a special case of the traditional problem of regression testing in software maintenance.

(2) *Component use.*
A reusable component will be used in many different circumstances. The possibility exists, however, that a component may be selected that does not *quite* meet the precise needs of a particular application. Where informal specification techniques are used for components in reuse libraries and reliance is placed on human insight for component selection and matching, it will be difficult to ensure that a selected component does precisely what is required and that the component is being used correctly [9, 16, 17]. This indicates the need for increased attention being paid to integration testing during system development because components will contain assumptions about their use that must be complied with for correct operation. No matter how carefully documentation is prepared, such assumptions are easily violated and the resulting fault is likely to be very subtle.

(3) *Component revision.*
As with any software, a reuse library will be the subject of revision. Parts will be enhanced to improve their performance in some way yet maintain their existing interface. Systems built with such components are then faced with a dilemma. Incorporating the revised components might produce useful performance improvements but the resulting software will differ substantially from that which was originally built and tested. Can revised components with ''identical'' interfaces be trusted, and, if not, what testing needs to be performed when revised components are incorporated?

In summary, the various phases of testing that occur in a traditional development environment are still present but are changed in several ways when development is based on reuse. New testing techniques have to be developed and existing methods have to be enhanced to deal with this situation and to help make software reuse practical. The remainder of this paper addresses one of the more unusual and significant areas, namely adaptable components and adaptation.

5. TESTING AND USING ADAPTABLE COMPONENTS

There are two forms of adaptation that need to be addressed, *anticipated* and *unanticipated*. Anticipated adaptation occurs when a user exploits facilities for change that were designed into the part, such as occurs with an Ada generic component or a component dependent on symbolic parameters. Unanticipated adaptation occurs when a component is modified in a way that was not planned, usually using a text editor.

Both forms of adaptation operate as source-to-source transformations. The output of such a transformation is source text in a programming language, and it is not required to meet any conditions other than the syntactic rules of the language when it is compiled. This is true even of Ada generic units even though the output of the transformation is not accessible to the programmer.

Checking Anticipated Adaptation

In many cases there are restrictions inherent in the design of a component to which any anticipated adaptation must adhere. In the simplest case, a symbolic constant might be used to define a quantity such as the size of an array dimension. Adaptation then consists of setting the symbolic constant prior to using the part, an action that was anticipated by the implementor of the part. The design of the part, however, might necessitate that certain restrictions be imposed, such as the size being within prescribed limits, or having some property, such as the size being a power of two.

Selecting a parameter value may seem to be an innocuous activity. Initially, it seems unlikely that the value selected will be the subject of complex restrictions. However, in a language like Ada, many elements of the operational environment of a program can be controlled by source-text parameters and the values required might be interrelated in non-obvious ways. For example, representation clauses in Ada can be used to define record formats, enumerated type representations, storage available for objects of a given type, and the characteristics of numeric types, among other things. Parameterization of many of these quantities is very likely in a component designed for reuse and the associated interrelationships might be quite involved.

In a more general context, a restriction imposed on an adaptation might be a functional restriction on some piece of supplied program text. A procedure parameter to an Ada generic unit, for example, might be required to meet certain functional constraints inherent in the design of the generic unit. A more complex situation is likely to arise if a component in a library is actually a canonical design. In that case, substantial volumes of code will have to be added to the basic design. The code added might itself be obtained from a reuse library, but will almost certainly have to meet many restrictions imposed by the canonical design.

If they are documented at all, the various restrictions imposed on an adaptation are documented as comments. However, no mechanism is provided in existing production programming systems to permit such restrictions to be checked. Ada does provide static expressions thereby permitting extensive computation to be performed at compile time. Gargaro and Pappas present an excellent example of checking this way in Ada [9]. However, checking restrictions is not the intent of such static expressions, they do not provide the complete range of facilities needed, and there is no mechanism to permit signaling a violation other than forcing a contrived, compile-time exception.

In general, the checking that is required amounts to ensuring that an *implementation* (albeit often a small one) meets a *specification*. Checking an anticipated adaptation is, therefore, a special case of *verification*. The restrictions correspond to the specification and the adaptation itself corresponds to the implementation. It is important to note that the specification in this case does not derive from, and is not related directly to, the original specification for the application. The specification is a consequence of the design of the reusable part.

In a non-reuse setting, this verification will be performed by the author of a part. If the component is placed into a reuse library, however, the checks must be performed by the user.

Correct use then relies on the restriction being documented fully by the author, noticed by the user, and checked accurately by the user. Achieving correct use on a regular basis seems unlikely given this almost total reliance on human effort.

Anticipated adaptation can be dealt with using special-purpose variants of existing techniques that are used for program verification. Just as with verification of complete programs, certain properties of adaptation can be checked completely and others not. For example, it is simple to check that a symbolic constant meets a range or special property criteria. However, it is not possible, in general, to check that a subprogram supplied as a generic parameter complies with required functional constraints.

Checking beyond that inherent in most programming languages is possible using some form of supplementary notation. For example, Anna [14] is a notation designed to permit specifications to be added to Ada source programs. Anna, however, is not designed to perform the kind of verification described here, and although some of the required checking can be specified in Anna, it is not possible to distinguish easily the checks that Anna will perform *before* execution time. For checks that are delayed by the Anna system until execution time, the verification of the various adaptations becomes confused with the verification of the entire program unit that is being executed. Also, such checks require processor and memory resources at execution time, and may not be checked at all unless the assertion is carefully placed. Checking restrictions that derive from the design of a component is an activity that is best performed as a fundamental element of the adaptation process.

A far better approach to checking the constraints required in an anticipated adaptation is to incorporate machine-processable statements of the required restrictions within the source text of the component. Checking for compliance is then performed after adaptation but before traditional compilation. Such a notation can be thought of as an assertion mechanism that operates prior to compilation rather than during execution.

This mechanism will not support the checking of all restrictions, for example many forms of required functionality. Using the analogy with program verification once again, adaptation restrictions that cannot be checked with a pre-compilation assertion mechanism can be dealt with by testing the adapted component but again *prior to conventional compilation*. The concept is to associate with a reusable component a set of test cases that must be executed satisfactorily by any user-specific code supplied during adaptation. The tests will be defined by the author of the component and executed by the user of the part. In the same sense that software testing is an informal approach to verification, this approach is an informal way of assuring that adaptation constraints are met. The overall flow of activities that permit adaptable parts to be used and the associated constraints machine checked is shown in Figure 2.

Checking Unanticipated Adaptation

Arbitrary changes made using an editor are likely to be required frequently in attempting to reuse existing software. Such unanticipated adaptation is far harder to deal with than anticipated adaptation because its effect on the software is unpredictable. There is still the desire, however, to limit the amount of retesting that is needed since a component tailored specifically for reuse is likely to have been subjected to extensive unit testing to ensure component quality. If all the testing carried out previously has to be repeated after adaptation, the economic impact will be severe and could even be a deterrent to reuse. The central difficulty with unanticipated adaptation is to restrict the number of test cases that have to be reexecuted after modification.

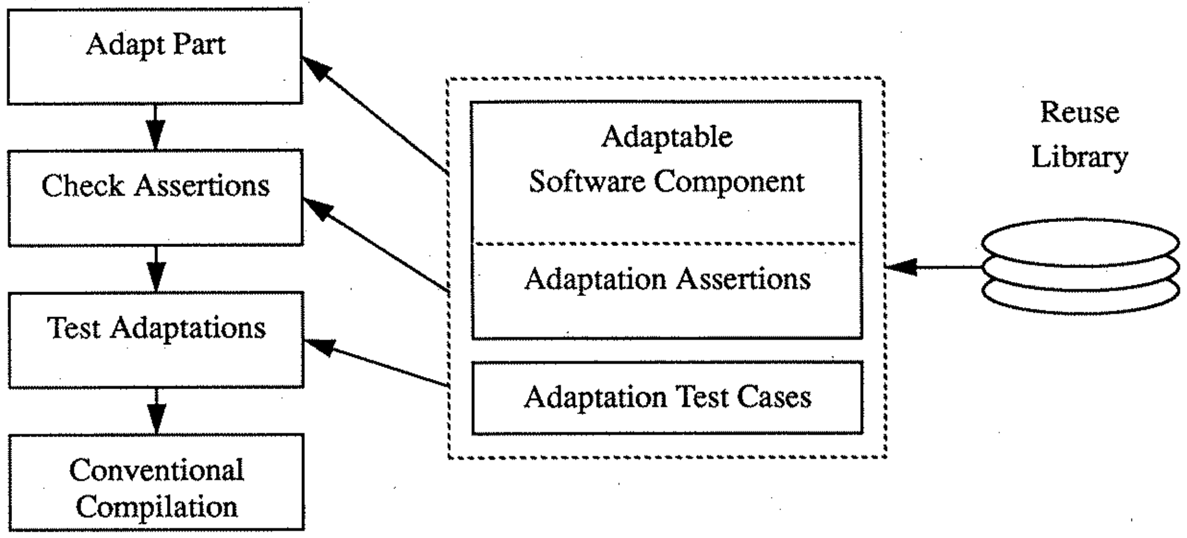

Figure 2 - Checking Anticipated Adaptations

The problem that has to be dealt with in this case is precisely that of conventional program verification. Note, however, that the verification required in this case is quite different from the verification required with anticipated adaptation. A modified component is different from the original component and obviously satisfies different specifications after unanticipated adaptation. If the specifications were not different after adaptation, there would be no point in modifying the component in the first place.

Storing the specification of a component in machine-processable form and modifying the specification along with the component with extensive automated checking and support is the best way to deal with unanticipated adaptation. Unfortunately, in general, this is probably not a practical approach to the problem at this point. However, a promising first approach to dealing with many of the issues, at least partially, is the instrumentation of reusable components with executable assertions [1, 14, 16]. In fact, Anna [14] is described as a notation for specification although it does not have the completeness characteristics of a rigorous approach such as VDM [11]. However, Anna does provide a rich notation for writing executable assertions.

The role of instrumentation using assertions is to include design information with the part, in particular to permit design assumptions to be documented in a machine-processable way. The effects of arbitrary changes cannot be checked with any degree of certainty in this way. However, there is some empirical evidence that executable assertions provide a useful degree of error detection when properly installed [13]. Executable assertions can be used therefore as part of a system for testing components subjected to unanticipated adaptation. Any design assumptions that were not modified correctly during adaptation at least stand a chance of being detected by an assertion failing during test execution.

Testing Adaptable Components

As discussed above, the problem with anticipated adaptation is to ensure that certain requirements imposed by the design of the component are met by the adaptation. The problem of testing adaptable components is the complement of this. It amounts to ensuring that the adaptable component will function correctly assuming that an adaptation complies with the restrictions associated with design of the component.

Adaptable components cannot be executed without adaptation. Each specific adaptation represents a degree of freedom that has to be constrained in order to use the part, and the key question is whether the component will work correctly once these constraints or selections are installed.

The various adaptations that are provided with an adaptable component are similar in many ways to inputs to the component. From the point of view of correctness, setting a symbolic parameter, say, has some of the characteristics of reading an input of the same type as the parameter. The component should, in principle, operate correctly for every valid value of the parameter just as it should for every valid value of an input. Unfortunately, this analogy breaks down when the adaptation provided by the component requires the user to supply functional rather than merely parametric information. In that case there is no notion of type that can be used to determine a valid set of values for the parameter and no obvious selection mechanism for test cases.

The only workable approach at this stage to testing an adaptable component is to instantiate the component with specific adaptations and then test it using some conventional approach to unit testing. Complete testing will then consist of repeating this test process with systematic settings of the various adaptations. A key research issue that remains is to find useful ways of doing this for adaptations that require functions to be supplied.

6. CONCLUSION

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 components 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 used to help establish desirable properties in the final product.

To do this effectively requires a precise framework for dealing with component quality, a topic typically referred to in the literature on reuse as certification. Such a framework has been presented. A byproduct of the use of this framework is that it provides a means of documenting the qualities possessed by reusable components. Within a development organization this permits users of reusable components to have confidence in the components, confidence that is usually missing. This is expected to facilitate systematic reuse considerably.

A number of significant issues arise when considering both the testing of reusable components and the testing of systems incorporating reusable components. The most significant issues arise from the need to deal with adaptable parts, i.e., those designed for change, and adapted parts, i.e. those changed after being taken from a reuse library. To ensure that adaptable parts have been tested prior to placement in a reuse library requires entirely different techniques from those developed for traditional unit testing. Similarly, ensuring that an adaptable part has been adapted properly prior to its inclusion in a new product is a new form of verification to which traditional methods do not immediately apply.

For software reuse to succeed in delivering a substantial improvement in programmer productivity requires progress in a number of areas. Component certification is an important one.

REFERENCES

[1] Andrews, D.M. and J.P. Benson, ''An Automated Program Testing Methodology and Its Implementation'', *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, March 1981.

[2] Barnes, B., T. Durek, J. Gaffney, A. Pyster, ''A Framework and Economic Foundation for Software Reuse'', *Proceedings of the Workshop on Software Reusability and Maintainability*, National Institute of Software Quality and Productivity, October, 1987.

[3] Bassett, P.G., ''Frame-Based Software Engineering'', *IEEE Software*, July, 1987.

[4] Biggerstaff, T.J. and C. Richter, ''Reusability Framework, Assessment, and Directions'', *IEEE Software*, Vol. 4, No. 2, March 1987.

[5] Conn, R., ''The Ada Software Repository and Software Reusability'', *Proceedings of the Fifth Annual Joint Conference on Ada Technology and Washington Ada Symposium*, Washington, DC, 1987.

[6] Dowson, M., personal communication.

[7] Fagan, M.E., ''Advances in Software Inspections'', *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 7, July 1986.

[8] Freeman, P., (editor), *Software Reuse: Emerging Technology*, IEEE Computer Society Press, 1988.

[9] Gargaro, A. and T.L. Pappas, ''Reusability Issues and Ada'', *IEEE Software*, July 1987.

[10] Horowitz, E. and J.B. Munson, ''An Expansive View of Reusable Software'', *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984.

[11] Jones, C.B., ''Systematic Software Development Using VDM'', *Prentice Hall International*, 1986.

[12] Lenz, M., H.A. Schmid, and P.F. Wolf, ''Software Reuse Through Building Blocks'', *IEEE Software*, July, 1987.

[13] Leveson, N.G., S.S. Cha, T.J. Shimeall, and J.C. Knight , ''The Use Of Self Checks And Voting In Software Error Detection: An Empirical Study'', *IEEE Transactions on Software Engineering*, to appear.

[14] Luckham, D.C. and F.W. von Henke, ''An Overview of Anna, a Specification Language For Ada'', *IEEE Computer*, March, 1985.

[15] McCabe, T.J., ''A Complexity Measure'', *IEEE Transactions on Software Engineering*, Vol. SE-2, December, 1976.

[16] Meyer, B., ''EIFFEL: Reusability and Reliability'', in *Software Reuse: Emerging Technology*, Tracz, W., (editor), IEEE Computer Society Press, 1988.

[17] Rice, J. and H. Schwetman, ''Interface Issues In A Software Parts Technology'', in *Software Reusability*, edited by Biggerstaff and Perlis, Addison Wesley, 1989.

[18] Russell, G., ''Experiences Using A Reusable Data Structure Taxonomy'', Proceedings of the *Fifth Annual Joint Conference On Ada Technology and Washington Ada Symposium*, April 1987.

[19] Sommerville, I., *Software Engineering*, third edition, Addison Wesley, 1989.

[20] Software Productivity Consortium, *Ada Quality and Style: Guidelines for Professional Programmers*, Van Nostrand Reinhold, 1989.

[21] Tracz, W., ''Software Reuse: Motivators and Inhibitors'', *Proceedings of COMPCON S'87*, 1987.

[22] Tracz, W., ''Software Reuse Myths'', *ACM SOFTWARE Software Engineering Notes*, Vol. 13, No. 1, Jan 1988.

[23] Tracz, W., (editor), *Software Reuse: Emerging Technology*, IEEE Computer Society Press, 1988.

[24] U.S. Department of Defense, Ada Joint Program Office, *Reference Manual For The Ada Programming Language*, ANSI/MIL-STD-1815A, January, 1983.

[25] Zelkowitz, M., J. Gannon, and A. Shaw, *Principles of Software Engineering and Design*, Prentice Hall, 1979.