Issues in the Certification of Reusable Parts

John C. Knight

Computer Science Report No. TR-92-14
May 21, 1992

Submitted to *International Journal of Software Engineering and Knowledge Engineering*.

ISSUES IN THE CERTIFICATION OF REUSABLE PARTS[†]

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 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 achieving quality as an entirely separate goal from improving productivity. If useful properties pertaining to quality 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 parts 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 parts. 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 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 parts, part certification.

---

† This work supported in part by the Software Productivity Consortium and in part by Virginia Center for Innovative Technology under grant numbers INF-90-016 and INF-92-001.

1. INTRODUCTION

A substantial difficulty that appears to be limiting software 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 parts*, 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 part, he or she would be better off rebuilding it. The observation that lack of perceived quality is a detractor from reuse is based only on anecdotal evidence but appears to be the software-reuse manifestation of the "not-invented-here" syndrome.

By incorporating reusable parts into a new product rather that rebuilding them, productivity is improved. However, possible improvement in productivity is not the only advantage of reuse although it is the most widely publicized. The parts 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.

The adjective *certified* is sometimes used to describe parts that have been tested in some way prior to entry into a library (e.g., [24]). 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 [3, 13, 24]. 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 parts. We advocate the development of software work products 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 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 part quality between the developer of a part and users of the part. Users no longer have to question the quality of parts - certification describes for the prospective user exactly what can be expected of a part. This eliminates the "not-invented-here" difficulty mentioned above and facilitates higher reuse levels.

†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 [28].

2. TERMINOLOGY

In software development that incorporates reuse, any software work 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. In the modern approach to reuse, reuse libraries might contain any form of work product including specification parts, design parts, test plan parts, as well as more traditional source-code parts. This is an important aspect of the field of reuse since the economic benefits of reusing artifacts other than source code is likely to be substantial.

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 or *reengineering* 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.

No matter what its origin, a part 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*. In some cases, provision for modification is included when a part 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 part 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 part might also wish to adapt the part 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 source-code reuse to the extent that provision for it is finding its way into programming languages. Generic program units are present in Ada [27], 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.

Finally, even when reuse is consistently and extensively practiced, *custom artifacts* have to be built for those elements of the work product that could not be constructed by reusing parts from a library. These custom artifacts might themselves be a source of new parts for inclusion in a reuse library.

3. DEFINING CERTIFICATION

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 parts and to permit the exploitation of reuse in support of work-product quality. 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 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 a source-code 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 source-code 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 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 parts to possess properties related to efficient execution. Finally, note that testing is not an especially meaningful notion for libraries other than source-code libraries.

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:

Definition: *Property*
A property is a true statement about some aspect of a reusable part. A property might be an assumption that a part makes about its operating environment or a specific quality that a part can have.

Definition: *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.

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

Definition: *Certification*
Certification is the process by which it is established that a part is certified.

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 enquire about the precise set of properties that the part 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 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.

(2) *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 reusable 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.

(3) *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 goals of the organization.

The utility of multiple definitions and the major benefit of communication between the part developer and part user is illustrated in Figure 1. The developer of the part knows exactly what qualities have to be present and the user of the part knows exactly what qualities can be assumed.

The properties included in a specific instance of certification can be anything relevant to the organization expecting to use the certified parts. The following are examples of properties that might be used for source-code parts:

- Compliance with a detailed set of programming guidelines such as those prepared for Ada [22].
- Subjected to a rigorous but informal correctness argument.
- 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.

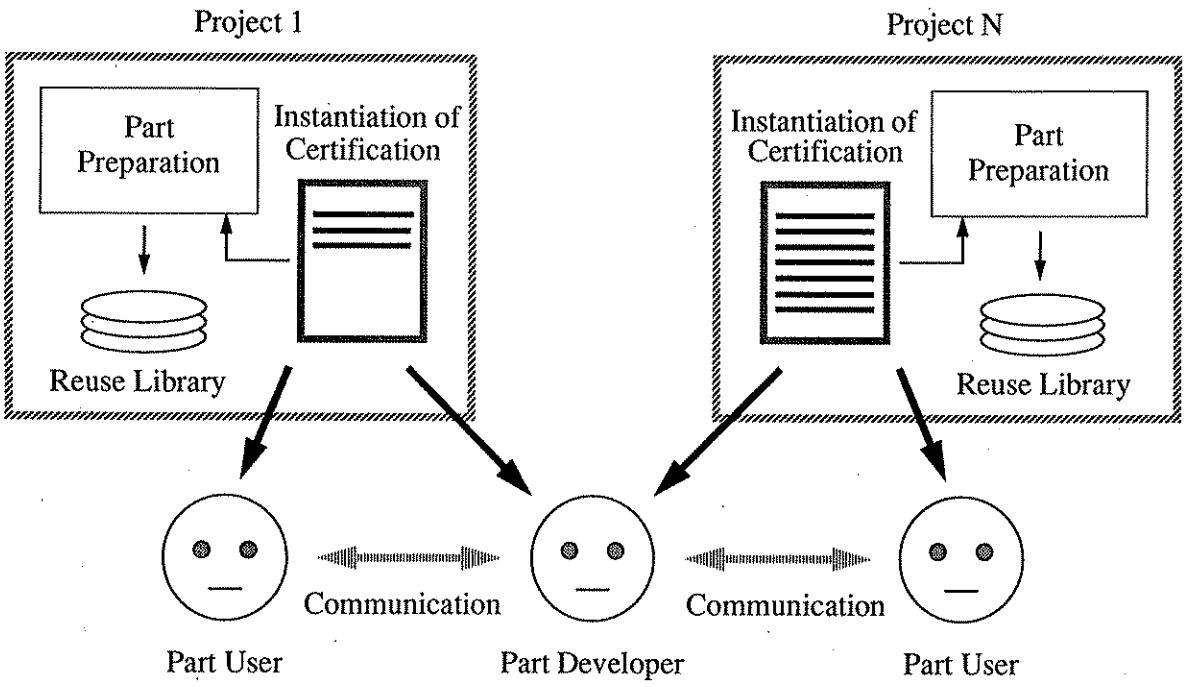

Figure 1 - Multiple Certification Instances.

4. 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.

Many properties come to mind as being desirable. However, since preparation of reusable parts is a major capital undertaking, it is inappropriate to include properties that are not essential. Consider, for example, requiring the existence of a formal proof that a source-code part has some specific quality as part of a certification instance. 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. In other words, it is not desirable to have parts that are "too good". This issue is a concern for all types of work products and all properties.

The opposite circumstance is also a factor. If establishing a necessary characteristic of a work product is facilitated by incorporating reusable parts having a certain property, then that property had better be included in the certification instance. In other words, it is important to have parts that are "good enough".

Precisely what determines the properties that should be included in a certification instance for a given reuse library? The key to the definition of any specific instance is the use to be made of the properties in the definition. The *only* justification for the inclusion of a particular property in a certification instance is that possession of that property by parts in a library contributes to the establishment of useful characteristics in work products built from those parts. Thus a certification instance is developed from the characteristics desired of work products built from the associated library, and the determination of these characteristics is part of *domain analysis* [18]. The sequence of events, therefore, is to determine the desired domain properties and then from these determine the properties required of reusable parts. These become the certification instance. Of course, this does not preclude the possibility of a common instance being used for many libraries or "standard" instances being developed for groups of domains or classes of application. These concepts are illustrated in Figure 2.

This approach 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 domain properties from which the instance of certification is derived. However, in practice, the domain properties are the ones of real concern, and they are very likely to be defined by the domain analysis. If certification is an important

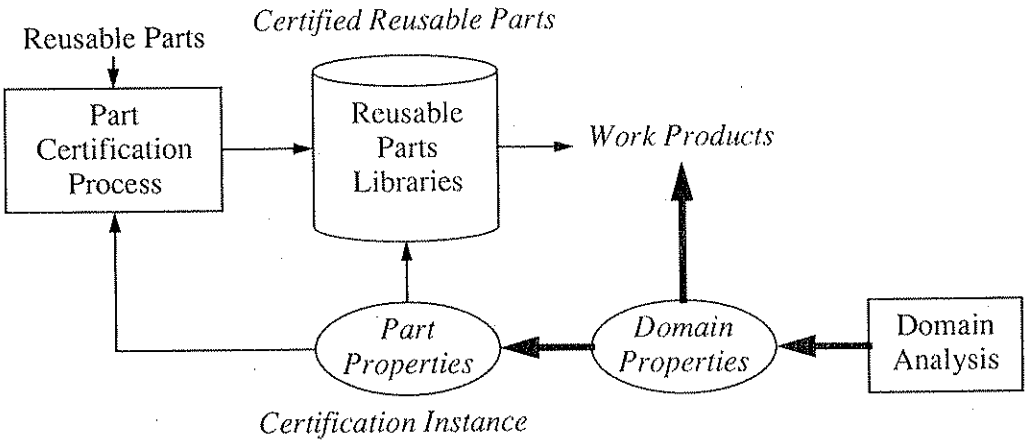

Figure 2 - Development Of A Certification Instance.

facility for a particular domain, then acquisition of the necessary domain properties will have to be a well-defined aspect of the associated domain analysis.

It is not the case that an identified domain property and the associated part property would necessarily be the same, although they might be. What is required is that part properties permit the demonstration of useful properties of work products built from them, and this might require explicit manipulation of information about the parts. Consider, for example, a reuse library of source-code parts intended for developing real-time applications. The certification instance in that case might require a determination of the absolute bound on execution time for a certified part on a given host computer/compiler combination and recording of that bound along with the part in the reuse library. Availability of the time bounds does not permit anything to be concluded immediately about any system built using the parts. However, analysis of the final system structure using knowledge of the time bounds of the parts can facilitate the assurance of meeting required real-time deadlines for some system structures.

A simple example where the part property and the domain property are the same is source-code programming standards. Clearly, if certified parts follow required programming standards, then that portion of a complete system built from such parts will follow the standards also. Showing that a complete system complies with required coding standards is facilitated in this case since the source text derived from reusable parts need not even be checked.

5. EXAMPLES OF CERTIFICATION PROPERTIES

To illustrate the ideas described in the previous section, we examine some significant properties of two types of work product that might be built with reusable components. The work products are requirements specifications and source code.

5.1. Requirements Specifications

Figure 3 shows a reusable specification component written in "Z" [23], a specification language. The component is the specification of a display in which some numeric quantity is presented on a panel of seven-segment displays. Many embedded systems from consumer electronics to avionics require such displays. To minimize hardware costs, it is often the case that much of the management of such a display is controlled by software. For example, translation of a digit into its seven-segment representation is often done in software although there are readily available hardware implementations.

Initially, specification of such displays seems unnecessary. Merely stating that the display needs to be driven by the software appears to be sufficient. However, the wealth of detail that must be defined makes this a difficult and error-prone approach. Decisions left to the programmer in that case include the number of digits displayed, whether a sign is displayed, where the sign is displayed if present, whether the sign floats if present,

$$digit \mathrel{\widehat=} 0 \mathinner{..} radix \cup \{point, negative, blank\}$$
$$feature \mathrel{\widehat=} \{none, fixed, float\}$$

$$NumericDisplay : feature \times \mathbb{N} \times \mathbb{N} \rightarrow (\mathbb{R} \rightarrow \text{seq } digit)$$

$$\begin{array}{l}
\forall\, sign : feature;\ width, radix : \mathbb{N};\ r : \mathbb{R};\ output : \text{seq } digit \bullet \\
NumericDisplay(sign, width, radix)r = output \Leftrightarrow \\
\quad sign = none \Rightarrow r \geq 0 \\
\quad \wedge\ \exists\, s : \text{seq } digit \bullet \\
\qquad value(s) = r \\
\qquad \wedge\ digit\ 1 = 0 \Rightarrow digit\ 2 = point \vee \#s = 1 \\
\qquad \wedge\ digit(\#s) = 0 \Rightarrow trunc\ r = r \\
\qquad \wedge\ r < 0 \wedge sign = fixed \Rightarrow \\
\qquad\quad (output\ 1 = negative \\
\qquad\quad \wedge\ display(sign, width - 1, radix)\ (-r) = tail\ output) \\
\qquad \wedge\ r < 0 \wedge sign = float \Rightarrow \\
\qquad\quad \#s \geq width \Rightarrow \\
\qquad\quad\quad display(fixed, width, radix)\ r = output \\
\qquad\quad \wedge\ \#s < width \Rightarrow \\
\qquad\quad\quad output = \frown/\langle blanks(width - 1 - \#s), \\
\qquad\quad\quad\quad display(sign, width - 1, radix)\ (-r)\rangle \\
\qquad \wedge\ r \geq 0 \Rightarrow \\
\qquad\quad round(r) < radix^{width} \\
\qquad\quad \wedge\ \#s \leq width \Rightarrow \\
\qquad\quad\quad output = \frown/\langle blanks(width - \#s), \langle negative\rangle, s\rangle \\
\qquad\quad \wedge\ \#s > width \Rightarrow output = s \upharpoonright 1 \mathinner{..} width
\end{array}$$

Figure 3 - Reusable Specification Part In Z.

whether leading zeros are suppressed, whether a single zero is displayed before the decimal point for values less than one, and the precise mapping of digits to illuminated segments†. For a consumer product, leaving these decisions to the programmer is far from satisfactory because subtle display errors could lead to a recall. For an avionics system, leaving these decisions to the programmer could be quite dangerous.

† Note that there are two mappings for the digits 1 and 9, for example.

The specification part in Figure 3 is parameterized to permit selection of appropriate responses to some of the questions raised above so that a complete specification for a display can be created immediately and supplied to a software engineer for implementation. The part in Figure 3 does not deal with all of the questions raised in order to keep its size reasonable. It is merely for illustration. Figure 4 is a display specification that uses the part from Figure 3. It specifies an 8-digit decimal display with floating decimal point that will be used to display a time-varying quantity called ''altitude'' that is specified elsewhere.

An important point to note here is that a complex specification problem is dealt with very effectively through reuse. If the part shown in Figure 3 were available in a library, the specification shown in Figure 4 is all that would be needed for a relatively sophisticated application and could be prepared very quickly (assuming knowledge of the part).

Z is a hierarchical language in which elements of a specification are combined in a manner not unlike the uses graph of modules in a programming language. It has, therefore, a structure that lends itself to application of reuse in much the same way that systematic reuse is applied at the source-code level. Elaborate specifications can be built in Z by employing extensive reuse if suitable libraries are available.

If libraries of specification parts were available to support a reuse paradigm for specifications, what would be the role of certification? In particular, what properties would a certification instance for a Z reuse library contain? The answer, of course, would be defined by the associated domain analysis, but, for purposes of illustration, we consider some sample properties and examine how they would be exploited.

First of all, since Z is a language with a precise syntax and semantics, any certification instance would include a set of rules akin to programming standards for source code. Such rules would define layout of the syntax, define conventions for selecting identifiers, specify elements of the Z syntax to avoid, and prescribe appropriate ways of using the language features. The reusable part shown in Figure 3 follows such a set of rules designed to promote readability and eliminate common errors.

Display

$$output : \text{seq}\ ssd$$

$$\forall t : time \bullet$$
$$output = NumericDisplay(float, 8, 10)(altitude\ t)$$

Figure 4 - Sample Display Specification In Z.

A significant certification property for specification parts is rigorous but informal correctness. It is very difficult to specify even simple things, such as a display, and be sure that all issues have been covered correctly. However, it is worthwhile expending effort, perhaps in the form of a systematic inspection, to examine a reusable specification part to ensure that it is complete, free of ambiguities, and specifies meaningful functionality. Composing a complete specification by incorporating such certified parts frees the specifier from much of the tedious yet important detail since these qualities are known to hold for the parts.

Another example of a certification property for specification parts is the treatment of time. Z does not treat time in a special way, it is just a variable that a specifier may define if necessary. This flexibility leads to inconsistency if time is introduced inadvertently more than once in a specification. If one notion of time is an integer count of seconds from a starting time and another is an integer count of milliseconds from a different starting time then serious specification inconsistencies will arise. The certification property that is needed to take care of this is to have a prescribed notion of time for all parts in a reuse library. In this way, any specification built form these parts will be assured of treating time consistently.

In summary, reusable specification parts in languages like Z offer an enormous potential productivity gain. In some cases, they will permit specifications to be built in circumstances where specifications would otherwise be missing with the ensuing risk of the wrong software being built. For many important applications, precise specifications are essential and certified specification parts permit very high quality specifications to be built very rapidly.

5.2. Source Code

Consider now the development of software in Ada using reusable parts. Certification properties might well include the timing information mentioned earlier as well as adherence to some set of programming standards. Such properties map fairly obviously into useful domain properties.

Much more significant opportunities exist, however, for exploiting certification in Ada source code development. As an example, consider the treatment of exceptions in Ada. If an Ada program unit raises an exception, a handler is sought within the unit. If one exists, it is executed and the unit is completed, but otherwise the exception is reraised in the caller at the point of the call in a process called *propagation*.

This approach associates handlers for exceptions with program scope dynamically, and leads to a variety of problems [11]. For example, an exception might be propagated an arbitrary distance up the stack of active subprogram calls thereby terminating all of the active subprograms in which a handler was not located. Worse is the possibility that a programmer-defined exception might be propagated out of the name scope for the exception into a region where the name of the exception is not known but a hander is still being sought. In that case, a handler can only invoked if it is for the catch-all exception

name `others`. It is unlikely that programmers ever intend such situations to arise but, because of the dynamic association of handlers, it is very difficult to show that such situations will not arise.

Reuse of certified parts can help to deal with this problem. Leaf parts, for example, might be required to comply with one or more of the following properties:

- - *No exceptions are declared within the part.*

- - *No exceptions are propagated out of the part.*

- - *The part contains handlers for all predefined exceptions.*

- - *Handlers are present in every block within the part for all declared exceptions.*

- - *All possible call sequences have been generated during testing of the part and no path exists in which an exception becomes anonymous or in which propagation is unbounded.*

Similarly, subsystems or canonical designs might be required to comply with one or more of the following properties:

- - *Within the call structure, handlers for* `others` *have been included in a "firewall" structure to ensure that no unbounded exception propagation can occur.*

- - *If the subsystem is terminated by an unanticipated exception propagation, provision is made within the part to restart in a meaningful way.*

Suitable combinations of such properties would permit the known difficulties with Ada exception handling to be dealt with very effectively. It could be shown with a high degree of assurance that these known difficulties would not occur in systems built with reusable parts, at least not because of defects in the reusable parts.

A second example of a significant certification benefit is in the area of tasking performance. Building concurrent systems is always difficult. Actually developing the algorithms is much harder than developing sequential algorithms but concurrent systems also introduce new classes of faults such as race conditions, deadlock, and starvation. Ada software is affected, in addition, by priority inversion.

Once again, if reusable parts are known to possess suitable properties, some of these difficulties can be alleviated. In this case, reusable parts might be required to comply with one or more of the following properties:

- - *No shared variables are referenced within the part.*

- - *There is no internal concurrency within the part.*

- - *No execution conditions with the part can lead to tasking error.*

- - *Correct operation of the part has been shown not to depend on specific priority values nor on specific system scheduling algorithms.*

- - *All entry calls made by the part and its entry definitions are documented correctly and in a machine-processable notation.*

The use of most of these properties in establishing system properties is fairly obvious. The last property in the list is intended to permit automatic or semi-automatic analysis of deadlock potential and priority inversion. Freedom from deadlock can be shown easily for tasking structures that follow simple rules, and a certification instance can include properties that facilitate building systems that do follow the appropriate rules. If the rules are followed, all that is required is that the interactions undertaken by the constituent tasks be available for analysis. Where the tasks are derived from reusable parts, the certification instance can ensure that the requisite information is available for deadlock analysis of the system.

A similar analysis can be undertaken to seek possible cases of priority inversion in Ada task structures. Once again, a certification instance can be developed that ensures the necessary information is available for analysis.

6. CERTIFICATION AND TESTING

Although certification allows the inclusion of any desired properties for parts of any type, properties associated with testing will frequently be present in certification instances dealing with source-code parts. Because of its importance, testing is the focus of the remainder of this paper.

6.1. Testing Issues

Many testing issues are raised by the inclusion of reuse in a software development method. For example, in the context of preparing parts for entry into a library some of the testing issues raised are:

(1) *Part quality.*
By definition, a part that is entered into a reuse library is being offered for use by others and has to be prepared for *every* possible use [20]. 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.

(2) *Part type.*
Testing is complicated by differing part structures. Very few reusable source-code parts will be subprograms. Other parts 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 thereby making meaning testing quite challenging.

(3) *Adaptable parts.*
Adaptable parts, i.e., parts designed to be modified before use such as Ada generic units, present significant challenges for testing. The parameters used with Ada generic units are not merely numeric or symbolic but can be subprograms thereby allowing different instances 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) *Part use.*
A reusable part will be used in many different circumstances. The possibility exists, however, that a part may be selected that does not *quite* meet the precise needs of a particular application. Where informal specification techniques are used for parts in reuse libraries and reliance is placed on human insight for part selection and matching, it will be difficult to ensure that a selected part does precisely what is required and that the part is being used correctly [9, 17, 19].

(2) *Part 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 parts are then faced with a dilemma. Incorporating the revised parts might produce useful performance improvements but the resulting software will differ substantially from that which was originally built and tested. Can revised parts with "identical" interfaces be trusted, and, if not, what testing needs to be performed when revised parts are incorporated?

(3) *Part adaptation.*
Once a part is changed, the results of any testing that took place prior to placing the part in the library cannot necessarily be trusted. That testing might have been extensive and be expensive to repeat in its entirety.

Some of the issues summarized here can be addressed by suitable certification properties and associated exploitation. For illustrative purposes, we consider the unique issues raised by adaptable parts and adaptation.

6.2. Adaptation And Adaptable Parts

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 part or a part dependent on symbolic parameters. Unanticipated adaptation occurs when a part is modified in a way that was not planned, usually using a text editor.

In many cases there are restrictions inherent in the design of a part 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. 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.

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 part 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 part 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.

For purposes of certification, two issues need to be addressed. Assuming that a certification property exists that calls for parts to be tested, then exactly how to test adaptable parts has to be dealt with. The second issue arises when a certification property is exploited to establish a property of a system in which certified parts are included. In order to exploit the fact that a part has been tested and to be sure that the part works as the developer intended, it is essential that the adaptation has met any restrictions inherent in the part.

Testing Adaptable Parts

The problem of testing adaptable parts amounts to ensuring that the adaptable part will function correctly assuming that an adaptation complies with the restrictions associated with design of the part.

Adaptable parts usually 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 part will work correctly once these constraints or selections are installed.

The various adaptations that are provided with an adaptable part are similar in many ways to inputs to the part. From the point of view of correct functionality, setting a symbolic parameter, say, has some of the characteristics of reading an input of the same type as the parameter. The part 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 part 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 part is to instantiate the part 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. Systematic in this case is essentially equivalent to the conventional problem of test case selection. A key research issue that remains is to find useful ways of doing this for adaptations that require functional information to be supplied.

Checking Anticipated Adaptation

If an adaptable part has been tested as part of certification, then exploitation of that testing is completely dependent on the adaptation in a particular case having been done correctly. If they are documented at all, the various restrictions imposed on an adaptation are usually documented as comments. 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 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 part 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 [15] 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 part 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 part. 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 part but again *prior to conventional compilation*. The concept is to associate with a reusable part 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 part 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 5.

Checking Unanticipated Adaptation

On occasion, arbitrary changes made using an editor might be required when attempting to reuse an existing part even if the part was designed for reuse. 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 if a certified part is changed. 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 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 part is different from the original part and obviously satisfies different specifications after unanticipated adaptation. If the specifications were not different after unanticipated adaptation, there would be no point in modifying the part in the first place.

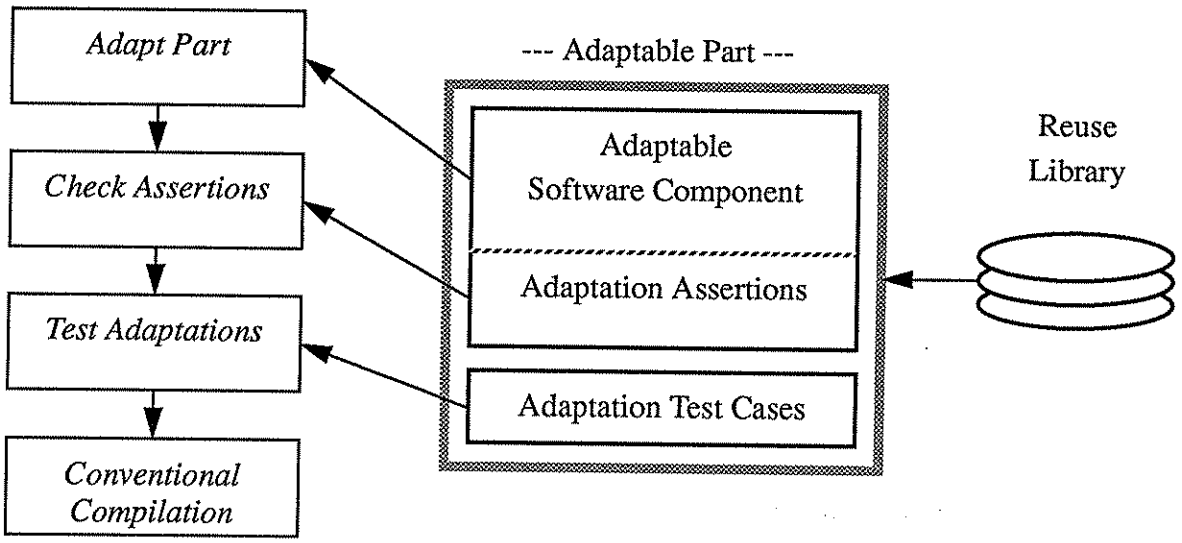

Figure 5 - Checking Anticipated Adaptation.

Storing the specification of a part in machine-processable form and modifying the specification along with the part 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 in the present embryonic state of reuse technology.

A promising first approach to dealing with many of the issues, at least partially, is the instrumentation of reusable parts with executable assertions [1, 15, 17]. In fact, Anna [15] is described as a notation for specification although it does not have the completeness characteristics of a rigorous approach such as VDM [12]. 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 [14]. Executable assertions can be used therefore as part of a system for checking parts subjected to unanticipated adaptation.

7. CONCLUSION

Software reuse can be exploited to improve work-product quality in conjunction with the highly publicized goal of using it to improve productivity. The reuse of parts that have been shown to possess desirable properties has the potential for conveying those properties to the work product in which the parts are used. This information can then be used to help establish desirable properties in the final product. Desirable properties go far beyond simple functional correctness and might include properties in areas such as maintainability, execution efficiency, or portability.

To do this effectively requires a precise framework for dealing with part 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 parts. Within a development organization this permits users of reusable parts to have confidence in the parts, confidence that is usually missing. This is expected to facilitate systematic reuse considerably and thereby to promote higher levels of reuse.

A number of significant issues arise when considering both the testing of reusable parts and the testing of systems incorporating reusable parts. 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 productivity requires progress in a number of areas. Part 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] Howell, C., and D. Mularz, "Exception Handling in Large Ada Systems", Technical Report, MITRE Corporation, McLean, VA, 1991.

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

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

[14] 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.

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

[16] McCabe, T.J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. SE-2, December, 1976.

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

[18] Prieto-Diaz, R., "Domain Analysis: An Introduction", *ACM SIGSOFT* , Vol. 15, No. 2, April 1990, pp. 47-54.

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

[20] 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.

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

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

[23] Spivey, J.M., "The Z Notation: A Reference Manual", Prentice Hall, 1989.

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

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

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

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

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