Certification Of Reusable Software Parts

Michael F. Dunn
John C. Knight

August 31, 1992

CERTIFICATION OF REUSABLE SOFTWARE PARTS†

Michael F. Dunn John C. Knight

Department of Computer Science
University of Virginia
Charlottesville
Virginia 22903

August, 31 1992

† This work was funded in part by the Software Productivity Consortium and in part by the Virginia Center for Innovative Technology, grant number INF-92-001.

Executive Summary

Improved system quality is often cited as a benefit of software reuse, but little work has been done to quantify this benefit. The *software component certification* strategy presented in this report is a framework for such a quantification.

The main idea behind component certification can be stated quite simply: *Having guaranteed that a specific set of quality guidelines have been adhered to in a set of components, it will then be much easier to verify the quality of a system composed of those components*. The main questions one must answer about a certification scheme are:

- *How does one establish component quality guidelines for the various lifecycle work products?*

  While reuse-oriented development is often thought of in terms of source code, it is important to consider the benefits of reusing other work products as well, such as specifications and designs. These are fundamentally different artifacts of the software development process, and a comprehensive certification scheme must be flexible enough to accommodate them.

- *Once component quality guidelines have been established, how can they be used to determine the quality of a system that incorporates them?*

  A point central to the strategy presented in this report is that in order to establish component quality guidelines, one must first establish the quality requirements of all systems within a particular problem domain, and then work backwards. This underscores the importance of the domain analysis in the software process.

- *How can a certification strategy be established that will suit the unique needs of a variety of different organizations?*

  The strategy presented here is not a set of specific quality rules, it is a framework, coupled with guidelines on how to derive such rules. The idea is to provide guidance on how to determine what quality attributes are important, and then use this knowledge to create certification criteria. As such, the strategy presented should be useful to any software development group.

- *How does an organization go about assessing the economic benefits of such a*

strategy?

The fundamental assumption is that an organization must build systems to adhere to a set of quality standards, whether a component certification process is used or not. The claim made here is that a certification process will lead to reduced costs by encouraging component reuse, reducing quality assurance cost, reducing the amount rework required, and easing the maintenance process. The section of the report dealing with this issue provides a set of questions to help the organization determine whether adopting a certification strategy is economically appropriate.

1. Introduction

This report focuses on the development of high quality systems through the use of rigorously certified reusable software components. While enhanced quality is often mentioned as a benefit of reuse-oriented development processes, it has been difficult to establish the nature of these enhancements in a convincing way. The techniques described here are a step toward this establishment.

Intended for the member companies of the Software Productivity Consortium, this report is designed to serve as both a detailed explanation of the framework underlying component certification, and as a practical guide to the software practitioner. As such, the organization is as follows:

- Chapter 2 provides definitions for the fundamental terms used throughout the document, and provides a rationale for the certification framework itself.
- Chapter 3 describes the nature of certification properties, and gives a framework for developing complete and consistent definitions for these properties.
- Chapter 4 presents a detailed look at how part certification fits into the context of a reuse-oriented development process.
- Chapter 5 gives specific techniques for establishing relevant part properties.
- Chapter 6 is a follow-on to Chapter 5, and shows how a set of part properties can be used to establish properties of the system comprised of those parts.
- Chapter 7 gives an economic justification to the component certification process.

As an appendix, the report includes two case studies showing example certification definitions and their application.

2. Definitional Framework

As an old joke among computer scientists goes, on New Year's Eve, 1999, all of the computer programmers in the world will line up outside of their respective banks and withdraw all of their money. The reason, of course, is because of the uncertainty over the impact of the millennium on all of the computerized banking systems currently in use that assume a two character year format, with '19' being the implicit first two characters. The fact that this joke is more frightening than it is humorous provides a motivating example for the issue of software component certification.

As software development techniques begin to rely more and more on the use of reusable components, the quality of these components will become an issue of increasing urgency. The anecdote noted above is a fairly obvious example of component usage that has been prevalent for years; namely, the use of standard date processing routines in business applications. While the utility of such routines is obvious, no one can vouch for the dependability of all such routines under all circumstances. The user of such a routine must assume that it will provide reasonable results under such conditions as leap years and century changes. For the user to have to examine the source code of the routine to verify its correctness would defeat the purpose of having such a general routine in the first place.

Therein lies the problem of component certification: How to determine what quality aspects of a component are truly important, and how to understand the impact these aspects will have on the overall quality of systems built with such components.

2.1 Framework

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.

2.1.1 Basic Definitions

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 report:

Definition: *Part*

> A life-cycle work product out of which other, larger life-cycle work products can be composed. In this report, the word *part* is used interchangeably with the word *component*.

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.

2.1.2 Framework Rationale

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:

- *Flexibility.*

  As many different certification instances can be defined as are required,. and different organizations can establish different sets of properties to meet their

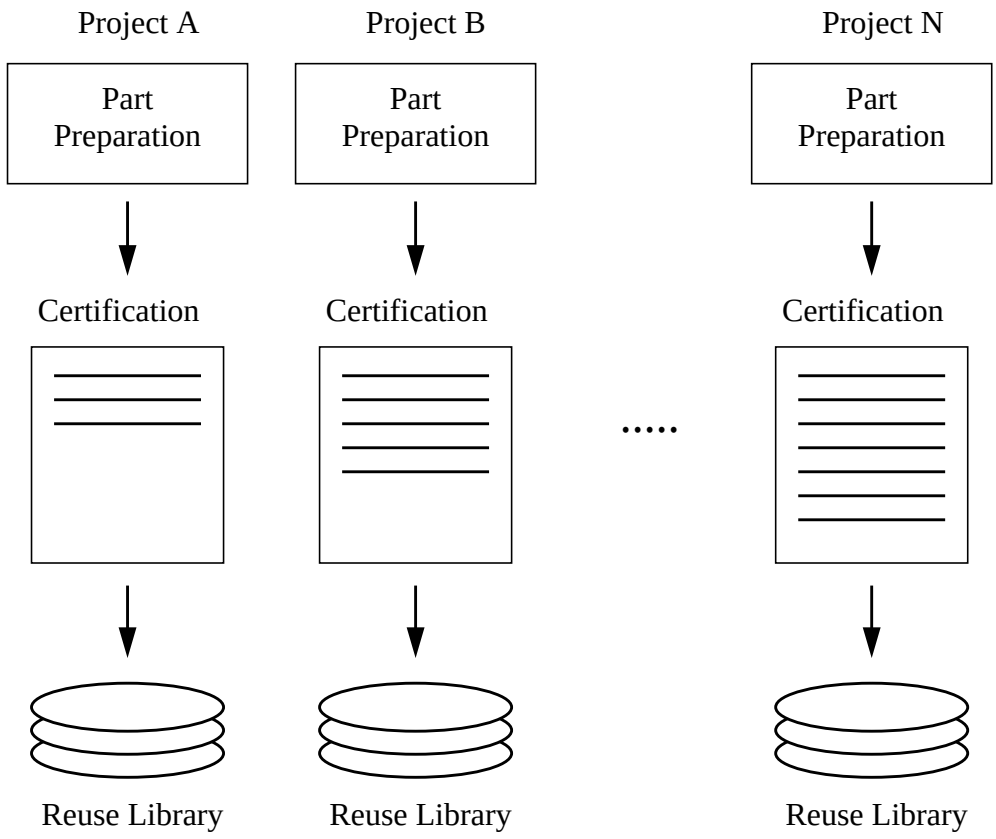

Figure 1 - Multiple Certification Definitions

needs (see Figure 1). 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 *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.

- *Precision.*

  Once the prescribed property list in the certification instance is established, there

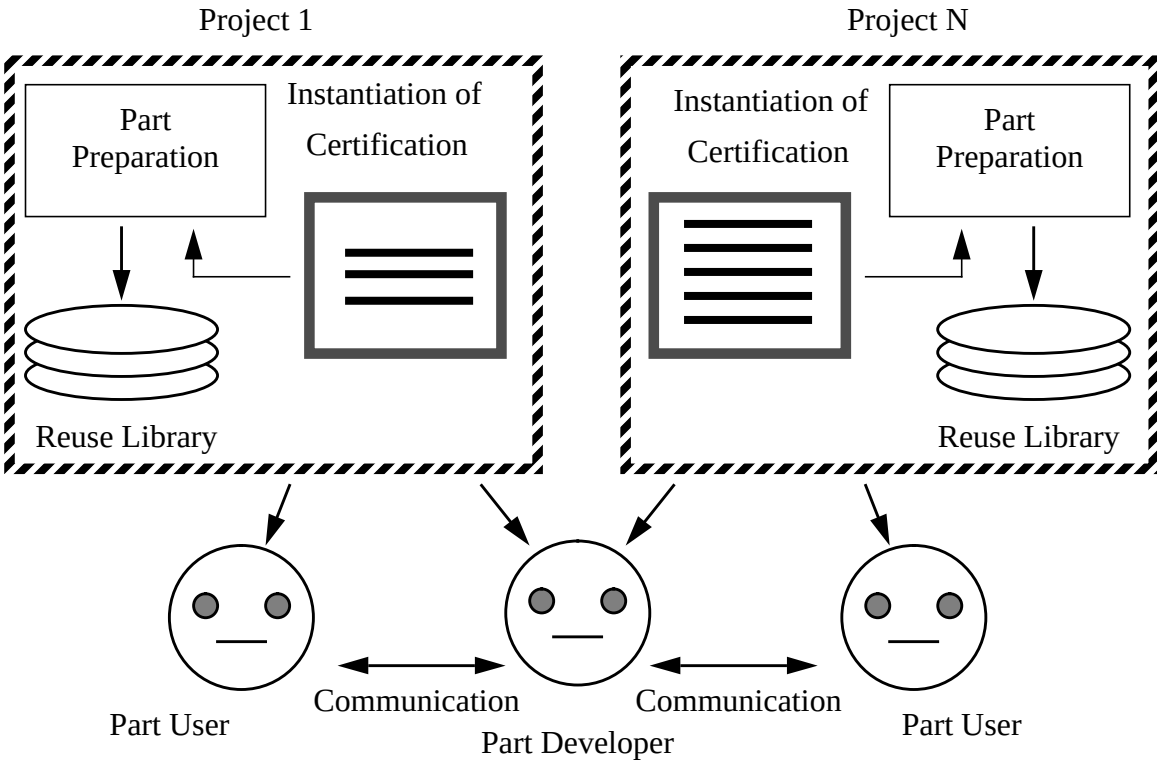

Figure 2 - Communication By Certification

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 2. 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 [SPC89].
- 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.

Determining the properties to be included in a particular instantiation is discussed in the next section and in Chapter 3.

2.2 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 component 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* [Pri90]. 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 3.

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 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. By adding the requirement for

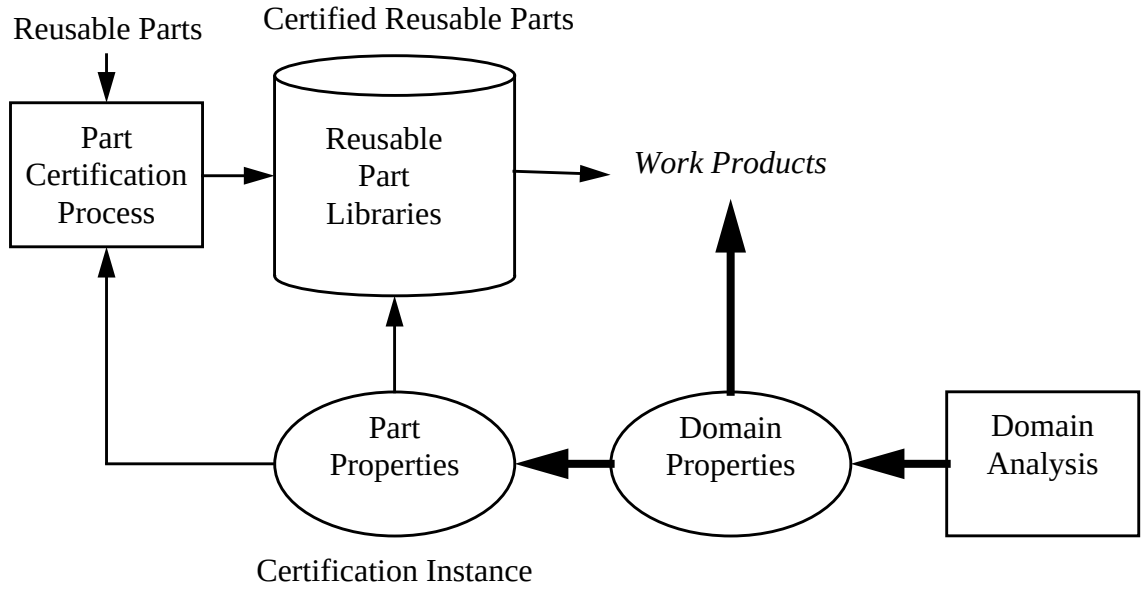

Figure 3 - Instantiating Certification

capturing certification properties to domain analysis, the concept of *extended domain analysis* is defined.

It is not the case that an identified domain property and the associated part property will 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 structure

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.

2.3 Summary

The concept of this framework is to define certification to be a set of properties possessed by reusable components in a certified reuse library. The properties in question are themselves inferred from the properties required of systems developed within the domain. These latter system properties are determined during an extended domain analysis.

The key aspect of this framework is its flexibility. The idea is not to create a rigid set of certification criteria that can be used by all organizations, but rather to create a framework by which any organization can create its own set of certification criteria. One must not regard certification as a process that takes place in a vacuum; properties one determines for a set of components must be useful in deriving properties for a system built with these components. It is also important to keep in mind the balance between having parts that are “too good”, that is, over certified and thus expensive, and those that are “not good enough”, and can lead to problems in deployed systems.

The following chapter refines the notion of certification properties, and how one should go about defining them for a variety of work products.

3. Developing Certification Definitions

As mentioned in Chapter 2, certification instances will vary according to the needs of the problem domain in question and according to the needs of the software development organization. As is shown in this chapter, however, there is a structure underlying the creation of certification instances; a structure that should remain constant regardless of the characteristics of specific organizations or problem domains.

Briefly, certification property definitions are used to describe two attributes of a component:

- Facts about the component itself. In other words, *what the component is*. These are referred to here as *structural properties*.

  Properties of this sort are important because they determine how easily a component can be used, modified, and reused, regardless of the application domain. Some important factors affecting properties of this sort are:

  - The lifecycle phase for which the component is intended.
  - The notation or language in which the component is written.
  - The size or complexity of the component.

- Facts about the intended use of the component. In other words, *what the component does*. These are referred to as *behavioral properties*.

  Properties of this sort are important because they determine how useful the component is. The important factors here are:

  - The application domain for which the component is intended.
  - The size or complexity of the component.

The next several sections refine this distinction by developing a framework that can be used to organize the development process of a certification instance.

3.1 Three Major Certification Attributes

The previous chapter defined the rationale for a component certification scheme. The next step is to determine the types of components to which one should apply the scheme. This section considers this step by looking at three attributes:

- The *lifecycle phase* in which the component was produced.
- *Level of granularity* of the component; that is, its size and complexity.
- The *domain* for which the component is intended.

3.1.1 Lifecycle Phase

Although reuse research has tended to focus on source code, numerous researchers have stressed the economic benefits to be gained from reusing higher-level artifacts, such as specifications and design parts. As reuse of these work products becomes more common, it is reasonable to assume that the development of appropriate certification criteria will become necessary. Like source code, these other work products can be categorized into a variety of different components. When dealing with specifications, for example, we might speak of parts such as module catalogs, lists of undesired events, or abstract interface descriptions.

3.1.2 Level Of Granularity

Traditional reuse libraries have focused on source-code elements, such as Ada generics, collections of C functions, and C++ classes. The certification instances of one such artifact versus another are likely to be quite different, given their differing semantic content. In certifying a generic, for example, one might be concerned about the number of different data types used as parameters during testing, as well as the specific data values.

Granularity is also a concern for other lifecycle work products. In specifications for entire subsystems, for example, a major problem is keeping the usage of names and their definitions consistent. While this is true of any size specification, it is much easier to keep track of this for small components.

3.1.3 Intended Domain

The type of application for which a component set is intended will also impact the manner in which certification criteria are applied. One would expect a safety-critical application domain, such as flight control, to have a much more rigorous set of certification properties associated with it, than, for example, a domain such as warehouse inventory control.

3.1.4 Graphical Model

These three major areas, *level of granularity, lifecycle phase*, and *application domain*, constitute three major *dimensions* that can be used to help define certification instances for a set of components. Figure 4 shows how these dimensions, arranged as a 3-space, can be used to describe a source-code function of an embedded real-time system. The next several sections discuss specific issues in work product certification definitions, namely:

- Notation used to define work products, and the impact various notations will have on the definitions.
- The types of properties for which one will need definitions, and the partitioning of these properties into property classes.

3.2 Notation Used To Define Work Products

To understand the problems posed in forming a general component certification scheme, consider the *Lifecycle-Phase* dimension. The labels that appear on that axis are, of course, heterogenous. For example, one would not expect a code artifact to even remotely resemble a requirements artifact. However, the language used to define the part also has a major impact on the definition of its ultimate certification instance. This section describes some of these differences for two of these part types, specifications and code. The influence of these differences can be seen readily throughout the rest of this report.

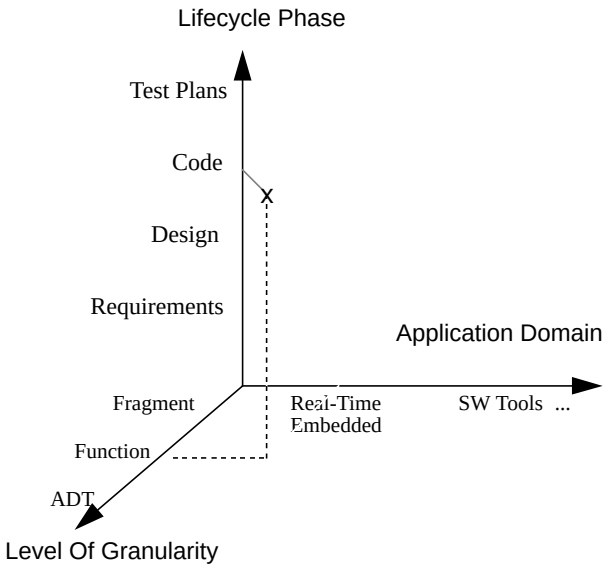

Figure 4 - Certification Space

3.2.5 Choice of Language For Specifications

Various notations for capturing requirements specifications exist, ranging from informal natural language to “semi-formal” methods such the A-7E technique [Hen80], to formal methods such as ‘‘Z’’ [Spi89]. Given the ambiguity inherent in natural language, one can make a strong case for favoring specifications captured using more formal methods as lending themselves more easily to certification.

3.2.6 Choice of Language For Source Code

The ease with which source code parts can be certified depends to a large extent on the language in which the parts are written. To complicate the issue, not only are there variations between the features and facilities of different languages, but different implementations of the same language will often yield semantically different object code from typographically equivalent source code. An organization devising a certification scheme will have to understand these differences in order to create truly useful certification definitions.

The impact that language differences will have on certification will manifest itself in the following areas:

- Concurrency semantics.
- Exception handling semantics.
- Mathematical precedence rules and order of evaluation in expressions.
- Type or class inheritance semantics.
- Abstraction and encapsulation mechanisms.
- Recursion semantics.
- Efficiency, for example, of numeric and string processing functions.
- Strength of type mechanism.
- Approach to evaluation, lazy or eager.
- Boolean expression semantics, short-circuit or non-short-circuit.
- Scoping (visibility) rules.
- Parameter transmission semantics.

3.3 The Definition of Certification Properties

The purpose of this section is to clarify what we mean by the term, certification property. As defined earlier, a ***property*** is *a true statement about some aspect of a reusable part*. More specifically, we can define a property as one of the following:

- An assumption that a component makes about its operating environment.
- A statement about an empirically verified aspect of the component. Quality assurance can take the form of testing, static analysis, benchmarking, or inspection.
- A statement about a formally proven aspect of the component.

As previously mentioned, a property can be a statement about what the part *is* or what the part *does*. The following are examples of each type of property:

- *What a part is:*

  "This function does not depend on parameter-passing implementation semantics."

- *What a part does:*

  "For any given input integer n, where n is between -MAXINT and MAXINT, this function will complete execution within 0.1 milliseconds."

Extensive examples of part properties are included in section 3.6 and Appendix A. From these examples, two issues arise:

- The number of properties one might want to certify for any given component is potentially limitless.
- Determining which properties to certify is an unstructured, ad hoc process.

To address the first issue, consider once again observations made in Chapter 2. The only significant interest that the development engineer has is the properties desired of *system*s in the domain. Specific properties of the individual components are irrelevant except to the extent that they contribute to the establishment of domain properties. Thus the certification instantiation should include those properties and only those properties that contribute to domain properties. Note that a domain property and the associated component property or properties will often not be the same.

Precisely what should be in a certification instance may not be clear at first. This implies a learning process via one or more iterations of developing similar versions of a particular system. In determining certification properties for the initial set of components, one would use previously developed systems to help establish a baseline set of properties, plus a formal or informal domain analysis.This idea is expanded in Chapter 4.

The second issue refers to how the space of possible certification properties should be conceptually partitioned. The next section introduces a mechanism to perform this partitioning via the notion of establishing *Property Classes*.

3.4 Property Classes

Part properties established as part of certification tend to influence the software lifecycle either by improving the efficiency of the development process, by improving the functional integrity of developed systems, or by reducing maintenance effort. By organizing these properties into a set of classes, one can determine more easily the completeness with which a certification instance has been specified for a component.

3.4.1 A Catalog Of Property Classes

The property classes we propose are specific to the following areas:

- *Reusability Properties*

  Properties that influence the ease with which a component can be reused in a variety of different systems.

- *Maintainability Properties*

  Properties that influence the ease with which a system can be modified and enhanced. This includes, for example, the usage of standard naming conventions. It might also include an indication of how many other components might need to be modified if a change is applied to a given component.

- *Portability Properties*

  Properties related to machine dependencies or environmental requirements. Examples include the assumed size of MAXINT, the availability of X-Windows versus some other windowing system, and the usage of compiler-specific language features.

- *Performance Properties*

  Properties governing the speed with which a component executes, or the amount of memory it uses while processing. For non-code work products, this might be a measurement of algorithmic complexity rather than an empirical measurement.

- *Safety Properties*

  Properties related to system qualities such as reliability, availability, safety, or security. Such properties might include assured fail-stop computation, bounded loops assuring termination, or a built-in restart capability.

- *Completeness Properties*

  Properties related to how fully a set of items is covered. For example, does a particular specification include actions for all anticipated undesired events.

- *Precision Properties*

  Properties related to numeric precision.

- *Presentation Properties*

Properties dealing with the textual appearance of the work product. This includes type faces and formats, and symbol usage.

Each lifecycle work product will have a set of property classes associated with it, defining that work product's certification instance. To fully state a property in a certification instance, it is necessary to include a statement of the system property that can be determined from the certified component property. Intuitively, a single system property will depend on a set of one or more component properties.

3.4.2 Property Class Orthogonality

The idea of property class *orthogonality* is to determine which types of properties have no influence on any other types of properties. For example, one might make the claim that properties pertaining to numeric precision will have no influence on those pertaining to concurrency. If so, one could make changes to the precision characteristics of a component, and not have to worry about needing to go through a recertification process of the component's concurrency properties to ensure they still work properly. Other types of properties, such as those pertaining to portability, might need to be recertified, but at least the number of properties known to require such an effort has been narrowed down.

As an example of properties that could be in conflict with each other, consider performance properties versus reusability properties. The additional programming often required to make a component applicable to a variety of different circumstances is often cited as a factor in decreasing the component's execution speed. In such cases, the developer has to make a conscious trade-off between speed of execution and speed of development.

Within the scope of this report, it is not possible to define general rules for recognizing which classes are orthogonal to which other classes, and characterize the trade-offs one must make for those properties that are not orthogonal.

3.5 Notations For Expressing Properties

In this document, certification properties are conveyed in ordinary English. It should be stressed that this is done for purposes of clarity only. Since component properties are intended as a means for drawing inferences about domain properties, a more formal notation is called for.

Formal specification methods have grown in usage in recent years, and there is strong evidence that these techniques can be applied to other realms besides specification. One might express a desired set of source-code properties using Z constructs, for example.

Another possibility is to express properties as a set of facts in the Prolog language. An

advantage to this is that one can then use the Prolog inferencing mechanism to check for property completeness and consistency.

3.6 Examples Of Certification Properties

For illustrative purposes, this section provides some examples of certification properties for four different types of lifecycle work products: specifications, canonical designs, source code, and test cases. Examples of structural and behavioral properties are given for each work product type.

3.6.1 Specifications

Behavioral-specification properties refer to how adequately the specification part captures the functional requirements of the system. Structural-specification properties refer to the typographical, format, and syntactic correctness characteristics of specification parts.

Property Class	Property Definition	Certific'n Method
Performance	All appropriate real-time constraints defined.	Inspection
Safety	Specified ordering of events always leaves system in a safe state.	Inspection
Precision	All appropriate numeric accuracy requirements stated.	Inspection
Completeness	All I/O sources (human, sensor, network, other) identified.	Inspection

Table 1: Behavioral Properties

Property Class	Property Definition	Certific'n Method
Reusability	Text conforms to standard fonts and font sizes.	Inspection
Maintainability	Domain specific quantities defined via symbolic parameters.	Inspection
Portability	Free of references to or dependencies on a specific hardware platform.	Inspection
Presentation	All names bracketed by appropriate A-7E delimiters.	Inspection

Table 2: Structural Properties

3.6.2 Canonical Design

Behavioral canonical-design properties refer to how adequately the design constrains the number of design choices one can make about a particular system, without sacrificing necessary flexibility. Structural properties refer to how easily one can insert code into the design, and how well the skeleton conforms to an accepted set of standards.

Property Class	Property Definition	Certific'n Method
Performance	No assumptions made about relative speeds of tasks to be included in skeleton.	Proof
Safety	Ensures correct sequence of task execution.	Inspection
Precision	No assumptions made about maximum relative error of floating-point quantities.	Inspection
Completeness	Correctness of parameter values and return values verified using "interface model" functions.	Testing

Table 3: Behavioral Properties

Property Class	Property Definition	Certific'n Method
Reusability	''Slots'' for plug-in code clearly indicated with stylized comments and function headers.	Inspection
Maintainability	No common coupling (sharing of global data) assumed between "plug-in" functions.	Inspection
Portability	Complies with the Portability section (Chapter 7) of the *SPC Ada Style Guide*.	Inspection
Presentation	Includes machine-processable design documentation comment headers.	Static Analysis

Table 4: Structural Properties

3.6.3 Source Code

Behavioral source-code properties refer to the code's run-time characteristics. Structural source-code properties refer to the code's static textual properties, particularly the ease with which code can be understood and modified.

Property Class	Property Definition	Certific'n Method
Performance	For each target configuration worst-case CPU time established.	Proof
Safety	No anonymous exceptions raised.	Static Analysis
Precision	Maximum relative error of floating-point quantities and circumstances under which this error occurs are documented.	Proof
Completeness	Selectable range checks implemented on all input parameters.	Inspection

Table 5: Behavioral Properties

Property Class	Property Definition	Certific'n Method
Reusability	Complies with the Reusability section (Chapter 8) of the *SPC Ada Style Guide*.	Inspection
Maintainability	Information hiding applied to hardware details.	Inspection
Portability	Part does not depend on parameter-passing implementation semantics.	Inspection
Presentation	Complies with relevant sections of Chapters 1-6 of the *SPC Ada Style Guide*.	Inspection

Table 6: Structural Properties

3.6.4 Test Case

Behavioral test-case properties refer to the adequacy with which a set of test cases exercise a set of functions. Structural test-case properties refer to the ease with which a set of test cases can be understood and modified.

Property Class	Property Definition	Certific'n Method
Performance	Entire test suite executes in five CPU seconds on an unloaded SPARC IPC with source compiled using highest optimization level available on standard compiler.	Benchmarking
Safety	Tests absence of all unacceptable output combinations in all operating modes.	Testing
Precision	Test cases cover responses to all floating-point maximum relative error situations.	Proof
Completeness	Test cases cover all allowable parameter data types.	Inspection

Table 7: Behavioral Properties

Property Class	Property Definition	Certific'n Method
Reusability	Test cases partitioned into required subsets for all systems in domain, and optional subsets.	Inspection
Maintainability	Function of each test harness indicated by stylized comments.	Inspection
Portability	Executes correctly on all 386- and 68030-based machines.	Testing
Presentation	Test case output formatted to facilitate automatic verification.	Testing

Table 8: Structural Properties

3.7 Specific Guidelines

The basic idea is to drive the enhanced domain analysis with questions from the property classes. One then uses the quality properties of the domain to establish which properties are needed in the various certification instantiations.

(1) Do an enhanced domain analysis; that is, determine not only the overall functional characteristics of the problem domain, but also identify the quality properties that need to be present in any system in this domain.

(2) Identify which lifecycle work products are to be built with reusable parts.

(3) Determine desired qualities of work products as they affect the lifecycle phases in terms of certification. For example, if maintenance is a large cost item in this domain, a number of properties pertaining to ease of enhancement and modifiability will be needed.

(4) Derive part properties from necessary domain properties.

- Distinguish each property according to whether it is behavioral or structural.
- Identify the impact of the language used to express the work product on its quality. A set of Z specification parts, for example, will require an entirely different set of structural properties from a set of A-7E-style specification parts.
- Identify the impact that the size or complexity of the work product has on its quality. When one is dealing with the source code for an abstract data type, for example, one needs to consider the interactions between the functions of the ADT, a concern which is absent when dealing with individual functions.

(5) Organize the part properties according to a catalog of property classes, such as that shown in section 3.4. This will help in the determination of whether there are major gaps in the set of property definitions.

4. Reuse-Oriented Development

In this section, the role of certification in the overall software lifecycle is considered. Certification as defined here is not dependent on any specific software process and can be applied in any development activity employing reuse. However, we note here specifically that certification can be used with the SPC's Synthesis process.

4.1 Opportunities For Certification

To gain the greatest benefit from reuse, it is essential that it be applied to the development of as many lifecycle products as possible. It is important, therefore, to seek opportunities for reuse of specifications, designs, implementations, test plans, and all relevant ancillary documentation. The same is true of reusable component certification. If any work product is developed with reuse, the quality of the associated product can be enhanced in a cost-effective manner by exploiting certification. Certification should be an integral part of each and every reuse activity.

To exploit certification, a set of certification instances is developed and applied to a set of reusable components. Once this is done, an engineer can begin to draw quality inferences about the work products resulting from the development. With a little careful thought, this simple idea can be applied to any work product. It is not essential that the certification properties be in any sense complete in order to be useful. Properties can be added as required as experience within the domain is gained.

As examples of simple certification properties in diverse work products, consider first a set of reusable specification components written in English. If it is known that the set of components is grammatically correct, has correct seplling, is in a form suitable for inclusion in documents developed with a standard word processor, and has been reviewed for correctness and completeness, then such components comply with a simple certification definition and are immediately useful for the preparation of new specifications. Much of the

traditional proofreading and checking work (work that is tedious and error prone) can either be eliminated or reduced because of the quality of the reused components. A good example of this approach is presented in a case study of the SPC Synthesis process [SPC92].

As an example in a completely different work product (source code), if we know the execution speed of a set of components and we have a single-thread system composed entirely of those components, then we can deterministically show the execution bounds on the entire system. It is quite simple to develop a tool to assist in this form of analysis so that statements can be made immediately about the system's quality.

The opportunities for reuse and correspondingly certification in the most abstract development process are shown in Figure 5.

4.2 Certification And Synthesis

Synthesis is the reuse-oriented development process advocated by the Software Productivity Consortium. It includes elements of traditional component reuse and application generation technology. A basic description of the process can be found in the SPC's Synthesis documentation [SPC90]. There are two main aspects to Synthesis:

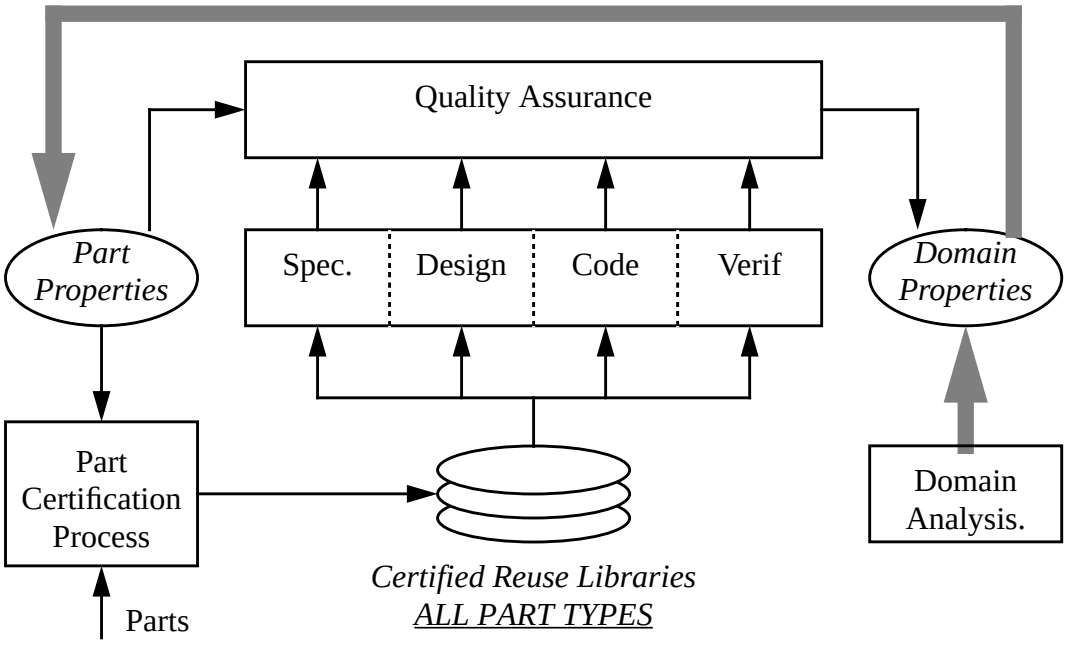

Figure 5 - Development Process

- *Domain Engineering*, which involves performing a domain analysis, and then creating an Application Engineering Environment to facilitate the creation of families of systems in the domain. This environment includes software components, and tools for composing those components.

- *Application Engineering*, which takes specific user requirements, and uses the Application Engineering Environment to convert those requirements into systems.

There are three steps in the Synthesis process where certification can play a role:

- *Domain Analysis*

  In the Domain Analysis steps, the engineer must determine the quality characteristics each system family in the domain should possess.

- *Domain Implementation*

  In the Domain Implementation steps, the engineer creates a desired set of components as part of the Application Engineering Environment. Once created, these components must be certified according to the desired properties derived in the Domain Analysis.

- *Application Engineering*

  During the Application Engineering step, these properties are then used as part of the validation and quality assurance of the system as a whole.

4.3 Using Certification In Development

Figure 6 shows in more detail how certification fits in a reuse-oriented development process. As is shown, components supporting each lifecycle phase are subjected to a certification process. In this process, it is determined whether the components conform to a set of certification properties. These properties are established via the domain analysis. Components that conform are placed in the organization's reuse library or libraries. A system developed using these components is then certified according to a set of domain certification criteria. The properties of the components are used to establish that these domain properties are present.

There are three important points to emphasize here:

- First, the arrow from the *Certified Reuse Library* symbol is to a generic phase of the development lifecycle and this reinforces the point that reuse is appropriate for all development work products, and certification criteria can be established for each one.
- Second, the only input to the *Part Properties* symbol is from the *Domain Properties* symbol. This stresses the fact that the only part properties that matter in a certification definition are those that assist in showing the presence of the domain properties in the various work products.
- Third, the arrows from the *Domain Properties* symbol to the *Part Properties* symbol and the *Quality Assurance Process* symbol emphasize the iterative

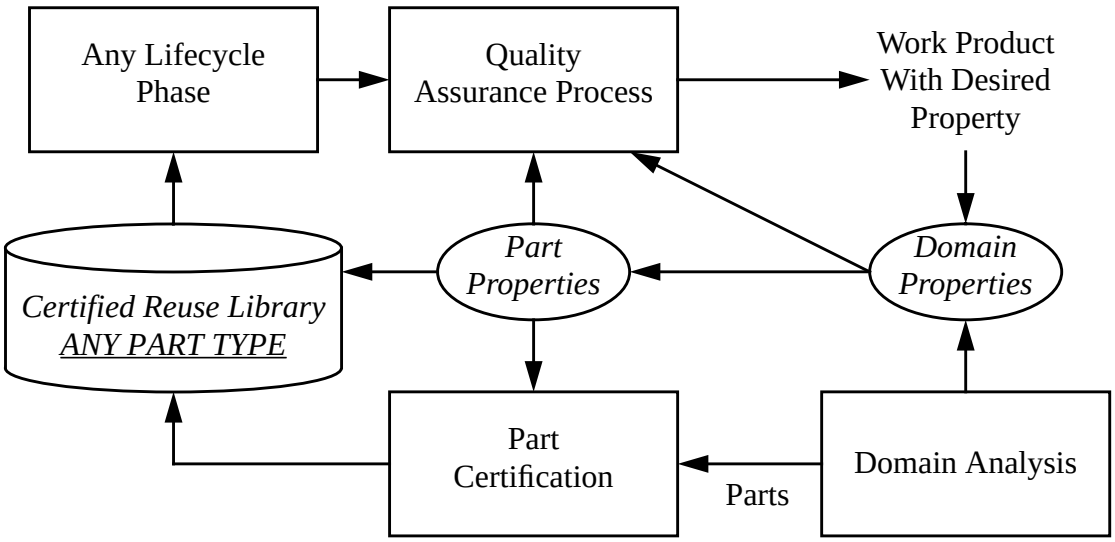

Figure 6 - Development Process

nature of certification. It is a learning process. Knowledge gained from deployed systems can be used to further refine the domain analysis, which can then be used to further refine the set of component and domain certification properties.

4.4 Roles

Figure 7 shows the key roles in a software development effort, and the information flow relevant to software certification.

4.4.1 Role of the Customer / Client

The customer or client is responsible for understanding the requirements of the problem

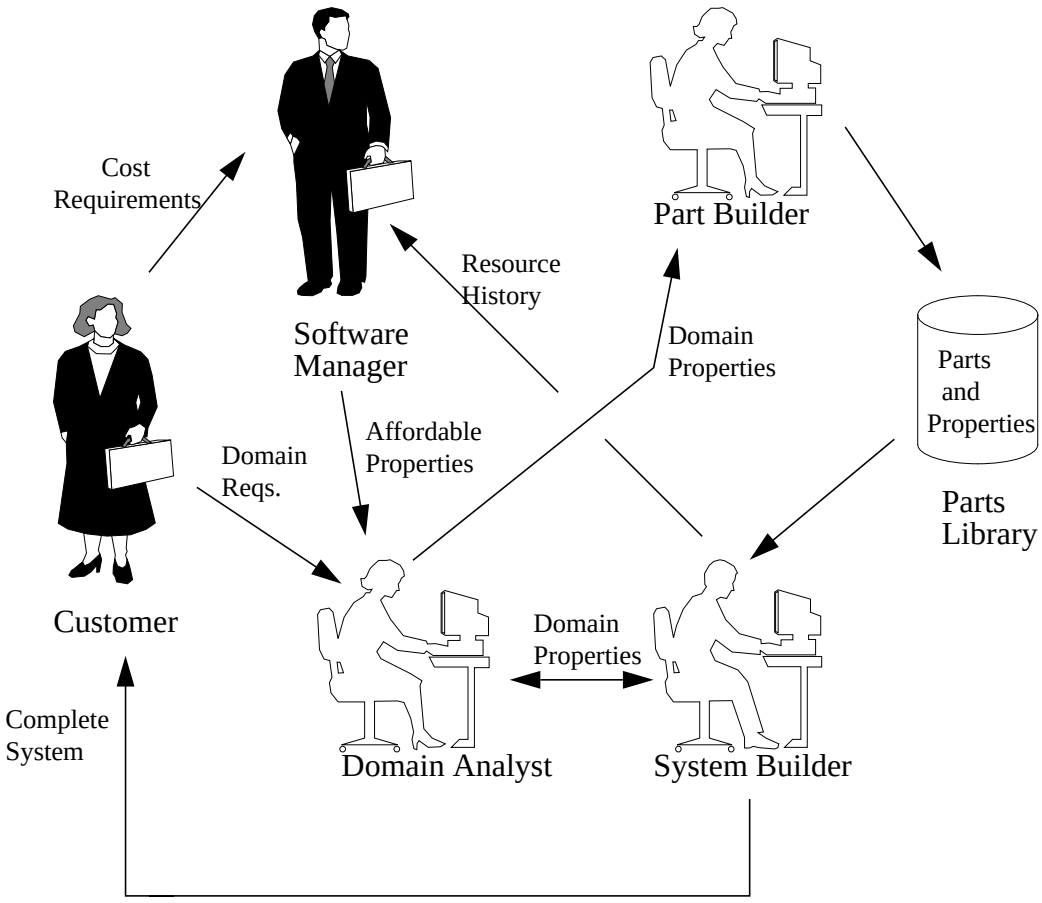

Figure 7 - Roles Of Various Personnel

domain, and the constraints of the business environment. It is up to the customer to communicate the domain requirements to the domain analyst, and work with the software manager to establish cost projections that fit his budget.

4.4.2 Role of the Software Manager

The software manager works with the customer in agreeing to target costs for the final system. The manager establishes targets partly with the use of historical data from previous projects in this domain, including amount of human resource spent in each phase of the development process, errors found during development, errors found after system deployment, changes made to the system after the specification phase, and changes made to the system after deployment. Given a target cost estimate, the manager must also determine the allowable cost for a certification effort. He must be aware of the relative cost for certifying a set of properties, and the trade-offs and risks involved in foregoing certification.

4.4.3 Role of the Domain Analyst

The domain analyst receives functional and quality requirements about the domain from the customer, and structures these into a formal domain analysis. The domain analyst also enhances this analysis by analyzing previously-deployed systems in this domain. Using this information, plus information about project cost constraints, the domain analyst produces a set of certification properties for systems in this domain, as well as a set of specifications for appropriate parts to be used by systems in this domain.

4.4.4 Role of the Part Builder

The part builder takes part specifications and domain certification properties from the domain analyst, determines the properties that must be present in the parts to accommodate the domain, and creates the parts. These certified parts are then made available to the system builder.

4.4.5 Role of the System Builder

The system builder uses the parts created by the part builder to create new systems. The system builder establishes that desired domain properties are present in the final system, working to large extent with the properties certified in the parts. The system builder communicates experiences gained from verifying the final system to the domain analyst, in order to further refine the domain analysis. The system builder also communicates project history data to the software manager, to assist in future cost estimation efforts.

4.5 Summary

A component certification strategy fits comfortably into the SPC's Synthesis process, and manifests itself in both the Domain Engineering and Application Engineering phases. Furthermore, a certification scheme must be regarded as an iterative process, where new insights are gained at each iteration.

The following chapter revisits the discussion of certification properties, providing details on methods by which component properties can be established.

5. Establishing Component Properties

While the previous chapters described some fundamental ideas underlying component quality properties and where certification fits in the software development process, in this chapter specific methods of establishing certified component properties are explored.

This chapter begins by enumerating the general issues in component quality assurance, and then examines the issues of anticipated and unanticipated adaptation. Several quality-assurance methods then discussed. However, because of its extreme importance, testing of source-code components is then focused on as a separate topic.

5.1 Issues In Component Quality Assurance

In determining how to set about establishing the quality of a component, many issues are raised by the fact that the components to which the techniques are applied will be reused. It is important to keep in mind that a reuse-oriented development paradigm is vastly different from traditional software development methods.

The general issues affecting the quality-assurance process for reusable components are:

- *Component Use*.

  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* (or more accurately reuse) [Rus87]. This is very different from the normal development situation in which an artifact is intended for a single use and its quality is usually assessed with that in mind.

  Quality assurance under such circumstances must be undertaken with a higher degree of generality than is normally encountered. Every possible variation in use of the component must be identified if possible and the relevant certification

properties shown to hold under all possible usage scenarios.

- *Component Type*.

  Quality assurance is complicated by the different types of work product and, within each work-product type, the different types of component. For example, very few reusable source-code components will be subprograms. Other components might 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 the detail is application-specific.

  In practice, the certification property of interest might apply to many or all of the types of components in a library, but the approach to establishing the property might differ with different component types.

  Abstractions employing symbolic constants and canonical designs are special cases of *adaptable components*. Adaptable components, which are discussed in detail in the next section, present significant challenges for quality assurance. The problem is that the certification property must hold for every possible instantiation of the component, that is it must hold for every possible specific instance of a deferred design decision.

5.2 Adaptable Components And Adaptation

A major complication in component quality assurance is introduced by the notion of *adaptable components*, that is components designed to be modified before use. The intention of making a component adaptable is to increase the opportunities for reuse by isolating some of the design decisions and leaving them to be made by the user of the component. Adaptable components range from the simple, such as an Ada subprogram in which the size of an array is set by a symbolic parameter, to the complex, such as an Ada generic unit in which generic parameters are used to set types and operations when the component is selected for reuse. Figure 8 shows an adaptable component with two different instantiations.

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 component intentionally by its author.

- *Unanticipated adaptation* occurs when a component is modified in a way that was not planned by the developer, usually using a text editor. Unanticipated adaptation changes a component in a way that is likely to invalidate many of the certification properties. Since the certification must be viewed as lost, unanticipated adaptation will be considered only briefly.

An important issue with adaptable components is the possibility of *adaptation restrictions*. 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 component. The design of the component, 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 component designed for reuse and the associated interrelationships might be quite involved.

In a more general context, consider the parameters used with Ada generic units. They are not merely numeric or symbolic but can be subprograms thereby allowing different instances to function entirely differently. A restriction imposed on an adaptation might be, therefore, 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. This situation raises the question of exactly how functional properties of adaptable components such as generic program units can be verified in any useful way.

```
-- simple adaptable component

generic
   type item is private;
   with function "*" (U, V) : ITEM return ITEM is <>;
function SQUARING (X : ITEM ) return ITEM;

-- two possible instantiations:

function SQUARE is new SQUARING ( MATRIX, MATRIX_PRODUCT );
function SQUARE is new SQUARING ( INTEGER, "*" )

-- these instantiations produce completely different code
```

Figure 8 - Specification Of An Adaptable Ada Components

5.3 Techniques For Component Quality Assurance

The general problem faced by the component certifier is that he has a component on the one hand and a quality that has to be demonstrated for the component on the other. The range of properties that can be expected to occur is enormous as is the range of component sizes and types. In addition, the component might be adaptable and the adaptation (or adaptations) might have significant restrictions placed upon them.

Establishing that a particular component has a given property is a quality-assurance activity, and an arsenal of quality-assurance techniques can be used to achieve it. These techniques can be loosely grouped into five main categories, *static analysis, formal inspections, testing, formal verification,* and *benchmarking*.

- *Static Analysis.*

  Static analysis is the process of showing automatically that a work product has a particular property without executing the work product (hence the term *static*). Automated tools, called static analyzers, exist that can check various useful properties of different types of work product, especially source code. Properties that can be checked by static analysis include absence of set/use anomalies with variables, absence of some forms of aliasing, and absence of unreachable code.

  To support certification, existing tools can be applied to reusable components and, where it is cost effective to do so, new static analyzers can be built to check appropriate certification properties.

- *Formal Inspections.*

  Formal inspection is a process in which a work product is examined in a systematic manner by human inspectors in order to establish that the work product possesses some useful property. Inspections can be applied to any textual work product in order to establish properties that are difficult or impossible to verify automatically. Properties that can be checked by inspection include consistency of specifications, correctness of designs, and completeness of test plans.

  The term ‘‘Formal Inspections’’ was introduced by Fagan [Fag76] and refinements of Fagan’s original idea have been suggested by Parnas and Weiss [PaW85] and by Knight and Myers [KnM91].

  Inspection can be applied to reusable components immediately and can be used to establish a wide range of properties. Because formal inspection involves human insight, it is the most general quality assurance technique.

- *Testing.*

  Testing is an experimental approach to quality assurance that attempts to show that a work product possesses some general property by example. All programmers are familiar with software testing, although most approach it as

more of an art than a science.

Although testing is used primarily in the verification of source code, it can be used to demonstrate properties of specifications or designs if they are expressed in executable notations, such as VDM [HeI88].

Testing can be applied to any reusable component that is executable. Its use is complicated by the fact that so many different types of testing are needed, and all have a place in a certification scheme.

- *Formal Verification.*

  Formal verification is the process of establishing a proof that a work product possesses some property. If a proof is produced (and if the proof itself is correct) then there is an absolute guarantee that the work product possesses the specific property. This should be contrasted with testing which attempts to do the same thing but by using examples.

  Properties that can be established using formal verification range from relatively simple, such as showing that a particular loop terminates, to extremely complex, such as showing that an implementation is correct with respect to a particular specification. The latter is often referred to informally as ‘‘proving correctness’’ although this informal term is extremely misleading since correctness cannot be defined.

  Establishing proofs in formal verification is usually tedious but the tedium can be reduced by employing a mechanical theorem prover. The effort involved is often questioned in any case, but, for a work product that will be reused, the effort can be amortized and the whole technique immediately becomes more attractive.

  Formal verification can be applied to any reusable component. Useful properties can be proved about specification components and design components as well as traditional source-code components provided the former are prepared in a notation that has a formal semantic definition.

- *Benchmarking.*

  Benchmarking is the process of determining the value of some performance measure associated with a work product. It is used primarily to measure quantities such as execution speed on a specific hardware/software platform for some standard application load.

  In the case of certification, benchmarking can be used to measure such properties as execution time, numeric precision, memory usage, and response time to specific events (such as an exception) of individual reusable components.

Table 9 summarizes these quality-assurance techniques and documents their advantages and disadvantages.

5.4 Testing Reusable Source-Code Components

In this section, a familiarity with the technology and terminology of software testing in general is assumed. The specific issues of testing reusable components are addressed using various existing techniques either directly or with appropriate modification.

The approach to testing of a reusable component depends on the type of the component. Three different component types are addressed in this subsection, specifically *functional abstractions* such as procedures and functions, *data abstractions* such as Ada packages, and *canonical designs* in which an entire system or subsystem structure is present but much of the application-specific detail is absent. Object-oriented abstractions such as C++ classes are an important approach to reuse. However, the complexity of testing classes incorporating multiple-inheritance semantics is beyond the scope of this report.

Any of these component types can be adaptable. Adaptable components usually cannot be executed without adaptation. Each specific adaptation represents a degree of freedom that has to be constrained in order to use the component, and the key question is whether the component will work correctly once these constraints or selections are installed. The problem of testing adaptable components amounts to ensuring that the adaptable component will function correctly assuming that an adaptation complies with the restrictions associated with design of the component.

Technique	Advantages	Disadvantages	Typical Uses
Static Analysis	Automated.	Limited application. Lack of available tools.	Simple design rule checking.
Formal Inspection	Widely applicable. Exploits human skills.	Labor intensive. Checking possibly incomplete.	Functional correctness checking.
Testing	Flexible. Gives confidence in product.	Not rigorous. Resource intensive.	Checking implementations.
Formal Verification	High degree of assurance.	Time consuming. Lack of appropriate support tools.	Safety-critical system quality assurance.
Benchmarking	Provides quantification.	Limited application	Time and space performance measurement.

Table 9: Quality-Assurance Techniques

In considering the testing of reusable components, the framework of certification must be kept in mind. The specific activities undertaken by the engineer responsible for testing a reusable component are to ensure compliance with the required certification property or properties. These in turn are determined solely by the associated domain analysis. No other testing is required nor is it appropriate.

5.4.1 Functional Abstractions

The starting point for procedures and functions is the existing practice of unit testing. Functions and procedures that are to become certified reusable components need to be tested using unit-testing methods to demonstrate whatever properties are required. Typical properties might be:

- Achieving a specific coverage metric such as statement or branch.
- Systematic demonstration of required functionality.
- Satisfactory processing of stress or extreme values.
- Demonstrated bound on failure probability.
- Any combination of the above.

Once properties such as these are established, properties derived from the reuse development paradigm need to be addressed. The fact that a reusable component must be prepared for every possible use dictates specialized testing in two major areas, *portability* and *symbolic parameterization*:

- *Portability.*

  A reusable component might be reused on different target computers from those for which the component was developed. Although attention to portability issues is well-established software engineering practice, it is not a quality-assurance check that is routinely applied at the unit level in non-reuse-based development. Testing a component on the entire range of target computers upon which it might execute is an essential element of component certification.

- *Symbolic Parameterization.*

  Parameterization in the sense used here is a special case of adaptation, and that topic is treated in more detail in section 5.4.4. Parameterization is discussed here because it is the simplest form of adaptation and because of its extensive use in components not otherwise adaptable.

  Carefully engineered software will make extensive use of programming-language parameterization facilities such as symbolic constants. The intent is to localize and focus size and bound parameters so that they can be changed easily. In non-reuse-based development, such parameters are usually set once and

forgotten until changes are dictated by maintenance. In a reuse-based development, such parameters might be set for each reuse and the setting is likely to be different in each case. It is important, therefore, that the existence of a symbolic constant be ascertained during component certification and the value of the parameter be treated as an additional input to the part. Thus testing needs to include test cases in which the values of symbolic constants are varied in an appropriate way.

5.4.2 Data Abstractions

Data abstractions such as Ada packages are a common form of reusable component. The starting point for packages and similar forms of modularization is the existing techniques of unit and integration testing. Certification of packages where some properties are to be established by testing needs to begin by using unit-testing methods to demonstrate whatever properties are required for each operation. Typical properties in this case will be the same as those itemized above for functional abstractions. Similarly, the special topics of portability and parameterization need to be addressed.

Once properties related to unit testing are established, the package can be assembled and integration testing applied. As noted with functional abstractions above, integration test plans must address the issues derived from the reuse development paradigm. Thus, for example, integration test plans must deal with the ‘‘all possible uses’’ problem by addressing the need to show that all possible sequences of uses of operations in the module will work correctly.

A major issue raised by data abstraction is the problem of testing *iterators*. Special problems that arise with iterators whose presence can be shown with testing are:

- *Empty structures.*

  It is common for iterators to fail when applied to an empty data structure. Often this possibility is eliminated in a specific use, but, when reusing a package containing iterators, this might be overlooked.

- *Large structures.*

  Because of their implementation, iterators sometimes have limits on the size of the structure to which they can be applied. Once again, this possibility might be eliminated in a single, specific use but this will not be the case if the component is intended to be reusable.

- *Operator interaction.*

  In some cases, iterators and other operators in a data abstraction interact. For example, addition and deletion operations can easily disturb the state so as to invalidate the information held by an iterator. Similarly, it is often the case that an application requires several instances of the same iterator to be running at the same time. This leads to subtle and hard-to-find faults unless the iterator is

designed and implemented correctly.

All of these issues with iterators occur in a non-reuse setting but are usually taken care of in an ad hoc fashion by those developing the system. This is not an acceptable approach in a reuse development environment and package test plans need to address the issues explicitly.

5.4.3 Canonical Designs

The concept of *canonical design* is unique to software reuse. A canonical design represent the framework or skeleton of a complete system or subsystem where this structure is determined by the associated domain analysis. Since a canonical design is intended to act as the basic structure of a whole family of systems within the domain, application-specific detail is omitted.

With extensive parts of the system missing, a canonical design is, in principle, not executable in any meaningful way. How then can a canonical design be tested if it cannot be executed? The answer lies first in not executing the entire component during the initial phases of testing, and second in providing specialized stubs termed *interface models* during the latter phases of testing.

An interface model is merely a stub with two unique properties. First it is able to absorb all parameters supplied when the stub is called. These parameters might be passed explicitly as traditional parameters or implicitly via shared variables. In either case, the interface model will accept the values supplied, check their types and, if appropriate, check the correctness of their values. The second unique property is the ability to generate all possible responses (return parameters, for example) that the canonical design might receive from the stub. Thus, for example, if a return parameter is an integer subtype, the interface model will have the capability of generating all possible values in the subtype during a comprehensive test of the canonical design.

The purpose of these two properties of interface models is to permit the suitability of the canonical design for every possible use to be checked.

Testing of a canonical design proceeds in a manner akin to the way that a system is tested:

- *Unit Test.*

  A canonical design will be composed of units organized to provide the required framework. These units do not depend in any significant way on the elements of an application not present in the canonical design. Thus traditional unit-test methods can be applied to them. Note that this might involve developing specialized test harnesses just as in any unit-test situation.

- *Integration Test.*

  The construction of a conventional system proceeds by integrating the various units and testing the resulting partial systems at each step, and a canonical design can be developed in a similar way using conventional integration test methods. Again, specialized test harnesses will probably be required.

- *System Test.*

  It is at the level of system testing that the unique reuse characteristics of a canonical design become evident and require special attention. It is not possible to use any traditional system test techniques on a canonical design. Application-specific functional testing is meaningless, for example, since there is no functional specification.

  The canonical design is not part of any system. It is essential that the canonical design function correctly in every system in which it is reused. Thus system testing in this context means testing the canonical design over all possible reuses in contrast with traditional system testing for a single application.

  This form of testing requires that data generators within the interface models be programmed to test the canonical design for all possible data sequences. These data sequences will be determined by the specification of the canonical design itself.

5.4.4 Adaptable Components

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 correct functionality, 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 testing approach suitable for a similar reusable component without adaptation. 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 issue that then remains is testing parts with adaptations that require functional parameters to be supplied. An example of such a part is an Ada generic procedure for sorting in which the element type and the comparison relation are supplied as generic parameters. The comparison relation is a functional parameter.

Since any functionality might be supplied for a functional parameter, the testing that can be done is limited by the need to cater to all possible functionality. It is essential to limit this variability to the maximum extent possible, and this observation is the key to the first step in testing this type of reusable component:

- *Identify all necessary constraints on the functional parameter.*

  The intent is to ensure that it will be possible to identify valid and invalid values for the functional adaptation parameter and to restrict the difficulties with testing. In the example of a sort reusable component, the functional parameter passed to define the comparison relation must impose a total ordering on the elements being sorted.

  Since the user of the component will have to comply with these constraints when using the component, they must be documented.

Careful examination of adaptable components always reveals a separation of concerns within the implementation in which some pieces depend on the functional adaptation and others do not. In the sorting example, the basic control structure imposed by the fundamental sort algorithm (Quicksort, insertion sort, etc) is independent of the type of the elements being sorted and the associated comparison relation. Since this is the case, a possible large fraction of the implementation of such components can be tested without regard to the functional adaptation parameter. This observation leads to the second step in testing this type of reusable component:

- *Instantiate the adaptable component with a conforming value for its functional parameter(s) and test the resulting instantiation as one would any non-adaptable reusable component.*

  Any instantiation of the component will retain the fraction of the implementation that is unaffected by the functional parameter, and testing the component after such an instantiation will permit effective testing of this fraction of the component.

The final step in testing such components is to ascertain the effect of the functional parameter on the results of the previous step. The constraints determined in the first step will reveal rules with which the functional parameter must comply, and these together with knowledge of the separation of concerns within the implementation permit determination of whether the results developed in the step above will apply to any valid functional parameter. This observation leads to the final step in the testing of this type of component:

- *Ascertain whether valid functional parameters affect the algorithm implementation in such a way as to invalidate prior testing.*

  This step will usually be undertaken by inspection or formal verification rather than conventional testing. It is not possible in general to test for the desired property. In the sorting example, it is necessary to ensure that any comparison relation that imposes the required total ordering does not affect the correct operation of the remainder of the implementation. In the case of sorting, this is a very simple result to establish.

Turning now to the specifics of testing adaptable, reusable, source-code components written in Ada, the forms of adaptation in Ada are:

- Units containing symbolic parameters.
- Units depending on conditional compilation.
- Generic units.

Further complicating the task of testing are the different forms of generic parameters that can be used in an Ada component, specifically:

- Generic formal objects.
- Generic formal types.
- Generic formal subprograms.

The advocated approach to testing an adaptable Ada component is as follows:

(1) Determine the type of component: functional abstraction, data abstraction, or canonical design.

(2) Establish an appropriate test plan for the component based on its type using the techniques discussed above assuming the component is not adaptable.

(3) Determine all forms of adaptation used by the component.

(4) For each form of adaptation, determine each instance of the adaptation used by the component.

(5) For each symbolic parameter:

- Develop a suitable set of values for the symbolic parameter viewing it as an input variable. Techniques to consider include (a) all possible values, (b) statistical selection of values, and (c) extreme values. For each selected test value of the symbolic parameter, perform the entire test plan for the component that was established in step 2. In the sorting example, the size of an array might be a symbolic parameter and values to consider for the size might include 1, 1023, 1024, and the maximum permitted by the part design.

(6) For each instance of conditional compilation:

- Locate the condition used to control the compilation and instantiate the component for each value of the condition. For each instantiation, perform the entire test plan for the component developed in step 2. In the sorting example, conditional compilation might be used to select different sort algorithms for different ranges of file size to be sorted or different element types. In that case, all possible combinations of selected compiled program need to be tested completely.

(7) For each generic parameter:

- Instantiate the component with a specimen ‘‘value’’ for each generic parameter. Perform the entire test plan for the component developed in step 2

above. By inspection of proof, determine that the test results are independent of the particular generic parameter value that was used.

The approach just outlined calls for the repetition of the entire basic test plan many times. This is unavoidable. Consider, for example, an adaptable part that uses symbolic parameters, conditional compilation and generic parameters to provide flexibility. It is essential that the developer of this part assure himself that it will function correctly in every possible reuse. This can only be done by careful, systematic testing of the various forms that the adaptable component can take when instantiated. It is not sufficient to test a single value of a symbolic parameter - others, especially special cases, might not work. It is not sufficient to test a single set of conditional compilation instantiations - other combinations might not work. Further, any difficulties that arise will be faced by the user of the part and not the developer. These two individuals might not be able to communicate about the issues so raised.

5.5 Specific Guidelines

Establishing part properties assumes the existence of two sets of items: a set of components and a set of desired properties for those components. The goal is to ensure that the components do in fact possess the desired properties. To do this, the following steps should be followed:

(1) For each certification property, determine which quality-assurance technique or techniques (static analysis, formal inspections, testing, formal verification, benchmarking) can be used to establish the technique. Carefully document the resulting list. Have the list checked by several engineers.

(2) Determine in which components these properties will be established.

(3) Create a checklist of certifiable properties for each component to be certified. Ideally, this checklist will be in electronic, machine-processable form, and will reside in the reuse library along with the component itself once it is completed.

(4) For each certification property, develop a plan for the application of the associated quality-assurance technique. Pay particular attention to the issues raised in section 5.2 - make sure that the technique addresses the goal of certification for every possible use, that it is suitably applied for the different component types, and that the special requirements of adaptable components are dealt with.

(5) Apply the selected techniques, noting the results on the checklist.

6. Establishing Properties Of Systems

As complex as establishing component properties is, it is essentially an empty exercise if inferences about component-based work products cannot be drawn from them. Except for the software component industry itself, customers purchase systems, not the components from which the systems are built. This chapter focuses on the issue of using certified components and exploiting the certification properties to establish properties of complete work products.

Recall that the basic goal is to exploit the various *component properties* established by certification to permit the rapid demonstration of *domain properties* of work products. In

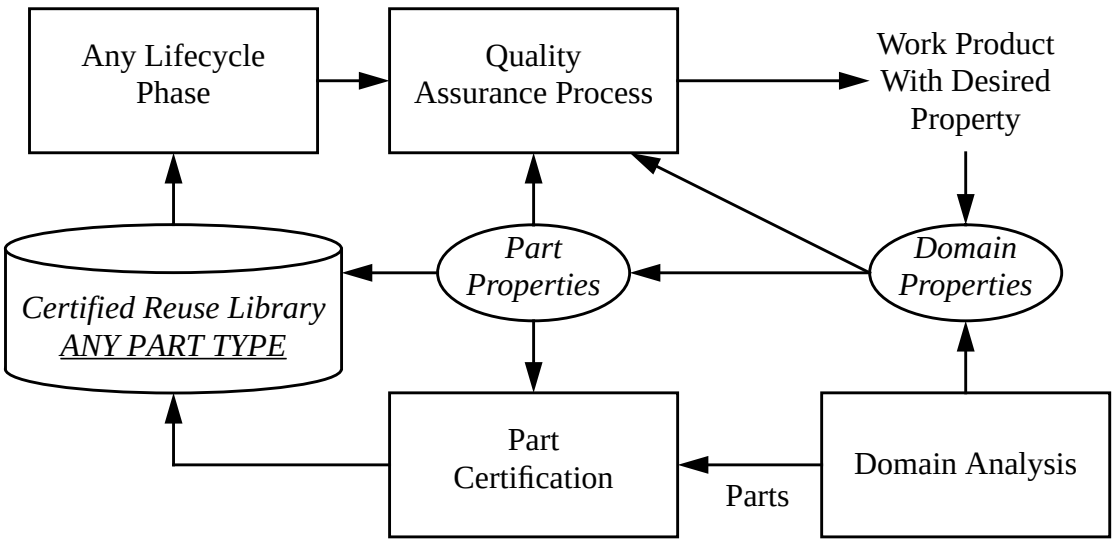

Figure 9 - Development Process

this section, a system is defined to be all the work products associated with a particular product development. The basic process for exploiting the properties of parts is repeated in Figure 9.

6.1 Issues In System Quality Assurance

Once prepared and placed into a reuse library, taking advantage of the certification properties of components raises several issues, specifically:

- *Custom-built elements.*

  Even where reuse rates are high, no work product is ever likely to be comprised completely of reusable components. The existence of custom-built elements in the final work product detracts from the immediate exploitation of the certification properties.

- *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 [Gar87, Mey87, Ric89].

- *Component revision*.

  As with any software, a reuse library will be the subject of revision. Components 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 verified. Can revised components with "identical" interfaces be trusted, and, if not, what verification needs to be performed when revised components are incorporated?

- *Part adaptation*.

  Adaptable components as targets of certification were discussed in the previous section. If a work product incorporates adaptable components, then adaptation was performed when the component was introduced into the work product. In some cases, this affects the way that certification properties are exploited in showing properties at the work-product level.

6.2 Techniques For System Quality Assurance

The general problem faced by the engineer performing system quality assurance is very similar to the problem faced during component quality assurance: he has a work product (or a work product under development) on the one hand and a quality that has to be demonstrated for the final work product on the other. As is the case with individual components, the range of properties that can be expected to occur is enormous as is the range of work product sizes and types.

Establishing that a particular work product has a given property is again a quality-assurance activity, but virtually no existing techniques can be used. The problem is essentially one of inference rather than direct quality assessment. The required inference techniques have to be developed as needed and fall into two main categories, *immediate inference* and *analytic inference*:

- *Immediate Inference.*

  Many useful and important properties of a work products can be inferred immediately from component properties.

  As an example, consider stylistic issues. If all the reusable components used to develop a work product (such as a specification in the A-7E style or an implementation in Ada) follow desired stylistic rules, then the resulting work product will also comply with these rules to the extent that it is built from reusable components.

- *Analytic Inference.*

  Some properties of a work product cannot be inferred from component properties without performing some form of analysis. In such cases, the work product property of interest might not be related in an obvious manner to the associated component properties and might, in fact, depend upon several component properties.

  As an example, consider the problem of showing that an implementation in Ada is free of deadlock. This can only be done after extensive analysis and will rely on component properties such as component freedom from deadlock, documented and verified component entry-call sequences, component independence of the scheduling policy, component freedom from anomalous exception propagation, component guaranteed termination, and verified guard semantics in component select statements.

6.3 Using Adaptable Components

In Chapter 5, the nature of adaptable components was discussed and associated issues

in component quality assurance were addressed. In this section, the use of adaptable components and their effect on system qualities are considered.

Components are made adaptable to enhance their reusability. Unfortunately, introducing adaptability often brings with it important restrictions that the component developer was unable to avoid. These adaptation restrictions must be adhered to if the component is to operate correctly. The first task in system quality assurance is to ensure that all adaptation restrictions have been identified and complied with.

6.3.1 Anticipated Adaptation

If an adaptable component 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 [Gar87]. 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 component.

In a non-reuse setting, this verification will be performed by the author of a component. If the component is placed into a reuse library, however, the checks must be performed 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 [Luc85] 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 component. 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 components to be used and the associated constraints machine checked is shown in Figure 10.

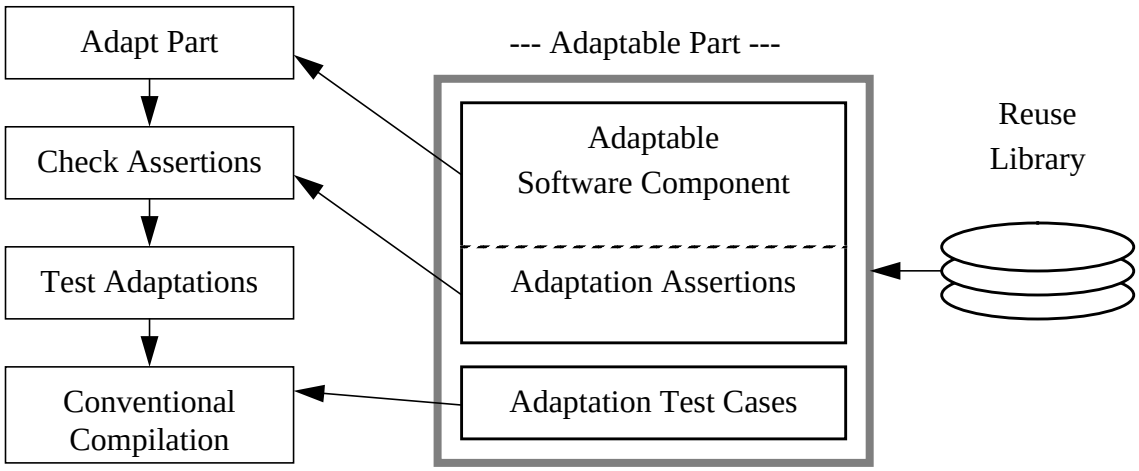

Figure 10 - Checking Anticipated Adaptation

6.3.2 Unanticipated Adaptation

On occasion, arbitrary changes made using an editor might be required when attempting to reuse an existing component even if the component 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 component 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 component is different from the original component 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 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 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 components with executable assertions [And81, Luc85, Mey87]. In fact, Anna [Luc85] is described as a notation for specification although it does not have the completeness characteristics of a rigorous approach such as VDM [Jon86]. However, Anna does provide a rich notation for writing executable assertions.

The role of instrumentation using assertions is to include design information with the component, 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 [Lev87]. Executable assertions can be used therefore as part of a system for checking components subjected to unanticipated adaptation.

6.4 Examples Of Domain Properties

In this section, some specific examples of the use of component properties to establish, at least in part, significant domain properties in developed systems are presented. The examples given are for source programs written in Ada since they are considered to be of the most immediate use.

Many significant opportunities exist for exploiting certification in Ada source-code development. Simple though very useful properties, such as presentation, maintenance, and adherence to some set of programming standards are in the category of immediate-inference properties. Such component properties map fairly obviously into useful domain properties and are not discussed further.

6.4.1 Exception Handling In Ada

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 raised again 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 [How91]. 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 handler 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 unless great care is taken with system design. By following some fairly elaborate design rules, systems can be built that are free of these and other exception-related difficulties but following such rules and confirming their correct implementation is complex.

Reuse of certified components can help to deal with this problem. The correct use of the design rules is set as a domain property and suitable part properties derived to support establishment of the domain property in systems built from the associated reuse library. In practice, there are several different sets of component properties that will permit the domain property to be inferred. The following is a simple example set of behavioral properties:

- For leaf components, the following properties are known to hold:
  - Handlers exist within the component for all exceptions that are declared within the component. No such exceptions can be propagated out of the component thereby becoming anonymous.
  - Handlers exist within the component for all predefined exceptions. No predefined exceptions can be propagated out of the component.
  - All possible subprogram call sequences have been generated during testing of the component and no path exists in which an exception becomes anonymous or in which propagation is unexpected
  - All cases in which the component could raise an exception are fully documented as to exception name and circumstances.

- For subsystems or canonical designs:
  - Within the call structure, handlers for `others` have been included in a "firewall" structure to ensure that no unbounded exception propagation can occur at the subsystem or system level. If the subsystem is terminated by an unanticipated exception propagation, provision is made within the component 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 components, at least not because of defects in the reusable components.

6.4.2 Ada Task Synchronization

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

- No shared variables are referenced within the component.
- There is no internal concurrency within the component.
- No execution conditions with the component can lead to tasking error.
- Correct operation of the component has been shown not to depend on specific priority values nor on specific system scheduling algorithms.
- All entry calls made by the component 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 components, 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.

6.5 Specific Guidelines

To establish a set of work product properties from a set of component properties, these sets of items must be available: the set of properties of the components used to build the work product, the work product itself, and the domain properties created during the enhanced domain analysis. The specific steps are as follows:

(1) Determine which pieces of the work product are composed entirely of reusable parts, and which are composed of custom-built elements. Realistic quality inferences can only be drawn about those areas of the system consisting of certified components.

(2) Determine which domain properties can be drawn by means of immediate inference from the component properties. This will often include simple but useful properties such as “adheres to stylistic guidelines”.

(3) Determine which domain properties must be drawn by means of analytic inference from the component properties. It is useful to think of the system property as a logical assertion whose truth can be shown by ANDing and ORing together an appropriate set of component properties. For example, “System property SP is true if component properties CP1 AND CP2 AND CPn are true”.

(4) Apply traditional quality assurance techniques to the parts of the system not comprised of certified components.

7. Economics Of Certification

The primary motivation for any reuse scheme is economic. The goal of certification according to the detailed structure developed here is to enhance the expected economic benefits of reuse. The system properties facilitated by certification are usually expensive to establish, so economic benefit accrues directly when certification is exploited.

While beneficial, a component certification process is not without costs. Any organization considering certification must weigh the trade-offs and decide whether the benefits are truly worthwhile. This chapter summarizes the economic trade-offs to consider in a certification program, and provides guidelines by which an organization can determine the potential payback of implementing such a program.

7.1 Economic Trade-off

The benefits of a certification strategy are realized in the following areas:

- *Reduced development effort.*

  Use of high-quality parts should result in reduced quality-assurance and rework costs for the work products that include these parts.

- *Reduced maintenance effort.*

  Parts that conform to a clear and accepted set of standards yield work products that contain fewer format and usage deficiencies and contain fewer errors requiring correction.

- *Higher reuse levels.*

  Software engineers are more likely to use high quality parts.

These are significant benefits. But what are the costs of realizing them? The following

recurring costs are generated by certification:

- *Extended domain analysis.*

  Quality-assurance properties required of products within the domain have to be determined.

- *Certification instantiation.*

  An instantiation of certification has to be generated.

- *Establishing component properties.*

  Components that are useful but lack properties might have to be reengineered to comply with desired properties.

- *Checking compliance with component properties.*

  For each component in a reuse library, all the relevant certification properties must be shown to hold.

- *Exploiting component properties.*

  For each work product developed with certified parts, the desired properties of the work product have to be established.

- *Dealing with custom-built elements.*

  All realistic work products will contain custom-built elements and these contribute to the properties of the overall work product.

It is easy to be misled into thinking of certification as something an organization does in addition to its normal software development process. In the model presented in this report, an organization will identify quality properties that systems within a domain will have to possess by means of an extended domain analysis. An organization not using certification will still need to have an established set of quality criteria for deployed systems. In either case, these properties will have to be shown to hold before the system is deployed. The difference is that if certification is used, then the properties of the system's components can be used to help establish the system's properties. If component certification is not used, these properties will have to be established "from scratch" at a potentially higher cost as part of the normal quality assurance process.

The following two sections describe these costs and benefits in more detail.

7.2 Benefits

7.2.1 Reduced Development Effort

Development costs are reduced in a number of ways if properly certified reusable parts are used during development. Specifically:

- *Improved initial quality.*

  The initial quality of work products will be higher because they will have been composed of high quality components.

- *Simpler quality assurance.*

  Demonstration of the quality of work products will be simpler because the exploitation of certification properties deals with work-product qualities at a higher level than is usually the case. For example, a property definition for a source-code function will be expressed as a single statement about the entire function rather than as a set of statements about individual lines of code within the function.

- *Easier defect location.*

  Correcting defects in a work product detected during development will require less effort because locating the defect will be simpler. Consider, for example, a subsystem consisting of custom elements together with a set of certified components. If flaws are found during quality assurance and the components are certified to be free of these types of flaws, then the number of places in the subsystem that need to be investigated has been reduced.

- *Reduced rework.*

  The total amount of rework that has to be undertaken is reduced. Figure 11 illustrates the development and rework relationships that occur in the software creation process. Since all lifecycle work products are subject to some form of verification, flaws found during verification require correction, and in so doing, might result in the modification of subsequent lifecycle work products. For

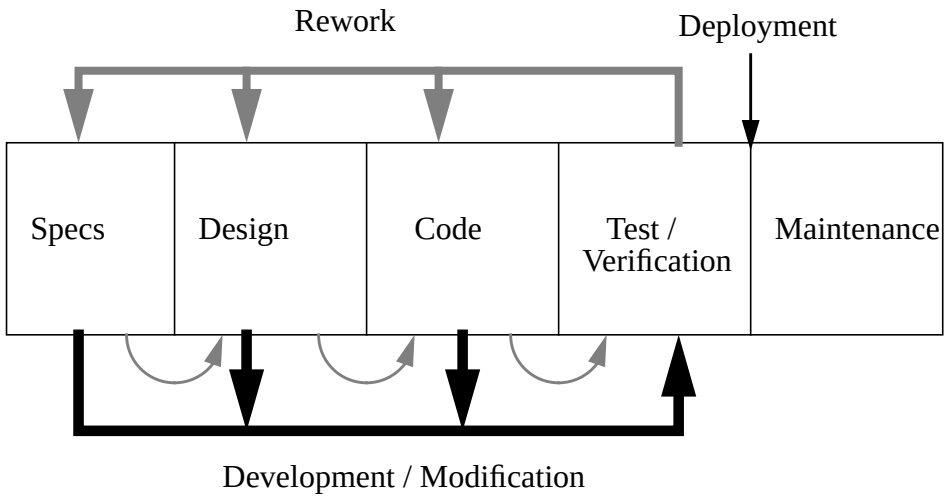

Figure 11 - Software Development Versus Rework

example, a problem uncovered while testing a program might be traced back to a fundamental misunderstanding during the specification phase. As shown by the small semicircular arcs in Figure 11, reworking the specification would also imply reworking the design, which implies reworking the source code. Building higher-quality work products throughout the lifecycle will provide an overall reduction in rework activities and hence a reduction in development costs.

7.2.2 Reduced Maintenance Costs

The so-called maintenance phase of the software lifecycle is usually the most expensive phase of all. Clearly, during the deployed life of the system, changes will need to be introduced periodically to account for new requirements, changes in the operating environment, or to repair defects. Maintenance is expensive because of the number of modifications and enhancements required by most long-lived systems and because of the often formidable difficult of making these changes.

The same benefits cited for development hold true for maintenance, and both cost elements of maintenance are addressed by a reuse development process utilizing certified reusable parts. Components guaranteed to conform to a set of quality standards will contribute to the development of work products that are initially of higher quality than is typical, and secondly will yield work products that are easier to comprehend, modify, and enhance than is typical.

7.2.3 Higher Reuse Levels

As indicated earlier, wariness about part quality by software engineers tends to inhibit the acceptance of reuse-oriented development processes. It is hypothesized that the rate will of reuse will increase with the introduction of certified reuse libraries for two reasons:

- *Higher part uniformity.*

  Most certification instances are likely to include a number of properties pertaining to conformance to a set of standards. As a simple example, consider a set of natural-language specification parts that conform to the same set of text formatting standards. Analysts who know that these parts can be combined to form a seamless document are more likely to use them than they would a set of parts with wildly different font sizes, typefaces, and layouts even if the parts had useful content.

- *Higher confidence in parts by programmers*.

  This is an extension of the first point. Engineers who know that a set of parts conform to a rigorous set of quality standards are more likely to use them than build their own. The cost of rebuilding rather than reusing goes up dramatically once quality aspects are considered. The engineer is faced with not only having to develop an artifact to match his functional needs but also to show that the artifact possesses all the required non-functional characteristics. Most engineers

realize that the bulk of the effort is in this latter process, and the benefit of reusing a *certified* part is very clear.

7.3 Costs

7.3.1 Extended Domain Analysis

Since a domain analysis will have to be undertaken anyway, this is an increment on an existing cost. The additional effort involved is likely to be quite small and this cost category is probably insignificant.

7.3.2 Instantiating Certification

The major cost of instantiating a certification definition is in the analysis necessary to determine the component properties that are required to support the work-product properties that are identified in the extended domain analysis.

If the staff undertaking this stage of the process are familiar with the technology, then the total cost incurred is once again small. The first time that a certification instance is developed, there will be a steep learning curve and an inevitable need to adjust the instance as experience is gained.

7.3.3 Establishing Part Properties

This is a highly variable and possibly substantial cost. It is, however, a capital investment rather than a non-recurring cost in the same sense as the creation of a reusable component is. Thus this cost is amortized over the various reuses of the component that occur.

Where cost benefit analysis is undertaken to evaluate the merit of a certification program, the costs of establishing part properties can be estimated using simple variations of software economic models such as the COCOMO model.

7.3.4 Checking Compliance With Certification Properties

Once again, this is a highly variable but non-recurring cost. The cost will be determined to a large extent by the properties included in the certification instantiation and the techniques available for checking compliance. The cost will be higher for specification components written in an unusual notation because few tools are likely to be available for the notation. In contrast, where a property can be checked by static analysis, the cost in this area is essentially negligible.

In undertaking a cost/benefit analysis for certification, each property in the instantiation has to be examined in isolation and a domain-specific cost model developed.

If new software tools are deemed necessary to support the checking of component compliance with certification properties, major costs will be incurred in tool development. This is a highly variable and hard-to-quantify cost area.

7.3.5 Exploiting Certification Properties

For properties resulting from direct inference this cost is negligible but the cost associated with analytic inference might be quite high. The specific costs will depend entirely on the system property to be inferred and the techniques to be employed.

If the technical approach involves human effort, the cost will necessarily be high. Software tools can be developed to facilitate exploitation of certification but their creation is a capital investment. Once again, this is a highly variable and hard-to-quantify cost area.

7.3.6 Custom-Built Elements

The cost of dealing with custom-built elements included in a work product is largely unaffected by certification. The effort involved in dealing with the custom-built elements might overwhelm the gain from certification though this is unlikely.

7.3.7 Other Cost Factors

There are other factors that affect the cost incurred in certification and hence the decision about whether to undertake certification, in particular:

- *Size and number of components.*

  As indicated in Chapter 3, a component can range in size and complexity from a one or two line fragment to a multi-module subsystem. The trade-off here is that small components are generally easy to certify, but one might have a large number of them to deal with. Conversely, one might be faced with a small number of large, difficult to certify components. The benefit to the organization of having certified parts that are large is, however, very substantial.

- *Variation in properties.*

  Certification can be either inexpensive or expensive depending on the types of properties the organization chooses to certify. The fact that some properties are easier to certify than others means that the organization can optimize the way it uses its resources. One would expect such properties as compliance with formatting standards to be checked automatically, virtually for free. For those properties that require human intervention, the organization can determine which ones can be certified by junior-level people, and which would require

highly skilled, senior-level people.

7.4 Specific Guidelines

In this section, specific guidelines are presented that can be followed by an organization in order to perform the necessary cost/benefit analysis associated with certification. The results of the analysis will facilitate subsequent decisions about a certification program. Three main areas are touched upon: determining whether a process will be accepted by the engineering community, determining the cost versus quality impact a process will have on systems developed by the organization, and determining the cost of implementing the process throughout the organization.

(1) Determine where resources currently are being spent.

A number of organizations keep track of this information by using weekly timesheets or other labor claiming methods. Those that do not will probably have to resort to surveys. The key pieces of information are the amount of time being spent up front in each phase of the software lifecycle, and how much time is being spent in rework.

(2) Determine the level of reuse that is currently occurring in the organization.

Those organizations that already have an organized reuse process in place should be able to gauge this by the amount of activity against their reuse libraries.

(3) Determine the perception among the engineering community of factors inhibiting reuse.

This information is often best determined in face-to-face meetings with groups of engineers.

(4) Determine whether the engineering community will accept the notion of a rigorous certification process.

If the people who are supposed to use the certification process do not see the value of it and refuse to go along with it, the process will fail. Introducing new processes to large organizations can be a source of bitter political battles and endless frustration if everyone involved is not committed to making it work.

(5) Determine the cost of establishing each certification property.

Fairly trivial properties, such as “Conforms to local formatting / typesetting standards”, can be verified and enforced automatically. Complex properties such as “Guaranteed free of deadlock” can involve exhaustive inspection by experienced engineers.

(6) Determine the benefit gained in each work product as a result of each certification property.

The benefits are likely to be both tangible and intangible, and must be judged by these attributes:

- The context in which the system will be used.
- The functional requirements of the domain.
- The lifecycle characteristics for this type of system.
- Specialized requirements of the customer.

For example, certifying that a part conforms to local formatting standards might appear on the surface to be a nicety with little or no payoff, but if it is known that systems using the part are subject to high maintenance activity, then this property might become quite important. On the other hand, exhaustively certifying the functional correctness of a part of the system that the customer knows is of minor importance might be overkill.

(7) Determine the risk associated with not establishing each certification property.

Again, this depends on the same factors as the previous step. For example, guaranteeing freedom from deadlocks might appear to be important for all real-time systems. For an avionics system, the risk of deadlock might be a catastrophic crash. However, if the system in question is a video game, the risk might be that the user has to hit a function key to restart, and certifying such a property might not be cost justified. One sees, then, how intimately entwined the assessment of benefits must be with the domain analysis.

(8) Identify a set of projects to serve as “pilot projects” for the organization.

A complete certification process cannot be implemented all at once in any software organization; it must be phased in gradually. A set of small pilot projects will give members of the organization time to become familiar with the methods and technology, and provide insight into the best way of rolling the process out to the rest of the organization. Most pilot projects, if done well, take at least six months to a year, so this is an important element of a cost justification.

(9) Devise an implementation plan for the organization.

This plan will include time frames for programmer education, creation and evaluation of certification instantiations, and installation of necessary support tools. For a large (one hundred or more programmers) organization, fully integrating a complex process can take at least a year.

Appendix A: Case Studies

This appendix contains case studies of two certification instantiations. The first is an instantiation for specification components, and the second is for source-code components. They are intended to serve as terse certification “cookbook” examples for the software practitioner. A sample application domain is presented, along with typical certification instances. It is shown how the properties in the certification instances can be applied to the quality assurance of a system built with a set of certified parts.

The instantiation for specifications is based loosely on the assumption that the A-7E technique and notation will be used for the specification components since this technique is reasonably rigorous, yet well-known. The instantiation for source-code components is based on the assumption that the components will be written in Ada.

A.1 Example Application Domain

A.1.1 Application Overview

The application domain of interest is that of nautical navigation for ships [DuK91]. Computers are being used increasingly to help navigators plot their ship's course, monitor their progress, track voyage history, and so on. Many modern bridges feature graphics terminals displaying the ship's current position relative to features of the coastline, navigation channels, and other ships in the area picked up by radar. In addition, ship control itself can be integrated into such systems.

A.1.2 Domain Analysis Overview

The main functional aspects of this hypothetical example domain are:

- A digitized navigation chart is displayed to scale on a bitmapped graphics display.
- Highlighted icons are superimposed on this chart representing radar images of other ships in the area.
- Also superimposed on this chart is a highlighted icon representing the navigator's own ship.
- Navigation directions are entered via this display and ship control is automatic thereafter.
- The system operates in real time and is written in Ada.

A.1.3 Extended Domain Analysis

The quality properties required of products built using certified, reusable components are determined in an extended domain analysis.

In this hypothetical example, the main quality features of interest for complete specifications are:

- *Portability*: Assuming Interleaf as the desktop publishing platform, it should be possible to create a complete specification document without regard to Interleaf release level, and it should be possible to print the complete document on any laser printer attached to the system.
- *Safety*: Systems built using the specifications should account for all domain-specific hazards, and adhere to guidelines set forth by the appropriate regulating agencies.
- *Maintainability*: It should be a straightforward matter to modify the specifications to account for changes to the domain, error corrections, and

version upgrades.

- *Completeness*: The resulting specification should be functionally complete and complete in its statement of processing requirements for each possible input value and value combination.
- *Consistency*: The resulting specification should be consistent in its meaning such that conflicting requirements do not appear.

In this hypothetical example, the main quality features of interest for delivered software are:

- *Portability*: It should be reasonably easy to port the software to new hardware and operating system platforms.
- *Safety*: The display must give an accurate portrayal of where the ship is, or collisions or groundings might result. Also, highly available software is required; the need to reload the system because of a software fault must not take too long or the vessel could be endangered. Systems incorporating real-time control functions must ensure that effective and safe control is maintained under all operating conditions.
- *Performance*: The display of moving objects must happen in real time.
- *Extensibility*: It should be reasonably easy to extend the software's functionality. For example, it might be desirable to display new navigation features. Or it might be desirable to sound an audible alarm if the ship passes within a certain number of meters of a known hazard or another ship.
- *Maintainability*: It must be easy to dispatch fixes and updates to the software, since it will be deployed on a moving platform that will often be located in remote parts of the globe.
- *Functional Correctness*: Systems must demonstrate very high levels of functional correctness.

A.2 Case Study 1 - Specifications

This case study examines the preparation of specifications from certified reusable components. The components are written in stylized English using the A-7E techniques.

A.2.1 Contents Of Reuse Library

From the domain analysis, it is determined that the following reusable specification components are needed:

- Background map layout specification. Adaptations permit content, symbol, and graphic variations.
- Map overlay specification. Adaptations permit content, symbol, graphic, and motion variations.
- Menu-panel specification. Adaptations permit menu graphics, button text, and button actions to be varied.
- Image manipulation specification including zooming, offsetting, centering, and decluttering.
- Sensor input interface and actuator output interface specifications including data formats, data types, data rates, and data transmission protocols. Adaptations permit all data characteristics to be varied.
- Coordinate systems and appropriate transformations between them.
- Course planning, navigation, distance, and guidance algorithms. Adaptations permit the use of all available coordinate systems.
- Real-time and associated scheduling specifications to permit all necessary timing requirements to be specified.
- Templates for control subsystem, graphic subsystem, and navigation subsystem specification documents. Used as ‘‘chapters’’ in complete application specifications.
- Templates for basic document formats.

The next section shows a sample certification definition for specification components in this reuse library.

A.2.2 Certification Instantiation

To uphold the desired properties of the domain, the following properties need to be certified in the specification components. The definitions are partitioned according to whether the property refers to an aspect of the component itself (structural properties), or an aspect of what the component does (behavioral properties).

Property Class	Property Definition	Certific'n Method
Performance	All appropriate real-time constraints defined.	Inspection
	All appropriate main-memory constraints defined.	Inspection
	All appropriate I/O performance constraints defined.	Inspection
Safety	Specified inputs are necessary and sufficient for producing the specified outputs.	Inspection
	For every possible input value an action is specified.	Inspection
	All undesired events and appropriate responses identified.	Inspection
	Complies with standards of appropriate regulatory agency.	Inspection
Precision	All appropriate numeric accuracy requirements stated.	Inspection
Completeness	All I/O sources (human, sensor, network, other) identified.	Inspection
	All allowable I/O data types defined and correctly parameterized.	Inspection
	All allowable specific data values identified.	Inspection
	All assumptions about platform hardware stated.	Inspection
	All assumptions about platform system environment stated.	Inspection
	No assumptions made about specific implementation algorithms.	Inspection
	No assumptions made about specific programming languages.	Inspection
	All modes of operation have been identified.	Inspection
	All transitions between modes identified.	Inspection
	All major periodic and demand functions identified.	Inspection
	Initiation and termination events for functions identified.	Inspection
	All expected changes to the system documented.	Inspection
	Removable subsets identified.	Inspection

Table 10: Behavioral Properties - Specifications

Property Class	Property Definition	Certific'n Method
Reusability	All data items are associated with meaningful symbolic names.	Inspection
	All likely-to-change text elements parameterized.	Inspection
Maintainability	Complete cross-reference of symbols and symbol uses.	Static Analysis
	Meaningful symbolic names for paragraph types.	Inspection
	Consistent and systematic use of paragraph types.	Inspection
Portability	Prints correctly with Interleaf running on all available platforms.	Benchmark
	Compatible with current and future Interleaf releases.	Inspection
Presentation	Paragraphs appear in 12 point plain Roman.	Inspection
	Template keywords appear in 12 point boldface Helvetica.	Inspection
	All step and bullet symbols are followed by 0.25" space.	Inspection
	Correct spelling used.	Static analysis
	Correct grammar and style used.	Inspection
	All names bracketed by appropriate A-7E delimiters.	Inspection

Table 11: Structural Properties - Specifications

A.2.3 Certification Exploitation

Certification exploitation is the process by which the certification properties known to hold for the reusable components used to build a work product are used to demonstrate useful properties of the entire work product. For the sample domain studied here, the goal is to produce high-quality specifications in the A-7E style for products relating to nautical navigation for ships.

Suppose a comprehensive control system for a new vessel is required. A complete and accurate specification can be built quickly and efficiently using certified parts from the hypothetical library of specification parts. The starting point is an appropriate basic document template and development continues with the selection of appropriate subsystem templates and lower-level detailed specifications.

The preparation of the specification through reuse would proceed according to some standard development process. In this section, some example properties of the complete

specification that can be derived from the certification properties of the reusable components are shown. Many useful properties both simple and complex can be established using certification technology. Other examples will no doubt suggest themselves to the reader.

Table 12 shows several work product properties and the certification properties from which they are derived. See also Chapter 6.

Domain Property	Component Property	Inference Mechanism
Portability across publishing platforms.	All portability properties.	Immediate
Upward compatibility with Interleaf.	Component compatibility with Interleaf.	Immediate
Correctly formatted document.	Component presentation properties.	Immediate
Easily modified document text.	Presentation, maintainability, and reusability properties.	Immediate
Easily modified document content.	Completeness and safety properties.	Analytic
I/O subsystem functional completeness	Completeness and safety properties.	Analytic

Table 12: Exploiting Specification Certification

A.3 Case Study 2 - Source Code

This case study examines the preparation of source-code from certified reusable components. The components are written in Ada.

A.3.1 Contents Of Reuse Library

From the domain analysis, it is determined that the following reusable source-code parts are needed:

- Primary map element abstract data types including land masses, shoals, buoys, ships, navigation channels, and hazards. Adaptations permit control of symbol, color, shape, location, and size.
- Background map abstract data type. Adaptations permit control of view, orientation, position, motion, and background.
- Map overlay abstract data type. Adaptations permit content, symbol, graphic, and motion variations.
- Menu-panel abstract data type. Adaptations permit menu graphics, button text, and button actions to be varied.
- Window abstract data type with operators providing zooming, offsetting, centering, and decluttering.
- Sensor and actuator device drivers. Adaptations permit various data formats, data types, data rates, and data transmission protocols.
- Functional abstractions providing transformations between coordinate systems.
- Canonical designs for subsystems that implement course planning, navigation, distance, and guidance algorithms. Adaptations permit the use of all available coordinate systems.
- Canonical designs for general synchronous and asynchronous real-time systems.
- A general-purpose graphics package.
- A general-purpose matrix manipulation package.
- Low-level timing, scheduling, interrupt handling, and associated real-time components.

The next section shows a sample certification definition for source-code components in this reuse library.

A.3.2 Certification Instantiation

To uphold the desired properties of the domain, the following properties need to be certified in the source-code parts. The definitions are partitioned according to whether the

property refers to an aspect of the code itself (structural properties), or an aspect of what the code does (behavioral properties).

Property Class	Property Definition	Certific'n Method
Performance	For each target configuration worst-case CPU time established.	Proof
	For each target configuration average elapsed time established.	Benchmarking
	For each target configuration size of the code segment and maximum heap space established.	Testing
	All heap space acquired is explicitly freed.	Inspection
	Screen updates take less than 5 ms. on all target configurations.	Benchmarking
Safety	Demonstrated freedom from numeric exceptions, or complete documentation of circumstances leading to such exceptions.	Inspection
	Handlers are present in every block within the component for all declared exceptions.	Static Analysis
	No anonymous exceptions raised.	Static Analysis
	No unexpected propagation of exceptions out of the part.	Static Analysis
	Exceptions and conditions under which they can be raised are fully documented.	Inspection
	Dynamically-allocated objects are destroyed when all references to them are destroyed.	Inspection
	Pointers never refer to released storage.	Inspection
	All finite loops shown to terminate.	Proof
	No internal concurrency, or internal concurrency deadlock free.	Inspection
	Verified guard semantics in all select statements.	Inspection
	No internal references to shared data objects.	Static Analysis
	Functionality independent of scheduling algorithm.	Inspection
	No execution conditions lead to tasking error.	Inspection
	Correct operation not dependent on specific priorities used.	Inspection
	All entry calls documented to facilitate demonstration of freedom from deadlock and priority inversion.	Inspection
Precision	Maximum relative error of floating-point quantities and circumstances under which this error occurs are documented.	Proof

Table 13: Behavioral Properties - Source Code

Property Class	Property Definition	Certific'n Method
Completeness	Selectable range checks implemented on all input parameters.	Inspection
	Parm. value or parm. interrelationship assumptions documented.	Inspection
	Non-generics tested to 90% branch & 100% functional coverage.	Testing

Table 13: Behavioral Properties - Source Code

Property Class	Property Definition	Certific'n Method
Reusability	Complies with the Reusability section (Chapter 8) of the *SPC Ada Style Guide*.	Inspection
Maintainability	Complies with relevant sections of Chapters 1-6 of the *SPC Ada Style Guide*.	Inspection
	Information hiding applied to hardware details.	Inspection
Portability	Complies with the Portability section (Chapter 7) of the *SPC Ada Style Guide*.	Inspection
	Part does not depend on parameter passing implementation semantics.	Inspection
	No dependencies on implementation-specific Ada features.	Static Analysis
Presentation	Complies with relevant sections of Chapters 1-6 of the *SPC Ada Style Guide*.	Inspection
	Includes corporate copyright notice.	Static Analysis
	Include machine-processable design documentation comment headers.	Static Analysis

Table 14: Structural Properties - Source Code

A.3.3 Certification Exploitation

Certification exploitation is the process by which the certification properties known to hold for the reusable components used to build a work product are used to demonstrate useful properties of the entire work product. For the sample domain studied here, the goal is to produce high-quality implementations in Ada for products relating to nautical navigation for ships.

Suppose a comprehensive control system for a new vessel is required and that a complete and accurate specification is available (see Section A.2). An implementation in Ada can be built quickly and efficiently using certified parts from the hypothetical library of source-code parts. The starting point is an appropriate canonical design and development continues with the selection of appropriate subsystem implementations and lower-level, leaf-node implementations.

The preparation of the implementation through reuse would proceed according to some standard development process. In this section, some example properties of the complete implementation that can be derived from the certification properties of the reusable

components are shown. Many useful properties both simple and complex can be established using certification technology. Other examples will no doubt suggest themselves to the reader.

Table 15 shows several work product properties and the certification properties from which they are derived. See also Chapter 6.

Exploitation of part properties will be done by the system builder. The steps to follow are:

- Establish system quality criteria. Focus especially on the qualities specified in the extended domain analysis.
- Determine which system quality criteria can be derived from the parts used. A system will almost never be comprised entirely of library parts, so quality inferences cannot be drawn about the entire system. However, inferences can be drawn about large parts of the system, for example:
  - The areas of the system using the certified parts will be free of memory leakage.
  - If processing is assumed to be sequential, then worst-case time to update the display screen is a function of the number of targets in the area, and can be calculated deterministically.
  - No non-termination problems in the final system will be the result of inadvertent infinite loops in these parts.
  - Checks for adherence to formatting standards will not need to be applied to these parts.

Domain Property	Component Property	Inference Mechanism
Conforms to real-time deadlines.	Performance.	Analytic
No inadvertent non-terminating loops.	Safety.	Immediate
Free of anomalous exception handling	Safety.	Analytic
Free of memory leakage.	Performance and safety.	Analytic
Free of invalid memory references.	Safety.	Analytic
Source format and programming techniques comply with standard.	Presentation and maintainability.	Immediate
System portable across all expected platforms.	Portability.	Immediate
Simple to add new I/O device.	Maintainability.	Immediate

Table 15: Exploiting Source-Code Certification

References

[And81] 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.

[DuK91] Dunn, M.F., and J.C. Knight, “Software Reuse In An Industrial Setting: A Case Study”, *Proceedings of the Thirteenth International Conference on Software Engineering*, Austin, TX, May 1991.

[Fag86] Fagan, M.E., “Advances in Software Inspections”, *IEEE Transactions On Software Engineering*, Vol. SE-12, No. 7, July 1986.

[Gar87] Gargaro, A. and T.L. Pappas, “Reusability Issues and Ada”, *IEEE Software*, July 1987.

[HeI88] Hekmatpour, S and D. Ince, *Software Prototyping, Formal Methods, and VDM*, Addison Wesley, 1988.

[Hen80] Heninger, K.L., “Specifying Software Requirements for Complex Systems: New Techniques and Their Application”, *IEEE Transactions On Software Engineering*, Vol. SE-6, No. 1, January 1980, pp. 2-13.

[How91] Howell, C., and D. Mularz, “Exception Handling in Large Ada Systems”, Technical Report, MITRE Corporation, McLean, VA, 1991.

[Jon86] Jones, C.B., *Systematic Software Development Using VDM*, Prentice Hall International, 1986.

[KnM91] Knight, J.C., and E.A. Myers, “Phased Inspections”, *ACM SIGSOFT*, Vol. 16, No. 3, July 1991, pp. 29-35.

[Lev87] 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*, Vol. 16, No. 4, April 1990.

[Luc85] Luckham, D.C. and F.W. von Henke, “An Overview of Anna, a Specification Language For Ada”, *IEEE Computer*, March, 1985.

[Mey88] Meyer, B., “EIFFEL: Reusability and Reliability”, in *Software Reuse: Emerging Technology*, Tracz, W., (editor), IEEE Computer Society Press, 1988.

[PaW85] Parnas, D.L. and D.M. Weiss. ‘‘Active Design Reviews: Principles and Practices’’, Proceedings of the *Eighth International Conference on Software Engineering*, London, England, August 1985.

[Pri90] Prieto-Diaz, R., “Domain Analysis: An Introduction”, *ACM SIGSOFT*, Vol. 15, No. 2, April 1990, pp. 47-54.

[Ric89] Rice, J. and H. Schwetman, “Interface Issues In A Software Parts Technology”, in *Software Reusability*, edited by Biggerstaff and Perlis, Addison Wesley, 1989.

[Rus87] 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.

[SPC89] Software Productivity Consortium, *Ada Quality And Style: Guidelines for Professional Programmers*, Van Nostrand Reinhold, 1989.

[SPC90] Software Productivity Consortium, “Introduction to Synthesis”, TR INTRO_SYNTHESIS-90019-N, June 1990.

[SPC92] Software Productivity Consortium, “Introducing Systematic Reuse To The Command and Control Systems Division of Rockwell International”, TR SPC-92020-N, May 1992.

[Spi89] Spivey, J.M., *The Z Notation: A Reference Manual*, Prentice Hall, 1989.

