On Local Certifiability of Software Components

Bruce W. Weide
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
weide@cis.ohio-state.edu

Joseph E. Hollingsworth
Department of Computer Science
Indiana University Southeast
New Albany, IN 47150
jholly@ius.indiana.edu

Abstract — Large software systems, like other large engineered systems, consist of components that are meant to be independent except at their interfaces. An important aspect of any large system is the need for *local certifiability*: to be able to establish properties of components out of the context of the larger system(s) in which they are embedded, and to be sure that any properties thus certified are certain to hold even when the components are composed with others to build larger ones. This is especially important for "black-box" reusable components, which need to be prequalified for inclusion in a component library. A good software engineering discipline must support local certifiability of important component properties, or life-cycle costs are inherently doomed to spiral out of control for larger and larger systems. No software engineering discipline in general use today can support local certifiability of most interesting properties. But local certifiability of many important properties — including, crucially, correctness with respect to an abstract specification — is possible in at least one practical discipline [Hollingsworth 92b].

(This paper is adapted, expanded, and updated from our position paper [Weide 92] for the 5th Annual Workshop on Software Reuse, held in Palo Alto, CA, in October 1992.)

1. Introduction

You might think it should be relatively straightforward to engineer software systems from off-the-shelf reusable components much like mechanical and electrical systems. Unfortunately this is not the case, because a piece of software is a mathematical object, not a physical one. Consider an Escher print depicting a plausible (on a local scale) but physically unrealizable (on a global scale) device. A mechanical engineer couldn't construct such a beast because physical laws prevent it. But nothing except extreme care and strict personal discipline can keep you from designing and building the software equivalent of such a device. So nearly every large software system is just that: It seems OK when viewed locally, i.e., on a component-by-component basis, but as a whole it probably does not work correctly under all circumstances due to unanticipated and arbitrarily long-range "weird interactions" among its parts.

Every large system (software or physical) consists of components that are meant to be independent except at their interfaces because a "large" system is simply too big for one person to comprehend as a whole, especially when you need to make changes in a timely fashion. If some of these components are *reusable* off-the-shelf, then presumably they are subject to quality control procedures by which the component provider *certifies* that they have certain properties. The purpose of certification is to give prospective clients critical assurance that the component is truly suitable for use in the client's circumstances. The weird interaction problem arises because the certification process must be performed outside the context of any particular client system; a reusable component is meant to be incorporated into a wide variety of client software which is perforce unknown at the time of certification. The intrinsic requirement for *local* (also called *modular*, or *independent*) *certification* of reusable component quality raises a challenging problem, a potential obstacle that threatens development of all large component-based systems: You must be able to ensure that the creation-time certification of an off-the-shelf component cannot be broken by any client system that uses the component "properly."

As software engineering is a comparatively immature discipline at best, some fundamental questions about local certifiability — which are at the heart of what an engineering discipline for software should look like — remain unanswered or even unacknowledged.

- What properties should be locally certified for software components?
- What is the full compass of ways in which a client program might use a component that might invalidate that component's certification?
- How can you control or eliminate each such threat so that locally certified properties of software components will result in composite systems having the desired properties?

If satisfactory answers to these questions about local certifiability and the problems they raise can be found, then the software engineering community will have a good handle on how to deal with some of the major technical problems of engineering of large component-built software systems. Attention to local certifiability during design and development will be especially beneficial in reducing subsequent maintenance problems, which clearly are the most costly part of the software life-cycle, especially for object-oriented designs [Wilde 93]. If satisfactory answers cannot be found, then the software engineering field will find itself in serious trouble. For while management aspects of software engineering are undeniably important, no clever scheme for deploying people, incentive system for encouraging software reuse, or other non-technical gimmickry will ultimately be able to hide the fact that the emperor has no clothes.

The primary contribution of this paper is an explanation — in the context of reuse — of the importance of answering these questions. While there has been some progress (e.g., [Hollingsworth 92b]) the problems in many respects remain open [Weide 92].

The paper is organized as follows. Section 2 illustrates the fundamental problem of locality of reasoning about program behavior, which faces anyone who proposes a software engineering discipline that is purported to work for large systems. Section 3 explains why you should demand local certifiability of any sensible software engineering discipline. Section 4 reviews related work, and Section 5 contains our conclusions and a brief discussion of one of the many open problems associated with local certifiability.

2. Local Reasoning About Program Behavior

A *software engineering discipline* is a set of software design and development principles that (some expert claims) you should observe if you want to build high-quality software systems quickly and effectively. These principles might in turn entail general rules and guidelines and/or very specific and detailed ones, even at the level of coding conventions.

There is serious doubt within the software engineering community about whether any software engineering discipline can predictably deal with the technical problems of engineering systems that are too large for one person to fully understand [CSTB 90, McCarthy 93, Neumann 93]. Why? It is just amazingly hard to arrange a large software system so that you have the ability to reason locally about its behavior. A simple example of one "devil in the details" helps to illustrate what can happen to your local reasoning ability if you are not careful in designing and reusing software components. Consider the following Pascal-like procedure and its functional specification:

```
procedure Increment (var x: integer; var y: integer);
    ensures     x = #x + 1  and  y = #y + 1
```

The meaning of the specification is that, upon return from a call "Increment (x, y)," the value of x will be one greater than it was before the call, and similarly for y.

Suppose you have a client program in which the Increment procedure is treated as a (black-box) component. The client program might contain the following code:

```
var a, b: integer;
...
a := 10;
b := 20;
Increment (a, b);
```

In trying to understand or explain how this client code works, you might create the following explanation (which could be formalized): "The first assignment statement causes the value of a to be 10, and the second causes b to be 20. Then the Increment procedure causes a to be one greater than it was before the call, and similarly for b. The only values of a and b that can make this true are a = 11 and b = 21, so this procedure call causes the value of a to be 11 and b to be 21." Indeed this is a valid story if the variables a and b are not otherwise connected (e.g., by being EQUIVALENCEd if the programming language were Fortran), and if the implementation of Increment meets its specification. It is valid because it explains how the code *actually will work* if executed; the reasoning process is *sound*.

But a different client program might contain slightly different code:

```
var a: integer;
...
a := 10;
Increment (a, a);
```

In trying to understand or explain how this client code works, you might create the following explanation: "The assignment statement causes the value of a to be 10. Then the Increment procedure causes a to be one greater than it was before the call, and similarly for a. The only value of a that can make this true is a = 11, so this procedure call causes the value of a to be 11." This is *not* valid because the actual value of a (for most modern languages and for the most obvious implementation of Increment) actually will be 12.

What has happened? The problem is that you can show that the Increment procedure is correct out of the context of a particular client program *only* under certain conditions on the ways in which the client program uses it. In this case these conditions include the fact that the client program may not call Increment with the same variable appearing twice in the argument list. You can locally certify the Increment procedure to act as its functional specification says only if that specification includes some context constraints. But these constraints generally are not listed (either explicitly or implicitly) with current software engineering practices.[1] It is essential that you should state them clearly and unambiguously in order that client programmers be able to reason in a sound manner about the behavior of component-built software (and in order that a formal, modular verification system for such programs should be sound [Ernst 91]).

Paradoxically, while the above claim about the need for local certifiability may seem to be only common sense, to our knowledge you cannot locally certify important properties such as correctness in any practical extant software engineering discipline or using any current software component library, except our own [Hollingsworth 92b].

3. Inherent Requirements of a Good Software Engineering Discipline

This section presents the general technical basis for the claim that no software engineering discipline, even though it might "work" for small projects, can scale up to large ones unless it includes a core of techniques for locally certifying certain properties of a system's components. Two main issues arise: the properties to be certified, and the practicality of certifying them.

3.1. Local Certifiability

A property is said to be *locally certified* whenever:

(a) the property is established for the component in isolation — outside the context of any particular client program; and

(b) the property is certain to hold for any instance of the component that occurs in any client program, so long as the client program uses the component in compliance with certain conditions, including context constraints.

---

1 Notice that you could not phrase this context constraint as a precondition of the procedure, because it depends not just on the abstract values of the actuals. To try to state it as a precondition would betray knowledge of the representations of variables.

If a property is locally certifiable, then the amount of information you need to establish that the property holds for the component depends only upon the component itself. It is not related to the size or structure of client systems in which you might embed instances of that component.

3.2. Properties That Should Be Certifiable

What general properties should hold for high-quality software components? Without fixing the kind of component in question it is difficult to identify many properties that every component should have, except the following.

- *Simplicity/Comprehensibility* — It is crucial to those who need to use components (end users in the case of application systems, other software designers and developers in the case of embedded piece-parts) that the functions of individual components be understandable. Sometimes a software component is, of necessity, relatively complex because of the nature of the task it must accomplish. What you should insist upon is that the understandability of its behavioral description be commensurate with the intrinsic complexity of its task.

- *Reusability* — Significant leverage is available for reducing total life-cycle cost and for improving overall product quality if constituent components are reusable. That is, you should be able to use many components of a system in other systems, as-is or with only parametric adaptation. Critically, these other systems include (but of course are not limited to) extensions, new versions, and variations of existing systems, which might result from adaptive and perfective maintenance activities.

The above component properties are subjective. Whether they would hold for a particular component might well be debatable. However, for particular component types — in this case, modules or packages or classes — you can identify some specific technically precise and objectively determinable component properties that also should be satisfied. Among them are:

- *Correctness* — A high-quality component should be demonstrably correct, i.e., it should correctly implement its intended (abstractly specified) behavior.

- *Efficiency* — A high-quality component should be time- and/or space-efficient in the sense that it should not be dominated in all dimensions of performance by another component that does the same job.

- *Composability* — A high-quality component should be readily composable with other high-quality components designed according to the software engineering discipline. This is necessary in order to define and implement "larger" components.

It is still a research matter to define properties such as efficiency [Sitaraman 92] and composability [Edwards 93] more precisely, to identify other objective properties that should be locally certifiable, and to determine whether and how local certification is to be achieved. But certainly correctness is a key property, and we use it below to help illustrate why local certifiability is essential for dealing with large systems.

3.3. Tractable Reasoning About Program Properties

Local certification is essential for effective reasoning about the run-time behavior of large systems because you need it for *tractable* reasoning. This is due to the potentially combinatorial explosion of work that arises if you treat large systems only as massive aggregates of primitive pieces. Suppose the components in question are subprograms, and consider a case where each of A and B calls both C and D; C calls E and F; and so on. Now if you cannot cleanly factor the proof that C is correct from the usage of C in the coding of A and B, then to argue for the correctness of A and

B you must expand the code for C in-line in A and B and deal with the resulting code dilation and state space expansion. From there the problem only gets worse, since C also calls E and F, and so on down through the hierarchy of components, and at each level you must expand the calls in-line in the same way.

Unfortunately, this is how almost all existing formal proof-of-correctness systems *must* work[2] if the proof rules are supposed to encompass any legal program in a typical real language, e.g., Ada [Guaspari 90]. The problem is that use of encapsulation mechanisms such as separation of specification from implementation, private types, etc., is not sufficient by itself to ensure that you can reason modularly about programs. This is well-known in principle, but "common knowledge" has not worked its way into software engineering practice. Probably more important is the fact that knowing that you should be able to reason modularly says nothing about how to do it. Many widely-practiced programming techniques — ones taught in contemporary programming and software engineering textbooks and held up as good practice — are the antithesis of those needed to support modular reasoning and local certification [Harms 91, Hollingsworth 91a, Hollingsworth 92a, Hollingsworth 92b, Weide 93].

The total amount of code that you must consider in an argument that treats a component-built program as a monolith grows combinatorially as you progress up the hierarchy to larger and larger components. Despite lip-service to "abstraction" and "information hiding," there really are no firewalls that protect against having to expand each component into its immediate constituents, then on and on down to the primitive components. It is little wonder that traditional formal program verification technology has had a reputation for being unable to deal with programs of realistic size. Because informal reasoning techniques (e.g., simulated execution) face precisely the same difficulties, it also is little wonder that the costs of maintenance and system integration tests are so prominent in the software life-cycle. Without taking great care in design and coding, you simply *cannot assume* that the effects of program changes are localized; even if changes seem to be (and actually are) OK in isolation they may introduce unanticipated side-effects in the far reaches of the system.

On the other hand, you *can* locally certify correctness provided firewalls do exist above and below each component [Cook 78, Krone 88, Ernst 91]. For instance, in the above example if you can show that C and D are correct outside the particular context of A or B, then you can make each correctness argument once and for all. You can argue that A is correct because (1) it uses C and D properly and in accordance with the assumptions needed to establish their correctness independently of A, and (2) you already know that C and D are correct under those assumptions. This means that the total complexity of establishing correctness (or some other locally certifiable property) grows linearly with the number of its direct and indirect constituents, i.e., linearly with the total amount of code for the component.

Furthermore, if there is any component reuse (e.g., A and B both use C) then you can amortize the cost of establishing a property for a component over the many times it is used. In the limit, the cost of certifying a property of a new component becomes proportional only to the amount of new code added by that component. The cost is independent of how high a component is in the implementation hierarchy, so certifying "large" components (those higher in the hierarchy) costs you no more per statement than certifying "small" ones.

Because of the language definitions, you cannot modularly verify every syntactically legal program in, say, Ada or C++, even if they might be correct [Weide 93]. The proof rules in [Krone 88] apply to a version of our language, RESOLVE, in which practices that thwart modular verification are prohibited by the language itself. The key to making such a proof system relate to programming practice in "real" languages like Ada or C++ is to restrict programs to a certain style in which dangerous constructs are prohibited, not by the language — it is too late for that — but by a well-defined software engineering discipline. Characterizing such a style for Ada is one of the main

2 A superficial analysis of the results of [Cook 78] suggests otherwise; but see Section 4 and [Weide 93].

contributions of [Hollingsworth 92b]. How the RESOLVE/Ada Discipline lets you achieve local certifiability cannot be summarized acceptably in a short paper such as this one, but [Hollingsworth 92b] is readily available via the Internet (see the bibliography) if you are interested in details and numerous examples.

The bottom line is this: *Local certifiability is a necessary condition for tractable certifiability of large components and systems.* If following a particular software engineering discipline results in the important properties of high-quality components being locally certifiable, then it is at least possible that you can feasibly certify those properties even for large components under that discipline. On the other hand, if following a particular software engineering discipline results in important properties of high-quality components not being locally certifiable, then you definitely cannot feasibly certify those properties for large components under that discipline. You should, therefore, reject any such discipline as the basis for engineering large software systems.

3.4. Some Consequences of Requiring Local Certifiability

Local certifiability has many implications for the issues that a software engineering discipline must address. For instance, part of the RESOLVE/Ada Discipline is intended to guarantee that the abstract concept which a component implements, and the expectations of the context within which that component operates [Edwards 90, Tracz 90b], are fully defined, so that you can certify important component properties once and for all prior to entering the component into a library. A related part of the discipline is intended to ensure that no client program which uses the component in a way that meets this contract can possibly invalidate its certification [Hollingsworth 92b].

Local certification of correctness illustrates the central issues in being able to reason — either formally or informally — about the behavior of individual components in the context of client programs. One facet of correctness that usually is not considered in other approaches, storage management, raises an interesting sub-property that should be locally certifiable: Every component should introduce no storage leak. Here you can show by an induction argument that no component designed, implemented, and used under the RESOLVE/Ada Discipline can have a storage leak [Hollingsworth 92b]. Sadly (as you are painfully aware as a sophisticated user of any other component library for, e.g., Ada or C++), it is essentially impossible to establish this property *at all* for a non-trivial program that is developed following accepted contemporary software engineering practices. Program analysis tools can help, but neither you nor a tool can demonstrate it by purely local means.

One seeming consequence of the thesis that local certifiability of components is essential for design and maintenance of high-quality large systems — and that no current generally-practiced software engineering discipline employs this important fact — is that no high-quality large software systems can exist today. This is not necessarily the case. Large systems that really work still might exist in principle because mere difficulties do not imply impossibility. Heroic and costly efforts of software engineers working as individuals or as teams, plus some luck [Bernstein 92], still might result in high-quality systems even though the developers follow practices that make the job hard.

Whether any such systems actually exist is surely open to question, since even if they do, there would be no practical way to *certify* that they actually were of high quality. With current practices, whether a system behaves as specified, for example, is unknown until you have observed all aspects of the desired behavior and all possible conditions in execution. Clearly, for a large system there is no feasible way to do this. For life-critical embedded systems the infeasibility of demonstrating correctness in advance is crucial. This was a key point of attack by Parnas and by others on anticipated problems with software for the Strategic Defense Initiative [Parnas 85]. Furthermore, adaptive and perfective maintenance problems for systems that do not support local certifiability remain incredibly difficult and costly even if you can make those systems work as they should in the current release.

4. Related Work

There are many recent books and papers promoting (partial) disciplines for component-built software, e.g., [Liskov 86, Booch 87, Meyer 88, Musser 89, Weide 91, Batory 92, Meyer 92]. There are papers on certification of various component properties (e.g., [Knight 91]), and questions about the very feasibility of dependable large systems [Neumann 93]. But there is surprisingly little work that pinpoints local certifiability as an essential aspect of a scalable software engineering discipline or that explicitly discusses the technical problems associated with it.

The importance and difficulty of modular reasoning about program behavior is discussed in the context of data-flow programming by [Dennis 72], who offers a form of programming-without-side-effects as a possible way to achieve it. Adopting a purely functional programming style is clearly one way of eliminating certain undesirable interactions among components. Unfortunately, this approach is not really viable given current (or likely future) software engineering practice, which largely uses object-based or object-oriented imperative programming techniques to achieve acceptable levels of efficiency in "industrial-strength" software.

Moreover, adopting a purely functional programming style is not sufficient to achieve local certifiability of some important system properties. For example, consider a real-time embedded application wherein the use of a garbage collector is inappropriate due to its unpredictable effects on system performance. How can you be sure that the storage management techniques used in the individual components do not interact to introduce a storage leak that will eventually crash the system? For example, the Booch Ada components [Booch 87] are subject to this problem, and simply adopting a functional style of programming would not fix it [Hollingsworth 91].

The program verification literature of the late 1970's addresses sound and relatively complete *modular* proof-of-correctness systems [Cook 78] which are defined for very simple, but still interesting, Pascal-like languages. The most exciting part of Cook's language, for example, is that it has procedures (although no recursive ones) with parameters like those Ada (later) termed "in" and "in out" mode. But there are some restrictions:

- All variables are scalars of the same type, e.g., integer. There are neither type constructors such as arrays and pointers, nor user-defined types.
- In any call, all arguments whose corresponding formals have mode "in out" must be distinct variables.
- In any call, no argument whose corresponding formal has mode "in out" may be visible within the called procedure's body by virtue of being global to it.

Ada and C++ and similar "real" languages that you might use to develop component-built software systems are similar to Cook's language in many ways, but also differ from it in important respects. They have recursive procedures; many scalar types; record, array, and pointer type constructors; user-defined abstract types; and generic modules. And they do not ask the compiler to enforce the above restrictions on procedure calls. So although many software engineers with a shallow understanding of Cook's work would like to believe otherwise (i.e., that local certifiability of correctness is a solved technical problem), the existence of a sound and relatively complete modular proof system for Cook's simple language and slight variants of it certainly does not imply that you can reason modularly about arbitrary Ada or C++ programs [Ernst 91, Weide 93].

There has been some work in the object-oriented programming community on making class "correctness" tantamount to system "correctness." This work seems to be limited to one aspect of type correctness, i.e., making subtypes and supertypes behave "properly" in clients in all relevant respects. The scenario described in [Leavens 90] includes an applicative language where side-effects are not a problem and therefore, by the authors' admission, bears little relationship to the way object-oriented programming is practiced. This paper also does not deal with other desirable system properties that you might like to certify locally. The work of [Weber 92] is more practical in that it describes techniques for avoiding unwanted component coupling in a real object-oriented programming setting. But it, too, deals only with type correctness and not with other crucial properties of software such as overall behavioral correctness, performance, etc.

Some of the challenges of locally certifying correctness with respect to an abstract specification are discussed in [Ernst 91] for Ada generics. This work points out the central role of formal methods in identifying threats to local certification of correctness and the inescapable relationship between program verification and modular informal reasoning about program behavior. It also explains why a verification system that involves simple textual expansion of generics and procedure calls into their bodies (e.g., the approach described in [Guaspari 90]) cannot form the basis for modular reasoning about large programs.

We believe [Hollingsworth 92b] contains to date the most comprehensive, technically rigorous discussion of scalability of software engineering disciplines. It identifies several important properties that should be locally certifiable: correctness with respect to an abstract specification (including correctness of storage management, which is usually left unspecified), composability, reusability, and comprehensibility. It also contains, to our knowledge, the only published and easily accessible discipline that can support local certifiability of these properties and at the same time is presented in the context of a practical programming language, Ada.

5. Conclusion

If you need to design, develop, and maintain large software systems, you should demand that the software engineering discipline to be followed must support local certifiability of important properties of the components of that system. If you can't locally certify properties of system components, then life-cycle costs are doomed to grow not linearly with system size (as you might hope in a more optimistic moment) but exponentially (as you should expect by now from dismal experience).

Is there any hope that software engineering disciplines can support local certifiability? The foundation for such a discipline for sequential programs written in Ada already exists [Hollingsworth 92b]. But this is just the start of the effort needed to make local certifiability a practical reality. For some properties — particularly correctness with respect to an abstract specification, and reuse-related ones such as composability — there is a firm foundation for achieving local certifiability [Krone 88, Ernst 91, Hollingsworth 92b]. For some other important properties there is little previous work to build on.

As an example, consider the problem of locally certifying that a component implementation meets certain performance specifications for its execution time. Achieving local certification of this kind of property requires development of a calculus for composing summary performance information about components into comparable information about a client program, which is still an open problem [Sitaraman 92]. It turns out that you can easily establish certain negative results about local certifiability of performance, because many classical techniques for analysis of algorithms simply do not scale. For example, in average case analysis, just knowing the expected execution time for a procedure as a function of its inputs' values is not sufficient to allow you to determine the expected execution time of a caller — even if you know all input probability distributions and even if the calling program contains nothing but a call to the already-analyzed procedure. The problem is that the caller may have a precondition that the called procedure does not have; Bayes' theorem applies, and one of the conditional probabilities is unknown.

Modular composition of performance information seems to require "pointwise" performance formulæ for all components. Unfortunately, the mathematical models that provide an adequate basis for functional specifications of behavior usually are too abstract to provide a basis for local certification of compliance with performance specifications — or even to *express* performance specifications adequately. So it is a challenging and as yet unsolved problem just to identify the sort of performance attributes that would provide a viable framework for local certifiability, let alone to develop the techniques for certification. We postulate that methods similar to those we have applied to verify functionality will form the basis for verifying performance. But significant aspects of the approach, and many details, remain to be worked out [Sitaraman 92].

Acknowledgment

Bill Ogden, Stu Zweben, Steve Edwards, Wayne Heym, Murali Sitaraman, Tim Long, Neelam Soundararajan, B. Chandrasekaran, and Dean Allemang have provided helpful insights and/or feedback on the ideas presented here. We also gratefully acknowledge the support of the National Science Foundation through grants CCR-9111892 and CCR-9311702; and the Advanced Research Projects Agency under ARPA contract number F30602-93-C-0243, monitored by the USAF Material Command, Rome Laboratories, ARPA order number A714.

Bibliography

[Batory 92] Batory, D., and O'Malley, S. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. on Softw. Eng. and Meth. 1*, (Oct. 1992), 355-398.

[Bernstein 92] Bernstein, L. Software discipline and the war of 1812. *Software Eng. Notes 17*, 4 (Oct. 1992), 18.

[Booch 87] Booch, G. *Software Components with Ada*. Benjamin/Cummings, Menlo Park, CA, 1987.

[Cook 78] Cook, S.A. Soundness and completeness of an axiom system for program verification, *SIAM J. Comp. 7*, 1 (Feb. 1978), 70-90.

[CSTB 90] Computer Science and Technology Board. Scaling up: a research agenda for software engineering. *Comm. ACM 33*, 3 (Mar. 1990), 281-293.

[Dennis 72] Dennis, J.B. Modularity. In *Software Engineering: An Advanced Course*, G. Goos and J. Hartmanis, eds., Springer-Verlag, New York, 1972.

[Edwards 90] Edwards, S. The 3C model of reusable software components. *Proc. 3rd Ann.Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center, June 1990.

[Edwards 93] Edwards, S.H. Common interface models for reusable software. *Intl. J. Software Eng. and Knowledge Eng. 3,* 2 (June 1993), 193-206.

[Ernst 91] Ernst, G.W., Hookway, R.J., Menegay, J.A., and Ogden, W.F. Modular verification of Ada generics. *Comp. Lang. 16*, 3/4 (1991), 259-280.

[Guaspari 90] Guaspari, D., Marceau, C., and Polak, W. Formal verification of Ada programs. *IEEE Trans. on Software Eng. 16*, 9 (Sept. 1990), 1058-1075.

[Harms 91] Harms, D.E., and Weide, B.W. Copying and swapping: influences on the design of reusable software components. *IEEE Trans. on Software Eng. 17*, 5 (May 1991), 424-435.

[Hollingsworth 91] Hollingsworth, J.E., Weide, B.W., and Zweben, S.H. Abstraction leaks in Ada (extended abstract). *Proc. 14th Minnowbrook Workshop on Software Eng.*, Syracuse University, July 1991.

[Hollingsworth 92a] Hollingsworth, J.E., and Weide, B.W. Engineering 'unbounded' reusable Ada generics. *Proc 10th Annual Natl. Conf. on Ada Technology*, Arlington, VA, Feb. 1992, 82-97.

[Hollingsworth 92b] Hollingsworth, J.E. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada*. Ph.D. dissertation, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH, Aug. 1992; available by anonymous FTP from host "ftp.cis.ohio-state.edu" in "pub/tech-report/1993/TR01-DIR/*".

[Knight 91] Knight, J. Issues in the certification of reusable parts. *Proc. 4th Ann.Workshop on Software Reuse*, Herndon, VA, Nov. 1991.

[Krone 88] Krone, J. *The Role of Verification in Software Reusability*. Ph.D. dissertation, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH, Aug. 1988.

[Leavens 90] Leavens, G.T., and Weihl, W.E. Reasoning about object-oriented programs that use subtypes. *Proc. OOPSLA '90/SIGPLAN Notices 25*, 10 (Oct. 1990), 212-223.

[Liskov 86] Liskov, B., and Guttag, J. *Abstraction and Specification in Program Development*. McGraw-Hill, New York, 1986.

[McCarthy 93] McCarthy, J. Merging CS and CE disciplines is not a good idea. *Comp. Res. News 5*, 1 (Jan 1993), 2-3.

[Meyer 88] Meyer, B. *Object-oriented Software Construction*. Prentice-Hall, New York, 1988.

[Meyer 92] Meyer, B. Applying "design by contract". *Computer 25*, 10 (Oct. 1992) 40-51.

[Musser 89] Musser, D.R., and Stepanov, A.A. *The Ada Generic Library: Linear List Processing Packages*. Springer-Verlag, New York, 1989.

[Neumann 93] Neumann, P.G. Are dependable systems feasible? *Comm. ACM 36*, 2 (Feb. 1993), 146.

[Parnas 85] Parnas, D.L. Software aspects of strategic defense systems. *Comm. ACM 28*, 12 (Dec. 1985), 1326-1335.

[Sitaraman 92] Sitaraman, M. Performance-parameterized reusable software components. *Intl. J. Software Eng. and Knowledge Eng. 2*, 4 (Dec. 1992), 567-587.

[Tracz 90] Tracz, W. The three cons of software reuse. *Proc. 3rd Ann. Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center, June 1990.

[Weber 92] Weber, F. Getting class correctness and system correctness equivalent: how to get covariance right. In *Proc. TOOLS USA '92*, R. Ege, M. Singh, and B. Meyer, eds., Prentice-Hall, 1992.

[Weide 91] Weide, B.W., Ogden, W.F., and Zweben, S.H. Reusable software components. In *Advances in Computers*, vol. 33, M.C. Yovits, ed., Academic Press, 1991, 1-65.

[Weide 92] Weide, B.W., and Hollingsworth, J.E. Scalability of reuse technology to large systems requires local certifiability. *Proc. 5th Ann.Workshop on Software Reuse*, Palo Alto, CA, Oct. 1992.

[Weide 93] Weide, B.W., Heym, W.D., and Ogden, W.F. Procedure calls and local certifiability of component correctness. *Proc. 6th Ann.Workshop on Software Reuse*, Owego, NY, Nov. 1993.

[Wilde 93] Wilde, N., Matthews, P., and Huitt, R. Maintaining object-oriented software. *IEEE Software 10*, 1 (Jan 1993), 75-80.