Software Component Certification: 10 Useful Distinctions

Kurt C. Wallnau

September 2004

CMU/SEI-2004-TN-031

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Software Component Certification: 10 Useful Distinctions

CMU/SEI-2004-TN-031

Kurt C. Wallnau

September 2004

Predictable Assembly from Certifiable Components Initiative

Unlimited distribution subject to the copyright.

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2004 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provide the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Abstract . . . . **iii**

1 Introduction **1**

2 Definition of Certification **3**

3 Ten Distinctions **4**
- 3.1 Truth Vs. Knowledge 4
- 3.2 Knowledge Vs. Trust 5
- 3.3 Normative Vs. Descriptive 5
- 3.4 Objective Vs. Subjective 6
- 3.5 Product Vs. Process 7
- 3.6 Local Vs. Contextual 8
- 3.7 Determinate Vs. Predictive 8
- 3.8 Formal Vs. Empirical 10
- 3.9 Procedural Vs. Mechanical 11
- 3.10 Standard Vs. De Facto 12

4 Related Work **14**

5 Conclusions **16**

Bibliography **17**

Abstract

Using software components to develop mission-critical systems poses a number of technical, organizational, and economic challenges. One persistent and largely unaddressed challenge is how the consumers of software components—that is, the developers of mission-critical systems—can obtain a meaningful level of *trust* in the runtime behavior of software components. The most frequently cited concerns are centered on issues of security; for example, trust that a component does not contain malicious code or exhibit vulnerabilities that can be exploited by malicious code. There are, however, other concerns about software component behavior that can be just as important. For example, in an embedded weapon system, it may be crucial to trust that a component will always execute a function within a particular time bound or never introduce unbounded priority inversion.

Certification is a practical, proven means of establishing trust in various sorts of things in other disciplines and is, therefore, a natural contender for developing trust in software components. This technical note does not propose a particular certification regimen for components. Rather, it introduces a series of 10 distinctions that can help in understanding different aspects of certification in the context of software components.

1 Introduction

The Predictable Assembly from Certifiable Components (PACC) Initiative at the Carnegie Mellon® [1] Software Engineering Institute (SEI) is investigating how the behavior of systems comprising assemblies of software components can be reliably inferred, or predicted, from the trusted properties of the software components themselves.

There is more than a little subtlety in the phrasing of this research agenda. In particular, we use the phrase "trusted properties of software components" rather than "trusted software components" to indicate that we may trust some properties of a component but not others. We also place the topic of component-level trust within the broader context of system-level (or, more properly, assembly-level) predictability; doing so indicates the intended subservience of trusted components to an encompassing activity involving engineering analysis.

Another subtlety is that the title of our initiative uses the adjective "certifiable" rather than "certified." Our interest in trusted components lies in how they are used to make reliable predictions about assembly behavior. Our interest in certification lies in how it can be used to engender trust. Certification is, therefore, at least one step removed from our fundamental agenda—reliable prediction. On the other hand, the conditions that make a component *certifiable* are of direct importance to our agenda, since trustworthy predictions are impossible without trustworthy component properties.

Hermeneutics aside, it is apparent that our perspective (that of the PACC Initiative) on which aspects of software components are worth trusting—namely, the properties of components that are parameters to analytic theories of assembly behavior—is not wholly conventional. This perspective may unintentionally put us at odds with the more conventional, and familiar, notions of software certification used in the software industry. The resulting terminological confusion may obscure the ideas underlying the PACC premise regarding trusted components—ideas whose validity does not hinge on certification per se.

An important question for the PACC Initiative, then, is: Can the conventional understanding of software certification be adopted, or perhaps coerced, in the service of predictable assembly? The question presupposes that there is, in fact, a well-developed conventional notion of software certification. There are, in fact, many notions of software certification currently implemented in different ways.

---

[1] Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

For the term *certifiable components* to have technical meaning in PACC, it will have to be situated in a much clearer model of software certification than is currently available. This technical note is our first step toward exploring the software certification landscape by presenting a space of alternative concepts rather than a compendium of specific approaches. The device used to describe this space is a series of 10 distinctions, each expressing 2 concepts that may be similar but not identical, partially conflicting, or mutually exclusive to one another.

These distinctions do not completely characterize the conceptual space of software certification. They are, however, representative of what we have encountered in the literature and at various fora on trusted software and software certification. The distinctions are not used in this technical note to classify different approaches, but rather to expose concerns that underlie the topic as a whole.

As background, Chapter 2 defines the term *certification*. Chapter 3 briefly discusses the 10 distinctions; our goal in that chapter is to help the reader develop general intuition rather than to present an argument in detail. A representative but by no means comprehensive review of related work is provided in Chapter 4. Finally, Chapter 5 presents a few tentative conclusions about the distinctions discussed in this technical note.

2 Definition of Certification

We should not, at the outset, prejudice how certification may be applied to software and, in particular, to software components. The dictionary definition below gives us insight into the conventional meaning of certification and exposes what may be a source of many hidden assumptions about how it does, or should, relate to software components:

> *1. to attest as certain; give reliable information of; confirm... 2. to testify to or vouch for in writing...3. to guarantee; endorse reliably; to certify a document with an official seal...*[MWU 96]

This definition is a marvel of efficiency, yet is suggestive in several ways:

- *to attest as certain* suggests an authoritative testimonial or “standing by” a statement
- (to give) *reliable information* suggests an objective quality
- (to) *confirm* suggests a corroboration of other information
- (to) *vouch for in writing* suggests evidence and legal implications
- *to guarantee* suggests contractual obligations and remedies
- (to use) *an official seal* suggests a designated authority

More specialized definitions are available; for example, legal and medical definitions of the term have emerged over time. These specializations are consistent with the general meaning but are particular in their application. For our purpose, the general definition is a sufficient starting point.

3 Ten Distinctions

3.1 Truth Vs. Knowledge

The full title of Mary Shaw's position paper nearly says it all: "*Truth Vs. Knowledge: The Difference Between What a Component Does and What We Know It Does*" [Shaw 96].

On the one hand, this distinction expresses nothing beyond an age-old epistemological question about the basis of human knowledge and its relation to reality. On the other hand, it does posit an axiom of any software certification regimen: all knowledge about the properties of a software component will be *provisional*. When this axiom is accepted, any robust treatment of certification must account for this provisionality and possibly make it explicit.

We are accustomed to some level of provisionality for software. For example, performance benchmarks of, say, operating systems involve experimental measurement and, hence, introduce potential sources of systematic and random error. In fact, any properties of software components that are measured (for example, through functional testing) will introduce sources of experimental error. In all cases, we would be surprised by, and should be skeptical of, any purported measures if they were not also accompanied by some sort of statistical qualification.

We are less accustomed to the idea that provisionality applies equally to properties of software components established by purely formal means, that is, through means of logic and proof. What does it mean for a *proof* of total correctness in Hoare logic to be provisional? Just this: The proof in question applies to a specification that will undergo a sequence of transformations (e.g., from source to intermediate to assembly to machine instructions), each of which is assumed (usually with no formal or empirical basis) to be correct.

Although this last point is a well-known bugaboo noted by critics of formal methods, it is not our main concern. Instead, we are concerned with understanding the provisionality of formal assertions, especially since a recent trend in formal methods is the use of fully automated verification technology, such as temporal logic model checking. It is reasonable to assume that software implementing complex verification procedures is also be susceptible to bugs.[2]

---

[2] You might find it helpful to examine the release history of the well-known Spin model checker provided at http://spinroot.com.

3.2 Knowledge Vs. Trust

This distinction can also be linked to basic epistemological questions because a great deal of what we claim to know about the world is acquired not through direct experience, but rather through description.[3] This sense of the distinction is not what is emphasized in this technical note.

Meyer, Mingins, and Schmidt express the intended meaning of this distinction in their article titled "*Trusted Components for the Software Industry*" [Meyer 98]. In particular, they argue (correctly in my judgement) that trust is a *social* phenomena. One (revealing) supporting argument is that even the acceptance ("trust") of mathematical proof is influenced by social factors such as implicit agreement on what is an acceptable level of rigor in proofs, the perceived reputation of the author of the proof, and so forth.

While this social conditioning has been noted elsewhere [Demillo 77], Meyer, Mingins, and Schmidt draw software-specific implications related to the question of certification. Specifically, they conclude that a broad and varied array of evidence is required to establish a social foundation for trust. From that conclusion, we can infer that certification, in the sense of authoritative labeling, may sometimes be insufficient in itself to establish trust; conversely, it might be necessary in certain communities accustomed to the stricture of certification; for example, the nuclear power industry.

Note that trust is not absolute—absolute trust is better denoted as *faith*. The provisional nature of trust is also noted by Meyer, Mingins, and Schmidt. As a result, we can also conclude, as did Mary Shaw, that a robust treatment of certification will address the inherent boundedness of the certainty of empirical, and even formal, evidence.

3.3 Normative Vs. Descriptive

Normative certification attests that software conforms to, or satisfies, some established *norm*. Descriptive certification merely "describes" some aspect of the software.

Examples of normative certification are legion and include the following: conformance to interface standards, for example, POSIX;[4] test suites, for example, the Ada Compiler Validation Capability (ACVC) benchmarks;[5] or quality-attribute-specific norms, for example, for security, the storied “Orange Book,”[6] or, for safety, a menagerie of standards, including UL1998, IEEE 1228, IEC 60880, IEC 60950, and IEC 61508.

---

[3] A classic example of this is: I know the Taj Mahal exists, and yet I have never seen it (except, again, provisionally through photographs). Is this knowledge trustworthy? How do you decide?

[4] For more information, go to http://www.opengroup.org/testing/fips/.

[5] For more information, go to http://www.iste.uni-stuttgart.de/ps/AdaBasis/pal_1195/ada/ajpo/compilers/95val/val-proc.txt.

Examples of descriptive certification are, by comparison, rare. Performance benchmarks have been defined for particular kinds of software; for example, middleware and database technology.[7] Voas has long advocated the descriptive certification of reliability and has proposed several mechanisms for acquiring such measures [Voas 98], [Voas 99], [Voas 00]. His proposals emphasize the descriptive rather than normative aspects of certification because no threshold (norm) that components must satisfy has been defined.

Normative certification is the rule in software; descriptive certification is nearly nonexistent. This situation is no doubt due, in part, to the dearth of accepted measures for software—with the exception of “source line of code” and “function point”—and even these are disputed. In contrast, electronic (hardware) components have numerous descriptive measures such as whetstones, dhrystones, and clock speed for processors; mean time to failure; access time and seek time for disk drives; and so forth. Generally, these measures are used to provide manufacturer-rated specifications.

In general, we can conclude that certification tends toward normative interpretation, partly because the notions of authority and the legal and economic sanctions implied by the general definition of certification (see Chapter 2) favor notions of compliance to established norms. Still, where standard measures of component properties are available—as for certain classes of hardware components—some level of descriptive certification can take shape simply to allow manufacturers to differentiate their products in the marketplace.

The following argument is worth considering: normative certification is really descriptive certification with associated acceptance criteria or, more generally, a classification threshold for acceptance. If this argument is valid, descriptive certification would appear to be of independent significance since it permits the definition of various norms rather than a single “one size fits all” norm.

3.4 Objective Vs. Subjective

Objective measures are those that depend only on the object of study and possibly some physical apparatus. In contrast, subjective measures are the product of some mental activity. While this distinction is clear, an illustration can help explain some less than obvious subsidiary points.

---

[6] For more information, go to http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html.

[7] For more information on benchmarking transaction processing systems, go to http://www.tpc.org/. To see various papers on benchmarking middleware system, go to http://nenya.ms.mff.cuni.cz/projects/corba/oopsla-workshop-03/.

Consider, for example, measures of a quality attribute that we might call the *understandability* of a computer program. A candidate objective measure of this quality is “cyclomatic complexity,” which is defined as a numerical measure derived from the structure of a computer program (in particular, control flow).[8] A candidate subjective measure might be a statement posed to a human subject, such as “this code is understandable,” with a numerical measure derived from responses to this statement ranging from “strongly agree” to “strongly disagree.”

The virtue of objectivity is that the measure is repeatable. But repeatability does not suggest anything about other important qualities of the measure such as reliability. It is not clear whether cyclomatic complexity is more reliable than the proposed subjective measure, and, in fact, there is reason to think just the opposite.

Note, too, that quantification is not the sole purview of objectivity; in practice, subjective measures are quite likely to be quantified. A significant body of literature on multi-attribute decision making depends on quantified judgement. Also, Fenton and Pleeger provide a nice discussion of the coevolution of measures, from qualitative to quantitative in tandem with an improved understanding of the phenomenon being measured [Fenton 97]. Their thesis suggests that objective measures of software will become increasingly prominent, since qualitative measures (which are supplanted as understanding improves) are inherently subjective.

3.5 Product Vs. Process

In this report, we have tacitly assumed that the subject of certification is the software product itself. Yet, normative certification standards for software tend to place significant importance on the development processes used to create the software. This again may reflect the dearth of reliable (objective or subjective) measures of software quality. An emphasis on process certification illustrates the use of an *indirect,* or *proxy,* measure of software quality.

Product certification is likely to be most useful where conformance testing can be fully mechanized, such as the earlier examples of certifying interface compliance (the POSIX example) and test suite compliance (the ACVC example). Product certification has the merit of emphasizing objective measures, which, as previously noted, improves repeatability. We have a strong intuition (not substantiated by measure theory or current practice) that product certification will yield more reliable measures of software quality than process certification, despite the known correlation between better software process and better quality software.

Process certification is more useful when expert judgement is required and the cost of formulating expert judgements from product artifacts is high. For example, certifying that software is safe in so-called human-rated systems requires detailed examination of design artifacts (for example, requirements specifications, architectural specifications, and source code). A less costly but *possibly* reliable measure may be the quality of the software engineering processes used to manage requirements, architectural design, and implementation.

---

[8] For more details on this measure, go to http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html.

3.6 Local Vs. Contextual

Local properties of software are those that can be ascertained independently of their use context. Contextual properties, in contrast, are just the opposite: they require a context.[9] A local property of software might be its size in source lines of code—which is an illustrative, if not generally useful, measure. A contextual measure of the same software might be its reliability, since reliability will, in most cases, depend entirely on how the software is used—that is, on what input it receives from its environment (its context).

This dichotomy is of direct relevance to the question of certifiable software components, since the properties of software that are of most interest tend to be contextual. In other words, the properties of interest tend to be those exhibited by systems as a whole rather than by any one part of a system. Put in yet another (but more systems-theoretic) way, the properties of interest, such as safety, tend to be emergent properties—those that arise as a result of the interactions among the parts of a system. (For a discussion of emergence, see the monograph on safety engineering by Leveson and associates [Leveson 04]).

This notion of emergence has some rather obvious implications for the prospects of certifying software components. Certifying that a software component is safe requires the certification to be “contextualized” in at least two ways. First, the definition of what it means for a component to behave safely will vary from context to context. Second, the certified component property must be correlated in some way to the emergent behavior, since that, rather than the component behavior, is what is of interest.

3.7 Determinate Vs. Predictive

We say that certification is *determinate* if norms can be established for local properties of components and *predictive* if otherwise.[10]

---

[9] Fenton and Pleeger [Fenton 97] refer to this distinction as *internal* (for local) and *external* (for contextual) measures, but the terminology adopted in this technical note is more transparent. Also, the antonym of local is *global*, which might better represent the distinction. However, *global* suggests an absolute notion, whereas *contextual* suggests a more relative notion, that is, the context for one software element may simultaneously be an element in another (its own) enclosing context. For this reason, I prefer the term *contextual*.

[10] Of the distinctions presented, this is the only one that directly reflects our research bias. The other distinctions reveal something of the state of the practice of software certification, while this one reveals something of our proposed improvement of this practice. This distinction is also the most difficult to describe; more work is needed to understand the underlying issue.

To illustrate predictive certification, consider the contextual property "the system satisfies all performance deadlines." Also imagine that our interest is in certifying the performance properties of the system's software components. Does the software component have any purely local properties that can be certified as conforming to this contextual property? Probably not. We might measure the non-blocking execution time of all component traces (non-blocking to eliminate non-local effects). The validity of this measure assumes that traces are not dependent on environmental input—a situation that is unlikely. We are also excluding the role of processor, memory, and so on as contextual elements of such benchmarks.

Although the measure of execution time is arguably not contextual, it is clear that this particular measure is insufficient for determining whether the real contextual requirement—meeting deadlines—is satisfied. For this purpose, we need a sufficiently detailed model of the context, one into which we can "plug" the component execution time to determine (predict) deadline satisfaction. Even if the context were defined well enough for execution time norms to be established for components, the effect is still predictive rather than determinate. That is true because, in any case, the component execution time is insufficient in itself to fulfill the contextual property of deadline satisfaction.

To illustrate determinate certification, consider first the rather simple properties of conformance to interface specifications or the satisfaction of specified test cases (again, the POSIX and ACVC examples). These properties are determinate by definition. A less trivial illustration is to certify the total correctness of a software component—that is, that it always computes the correct result and terminates for input within specified bounds. Here, the contextual assumptions are finessed, but not eliminated, by circumscribing the behavior of the context (the environment). The result is still determinate, however, because the proof of correctness requires only these assumptions.

If this last illustration is modified, however, from certifying component-level correctness to certifying system-level safety, matters become more obscure. We may now require (as we did in the performance illustration) a model of the context that is detailed enough to expose those patterns of interaction among components from which safety will emerge. Now the proof of component correctness is just a parameter of a larger proof of safety. Further contextual assumptions may need to be more detailed; for example, those in the "assume-guarantee" clauses that are often required to construct a *compositional* proof of correctness of concurrent software. Certification may still be determinate in this case, but the boundaries are becoming blurred.

To recap, the performance properties of components can be certified only predictively, since these properties are inherently contextual. In contrast, properties that are inherently local to components can be determinately certified. A grey area arises when the properties are local to components, but the norms are contextual. If we accept Fenton and Pleeger's thesis that contextual properties are the main ones of interest, perhaps we should also accept that predictive certification is likely to be more interesting than determinate certification for software components.

3.8 Formal Vs. Empirical

This distinction differentiates those software component properties established through proofs in some system of logic (formal properties) from those established through observation and measurement (empirical properties).

It is natural to think of formal techniques as constituting a gold standard for the software properties dependent exclusively on software (i.e., those not dependent on the peculiarities of a particular computing environment such as memory and processor speed) and to consider empirical properties as a compromise at best. The second-class status of empirically derived evidence is certainly implied by Meyer, who assigns formal proof the high-road status, while relegating non-proof-theoretic approaches the low-road status [Meyer 03]. It is difficult to disagree with this assessment, especially if a computer program is accorded its true stature as a formal specification.

In some cases, though, we must rely on measurement for those properties dependent on the physical computing environment; for example, the consumption of time and power and the generation of heat—all of which may depend on the number of instructions needed to compute a particular function, on the mix of instructions used, and on the characteristics of hardware (which often vary from batch to batch in a manufacturing process).

Measurement may also be useful, however, when applied on a scale that would be impractical for formal techniques. There are analogues in the physical sciences. For example, the ideal gas laws are, in principle, an approximation of forces that could be computed by applying the laws of motion, assuming inelastic collision, on the scale of molecular interaction [Fermi 56]. There is no need to do so, however, since the law of large numbers assures us that the ideal gas laws adequately describe the emergent properties of temperature and pressure at a particular level of system description.

This ideal gas law example may seem contrived and far removed from software, but software analogues are not hard to find. For example, automata-theoretic models of program execution provide a sound foundation for microscopic theories of program behavior, and those models are exploitable by formal verification techniques such as temporal logic model checking. These techniques suffer from state space explosion, however, for even moderately complex computer programs, to say nothing of large-scale systems comprising many hundreds of such programs. At these levels of system scale, various forms of statistical quality control and statistical testing have proven more practical and effective.

Probability theories that provide meaningful models of the behavior of software systems with large and potentially unbounded state space have been applied also in the new discipline of real-time queuing theory (RTQT) [Lehoczky 97] and, quite recently, in survivability models of software systems [Jha 01]. These techniques are favored because they provide more convenient and tractable analysis of probabilistic behavior, while still exhibiting the soundness of an underlying mathematical theory. And, as noted earlier, such theories can provide a basis for predictive certification.

Formal and empirical may be described here as extremes of a dichotomy, but they are not mutually exclusive. For example, model checking and testing have been combined by using empirical testing to verify that a developed system is indeed a refinement of an abstract model, where that model is subjected to formal verification [Havelund 01].

3.9 Procedural Vs. Mechanical

Procedural certification mandates a role for some authorized, human, certifying agent. In other words, procedural certification involves a "human in the loop." In contrast, mechanical certification can, in principle at least, be fully automated.[11]

Normative certification standards currently in use in the software industry (see, for example, the safety standards mentioned earlier) are procedural. It is hardly possible to conceive that compliance with these standards can be reduced to mechanical checking. Moreover, these standards have spawned a support industry of independent verification and validation (IV&V) laboratories that provide testing services for standards compliance. This business model serves industry well, since IV&V allows for a specialization of skills, provides a means of reducing the liability exposure of system producers, and helps generate social trust.

Mechanical certification has appeared in the commercial marketplace. A notable example is Microsoft's approach to certifying device drivers. Drivers are subjected to a battery of compliance tests, including the use of a fully automated suite of model checking tools that formally verify a specified set of *safety*[12] conditions (a "safety policy"). The logic of a mechanical approach seems well suited to Microsoft's business need to establish social trust in its products without requiring a large investment in a "human in the loop" certification process.

Intriguing possibilities for mechanical certification are also emerging in research in proof-carrying code (PCC), sometimes referred to as self-certifying code [Necula 96]. In one variant of this concept, a consumer publishes a safety policy,[12] and certifying compilers insert a proof of this safety policy's satisfaction directly into component code. The target system in which the component will be placed executes an efficient check of the proof. One interesting characteristic of this approach is that fraudulent proofs are definitely rejected, while adulterated code (for example, code that has been tampered with or otherwise compromised) will be accepted only if it satisfies the published safety policy.

---

[11] The formulation of *mechanical versus procedural* is preferred to *automated versus manual* because some processes that are automatable in principle may, for practical reasons, make use of manual processes—the running of test programs, for example. The formulation chosen in this technical note reflects the inherently procedural nature of an activity.

[12] Note that this use of the term *safety* refers to a particular class of properties that can be established by examining the prefix of finite execution traces. This is **not** the same notion of safety used by the various safety standards mentioned earlier.

PCC is an approach to generate trust in components even if component suppliers and distribution channels (e.g., for mobile code) are untrusted. Research in so-called "foundational PCC" takes this approach to the extreme by systematically reducing the "footprint" of the trusted computing base required for PCC [Appel 01]. For example, in Necula and Lee's original work [Necula 96], the proof checker must be a trusted component. Foundational PCC allows this component, and various others of the original PCC work, to be untrusted, achieving what Lee has referred to euphemistically as "a conjurer's trick"[13] of generating trust from a trustless environment.

3.10 Standard Vs. De Facto

One assertion made in this technical note (and elsewhere) is that certification is a means to an end—it is a tactic that can be used to establish trust and, possibly, reduce liability. Trust and reduced liability are economic and social ends that transcend the objects of certification—particular systems or components of systems. Accordingly, in many cases, there are significant economic and social interests at stake in how certification norms are defined and compliance is enforced (sanctions, etc.). The established fora for reconciling these interests are the recognized standards bodies—the Institute of Electrical and Electronics Engineers (IEEE), the International Electrotechnical Commission (IEC), the American National Standards Institute (ANSI), the International Standards Organization (ISO), the Federal Information Processing Standards (FIPS), and so forth.

As we saw with the example of Microsoft device drivers, however, not all standards must be established by recognized standards bodies. Other standards may be imposed by particular businesses (for example, Microsoft), by standards bodies that are, in effect, proxies for one or more businesses (for example, the Object Management Group), or for particular business segments (for example, automotive and workflow management). Such standards are often called (sometimes pejoratively) *de facto* standards. Those standards tend to emphasize concerns such as branding (for example, "Microsoft Certified" or "100% Java").

The distinction between standard and de facto certification may not seem, on the surface, to be significant. Branding is, after all, just another means of establishing trust—in this case, through product uniformity. On the other hand, there is no question that the approaches used to establish sanctioned and de facto standards are radically different. Intuitively, deciding which route to take for certified software components seems to be important in defining a starting point for both the effort (e.g., a particular safety standard) and the strategy used to achieve consensus.

---

[13] This is an overstatement, but just barely. One has to trust something, but if foundational PCC is successful, all that must be trusted are the foundations of formal logic.

4 Related Work

Meyer has long been an eloquent proponent of many aspects of software quality and has, most recently, been associated with the notion of trusted components. He is the creator of design by contract (DBC), a technique that embraces formal methods and testing to improve the programming process and the quality of the resulting programs. Features to support DBC are incorporated directly into Meyer's Eiffel programming language, although DBC features have appeared in other languages as well [Arnout 02], [Kramer 98].

Meyer's most recent work on the subject of trusted components is geared toward establishing a component certification center. The center has two metaphorical roads: (1) the high road, based on proofs of component correctness and (2) the low road, based on a rudimentary but usable component quality model [Meyer 03]. It is not clear how much progress has been made toward the goal of the certification center.

Voas has been an outspoken advocate of software warranties and software certification. Much, but not all, of Voas's work has been directed to the use of commercial off-the-shelf software in settings requiring a high level of quality assurance. Voas's notion of a software certification laboratory [Voas 00] has a similar intent to Meyer's certification center, but is more distributed in operation (at least in concept, since no such laboratory has been developed).

Microsoft Corporation has several active areas of investigation and product development that have some bearing on the topic of software certification. The SLAM toolkit [Ball 01] implements a formal verification technique used to verify device drivers against a prespecified set of safety conditions (for a definition of *safety*, see footnote 12 on page 11). Various efforts are based on Gurevich's Abstract State Machine Language (AsmL) [Gurevich 04]; for example, the use of monitors and "spying" for runtime contract verification [Barnett 01]. For example, intriguing work reported by Barnett and Schulte uses AsmL, runtime monitors, and features of the COM component technology to perform runtime conformance checking between a component implementation and a formal specification of its behavior. This is another example of combining formal and empirical testing techniques (the other, already cited, is Havelund and Rosu's work [Havelund 01]).

The National Infrastructure Assurance Partnership (NIAP)[14] is a U.S. government initiative, initially conceived as a collaborative effort of the U.S. National Institute for Science and Technology (NIST) and the U.S. National Security Agency (NSA). The goal of the NIAP is “to help increase the level of trust consumers have in their information systems and networks through the use of cost-effective security testing, evaluation, and validation programs.” The NIAP represents a traditional, normative approach to software certification.

---

[14] For more information on the NIAP, go to http://niap.nist.gov/index.html.

NIAP certification norms are referred to as the *common criteria* and *protection profiles*, which are product-category-specific “slices” through the common criteria (e.g., “firewall” and “operating system”). The common criteria are based on the NSA’s “Orange Book” security norms but have been updated to reflect the emergence of a robust marketplace in commercial software technology. To date, over 120 products have been validated as compliant with some aspect of the common criteria,[15] and the NIAP has established 8 separate testing laboratories.

Underwriter’s Laboratory (UL) has developed the ANSI/UL1998 standard for the safety of software in programmable components. Like the NIAP, UL1998 is normative. Unlike the NIAP, however, UL1998 places a much stronger emphasis on process norms, which is consistent with the system-level safety certification standards discussed earlier. UL1998 also shares with the NIAP its specialization to the concerns of software, focusing not on the system as a whole (the *programmable component* in UL1998-speak), but rather on its software element.

---

[15] For a list of those products, go to http://niap.nist.gov/cc-scheme/vpl/vpl_name.html.

5 Conclusions

Based on the discussions in this technical note, some preliminary observations about certification in practice are that it

- emphasizes process more than product, subjective rather than objective, normative rather than descriptive, determinate rather than predictive, and procedural rather than mechanical approaches
- involves a recognized, possibly designated certifying authority; the certifying authority need not be, and often is not, the same as the testing agent.
- does not accommodate provisionality, at least in part because the emphasis on process and subjective measures presents significant challenges to quantifying uncertainty

What do these observations mean for the PACC agenda? Compared with the first bulleted observation above, PACC emphasizes product rather than process, objective rather than subjective, descriptive rather than prescriptive, predictive rather than determinate, and mechanical rather than procedural approaches to component trust. While the PACC notion of a *certifiable component* is clearly quite different from current practice, it is not unprecedented, as evidenced by the Microsoft device driver certification program. Since the Microsoft program is driven by clear-eyed business considerations, its contrary emphasis (like that in PACC) should be taken seriously.

The PACC concept of certifiable component is likely to be best received when the certifying authority is motivated primarily by bottom-line considerations; for example, manufacturers of software or devices that have tight design tolerance and are constructed from *or extended by* third-party software components. In contrast, the PACC concept of certifiable components will be difficult to transition in two situations: (1) when the certifying authority is part of an official regulatory process, since, in that setting, the bottom line is but one of many large-scale social interests at stake and (2) where the regulators must be wary of being overly prescriptive and imposing competitive or other economic hardship on manufacturers of software and software-intensive systems.

Bibliography

URLs are valid as of the publication date of this document.

[Appel 01]	Appel, A. "Foundational Proof-Carrying Code," 247-256. *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*. Boston, MA, June 16, 2001. Washington, DC: IEEE Computer Society, 2001.
[Arnout 02]	Arnout, K. & Simon, R. "The .NET Contract Wizard: Adding Design by Contract to Languages Other than Eiffel," 14-23. *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS 39)*. Santa Barbara, CA, July 29-August 3, 2001. Washington, DC: IEEE Computer Society, 2002.
[Ball 01]	Ball, T. & Rajamani, S. "The SLAM Toolkit," 260-264. *Proceedings of the 13th Conference on Computer Aided Verification (CAV 2001)*. Paris, France, July 18-22, 2001. 2001. Berlin, Germany: Springer, 2001.
[Barnett 01]	Barnett, M. & Schulte, W. "Spying on Components: A Runtime Verification Technique." *Proceedings of the OOPLSA 2001 Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*. Tampa Bay, Florida, October 14, 2001. http://www.cs.iastate.edu/~leavens/SAVCBS/2001/
[Demillo 77]	Demillo, R.; Lipton, R.; & Perlis, A. "Social Processes and Proofs of Theorems and Programs," 206-214. *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*. Los Angeles, California, January 17-19, 1977. New York, NY: Association for Computing Machinery, 1977.
[Fenton 97]	Fenton, N. & Pleeger, S. *Software Metrics: A Rigorous and Practical Approach*. Boston, MA: PWS Publishing Company, 1997 (ISBN 1-85932-275-9).
[Fermi 56]	Fermi, E. *Thermodynamics*. New York, NY: Dover Publications, 1956 (ISBN 048660-361-X).

[Gurevich 04] Gurevich, Y.; Rossman, B.; & Schulte, W. *Semantic Essence of AsmL* (Technical Report MSR-TR-2004-27). ftp://ftp.research.microsoft.com/pub/tr/TR-2004-27.pdf

[Havelund 01] Havelund, K. & Rosu, G. *Testing Linear Temporal Logic Formalae on Finite Execution Traces* (RIACS Technical Report 01-08). Moffet Field, CA: Research Institute for Advanced Computer Science, 2001.

[Jha 01] Jha, S. & Wing, J. “Survivability Analysis of Networked Systems,” 307-317. *Proceedings of the 23rd International Conference on Software Engineering (ICSE).* Toronto, Canada, May 12-19, 2001. Washington, DC: IEEE Computer Society, 2001.

[Kramer 98] Kramer, R. “iContract - The Java Design by Contract Tool,” 295-307. *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 26)*. Santa Barbara, CA, August 3-7, 1998. Los Alamitos, CA: IEEE Computer Society, 1998.

[Lehoczky 97] Lehoczky, J. “Using Real-Time Queueing Theory to Control Lateness in Real-Time Systems,” 158-168. *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems.* Seattle, Washington, June 15-18, 1997. New York, NY: ACM Press, 1997.

[Leveson 04] Leveson, N.; Daouk, M.; Dulac, N.; & Marais, K. *A Systems Theoretic Approach to Safety Engineering.* http://esd.mit.edu/symposium/pdfs/monograph/safety.pdf (2004).

[Meyer 98] Meyer, B.; Mingins, C.; & Schmidt, H. “Providing Trusted Components to the Industry. *IEEE Computer 31*, 5 (May 1998):104-105.

[Meyer 03] Meyer, B. “The Grand Challenge of Trusted Components,” 660-667. *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. Portland, Oregon, May 3-10, 2003. Los Alamitos, CA: IEEE Computer Press, 2003.

[MWU 96] *Webster’s New Universal Unabridged Dictionary.* New York, NY: Barnes & Noble Books, 1996 (ISBN 0-7607-0288-8).

[Necula 96] Necula, G. & Lee, P. “Safe Kernel Extension Without Runtime Checking,” 229-243. *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI)*. Seattle, Washington, October 28-31, 1996. Berkeley, CA: Association for Computing Machinery, 1996.

[Shaw 96] Shaw, M. “Truth Vs. Knowledge: The Difference Between What a Component Does and What We Know It Does,” 181-185. *Proceedings of the Eighth International Workshop on Software Specification and Design*. Schloss, Germany, March 22-23, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[UL 98] Underwriters Laboratories. *UL Standard for Safety for Software in Programmable Components (UL 1998), Edition 2*. Northbrook, IL: Underwriters Laboratories, 1998.

[Voas 98] Voas, J. “Certifying Off-the-Shelf Software Components.” *IEEE Computer 31,* 6 (June 1998): 53-59.

[Voas 99] Voas, J. “Certifying Software for High-Assurance Environments.” *IEEE Software 16,* 4 (July/August 1999): 48-54.

[Voas 00] Voas, J. “Developing a Usage-Based Software Certification Process.” *IEEE Computer 33*, 8 (August 2000): 32-37.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. **AGENCY USE ONLY** (leave blank)

2. **REPORT DATE**

September 2004

3. **REPORT TYPE AND DATES COVERED**

Final

4. **TITLE AND SUBTITLE**

Software Component Certification: 10 Useful Distinctions

5. **FUNDING NUMBERS**

F19628-00-C-0003

6. **AUTHOR(S)**

Kurt C. Wallnau

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. **PERFORMING ORGANIZATION REPORT NUMBER**

CMU/SEI-2004-TN-031

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. **SUPPLEMENTARY NOTES**

12.a **DISTRIBUTION/AVAILABILITY STATEMENT**

Unclassified/Unlimited, DTIC, NTIS

12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

Using software components to develop mission-critical systems poses a number of technical, organizational, and economic challenges. One persistent and largely unaddressed challenge is how the consumers of software components—that is, the developers of mission-critical systems—can obtain a meaningful level of *trust* in the runtime behavior of software components. The most frequently cited concerns are centered on issues of security; for example, trust that a component does not contain malicious code or exhibit vulnerabilities that can be exploited by malicious code. There are, however, other concerns about software component behavior that can be just as important. For example, in an embedded weapon system, it may be crucial to trust that a component will always execute a function within a particular time bound or never introduce unbounded priority inversion.

Certification is a practical, proven means of establishing trust in various sorts of things in other disciplines and is, therefore, a natural contender for developing trust in software components. This technical note does not propose a particular certification regimen for components. Rather, it introduces a series of 10 distinctions that can help in understanding different aspects of certification in the context of software components.

14. **SUBJECT TERMS**

software certification, certified software components, trusted software, trusted software components

15. **NUMBER OF PAGES**

28

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT**

UNCLASSIFIED

18. **SECURITY CLASSIFICATION OF THIS PAGE**

UNCLASSIFIED

19. **SECURITY CLASSIFICATION OF ABSTRACT**

UNCLASSIFIED

20. **LIMITATION OF ABSTRACT**

UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102