

Certifying Software for High Assurance Environments

Jeffrey Voas

Reliable Software Technologies Corporation

Abstract

Software certification processes have become so intertwined with development processes that true product certification, which should demonstrate that the software will behave appropriately, is almost never employed. This deficiency calls for a new generation of certification processes. This paper presents a methodology for certifying software systems that will be employed in environments with high assurance requirements. Our methodology is independent of how the software was developed. Until software certification becomes a software-centered activity as opposed to a process-centered activity (e.g., CMM, TCSEC/Common Criteria, etc.), labeling software as “highly reliable,” “high assurance,” “high integrity,” “safe,” “secure” etc., is suspicious.

Keywords

High assurance certification, reliability, testing, process models, fault tolerance, fault injection.

1 Introduction

Immature professions attempting accurate anomaly detection will likely suffer from misclassification errors, particularly for those anomalies with low detectability. Take for instance the history of the early detectors that were used to classify human blood with the AIDS virus. 1989 was the year that the first test for HIV-1 was licensed (even though the AIDS virus was first identified in 1981). That test was not sufficient, however, for the HIV-2 strain. The first Food and Drug Administration (FDA) approved test kit license for HIV-2 occurred in 1990, and it was not until 1991 that the FDA was able to license a single test kit capable of testing for both forms of the virus. And in 1996, the FDA approved the first antigen test kit,

Coulter HIV-1 p24 Antigen Assay, that further reduced the number of transfusion-related HIV-1 infections by 25% per year.

Because of these advances, it is estimated that the risk today of contracting AIDS from a blood transfusion has dropped by more than 99% from 1983 to 1991.¹ It is now estimated that only 18-27 blood donations per year to the nation's blood supply are missed. That translates into approximately 1 in 450,000 to 660,000. Our vision for how software certification processes will evolve over the coming years is similar. As newer and more accurate certification filters are developed, classification errors for software quality will decrease.

This paper presents a methodology for certifying whether software *will do what we, the customer or user, want*. If the software will, it deserves to be labeled as being suitable for environments with high assurance requirements. For brevity, we will term such software as *high assurance software*. For us, the anomalies that we wish to detect via our software certification process are those software outputs that would allow a catastrophic event to occur to the system that the software is a part of.

The methodology that we will describe is applicable to all types of software, covering the spectrum from safety-critical control software and security-critical systems to games. We suspect, however, that our certification methodology will mainly interest the stake-holders (insurers, owners, and end-users) of safety-critical, security-critical, and enterprise-critical systems.

When software does not do what we want, it is because the software exhibits undesirable behaviors. Those undesirable behaviors may be traced to a defective specification, defective software, incorrect usage, or a combination of these. Note that there are several levels at which undesirable software behavior can be defined. First, there is the Utopian set of software behaviors, which for *any* set of circumstances, define precisely what we, the customer or user, want the software to do in our environment.

It is fair to think of Utopian software behaviors as absolute universal truths concerning what is "good" software behavior versus "bad" behavior (for *all* circumstances that the software could ever find itself in for a fixed environment). Utopian behaviors are a system-level, omnipotent viewpoint that drills down to the software level and defines what the software must do for a particular system event.

Secondly, the software's *specification* also defines "good" versus "bad" but it is possible that the specified behaviors conflict with the Utopian behaviors. Specification behaviors are usually what we use to determine whether software is *correct*, but correct software can still cause serious system-level problems.² Because so much software that is written today

¹This was reported by the FDA in their press release P91-23 on 09/26/96.

²Here, system includes the environment that the software is embedded in, and not just the software.

carries an implicit goal of reuse (although that may frequently fail to occur), e.g., COTS functionality, it is not possible that the software publisher could have considered the Utopian behaviors unique to each target system when they wrote their software’s specification. Ideally, for one-of-a-kind systems with custom software, the specification behavior should equal the Utopian behaviors, but errors such as missing requirements will cause this to not be true.

Because of how we defined Utopian behaviors, if the software *always* exhibits those behaviors, then system-level problems cannot occur. Here, the software *is* still of high assurance even if it violates the behaviors defined in the software’s specification. Thus it is preferable for the software to exhibit the Utopian behaviors instead of the specification’s behaviors in cases where they disagree.

Software certification schemes that claim to be able to certify high assurance must be able to determine whether the software can exhibit undesirable system-defined behaviors. The problem, however, is that specification-defined software behaviors may conflict with the Utopian “system-level” behaviors. Our certification methodology will address this discrepancy.

2 The High-Assurance Pipeline

We are now ready to walk the reader through our high assurance certification model (See Figure 1). Note that although we order the certification processes in Figure 1, in practice, there may be reasons to iterate or change the ordering of these pipes.

In our model, software certification processes will be independent of the processes employed in the “Requirements and Specification” and “Develop Software” pipes. Note however that the end-result of certification will definitely depend on the techniques used within the “Develop Software” phase of the life-cycle (as well as depend on those techniques being applied performed.) Before unveiling the model, we will first discuss the initial processes that must be performed before certification begins.

2.1 Pipe 1 Requirements and Specifications

In school, budding software engineers are taught to write their requirements in a concise, unambiguous, and complete manner. Software engineers are taught that if they do not define what they want the software to do, their software is unlikely to do it. Defining requirements is an exercise that occurs in the first pipe of the pipeline and is needed before software development can begin.

But defining what we want the software to do is not sufficient for high assurance. It is also necessary to define what we do not want the software to do. There may be software behaviors

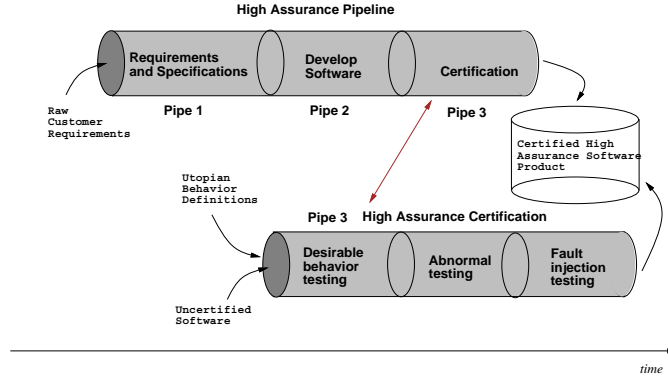


Figure 1 The high-assurance certification pipe

that the software could exhibit that are not ideal but can be tolerated by the system and do not impact negatively on the assurance afforded by the software. Since defect-free software is, in general, an oxymoron, it is prudent to succumb to certain problems while concentrating effort on thwarting those problems that are totally unacceptable.

An instantiation of an Utopian software behavior is defined as a three-tuple (a software input vector, the desired software output or range of acceptable software output values, and the state of the system or environment at the time when this input is executed). For a fixed software input, many different system states might be possible, and hence there could be ranges of output values that would be acceptable. Or for a fixed software input, there might be only one unique system state associated with it yet there might be a range of acceptable output values or possibly only one specific output that could be tolerated by the system. Software specifications, particularly for reusable software, may only consider two parts of the three-tuple (a software input vector and the correct software output or a range of acceptable output values).

For example, suppose that a specification defines the following input/output tuple (1,2). Suppose further that this is acceptable only if the system is in state A. For system state B, the software specification needs to require (1,6). Since the specification defined (1,2) and assuming that the software is deterministic and correct, the state (1,6,B) is not possible; only the states (1,2,A) and (1,2,B) are.

We are now ready to illustrate this principle more generally. Figure 2(A) shows two spaces \mathbf{D} is the space of desirable (correct) software output behaviors given the software's inputs, and \mathbf{U} is the space of undesirable (incorrect) behaviors. Here, the spaces in Figure 2(A) ignore the state of the target environment that the software will operate in. Software quality is defined in Figure 2(A) as if the software were a stand-alone entity.

To address the deficiency of ignoring system state, Figure 2(B) redefines four different

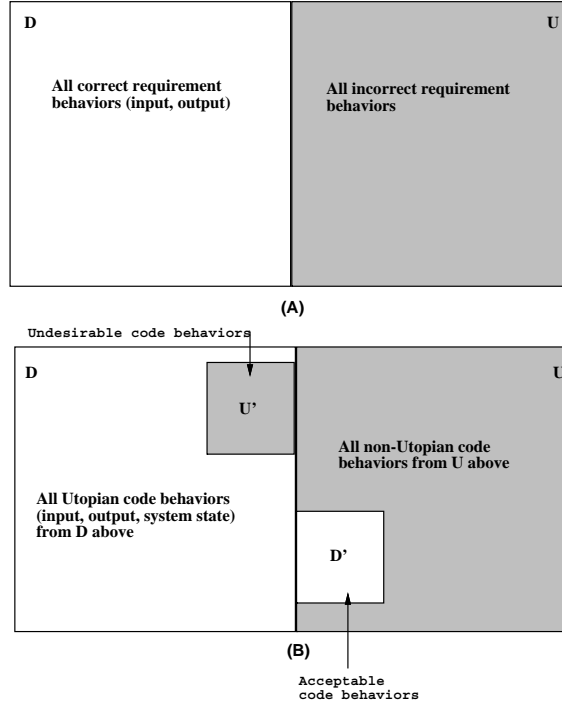


Figure 2 (A) Requirement behaviors based on inputs and outputs, (B) Code behaviors based on inputs, outputs, and system state.

classes of software behavior **D**, **U'**, **D'**, and **U**. In Figure 2(B) class **D'** represents those behaviors that are incorrect with respect to the specification but have no negative affect on the system as a whole. They are simply nuisances. Class **U'** is the reverse, where the behaviors are correct with respect to the requirements, but from the Utopian perspective are anything but good. **U** represents those behaviors that were not good according to specification and the Utopian perspective, and **D** represents those behaviors that were good according to specification and the Utopian perspective.

Thus in Figure 2(B), space $(\mathbf{D} \cup \mathbf{U}')$ is equal to the previous **D** space in Figure 2(A). And the previous space **U** from Figure 2(A) is now equal to $(\mathbf{D}' \cup \mathbf{U})$. In Figure 2(B), $(\mathbf{D} \cup \mathbf{D}')$ defines all Utopian behaviors, whereas $(\mathbf{U} \cup \mathbf{U}')$ represents those behaviors that must be shown are not possible before high assurance certificates are granted.

Note the implicit partitioning of *failure severities* between Figures 2(A) and Figure 2(B). Some of the *failures* defined in **U** in Figure 2(A) appear as *successes* in **D'** because their impact to the system was inconsequential and could be tolerated. Thus once system state is considered during software certification, software outputs defined as failures (when state was not considered) may find themselves redefined as successes or *vice versa*.

Certifying high assurance software, then, requires a convincing argument that the behav-

iors in $(\mathbf{U} \cup \mathbf{U}')$ cannot occur. Thus for high assurance certification, it is necessary to have access to definitions for \mathbf{D} , \mathbf{D}' , \mathbf{U} , and \mathbf{U}' .

2.2 Pipe 2 Design for High Assurance

The second pipe in Figure 1 builds software. The second pipe contains a cornucopia of processes for transforming software requirements into code. This pipe contains processes such as CASE tools, compilers, debuggers, design paradigms (*e.g.*, Bertrand Meyer’s *Design-by-Contract*, *defensive programming*, etc.). Because this pipe contains the more creative and enjoyable tasks associated with software engineering, this pipe usually receives more attention than the other two. But since this pipe is not the focus of this paper, we will quickly move into discussing the third pipe.

2.3 Pipe 3 Certification Through Stress

The processes in the “Develop Software” pipe are necessary to *achieve* high quality code, but they do not guarantee that high quality software will result. And simply employing specific best practices in the “Develop Software” pipe is not sufficient to trust the code either.

This section is devoted to three *assurance* certification processes that observe how frequently behaviors in $(\mathbf{U} \cup \mathbf{U}')$ manifest themselves. The processes that will be recommended here are radically different than those that the current state-of-the-practice calls for.

Current certification practice is highly *process*-oriented [5, 1]. It usually requires processes in the first two pipes such as criticality analysis, failure modes and effect analysis, fault tree analysis, event tree analysis, Markov modeling, formal methods, etc. In contrast, our *product*-oriented certification scheme contains procedures that only serve to exercise and *stress* the software. Note that we are not suggesting replacing design assurance techniques or any other processes in the “Develop Software” pipe. We are only arguing that certification must be independent of design assurance.

Our scheme will determine whether to grant high assurance certification without regards for what occurred in the first two pipes, with one exception being that we need information concerning $(\mathbf{D} \cup \mathbf{D}')$. No other information from the first two pipes will be considered by our certification methodology.

Our goal is simple justifiable high assurance certification based on demonstrated software behavior. For this, three processes will be used

1. **Desirable behavior testing** Demonstration that under *operational* input scenarios, the software performs only those environment-based behaviors contained in $(\mathbf{D} \cup \mathbf{D}')$.

2. **Abnormal testing** Demonstration that under *abnormal* input scenarios, the software performs only those environment-based behaviors contained in $(\mathbf{D} \cup \mathbf{D}')$.
3. **Fault injection** Demonstration that failures of any subsystem, whether hardware or other software cannot cause the software to behave in a manner that contradicts those environment-based behaviors contained in $(\mathbf{D} \cup \mathbf{D}')$.

2.3.1 Desirable behavior testing

Our first certification process is “desirable behavior testing.” *Desirable behavior testing* (DBT) checks to see if the software exhibits behaviors in $(\mathbf{U} \cup \mathbf{U}')$, and if so, counts those as software failures. Output behaviors in $(\mathbf{D} \cup \mathbf{D}')$ are counted as successes. In comparison, *reliability testing* seeks to ensure that the software performs correctly according to the correct behaviors defined in the requirements.³

Existing certification standards often recommend that reliability testing be performed [3]. But note that DBT is a slight twist on traditional reliability testing all behaviors defined in Figure 2(B) are defined with respect to the system, and not simply based on (input, output) pairs as illustrated in Figure 2(B). Thus reliability testing by the software publisher (or even an independent third-party certification laboratory) cannot be equivalent to a system’s stake-holders performing DBT.

Like reliability testing, desirable behavior testing uses inputs selected at random from the “operational profile.” The operational profile for the software reflects the operational profile of the target environment that the software will be a part of. An *operational profile* describes the probability that each input will be selected when the software is deployed [12]. DBT provides an analysis of how well-behaved the code is when it is executing in operational modes. So for example, if an input value of ‘100’ to the software is likely to occur in the software’s target environment, DBT would execute the software using ‘100’ with appropriate system states and check its output.

As an actual example of the value of considering the system state when selecting test cases, consider the Ariane 5 rocket disaster [10]. In this failure, software was reused from the Ariane 4 rocket and embedded into the different trajectory environment of the Ariane 5. The software was reliable in the Ariane 4 environment (thus the software exhibited behaviors consistent with those in $(\mathbf{D} \cup \mathbf{D}')$). The software failed, however, when put into the Ariane 5 environment, because $(\mathbf{D} \cup \mathbf{D}')$ was different for Ariane 5 than for Ariane 4. Some of those members from Ariane 4’s $(\mathbf{D} \cup \mathbf{D}')$ now found themselves in Ariane 5’s $(\mathbf{U} \cup \mathbf{U}')$.

³Reliability testing would deem output behaviors shown in \mathbf{E} as failures, whereas DBT would deem them as successes.

2.3.2 Abnormal testing

After DBT is performed, the second certification process is *abnormal testing*. Abnormal testing employs infrequent, rare test cases.

Rare test cases have low likelihoods of being selected according to the operational profile, but they are still possible candidates for execution after the software is released. Like desirable behavior testing, abnormal testing counts how many software outputs are members of $(\mathbf{D} \cup \mathbf{D}')$.

Abnormal testing leverages the information that defines the operational profile. Abnormal testing inverts the operational profile (as defined in [11]) and then performs the same processes as DBT. Just as with DBT, the operational profile used for the inversion must reflect the operational environment if the system that the software is a part of. This provides an analysis of how well-behaved the code is when it is executing in the more unlikely input modes. So for example, if an input value of '1' is unlikely in the target environment yet still possible, abnormal testing would force the software to receive a '1' (with appropriate system states defined in Figure 2(B)) and then observe whether any outputs satisfy those behaviors in $(\mathbf{D} \cup \mathbf{D}')$.

One other type of abnormal testing involves skewing the operational profile such that greater weight is given to (input, system states) pairs for which failure occurring would result in grievous system consequences. In certain cases, this consequence-based approach to test case generation will be identical to what we just defined as being abnormal testing, but more often, this variant on abnormal testing will allow us to try test inputs that hone in to exercise the software's more critical tasks. This form of abnormal testing may be difficult in practice, however, because the specialized test case generation functions needed may not be known.

In summary, abnormal testing uses profile-skewing techniques which can provide insight into how successful the early life-cycle processes were at developing well-behaved code under unusual circumstances. Because unusual inputs may be ignored during the publisher's testing as well as design, we consider it pivotal to our methodology.

2.3.3 Fault injection

After abnormal testing is completed, the third and final certification process is fault injection. *Fault injection* employs the same procedures already described in Sections 2.3.1 and 2.3.2 but adds a twist it uses specific "fault class" rules to mangle the internal states that are created as the software executes. This is particularly important for high availability systems that must function even if their input states are corrupted.

Fault injection observes whether the software still produces outputs consistent with $(\mathbf{D} \cup$

\mathbf{D}') even though its internal or input states were corrupted. So while the software executes under uncorrupted operational and abnormal inputs, fault injection corrupts states and observes whether the software still behaves appropriately. If so, the software is fault-tolerant and has good recoverability.

The difference between fault injection and abnormal testing and DBT is that fault injection tests the tolerance of the software to failures of system's subcomponents including external hardware, other software functionality than that being certified, and human user errors, meanwhile observing whether the software's outputs satisfy $(\mathbf{D} \cup \mathbf{D}')$.

Note that there will be situations where it is acceptable if the software does not behave in a manner consistent with $(\mathbf{D} \cup \mathbf{D}')$ after fault injection is applied. Specifically, the situation where it can be shown that the corrupted states employed during fault injection were such that there are no circumstances by which those mangled states could manifest themselves in "real-life." For each observed behavior that is not in $(\mathbf{D} \cup \mathbf{D}')$, if we can mitigate the possibility of it ever occurring naturally, then we ignore the fact that an undesirable behavior was observed, i.e., we will not hold this fact against the software when deciding whether certification is warranted. The goal, obviously, is to attempt to only employ corrupt states that are reasonable failure modes for the subsystems that the software under certification interacts with. But this cannot always be assumed.

Note that of the three processes in the certification pipe, software fault injection is the most intensive and most expensive to perform. It is also the one that is the most likely to ferret out unknown behaviors that are outside of $(\mathbf{D} \cup \mathbf{D}')$. This is because fault injection asks questions about the software's behavior that are much more probing than those asked by abnormal testing or DBT.

3 To Grant Certification or Not

We have recommended three processes for the high assurance certification pipe. To be certified, software should exhibit only those behaviors in $(\mathbf{D} \cup \mathbf{D}')$ during DBT, abnormal testing, and fault injection (recall that we ignore those violations of $(\mathbf{D} \cup \mathbf{D}')$ that we can mitigated away). We do not claim that our 3-part approach is fully *sufficient*. Additional certification technologies are needed. We do claim, however, that given existing technologies, the ones that we have included are *necessary*. And we recognize that other testing and quality assessment analyses exist that could be used as supporting evidence in favor of granting certification or they might even subsume the processes discussed here.

The immediate question that arises concerns how much certification effort should be allocated to each of the three processes. For example, if only one system-level test case were

used during each process, and all outputs satisfied ($\mathbf{D} \cup \mathbf{D}'$), would we be willing to grant certification? Of course not! But other than knowing an absurdity like this when we see it, it is not easy to select a single number as the required level of certification effort which is sufficient for all programs. Also, it might make sense to vary the level of effort between the different procedures, e.g., DBT might employ twice as many test cases as abnormal testing since few rare test cases might exist.

In summary, system stake-holders must take responsibility for decisions concerning how much effort to expend. After all, they carry to liability for the system and the software. If they pick an absurdly low value to reduce costs, their pocketbooks may bear the brunt of their negligence. Alternatively, if certification laboratories are employed to perform software certification, they too must take responsibility for decisions concerning how much effort to expend. It is *their* stamp of approval that goes onto the software. If they pick an absurdly low value to reduce costs, the world will quickly learn what their seal of approval is worth after the first catastrophe is reported.

4 Approach Validation

To show that our 3-part certification process is well-researched, we will employ a combination of analytical and empirical results. Also, where appropriate, we will show how our approach aligns with published statements from high-assurance software experts.

To begin, consider the fault injection process. In numerous experiments, the value of employing fault injection to thwart out undesirable behavior has been shown. Voas analytically showed how fault injection could have warned of the problems that occurred to the Therac-25 and Ariane-5 *before* those systems were deployed [13]. Voas *et al.* demonstrated how fault injection detected a serious flaw in the safety-monitoring routine of a nuclear control application [2]. The fault had not been discovered by any other V&V technique, even though many different testing techniques had been applied of a period of many years. In fact, that error made that part of the safety-monitoring software useless. In [4], similar results discussed how fault injection had detected an improperly placed safety-monitoring assertion that was needed to keep a voltage spike from occurring in a surgical device (UVA's Prototype Magneto Stereotaxis System). That same paper also described how fault injection had localized multiple fault-tolerance problems in the new subway control system. After localization, the developer corrected the potential hazards using a variety of different error recovery mechanisms.

Fault injection has also shown amazing abilities to detect the potential for buffer overflows in security-critical systems. Ghosh et. al. [7] reported successful results from analyzing the

vulnerability of security-critical software applications to malicious threats and anomalous events using fault injection. Their work was based on the well-understood premise that a large proportion of security violations result from errors in software source code and configuration. Their methodology employed software fault injection to force anomalous program states during the execution of software and observes their corresponding effects on system security. In [6], Ghosh et. al. presented a more specialized fault injection technique for analyzing security-critical software for vulnerability to buffer overrun attacks. This technique dynamically analyzed software source code to determine the potential to successfully overrun program buffers in order to execute arbitrary system commands. Their fault injection algorithm inserted malicious strings into potentially vulnerable buffers during execution. If the buffer overrun attack was successful, arbitrary malicious code could be executed at the whim of the attacker on the host system.

With respect to the abnormal testing process, [4] also showed how inverted operational profiles are an ideal way to test for the effects of unlikely input events. That experiment involved the `yawdamp.c` software function from an avionics simulation. The experiment revealed divergent safety behaviors between when the code's operational profile was inverted and when it was not.

Many others have noted the potential for hazards from odd events. In Leveson's book on page 60, she states [9]

“Usually, the most likely hazards are controlled, but hazards with high severity and (assumed) low probability are dismissed as not worth investing resources to prevent ...”

Abnormal testing can provide information concerning whether such hazards should be ignored or not. And on page 493 of Leveson's book [9], she recommends that test cases are used that account for “boundary conditions” and “incorrect and unexpected inputs and input sequences and timing (minimum, maximum, and outside the expected range)”. Herb Hecht echos a similar message from his years testing NASA Shuttle software [8]. He states

“Where the nature of a program demands extremely high reliability, it appears therefore that the test scenario should include a substantial number of case that simulate multiple (at least two) rare conditions.”

Hecht collected evidence showing that single rare conditions were difficult to detect via testing and multiple rare conditions were even harder. These are precisely the types of events that must be mitigated during high assurance certification. Abnormal testing provides that capability in our high-assurance methodology.

DBT is a necessary process because high-assurance is a system-level property, not software-level property alone. It is foolish to believe that correct software cannot cause terrible system-level damage. On page 495 of Leveson’s book [9], she states

“Reliability assessment of software does not assess safety.... Accidents are rarely the result of component failure of a type that would be easy to predict through such reliability testing. Rather, the events leading to an accident are usually a complex combination of equipment failure, faulty maintenance, instrumentation and control problems, management errors, design errors, and operator errors.”

This explains why our certification model employs the system perspective on software “badness” ($\mathbf{U} \cup \mathbf{U}'$) instead of \mathbf{B} . \mathbf{B} only considers the software specification’s definition of “badness”, and that fails to account for many of the problems that can cause disaster. To our knowledge, DBT is the first black-box software testing technique that employs the software’s operational profile but whose oracle is based on the high-assurance requirements of the system (and not necessarily of the software). Although we have not yet experimented with DBT yet, it does align with the common wisdom on best testing practices.

5 Summary

We acknowledge that in order to rank development organizations according to their technical sophistication, it is reasonable to grade them according to their processes and corporate infrastructure. It is equally reasonable to grade them on the historical success of their previous products (even though to our knowledge this is less frequently done except at the contractual level.) Certifying software products is different, however, than certifying the publishers.

Before we close, we need to highlight a subtle point that has been carried through the paper. Each certification process was a function of the environment’s operational profile, not the assumed profile by the software’s publisher. Thus it is possible that a program could receive high assurance certification for one profile while failing to receive it for another. This makes sense. High assurance software certification must be tied to specific system profiles; it does not make sense to grant ‘carte blanche’ high assurance certificates because with the exception of trivial systems, it cannot be justified.

Recognize that this “limited claims” approach is not unusual in other disciplines. The US Food and Drug Administration approves drugs for specific sets of circumstances. Drug A may be approved for specific diseases or specific age groups (like for adults and not children), but not approved for other circumstances. Given that software is discrete and

highly unpredictable, and given that our profession is still in its infancy, it is prudent to bound expectations to only those that are known to be true.

Our perspective on software certification differs with many of our peers. We believe that certification must be independent of how the software was developed. We contend that if an accurate system profile cannot be found, high assurance certification for the software cannot be justified. Further, we contend that high assurance certification should not be granted if attempts were not made to define the undesirable behavior space with respect to system state. We accept that this behavior space can never be fully defined, and in some cases will be incorrectly defined, leading to improper certification decisions. But it is time that this space start receiving attention early in the software life-cycle. The more we practice defining it, the better we will get at defining it.

And finally, a certificate designating that a program has attained the level of assurance known as “high” using our methodology does not also guarantee that the program will always be well-behaved. Risk is still incurred. Although disheartening, we must accept that there will always be a non-zero probability that the software has poorer quality than we believe. The goal, then, is to seek product-oriented quality measures that reduce this probability. Our certification “endurance course” (that makes the software jump through difficult hurdles) is one step in that direction.

Acknowledgments

This work has been partially supported by DARPA Contract F30602-95-C-0282, NIST Contract 50-DKNA-4-00119, and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160. I appreciate the feedback I received from Dr. Anup Ghosh, Dr. S. L. Pfleeger, Lora Kassab, Dolores Wallace, Jeffery Payne, and the thoroughness of the anonymous referees.

References

- [1] US FOOD AND DRUG ADMINISTRATION. Reviewer Guidance for Computer Controlled Medical Devices Undergoing 510(k) Review, 1991.
- [2] J. VOAS, F. CHARRON, AND L. BELTRACCHI. Error Propagation Analysis Studies in a Nuclear Research Code. In *Proc. of IEEE Aerospace'98*, Snowmass, CO, March 1998.
- [3] D. R. WALLACE, L. M. IPPOLITO AND D. R. KUHN. High Integrity Software Standards and Guidelines. US Department of Commerce, NIST, September 1992. Report 500-204.

- [4] J. VOAS, F. CHARRON, G. MCGRAW, K. MILLER, AND M. FRIEDMAN. Predicting How Badly ‘Good’ Software can Behave. *IEEE Software*, 14(4)73–83, July 1997.
- [5] FEDERAL AVIATION AUTHORITY. Software Considerations in Airborne Systems and Equipment Certification, 1992. Document No. RTCA/DO-178B, RTCA, Inc.
- [6] A.K. Ghosh and T. O’Connor. Analyzing programs for vulnerability to buffer over-run attacks. In *to appear in Proceedings of the National Information Systems Security Conference*, October 6-9 1998. Crystal City, VA USA.
- [7] A.K. Ghosh, T. O’Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 104–114, Oakland, CA, May 3-6 1998.
- [8] H. HECHT. Rare Conditions An Important Cause of Failures. In *Proc. of the Eighth Annual Conference on Computer Assurance*, pages 81–85, National Institute of Standards, June 1993.
- [9] N.G. LEVESON. *Safeware System Safety and Computers*. Addison-Wesley, 1995.
- [10] Prof. J. L. LIONS. Ariane 5 flight 501 failure Report of the inquiry board. Paris, July 19, 1996, available at http://www.cnes.fr/actualites/news/rapport_501.html.
- [11] J. VOAS, F. CHARRON, AND K. MILLER. Investigating Rare-Event Failure Tolerance Reductions in Uncertainty. In *Proc. of IEEE High-Assurance Systems Engineering Workshop*, Niagara-on-the-Lake, Canada, October 1996.
- [12] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987. ISBN 0-07-044093-X.
- [13] J. VOAS. Software Fault Injection Growing Safer Systems. In *Proc. of IEEE Aerospace’97*, Snowmass, CO, February 1997.