Is Certified Software Actually Correct?
(What does “certified” mean?)
(What does “correct” mean?)

Brian Larson

May 1, 2011

Certificate

A certificate is a document attesting to some fact(s).

What would a certificate for software attest to?

Must Distinguish Validation from Verification

- Verification says merely that software meets its specification.
- Validation says that the specification meets requirements, and that the requirements meet the design intent.

Validation Will Always Be Human

Validation will always be inherently subjective.

Formal methods can help make requirements and specifications complete and unambiguous.

Will, intent, and need are inherently human.

Regulatory Approval vs. Certification

- FAA "certifies" aircraft, engines, and propellers. Only whole systems; no composition of certified subsystems. Certificates attest to airworthiness.
- FDA "approves" medical devices. Although demanding evidence of safety and efficacy for approval, the FDA certifies nothing.
- NRC (operating license for NPPs individually? designs? significant upgrades?)

(If some kind of certification could limit liability of medical device makers, they will embrace certification enthusiastically.)

Certifying Verification

Verification can be made precise.

Software certificates attest to conformance to specification.

Formal methods can give higher confidence than testing alone.

Verification is Objective; Validation is Subjective

Verification can be objective; validation will always be inherently subjective.

Validation is crucial, of course.

Validation $\equiv$ Systems Engineering

See INCOSE Handbook for state-of-practice systems engineering.

Zhong Shao's Paper

Certified Software, Zhong Shao, in Communication of the ACM, December 2010

“Some think that this paper misses the essential difference between correctness and certification."

Shao specifically limits software certification to verification.

Shao's certificates attest to conformance to specification (correctness) not fitness for purpose, safety, or efficacy.

Proof ≡ Truth

Since proofs are incontrovertible mathematical truths, once a software component is certified, its trustworthiness (with respect to its specification) would presumably last for eternity.[1]

[1] p.61

Nobody Knows How Today's Software Works

For most of today's software, especially low-level forms like operating systems, nobody knows precisely when, how, and why they actually work. They lack rigorous formal specifications and were developed mostly by large teams of developers using programming languages and libraries with imprecise semantics.[2]

[2] p. 56

Lack of Metrics

- *Metrics are still lacking for measuring software dependability, making it difficult to compare different techniques and build steady progress in the field.*
- *Dependability often includes attributes like reliability, safety, availability, and security.*
- *A program with one bug is not necessarily 10 times more secure than a program with 10 bugs.*
- *A system's reliability depends on its formal specification, which is often nonexistent.*[3]

[3] p.57

Dependability

Worse, software dependability is often confused with the dependability of the software's execution environment, which consists of not just hardware devices but also human operators and the physical world.

Since the dependability of the execution environment is often beyond human control, many people view software as a complex biological system, rather than as a rigorous mathematical entity.[4]

[4] p.57

System Software

A software application's dependability also relies on the dependability of its underlying system software, including OS kernel, device driver, hypervisor, garbage collector, and compiler.

These low-level programs are often profoundly complex and bug-prone, but little has been done to make them truly dependable.[5]

[5] p.57

Last-Mile Problem

> *Most software-verification research concentrates on high-level models rather than on actual programsÑvaluable for finding bugs but leaving a big gap that must be closed before meaningful dependability claims can be made about actual software.*
>
> *Failure to reason about actual code also has serious implications for maintainability; for example, it is difficult for programmers to pinpoint the source and a fix when a new bug is identified and ensure that subsequent updates (to actual code) will not break the code's high-level model.* [6]

Model-checking just checks models–not the actual program.

Compilers and run-time services need verification to same level as application code.

Version control to ensure code in field is code verified.

[6] p.58

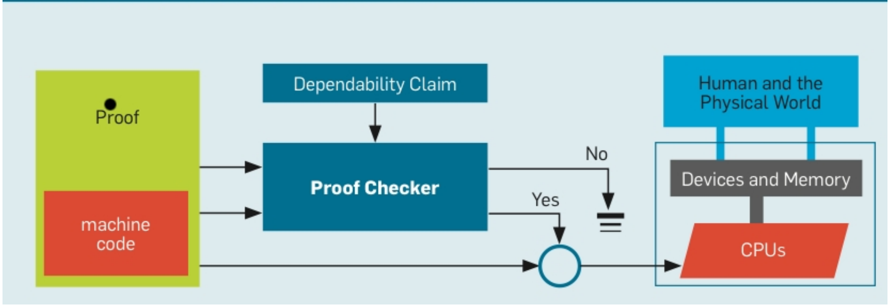
Certify Verification of Executable Machine Code

With a formal specification stating its desirable behavior, we can (at least in theory) rigorously “certify" that the machine executable behaves as expected. [7]

[7] p.58

Shao Certifies Limited Dependability Claims

Figure 1. Components of a certified framework.



8

[8] p.59

Mechanized Metalogic

Much of the current work on certified software is carried out in the Coq proof assistant. Coq itself provides a rich higher-order logic with powerful inductive definitions, both crucial to writing modular proofs and expressive specifications. [9]

[9] p.60

Not All Software Needs Highest Confidence

> *When dependability is not an issue, the software can be used as is, assuming proper isolation from the rest of the system; when programmers really care about dependability, they must provide the formal machine-checkable proof.*
>
> *[B]uilding certified software does not mean that programmers must verify the correctness of every component or algorithm used in its code; for example, in micro-kernels or virtual-machine monitors, it is often possible for programmers to verify a small set of components that in turn perform run-time enforcement of security properties on other components.* [10]

For medical devices, SOUP for TCP/IP stack and software radio, conventional verification (testing) of history and paramter setting, proof for therapy decisions (1% of code in can).

[10] p.61

Automation and Proof Engineering

The end goal of certified software is a machine-checkable dependability metric for high-assurance software systems.

Certified software advocates the use of an expressive metalogic to capture deep invariants and support modular verification of arbitrary machine-code components.

Machine-checkable proofs are necessary for allowing third parties to quickly establish that a software system indeed satisfies a desirable dependability claim.

Automated proof construction is extremely important and desirable but should be done only without violating the overall integrity and expressiveness of the underlying verification system.[11]

[11] p.65

SMT Solvers only First-Order

Existing automated theorem provers and Satisfiability Modulo Theories solvers work on only first-order logic, but this limited functionality conflicts with the rich metalogic (often making heavy use of quantifiers) required for modular verification of low-level software.[12]

[12] p.65

Higher-Order Theorem Provers and Proof Assistants are Painful

Proof tactics in existing proof assistants (such as Coq) must be written in a different “untyped" language, making it painful to develop large-scale proofs.[13]

[13] p.65

Current Proof Tools

- either too weak (SMT), or too hard to use (Coq)
- reason about a model–not the program itself
- can't prove temporal behavior with respect to an environment needed by cyber-physical systems.

Summary

- Software Certificates Attest to Verification, NOT Validation
  (validation is always system-level; software conformance to specification is one small part of overall fitness of purpose)
- Software Certificates Vary in Strength
  (type-checked, tested, SAVI-integrable, model-checked, total-correctness-proved)
- Verification Power in Proportion to Dependability Need
  (SOUP for TCP/IP stack, automated regression testing for history and programming, sampled and compared with model simulation, correctness proofs for therapy decisions)
- Prove (only) Crucial Dependability Properties of Critical Code
  (cost-effective to ensure critical fraction of code is defect-free)

Is Certified Software Actually Correct?

Not necessarily.

Can Systems Using Incorrect Software be Dependable, Safe, Secure, and Effective?

Yes!

A bug in the SOUP need not spoil the whole meal.