



Dennis de Champeaux

Software Engineering Considered Harmful

In search of a resolution to the ongoing software crisis.

Software engineering was introduced over 30 years ago—surprisingly at a NATO conference instead of, say, an academic institution—to attack the well-known software crisis. In spite of vigorous attempts and developments since then (OO, software development processes, development process metrics, IDEs, and UML, to name a few) the crisis continues. For example Denning observes: “confusing systems, software without warranties, begrudging technical support, surly customer service, inter-vendor finger-pointing, disregard for privacy, and even poorly managed, investment squandering dot-com companies. As a consultant during the last years, I can add to his observation: I earned absurd fees by finding obscure bugs, documenting incomprehensible systems, writing large log analyzers for out-of-control multithreaded systems, in general: cleaning up a growing mess.”

I volunteer, with understandable trepidation, the conjecture that the concept of software *engineering* has not been helpful, and plausibly has deeply exacerbated the software crisis.

Computer science is the traditional discipline (in the U.S.) under which software engineering resides. Here we have an immediate problem. The “science” designation is appropriate for the hardware side of computer systems. It allows investigation of the linkages between the core of physics and the materialization of data processing components and candidate assemblies. Hypotheses can be validated by concrete testing in the tradition of the sciences. This supports in turn computer engineering as a solid enterprise for the hardware side (which may, as other sciences, use software modeling to advance its aims).

The term “software engineering” suggests there is similarly a “software science” discipline that backs up software engineering. There is none. Testing, as a key corollary and observed by many, does not give the certainty of the corresponding activity in science. The results have been staggering, as indicated by the Denning quotation cited here. The bad news is that things are getting worse. Distributed computation with asynchronous communications yield non-reproducible timing errors, which can take forever

to isolate, if ever.

Certainly, I agree that the standard tricks of the trade (OO, abstraction, reviews, rigorous procedures, regression testing, metrics, and so forth) have been helpful, but we are still dealing with an intrinsically unsatisfactory situation. Especially because there is, I believe, a serious alternative.

Consider a mathematician who after working for many years produces a proof for a conjecture that has a length of several hundred thousand pages of semiformalized notations. Is the proof correct? Testing the proof by looking at a page here or there would allow us to find errors but would not yield a watertight conclusion.

The situation would be similar if the mathematician wrote the proof in a formal language, say predicate calculus with set theory notations or any other formalism. Just looking at snippets here and there would still not settle the correctness (and one has to worry that the rigorous formalization actually corresponds with the conjecture to be proven, but let’s ignore this for now).

The formalized situation is, however, different. The formal proof can be fed into a proof validation program (yes, this program must be correct) that encodes logic axioms, inference rules and axioms of set theory and any other theory employed. This time we would have a rigorous confirmation (or not) that the metaproof is correct.

This scenario works because we know, among others, the “ins-and-outs” of logic inference rules. Could this scenario or a similar scenario be applied to software? I see two bottlenecks, which I believe can be addressed by the academic community.

Consider first a scenario whereby a large program, which includes process creation/destruction, asynchronous communication, and so forth, is created manually. This program is automatically correct in the absence of a formal specification of its intended behavior. Hence, assume we also have available a formal description of intended behaviors along the lines of generalized use cases. The key question is now whether we can construct a validator similar to the validator for a mathematician’s proof.

Obviously, such a program must at least know the syntax of the language in which the program has been constructed. In addition, the semantics of that language must be known. Hoare Logic is, unfortunately, not rich enough for the task at hand. It can deal with simple assignments,

conditional expressions, and recursion, but at this point we do not have formal, declarative, off-the-shelf semantics for pointer manipulations, process creation and destruction, and asynchronous communication. Based on the work I have done in this area, I admit this is difficult stuff. However, I believe it is achievable and, given the economic importance of correct software, it seems to me the academic community should see this as a top priority.

This scenario would accommodate software written in low-level languages such as C and C++. It cannot be denied that writing a system in these languages together with the associated formal specification is a tedious affair. Memory management is especially tricky to formalize. The situation improves when a target system is not coded in a programming language but in a design language like UML—let’s call it UML2.

A similar scenario would entail a system to be coded in UML2 while a UML2-based validator would again ascertain the satisfaction of this design against the generalized use cases. Meaning-preserving transformations/compilations would map such a design into traditional languages. Optimizers would ensure efficient code.

This second scenario also needs attention from the academic community: UML does not yet have watertight formal semantics for this scenario to be currently achievable.

This state of affairs is quite surprising. I would have bet my right arm in 1990 that by now we would not have to tinker anymore at the programming language level. I later met people at an OOPSLA workshop who had a prototype implementation of an evaluator for OOAD code. It appears the ruling *engineering* approach to software development has precluded progress toward a rigorous *applied logic* approach.

Do we have to wait until two large passenger jets collide in midair as a result of a software glitch—arousing many lawyers—before we become serious about quality?

I have been asked who I am addressing—let me be more direct. Software academics have played only a secondary role in the last decades in the development of OOA, UML, and Java, for example. They have also not extended Hoare Logic (or done any other theoretical development) so that the validation tools I have described can be built. It seems to me the software theoreticians need to get their collective act together so that effective validation tools for large systems can be constructed. Here, I offer the following challenge to any person or team that:

- Generalizes Hoare Logic so that the declarative semantics of the following operations are captured:
—assignments in nodes of graphs; and

—synchronous and asynchronous interprocess/thread communication, process and thread creation and deletion—these extensions should be compositional; that is, the semantics of a code can be composed from the components (with the exception of deadlock-freeness of multithreaded systems)

- Construct a symbolic code evaluator based on this extended Hoare Logic that generates the appropriate proof obligations in a logic for which theorem provers are available.
- This symbolic evaluator generates the proper proof obligations for annotated versions of:

—Robson’s graph-copying algorithm;
—the Patterson-Wegman linear unification algorithm; and
—the Olympic Torch algorithm (a token is passed on from left to right through the leaf nodes of a tree where the nodes are processes emanating from a root process node).

The last item on this list should ascertain not only theoretical correctness of the declarative semantics, but also pragmatic adequacy. No ad hoc correctness proof is acceptable! **■**

DENNIS DE CHAMPEAUX
(ddc@cutter.rexx.com) is President of
OntoOO, Inc., in San Jose, CA.

© 2002 ACM 0002-0782/02/1100 \$5.00