Usable Verification: Balancing Thinking and Automating

Jeannette M. Wing

Computer Science Department
Carnegie Mellon University
23 October 2010

Here are five ideas on how to make verification more usable by and useful for people. They touch on education, techniques, and process. A common theme is the need to balance the need for human to think and the ability for machines to automate.

Education

Some understanding of formal logic is good. It is necessary and possible.

Verification is based on formal logic. We know from experience that the kind of mathematical thinking needed to specify and verify systems takes facility with logic. Many developers do not have this facility. So to make verification usable, at one extreme we could try to hide the formalism. I don't think that is a good idea. While we may want push-button verification tools, we don't want push-button thinking. We want to make sure that people have enough understanding of formalism to do a sanity check on the results of verification tools and techniques. Achieving this goal means making sure developers are trained and educated in the basics of mathematical logic, including proof techniques and abstract modeling. The challenge is striking a balance between how much people should know and how much we can safely hide from them in our tools.

A good start for usable verification is usable specification. At Carnegie Mellon, we have recently revamped our introductory classes[1]. The revision includes a first-semester first-year class, 15-122 Principles of Imperative Computation[2], that teaches specification as part of the programming process, with a goal of learning how to write reliable software. In the context of a subset of C amenable to verification, students learn to write pre-/post-conditions, assertions, loop invariants, and representation invariants for abstract data types. The specification language is inspired by JML and Spec#. Right now, the compiler causes the specifications to be checked dynamically, but the research goal is to integrate the compiler with a back-end tool, e.g., Boogie, to do checks statically. By mid-October, the instructor (Frank Pfenning) was having the 100+ students in lecture interactively write code to test the representation invariants of red-black trees. [Aside: The second-semester first-year class, 15-150 Principles of Functional Programming, will be a companion course offered for the first time Spring 2011.]

Techniques

Make machines do what they can do best. Provide "Think Now" or "Do Now" guideposts for humans to do what they do best and what verification tools cannot yet do.

[1] R.E. Bryant, K. Sutner, and M.J. Stehlik, "Introductory Computer Science Education at Carnegie Mellon University: A Deans' Perspective," CMU-CS-10-140, August 2010. http://reports-archive.adm.cs.cmu.edu/anon/2010/CMU-CS-10-140.pdf
[2] http://www.cs.cmu.edu/~fp/courses/15122-f10/

Errors reported by verification tools should give semantic insights into the system/property being verified. Some balance is needed where if we use a verification tool and an error is reported, our understanding of that error and how to fix it should be correlated with our understanding of the system being specified and/or property being verified. Take type checking. It is the most successful usable verification technique we have today. Type errors (often) reveal semantic errors. Granted, a type error does not always point to the place in the program where there is a semantic misunderstanding of one's code, but as type systems for programming languages have become more and more sophisticated, there is a closer correlation. For verification (beyond type checking), we should strive for this goal too. Achieving it means providing traceablity mechanisms in our verification tools that scale, compose, and are easy to manage. It means allowing step-by-step verification to play proofs incrementally both backward (debugging—what happened?) and forward (exploration—what if?).

Operationalize verification. Humans are task-oriented and get a sense of satisfaction when completing tasks.

Checklists and tables are useful. People are very good at using mental crutches to remember what to do or what needs to be done. Checklists and tables help operationalize the verification process. Mnenomics help people remember the items in a checklist, suggesting that lists should not be too long and tables, too large. Each item on a checklist or entry in a table should be of the right "mental" size: not so small that the human will find it trivial; and not so large that if an error occurs (see above), the human will be at a loss in fixing the problem. Each item/entry should correlate to a chunk of reasoning the human can do. Achieving this goal takes work from the verification community, e.g., how to "chunkify" verification tasks that correspond to items/entries, but also the cognitive science community, e.g., to figure out how "big" a mental task is and to figure out how to how to structure and organize checklists, tables, and the like so that they do not become unwieldy in size and overwhelming in number.

Collaborative verification. "Socially intelligent computing" where Internet-scale networks of humans and machines work together to solve problems neither can solve alone suggests a completely different angle on usable verification. The PolyMath Project gives evidence that massive collaborations of people can be used to solve math problems in record time. For verification, we can go one step further and combine both human intelligence and machine intelligence. Here we would exploit the collective intelligence of humans and machines, letting humans do what they do best (where machines are not as good at *yet*), and letting machines do what they do best (where humans are not good at).

<u>Process</u>

Integrate with the new way systems are built, services are rolled out. To make verification as much a part of the system development process, we need to be attuned to the newer ways and time scales systems are developed and deployed today. Many of our verification tools and techniques fit in with a presumed systematic development process, be it through stages in a pipeline or iterations in a spiral process, that have on the one hand, amplified, and on the other, ignored.

Considering these two different ends of the spectrum, we should rethink how to develop verification tools and techniques differently for today's world. First, large complex systems, with hardware and software components, that have to interact with the physical world and with people, are now developed by multiple teams, whose members may have different skill sets and domain knowledge, and where teams may be located in different time zones and countries and may even work for different organizations.   Sub-systems and system components are developed in parallel, often asynchronously. The delay in rolling out the Boeing 787 is a classic example of what can happen if specification interfaces mismatch. Transitions across sub-systems are where costly errors occur, more so than in developing the individual system components and integrating systems under the jurisdiction of a single manager/organization.

Second, software services today, especially web services provided by companies like Amazon, Facebook, Google, and Microsoft are launched in short-time scales, measured in days, not months. Companies can now beta test with targeted large-scale subsets of the real user community who provide instantaneous feedback. These companies increasingly are (or should be) concerned with security and privacy issues as well as performance, availability, and reliability. Having all developers in a company build with security and privacy in mind will (I hope) become more common than having only a separate team off to the side whose responsibility is to ensure system trustworthiness.

These newer system development processes, both heavy-weight and light-weight, provide new opportunities for usable verification. Achieving seamless integration for the heavy-weight processes will require evidence-based verification[3] so that certification agencies are doing meaningful semantic checks, not trivial checks (e.g., How many test cases did you run?). Achieving seamless integration for the light-weight processes might very well rely on crowd-sourcing verification (see above) akin to the way testing is already done today.

<u>Summary</u>

For usable verification, by definition humans are in the loop. The challenge is to figure out not just how much machines *can* help humans and vice versa, but also how much machines *should* help.

[3] *Software for Dependable Systems: Sufficient Evidence?,* Daniel Jackson, Martyn Thomas, and Lynette I. Millett, Editors, National Research Council, 2007.