Software Verification and Software Engineering a Practitioner's Perspective

Anthony Hall

anthony@anthonyhall.org

Extended Abstract

The web page for this conference announces

> a "Grand Challenge" of crucial relevance to society: ensuring that the software of the future will be error-free.

According to the Scope and Objectives

> In the end, the conference should work towards the achievement of the long-standing challenge of the Verifying Compiler.

I want to question whether this long-standing challenge is really relevant to the greater goal of achieving trustworthy software. Instead, I suggest that research in verification needs to support a larger effort to improve the software engineering process.

I am a strong advocate – and a practitioner – of formal methods as part of a rigorous software engineering process. But formal methods are very much more than program verification. The goal of verification is to take a given program and to prove that it is correct. The goal of software engineering is quite different: it is to create a program that is verifiably correct. Program verification is neither necessary nor sufficient for software engineering. Its pursuit may even have harmful consequences.

To illustrate, let me take two examples from the field of security.

First, consider the appalling fact that most security flaws are caused by buffer overflow. Why is this appalling? Because there is absolutely no need for anyone, ever, to write a program that contains buffer overflows. That we continue to do so is a reflection our addiction to atrocious languages like C++. There are perfectly good languages around that make it simply impossible to write code that can cause buffer overflows. We should use them. No research is necessary. No proofs are necessary. It's a decidable – indeed solved – problem.

Now let me take a more sophisticated example: cryptography on smart cards. This is a field that seems to be natural for verification. Indeed we could hope to prove, for example, that a cryptographic algorithm needed exponential time to break by brute force and that it was correctly implemented on a smart card. So the smart card would be secure, right? Wrong. Along comes Paul Kocher with a watt meter and breaks the key that you have proved is secure. How did he do that? He did it by bypassing the assumptions you made in your proof. But you never even stated those assumptions: how many proofs contain the following assumption?

Ass1: No-one will measure the power used by the processor

The harmful consequence here is obvious: by doing a proof we have given ourselves a false sense of security. But I think there are more insidious dangers, exemplified even by excellent work in the field. Program verification can all too easily seem to support

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 70–73, 2008.

a process of "Ready – Fire – Aim". For example, one – rightly acclaimed – application of verification is Microsoft's Static Driver Verifier. The web page for SDV says:

> SDV … is designed to be run near the end of the development cycle on drivers that build successfully and are ready for testing.

This is encouraging a wasteful process of guess and debug: one that is almost guaranteed to fail for large and complex software. We have direct experience of this with another static analysis tool, the SPARK Examiner. In principle, the Examiner can be run retrospectively over any SPARK code. In practice, we find that all this proves is that the code has lots of information flow errors.

A far more powerful approach is to design the code with correct information flow in mind, and run the analyser on your design – before you have even created all the package bodies – to eliminate design errors. This is an example of Correctness by Construction: a step-by step process starting from early requirements and progressing through formalisation of the specification, rigorous design, coding in a sound language, static analysis and specification-based testing. Every step of this process is subject to rigorous analysis and of course once one is in the formal domain that analysis can be supported by verification tools.

I suggest that the real opportunity for the verification community is to provide better support for Correctness by Construction. The real Grand Challenge for formal methods is to make Correctness by Construction the mainstream approach to software development. Proponents of this specification-oriented style of formal methods have sometimes underplayed the importance of tools, including verification tools. There is a real opportunity is to bring the specification and verification communities together and to apply the extremely sophisticated tools coming from the verification community to the systematic construction of software. Here are some of the big issues that need to be addressed.

1. Early requirements. How can we add rigour to scenarios, use cases and other techniques that are essential for communicating with stakeholders? How can we formalise domain knowledge? What about the problem of "unbounded relevance" – how do we know what assumptions to make?
2. Specification languages. How can we have rich and expressive specification languages and at the same time have tractable proofs? How can we make specification languages and reasoning about them more accessible?
3. Design notations. How can we express the multiple dimensions of design? What are the refinement rules when we are using a distributed design, working with COTS, building on a database…?
4. Concurrency. How can we express concurrency properties in a compositional way? How can we turn a black-box sequential specification into a concurrent implementation?
5. Testing. How can we develop efficient test cases from our requirements? How can we relate test effectiveness and proof coverage?
6. Proof. How can we choose what to prove? How can we make proof accessible? How can we use proof for finding errors?

This challenge is both easier and harder than pure verification. It's easier, because a step by step process reduces the semantic gap to be bridged and makes verification feasible. It's harder, because we have to face up to the difficulties of the real world, the problem of imperfect knowledge and the difficulty of reconciling rigour and creativity. The reward is that we really could turn software into engineering.

A Discussion on Anthony Hall's Presentation

Bertrand Meyer

I want to take issue with one of your slides, which, I think, detracts from the rest of your presentation. This is the one where you cited parts of the Standish report. Actually the first comment is that there have been new versions of the Standish report which give a quite different picture from the original 1994 report that everyone quotes. But more specifically the overall view that technology does not matter seems a bit weak. I think that if the time were 1820, you could make a very serious case about sailing boats not being fast and reliable enough, that it is all a managerial problem and it does not matter whether to improve the sailing technology or not. This ignores the possibility of a technology breakthrough, which redefines the problem.

Anthony Hall

I do not remember saying anything about technology not mattering, quite the contrary. What I have said was that if we wanted to attack these problems, we had better attack the problems where they are, not where we would like them to be. What I said is, technology as here is probably the one bit, by lurking in there (*pointing at a slide*), perhaps 10% of that slide is maybe errors that can be covered by proofs of correctness. This covers use of tools or people learning new methods and so on. I certainly did not mean to say technology is not important. What I mean is, technology better be applied to the 49% that matters rather than the 10% or 15% that occasionally cause failures.

Richard Bornat

I have lots of objections to what you said. I shall not object to every slide. I would just like to say that you could have given that talk 40 years ago, and indeed, talks like that were given 40 years ago, simply by replacing the words "program verification" by the word "compiling". The fact is, new program verification is but old compiling writ large. And in the old days, you would have to say: the problem is not what language you write it in, the problem is the design. And they were right, and they were wrong. The fact is, compiling has had a massive effect on all of the problems you dealt with. And compiling has meant that the problem of software evolution—although not overcome, and the proportions are still the same—is yet dented. We can more rapidly recompile our design than we could. And new program verification is just another step down the same road. Yes, it does not solve the problem, but it does help.

Now, I want to argue with you about your Lakatos thing, but I shall not do that now. I just want to point out to you that please be careful when you quote Hardy!

Hardy, you remember, was the author of your quote that "proof is just gas". Now, Hardy was a sucker for a well-turned phrase, as I am. But his other most famous quote was that "I am absolutely sure that everything I have done in my life has been completely useless." Hardy was absolutely wrong, because his work is the foundation of RSA. And he was absolutely wrong about that, and he is absolutely wrong about the gas, too. And I think, you would be very, very careful about building an argument onto Hardy, Sir, it will swallow you up!

Anthony Hall

Well, I have to say, I had some doubts about quoting Hardy. However, both of those statements, of course, are propaganda statements. His statement about that he is absolutely certain that everything is useless, was a statement of propaganda. It appears in *A Mathematician's Apology*.

It was a statement which he would *like* to be true. He would like to be pure and isolated from the real world.

Willem-Paul de Roever (*laughing*)

I would like to support the speaker after these attacks, because it is laughable. Well, the point is, we all know mathematical proofs, and I have given extremely many methodical proofs in my lifetime. But if I go to John Rushby and Shankar here, we all know that when you try to formalize any mathematical proof it is full of holes, there are all sorts of mistakes, but it is very seldom the case that the theorem is completely incorrect. So the point is that it is a sketch for communication, and the poor sucker who has to formalize it in PVS or Isabelle is very proud that he found so many errors. But at the end, he did not find an error in the theorem. So, I would like to have Rushby and Shankar cooperate on this, because I think that is relevant.

What I myself thought is, logically speaking, you do not give place to bottom-up development, and you are so much experienced in developing software, so you certainly know where to put bottom-up in this picture. That is, what I wanted to say.