Software Verification and Software Engineering

A Practitioner's Perspective

Anthony Hall
`anthony@anthonyhall.org`

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 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.