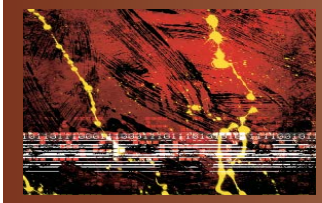
Verified Software: A Grand Challenge

Cliff Jones, University of Newcastle
Peter O'Hearn, University of London
Jim Woodcock, University of York



Given the right computer-based tools, the use of formal methods could become widespread and transform software engineering.

A common joke among computer users is that, for all its extensive documentation, software often behaves in completely surprising ways. The need to reduce a problem to a set of rules that the computer must blindly follow is what makes programming hard. Components interact and interfere, undesirable properties emerge, and systems fail to satisfy their users' needs.

The inherent difficulty of developing software is responsible for this unpredictability: It's hard to define requirements, anticipate interactions, and accommodate new functionality. Documentation includes lots of text, pictures, and diagrams that are often imprecise and ambiguous. Important information is often hidden in a mass of irrelevant detail, and design mistakes are often discovered too late—making it expensive or even impossible to correct them. It's a tribute to the skill of software engineers that systems work at all.

But there is another way. Many industry practitioners and university researchers believe formal methods offer a practical alternative to producing software—even noncritical software—and that this may turn out to be the cheapest way to do it as well.

Given the right computer-based tools, the use of formal methods could become widespread and transform software engineering. The computer science community recently committed itself to making verified software a reality within the next 15 to 20 years when representatives met in Zurich in 2005 to discuss an international grand challenge on verification (vstte.ethz.ch/report.html).

WHY NOW?

Software has been the subject of academic research for 70 years now. Fundamental results have been studied in theoretical computer science and have been exploited in software development theories studied as formal methods of software engineering.

A scientific theory has two purposes—to explain and predict. Formal methods are intended to explain software, both to the user and the developer. They produce precise documentation, structured and presented at an appropriate level of abstraction. By being amenable to mathematical analysis, formal methods are also intended to predict software behavior.

Today, universities around the world routinely teach formal methods. Googling "formal methods education" returns links to repositories, a virtual library, and surveys of hundreds of different courses with online resources. Many key texts can be freely downloaded.

Software developers have successfully used formal methods in avionics and other safety-critical computing domains. But, you might ask, what can formal methods do for the average user? The truth be told, even mainstream commercial software developers are beginning to use such formal logic behind the scenes in the form of program analysis and model-checking tools.

Microsoft, the world's biggest software developer, provides a good example. Most people use the Windows operating system despite its erstwhile reputation for frequent crashes. However, a closer examination reveals that faulty device drivers were usually responsible for the crashes, not the OS itself. As a result, Microsoft responded by developing its own model checker—the *static driver verifier*—to check drivers for conformance against a mathematical specification. If a driver fails the SDV test, it might have a bug.

SDV illustrates Kant's dictum that "there's nothing so practical as a good theory." SDV exploits sophisticated theory hidden from the user. This encapsulation is so complete that commercial software developers now use SDV outside Microsoft.

SDV's success relies on a device driver's conceptual simplicity: Although a driver might appear as complicated code, there is a simple overall structure that can be abstracted. In par-

ticular, device drivers have no concurrency.

Microsoft clearly recognizes the importance of software verification. As Bill Gates observed in a keynote address at the Windows Hardware Engineering Conference in 2002 in Seattle:

> Software verification has been the Holy Grail of computer science for many decades, but now in some key areas, for example, driver verification, we're building tools that can do actual proof about the software and how it works, in order to guarantee reliability.

Considerable experience has developed from the successful use of formal methods in safety-critical computing, and a new wave of tools can now shield users from technical subtlety. Adding to that the significant advances in proof technology such as SAT solvers and combined decision procedures, the time now seems ripe for a concerted push for software verification.

Considerable activity is already under way in numerous specific projects worldwide, but that activity is not (yet) coordinated.

VISION OF THE FUTURE

The computer science community seeks to create computer systems that justify the trust that society increasingly places in them. Earning this trust will require a substantial reduction in the current high cost of programming error incurred during the design, development, testing, installation, maintenance, evolution, and retirement of computer software.

An important technical contribution to realizing this vision is a toolset that automatically proves that a program meets its given specifications. This has been a challenge for computing research for more than 30 years, but the current state of the art gives grounds for hope in seeing substantial progress.

A striking difference between software and other products is that software is generally sold with little or no meaningful warranty; most of the time only the physical CD is guaranteed. In contrast, if you ask your mechanic to supply you with snow tires for your vehicle, there's an implied warranty of fitness for purpose, if not a written or verbal guarantee. Could we ever expect software to come with such warranties? In our vision of the future, we would expect exactly that.

In our vision, software comes with a clean architecture that encourages the user to create a sound mental model.

There's a modern fairytale that involves your least favorite—but essential—software application: The fairy offers you the choice between correcting the bugs or making the application more predictable and easier to use. These two things are not the same, and many of us would choose the latter.

In our vision, software comes with a clean architecture that encourages the user to create a sound mental model. The architecture is made more evident by components and their interfaces being clearly documented, which in turn makes it easier to support maintenance, reuse, and evolution.

FORMAL METHODS: COMPLEMENTARY APPROACHES

There are two approaches to using formal methods, *post facto* and *correctness by construction* (C×C), each providing different benefits.

Some airline reservation systems, for example, rely on large quantities of legacy code written 35 years ago. These legacy systems would benefit from architecture improvements, and they would be more reliable if we debugged their null-pointer dereferences and buffer overflows. Similarly, we could make a huge contribution to open source software, such as the Apache Web server, by using formal methods post facto.

The C×C approach, on the other hand, uses formal methods during software production rather than afterward. C×C can be highly cost-effective because it significantly reduces testing and reworking while yielding low-defect software. And with the confidence this brings, developers can issue product warranties. Praxis, for example, has offered a 10-year warranty on some software, which greatly reduces its customers' support costs. We envision all software providing the same kind of guarantee.

C×C and post facto are actually close siblings. A C×C effort will eventually result in code, and assertions on programming values (the same kinds handled by post facto methods) will be encountered. Consequently, both approaches share many of the same techniques, including proof tools, verification condition generators, and invariant synthesizers.

VERIFICATION CHALLENGE

Tony Hoare initially suggested the Verification Challenge as an effort to create a toolset that would—as automatically as possible—guarantee that programs meet given specifications.

The toolset should be applicable to a wide range of software and, ideally, to itself. Hoare conceived the Challenge in a broad sense, which would include testing and resolving the problems of getting the specifications right.

Although we can never bridge the formal/informal gap, having a full-fledged C×C system would allow feedback to flow from program to design, so that we could precisely track unwanted behaviors to a suitably formalized design.

Tools that crash are frustrating, but even worse are tools that claim the software is "verified" when it actually has bugs. An ideal verifier tool will be warranted against letting a class of errors occur in any system created using the tool. Achieving this is impossible short of a full C×C support system in which the tool includes its own

precise specification of what its users can expect from it. This is deliberately ambitious but, we believe, it is achievable within 10 to 15 years.

This goal might appear introverted, or just "navel gazing," but we believe otherwise: The community that builds tools to support formal approaches to software development must "take its own medicine" if it is to be trusted. Further, applying a tool or formalism to itself is a time-honored method of implementing generality.

Such an endeavor would require the designers to use methods that scale up and encourage usability as an explicit design goal. We prefer a multiuser tool because this would force at least some aspects of concurrency control into the tool as well as the methods used in its creation. But perhaps we should stop short of this requirement, as it would be a significant achievement to create a single-use "warranted verifier."

As part of the Verification Challenge workshop held at the 2005 International Federation for Information Processing working conference in Zurich, verification and formal methods experts, tools researchers, and theoreticians proposed many application areas for such research, including telecommunications, Web servers, and avionics. Everyone had a favorite target, but a clear—or clearly fundamental—target that involves verification tools has yet to be determined.

A finalized Verification Challenge is still being developed, and future workshops are being planned to advance these discussions. We invite you to become involved. Visit www.fmnet.info/vsr-net and vstte.ethz.ch for more details.

Cliff Jones *is a professor in the Department of Computer Science at the University of Newcastle. Contact him at cliff.jones@ncl.ac.uk.*

Peter O'Hearn *is a professor in the Department of Computer Science at the University of London, Queen Mary College. Contact him at ohearn@dcs.qmul.ac.uk.*

Jim Woodcock *is the Anniversary Professor of Software Engineering in the Department of Computer Science, University of York. Contact him at jim@cs.york.ac.uk.*

Editor: Michael G. Hinchey, NASA Software Engineering Laboratory at NASA Goddard Space Flight Center and Loyola College in Maryland; michael.g.hinchey@nasa.gov