COMMUNICATIONS OF THE ACM
TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS

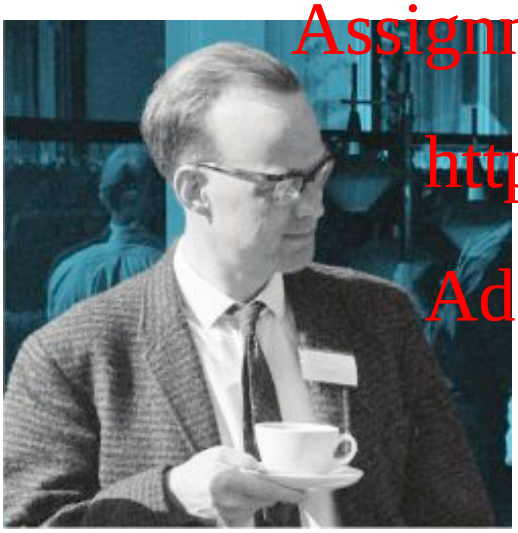
VIEWPOINTS

Retrospective: An Axiomatic Basis for Computer Programming

C.A.R. Hoare revisits his past *Communications* article on the axiomatic approach to programming and uses it as a touchstone for the future.

C.A.R. Hoare

Communications of the ACM
Vol. 52 No. 10, Pages 30-32
10.1145/1562764.1562779



C.A.R. Hoare attending the NATO Software Engineering Techniques Conference in 1969. Credit: Robert M. McClure

This month marks the 40th anniversary of the publication of the first article I wrote as an academic.[a] I have been invited to give my personal view of the advances that have been made in the subject since then, and the further advances that remain to be made. Which of them did I expect, and which of them surprised me?

Retrospective (1969–1999)

My first job (1960–1968) was in the computer industry; and my first major project was to lead a team that implemented an early compiler for ALGOL 60. Our compiler was directly structured on the syntax of the language, so elegantly and so rigorously formalized as a context-free language. But the semantics of the language was even more important, and that was left informal in the language definition. It occurred to me that an elegant formalization might consist of a collection of axioms, similar to those introduced by Euclid to formalize the science of land measurement. My hope was to find axioms that would be strong enough to enable programmers to discharge their responsibility to write correct and efficient programs. Yet I wanted them to be weak enough to permit a variety of efficient implementation strategies, suited to the particular characteristics of the widely varying hardware architectures prevalent at the time.

I expected that research into the axiomatic method would occupy me for my entire working life; and I expected that its results would not find widespread practical application in industry until after I reached retirement age. These expectations led me in 1968 to move from an industrial to an academic career. And when I retired in 1999, both the positive and the negative expectations had been entirely fulfilled.

The main attraction of the axiomatic method was its potential provision of an objective criterion of the quality of a programming language, and the ease with which programmers could use it. For this reason, I appealed to academic researchers engaged in programming language design to help me in the research. The latest response comes from hardware designers, who are using axioms in anger (and for the same reasons as given above) to define the properties of modern multicore chips with weak memory consistency.

One thing I got spectacularly wrong. I could see that programs were getting larger, and I thought that testing would be an increasingly ineffective way of removing errors from them. I did not realize that the success of tests is that they test the programmer, not the program. Rigorous testing regimes rapidly persuade error-prone programmers (like me) to remove themselves from the profession. Failure in test immediately punishes any lapse in programming concentration, and (just as important) the failure count enables implementers to resist management pressure for premature delivery of unreliable code. The experience, judgment, and intuition of programmers who have survived the rigors of testing are what make programs of the present day useful, efficient, and (nearly) correct. Formal methods for achieving correctness must support the intuitive judgment of programmers, not replace it.

My basic mistake was to set up proof in opposition to testing, where in fact both of them are valuable and mutually supportive ways of accumulating evidence of the correctness and serviceability of programs. As in other branches of engineering, it is the responsibility of the individual software engineer to use all available and practicable methods, in a combination adapted to the needs of a particular project, product, client, or environment. The best contribution of the scientific researcher is to extend and improve the methods available to the engineer, and to provide convincing evidence of their range of applicability. Any more direct advocacy of personal research results actually excites resistance from the engineer.

Progress (1999–2009)

On retirement from University, I accepted a job offer from Microsoft Research in Cambridge

(England). I was surprised to discover that assertions, sprinkled more or less liberally in the program text, were used in development practice, not to prove correctness of programs, but rather to help detect and diagnose programming errors. They are evaluated at runtime during overnight tests, and indicate the occurrence of any error as close as possible to the place in the program where it actually occurred. The more expensive assertions were removed from customer code before delivery. More recently, the use of assertions as contracts between one module of program and another has been incorporated in Microsoft implementations of standard programming languages. This is just one example of the use of formal methods in debugging, long before it becomes possible to use them in proof of correctness.

I did not realize that the success of tests is that they test the programmer, not the program.

In 1969, my proof rules for programs were devised to extract easily from a well-asserted program the mathematical 'verification conditions', the proof of which is required to establish program correctness. I expected that these conditions would be proved by the reasoning methods of standard logic, on the basis of standard axioms and theories of discrete mathematics. What has happened in recent years is exactly the opposite of this, and even more interesting. New branches of applied discrete mathematics have been developed to formalize the programming concepts that have been introduced since 1969 into standard programming languages (for example, objects, classes, heaps, pointers). New forms of algebra have been discovered for application to distributed, concurrent, and communicating processes. New forms of modal logic and abstract domains, with carefully restricted expressive power, have been invented to simplify human and mechanical reasoning about programs. They include the dynamic logic of actions, temporal logic, linear logic, and separation logic. Some of these theories are now being reused in the study of computational biology, genetics, and sociology.

Equally spectacular (and to me unexpected) progress has been made in the automation of logical and mathematical proof. Part of this is due to Moore's Law. Since 1969, we have seen steady exponential improvements in computer capacity, speed, and cost, from megabytes to gigabytes, and from megahertz to gigahertz, and from megabucks to kilobucks. There has been also at least a thousand-fold increase in the efficiency of algorithms for proof discovery and counterexample (test case) generation. Crudely multiplying these factors, a trillion-fold improvement has brought us over a tipping point, at which it has become easier (and certainly more reliable) for a researcher in verification to use the available proof tools than not to do so. There is a prospect that the activities of a scientific user community will give back to the tool-

builders a wealth of experience, together with realistic experimental and competition material, leading to yet further improvements of the tools.

For many years I used to speculate about the eventual way in which the results of research into verification might reach practical application. A general belief was that some accident or series of accidents involving loss of life, perhaps followed by an expensive suit for damages, would persuade software managers to consider the merits of program verification.

This never happened. When a bug occurred, like the one that crashed the maiden flight of the Ariane V spacecraft in 1996, the first response of the manager was to intensify the test regimes, on the reasonable grounds that if the erroneous code had been exercised on test, it would have been easily corrected before launch. And if the issue ever came to court, the defense of 'state-of-the-art' practice would always prevail. It was clearly a mistake to try to frighten people into changing their ways. Far more effective is the incentive of reduction in cost. A recent report from the U.S. Department of Commerce has suggested that the cost of programming error to the world economy, as measured in tens of billions of dollars per year, most of it falling (in small but frequent doses) on the users of software rather than on the producers.

The phenomenon that triggered interest in software verification from the software industry was totally unpredicted and unpredictable. It was the attack of the hacker, leading to an occasional shutdown of worldwide commercial activity, costing an estimated $4 billion on each occasion. A hacker exploits vulnerabilities in code that no reasonable test strategy could ever remove (perhaps by provoking race conditions, or even bringing dead code cunningly to life). The only way to reach these vulnerabilities is by automatic analysis of the text of the program itself. And it is much cheaper, whenever possible, to base the analysis on mathematical proof, rather than to deal individually with a flood of false alarms. In the interests of security and safety, other industries (automobile, electronics, aerospace) are also pioneering the use of formal tools for programming. There is now ample scope for employment of formal methods researchers in applied industrial research.

Prospective (2009–)

In 1969, I was afraid industrial research would dispose such vastly superior resources that the academic researcher would be well advised to withdraw from competition and move to a new area of research. But again, I was wrong. Pure academic research and applied industrial research are complementary, and should be pursued concurrently and in collaboration. The

goal of industrial research is (and should always be) to pluck the 'low-hanging fruit'; that is, to solve the easiest parts of the most prevalent problems, in the particular circumstances of here and now. But the goal of the pure research scientist is exactly the opposite: it is to construct the most general theories, covering the widest possible range of phenomena, and to seek certainty of knowledge that will endure for future generations. It is to avoid the compromises so essential to engineering, and to seek ideals like accuracy of measurement, purity of materials, and correctness of programs, far beyond the current perceived needs of industry or popularity in the market-place. For this reason, it is only scientific research that can prepare mankind for the unknown unknowns of the forever uncertain future.

The phenomenon that triggered interest in software verification from the software industry was totally unpredicted and unpredictable.

So I believe there is now a better scope than ever for pure research in computer science. The research must be motivated by curiosity about the fundamental principles of computer programming, and the desire to answer the basic questions common to all branches of science: what does this program do; how does it work; why does it work; and what is the evidence for believing the answers to all these questions? We know in principle how to answer them. It is the specifications that describes what a program does; it is assertions and other internal interface contracts between component modules that explain how it works; it is programming language semantics that explains why it works; and it is mathematical and logical proof, nowadays constructed and checked by computer, that ensures mutual consistency of specifications, interfaces, programs, and their implementations.

There are grounds for hope that progress in basic research will be much faster than in the early days. I have already described the vastly broader theories that have been proposed to understand the concepts of modern programming. I have welcomed the enormous increase in the power of automated tools for proof. The remaining opportunity and obligation for the scientist is to conduct convincing experiments, to check whether the tools, and the theories on which they are based, are adequate to cover the vast range of programs, design patterns, languages, and applications of today's computers. Such experiments will often be the rational reengineering of existing realistic applications. Experience gained in the experiments is expected to lead to revisions and improvements in the tools, and in the theories on which the tools were based. Scientific rivalry between experimenters and between tool builders can thereby lead to an exponential growth in the capabilities of the tools and their fitness to

purpose. The knowledge and understanding gained in worldwide long-term research will guide the evolution of sophisticated design automation tools for software, to match the design automation tools routinely available to engineers of other disciplines.

The End

No exponential growth can continue forever. I hope progress in verification will not slow down until our programming theories and tools are adequate for all existing applications of computers, and for supporting the continuing stream of innovations that computers make possible in all aspects of modern life. By that time, I hope the phenomenon of programming error will be reduced to insignificance: computer programming will be recognized as the most reliable of engineering disciplines, and computer programs will be considered the most reliable components in any system that includes them.

Even then, verification will not be a panacea. Verification technology can only work against errors that have been accurately specified, with as much accuracy and attention to detail as all other aspects of the programming task. There will always be a limit at which the engineer judges that the cost of such specification is greater than the benefit that could be obtained from it; and that testing will be adequate for the purpose, and cheaper. Finally, verification cannot protect against errors in the specification itself. All these limits can be freely acknowledged by the scientist, with no reduction in enthusiasm for pushing back the limits as far as they will go.

Author

C.A.R. Hoare (thoare@microsoft.com) is a principal researcher at Microsoft Research in Cambridge, U.K., and Emeritus Professor of Computing at Oxford University.

Footnotes

a. Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

DOI: http://doi.acm.org/10.1145/1562764.1562779

Figures

Figure. C.A.R. Hoare attending the NATO Software Engineering Techniques Conference in 1969.