THE VERIFYING COMPILER: A GRAND CHALLENGE FOR COMPUTING RESEARCH OF THE 21ST CENTURY

by

Sir Tony Hoare

18 March 2004

The Verifying Compiler: a Grand Challenge for computing research of the 21st century

Tony Hoare

Summary. The questions "How does it work?" and "Why does it work" have always inspired the curiosity of everyone engaged in basic science or engineering research. And in modern times, the answers to these questions are incorporated into sophisticated computer programs, design tools that have contributed major advances in the quality and efficiency and reliability of all the products of our technological society. Often in ways that were inconceivable to those who engaged in the original research.

How do computer programs work? And why? Nobody really knows for certain. Or at least, nobody is certain enough to guarantee that they are going to work correctly. And for large parts of programs in use today, it is freely admitted that there is nobody who understands how they work, let alone why. And even what little we believe we understand may be inadequate, inappropriate, or even downright wrong. Researchers in the science of programming have put forward a number of good theories, conjectures and hypotheses. And in engineering practice there are many programs of all sizes in constant use that actually do their job pretty well.

In this, they are like the Sainte Chapelle in Paris and other medieval cathedrals and bridges; amazingly, they still stand, though they were built long before our modern theories of civil engineering could explain how and why. And so it is in programming. To build the link between the theory of programming and the products of software engineering practice is still a grand challenge for scientific research in computing; the development of a verifying compiler is an essential tool and target for this research; it will make the results of the research available to software engineers of the future, and so contribute to the quality and reliability of all the programs that they produce.

Background. The history of science records many examples where the pursuit of a scientific Grand Challenge has triggered a major breakthrough in scientific discovery, one that could have been achieved in no other way. A recent successful example has been the Human Genome project. I have taken that example to illustrate the defining characteristics of a Grand Challenge project. They are its long time-scales (fifteen years planned, starting 1990), its wide international participation (seven nations to start with), its ideals of rapid and open publication (the Bermuda Agreement), and its quest to answer basic questions that lie at the foundations of an entire scientific or engineering discipline. What could be more fundamental to human biology or medicine than the discovery of nature's blueprint for the entire human being?

Are there any projects in computer science that share these characteristics? Of course, many achievements like the World Wide Web have emerged quite without the stimulus of a Grand Challenge. A well-known computing Grand Challenge dating from the nineteen fifties was the construction of a computer that plays chess at international championship standard; and that challenge was met on May 11 1997, when the IBM Deep Blue special-purpose computer defeated the reigning world chess champion Garry Kasparov, in an open match. A challenge that is still outstanding is the efficient solution of problems like the travelling salesman problem, or (more likely) the proof that there isn't one. Are there any more such challenges for computing research?

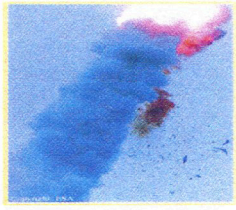
In this lecture, I will concentrate on one such challenge. It is targeted at the disciplines of software engineering and the science of programming. The challenge consists in the construction, evaluation and deployment of a high level programming language compiler, one that verifies the correctness of each program that it compiles. If this project is successful, systematic use of the verifying compiler could avert many of the kinds of programming error that afflict the world today. And the benefit will accrue from pure scientific research, directed at the basic questions that lie at the foundations of our scientific discipline: how do computer programs work and why?

Introduction. If you tell the man on the Clapham bus how the engine of the bus works, I doubt whether he will care. The people who care are the engineers who designed the engine, students of mechanics in colleges and universities, and researchers in automobile engineering; you would be justifiably shocked if they did not care how an engine works. The research scientist is also interested in the question why they work. What are the basic principles and laws which apply to the processes of internal combustion? The answer is sought by formalising scientific theories and checking them against

data obtained by large-scale experiments. These days, the experiments are increasingly conducted and controlled by computer, and the observed data are stored and analysed for their conformity to theory by sophisticated computer programs.

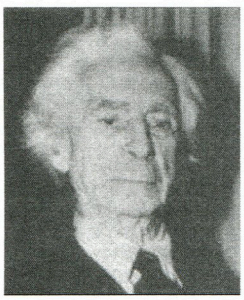
How do computer programs work? And why? These are the questions that drive research in software engineering and the science of programming. And the answers are beginning to be incorporated in the design of software engineering tools which help computer programmers in their daily tasks. There are program analysers, code generators, development environments, type checkers, simulators, test harnesses and test case generators or selectors; these are in addition to the compiler for the programming language itself.

The verifying compiler will be built from a combination of the technologies incorporated in all those tools. Its goal is to guarantee that every program that it compiles will always work in the manner specified. For example, the compiled program when executed will not hang, when mistreated it will not crash, when maliciously attacked, it will not succumb, and under no circumstances will it ever crash the car or rocket that it is helping to drive or fly.



Here is an example of what can happen. On the 4th of June 1996, at its test site in South America, the newly enhanced Ariane V space rocket flight 501 went out of control some forty seconds after launch, and had to be destroyed. The fault was soon traced to a tiny error in a part of the flight control software that had been frequently tested on previous flights of Ariane IV. The loss was estimated at half a billion dollars, and a year's delay to the European space programme. Since then, all the control programs for Ariane are subjected to the best available verification technology.

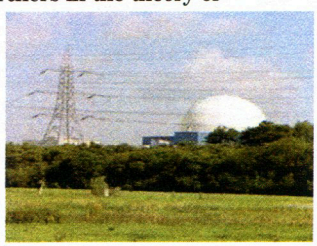
In principle, a verifying compiler will guarantee that the program will do exactly what it is intended to do, insofar as the programmer has taken the trouble to specify to the compiler exactly the intention of that program. The check of correctness is made not by testing the program on a few hundred artificially constructed test cases, or even on thousands or even on many millions. The check is made by methods of mathematical proof, the same methods which have made mathematics the most convincingly reliable of all branches of learning. Like the proof of a mathematical theorem, the guarantee offered by the verifying compiler applies for all possible or conceivable test cases, and so will always work in practice, even if not a single test were conducted. And the proof will itself be checked by computer, using proof techniques that are taken from standard mathematical practice; the precise formalisation of mathematical reasoning can be traced back to the discoveries of Aristotle, Euclid, Leibniz, Frege and Russell. Bertrand Russell was a British philosopher who in his later years became famous as an antinuclear protester (see picture, left). When young, he worked with Alfred Whitehead on a complete formalisation of the proof procedures of mathematics, with the aim of showing how every mathematical proof can be checked by a mechanical procedure; the research was completed long before anyone new that computers would be invented both to need it and to exploit it.



The Challenge. The ideal of program verification by mathematical proof has a long history. The British founder of Computer Science, Alan Turing (see picture, right), described how to prove programs in a talk on Checking a Large Routine, which he delivered in Cambridge in 1948. He described the concept of an assertion. This is a conditional expression which describes some property of the program's data which is expected to be true whenever the program reaches the point at which it is written. Assertions are widely used today as test oracles; they are evaluated at run time to detect errors as close as possible to their place of occurrence; and they still provide the foundation of all attempts to prove the correctness of programs.

The idea of using computers to generate or check the proofs was put forward by the American computer scientist John McCarthy in 1962. The idea of a verifying compiler was put forward by Bob Floyd in an applied mathematical journal in 1967. Since reading Floyd's paper in 1968, I have devoted a large part of my research career to extending Floyd's technology to cover more features of standard programming practice. And there are many others working today to fill holes in our theoretical understanding of commonly used programming techniques like object orientation and concurrency. But the results of this research are almost wholly theoretical: its relevance and applicability has not yet been checked against real programs in widespread or critical use today.

That is why the first application of the verifying compiler, even while it is still being designed, written and developed, will be as an experimental tool, to check the scientific theories that have been put forward by researchers in the theory of programming against existing programs that have already been written by good programmers. Many of these programs have been subject to continuous evolution to meet changed or newly recognised needs. With each change, the program becomes less and less comprehensible. Do the theories really apply to programs that actually exist? Programs that are actually in daily use by millions of owners of personal computers? Programs that provide and administer our communication networks? Programs that stretch to twenty million lines of code? Programs that control our nuclear power stations, flood gates, and



cars and aeroplanes? The programs that control the nuclear reactors at Sizewell have to be exceptionally reliable. It was the intention that after an incident, control of the reactor would be taken over by the program, and that the human operators would not be allowed to interfere for at least half an hour. That is why the program was subjected to the best available manual verification technology available.

It is this kind of program that will provide the experimental data to test our theories of programming, and their application on a grand scale. The first task of the experimenter will be to understand the intentions of each program, and incorporate this understanding in formal assertions and other forms of declaration and specification; these will be input together with the program to the latest prototype of the verifying compiler, to make sure that they really do apply.

This is the really hard work, but no harder than similar work now in progress on the interpretation of the three billion base pairs of the human genome – the same sort of length as the software used in a typical pc today. At present after fifteen years of hard work, the human genome is about as comprehensible as a binary memory dump of an unknown computer of unknown architecture. The current phase of the project is to discover what it means and how it works, and scientists throughout the world are engaged in this analysis. The results of each new analysis are incorporated into the human genome data base, to serve as a basis for yet more advanced analyses, devised by later scientists to check more and more aspects of the functioning genome.

In the case of computer programs, the first task of each analysis is to discover the assertions on which the working of the program is based, and to check them by submitting them to the currently available verification tools. But to begin with the assertions will often not pass the check. Usually this will be because the assertions are just wrong and must be adjusted. In other cases the assertions will be inadequate to complete the proof, and will need to be strengthened before they pass the check. Sometimes the theorem proving capability of the compiler will be too weak, and will needs enhancement. Fortunately, the assertions can be left to be tested in the running code, so that confidence in their correctness can be established even before the ultimate verification technology is available. Just occasionally, the failure to check the assertion will be because the theory is wrong and must be corrected before it is used again. All of these setbacks, especially the last, will lead to advances in the science and improvements in the technology.

But sometimes failure of the proof of correctness will actually reveal a bug in the software. If the software is currently in use, the bug may be corrected, to the benefit of all who use it – millions of people in some cases. There are not many branches of science that can offer such widespread benefits so quickly from the results of new research.

The benefits. The main benefits of the success of the verifying compiler project will only be reaped long after the research is complete. Only then will the programming profession have the scientific evidence that the verifying compiler is adequately powerful to deal with kind of programs which they actually want to write, and which their clients actually want to use. They will have the confidence to begin to write the specifications and declarations and assertions that define the correctness of their programs at the same time that they write the programs themselves. (They may even write parts of the specification in advance, a practice which is believed to contribute to other aspects of quality in software.) With the aid of the assertions, the verifying compiler will be able to detect inconsistencies and errors in the program immediately on compilation, even before the program is run on its first test case. Of course, testing will still be required to make sure that nothing else has gone wrong; and it will still be required to test important aspects of dependability and serviceability which are impossible or inappropriate for formal specification as assertions. But the cost and duration of testing, which at present amounts to considerably more than half the cost of program development, should be reduced to considerably less than half the cost.

The value of the verifying compiler will extend far beyond its contribution to the initial development of a new program. It will be even greater when the program undergoes adaptations and enhancements to meet new and changing needs. Because the assertions reliably describe not only how the program works but also why, changes to the program can be planned with far greater confidence. And if the change turns out to be incompatible with the assumptions and aims and design disciplines of the original program, the very next run of the verifying compiler will detect this immediately, even before the first test is conducted.
It is not the task of a scientist to estimate the financial benefits of a new technology. Fortunately this has been done for us by the US Department of Commerce, in a report entitled "The Economic Impacts of Inadequate Infrastructure for Software Testing", commissioned and published by the National Institute of Standards and Technology (NIST), Program Office for Strategic Planning and Economic Analysis Group, US Department of Commerce, May 2002, Planning Report 02-03, from which I quote:

> *Based on the software developer and user surveys, the national annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion (excluding the costs associated with errors in mission critical software). Over half these costs are borne by software users in the form of error avoidance and mitigation activities. The remaining costs are borne by software developers ...*

These costs apply to the US economy alone; they should roughly be doubled to cover the economy of the world as a whole. And they will surely double again in the next fifteen years as software extends yet further into our personal, commercial, social and political lives. Let's say in total a hundred billion dollars per year.

But it would be completely wrong for a scientist to speculate how much of this expenditure might be saved by use of a verifying compiler. Indeed, actual exploitation of the verifying compiler, even when available, may be delayed indefinitely by many extraneous factors: for example, the absence of legal liability for software, the inertia of the programming profession, its restricted educational background, and the heavy burden of the large legacy programs that it is called upon to maintain. But these are not the primary concerns of scientists; our job is complete when we have provided the best technology that we can, and when we have objectively evaluated it over its range of application. It is not the scientists' job to persuade people to exploit such an opportunity; and if we make it so, we lose one of our most valuable scientific attributes, our scientific objectivity. That is why a Grand Challenge must be 'blue sky' research, wholly directed to scientific goals. Above all, we must avoid any hint of a promise of competitive commercial advantage accruing to a single company or nation.

Furthermore, even when widely used, the verifying compiler is not going to solve all problems arising from computer or software error. Program specifications may be inadequate or inappropriate to real needs, especially when individually desirable features interact; awkward human-computer interfaces may tempt the user into error; hardware and communications can fail; and no compiler can check that the correct program has been loaded into the right computer before it is run. These are problems that can and must be solved by other measures, some of them administrative, and many of them also based on the results of scientific research in other specialist branches of computing. A Grand Challenge will always be of minority appeal among scientists: most good scientists will prefer to protect their freedom and individual right to pursue whatever new ideas will occur to them on the spur of the moment. In fact, it would very dangerous if a Grand Challenge were to divert more than a small fraction of the available pool of scientific talent in any specialist branch of science.

The risk of failure. This is because of the strong prospect that the verifying compiler project as a whole may fail. Without that prospect, the challenge would not be Grand enough. Certainly, all previous attempts to construct a verifying compiler have failed. Before reviving the challenge, it is essential to understand the reasons for previous failures, and provide some grounds for confidence that they can be overcome. Otherwise, you should dismiss this idea as just another incarnation of the perpetual perpetual motion machine.

The first attempts to construct a Verifying Compiler date back to the early 1970s, when it was the topic of research for a few clever and courageous doctoral students. At that time, computers were about a thousand times slower than they are today. Their main memories were about a thousand times smaller. And each large computer was shared by up to thousand users. Here is a supercomputer of those times. In spite of its relatively modest capabilities, it filled a whole room with its cabinets, cables, and power supplies, and cost many millions of dollars. Multiplying the three factors of a thousand, we get a total of one billion-fold improvement from then till now in the availability of usable computer power; with a similarly spectacular reduction in physical size and cost. The performance figure is likely to improve by another factor of a thousand in the next fifteen years. And in talking about these billions, please don't forget how big a mere thousand is: one thousand is the factor that separates the speed of a crawling baby from that of a supersonic transport.

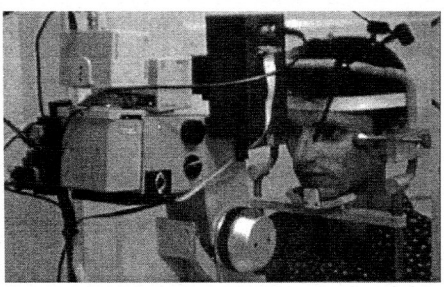
Another dramatic change has been in the commercial environment for the provision of general purpose basic software. In the 1960s, the main software for computers was written by computer manufacturers, and its code was kept as a closely guarded commercial secret. It was written in the machine language for a proprietary computer, which might have a commercial lifetime of just a few years. The code was not available for research, and was certainly not worth the effort of scientific study, since much of it was only ever used on less than a thousand computers. The situation today is radically changed. Nearly all software is written in a machine-independent high-level language. Much of the source code is open, and is made freely available by the computer scientists who wrote it. And much of it is used daily over a period of decades by many millions of users. The incentive to scientists to make every possible improvement to its reliability and serviceability is enormous. The open source code itself gives scientists the experimental material on which to work, and the open source movement gives us the inspiration to follow the example of open publication in the overall interests of mankind.

A third encouraging factor is that the state of the art in all the software technologies relevant for the verifying compiler has made enormous progress. Automatic theorem provers like HOL and ACL2 can now tackle proofs of some of the big fundamental theorems of mathematics. Model checkers like SPIN and FDR can explore the entire behaviour space of a distributed system, to check that there are no deadlocks or races that would stop the system from working. They have already made a significant contribution to the reliability of computer hardware, and to the verification of protocols for computer security, and they are beginning to be applied to the less tractable problems of software correctness. Decision procedures and constraint solvers like SAT and PVS are capable of finding counter-examples to proposed theorems, and they are being adapted to find test cases that will drive a program into an error state. If they do not find a counterexample, that increases confidence that there is none. Many programs today come with large regression test suites accumulated over many years, containing test data that have discovered previous errors in the program. Research tools like Daikon are designed to

exploit the test data to make plausible suggestions of the missing assertions in a program. And an early actual exemplar of a verifying compiler, ESC/Java, is now universally and freely available for educational and research use in Universities.

A fourth ground for optimism is that theory and practice in programming are already coming together. Speaking only from my experience in Microsoft, I have found that assertions are liberally sprinkled through large programs, and in some products may amount to as much as ten percent of the code. Program analysis tools like PREfix construct abstract models of the behaviour of twenty million line programs, and already detect up to a quarter of all anomalies corrected in the final stages before delivery of a commercial product. A faster version of PREfix called PREfast is in daily use by program developers; it exposes an internal compiler interface, an abstract syntax tree of the code, which enables individual researchers and programmers to extend the capabilities of the tool to detect further instances of errors already discovered once. The developing analysis tool SLAM developed in Microsoft Research has applied a model checker and a theorem prover in combination to several hundred device drivers which are run in the kernel of Windows XP. Further programmer productivity tools are under development. It is likely that the normal commercial pressures of market demand will lead to continuous improvements to such tools, as measured by the increase in the number of errors that they find, and the decrease in the number of false positive warnings. Many of these improvements will be based on the fundamental understanding gained by pure research.

That is why long-term pure research must retain its distinctive role, complementary to that of commercial development. Science is inspired not just by commercial pressures but also by curiosity and by idealism. The ideal of the chemist always has been to strive for absolute purity of materials; the ideal of the physicist always has been to strive for absolute accuracy of measurement. This is an instrument that uses lasers to measure distances inside the human eye with an accuracy down to three microns. It is used to measure blood flow to the retina. It is only possible now because the early researchers into laser interferometry went far beyond the stated needs of practicing engineers and industrialists of the day. Practicing engineers often complain that their main daily task is to find acceptable compromises with inadequate materials and hasty workmanship, so as to meet fixed constraints in budget and delivery date. The research scientist must ignore the complaint, because such specific problems can be solved only by exploiting factors specific to the particular situation facing the engineer. The job of the research scientist is to look for general solutions — as general as possible. In the end, the engineer and the industrialist and the marketer will find a way of exploiting the unnecessary generality, purity and accuracy that have been achieved by long-term scientific research by scientists pursuing impossible ideals.



Similarly, the ideal of the computer scientist always has been to seek a clearly stated and mathematically supported guarantee of the correctness of programs, not merely a reduction in the number of false warnings of error, either positive or negative. Of course in the final practical application of scientific discoveries, there will be many compromises to the ideal; it will still be the responsibility of the engineer on each project to decide how far it is appropriate to specify accurately the desired behaviour of programs and how much still to rely on intuition or searching tests, or even, let's face it, wishful thinking. The job of the scientist is only to ensure that it is not the science, nor the tools, but something else in each particular case that imposes these limiting constraints on the ideal of perfection.

Summary. The challenge of the verifying compiler is one of long standing, and all previous attempts to meet it have been rewarded by noble failure. But I now conjecture that significant advances in hardware and software technology, and in the professional environment, give rise for hope that the challenge could be met in the foreseeable future. All the strands necessary for development of a verifying compiler are already here. All we need is some way of weaving them together into a single program development and analysis tool, for use first by researchers, and later by professional programmers in the field. I suggest that the explicit announcement and the detailed planning of a Grand Challenge project is an effective way, perhaps the only way, of mobilising the available scientific talents and idealism to work co-operatively and competitively towards the answer of the basic scientific questions of software engineering and the theory of programming, namely: "How do computer programs work and why."

Sir Tony Hoare
March 2004