The Verifying Compiler: A Grand Challenge for Computing Research

TONY HOARE

Microsoft Research Ltd., Cambridge, UK

Abstract. This contribution proposes a set of criteria that distinguish a grand challenge in science or engineering from the many other kinds of short-term or long-term research problems that engage the interest of scientists and engineers. As an example drawn from Computer Science, it revives an old challenge: the construction and application of a verifying compiler that guarantees correctness of a program before running it.

Introduction. The primary purpose of the formulation and promulgation of a grand challenge is the advancement of science or engineering. A grand challenge represents a commitment by a significant section of the research community to work together towards a common goal, agreed to be valuable and achievable by a team effort within a predicted timescale. The challenge is formulated by the researchers themselves as a focus for the research that they wish to pursue in any case. It may pursue purely scientific goals, independent of economic, commercial, medical, military or social interests; and its initiation need not wait for political initiatives or prior allocation of special funding.

An opportunity for a grand challenge arises only rarely in the history of science, when a branch of study first reaches an adequate level of maturity to predict and plan the direction of future progress. Most scientific advances, and nearly all breakthroughs, are accomplished by individuals or small teams working competitively and in relative isolation; and the greater part of the research effort in any branch of science should remain free of involvement in grand challenges.

A grand challenge may involve as much as a thousand man-years of research effort, drawn from many countries and spread over ten years or more. The research skill, experience, motivation and originality that it will absorb are qualities even scarcer than the financial guarantees. For this reason, a proposed grand challenge should be subjected to assessment by the most rigorous criteria before its proposal and promotion. These criteria include all those proposed by Jim Gray [2003] as desirable attributes of a long-range research goal. The additional criteria that are proposed here relate to the maturity of the scientific discipline and the feasibility of the project. Many of the long-term systems research problems identified by Grey meet the original criteria in full measure; but they do not at the present time meet the additional criteria needed to accord them the status of a grand challenge.

Fundamental. It arises from scientific curiosity about the foundation, the nature or the limits of an entire scientific discipline, or a significant branch of it.

Astonishing. It gives scope for engineering ambition to build something never imagined before.

Testable. It has a clear measure of success or failure, which can be applied at any time.

Journal of the ACM, Vol. 50, No. 1, January 2003, pp. 63–69.

Revolutionary. It will lead to radical paradigm shift, breaking free from the dead hand of legacy.

Research-directed. The project can be forwarded by the methods of academic research. It is not likely to be met solely from commercial motivated evolution of existing products.

Inspiring. It has enthusiastic support from (almost) the entire research community, even those who do not participate in it, and do not benefit from it.

Understandable. It is generally comprehensible, and captures the imagination of the general public, as well as the esteem of scientists in other disciplines.

Challenging. It goes beyond what is initially possible, and requires development of understanding, techniques and tools unknown at the start of the project.

Useful. Work on the project brings scientific or other benefit, even if the project as a whole fails.

International. It has international scope: participation would increase the research profile of a nation.

Historical. It was formulated long ago, and will stand for many years to come.

Feasible. The reasons for previous failure are understood and can now be overcome.

Incremental. It decomposes into identified intermediate research goals.

Co-operative. It calls for planned co-operation among identified research teams and communities.

Competitive. It encourages and benefits from competition among individuals and teams, with clear criteria on who is winning, or who has won.

Effective. Its promulgation changes the attitudes and activities of scientists and engineers.

Risk-managed. The risks of failure are identified, and strategies to meet them are in place.

The tradition of grand challenges is common in many branches of science. If you want to know whether a challenge qualifies for the title ‘Grand’, compare it with

Put a man on the moon within ten years	(accomplished in 1960s)
Cure cancer within in ten years	(failed in 1970s)
Prove Fermat’s last theorem	(accomplished)
Map the Human Genome	(accomplished)
Map the Human Proteome	(too difficult for now)
Find the Higgs boson	(under investigation)
Find Gravity waves	(under investigation)
Unify the four forces of Physics	(under investigation)
Hilbert’s programme for mathematical foundations	(found impossible in 1930s)

All of these challenges in varying degrees satisfy many of the criteria listed above. Even though no individual challenge is expected to satisfy all the criteria, at least the criteria are relevant for their description and assessment.

In Computer Science, the following examples are listed *not as recommendations* but as examples that may be familiar from the past.

Prove that P is not equal to NP	(open)
The Turing test	(outstanding)
The verifying compiler	(abandoned in 1970s)
A championship chess program	(completed)
A GO program at professional standard	(too difficult)
Automatic translation from Russian to English	(failed in 1960s)
A mathematical model of the evolution of the web	(new)
A wearable computer serving as a guide dog for the blind	(new)

It is quite easy to extend this list with new challenges. The difficult part is to find a challenge that passes the main tests for maturity and feasibility. The remainder of this contribution picks just one of the challenges, and subjects it to detailed evaluation according to the seventeen criteria.

The Verifying Compiler: Implementation and Application

A verifying compiler uses mathematical and logical reasoning to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the code of the program. The compiler will work in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components. The capabilities and performance of the verifier will be demonstrated by application to a broad selection of legacy code, chiefly from open sources. *The verifying compiler does not itself have to be verified,* though it would be desirable to do so, at least partially. This proposed grand challenge is now evaluated under the seventeen headings listed in the introduction.

Fundamental. Correctness of computer programs is the fundamental concern of the theory of programming and of its application in large-scale software engineering. The limits of application of the theory will be explored and extended.

Astonishing. Most of the general public, and even many programmers, are unaware of the possibility that computers might check the correctness of their own programs.

Testable. If the project is successful, a verifying compiler will be available as a standard tool in some widely used programming productivity toolset. It will have been tested in verification of structural integrity and security and other desirable properties of millions of lines of open source software, and in more substantial verification of critical parts of it. This will lead to removal of thousands of errors, risks, insecurities and anomalies in widely used code. Proofs will be subjected to check by rival proof tools. The major internal and external interfaces in the software will be documented by assertions, to make existing

components safer to use and easier to reuse. Further improvements to quality and functionality of the code will be facilitated by good documentation of the internal interfaces.

Revolutionary. At present, the most widely accepted means of raising trust levels of software is by massive and expensive testing. Assertions are used mainly as test oracles, to detect errors as close as possible to their place of occurrence. Availability of a verifying compiler will encourage programmers to formulate assertions as specifications in advance of code, and many of them will be verifiable by automated or semi-automated mathematical techniques. Existing experience of the verified development of safety-critical code will be transferred to commercial software for the benefit of the mass market.

Research-directed. The methods of research into program verification are well established, though they need to be scaled up to meet the needs of modern software construction. This is unlikely to be achieved in industry. Commercial programming tool-sets are driven mainly by fashionable slogans and by the politics of standardisation. Their elegant pictorial representations often have no semantics attributed to them. Their designers are constrained by compatibility with legacy practices and code, and by lack of scientific understanding on the part of their customers (and themselves).

Inspiring. Program verification by proof is an absolute scientific ideal, like purity of materials or accuracy of measurement, pursued for its own sake in the controlled environment of the research laboratory. The practicing engineer has to be content to work around the impurities and inaccuracies of the real world. The value of purity and accuracy (and even correctness) are often not appreciated until after the scientist has shown that they are achievable.

Understandable. All computer users have been annoyed by bugs in mass market software, and will welcome their reduction or elimination. Recent well-known viruses have been widely reported in the press, and have been estimated to cost billions. Fear of cyber-terrorism is widespread. Viruses obtain entry by exploiting errors like buffer overflow, which could be caught quite easily by a verifying compiler.

Trustworthy software is now recognized by major vendors as a primary long-term goal. The interest of the press and the public in the project can be maintained, whenever dangerous anomalies are detected and removed from software in common use.

Challenging. The analysis and verification tools essential to this project are not yet available. Indeed, their development is the essential feature of the challenge.

Useful. Unreliable software is currently estimated to cost the US some sixty billion dollars [Planning Report 02-3. The Economic Impacts of Inadequate Infrastructure for Software Testing, prepared by RTI for NIST, US Department of Commerce, May 2002]. A verifying compiler would be a valued component of an Infrastructure for Software Testing.

The project may help accumulate confidence that will assess and reduce the risks of incorporation of commercial off-the-shelf software (COTS) in safety critical systems. The project will extend the capabilities of load-time checking of mobile

proof-carrying code. It will provide a secure foundation for the achievement of trustworthy software.

The main long-term benefits of the verifying compiler will be realised most strongly in the development and maintenance of new code, specified, designed and tested with its aid. We look forward to the day when normal commercial software will be delivered with an eighty percent chance that it never needs recall or correction by service packs, etc. within the first ten years after delivery. Then the suppliers of commercial and mass-market software will have the confidence to give the normal assurances of fitness for purpose that are now required by law for most other consumer products.

International. The project will require collaboration of leading researchers in America, China, India, Australasia, and many countries of Europe.

Historical. The idea of using assertions to check a large routine is due to Turing [1949]. The idea of the computer checking the correctness of its own programs was put forward by McCarthy [1963]. The two ideas were brought together in the verifying compiler by Floyd [1967]. Early attempts to implement the idea were severely inhibited by the difficulty of proof support with the machines of that day. At that time, the ephemeral nature and limited market for software written by hardware manufacturers reduced motivation for any major verification effort. Furthermore, the source code was usually written in assembler, and kept secret. Since those days, further difficulties have arisen from the complexities of modern software practice, for example, concurrent programming, object orientation, inheritance, etc. These new language features have been explored by theoreticians in the "clean room" conditions of new experimental programming languages. In this project, the results of such pure research will have to be adapted, extended, combined and tested by application on a broad scale to legacy code expressed in legacy languages.

Feasible. Most of the factors that have inhibited progress on practical program verification are no longer as severe as they were.

(1) Experience has been gained in specification and verification of moderately scaled systems, chiefly in the area of safety-critical and mission-critical software; but so far, the proofs have been mainly manual.

(2) The corpus of Open Software is now stable and used by millions, so justifying almost any effort expended on improvement of its quality and robustness.

(3) Advances in unifying theories of programming suggest that many aspects of correctness of concurrent and object-oriented programs can be expressed by assertions, supplemented by automatic or machine-assisted insertion of instrumentation in the form of ghost (model) variables and assignments to them.

(4) Many of the global program analyses that are needed to underpin correctness proofs for systems involving concurrency and pointer manipulation have now been developed for use in optimizing compilers.

(5) Increased machine capacity and modern software component technology now permit simultaneous use of multiple proof tools.

(6) SAT checking is providing spectacular increase in the power of proof tools.

(7) Iterative model checking now discovers relevant invariants and abstractions.

(8) Market pressure for trustworthy software is now greater than ever before.

Incremental. The progress of the project can be assessed by the number of lines of code that have been verified, and the level of annotation and verification that has been achieved. The relevant levels of annotation are: structural integrity, partial functional specification, total specification. The relevant levels of verification are: by testing, by human proof, by machine assistance, and fully automatic.

Cooperative. The work can be parcelled out to teams working independently on the annotation, on the compiler, and on the proof tools.

(1) The existing corpus of Open Source Software can easily be parcelled out to different teams for analysis and annotation; and the assertions can be checked by massive testing in advance of availability of adequate proof tools.
(2) It is now standard for a compiler to produce an abstract syntax tree from the source code, together with a database of program properties. This enables many researchers to collaborate on program analysis algorithms, test harnesses, test case generators, verification condition generators, and other verification and validation tools.
(3) Modern proof tools permit extension by libraries of theories that can be developed by many hands to meet the needs of each application.

Competitive. The annotated libraries of open source code will be good competition material for the teams constructing and applying proof tools. The proofs themselves will be subject to confirmation or refutation by rival proof tools. There will be competition to find errors in legacy code, and to be the first to obtain mechanical proof of the correctness of all assertions in each module of software.

Effective. The promulgation of this challenge is intended to cause a shift in the motivations and activities of scientists and engineers in all the relevant research communities.

(1) Researchers in programming theory will accept the challenge of extending proof technology for programs written in complex and uncongenial legacy languages. They will need to design program analysis algorithms to test whether actual legacy programs observe the constraints that make each theoretical proof technique valid.
(2) Builders of programming tools will do experimental implementation of the hypotheses originated by theorists, to explore the range of their application to real code.
(3) Sympathetic software users will allow newly inserted assertions to be checked in production runs, even before the tools are available to verify them.
(4) Compiler writers will support the proof goals by adapting and extending the program analyses currently used for optimization of code; later they will even exploit the redundant information in a verified program for purposes of further optimization.
(5) Providers of proof tools will regard the project as a fruitful source of low-level conjectures needing verification, and will evolve their algorithms and libraries of theories to the needs of actual legacy software and its users.
(6) Teachers and students of the foundations of software engineering will be enthused by student projects that annotate and verify a small part of a large code base, so contributing to the success of a world-wide project.

Risk-Managed. The main risks to the project arise from dissatisfaction with existing legacy code and legacy languages. The low quality of existing software, and its low level of abstraction, may limit the benefit to be obtained from the annotations. Many of the errors detected may be so rare that they are not worth correcting. Many failures of proof are not due to an error at all, but just to omission of a more or less obvious precondition. In other cases, preservation of an existing anomaly may be essential to the functionality of the software. Often the details of functionality of interfaces, either with humans or with hardware devices, are not worth formalizing in a total specification, and testing gives adequate assurance of serviceability. Legacy languages add to the risks of the project. From a logical point of view, they are extremely complicated, and require sophisticated analyses to ensure that they observe the disciplines that make abstract program verification possible.

The long-term solution to these problems is to discard legacy and start again from scratch. This could well be the topic of different grand challenges. One of these would involve design of a new programming language and compiler, especially designed to support verification; and another would involve a rewrite of existing libraries and applications to the higher standards that are achievable by explicit consideration and simplification of abstract interfaces. Research on new languages and libraries is in itself desirable, and would assist and complement research based on legacy languages and software.

ACKNOWLEDGMENTS. The content and presentation of this contribution has emerged from fruitful discussions with many colleagues. There is no implication that any member of this list actually supports the proposed challenge: Ralph Back, Manfred Broy, Alan Bundy, Michael Ernst, David Evans, Armando Haeberer, Joseph Halpern, He Jifeng, Jim Horning, John McDermid, Bertrand Meyer, J. Moore, Oege de Moor, Robin Milner, Larry Paulson, Jon Pincus, John Rushby, Natarajan Shankar, Martyn Thomas, Niklaus Wirth, Jim Woodcock, Zhou Chaochen, and many others.

REFERENCES

FLOYD, R. W. 1967. Assigning meanings to programs. *Proc. Amer. Soc. Symp. Appl. Math. 19*, 19–31.

GRAY, J. 2003. What next? A dozen information-technology research goals. *JACM 50*, 1 (Jan.), 41–57.

MCCARTHY, J. 1963. Towards a mathematical theory of computation. *Proc. IFIP Cong. 1962*. North-Holland, Amsterdam, The Netherlands.

TURING, A. M. 1949. Checking a large routine. *Report on a Conference on High Speed Automatic Calculating Machines*. Cambridge Univ. Math. Lab. 67–69.