The ideal of program correctness.

Tony Hoare, **Draft: May 2006**

Summary. The ideal of verified software has long been the goal of research in Computer Science. This paper argues that the time is ripe to embark on a Grand Challenge project to construct a program verifier, based on a sound and complete theory of programming, and evaluated by experimental application to a large and representative sample of useful computer software.

Introduction. Computer Science owes its existence to the invention of the stored-program digital computer. It derives continuously renewed inspiration from the constant stream of new computer applications, which are still being opened up by half a century of continuous reduction in the cost of computer chips, and by spectacular increases in their reliability, performance and capacity. The Science of Programming has made comparable advances by the discovery of faster and more general algorithms, and by the development of a wide range of specific application programs, spreading previously unimaginable benefits into almost all aspects of human life.

These amazing advances in computer application can distract attention from the fact that Computer Science also has a central core of fundamental discoveries which are particular to itself as an independent intellectual discipline. Computing Research is driven, like research in other mature branches of pure science, by natural curiosity, exploring the basic foundations and limitations of the programmable computer, independent of any particular area of application. Because of its effective combination of pure knowledge and applied invention, Computer Science can reasonably be classified as a branch of Engineering Science.

Like all scientists, we are faced with the problem of complexity, both of computers and of the programs that control them. Many software systems in wide-spread and productive use today have grown and evolved over several decades. Although they are human artefacts, they are now comparable in complexity with the most complex known natural phenomena, for example the Human Genome, whose raw binary code (nearly a gigabyte) has recently been laboriously decoded and published. Geneticists are now engaged in the even more challenging task of understanding the complexity of this code. They too are driven by curiosity about the fundamental questions about the role of the genome as a blueprint for an entire human being. They want to find out firstly what the genes do, and secondly how they do it. They want to discover the basic chemical principles which govern genetic activity, and so to understand not only how but why the genome works as it does. And finally, they wish to support all their discoveries and generalisations by accumulation of sound scientific evidence. Even when the scientist has accomplished all these objectives, it remains for the engineer and the industrialist to find out how to exploit enlarged scientific understanding for commercial profit.

The challenge facing Computer Science is very similar to that facing genetics. Our first and entirely non-trivial task is to understand what a computer program does. As

for other engineering artefacts, the externally visible aspects of program behaviour can be codified as a formal engineering specification, expressed in the relevant technical terminology. An explanation of how a program works can be formally expressed in terms of types, assertions, and other redundant annotations. They serve as internal specifications, attached at all the major and minor program interfaces. The correctness of the explanation can in principle be checked by a program analysis tool known as a program verifier. It uses automated logical and mathematical proof techniques to check consistency between a program and its internal and external specifications. A program verifier can play the same role in Software Engineering research as the automatic tools that are now essential or even obligatory in other branches of Engineering, to check the soundness and safety of engineering designs, long before they start construction. An adequately specified and annotated program, which has passed the scrutiny of an automatic program verifier, is said to be a verified program. It offers highly credible evidence that the program will work in accordance with its specification.

The ideal of verified software has been a long-standing inspiration to research in basic Computer Science, and has driven the development of a number of advanced tools performing many of the functions of a program verifier. The most widely used tools concentrate on the detection of programming errors, widely known as bugs. Foremost among these are modern compilers for strongly typed languages, which give warnings of potential anomalies in a program, insofar as those bugs that can be diagnosed without any knowledge of the program's specification. More advanced program analysers begin to take specifications into account. These have been applied by the computer hardware industry to verify programs that simulate the behaviour of computer chips, and they have averted expensive hardware design errors. Other program analysers are routinely used in the software industry to detect security risks and other errors in large-scale legacy code and in modifications to it.

It is expected that normal commercially motivated development of these tools will increase their power to detect more and more errors. This could be an unending task. There is evidence that in large-scale software there will always be more errors to detect, especially since correction of each error is itself prone to error. Eventually only the rarest errors will remain: each one that occurs in practice is extremely unlikely ever to occur again. Such errors are often not worth correcting, unless there is a risk that the error can be exploited by viruses, bugs, worms. Unfortunately the analysis of each error is both expensive and error-prone. Like insects that carry disease, the least efficient way of eradicating program bugs is by squashing them one by one. A completely different approach is needed. The only sure safeguard against attack is to pursue the ideal of not making the errors in the first place.

That is the goal of more advanced program verification tools. They have been used in the design of critical embedded software applications, often to assist human reasoning in achievement of correctness by construction. They have also been used in support of academic teaching of the principles of programming. But program verification tools of the present day are a long way from the original vision of a program verifier described by Jim King in his Doctoral thesis in 1969. The more practical analysers in use today have made significant compromises, affecting the soundness of their guarantee of correctness as well as the expressive power of the language in which specifications and programs are written. The more idealistic tools are restricted in

application by problems of scale, both in the size of the programs treated and in the complexity of the programming language accepted for analysis.

I suggest that the construction of a program verifier, with capabilities close to the original ideal, may be achieved in the foreseeable future by a co-ordinated long-term program of multi-national research, with three strands:

1 Theories: development and unification of the relevant general theories of programming, to cover programming languages in use today. It would have to include features of object orientation, inheritance, concurrency, etc.

2 Tools: incorporation of the theories into a coherent and co-ordinated toolset for program analysis, with evolving capabilities for program verification by a variety of techniques of constraint solving, model checking, and automatic theorem proving,

3 Experiments: evaluation of the tools by experimental application to a large and representative collection of real computer programs and their specifications, which are accumulated together with their specifications and proofs in a scientific repository. As a long-term target, we may hope to accumulate a million lines of verified code.

The project would employ computer scientists with varied specialist skills and experience drawn from around the world. We must combine long-term co-operation on strategic development with short-term scientific competition on methods and tactics. We must co-ordinate long-term planning of the eventual product with the setting of a hierarchy of intermediate goals. We must organise a division of labour to construct each of the tools of the verification toolset, and to verify each program in the expanding repository. We must ensure that intermediate results are accumulated in the repository, so that experiments can be repeated, and further research can build on their results. The broad scale, the long duration, and the high scientific ideals of this project are comparable to those of the Human Genome project; and maybe we too would be justified in appropriating the title of a Grand Challenge.

The methodology of the project derives its inspiration from the traditional practices of pure scientific research – the construction of theories, the exploration of their applicability by experiment, and (increasingly in the present day) the development and use of computer tools to confirm the match between theory and experimental result. The scientific understanding and technological advances arising from successful completion of the project will afford the opportunity for significant reduction in the direct and indirect costs currently associated with programming error.

The ideals of pure Science.

Traditionally, the pure science of Physics claims the crown as the most advanced of the natural sciences. It satisfies a basic human curiosity by exploring the fundamental components and the structures of the material universe, and by giving an account of its origin and history and even its future. The most sophisticated mathematical concepts and theories have been developed, not just to describe but also to explain the behaviour and mutual interactions of all material objects, ranging is scale from quarks and elementary particles to clusters of galaxies and super-clusters. Like other branches of pure science, Physics invents its own language to ask its own abstruse

questions, it sets its own agenda of investigation, and it engages in massive long-term collaborative projects to confirm its most general theories; a current example is the its own language to ask its own abstruse questions, it sets its own agenda of investigation, and it engages in massive long-term collaborative projects to confirm its most general theories; a current example is the construction of high-energy particle accelerators, by which it is hoped to confirm existence of the theoretically predicted Higgs boson.

Computer Science is better known as an applied science, having more in common with other branches of Engineering Science than with a pure science like Physics. Its value has been fully demonstrated by the enormous contributions that have been made by computers and their software to almost every aspect of the modern technological world. And new opportunities for beneficial application are still repeatedly opened up by continuing improvements in the versatility and power and ubiquity and cheapness of computer hardware, reinforced by increases in the speed of computer-mediated communications. The success of any particular software product or project requires an understanding not only of computers and of their general-purpose software, but also of the domain in which they are to be applied. In this respect, applied Computer Science, like applied Mathematics and Statistics, is an inherently multi-disciplinary discipline.

Again like Mathematics and Statistics, Computer Science has a pure branch, in which research is motivated by curiosity and high scientific idealism. We pursue a scientific ideal in the same way that Physicists to pursue the utmost accuracy of measurement, or chemists seek the utmost purity of their materials. For the computer scientist, the total correctness of computer programs is just such an ideal. Scientists seek such ideals for their own sake, going far beyond the current needs of the practicing engineer. The main daily concern of the engineer is to accommodate unavoidable impurities in materials and inaccuracies in measurement, just as computer users have to find workarounds for discovered errors in computer programs. Practical engineering is all about compromises that take into account the particular circumstances and timescales of the current project, and the particular interests of the current customer. For the engineer, good enough is always good enough; and fixed budgets and delivery dates are always an adequate excuse for imperfection. The scientist knows that only perfection will protect his work from being superseded by later work of other scientists.

In contrast to the particularities exploited by the good engineer, the pure scientist pursues generality of theory for its own sake. Although the success of any particular experiment may demand skilful compromise, the long-term goals are to transcend the particular circumstances of the current experiment, and to extend the boundaries of application of the current theory. The ultimate accolade goes to those who discover the most general concepts, explaining by the laws of a unified theory such highly disparate phenomena as the fall of an apple and the motion of the planets and moon.

Another ideal pursued by the scientist is certainty of knowledge, gained by accumulation of scientific evidence from widely varied sources. For the engineer, certainty is an irrelevance. His main concern is to make good decisions in the face of prevailing uncertainties that would take far too much time and money to remove.

Nevertheless, the scientist pursuing more abstract ideals, and accumulating knowledge in collaboration with the scientific community, will often make totally unexpected and unplanned contributions to the later success of the engineer and even to the monetary profits sought by the commercial entrepreneur. One day, the engineer realises that the purity and accuracy, which the pure scientist has shown to be achievable in the laboratory, can be exploited on an industrial scale in a deliverable product of a completely new kind. For example the silicon chip is now manufactured in a fabrication line that achieves levels of environmental purity that were only dreamed of in the scientific laboratories of twenty years ago. One day, the entrepreneur realises that a completely new market can be found for the product, and money can be made from it. Amazingly, the roles of scientist, engineer and entrepreneur are sometimes concentrated in a single person, who makes outstanding contributions in all three areas. But this can only be because that person recognises how different the three roles actually are.

The extra generality of theory sought by the pure scientist also offers long-term benefits for the engineer. It is a more general theory that allows the practical experience gained by the engineer on one project to be transferred to a later project which is not identical to it. It is generality of theory that allows the engineer to explore a range of product designs, and select the one that most fully satisfies the needs of a broad market of potential customers. Experience of modern technology reveals again and again the benefits of an understanding of general theories: initially, they seem to go far beyond the needs of any particular case, but in the long run they lead to continuous stream of new products which are more functional, more economic, and more reliable than anything that preceded them.

In summary, in the advancement of Engineering Science, the engineer and the pure scientist play distinct but closely related roles; their contributions are complementary to each other, and equally necessary. The role of the software engineer in extending the benefits of computer application can be immediately recognised and financially rewarded. But Computer Science also has a pure branch, which deserves equal recognition. It seeks answers to the same basic questions that inspire all branches of engineering science, no matter what their particular area of application.

The five basic questions of an Engineering Science.

There are five basic questions that are common to all branches of Engineering Science, whether the objects of study are ships, bridges, motor cars, genes or computer software. In summary, they are

1. What does it do?
2. How does it work?
3. Why does it work?
4. How do we know?
5. And how can we exploit the knowledge to improve the product?

The third and fourth questions are primarily the domain of pure science, and the rest have more to engineering. The first engineering question is ‘precisely what is the product for, and exactly what does it do to meet its goals?’ The answer to this question is given in the form of an engineering specification of the product. Such a

specification is usually drafted as a guide to design as well as the use of the product; it is therefore formulated at a high level of precision and detail. It is suitable for use by other engineers, and may contain more information than the average user of the product would wish to know.

Secondly, the engineer wants to understand exactly how the product works. This is described at varying levels of granularity and detail by giving the specifications of the internal interfaces of the product. These explain the functions of each component of the product, and how they interact. Often, the interface specifications are sufficiently complete and precise to permit mathematical calculations, guaranteeing that the joint working of all the components will lead to the correct operation of the product as a whole. In a mature branch of engineering, these calculations are implemented in a computer program, whose use is often obligated by standards of professional practice, and in some cases even by law.

The pure scientist asks two further questions, perhaps even more basic, about an engineering product. The first is the question 'Why does the product work?' The explanation must appeal to general scientific principles, ones that apply not just to a particular product, but to a general range of similar products, actual or hypothetical. The answers are found in the basic laws and fundamental theory of the relevant branches of pure Science. And finally, the scientist asks the most important question of all: 'How do we know that the answers to the previous questions are actually right? How do we know that the theory corresponds to reality in general, as well as in each particular case?'

The answer is given by the experimental method, as recommended by Francis Bacon, an English statesman and essayist of the sixteenth century; and since his day it has been on experiment that our confidence in the whole of modern Science is based. The desired connection between the theory and the observable experimental results often involves a long chain of mathematical reasoning and calculation and proof. In earlier times, these calculations were performed by hand; but now the essence of the scientific theories is built into computer programs which analyse high volumes of experimental data collected by modern scientific instruments at a rate of terabytes per second. Computer programs are also essential to check the conformity of this data with theory.

The last engineering question, on how to exploit the accumulated knowledge for commercial advantage, is one which the scientist, pursuing knowledge for its own sake, should not be required to answer in advance. Pure knowledge is independent of application. That is why it is so valuable. There is plenty of experience that the first and most important application of new knowledge will be to meet needs that are entirely unpredicted when the research starts. Knowledge is what prepares us to meet the problems of an unknown future. So the fifth question is one that should not be answered until after the knowledge has been accumulated.

The general scientific questions described above are applied by Computer Scientists, to computer programs. The first question is 'What does the program do?'; it is answered by a functional specification of the system, expressed as a formal description of the observable properties of its intended behaviour in action. The second question 'How does it work?' is answered by specifications of the internal

interfaces between components of the system, often expressed by technically redundant declarations and assertions sprinkled in the text of the program. The third question 'Why does the program work?' is answered by the theory of programming, which formalises the semantics of the programming language in which the program is written: this provides a basis for the rules which define the correctness or conformity relation between a program and its accompanying documentation.

And the final question is 'How do we know that the program is in fact correct?' The theory of programming tells us that this final assurance can in principle be given by mathematical reasoning and proof, guaranteeing that the specifications are a logical consequence of the text of the program. This theory has already been put into practice. Since the earliest times, proofs for small and critical programs have been constructed manually, and checked by human eye. In some cases, the proofs have been constructed as part of the development process for the software. More recently, the reliability and effectiveness of the verification has been increased by automation of the construction or the checking of the proofs. In analogy with other branches of science, consider the text of the program as the experimental data; consider the specifications of the external and internal interfaces of the program as a theory of how and why the program works. Now an automatic tool for program verification is one that checks the consistency of the theory with the actual text of the program, just like the analysis tools of other branches of science and engineering. Its application greatly increased confidence that the verified program when executed will conform to specification. That is the dream that has for over thirty years driven research in basic Computing Science.

The tool that realises this dream is called a program verifier. Unfortunately it does not yet exist.

Proposal for a Grand Challenge project.

A project to construct a program verifier will require the general support from the entire computer science research community, and especially from those who have the background, the skills and the experience to make a substantial contribution to its progress. The relevant topics of research include programming language semantics, programming principles, type theory, compiler construction, program analysis and optimisation, test case generation, mathematical modelling, programming methodology, design patterns, dependability, software evolution, and construction of programmer productivity tools. In addition there are various approaches to mechanical theorem proving, which include proof search, decision procedures, SAT solving, first-order induction, higher order logic, algebraic reduction, resolution, constraint solving, model checking, invariant abstraction, abstract interpretation. These lists are not intended to be complete; new ideas are very necessary, and will be welcomed from any quarter.

The main challenge of a verification project will be to bring this wide range of skills to bear on the evolution of a coherent toolset. At least an equal effort must be devoted to exercise and evaluate the prototype tools on a realistic selection of actual computer programs and their specifications. The history of computing gives examples of amazing progress that can be made in the evolution of tools by their repeated application to a series of agreed challenges. And the success of the project would

have an amazing impact on professional practice of programming, on the justified confidence which Society places in computers, and on the further progress of scientific research in consolidating and extending these benefits.

It is hoped that the majority of specialists in all these fields will welcome the prospect that a program verifier will exploit the results of their research, for the ultimate benefit of all programmers and users of computers. But most of them will not wish to commit their own efforts to such a long-term and laborious collaborative project. And rightly so. Most of scientific progress, and nearly all breakthroughs, are made by individual scientists, working by themselves or in a small local team; they need to preserve their freedom to pursue their own bright ideas in their own directions, and to communicate their research results by publication in the research literature. Other excellent engineering scientists may be discouraged by the long timescales of the project. They will prefer to grasp ever expanding opportunities for transfer of software verification technology into direct industrial and commercial exploitation, and they will enjoy the more significant and immediate benefits that can be achieved. The transfer of verification technology to the computer chip industry took just such a course. A balance between short-term opportunistic researches and a long-term co-ordinated research programme is essential. There should be a fruitful interchange of research skills, prototype tools and theoretical understanding between one style of research and another. Indeed, even after completion of the long-term project, the practical exploitation of a program verifier will be critically dependent on the continuing progress of research in such areas as system dependability and software evolution, programming methodology and software engineering.

In conclusion, we should not expect more than a small percentage of the relevant research communities to be engaged in a Grand Challenge project at any one time. Success of the project is far from a foregone conclusion, and to commit more than a small proportion of the world's scarce resources of scientific talent in any particular specialist area would be simply too great a risk.

The success of a Grand Challenge project depends on the agreement of a substantial community of the world's scientists, not only that the project is worth while, but also that the time is ripe to start it now. The project can hardly start without a measure of agreement on the following points

1. selection of an initial set of complementary tools, and allocation of responsibility for their development
2. establishment of a repository of representative programs and specifications, together with assertions, test cases, development histories, and other relevant formal material.
3. planning for adaptation of the tools and representative programs, so that each tool applies to all programs
4. division of responsibility for supply of missing specifications, incomplete assertions or missing code for specifications and programs in the repository
5. experimental application of tools to the material in the repository
6. accumulation of the results of experiment for exploitation in subsequent research and development
7. identification of opportunities for improvement in the tools, and a planned programme for their implementation

8. design of internal interfaces behind which specialised tools can be combined, while preserving their freedom to evolve independently
9. election of an international guidance committee to oversee the progress and direction of the research.

In later phases, the project would develop by expanding the range and ambition of the representative programs in the repository, by implementation of new tools that combine technologies previously found successful on separate tests.

There is plenty of experience of large-scale, long-term collaborative projects in other branches of science. For example, in Astronomy and in Nuclear Physics, all new particle accelerators, satellites and telescopes are planned as long-term national or international collaborations. But such projects have so far been rare in Computer Science, and this may be a symptom of the immaturity of our subject. To embark on such a project now will need a fairly radical change to the culture and the daily practice of our research. We will have to accept that a worthy method of publication of new theoretical results will be to incorporate them in a set of tools that has been designed by others. We will have to accept that the best progress will be made by combining the technologies currently promoted in rival tools, each of which has hitherto aimed at universal applicability. And above all, we will have to give the highest scientific rewards to those who apply other peoples' theories and other peoples' toolsets to programs that have been written by yet some-one else. It is the users of telescopes and particle accelerators that win the Nobel prizes, not their builders. A broad division of labour among specialists is commonplace in all mature branches of science: no-one expected Einstein to test his own theories, and no-one expected Eddington to devise his own theories for experiment. But in our subject such division of labour would be a novelty. It is likely that successful conduct of a Grand Challenge project may require radical changes in current modes of refereeing, publication practices, administration of research funding, and even the criteria for promotion of academic researchers.

Costs and benefits.

Although the main goal of a Grand Challenge project is the advancement of Science, it would be unrealistic to embark on the project without some consideration of the costs and benefits for society as a whole.

The costs may be roughly estimated as between one and two thousand man-years of scientific effort, expended throughout the world over a period of ten to twenty years. This could be approaching ten percent of the world-wide availability of research skills of those currently engaged in the relevant areas of formally based research in Computer Science. The entire cost of the whole project over the entire period should be around one billion dollars, shared among all the nations of the world.

The benefits of program verification will be delivered in the form of reduction of the phenomenon of programming error, and a consequential increase of confidence in the dependability of software systems in widespread use. Fortunately, an estimate of the cost of programming errors is already available from an independent source, which attributes them to an inadequate infrastructure for program testing. Here is an extract from a recent report.

> ***Based on the software developer and user surveys, the national [US] annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion. Over half of these costs are borne by software users in the form of error avoidance and mitigation activities. The remaining costs are borne by software developers… [The Economic Impacts of Inadequate Infrastructure for Software Testing, US Dept. Commerce Planning Report 02-03, May 2002].***

This figure should probably be doubled to cover the world-wide costs of programming error, and doubled again, if nothing is done about it, to cover the growth in computer usage in the next decade. The prospect of saving just one percent of this waste of resource for just one year would justify the allocation of more funds to a Grand Challenge project in program verification than it could ever find productive ways of spending. The limitation on the rate of progress will be the availability of researchers with the necessary background, skills and enthusiasm. Each year's delay in the delivery and exploitation of the results of the research will cost far more than the entire cost of the research project.

However, we must recognise that the entire project is a risky one, and there is no guarantee of return on investment. Possible causes of failure will be the unwillingness of the research community to change its cultures and practices, the difficulty of obtaining collaboration in the large amount of meticulous work required to specify and annotate and prove computer programs. And perhaps the project is impossible anyway, because of the exponentially high complexity of the powerful theorem proving algorithms that will be needed.

Furthermore, we must recognise that full exploitation of the ultimate benefits arising from the project will require more than simply the availability of a scientific prototype of a program verifier. It will require that software engineers as a profession must adopt a more scientific approach to the whole task of program development and evolution, from the elucidation of requirements and formalisation of specifications, to the design and testing of program changes to be installed in running software. It will require the development of re-usable libraries of useful concepts and specifications, covering all the major application areas for computers. It will require that the technology of verification developed in the project (though probably not the prototype verifier itself) should be incorporated into commercially marketed tool-sets. It will require the training and motivation of software engineers in the use of the tools; and when the proof technology is widely available, its use may be mandated, as in other branches of engineering, by official codes of engineering practice, reinforced perhaps by professional, legal or commercial sanctions. Even if software engineers can become fluent in the new technology within one week, the training of the entire profession worldwide, perhaps a million programmers, will cost more than the whole research project.

It is not the role of the scientist to predict or recommend what changes in education, law, and society will be necessary to enable newly developed technology to be used for the benefit of the world. It is our role merely to make such changes possible. And without a program verifier, they will not be possible. In summary, the cost of technology transfer will be at least ten times the cost of the basic research.

Fortunately, success in the original research project will greatly reduce the risk of this later and larger investment.

Public esteem

In the present day, it must be admitted that the general public holds the profession of programming in rather low esteem. The newspapers delight in reporting examples of major projects that are over budget, late, and sometimes even cancelled before delivery. One of the many causes for these failures is the inadequacy, the instability, or even the total absence of timely specifications, agreed in advance with the informed consent of the customer. And even after delivery, the programs are full of annoying bugs, in some cases affecting many millions of users throughout the world. Sometimes these bugs provide a target for the entry and spread of viruses and worms in the computer network, which cause billions of dollars of damage to those whose business relies on the web.

The low esteem of the programming profession is confirmed by an examination of our normal every-day mode of working. Surely we are the only profession in the world that expends half of its working life detecting and removing mistakes committed in the other half. Our excuse is that without massive debugging efforts, the software delivered to customers would be even less reliable. But other professions have learnt that it pays to devote their main efforts to preventing the errors from occurring in the first place. If a program verifier can help us to do that, perhaps we can begin to earn the trust and respect of the public, and even perhaps improve our own self-respect. In the recognised professions such as medicine and law, as well as in established branches of engineering, professional practitioners strengthen their claim to the trust of the public, because they owe allegiance to principles and ideals that transcend considerations of personal, political, or financial advantage. It is important that Computer Scientists should insist on their right to pursue similar impersonal ideals; because that will in general enable our long-term advice to be treated by society as unbiased by short-term gains

One of the beneficial side-effects of the announcement of a Grand Challenge project is to raise public awareness and interest in the progress, the methods and the results of scientific research. Astronomy has gained enormous visibility from the spectacular achievements of the manned space programme, and the human genome project has attracted many clever and enthusiastic students into a scientific career in branches of biology. In general, young people are attracted to science and engineering by their natural youthful idealism and their innate curiosity about the real world, and about the workings of the products of engineering. A Grand Challenge in verified software may not have the same glamour as challenges in Genetics or Physics, but it makes the same kind of appeal to students who really want to understand how things work and why.

Conclusion

The long-term benefits of an improved understanding of the relationship between programs and their specifications are expressible as a vision of a future world in which

0. The education and training of software engineers is based on scientific principles
1. Software engineers can be relied on to deliver new products on time, within budget, and in accordance with specification
2. No design or implementation errors are ever found in delivered software, or very few
3. Changes and improvements to working software are undertaken with equal confidence in their serviceability
4. Computer software is always the most reliable component in any system which it controls.

These goals will be achieved by basic advances in our understanding of Computer Science. The advances are made by the normal scientific method of development of a comprehensive theory, the conduct of experiment to confirm its range of application, and the development of sophisticated computer programs to check the match between experiment and theory. At present, the development of the technology is inhibited by cultural and financial constraints on the scientific practice of Computer Scientists. A Grand Challenge project, planned and undertaken by Computer Scientists themselves, may help to remove these constraints, and empower Computer Scientists to realise their ambitions for the advancement of their own subject, both in theory and in application.