Verified software: theories, tools, experiments

Vision of a Grand Challenge project

Tony Hoare and Jay Misra **December 2005**

Microsoft Research Ltd. and The University of Texas at Austin.
Email: thoare@microsoft.com. misra@cs.utexas.edu

Summary. The ideal of correct software has long been the goal of research in Computer Science. We now have a good theoretical understanding of how to describe what programs do, how they do it, and why they work. This understanding has already been applied to the design, development and manual verification of simple programs of moderate size that are used in critical applications. Automatic verification could greatly extend the benefits of this technology.

This paper argues that the time is ripe to embark on an international Grand Challenge project to construct a program verifier that would use logical proof to give an automatic check of the correctness of programs submitted to it. Prototypes for the program verifier will be based on a sound and complete theory of programming; they will be supported by a range of program construction and analysis tools; and the entire toolset will be evaluated and evolve by experimental application to a large and widely representative sample of useful computer programs. The project will provide the scientific basis of a solution for many of the problems of programming error that afflict all builders and users of software today.

This paper starts with an optimistic vision of a possible long-term future of reliable programming. It argues that scientific research will play an essential role in reaching these long-term goals. It suggests that their achievement should be accelerated by a major international research initiative, modelled on a Grand Challenge, with specific measurable goals. The suggested measure is one million lines of verified code. By definition, this consists of executable programs, together with their specifications, designs, assertions, etc., and together with a machine-checked proof that the programs are consistent with this documentation. We anticipate that the project would last more than ten years, consume over one thousand person-years of skilled scientific effort, drawn from all over the world. Each country will contribute only a proportion of the effort, but all the benefits will be shared by all.

The paper concludes with suggestions for exploratory pilot projects to launch the initiative and with a call to volunteers to take the first steps in the project immediately after this conference. A possible first step will be to revise and improve this paper as a generally agreed report of goals and methods of the scientists who wish to engage in the project.

1. The long-term vision (20-50 years from now).

Programmers are human and they make mistakes. There are many competent programmers who spend as much as half their time in detecting and correcting the mistakes that they and their colleagues have made in the other half. Commonly used programming tools and languages do little to prevent error. It is not surprising that software products are often delivered late, and with a functionality that requires years of evolution to meet original customer requirements. Inevitable evolution of the requirements, and changes in the environment of software use, are increasingly hard to track by changes made to the program after delivery. The US Department of Commerce in 2002 estimated that the cost to the US economy of avoidable software errors is between 20 and 60 billion dollars every year. Over half the cost is incurred by the users.

In the progress of time, these problems will be largely solved. Programmers of the future will make no more mistakes than professionals in other disciplines. Most of their remaining mistakes will be detected immediately and automatically, just as type violations are detected today, even before the program is tested. An application program will typically be developed from an accurate specification of customer requirement; and the process of rational design and implementation of the code will be assisted by a range of appropriate formally based programming tools, starting with more capable compilers for procedural and functional programming languages. Critical applications will always be specified completely, and their total correctness will be checked by machine. In many specialised applications, large parts of a program will be generated automatically from the specification. Declarative and procedural programming languages will emerge or evolve to assist in reliable program design and verification. Education of software engineers will be based on an appreciation of the underlying science. As a result, inevitable changes in environment and in customer requirements will be more easily and rapidly tracked by changes in the delivered software.

Progress towards these solutions will require many changes in the current prejudices and professional practice of the designers and implementers of software. It will involve a great many interlinked educational, social, commercial, financial and legal issues, which are outside the domain of scientific research. Experience shows that scientific progress is best achieved by postponing discussion of such issues until the underlying scientific issues have been resolved, and the tools are available for its wide-spread application. As scientists, we take the view that the ultimate solution to the problem of programming error must be based on a full and accurate answer to some of the basic questions explored in computing research. They are similar to questions asked about the products of any other branch of pure and applied science, namely: what does a computer program do; how does it work; why does it work; how do we know the answers are correct? Only when these answers are found can we address the issue of how to exploit the answers to make the programs work better.

For many simple but critical applications, we already know the answers to these questions. A formal specification tells us precisely what a program does. Assertions internal to the program define the modules and interfaces, and explain how the program works. The semantics of a programming language explains why programs work. A mathematical proof is what checks that the answers to the above questions are correct. And finally, a theory of programming explains how to exploit the scientific understanding to improve the quality of software products. The continuing challenge is to broaden and deepen our understanding, for application to the full range of computer programs in use, both now and in the future.

Steady progress towards the answers is emerging from the normal scientific processes of competitive research and open publication. Much of this research is already driven by the need to solve immediate industrial and commercial problems. These strands of research must certainly continue. This paper addresses an additional question: whether progress towards the longer term goals could be dramatically accelerated by the concerted effort of an international team of computer scientists. This will be a Grand Challenge project, similar to those that build satellites for astronomers and particle accelerators for physicists. Such a project should not be undertaken lightly, because it will involve commitment of more than a thousand person-years of effort from our most inventive scientists and engineers; their enthusiasm and talent will always be our scarcest scientific resource. The project will involve uncomfortable changes to the current culture and practice of science; perhaps we will have to break down some of the current divisions between our research communities, and abandon some of our favourite controversies between rival schools of thought. Recognition of success will be far in the future, and failure will be an ever-present risk.

But the rewards will be spectacular. The theory of programming will be proved applicable and adequate to explain the workings of most of the kinds of program that people actually want to write and use. Computing will demonstrate its maturity as a respectable branch of engineering science. Its basic scientific discoveries will underpin a comprehensive range of programming tools in widespread professional use. Experiments in the development, application and evolution of prototype tools will give strong scientific evidence of their value and limitations. Evidence of intermediate progress will encourage research in the many other areas of computer science, whose results will make essential contributions to the eventual application of the knowledge and tools developed by the project. Confidence in the availability of tools will also trigger discussion, planning and action on the many issues outside the domain of science that are needed to exploit the results of the research and so to achieve our long-term vision.

As in other branches of science, the achievement of the goals of a Grand Challenge is only a milestone of scientific advance, providing new methods that accelerate research aimed at even broader and more fundamental goals. For example, a program verifier is only a partial contribution to the development of reliable and dependable software: its availability will stimulate intensified research into the reliable capture of requirements and their faithful encoding as specifications. Indeed, the philosophy and technology of formal software verification may be an inspiration and foundation for a new general science of System Certification, covering not only software but also many other aspects of the design and implementation of major technological products that raise critical concerns of safety.

A program verifier, together with its associated toolset of specification aids, code generators, test environments, etc., will eventually be integrated into a single professional support environment, that might play a role as an 'intelligent programmer's assistant'. Its intelligence will be based on the capability of a computer to perform not only numerical calculations but also formal deductions, in a manner fore-shadowed by philosophers in the tradition of Aristotle, Leibnitz and Russell. Such a professional support environment might provide a pattern for similar intelligent assistants in other engineering professions where dependability is a crucial concern, perhaps including even mathematics and medicine.

2. The contribution of the Grand Challenge project (20 years from now).

We envisage that the Grand Challenge project will deliver its testable and measurable scientific results within twenty years. The results may be classified under three headings: theory, tools and experiments.

2.1. Theory. The project will deliver a comprehensive and unified theory of programming, which covers all the major programming paradigms and design patterns appearing in real computer programs, and nearly all the features of the programming languages of the present and foreseeable future. The theory will include a combination of concurrency, non-determinism, object orientation and inheritance. Many disciplined software design patterns will be proven sound, and analysis algorithms will be developed that check whether a program observes the relevant disciplines. Development of the theory should consume only a small percentage of the total research effort.

2.2. Tools. The project will deliver a prototype for a comprehensive and integrated suite of programming tools for the design, development, analysis, validation, testing and evolution of computer programs that are free of all errors of certain clearly specified kinds. We describe the tools in two classes: programming tools and logical tools. Development of the tools should consume a bit less than half the total effort.

The programming tools will include program development aids such as specification analysers, interrogators, checkers, and animators, as well as a range of application-oriented automatic program generators. They will include type checkers, and program analysers to detect possible violations of theoretically sound design disciplines. They will include test harnesses using assertions as test oracles, and test case generators based on a combined analysis of specifications and program text. They will include aids to evolution and reverse engineering of legacy code, for example, tools that infer assertions from comprehensive regression tests. All the tools of the suite will accept and produce data in compatible formats, and with the same semantics for intermediate specifications and annotated code.

The logical tools will employ a variety of strategies such as proof search, resolution, decision procedures, constraint solving, model checking, abstract interpretation, algebraic reduction, SAT solving, etc. These technologies will work together across broad internal interfaces, and they will share in the solution of each verification problem. They will be supplemented by a comprehensive library of theories which have specifically targeted at concepts of common programming languages, standard libraries, and specific applications.

The pivotal tools in the collection will be the program verifiers. They will exploit the discoveries of research into the theory of programming to transform a program and its specification into verification conditions that can be discharged by the logical tools.

2.3. Experiments. The project will deliver a repository of verified software, containing hundreds of programs and program modules, and amounting to over a million lines of code. The code will be accompanied by full or partial specifications, designs, test cases, assertions, evolution histories and other formal and informal documentation. Each program will have been mechanically checked by one or more of the program verifiers in the toolset. The verification of the code in the repository should consume a bit more than half of the total effort of the project.

The eventual suite of verified programs will be selected by the research community as a realistic representative of the full range of computer application. Realism is assured if the program itself (or perhaps an earlier less fully verified version of it) has actually been used in real life. In size, the individual entries into the repository should range from a thousand to a hundred thousand lines of code. Some of them will be written in declarative languages and some in procedural languages, and some in a combination of both. They will cover a wide range of applications, including smart cards, embedded software, device routines, modules from a standard class library, an embedded operating system, a compiler for a useful language (possibly smartcard

Java), parts of the verifier itself, a program generator, a communications protocol (possibly TCP/IP), a desk-top application, parts of a web service (perhaps Apache). The programs will use a variety of design patterns and programming techniques, including object orientation, inheritance, and concurrency.

There will be a hierarchy of recognised levels of verification, ranging from avoidance of specific exceptions like buffer overflow, general structural integrity (or crash-proofing), continuity of service, security against intrusion, safety, partial functional correctness, and (at the highest level) total functional correctness. Each claim for a specific level of correctness will be accompanied by a clear informal statement of the assumptions and limitations of the proof, and the contribution that it makes to system dependability. The progress of the project will be marked by raising the level of verification for each module in the repository. Since the ultimate goal of the project is scientific, the ultimate level achieved will always be higher than what the normal engineer and customer would accept.

In addition to verified programs, the repository will include libraries of bare specifications independent of code. These may include formal definitions of standard intermediate languages and machine architectures, communication and security protocols, interfaces of basic standard libraries, etc. There will also be libraries of application-oriented specifications, which can be re-used by future applications programmers in particular domains (such as railway networks). Some of these will be supported by automatic code generators, perhaps implemented within a generic declarative framework.

The actual contents of the repository will be as chosen by the research community during the course of the project. An overall total of one million lines of mechanically verified code will be a convincing demonstration of the success of the project; it will be sufficient to reveal clearly the capability and limitations of the concept of a program verifier and other associated formal programming tools; and it will give paradigm examples of their effective use.

There will remain the considerable task of transferring the technology developed by the project into professional use by programmers, for the benefit of all users of computers. This is where all the educational, social, commercial, legal and political issues will come to the fore. The scientists will have made their main contribution in demonstrating the possibility of what was previously unthinkable. Scientific research will continue to deliver practical and theoretical improvements at an accelerated rate, because they are based on the achievement of the goals of the Grand Challenge project.

3. The Grand Challenge project plans (5-20 years from now).

The Grand Challenge project will be planned in outline to take fifteen years, starting perhaps in five years' time, and requiring over a thousand person-years of skilled scientific effort. The project will be driven by the cumulative development of the repository of mechanically verified code. Its progress will be measured by a number of indicators: a count of the total number of executable lines of code that have been verified, the length of the largest verified module, the level of verification achieved, the range of applications that are covered, and the range of programming features that they exploit. According to these criteria, every program and module that is successfully verified will be celebrated as the first of a new kind. That will ensure that the repository eventually covers the necessary wide range of variation.

In order to accumulate a million lines of verified code, it will be essential to exploit all sources of experimental material, and to exploit and develop the capabilities of all available tools. Some of the code and specifications may be collected from critical projects that have already undergone a formal analysis. Some of the code will be developed by rational design methods from specification, using interactive specification checkers and development aids. Other code will be generated automatically, together with its assertions and its proofs, by special-purpose program generators, taking an application-oriented specification as input. Some of the code will be taken from existing class libraries for popular programming languages, and even from well-written open source application codes. In such cases, the specification and other annotations will need to be reverse-engineered from the code, using program analysis, assertion inference, and even empirical testing. In all cases, the resulting code must be checked mechanically against specification by a program verifier.

The experience gained in use of existing tools will be exploited to gradually improve the capability of the tools themselves, including their usability and their performance. For each tool that is mature enough for general use on the project, there will be a research and development group with adequate resources and authority to maintain and develop the tool, to provide support and education, and to control the impact of enhancements to the tool in the interests of existing and new users. On occasion, new tools will be developed, offering a step-change increase in capability and quality, and opening up opportunities for yet further advances. For the purposes of the project, the tools should be as usable by scientists as a routine scientific instrument like an oscilloscope. They can assume a high level of scientific understanding.

An over-riding concern in the development of tools, and in particular of the program verifiers, will be that the algorithms incorporated in the tools must be logically sound and consistent. The second concern is that they should approach more and more closely to completeness. These achievements will require significant contributions from theorists, who undertake to apply their skills in the context of the programs and languages actually included in the repository. For many of these, new concepts in specification, new type systems, and new design disciplines will need to be discovered; and existing knowledge will have to be adapted to the needs of mechanical verification. In some cases, the theorist will be able to test and evaluate new discoveries by making ad hoc changes in existing tools. But the wider testing of the developing theory will depend on the cooperation of tool-builders and tool support groups, who will incorporate new theory into existing and new tools.

Advances in the capability of proof tools will be driven primarily by the needs of their expanding range of users. The first demand of the users will be increased automation and improved performance of analysis and proof. This requirement is readily formalised and promoted by annual competitions, currently conducted by the theorem proving community; they have achieved fantastic improvements in the speed of decision procedures that are needed in the inner loop of all verifiers. In addition to improvements to the capability of individual tools, a strong user requirement will be for the inter-working of all the tools required in a verification experiment, so that each tool accepts the output of the others, and produces its own output in a form acceptable by others. The design of new competitions should be targeted towards this goal; the rules could require broader interfaces than the yes/no of decision procedures; and they could require that counter-examples and proofs be produced and checked by independent tools. In some cases, human inspection of the results may be desirable, to assess comprehensibility. Some competitions could be designed so that they can only be won by combining the merits of the latest versions of different tools. In this way each tool can continue to be developed independently, ensuring that independent improvements can be exploited immediately by other tool-builders and by all users of the tools. Perhaps the most stimulating competitions would be based on the popular programming competitions already set regularly on the web, by simply requiring that the resulting program be verified. A little more time may be needed at first, but the eventual goal would be to develop correct programs just as fast or faster than incorrect ones are today.

It is probable that the practical capability of proof tools will be increased by specialisation towards the needs of particular programming languages. For example, it would pay to incorporate the type system of the programming language directly into the logic of the proof tool. It would also pay to develop libraries of theories to cover specifically the features of the class library of particular languages in widespread use.

At any given time during the course of the project, the repository will contain all the code verified so far, which will be used repeatedly as regression tests for the evolution of the toolset, and as a measure of the progress of the project. In addition, the repository will contain a number of challenge problems, which have been selected by the research community as worthy candidates and ripe for verification. In the early stages, these challenges will include useful programs which already have specifications verified by existing technology, often by hand. For larger and later examples, the challenge may require construction or generation of code from a specification or vice-versa. Some of the problems will be large enough that they have to be solved collaboratively, by an agreed share-out of the work. Smaller problems of a more innovative kind may be set up annually as challenges for scientific competition.

It is the availability of a program verifier that will determine in what languages the initial programs and specifications will be written. A program verifier is a combination of proof tools with the semantics of a particular programming language. Examples of currently mature program verifiers are ESC/Java, Coq, and ACL2; and other proof tools are becoming available for this purpose. In the early stages, the project will be broadly inclusive of all approaches and languages. But the eventual goal will be that nearly all the tools will be readily applicable to nearly all the programs in the repository, and tool providers engaging in the project will collaborate to achieve this goal. Their idealism will be re-enforced by the users of the tools, who will certainly need tools that are highly interoperable; and tool support groups will compete to provide what the users want. Initially, inter-operation could be assisted by mechanical translators, which could be just targeted specifically at the particular challenge material that has been accepted into the repository. The problem of assimilating formats of incompatibly coded data bases is not peculiar to Computer Science – it is also a severe problem in astronomy, physics and biology, and other branches of e-science. The other sciences have exploited the repository concept to tackle this problem, and have allocated the necessary translation and co-ordination effort from their programming support groups.

In the course of time, agreement will be reached between tool support groups for the design and implementation of wider, more efficient, and more convenient interfaces between the tools that they support. In the end, the quality of each tool will determine the numbers of its users. Since users will tend to gravitate towards the most popular tools, it is quite likely that there will be less variety of notation at the end of the project than at the beginning.

We expect that the plans for the project will include a specification phase, an integration phase and an application phase, each lasting around five years. During the specification phase, a network of repositories will be established and populated with challenge specifications and codes. Some of the smaller challenges will be totally verified using existing tools, and some of the larger ones will be verified to a formally specified level of structural soundness and serviceability. A formal tool-bus will establish communication at the file level between the more widely used tools. During the second integration phase, the tools will evolve to exploit even closer inter-working, and performance will be improved by introduction of more inter-active interfaces between them. New tools will emerge as user needs become more apparent. Medium-sized programs will be fully verified and some of the largest programs will be verified to a high level of structural integrity. During the final phase, the pace of progress will accelerate. A comprehensive, integrated tool suite that is convenient to use will permit the collaborative verification of large-scale applications, culminating in achievement of the million-line goal.

4. Pilot projects (1-5 years from now).

The aim of a pilot project for a Grand Challenge is to gather evidence for the viability of the long-term goals, and to gather the experience needed to formulate intermediate goals and more detailed project plans. The pilot projects also lay the foundation for the actual conduct of the project, including recruitment and training of the initial cadres of scientists who will work on it. They do not require the same degree of log-term commitment or global co-ordination as the main project itself.

4.1. The repository. An early pilot project for this challenge would obviously be the establishment of an initial repository of suggested challenge material for formal verification. It could start by requesting contributions of any available specifications for existing software that can be put into the public domain. In some cases, this could be accompanied by the actual code, and perhaps any existing proofs, constructed earlier by hand. It would remain open to suggestions for further challenge problems, and for accumulating the results of the experimental research.

4.2. Service centres. A second class of pilot project would be to set up a network of support and service centres for the repository. The centres should have the manpower to curate the submitted material, and to perform any necessary notational conversions to ensure that the announced challenge codes will be amenable to processing by the available programming tools. Such centres could also provide an advisory service to local experimenters and users of the tools. Such a centre could also take responsibility for education; they could organise and deliver summer schools to train scientists in the use of currently available tools, and teach the underlying theories of correctness.

4.4. Tool inter-working. The methods and potential benefits and difficulties of tool integration can be explored by a number of ad hoc collaborations between the builders of any pair of tools, whether similar or complementary. There are examples of such collaborations now in progress.

4.5. Tool repository. For more systematic integration, the concept of a repository could be extended to support the maintenance and integration of suites of existing tools. Such a repository should be responsible for facilitating the combined use of multiple tools from the suite, and ensuring their applicability to the challenge problems. The staff of the tool repository could assist in negotiations of interfaces between tool providers, and they should have the manpower to help implement any standard interfaces that emerge.

4.6. Tool support centres. Another important class of pilot project start by the establishment of a recognised support centre for each of the major programming and proof tools that are likely to be used in the initial stages of the project. Such a centre should obtain long-term funding to enable it to discharge responsibilities to a growing community of users. Each centre should have resources to maintain its tool, to improve it in the light of user experience, to integrate it with the latest versions of other tools, to improve its usability, and to incorporate extensions to its functionality that have been suggested by theorists or by users.

5. The first steps (0-1 year from now).

On a long journey, the longest step is the first one. Here are some suggestions on how to take it. Each suggestion is followed by a personal question about your own possible role in getting the project started. I hope that this conference will help you to answer the questions in a positive way.

The first step for every participant is to think deeply whether your personal research goals are aligned to those of a Grand Challenge project; and then to consider what changes to your own research practices and priorities would you be prepared to make to maximise your own contribution to these goals. Then decide whether you are prepared to make these changes?   The pilot projects listed above are only rough examples. Can you suggest some more specific ones? Are there any that you would like to make the subject of your next research grant proposal?

These issues should be discussed with other members of each of the research communities whose expertise is necessary to the success of the project. A series of suitable occasions could be at the regular international conferences of your research community. Would you be willing to organise or attend a Workshop at the next conference that you attend; its purpose would be to discuss the general prospect and the detailed progress of this Grand Challenge? Or would you attend such a Workshop organised by someone else?

The most vital contributors to the project will be the experimental scientists, who apply the best available tools to the challenge problems accumulating in the repository, and contribute suggestions for the improvement of the tools. One way to assess and increase the enthusiasm of scientists in your own country or region would be to hold a Working Conference and/or a series of Workshops, to encourage local scientists to contribute their skill and their judgement to the planning of the project. Would you be prepared to sponsor or organise such a Conference? Or serve on the programme Committee?

The teams of experimental scientists will require education in the relevant theories, and training in the use of the best available tools. Would you be prepared to design and deliver Master's courses on program verification? Would it be appropriate to set Master's projects to verify small portions of the challenge material held in the repository?

Education in the technology should begin at the undergraduate level. Would you be prepared to teach the use of specifications and assertions as a routine aid to program testing, as they are currently being used in Industry? Would you use them yourself as an aid to marking the students' practical assignments?

The promises of a Grand Challenge project and the planning of their fulfilment must be based on a deep and wide appreciation of the current state of the art, and the technical and human resources available to advance it. Would you be prepared to draft a survey paper of the current state of the art in your subject, and an evaluation of its potential future contribution to the project as a whole? The paper could serve as an introduction to the relevant parts of the specialist background, for the benefit of those who will participate in the project.

The survey paper might be suitable as the first article in a special issue of an international technical Journal, which could be devoted to technologies relevant for the Grand Challenge Project? Who would you invite as other contributors to the special issue? Would you volunteer to edit such an issue yourself?

For tool-builders, a long-term goal of the project is that all the tools needed for verification of useful software will work conveniently together in an integrated toolset. This process has already been started by pair-wise coupling of proof tools. Can you identify a tool that could usefully be used in combination with yours, across some mutually agreed interface? Who would you approach to start discussion? Is the time ripe for design of interfaces to link larger sets of tools? Would you be prepared to discuss standard interfaces? Or initiate a discussion of them?

For theorists, a long-term goal of the project is that an integration of sound and general theories will be proved adequate for verification of actual programs that people want to write and use. These programs will be written in languages that people want to write their programs in. Elegant and general theories will have to be adapted and specialised to provide an axiomatic (correctness-oriented) semantics for an adequately large subset of existing languages. Further work will be needed to design analysis algorithms that check conformance to the disciplines of the subset. These will then have to be incorporated in tool-sets that are needed for the experimentation; and the results of experiment may require re-evaluation of the theory. Are you prepared to engage in the extra work required to adapt your theories for experimental verification or falsification?

Most of the funds that support research in Computer Science are provided by national funding agencies. In some countries, these agencies have a tradition of favouring shorter-term projects that will give competitive advantage to national industry. For a long-term international project, each agency will have to make a decision on behalf of the country whether and how far to engage in a Grand Challenge project, recognising the advantage of sharing the costs among all nations, in the expectation that the benefits are also shared among all. The modes and procedures for funding the necessary research proposals in some countries may have to be adjusted to accommodate the scale and duration of a Grand Challenge. Would you be able to raise and discuss these issues with your own national funding agency?

The most crucial decisions right from the start of the project will be decisions about the current and final contents of the repository. These decisions could be delegated to an editorial board representing the views and good judgement of the entire research community. The board will lay down criteria for acceptance of new material into the repository, and devise some prestigious mode of publication for the results of each successful verification. In due course, they or their successors would be responsible suggesting priorities, for issuing challenges, and for co-ordinating a reasonable split of the labour of verification among all volunteers. The board might request recognition from IFIP as a new Working Group, as recognition of their international standing and as an aid to obtaining support for regular meetings. Who would you like to see as a member of such a board? Would you be prepared to serve yourself for an initial period?

Conclusion.

Now is the best time to start this project. Since the idea of a program verifier was first proposed by Jim King in 1969, the state of the art has made spectacular advances. Theories are available for most of the features of modern programming languages. There is already considerable experience of manually proven design of programs currently running in critical applications. Machines are now a thousand times larger and a thousand times faster than they were then, and they are still getting more powerful. The basic technology of automatic proof has also made a thousand-fold advance, and the contributors to the conference show that this advance is also continuing. Finally, the number of computers, the number of computer programmers, and the number of computer users have each increased more than a thousand-fold. As a result, the annual cost of software error is many times greater than the total cost of the proposed scientific project over its entire fifteen-year timescale. The state of the art and the economic imperative convey the same message: the best time to start this project is now.

Acknowledgements: Tony Hoare has presented his original vision to a number of audiences over the last five years. Their comments, questions and criticisms have shaped this document. We are thankful to many individuals who have read and commmented on the preliminary draft of this document. We are thankful in particular to Jim Browne, Fred Chang, Rick Hehner, Jim Horning, Cliff Jones, Mathai Joseph, Bertrand Meyer, J Moore and Natarajan Shankar.

Appendix: Some frequently asked questions

1. What is to be the ultimate product of the Grand Challenge?

The ultimate product will be a range of scientifically based programming support tools to assist in the specification, design, development, testing, analysis, evolution of software systems. This will be accompanied by convincing evidence of its successful use in a wide range of computer applications. The essentially new tool, serving as a lynchpin of the entire toolset, will be a program verifier, whose central role will be to certify that software conforms to its stated specification. The wider application of this tool will help to reduce the high costs of programming error, currently estimated at tens of billions of dollars a year.

1.1. Will the program verifier certify all reasonably-sized programs automatically?

During the course of the project, the program verifier will be applied to a wide range of programs of reasonable size, which the research community has selected as representative of a useful range of computer applications. By the end of the project, this corpus of verified software will amount to more than a million lines of executable code, together with specifications, design history, assertions, test cases and other documentation. The science and technology incorporated in the tools will be generic, so there is good hope that success achieved on the million-line corpus will generalise to previously unseen programs of a similar kind.

The ultimate aim of the project is that the program verifier will certify all programs automatically. In practice, at any given time there will always be a percentage of verification conditions that the verifier cannot prove. For these, the traditional techniques of testing and run-time checking will be employed. The project will measure the percentage of failed conditions, and seek to reduce it, even after the tool is accepted as adequate for practical use.

1.2. If the verifier is not fully automatic, how much human intervention will be needed?

In the initial stages of the project, there will be many verification conditions (perhaps as much as 5% of the total) that cannot be proved automatically. These will be published to serve as a challenge for development of the technology of theorem proving, and the development of specialised libraries of theorems that are targeted at verification of real programs. In the course of the project, the necessary human intervention will be supplied by experimental scientists engaged in the verification of the corpus of challenge codes. When the verifier is more widely used, its more serious failures will still require assistance from a specialist in verification. The tool will not be used in areas where these interventions are found to be too frequent.

1.3. What kinds of properties will it be able to certify?

The verifier will be capable in principle of certifying any properties that can be formally specified in the relevant subset of mathematical notation. Such properties will include avoidance of software-generated computer crashes, resistance to intrusion from unauthorised sources, observance of privacy constraints, conformity to internal interface conventions. In more critical applications, safety properties will be specified and verified, and in some cases a full behavioural specification will be formalised.

Many important properties are resistant to formal specification, for example ‘user-friendliness’, early delivery, extensibility. Other properties will be too expensive to formalise; and others will be so complicated that their formalisation is no more obviously correct than the program itself. The decision on how much effort to expend on formalisation of properties will have to be taken by the individual engineer in the light of the circumstances of the individual project. These decisions will limit the applicability of the verifier more substantially than any failure of full automation.

1.4. Will it be too onerous for the average programmer to supply the necessary assertions and the lemmas needed for their proof?

Yes indeed. Initially it will be only skilled scientists who can perform this task. As the project proceeds, the verifier will be able to deduce many of the trivial but important assertions from the text of the program itself. Furthermore, the scientists will accumulate libraries of useful theories and theorems, so that that most of the necessary lemmas can be supplied automatically. Finally, the availability of a capable program verifier will motivate the more intellectually lively programmers to use it , and it will support the education of the next generation of programmers.

1.5. Will programmers have to learn a new language?

No; this is unlikely, and fortunately unnecessary. Certainly, scientists will continue to use and invent new languages to serve as scientific work-benches for exploring the theories under investigation, as it were under laboratory conditions. But the theories will also be extended and adapted to deal with languages in which the corpus of challenge material is expressed. In some cases, only a recognisable subset of the programming language will be verifiable.

Nevertheless, one can foresee ways in which the use of existing languages may gradually change. Programmers will be encouraged to avoid certain combinations of programming language feature which significantly complicate correctness reasoning. They will be encouraged to follow certain well-structured system design patterns, which permit verification by simpler proof patterns. And we may hope to see more program generators, which automatically generate parts of a system from their more abstract specifications.

In this way, it is likely that existing programming languages will evolve further in the direction of support for efficient construction of reliable programs. Such evolution will be based on scientifically supported theories that emerge from this Grand Challenge project.

1.6 Would the verifier be partially available before the 15-year time-line?

Yes; in fact the project will start with verifiers that are already available, but somewhat limited in capability. The main task of the project will be constantly to test and extend their capability. During the course of the project, the primary users of the tools will be the scientists engaged in the project; their task is to test current tools on a corpus of representative software, and determine how they can be improved. The tools will be structured to permit continuous improvement. They will be sufficiently convenient for use by specialised scientist, but probably not for use by the professional programmer.

In the later stages of the project, the emergent scientific understanding will be readily transferable to the developers and suppliers of widely used commercial programming tool-sets. They will be responsible for integrating the technology with current program development environments, for enhancing the convenience of their use, and for marketing and selling the results. That is a reasonable division of labour between the scientific researchers and the commercial exploiters of their discoveries.

2. By what technical means is this goal to be achieved?

By collaborative endeavours of scientists from many specialities, who broadly agree on a common outcome, and on the intermediate steps to achieve it in a given timescale. They will work towards agreement on notations for programs and specifications, and on the formalisation of interfaces between tools of various kinds, and on the selection and accumulation of an agreed corpus of representative programs to test the tools. Theorists will work together to ensure that their theories can be applied in combination, and they will adapt their theories for application to the languages in which the programs of the corpus are expressed. Tool-builders will build select theories suggested by others, and build them into their tools; and they will undertake to supply tools of sufficient stability and convenience for use by experimental scientists. And finally, experimentalists will apply the tools to the programs in the agreed corpus.

2.1 Is it mainly a question of improving current tools?

Certainly, it is inevitable the initial plans for the project should be based on what already exists. The plans envisage a continuous process of evolution of tools in the light of experimental evidence obtained by their community of users. General experience of software evolution suggests that this is an excellent driver of progress.

During the course of the project, the possibility of emergence of entirely new ideas and tools must be encouraged and welcomed. But by definition it cannot be planned for.

2.2. How can we achieve tool integration, when they have different interfaces, internal data structures, and search methods?

Initially, the goal should be inter-working of tools rather than integration. Inter-working involves translation of formats and data structures whenever information is required to pass between the separate tools. The essential advantage of inter-working is that it permits continued evolution of tools on both sides of the interface.

As understanding of the basic technology matures, and as its implementation stabilises, further efficiency, capability, and convenience of use may be achievable by closer integration of tools. Certainly, closer integration will be a necessary concomitant of actual commercial products.

2.3. Is there any grounds for optimism that combination of inadequate tools will be found to be adequate?

The use of tools in combination is essential to the success of the project. Without a full toolset, it will be impossible to reach the goal of a million lines of verified code. Inter-working is required to make combined tool use reasonably convenient.

There are analogies that suggest that inter-working tools can evolve to cover each others' inadequacies. The main ground for optimism is that the individual tools can continue to be improved separately in the light of their substantial use in experiments.

3. How can different groups manage to co-ordinate their activities?

There is plenty of experience of large-scale collaboration among scientists: astronomers build telescopes, physicists build particle accelerators, space scientists build satellites. In all cases, co-ordination is planned by the recognised leaders of the scientific community actually engaged in the project.

> *3.1. Will there be top-down management which will address standardisation of notation, interfaces for inter-operability, and directions of research?*

In Computer Science, there is less tradition of such a large-scale collaboration as in nuclear physics and astronomy. Consequently, the process of agreement should probably begin with small-scale and relatively short-term agreement among two or three groups engaged in related aspects of the project. When the major contributors to the project have identified themselves, it is likely that wider and more substantive issues will be discussed at regular meetings by the community as a whole.

> *3.2. Can individuals and groups who do not identify themselves as participants still contribute to the effort?*

Yes definitely. New ideas and breakthroughs are more likely to be originated by scientists who (with no need for justification) prefer to work independently of any long-term large-scale project. There will be many who prefer to conduct their research in close collaboration with an on-going engineering or commercial product development. Although the primary responsibility of such scientists is for the timely success of the particular project, the most original and general ideas are often inspired by practical needs.

Those engaged in the project must watch carefully for such advances by non-participating scientists, and be eager to embrace them as quickly as possible. How else can the challenging goals be achieved?

> *3.3. Who will fund this work?*

Funds for an international scientific project are usually contributed by the scientific funding agencies of the nations engaged in the project. The means of obtaining initial funds, and the assurance of reasonable continuity of funding in later years, will have to be planned individually by scientists resident in each country.

In Britain, the Prime Minister has stated that the goals of the national funding of science are:

1. the maintenance of national scientific prestige on the international scene.
2. the promotion of national commercial interest.

Contributions to an internationally recognised Grand Challenge project are a definite source of national prestige. And a scientific project that advances a generally applicable technology is an excellent nursery for technically advanced entrepreneurs, who will adapt and apply the technology in a specific marketplace. Finally, the main benefits of the project will be in the reduction of cost and improvement of the quality of computer software. These benefits will be shared by all nations.

> *3.4. Will there be an effort to get umbrella funding?*

For international projects in Astronomy and Nuclear Physics, funding agencies in the participating countries undertake to bear some agreed proportion of the costs over an extended timescale, without specifying in advance exactly what the funds will be spent on. The actual spending is decided by the scientists, as the project progresses. This is possible, because the funding agencies have reasonable confidence that the scientists involved can collaborate effectively to produce the results that they promise.

In Computer Science, such collaboration would be unprecedented; but if the early achievements of the project give grounds for the requisite confidence, an effort to get umbrella funding would be entirely appropriate. The important criterion is that the attempt to obtain funding should not dilute or divert the pursuit of the pure scientific goals of the project.