Dependable systems evolution

A grand challenge for computer science

1 Introduction

This document is a call for action on the part of the strong software engineering research community. The call is to select a programme of scientifically relevant projects, and to form multi-national teams for their implementation. Team formation and planning could last two to five years, and the programme itself could last up to fifteen years. The action will enjoy enthusiastic support from the general community, as well as participation from suitably sized groups of specialists. The immediate objective is to suggest a plan for a workshop to be held in November 2003, to test the possibility of the formation of teams, and work towards clarification of the work-plans of each team.

2 The vision

Society’s dependence on computing systems is increasing, and the consequences of their failures are at best inconvenient; in certain application areas, they may also lead to loss of financial resources, and even loss of human life. A computing system is dependable if reliance can justifiably be placed on the service that it delivers, characterised in terms such as functionality, availability, safety, and security. Evidence is needed in advance to back up any manufacturer’s promises about a product’s future service, and this evidence must be scientifically rigorous. At the moment it is very expensive and difficult to produce such evidence: exhaustive testing is usually out of the question, and the application of mathematical techniques for high assurance is usually effective, but extremely costly.

This is compounded by the need for practical computing systems to evolve in response to changes in their requirements, technology, and environments, without compromising their dependability. For example, an avionics systems will have its processor upgraded several times during the life-time of the airframe; and a telephone system will be continually upgraded with new features. There are even applications where system boundaries are not fixed and are subject to constant urgent change. These applications are typically found in emergent organisations, which are always in a state of continual process change, never arriving, always in transition. These may be e-businesses or organisations that continually need to reinvent themselves to gain competitive advantage. For example, stock-brokers often need to introduce a new service overnight; the service may exist for only another 24 hours before it is updated. As an extreme case of evolution, we will provide the scientific and technological means to be able to create dependable computing services out of components existing at the moment when the service is required. In all these applications, from high assurance to emergent systems to just-in-time services, the reliability required of the system is just as great after the change as it was before.

We need the scientific foundation to be able to build systems whose dependability can be justified, even in the face of the most extreme threats. We need to be able to put systems in inaccessible places, knowing that they will continue to work over decades. We need to be able to build very large scale systems with controllable costs and risks. We need the ability to evolve such systems rapidly, at costs which reflect the size of change, not the scale of the system. We seek a change in culture, where suppliers sell software for its safety, security, and reliability, as well as for its functionality.

The scientific and technical advances that we hope will result from this project could be the basis and trigger of a radical change in the practice of software engineering. Perhaps in the future:

- Commercial and industrial-scale software can be developed to be truly dependable, at lower cost and with less development risk than today.
• The vulnerabilities in legacy systems and COTS components can be discovered and corrected, improving their dependability.
• Dependable systems can be evolved dependably including, for a class of applications, just-in-time creation of required services.

Within fifteen years the project will produce prototype tools and examples of their successful use that are sufficiently persuasive to encourage commercial tools vendors and their customers to make these improvements. The choice of early prototype tools and the range of their experimental use are yet to be determined by the participating scientists.

3 The tools

The realisation of our vision will depend on the development of a powerful set of tools for strong software engineering. These might include the following.

• **Software system specification and model-based software development/architecture**: Tools to construct specifications including timing, safety, security, and resource-usage properties of systems. Tools for modelling and analysis to enable architectures to be developed that satisfy both functional and non-functional requirements.

• **Automated verification of software properties**: Combined model-checking and theorem-proving tools to enable fully automated assessment of software properties. Tools for supporting correctness-preserving refinement from specification to implementation and incremental (component-based) verification.

• **Automated testing of software properties**: Tools for automated generation of test cases and test oracles to demonstrate that software meets its functional and non-functional requirements. Extension of existing automatic test-bench generation tools used during hardware design to demonstrate that software meets its functional and non-functional requirements.

• **Automated dependability analysis**: Tools to generate dependability analyses automatically from software specifications and hardware failure properties.

• **Heuristic design synthesis**: Tools for heuristic approaches, such as simulated annealing and genetic algorithms, to synthesise designs from functional and non-functional specifications.

Although the choice of prototype tools has yet to be made, it may include the following exemplars.

• **Invariant generators** are program analysers that discover putative program invariants. The properties of interest may be restricted in order to get a high degree of automation.

• **Verifying compilers** give high assurance of the correctness of the programs that they compile. Consistency of the program with its specification is established automatically by a combination of program analysis, type inference, model checking, decision procedures, proof search, test case generation, and any other method that justifies increased trust in the quality of the program.

• **Refinement tools** guide the systematic production of designs and code from specifications using special-purpose design calculi.

We consider a work-plan for one of these exemplar projects in the appendix.

4 The criteria

Sixteen criteria have been proposed to judge the maturity of a grand challenge, and we consider each of them in this section, applying them to each of the exemplars, individually or in combination.
4.1 Scientific significance

▷ Is it driven by curiosity about the foundations, nature, or limits of basic Science?

The correctness of computer programs is the fundamental concern of the theory of programming and of its application to software engineering. Much is understood about how to verify the correctness of functional properties of modest systems. The limits of application to large-scale systems will be explored and extended, especially in the treatment of non-functional properties, such as safety, security, responsiveness, and locality, and the cost-effective development and evolution of industrial-scale systems. Many non-functional properties can be formally described so that they are unambiguously testable, and so are equally amenable to scientific method as functional properties. The limits of mechanisation will also be explored.

▷ Are there clear criteria for the success or failure of the project after fifteen years?

Where practical, scientific method will be used to evaluate the project's results: before the project starts, we will design experiments to try to refute claims made about how our methods and tools increase system dependability. These experiments will, by their very nature, have clear criteria for success and failure.

If the project is successful, then its technology will become standard practice for the development of dependable systems. For example, a prototype strong software engineering tool-set will be widely available, including support for powerful static analysis of mainstream language dialects to show a program's conformance to a range of assertions and other partial specifications. It will have been tested in the verification of certain desirable properties of millions of lines of software; these properties that those that are desirable of any system, such as deadlock freedom or absence of nil-pointer dereferencing. It will have been tested in the more substantial verification of critical parts of it, leading to the removal of thousands of anomalies in widely used code. Some of this work will have been carried out on open source software, so the results will be widely visible and open to refutation. Exemplars and prototype products will be developed within the project, and evaluations will be published in the scientific literature. New dependable products will have been developed by industry, and their adoption will be widespread.

▷ Does it promise a revolutionary shift in the accepted paradigm of thinking or practice?

At present, the most widely accepted means of raising the levels of trust in software is by massive and expensive testing, which often fails to produce the dependability required by users. Availability of effective software development tool-sets will encourage software engineers to formulate specifications in advance of code, and many of them will be verified by mathematical techniques. Experience of the verified development of safety-critical code will be transferred to commercial software with mass markets. The Grand Challenge offers the opportunity for a shift from the current, largely manual development of industrial-scale systems to a situation where development is largely automated. This new approach will be faster, and much more predictable in time and cost. This will mark the maturity of software development as an engineering discipline.

▷ Does it avoid duplicating evolutionary development of commercial products?

At present, large companies manage their products through evolutionary development, producing software that is not always adequate for the job, with users frequently encountering problems. In this climate, guarantees of dependability are seldom offered and consequent liabilities not accepted. With the success of our work, we will break away from this institutionalised situation. No single commercial company could contemplate carrying out this work and making its results freely available; nor would they have the technical competence to do so. It is inconceivable that the integration of techniques would take place without the cohesive drive behind a Grand Challenge. Few of the individual areas will be addressed by commercial tool vendors, as they will not see sufficient economic benefits from advances made in isolation. The Grand Challenge offers a unique opportunity for theories to be implemented in prototype tools, and for these tools to be used on realistic case studies, where theorists, tool-builders, and users come from different countries.
4.2 Impact on practice

▷ Will its promotion as a Grand Challenge contribute to the progress of Science?

The project proposes to go far beyond the state of the art in the development of dependable systems. This will require significant scientific advances in the theory of computation, particularly in the treatment of non-functional properties, as well as significant engineering advances in large-scale modelling and mechanical reasoning. It will require the consolidation of decades of research in theoretical computer science and software engineering, unifying theories that have to work together, identifying gaps in the range of existing theories.

Just as important is the progress of engineering and its impact on practice. We need to extend the successful approaches that have worked in limited domains (e.g., SPARK) to encompass the most widely used languages; we need to develop methods and tools that support rapid evolution with controlled dependability; and we need to evaluate these methods and tools scientifically. This will involve analysis of existing languages to provide the necessary strong semantics, and the development of stronger theories for composition and evolution. It will involve the use of design patterns and program generators to enable users to exploit higher level concepts and features of languages that have been developed and tested in the laboratory.

▷ Does it have the enthusiastic support of the established scientific communities?

The community comprises the following.

- **Domain experts** will offer challenges for work on particular practical problems.
- **Researchers** in many disciplines, including programming theory, dependability, software evolution, testing, and empirical software engineering, will need both to advance research their own topics and to collaborate together. For example, researchers in programming theory will accept the challenge of extending proof technology to programs written in industrial languages. They will need to design program analysis algorithms to check whether actual programs observe the constraints that make each theoretical proof technique valid. Researchers in empirical software engineering will need to work with all research groups to determine critical tests of the proposed technologies.
- **Tool builders** will include analysis capabilities earlier in the software life-cycle, to explore the range of their application to real code.
- **Experts in design patterns** will extend their use to legacy software.
- **Users** who are willing to try out the experimental prototypes, or allow them to detect and record the behaviour of the software that they use.
- **Regulators** will contribute to understanding the requirements for assessment and certification of dependability.
- **Teachers and students** of the foundations of software engineering will be enthused by various projects associated with the challenge, so contributing to the success of a world-wide project. For example, they may take part in annotating and verifying a small part of a large code base.
- **Researchers** from other disciplines will offer their specialist expertise. For example, psychologists and sociologists will help us understand how people contribute to system failures, and lawyers and business managers will explain the legal and marketing implications or our work.
- **All computer users** feel the frustrations of undependable software. This includes the scientists involved in this project, who accept responsibility for the problem and for solving it.

Support has already been canvassed amongst these communities.

▷ Does it appeal to the imagination of the general public?

All computer users have been annoyed by bugs in mass-market software, and will be concerned by the threats of bugs in critical software; they will welcome their reduction or elimination. Recent well-known viruses have been widely reported in the press, and estimated to cost billions of pounds. Fear of cyber-terrorism is widespread. The interest of the public can be maintained as dangerous
errors are detected and removed from software in common use. They will be reassured by the
gress towards achieving surety in safety-critical systems, the ability to deliver them on time,
and to adapt them quickly to evolving needs. Trustworthy software is now recognised by major
vendors as a primary long-term goal, and given recent problems in the UK Passport Office and Child
Support Agency, and the loss of the Mars missions in 1999 and 2000, for example, our goals should
be comprehensible to the general public.

\- What kind of long-term benefits to science, industry, or society may be expected?

This project represents a realistic attempt to reduce the £60 billion annual global cost of unreliable
software. It represents a significant opportunity for a a large-scale demonstration of the practical
benefits and pay-back from advances in theoretical computer science and software engineering. We
look forward to the day when normal commercial software will be delivered with a high chance,
perhaps eighty percent, that it never needs recall or correction within ten years of 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.

The success of producing just-in-time services from COTS components promises a new economic
model for software, replacing the cost of ownership with pay-per-use.

4.3 Scale and distribution

\- Does it have international scope?

The project has found enthusiastic support from leading researchers in Australia, Brazil, Canada,
China, India, Japan, the USA, and many European countries, including Denmark, Finland, Germany,
and the Netherlands.

\- How does the project split into sub-tasks or sub-phases, with identifiable goals and criteria?

The project is organised as three sub-tasks; rigorous scientific experiments will be conducted to
gather evidence for their success or failure. Each sub-task will contribute towards the development
of the strong software engineering tool-set.

1. Legacy and COTS systems: This sub-task will study legacy systems to develop tools and
techniques to justify and improve their dependability; experience gained working on existing
code will inform the work on new systems. We will accomplish the verification of some non-
trivial properties of a major legacy system.

   Tools: Tools are needed in this sub-task to generate a specification from legacy code, and
   then to demonstrate that the code is correct with respect to this specification.

2. Dependable development: This sub-task will develop tools and techniques to assure the
dependability of new systems by construction. We will develop dependable systems as exemplars
of our approach; these may include the development of a trustworthy European electronic
evoting system and the control system for an unmanned, autonomous, flying vehicle. A key
problem is to understand the dependability requirements—dependability by construction is
great but only if the right thing is constructed.

   Tools: Tools are needed in this sub-task to help guide the design process, and to verify and
   validate the systems being developed.

3. Evolution and just-in-time: This sub-task will study existing evolving dependable systems
with dynamic requirements, and develop tools and techniques for maintaining dependability in
the face of continual change. We will carry out the verification of the evolution of a dependable
system with an existing, certified justification.

   Tools: Tools are needed in this sub-task to support incremental, component-based verifica-
tion; that is, they analyse only what has changed during an evolutionary step.

Towards the end of the project, a number of prototype products will be developed to act as ex-
periments to judge the success of the overall project. These products may include a national medical
record system and an aircraft flight-control system.
What calls does it make for collaboration of research teams with diverse skills?
Contributions are needed from all the scientific communities mentioned on page 4. The scientific programme requires collaboration from researchers in dependability, safety, security, and programming theory, and the builders of testing tools, model checkers, and theorem provers.

How can it be promoted by competition between teams with diverse approaches?
The annotated libraries of open source code will be good competition material for the teams constructing and applying test and proof tools. Proofs will be subject to 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. Exemplars and prototype products will be developed by rival teams, and their results compared. Scientists will compete to strengthen the level of dependability of particular software items from just structural integrity (crash-proofing) to non-functional and functional properties.

4.4 Timeliness

When was it first proposed as a challenge? Why has it been so difficult so far?
The difficulty of achieving dependability has been recognised ever since we started to program computers. Due to our ambitions and to continual technological development, systems are becoming ever-more complex, compounding the problems of dependability. In practice, with the exception of safety critical systems, software product quality is often compromised with tight time to market constraints; unfortunately it has become common practice to fix software (bugs) via patches later in the product life cycle.

Achieving dependability when a system is evolving is the really intellectually challenging part: if it takes longer to verify and validate a system than the time between changes, then dependability is severely compromised. Currently, the problem is that the cost of assurance is proportional to the size of the whole system; what we want is for it to be proportional to the size of the change.

Why is it now expected to be feasible in a ten to fifteen-year time-scale?
There is both market-pull and technology-push: society's need for dependable software is greater than ever before; and the results of decades of research are now ready for exploitation. The greatest obstacle to producing dependable software has been the lack of effective tool support for verification. Significant progress has been made recently in model checking, SAT checking, and theorem proving. Advances in unifying theories of programming suggest that many aspects of the correctness of difficult programming language features, such as concurrent and object-orientation, may be expressed by simple specifications.

What are the first steps?
It will be necessary to assemble a working group of international leaders in the field to promote and guide the project. This will start with a workshop to discuss the initial technical programme; subsequent interaction will be through a conference series and a new journal.

The existing corpus of Open Source Software can easily be parcelled out to different teams for analysis and annotation; and the specifications can be checked by massive testing in advance of the availability of adequate proof tools. There will be several international consortia with different time-scales that work on different languages and code bases.

What are the most likely reasons for failure?
The annotation and verification of existing code is at present not a well-regarded research achievement. This essential part of the project may fail to attract good researchers. The low quality of existing software, and its low level of abstraction, reinforced by the use of legacy languages, 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 of them may be just a failure to make explicit a more or less obvious precondition. In other cases, an anomaly may be essential to the functionality of
the software. Often the details of functionality of interfaces, human or hardware, are not worth formalising in a specification.

These engineering concerns are always likely to place a boundary on the applicability of formal analysis in software engineering. It is the engineering goal of our project to push back the boundaries as far as possible. It is the scientific goal to show that there are no bounds at all to what is in principle achievable.

In any case, a significant group of the scientific community is keen to work together towards these long-term goals. It is their scientific idealism that will drive the project through the many practical difficulties that lie in its path.

A References for the state of the art

These references are based on those found in Cla96c, which contains a survey of the state of the art in the theory and practice of formal methods.


Arch90 G. Archinoff et al. 1990. Verification of the shutdown system software at the Darlington Nuclear Generating System. In International Conference on Control and Instrumentation in Nuclear Installations (Glasgow, Scotland, May).


Lam84 L. Lamport 1984. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 672–923.


Que82 J. Queille and J. Sifakis 1982. Specification and verification of concurrent systems in CAESAR. In Proceedings of Fifth ISP.


Rus96 D. Russinoff 1996. A mechanically checked proof of the correctness of the AMD K5 floating-point square root algorithm.


SPC93 Consortium requirements engineering guidebook. Technical Report SPC-92060-CMC version 01.00.09, Software Productivity Consortium, Herndon, VA.


B Towards a work-plan for an exemplar project

B.1 The verifying compiler

A verifying compiler [Lei98] uses automated mathematical and logical reasoning methods to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations that are associated with the code of the program, often inferred automatically, and increasingly often supplied by the original programmer. 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 only limit to its use will be set by an evaluation of the cost and benefits of accurate and complete formalization of the criterion of correctness for the software.

An important and integral part of the project proposal is to evaluate the capabilities and performance of the verifying compiler by application to a representative selection of legacy code, chiefly from open sources. This will give confidence that the engineering compromises that are necessary in such an ambitious project have not damaged its ability to deal with real programs written by real programmers. It is only after this demonstration of capability that programmers working on new projects will gain the confidence to exploit verification technology in new projects.

Note that the verifying compiler itself does not itself have to be verified. It is adequate to rely on the normal engineering judgment that errors in a user program are unlikely to be compensated by errors in the compiler. Verification of a verifying compiler is a specialized task, forming a suitable topic for a separate grand challenge.

B.2 The criteria

This proposed grand challenge is now evaluated under a relevant selection of the standard headings suggested for evaluation of a Grand Challenge Project.

B.2.1 Historical

The idea of using assertions to check a large routine is due to Turing [Tur49]. The idea of the computer checking the correctness of its own programs was put forward by McCarthy [McC63]. The two ideas were brought together in the verifying compiler by Floyd [Flo67]. Early attempts to implement the idea [Kin69] were severely inhibited by the difficulty of proof support with the machines of that day. At that time, the source code of widely used software was usually kept secret. It was generally written in assembler for a proprietary computer architecture, which was often withdrawn after a short interval on the market. The ephemeral nature and limited distribution for software written by hardware manufacturers reduced motivation for a major verification effort.

Since those days, further difficulties have arisen from the complexities of modern software practice and modern programming languages [Str85]. Features such as concurrent programming, object orientation and inheritance, have not been designed with the care needed to facilitate program verification. However, the relevant concepts of concurrency and objects have been explored by theoreticians in the 'clean room' conditions of new experimental programming languages [iga99, Has03]. In the implementation of a verifying compiler, the results of such pure research will have to be adapted, extended and combined; they must then be implemented and tested by application on a broad scale to legacy code expressed in legacy languages.
B.2.2 Feasible

Most of the factors which 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 [Ste00, Gal98].

2. The corpus of Open Source Software [http://sourceforge.net] is now universally available and used by millions, so justifying almost any effort expended on improvement of its quality and robustness. Although it is subject to continuous improvement, the pace of change is reasonably predictable. It is an important part of this challenge to cater for software evolution.

3. Advances in unifying theories of programming [Hoa98] 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 which are needed to underpin correctness proofs for systems involving concurrency and pointer manipulation have now been developed for use in optimising compilers [Ruf95].

5. Theorem proving technology has made great strides in many directions. Model checking [Hol91, Ros94, Mus02, Sha97] is widely understood and used, particularly in hardware design. Decision procedures [Gor88] are beginning to be applied to software. Proof search engines [Sha96] are now well populated with libraries of application-dependent theorems and tactics. Finally, SAT checking [Mos01] promises a step-function increase in the power of proof tools. A major remaining challenge is to find effective ways of combining this wide range of component technologies into a small number of tools, to meet the needs of program verification.

6. Program analysis tools are now available which use a variety of techniques to discover relevant invariants and abstractions [Bal01, Nim02, Fla01]. It is hoped that these will formalize at least the program properties relevant to its structural integrity, with a minimum of human intervention.

7. Theories relevant for the correctness of concurrency are well established [Mil99, Ros98, Cha88]; and theories for object orientation and pointer manipulation are under development [OHc01, Hoa99].

B.2.3 Co-operative

The work can be delegated to teams working independently on the annotation of code, on verification condition generation, 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. A compiler that exposes the syntax tree would enable 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 specialized theories [Gor88]; these can be developed by many hands to meet the needs of each application. In particular, proof procedures can be developed that are specific to commonly used standard application programmer interfaces for legacy code [Ste94].
B.2.4 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. They will be pioneers in the collaborative implementation and use of a single large experimental device, following a tradition that is well established in Astronomy and Physics but not yet in Computer science.

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 carry out experimental implementation of the hypotheses originated by theorists; following practice in experimental branches of science, their goal is to explore the range of application of the theory to real code.

3. Sympathetic software users will allow newly inserted assertions to be checked dynamically in production runs, even before the tools are available to verify them.

4. Empirical Computer Scientists will apply tools developed by others to the analysis and verification of representative large-scale examples of open code.

5. Compiler writers will support the proof goals by adapting and extending the program analyses currently used for optimisation of code; later they may even exploit for purposes of further optimization the additional redundant information provided with a verified program.

6. 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 meet the needs of actual legacy software and its users.

7. Teachers and students of the foundations of software engineering will be enthused to set student projects that annotate and verify a small part of a large code base, so contributing to the success of a world-wide project.

B.2.5 Incremental

The progress of the project can be assessed by the number of lines of legacy 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, specification of total correctness. The relevant levels of verification are: by testing, by human proof, with machine assistance, and fully automatic. Most software is now at the lowest level: integrity verified by massive testing. It will be interesting to record the incremental achievement of higher levels by individual modules of code, and to find out how widely the higher levels are reasonably achievable; few modules are likely to reach the highest level of full verification.

B.3 References


Nip02 See http://www.fbi.gov/congress/congress02/nipc072402.htm, a congressional statement presented by the director of the National Infrastructure Protection Center.
Nis02 Planning Report 02-3. The Economic Impacts of Inadequate Infrastructure for Software Testing, prepared by RTI for NIST, US Department of Commerce, May 2002


