

State of the Challenge

GC6: Dependable Systems Evolution

*GC6 Steering Committee**

17th of May 2004

Background

A new technological product or system is dependable if, from its initial delivery, it justifies the confidence of its users in its soundness, security, and serviceability. When the design of a system evolves to meet new or widening or changing market needs, it should remain as dependable as on initial delivery. Dependability in the face of evolution is currently a significant challenge for software systems, on which society increasingly depends for its entertainment, comfort, health, wealth, and survival.

To meet this challenge we propose to elucidate and develop the basic scientific principles that justify confidence in the correct behaviour of software systems, even in the face of extreme threats. We propose to exploit these principles in co-operative development of a strong software engineering toolset to assist in design, development, analysis, testing, and formal verification of computer software systems of any size throughout their evolutionary life. We propose to evaluate and improve the tools by competitive testing against a representative range of challenge problems taken from real computer applications. And we propose to assist the producers of commercial programming toolsets to adopt the improved verification technology in their widely marketed products. We hope to achieve these goals in a fifteen-year international project.

If the project is successful, it could lead to a revolutionary long-term change in the culture of the producers and users of software. The users of software will expect and demand guarantees of correctness of software against agreed specifications, and perhaps will be entitled by law to do so. The technology of verification will be incorporated in commercial toolsets, and their use will enable software producers to meet their guarantees. Software will be developed and evolve faster and at significantly less cost than at present, because verification will reduce the need for lengthy debugging and regression testing of each change. A recent estimate has put the potential savings in 2003 to the US economy alone at up to 60 billion US dollars per year.

Our ultimate goal is to meet the long-standing challenge of the verifying compiler. Its design will be based on answers to the basic questions that underlie any branch of engineering or engineering science, which ask of any product: what does it do (its specification), how does it work (its internal structure and interfaces), and why does it work (the basic scientific principles). We will be inspired by the ideal of total correctness of programs, in the same way as physicists are inspired by the ideal of accurate measurement and chemists by the ideal of purity of materials—often far beyond the current needs of the market-place. We hope that these ideals will be attractive to young scientists entering the field, and that each step in their

*Keith Bennett (Durham), Juan Bicarregui (Rutherford), Tony Hoare (Microsoft), Cliff Jones (Newcastle), John McDermid (York), Peter O'Hearn (Queen Mary), Brian Randell (Newcastle), Martyn Thomas (MTA), Jim Woodcock (Kent)

achievement will be welcomed by professionals, teachers, and scientists alike as a milestone in the development of the discipline of Computer Science.

Research Agenda

Our Grand Challenge has two central principles: theory should be embodied in tools; and tools should be tested against real systems. These principles are embodied in our two main current activities: the creation of a strong software engineering tool-set (*the verifying compiler*); and the collection of an appropriate range of examples (*the challenge repository*).

The Verifying Compiler

A verifying compiler is a tool that proves automatically that a program is correct before allowing it to run. Program correctness is defined by placing assertions at strategic points in the program text, particularly at the interfaces between its components. These assertions are simply executable truth-valued expressions that are evaluated when control reaches a specific point in a program. If the assertion evaluates to false, then the program is incorrect; if it evaluates to true, then no error has been detected. If it can be proved that it will *always* evaluate to true, then the program is correct, with respect to this assertion, for all possible executions. These ideas have a long history. In 1950, Turing first proposed using assertions in reasoning about program correctness; in 1967, Floyd proposed the idea of a verifying compiler; and in 1968, Dijkstra proposed writing the assertions even before writing the program.

Early attempts at constructing a verifying compiler were frustrated by the inherent difficulties of automatic theorem proving. These difficulties have inspired productive research on a number of projects, and with the massive increases in computer power and capacity, considerable progress has been made. A second problem is that meaningful assertions are notoriously difficult to write. This means that work on a verifying compiler must be matched by a systematic attempt to attach assertions to the great body of existing software. The ultimate goal is that all the major interfaces in freely available software should be fully documented by assertions approaching in expressive power a full specification of its intended functionality.

But would the results of these related research projects ever be exploited in the production of software products used throughout the world? That depends on the answers to three further questions. First, will programmers use assertions? The answer is yes, but they will use them for many other purposes besides verification; in particular, they are used to detect errors in program test. Second, will they ever be used for verification of program correctness? Here, the answer is conditional: it depends on wider and deeper education in the principles and practice of programming, and on integration of verification and analysis tools in the standard software development process. Finally, is there a market demand for the extra assurances of reliability that can be offered by a verifying compiler? We think that the answer is yes: the remaining obstacle to the integration of computers into the life of society is a widespread and well-justified reluctance to trust software.

The Challenge Repository

A scientific repository is a collection of scientific data constantly accumulating from experimental or observational sources, together with an evolving library of programs that process the data. Running one of these programs on the data is a kind of scientific experiment whose success, and sometimes failure, is reported in the refereed literature. If they prove to

be useful, the results of the experiment are added to the repository for use in subsequent experiments.

Our challenge repository will complement the verifying compiler by providing the scientific data for its experiments. These will be *challenge codes*: realistic examples of varying size. These codes will not merely be executable programs, but will be documentation of all kinds: specifications, architectures, designs, assertions, test cases, theorems, proofs, and conjectures. They will have been selected because they have proved their significance in past or present applications. The following are just two well-known experimental applications that may be used.

The recently constructed Waterkering storm-surge barrier is the final piece of the Dutch coastal flood defences. As part of Rotterdam's harbour is behind the barrier, it has to be movable. Since it takes about 11 hours to close, the decision to do so is based on weather forecasts and tide predictions. It is a computer that takes this decision: there is no manual override. This is a safety-critical application: if the barrier is closed unnecessarily, then the work of the harbour is badly disrupted; but if the gates are not closed when they are needed, then the severe floods of 1953 may be repeated.

Mondex is a smart card used for financial transactions: the value stored on a card is just like real money. An on-board computer manages transactions with other cards and devices, and all security measures have to be implemented on the card, without any real-time auditing. This application is security-critical: it must not be possible for a forger to counterfeit electronic value.

More ambitious examples may include crash-proofing selected open-source web services, perhaps leading eventually to crash-proofing the entire internet. Many of the challenge codes will be parts of real systems, and so they will be evolving at the hands of their owners and users; it will be part of the challenge to deal with this evolution.

Extending the Horizons

The basic idea underlying all techniques aimed at achieving high dependability is *consistency checking of useful redundancy*. A verifying compiler aims to check the consistency of a program with its specification once and for all, as early as possible, giving grounds for confidence that error will always be avoided at run-time. In contrast the dependability technique of *fault tolerance* delays the checking until just before it is needed, and shows how to reduce the effects of any detected errors to a tolerable level of inconvenience, or possibly even to mask the effects completely. This technique is used for different kinds of faults, including software, hardware, and operator faults, but the error checking will have been derived from the same specification that the verifying compiler would use. The same kind of redundancy is exploited, no matter where the consistency is checked.

Verification technology is applicable only to properties that can be rigorously formalised in notations accessible to a computer. Some of the important requirements of dependability, fault tolerance, component architectures, and associated design patterns have resisted formalisation so far. The availability of verification tools will surely give a fillip to research in these important areas.

Current Activities

Our first task is to define the scientific goals and objectives of the Grand Challenge, and we expect that this will take about three years of serious discussion. We have launched a series

of workshops to begin this process, and to expand the range of research activities to ensure that we address the issues encompassed by dependable systems evolution.

Workshop 1: Introduction. (November 2003, Queen Mary University of London). This first workshop was dedicated to getting a core group of interested researchers together to get to know each other. Work started on a comprehensive survey of the state-of-the-art. This will contain a list of leading researchers and teams around the world, with an annotated bibliography as background for the Grand Challenge.

Workshop 2: Tool-set. (March 2004, Gresham College). This second workshop involved a significant number of international Grand Challenge partners. It was devoted to discussing the theory and practice of existing tools, and to plan the components of a future tool-set for the Challenge.

Workshop 3: Applications. This third workshop will be devoted to establishing a collection of suitable benchmark problems for testing the progress of the Grand Challenge, and a collection of common software design patterns suitable for formalisation. The workshop will have a significant number of practitioners from industry.

Workshop 4: First Steps. The fourth workshop will continue the work on publishing a definitive survey of the field, and be concerned with the detailed planning of the first phase of the project.

Workshop 5: Landmark Events. The final workshop will be devoted to the organisation of landmark events in the progress of the project, which may include the following:

- A Royal Society meeting for discussion to interface to the general scientific community.
- The founding of an IFIP Working Group on strong software engineering tool-sets.
- A programme of meetings at the Royal Academy of Engineering to discuss experimental and future industrial applications.
- A one-week workshop at Schloss Dagstuhl, the International Conference and Research Centre for Computer Science.
- A meeting at the Isaac Newton Institute for Mathematical Sciences at Cambridge to discuss gaps in the theoretical foundations.

We will advertise the activities of the Grand Challenge through articles in the popular and learned press. We will maintain a comprehensive web-site to promote and disseminate our activities. We will engage in discussions in relevant newsgroups and email lists. One of the ambitions of a grand challenge is to capture the public's awareness of its aims and objectives; to achieve this, accessible accounts will be published of the background to the Challenge, its scientific goals, and a vision of future exploitation. This work will in no way compromise the scientific promise or integrity of the project to attract public attention or support. More technical articles will be published, particularly summaries of the proceedings of the workshops and final conference. Every article will also be available online at the Grand Challenge's web-site.

This period of planning will close with a final **Conference**, which will provide a showcase for a summary of the Challenge and the public presentation of the scientific research programme. It will also provide a forum for the formation of teams to carry out the proposed research. The most important product of the initial programme of work is to produce **a road-map of strategic directions**, a list of long-term scientific goals. These will be dedicated to extending current scientific knowledge, and understanding necessary engineering skills. The road-map may contain the following: a survey of known effective methods that can contribute to our scientific goals; pathways from successful research to new fields for research and application; a list of open research questions, preferably with simply verifiable answers. The road-map will have independent value as a reference for subsequent research proposals.