GC 6

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 progress 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 voting 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 verification; 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 experiments 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.

01822

Abr96 J.-R. Abrial 1996. *The B-Book.* Cambridge University Press.

Alu96 R. Alur, T. Henzinger, and P.-H. Ho 1996. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering* 22(3):181–201.

Ame02 Peter Amey 2002. Correctness by construction: better can also be cheaper. Available online at `www.sparkada.com/downloads/Mar2002Amey.pdf`.

App95 D. P. Appenzeller and A. Kuehlmann 1995. Formal verification of a PowerPC microprocessor. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'95)* (Austin, Texas, October) 79–84.

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).

Arn96 A. Arnold, D. Begay, and J.-P. Radoux 1996. The embedded software of an electricity meter: an experience in using Formal Methods in an industrial project. *Science of Computer Programming.*

Bar89 G. Barrett 1989. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering* 15(5):611–621.

Bar95 G. Barrett 1995. Model checking in practice: The t9000 virtual channel processor. *IEEE Transactions on Software Engineering* 21(2):69–78.

Bea91 S. Bear 1991. An overview of HP-SL. In *Proceedings of VDM'91: Formal Development Methods* Volume 551 of Lecture Notes in Computer Science. Springer-Verlag.

Ben96 J. Bengtsson, W. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi 1996. Verification of an audio protocol with bus collision using UppAal. In *Computer-Aided Verification '96.* Lecture Notes in Computer Science 1102, R. Alur and T. Henzinger (editors), Springer-Verlag, 244–256.

Bjo96 N. Bjorner et al. 1996. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proceedings of the Eighth International Conference on Computer-Aided Verification* Number 1102 in Lecture Notes in Computer Science (July), Springer-Verlag, 415–418.

Bos94 D. Bosscher, I. Polak, and F. Vaandrager 1994. Verification of an audio-control protocol. In *FTRTFT 94: Formal Techniques in Real-Time and Fault-Tolerant Systems* Lecture Notes in Computer Science 863, H. Langmaack, W.-P. de Roever, and J. Vytopil (editors), Springer-Verlag, 170–192.

Bos95 A. Boswell 1995. Specification and validation of a security policy model. *IEEE Transactions on Software Engineering* 21(2):63–68.

Boy79 R. S. Boyer and J. S. Moore 1979. *A Computational Logic.* Academic Press, New York.

Boy88 R. S. Boyer and J. S. Moore 1988. *A Computational Logic Handbook.* Academic Press, New York.

Boy96 R. Boyer and Y. Yu 1996. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM* 43(1):166–192.

Bra96 R. Brayton et al. 1996. VIS: A system for verification and synthesis. In *Proceedings of the Eighth International Conference on Computer-Aided Verification.* Number 1102 in Lecture Notes in Computer Science, Springer-Verlag, 423–427.

Bro86 M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra 1986. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers* C-35(12):1035–1044.

Bro96 B. Brock, M. Kaufmann, and J. S. Moore 1996. Heavy inference: Theorems about commercial microprocessors. In *Formal Methods in Computer-Aided Design (FMCAD'96).* M. Srivas and A. Camilleri (editors), Springer-Verlag.

Bry86 R. E. Bryant 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35(8).

Bur94 J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill 1994. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design and Integrated Circuits and Systems* 13(4):401–424.

Bux70 J. N. Buxton and B. Randell (editors). *Software Engineering Techniques.* NATO Scientific Committee Report.

Cal97 J. Calero, C. Roman, and G. D. Palma 1997. A practical design case using formal verification. In *Proceedings of Design-SuperCon'97.*

Car92 M. Carnot, C. Dasilva, B. Dehbonei, and F. Meija 1992. Error-free software development for critical systems using the B-methodology. In *Third International IEEE Symposium on Software Reliability Engineering.*

Cha88 K. Chandy and J. Misra 1988. *Parallel Program Design.* Addison-Wesley, Reading, MA.

Cha92 J. Chaves 1992. Formal methods at AT&T: An industrial usage report. In *Proceedings Formal Description Techniques IV.* North-Holland, Amsterdam, 83–90.

Che96 G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian 1996. Specification and verification of the PowerScale bus arbitration protocol: An industrial experiment with LOTOS. In *Proceedings of FORTE/PSTV'96* (Kaiserslautern, Germany). Chapman & Hall, London.

Chi87 G. Chisolm, J. Kljaich, B. Smith, and A. Wojcik 1987. An approach to the verification of a fault-tolerant, computer-based reactor safety system: A case study using automated reasoning (Vol. 1, interim report). *Technical Report NP-4924* (Jan.), Electric Power Research Institute, Palo Alto. Prepared by Argonne National Laboratory.

Cla81 E. M. Clarke and E. A. Emerson 1981. Synthesis of synchronisation skeletons for branching time temporal logic. In *Logic of Programs: Workshop* (Yorktown Heights, NY), Volume 131 of Lecture Notes in Computer Science, Springer-Verlag.

Cla86 E. M. Clarke, E. A. Emerson, and A. P. Sistla 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Program Languages and Systems* 8(2):244–263.

Cla92 E. M. Clarke, O. Grumberg, and D. E. Long 1992. Model checking and abstraction. In *Proceedings of Principles of Programming Languages.*

Cla93a E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness 1993. Verification of the Futurebus+ cache coherence protocol. In *Proceedings CHDL.*

Cla93b E. Clarke and X. Zhao 1993. Analytica: A theorem prover for Mathematica. *Mathematica Journal,* 56–71.

Cla96a E. Clarke, S. German, and X. Zhao 1996. Verifying the SRT division algorithm using theorem proving techniques. In *Proceedings of the Eighth International Conference on Computer-Aided Verification.* Number 1102 in Lecture Notes in Computer Science, Springer-Verlag, 111–122.

Cla96b E. Clarke and R. Kurshan 1996. Computer-aided verification. *IEEE Spectrum* 33(6):61–67.

Cla96c Edmund M. Clarke and Jeannette M. Wing 1996. Formal methods: state of the art and future directions. *ACM Computing Surveys* 28(4):626–643.

Cle93 R. Cleaveland, J. Parrow, and B. Steffen 1993. The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems. *ACM Transactions on Program Languages and Systems* 15(1):36–72.

Cle95 R. Cleaveland, E. Madelaine, and S. Sims 1995. Generating front ends for verification tools. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95).* Volume 1019 of Lecture Notes in Computer Science, E. Brinksma, R. Cleaveland, K. Larsen, and B. Steffen (editors), Springer-Verlag, 153–173.

Con86 R. Constable et al. 1986. *Implementing Mathematics with the NuPRL Proof Development Environment.* Prentice-Hall, Englewood Cliffs, NJ.

Cor95 C. Cornes, J. Courant, J.-C. Filliatre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Munoz, C. Murthy, C. Parent, A. Saibi, and B. Werner 1995. The Coq proof assistant reference manual version 5.10. *Technical Report 177* (July), INRIA.
Available online: `http://pauillac.inria.fr/coq/systeme_coq-eng.html`.

Cra88 D. Craigen, S. Kromodimoeljo, I. Meisels, A. Neilson, B. Pase, and M. Saaltink 1988. m-EVES: A tool for verifying software. In *Proceedings of the Tenth International Conference on Software Engineering* (Singapore, April), 324–333.

Cra93a D. Craigen, S. Gerhart, and T. Ralston 1993a. An international survey of industrial applications of formal methods. *Technical Report NIST GCR 93/626 (Vols. 1 and 2)* (March). U. S. National Institute of Standards and Technology. Also published by the U.S. Naval Research Laboratory (Formal Rep. 5546–93- 9582, Sept.), and the Atomic Energy Control Board of Canada.

Cra93b D. Craigen, S. Gerhart, and T. Ralston 1993b. Observations on industrial practice using formal methods. In *Proceedings of the Fifteenth International Conference on Software Engineering* (May).

Cra94 D. Craigen, S. Gerhart, and T. Ralston 1994. Formal methods in critical systems. *IEEE Software* 11(1).

Cra95 D. Craigen, S. Gerhart, and T. Ralston 1995. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering* 21(2):90–98.

Cro95 M. Croxford and J. Sutton 1995. Breaking through the V and V bottleneck. In *Proceedings of Ada in Europe.* Springer-Verlag.

Dam95 W. Damm, B. Josko, and R. Schloor 1995. Specification and Validation Methods for Programming Languages and Systems. In *Specification and verification of VHDL-based system-level hardware designs.* Oxford University Press, New York, 331–410.

Dam96 W. Damm and C. Delgado-Kloos 1996. Practical Formal Methods for Hardware Design. Lecture Notes in Computer Science. Springer-Verlag.

Daw95 C. Daws and S. Yovine 1995. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of 1995 IEEE Real-Time Systems Symposium, RTSS'95 (Pisa, Italy, Dec.).* IEEE Computer Society Press, Los Alamitos, CA.

Deh95 D. Deharbe and D. Borrione 1995. Semantics of a verification-oriented subset of VHDL. In *CHARME'95, Correct Hardware Design and Verification Methods.* P. Camurati and H. Eveking (editors) Number 987 of Lecture Notes in Computer Science Springer-Verlag, 293–310.

Del90 N. Delisle and D. Garlan 1990. A formal specification of an oscilloscope. *IEEE Software* 7(5):29–36.

Dep96 G. DePalma and A. Glaser 1996. Formal verification augments simulation. *Electrical Engineering Times* 56.

Dij72 E. W. Dijkstra The Humble Programmer. *CACM* 15(10):859–866.

Dil92 D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang 1992. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors* 522–525.

Din84 G. Dinolt et al. 1984. Multinet gateway: towards A1 certification. In *IEEE Symposium on Security and Privacy*

Dod96 C. J. Dodge, P. E. Undrill, A. R. Allen, and P. G. B. Ross 1996. Application of Z in Digital Hardware Design. *IEE Proceedings–Computers and Digital Techniques* 143(1):79–86.

Els96 W. Elseaidy, R. Cleaveland, and J. Baugh 1996. Modelling and verifying active structural control systems. *Science of Computer Programming.*

Fer96 J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu 1996. CADP (CAESAR/ALDEBARAN development package): A protocol validation and international verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification.* Number 1102 in Lecture Notes in Computer Science. R. Alur and T. A. Henzinger (editors), Springer-Verlag.

Fil94 T. Filkorn, H. Schneider, A. Scholz, A. Strasser, and P. Warkentin 1994. SVE User's Guide. *Technical Report ZFE BT SE 1-SVE-1,* Siemens AG, Corporate Research and Development, Munich.

Gar88 S. J. Garland and J. V. Guttag 1988. Inductive methods for reasoning about abstract data types. In *Proceedings of the Fifteenth Symposium on Principles of Programming Languages,* 219–228.

Gar95 D. Garlan, G. Abowd, D. Jackson, J. Tomayko, and J. Wing 1995. The CMU Master of Software Engineering Core Curriculum. In *Proceedings of the Eighth SEI Conference on Software Engineering Education (CSEE).* Number 895 of Lecture Notes in Computer Science, Springer-Verlag, 65–86.

Ger95 R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper 1995. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings IFIP/WG6.1 Symposium on Protocol Specification, Testing, and Verification* (Warsaw, Poland, June).

Gor79 M. J. Gordon, A. J. Milner, and C. P. Wadsworth 1979. *Edinburgh LCF.* Number 78 of Lecture Notes in Computer Science. Springer-Verlag.

Gor87 M. Gordon 1987. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis.* Kluwer.

Gui90 G. Guiho and C. Hennebert 1990. SACEM software validation. In *Twelfth International Conference on Software Engineering.*

Gut93 J. Guttag and J. Horning 1993. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag. Written with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing.

Hal96 A. Hall 1996. Using formal methods to develop an ATC information system. *IEEE Software* 12(6):66–76.

Har87 D. Harel 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8:231–274.

Har90 Z. Har'El and R. P. Kurshan 1990. Software for analytical development of communications protocols. *AT&T Bell Laboratories Technical Journal* 69(1):45–59.

Har92 D. Harel 1992. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer* 25(1):8–20.

Hei96 M. Heimdahl and N. Leveson 1996. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering* SE-22(6):363–377.

Hen80 K. Heninger 1980. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering* 6(1):2–13.

Hen94 T. A., Henzinger, X. Nicollin, J. Sifakis, and S. Yovine 1994. Symbolic model checking for real-time systems. *Information and Computation* 111:193–244.

Hoa85 C. A. R. Hoare 1985. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, NJ.

Hoj93 R. Hojati, R. Brayton, and R. Kurshan 1993. BDD-based debugging of designs using language containment and fair CTL. In *Proceedings of the Fifth International Conference on Computer-Aided Verification.* Number 697 in Lecture Notes in Computer Science, C. Courcoubetis (editor), Springer-Verlag, 41–57.

Hol89 G. Holzmann and J. Patti 1989. Validating SDL specifications: An experiment. In *Proceedings of the Ninth International Conference on Protocol Specification, Testing, and Verification, INWG/IFIP.* (Twente, Netherlands, June) C. Vissers and E. Brinksma, (editors)

Hol91 G. Holzmann 1991. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey.

Hol92 G. Holzmann 1992. Practical methods for the formal validation of SDL specifications. *Computer Communications.* Special issue on Practical Uses of FDTs.

Hol94a G. Holzmann 1994. The theory and practice of a formal method: NewCoRe. In *Proceedings of IFIP World Computer Congress* (Hamburg, Germany, August).

Hol94b G. Holzmann and D. Peled 1994. An improvement in formal verification. In *Proceedings of FORTE94.* (Berne, Switzerland, October).

Hou91 I. Houston and S. King 1991. CICS project report: Experiences and results from using Z. In *Proceedings of VDM'91: Formal Development Methods.* Volume 551 of Lecture Notes in Computer Science, Springer-Verlag.

How95 P.-H. Ho and Wong-H. Toi 1995. Automated analysis of an audio control protocol. In *Computer-Aided Verification '95.* Lecture Notes in Computer Science 939, P. Wolper Ed., Springer- Verlag, 381–394.

ISO87 ISO. 1987. Information Systems Processing–Open Systems Interconnection–LOTOS. *Technical Report* International Standards Organisation DIS 8807.

Jac95 J. Jacky 1995. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering* 21(2):99–106.

Jag96 L. Jagadeesan, C. Puchol, and J. V. Olnhausen 1996. A formal approach to reactive systems software: A telecommunications application in Esterel. *Formal Aspects of Computing* 8(2):123–151.

Jan96 R. Janicki, D. L. Parnas, and J. Zucker 1996. Tabular representations in relational documents. In *Relational Methods in Computer Science.* C. Brink, Ed., Springer-Verlag.

Jon86 C. B. Jones 1986. *Systematic Software Development Using VDM.* Prentice-Hall International, New York.

Kal94 M. Kaltenbach 1994. Model checking for UNITY. *Technical Report TR94-31* (Dec.), The University of Texas at Austin.

Kap87 D. Kapur and D. Musser 1987. Proof by consistency. *Artificial Intelligence* 31:125–157.

Kau95 M. Kaufmann and J. S. Moore 1995. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual (Version 1.8).* Available from:
`ftp://ftp.cli.com/pub/acl2/v1-8/acl2-sources/doc/HTML/acl2-doc.html`.

Kin94 T. King 1994. Formalising British Rail's signalling rules. In *FME'94: Industrial Benefit of Formal Methods.* Number 873 of Lecture Notes in Computer Science (1994), Springer-Verlag, 45–54.

Kin96 D. Kindred and J. Wing 1996. Fast, automatic checking of security protocols. In *Proceedings of the USENIX Workshop on Electronic Commerce Protocols* (1996).

Klj89 J. Kljaich, B. Smith, and A. Wojcik 1989. Formal verification of fault tolerance using theorem-proving techniques. *IEEE Transactions on Computers* 38:366–376.

Kue95 A. Kuehlmann, A. Srinivasan, and D. P. Lapotin 1995. Verity–a formal verification program for custom CMOS circuits. *IBM Journal Research and Development* 39(1/2):149–165.

Kuh90 D. Kuhn and J. Dray 1990. Formal specification and verification of control software for cryptographic equipment. In *Sixth Computer Security Applications Conference* (1990).

Kur93 R. Kurshan and L. Lamport 1993. Verification of a multiplier: 64 Bits and beyond. In *Computer Aided Verification.* Number 697 of Lecture Notes in Computer Science, C. Courcoubetis, Ed., Springer-Verlag, 166–179.

Kur94a R. P. Kurshan 1994a. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ.

Kur94b R. P. Kurshan 1994b. The complexity of verification. In *Proceedings 26th ACM Symposium on Theory of Computing (STOC).* (Montreal), 365–371.

Lam84 L. Lamport 1984. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 872–923.

Les83 P. Lescanne 1983. Computer experiments with the REVE term rewriting system generator. In *Proceedings of the Tenth Symposium on Principles of Programming Languages* (Austin, Texas, Jan.), 99–108.

Lon93 D. L. Long 1993. *Model checking, abstraction, and compositional reasoning.* Ph.D. Thesis, Carnegie Mellon Univ., Computer Science Dept.

Low96 G. Lowe 1996. Breaking and fixing the Needham-Schroder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems.* Number 1055 of Lecture Notes in Computer Science. Springer-Verlag.

Luo92 Z. Luo and R. Pollack 1992. LEGO proof development system: User's manual. *Technical Report ECS-LFCS-92-211* (May), Computer Science Department, University of Edinburgh.

Lyn87 N. Lynch and M. Tuttle 1987. Hierarchical correctness proofs for distributed algorithms. *Technical Report* (April), MIT Laboratory for Computer Science, Cambridge, MA.

Man91 Z. Manna and A. Pnueli 1991. *The Temporal Logic of Reactive and Concurrent Systems,* Springer-Verlag, New York.

Mat95 P. Mataga and P. Zave 1995. Multiparadigm specification of an AT&T switching system. In *Applications of Formal Methods.* M. G. Hinchey and J. P. Bowen, (editors), Prentice-Hall International, Englewood Cliffs, NJ, 375–398.

McM93 K. L. McMillan 1993. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer.

Mil80 A. J. R. G. Milner 1980. *A Calculus of Communicating Systems.* Number 92 of Lecture Notes in Computer Science. Springer-Verlag.

Mil89 Robin Milner 1989. *Communication and Concurrency.* Prentice Hall.

Mil95 S. P. Miller and M. Srivas 1995. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques* (Boca Raton, FL), IEEE Computer Society, Washington, DC, 2–16.

Moo96 J. S. Moore, T. Lynch, and M. Kaufmann 1996. A mechanically checked proof of the correctness of the AMD5K86 floating point division algorithm.
Available online at `http://devil.ece.utexas.edu:80/lynch/divide/divide.html`.

Nie89 M. Nielsen, K. Havelund, K. Wagner, and C. George 1989. The RAISE language, method and tools. *Formal Aspects of Computing* 1:85–114.

Owr92 S. Owre, J. Rushby, and N. Shankar 1992. PVS: A prototype verification system. In *Eleventh International Conference on Automated Deduction (CADE).* Number 607 of Lecture Notes in Artificial Intelligence, D. Kapur (editor), Springer-Verlag, 748–752.

Oxf96 University of Oxford. 1996. `http://www.comlab.ox.ac.uk/igdp/`. Master of Science in Software Engineering.

Pel96 D. Peled 1996. Combining partial order reductions with on-the-fly model-checking. *Journal Formal Methods in System Design* 8(1):39–64.

Pnu81 A. Pnueli 1981. A temporal logic of concurrent programs. *Theoretical Computer Science* 13:45–60.

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

Raj95 S. Rajan, N. Shankar, and M. Srivas 1995. An integration of model-checking with automated proof checking. In *Computer-Aided Verification '95.* Number 939 of Lecture Notes in Computer Science P. Wolper (editor), Springer- Verlag, 84–97.

Ric89 D. Richardson, T. O'Malley, and C. T. Moore 1989. Approaches to specification-based testing. In *ACM SIGSOFT 89: Third Symposium on Software Testing, Analysis, and Verification.*

Ros94 A. W. Roscoe 1994. Model-checking CSP. In *A Classical Mind: Essays in Honour of C. A. R. Hoare,* A. W. Roscoe (editor), Prentice-Hall, Englewood Cliffs, NJ.

Ros98 A. W. Roscoe 1998. *The Theory and Practice of Concurrency.* Prentice Hall.

Roy90 V. Roy and R. de Simone 1990. Auto/Autograph. In Computer-Aided Verification '90. Number 3 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science* (Piscataway, NJ, June), E. Clarke and R. Kurshan (editors), American Mathematical Society, Providence, RI, 235–250.

Rue96 H. Ruess, N. Shankar, and M. Srivas 1996. Modular verification of SRT division. In *Proceedings of the Eighth International Conference on Computer-Aided Verification.* Number 1102 in Lecture Notes in Computer Science (July), Springer-Verlag, 123–134.

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

Sch00 Steve Schneider 2000. *Concurrent and Real-time Systems.* Wiley.

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

Spi88 J. M. Spivey 1988. *Understanding Z: a Specification Language and its Formal Semantics.* Cambridge University Press, New York.

Spi92 J. M. Spivey *The Z Notation: A Reference Manual.* 2nd edition. Prentice Hall. 1992.

Ste96a B. Steffen, T. Margaria, A. Classen, V. Braun, and M. Reitenspiess 1996. An environment for the creation of intelligent network services. In *Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications–A Comprehensive Report* (Chicago), I. E. Consortium (editors), 287–300.

Ste96b B. Steffen, T. Margaria, A. Classen, and V. Braun 1996. The Meta '95 environment. In *Proceedings of Computer-Aided Verification '96.* Lecture Notes Computer Science, Springer-Verlag.

Var86 M. Y. Vardi and P. Wolper 1986. An automata-theoretic approach to automatic program verification. In *Proceedings of Logic in Computer Science.*

Win85 J. Wing 1985. Specification firms: A vision for the future. In *Proceedings of the Third International Workshop on Software Specification and Design* (London, August), 241–243.

Woo94 J. C. P. Woodcock, P. H. B. Gardiner, and J. R. Hulance The formal specification in Z of Defence Standard 00-56. *Proceedings of the Z User Workshop.* Workshop Series in Computer Science, Springer Verlag. 1994. Keynote speech.

Woo96 Jim Woodcock and Jim Davies 1996. *Using Z—Specification, Refinement, and Proof.* Prentice Hall.

Zav95 P. Zave 1995. Secrets of call forwarding: A specification case study. In *Proceedings of the Eighth International IFIP Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE '95).* Chapman & Hall, London, 153–168.

Zav96 P. Zave and M. Jackson 1996. Where do operations come from? A multiparadigm specification technique. *IEEE Transactions on Software Engineering* 22(7):508–528.

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 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 [OHe01, 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 data base 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*

Bal01 T. Ball and S. K. Rajamani 2001. Automatically Validating Temporal Safety Properties of Interfaces, *SPIN 2001*, LNCS 2057 pp.103–122.

Bus00 W. R. Bush, J. D. Pincus, and D. J. Sielaff 2000. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience* 30:775–802.

Cha88 K. M. Chandy and J. Misra 1988. *Parallel Program Design: a Foundation*, Adison-Wesley.

Eva02 D. Evans and D. Larochelle 2002. Improving Security Using Extensible Lightweight Static Analysis, *IEEE Software.*

Eva96 D. Evans 1996. Static detection of dynamic memory errors, *SIGPLAN Conference on Programming Languages Design and Implementation.*

Fla01 C. Flanagan and K. RM. Leino 2001. Houdini, an annotation assistant for ESC/Java. *International Symposium of Formal Methods Europe 2001*, LNCS 2021, Springer-Verlag pp.500–517.

Flo67 R. W. Floyd 1967. Assigning meanings to programs. *Proceedings of the Amer. Soc. Symp. Appl. Math.* 19, (1967) pp.19-31.

Gal98 A. J. Galloway, T. J. Cockram, and J. A. McDermid 1998. Experiences with the application of discrete formal methods to the development of engine control software. HISE York.

Gat02 W. H. Gates, *Internal communication.* Microsoft Corporation.

Gor88 M. J. C. Gordon 1988. HOL: A proof generating system for Higher-Order Logic, *VLSI Specification, Verification and Synthesis*, Kluwer pp.73–128.

Gra99 J. Gray 1999. What Next? A Dozen Information-technology Research Goals, *MS-TR-50*, Microsoft Research.

Hal02a A. Hall and R. Chapman 2002. Correctness by Construction: Developing a Commercial Secure System, *IEEE Software* 19(1): 18–25.

Hal02b S. Hallem, B. Chelf, Y. Xie, and D. Engler 2002. A System and Language for Building System-Specific Static Analyses. *PLDI 2002.*

Has03 Haskell 98 language and libraries: the Revised Report, *Journal of Functional Programming* 13(1) Jan 2003.

Hoa98 C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*, Prentice Hall.

Hoa99 C. A. R. Hoare and He Jifeng 1999. A Trace Model for Pointers and Objects, *ECOOP*, LNCS 1628, Springer-Verlag, pp.1–17.

Hoa02 C. A. R. Hoare 2002. Assertions, to appear, Marktoberdorf Summer School.

Hol91 G. J. Holzmann 1991. *Design and Validation of Computer Protocols*, Prentice Hall.

Iga99 A. Igarashi, B. Pierce, and P. Wadler 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ, *OOPSLA'99*, pp.132-146.

Jim02 T. Jim, G. Morrisett, D. Grossman, M. Hicks, J Cheney, and Y. Wang 2002. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey.

Kin69 J. C. King 1969. *A Program Verifier*, PhD thesis, Carnegie-Mellon University.

Lei98 K. M. Leino and G. Nelson 1998. An extended static checker for Modula-3. *Compiler Construction CC'98*, LNCS 1383, Springer-Verlag 302–305.

McC63 J. McCarthy 1963. Towards a mathematical theory of computation. *Proc. IFIP Cong. 1962*, North Holland.

Mey97 B. Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall.

Mil99 R. Milner 1999. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press.

Mos01 M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik 2001. Chaff: Engineering an Efficient SAT Solver, *38th Design Automation Conference (DAC2001)*, Las Vegas.

Mus02 M. Musuvathi, D. YW. Park, A. Chou, D. R. Engler, and D. L. Dill 2002. CMC: A pragmatic approach to model checking real code, to appear in *OSDI 2002.*

Nec02 G. C. Necula, S. McPeak, and W. Weimer 2002. CCured: Type-safe retrotting of legacy code. In *29th ACM Symposium on Principles of Programming Languages*, Portland.

Nec97 G. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97).*

Nim02 J. W. Nimmer and M. D. Ernst 2001. Automatic generation of program specifications, *Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pp.232–242.

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

OHe01 P. O'Hearn, J. Reynolds, and H. Yang 2001. Local Reasoning about Programs that Alter Data Structures. *Proceedings of CSL'01* Paris, LNCS 2142, Springer-Verlag, pp.1–19.

Ros94 A. W. Roscoe 1994. Model-Checking CSP, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice-Hall International, pp.353–378.

Ros98 A. W. Roscoe 1998. *Theory and Practice of Concurrency.* Prentice Hall.

Ruf95 E. Ruf 1995. Context-sensitive alias analysis reconsidered, *Sigplan Notices*, 30(6).

Sch99 F. B. Schneider (editor) 2002. Trust in Cyberspace, *Committee on Information Systems Trustworthiness*, National Research Council.

Sha01 U. Shankar, K. Talwar, J. S. Foster, and D Wagner 2001. Detecting format string vulnerabilities with type qualifiers, *Proceedings of the 10th USENIX Security Symposium.*

Sha96 N. Shankar 1996. PVS: Combining specification, proof checking, and model checking. *FMCAD '96*, LNCS 1166, Springer-Verlag, pp.257–264.

Sha97 N. Shankar 1997. Machine-assisted verification using theorem-proving and model checking. *Mathematical Methods of Program Development*, NATO ASI Vol 138, Springer-Verlag, pp.499-528.

Ste94 A. Stepanov and Meng Lee 1994. *Standard Template Library*, Hewlett Packard.

Ste00 S. Stepney, D. Cooper and J. C. P. Woodcock 2000. *An Electronic Purse: Specification, Refinement, and Proof*, PRG-126, Oxford University Computing Laboratory.

Str85 B. Stroustrup 1985. *The C++ Programming Language*, Adison-Wesley.

Tur49 A. M. Turing 1949. Checking a large routine. *Report on a Conference on High Speed Automatic Calculating machines.* Cambridge University Mathematical Laboratory 67–69.

Wag00 D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego.

J. C. P. W.
26th May 2003