The Verifying Compiler: A Grand Challenge for Computing Research

Tony Hoare

Microsoft Research Ltd., 7 JJ Thomson Ave, Cambridge CB3 0FB, UK
thoare@microsoft.com

Abstract. I propose a set of criteria which distinguish a grand challenge in science or engineering from the many other kinds of short-term or long-term research problems that engage the interest of scientists and engineers. As an example drawn from Computer Science, I revive an old challenge: the construction and application of a verifying compiler that guarantees correctness of a program before running it.

1 Introduction

The primary purpose of the formulation and promulgation of a grand challenge is to contribute to the advancement of some branch of science or engineering. A grand challenge represents a commitment by a significant section of the research community to work together towards a common goal, agreed to be valuable and achievable by a team effort within a predicted timescale. The challenge is formulated by the researchers themselves as a focus for the research that they wish to pursue in any case, and which they believe can be pursued more effectively by advance planning and co-ordination. Unlike other common kinds of research initiative, a grand challenge should not be triggered by hope of short-term economic, commercial, medical, military or social benefits; and its initiation should not wait for political promotion or for prior allocation of special funding. The goals of the challenge should be purely scientific goals of the advancement of skill and of knowledge. It should appeal not only to the curiosity of scientists and to the ambition of engineers; ideally it should appeal also to the imagination of the general public; thereby it may enlarge the general understanding and appreciation of science, and attract new entrants to a rewarding career in scientific research.

An opportunity for a grand challenge arises only rarely in the history of any particular branch of science. It occurs when that branch of study first reaches an adequate level of maturity to predict the long-term direction of its future progress, and to plan a project to pursue that direction on an international scale. Much of the work required to achieve the challenge may be of a routine nature. Many scientists will prefer not to be involved in the co-operation and co-ordination involved in a grand challenge. They realize that most scientific advances, and nearly all break-throughs, are accomplished by individuals or small teams, working competitively and in relative isolation. They value their privilege of pursuing bright ideas in new directions at short

notice. It is for these reasons that a grand challenge should always be a minority interest among scientists; and the greater part of the research effort in any branch of science should remain free of involvement in grand challenges.

A grand challenge may involve as much as a thousand man-years of research effort, drawn from many countries and spread over ten years or more. The research skill, experience, motivation and originality that it will absorb are qualities even scarcer and more valuable than the funds that may be allocated to it. For this reason, a proposed grand challenge should be subjected to assessment by the most rigorous criteria before its general promotion and wide-spread adoption. These criteria include all those proposed by Jim Gray [1] as desirable attributes of a long-range research goal.   The additional criteria that are proposed here relate to the maturity of the scientific discipline and the feasibility of the project. In the following list, the earlier criteria emphasize the significance of the goals, and the later criteria relate to the feasibility of the project, and the maturity of the state of the art.

- **Fundamental.** It arises from scientific curiosity about the foundation, the nature, and the limits of an entire scientific discipline, or a significant branch of it.
- **Astonishing.** It gives scope for engineering ambition to build something useful that was earlier thought impractical, thus turning science fiction to science fact.
- **Testable.** It has a clear measure of success or failure at the end of the project; ideally, there should be criteria to assess progress at intermediate stages too
- **Inspiring.** It has enthusiastic support from (almost) the entire research community, even those who do not participate in it, and do not benefit from it.
- **Understandable.** It is generally comprehensible, and captures the imagination of the general public, as well as the esteem of scientists in other disciplines.
- **Useful.** The understanding and knowledge gained in completion of the project bring scientific or other benefits; some of these should be attainable, even if the project as a whole fails in its primary goal.
- **Historical.** The prestigious challenges are those which were formulated long ago; without concerted effort, they would be likely to stand for many years to come.
- **International.** It has international scope, exploiting the skills and experience of the best research groups in the world. The cost and the prestige of the project is shared among many nations, and the benefits are shared among all.
- **Revolutionary.** Success of the project will lead to radical paradigm shift in scientific research or engineering practice. It offers a rare opportunity to break free from the dead hand of legacy.
- **Research-directed.** The project can be forwarded by the reasonably well understood methods of academic research. It tackles goals that will not be achieved solely by commercially motivated evolution of existing products.
- **Challenging.** It goes beyond what is known initially to be possible, and requires development of understanding, techniques and tools unknown at the start.
- **Feasible.** The reasons for previous failure to meet the challenge are well understood and there are good reasons to believe that they can now be overcome**.**
- **Incremental.** It decomposes into identified intermediate research goals, which can be shared among many separate teams over a long time-scale.
- **Co-operative.** It calls for planned co-operation among identified research teams and research communities with differing specialized skills.

- **Competitive.** It encourages and benefits from competition among individuals and teams pursuing alternative lines of enquiry; there should be clear criteria announced in advance to decide who is winning, or who has won.
- **Effective.** Its promulgation changes the attitudes and activities of research scientists and engineers.
- **Risk-managed**. The risks of failure are identified, symptoms of failure will be recognized early, and strategies for cancellation or recovery are in place.

The tradition of grand challenges is common in many branches of science. If you want to know whether a challenge qualifies for the title 'Grand', compare it with

- Prove Fermat's last theorem   (accomplished)
- Put a man on the moon within ten years   (accomplished)
- Cure cancer within ten years   (failed in 1970s)
- Map the Human Genome   (accomplished)
- Map the Human Proteome   (too difficult for now)
- Find the Higgs boson   (under investigation)
- Find Gravity waves   (under investigation)
- Unify the four forces of Physics   (under investigation)
- Hilbert's programme for mathematical foundations   (abandoned in 1930s)

All of these challenges satisfy many of the criteria listed above in varying degrees, though no individual challenge could be expected to satisfy all the criteria. The first in the list was the oldest and in some ways the grandest challenge; but being a mathematical challenge, my suggested criteria are considerably less relevant for it.

In Computer Science, the following examples may be familiar from the past. That is the reason why they are listed here, **not as recommendations,** but just as examples

- Prove that P is not equal to NP   (open)
- The Turing test   (outstanding)
- The verifying compiler   (abandoned in 1970s)
- A championship chess program   (completed)
- A GO program at professional standard   (too difficult)
- Automatic translation from Russian to English   (failed in 1960s)

The first of these challenges is of the mathematical kind. It may seem to be quite easy to extend this list with new challenges. The difficult part is to find a challenge that passes the tests for maturity and feasibility. The remainder of this contribution picks just one of the challenges, and subjects it to detailed evaluation according to the seventeen criteria.

2   The Verifying Compiler: Implementation and Application

A verifying compiler [2] 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.

This proposed grand challenge is now evaluated under the seventeen headings listed in the introduction.

Fundamental. Correctness of computer programs is the fundamental concern of the theory of programming and of its application in large-scale software engineering. The limits of application of the theory need to be explored and extended. The project is self-contained within Computer Science, since it constructs a computer program to solve a problem that arises only from the very existence of computer programs.

Astonishing. Most of the general public, and even many programmers, are unaware of the possibility that computers might check the correctness of their own programs; and it does so by the same kind of logical methods that for thousands of years have conferred a high degree of credibility to mathematical theorems.

Testable. If the project is successful, a verifying compiler will be available as a standard tool in some widely used programming productivity toolset. It will have been tested in verification of structural integrity and security and other desirable properties of millions of lines of open source software, and in more substantial verification of critical parts of it. This will lead to removal of thousands of errors, risks, insecurities and anomalies in widely used code. Proofs will be subjected to check by rival proof tools. The major internal and external interfaces in the software will be documented by assertions, to make existing components safer to use and easier to reuse [3]. The benefits will extend also to the evolution and enhancement of legacy code, as well as the design and development of new code. Eventually programmers will prefer to confine their use of their programming language to those features and structured design patterns which facilitate automatic checks of correctness [4,5].

Inspiring. Program verification by proof is an absolute scientific ideal, like purity of materials in chemistry or accuracy of measurement in mechanics. These ideals are

pursued for their own sake, in the controlled environment of the research laboratory. The practicing engineer in industry has to be content to work around the impurities and inaccuracies that are found in the real world, and often considers laboratory science as unhelpful in discharging this responsibility. The value of purity and accuracy (just like correctness) are often not appreciated until after the scientist has built the tools that make them achievable.

Understandable. All computer users have been annoyed by bugs in mass market software, and will welcome their reduction or elimination. Recent well-known viruses have been widely reported in the press, and have been estimated to cost billions of dollars. Fear of cyber-terrorism is quite widespread [6,7]. Viruses can often obtain entry into a computer system by exploiting errors like buffer overflow, which could be caught quite easily by a verifying compiler [8].

Trustworthy software is now recognised by major vendors as a primary long-term goal [9]. The interest of the press and the public in the project can be maintained, whenever dangerous anomalies are detected and removed from software that is in common use.

Useful. Unreliable software is currently estimated to cost the US some sixty billion dollars [10]. A verifying compiler would be a valued component of the proposed Infrastructure for Software Testing.

A verifying compiler may help accumulate evidence that will help to assess and reduce the risks of incorporation of commercial off-the-shelf software (COTS) into safety critical systems. The project may extend the capabilities of load-time checking of mobile proof-carrying code [11]. It will provide a secure foundation for the achievement of trustworthy software.

The main long-term benefits of the verifying compiler will be realised most strongly in the development and maintenance of new code, specified, designed and tested with its aid. Perhaps we can look forward to the day when normal commercial software will be delivered with an eighty percent chance that it never needs recall or correction by service packs, etc. within the first ten years after 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.

Historical. The idea of using assertions to check a large routine is due to Turing [12]. The idea of the computer checking the correctness of its own programs was put forward by McCarthy [13]. The two ideas were brought together in the verifying compiler by Floyd [14]. Early attempts to implement the idea [15] 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 [16]. 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 [17,18]. 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.

International. The project will require collaboration among leading researchers in America, China, India, Australasia, and many countries of Europe. Some of them are mentioned in the Acknowledgements and the References.

Revolutionary. At present, the most widely accepted means of raising trust levels of software is by massive and expensive testing. Assertions are used mainly as test oracles, to detect errors as close as possible to their place of occurrence [19]. Availability of a verifying compiler will encourage programmers to formulate assertions as specifications in advance of code, in the expectation that many of them will be verifiable by automated or semi-automated mathematical techniques. Existing experience of the verified development of safety-critical code [20,21] will be transferred to commercial software for the benefit of mass-market software products.

Research-directed. The methods of research into program verification are well established in the academic research community, though they need to be scaled up to meet the needs of modern software construction. This is unlikely to be achieved solely in industry. Commercial programming tool-sets are driven necessarily by fashionable slogans and by the politics of standardisation. Their elegant pictorial representations can have multiple semantic interpretations, available for adaptation according to the needs and preferences of the customer. The designers of the tools are constrained by compatibility with legacy practices and code, and by lack of scientific education and understanding on the part of their customers.

Challenging. Many of the analysis and verification tools essential to this project are already available, and can be applied now to legacy code [22-27]. But their use is still too laborious, and their improvement over a lengthy period will be necessary to achieve the goals of the challenge. The purpose of this grand challenge is to encourage larger groups to co-operate on the evolution of a small number of tools.

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 [20,21].
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 [28] 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 [29].

5. Theorem proving technology has made great strides in many directions. Model checking [30-33] is widely understood and used, particularly in hardware design. Decision procedures [34] are beginning to be applied to software. Proof search engines [35] are now well populated with libraries of application-dependent theorems and tactics. Finally, SAT checking [36] 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 [37-39]. 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 [40-42]; and theories for object orientation and pointer manipulation are under development [43,44].

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 – structural 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.

Cooperative. 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 [34]; 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 [45].

Competitive. The main source of competition is likely to be between teams that work on different programming languages. Some laboratories may prefer to concentrate on older languages, starting with C and moving on to C++. Others may prefer to concentrate on newer languages like Java or C#.

But even teams working on the same language and on the same tool may compete in achieving higher levels of verification for larger and larger modules of code. 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 in each module of software. The annotated libraries of open source code will be good competition material for the teams constructing and applying proof tools. The proofs themselves will be subject to confirmation or refutation by rival proof tools.

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.

Risk-managed. The main risks to the project arise from dissatisfaction of many academic scientists with existing legacy code and legacy languages. The low quality of existing software, and its low level of abstraction, may limit the benefit to be obtained from the annotations. Many failures of proof are not due to an error at all, but just to omission of a more or less obvious precondition. Many of the genuine

errors detected may be so rare that they are not worth correcting. In other cases, preservation of an existing anomaly in legacy software may be essential to its continuing functionality. Often the details of functionality of interfaces, either with humans or with hardware devices, are not worth formalising in a total specification, because testing gives an easier but adequate assurance of serviceability.

Legacy languages add to the risks of the project. From a logical point of view, they are extremely complicated, and require sophisticated analyses to ensure that they observe the disciplines that make abstract program verification possible. Finally, one must recognize that many of the problems of present-day software use are associated with configuration and installation management, build files, etc, where techniques of program verification seem unable to contribute.

The idealistic solution to these problems is to discard legacy and start again from scratch. Ideals are (or should be) the prime motivating force for academic research, and their pursuit gives scope for many different grand challenges. One such challenge would involve design of a new programming language and compiler, especially designed to support verification; and another would involve a re-write of existing libraries and applications to the higher standards that are achievable by explicit consideration and simplification of abstract interfaces. Research on new languages and libraries is in itself desirable, and would assist and complement research based on legacy languages and software.

Finally, it must be recognized that a verifying compiler will be only part of a integrated and rational tool-set for reliable software construction and evolution, based on sound scientific principles. Much of its use may be confined to the relatively lower levels of verification. It is a common fate of grand challenges that achievement of their direct goal turns out to be less directly significant than the stimulus that its pursuit has given to the progress of science and engineering. But remember, that was the primary purpose of the whole exercise.

References

[1]   J Gray, What Next? A Dozen Information-technology Research Goals, MS-TR-50, Microsoft Research, June 1999.

[2]   KM. Leino and G Nelson. An extended static checker for Modula-3. *Compiler Construction:, CC'98*, LNCS 1383, Springer, pp 302–305., April 1998.

[3]   B Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997

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

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

[6]   See http://www.fbi.gov/congress/congress02/nipc072402.htm, a congressional statement presented by the director of the National Infrastructure Protection Center.

[7]   FB Schneider (ed), *Trust in Cyberspace*, Committee on Information Systems Trust-worthiness, National Research Council (1999),

[8]   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, CA, February 2000

[9]   WH Gates, internal communication, Microsoft Corporation, 2002

[10]   Planning Report 02-3. The Economic Impacts of Inadequate Infrastructure for Software Testing, prepared by RTI for NIST, US Department of Commerce, May 2002

[11]   G Necula. Proof-carrying code. In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97), January 1997

[12]   AM Turing, Checking a large routine, *Report on a Conference on High Speed Automatic Calculating machines,* Cambridge University Math. Lab. (1949) 67–69

[13]   J McCarthy, Towards a mathematical theory of computation, Proc. IFIP Cong. 1962, North Holland, (1963)

[14]   RW Floyd, Assigning meanings to programs, *Proc. Amer. Soc. Symp. Appl. Math.* **19,** (1967) pp 19–31

[15]   JC King, A Program Verifier, PhD thesis, Carnegie-Mellon University (1969)

[16]   B Stroustrup, *The C++ Programming Language*, Adison-Wesley, 1985

[17]   A Igarashi, B Pierce, and P Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ, OOPSLA`99, pp. 132–146, 1999.

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

[19]   CAR Hoare, Assertions, to appear, Marktoberdorf Summer School, 2002.

[20]   S Stepney, D Cooper and JCPW Woodcock, An Electronic Purse: Specification, Refinement, and Proof, PRG-126, Oxford University Computing Laboratory, July 2000.

[21]   AJ Galloway, TJ Cockram and JA McDermid, Experiences with the application of discrete formal methods to the development of engine control software, Hise York (1998)

[22]   WR Bush, JD Pincus, and DJ Sielaff, A static analyzer for finding dynamic programming errors, Software -- Practice and Experience 2000 (30): pp. 775–802.

[23]   D Evans and D Larochelle, *Improving Security Using Extensible Lightweight Static Analysis*, IEEE Software, Jan/Feb 2002.

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

[25]   GC Necula, S McPeak, and W Weimer, CCured: Type-safe retrotting of legacy code. In 29th ACM Symposium on Principles of Programming Languages, Portland, OR, Jan 2002

[26]   U Shankar, K Talwar, JS Foster, and D Wagner. Detecting format string vulnerabilities with type qualifiers, Proceedings of the 10th USENIX Security Symposium, 2001

[27]   D Evans. Static detection of dynamic memory errors, SIGPLAN Conference on Programming Languages Design and Implementation, 1996

[28]   CAR Hoare and He Jifeng. *Unifying Theories of Programming*, Prentice Hall, 1998.

[29]   E Ruf, Context-sensitive alias analysis reconsidered, Sigplan Notices, 30 (6), June 1995

[30]   GJ Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991

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

[32] M Musuvathi, DYW Park, A Chou, DR. Engler, DL Dill. CMC: A pragmatic approach to model checking real code, to appear in OSDI 2002.

[33] N Shankar, Machine-assisted verification using theorem-proving and model checking, *Mathematical Methods of Program Development,* NATO ASI Vol 138, Springer, pp 499–528 (1997)

[34] MJC Gordon, HOL: A proof generating system for Higher-Order Logic, *VLSI Specification, Verification and Synthesis*, Kluwer (1988) pp. 73–128

[35] N Shankar, PVS: Combining specification, proof checking, and model checking. FMCAD '96,LNCS 1166, Springer, pp 257–264, Nov 1996

[36] M Moskewicz, C Madigan, Y Zhao, L Zhang, S Malik, Chaff: Engineering an Efficient SAT Solver, 38th Design Automation Conference (DAC2001), Las Vegas, June 2001

[37] T Ball, SK Rajamani, Automatically Validating Temporal Safety Properties of Interfaces, *SPIN 2001,* LNCS 2057, May 2001, pp. 103–122.

[38] JW Nimmer and MD Ernst, Automatic generation of program specifications, *Proceedings of the 2002 International Symposium on Software Testing and Analysis*, 2002, pp. 232–242.

[39] C Flanagan and KRM Leini, Houdini, an annotation assistant for ESC/Java. *International Symposium of Formal Methods Europe 2001,* LNCS 2021, Springer pp 500–517, 2001

[40] R Milner, *Communicating and Mobile Systems: the pi Calculus*, CUP, 1999

[41] AW Roscoe, *Theory and Practice of Concurrency*, Prentice Hall, 1998

[42] KM Chandy and J Misra, *Parallel Program Design: a Foundation*, Adison-Wesley, 1988

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

[44] CAR Hoare and He Jifeng, A Trace Model for Pointers and Objects, ECOOP, LNCS 1628, Springer (1999), pp 1–17

[45] A Stepanov and Meng Lee, Standard Template Library, Hewlett Packard (1994)

[46] CAR Hoare, The Verifying Compiler: a Grand Challenge for Computer Research, JACM (50) 1, pp 63–69 (2003)