Exemplar Project: the Verifying Compiler.

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 a relevant selection of the standard headings suggested for evaluation of a Grand Challenge Project.

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 It was generally written in assembler for a proprietary computer architecture, which was often withdrawn after a short interval on the market.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.

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].

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].

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.

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.

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 Trustworthiness, 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 Leino, 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)