Preface to Special Issue on Software Verification

C.A.R. HOARE

Microsoft Research Labs, Cambridge

and

JAYADEV MISRA

University of Texas at Austin

The origins of software verification go back to the pioneers of Computing Science, von Neumann and Turing. The idea has been rediscovered several times since then, for example by McCarthy, Naur and Floyd. The ideals of verification have inspired half a century of productive computing research at the foundations of the subject. There are now flourishing research schools in computational logic, computer-aided proof, programming theory, formal semantics, specification and programming languages, programming methodology and software engineering.

By the end of the last century, enormous progress had been made in verification theory and in tools to assist in its application. The technology of proof was extended to include constraint solving and model checking, which were routinely exploited in the electronics industry to increase confidence in the absence of errors in circuit designs before commitment to silicon. Programming theory and semantics provided logics for proof of correctness of well-structured sequential programs. The foundations of concurrent programming were explored by employing temporal logic, and communication over channels was explored in a number of process algebras. Formal specifications were used in certain safety-critical applications as an aid to system development and verification of correctness. Internal program specifications in the form of program assertions were used in the software industry as test oracles, to detect and diagnose errors in regression tests conducted overnight. In suitable cases they are left in customer code for re-checking at run time.

The early years of the current century have seen a dramatic spurt in progress towards realization of the ideal of verification of software as well as hardware. Proof technology is now routinely exploited in industrially supported program analysis tools, which successfully detect many kinds of generic program error even before a program is tested. Mature proof tools, both automatic and interactive, are now providing indispensable aid in computing research, including research into verification. For mechanized proof of classical conjectures in mathematics, computers have become an indispensable tool.

Authors' addresses: C.A.R. Hoare, Microsoft Research Cambridge, Roger Needham Building, 7 J J Thomason Avenue, Cambridge CB3 0FB, UK; J. Misra, University of Texas at Austin, Department of Computer Sciences, College of Natural Sciences, 1 University Station C0500, Austin, TX 78712, Corresponding email: misra@cs.utexas.edu.

DOI 10.1145/1592434.1592435 http://doi.acm.org/10.1145/1592434.1592435

It has suddenly become easier to use mechanical aids than not to do so! Programming theory has moved on to cover many of the complex aspects of modern programming languages, without restriction to well-structured forms of program and data. Concurrency theory is now beginning to tackle the problems posed by threads that share memory, even the weakly consistent memory offered by modern multi-core architectures. Assertions and internal interface specifications (contracts) have been incorporated in most of the widely used programming languages. The design of recent programming languages has explicitly recognised among its design criteria the assistance offered to the programmer in the achievement of correct programs.

At the same time, computer software has infiltrated every aspect of modern life and has set the pace of innovation in all of them. Industry and commerce, science and engineering, domestic appliances, travel, entertainment, education and even research in science and the humanities, have all changed beyond recognition. In total, software must now be counted as one of the world's few trillion-dollar industries. With this growth, the significance of verification has also increased. The financial cost of programming error to the world economy has been estimated as tens of billions of dollars per year. Most of it falls (in small but frequent doses) on the billions of users of software rather than on its producers. This economic argument is an added incentive supporting research towards the ideal of software verification.

The verification research communities have recognized the enormous opportunities now presented for beneficial research in this area. At the same time, they recognize that success will depend on a significant change in the culture and practices of research in computing science. It will be necessary to move towards the culture of the *big sciences*, for example physics, astronomy, and (more recently) biology. Like them, more rapid progress will require longer term projects, and wider collaborations than are customary today. In particular, it will require collaboration among theorists, tool-builders, and experimenters, each contributing an independent specialist skill to a common enterprise. The experimenters have to take the lead: they will apply theories originated by theorists; they will use tools constructed and maintained by tool-builders; and they will apply them to verify (at some level) a repository of challenge applications derived from practical use, and generally agreed as significant for the advancement of the basic science and technology of verification. The results of the experiment will provide independent and objective evidence of the applicability of the theory to useful software, and the capability of the tools in prevention or detection and elimination of errors.

This new paradigm for the conduct of verification research is called the Verified Software Initiative. It has been discussed intensively at two international conferences. The first was in Zurich (2005), and was attended by international leaders of many of the relevant research communities. The second was in Toronto (2008), and many of its participants signed a manifesto declaring their commitment to the initiative. The manifesto is reproduced as an appendix to this special issue of the Computing Surveys.

The main content of the Survey provides the technical background of the Verified Software Initiative. The article by Woodcock et al. describes the current state of the art in the practical application of verification technology for software. It covers all stages of the software life cycle, from requirements analysis through specification, design, code construction, testing, delivery, and subsequent evolution. The article reports the results of a recent survey of industrial practice, and continues with an account of a number of recent industrial projects. They are landmarks in the delivery of software to a high level of verification: many of these software products are still in use. The article concludes with a description of a collection of advanced verification challenges, which have been undertaken by the research community as experiments for contribution to the software repository.

The article by Shankar is devoted to the advances in automated deduction techniques currently prevalent in widely used verification tools. It starts with brief recapitulation of the concepts of logical reasoning, as applied in formal verification. This covers both propositional logic and predicate calculus, extended with first-order theories. It describes the remarkable technology of modern satisfiability (SAT) solvers, which have improved the efficiency of automated proof by three or more orders of magnitude; this is cumulative to the rather slower growth of raw computing power in accordance with Moore's law. The article then describes a collection of the mature tools available today, and describes how they combine the benefits of SAT with decision procedures, model checking, proof search, algebraic rewriting, and human interaction. The article concludes with a survey of outstanding challenges.

The article by Majumdar and Jhala describes the achievements of research in model checking. The goal of model checking is to establish correctness properties of a program or find counterexamples to these properties by exploring concrete or symbolic representations of the state space of the program. Model checking was exploited first in the electronics industry for circuit design, as specified by their simulation programs; but it is now incorporated in more general program analysers to generate test cases, or to detect errors without testing. It is particularly appropriate for application to legacy code, which is often found to have no external or internal assertions or other documentation. The article describes how such documentation can be inferred automatically, using symbolic execution together with predicate abstraction, Craig interpolation, or widening in an abstract domain. The article concludes with a survey of current mature model checking tools, and an introduction to remaining research challenges.

We hope that publication of this special issue will inspire the modern generation of students of Computing Science to devote their research to the goals of the Verified Software Initiative. The articles in this issue will provide them with the essential background for an understanding of the technical challenges. The range of topics covered has been limited by the available space. There are many other research areas essential to the goal of software verification. We hope that they will be the subject of future articles submitted to Computing Surveys.

ACKNOWLEDGMENTS

We are extremely grateful to the reviewers of the articles for this special issue. They have reviwed the dense technical material at great depth, and provided detailed constructive suggestions on succesive versions of the manuscripts.