The Challenge of Software Verification

N. Shankar

shankar@csl.sri.com

URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory

SRI International

Menlo Park, CA

Overview

Professor Tony Hoare has proposed the goal of automatically verified software as a *grand scientific challenge* for computing.

A series of workshops (funded by NCO HCSS through NSF) have been organized to explore the nature of this challenge.

A preliminary workshop was held at SRI International in Washington DC in April 2004 and was attended by about 50 participants.

A larger workshop was held recently (Feb 21–23, 2005) at SRI International in Menlo Park (chaired by Jay Misra, Greg Morrisett, and NS).

A working conference will be held in Zurich, Switzerland during the week of Oct. 10, 2005 (www.vstte.ethz.ch).

Outline

- A brief history of verification
- Hoare's verification grand challenge
- The VGC workshop: participants, themes, and debating points
- Conclusions
- Discussion

A Disclaimer

I had no part in initiating this challenge — that is well above my pay grade.

While I'm sympathetic to many elements of this challenge, many crucial aspects have to be resolved through open discussion and debate before a clear picture emerges.

This talk is an attempt to stir up such discussion.

It captures some of the viewpoints expressed at the workshop.

See `www.csl.sri.com/~shankar/VGC03` for more details.

Dijkstra's Retirement Speech

At his retirement/70th birthday ceremony, Edsger Dijkstra gave a speech entitled *Under the Spell of Leibniz's Dream*.

Among his significant early contributions were an Algol 60 compiler and the THE operating system.

Though both were extremely influential, neither piece of software was actually that widely used. . . . so what was the fuss about?

They helped resolve a debate over whether computer science was an academic subject or whether industry had things *well under control*.

A Brief History of Verification

1950s: Turing, von Neuman: Hand proofs of program correctness.

1960-63: McCarthy's *Mathematical Theory of Computation*

1966/67: Floyd introduces assertional reasoning on flowcharts for proving partial and total correctness.

1969: Hoare introduces axiomatic semantics for programming constructs.

King writes a dissertation on automatic program proving through verification condition generation.

Fast Forward . . .

1970s: Predicate transformer semantics, LCF/ML, Boyer-Moore prover, VDM, abstract interpretation, algebraic data types, temporal logic, combination decision procedures.

de Millo, Lipton, and Perlis on *Social Processes and Proof*.

1980s: Model checking, hardware verification, HOL/Nuprl/Coq/Isabelle/EHDM, UNITY, TLA, I/O automata, Z specification language, OBJ3, KIDS.

1990s: Symbolic model checking, timed/hybrid model checking, predicate abstraction, bounded model checking, B Method, proof carrying code, typed assembly language.

Intel FDIV and aborted Ariane-5 launch.

. . . To the Present

Industrial use of hardware verification (AMD, Intel, Synopsys, Cadence, Mentor Graphics).

Microsoft's SLAM project for device driver verification (uses theorem proving, predicate abstraction, and model checking).

Large-scale program analysis: A380, Ariane-5, Linux/OpenBSD kernel.

Crypto-protocol verification.

Model-based design of embedded systems software.

Workshop Participants

Academia: Martin Abadi, Alex Aiken, Sergey Berezin, Randy Bryant, Ed Clarke, Matt Dwyer, Allen Emerson, Dawson Engler, David Evans, Ganesh Gopalakrishnan, Ric Hehner, Daniel Jackson, Deepak Kapur, Flavio Lerda, Rupak Majumdar, Pete Manolios, Scott McPeak, Jose Meseguer, Bertrand Meyer, J Moore, Amir Pnueli, John Reynolds, Robby, Henny Sipma, Konrad Slind, Maria Sorea, Josh Tauber.

Private Research: Leonardo de Moura, Bruno Dutertre, Cordell Green, Peter Neumann, Sam Owre, Harald Ruess, Hassen Saidi, Doug Smith, Ashish Tiwari.

Government: Paul Black, Helen Gill, Brad Martin, Klaus Havelund, Connie Heitmeyer, Mike Hinchey, Peter Homeier, Bill Legato, Mike Lowry, John Penix, James Rash, Willem Visser.

Industry: Dave Hardin, Tony Hoare, Jim Horning, Rance de Long, John Harrison, Nils Klarlund, Rustan Leino, Paul Loewenstein, Ken McMillan, Greg Nelson, Sriram Rajamani, Wolfram Schulte, Yuan Yu, Lintao Zhang.

Hoare's Verification Grand Challenge

A mature scientific discipline should set its own agenda and pursue ideals of purity, generality, and accuracy far beyond current needs.

Science explains why things work in full generality by means of calculation and experiment.

An engineering discipline exploits scientific principles to the study of the specification, design, construction, and production of working artifacts, and improvements to both process and design.

The verification challenge is to achieve a significant body of verified programs that have precise external specifications, complete internal specifications, machine-checked proofs of correctness with respect to a sound theory of programming.

The Deliverables

A *comprehensive theory* of programming that covers the features needed to build practical and reliable programs.

A *coherent toolset* that automates the theory and scales up to the analysis of large codes.

A *collection of verified programs* that replace existing unverified ones, and continue to evolve in a verified state.

"You can't say anymore it can't be done! Here, we have done it. "

Topics

- Model Checking
- Software model checking
- Decision Procedures
- Theorem provers
- Static/dynamic analysis
- Programming languages/semantics
- Programming methodology
- Applications
- Metrics/Benchmarks

Model Checking (MC)

Examples: SMV, COSPAN, VIS, SAL, CMC.

Strengths: hardware, control-intensive software (100-1000 state bits), protocols, interface checking. Automatic with counterexamples.

Issues: Predicate abstraction, counterexample-guided abstraction refinement, test-case generation, invariant generation.

Challenges: Complex data types, pointers, finding good abstractions, generating complex invariants, parametricity, compositionality, and environment models.

Software Model Checking (SMC)

Examples: SPIN, Bandera, Java Pathfinder, Verisoft, Blast, MAGIC, Cadena, Zing.

Strengths: Systems with dynamic data structures and threads, small reachable set of states. SMC can be built by instrumenting the virtual machine.

Issues: State space explosion, hybrid representations, model extraction from software, environment models, real-time systems.

Challenges Checking functional properties, exploiting modularity, and achieving scale with respect to data and concurrency.

Decision Procedures (DP)

Examples: GRASP, Chaff, zchaff, Berkmin, Siege, Simplify, ICS, UCLID, SVC, CVC, CVCL, Mathsat, DPLL(T), TSAT, QEPCAD, Zap.

Strengths: Satisfiability over booleans, arithmetic, arrays, abstract data types, uninterpreted functions, and their combination.

Issues: Improved APIs (online, resettable, proof/counterexample interpolant producing), QBF, lazy vs. eager combination, modularity, quantifiers, performance.

Challenges: API/performance tradeoff, quantifiers, nonlinear arithmetic, compiling new theories, computing joins, providing counterexamples, and explanations.

Theorem Proving (TP)

Examples: ACL2, Coq, HOL, Isabelle, Maude, Nuprl, PVS, STeP.

Strengths: Mathematically rich theories, data-intensive systems, operational semantics, fault tolerance, security.

Issues: Performance, integration with DP/MC, feedback through proofs/counterexamples, deep and shallow embeddings, proof strategies.

Challenges: Reconciling automation and user guidance, libraries, invariant generation, lemma generation, user feedback, integration with MC and DP, and fast rewriting.

Static and Dynamic Analysis (SA/DA)

Examples: BANE, Ccured, Fluid, Polyspace, Temporal Rover, PREfix.

Strengths: Buffer overruns, overflows, memory leaks, and race conditions. Handles 1MLOC. SA is good for generic properties whereas DA is good for user annotations.

Issues: Combining different SAs, integrating SA and DA, bug finding.

Challenges: Efficient, precise, and modular analysis; path sensitivity; concurrency; reducing spurious "bugs".

Programming Languages/Semantics

Type systems: Move to undecidable type systems, types for security and information flow, linear typing, exceptions, concurrent interaction.

Heap/Pointers: Separation logic.

Correctness/Optimization: Undischarged assumptions yield runtime checks with performance penalty.

Generic programming: Standard templates library (STL); exploit algebraic properties in efficient algorithms.

Programming in mathematics: Programming languages as syntactic sugar for mathematical concepts.

Programming Methodology Examples: Alloy, B Method, I/O

Automata, Spec#, Specware, VDM, Z.

Strengths: Use models, specification to guide code development.

Issues: Interaction between structure and verification, domain formalization.

Challenges: Invariants, initialization, modularity, concurrency, maintaining model/code correspondence.

Suggested Challenge Applications

Small (Algorithms, Architectures, Programs): Infusion pump, medical devices, embedded controllers

Medium (Libraries): Separation kernel, STL, MPI, file systems.

Large (Systems): Apache, Linux, SCADA (Supervisory Control And Data Acquisition)

Themes/Debates

- Normative vs. Descriptive Approaches: Design new languages that are better suited for verification.
- Analytic vs. Synthetic: Generate correct code from high-level specifications instead of verifying low-level code.
- Church vs. Curry: Should programmer provide annotations or should they be infered automatically?
- Bug finding vs. Verification: Commercial tools are going to focus on bug-finding.
- Shallow vs. deep properties: Deep properties need user guidance, which is a *good thing*.

Themes/Debates (continued)

- Carrot vs. Stick: Is product liability needed to drive industry practice toward verification?
- "Winner take all" vs. "Let a million flowers bloom": Should we have verification challenges with prize money?
- Tool suite vs. verifying compiler: Need a precise goal.
- Specification vs. Verification Grand Challenge: Need reference specifications and implementations to kick-start verification.

Convergence

Build a unified verifying compiler based on a formal tool bus for integrating different analysis/synthesis tools.

Transformational approach to analyzing models, composing models, generating optimized code, integrating existing code, support finely tuned static and dynamic analyses.

Formal tool bus (FTB) manages semantic flow between different formalisms, languages, and tools to map models, assertions, counterexamples, and representations.

Support interactive and automated development of verified software with seamless use of analysis tools.

Integrate with existing modeling formalisms.

Eat your own dog food: Develop verified tools.

Divergence

What's wrong with business as usual?

Is there enough mutual understanding to embark on a grand challenge?

Are we overly ambitious in our goals?

Is software much too diverse so that we should focus on specific application areas that are most amenable to automation?

Will an unhealthy focus on benchmarks divert attention for the "real" problems?

Why not let market forces dictate the development of verification technology?

A Tentative 15-Year Roadmap

Years 1-5: **Specification grand challenge.** Development of Metrics/Benchmarks. Formal Tool Bus. Large theorems about small programs, small theorems about big programs.

Years 6-10: **Integration grand challenge.** Use FTB to support tool integration. Medium examples. Verified libraries.

Years 11-15: **Application grand challenge.** Deliver comprehensive, integrated tool suite with a range of verified large-scale applications.

Some Thought Experiments

Transportation: Each community builds its own transportation network of roads, railways, aviation, and shipping without any coordination on standards, timetables, and with incompatible signaling and communication mechanisms.

Climbing the mountain: Each individual climber manages to get part of the way up the mountain, and perhaps a bit more each year, or even by different routes, but no one is able to reach the peak for the lack of base camps, supply lines, emergency support, and communication.

Inverting Gresham's law: *Bad software lives forever. Good software gets updated until it goes bad, in which form it lives forever.* Casey Schaufler

Can we make good software proliferate?