Verified Software Grand Challenge

Jim Woodcock
University of York
April 2006

Introduction

- industrial software usually has extensive documentation
- …but software behaviour is often a complete surprise!
- programing is **hard**
  - reduce a problem to a set of rules that can be **blindly** followed by a computer
- components interact and interfere
- undesirable properties emerge
- systems fail to satisfy their users' needs

Software is inherently hard to develop

- it's hard to define requirements, to anticipate interactions, to accommodate new functionality
- documentation involves large amounts of text, pictures, and diagrams, but these are often imprecise and ambiguous
- important information often hidden by irrelevant detail
- design mistakes are often discovered too late, making them expensive or even impossible to correct

it's a tribute to the skill of software engineers that systems work at all

A modern fairy-tale

- $\mathcal{W}$: your least favourite, but essential, software application
- the good fairy offers you the choice between correcting the bugs in $\mathcal{W}$ or cleaning up its architecture
- clean architecture
  - sound mental model makes it easier to use
- architecture is made more evident by components and their interfaces being clearly documented
- supports maintenance, reuse, and evolution
- **it's not just about correctness**

Cliff Jones

Another way?

some practitioners in industry and researchers from universities believe it's now practical to use formal methods to produce software, even non-critical software

- ***and that this will turn out to be the cheapest way to do it***
- *given the right computer-based tools, the use of formal methods could become widespread and transform the practice of software engineering*
- *the computer science community recently committed itself to making this a reality within the next fifteen to twenty years*

Vision 15 years out

- software versus industry's other products
  - software generally sold without meaningful warranty
  - but the CD has a guarantee!
- implied warranty of fitness for purpose
- could we ever expect software to come with such warranties?
- in our vision of the future, we would expect exactly that

Maturity of theoretical computer science

- 70 years of academic research in software
  - fundamental results in theoretical computer science
  - exploited in theories of software development
  - studied as formal methods of software engineering
- scientific theories explain and predict
  - formal methods explain software, both to the user and to the developer
    - they produce precise documentation, structured and presented at an appropriate level of abstraction
  - formal methods also predict behaviour of software, by being amenable to mathematical analysis

Maturity of formal methods

- routinely taught
- google `formal methods education`
  - repositories, virtual library, surveys of hundreds of different courses with on-line resources
  - key texts freely downloadable
- wide range of industrial examples
- used with particular success in safety-critical applications

But why now?

- ***new wave of research***
- *using logic behind the scenes in program analysis and model checking tools*
- *Microsoft: world's biggest software developer*
  - *Windows used by most computer users*
  - *infamous for frequent crashes*
  - *caused by faulty device drivers*
  - ***static driver verifier** checks drivers for conformance*
  - *success comes from very restricted domain*
  - *if a driver fails the SDV test, then it might contain a bug*
- *nothing so practical as a good theory*
- *SDV has sophisticated theory, hidden from user*

A concerted push

- considerable experience in successful use of formal methods
- new wave of tools shielding users from technical subtlety
- significant advances in proof technology
  - SAT solvers and combined decision procedures
- time is right for a concerted push at software verification
- considerable activity is already under way
- but it is not (yet) concerted

Golden Gates?

Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.

Bill Gates
Keynote address
WinHec 2002

Ode to Joy?

> ***I have a few more things I want to do.***
> ***I still think the tools we have for building reliable software are inadequate.***

Bill Joy

Showstopper?

> ***Without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry.***

2005 ITRS Roadmap

A grand challenge

- Tony Hoare

  > ***automatically verified software:***
  > ***a grand scientific challenge for computing***

- NSF-funded meetings in the US
- EPSRC-funded meetings in the UK
- Zürich conference `vstte.inf.ethz.ch`
- FACJ article, 2006
- IEEE Computer article, April 2006
- research roadmap `qpq.csl.sri.com`

Ancient history (Jones)

- **1947–49**
  - Goldstine/von Neumann and Turing: Assertional hand proofs of program correctness
- **1961–63**
  - McCarthy: A Basis for a Mathematical Theory of Computation
- **1966/67**
  - Floyd and Naur: flowcharts + assertions for partial and total correctness
- **1969**
  - Hoare: axiomatic basis for computer programming
  - King: a program verifier

Modern history

- 1970s
  - predicate transformer semantics, LCF/ML, Boyer-Moore prover, VDM, abstract interpretation, algebraic data types, temporal logic, combination decision procedures
- 1980s
  - model checking, hardware verification, HOL, Nuprl, Coq, Isabelle, EHDM, UNITY, TLA, I/O automata, Z notation, 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 bug and aborted Ariane-5 launch

Current affairs

- industrial use of hardware verification (AMD, Intel, Synopsys, Cadence, Mentor Graphics)
- Microsoft: SLAM project for device driver verification (uses theorem proving, predicate abstraction, and model checking)
- large-scale program analysis: A380, Ariane-5, Linux/OpenBSD kernel
- experimental tools (SAT/SMT solvers, model checkers, static/dynamic analyzers, proof checkers) with competitions, intermediate formats, API standardization
- large-scale verification systems/projects (Spark/ADA, ESC/Java, Spec#, Verisoft, LOOP, JML, KeY, Krakatoa)
- applications: hardware, security, cryptographic protocols, communication protocols, AI planning

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 in 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*

Deliverables

1. **a comprehensive theory of programming**
   - covering the features needed to build practical and reliable programs
2. **a coherent toolset**
   - automating the theory and scaling up to large codes
3. **a collection of verified programs**
   - replacing existing unverified ones
   - continuing to evolve as verified code

You can't say any more it can't be done!
Here, we've done it!

Hoare and Misra's Timetable

- **long-term goal**
  - ensure science and practice converge
  - employ and teach principles of specification, design, architecture, language, and semantics
- **20 years**
  - well-developed theory, comprehensive and powerful suite of tools, compelling body of experimental evidence
    - reliable software can be engineered using formal verification techniques
- **5years**
  - foundations laid for work ahead through development of mature tools and standards

verified software $\neq$ verifying the code

First step: research roadmap

roadmap should set out long-term co-ordinated programme of incremental research

1. **accelerated scaling** of the performance, robustness, and functionality of basic verification technology
2. **integration and embedding** of this technology in program development and verification methodologies and environments, and in teaching
3. **pilot projects** to evaluate feasibility and guide technology development
4. **large-scale experiments** that benchmark the technology

The roadmap isn't…

- **a pipe dream**
  - roadmap challenges are concrete, realistic, measurable, verifiable
- **a funding proposal**
  - although there may eventually be funded research programmes to realise some of the goals
- **a needless diversion**
  - co-ordinate existing research to further roadmap goals
- **a jingoistic exercise in chest-beating**
  - international co-operation needed to draft roadmap
  - much more needed to realise goals

Roadmap panels

- **chair:** Shankar (SRI)
- **correct-by-construction:** Leavens (Iowa)
- **integrated verification environments:** Leino (Microsoft)
- **theory:** Naumann (Stevens)
- **system reliability/certification:** Rushby (SRI)
- **interoperable tools/integrated verification environments:** Shankar/Leino
- **verified software repository:** Woodcock (York)
- **pilot projects:** Joshi (JPL)

Goals of the Repository

1. To accelerate the development of verification technology through the development of better tools, greater interoperability, and realistic benchmarks.
2. To provide a focus for the verification community to ensure that the research results are relevant, replicable, complementary, and cumulative, and promote meaningful collaboration between complementary techniques.
3. To provide open access to the latest results and educational material in areas relevant to verification research.
4. To collect a significant body of verified code (specification, derivation, proofs, implementation) that addresses challenging applications.

Goals of the Repository

5. Identify key metrics for evaluating the scale, efficiency, depth, amortization, and reusability of the technology.
6. Enumerate challenge problems and areas for verification, preferably ones that require multiple techniques.
7. Identifying (and eventually standardising) formats for representing and exchanging specifications, programs, test cases, proofs, and benchmarks, to support tool interoperability and comparison.
8. Defining quality standards for the contents of the repository.

Content

1. **tools**
   - integrated verification environments: Spec#, ESC/Java2.
   - language front-ends
   - static analyzers
   - test case generators
   - theorem provers
   - model checkers
   - graphical user interfaces
   - program synthesisers
   - integrated builds
2. **benchmarks**

3. **case studies**
   - Mondex case study
   - electronic voting
4. **interoperability**
   - interchange formats, mappings between models, logics, glue
   - models: state machines, automata, timed automata, hybrid automata, abstractions
   - language syntax and semantics: logics, specification languages, programming languages
   - representations of proofs, test cases, counterexamples
5. **verified libraries:** STL, openSSL, core Java, Bouncy Castle, openPGP, glibc, GMP
6. **educational resources** tutorials, lecture notes

7. ***search capabilities:*** *ontologies, keywords*

8. ***challenge problems:*** *file synchronization, file system, web server, kernel, TCP/IP, SSL, compression, theorem prover kernel, cache consistency, separation kernel, compilers, virtual machines, build tools, fault-tolerant architectures, model reduction for hybrid systems, aspect extraction, scale/parametricity*

9. ***generic properties:*** *Absence of runtime errors, data consistency, timing behavior, accuracy, type correctness, termination, translation validation, serializability, memory leaks, information hiding, representation independence, information flow*

The Mondex case study

- smart card for electronic financial transactions
- Natwest, Platform7, Logica, Oxford (1996)
- first product certified to ITSEC Level E6
- very large project, small formal methods component
- security policy, specification, refinement, hand-written proofs
- completed within time and budget
- cost effective mechanical proof beyond state of the art
- sanitised documentation publicly available

Current Mondex project

- Alloy (Jackson), B (Butler), OCL (Gogolla), PerfectDeveloper (Escher), Raise (George/Hasthausen), VDM (Jones), Z (King)
- one-year, no funding
- how far can we automate the proof?
- group building exercise
- friendly competition
- CASL-like ambitions
- warranted verifier
- meanwhile…
  - Schellhorn (Augsburg): KIV + ASM

Warranted verifier

- clear and clearly fundamental target
- highest level of ambition
- warranted by its designers to be incapable of letting a class of errors occur in any system created with the use of the tool
- deliberately ambitious but achievable within 10–15 years
- take our own medicine
- force designers to scale up and have usability as an explicit design goal

Planning and Road Map

- **Years 1 to 5** Benchmarking, static analysis, and specifications. Individual tool development with opportunistic integration. Small properties of big systems and big properties of small system. Logics for heaps, security, resources, modularity, weak memory models. Multiple code views. Engagement with early adopters.
  - **Year 1** One paradigmatic example of code from specs, assertions from code, assertions from test, verification condition generation, and theorem proving.
  - Initial case studies.
  - User community.

Planning and Road Map

- **Years 5 to 10** Integrated tool development, medium-scale/medium-degree verification, loosely coupled to tightly coupled integration, workflows involving multiple tools. Engagement with industry.
- **Years 10 to 15** Large-scale applications, tool validation, embedded verification, novel models of programming. Developing a broad user base.

Pilot projects: key characteristics

1. **complex**
   - should be challenging: concurrent, fault-tolerant, secure
2. **simple**
   - should be completed in less than two years
3. **important**
   - should have an impact beyond the community
4. **accessible**
   - should have freely available code and models
5. **interesting**
   - should be amenable to different approaches

Some possibilities

- high-impact systems
  - safety and mission-critical systems
  - financial systems
  - electronic voting
- complex, simple, important, interesting
- but difficulties with accessibility

Some better possibilities

- verifying compiler
  - but what target? what impact?
- small operating system
  - real time for embedded devices?
- key operating system component
  - filesystem? resource management?

A verifiable filesystem

1. **complex**
   - reliability (concurrent users, power failures)
2. **simple**
   - manageable — less effort than a kernel
3. **important**
   - almost all data managed by filesystems
4. **accessible**
   - clean, well defined interface (POSIX compliant)
   - well-understood data structures and algorithms
   - open source filesystems
5. **interesting**
   - **C×C** versus **post hoc**, multi-faceted

Directions and challenges

- build a **verifiable** filesystem
- build appropriate models
- check with automated verification tools

Specification

- formal behavioural specification of functionality
- POSIX standard
  - informal prose, ambiguous and incomplete
- formalise key parts of POSIX
- useful starting points
  - Unix/Z, Synergy/ACL2

Underlying hardware

- filesystem properties include robustness wrt power failure
- formal proof requires assumptions about hardware behaviour
  - hard drives, flash memory, …
- must identify and state assumptions
- different hardware, different relaibility guarantees

Data structures and algorithms

- typical filesystem data structures
  - hash tables, linked lists, search trees
- identify design properties
  - datatype invariants
  - locking structures
  - pre/postconditions
- libraries of reusable formally documented components

Reliable flash filesystem for flight software

- NASA/JPL Laboratory for Reliable Software
- exploit automated verification tools
- pilot project for future missions
- flash memory: no moving parts, low power, easily available
- filesystem nontrivial
  - performance
  - concurrency, power failure
  - arbitrary flips, bad blocks
  - cache and write buffer consistency
  - mission requirements

Other pilot projects

- sqlite: open-source database
- Liberouter: open-source router hardware
- open extensible language framework
- MINIX3
- Click router

Finally

come and join us