Verified Software: the next step

Jayadev Misra

Department of Computer Science
University of Texas at Austin

Email: misra@cs.utexas.edu
web: http://www.cs.utexas.edu/users/psp

Acknowledgements

Inspired by: Sir Tony Hoare

Thanks to:

J. R. Abrial	ETH, Zurich
Gerard Holzmann	NASA JPL, Pasadena
Warren Hunt	Univ. of Texas, Austin
Rustan Leino	Microsoft Research, Redmonds
J Moore	Univ. of Texas, Austin
John Rushby	SRI International
N. Shankar	SRI International

Papers

Latest description, written jointly with Tony Hoare.

http://vstte.inf.ethz.ch/pdfs/vstte-hoare-misra.pdf

A faq sheet, jointly written with Tony Hoare.

http://wiki.se.inf.ethz.ch/vstte/index.php/
Verified_Software:_Frequently_Asked_Questions

Growth in System Complexity

Mariner 6 (1969):
128 words of assembly code: approximately 30 lines in C

Lunar Lander (2006): around 1M lines

Growth in System Complexity; continued

- Software crisis coined in 1968,
  NATO Software Engineering Conference, in Garmish, Germany.

- Since then:
  - large applications are larger by about a factor of 2^6 (64 x)
    OS/360 was 5 Million Lines of Assembly (1 Million lines of C)
    Windows XP around 64 Million Lines of C/C++
  - but computers are more powerful by a factor of: 2^{22} (> 4 million)
  - And we have new and hugely improved analysis tools.

Can we win?

Hoare's Verification Grand Challenge: Goals

Verify a significant body of programs.

- given precise external specifications,
- given complete internal specifications,
- generate machine-checked proofs of correctness with respect to a sound theory of programming.

On Science and Engineering

- 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.

Typical Grand Challenges

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

Typical Grand Challenges In Computing Science

Prove that P is not equal to NP	(outstanding)
The Turing test	(outstanding)
The verifying compiler	(abandoned in 1970s)
A championship chess program	(completed 1997)
A GO program at professional standard	(too hard)
Machine translation of English to Russian	(failed in 1960s)

A Typical Grand Challenge Project

It offers fundamental and radical advance In basic Science or Engineering.

- Is a fifteen-year project,
- With world-wide participation,
- And clear evaluation of success or failure.

A Grand Challenge Project needs

- Maturity of the state of the art,
- General support from the international scientific community,
- Long-term commitment from the teams who engage in it,
- Understanding from the funding agencies.

Criteria

- Fundamental: How does it work? Why does it work?
- Historical: formulated long ago.
- Astonishing: Currently beyond our reach.
- Idealistic: does not duplicate commercially motivated evolution of existing products.

A Grand Challenge is not ...

- A solution to all problems
- A science-fiction scenario
- A list of open questions
- A roadmap of strategic directions
- A call for proposals
- A specially funded programme of research
- A plan for a commercial product
- A promise of competitive advantage

Two Grand Challenge Problems in Software Engineering

- Robustness
- Security

Underlying both is construction of “correct” software.

A Program Verifier

- for routine use by computer scientists and practicing programmers,
- goes well beyond debuggers and test engines,
- provides complete assurance.

A Measureable grand challenge

Million lines of verified code.

- A program that is used in practice.
- Source code + specifications + design documents + assertions = 1 Million lines
- A machine checked proof that the code meets the specs.

But correctness is not the end

Software is only one component.

Larger system issues:

- Dependability
- Correctness under evolution
- Graceful degradation under failures, survivability ...

Workshops and Conferences

Partial funding by NITRD HCSS Member Agencies (NSA/NSF)

- A preliminary workshop at SRI Intl. in Washington DC in April 2004; attended by about 50 participants.
- A larger workshop (Feb 21–23, 2005) at SRI Intl. in Menlo Park.
- A working conference held in Zurich, Switzerland during the week of Oct. 10, 2005 (www.vstte.ethz.ch).
- A series of Miniworkshops just held at SRI International in Menlo Park.
- A conference scheduled at Dagstuhl, Germany for July 2006.

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.

1969: King writes a dissertation on automatic program proving through verification condition generation.

Fast Forward . . .

- 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 program analysis: A380, Ariane-5, Linux/OpenBSD kernel.
- Crypto-protocol verification.
- Model-based design of embedded systems software.

Formal Proofs in Industry: ACL2

- ACL2 is the industrial strength version of the Boyer-Moore system.
- Boyer, Moore, Kaufmann recently won the ACM Software award.
- Applied to a variety of problems ranging from pure math to industrial products.

Applications of ACL2

- Verification that register-transfer level description of AMD Athlon^{TM} processor's elementary floating point arithmetic circuitry implements the IEEE floating point standard.
- Similar work for components of the AMD-K5 processor, the IBM Power 4, and the AMD Opteron^{TM} processor.
- Properties of microarchitectural model of a Motorola digital signal processor (DSP).
- Verification that microcode for the Rockwell Collins AAMP7 implements a given security policy.

Applications of ACL2; contd.

- Verification that the JVM bytecode produced by the Sun compiler `javac` on certain simple Java classes implements the claimed functionality.

- Verification of the soundness and completeness of a Lisp implementation of a BDD package.

- Verification of the soundness of a Lisp program that checks the proofs produced by the Ivy theorem prover from Argonne National Labs; Ivy proofs may thus be generated by unverified code but confirmed to be proofs by a verified Lisp function.

Formal Proofs in Industry (Figures from the B effort)

n lines of final code implies $n/3$ proofs

95% of proofs discharged automatically

5% of proofs discharged interactively

350 interactive proofs per man-month

A particular Example:

- 60,000 lines of final code ; 20,000 proofs ; 1,000 interactive proofs

- 3 man-months for 1,000 interactive proofs (1000/350)

- Far less expensive than heavy testing

Line 14 Paris Metro, Roissy Shuttle

	Paris Metro	Roissy Shuttle
Year completed	mid 90s	2005
Number of ADA lines	86,000	158,000
Number of proofs	27,800	43,610
Percentage of interactive proofs	8.1	3.3
Interactive proofs time in Man.Month	7.1	4.6

Table 1: Two case studies using Classical-B

Software Specification Document Size

WCU	(84 functional modules)	228
Block Logic	(30 functional algo)	51
Route Logic	(37 functional algo)	80
Mode Logic	(50 functional algo)	98

Table 2: Size of input documents (in pages)

Proofs: Automatic and Interactive

	Lemmas	Rate
Grand total	43,610	100%
Force 0 prover	38,822	89%
Force 1 prover	1,397	3%
Generic demonstrations	1,950	5%
Total of automatic proofs	42,169	97%
Interactive proofs	1,441	3%
Number of interactive proofs/day		15

Table 3: Figures on Proofs

Meteor Project: Using B

Version	Release Date	Automatic proof	Interactive Proof
1.6	12/94	35%	19,500
2.9	9/96	60%	12,000
3.3	10/97	80%	6,000

Table 4: Number of lemmas: Total 30,000

Proof metrics: Meteor

Software Product	Code size	Automatically Proven Lemmas	Automatically, customized Tool	Interactively Proven Lemmas
Wayside	13,600	79 %	89 %	1,463
On-Board	7,600	84 %	90 %	775
Line	6,600	80 %	99 %	16
Total	27,800	81 %	92 %	2,254

Table 5: Proof metrics: Meteor

Status Summary

- We have travelled far and deep:

  From toy problems to industrial strength software.

  From trivial properties to deep logical properties.

- But we have miles to go before we sleep:

  Verification should become routine, more like compiling.

  Verifier should handle programs of much larger size.

  It should give assurances about absence of certain kinds of errors.

- A concerted effort is more likely to foster faster progress.

Grand Challenge Project: 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.

Broad Consensus

- Tool development
- Tool-set Integration
- Experiments
- Foundational work

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: ACL/2, 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.

Industrial effort: Spec# Programming System

- Being developed at Microsoft Research at Redmonds by Leino, Schulte, et. al.
- A superset of C#, targeted for .NET platforms.
- The goal is to improve the quality of general purpose, industrial-strength software.

Features of Spec#

- Enhance type system to aid design and automatic checking.
- Method specifications using contracts (pre- and postconditions), as in Eiffel.
- Three levels of checking:
  - Type checker: can check some aspects of dataflow.
  - Compiler emits run-time checks, which enforce contracts.
  - Static program verifier.
- Guarantee soundness for a large portion of code.

Spec# Static program verifier (Boogie)

- Translates program to intermediate code.
- Infers loop invariants using abstract interpretation.
- Generates verification conditions, to be checked by a theorem prover.
- Translates counterexamples as error messages of the source code.
- Enforces all contracts, and the entire programming methodology.

Programming Methodology

Examples: Alloy, B Method, I/O Automata, Spec#, Specware, UNITY, 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.

Programming Methodology: Lunar Rover

Simplified Hierarchical Redundancy

- Every critical module is given a simpler and more strongly verified backup with reduced functionality.
- The prime module is designed for performance and functionality. Neither fully checked.
- The backup is designed for verifiability, fault containment, survivability and recovery from unanticipated software faults.
- Normal Operation: Prime Module.
  Under fault: Backup module.

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)

Rajeev Joshi's Challenge at NASA

- An ultra-reliable posix compliant flash file system software module
- preserves filesystem integrity across random reboots/power losses

Tool-set Integration

Goal:

- The user of a verifier knows no more than a programmer about the structure of a compiler.
- Unachievable goal, at present.

Approach:

- Provide a set of verification tools.
- Provide means to use multiple tools and transfer data among them.

Some questions about tool-set integration

- How can a user run a dialog with a bunch of tools?
- How can various tools use the same packages, Model checkers, SAT solvers, yet customize them for their specific needs?

Evidential Tool Bus of John Rushby

- In the beginning there was just (interactive) theorem proving
- Then there were VC generators, decision procedures, model checkers, abstract interpretation, predicate abstraction, fast SAT solvers,. . .
- Now there are systems that use several of these (SDV, Blast,. . .)
- For 15 years from now, we need an architecture that allows us to make opportunistic use of whatever is out there.

  And to assemble customized tool chains easily.
- It should be robust to changes (in problems and tools), and deliver evidence.

Integration of Heterogeneous Components

Modern formal methods tools do more than verification

They also do refutation (bug finding)

And test-case generation

And controller synthesis

And construction of abstractions,

And generation of invariants, and ...

Observe that these tools can return objects other than verification outcomes

Counterexamples, test cases, abstractions, invariants

A Tool Bus

Construct these customized combinations and integrations easily and rapidly.

The integrations are coarse-grained (hundreds, not millions of interactions per analysis), so they do not need to share state

So we could take the outputs of one tool, massage it suitably and pass it to another and so on

A combination of XML descriptions, translations, and a scripting language could probably do it

Suitably engineered, we could call it a tool bus

Tool Bus Judgments

The tools on the bus evaluate and construct predicates over expressions in the logic—we call these judgments

Parser: A is the AST for string S

Prettyprinter: S is the concrete syntax for A

Typechecker: A is a well-typed formula

Finiteness checker: A is a formula over finite types

Abstractor to PL: A is a propositional abstraction for B

Predicate abstractor: A is an abstraction for formula B wrt. predicates ϕ

GDP: A is satisfiable

GDP: C is a context (state) representing input G

SMT: ρ is a satisfying assignment for A

Tool Bus Queries

Bus exploits tools' capabilities for delivering queries to them.

Query: well-typed?(A)

Response: PVS-typechecker(. . .) —- well-typed?(A)

The response includes the exact invocation of the tool concerned

Queries can include variables

Query: predicate-abstraction?(a, B, ϕ)

Response:
SAL-abstractor(. . .)—- predicate-abstraction?(A, B, ϕ)

The tool invocation constructs the witness, and returns its handle A

An Evidential Tool Bus

Each tool should deliver evidence for its judgments

- Could be proof objects (independently checkable trail of basic deductions)
- Could be reputation (“Proved by PVS”)
- Could be diversity (“using both ICS and CVC-Lite”)
- Could be declaration by user
  - “Because I say so”
  - “By operational experience”
  - “By testing”

And the tool bus assembles these (on demand)

And the inferences of its own scripts and operations

To deliver evidence for overall analysis that can be considered in a safety or assurance case—hence evidential tool bus

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: analyze models, compose models, generate optimized code, integrate existing code, support finely tuned static and dynamic analyses.
- Formal tool bus 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.

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. Big 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.

From Tony Hoare

- The five-year goal is mainly to show that there are no show-stoppers.

  It will answer two questions:

  1. can the community work together on tools and their application?
  2. And do there exist verified programs that people might actually want to use?

- In the rest of the project, we can begin the laborious process of turning the existential quantifier into a universal one.