Z User's Meeting, Sept. 1995

Mechanizing Formal Methods: Opportunities and Challenges

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

The Opportunity

- Formal methods contribute useful mental frameworks, notations, and systematic methods to the design, documentation, and analysis of computer systems

- But the primary benefit from specifically *formal* methods is that they allow certain questions about a design to be answered by symbolic calculation (e.g., formal deduction, model checking)

- These symbolic calculations can be used for debugging and design exploration as well as post-hoc verification

- Comparable to the way computational fluid dynamics is used in the design of airplanes and jet engines

The Challenge

- Just like any other engineering calculations, it's tools that make formal calculations feasible and useful in practice
- To undertake formal calculations on real problems, you need automated tools (i.e., theorem provers and model-checkers) of *massive* power
  - Proof editors: off by 3–4 orders of magnitude
  - HOL: off by 2–3 orders of magnitude
  - SMV, PVS: off by 1–2 orders of magnitude
- Whose notations are acceptable to practitioners
  - So, cannot increase automation at expense of expressiveness

The Challenge Within The Opportunity

- Interested in more than classical correctness
  - Challenging requirements to *validate* them
  - *Adjusting* designs to changed requirements
  - *Debugging* designs
  - *Exploring* alternative designs
  - *Pruning* and *sharpening* assumptions
  - *Deriving* test cases

- Hence, seldom proving true theorems
  - Need to return useful information from (rapid) failure
  - So classical theorem provers (e.g., Otter) are of little use in mechanizing formal methods

Room For Maneuver: Lifecycle Stages

- Don't apply automated formal methods to all stages of the development lifecycle: just those where other methods are inadequate
  - In avionics, for example, coding consumes less than 5% of costs and contributes less than 2% of faults
  - Get much bigger bang for the buck with requirements (50% of faults, and the most expensive ones to correct)
  - And with architectures, interfaces, designs, algorithms
- Shake off the heritage of "program verification"

Room For Maneuver: Properties

- "Small theorems about big systems"

- Don't apply automated formal methods to every property of a design: just those where other methods are inadequate
  - Redundancy management in aircraft flight control
  - Address-space separation in security kernels
  - Feature interaction in phone systems (and printers)
  - Pipeline correctness, superscalar dispatch, out-of-order cleanup, floating point in processors
  - Cache coherence in memory systems

- Important theorems about complex systems

Meeting The Challenge

- Focus on the *hardest* and most significant problems
  - Concurrency, fault tolerance, real time
- Automate (efficiently!) everything that is decidable
  - Propositional calculus, linear arithmetic, finite-state model checking
- Provide heuristic automation for the obvious cases of everything else
  - Instantiation, induction
- Leave the human free to concentrate on, and to direct, the steps that require real insight

Essential Techniques: Theorem Proving

- *Must* use decision procedures for ubiquitous theories
  - Linear arithmetic (integers and reals): sup-inf, loop residue
    And extensions (e.g., signs of nonlinear products and exponentials, infinities)
  - Equality: congruence closure
  - Propositional satisfiability: OBDDs
  - Function overriding, other conditional expressions

- And must make them work together: Nelson-Oppen, Shostak

- Must closely integrate all this with conditional term rewriting (for which you must have good term indexing and caching)

- Allow users to customize the system safely: tactics

Essential Techniques: Model Checking

- Proving properties of concurrent systems is tedious
  - Usually several interdependent invariants must be strengthened iteratively
- Can instead be done by brute force if the system can be reduced to finite state and the property of interest can be described in branching time temporal logic—model checking
  - Explicit state enumeration (Murϕ)
  - With approximation (Spin)
  - Symbolic state enumeration (SMV)
- Related methods: linear-time temporal logic model checking, language inclusion methods (COSPAN), refinement (FDR)
- Most can be expressed in terms of decision procedure for μ-calculus

Essential Techniques: Overall Synthesis

- Theorem proving and model checking work together
  - Use theorem proving to show that a finite state abstraction preserves property of interest
  - Model check that abstraction
- Aggressive abstraction ("downscaling") is worthwhile in the early stages and should be supported
  - Generally find more errors by exploring *all* behaviors of simplified model than *some* behaviors of more complex one
- Direct and symbolic execution have their uses, too
- Want a graduated series of techniques with increasing cost and power

Influence on Specification Language Design

- Powerful theorem proving influences choice of logic
  - Efficient equality reasoning requires total functions
- And prefers structured definitions over arbitrary axioms
  - In a logic of total functions, recursive definitions must be shown to terminate
  - Which may require the (sub-ε_0) ordinals
  - Will also want recursive datatypes ("free types" in Z), and inductive definitions
  - Side benefit is earlier and stricter error detection
- Need a type system that assists theorem proving and model checking
  - Careful treatment of numeric (sub)types
  - Tracking heriditarily finite types (for model checking)

Opportunities In Specification Language Design

- Availability of powerful theorem proving creates new opportunities

- Can contemplate using theorem proving in typechecking (i.e., allow typechecking to become undecidable)

- Biggest benefit is in *predicate subtypes*—PVS example:

  $\texttt{nonzero\_real: TYPE = } \{ \texttt{ r: real | r} \neq \texttt{0 } \}$

  $\texttt{/: [real, nonzero\_real} \rightarrow \texttt{real]}$

  $\texttt{div\_prop: LEMMA } \forall \texttt{ (x, y: real): x} \neq \texttt{y} \supset \texttt{(x-y)/(y-x) = -1}$

  Generates the following proof obligation

  $\texttt{TCC1: OBLIGATION } \forall \texttt{ (x, y: real): x} \neq \texttt{y} \supset \texttt{(y-x)} \neq \texttt{0}$

  (called a Type Correctness Condition, or TCC, in PVS)

Higher-Order Predicate Subtypes

- Higher-order logic provides considerable additional expressiveness at little cost
  - Induction schemes become first-class formulas
  - Sets become predicates and inherit theorem proving muscle

- Surjections, for example, become a subtype of the functions

```
function_props[dom, rng: TYPE]: THEORY
  surjective?(f): bool = ∀ (r: rng): ∃ (d: dom): f(d) = r
```

So that

```
even_nat: TYPE = {i: nat | ∃ (j: nat): i = 2×j}
half: (surjective?[even_nat, nat]) = λ (e: even_nat): e/2
```

Generates the proof obligation

```
TCC2: OBLIGATION ∀ (n: nat): ∃ (e: even_nat): half(e) = n
```

Symbiosis Between Language and Deduction

- To get the most from mechanization, language and tools must be designed together

- Some tradeoffs are necessary

- But new opportunities are created
  - Enhanced expressiveness
  - Earlier and stricter error detection
  - Integrated treatment of properties what would otherwise require special-purpose tools (e.g., state machine invariants)

Some Successful Applications of Mechanization—Software

- The best Byzantine fault-tolerant algorithms for synchronous distributed systems were all developed through analysis in PVS
  - Agreement, Clock Synchronization, Diagnosis
  - Recently, used Murϕ to explore beyond worst-case bounds
- Support for tabular specification methods (SCR, Tablewise) was prototyped in PVS with negligible effort
  - TCCs generate well-formedness checks
  - BDDs and linear arithmetic discharge them
  - Model checking used to test invariants and liveness
- Complex bounded retransmission protocol was verified in PVS using combined methods
  - Prove finite state abstraction preserves property of interest
  - Model check the abstraction

Some Successful Applications of Mechanization—Hardware

- PVS was used to prove consistency between axiomatic definition of Sparc V9 RMO memory model and executable description in Murϕ
  - Justifies subsequent symbolic testing using Murϕ
- Cache coherence protocol for Stanford Flash processor was checked in Murϕ and then verified in PVS
- RTL-level microarchitecture of Collins AAMP5 avionics processor (500,000 transistors) was specified in PVS, and its microcode verified for a representative set of 11 instructions
  - Found seeded bugs, and two new ones
  - Used model checking to discharge assumptions about interactions with LFU and cache

Conclusion

- Mechanization is not the most important thing about formal methods
  - It's the *only* important thing
- Mechanization has to be very powerful to be of any use
  - And must encompass many capabilities
  - But allows formal specifications to be more than documentation
  - And does more than post-hoc verification
- Efficient mechanization imposes some requirements on language design
  - But opens up new opportunities
- Really powerful mechanization is within reach

To Find Out More About PVS

- Browse papers and technical reports at
  `http://www.csl.sri.com/fm.html`
  - `CSL-95-1.html` is a chapter for the FAA Digital Systems Validation Handbook that discusses issues in formal methods and assurance
  - `CSL-93-7` is a much longer and more technical treatment
  - `cav95.html` describes the model checker integration in PVS
- You can get alpha release of PVS2 by anonymous FTP from `ftp.csl.sri.com` in `/pub/pvs/pvs2` (or WWW link at URL above)
  - Allegro Lisp for SunOS 4.1.3 and Solaris
  - Need 32M memory, 100M swap space, Sparc 2 or better
  - Mirror sites at York and Ulm
  - General release awaits completion of updated documentation