FM'99 Toulouse, France, 23 Sep 1999, 9am

Mechanized Formal Methods: Where Next?

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, California, USA

Contributions of Formal Methods

- Formal methods have contributed much to the development of computer science
  - Mental frameworks, notations, systematic methods
- But there's one big problem. . .

The Problem

Not many practitioners use our stuff

(By practitioners, I mean software and hardware engineers, designers, testers, analysts etc.)

Getting Practitioners to Use Formal methods

- Address their problems
- Provide effective solutions
- Build on what practitioners already do and know
- Fit into the design flow

Address Practitioner's Problems

- Correctness is not a goal in most applications
- The product has to be "just good enough"
  - So help with debugging, testing, analysis
- The big problems are not in coding (or even in algorithms)
- They are in requirements, design, maintenance
  - So help with requirements analysis, design exploration, specification and checking of interfaces

Provide Effective Solutions

- Debugging
  - Find more bugs, and higher-value bugs than otherwise
  - Find them earlier and more cheaply
- Testing
  - Improve coverage and reduce cost
- Requirements analysis
  - Be able to *do* something earlier than otherwise
- Design exploration
  - Support "what if" investigations
- Interface specification and checking
  - Checking semantics rather than syntax

All of these require mechanization

Provide Effective Solutions (ctd.)

- Designs change and evolve
- You can try to provide formal support for those processes
  ... good luck
- Or you can try to help debug and test the changed artifacts, and analyze the consequences of those changes
- This is feasible only if mechanized

Mechanized Formal Methods

Mechanization is not

The most important thing about formal methods

It's the only important thing

Mechanizing Formal Methods

- Logic is the applied math of computer science
- Therefore, mechanization of formal methods uses the techniques of automated deduction
  - Model checking and theorem proving
- Mathematicians prove theorems
- So does that mean formal methods should look to mathematics and mathematical logic for its inspiration?

Mechanized Formal Methods and Mathematical Logic

- The techniques of mathematical logic are important for mechanized formal methods
- But the disciplines have different goals
- Mathematical logic
  - Foundational and metamathematical issues
- Formal methods
  - Calculating answers to practical questions

"The arithmetic seems correct, yet I find myself haunted by the idea that the basic axioms on which the arithmetic is based might give rise to contradictions that would then invalidate these computations."

Mechanized Formal Methods and Engineering Calculations

- Engineering calculations are a better model for mechanized formal methods than foundational mathematics
- Examples: computational fluid dynamics (aeronautics), or finite-elements analysis (structures)
- Soundness is important
  - But it's not the only important thing
- The other important things about mechanization
  - Speed, scaling, power, automation, predictability
  - Speed, scaling, power, automation, predictability
  - Speed, scaling, power, automation, predictability
- (This point of view is an underappreciated gift from the model checking community)

Whence
Speed, Scaling, Automation, Power, Predictability?

- From decision procedures
- Many useful theories are decidable
  - Propositional calculus
  - Linear integer arithmetic
  - Real arithmetic
  - Equality over uninterpreted function symbols
  - Bitvectors
  - Weak second order theory of one successor (WS1S)
  - Recursive datatypes (lists etc.)
  - Finite μ-calculus (model checking)
- The issues outstanding are in their combination and extension

Decision Procedures

- Individually decidable theories can become undecidable in combination
  - E.g., quantified linear arithmetic and equality over uninterpreted function symbols
- So need to make pragmatic compromises
  - E.g., decide the combination of unquantified theories
  - And deal with quantification heuristically
- Austere combinations can sometimes be much more efficient than richer ones
  - E.g., propositional calculus with equality over uninterpreted function symbols
  - Cf. same plus linear arithmetic

Where Next? More Decision Procedures!

- Maximize utility of existing decision procedures
  - E.g., generalize equality to congruences,
    linear arithmetic to linear orders
- Decide additional theories
  - E.g., fragments of set theory, queues
- Provide pragmatically useful heuristic extensions
  - E.g., nonlinear integer products, elementary real functions
- Develop new combinations and allow mix-and-match
- Provide proof objects (checkable justifications)
  - And counterexamples
- Liberate the procedures from closed verification systems

Does That Solve The problem?

- Given effective decision procedures (and rewriting)
- And some other sensible design decisions
- One can build quite effective support tools for formal methods
    - E.g., . . . PVS . . .
- But they still don't get used much outside the formal methods community
- Part of the problem is they still are oriented towards verification rather than debugging, testing, etc.
- But another problem is the nature of the interaction and guidance required by the user

Human Interaction And Mechanized Formal Methods

- Not everything is decidable
- So all general-purpose tools for formal methods require human interaction and guidance in some way or other
  - Automatic theorem provers
    - ⋆ Setting dials and switches (resolution)
    - ⋆ Choosing the shape and ordering of lemmas (ACL2)
  - Interactive theorem provers
    - ⋆ Explicit guidance, or programming tactics/strategies
  - Model checkers
    - ⋆ Reducing to a suitable special form (e.g., finite state)
    - ⋆ Setting dials and switches (e.g., variable orderings)
- But practitioners tolerate some ways better than others

Reducing/Improving Human Interaction In Mechanized Formal Methods

- Find applications that are decidable
  (given domain-specific support)
  - Paradigmatically, equivalence checking in hardware
  - E.g., consistency and completeness checkers for Statechart/tabular specification methods
- Find applications where (occasional) inability to prove true theorems (i.e., false negatives) does not devalue the method
  - Those where current methods are even more approximate
    E.g., static analysis, program slicing
  - Those where counterexamples help the user discriminate
    E.g., model checking downscaled models
- Interact in ways closer to practitioner's concerns

What Practitioners Will Tolerate

- (Some) practitioners will put large effort into reducing
  (slicing, downscaling, simplifying, abstracting)
  designs to make them acceptable to a model checker
  - Even though it's difficult and expensive to maintain correspondence between the real and the reduced models as the design evolves
- But they are not willing to guide a theorem prover
  (that might produce more information more quickly)
- Conjecture
  - That's because the reductions resemble design activity
  - Whereas guiding a theorem prover does not

Build On What Practitioners Do and Know

Proposal: Change the interaction with a theorem prover to focus on the construction of abstractions and invariants

Triple benefit:

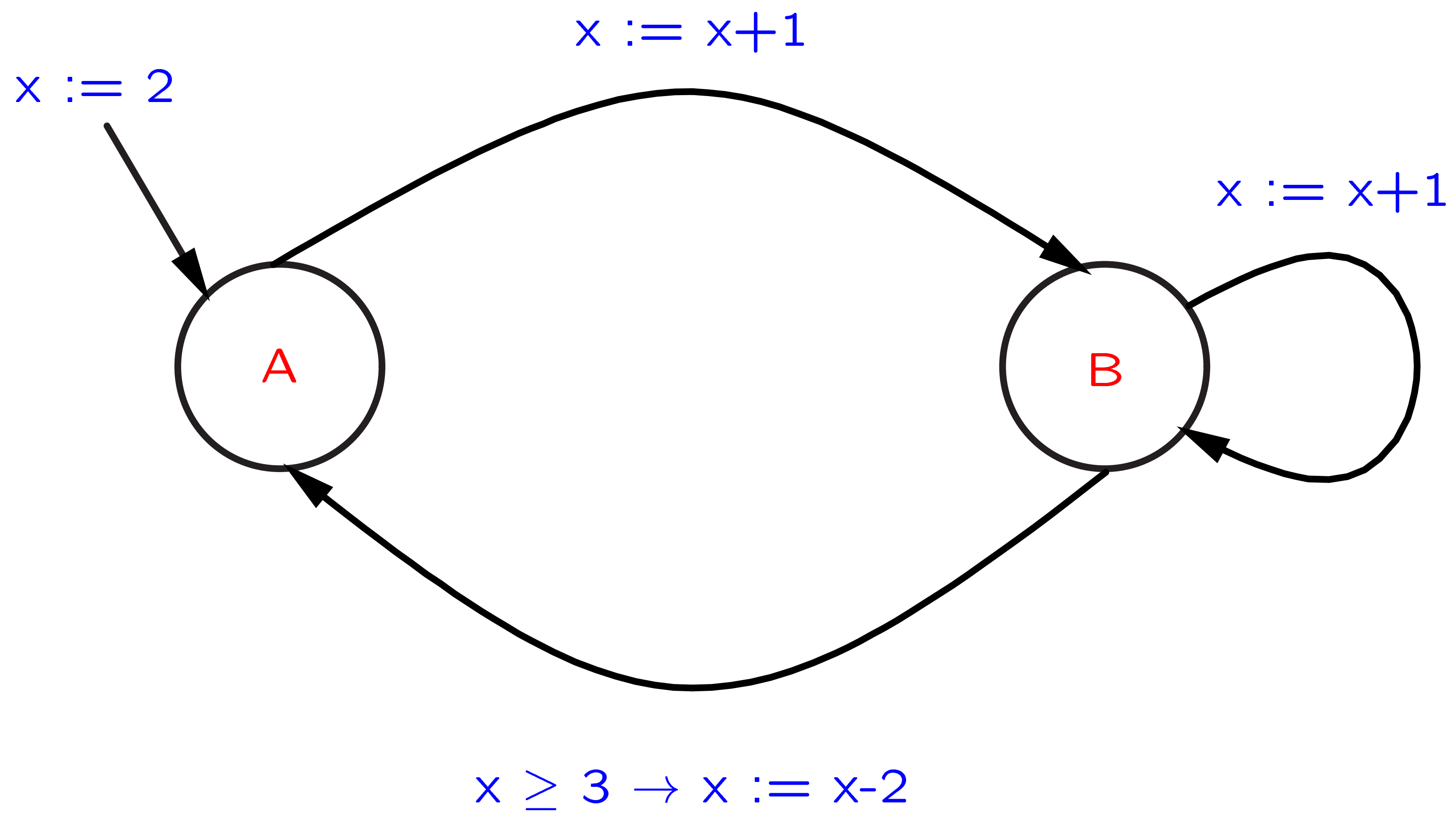
- More acceptable form of interaction (maybe)
  i.e., suggesting abstractions
- Can tolerate some false negatives
- And it mechanizes the reductions required for model checking

Mechanized Abstraction and Invariant Generation

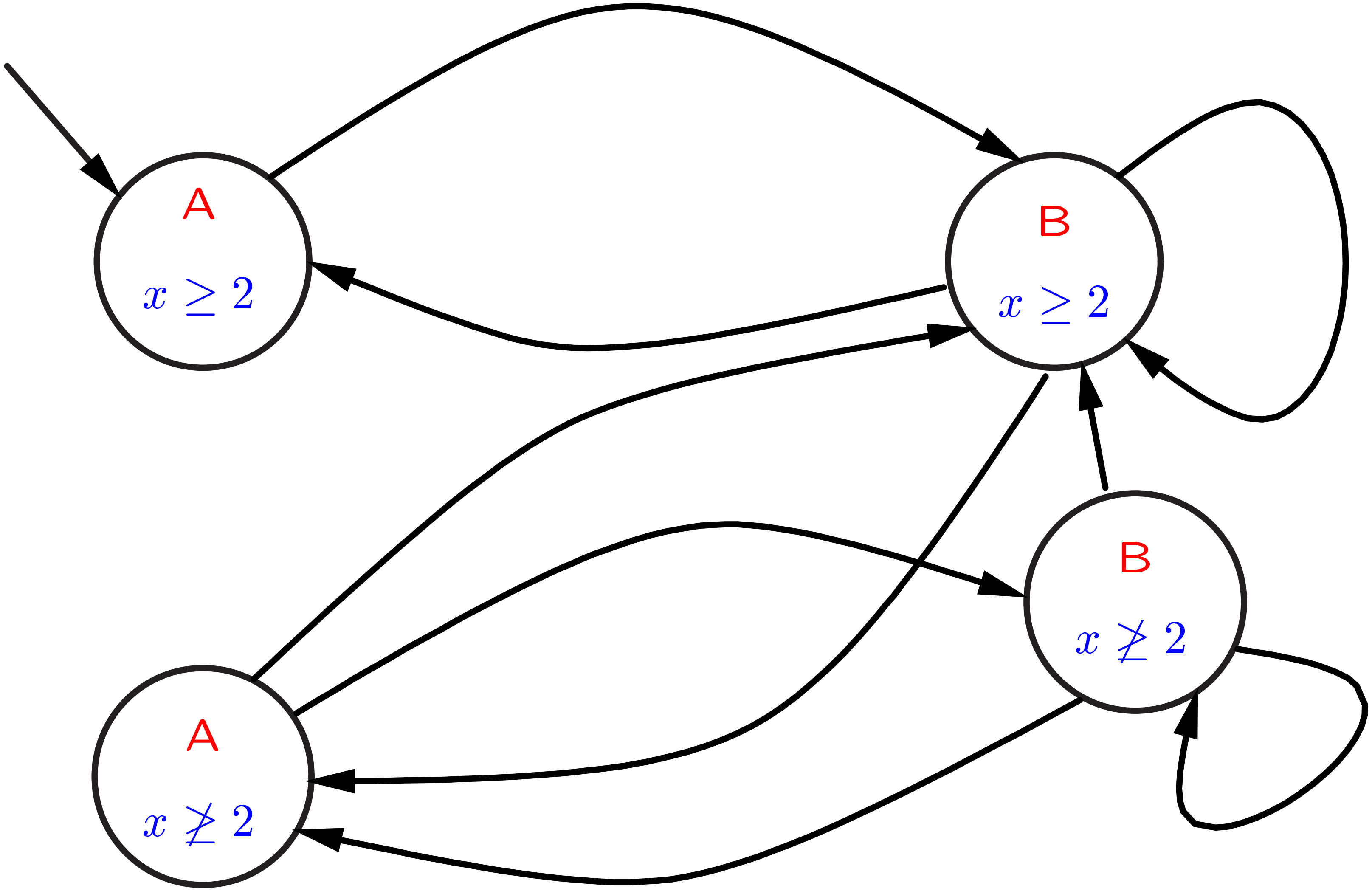
- Predicate abstraction
  - Replace predicates on state variables by uninterpreted boolean variables
  - Construct a transition relation over these abstract variables that approximates the original
    - ⋆ Start with universal transition relation on the abstract variables; then for each arc
      - ◇ Generate the VC that allows it to be removed
      - ◇ Leave it in if cannot prove the VC
    - ⋆ There is also another method
- Invariant generation
  - Use various kinds of propagation, fixpoint computations, to calculate invariants, or to strengthen existing ones
- Invariants can help predicate abstraction

A Trivial Example

Show that when control is at B, then $x \geq 2$



Predicate Abstraction on $\mathtt{x} \geq 2$



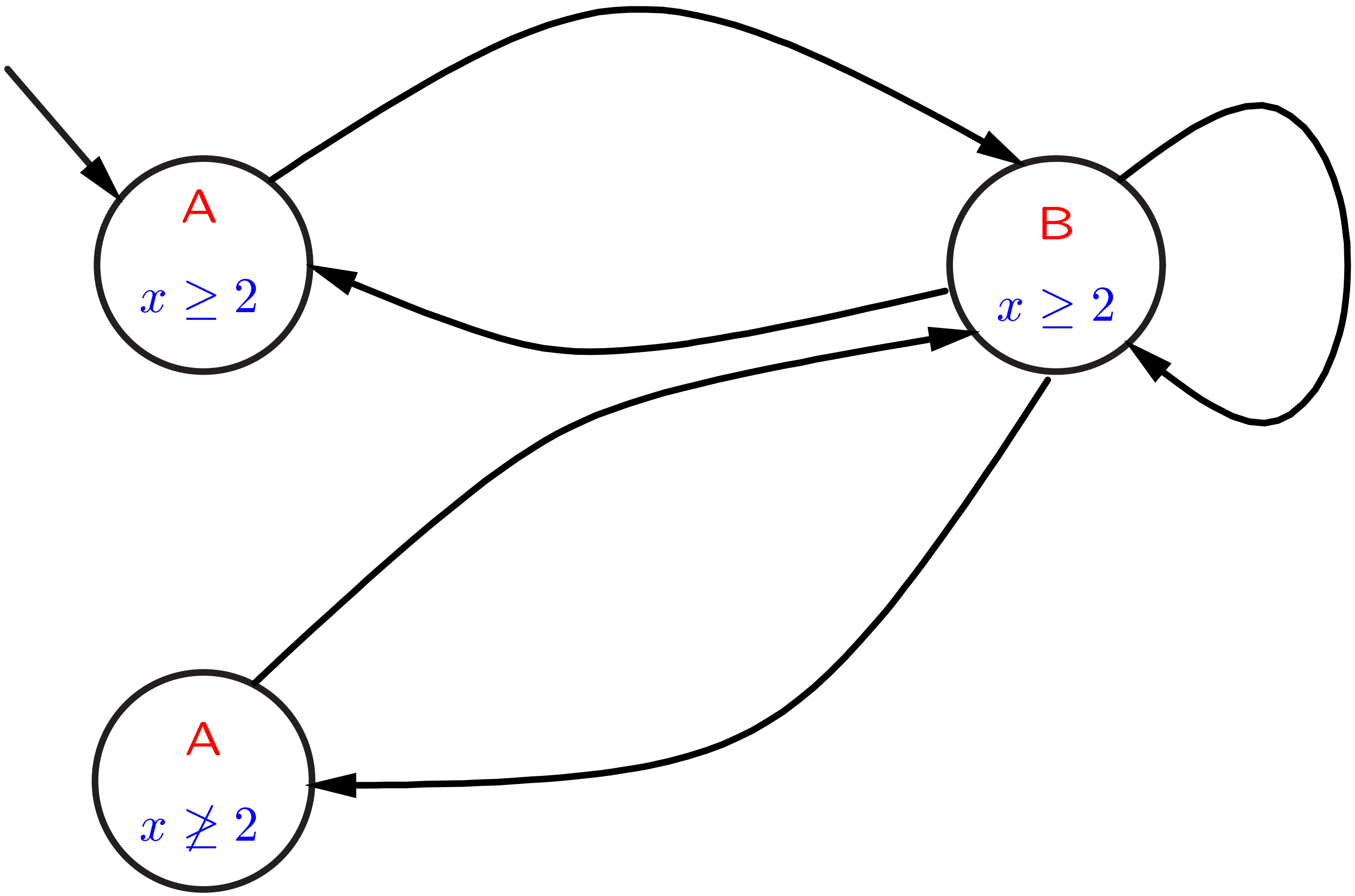
Desired property is not satisfied

Diagnosing The Problem

- Model checking produces this counterexample trace
  - $\{\texttt{A},\ x \geq 2\} \rightarrow \{\texttt{B},\ x \geq 2\} \rightarrow \{\texttt{A},\ x \not\geq 2\} \rightarrow \{\texttt{B},\ x \not\geq 2\}$
- If we "concretize" this we see that the last transition is impossible in the concrete system
  - $\{\texttt{A},\ x \geq 2\} \rightarrow \{\texttt{B},\ x \geq 2\} \rightarrow \{\texttt{A},\ x \not\geq 2\} \rightarrow \{\texttt{B},\ x \not\geq 2\}$

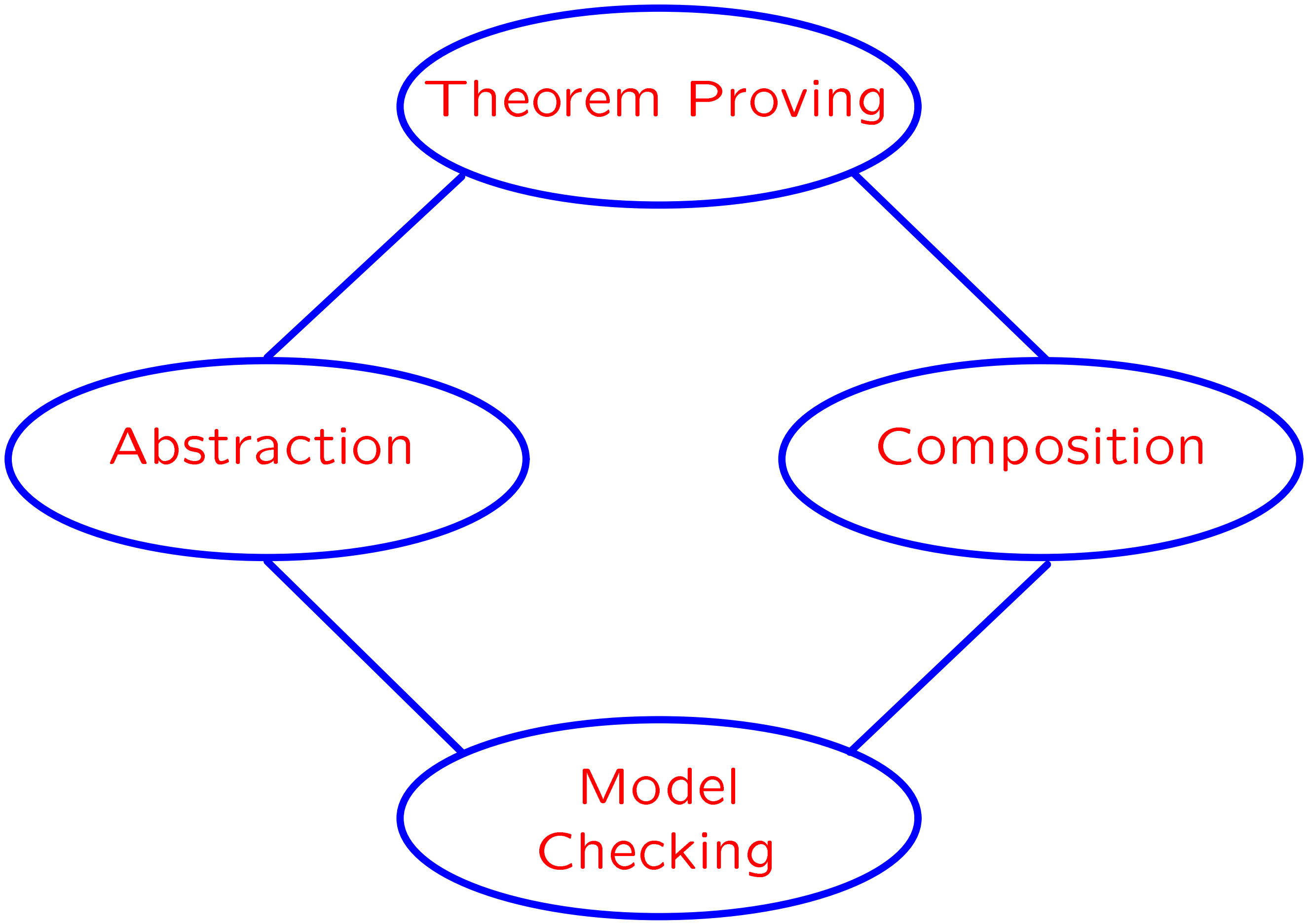
    2 3 1 2
- Our abstraction is too coarse: need to abstract on additional predicates or use invariants
- Invariant generation provides $\texttt{x} \geq 1$

Refined Abstraction on $\texttt{x} \geq 2$



The desired property is satisfied

Abstraction Is A Bridge From Deductive To Algorithmic Methods



The Bridge Goes In Both Directions

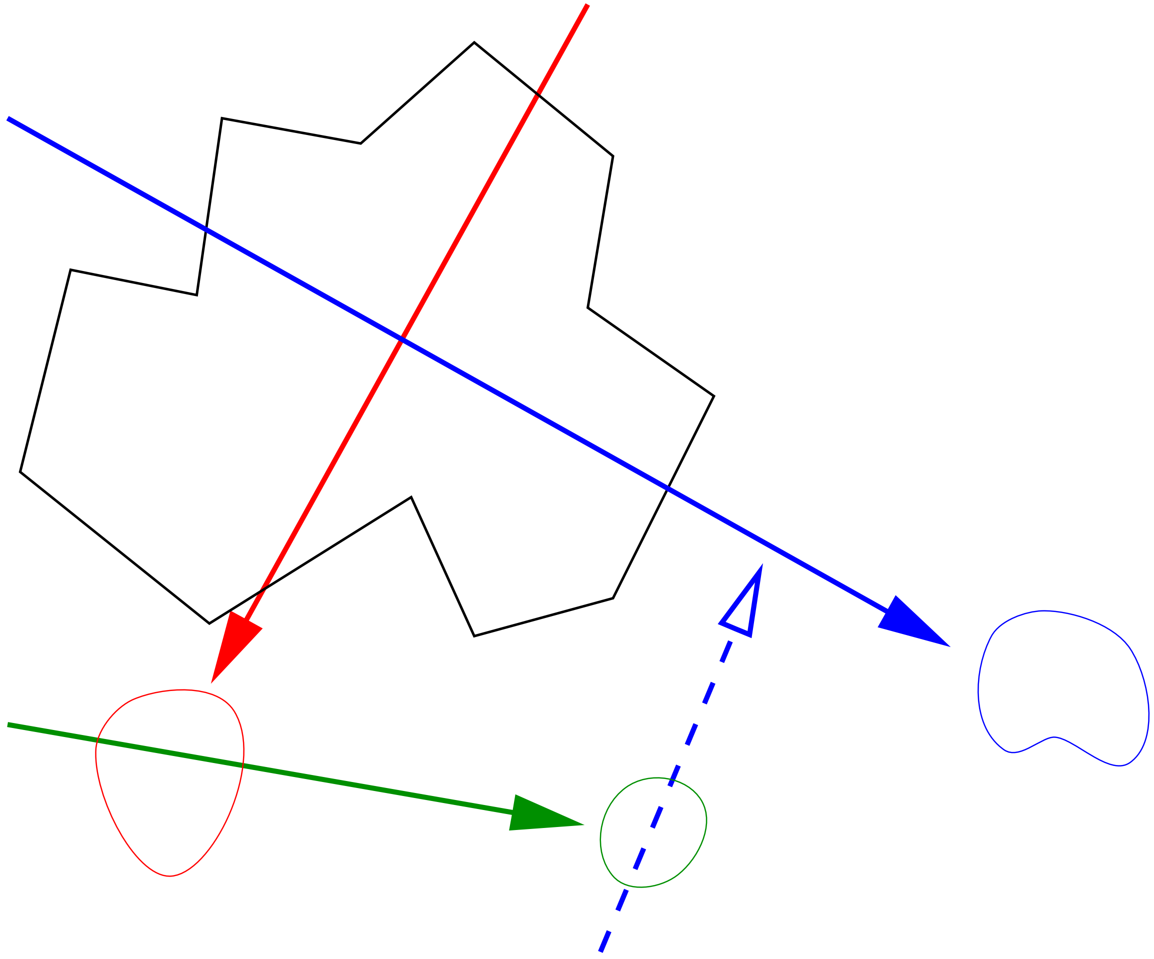
- Model checkers often calculate the reachable stateset
  - Which is the strongest invariant

  And then throw it away
- The concretization of the reachable states of an abstraction is an invariant of the concrete system
  - And often a strong one
- Modify a model checker to return the reachable states as a formula that a theorem prover can manipulate

Where Next? Integrated, Iterated Analysis!

- Think of calculating and accumulating properties rather than just checking a specific property
- Properties convey information and insight,
  and help construct abstractions
- Which can be used to calculate new properties
- Requires that tools return symbolic values and properties rather than just yes/no results of verifications

Integrated, Iterated Analysis



Examples

- Bounded retransmission protocol (FME '96)
  - 57 manually developed invariants for direct proof
  - 45 to justify an abstraction

  Two man-months of effort by Havelund and Shankar
- Can now be done automatically in minutes (CAV '99)
  By Saïdi and Shankar, and by Abdulla et al.
- Other highly mechanized examples in CAV '99
  (translation validation, pipelines, garbage collector)
  By Pnueli, Rodeh, and Shtrichman; Bryant, German, and Velev; Das, Dill, and Park
- And in FMCAD '98 (Tomasulo, cache coherence)
  By McMillan and by Eiriksson

Missing Pieces

- Connection to standard languages
  - Start from, end with, or at least provide mechanized connections to, notations in widespread use
    E.g., Java, Verilog, UML, Matrix-X, Statecharts
  - Need slicers to extract fragments of interest
- Connection to standard tools
  - Glue code needed to fit into the design flow

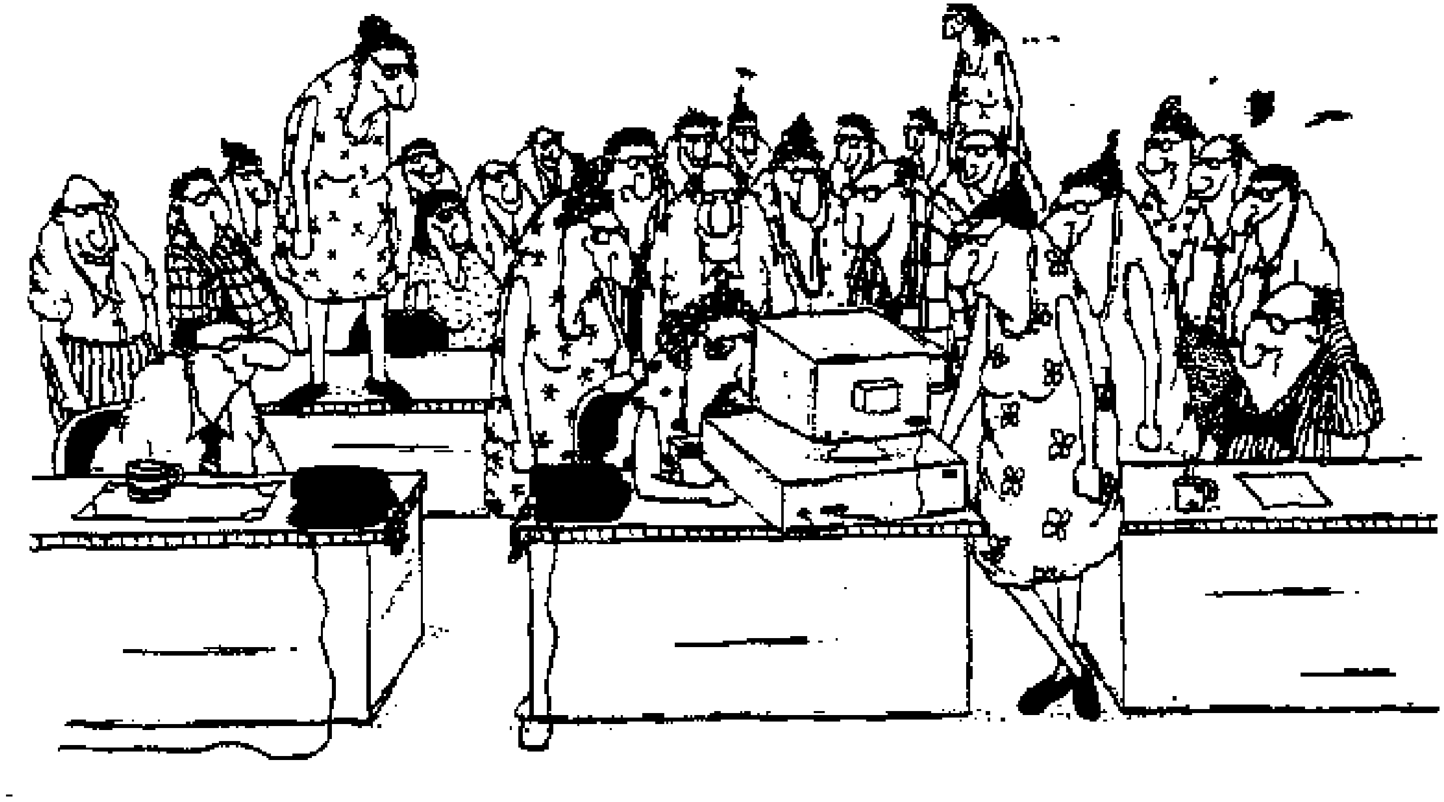
How To Exploit This Technology

- We (could) have the tools, all we need now are methodologies to exploit them effectively
- I'm looking for ideas, but I suspect the best prospects are in three areas
  - Use deduction to refine the analyses of standard tools
    - Compilers (ESC, optimization), static analyzers, slicers
  - Provide new, targetted, analysis tools
    - E.g., Daniel Jackson's Alcoa for object models
  - Develop powerful methods for specific domains
    - E.g., conformance testing for protocols, railroad switching, mode logic for embedded systems

Prospects

- Have never been brighter
- Expectations for what is “good enough” are increasing
- Even simple mechanized formal methods find large numbers of errors that have been missed by other methods
- The technological building blocks are available,
- The challenge is to figure out how best to use them
- So that practitioners will use our stuff

Excitement Grows As The Model Checker Is Applied To The Final Abstraction



Credits

None of the work, and few of the ideas, are mine; they come from my colleagues

- Shankar: SAL

  Being developed with David Dill (Stanford)

  And Tom Henzinger (Berkeley)

  Implementation by César Muñoz, Harald Rueß Vlad Rusu, Eli Singerman, and Ashish Tiwari
- Hassen Saïdi: The Invariant Checker, PVS abstractor
- Saddek Bensalem, Yassine Lakhnech, Sam Owre: InVeSt
- Judy Crow, David Cyrluk, Sam Owre, Shankar, Dave Stringer-Calvert: PVS
  - Freely available under license to SRI
  - From `http://pvs.csl.sri.com` (for Solaris or Linux)
  - Version 2.3 includes abstraction