ICFEM98, Brisbane Australia, 11 December 1998, 9am

Ubiquitous Abstraction: A New Approach To Mechanized Formal Verification

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

Formal Methods and Calculation

- Formal methods contribute useful mental frameworks, notations, and systematic methods to the design, documentation, and analysis of computer systems
- But the singular benefit from specifically formal methods is that they allow certain questions about a software or hardware design to be answered by symbolic calculation
  (e.g., formal deduction, model checking)
- And those calculations can be automated
  for speed, reliability, repeatability
- Calculations can be used for debugging (refutation) and design exploration as well as post-hoc verification
- Augments simulation, prototyping, testing
- Comparable to the way mathematics is used in other engineering disciplines

Automating Formal Calculations

- Tools are not the most important thing about formal methods
  - They are the only important thing
  - Just like any other engineering calculations, it's tools that make formal calculations feasible and useful in practice
- And the important things about tools are
  - Speed, scaling, automation, power
  - Speed, scaling, automation, power
  - Speed, scaling, automation, power
  - Oh, and soundness

Where To Apply Formal Methods Tools?

- There is little point in applying formal methods to topics that are handled adequately by traditional methods
  - E.g., refinement to code; verification of code
    (Except in regulated industries; even there, cost is critical)
- Focus on where the intractable difficulties are
  - Usually in the hardest elements of design
  - Concurrency, real time, fault tolerance

  Secondary advantage: these elements are usually small, have the best people
- And where the greatest costs are incurred
  - Errors introduced in the early lifecycle
  - Notably, omissions in requirements

So What Should Tools Do?

- Determine whether specifications of complex, often incomplete, designs have certain desired properties
  - Properties often amount to less than full correctness
- Can look at this from two sides

  Refutation: try and find bugs
  - Need not be sound (finds all errors)
    or complete (finds only real errors)
  - As long as it finds enough real bugs to be cost-effective
  - Should provide diagnostic information (counterexample)

  Verification: try and show "correctness"
  - Generally more difficult than refutation
  - And less helpful when bugs are present
- Switch to verification when refutation runs out of steam

Mechanizing Refutation: Model Checking

- If design has a finite state space, can often check properties by model checking
  - Check whether design is a Kripke model of property expressed as a temporal logic formula

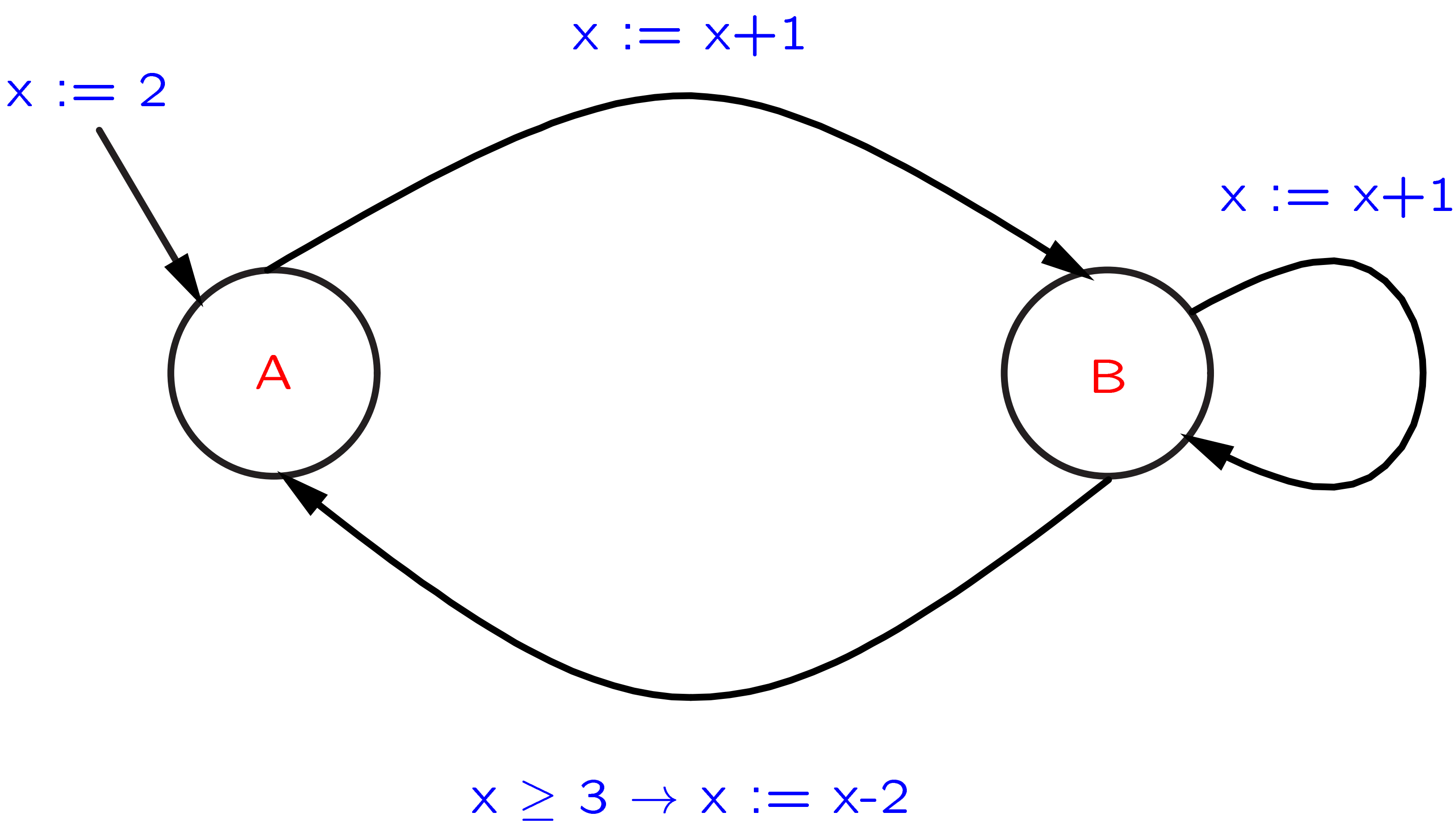
  Name often used for all related methods
- Complexity is linear in number of states
  - But that grows as product of size of data structures, and is exponential in number of interacting components
- Hence, must construct abstracted or downscaled models
  - Downscaling is aggressive (unsound) abstraction
- Experience is that you learn more by examining all possibilities of downscaled model than by probing some of the possibilities of the full thing (as by simulation or testing)

Mechanizing Formal Verification

- The tools are generally based on interactive theorem proving
  - With substantial automation
    - ⋆ Decision procedures, rewriting, heuristics, libraries
- Guiding the interaction requires skill, but
  - In domains with decision procedures or good libraries
  - And specifications are functional
  - It is often no harder than hand proof (of comparable detail)
- But for concurrent and distributed systems
  - Where specifications are transition relations
  - It is very hard indeed
    - ⋆ Not due to lack of theorem proving power
    - ⋆ But to the difficulty of inventing strong invariants

A Trivial Example

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



Attempted Proof

- We'll use the induction scheme:

```
(∀ s: init(s) ⊃ p(s))
   ∧ (∀ pre, post: p(pre) ∧ tr(pre, post) ⊃ p(post))
      ⊃ invariant(p)(init, tr)
```

  Whose own proof in PVS is

```
(SKOSIMP) (EXPAND "invariant") (INDUCT-AND-SIMPLIFY "j")
```

- The proof steps

```
(USE "ind[state]") (GROUND) ("1" (GRIND)) ("2" (GRIND))
```

  Yield

```
[-1]    pc(pre!1) = A
[-2]    pc(post!1) = B
[-3]    x(pre!1) = 0
  |-------
[1]     x(pre!1) + 1 ≥ 2
```

- Need to strengthen invariant: $x \geq 1$ when control at A

And In General?

- Can extract terms that need to be added (conjoined) to the invariant by examining these failed subgoals

  (Similar ideas for loop invariants go back 20 years)

- Larger example: verification of Bounded Retransmission Protocol (BRP)
  - Required 57 strengthenings
- Effort required generally defeats all but the most determined
  - The case explosion problem
  - Everything is possible but nothing is easy
- There is much work on methodologies for deriving suitable invariants systematically (for given classes of problems)
- But we're looking for general methods. . .

Another Direction

- Model checking avoids all this hassle
  (by calculating a fixpoint)
- Substitutes calculation for proof
- But only works for finite-state systems
- So let's create a finite-state abstraction (i.e., approximation)
- And model-check that
- Will also need to prove that the abstraction is property-preserving

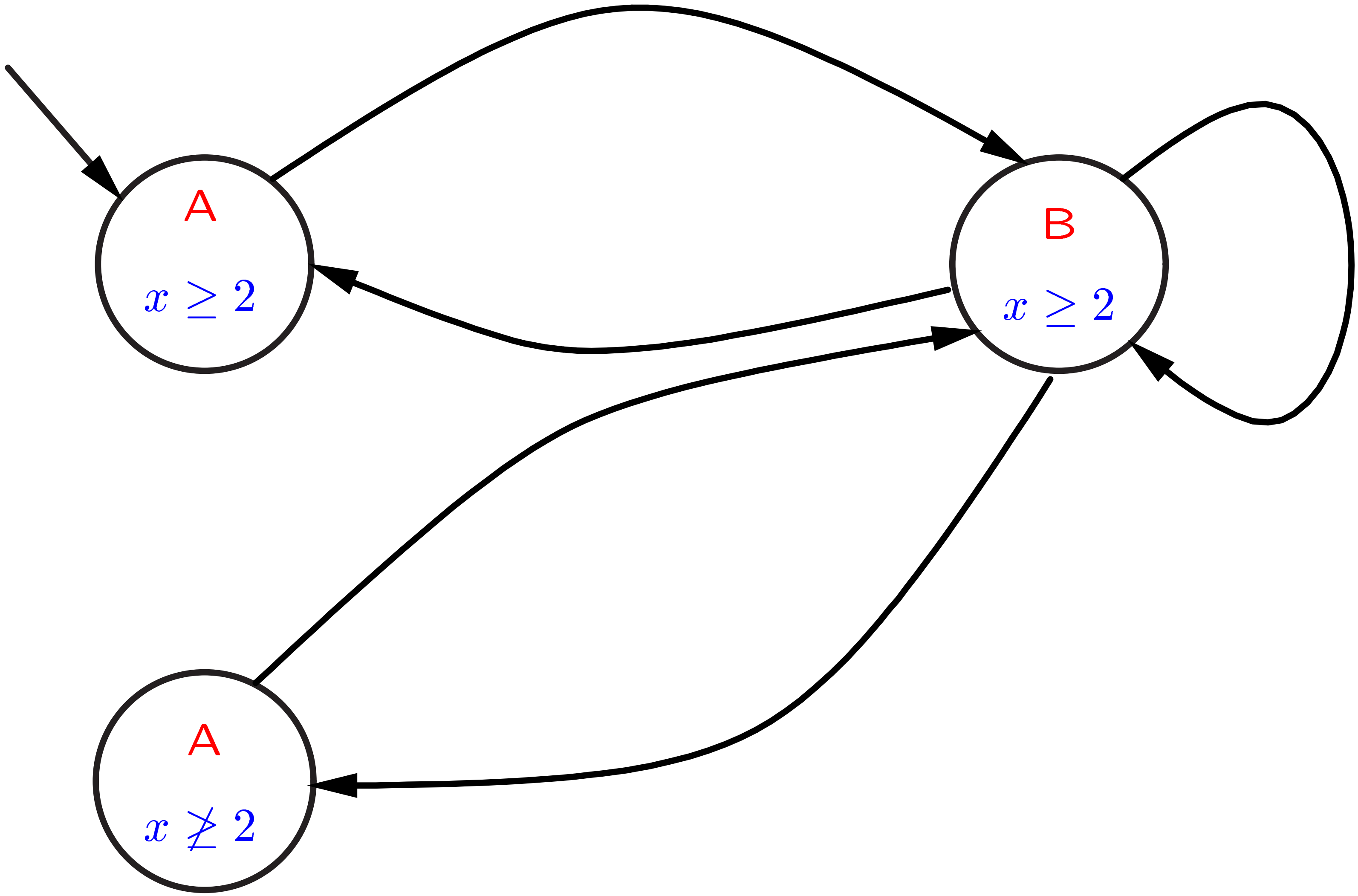
Verification Via Property-Preserving Abstraction

- In general, we need a (finite) abstract state space with transition relation $\mathtt{tr}_a$
- And an abstraction function abs from the concrete state space to the abstract one
- And a predicate $\mathtt{p}_a$ on the abstract states
- Such that
  1. $\mathtt{init}_c(\mathtt{cs}) \supset \mathtt{init}_a(\mathtt{abs}(\mathtt{cs}))$
  2. $\mathtt{tr}_c(\mathtt{pre}_c, \mathtt{post}_c) \supset \mathtt{tr}_a(\mathtt{abs}(\mathtt{pre}_c), \mathtt{abs}(\mathtt{post}_c))$
  3. $\mathtt{p}_a(\mathtt{abs}(\mathtt{cs})) \supset \mathtt{p}_c(\mathtt{cs})$
- Then
  - $\mathtt{invariant}(\mathtt{p}_a)(\mathtt{init}_a, \mathtt{tr}_a) \supset \mathtt{invariant}(\mathtt{p}_c)(\mathtt{init}_c, \mathtt{tr}_c)$
- And the antecedent can be proved by model checking

The Example: Boolean Abstraction

- Often convenient to choose an abstract state space consisting of
  - The control locations of the concrete system, plus
  - Some boolean state variables that correspond to predicates in the concrete system
- This is Boolean abstraction
- For the example, we'll have one abstract Boolean state variable corresponding to the concrete state predicate $\mathtt{x} \geq 2$

An Abstract Transition Relation For The Example



Clearly, the abstract invariant is satisfied

Verification Conditions for the Example Abstraction

- All trivial except number 2: default proof strategy yields

```
[-1]    pc(post_c!1) = B
[-2]    x(pre_c!1) = 0
  |-------
[1]     x(pre_c!1) + 1 ≥ 2
```

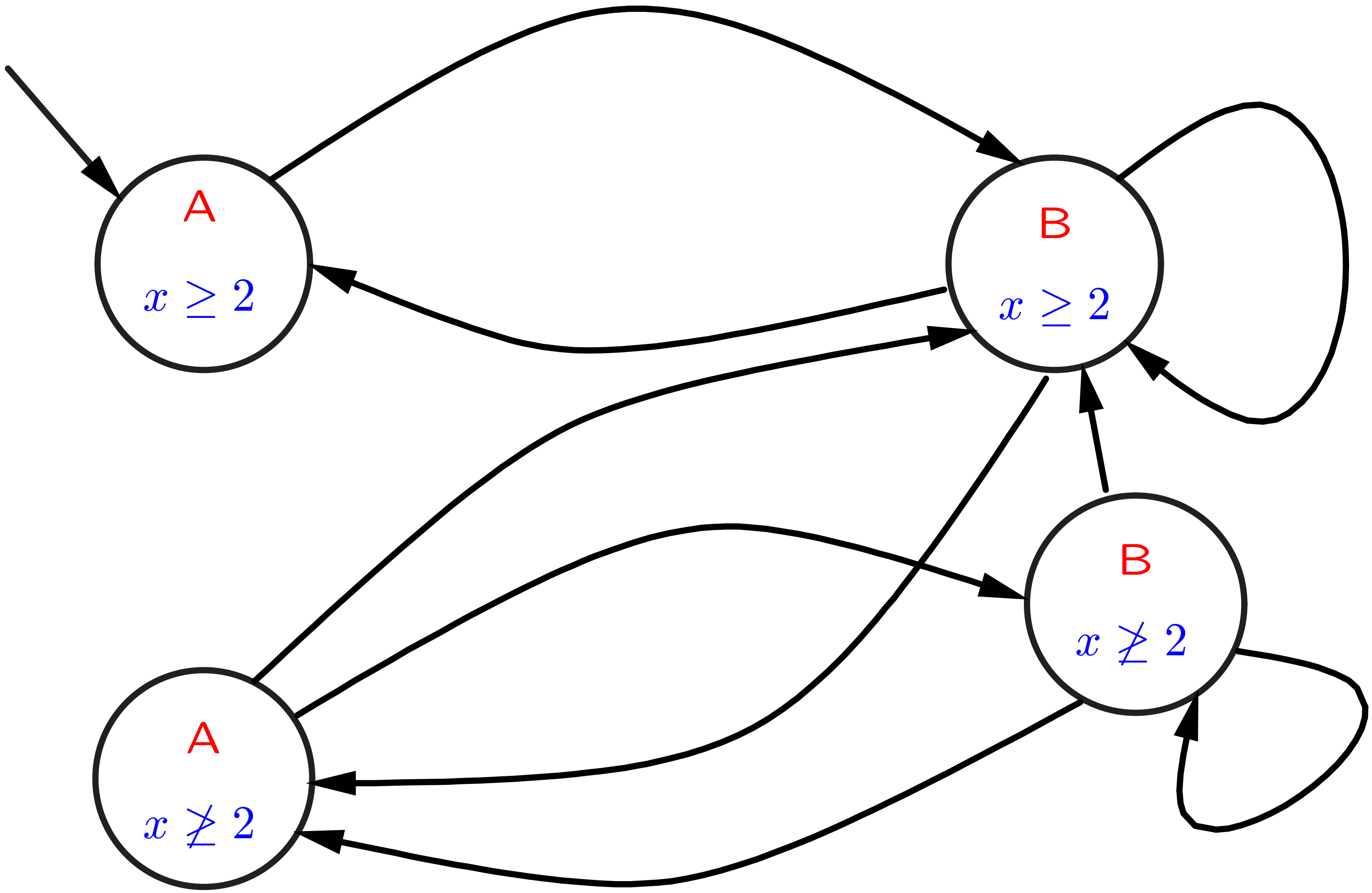
- Essentially the same as in the basic invariance proof
- Requires an invariant!
- Larger example: verification of Bounded Retransmission Protocol (BRP) by abstraction
  - Required 45 invariants

So What's To Be Done?

Calculate the abstract system (given the abstraction function) rather than "invent and verify"

- Saves manual effort of construction
- Abstract system is an abstraction (by construction)
- But may be too coarse to satisfy desired abstract invariant

Calculated Abstract Transition Relation For The Example



Abstract invariant is not satisfied

Diagnosing The Problem

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

    2 3 1 2
- We see that it is important to know $\mathtt{x} \geq \mathtt{1}$ at $\mathtt{A}$
- So add another abstract state variable corresponding to $\mathtt{x} \geq \mathtt{1}$ and repeat
- This does it!

Making It Practical

- (At least) two ways of calculating the abstracted system
  - Start with universal transition relation; then for each arc
    - ⋆ Generate the verification condition (VC) that allows it to be removed
    - ⋆ Leave it in if cannot prove the VC

    This approach preserves structure
  - Develop the relation by a forward reachability analysis
    - ⋆ At each point generate the VCs that lead to successor states with given predicate true resp. false

    This approach usually has fewer states
- There are clever techniques for assuming the invariant you want to prove while constructing the abstraction
- And for refining an abstraction using counterexamples

Making It Practical (ctd.)

- Generate as many invariants as possible by static analysis and throw those into the proofs/calculations
  - Can easily deduce $x \geq 1$ in the example
- Use heuristics to generate plausible initial abstractions
  - Boolean abstraction on (atomic) guard predicates
- Build tools for concretizing counterexample traces and checking them against the concrete system
  - To help distinguish between
    - ⋆ An excessively coarse abstraction
    - ⋆ A bug in the concrete system
- Can verify Bounded Retransmission Protocol (BRP) automatically using these techniques
- Takes a couple of hours to calculate the abstracted system

Doing It Ubiquitously

- Model checkers usually calculate the reachable stateset (and then throw it away)
  - Which is the strongest invariant
- 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 the theorem prover can manipulate
- Use simple abstractions to develop invariants that enable construction of finer ones
  - E.g., Boolean abstraction on $x \geq 1$ in the example provides the invariant that enables construction of the fine abstraction on $x \geq 2$

Iterated Abstractions

- Can also use different abstraction techniques

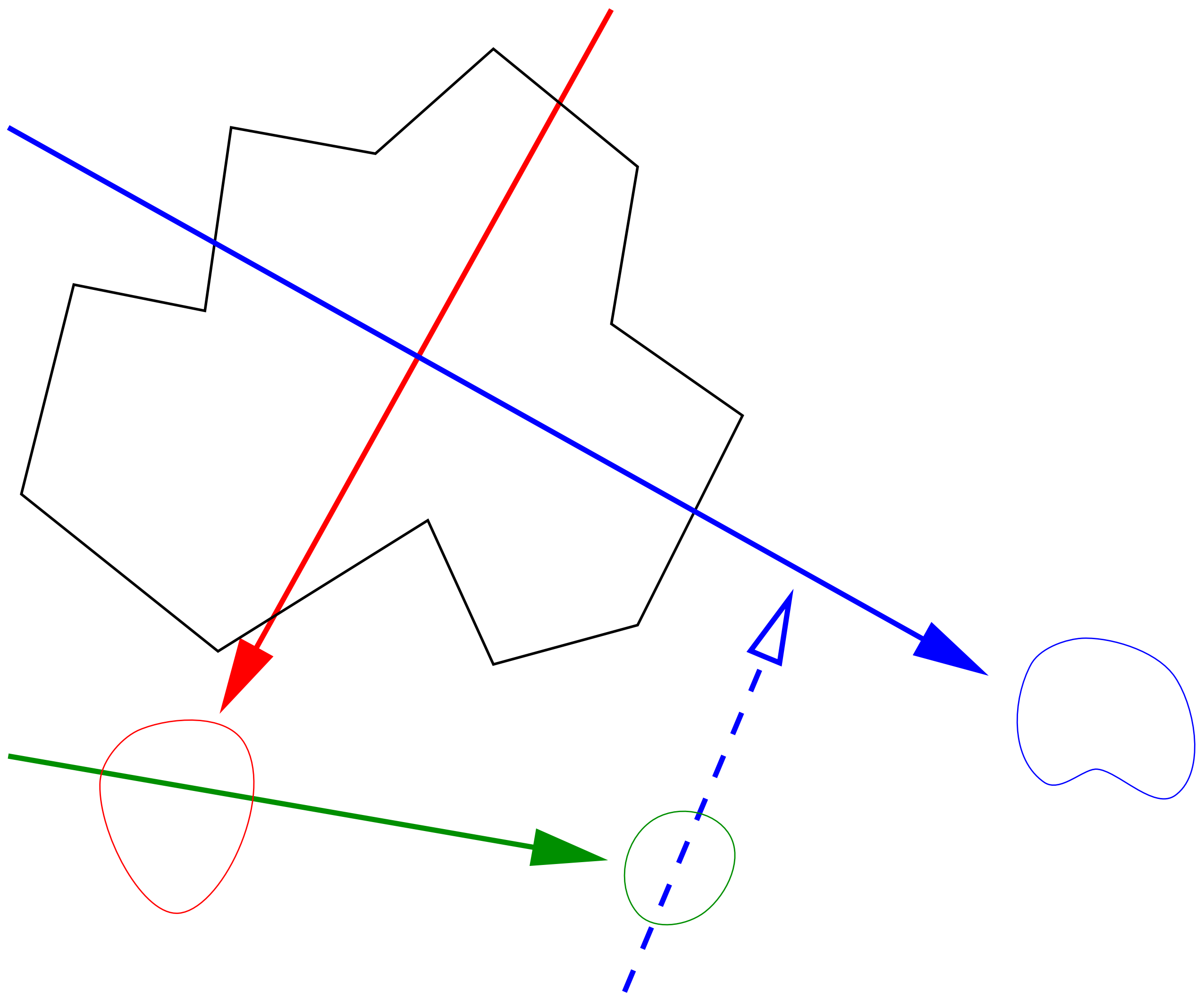
  Semantic: what we've seen so far

  Syntactic: slicing, abstract interpretation

  - Slicing extracts salient part of a complex system
  - Abstract Interpretation provides basis for strong static analyses (cf. dimensional analysis)

- And can iterate them
  - E.g., slice, abstract interpretation, then semantic abstraction

Iterated Abstraction, Concretization, Invariant Generation



Integrating Abstraction With Theorem Proving

- So far, we've used abstraction only on the top-level goal
- Can also apply it in the context of the subgoals generated by a theorem prover (e.g., in an inductive proof)
- Are then working on simpler problems
- And predicates in subgoal provide good clues to suitable Boolean abstractions

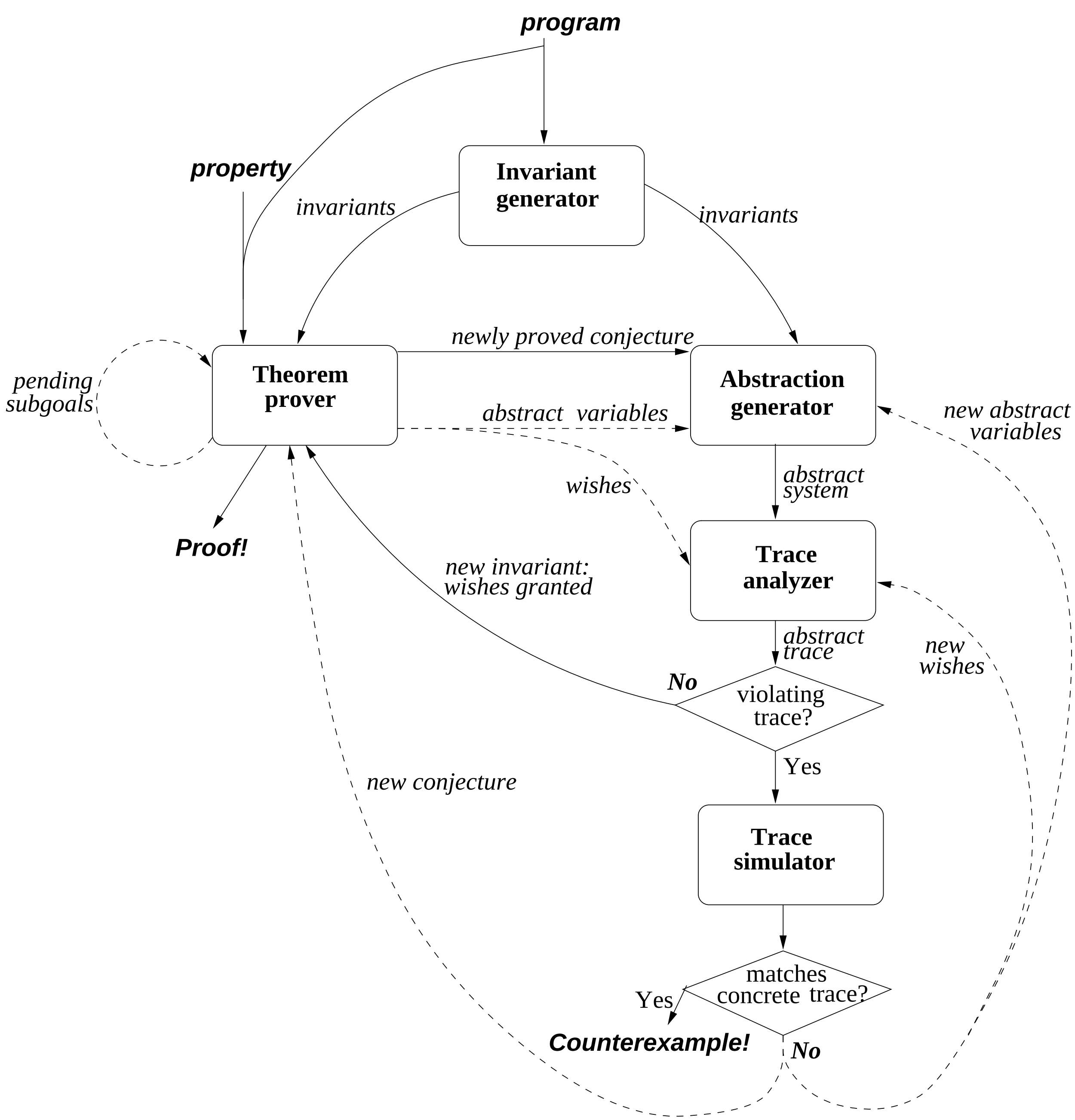
Integrating Abstraction With Theorem Proving (ctd.)

- In the example, the subgoal

```
[-1]    pc(pre!1) = A
[-2]    pc(post!1) = B
[-3]    x(pre!1) = 0
  |-------
[1]     x(pre!1) + 1 ≥ 2
```

- Suggests abstracting on $\mathtt{x} = 0$
  (which is equivalent to $\mathtt{x} \not\geq 1$ since x is a natural number)
- And model checking then shows this state to be unreachable
- Method is provably stronger than guard abstraction and precondition strengthening

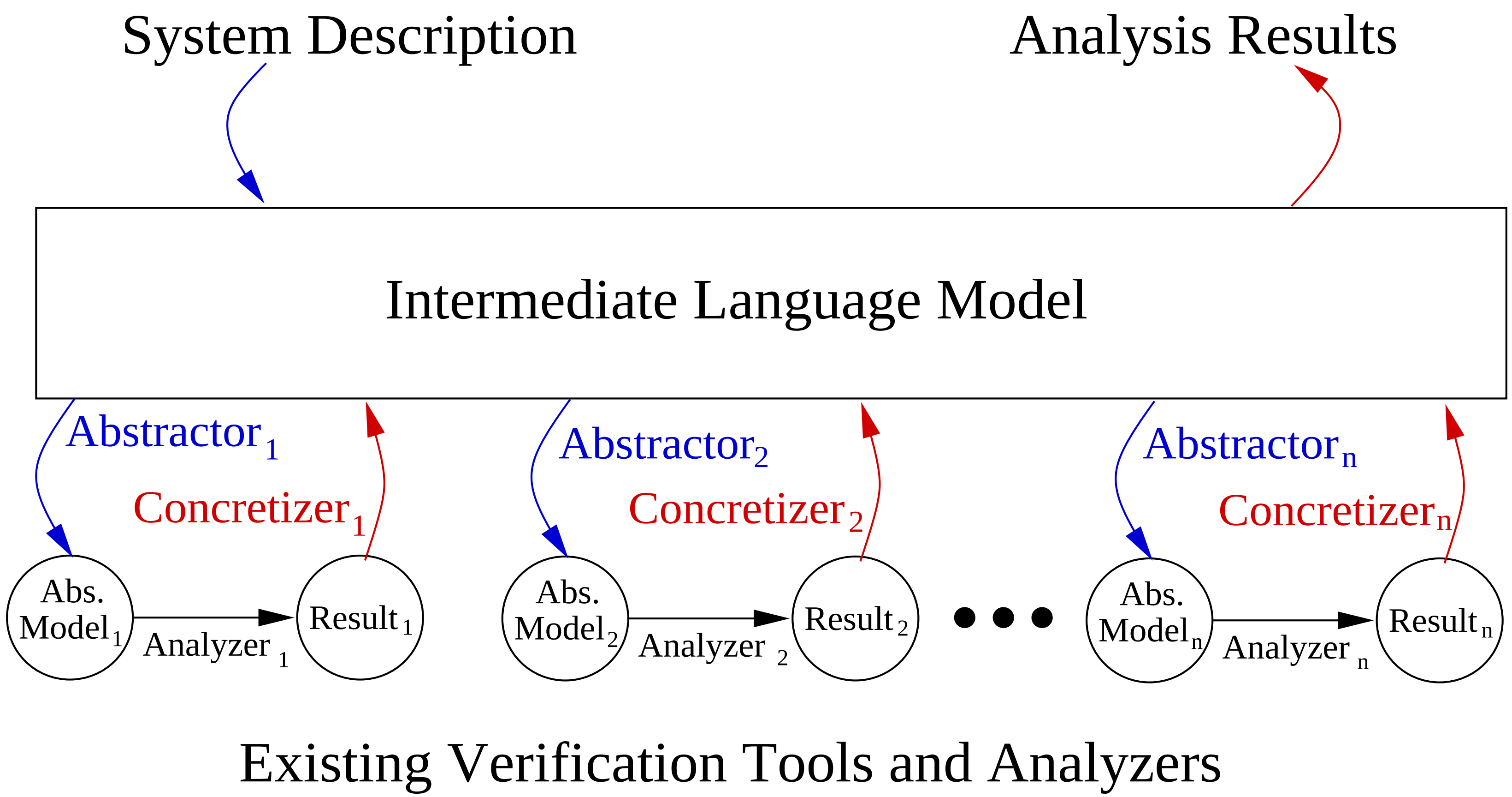
Integrating Abstraction With Theorem Proving (ctd. 2)



The “New” Approach

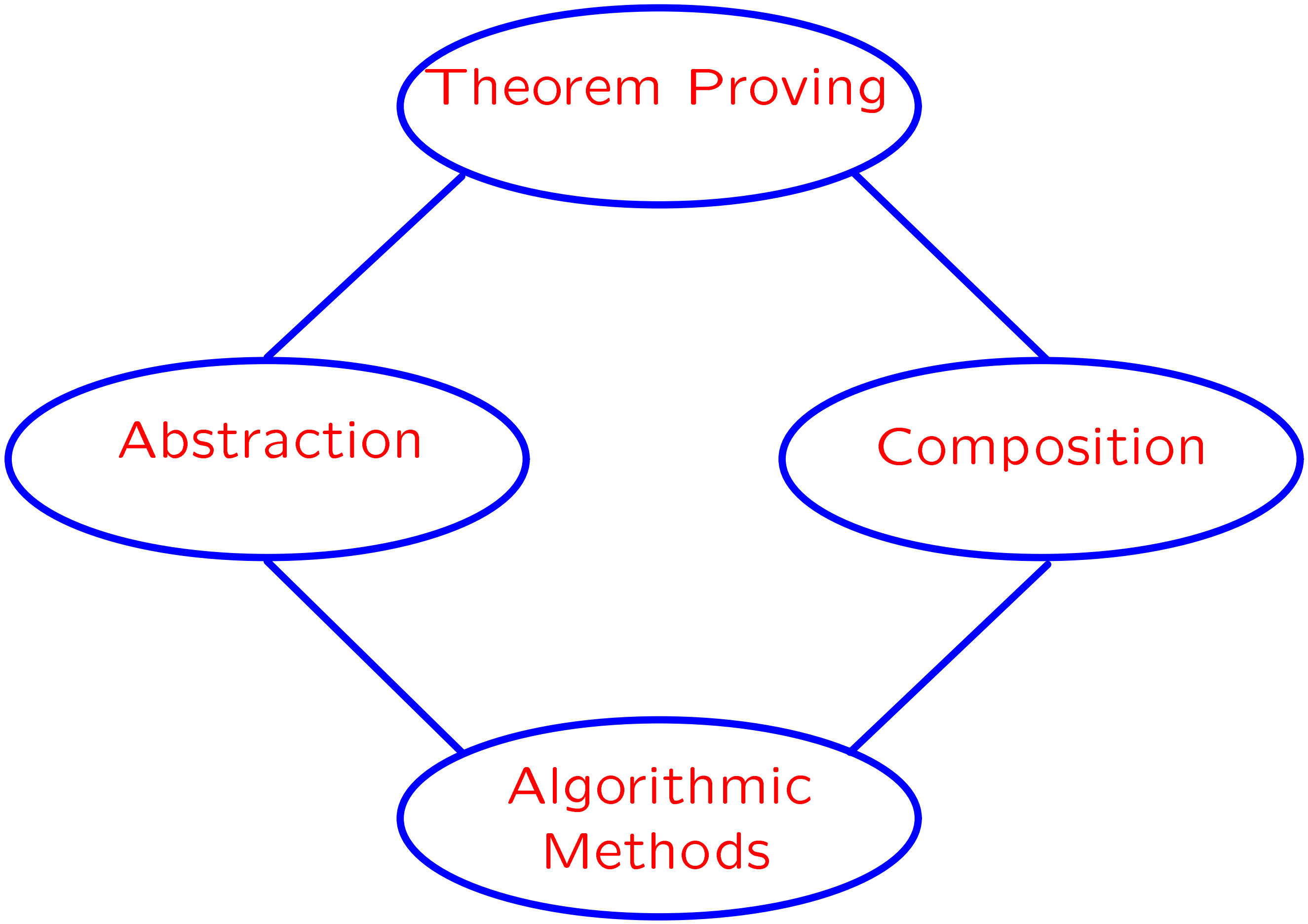
- Instead of trying to build ever more powerful tools
- Try to make the problems easier
  - Cut them down to a size the existing tools can handle
- By making ubiquitous use of automated abstraction
  - That is, construction of simpler descriptions that ignore/approximate aspects of the original
- Within a framework that allows multiple tools to cooperate
  - Generate models appropriate to different analyses and different tools from a single description
- Cooperation requires tools to exchange symbolic values, not just true/false verification outcomes
- The idea behind SAL: a (Symbolic Analysis Laboratory)

The SAL Idea



System Description
Analysis Results
Intermediate Language Model
Abstractor 1
Concretizer 1
Abstractor2
Concretizer 2
Abstractor n
Concretizer n
Abs.
Model1
Result 1
Analyzer 1
Abs.
Model2
Result 2
Analyzer 2
Abs.
Modeln
Result n
Analyzer n
Existing Verification Tools and Analyzers

The General View



Abstraction and composition are the bridges between deductive and algorithmic methods of verification

Related Work

- Research in model checking has long focused on abstraction
  - More recently on iterated combinations justified by theorem proving
  - E.g., "Minimalist Proof Assistants" by Ken McMillan
    - ⋆ FMCAD talk (on his web page at `http://www-cad.eecs.berkeley.edu/~kenmcmil/`)
    - ⋆ Implemented in SMV
    - ⋆ Used for Tomasulo, SGI cache coherence
- Much recent focus on logics with very powerful automation
  - Propositional calculus (Stålmarck's method)
  - With uninterpreted functions (Herbrand automata)
  - WS1S (Mona)

  And methods for reducing general problems to those efficient cases

Credits

None of this work is mine; it is due to my colleagues

- Klaus Havelund: BRP example
- Hassen Saïdi: The Invariant Checker
- Saddek Bensalem, Yassine Lakhnech, Sam Owre: InVeSt
- Vlad Rusu and Eli Singerman: Mini-SAL experiments
- Shankar: SAL
  Being developed with David Dill (Stanford)
  And Tom Henzinger (Berkeley)

To Learn More

- Browse general papers and technical reports at http://www.csl.sri.com/fm.html
  - ~owre/cav98.html and ~owre/cav98-tool.html for InVeSt
  - ~rusu/tacas99.html for mini-SAL experiments
  - ~saidi/Invariant-Checker/index.html for the Invariant Checker
- Information about our verification system, PVS, and the system itself are available from http://pvs.csl.sri.com
  - Freely available under license to SRI
  - Allegro Lisp for Solaris, or Linux
  - Need 64M memory, 100M swap space, 200 MHz or better