MOVEP'2k: Nantes, France 23 June 2000

Theorem Proving for Verification

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

Plan

1. Theorem proving in general
   - What is it? And what do I know?
   - Techniques
   - Pros and cons
   - Examples (with PVS)

   Break

2. Application to parallel processes
   - Modeling
   - Inductive invariants
   - Comparison with model checking
   - Integration with model checking, automated abstraction
   - Example: Bakery

Theorem Proving for Verification: What Is It?

- Specify
  - The system (at some suitable level of abstraction)
  - A required property
  - The environment (possibly)
  - Other assumptions
  - Necessary background theories

  as formulas in a single logic

- And prove that

  $$background + assumptions + (environment +)\ system \vdash requirement$$

- Variation:
  - Prove that implementation is refinement of specification (i.e., commutative diagram, or theory interpretation)

Compared to Model Checking

- Specify
  - The system as a finite transition system
  - A required property as a formula in a temporal logic
- Establish that the system is a model of the formula

  $$\mathit{background} + \mathit{assumptions} + (\mathit{environment} +)\ \mathit{system} \models \mathit{requirement}$$

  Accomplished by a search over the state space
- Usually no background theories
  (sometimes limited arithmetic)
- Usually, the only assumptions concern interpretation of liveness properties (branching vs. linear time, fairness)
  - Built in to choice of logic and selection of operators

Theorem Proving for Verification

- Our goal is to build mathematical models of systems and to calculate properties of the models
- Just like in other branches of engineering (e.g., CFD)
- Logic is the applied math of Computer Science
- So the models are formal specifications in some logical system
- And theorem proving (automated deduction) is used to do the calculations
    - A different motivation for "proof" than in mathematics
    - Must support an effective specification language
    - But must be highly automated

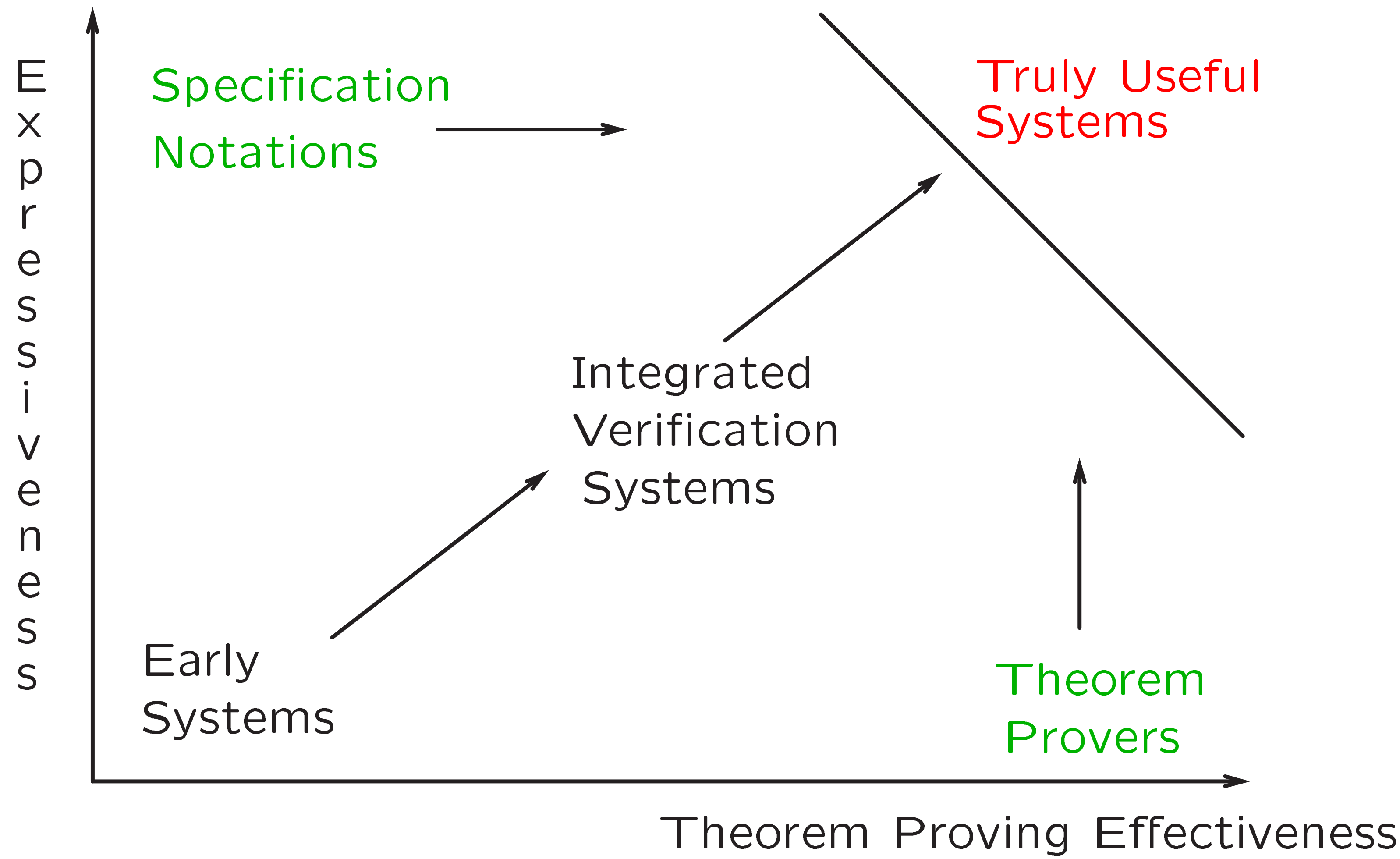
Blind Alleys—Fundamentalism

- "Proof" means emulating pencil-and-paper arguments
  - Heavily interactive; excludes real automation
  - Invites unsuitable foundations (e.g., set theory)
- Reductionist approaches to soundness
  - Reduce everything to elementary rules of logic
  - Excludes decision procedures and other powerful automation
- Complete correctness
  - Prove everything from top to bottom
    - ⋆ Or impose a constructive approach on specification and proof
  - Doesn't focus on where the benefits are

Blind Alleys—Language First, and Prover First

- You cannot design a specification language in isolation, and then hope to add effective mechanization
  - Mechanization is sensitive to choice of basic framework, and to issues such as partial functions
- And you cannot build a theorem prover in isolation and hope to persuade very many people that its logic is a specification language
  - Also this approach usually lacks interpreted theories (arithmetic, datatypes etc.)
- The formal method, its specification language, and its tools have to be designed together
  - Tradeoffs and compromises are necessary

The Three Traditions



Specification Languages and Theorem Proving

- A specification language must provide some of the features and conveniences of a programming language (e.g., declarations, packaging in modules, parameterization)
- And computer-science data types (e.g., numbers, tuples, records, functions, sets, lists, trees)
- And should use a reasonably familiar syntax
- But it must also support deduction—i.e., it must be a logic
- We want that logic to be expressive, and we also want deduction to be automated very effectively
- Conflict: expressiveness and effectively automated deduction are usually inversely related
- Reconciling these disparate requirements is a fundamental challenge in theorem proving for formal methods

Language Choices in PVS

- Framework is higher-order logic
  - Typing is intrinsic
  - Functions as arguments are very useful (e.g., to parameterize a sorting function wrt. the sort criterion)
  - And so is higher-order quantification
    - E.g., to specify sorting criterion is well-founded
    - Less need for extra-logical machinery (e.g., induction)
  - Provides an adequate treatment of sets
  - Total functions are primitive (cf. first-order + set theory)
  - Can embed other logics
- With dependent typing, and predicate subtyping
  - Uses theorem proving in typechecking
- And recursive abstract datatypes (e.g., lists, trees, ordinals)
- No built-in model of computation

Automated Theorem Proving is Hard

- Many necessary theories or combination of theories are undecidable in general (e.g., quantified linear arithmetic with uninterpreted function symbols)
- So cannot expect complete automation: will need some heuristics (which may fail), and some interactive guidance
- The field of automated deduction provides important techniques, but has a different focus than verification
  - Its applications(?) are math and AI
  - Its goal is unassisted automation (in semi-decidable systems—e.g., pure first-order logic)
  - It automates raw logics, not specification languages
  - It focuses on proving true theorems

Basic Mechanisms for Theorem Proving

- Decision procedures for important theories
  - And their combination
  - May need "proof objects" for certification
- Quantifier instantiation
  - Can decide some combinations of theories for unquantified case only, so provide quantifier instantiation separately
- Rewriting
  - Does instantiation, lemma selection, definition expansion
- Everything else

Decision Procedures

- Propositional calculus

  $(a \wedge b) \vee \neg a = a \supset b$

- Equality

  $x = y \wedge f(f(f(x))) = f(x) \supset f(f(f(f(f(y))))) = f(x)$

- Linear Arithmetic

  $x \leq y \wedge x \leq 1 - y \wedge 2 \times x \geq 1 \supset F(4 \times x) = F(2)$

- Function Updates

  f **with** $[(x) := y](z) \stackrel{\mathrm{def}}{=}$ **if** $z = x$ **then** y **else** $f(z)$

- Datatypes

  $car(cons(a, b)) = a$

- Combination

  $x \leq car(a) \supset car(a) \geq car(cons(x, b))$

Decision Procedures: Algorithms

- Propositional logic: BDDs, Davis-Putnam, Stålmarck
- Pure equality: congruence closure
- Linear arithmetic: integer/linear programming (loop residue, SUP-INF, Fourier-Motzkin)
  - Heuristic extensions to nonlinear cases are important
- Datatypes, algebra: rewriting
- Combination: Shostak, Nelson-Oppen
- Others: bitvectors, WS1S, finite μ-calculus

Rewriting

- Provides decision procedures if rules are finite terminating and confluent
- More often used heuristically
- E.g., only if antecedent of conditional rule can be discharged

$$
\begin{aligned}
n > 0 \supset n! &= n \times (n-1)! \\
0! &= 1
\end{aligned}
$$

- Or top-level if-then-else in definition expansion

```
factorial(n) = IF n=0 THEN 0 ELSE n * factorial(n-1)
```

- Must be integrated with decision procedures
- Can provide an evaluation capability for test-cases

Other Basic Mechanisms

- Heuristics
  - Quantifier Elimination: unification, search
  - Induction: Higher-order matching, Boyer-Moore, rippling
- Combination
  - Everything needs to work together
- Interactive guidance
  - With a decent interface
  - And a language for defining “strategies”
- In the large
  - Use lemmas before they are proved
  - Introduce/modify lemmas on the fly
  - Need global checks for completeness
- “Proof objects” for certification

Theorem Proving: Compared with Model-Checking

- The system is not restricted to finite-state (nor to a transition system)
- Can leave details uninterpreted
- The required property is not restricted to invariance or eventuality properties that can be expressed in a temporal logic (or other restricted notation)
- The assumptions are not restricted to fairness constraints, but can be arbitrary theories
- And arbitrary mathematical theories can be supplied as background
- Can do verification, not just refutation
- The price of this generality is (usually) less automation
    - And less feedback (no counterexamples) on failure

Theorem Proving: The Challenge of Human Guidance

- Decision procedures can be very effective on limited domains (e.g., Boolean equivalence checking in CAD tools)
- But general problems usually require heuristics or interactive human guidance
  - Difficult to diagnose failure with heuristics
  - And often difficult to do anything about it
- Interactive methods can usually prove any true property given enough time, skill and patience
  - But often require too much time, skill, patience
  - And proof may be fragile in face of design changes
- Guidance is usually in terms of the prover, not the problem
  - Unacceptable to most engineers

Verifying Parallel Processes

- Sequential programs can be modeled by recursive functions, or rewrite rules
  - Or can embed, for example, a Hoare calculus
- For parallel processes, can use transition relations
  - Or embed, for example, a process algebra
- I'll focus on transition relations, and on safety properties (i.e., invariants)

Deductive Verification of Invariants

- Systems are modeled as nondeterministic automata over possibly infinite sets of states S; the transition relation is T
- There is a set of initial states I
- The reachable states R are those you can reach by iterating T, starting from I
- A set of states G is invariant (i.e., characterizes a safety property) if it includes all the reachable states
- Our problem: how to verify that a given G is invariant
- With finite-state systems, you can just enumerate R and check that its members are in G (model checkers do this)
- Using deductive methods, instead show that G is inductive

Inductive State Sets

- A set of states G is inductive if it includes the initial states and is closed on the transition relation
  - $s \in I \supset s \in G$
  - $s \in G \wedge T(s, t) \supset t \in G$

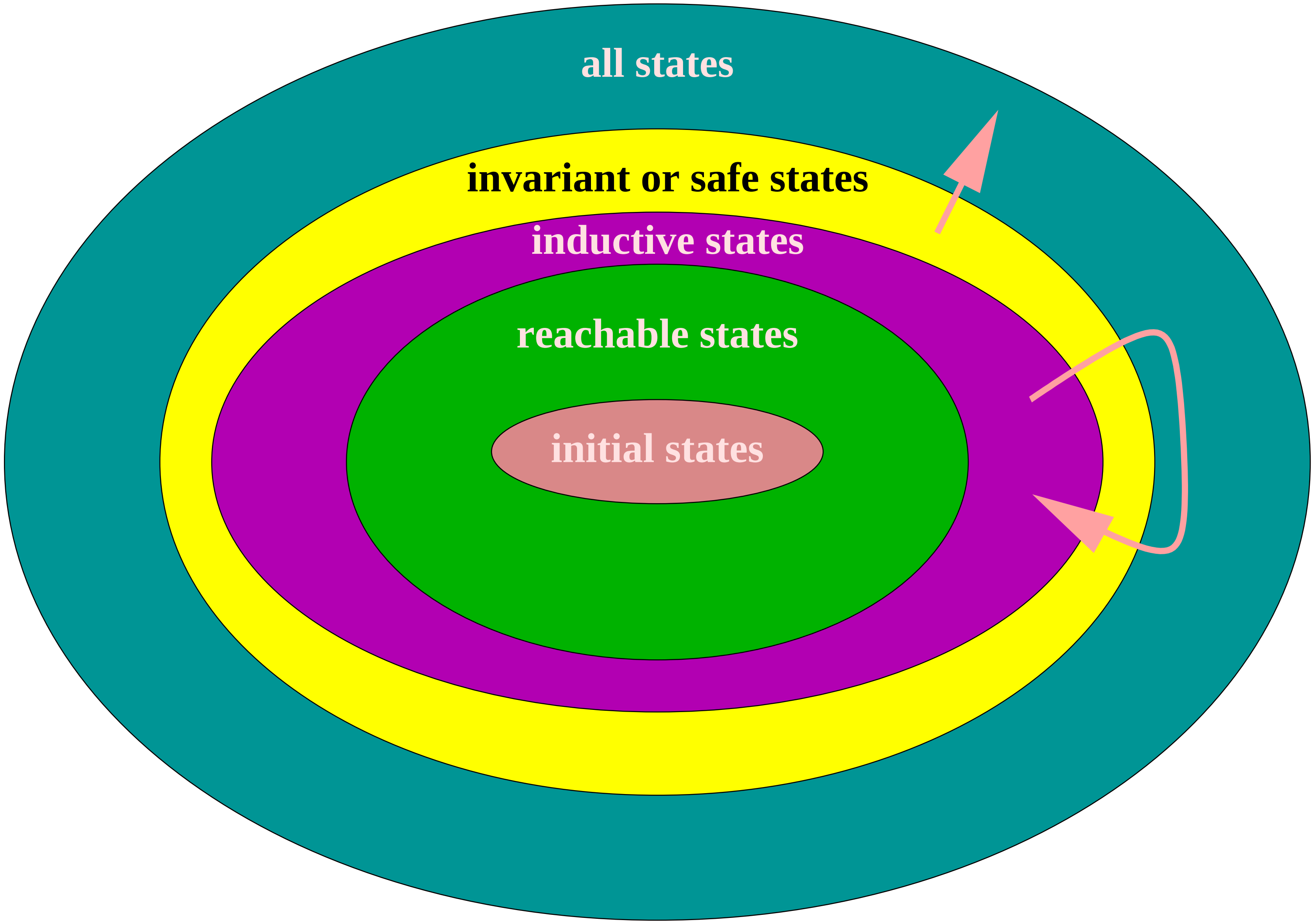
  The reachable states are the smallest set that is inductive

- The problem in verifying invariants is that they are seldom inductive; the second condition is violated

- The solution is to replace G by some smaller G^1

- And then iterate with smaller $G^2, \ldots, G^i$ until you eventually discover one that is inductive.

In Pictures


all states
invariant or safe states
inductive states
reachable states
initial states

Getting an Inductive Property

- Shifting terminology from sets of states to predicates. . .
- The usual way to form G^i is by conjoining additional predicates:

$$G^i_\wedge = G \wedge G_1 \wedge \cdots \wedge G_i$$

- This process can be made systematic (G_{i+1} is discovered by inspecting a failed proof for inductiveness of $G^i_\wedge$)
- And can be partially automated, but is always tedious

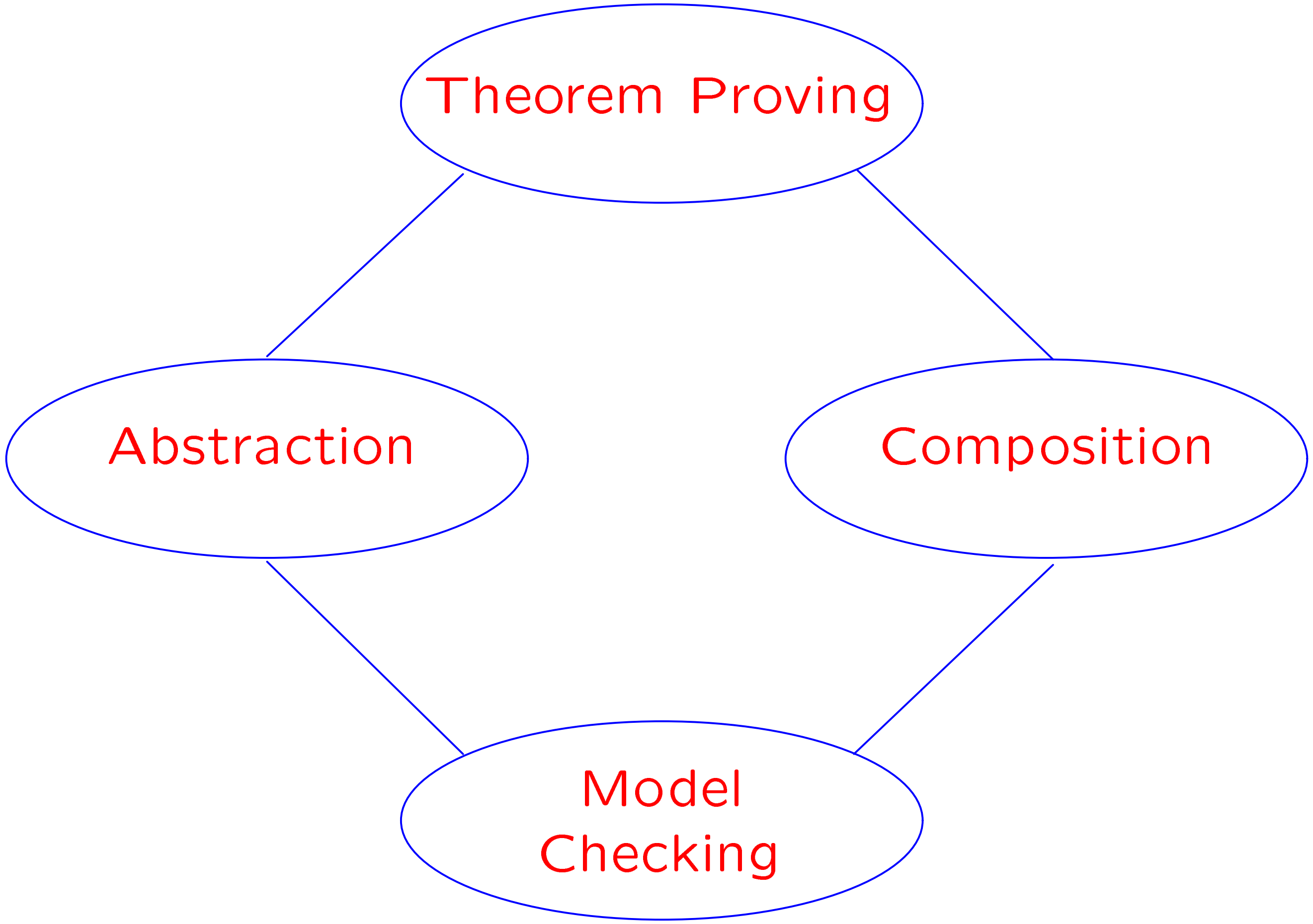
Inductive Invariants: Experience

- Havelund and Shankar formally verified a bounded-retransmission communications protocol (BRP) from Philips
- First "downscaled" (aggressively simplified) it to finite-state
  - Restricted size of data structures to explicit small values
  - Adequate for refutation, not for verification
- Checked safety properties by state exploration using Murϕ
- Then started verification of full protocol in PVS
- Desired invariant needed strengthening by additional conjuncts
  - Plausible conjuncts found by examining failed proof state
  - Checked with Murϕ
  - Added to invariant (without quitting proof)
  - Iterated the above—57 times!

Integration With Model Checking

- Use a model checker as a component of a theorem prover
- Need to be able to describe a temporal logic, its models, and satisfaction relation in the logic of the theorem prover
- Example: CTL, transition relations, and the satisfaction relation between them can be specified in PVS in terms of Park's μ-calculus (QBL with least and greatest fixed points)
  - $\mathbf{EX}(N, f)(u) : \texttt{bool} = \exists v : (f(v) \wedge N(u, v))$
  - $\mathbf{EG}(N, f)(u) : \texttt{bool} = \nu(\lambda Q : (f \wedge \mathbf{EX}(N, Q)))$
- A BDD-based validity checker for the Boolean fragment of Park's μ-calculus is available as a backend to PVS
- Combination provides CTL model checking on finite-state systems within PVS

Abstraction Is A Bridge From Deductive To Algorithmic Methods



Manually Constructed Abstractions

When model checking is integrated with theorem proving, can

- Use induction to prove n-process properties, with the inductive step discharged by model checking
- Use theorem proving to justify an abstraction to finite state, then model check the finite state abstraction

  Initiality simulation: $I_c(s) \supset I_a(\phi(s))$,

  Transition simulation: $T_c(s,t) \supset T_a(\phi(s), \phi(t))$,

  Safety Preservation: $P_a(\phi(s)) \supset P_c(s)$, and

  Abstract Invariant: $I_a(s) \supset \mathrm{AG}(T_a, P_a)(s)$

  The first three can be discharged by theorem proving, and the last by model checking

  - But proofs often require other invariants

Manual Abstraction: Experience

- Havelund and Shankar abstracted BRP to finite-state
  - E.g., untransmitted portion of file is chosen from the uninterpreted enumeration `NONE`, `ONE`, and `MANY`
  - Cf. downscaled version: file is 1, 2, or 3 messages long
- Model-checked the abstracted protocol using the CTL embedding
- Verified by theorem proving that abstraction preserved properties of interest
  - But needed 45 invariants to establish this

Integration: Observations

- The problems now are those of discovering invariants
- And justifying abstractions
- Idea: focus the theorem prover on the automated construction of invariants and abstractions
  - Being explored in InVeSt and SAL

Automated Abstraction

These reduced models can be constructed automatically by mechanized predicate abstraction (user suggests predicates)

- 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
    - ⋆ If a proof fails, reduced model will be more conservative, but is often still good enough
  - Available (using more efficient algorithm) in PVS

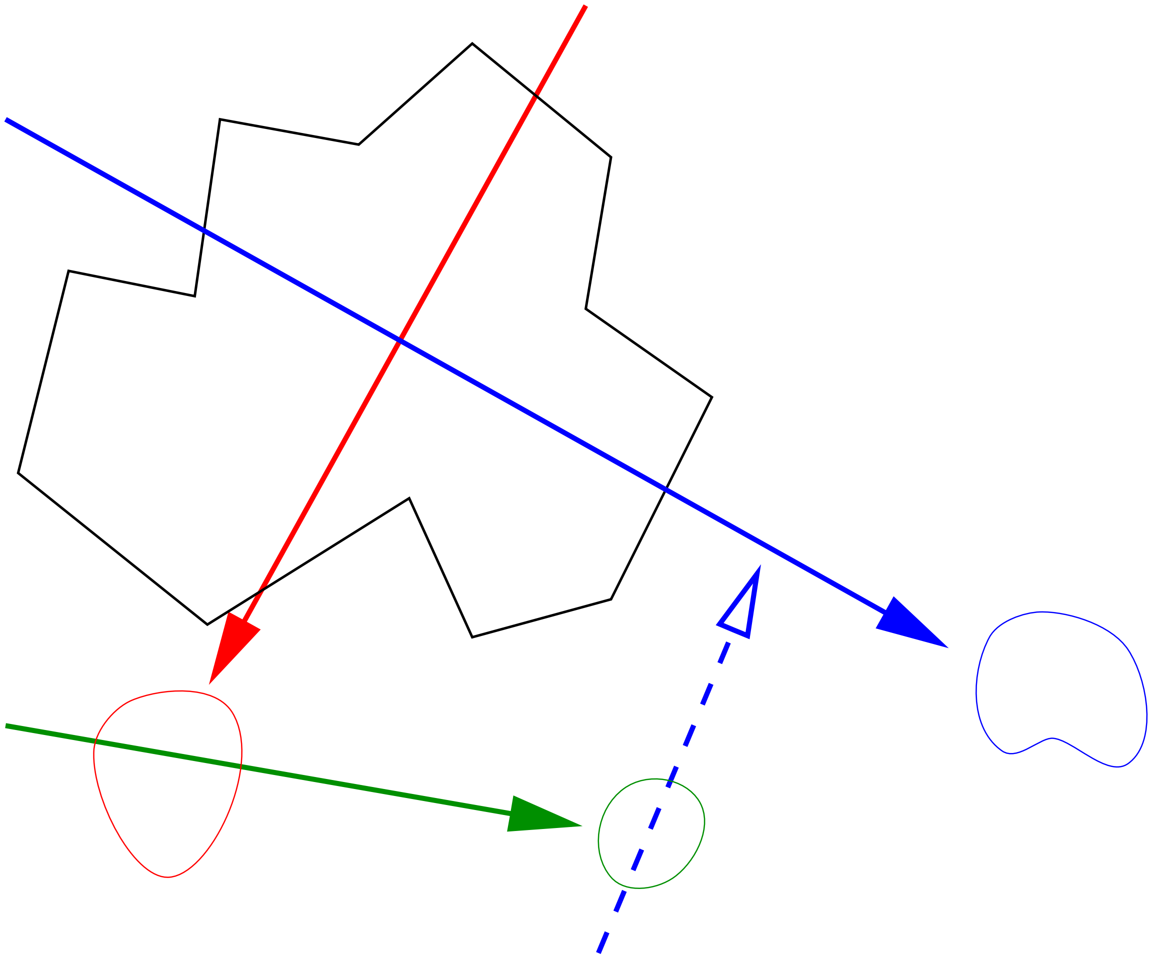
Solves BRP automatically in a minute or so

The Bridge Goes In Both Directions

- Invariants can help calculate strong abstraction
- 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
- So modify a model checker to return the reachable states as a formula that a theorem prover can manipulate
- Has been done (by Sergey Berezin) for CMU SMV and is used in InVeSt and SAL

Integrated, Iterated Analysis



Even More Integrated, Iterated Analysis!

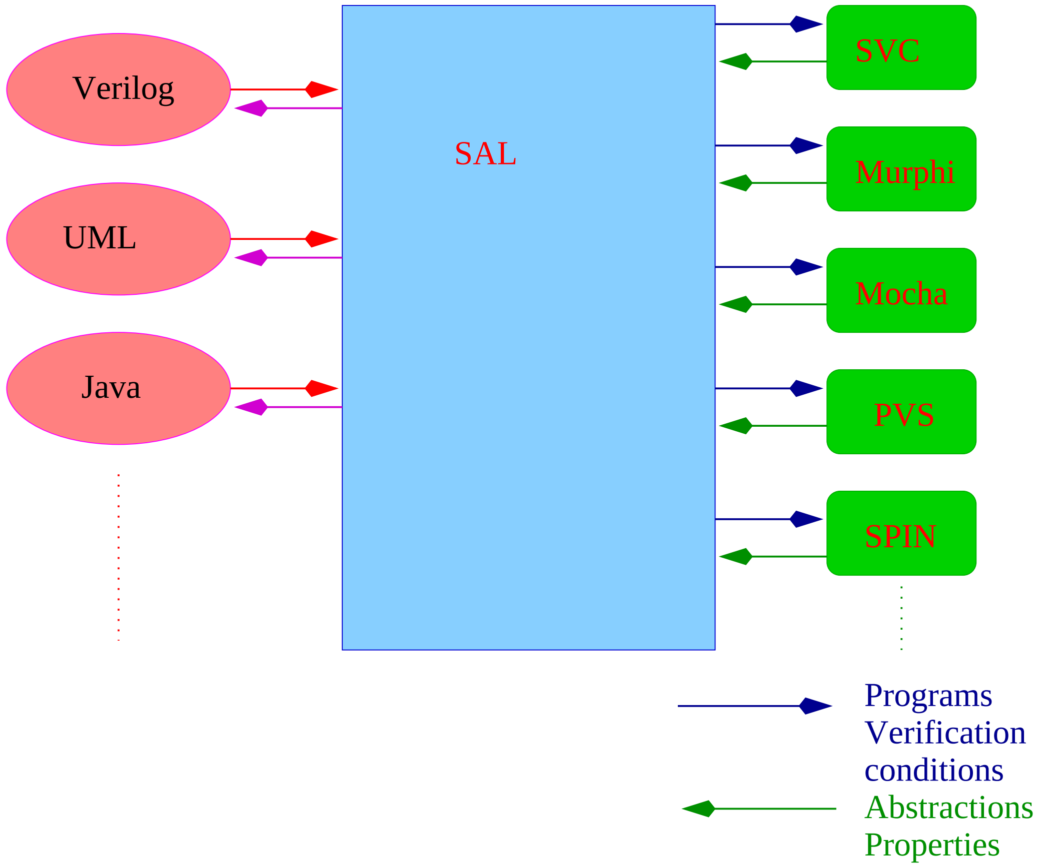
- (Approximations to) fixpoints of weakest preconditions or strongest postconditions also generate invariants and can strengthen those extracted from an abstraction
  - Mechanized by theorem proving
  - (Strongest postconditions are equivalent to symbolic simulation, which is independently useful)
- Counterexamples from failed model check help distinguish bugs from weak abstractions, and also help refine the abstraction
  - Suggest additional properties (invariants) that will help the theorem prover construct a tighter model
  - Suggest additional predicates on which to abstract

Truly Integrated, Iterated Analysis!

- Recast the goal as one of calculating and accumulating properties about a design (symbolic analysis)
- Rather than just verifying or refuting a specific property
- Properties convey information and insight, and provide leverage to construct new abstractions
  - And hence more properties
- Requires restructuring of verification tools
  - So that many work together
  - And so that they return symbolic values and properties rather than just yes/no results of verifications
- This is what SAL is about: Symbolic Analysis Laboratory

SAL Architecture

The SAL Language serves as a hub



Not all of these boxes are populated yet

Verification and Refutation

- Formal analysis is hard to automate (undecidable in general, decidable problems have huge complexity, or suffer from state explosion)
- Must often trade off accuracy or completeness of the model for tractability and automation of calculation
  - E.g., downscale aggressively to allow model checking
- For refutation (debugging) this is good enough, but for verification, we need highly accurate models
- Model checking is acceptable in industry, good for debugging, but not for verification
- Theorem proving can do verification, but interactive verification is not acceptable in industry, and automatic theorem proving often fails

Abstraction for Refutation and Verification

Automated abstractions promise to overcome this dilemma

- Can move smoothly between refutation and verification
- Interaction is in terms of design
- Failed proofs are not catastrophic

Related Approaches

- David Dill's treatment
  - Variant on predicate abstraction; can do garbage collector automatically in an hour or so, CAV '99
- Ken McMillan's treatment
  - Lightweight theorem proving whose subgoals are discharged by model checking; can do Tomosulo automatically, CAV 00
- Bandera (Matt Dwyer, John Hatcliffe, Kansas State)
  - Uses abstract interpretation, works on Java, ICSE 00
- SLAM (Tom Ball, Sriram Rajamani, Microsoft Research)
  - Boolean programs, WAVE 00
- Veritech (Shmuel Katz, Orna Grumberg, The Technion)
  - Another intermediate rep'n; eclectic front and backends

Other Promising Approaches

Symbolic evaluation has some of the same benefits as integrated abstraction

- Results of symbolic evaluation are considered in reviews (rather than have reviewers mentally execute the code)
- Theorem proving simplifies expressions during evaluation
  - Failure is not catastrophic
- Used by Rockwell Collins for AAMP and JEM avionics and Java processors

And Still Others

- Use deduction to refine the analyses of standard tools
  - Compilers (ESC, optimization), static analyzers, slicers
- Provide new, targeted, analysis tools
  - E.g., Daniel Jackson's Alcoa for object models
- Develop powerful methods for specific domains
  - E.g., checkers for tabular specifications (lots of verification conditions), railroad switching, mode logic for embedded systems

To Follow Up

- Browse general papers and technical reports at `http://www.csl.sri.com/fm.html`
- Information about 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
- Most of the work was conceived and performed by my colleagues: Judy Crow, David Cyrluk, Pat Lincoln, Sam Owre, Harald Rueß, Shankar, Srivas, Dave Stringer-Calvert