# Automated Deduction and Formal Methods

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

## A Missed Opportunity?

- Formal methods are a natural application for automated deduction
  - Without automation they are neither formal, nor methods
- Yet most tools for formal methods use little more than elementary techniques from automated deduction
- And few theorem provers are used in formal methods
- Why is this? What capabilities are needed?
- A look at some examples

## Extent of Automated Deduction in Formal Methods

- Traditional formal methods (e.g., VDM, Z)
  - Automation ranges from none to rudimentary ("Proof Assistants")
- Methods that stress tools (e.g., B and RAISE)
  - Automation ranges from very to... fairly rudimentary
- Automated verification methods (e.g., model checkers)
  - Fast, effective... but specialized
- Theorem provers used for FM (e.g., ACL2, Nqthm, RRL)
  - Strong on rewriting and arithmetic; little or no quantification
- Integrated verification systems (e.g., Coq, Eves, HOL, LP, Nuprl, PVS, STeP, VSE)
  - Interactive; some have quite substantial automation, but based on relatively few techniques

## Good Sources of Examples

- "Critical systems" (those where cost of failure is unacceptable) currently provide the best opportunities for formal methods
  - Safety and security-critical systems
  - Consumer electronics (processors, smartcards)

- "Program verification" (proving consistency between executable code and detailed specifications)
  is no longer of much industrial interest
  - Traditional techniques are effective (if crude)
  - Very few critical coding bugs escape detection
  - Accounts for $< 2\%$ of bugs and $< 10\%$ of costs in sophisticated industries (e.g., aerospace)

  Unless it can be *totally* automated (as for some hardware)

## Good Sources of Examples

- Main opportunities are in requirements analysis ($\sim 30\%$ of costs and $> 50\%$ of faults)
- And intrinsically hard problems of design
  - Has to be after-the-fact verification: unlikely to solve these hard problems by derivation or programs-from-proofs
  - Typically concern distributed, concurrent, real time, fault tolerant systems
- Let's start with an example that involves all of these

# Clock Synchronization

- Context: fault-tolerant systems for critical real-time applications (aircraft flight control)
- Idea: run redundant processors in lock-step, perform identical calculations in all components, use exact-match voting
- Subproblems:
  - Interactive Consistency (aka. Byzantine Agreement)
    - ⋆ Single-source data are distributed consistently
  - Clock Synchronization
- Assumptions about behavior of faulty components
  - *Byzantine* fault model: no assumptions
  - *Hybrid* fault model: some faults are "arbitrary" (Byzantine), some "symmetric," some "manifest" (tolerates more faults)

# Interactive Convergence Clock Synchronization Algorithm

- Due to Lamport and Melliar-Smith (JACM, Jan 85)
- Clocks drift (at rate bounded by $\rho$) and need to be resynchronized periodically to keep them within $\delta$ of each other
- Each processor reads the difference between its own clock and that of each other processor ($\Delta_{q\,p}^{(i)}$)
- Sets any difference greater than $\Delta$ to zero
- Takes arithmetic mean of these modified differences
- And adjusts its own clock by that amount
- Byzantine fault model:
  requires $n > 3m$ processors to withstand $m$ faults
- Hybrid fault model: $n > 3a + 2s + c$ processors to withstand $a$ arbitrary, $s$ symmetric, and $c$ manifest faults simultaneously

# Formal Treatment

- Clocks are uninterpreted functions from "clock time" to "real time" (both interpreted as reals)
  - $c_p(T)$ is the real time when $p$'s clock time is $T$
- Synchronize every $R$ clock time units, by own clock
  - $i$'th period denoted $R^{(i)}$, starts at $T^{(i)}$

$$T^{(i)} = T^{(0)} + iR$$

$$R^{(i)} = [T^{(i)}, T^{(i+1)}]$$

  - Algorithm takes $S$ clock units at end of period

$$S^{(i)} = [T^{(i+1)} - S, T^{(i+1)}]$$

- Adjustment to processor $p$'s clock in $i$'th period is $C_p^{(i)}$
  - Adjusted clock is $c_p^{(i)}(T) = c_p(T + C_p^{(i)})$

## Requirements ("Clock Synchronization Conditions")

**Bounded skew (S1):** If $p$ and $q$ are nonfaulty through period $i$, then

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta$$

for all $T$ in $R^{(i)}$.

**Bounded adjustment (S2):** If processor $p$ is nonfaulty through period $i$, then

$$|C_p^{(i+1)} - C_p^{(i)}| < \Sigma.$$

For real time constant $\delta$ and clock time constant $\Sigma$ that are parameters to the algorithm

## Assumptions

Several of these including

**A2:** If the clock synchronization conditions hold for the $i$'th period, and processors $p$ and $q$ are nonfaulty through period $i$, then

$$|\Delta^{(i)}_{q\,p}| \leq S$$

and

$$|c^{(i)}_p(T' + \Delta^{(i)}_{q\,p}) - c^{(i)}_q(T')| < \epsilon$$

for some time $T'$ in $S^{(i)}$

# Algorithm

For all processors $p$:

$$C_p^{(i+1)} = C_p^{(i)} + \Delta_p^{(i)},$$

where

$$\begin{aligned}
C_p^{(0)} &\quad \text{is arbitrary,} \\
\Delta_p^{(i)} &= \left(\frac{1}{n}\right) \sum_{r=1}^{n} \bar{\Delta}_{r\,p}^{(i)}, \qquad \textbf{and} \\
\bar{\Delta}_{r\,p}^{(i)} &= \textbf{if } |\Delta_{r\,p}^{(i)}| < \Delta \textbf{ then } \Delta_{r\,p}^{(i)} \textbf{ else } 0
\end{aligned}$$

and $\Delta$ is a clock time quantity that is a parameter to the algorithm

# Constraints

**C0:** $n > m \geq 0$

**C1:** $R \geq 3S$

**C2:** $S \geq \Sigma$

**C3:** $\Sigma \geq \Delta > 0$

**C4:** $\Delta \geq \delta + \epsilon + \frac{\rho}{2} S$

**C5:** $\delta \geq \delta_0 + \rho R$

**C6:** $\delta \geq 2(\epsilon + \rho S) + \frac{2m\Delta}{n-m} + \frac{n\rho R}{n-m} + \frac{n\rho\Sigma}{n-m} + \rho\Delta$

# Lemmas

1. If the clock synchronization conditions S1 and S2 hold for $i$, and processors $p$ and $q$ are nonfaulty through period $i+1$, then

$$|\Delta_{q\,p}^{(i)}| < \Delta$$

2. If processor $p$ is nonfaulty through period $i+1$, and $T$ and $\Pi$ are such that $A_p^{(i)}(T)$ and $A_p^{(i)}(T+\Pi)$ are both in the interval $[A_p^{(0)}(T^{(0)}), A_p^{(i+1)}(T^{(i+2)})]$, then

$$|c_p^{(i)}(T+\Pi) - [c_p^{(i)}(T)+\Pi]| \leq \frac{\rho}{2}|\Pi|$$

where $A_p^{(i)}(T) = T + C_p^{(i)}$

3. If the clock synchronization conditions S1 and S2 hold for $i$, processors $p$ and $q$ are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then

$$|c_p^{(i)}(T+\Delta_{q\,p}^{(i)}) - c_q^{(i)}(T)| < \epsilon + \rho S$$

## Lemmas (continued)

4. If the clock synchronization conditions S1 and S2 hold for $i$, processors $p$ and $q$ are nonfaulty through period $i+1$, $r$ is nonfaulty through period $i$, and $T \in S^{(i)}$, then

$$|c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)}]| < 2(\epsilon + \rho S) + \rho\Delta$$

5. If the clock synchronization condition S1 holds for $i$, processors $p$ and $q$ are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then

$$|c_p^{(i)}(T) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{r\,q}^{(i)}]| < \delta + 2\Delta$$

6. If processors $p$ and $q$ are nonfaulty through period $i+1$, and $T \in R^{(i+1)}$, then

$$\begin{aligned} &|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)| \\ &\quad \leq \ |c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]| + \rho(R + \Sigma) \end{aligned}$$

## Example Proof (Lemma 5)

Using the arithmetic identity

$$(a+x)-(b+y)=(a-b)+(x-y),$$

we obtain

$$\begin{aligned} & |c_p^{(i)}(T)+\bar{\Delta}_{r\,p}^{(i)}-[c_q^{(i)}(T)+\bar{\Delta}_{r\,q}^{(i)}]| \\ = \;& |c_p^{(i)}(T)-c_q^{(i)}(T)+\bar{\Delta}_{r\,p}^{(i)}-\bar{\Delta}_{r\,q}^{(i)}| \\ \leq \;& |c_p^{(i)}(T)-c_q^{(i)}(T)|+|\bar{\Delta}_{r\,p}^{(i)}|+|\bar{\Delta}_{r\,q}^{(i)}| \end{aligned}$$

The result follows on applying S1 to the first term on the right hand side, and observing that the Algorithm ensures that the remaining two terms are no larger than $\Delta$

# Main Theorem

Lemma 6 gives

$$\left|c_p^{(i+1)}(T) - c_q^{(i+1)}(T)\right|$$
$$\leq \left|c_p^{(i)}(T^{(i+1)}) + \Delta_p^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \Delta_q^{(i)}]\right| + \rho(R+\Sigma)$$

By the Algorithm, the right hand side equals

$$\left|\left(\frac{1}{n}\right)\sum_{r=1}^{n}(c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)}])\right| + \rho(R+\Sigma)$$
$$\leq \left(\frac{1}{n}\right)\sum_{r=1}^{n}|c_p^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,p}^{(i)} - [c_q^{(i)}(T^{(i+1)}) + \bar{\Delta}_{r\,q}^{(i)}]| + \rho(R+\Sigma)$$
$$\leq \left(\frac{1}{n}\right)[(n-m)(2[\epsilon+\rho S] + \rho\Delta) + m(\delta + 2\Delta)] + \rho(R+\Sigma)$$

where the first term is obtained by applying Lemma 4 to the $n-m$ nonfaulty processors, and the second is obtained by applying Lemma 5 to the $m$ faulty ones. The result then follows from the Constraint C6.

## Formal Verification

- Done in 1988 by von Henke and me using Ehdm (TSE, Jan 93)
- Found that 4 of the 5 lemmas in the JACM description are incorrect as stated

  And proof of main theorem is flawed—its induction has the form

  $$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta \;\text{ implies }\; |c_p^{(i+1)}(T') - c_q^{(i+1)}(T')| \lesssim \delta$$

- Repeated by Young in 1991 using Nqthm
  - With almost all automation turned off

## Capabilities Needed

- Nonlinear arithmetic with division, involving both reals and integers

- Both interpreted (e.g., $|\star|$, $c_p^{(i)}$) and uninterpreted (e.g., $c_p$, $\Delta_{q\,p}^{(i)}$) function symbols

- Induction
  - Main theorem is proved by induction over $i$
  - Also need induction over $1 \ldots n$ to justify splitting the summation into two parts

- Not much quantifier reasoning

## Capabilities Needed (Arithmetic)

- Ubiquity of arithmetic demands some automation
- Linear arithmetic would be a good start—for example
  - Decision procedures for Presburger arithmetic (as in HOL)
- But we have uninterpreted function symbols as well
- Better choice is a decision procedure for *ground* linear arithmetic (Bledsoe, Pratt, Shostak)
  - Over both integers and reals
  - *Combined* with equality over uninterpreted function symbols
- Small extensions to nonlinear case (Brooks, Sachs) are pragmatically significant (easier in SUP-INF than loop residue)
  - Associative, commutative, distributive, cancellation laws
  - Law of signs (minus times minus is plus)

## Capabilities Needed (First Order)

Very little nested or explicit quantification (A2 is the exception)

So lemma instantiation is the main issue

- Some lemmas can be formulated as conditional rewrite rules and applied automatically
  - Rewriting also handles definitions
  - Needs to be *integrated* with decision procedures
    - ⋆ To find matches under arithmetic
    - ⋆ To discharge conditions
- Other lemmas can be instantiated quite effectively by simple matching heuristics (but could do better)
- Selection of lemmas left to user guidance
  - Specific instances found automatically on basis of types
    - ⋆ E.g., Simple vs. bounded vs. structural induction

Also harder instantiations

# Capabilities Needed (Scaling)

- The combination of ground linear arithmetic, equality, arrays

$$i.e., f \textbf{ with } [(x) := y](z) \stackrel{\text{def}}{=} \textbf{if } z = x \textbf{ then } y \textbf{ else } f(z),$$

  propositional calculus, and rewriting is enough to tackle many interesting problems

- But can get overwhelmed by scale
  - Intermediate formulas in processor pipeline-correctness proofs can exhaust memory, take 60 pages to print
  - Proofs for SRT lookup table can run for three hours

- Highly optimized implementations are necessary
  - E.g., Barrett, Jones, Levitt, and Dill's SVC (FMCAD 96)
  - Uses structure sharing, Davis-Putnam, C++
  - As a backend to PVS, provides two orders of magnitude speedup

## Capabilities Needed (Scaling)

- Rewriting can be a bottleneck on huge formulas
- E.g., Bitvector manipulations in processor correctness proofs
  - Can take tens of minutes
  - Can exhaust memory, forcing user to become concerned with *how* the automation works
- Conjecture that a decision procedure for a fragment of the bitvector theory would eliminate this problem
- Many techniques from the automated verification community (BMDs, QDDs) can be incorporated into theorem proving as decision procedures

## Capabilities Needed (Libraries)

- A bare specification language is only the beginning
- Most applications of formal methods require substantial investment to develop supporting theories
- Decent libraries are necessary
- Their development is difficult
  - Design choices are critical to utility and efficiency
  - Maintenance is a nightmare

## Capabilities Needed (Interaction)

- Cannot expect to prove fairly large, hard theorems from arbitrary domains without some user guidance and interaction

- Many tools for formal methods are all interaction and no automation
  - Focus is on interface issues

- Whereas what's needed is a balance between automation and interaction
  - That can shift as attention moves between easy and hard parts of a proof

## Capabilities Needed (Interface)

- Interface should allow user to comprehend the big picture
  - Where am I?
  - Where did this subgoal come from?

- And how to proceed
  - Formulas should be recognizable
  - Not cluttered with extraneous information
  - Commands should have reasonably predictable outcomes and costs
  - Not too many choices

## Capabilities Needed (Pruning and Debugging)

- There's more to formal methods than proving true theorems
- Want to know exactly what assumptions, lemmas were used
  - In order to prune them, sharpen their statements
  - Assumptions must be validated in the real world
    - ⋆ Trivial example: Clock synchronization originally assumed all initial corrections are zero (i.e., $C_p^{(0)} = 0$)
- Many (most) conjectures are false when first proposed
  - Need to fail quickly, return useful information
    - ⋆ Heuristics often be ill-suited to this requirement
  - Would like to quickly check specific instances, downscaled (finite state) versions, simplified conjectures

# Fast Feedback: Example

- Havelund and Shankar formally verified a bounded-retransmission communications protocol from Philips (FME 96)
- 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 Mur$\phi$
- 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 Mur$\phi$
  - Added to invariant *without quitting proof*
  - Iterated the above—57 times!

## Capabilities Needed (Coping with Changes)

Want to use automated formal methods to support active design exploration

- Like computational fluid dynamics supports design of aerofoils
- Change assumptions, requirements, or design, rerun proof
  - Automation should adapt gracefully to small changes
  - Pinpoint major deviations, and allow local "patching"
- E.g., can significantly tighten bound on clock synchronization by using observation (due to Palumbo and Graham) that there is no error when reading own clock

# Active Design Exploration: Simple Example

Extension of clock synchronization algorithm to hybrid fault model (PODC '94)

- Two additional cases in Lemma 5.
- Generalize splitting of summation from two to several parts
- More complicated constraints and assumptions
- About a week's work (cf. 2–3 months for original)
- Discovered previously verified version of Lemma 4 needs to be strengthened
- Inconceivable to do this without automation
  - True of most algorithms for hybrid fault model—too much case analysis to tackle unaided
    (e.g., bug in a published algorithm)

# Active Design Exploration: Larger Example

- There are dozens of clock synchronization algorithms
- Schneider showed many of them can be given a uniform treatment
- Shankar formally verified this general treatment (FTRTFT 92)
  - Establishes correctness of all algorithms in a certain class satisfying 11 assumptions
  - Found and corrected several small errors during verification
  - And showed interactive convergence satisfies the assumptions
- Paul Miner argued that one of the assumptions is very difficult to establish of a given algorithm
  - Almost as hard as proving synchronization

## Active Design Exploration: Larger Example (ctd. 1)

- Miner showed troubling assumption could be discharged once and for all from suitably modified versions of the other ten

- And showed these are satisfied by the Lundelius-Lynch algorithm
  - Fault-tolerant midpoint: read the other clocks, exclude $m$ slowest and $m$ fastest, and set own clock to mid-point of remainder

- Then built a circuit to do it

- Incorporating a clever optimization due to Torres-Pomales
  - Run own clock at half speed after seeing $m+1$'st clock, until see $n-m$'th

## Active Design Exploration: Larger Example (ctd. 2)

Formal methods applied to the optimized clock synchronization circuit (DCC 96)

- DDD (stream-based transformation system for hardware) used for routine design
- BDD-based tautology checker used for low-level optimizations
- PVS used for the clock-rate optimization
  - Needs fairly heavy arithmetic, again
  - Connection to DDD via streams (formalized as a co-inductive type)
  - Optimization verified as a bisimulation on streams

# The Theorem Proving Lifecycle

There is a lifecycle to theorem proving during formal development

Need different capabilities for each stage

- Exploration: must fail quickly, be able to stay in the loop
  Specialized methods are very useful
  - Animation (direct execution/simulation)
  - State exploration (reachability analysis)
  - "Backwards execution" (fault-tree analysis)
  - Efficient methods for specialized cases (model checking)
- Development: efficiency!
  Plus ability to comprehend overall structure
- Presentation: want a real proof
- Maintenance and Generalization: robustness

## Capabilities Required (Tool and Method Integration)

- Already described how Mur$\phi$ (state exploration) and DDD (transformation) have been used together with theorem proving
- Need to make use of other specialized methods, and integrate them more closely
- Example: finite-state model checking
  - Real system of interest is usually infinite state
  - Conventionally, downscale or abstract to finite state by hand
  - Then model check the reduced system
  - Design exploration performed on reduced system
- Theorem proving could justify and automate the reduction
  - Model checking eliminates need for invariants (hmm...)

## Tool and Method Integration: Model Checking

- Need to be able to describe a temporal logic, its models, and satisfaction relation in the logic of the theorem prover
- Model checker is then a decision procedure for a finite-state fragment
- Example: CTL, transition relations, and the satisfaction relation between them can be specified in PVS in terms of Park's $\mu$-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 $\mu$-calculus is available as a backend to PVS
- Combination provides CTL model checking on finite-state systems

# Tool and Method Integration: Example

- Havelund and Shankar abstracted bounded-retransmisison protocol 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 $\mu$-calculus embedding
- Verified by theorem proving that abstraction preserved properties of interest
  - But needed 45 invariants to establish this
- Need to integrate with additional tools for generating abstractions and invariants

## Tool and Method Integration: Another Example

- $N$-process mutual exclusion can be recursively reduced to two-way
    - Abstract first $N-1$ processes to a single process
    - Establish mutual exclusion between that and $N$'th process

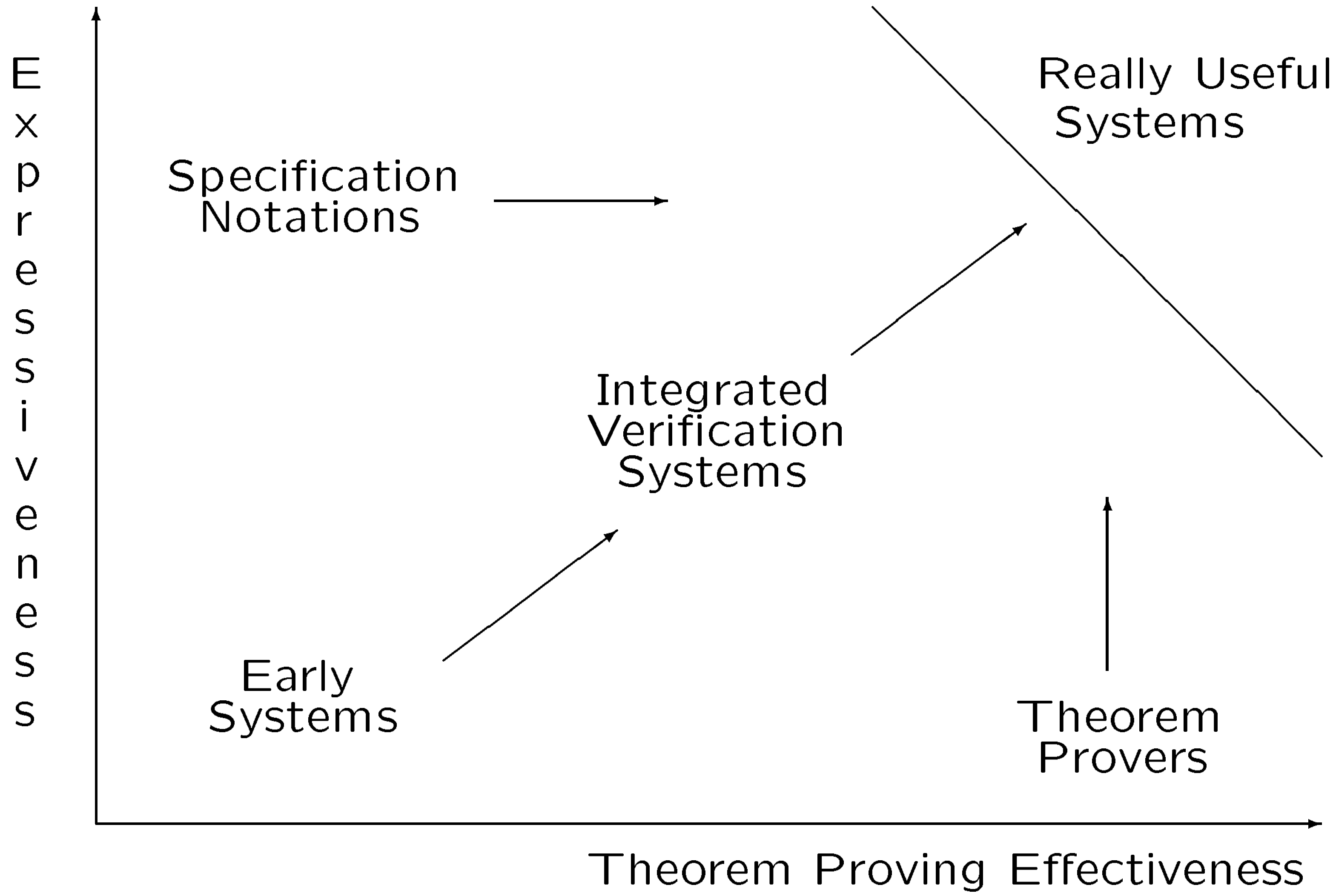
- Proof by abstraction plus induction
    - With the inductive step established by model checking

- Additional possibilities: use data structures from model checking, with node-equivalence decided by deduction rather than eq-test
    - Seen, to some extent, in SVC and Queue BDDs

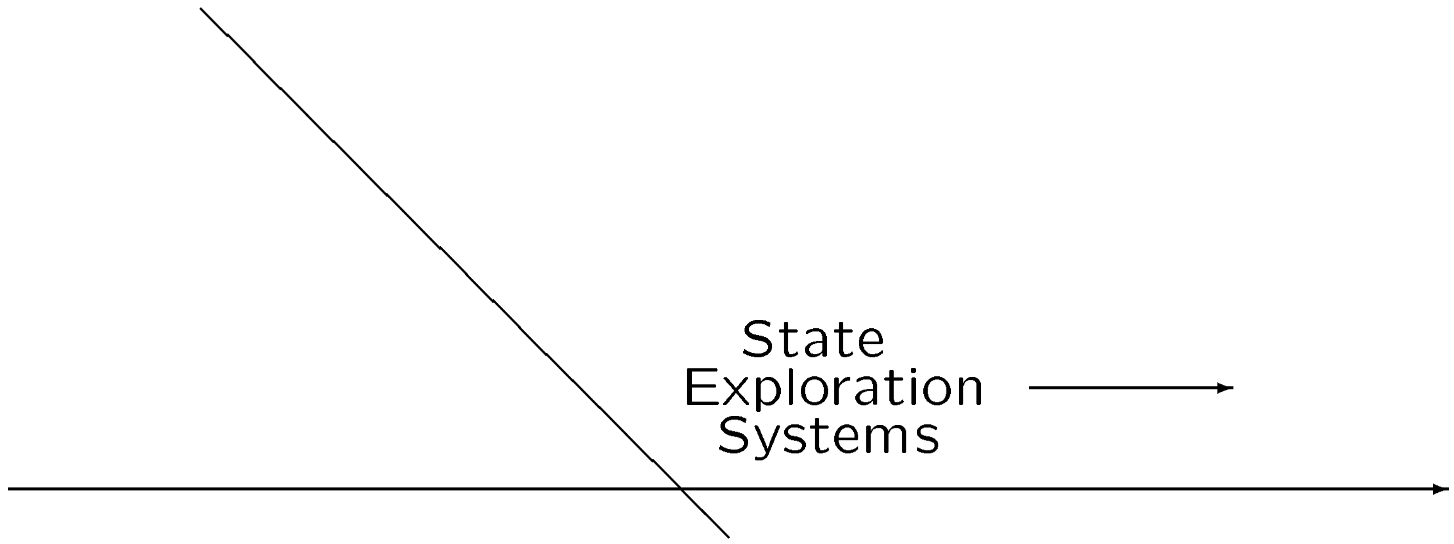
# Capabilities Needed (Language)

- Formal methodists want to use a specification language, not a bare logic
  - Rich typing discipline
  - Computer science data types (numbers, lists, trees)
  - Attractive syntax
  - Special notations—e.g., tables
- But they make choices inimical to efficient automation
  - Set theory
  - Functions are partial
  - No definitional constructs
- Need more engagement between the two communities to develop specification languages that allow efficient automated deduction

# The Three Traditions



## And A Fourth



## Reconciling the Conflicts

- You cannot design a specification language and then hope to add effective mechanization

- And you cannot build a theorem prover and hope to persuade very many people that its logic is a specification language

- The formal method, its specification language, and its tools have to be designed together
  - Tradeoffs and compromises are necessary
  - But new opportunities arise, too

## Summary

- Automated deduction and automated verification provide many techniques that are impressively effective
  - When applied to suitable problems
- Few applications of formal methods coincide exactly with these problems
- The need is for integrated systems that bring several techniques together
  - Integration needs to be tight
  - Requires major system-building effort
- And the integration needs to be extended through the design of the specification language

## The Opportunity

- Successful application of automated deduction to formal methods will enrich both fields
  - New opportunities for research and use of automated deduction
  - Make formal methods a truly useful and practical tool
- Tools are not the most important thing about formal methods
  - They are the *only* important thing