TU Vienna 27 March and NSA 29 March 2001

Based on HASE 2000

Disappearing Formal Methods

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Overview

- Assurance for Safety, Security, and other critical properties
    - Process- vs. product-based assurance
- Formal methods
- Problems with current methods
- Two big ideas
- From refutation to verification
- Disappearing formal methods

Evidence For Safety, Security, And Other Critical Properties

- How is it done for traditional systems?
  - E.g., an airplane wing
- How is it done for software?
  - Or software-intensive systems
  - E.g., a flight-control system

Safety Cases for Traditional Systems

- Mostly done by mathematical modeling and analysis
  - Build mathematical models of the design, its environment, and requirements
  - Use calculation to establish that the design in the context of the environment satisfies the requirements
  - Only useful when mechanized

  E.g., finite elements analysis
- The modeling is validated by tests
  - Limited testing is sound because we are dealing with continuous systems
- This is product-based certification
  - It concerns properties of (mathematical models of) the product

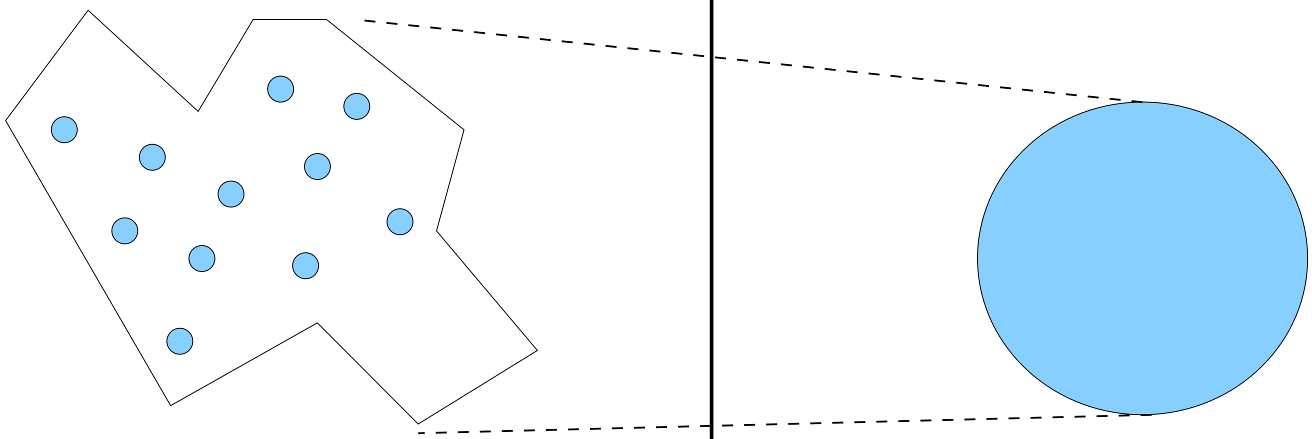
Safety Cases for Software Systems

- Mostly done by controlling, monitoring, and documenting the process used to create the software
  - Different industries have different recommended processes (e.g., DO-178B for avionics)
- This is process-based certification
  - Provides no direct evidence about the product

    "We cannot show how well we've done, so we'll show how hard we tried"
- NB. Testing is product-based, but cannot provide evidence beyond 10^{-4} because we are dealing with discrete systems
  - Complete testing is infeasible: 114,000 years test for 10^{-9}
  - And extrapolation from incomplete tests is unjustified

Formal Methods In Pictures

Testing/Simulation



Real System

- Partial coverage

Formal Analysis

Formal Model

- Complete coverage (of the modeled system)

  Accurate model: verification

  Approximate model: debugging

Product-Based Certification For Software

- Build mathematical models of a design, its environment, and requirements
  - The applied math of Computer Science is formal logic
  - So models are formal descriptions in some logical system:
- Use calculation to establish that the design in the context of the environment satisfies the requirements
  - Calculation in formal logic is done by theorem proving or model checking

    $$assumptions + design + environment \vdash requirements$$

    Formal calculations can cover all modeled behaviors, even if numerous or infinite (the power of symbolic reasoning)
- Only useful when mechanized
  - So need automated theorem proving or model checking

Formal Methods for Product-Based Assurance and Certification

- Want highly accurate formal models, so that calculations support strong claims—i.e., verification
- Then, using formal calculations, some activities that are traditionally performed by reviews
  - Processes that depend on human judgment and consensus

  can be replaced or supplemented by analyses
  - Processes that can be repeated and checked by others, and potentially so by machine

  Language from DO-178B/ED-12B
- That is, formal methods help us move from process-based to product-based assurance

However. . .

- Formal calculations
  - Are undecidable in general
  - And even decidable problems have much greater computational complexity than mechanizations of continuous mathematics
- So full automation is impossible in general
- Must rely on heuristics (guesses) which will sometimes fail
  - Heuristic theorem proving
- Or rely on human guidance
  - Interactive theorem proving
- Or trade off accuracy or completeness of the model for tractability and automation of calculation
  - Model checking

The Difficulty With Theorem Proving Is. . .

- Theorem proving can handle accurate models, but requires interactive human guidance
  - Focuses on proof, and idiosyncrasies of the prover, not on the design
  - Difficult to interpret failure (bug, or bad proof?)

  "Interactive theorem proving is a waste of human talent"

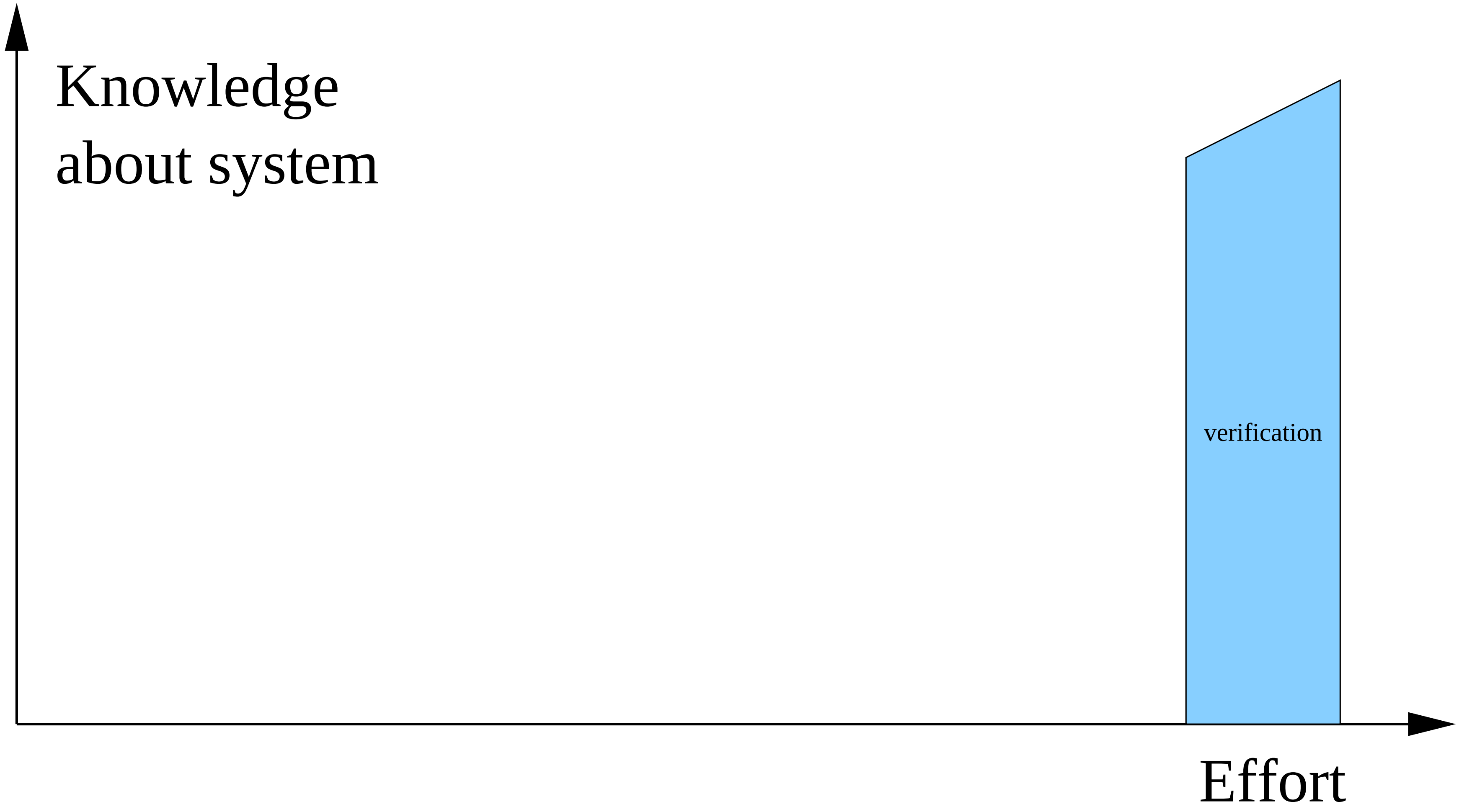
- Also, must strengthen invariants to make them inductive
- And it's all or nothing
- Payoff is definitive assurance. . . with caveats
  - May also find subtle bugs

Inductive Invariants

- To establish an invariant or safety property (one true of all reachable states) by theorem proving, we invent another property that implies the one of interest and that is inductive
  - Includes all the initial states
  - Is closed on the transitions

  The reachable states are the smallest set that is inductive
- Trouble is, naturally stated invariants are seldom inductive
  - The second condition is violated
- Postulate a new invariant that excludes the states (so far discovered) that take you outside the desired invariant
- Iterate until success or exasperation
- Bounded retransmission protocol required 57 such iterations

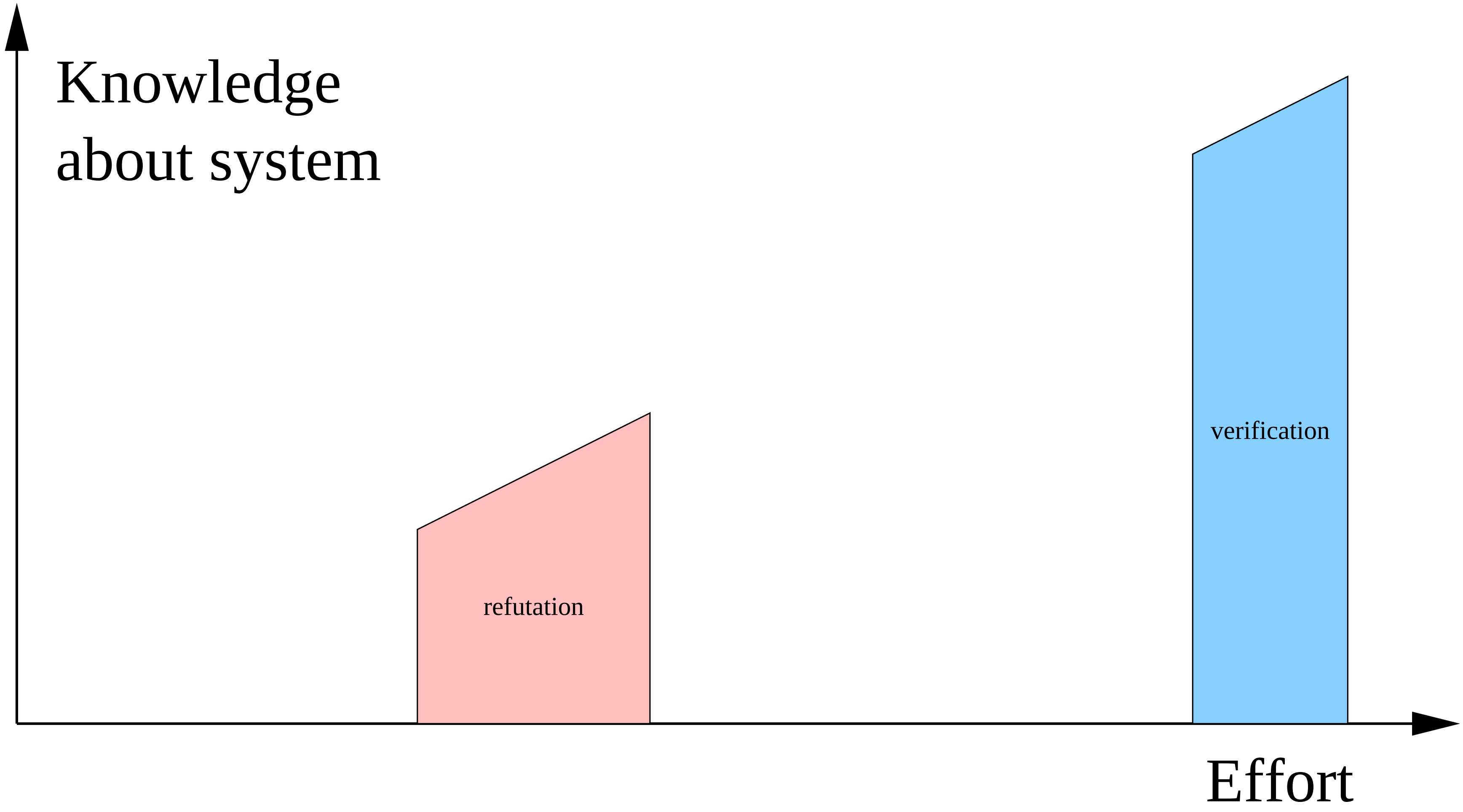
The Wall of Formal Verification



The Difficulty with Model Checking Is. . .

- The models (and properties) have to be simplified to make them tractable to fully automated analysis
- But simplified models may not be fully accurate with respect to the property of interest
  - And that's why they cannot be used for verification
- However, this approach works for refutation (finding bugs)
  - Experience indicates we learn more (find more bugs) by exploring all behaviors of a simplified model than by probing just some of the behaviors of the real thing (as with testing or simulation)
- But when to stop?
  - Lack of refutation is not the same as verification

Refutation and Verification



Formal Methods in Current Practice

- *Model checking saved the reputation of formal methods* (Daniel Jackson)
- Formal methods have achieved a modest degree of acceptance in some areas
  - E.g., hardware, protocols
- But mainly for purposes of refutation
  - That is, looking for errors
  - E.g., debugging, testing
- Verification is much less practiced
  - That is, showing the absence of errors

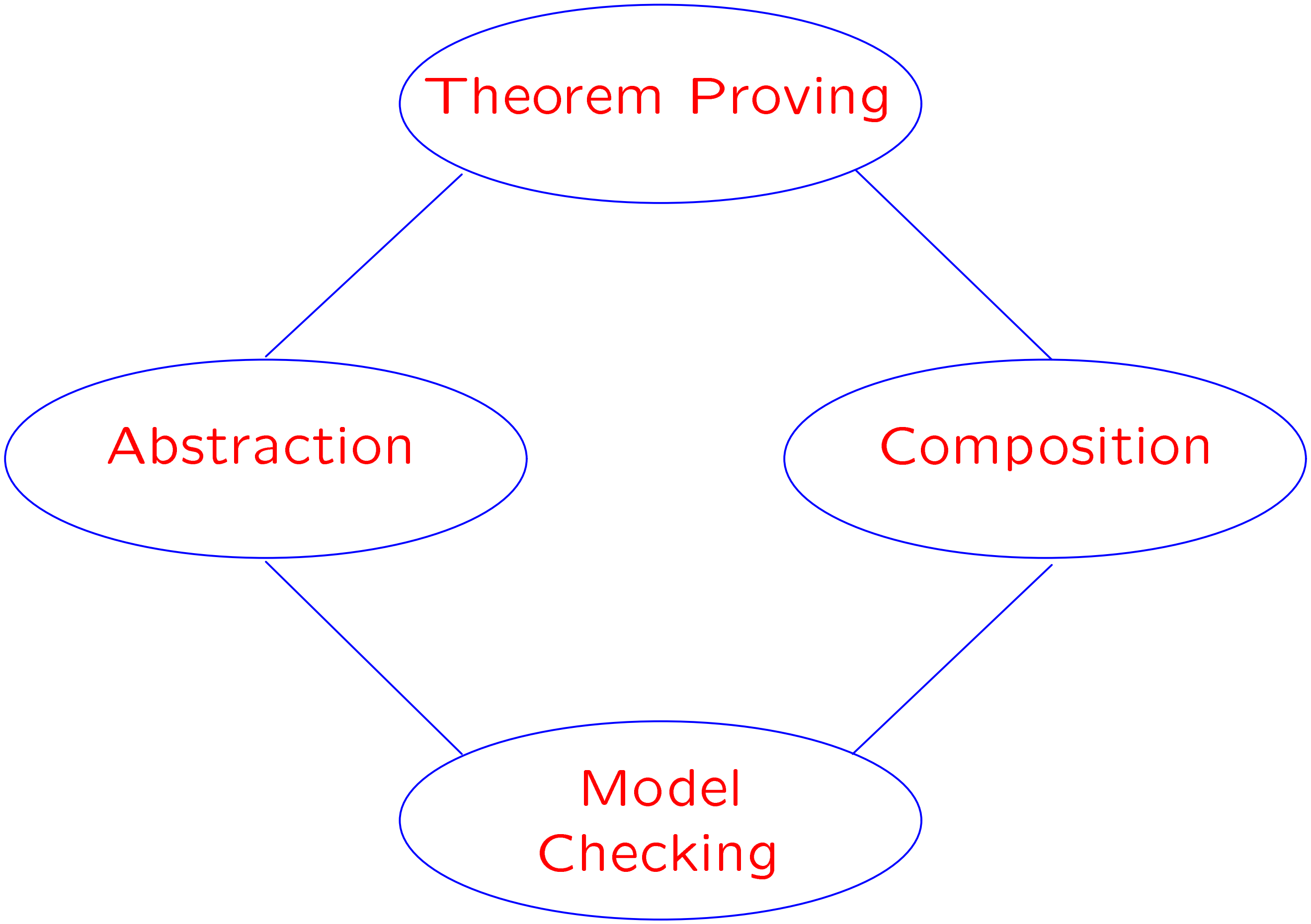
Summarizing

- Refut'n can be cost-effective, but doesn't get you to verif'n
  - Interaction concerns the model, the technology is automated, it resembles familiar activities
  - It is acceptable to practitioners

  Challenge: why cannot the technology of refutation (particularly model checking) be used for verification?
- Verif'n has high potential payoff, but few interm'd benefits
  - Interaction concerns the proof and the prover, technology is not automated, intimidating
  - It is not acceptable to practitioners

  Challenge: why cannot theorem proving be made automatic?
- Overall challenges: why cannot model checking and theorem proving work together? And why cannot we move smoothly from refutation to verification?

Abstraction is a Bridge Between Deductive and Algorithmic Methods And Between Refutation and Verification



Using Model Checking For Verification

- Model checking requires simple models (e.g., finite state)
- But can be used to verify properties of a complex model if it has a simple property-preserving abstraction
- Trouble is, it usually requires theorem proving to justify the abstraction
  - 45 of the 57 invariants required for BRP
- First Big Idea: use theorem proving to calculate the abstraction

Making Theorem Proving More Automatic

- The general theorem proving problem is undecidable
  - So full automation requires heuristics
  - Which will sometimes fail
- Classical verification poses correctness as a single "big theorem"
  - So failure to prove it (when true) is catastrophic
- Second Big Idea: "failure-tolerant" theorem proving
  - Prove lots of small theorems instead of one big one
  - In a context where some failures can be tolerated
- Aha! Automated abstraction provides this context

Abstraction

- Given a transition system G on S and property P, a property-preserving abstraction yields a transition system $\hat{G}$ on $\hat{S}$ and property $\hat{P}$ such that

$$\hat{G} \models \hat{P} \Rightarrow G \models P$$

- Strongly property preserving abstraction:

$$\hat{G} \models \hat{P} \Leftrightarrow G \models P$$

- A good abstraction typically (for universal properties) introduces nondeterminism while preserving the property
- Remaining problem: Construction of reasonably precise $\hat{G}$ and $\hat{P}$ given G and P

Data Abstraction [Cousot & Cousot]

- Replace concrete variable x over datatype C by an abstract variable x' over datatype A through a mapping $h : [C \rightarrow A]$.
- Examples: Parity, $mod\ N$, zero-nonzero, intervals, cardinalities, {0, 1, many}, {empty, nonempty}
- Given $f : [C \rightarrow C]$, construct $\hat{f} : [A \rightarrow set[A]]$:
(observe how data abstraction introduces nondeterminism)

$$b \in \hat{f}(a) \Leftrightarrow \exists x : a = h(x) \wedge b = h(f(x))$$

$$b \notin \hat{f}(a) \Leftrightarrow \vdash \forall x : a = h(x) \Rightarrow b \neq h(f(x))$$

- Theorem-proving failure affects accuracy, not soundness
- Mechanized in Bandera (Corbett, Dwyer and Hatcliff, KSU)

Predicate Abstraction [Graf-Saïdi]

- Abstracts out relations between variables, e.g., $x < y$, $x + y = z$
- Variables ranging over infinite datatypes can be replaced by Boolean variables representing the predicates on those variables
- Predicates can be extracted from guards, assignments, and the property of interest
- Guessing predicates is easier than invariant strengthening (and is also more general [Rusu & Singerman, TACAS 99])
- Mechanized in PVS (SRI)

Construction of Predicate Abstractions

- Given $\phi : [S \rightarrow \hat{S}]$ induced by the abstracted predicates, construct $\hat{G}$ by

$$\hat{G}(\hat{s}_1, \hat{s}_2) \Leftrightarrow \exists s_1, s_2 : \hat{s}_1 = \phi(s_1) \wedge \hat{s}_2 = \phi(s_2) \wedge G(s_1, s_2)$$

$$\neg\hat{G}(\hat{s}_1, \hat{s}_2) \Leftrightarrow \vdash \forall s_1, s_2 : \hat{s}_1 \neq \phi(s_1) \vee \hat{s}_2 \neq \phi(s_2) \vee \neg G(s_1, s_2)$$

- Theorem-proving failure affects accuracy, not soundness
- There is another method (exponentially more efficient) [Saïdi & Shankar, CAV 99]
- More powerful than data abstraction, but construction is more complex

Automated Abstraction

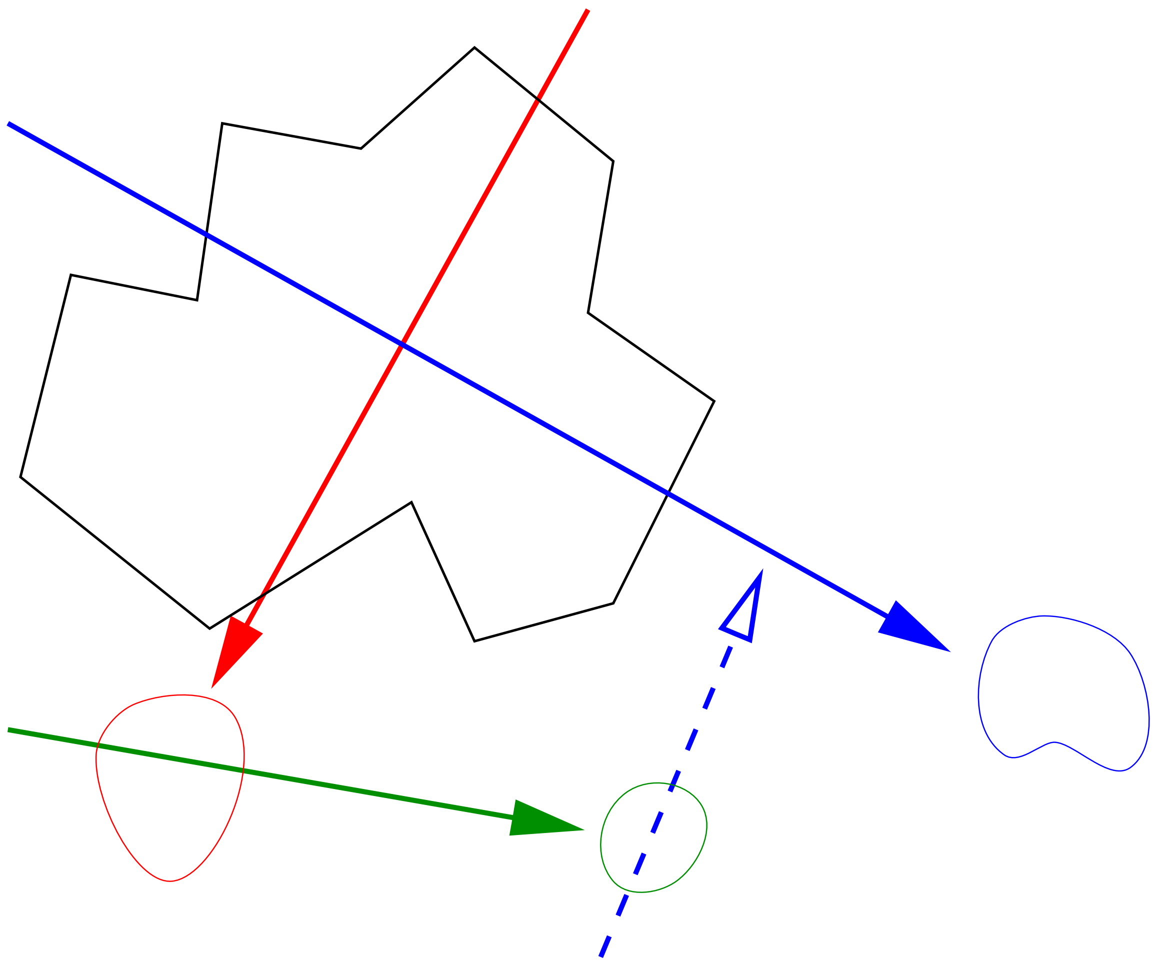
- Can often construct a simplified model that is faithful to the original (for a given property of interest)
  - The reduced model can by analyzed by model checking
  - And failure to detect bugs does certify their absence
- These reduced models can be constructed automatically by mechanized data or predicate abstraction
  - The construction is done by trying to prove lots of little theorems
    - ⋆ If a proof fails, the abstracted model will be more conservative, but often still good enough
- But still the construction often requires auxiliary invariants

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
- 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 [Bensalem, Lakhnech & Owre, CAV 99]

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

From Refutation to Verification

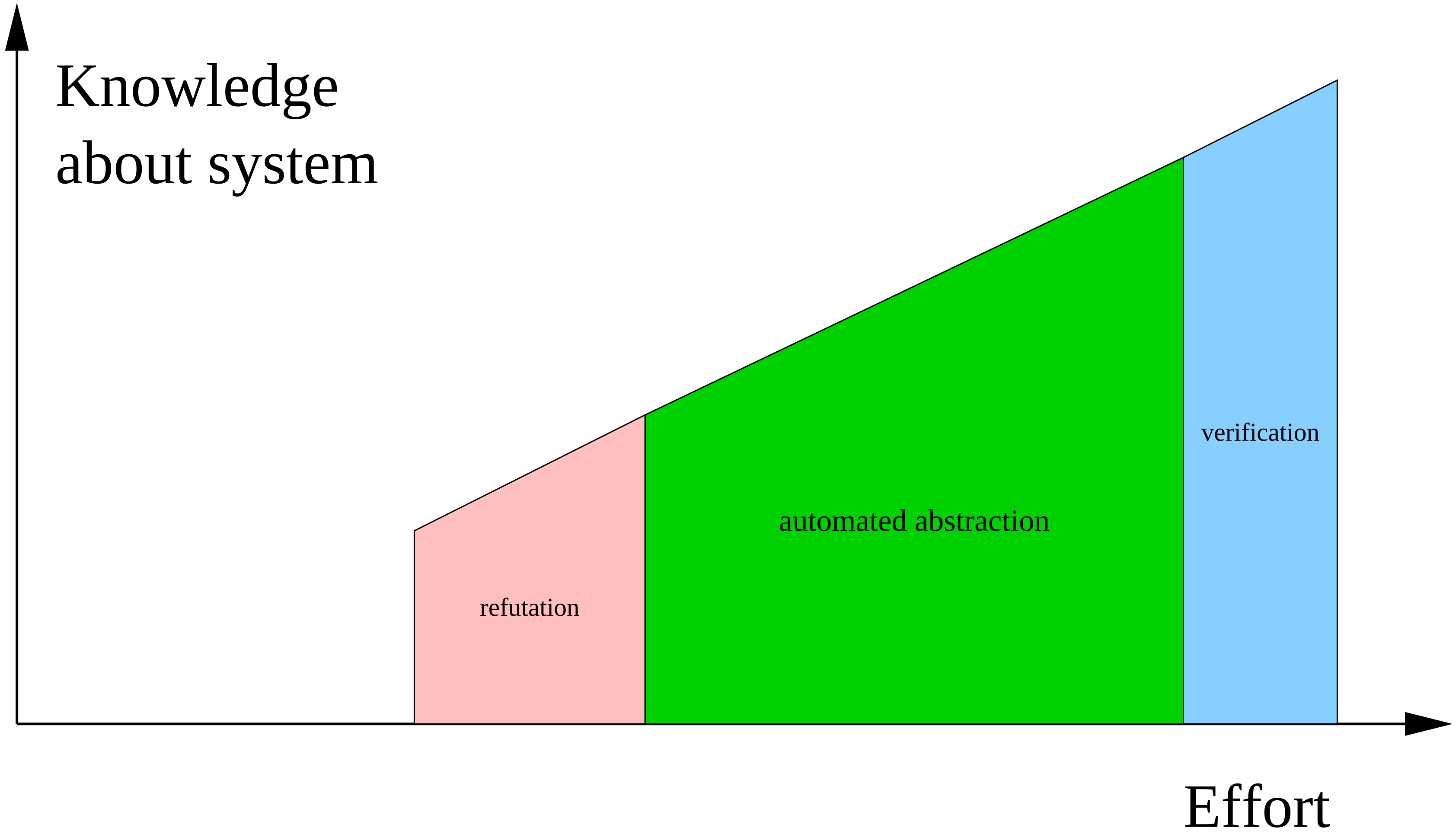
- By allowing unsound abstractions

$$\hat{G} \models \hat{P} \not\Rightarrow G \models P$$

  We can do refutation as well as verification

- By selecting abstractions (sound/unsound) and properties (little/big) we can fill in the space between refutation and verification
- Refutation lowers the barrier to entry
- Provides economic incentive: discovery of high value bugs
  - Can estimate the cost of each bug found
  - And can directly compare with other technologies
- Yet allows smooth transition to verification

From Refutation To Verification



Filling the Remaining Gap

- Model checking for refutation and (via automated abstraction) for verification imposes a much smaller barrier to adoption than old-style formal verification
- But the barrier is still there
- What about really low cost/low threat kinds of formal analysis?
- Make the formal methods disappear inside traditional tools and methods
  - Invisible formal methods, or
  - Ubiquitous formal methods

Examples of Disappearing Formal Methods

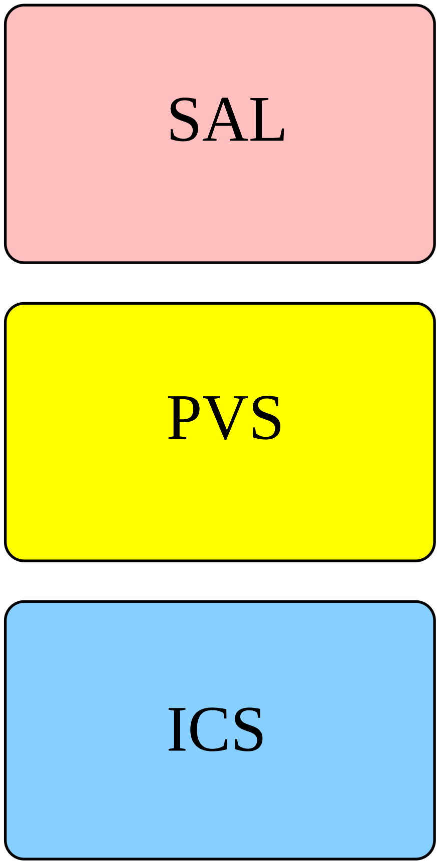
- Extended static checking (ESC) for Java (Compaq SRC)
- PVS-like type system (predicate subtypes) for any language
  - Traditional type systems have to be trivially decidable
  - But can gain enormous error detection by adding a component that requires theorem proving (lots of small theorems, failure generates a warning)
- Completeness/Consistency checkers for tabular specifications (cf. Ontario Hydro, RSML, SCR)
- Statechart/Stateflow property checkers (cf. OFFIS)
  - Show me a path that activates this state
  - Can this state and that be active simultaneously?
- Test case generators (cf. Verimag/IRISA TGV)

Tools To Realize These

- Abstraction and model checking
- Automated theorem proving built on powerful decision procedures
  - Combination of: propositional satisfiability, equality over uninterpreted function symbols with (linear) arithmetic, arrays, datatypes
  - Quantifier elimination for decidable fragment of the above

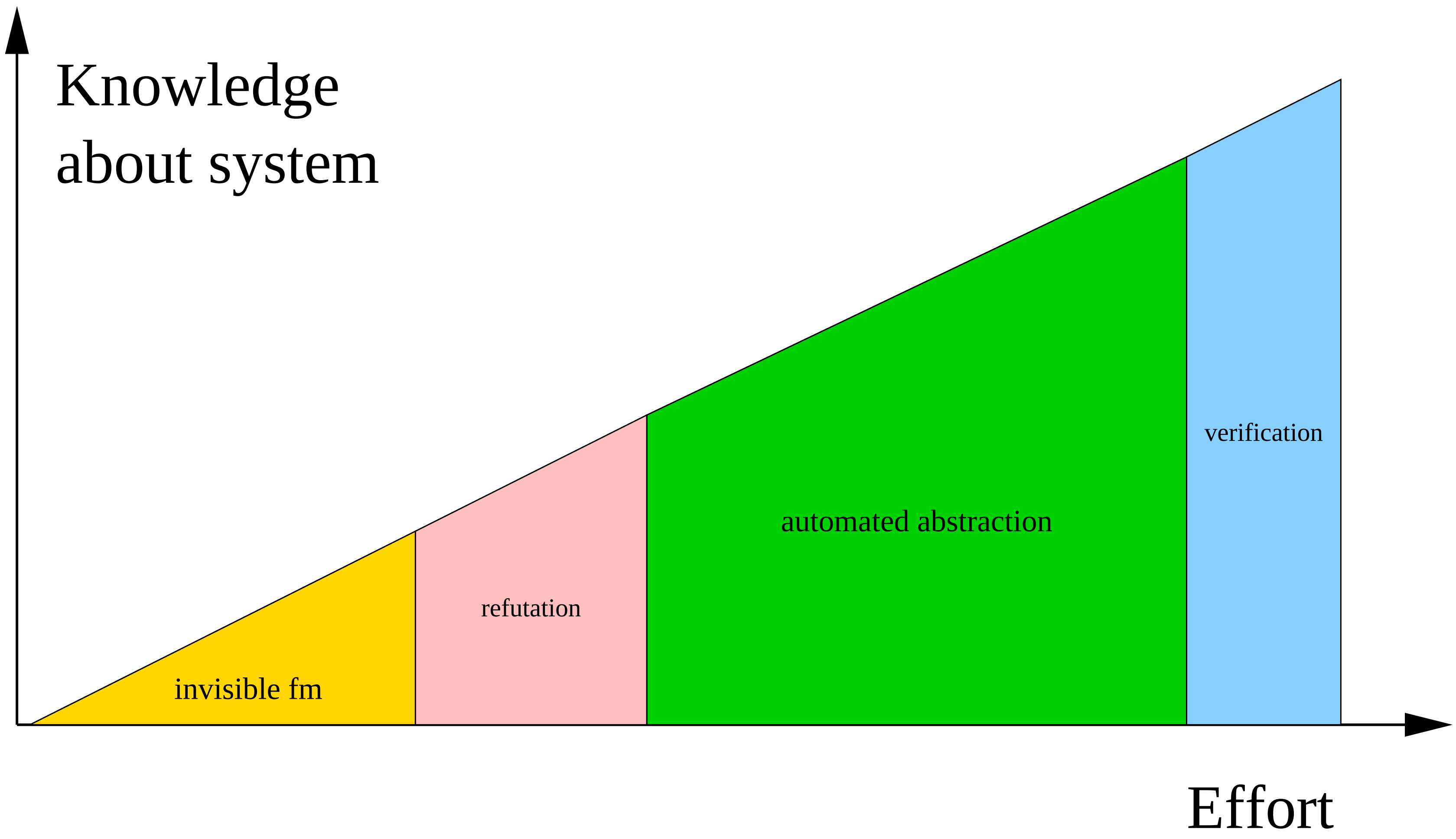
  We are making these available as ICS
- Also decision procedures for more powerful theories (e.g., Mona for WS1S, available in PVS)
- These can be extended to model checking
  - E.g., Lossy-Channel Systems (LCS)
  - Just as ordinary model checking builds on BDDs and SAT

What We Are building



ICS = Integrated Canonizer-Solver (= ICanSolve)

Disappearing Formal Methods



Acknowledgments

- N. Shankar, Sam Owre, Harald Rueß, Hassen Saïdi
- Saddek Bensalem, Jean-Christophe Filliâtre, Klaus Havelund, Friedrich von Henke, Yassine Lakhnech, César Muñoz, Holger Pfeifer, Vlad Rusu, Eli Singerman, and many others

To Learn More

- Check out papers and technical reports at http://www.csl.sri.com/programs/formalmethods
- Information about our verification system, PVS, and the system itself are available from http://pvs.csl.sri.com
  - Freely available under license to SRI
  - Built in Allegro Lisp for Solaris, or Linux
  - Version 2.3 includes predicate abstraction
- We plan to release SAL and ICS in July 2001