Tools for Formal Methods: ICS, SAL, and Stateflow

John Rushby

Computer Science Laboratory

SRI International

Menlo Park, California, USA

Introduction

- We've distributed the PVS Verification System since 1993
  - Formal specification and theorem proving environment comparable to Isabelle/HOL
- PVS is more automated than most other systems
  - Decision procedures, model checker, abstractor, evaluation
- But now we're looking for a massive increase in automation
  1. It's the only way for technology transfer into industrial/commercial use
  2. Part of a potential paradigm shift in how verification systems are built and used
- I'll talk mostly about the first of these, returning to the second near the end

Industrial Applications of Formal Methods

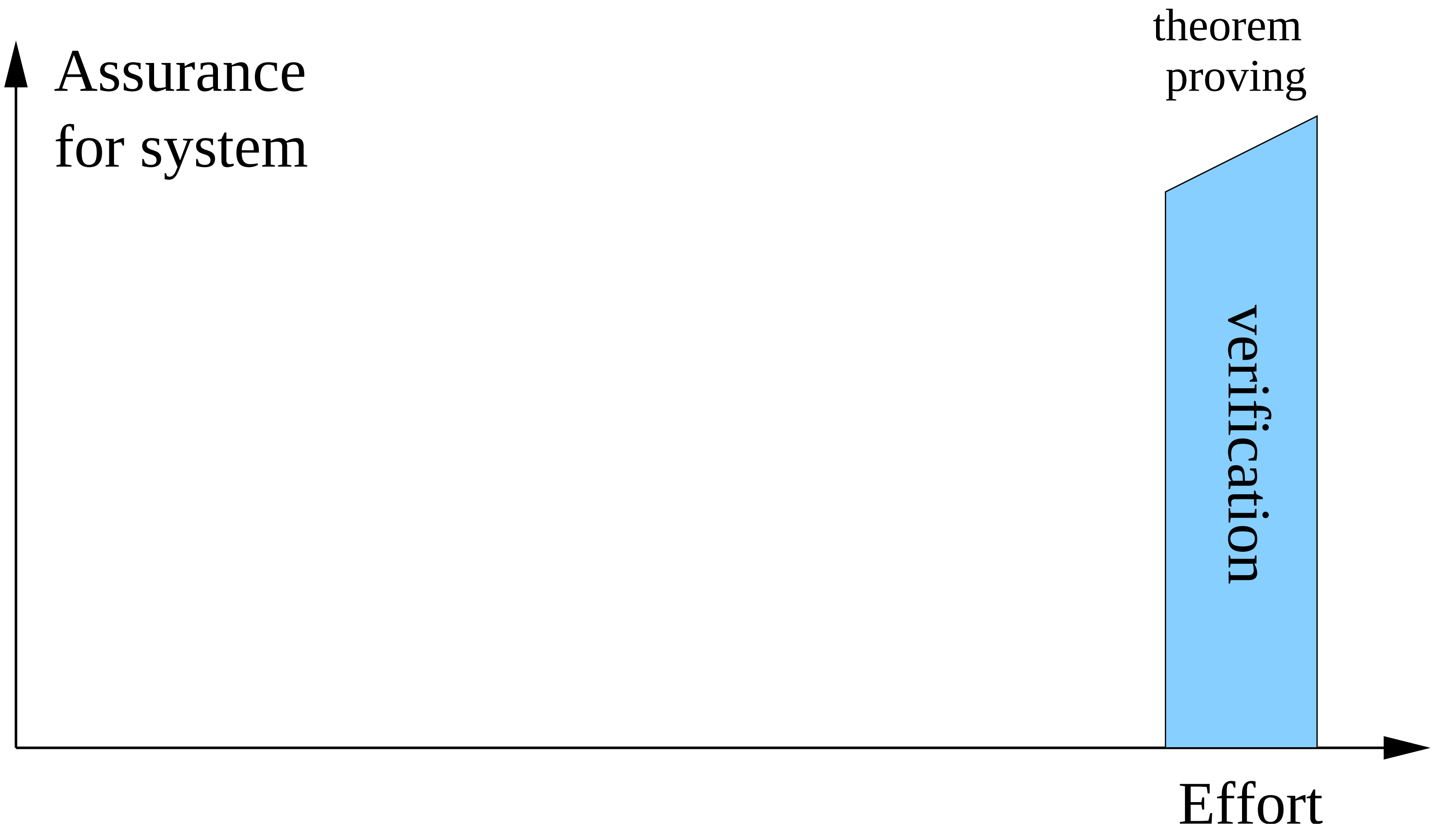
- Mostly driven by assurance concerns
  - In critical (often regulated) industries
  - And others with high cost of failure
- In these contexts, formal methods are about calculating properties of computer system designs
- Just like engineers in traditional disciplines use calculation to examine their designs
  - E.g., PDEs for aero, finite elements for structures
- So, with suitable design descriptions, we could use formal calculations to
  - Determine if all reachable states satisfy some property
  - Determine whether a certain state is always achievable

But Hasn’t That Been Tried and Failed?

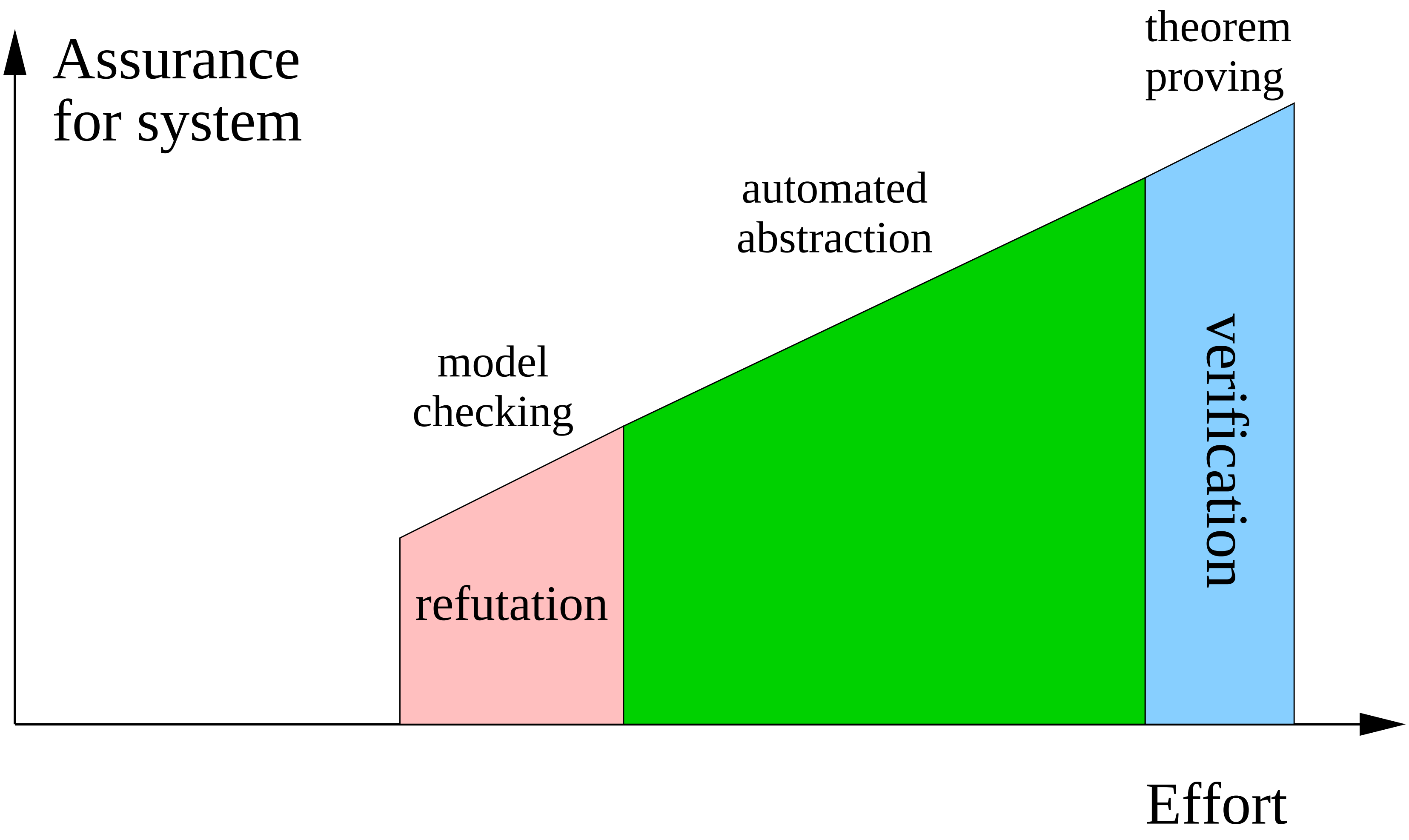
Yes, it failed for three reasons

- No suitable design descriptions
    - Code is formal, but too big, and too late
    - Requirements and specifications were informal
    - Engineers rejected formal spec’n languages (e.g., ours)
- Narrow notion of formal verification
    - Didn’t contribute to traditional processes (e.g., testing)
    - Didn’t fit the flow
    - Didn’t reduce costs or time (e.g., by early fault detection)
    - It was “all or nothing”
- Lack of automation
    - Couldn’t mechanize the huge search effectively
    - So needed human guidance—and interactive theorem proving is an arcane skill, and not everyone finds it fun!

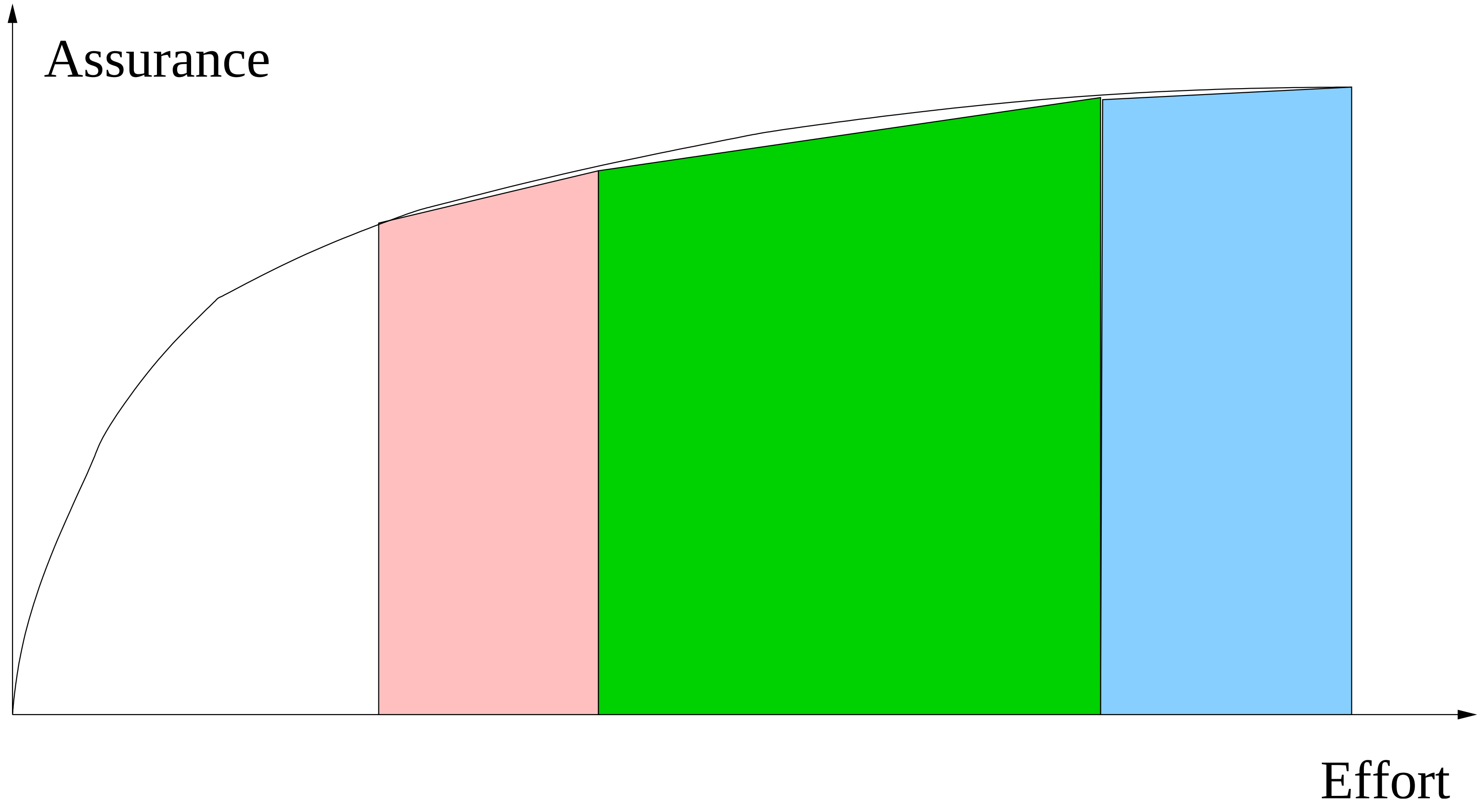
Formal Verification by Theorem Proving: The Wall



Newer Technologies Have Reduced The Wall Somewhat



But Effort/Assurance Relationship Is Not Linear



The interesting area is the blank spot to the left

The Opportunity

A convergence of three trends

- Industrial adoption of model-based devel’t environments
  - Use a model of the system (and its environment) as the focus for all design and development activities
  - E.g., Simulink/Stateflow, SCADE and Esterel, UML
  - Some are ideal for FM (others not, but can make do)
- New kinds of formal activities
  - Fault tree analysis, test case generation, extended static checking (ESC), formal exploration, runtime verification, environment synthesis, controller synthesis
- More powerful, more automated deductive techniques
  - Approaches based on “little engines of proof”
  - New engines: commodity SAT, Multi-Shostak
  - New techniques: (infinite) bounded model checking

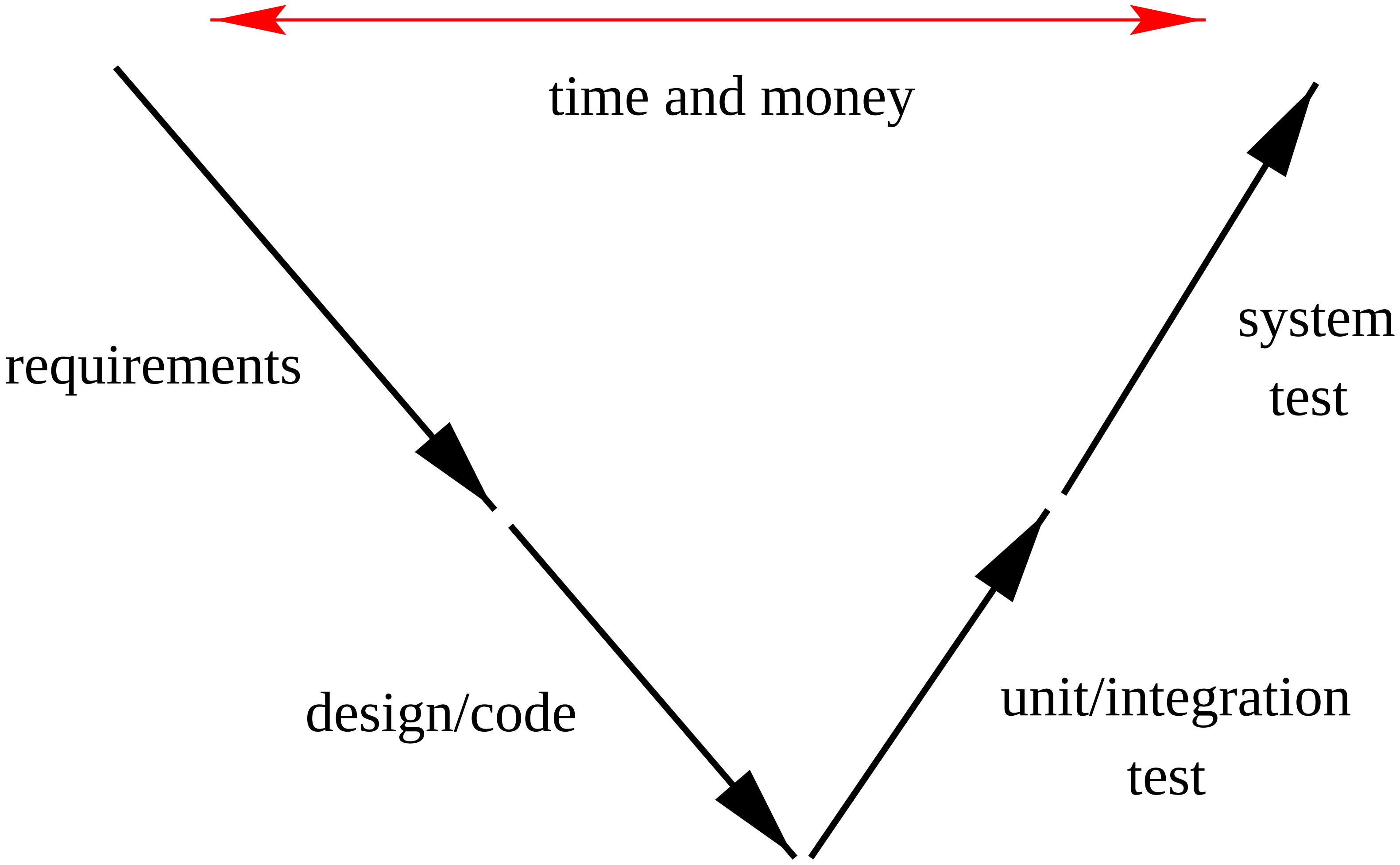
Industrial Adoption of Model-Based Development

- Give access to formal descriptions throughout the lifecycle
- Being adopted at a surprisingly rapid pace
- A380 (SCADE), 7E7 (TBD) software developed this way
- 550,000 Matlab licenses; how many UML?
- It was Ford that induced Mathworks to develop Stateflow
  - Appears to have complex semantics, but we have a surprisingly simple formalization
- Not just embedded systems
  - “Business logic”
  - System C and System Verilog: projections of 50,000 block designers, and 500,000 who assemble blocks
- Now, we just need to add analysis

New Kinds of Formal Analyses and Activities

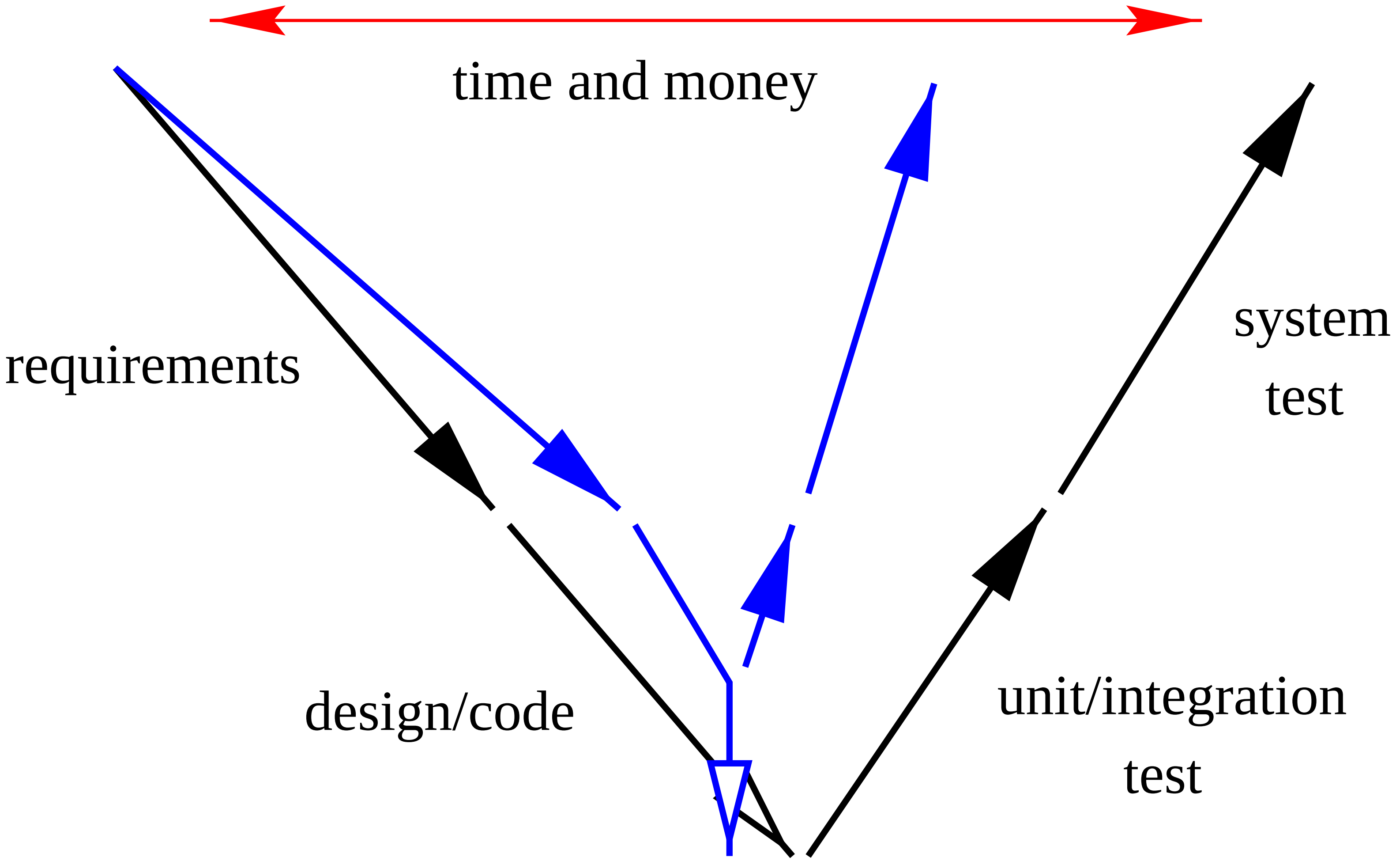
- Support design exploration in the early lifecycle
  - “Can this state and that both be active simultaneously?”
  - “Show me an input sequence that can get me to here with $x > y$”
- Provide feedback and assurance in the early lifecycle
  - Extended static checking
    - ⋆ E.g., Spark Examiner
  - Reachability analysis (for hybrid and infinite-state as well as discrete systems)
- Automate costly and error-prone manual processes
  - E.g., test case generation
- Together, these can provide a radical improvement in the traditional “V”; FM disappears inside traditional processes

Simplified Vee Diagram



Automated formal analysis can tighten the vee

Tightened Vee Diagram



time and money

requirements

design/code

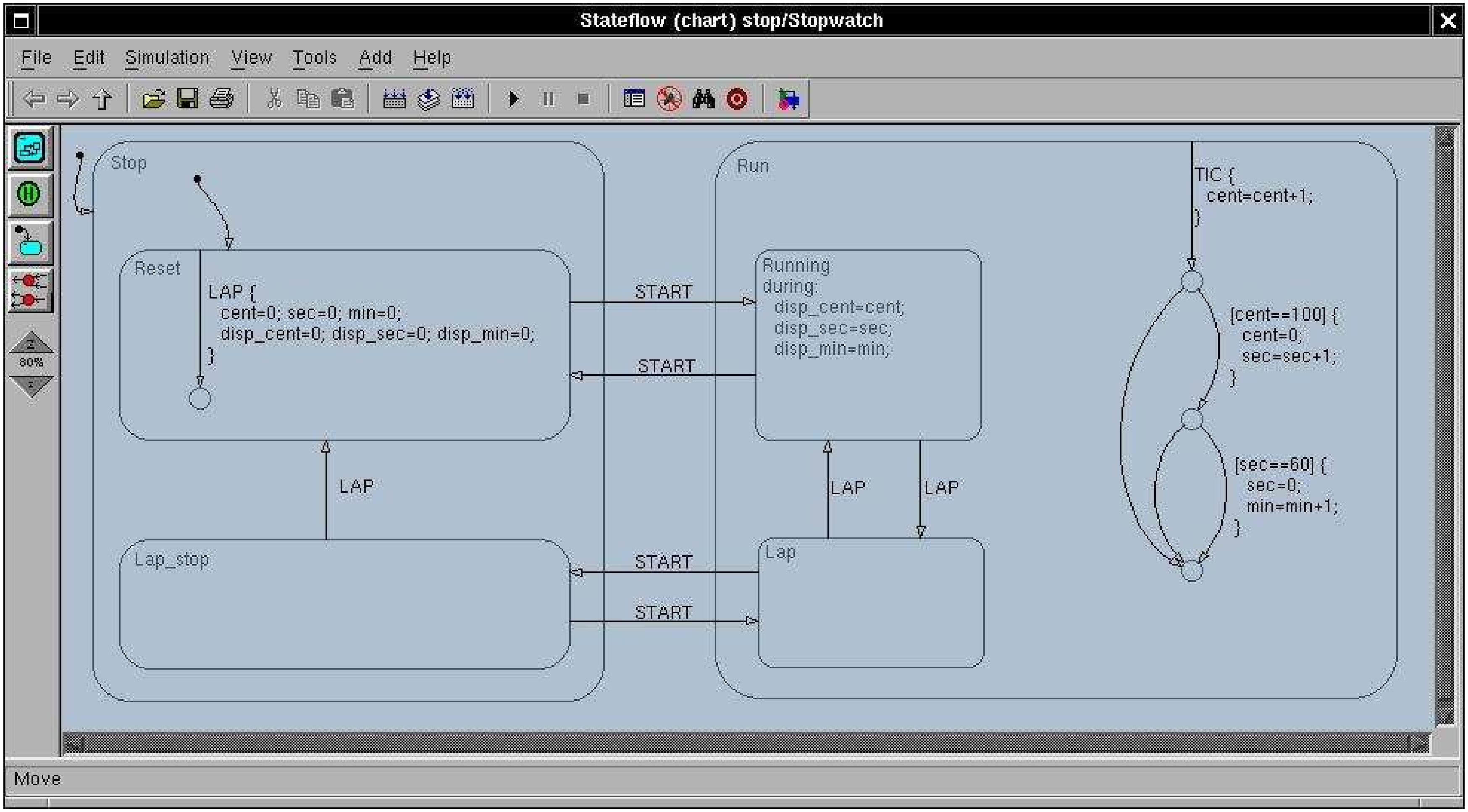
system

test

unit/integration

test

Example: Stateflow



Statechart and flowchart notation of Matlab/Simulink

Analyzing Stateflow

- First, we need a formal semantics
- Stateflow is superficially similar to other statecharts notations
- But is actually quite different
- It's a purely sequential programming language
- Semantics needs to explicate this
- We do this with an SOS semantics
  - Hamon and Rushby, FASE '04
- Have a static analyzer to enforce restrictions and guidelines
  - E.g., no dependence on 12 o'clock rule
- Now suppose we want a test case to reach a particular state
  - Need to solve the path predicates that get us there
- Both require deciding/constraint satisfaction for propositional combinations of arithmetic expressions

Little Engines of Proof (LEP)

- In contrast to one size fits all uniform proof procedures (e.g., resolution), LEP focuses on efficient solutions to important cases, and making them work together
- In the early lifecycle we have cts quantities (real numbers and their derivatives), integers, other infinite and rich domains
- Later in the lifecycle, we have bounded integers, bitvectors, abstract data types
- Several of these theories are decidable, such as
  - Real closed fields
  - Integer linear arithmetic
  - Equality with uninterpreted functions
  - Fixed-width bitvectors
- Challenge is: decide their combination and to do it efficiently

Decision Procedures

- Tell whether a formula is inconsistent, satisfiable, or valid
- Or whether one formula is a consequence of others
  - ○ E.g., does $4 \times x = 2$ follow from $x \leq y$, $x \leq 1 - y$, and $2 \times x \geq 1$ when the variables range over the reals?

  Can use heuristics for speed, but must always terminate and give the correct answer

- Most interesting formulas involve several theories
  - ○ E.g., does

    $$f(cons(4 \times car(x) - 2 \times f(cdr(x)), y)) = f(cons(6 \times cdr(x), y))$$

    follow from $2 \times car(x) - 3 \times cdr(x) = f(cdr(x))$?

  Requires the theories of uninterpreted functions, linear arithmetic, and lists simultaneously

Deciding Combinations Of Theories

- We want methods for deciding combinations of theories that are modular (combine individual decision procedures), integrated (share state for efficiency), and sound
- Need to make some compromises
  - The combination of quantified integer linear arithmetic with equality over uninterpreted functions is undecidable

  But the ground (unquantified) combination is decidable
- Our method (Shostak) works for theories that are canonizable and solvable
  - Most theories of practical concern
  - Others can be integrated using the slower method of Nelson-Oppen
  - Or by a new insight that relaxes solvability

Shostak’s Method

- Yields a modular, integrated, sound decision procedure for the combined theories
  - Invented at SRI more than 20 years ago
  - Developed continuously since then
  - First correct treatment published in 2002
  - Correctness has been formally verified in PVS
  - Previous/other treatments are incomplete, nonterminating, don’t work properly for more than two theories
- Combination of canonizers is a canonizer for the combination
  - Independently useful—e.g., for compiler optimizations
  - Assert path predicates leading to two expressions; expressions are equal if they canonize to identical forms

Deciding Combinations Of Theories

Including Propositional Calculus

- So far, can tell whether one formula follows from several others—satisfiability for a conjunction of literals
- What if we have richer propositional structure
  - E.g., $x < y \wedge (f(x) = y \vee 2 * g(y) < \epsilon) \vee \ldots$ for 1000s of terms
- Should exploit search strategies of modern SAT solvers
- So replace the terms by propositional variables
- Get a solution from a SAT solver (if none, we are done)
- Restore the interpretation of variables and send the conjunction to the core decision procedure
- If satisfiable, we are done
- If not, ask SAT solver for a new assignment—but isn't it expensive to keep doing this?

Deciding Combinations Of Theories Including Propositional Calculus (ctd.)

- Yes, so first, do a little bit of work to find fragments that explain the unsatisfiability, and send these back to the SAT solver as additional constraints (i.e., lemmas)
- Iterate to termination
- We call this “lemmas on demand” or “lazy theorem proving”
- Example, given integer x: $(x < 3 \wedge 2x \geq 5) \vee x = 4$
  - ○ Becomes $(p \wedge q) \vee r$
  - ○ SAT solver suggests $p = T, q = T, r = ?$
  - ○ Ask decision procedure about $x < 3 \wedge 2x \geq 5$, it says No!
  - ○ Add lemma $\neg(p \wedge q)$ to SAT problem
  - ○ SAT solver then suggests $r = T$
  - ○ Interpret as $x = 4$ and we are done
- It works really well

ICS: Integrated Canonizer/Solver

- ICS is our implementation of everything just described
  - ○ And some things not described: proof objects, rich API

  ICS decides the combination of unquantified integer and real linear arithmetic, bitvectors, equality with uninterpreted functions, arrays, tuples, coproducts, recursive datatypes (e.g., lists and trees), and propositional calculus

  - ○ Linear arithmetic solver uses a fast new method
- Its SAT solver is specially engineered for this application
  - ○ Large gains over loose combination with commodity SAT solver

  Benchmarking confirms ICS is competitive as a SAT solver, orders of magnitude faster than other decision procedures
- Accessed as a C library, can be called from virtually any language, also has an interactive ascii front end

ICS (continued)

- Developed under RedHat Linux, but ported to Solaris, MAC OS X, and to Cygwin (for Windows)
- Discharges tens of thousand ESC-type problems per second
- Can be used instead of legacy decision procedures in PVS
- Used in SAL (see later)
- Free for noncommercial purposes under license to SRI
- Visit **`ics.csl.sri.com`** or **`ICanSolve.com`**
- Plans include integer completeness, nonlinear arithmetic, quantifier elimination, definition expansion
  - And more builtin glue logic
- Anything previously done with a SAT solver (e.g., diagnosis, planning, controller synthesis) can be done better with ICS

Bounded Model Checking

- A key technology that finds many applications in tightening the Vee is bounded model checking (BMC)
- Is there a counterexample to this property of length k?
- Same method generates structural test cases
  - Counterexample to “there’s no execution that takes this path”

  And can be used for exploration
- Try $k = 1, 2, \ldots 100 \ldots$ until you find a bug or run out of resources or patience

Bounded Model Checking (ctd.)

- Given a system specified by initiality predicate I and transition relation T on states S, there is a counterexample of length k to invariant P if there is a sequence of states $s_0, \ldots, s_k$ such that

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$$

- Given a Boolean encoding of I and T (i.e., a circuit), this is a propositional satisfiability (SAT) problem
- Needs less tinkering than BDD-based symbolic model checking, can handle bigger systems, find deeper bugs
- Now widely used in hardware verification
  - Though they generally use several methods in cascade

Infinite BMC

- Suppose T is not a circuit, but software, or a high-level specification
- It'll be defined over reals, integers, arrays, datatypes, with function symbols, constants, equalities, inequalities etc.
- So we need to solve the BMC satisfiability problem

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$$

  over these theories

- Typical example
  - ○ T has 1,770 variables, formula is 4,000 lines of text
  - ○ Want to do BMC to depth 40
- Hey! That's exactly what ICS does

Infinite and Finite BMC

- Later lifecycle products replace infinite integers by representations as fixed width bitvectors, etc.
- Can encode some of these datatypes in pure SAT
  - E.g., bitvectors as array of booleans
- And give SAT-level implementations of operations on them
  - E.g., hardware-like adders, shifters
- Exponentially less efficient than ICS decision procedures on some things (e.g., barrel shifter)
- But more efficient on others
- Exact tradeoffs are fuzzy, and some applications already split things up (e.g., into arrays) before sending to ICS
- So we're providing a "dial" that determines how much of the analysis for finite types is handled by decision procedures and how much by SAT

SAL: Symbolic Analysis Laboratory

- SAL is our system for analyzing state machines
- Civilized (intermediate) language, similar to PVS
  - Parameterized modules, subtypes etc.
  - Specialized to transition systems
  - Both guarded commands and SMV-like assignments
  - Synchronous and asynchronous composition
  - Orthogonal assertion languages (currently LTL and CTL)
- State-of-the-art SMC and BMC model checkers for LTL
  - SMC uses CUDD, BMC can use several SAT solvers
- Unique infinite bounded model checker for LTL
  - Can use several decision procedures
- Unique witness model checker (WMC) for CTL
- Pretty good explicit state model checker
- Scriptable (Scheme) interface over powerful API

Application to (Stateflow) Test Case Generation

- Well known that model checkers can generate test cases
  - Set trap variables at targeted states or transitions
  - Model check assertions that trap variables are always false
  - Counterexamples are the test cases
- Generates many short test cases with much redundancy
- Instead, write a 10-line Scheme program on the SAL API that
  - Searches for satisfaction of any trap variable
  - From there searches for sat'n of any other trap variable
  - Iterates previous step, slicing transition relation each time
  - Iterates the whole thing to completion
- Finds a single test case of length 86 for Stateflow specification of shift scheduler for 4-speed automatic transmission that achieves full state and transition coverage

Broader Picture: Verification with SMC

- Symbolic model checkers can verify finite state systems
- And via predicate abstraction (automated by decision procedures), they can verify infinite state systems
- Symbolic model checkers can calculate the reachable stateset
  - 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
- Theorem provers need invariants
- So modify a symbolic 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]

Aside: Property-Preserving Abstractions

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

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

  Where $\hat{T}$ and $\hat{P}$ that are simple to analyze

- A good abstraction typically (for safety properties) introduces nondeterminism while preserving the property

Aside: Calculating an Abstraction

- Transition needed between given pair of abstract states?
- Given abstraction function $\phi : [S \rightarrow \hat{S}]$ we have

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

- We use highly automated (i.e., sound but incomplete) theorem proving to construct the abstracted system:
  - If we include transition iff the formula is proved
  - There's a chance we may fail to prove true formulas
  - This will produce unsound abstractions
- So turn the problem around and calculate when we don't need a transition: omit transition iff the formula is proved

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

- Now theorem-proving failure affects accuracy, not soundness
- We call this “failure tolerant theorem proving”

Aside: Hybrid Abstraction

- A variant on this approach can reduce hybrid systems (e.g., Simulink/Stateflow) to sound discrete abstractions
    - Which are then examined by model checking
- Abstracts polynomials over continuous variables and their first j derivatives to their qualitative signs $\{-, 0, +\}$.
- Computation uses a decision procedure over real closed fields
- The method is complete for linear hybrid systems
- Heuristically effective for others
- Allows computation of reachable states for hybrid systems (e.g., “will these two aircraft ever collide?”)
- Has solved much harder problems than other methods

Broader Picture 2:

Extending (Infinite and Finite) BMC to Verification

- In BMC, we should require that $s_0, \ldots, s_k$ are distinct
    - Otherwise there's a shorter counterexample
- And we should not allow any but s_0 to satisfy I
    - Otherwise there's a shorter counterexample
- If there's no path of length k satisfying these two constraints, and no counterexample has been found of length less than k, then we have verified P
    - By finding its finite diameter
- Seldom works in practice

Alternative: Automated k-Induction

- Ordinary inductive invariance (for P):

  Basis: $I(s_0) \supset P(s_0)$

  Step: $P(r_1) \wedge T(r_1, r_2) \supset P(r_2)$

- Extend to induction of depth k:

  Basis: No counterexample of length k or less

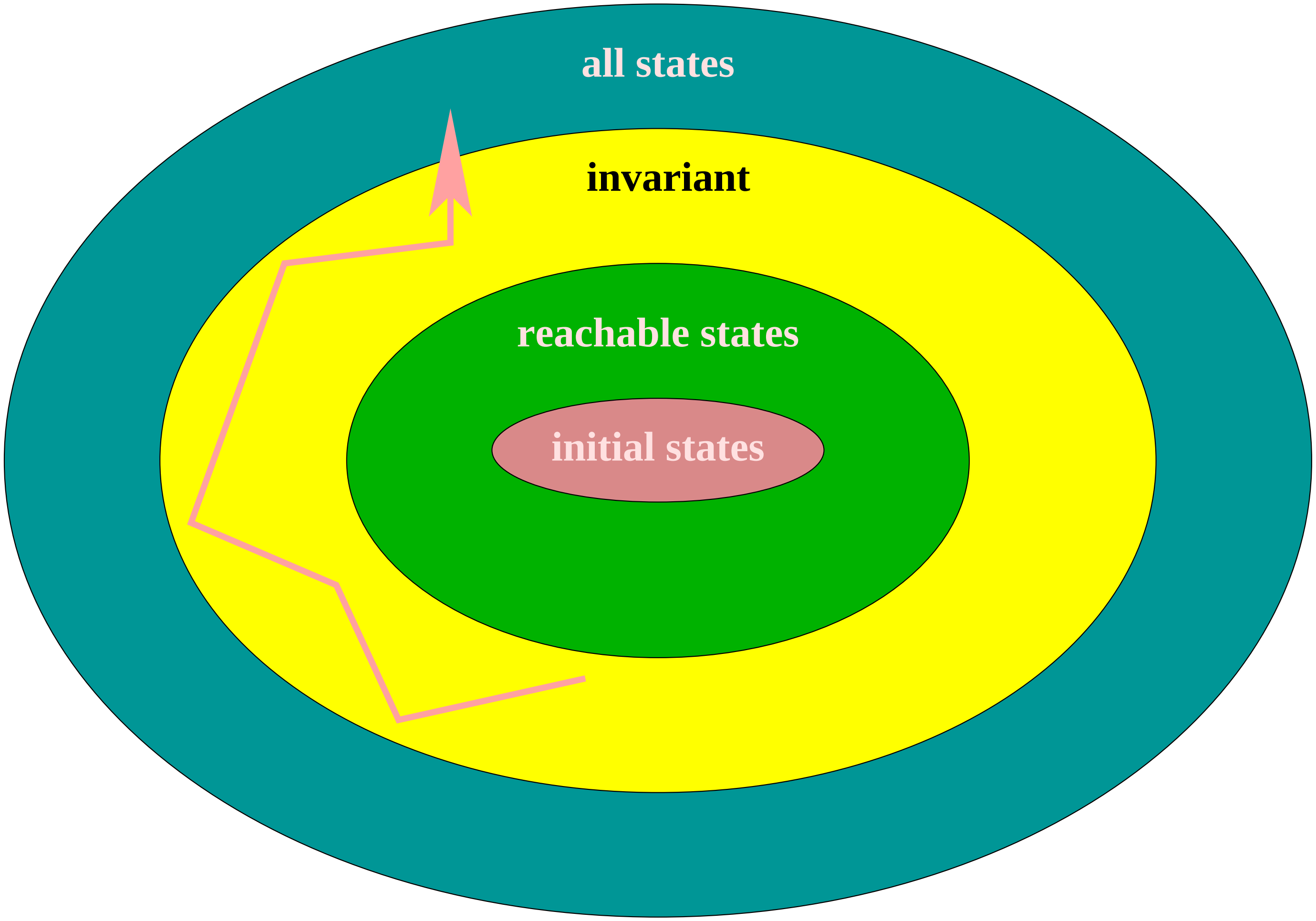
  Step: $P(r_1) \wedge T(r_1, r_2) \wedge P(r_2) \wedge \cdots \wedge P(r_{k-1}) \wedge T(r_{k-1}, r_k) \supset P(r_k)$

  These are close relatives of the BMC formulas

- Induction for $k = 2, 3, 4 \ldots$ may succeed where $k = 1$ does not
- Avoid loops and degenerate cases in the antecedent paths as in BMC
- Method does work in practice and is complete for some problems (e.g., timed automata)

k-Induction is Powerful

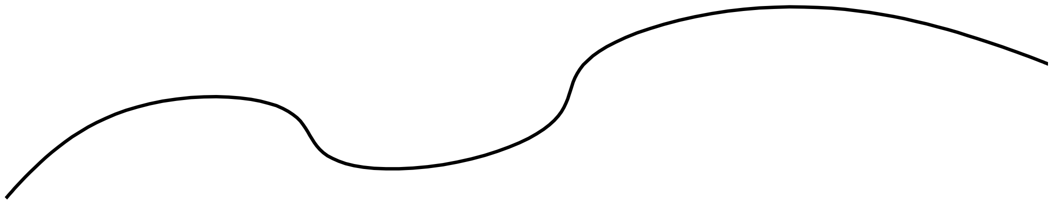
Violations get harder as k grows



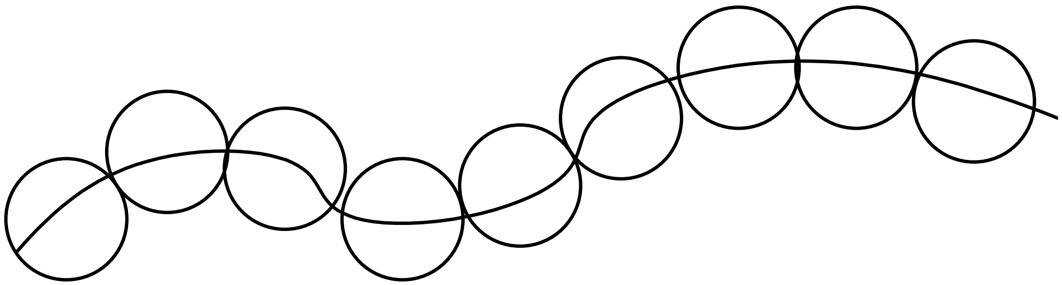
Broader Picture 3:
Integrating BMC With Informal Methods

- With big problems, may be unable to take k far enough for BMC to get to interesting states
- So, instead, start from states found during random simulation
- Can be seen as a way to amplify the power of simulation
- Or to extend its reach

Amplifying The Power Of Simulation



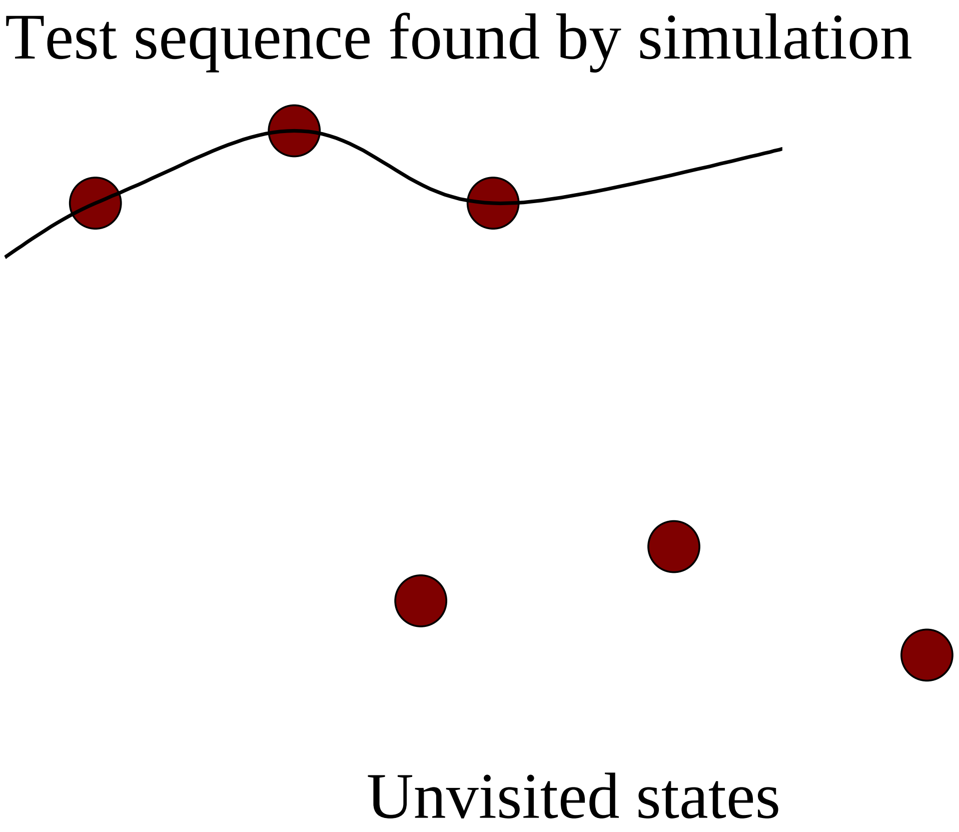
Test sequence found by simulation



Test sequence amplified by bounded model checking

Extending The Reach Of Simulation

Random simulation can have trouble reaching some parts of the state space

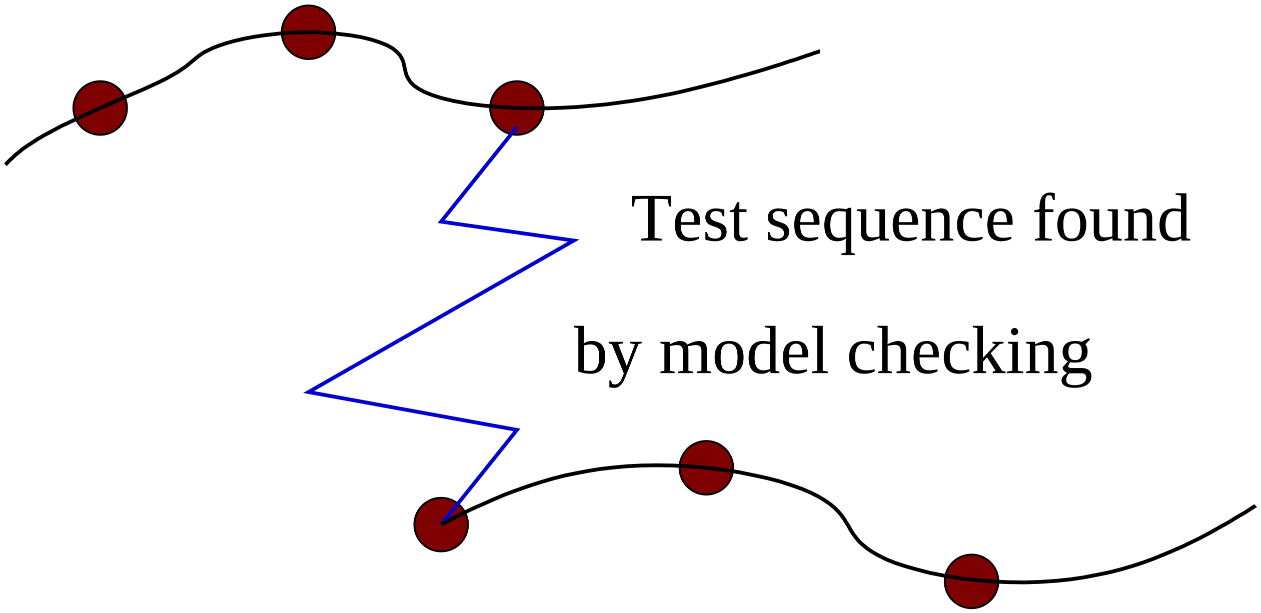


Test sequence found by simulation

Unvisited states

Extending The Reach Of Simulation

So use BMC to jumpstart entry into those parts



Test sequence found by simulation

Test sequence found

by model checking

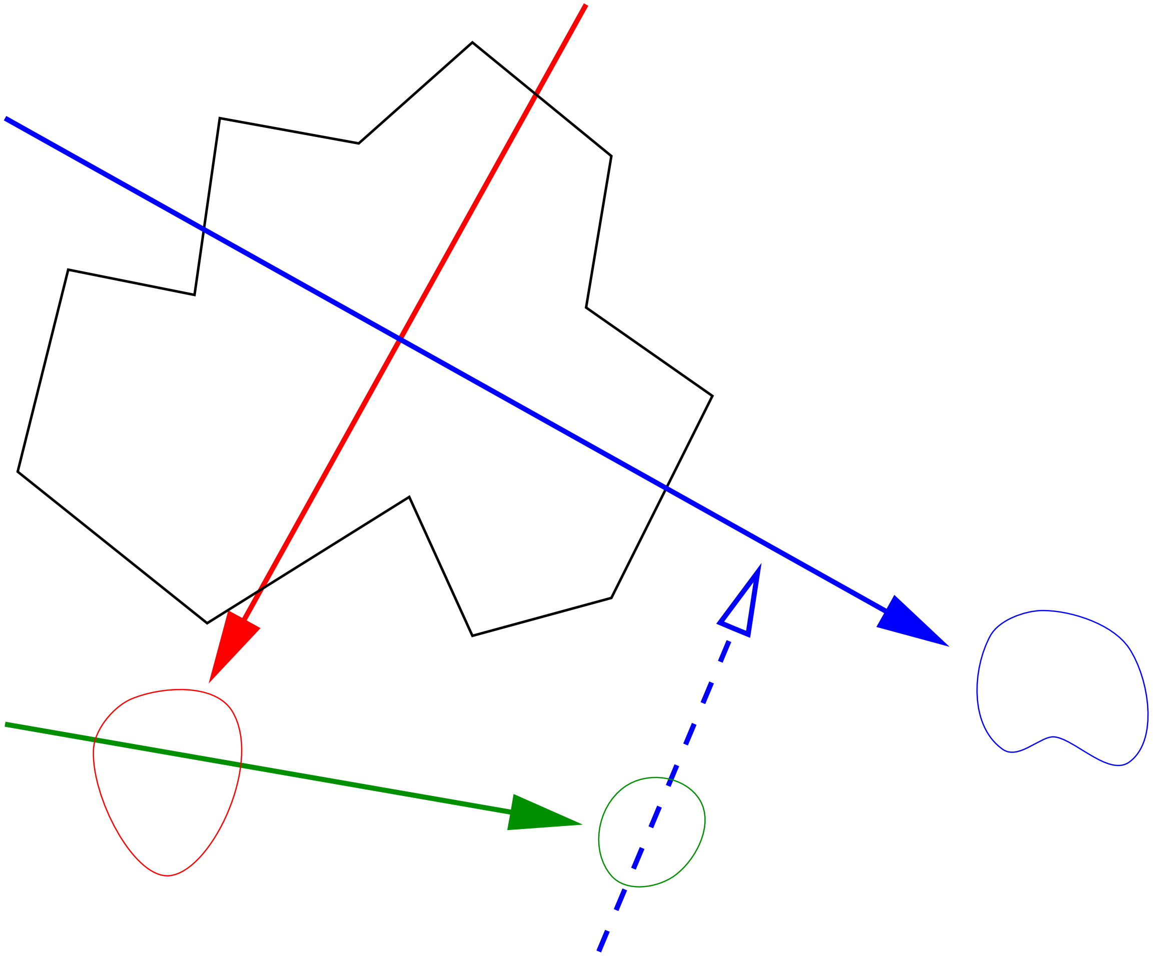
Test sequence continued by simulation

Putting It All Together: Maybe a Paradigm Shift

- It is now fairly routine to have model checkers as backends to theorem provers (e.g., PVS), or proof assistants as front ends to model checkers (e.g., Cadence SMV)
- But we envisage a larger collection of symbolic computational procedures
    - ○ Decision procedures, abstractors, invariant generators, model checkers, static analyzers, test generators, ITPs
- Interacting through a scriptable tool bus
- The bus manages symbolic and concrete artifacts
    - ○ Test cases, abstractions, theorems, invariants

  Over which it performs evidence management
- Focus shifts from verification to symbolic analysis
    - ○ Iterative application of analysis to artifacts to yield new artifacts, insight and evidence

Integrated, Iterated Analysis



Summary: Near-Term Opportunity

- Model-based design methods are a (once-in-a-lifetime?) opportunity to get at formal artifacts early enough in the lifecycle to apply useful analysis within the design loop
- And formal analysis tools are now powerful enough to do useful things without interactive guidance
- The challenge is to find good ways to put these two together
    - Deliver analyses of interest and value to the developers
    - Or certifiers
    - But must fit in their flow

  So can shift from technology push to pull
- Disappearing (or invisible) formal methods is our slogan for this approach: apply formal automation to familiar practices

Summary: Longer-Term Promise

- Symbolic analysis could become the dominant method in systems development and assurance
- And programming could be supplanted by construction of logical models
- And deduction will do the hard work

Summary: Technology

- The technology of automated deduction (and the speed of commodity workstations) has reached a point where we can solve problems of real interest and value to developers of embedded systems

- Embodied in our systems

  PVS.csl.cri.com: comprehensive interactive theorem prover

  ICS.csl.sri.com: embedded decision procedures

  SAL.csl.sri.com: model checking toolkit (explicit, symbolic, bounded, infinite-bounded), and (soon) tool-bus

- And in numerous papers accessible from `http://fm.csl.sri.com`, including our Roadmap

A Bigger Vision: 21st Century Mathematics

- The industrialization of the 19th and 20th century was based on continuous mathematics
  - And its automation
- That of the 21st century will be based on symbolic mathematics
  - Whose automation is now feasible

  Allows analysis of systems too complex and numerically too indeterminate for classical methods
- Example: symbolic systems biology
  - Knockouts in E.Coli (SRI; Maude)
  - Cell differentiation in C.Elegans (Weizmann; Play-in/out)
  - Delta-Notch signaling (SRI, Stanford; Hybrid SAL)
  - Sporolation in B.Subtilis (SRI; Hybrid SAL)