NICTA tutorial, 28 May 2003, UNSW, Sydney Australia, derived from Marktoberdorf Lectures 2002 (combined slides for Lectures 1–3)

Tutorial Introduction
To Mechanized Formal Analysis Using Theorem Proving, Model Checking and Abstraction

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

Some Different Approaches (for safety properties of concurrent systems defined as transition relations)

. . . to be demonstrated on a concrete example

Namely, Lamport's Bakery Algorithm

- Deduction (theorem proving)
- Model checking
- Abstraction and model checking
- Automated abstraction (failure tolerant theorem proving)
- Bounded model checking (for infinite state systems)
- Combined formal/informal methods

Formal Methods: Analogy with Engineering Mathematics

- Engineers in traditional disciplines build mathematical models of their designs
- And use calculation to establish that the design, in the context of a modeled environment, satisfies its requirements
- Only useful when mechanized (e.g., CFD)
- Used in the design loop (exploration, debugging)
    - Model, calculate, interpret, repeat
- Also used in certification
    - Verify by calculation that the modeled system satisfies certain requirements
- Need to be sure that model faithfully represents the design, design is implemented correctly, environment is modeled faithfully, and calculations are performed without error

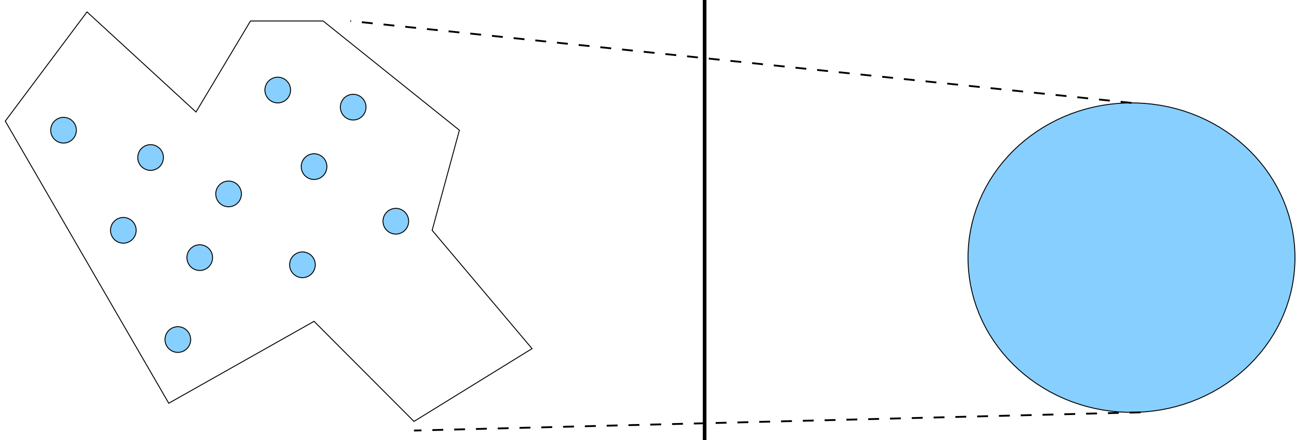
Formal Methods: Analogy with Engineering Math (ctd.)

- Formal methods: same idea, applied to computational systems
- The applied math of Computer Science is formal logic
- So the models are formal descriptions in some logical system
  - E.g., a program reinterpreted as a mathematical formula rather than instructions to a machine
- And calculation is mechanized by automated deduction: theorem proving, model checking, static analysis, etc.
- Formal calculations (can) cover all modeled behaviors
- If the model is accurate, this provides verification
- If the model is approximate, can still be good for debugging (aka. refutation)

Formal Methods: In Pictures

Testing/Simulation

Formal Analysis



Real System

Formal Model

- Partial coverage

- Complete coverage
  (of the modeled system)

  Accurate model: verification

  Approximate model: debugging

Comparison with Simulation, Testing etc.

- Simulation also considers a model of the system (designed for execution rather than analysis)
- Testing considers the real thing
- Both differ from formal methods in that they examine only some of the possible behaviors
- For continuous systems, verification by extrapolation from partial tests is valid, but for discrete systems, it is not
- Can make only statistical projections, and it's expensive
  - 114,000 years on test for 10^{-9}

  Limit to evidence provided by testing is about 10^{-4}
- In most applications, testing is used for debugging rather than verification

Comparison with Simulation, Testing etc. (ctd)

- Debugging depends on choosing right test cases
  - Can be improved by explicit coverage measures
  - Good coverage is almost impossible when the environment can introduce huge numbers of different behaviors (e.g., fault arrivals, real-time, asynchronous interactions)

  So depends on skill, luck
- Since formal methods can consider all behaviors, certain to find the bugs
  - Provided the model, environment, and the properties checked are sufficiently accurate to manifest them

  So depends on skill, luck
- Experience is you find more bugs (and more high-value bugs) by exploring all behaviors of an approximate model than by exploring some behaviors of a more accurate one

Formal Calculations: The Basic Challenge

- Build mathematical model of system and deduce properties by calculation
- The applied math of computer science is formal logic
- So calculation is done by automated deduction
- Where all problems are NP-hard, most are superexponential (2^{2^n}), nonelementary ($2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}\}n$), or undecidable
- Why? Have to search a massive space of discrete possibilities
- Which exactly mirrors why it's so hard to provide assurance for computational systems
- But at least we've reduced the problem to a previously unsolved problem

Let's do an example: Bakery

The Bakery Algorithm for Distributed Mutual Exclusion

- Idea is based on the way people queue for service in US delicatessens and bakeries
- A machine dispenses tickets printed with numbers that increase monotonically
- People who want service take a ticket
- The unserved person with the lowest numbered ticket is served next
  - Safe: at most one person is served (i.e., is in the "critical section") at a time
  - Live: each person is eventually served
- Preserve the idea without centralized ticket dispenser

Informal Protocol Description

- Works for $n \geq 1$ processes
- Each process has a ticket register, initially zero
- When it wants to enter its critical section, a process sets its ticket greater than that of any other process
- Then it waits until its ticket is smaller than that of any other process with a nonzero ticket
- At which point it enters its critical section
- Resets its ticket to zero when it exits its critical section
- Show that at most one process is in its critical section at any time (i.e., mutual exclusion)

Formal Modeling and Analysis

- Build a mathematical model of the protocol
- Analyze it for a desired property
- Must choose how much detail to include in the model
  - Too much detail: analysis may be infeasible
  - Too little detail: analysis may be inaccurate (i.e., fail to detect bugs, or report spurious ones)
  - Must also choose a modeling style that supports intended form of analysis
- Requires judgment (skill, luck) to do this

Modeling the Example System and its Properties: Accuracy and Level of Detail

- The protocol uses shared memory and is sensitive to the atomicity of concurrent reads and writes
- And to the memory model (on multiprocessors with relaxed memory models, reads and writes from different processors may be reordered)
- And to any faults the memory may exhibit
- If we wish to examine the mutual exclusion property of a particular implementation of the protocol, we will need to represent the memory model, fault model, and atomicity employed—which will be quite challenging
- Abstractly (or at first), we may prefer to focus on the behavior of the protocol in an ideal environment with fault-free sequentially consistent atomic memory

Modeling the Example System and its Properties (ctd.)

- Also, although the protocol is suitable for n processes, we may prefer to focus on the important special case $n = 2$
- And although each process will perform activities other than the given protocol, we will abstract these details away and assume each process is in one of three phases

  idle: performing work outside its critical section

  trying: to enter its critical section

  critical: performing work inside its critical section

Formalizing the Model (continued)

- We will need to model a system state comprising
  For each process:
  - The value of its `ticket`, which is a natural number
  - The phase it is in—recorded in its "program counter" which takes values `idle`, `trying`, `critical`
- Then we model the (possibly nondeterministic) transitions in the system state produced by each protocol step
- And check that the desired property is always preserved

A Formal Description of the Protocol (in SAL)

```
bakery : CONTEXT =
BEGIN
   phase : TYPE = {idle, trying, critical};
   ticket: TYPE = NATURAL;

process : MODULE =
BEGIN
 INPUT other_t: ticket
 OUTPUT my_t: ticket
 OUTPUT pc: phase

INITIALIZATION
  pc = idle;
  my_t = 0
```

More Formal Protocol Description (continued)

```
TRANSITION
    [pc = idle -->
          my_t'  = other_t + 1;
          pc' = trying
     []
     pc = trying AND (other_t = 0 OR my_t < other_t) -->
          pc' = critical
     []
     pc = critical -->
          my_t' = 0;
          pc' = idle
    ]
END;
```

More Formal Protocol Description (continued again)

```
P1 : MODULE = RENAME pc TO pc1 IN process;

P2 : MODULE = RENAME other_t TO my_t,
                     my_t TO other_t,
                     pc TO pc2 IN process;

system : MODULE = P1 [] P2;

safety: THEOREM
    system |- G(NOT (pc1 = critical AND pc2 = critical));

END
```

Analyzing the Specification Using Theorem Proving

- This is an infinite state system (because the tickets can grow without bound) so conventional model checking is not directly applicable
- So we'll start with an analysis by theorem proving using PVS
- PVS is a logic, it does not have a notion of state, nor of concurrent programs, built in—we must specify the program using the transition relation semantics of SAL
- Soon, we'll have a SAL to PVS translator to automate this

```
bakery: THEORY
BEGIN
  phase : TYPE = {idle, trying, critical}

  state: TYPE = [# pc1, pc2: phase, t1, t2: nat #]

  s, pre, post: VAR state
```

The Transitions in PVS

```
P1_transition(pre, post): bool =
 IF pre`pc1 = idle
   THEN post = pre WITH [(t1) := pre`t2 + 1, (pc1) := trying]
 ELSIF pre`pc1 = trying AND (pre`t2 = 0 OR pre`t1 < pre`t2)
   THEN post = pre WITH [(pc1) := critical]
 ELSIF pre`pc1 = critical
   THEN post = pre WITH [(t1) := 0, (pc1) := idle]
 ELSE post = pre
 ENDIF

P2_transition(pre, post): bool = ... similar

transitions(pre, post): bool =
  P1_transition(pre, post) OR P2_transition(pre, post)
```

Initialization and Invariant in PVS

```
init(s): bool = s`pc1 = idle AND s`pc2 = idle
                 AND s`t1 = 0 AND s`t2 = 0

safe(s): bool = NOT(s`pc1 = critical AND s`pc2 = critical)

% To prove that a property P is an invariant, we prove it is *inductive*
% This is similar to Amir Pnueli's rule for Universal Invariance
% Except we strengthen the actual property rather than have an auxiliary

indinv(inv: pred[state]): bool =
 FORALL s: init(s) => inv(s)
  AND FORALL pre, post:
     inv(pre) AND transitions(pre, post) => inv(post)

first_try: LEMMA indinv(safe)
```

First Attempted Proof: Step 1

- Starting the PVS theorem prover gives us this sequent

```
first_try :
   |-------
{1}   indinv(safe)
Rule?
```

- The proof commands `(EXPAND "indinv")` and `(GROUND)` open up the definition of `invariant` and split it into cases

- We are then presented with the first of the two cases

```
This yields  2 subgoals:
first_try.1 :
   |-------
{1}   FORALL s: init(s) => safe(s)
```

- This is discharged by the proof command `(GRIND)`, which expands definitions and performs obvious deductions

First Attempted Proof: Step 2

- This completes the proof of first_try.1.

```
first_try.2 :
  |-------
{1} FORALL pre, post:
      safe(pre) AND transitions(pre, post) => safe(post)
```

- The commands (SKOSIMP), (EXPAND "transitions"), and (GROUND) eliminate the quantification and split transitions into separate cases for processes 1 and 2

```
first_try.2.1 :
{-1}  P1_transition(pre!1, post!1)
[-2]  safe(pre!1)
  |-------
[1]   safe(post!1)
```

- (EXPAND "P1_transition") and (BDDSIMP) split the proof into four cases according to the kind of step made by the process

First Attempted Proof: Step 3

- The first one is discharged by `(GRIND)`, but the second is not and we are presented with the sequent

```
first_try.2.1.2 :
[-1]  pre!1`t2 = 0
{-2}  trying?(pre!1`pc1)
[-3]  post!1 = pre!1 WITH [(pc1) := critical]
[-4]  safe(pre!1)
{-5}  critical?(pre!1`pc2)
  |-------
```

- When there are no formulas below the line, a sequent is true if there is a contradiction among those above the line
- Here, Process 1 enters its critical section because Process 2's ticket is zero—but Process 2 is already in its critical section

First Attempted Proof: Aha!

- This is a contradiction because Process 2 must have incremented its ticket (making it nonzero) when it entered its `trying` phase
- But contemplation, or experimentation with the prover, should convince you that this fact is not provable from the information provided
- Similarly for the other unprovable subgoals
- The problem is not that `safe` is untrue, but that it is not inductive
  - It does not provide a strong enough antecedent to support the proof of its own invariance

Second Attempted Proof

- The solution is to prove a stronger property than the one we are really interested in

```
strong_safe(s): bool = safe(s)
  AND (s`t1 = 0 => s`pc1 = idle)
  AND (s`t2 = 0 => s`pc2 = idle)

second_try: LEMMA indinv(strong_safe)
```

Second Attempted Proof: Aha! Again

- The stronger formula deals with the case we just examined, and the symmetric case for Process 2, but still has two unproved subgoals; here's one of them

```
second_try.2 :

{-1}  trying?(pre!1`pc1)
{-2}  pre!1`t1 < pre!1`t2
{-3}  post!1 = pre!1 WITH [(pc1) := critical]
{-4}  critical?(pre!1`pc2)
  |-------
{1}   (pre!1`t1 = 0)
```

- The situation here is that Process 1 has a smaller ticket than Process 2 and is entering its critical section—even though Process 2 is already in its critical section
- But this cannot happen because Process 2 must have had the smaller ticket when it entered (because Process 1 has a nonzero ticket), contradicting formula -2

Third Attempted Proof

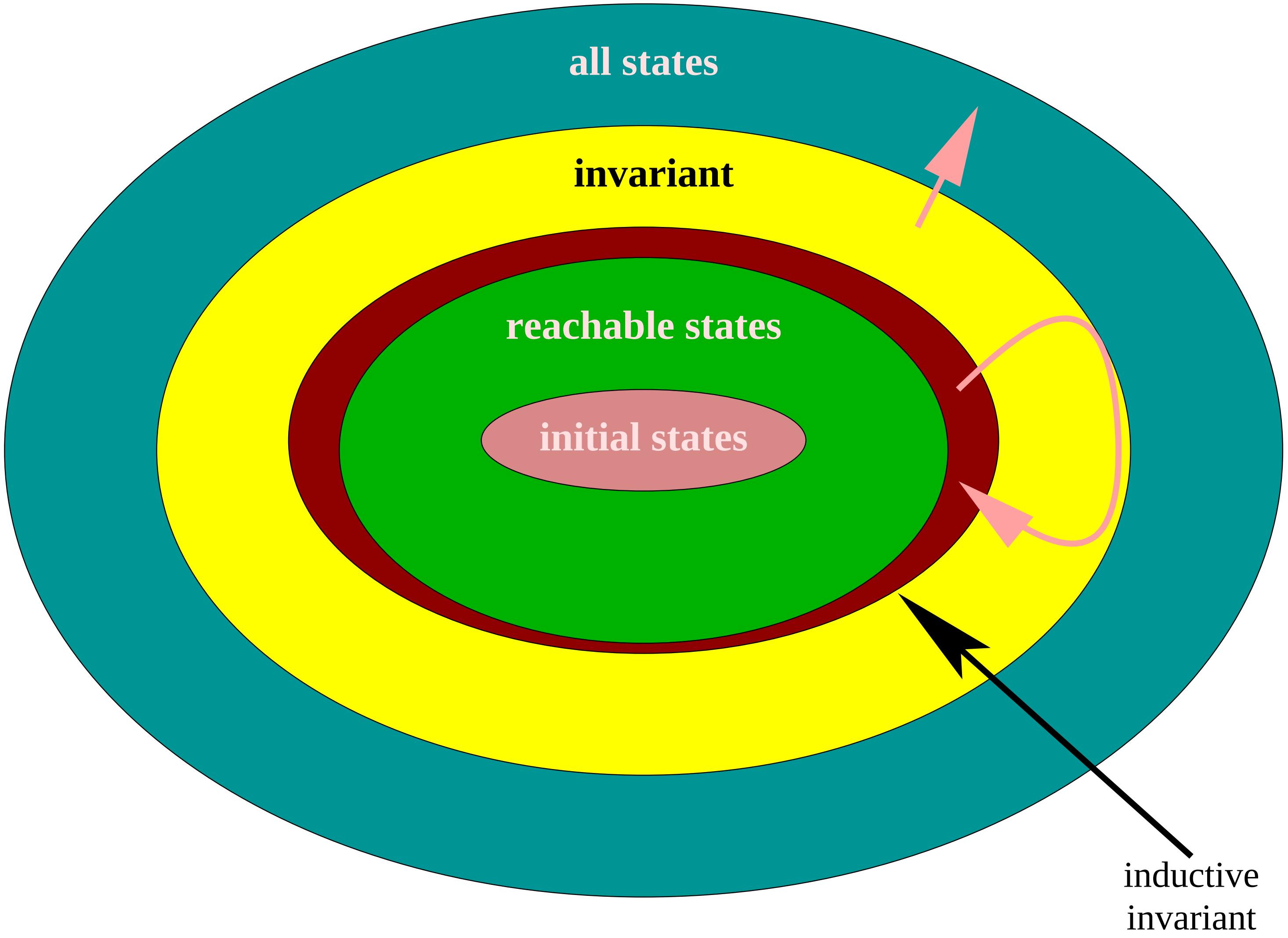
- Again we need to strengthen the invariant to carry along this fact
- ```
  inductive_safe(s):bool = strong_safe(s)
   AND ((s`pc1 = critical AND s`pc2 = trying) => s`t1 < s`t2)
   AND ((s`pc2 = critical AND s`pc1 = trying) => s`t1 > s`t2)

  third_try: LEMMA indinv(inductive_safe)
  ```
- Finally, we have a invariant that is inductive—and is proved with `(GRIND)`
  ```

Inductive Invariants

- To establish an invariant or safety property S (one true of all reachable states) by theorem proving, we invent another property P that implies S and that is inductive (on transition relation T, with initial states I)
  - Includes all the initial states: $I(s) \supset P(s)$
  - Is closed on the transitions: $P(s) \wedge T(s,t) \supset P(t)$
- The reachable states are the smallest set that is inductive, so inductive properties are invariants
- Trouble is, naturally stated invariants are seldom inductive
  - The second condition is violated
- Need to make them smaller (stronger) to exclude the states that take you outside the invariant

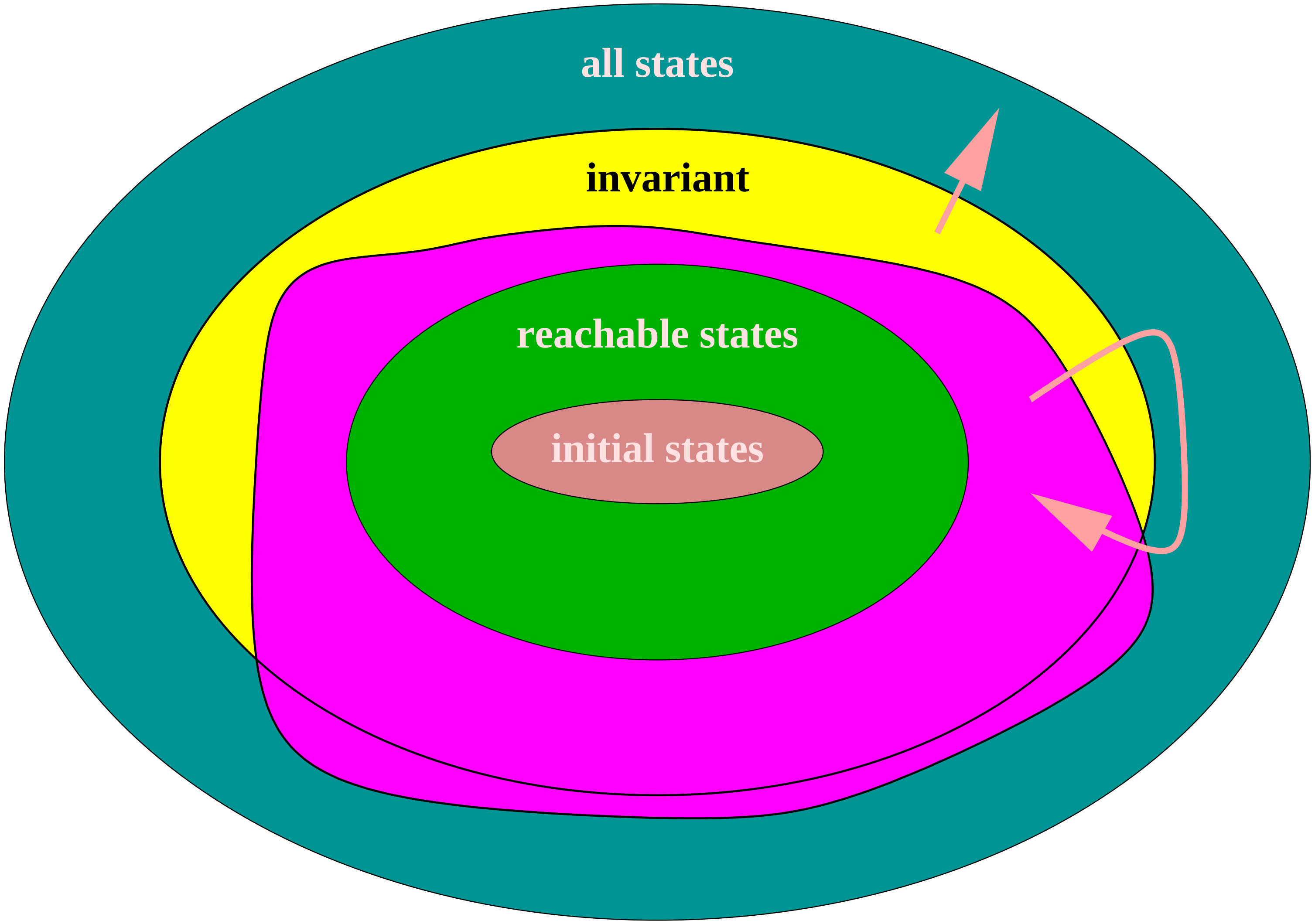
Noninductive Invariants In Pictures



Strengthening Invariants To Make Them Inductive

- Postulate a new invariant that excludes the states (so far discovered) that take you outside the desired invariant
- Show that the conjunction of the new and the desired invariant is inductive
- Iterate until success or exasperation

Inductive Invariants In Pictures



Strengthening Invariants To Make Them Inductive

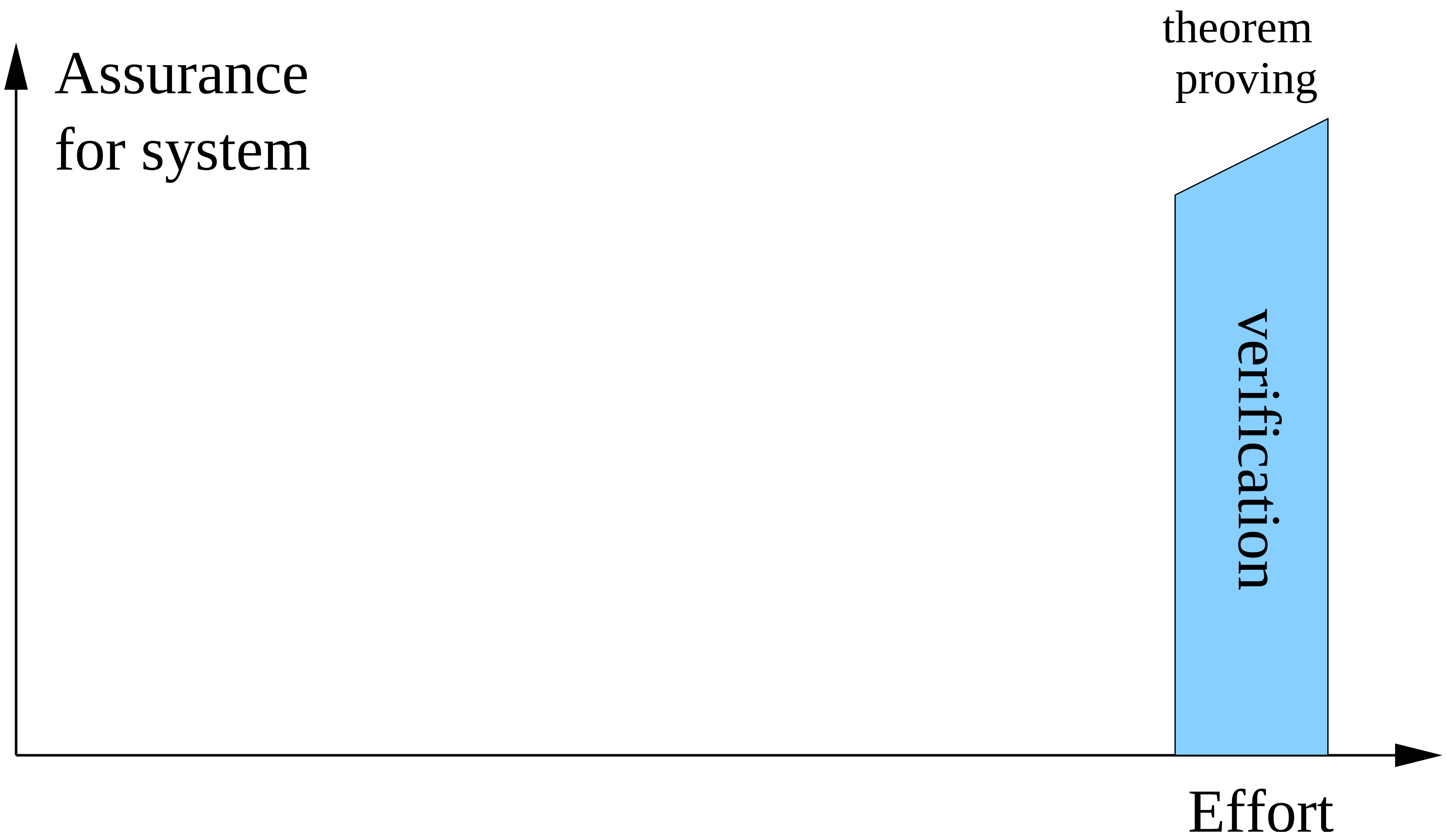
- Iterate until success or exasperation
- Process can be made systematic
  - Each strengthening was suggested by a failed proof

  But is always tedious
- Bounded retransmission protocol required 57 such iterations
  - Took a couple of months to complete
    (Havelund and Shankar)
- Notice that each conjunct must itself be an invariant
  (the very property we are trying to establish)

Pros and Cons of Theorem Proving

- Theorem proving can handle infinite state systems
- And accurate models
  - Sometimes less says more–e.g., fault tolerance
- And general properties (not just those expressible in temporal logic)
- But it's hard (and not everyone finds it fun)
  - Everything is possible but nothing is easy
  - Especially strengthening of invariants
- Interaction focuses on proof, and idiosyncrasies of the prover, not on the design being evaluated
  - "Interactive theorem proving is a waste of human talent"
- It's all or nothing
  - No incremental return for incremental effort

Formal Verification by Theorem Proving: The Wall



Minor Theme: Design Choices in PVS

- Aside from the need to strengthen the invariant, PVS did OK on this example (and does so on many more)
- We only used a fraction of its linguistic resources
    - Higher-order logic with dependent predicate subtyping
    - Recursive abstract data types and inductive types
    - Parameterized theories and interpretations
    - . . . and most of it is efficiently executable
- It automatically discharged subgoals by deciding properties over abstract data types (enumeration types are a degenerate case), integer arithmetic, record updates, prop'nl calculus
- In larger examples, it also has to choose when to open up a definition, and when to apply a rewrite
- What makes PVS (and other verification systems) effective is that it has tightly integrated automation for all of these

Decision Procedures

Many important theories are decidable

- Propositional calculus

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

- Equality with uninterpreted function symbols

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

- Function, record, and tuple updates

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

- Linear Arithmetic (over integers and rationals)

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

But we need to decide combinations of theories

$$2 \times car(x) - 3 \times cdr(x) = f(cdr(x)) \supset$$
$$f(cons(4 \times car(x) - 2 \times f(cdr(x)), y)) = f(cons(6 \times cdr(x), y))$$

Combined Decision Procedures

- Some combinations of decidable theories are not decidable
  - E.g., quantified theory of integer arithmetic (Presburger) and equality over uninterpreted function symbols
- Need to make pragmatic compromises
  - E.g., stick to ground (unquantified) theories and leave quantification to heuristics at a higher level
- Two basic methods for combining decision procedures
  - Nelson-Oppen: fewest restrictions
  - Shostak: faster, yields a canonizer for combined theory
- Shostak's method is used in PVS
  - Over 20 years from first paper to fully correct treatment
  - Now formally verified (in PVS)
    by Jonathan Ford, who is Australian

Integrated Decision Procedures

- It's not enough to have good decision procedures available to discharge the leaves of a proof
- They need to be used in simplification, which involves recursive examination of subformulas: repeatedly adding, subtracting, asserting, and denying subformulas
- And integrated with rewriting, where they used in matching and (recursively) to decide conditions or top-level if-then-else's
- So the API to the decision procedures must be quite rich
- We make such a set of decision procedures available for use in other tools: ICS the Integrated Canonizer and Solver (`www.ICanSolve.com`), developed by Harald Rueß

Top-Level Design Choices in PVS

- Specification language is a higher-order logic with subtyping
  - Typechecking is undecidable: uses theorem proving
- User supplies top-level strategic guidance to the prover
  - Invoking appropriate proof methods (induction etc.)
  - Identifying necessary lemmas
  - Suggesting case-splits
  - Recovering when automation fails
- Automation takes care of the details, through a hierarchy of techniques
  1. Decision procedures
  2. Rewriting (automates application of lemmas)
  3. Heuristics (guess at case-splits, instantiations)
  4. User-supplied strategies (cf. tactics in HOL)

Enough of theorem proving, let's do some model checking!

Analyzing the Specification Using Model Checking

- We'll use SALenv1 (developed by Leonardo de Moura)
  - An explicit-state LTL model checker for SAL

  Later, we'll look at symbolic and bounded model checking with SALenv2
- In general, explicit state model checking requires
  - A finite state space
  - A property expressed in a suitable (temporal) logic
- Then uses brute-force search over the state space to establish that the transition system is a (Kripke) model of the property
  - It's automatic
  - And can provide a counterexample when a bug is found

  For invariants, think of it as a simulator that saves all the states it encounters and backtracks until it can find no more; check the property at each state found

Making the Specification Suitable for Model Checking

- We need to make the state space finite
- Use drastic simplification ("downscaling")
  - We've already done this to some extent, by fixing the number of processors, n, as 2
  - We also need to set an upper bound on the tickets
- We'll start at 3, then raise the limit to 4, 5, ... until the search becomes too slow
- We have to modify the protocol to bound the tickets
  - So it's not the same protocol
  - May miss some bugs, or get spurious ones
  - But it's a useful check

The Bounded Specification in SAL

```
bakery : CONTEXT =
BEGIN
   phase : TYPE = {idle, trying, critical};
   max: NATURAL = 3;
   ticket: TYPE = [0..max];

process : MODULE =
BEGIN
 INPUT other_t: ticket
 OUTPUT my_t: ticket
 OUTPUT pc: phase

INITIALIZATION
  pc = idle;
  my_t = 0
```

The Bounded Specification in SAL (continued)

```
TRANSITION
    [pc = idle AND other_t < max -->
          my_t'  = other_t + 1;
          pc' = trying
     []
     pc = trying AND (other_t = 0 OR my_t < other_t) -->
          pc' = critical
     []
     pc = critical -->
          my_t' = 0;
          pc' = idle
    ]
END;
```

The Bounded Specification in SAL (continued again)

```
P1 : MODULE = RENAME pc TO pc1 IN process;

P2 : MODULE = RENAME other_t TO my_t,
                     my_t TO other_t,
                     pc TO pc2 IN process;

system : MODULE = P1 [] P2;

safety: THEOREM
    system |- G(NOT (pc1 = critical AND pc2 = critical));

END
```

Results of Model Checking

- SALenv reports
```
/tmp/gencode.XXCPCE6e.scm:
Verifier "checker" was generated with success.
 Checking...
verified
Number of visited states = 21
Maximum depth = 8
```
- Sometimes properties are true for the wrong reason
- It is prudent to introduce a bug and make sure it is detected before declaring victory
- Here, if we remove the +1 adjustment to the tickets, we get
```
Checking...
Counter-example detected:
... next slide
```

The Counterexample

```
my_t = 0
pc1 = idle
other_t = 0
pc2 = idle
----------------
pc1 = trying
----------------
pc1 = critical
----------------
pc2 = trying
----------------
pc2 = critical
----------------
my_t = 0
pc1 = idle
other_t = 0
pc2 = critical
----------------
Number of visited states = 9
Maximum depth = 4
```

Another Check

- We can check that the counters are capable of increasing indefinitely by adding the invariant
  `unbounded: THEOREM system |- G(my_t < max);`
  (After undoing the deliberate errors just introduced)
- `Checking...`

  `Counter-example detected:`

  `... omitted`

  `Number of visited states = 15`

  `Maximum depth = 5`
- The pattern is: `P1 tries`, then the following sequence repeats
  `P1 enters`, `P2 tries`, `P1 quits`, `P2 enters`, `P1 tries`, `P2 quits`

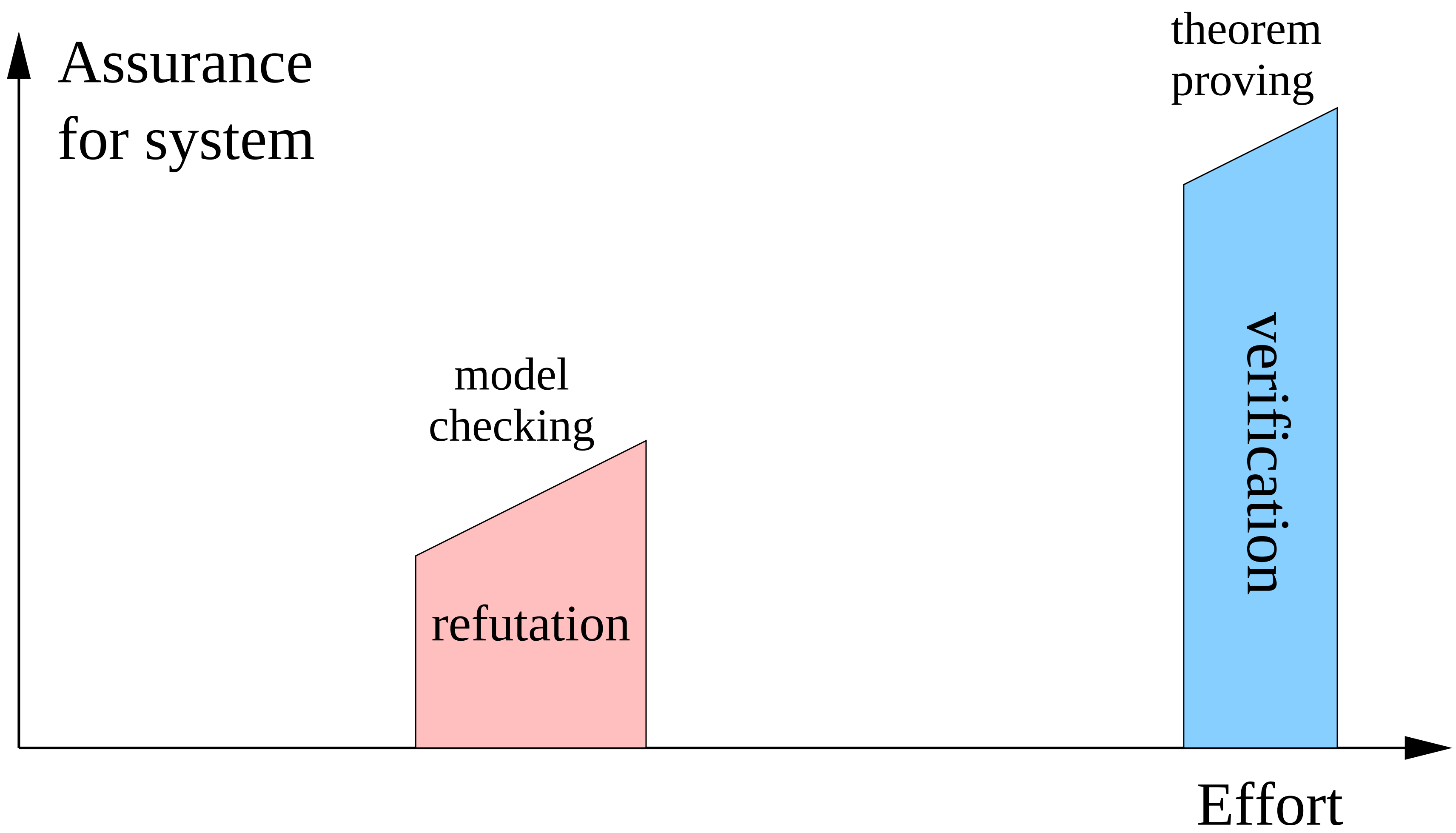
Model Checking the Original Specification

- SALenv can model check the original, infinite-state specification directly, by bounding the search depth
  - Can also draw a picture of the statespace to some depth
- This is sometimes simpler than downscaling
- But is similarly crude
- Let's see it

Pros and Cons of Model Checking

- "*Model checking saved the reputation of formal methods*"
- But have to be explicit where we may prefer not to be
  - E.g., have to specify the ALU (Arithmetic Logic Unit) when we're really only interested in the pipeline logic
- Usually have to downscale the model—can be a lot of work
- Often good at finding bugs, but what if no bugs detected?
  - Have we achieved verification, or just got too crude a model or property?
- Sometimes it's possible to prove that a small model is a property-preserving abstraction of a large, accurate one
- Then not detecting a bug is equivalent to verification

Model Checking: An Island



Minor Theme: Design Choices in SAL

- SAL is intended as an intermediate language to facilitate integration of multiple tools: a Symbolic Analysis Laboratory
- Specialized to state machines modeled as transition relations: you can apply more automation if you know what you are dealing with
- It's defined by an XML DTD
  - Translators provide frontends for other languages
  - And backends to analysis tools
- SALenv 1 is an API for explicit state exploration in SAL
  - Makes it easy to code various model checkers and related tools (each in about 20 lines of Scheme code)

Explicit State Model Checking

Complementary to symbolic model checking

- Can only explore a few million states, but that's enough when there are plenty of bugs to find
- Can use hashing (supertrace) to go further
- LTL is handled via Büchi automata
- Language can include any datatypes and operations supported by the API (not just those that can be easily encoded in BDDs)
- Breadth first search finds short counterexamples
- Can write special search strategies to target specific cases, or to ignore others (symmetry, partial order)
- Can evaluate functions, not just predicates, on the reachable states: can calculate worst-cases, do optimization

But If You Want To See Symbolic Model Checking...

- SALenv 2 compiles finite SAL down to a Boolean representation
- Can then do symbolic model checking by calling the CUDD BDD package

```
sal-smc smallbakery safety
proved
```

- Let's see it

Now let's put theorem proving and model checking together

Combining Model Checking and Theorem Proving

- Model checking a downscaled instance is a useful prelude to theorem proving the general case
- But a more interesting combination is to use model checking as part of a proof for the general case
- One approach is to create a finite state property-preserving abstraction of the original protocol
  - Theorem proving shows abstraction preserves the property
  - Model checking shows abstraction satisfies the property

    Instead of proving `indinv(safe)`, we invoke a model checker to show `abs_system |- G(safe)` [LTL]

  Can actually do all of this within PVS, because it includes a symbolic model checker (for CTL)

  - Built on a decision procedure for finite μ-calculus
  - We use it to prove `init(s) => AG(safe)(s)` [CTL]

Conditions for Property-Preserving Abstraction

- Want an abstracted state type `abstract_state`
  - And corresponding transition relation `a_trans`
  - And initiality predicate `a_init`
  - Together with abstracted safety property `a_safe`
- And an abstraction function `abst` from `state` to `abstract_state`, such that following properties hold

```
init_simulation: THEOREM
  init(s) IMPLIES a_init(abst(s))
trans_simulation: THEOREM
  transitions(pre,post) IMPLIES a_trans(abst(pre),abst(post))
safety_preserved: THEOREM
  a_safe(abst(s)) IMPLIES safe(s)
abs_invariant_ctl: THEOREM      % a_safe is invariant
  a_init(as) IMPLIES AG(a_trans, a_safe)(as)
```

Abstracted Model

- It doesn't matter to the protocol what the actual values of the tickets are
- All that matters is whether or not each of them is zero, and whether one is less than the other
- We can use Booleans to represent these relations
  - This is called predicate abstraction
- So introduce the abstracted (finite) state type

```
abstract_state: TYPE =
  [# pc1, pc2: phase,
     t1_is_0, t2_is_0, t1_lt_t2: bool #]

  as, a_pre, a_post: VAR abstract_state
```

Abstraction Function
And Abstracted Properties in PVS

```
abst(s): abstract_state =
  (# pc1 := s`pc1, pc2 := s`pc2,
     t1_is_0 := s`t1 = 0, t2_is_0 := s`t2 = 0,
     t1_lt_t2 := s`t1 < s`t2 #)

a_init(as): bool =
   as`pc1 = idle AND as`pc2 = idle
     AND as`t1_is_0 AND as`t2_is_0

a_safe(as): bool =
   NOT (as`pc1 = critical AND as`pc2 = critical)
```

More of the Abstracted Specification

```
a_P1_transition(a_pre, a_post): bool =
 IF a_pre`pc1 = idle
   THEN a_post = a_pre WITH [(t1_is_0) := false,
                             (t1_lt_t2) := false,
                             (pc1) := trying]
 ELSIF a_pre`pc1 = trying
         AND (a_pre`t2_is_0 OR a_pre`t1_lt_t2)
   THEN a_post = a_pre WITH [(pc1) := critical]
 ELSIF a_pre`pc1 = critical
   THEN a_post = a_pre WITH [(t1_is_0) := true,
                             (t1_lt_t2) := NOT a_pre`t2_is_0,
                             (pc1) := idle]
 ELSE a_post = a_pre
ENDIF
```

The Rest of the Abstracted Specification

```
a_P2_transition(a_pre, a_post): bool =
 IF a_pre`pc2 = idle
   THEN a_post = a_pre WITH [(t2_is_0) := false,
                             (t1_lt_t2) := true,
                             (pc2) := trying]
 ELSIF a_pre`pc2 = trying
         AND (a_pre`t1_is_0 OR NOT a_pre`t1_lt_t2)
   THEN a_post = a_pre WITH [(pc2) := critical]
 ELSIF a_pre`pc2 = critical
   THEN a_post = a_pre WITH [(t2_is_0) := true,
                             (t1_lt_t2) := false,
                             (pc2) := idle]
 ELSE a_post = a_pre
ENDIF
```

Proofs to Justify The Abstraction

- The conditions `init_simulation` and `safety_preserved` are proved by `(GRIND)`
- And `abs_invariant_ctl` is proved by `(MODEL-CHECK)`
  - Or could use SALenv on SAL equivalent (Let's see that)
- But `trans_simulation` has 2 unproved cases—here's the first

```
trans_simulation.2.6.1 :

[-1]  post!1 = pre!1
{-2}  idle?(pre!1`pc1)
  |-------
{1}   critical?(pre!1`pc2)
[2]   pre!1`t1 = 0
[3]   pre!1`t1 > pre!1`t2
{4}   idle?(pre!1`pc2)
{5}   pre!1`t1 < pre!1`t2
```

The Problem Justifying The Abstraction

- The problem here is that when the two tickets are equal but nonzero, the concrete protocol drops through to the `ELSE` case and requires the pre and post states to be the same
- But in the abstracted protocol, this situation can satisfy the condition for Process 2 to enter its critical section
  - Because `NOT a_pre`t1_lt_t2` abstracts `pre`t1 >= pre`t2` rather than `pre`t1 > pre`t2`
- But this situation can never arise, because each ticket is always incremented to be strictly greater than the other
- We can prove this as an invariant

```
not_eq(s): bool = s`t1 = s`t2 => s`t1 = 0

extra: LEMMA indinv(not_eq)
```

- This is proved with `(GRIND)`

A Justification of the Abstraction

- A stronger version of the simulation property allows us to use a known invariant to establish it

```
strong_trans_simulation: THEOREM
   indinv(not_eq)
    AND not_eq(pre) AND not_eq(post)
    AND transitions(pre, post)
      IMPLIES a_trans(abst(pre), abst(post))
```

- This is proved by

```
(SKOSIMP)
(EXPAND "transitions")
(GROUND)
(("1" (EXPAND "P1_transition")
      (APPLY (THEN (GROUND) (GRIND))))
 ("2" (EXPAND "P2_transition")
      (APPLY (THEN (GROUND) (GRIND)))))
```

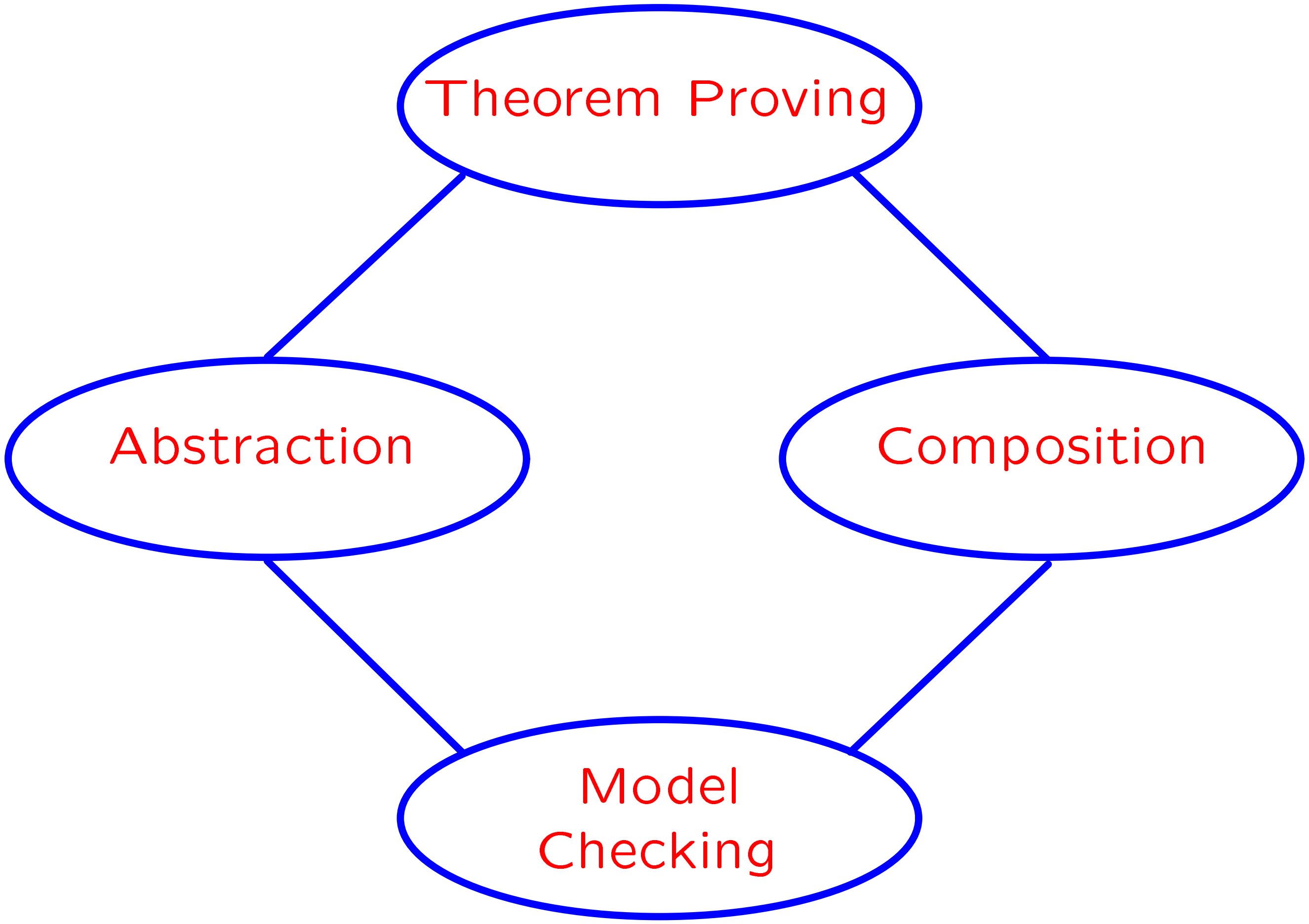
Pros and Cons of Manually-Constructed Abstractions

- Justifying the abstraction is usually almost as hard as proving the property directly
- And generally requires auxiliary invariants
- Bounded retransmission protocol required 45 of the original 57 invariants to justify an abstraction
- But there's the germ of an idea here

Abstraction Is The Bridge

Between Deductive and Algorithmic Methods

And Between Refutation and Verification



Failure-Tolerant Theorem Proving

- Model checking is based on search
- Safe to do because the search space is bounded, and efficient because we know its structure
- Verification systems (theorem provers aimed at verification) tend to avoid search at the top level
  - Too big a space to search, too little known about it
  - When they do search, they have to rely on heuristics
  - Which often fail
- Classical verification poses correctness as one "big theorem"
  - So failure to prove it (when true) is catastrophic
- Instead, let's try "failure-tolerant" theorem proving
  - Prove lots of small theorems instead of one big one
  - In a context where some failures can be tolerated

Contexts for Failure-Tolerant Theorem Proving

- Extended static checking (see later)
- Property preserving abstractions
  - Instead of justifying an abstraction,
  - Use deduction to calculate it
- Given a transition relation G on S and property P, a property-preserving abstraction yields a transition relation $\hat{G}$ on $\hat{S}$ and property $\hat{P}$ such that

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

  Where $\hat{G}$ and $\hat{P}$ that are simple to analyze (e.g., finite state)
- A good abstraction typically (for safety properties) introduces nondeterminism while preserving the property
- Note that abstraction is not the inverse of refinement

Calculating an Abstraction

- We need to figure out if we need a transition between any pair of abstract states
- Given abstraction function $\phi : [S \rightarrow \hat{S}]$ we have

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

- We'll use highly automated theorem proving on these formulas: 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

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

- Now theorem-proving failure affects accuracy, not soundness

Automated Abstraction

- The method described is automated in InVeSt
  - An adjunct to PVS developed in conjunction with Verimag, now being integrated into SAL
- A different method (due to Saïdi and Shankar) is implemented in PVS
  - Exponentially more efficient
- The abstraction is specified in the proof command by giving the concrete function or predicate that defines the value of each abstract state variable

Automated Abstraction in PVS

- `auto_abstract: THEOREM`
  ```
  init(s) IMPLIES AG(transitions, safe)(s)
  ```
- This is proved by
  ```
  (abstract-and-mc "state" "abstract_state"
    (("t1_is_0" "lambda (s): s`t1=0")
     ("t2_is_0" "lambda (s): s`t2=0")
     ("t1_lt_t2" "lambda (s): s`t1 < s`t2")))
  ```
- Now let's see this work in practice

Other Kinds of Abstraction

- We've seen predicate abstraction [Graf and Saïdi]
- I'll briefly sketch data abstraction and hybrid abstraction

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}
- Syntactically replace functions f on C by abstracted functions $\hat{f}$ on A
- 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

Data Abstraction Example

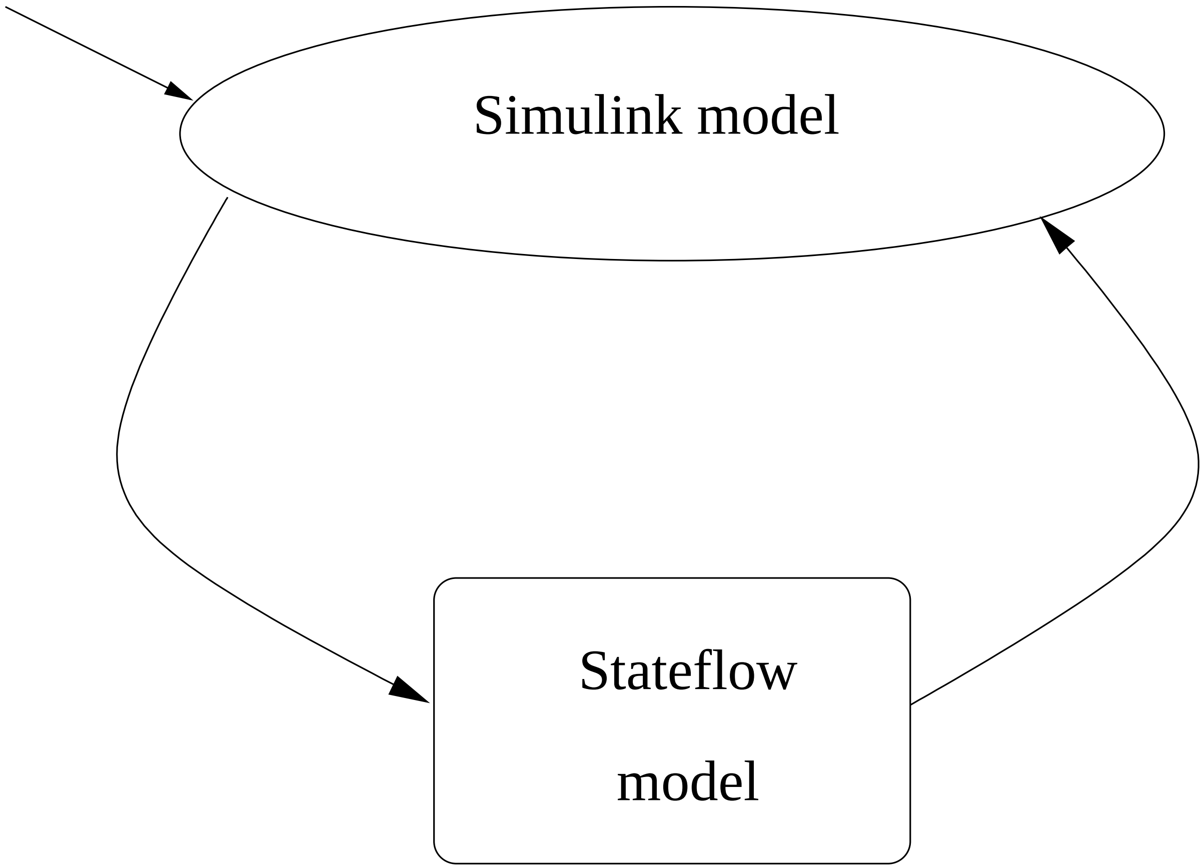
Replace natural numbers by {0, 1, many}

Calculate behavior of subtraction on {0, 1, many}

$-$	0	1	many
0	0	$-$	$-$
1	1	0	$-$
many	many	$\{1, \text{many}\}$	$\{0, 1, \text{many}\}$

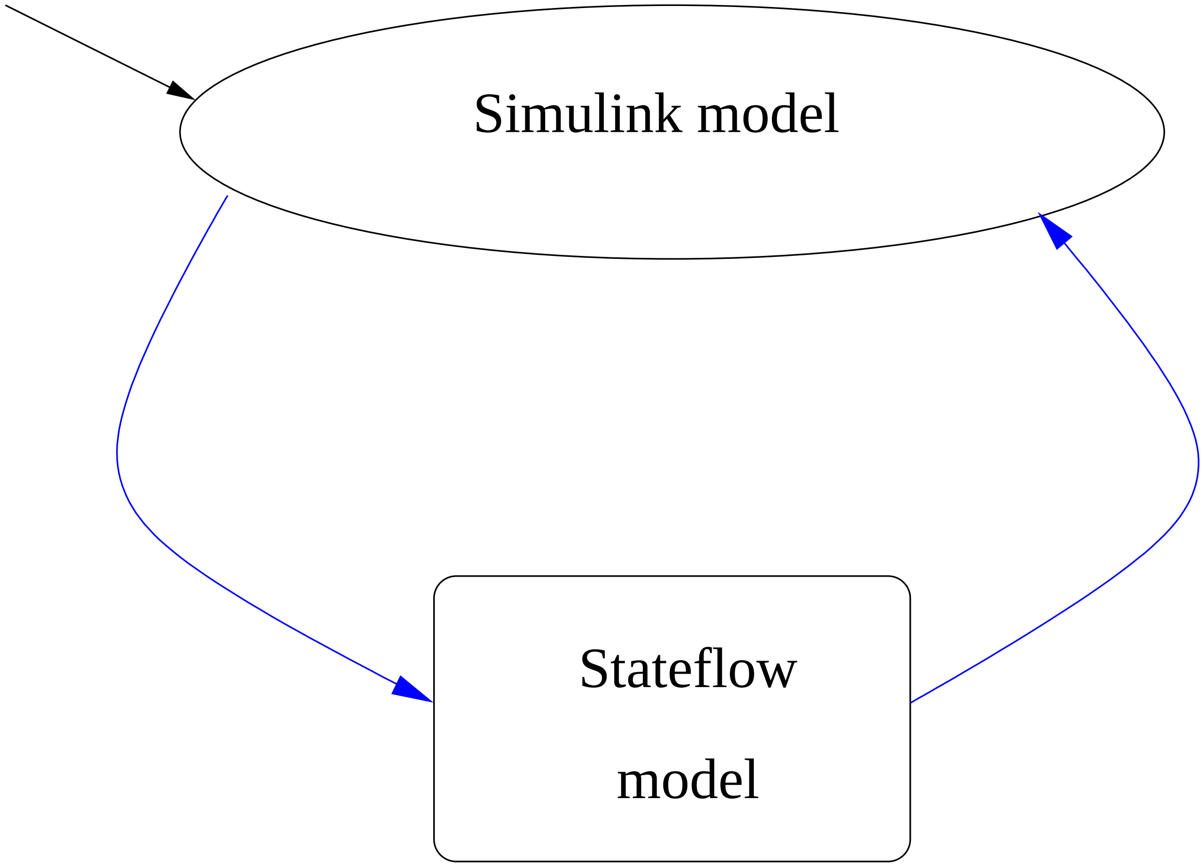
$$0 \notin (\text{many} - 1) \;\; \mathit{iff} \;\; \forall x \in \{2, 3, 4, \ldots\} : x - 1 \neq 0$$

Data Abstraction for Matlab (Hybrid Systems)



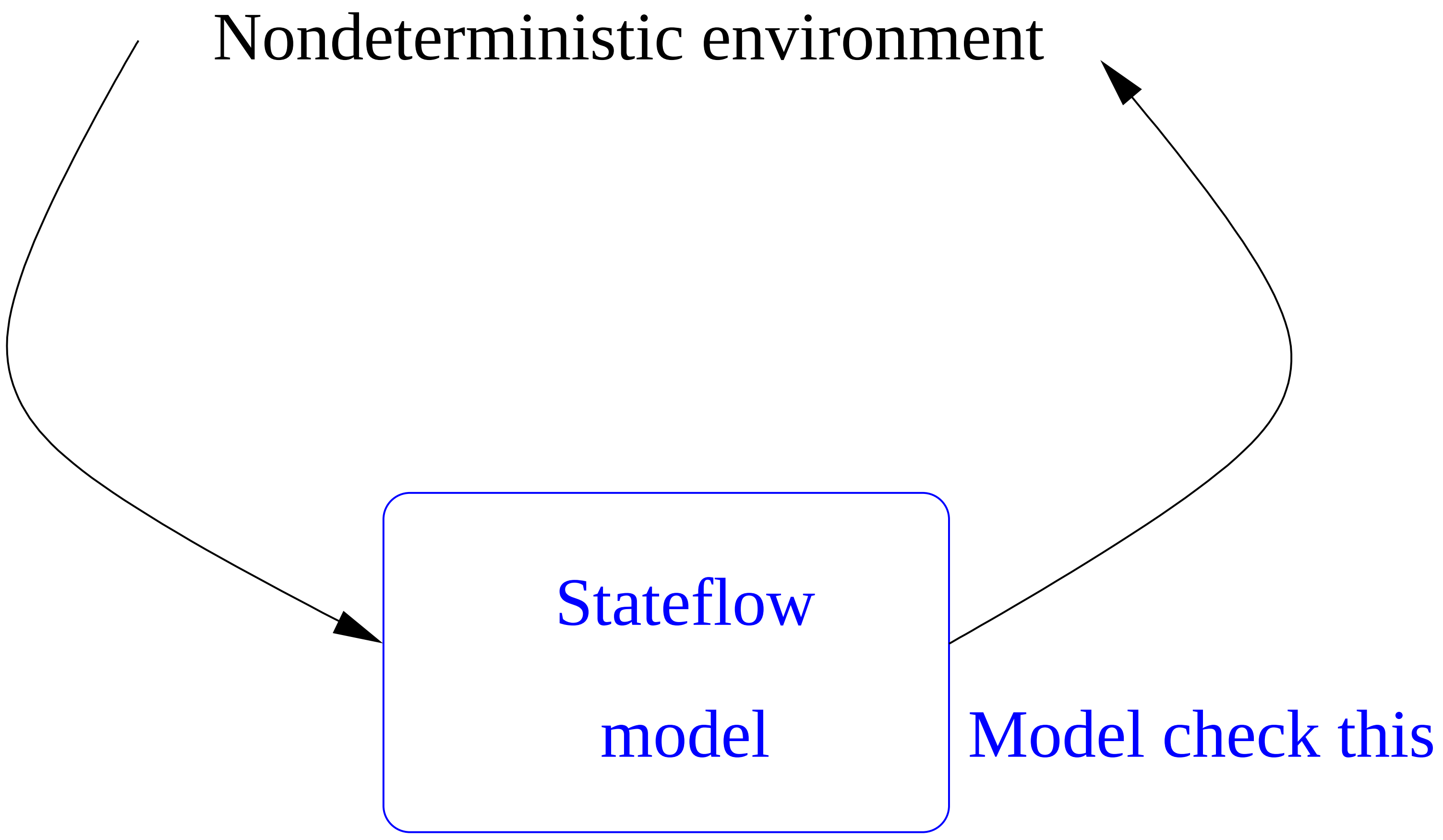
Mixed continuous/discrete (i.e., hybrid) system

Simulate One Trajectory at a Time



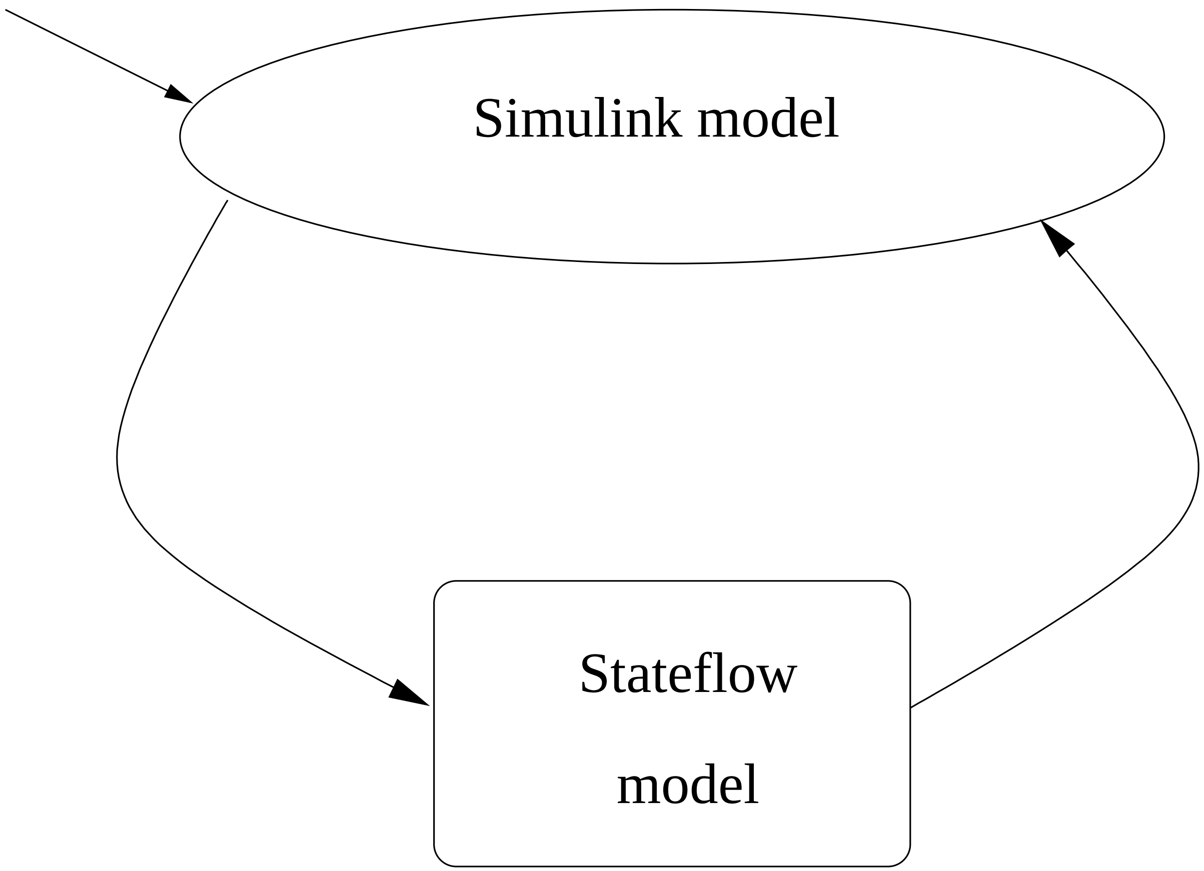
Just like testing: when have you done enough?

Model Check With Nondeterministic Environment



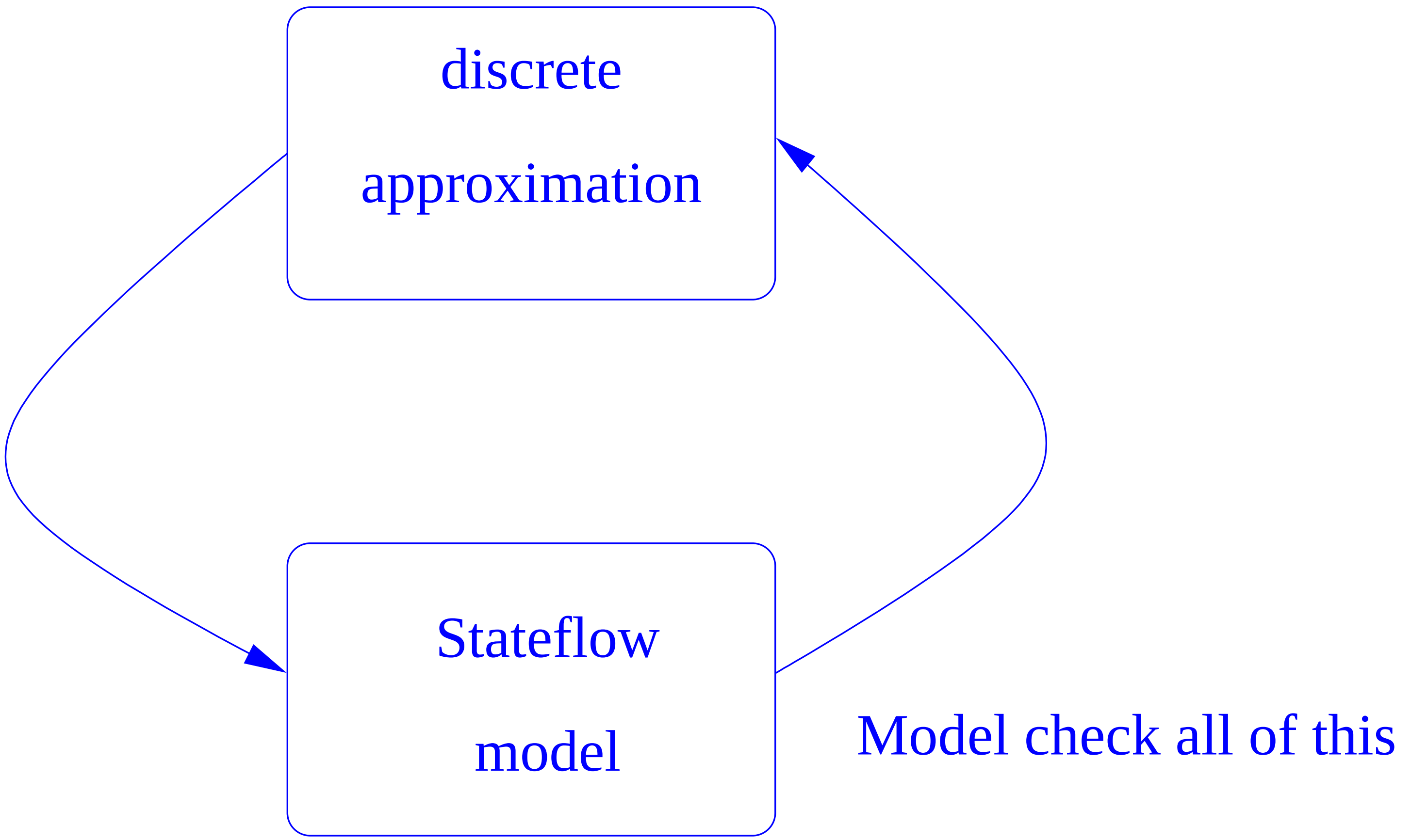
Too crude to establish useful properties

Analyze By The Methods Of Hybrid Systems



OK, but restricted

Model Check With Sound Discretization Of The Continuous Environment



Just right

Data Abstraction for Hybrid Systems

- Method developed by Ashish Tiwari
- The continuous environment is given by some collection of (polynomial) differential equations on $\mathbf{R}^n$
- Divide these into regions where the first j derivatives are sign-invariant (m polynomials, $(m \times j)^3$ regions)
  - I.e., data abstraction from $\mathbf{R}$ to $\{-, 0, +\}$
  - For each mode $l \in \mathbf{Q}$: if q_{pi}, q_{pj} abstract p_i, p_j and $\dot{p}_i = p_j$ in mode l, then apply rules of the form:
    - ⋆ if $q_{pi} = +$ & $q_{pj} = +$, then q'_{pi} is $+$
    - ⋆ if $q_{pi} = +$ & $q_{pj} = 0$, then q'_{pi} is $+$
    - ⋆ if $q_{pi} = +$ & $q_{pj} = -$, then q'_{pi} is either $+$ or 0
    - ⋆ ...

Data Abstraction for Hybrid Systems

- Larger choices of j give successively finer abstractions
- Usually enough to take $j = 1$ or 2
- Method is complete for some (e.g., nilpotent) systems
- Parameterized also by selection of polynomials to abstract on
  - The eigenvectors are a good start
  - Method is then complete for linear systems
- Construction is automated using decision procedures for real closed fields (e.g., Cylindric Algebraic Decomposition—CAD)
- Also provides a general underpinning to qualitative reasoning as used in AI

Example: Thermostat

Consider a simple thermostat controller with:

- Discrete modes: Two modes, $q = on$ and $q = off$
- Continuous variable: The temperature x
- Initial State: $q = off$ and $x = 75$
- Discrete Transitions:

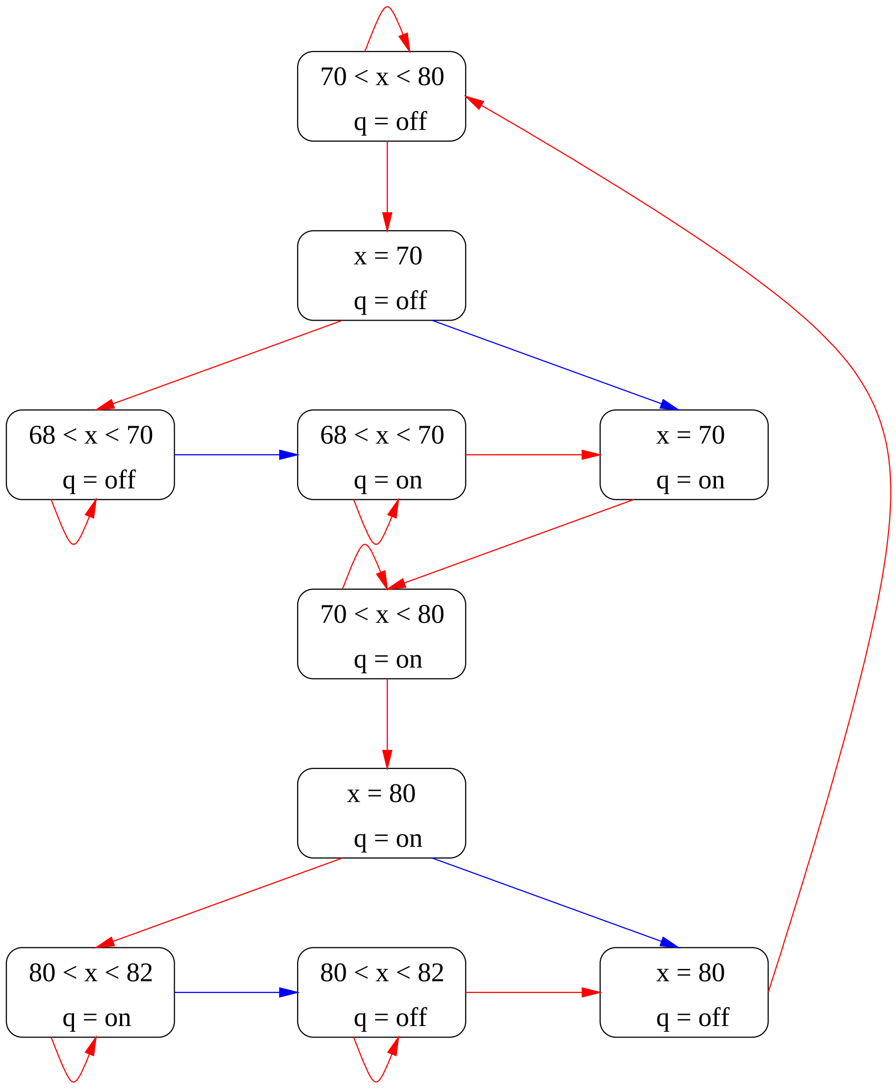
$$
\begin{aligned}
q = off \ and \ x \leq 70 \quad &\longrightarrow \quad q' = on \\
q = on \ and \ x \geq 80 \quad &\longrightarrow \quad q' = off
\end{aligned}
$$

- Continuous Flow:

$$
\begin{aligned}
q = off \ and \ x > 68 \quad &\longrightarrow \quad \dot{x} = -Kx \\
q = on \ and \ x < 82 \quad &\longrightarrow \quad \dot{x} = K(h - x)
\end{aligned}
$$

We want to prove $68 \leq x \leq 82$

Abstract Thermostat System



Pros and Cons of Automated Abstraction

- Good match between local theorem proving, and global model checking
- Quality of the abstraction depends on information provided by the user (predicates, polynomials etc.)
  - It's easier to guess useful predicates than invariants
  - Can guess additional ones if inadequate
  - Or let counterexamples suggest refinements
    - ⋆ A general approach can be discerned here: find quick solutions and fix them up, rather than deliberate in hope of finding good solutions

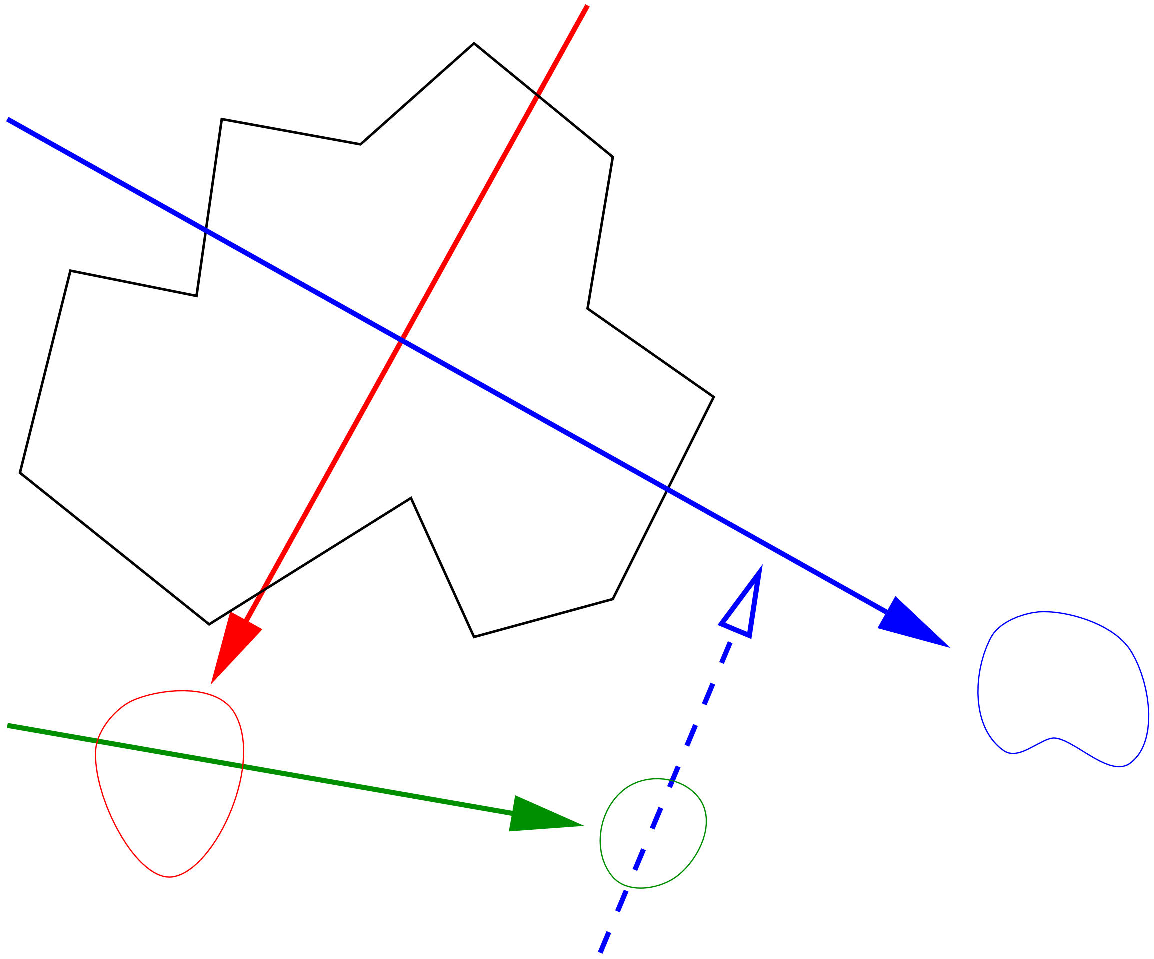
  And the deductive power applied
  - Which may increase if provided with known invariants

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



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

Refutation and Verification

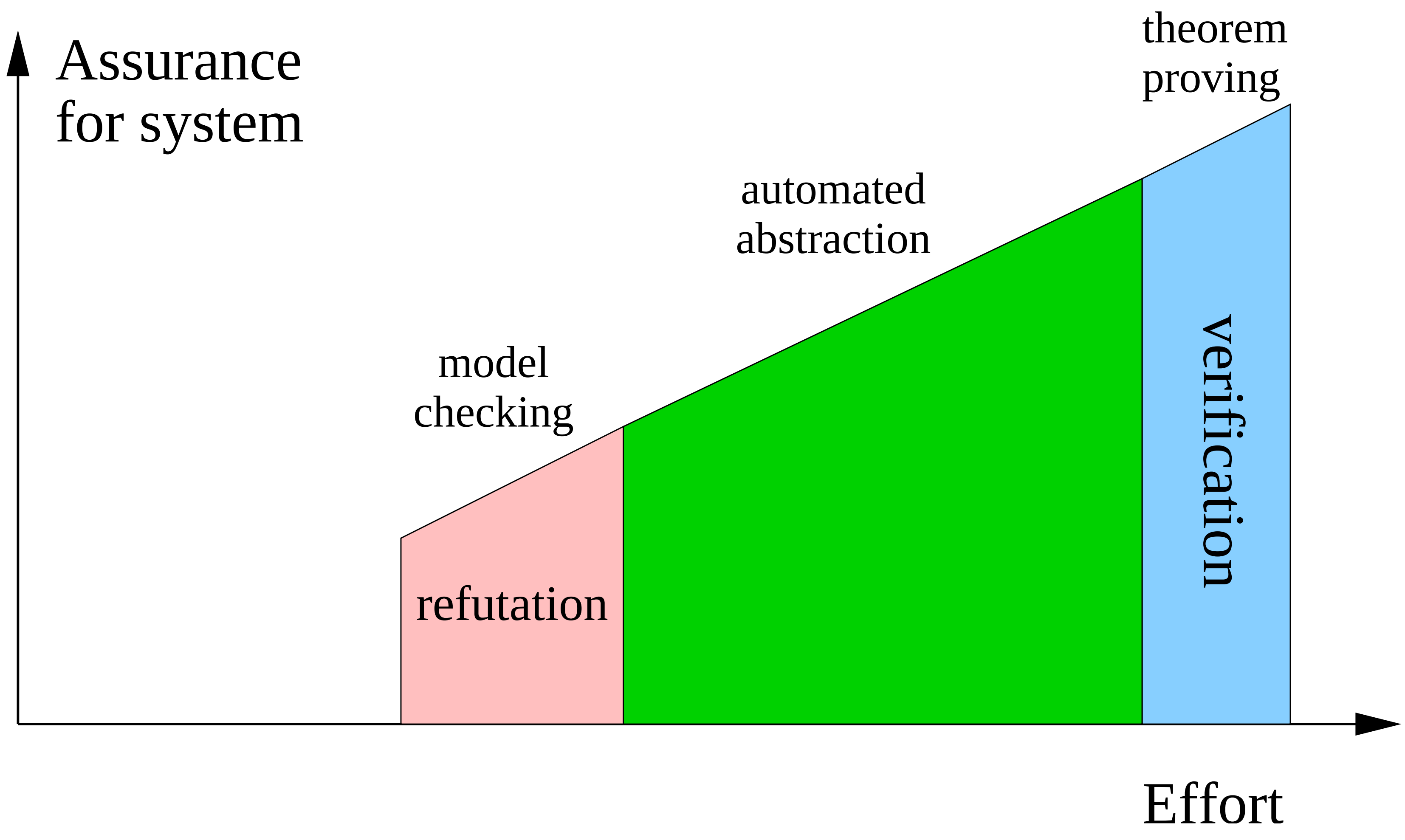
- By allowing unsound abstractions

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

  We can do refutation as well as verification

- Then, 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



Tools Employing Abstraction

PVS, SAL (SRI): data, predicate, and hybrid abstraction

InVeSt (Verimg/SRI): predicate abstr'n, invariant gener'n

Bandera (KSU): data abstraction for Java

ESC (Ex Compaq SRC): predicate abstr'n/guessing on Java

PAX (Kiel): predicate abstraction (to WS1S)

SLAM (MSR): predicate abstraction on C (device drivers)

BLAST (UCB): similar

STeP (Stanford): predicate abstraction

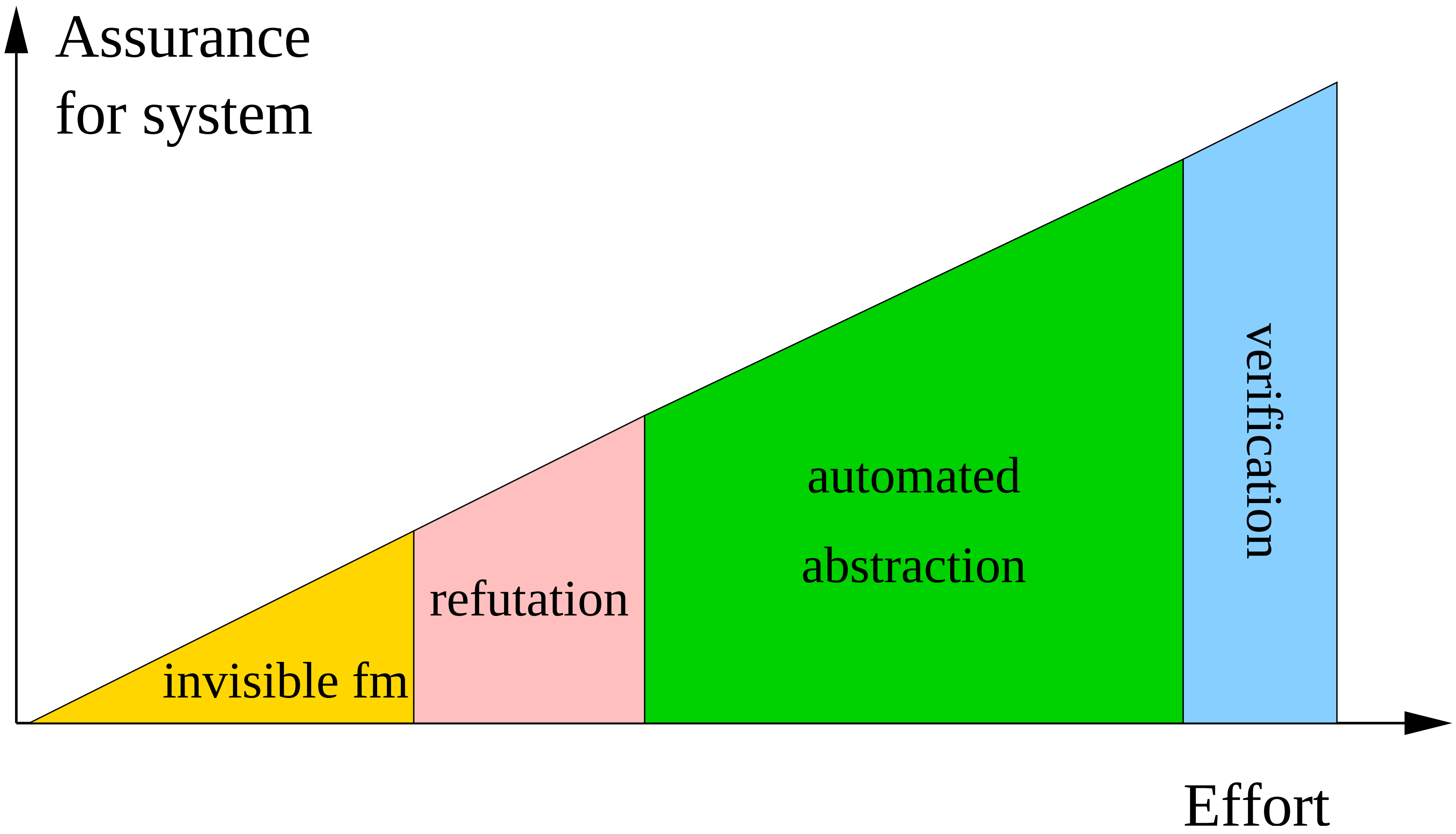
Veritech (Technion): ...

... (...): ...lots of experimental tools (Stanford, UCB, ...)

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
  - We call these invisible formal methods
  - And it's where a lot of the action and opportunity is

The Formal Methods Wedge



Examples of Invisible Formal Methods
Stronger Checking in Traditional Tools

- Various forms of extended static checking
  - Failed proof generates a possibly spurious warning
- Static analysis: typestate, shape analysis, abstract interpretation etc.
- 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 spurious warning)
- The verifying compiler

Examples of Invisible Formal Methods

Better Tools for Traditional Activities

- Statechart/Stateflow property checkers (cf. OFFIS, IBM Pathfinder)
  - Show me a path that activates this state
  - Can this state and that be active simultaneously?
- Checker synthesizers (cf. IBM FOCS)
- Completeness/Consistency checkers for tabular specifications (cf. Ontario Hydro, RSML, SCR)
- Test case generators (cf. Verimag/IRISA TGV and STG)

There's an entire industry in this space, with many companies make a living from modest technology (but very good understanding of their markets)

Key Technology: Constraint Satisfaction

- Many of these examples can be seen as instances of constraint satisfaction:
  - Find a test case that will exercise a particular path
  - Find a counterexample to this property
    - ⋆ This is bounded model checking—BMC
- When the system is finite state, these problems can be solved by propositional satisfiability solvers (SAT-solvers)
- Recently, methods for very efficient SAT solvers have become widely known (Chaff)
- BMC scales better than BDD-based model checking for refutation in industrial contexts (though they often use several methods in cascade)

Bounded Model Checking

- Given a system specified by initiality predicate I and transition relation T, 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 (which we have anyway for symbolic model checking), this is a propositional satisfiability (SAT) problem
- SAT solvers have become amazingly fast recently
- Try $k = 1, 2, \ldots$ and submit each instance to a SAT solver
- SALenv2 can do this: `sal-bmc absbakerybug safety`
  Reports counterexample
- Let's see it

Bounded Model Checking (ctd.)

- Needs less tinkering than BDD-based symbolic model checker, can handle bigger systems and find deeper bugs
- Now widely used in hardware verification
- But is limited to refutation. . . or is it?

Extending BMC to Verification

- 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

Alternatively, Automated Induction via BMC

- 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
- Is complete for some problems (e.g., timed automata)
- So let's see it

Bounded Model Checking for Infinite State Systems

- We can discharge the BMC and perimeter formulas efficiently for Boolean encodings of finite state systems because SAT solvers do efficient search
- If we could discharge these formulas over richer theories, we could do BMC for state machines over these theories
- So how about if we combine a SAT solver with a decision procedure—e.g., ICS—for the combined theories?

SAT-Based Constraint Satisfaction

- Idea is to extend the efficient search of a modern SAT solver to propositionally complex formulas with interpreted terms at the leaves
  - E.g., $x < y \wedge (f(x) = y \vee 2 * g(y) < \epsilon) \vee \ldots$ for thousands of terms
- Replace the terms by propositional variables
- Get a solution from the SAT solver (if none, we are done)
- Restore the interpretation of variables and send the conjunction to the decision procedure
- If satisfiable, we are done
- If not, ask SAT solver for a new assignment—but isn't that expensive?

SAT-Based Constraint Satisfaction (ctd)

- Yes, so first, do a little bit of work to find some unsatisfiable fragments and send these back to the SAT solver as additional constraints (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 Decision Procedure + SAT

- We combined ICS with Chaff: worked well, but. . .
  - Chaff wants input in CNF
    (which is expensive to compute)
  - Sometimes does more than we need (in asynchronous composition, we only want assignments to variables of one process, but Don't Cares can interfere with search)
  - As a black box, hard to do efficient incremental restarts
    - ⋆ Note: decision procedure needs to be incremental, too
  - Licensing terms
- We replaced Chaff by a nonclausal solver designed for restarts and Don't Cares and gained another two orders of magnitude
- So let's see it

Infinite BMC etc.

- So now we can do BMC over systems defined using terms from the theories decided by ICS
- Not only more general, but sometimes faster too
  - E.g., encoding bitvectors in SAT vs. using the ICS decision procedure for bitvectors
- So let's see it. . . sorry

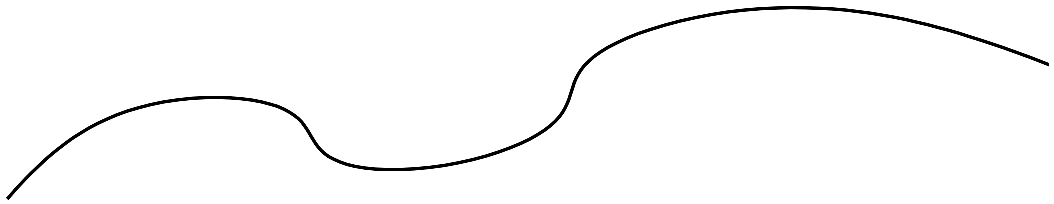
Other Applications for ICS and Infinite BMC

- Test-case generation (negate the property and use the counterexamples)
    - Structural coverage criteria can be formulated as temporal logic formulas
- Can augment any class of problems traditionally handled by SAT solvers (e.g., AI planning, diagnosis) to descriptions including decided theories

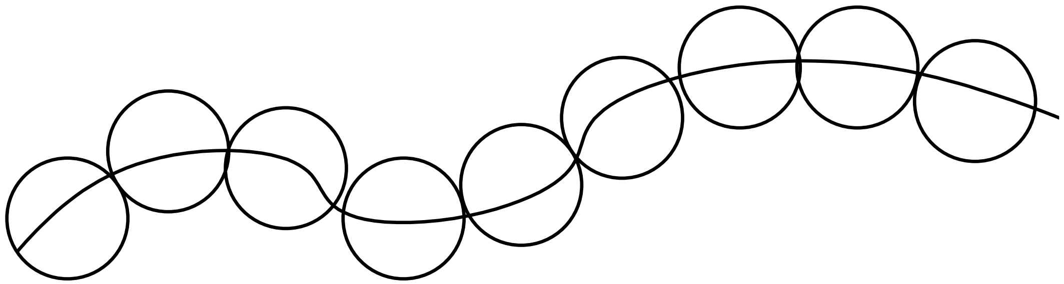
Combined Formal/Informal Methods

- Hardware designs can have millions of state bits and interesting traces thousands of steps long
- BMC can explore 10–100 steps on hundreds of state bits
- So BMC doesn't get you very far from a start state
- So, instead, do it from states found during random simulation
- Can be seen as a way to "fatten" thin traces explored by simulation
  - Or to amplify the power of simulation

Amplifying The Power Of Simulation



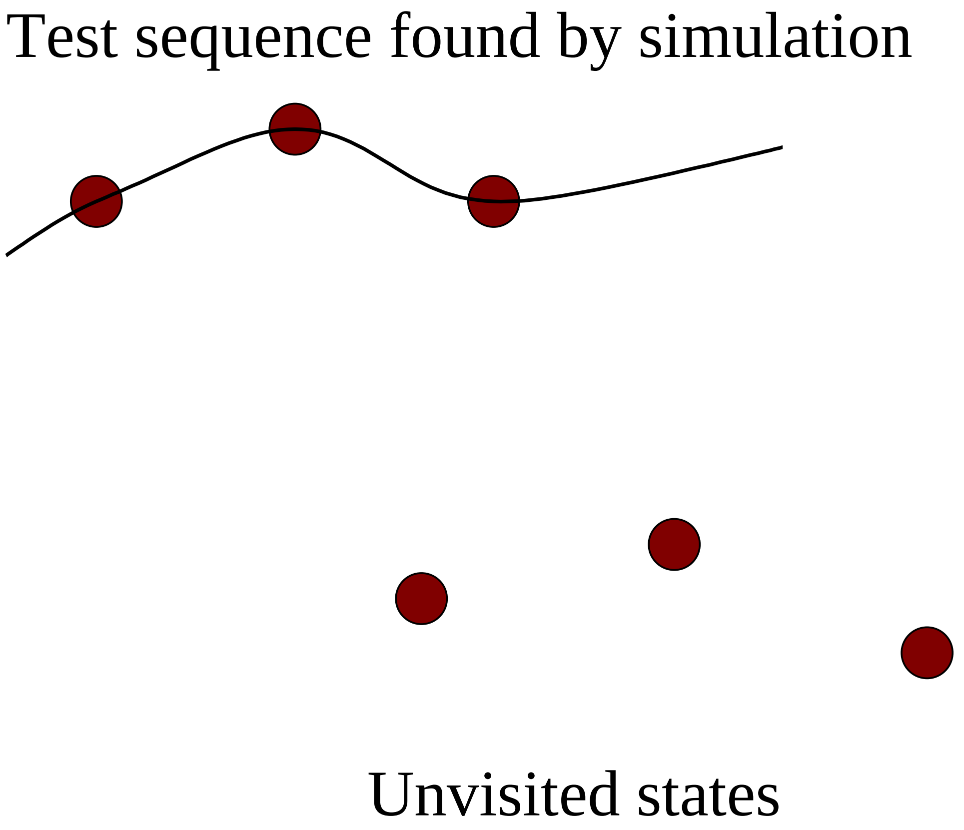
Test sequence found by simulation



Test sequence amplified by bounded model checking

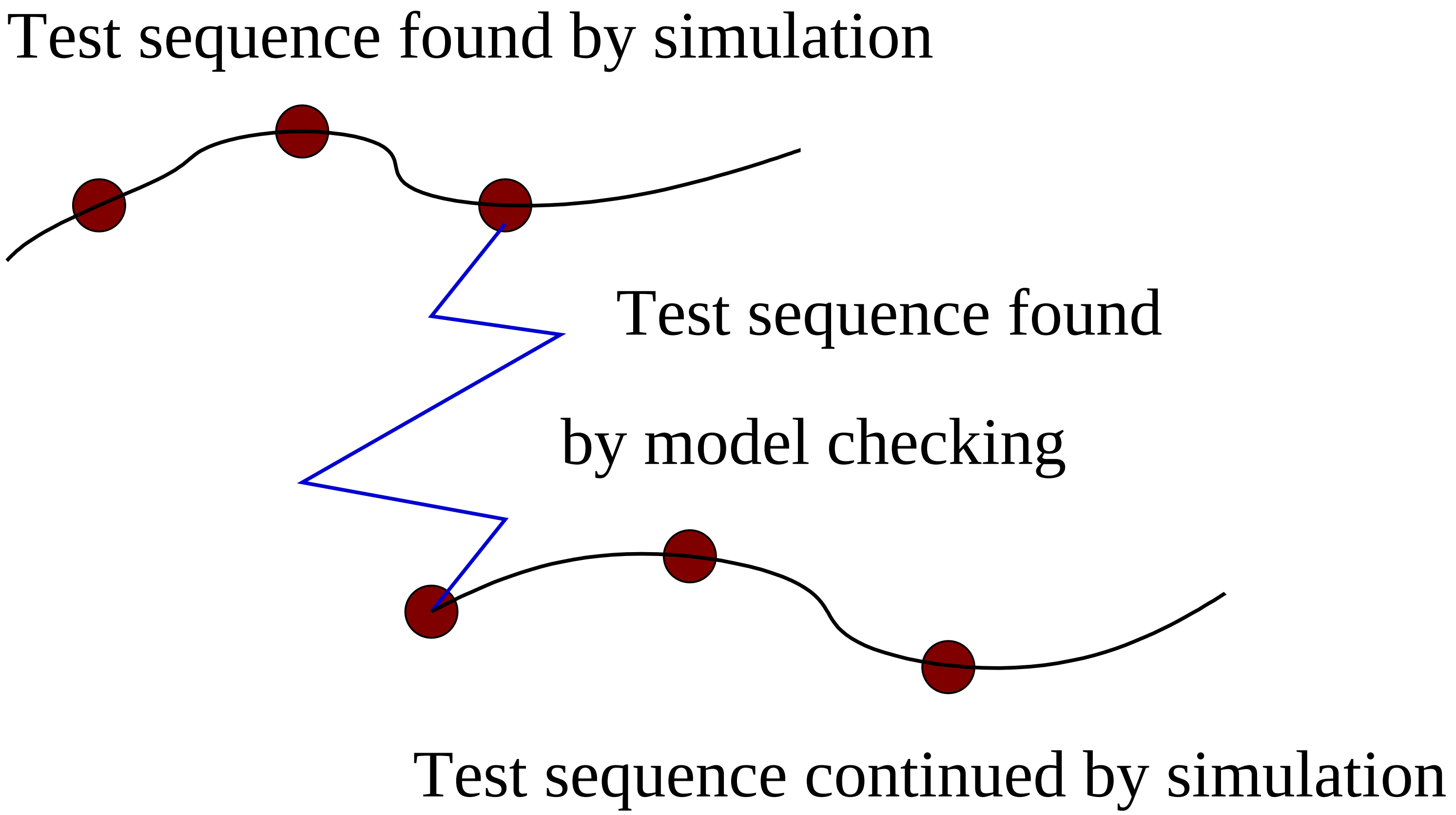
Extending The Reach Of Simulation

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



Extending The Reach Of Simulation

So use SAT-based model checking to jumpstart entry into those parts (also use abstraction to see if states are unreachable)



Invisible Formal Methods in Current Practice

BMC (...): used in many hardware companies

STE (...): Symbolic Trajectory Evaluation (originally from Randy Bryant) not invisible, but ... used in Intel, Compaq...

Logiscope (Polyspace): static analysis based on abstract interpretation—a whole other world I'm not qualified to discuss

Prefix (Microsoft): similar static analyzer used in-house

Checkerware (0-in): amplifies simulation for SOC verification

Ketchum (Synopsys): extends simulation for SOC verification

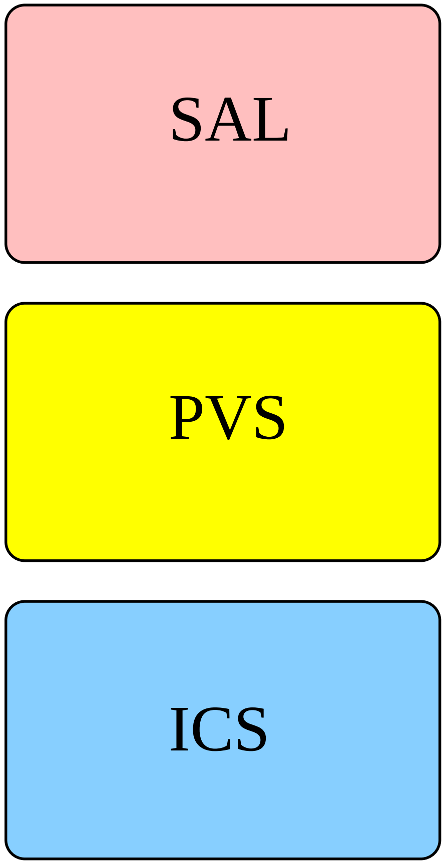
VN-Property-DX (TransEDA): property checking on simulations (this and later tools use our technology)

... (...): runtime monitoring, such as Temporal Rover

... (...): testcase generation such as TGV, STG, T-Vec

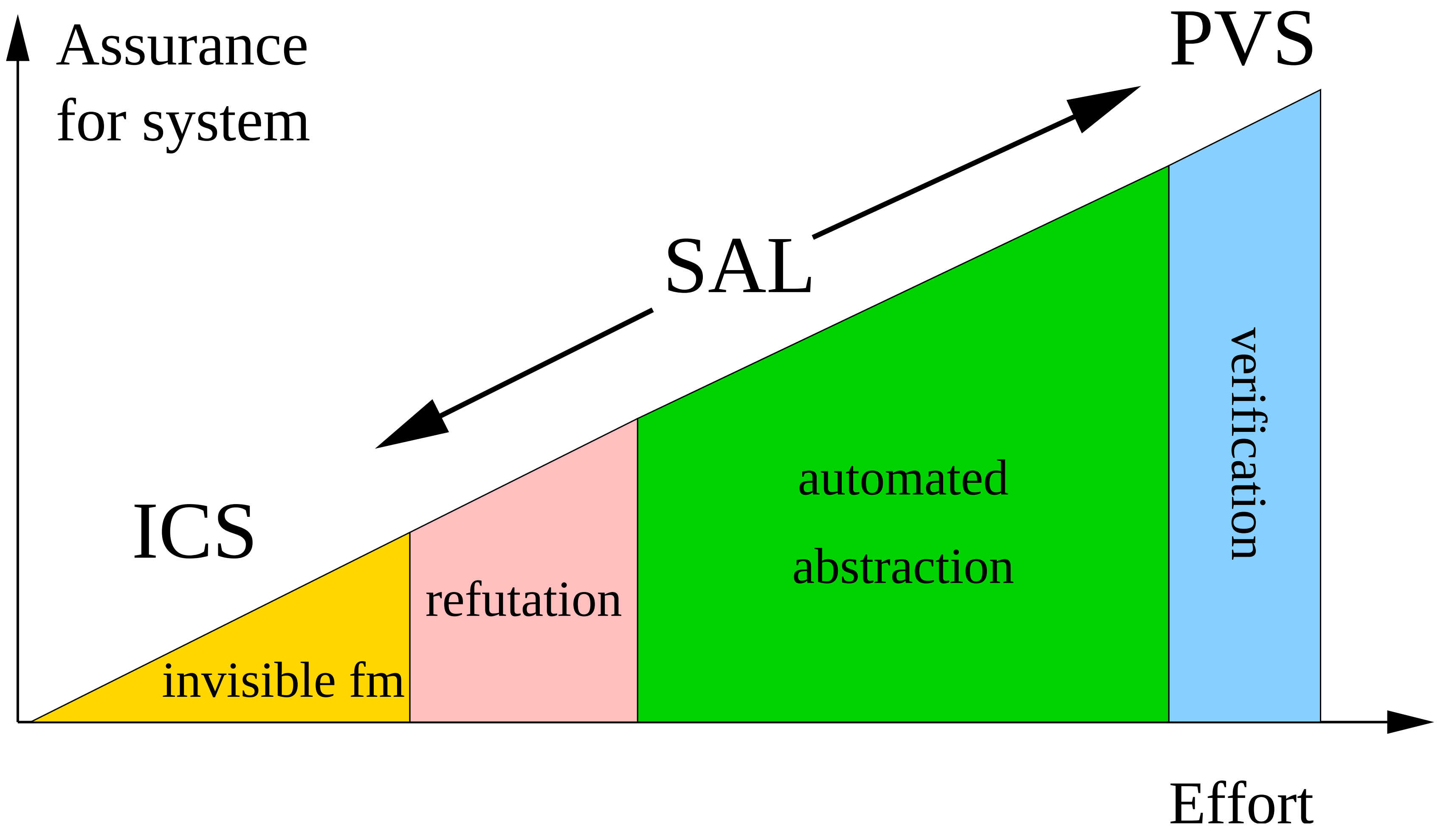
Our Tools

SAL = Symbolic Analysis Laboratory



ICS = Integrated Canonizer-Solver (= ICanSolve)

How It Fits Together



Summary

- The challenge faced by formal methods analysis tools is how to search a huge space efficiently
- Theorem proving has developed efficient methods of local search (decision procedures etc.)
- Model checking showed that efficient global search was possible
- Now, methods are emerging that combine insights from both approaches in a promising way
- And there a pragmatic focus on finding methods for using the (still limited) capabilities of formal analysis tools to address useful, but partial issues in big, real systems
- I have never felt more optimistic about the prospects for formal analysis tools

Acknowledgments

- Very little of this is my work
- Most of it the work of my colleagues: Judy Crow, Leonardo de Moura, Sam Owre, Harald Rueß, Shankar, Ashish Tiwari
- With valuable contributions from Saddek Bensalem, Pavol Cerny, Bruno Dutertre, Jean-Christophe Filliâtre, Klaus Havelund, Friedrich von Henke, Yassine Lakhnech, Pat Lincoln, César Muñoz, Holger Pfeifer, Vlad Rusu, Hassen Saïdi, Eli Singerman, Maria Sorea, Dave Stringer-Calvert, and many others

To Learn More

- Check out papers and technical reports at `http://www.csl.sri.com/programs/formalmethods`
- Information about PVS, and the system itself, is available from `http://pvs.csl.sri.com`
  - Freely available under license to SRI
  - Built in Allegro Lisp for Solaris, or Linux
  - Version 3.0 includes predicate and data abstraction
- ICS: `http://ics.csl.sri.com` and `http://www.ICanSolve.com`
  - Written in OCaml
  - Available as library for OCaml, Lisp, C
- SAL: `http://sal.csl.sri.com`
  - Written in Java, C, Scheme, dotty
  - Interface is XML