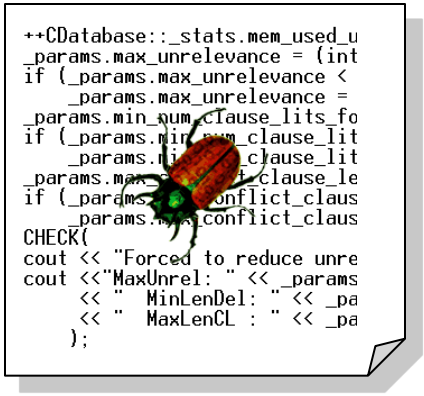
Abstractions and Decision Procedures for Effective Software Model Checking



Prof. Natasha Sharygina

The University of Lugano,

Carnegie Mellon University

Microsoft Summer School, Moscow, July 2011

Lecture 1

Traditional Approaches

- *Testing:* Run the system on select inputs
- *Simulation*: Simulate a model of the system on select (often symbolic) inputs
- Code *review* and *auditing*

What are the Problems?

- not exhaustive (missed behaviors)
- not all are automated (manual reviews, manual testing)
- do not scale (large programs are hard to handle)
- no guarantee of results (no mathematical proofs)
- concurrency problems (non-determinism)

What is Formal Verification?

- Build a mathematical model of the system:
  - what are possible behaviors?
- Write correctness requirement in a specification language:
  - what are desirable behaviors?
- Analysis: (Automatically) check that model satisfies specification

What is Formal Verification (2)?

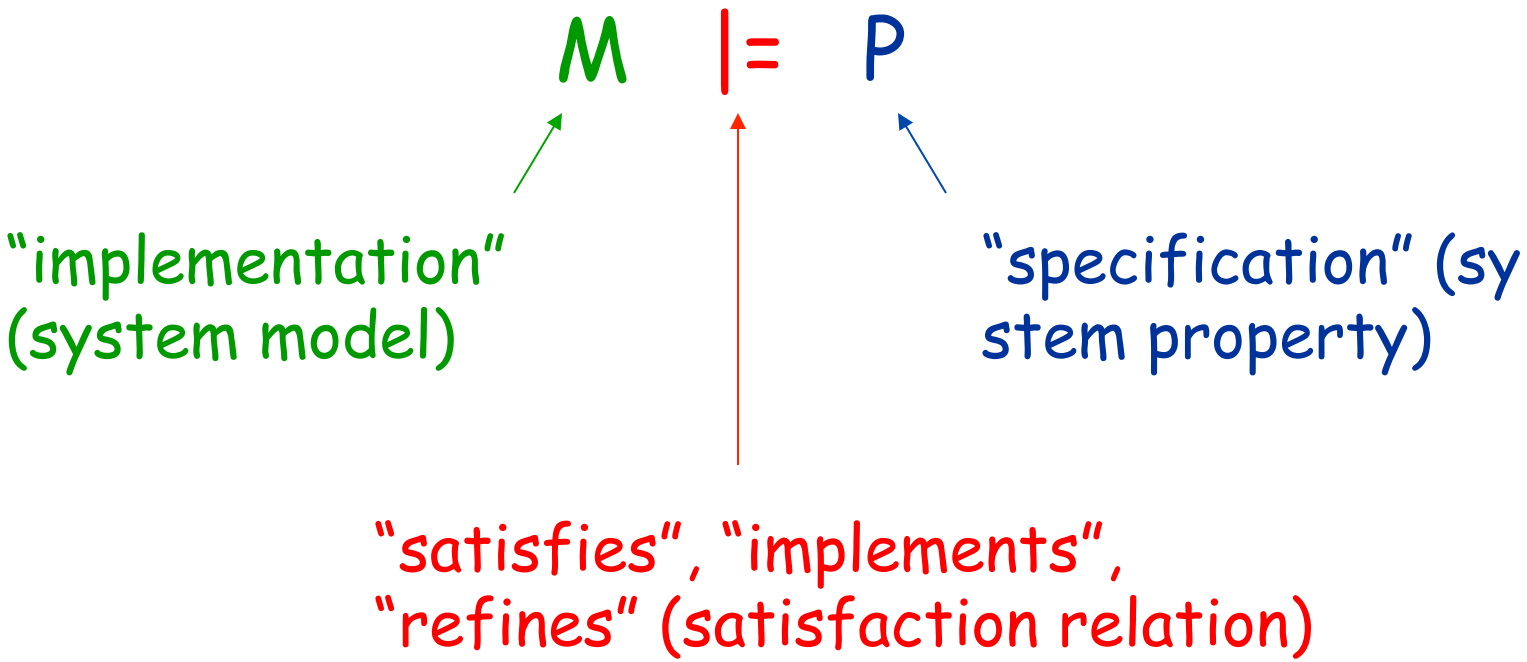
- Formal - Correctness claim is a precise mathematical statement
- Verification - Analysis either proves or disproves the correctness claim

Algorithmic Analysis by Model Checking

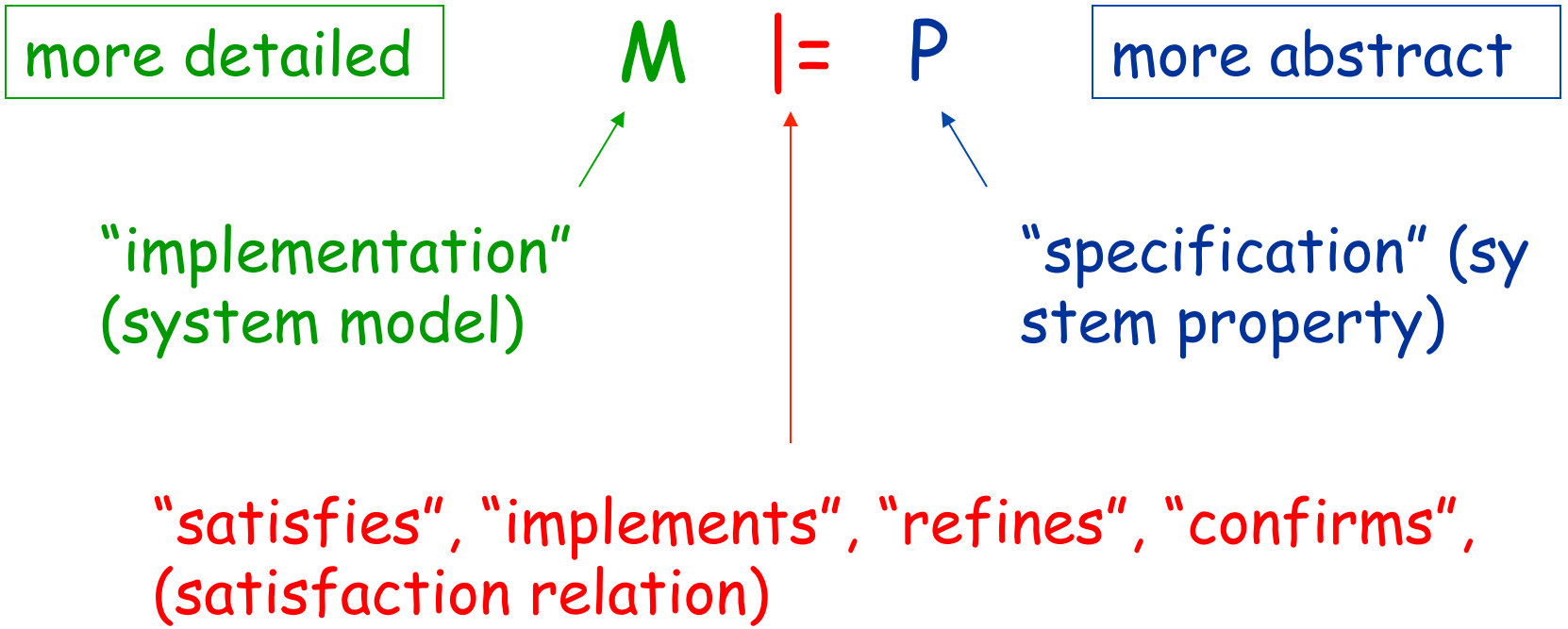
- Analysis is performed by an algorithm (tool)
- Analysis gives counterexamples for debugging
- Typically requires exhaustive search of state-space
- Limited by high computational complexity

Temporal Logic Model Checking

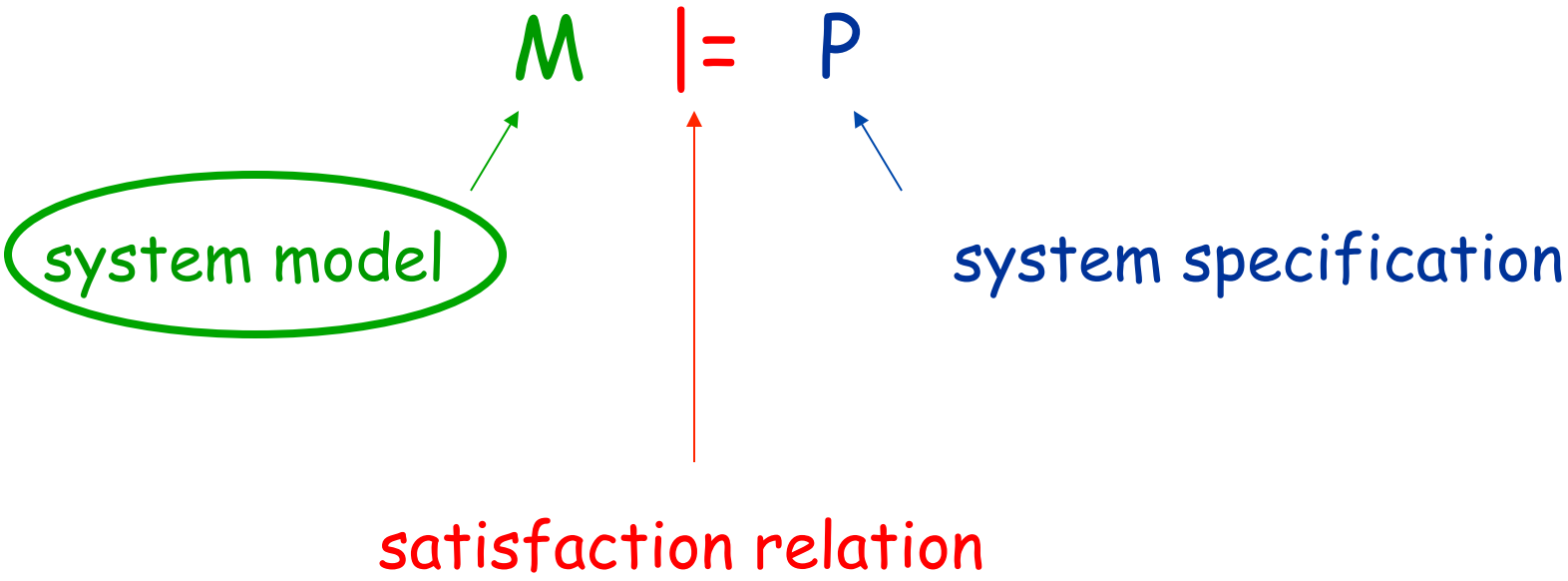
[Clarke,Emerson 81][Queille,Sifakis 82]



Temporal Logic Model Checking



Temporal Logic Model Checking



Decisions when choosing a system model:

- variable-based vs. event-based
- interleaving vs. true concurrency
- synchronous vs. asynchronous interaction
- clocked vs. speed-independent progress
- etc.

Characteristics of system models

which favor model checking over other verification techniques:

- ongoing input/output behavior
  (not: single input, single result)
- concurrency
  (not: single control flow)
- control intensive
  (not: lots of data manipulation)

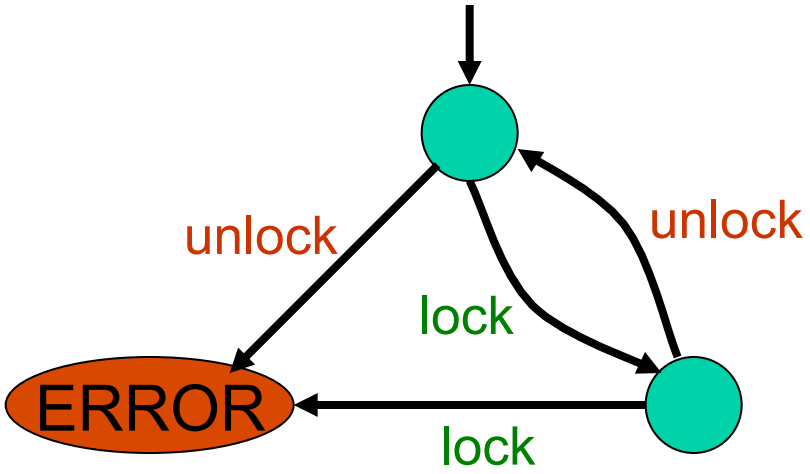
Decisions when choosing a system model:

While the choice of system model is important for ease of modeling in a given situation,

the only thing that is important for model checking is that the system model can be translated into some form of state-transition graph.

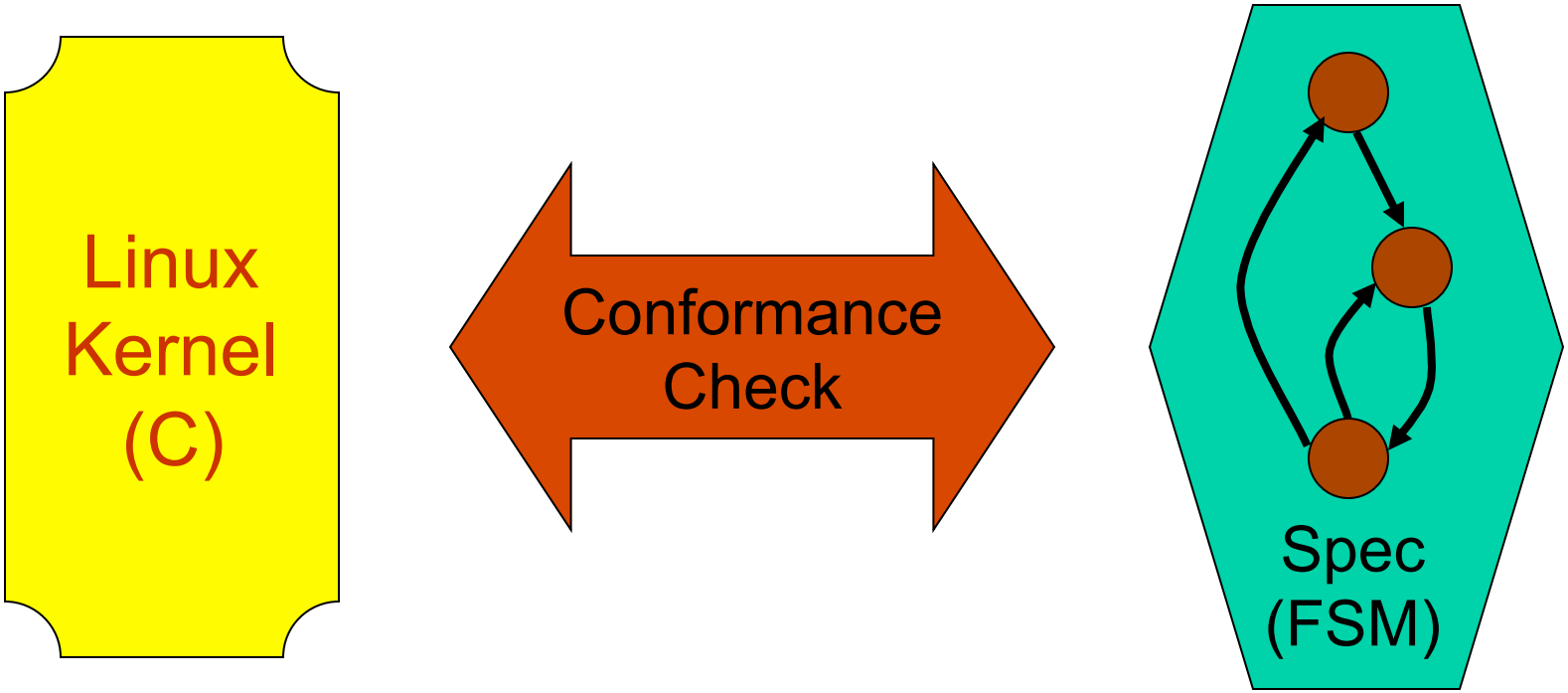
Finite State Machine (FSM)

- Specify state-transition behavior
- Transitions depict observable behavior

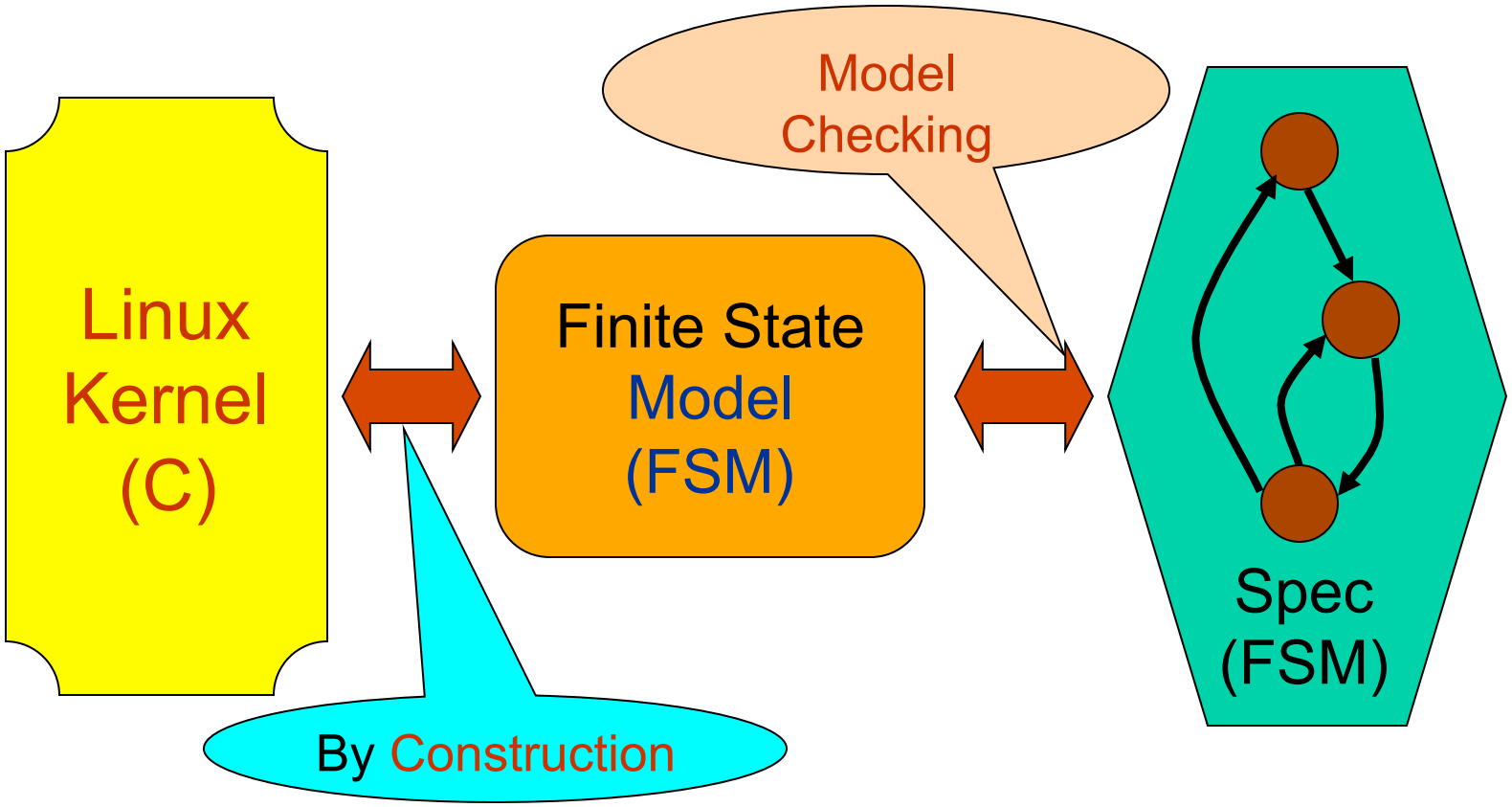


Acceptable sequences of acquiring and releasing a lock

High-level View



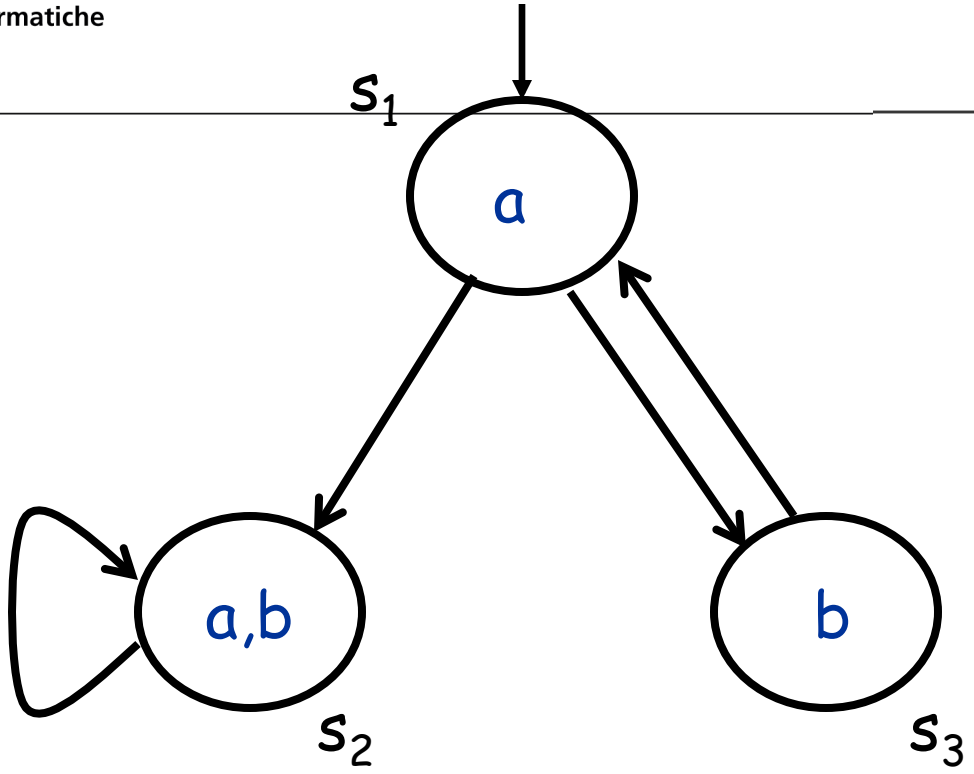
High-level View



Low-level View

State-transition graph

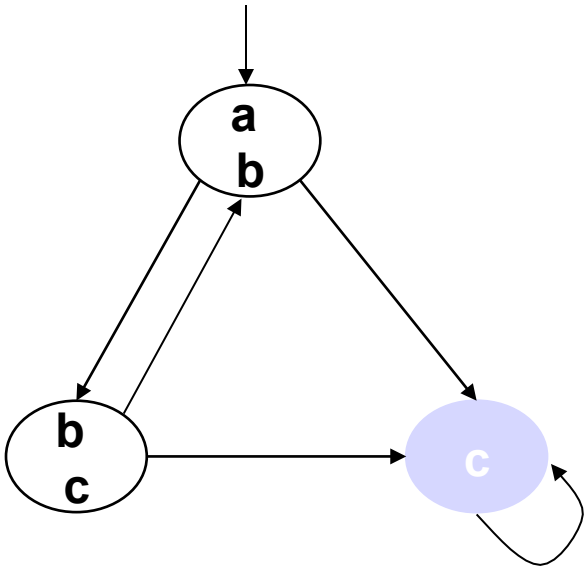
S	set of states
I	set of initial states
AP	set of atomic observation
$R \subseteq S \times S$	transition relation
$L: S \rightarrow 2^{AP}$	observation (labeling) function



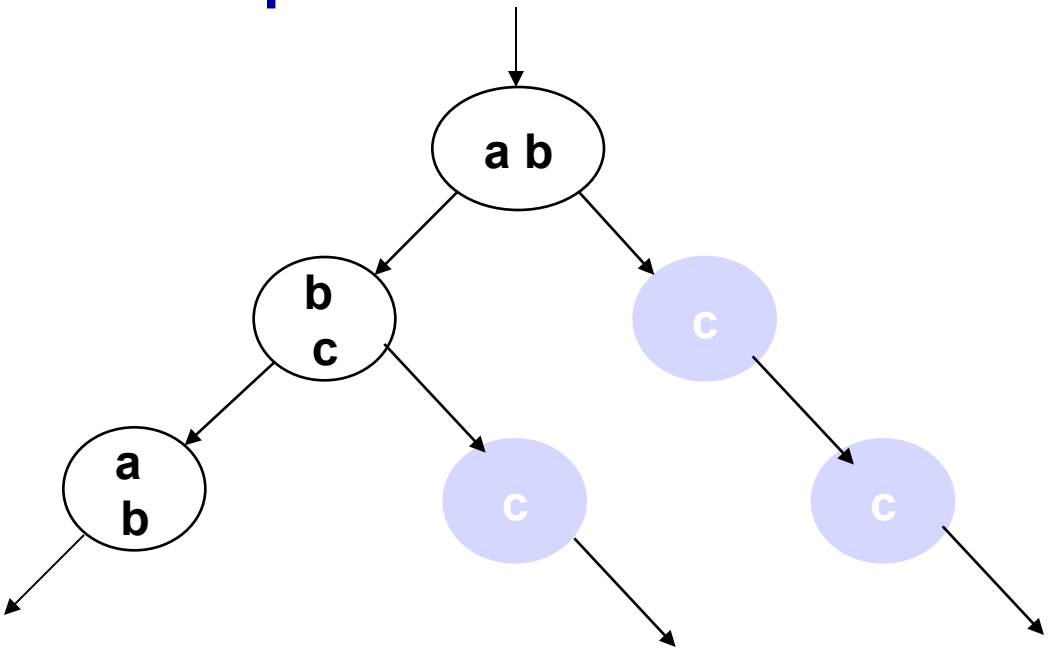
Run: $s_1 \rightarrow s_3 \rightarrow s_1 \rightarrow s_3 \rightarrow s_1 \rightarrow$ state sequence

Trace: a → b → a → b → a → observation sequence

Model of Computation



State Transition Graph

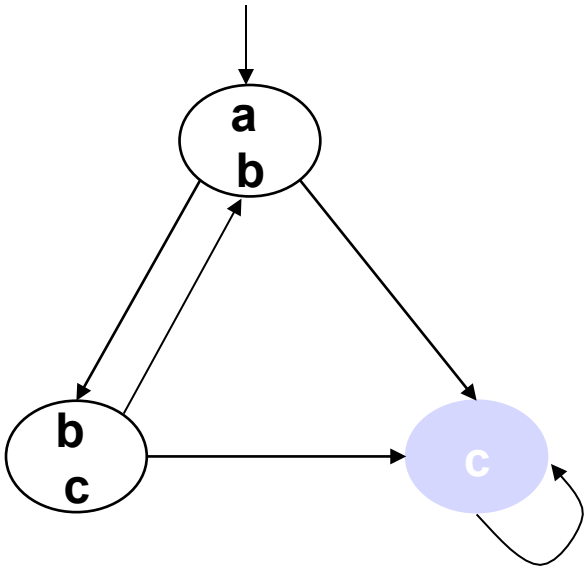


Infinite Computation Tree

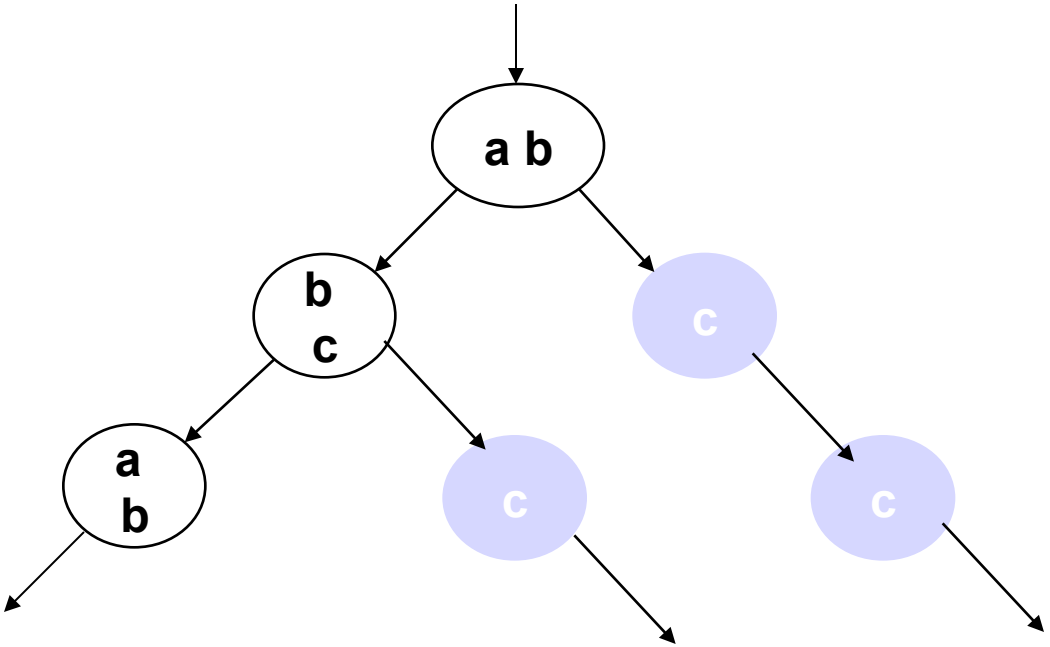
Unwind State Graph to obtain Infinite Tree.

A trace is an infinite sequence of state observations

Semantics



State Transition Graph



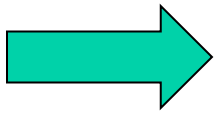
Infinite Computation Tree

The semantics of a FSM is a set of traces

Where is the model?

- Need to extract automatically
- Easier to construct from hardware
- Fundamental challenge for software

Linux Kernel
~1000,000 LOC
Recursion and data structures
Pointers and Dynamic memory
Processes and threads

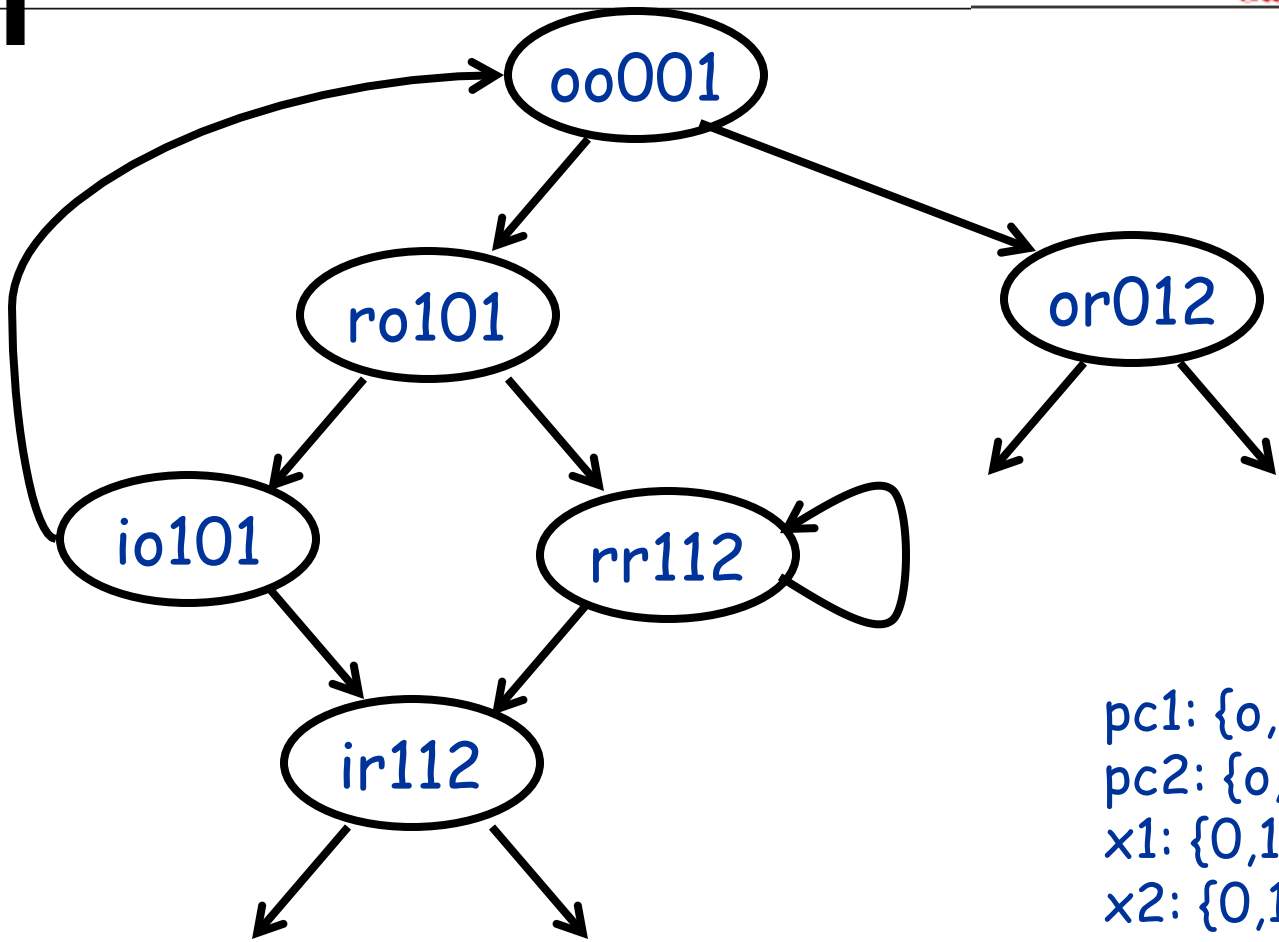


Finite State Model

Mutual-exclusion protocol

```
loop                                        ||   loop
   out:  x1 := 1; last := 1                       out:  x2 := 1; last := 2
   req:  await  x2 = 0  or  last = 2              req:  await  x1 = 0  or  last = 1
   in:     x1 := 0                                in:     x2 := 0
end loop.                                      end loop.
```

P1 P2



pc1: {o,r,i}
pc2: {o,r,i}
x1: {0,1}
x2: {0,1}
last: {1,2}

3·3·2·2·2 = 72 states

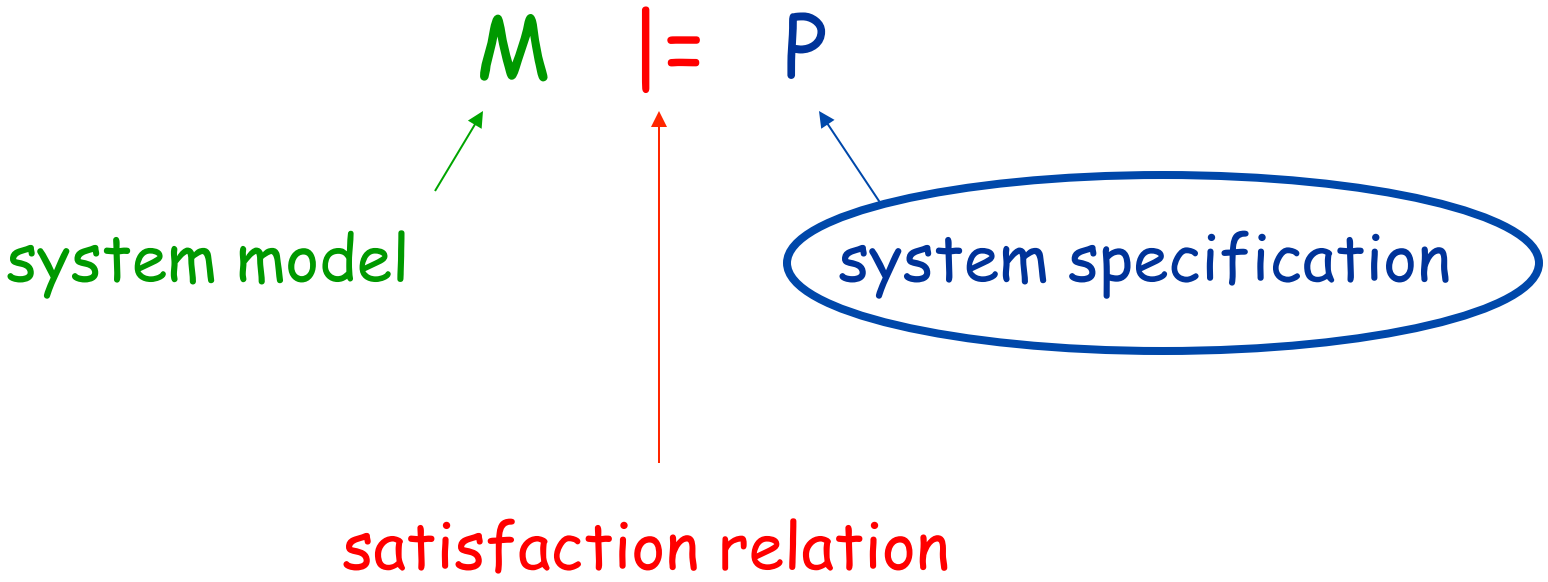
State space blow up

The translation from a system description to a state-transition graph usually involves an exponential blow-up !!!

e.g., n boolean variables $\Rightarrow$ 2^n states

This is called the "state-explosion problem."

Temporal Logic Model Checking



Decisions when choosing system properties:

- operational vs. declarative:
  automata vs. logic
- may vs. must:
  branching vs. linear time
- prohibiting bad vs. desiring good behavior:
  safety vs. liveness

System Properties/Specifications

- Atomic propositions: properties of states
- (Linear) Temporal Logic Specifications: properties of traces.

Specification (Property)

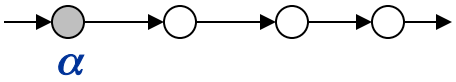
Examples:

Safety (mutual exclusion): no two processes can be at the critical section at the same time

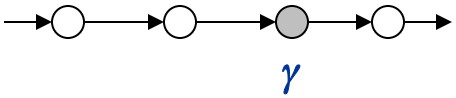
Liveness (absence of starvation): every request will be eventually granted

Linear Time Logic (LTL) [Pnueli 77]: logic of temporal sequences.

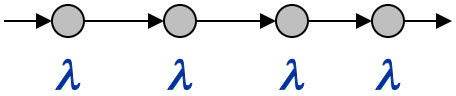
- ***next*** (α): α **holds in the next state**



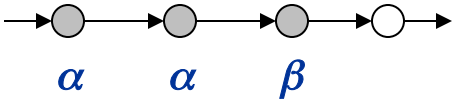
- ***eventually***(γ): γ **holds eventually**

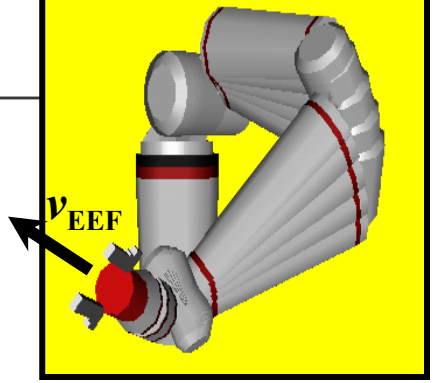


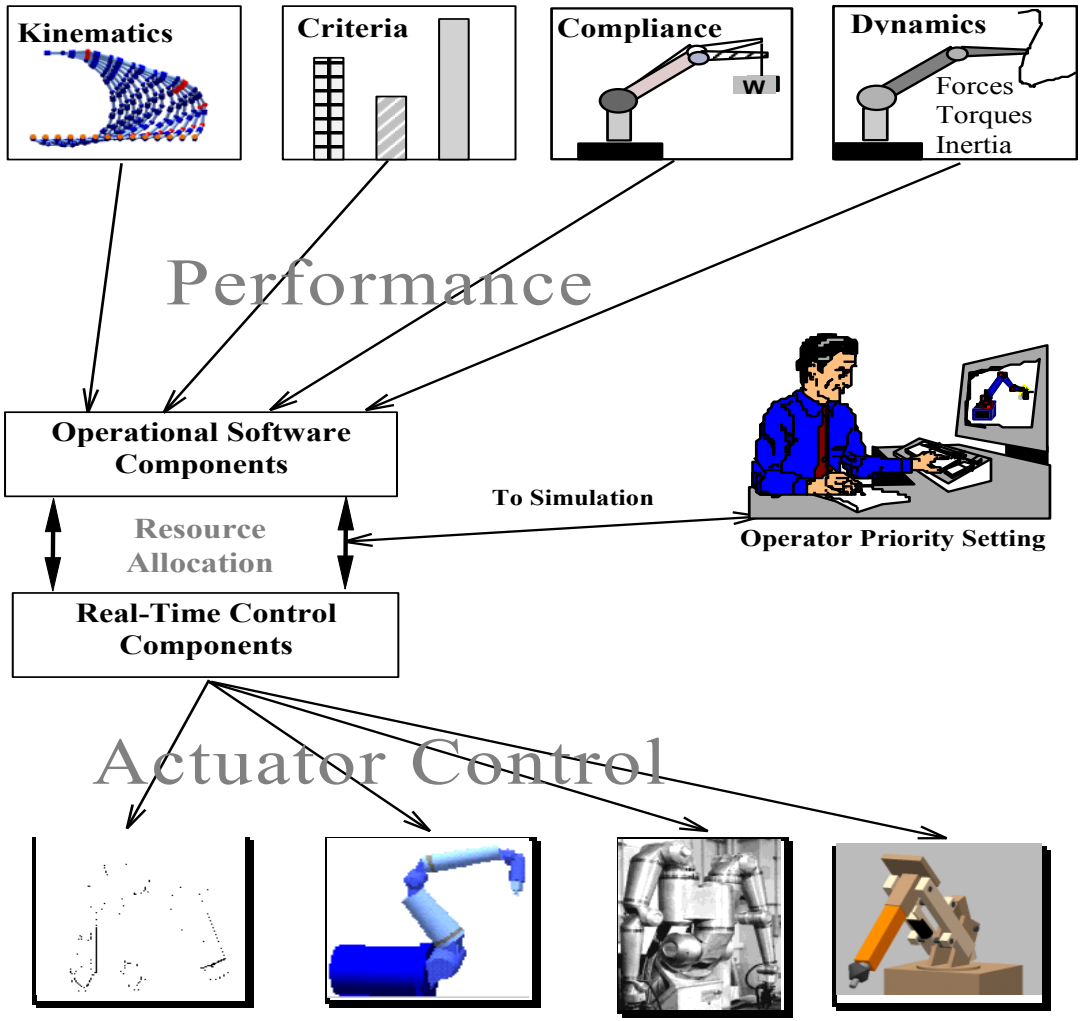
- ***always***(λ): λ **holds from now on**

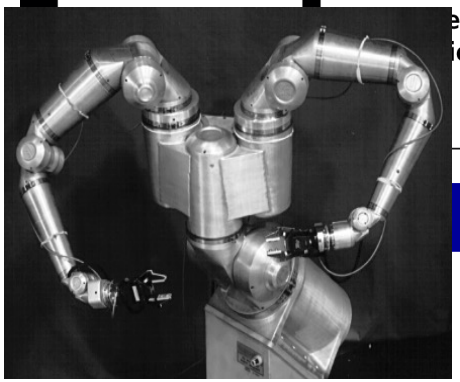


- α ***until*** β: α ***holds until*** β ***holds***




v_{EEF}


Kinematics
Criteria
Compliance
w
Dynamics
Forces
Torques
Inertia
Performance
Operational Software Components
To Simulation
Operator Priority Setting
Resource Allocation
Real-Time Control Components
Actuator Control



Examples of the Robot Control Properties

- ***Configuration Validity Check:***
  If an instance of *EndEffector* is in the "FollowingDesiredTrajectory" state, then the instance of the corresponding *Arm* class is in the 'Valid" state

  Always((ee_reference=1) ->(arm_status=1)

- ***Control Termination:*** Eventually the robot control terminates

  Eventually(abort_var=1)

What is "satisfy"?

M satisfies ***S*** if *all* the reachable states satisfy ***P***

Different Algorithms to check if *M* |= *P*.

- Explicit State Space Exploration

For example: Invariant checking Algorithm.

1. Start at the initial states and explore the states of ***M*** using DFS or BFS.
2. In any state, if ***P*** is violated then print an "error trace".
3. If all reachable states have been visited then say "yes".

State Space Explosion

Problem: Size of the state graph can be exponential in size of the program (both in the number of the program *variables* and the number of program *components*)

$$M = M_1 \,||\, \ldots \,||\, M_n$$

If each M_i has just 2 local states, potentially 2^n global states

Research Directions: State space reduction

Abstractions

- They are one of the most useful ways to **fight** the **state explosion problem**
- They should **preserve** **properties of interest:** properties that hold for the abstract model should hold for the concrete model
- Abstractions should be **constructed** **directly from the program**

Abstractions

- Why do we need to abstract?
  - To reduce a number of states
  - To represent (in a sound manner) infinite state systems as finite state systems

Abstractions

- Why we need to abstract?
  - To reduce a number of states
  - To represent (in a sound manner) infinite state systems as finite state systems
- How do we abstract?
  - By removing irrelevant to verification details

Data Abstraction

Given a program P with variables $x_1,...x_n$, each over domain D, the **concrete model** of P is defined over states $(d_1,...,d_n) \in D \times ... \times D$

Choosing

- Abstract domain **A**
- Abstraction mapping (surjection) **h: D → A**

we get an **abstract model** over abstract states $(a_1,...,a_n) \in A \times ... \times A$

Example

Given a program P with variable x over the integers

Abstraction 1:

$A_1 = \{ a_-, a_0, a_+ \}$

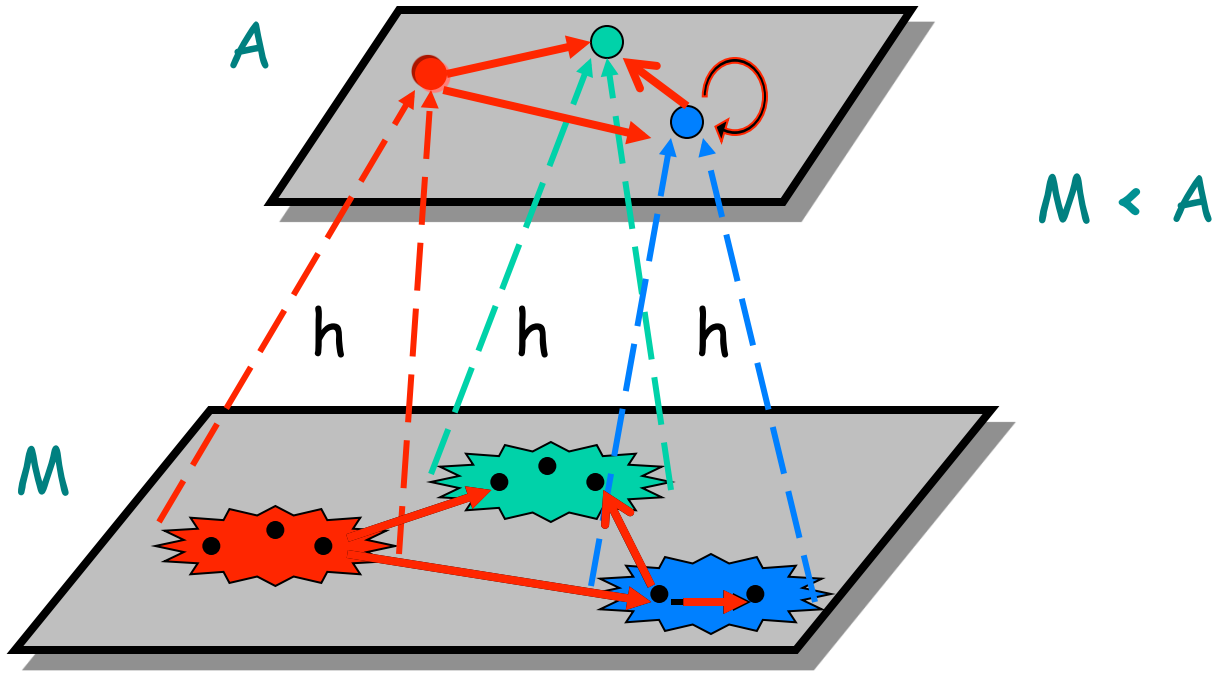
$$h_1(d) = \begin{cases} a_+ & \text{if } d>0 \\ a_0 & \text{if } d=0 \\ a_- & \text{if } d<0 \end{cases}$$

Abstraction 2:

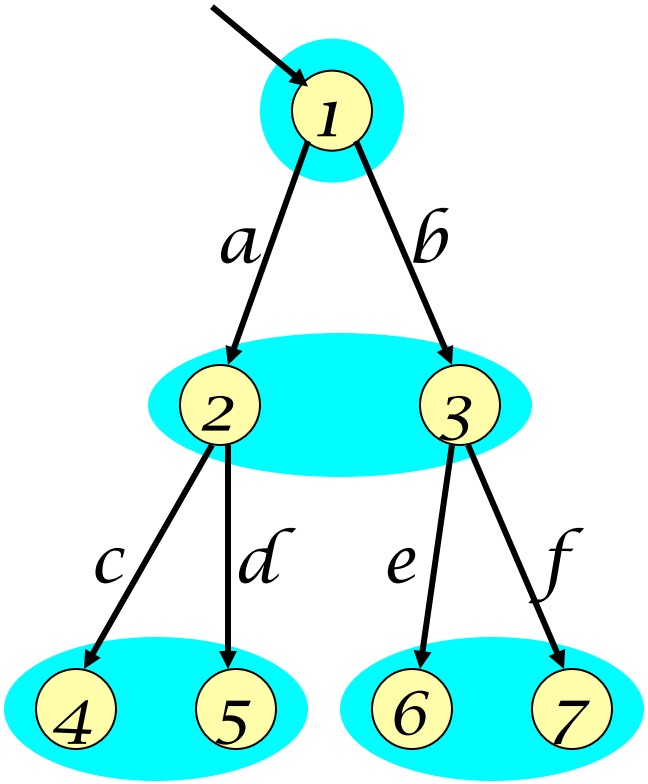
$A_2 = \{ a_{even}, a_{odd} \}$

$h_2(d) =$ if even($|d|$) then a_{even} else a_{odd}

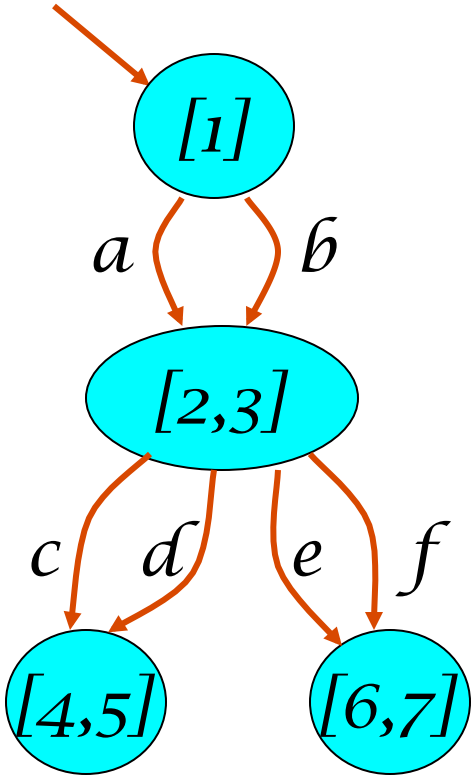
Existential Abstraction



Existential Abstraction



$\mathcal{M}$



$\mathcal{A}$

Existential Abstraction

- Every trace of $\mathcal{M}$ is a trace of $\mathcal{A}$

  – $\mathcal{A}$ over-approximates what $\mathcal{M}$ can do
  (Preserves safety properties!): $\mathcal{A}$ satisfies $\phi \Rightarrow \mathcal{M}$ satisfies ϕ

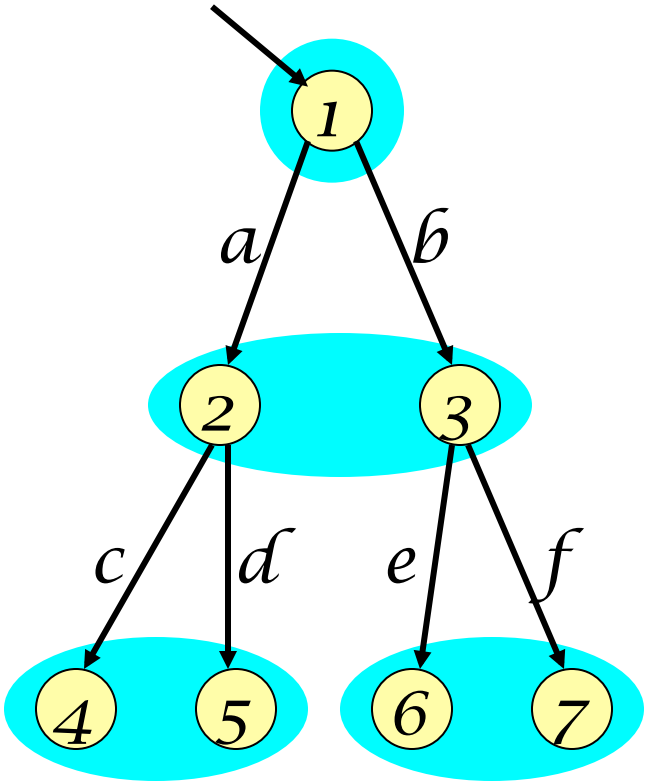
- Some traces of $\mathcal{A}$ may not be traces of $\mathcal{M}$

  – May yield spurious counterexamples - $< a, e >$

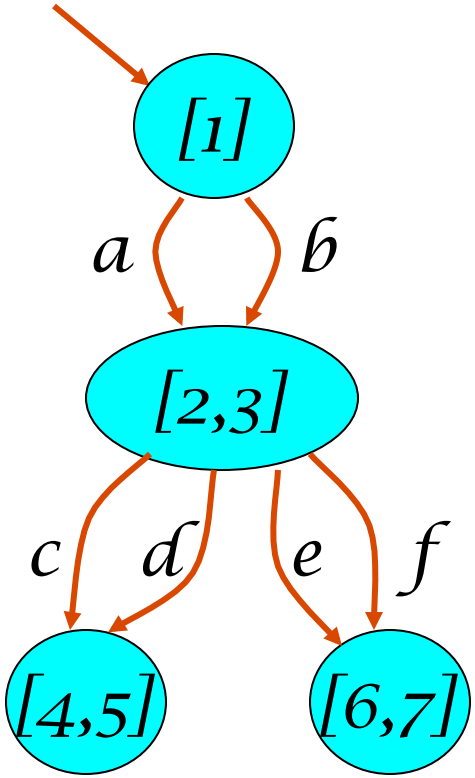
- Eliminated via abstraction refinement

  – Splitting some clusters in smaller ones
  – Refinement can be automated

Original Abstraction

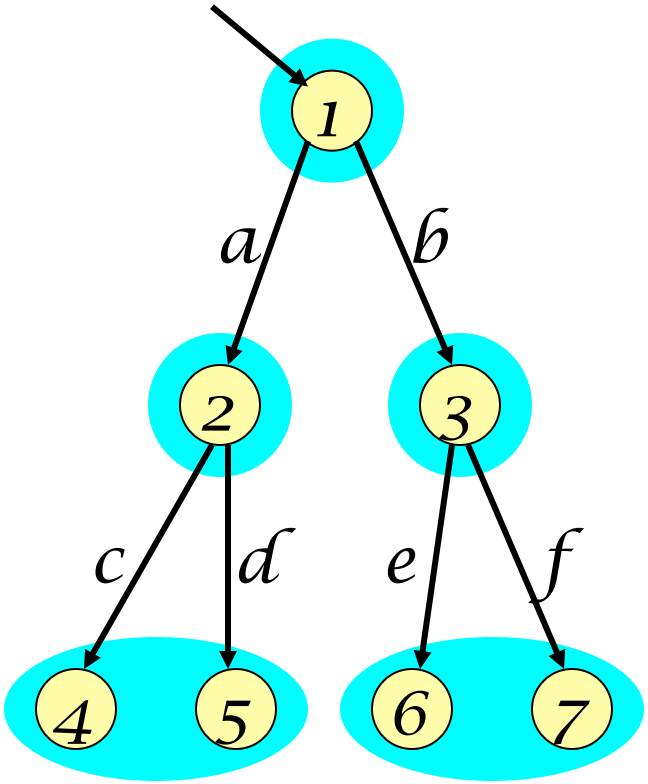


$\mathcal{M}$

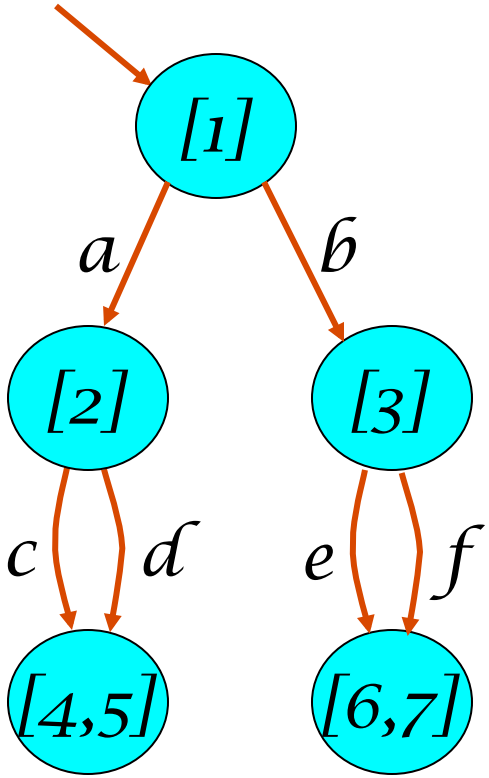


$\mathcal{A}$

Refined Abstraction



$\mathcal{M}$



$\mathcal{A}$

How to define an abstract model

Given M (model) and ϕ (spec), **choose**

- S_h - a set of abstract states
- **AP** – a set of atomic propositions that label concrete and abstract states
- **h** : $S \rightarrow S_h$ - a mapping from S on S_h that satisfies:

h(s) = h(t) only if L(s)=L(t)

Abstraction

Depending on h and the size of M, M_h (i.e., I_h, R_h) can be built using:

- BDDs or
- SAT solver or
- Theorem prover

Predicate Abstraction

[Graf/Saïdi 97]

- Idea: Only keep track of predicates on data

$$p_1(s), \ldots, p_n(s)$$

- Abstraction function:

$$h(s) = (p_1(s), p_2(s), \ldots, p_n(s))$$

Predicate Abstraction

- Given a program over variables **V**
- **Predicate** P_i is a first-order atomic formula over V
  Examples: $x+y < z^2$, $x=5$

- Choose: **AP** = { $P_1,\ldots,P_k$ } that includes
  - the atomic formulas in the property φ and
  - conditions in **if**, **while** statements of the program

Predicate Abstraction

Labeling of concrete states:

$L(s) = \{ P_i \mid s \models P_i \}$

Abstract Model

- Abstract states are defined over Boolean variables $\{ B_1,\ldots,B_k \}$:

$$S_h \subseteq \{ 0,1 \}^k$$

- $h(s) = s_h \Leftrightarrow$
  for all $1 \le j \le k$: $[\, s \models P_j \Leftrightarrow s_h \models B_j \,]$

- $L_h(s_h) = \{\ P_j \mid s_h \models B_j\ \}$

Example

Program over natural variables x, y

AP = { P1, P2, P3 }, where

P1 = $x \leq 1$, P2 = $x > y$, P3 = $y = 2$

AP = { $x \leq 1$, $x > y$, $y = 2$ }

For state s, where **s(x)=s(y)=0**: **L(s) = { P1 }**

For state t, where **t(x)=1, t(y)=2**: **L(t) = { P1,P3 }**

Example

$S_h \subseteq \{ 0,1 \}^3$

$h(s) = (1,0,0)$
$h(t) = (1,0,1)$

$L_h((1,0,0)) = \{ P_1 \}$
$L_h((1,0,1)) = \{ P_1, P_3 \}$

The concrete state and its abstract state are labeled identically

Computing abstract transition relation

$$(s_h, t_h) \in R_h \Leftrightarrow$$
$$\exists s,t \ [\ h(s) = s_h \wedge h(t) = t_h \wedge (s,t) \in R\ ]$$

Abstract transition relation

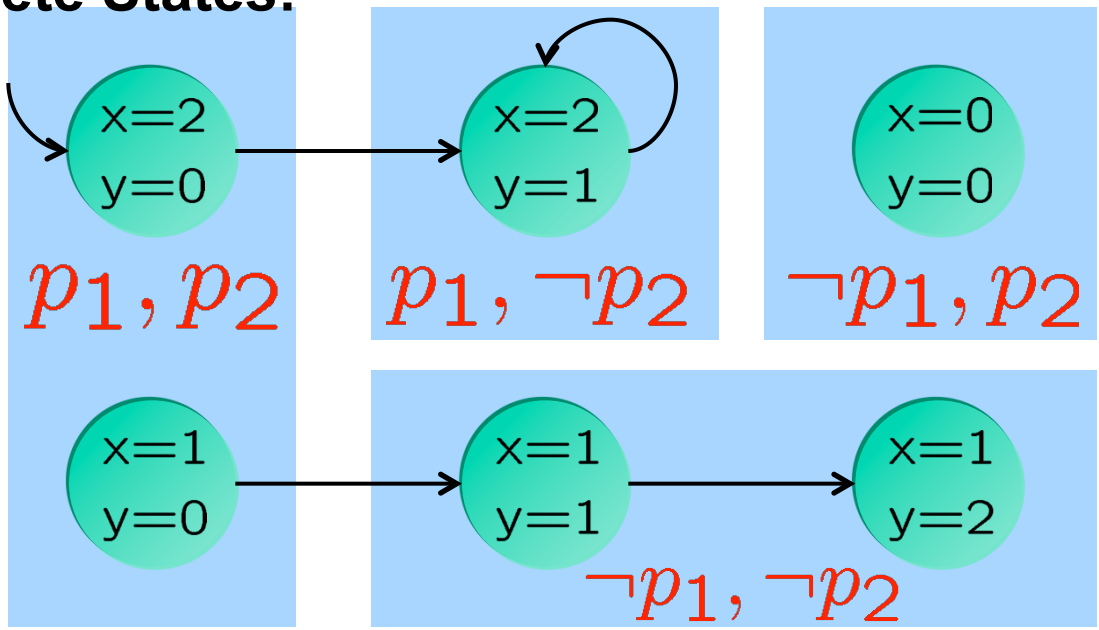
- Program with one statement: x := x+1

$$((b_1,b_2,b_3),(b'_1,b'_2,b'_3)) \in R_h \Leftrightarrow$$

$$\begin{aligned}\exists xyx'y' \; [\; & P_1(x,y) \Leftrightarrow b_1 \;\wedge \\ & P_2(x,y) \Leftrightarrow b_2 \;\wedge \\ & P_3(x,y) \Leftrightarrow b_3 \;\wedge \\ & x'=x+1 \;\wedge\; y'=y \;\wedge \\ & P_1(x',y') \Leftrightarrow b'_1 \;\wedge \\ & P_2(x',y') \Leftrightarrow b'_2 \;\wedge \\ & P_3(x',y') \Leftrightarrow b'_3 \;\;]\end{aligned}$$

Example

Concrete States:



Predicates:

$$p_1(s) = (s.x > s.y)$$
$$p_2(s) = (s.y = 0)$$

Abstract transitions?

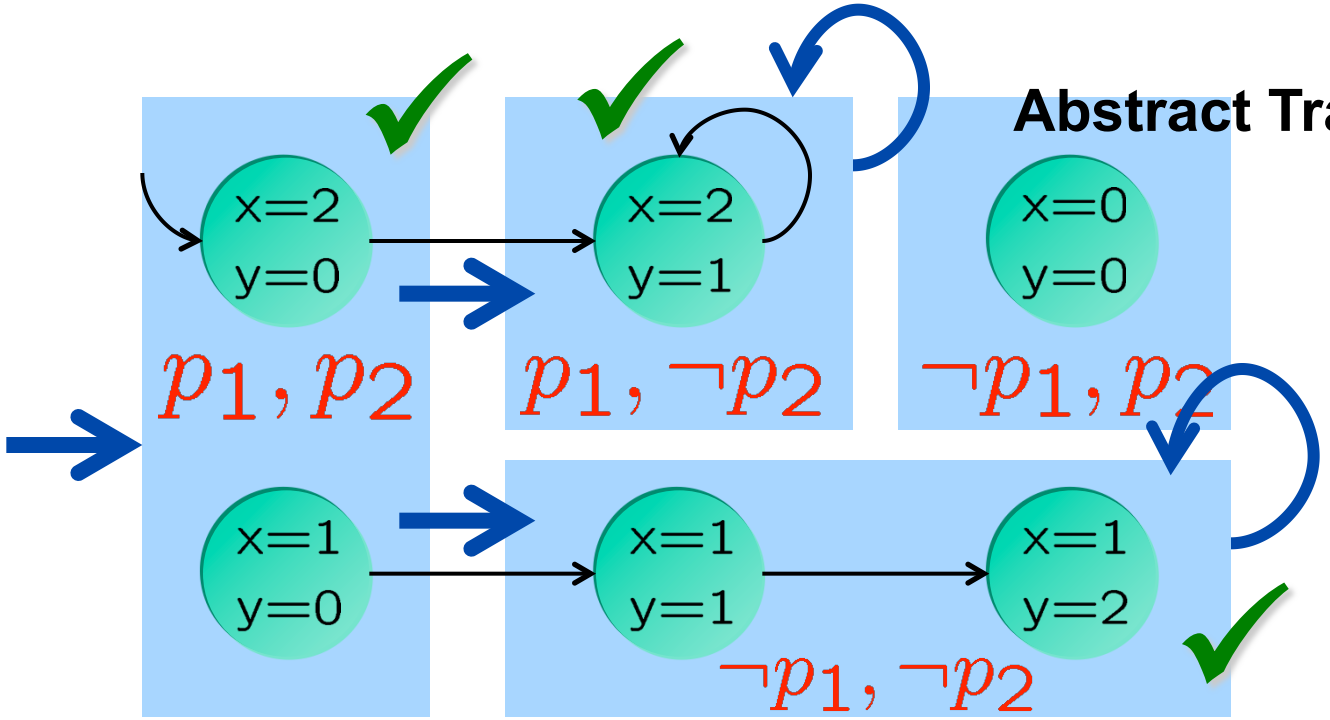
How to get abstract transitions?

- Typically done in a conservative manner
- Existential abstraction:

$$\hat{I}(\hat{s}) \quad :\Longleftrightarrow \quad \exists s : I(s)$$

$$\hat{R}(\hat{s}, \hat{s}') \quad :\Longleftrightarrow \quad \exists s, s' : R(s, s') \wedge h(s) = \hat{s} \wedge h(s') = \hat{s}'$$

Predicate Abstraction



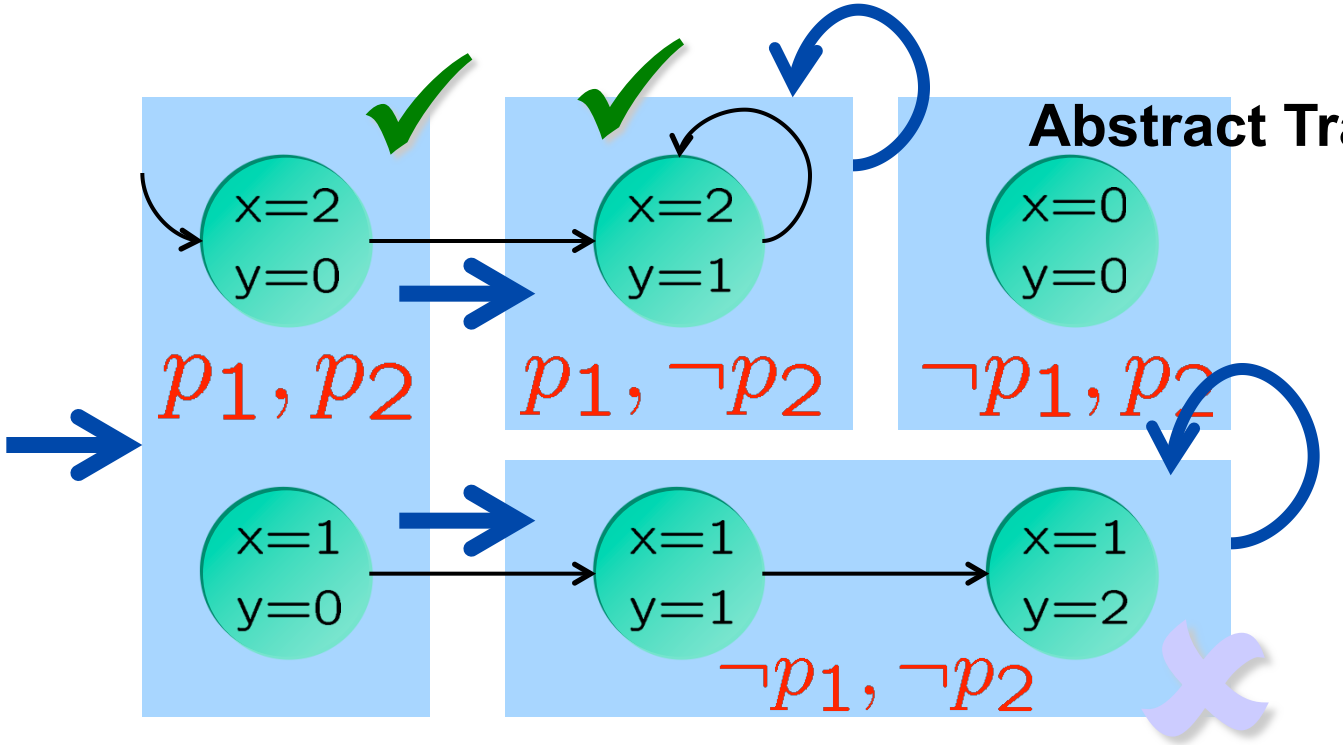
Abstract Transitions:

Property:

$$p_1 \vee \neg p_2 \quad \Longleftrightarrow \quad (s.x > s.y) \vee (s.y \neq 0)$$

Property holds. Ok.

Predicate Abstraction

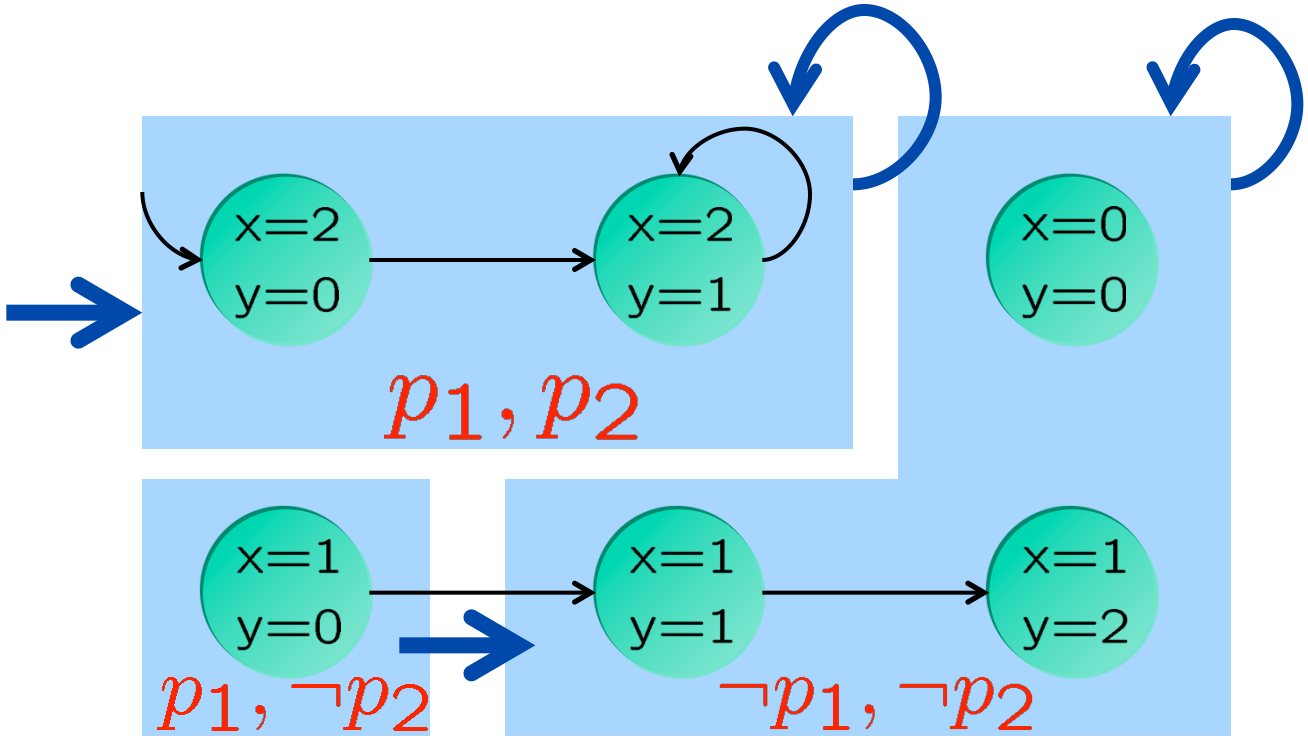


Property:

$$p_1 \iff (s.x > s.y)$$

This trace is spurious!

Predicate Abstraction



$$p_1 \iff (s.x > s.y)$$

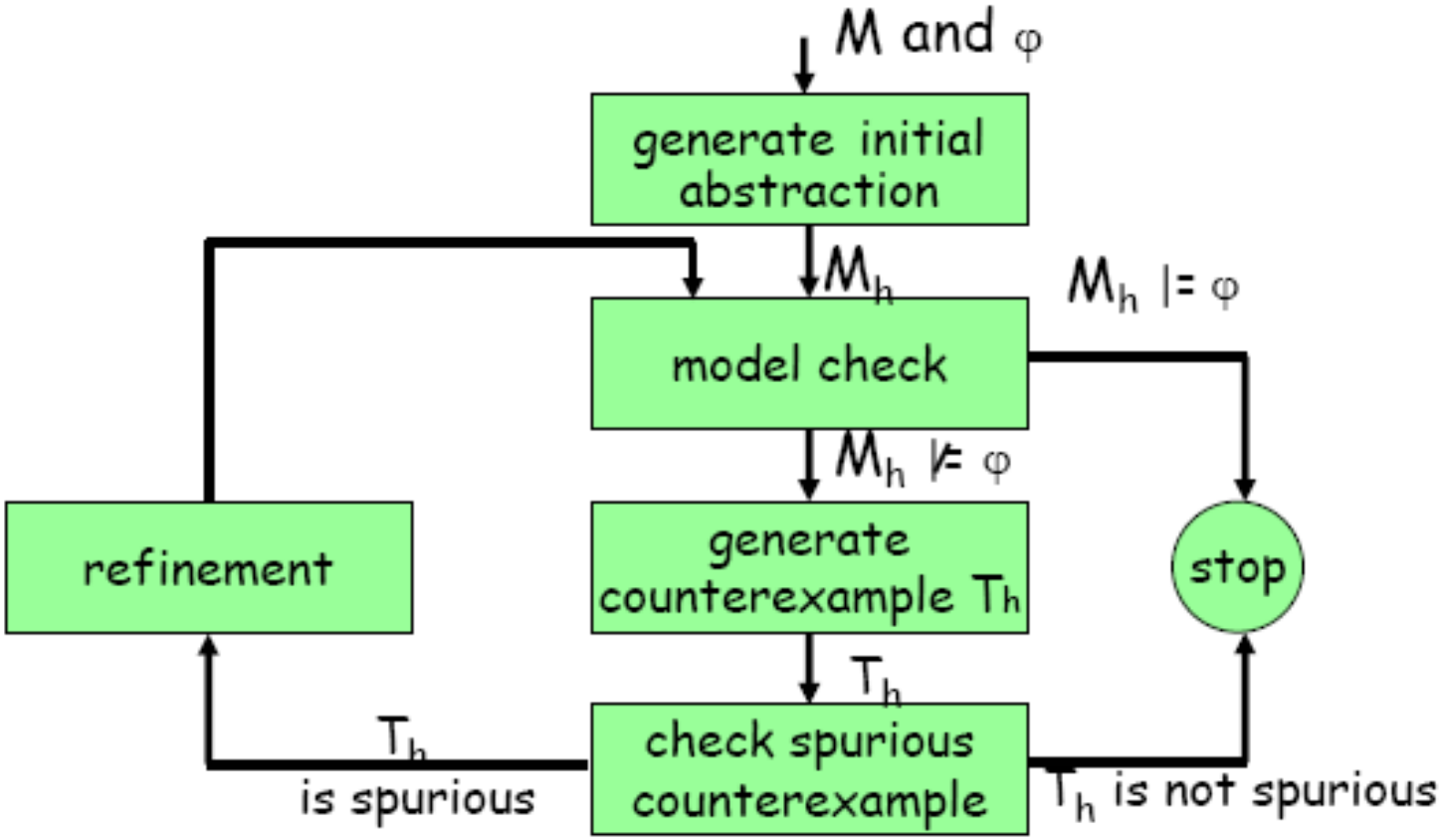
New Predicates:

$$p_1(s) = (s.x > s.y)$$
$$p_2(s) = (s.x = 2)$$

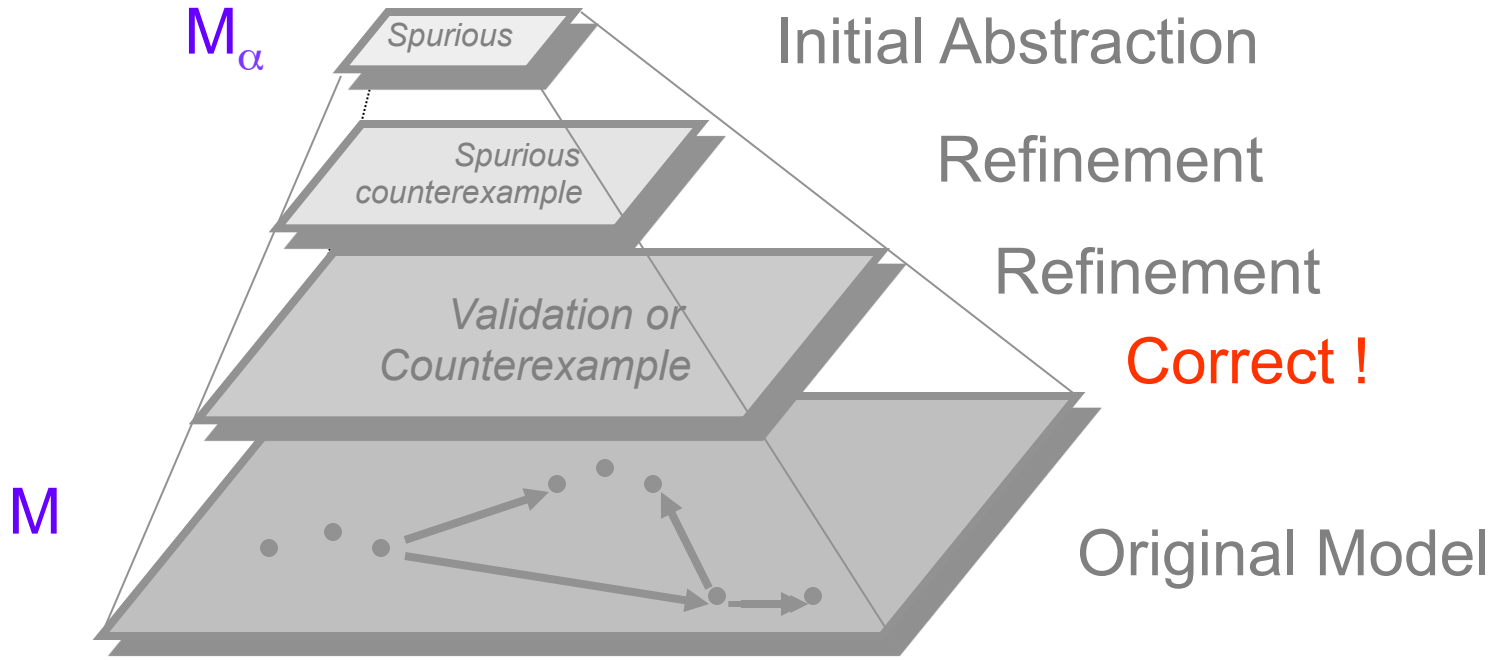
CEGAR

Counter Example Guided Abstraction Refinement

CEGAR approach



CEGAR


M_α
Spurious
Spurious counterexample
Validation or Counterexample
M
Initial Abstraction
Refinement
Refinement
Correct !
Original Model