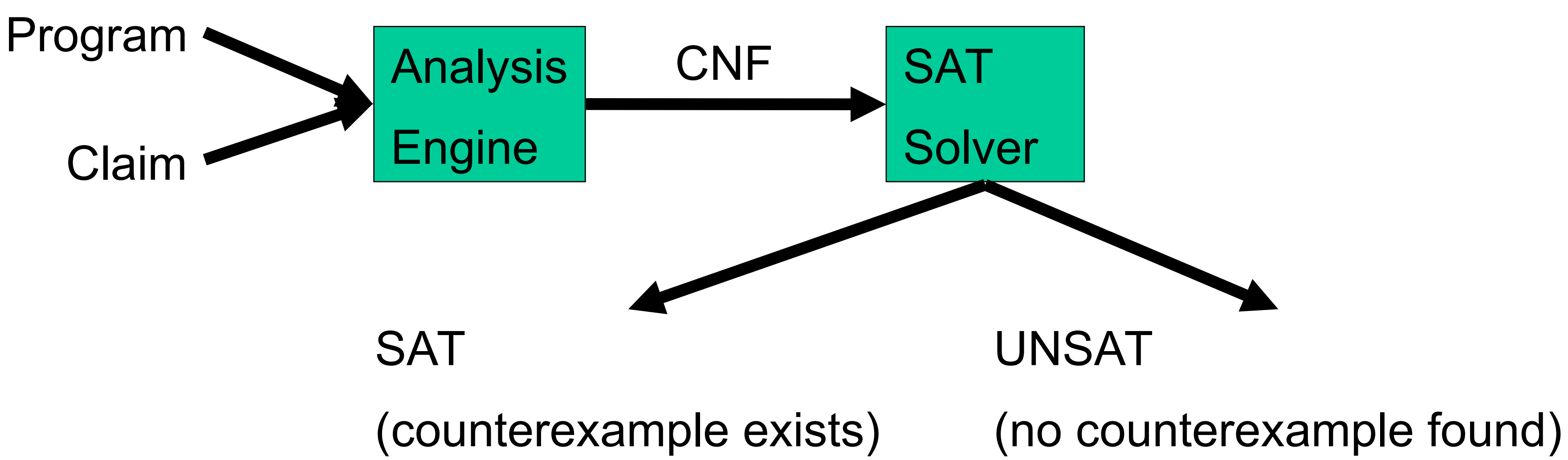
Model Checking Java Programs (Java PathFinder)

Slides partially compiled from the NASA JavaPathFinder project and E. Clarke's course material

Java PathFinder

- JPF is an explicit state software model checker for Java bytecode
  - JPF is a Java virtual machine that executes your program not just once (like a normal VM), but theoretically in all possible ways, checking for property violations like deadlocks or unhandled exceptions along all potential execution paths.

Symbolic Model Checking



Explicit State Model Checking

- The program is indeed executing
  - jpf <your class> <parameters>
    - Very similar to “java <your class> <parameters>”
  - Execute in a way that all possible scenarios are explored
    - Thread interleaving
    - Undeterministic values (random values)
  - Concrete input is provided
  - A state is indeed a concrete state, consisting of
    - Concrete values in heap/stack memory

JPF Status

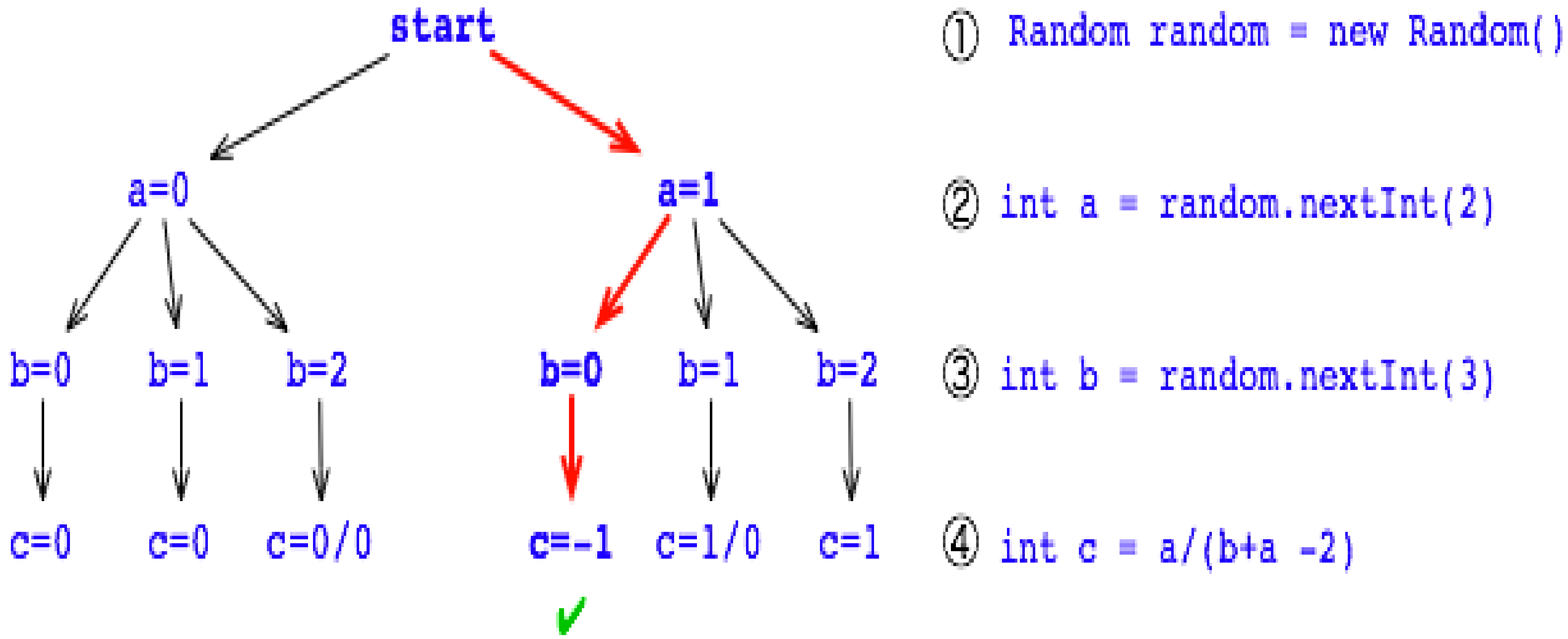
- developed at the Robust Software Engineering Group at NASA Ames Research Center
- currently in it’s fourth development cycle
  - v1: Spin/Promela translator - 1999
  - v2: backtrackable, state matching JVM - 2000
  - v3: extension infrastructure (listeners, MJI) - 2004
  - v4: symbolic execution, choice generators - 4Q 2005
- open sourced since 04/2005 under NOSA 1.3 license: http://javapathfinder.sourceforge.net
- First NASA-developed system hosted on public site before

An Example

```java
import java.util.Random;

public class Rand {
	public static void main (String[] args) {
		Random random = new Random(42);		// (1)

		int a = random.nextInt(2);			// (2)
		System.out.println("a=" + a);

		//... lots of code here

		int b = random.nextInt(3);			// (3)
		System.out.println("  b=" + b);

		int c = a/(b+a -2);				// (4)
		System.out.println("    c=" + c);
	}
}
```

```
> java Rand
a=1
  b=0
    c=-1
>
```

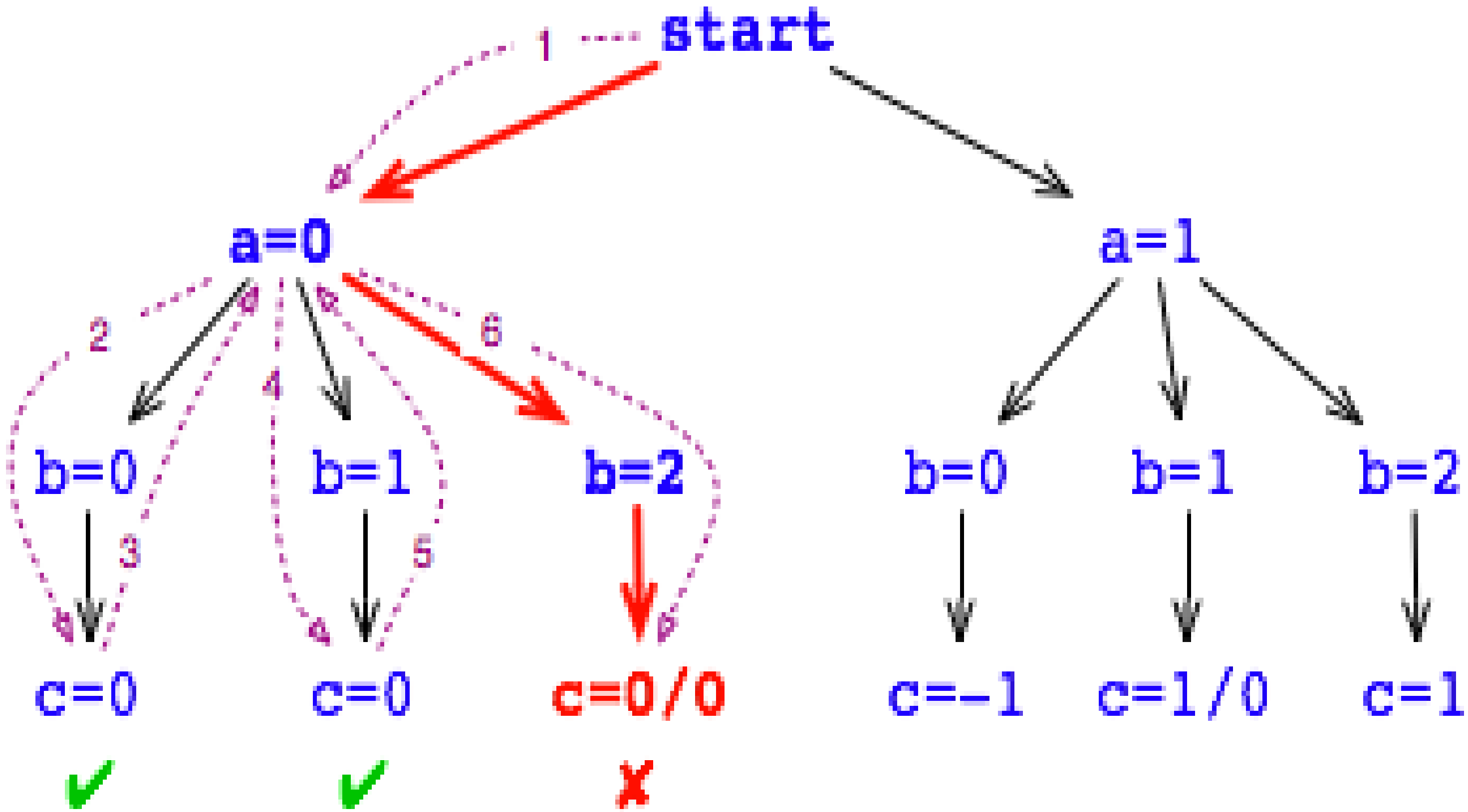
An Example (cont.)



- **One execution corresponds to one path.**

```
> bin/jpf Rand

JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
====================================================== system under test
application: /Users/pcmehlitz/tmp/Rand.java

====================================================== search started: 5/23/07 11:48 PM
a=1
  b=0
    c=-1

====================================================== results
no errors detected

====================================================== search finished: 5/23/07 11:48 PM
>
```

```
> bin/jpf +vm.enumerate_random=true Rand
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
====================================================== system under test
application: /Users/pcmehlitz/tmp/Rand.java

====================================================== search started: 5/23/07 11:49 PM
a=0
  b=0
    c=0
  b=1
    c=0
  b=2

====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
        at Rand.main(Rand.java:15)
....
>
```



- **JPF explores multiple possible executions GIVEN THE SAME CONCRETE INPUT**

Another Example

```
public class Racer implements Runnable {

     int d = 42;

     public void run () {
          doSomething(1000);                      // (1)
          d = 0;                                  // (2)
     }

     public static void main (String[] args){
          Racer racer = new Racer();
          Thread t = new Thread(racer);
          t.start();

          doSomething(1000);                      // (3)
          int c = 420 / racer.d;                  // (4)
          System.out.println(c);
     }

     static void doSomething (int n) {
          // not very interesting..
          try { Thread.sleep(n); } catch (InterruptedException ix) {}
     }
}
```

```
> bin/jpf Racer
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
===================================================== system under
application: /Users/pcmehlitz/tmp/Racer.java

===================================================== search starte
10
10

===================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
        at Racer.main(Racer.java:20)

==================================================== trace #1
---------------------------------------------------- transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
      [282 insn w/o sources]
  Racer.java:15                 : Racer racer = new Racer();
  Racer.java:1                  : public class Racer implements Runnable {
      [1 insn w/o sources]
  Racer.java:3                  : int d = 42;
  Racer.java:15                 : Racer racer = new Racer();
  Racer.java:16                 : Thread t = new Thread(racer);
      [51 insn w/o sources]
  Racer.java:16                 : Thread t = new Thread(racer);
  Racer.java:17                 : t.start();
---------------------------------------------------- transition #1 thread: 0
```

```
------------------------------------------------------ transition #1 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
  Racer.java:17                    : t.start();
  Racer.java:19                    : doSomething(1000);                    // (3)
  Racer.java:6                     : try { Thread.sleep(n); } catch (InterruptedException ix) {}
      [2 insn w/o sources]
  Racer.java:6                     : try { Thread.sleep(n); } catch (InterruptedException ix) {}
  Racer.java:7                     : }
  Racer.java:20                    : int c = 420 / racer.d;                // (4)
------------------------------------------------------ transition #2 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,>Thread-0}
  Racer.java:10                    : doSomething(1000);                    // (1)
  Racer.java:6                     : try { Thread.sleep(n); } catch (InterruptedException ix) {}
      [2 insn w/o sources]
  Racer.java:6                     : try { Thread.sleep(n); } catch (InterruptedException ix) {}
  Racer.java:7                     : }
  Racer.java:11                    : d = 0;                                // (2)
------------------------------------------------------ transition #3 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,>Thread-0}
  Racer.java:11                    : d = 0;                                // (2)
  Racer.java:12                    : }
------------------------------------------------------ transition #4 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
  Racer.java:20                    : int c = 420 / racer.d;                // (4)

====================================================== search finished: 5/24/07 12:32 AM
>
```

Two Essential Capabilities

- Backtracking
  - Means that JPF can restore previous execution states, to see if there are unexplored choices left.
    - While this can theoretically be achieved by re-executing the program from the beginning, backtracking is a much more efficient mechanism if state storage is optimized.
- State matching
  - JPF checks every new state if it already has seen an equal one, in which case there is no use to continue along the current execution path, and JPF can backtrack to the nearest non-explored non-deterministic choice
    - Heap and thread-stack snapshots.

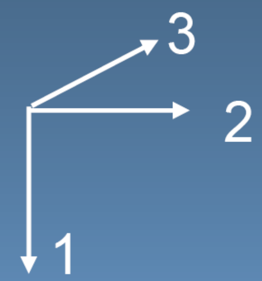
The Challenge

```
int   x,  y,  r;
int  *p, *q, *z;
int **a;

thread_1(void)      /* initialize p, q, and r */
{
    p = &x;
    q = &y;
    z = &r;
}
thread_2(void)      /* swap contents of x and y */
{
    r = *p;
    *p = *q;
    *q = r;
}
thread_3(void)      /* access z via a and p */
{
    a = &p;
    *a = z;
    **a = 12;
}
```

3 asynchronous threads
accessing shared data
3 statements each
how many test runs are needed to
check that no data corruption can occur?

The Challenge (cont.)

- the number of possible thread interleavings is...

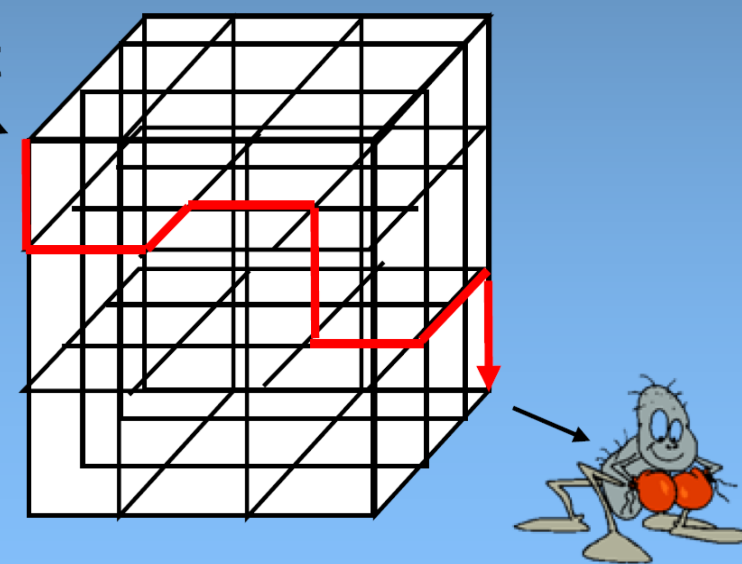


$$\frac{9!}{6!.3!} \cdot \frac{6!}{3!.3!} \cdot \frac{3!}{3!} =$$ 1,680 possible executions

placing 3 sets of 3 tokens in 9 slots

- are all these executions okay?
- can we check them all? should we check them all?
- in classic system testing, how many would normally be checked?

start

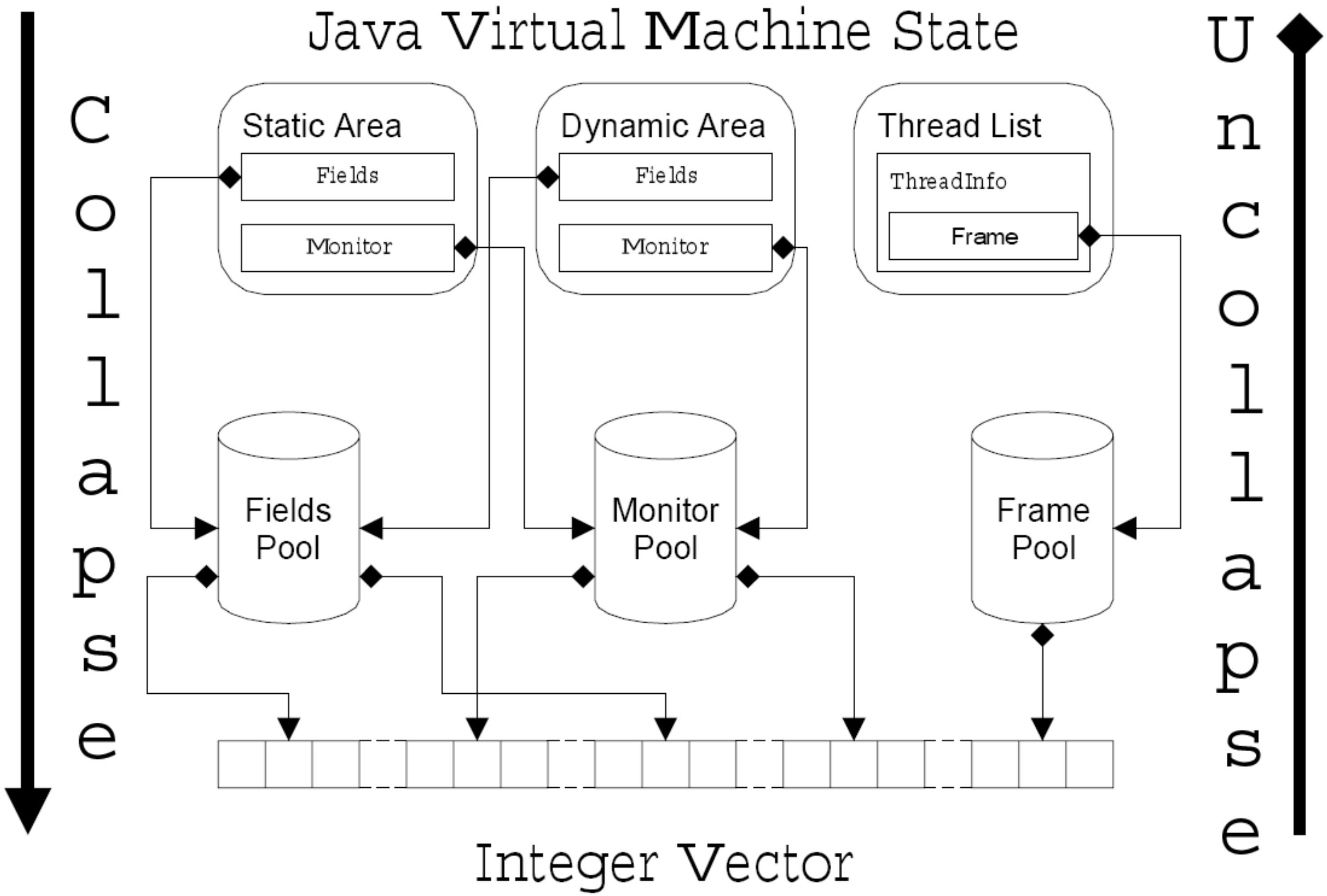


State Explosion!!

JPF’s Approach

- Configurable search strategy
  - Directing the search so that defects can be found quicker
    - A debugging tool instead of a “proof” system.
  - User can easily develop his/her own strategy
- Host VM Execution
  - Delegate execution to the underlying host VM (no state tracking).
- Reducing state storage
  - State collapsing
    - Premise: only a tiny part of the state is changed upon each transaction. (e.g. a single stack frame)
    - Dividing a state into components, use hashtable to index a specific value for a component.

Solution – State Collapsing



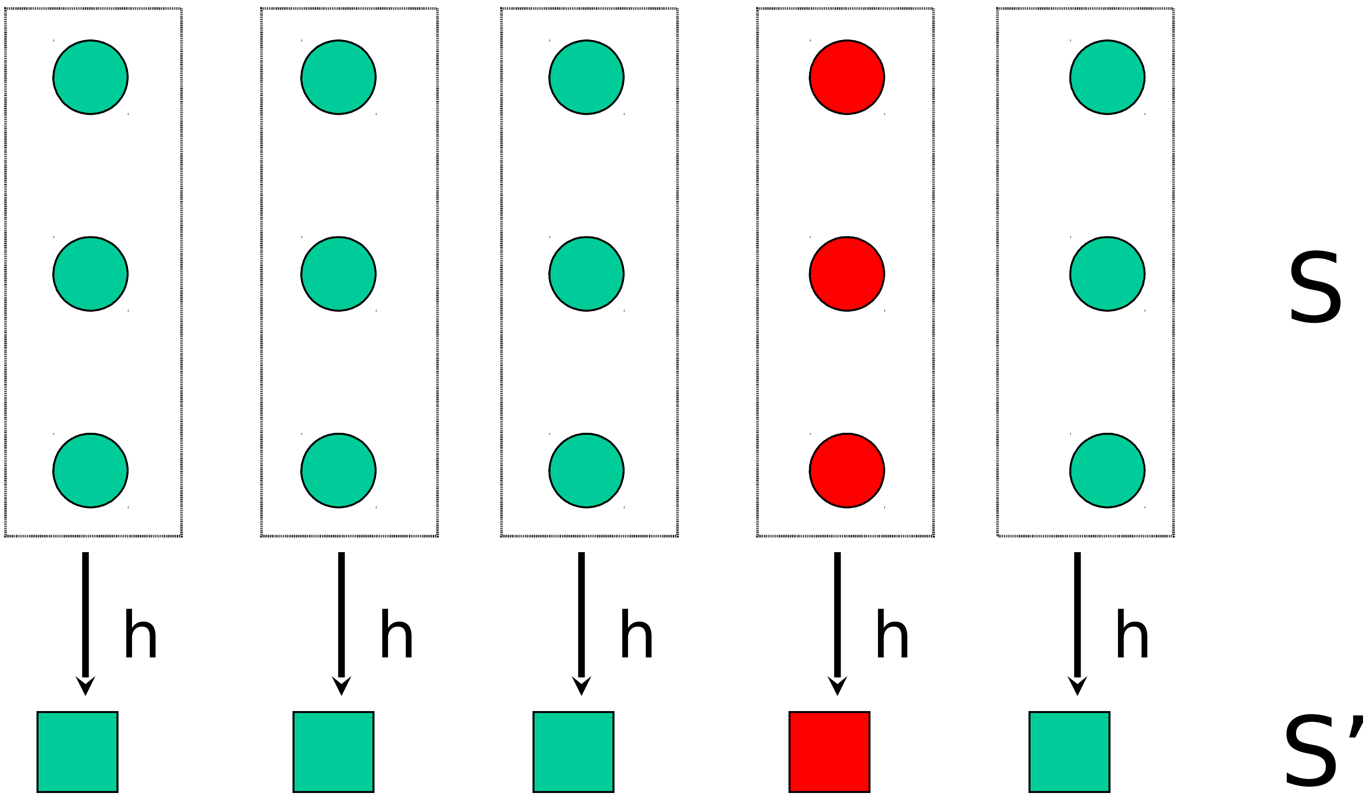
Solution – State Reduction

- Orthogonal (our focus)
  - **State Abstraction**
  - **Partial Order Reduction**

Abstraction

- Eliminate details irrelevant to the property
- Obtain simple finite models sufficient to verify the property
- Disadvantage
  - Loss of Precision: False positives/negatives

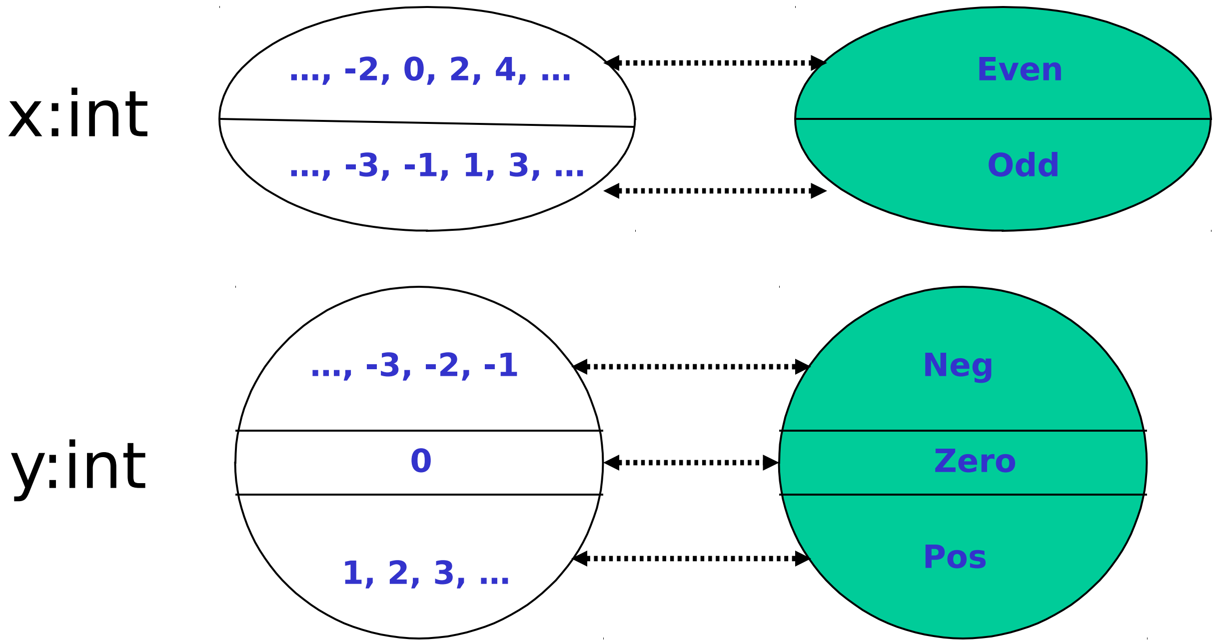
Data Abstraction



Abstraction Function h : from S to S’

Data Abstraction Example

- Abstraction proceeds component-wise, where variables are components



How do we Abstract Behaviors?

- Abstract domain A
  - Abstract concrete values to those in A

- Then compute transitions in the abstract domain

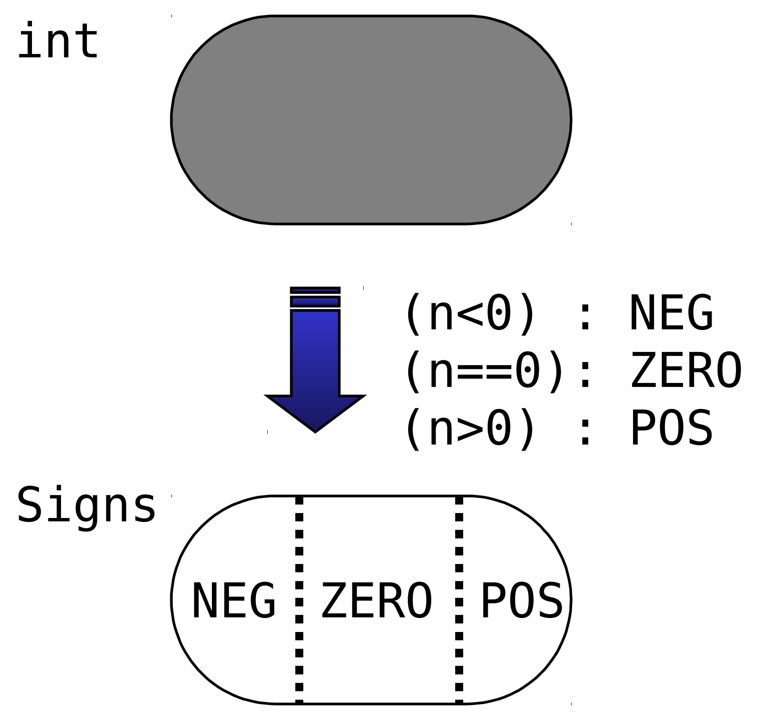
Data Type Abstraction

Code

```
int x = 0;
if (x == 0)
  x = x + 1;
```

```
Signs x = ZERO;
if (Signs.eq(x,ZERO))
  x = Signs.add(x,POS);
```

Abstract Data domain

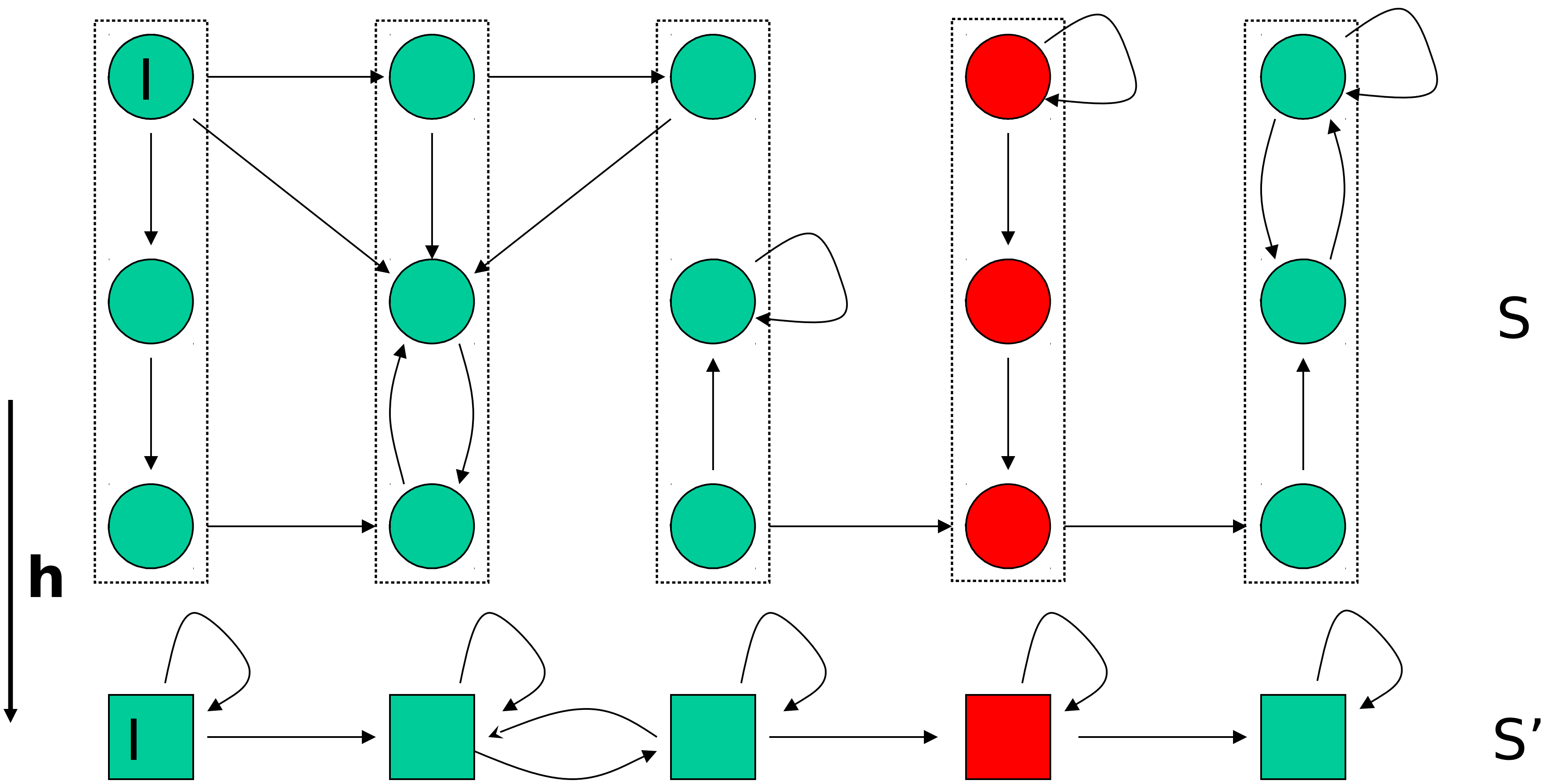


Existential/Universal Abstractions

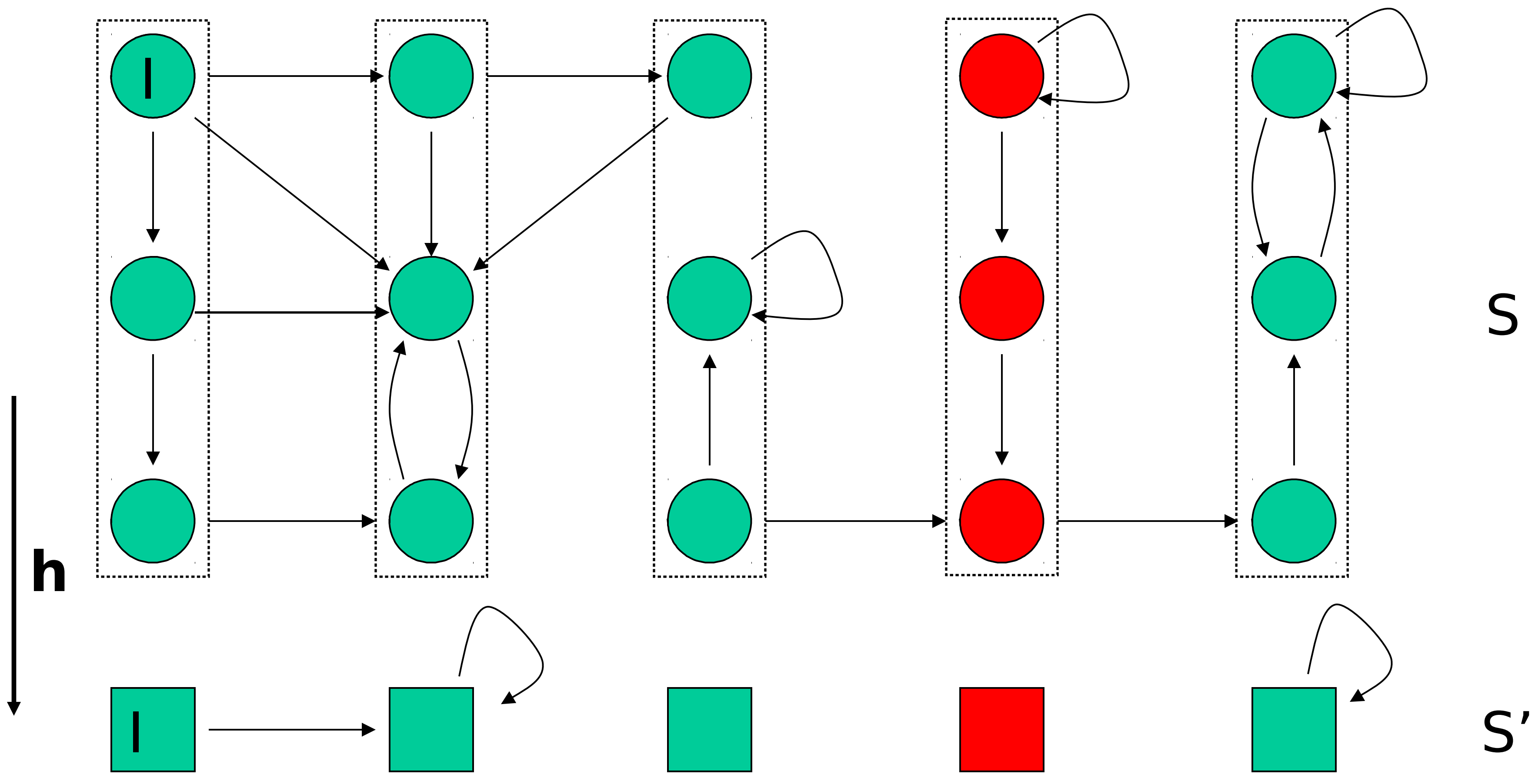
- Existential
  - Make a transition from an abstract state if ***at least one*** corresponding concrete state has the transition.
  - Abstract model M' simulates concrete model M

- Universal
  - Make a transition from an abstract state if ***all*** the corresponding concrete states have the transition.

Existential Abstraction (Over-approximation)



Universal Abstraction (Under-Approximation)



Guarantees from Abstraction

Assume M' is an abstraction of M

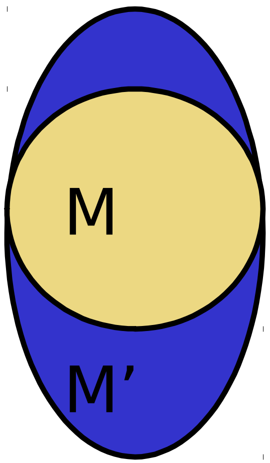
- Strong Preservation:
  - P holds in M' iff P holds in M
- Weak Preservation:
  - P holds in M' implies P holds in M

Guarantees from Exist. Abstraction

- Let φ be a *hold-for-all-paths* property
- M’ existentially abstracts M

- Preservation Theorem

$$M' \not\models \varphi \;\rightarrow\; M \not\models \varphi$$



- Converse does not hold

$$M' \not\models \varphi \;\not\rightarrow\; M \not\models \varphi$$

- $M' \not\models \varphi$: counterexample may be spurious

Guarantees from Univ. Abstraction

- Let φ be an existential-quantified property and M simulates M'

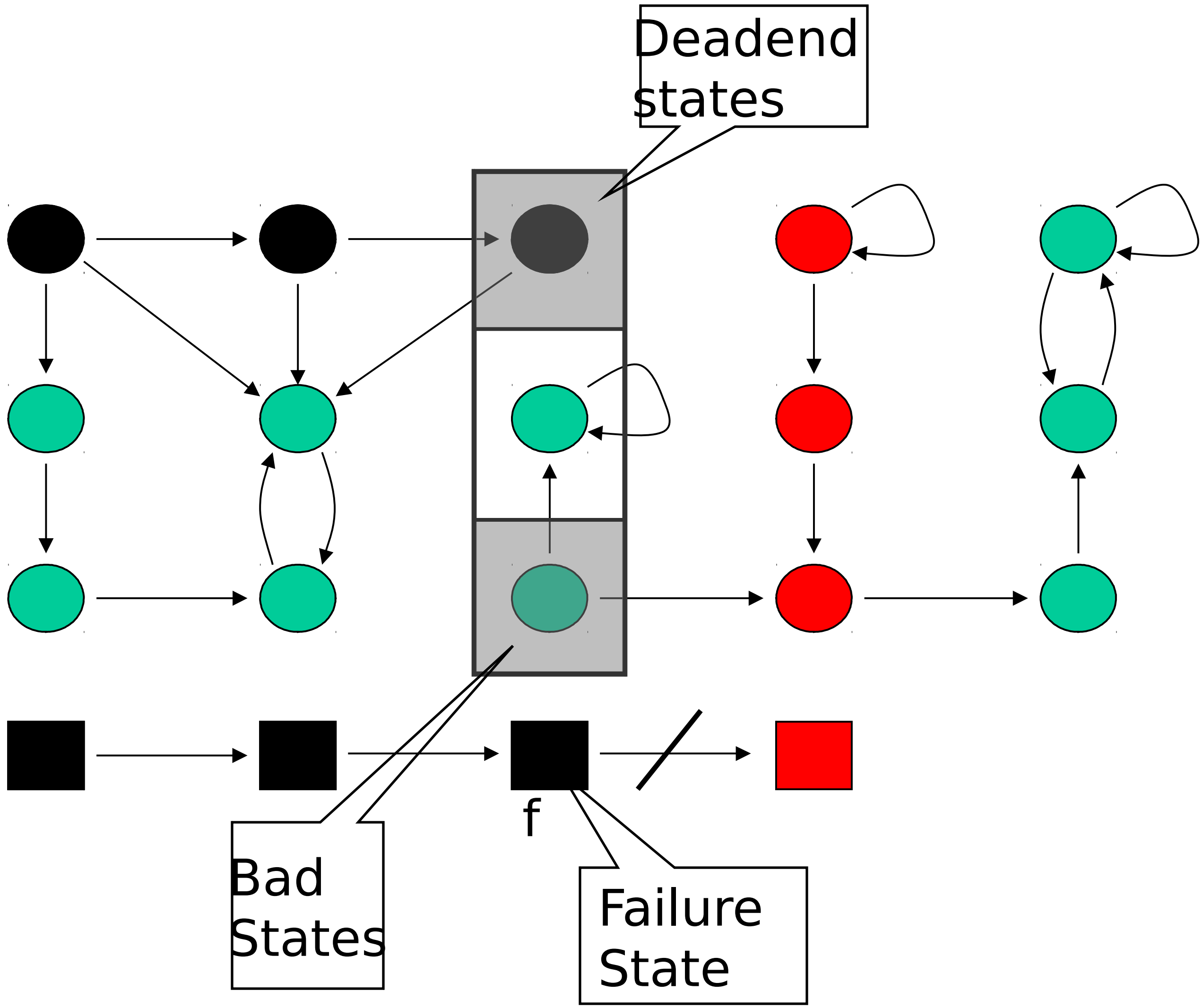
- Preservation Theorem

$$M' \not\models \varphi \;\rightarrow\; M \not\models \varphi$$

- Converse does not hold

$$M \not\models \varphi \;\rightarrow\; M' \not\models \varphi$$

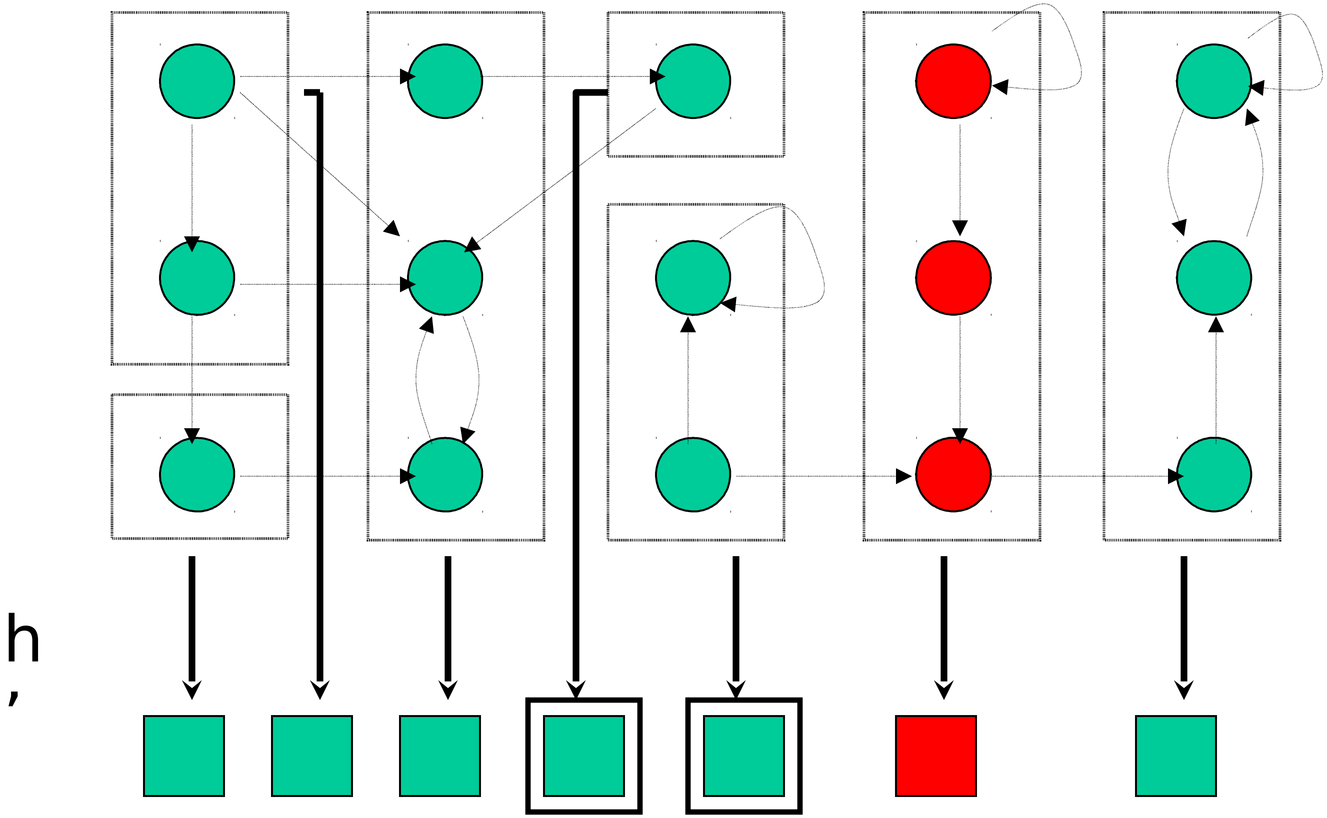
Spurious counterexample in Over-approximation



Refinement

- Problem: Deadend and Bad States are in the same abstract state.
- Solution: Refine abstraction function.
- The sets of Deadend and Bad states should be separated into different abstract states.

Refinement



Refinement : h’

Automated Abstraction/Refinement

- Good abstractions are hard to obtain
  - Automate both Abstraction and Refinement processes
- Counterexample-Guided AR (CEGAR)
  - Build an abstract model M’
  - Model check property P, M’ ⊨ P?
  - If M’ ⊨ P, then M ⊨ P by Preservation Theorem
  - Otherwise, check if Counterexample (CE) is spurious
  - Refine abstract state space using CE analysis results
  - Repeat

Counterexample-Guided Abstraction-Refinement (CEGAR)