

# **Java PathFinder**

## **Corso di programmazione concorrente e algoritmi distribuiti**

**Angelo Ferrando**

# Java PathFinder

- Sviluppato nel 2007, da una partnership tra West Virginia University (WVU) e il NASA Independent Verification and Validation (IV&V).
- Obiettivo: Studiare tecniche di Software Engineering per migliorare il modo in cui la NASA produce sistemi software sicuri e affidabili.
- JPF è un Software Model Checker di Java bytecode
  - JPF è una Java virtual machine che esegue il programma non solo una volta (come una virtual machine normale), ma teoricamente in tutti i modi possibili, controllando per violazioni di proprietà come deadlocks, eccezioni non gestite, o violazioni di asserzioni in tutti i potenziali cammini di esecuzione.
  - Storicamente, JPF era originariamente un traduttore da Java a PROMELA - il linguaggio di input utilizzato da SPIN (model checker)

# **Java PathFinder: Verifica del software**

- Costosa dal punto di vista del tempo di esecuzione
  - Difficile da fare
  - Non sempre è possibile verificare totalmente il sistema sotto analisi
- In particolare, si vorrebbe qualcosa di:
  - più veloce
  - più facile da usare
  - che catturi più errori possibile
  - Soluzione?
    - Java PathFinder!

# **Java PathFinder: Stato corrente**

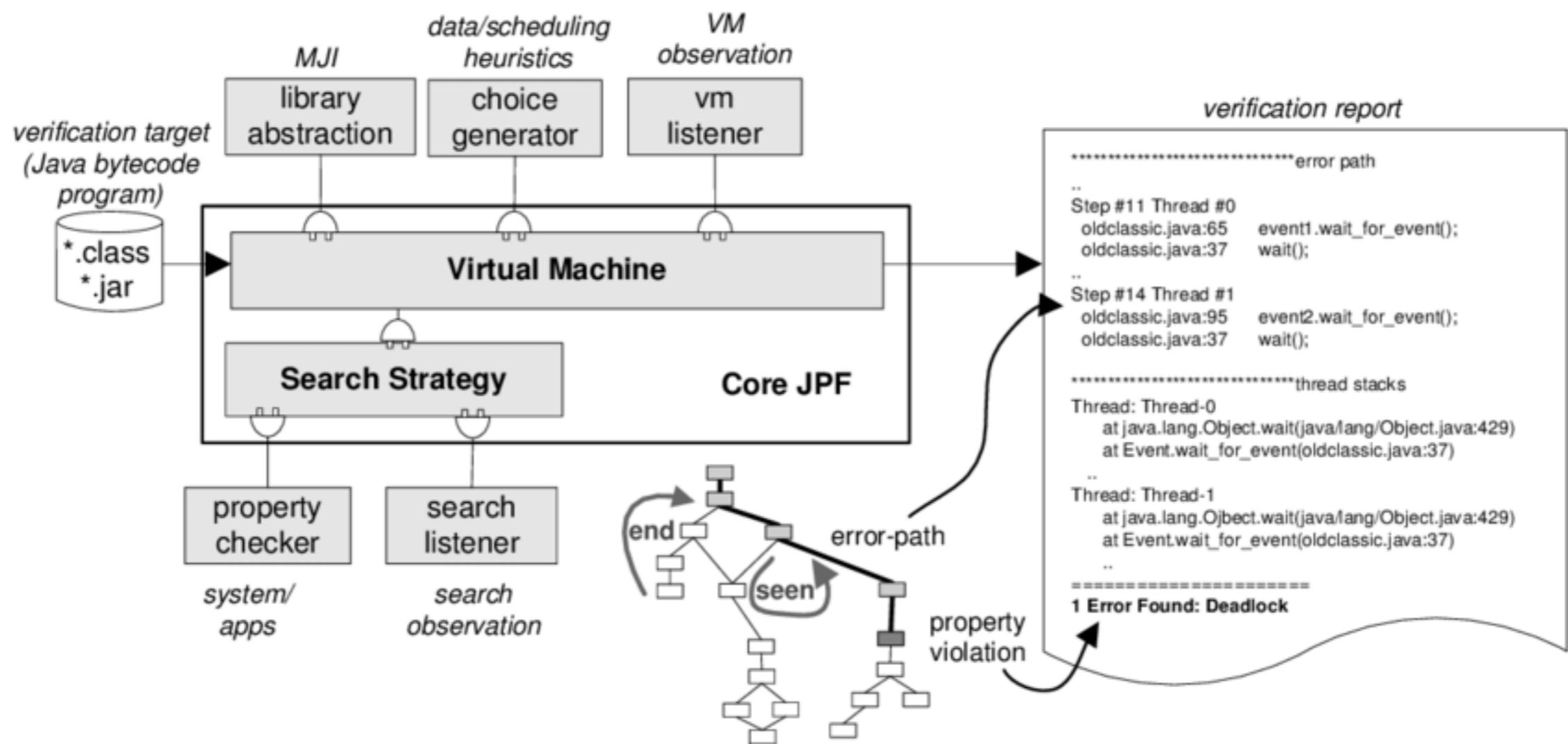
- Sviluppo
  - Robust Software Engineering group at NASA Ames Research Center
  - 1999 - presente
  - Open source (dal 2004)
  - Più di 11100 downloads dal 04/2005
- Website
  - <http://javapathfinder.sourceforge.net/>
- Repository
  - <https://github.com/javapathfinder>

# **Java PathFinder: Sistemi Operativi supportati**

- Windows/Linux
  - Linea di comando
    - Ant
    - JUnit
  - Integrated Development Environments
    - Eclipse
    - NetBeans

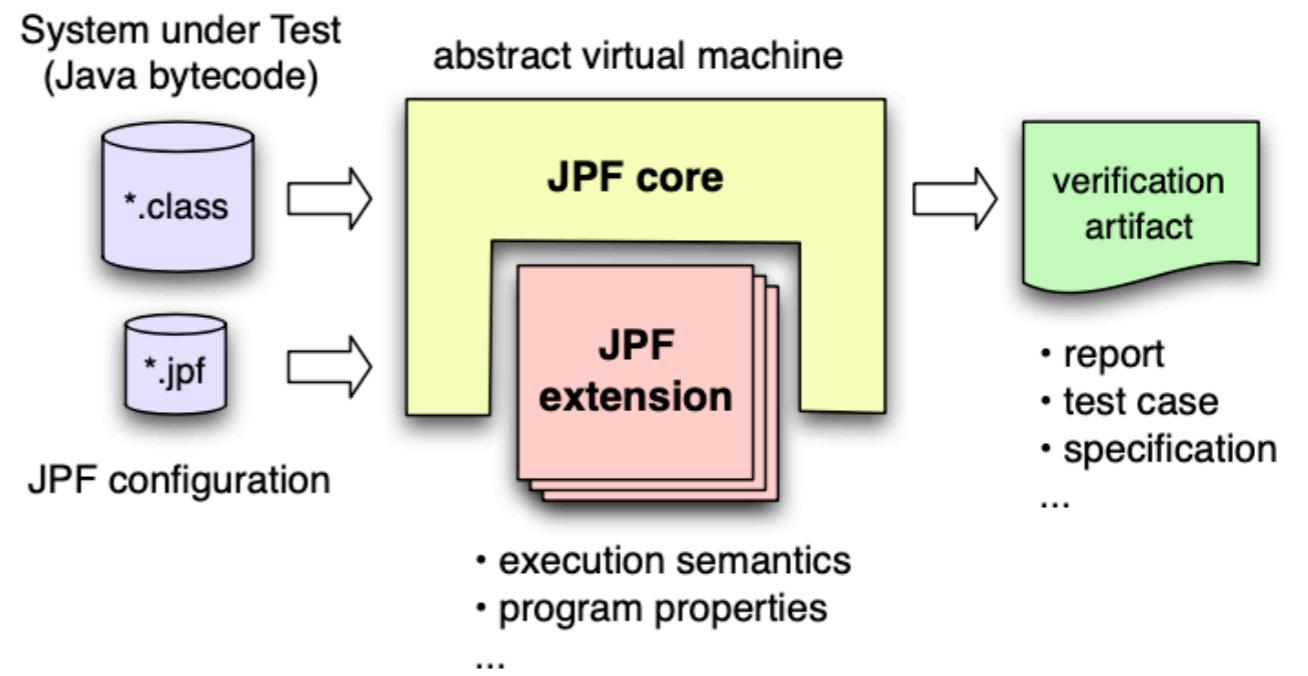
# Cosa JPF può fare? (parte 1)

- "JPF è un coltellino svizzero per quanto riguarda tutti i generi di verifica effettuata a tempo esecuzione"



# Cosa JPF può fare? (parte 2)

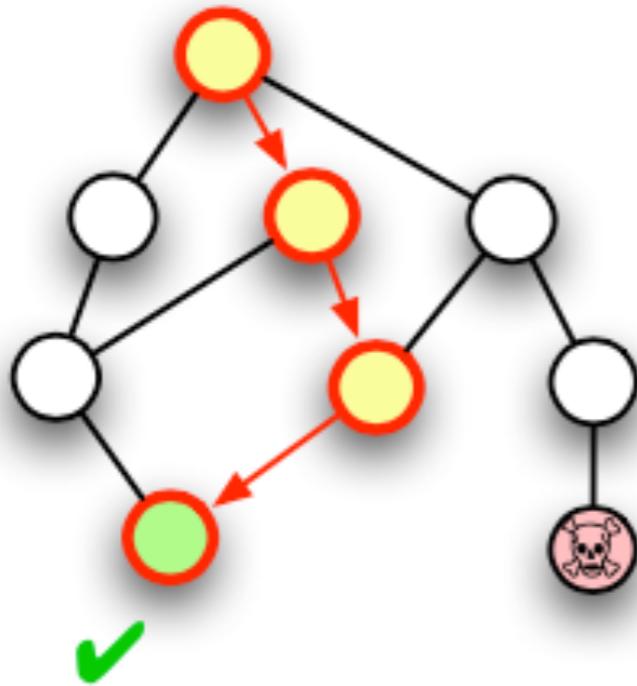
- Model Checking
- Facilmente estendibile
  - Test Case Generation
  - Coverage Analysis
  - Numeric Analysis
  - UML State Machine Model Checker
  - e molto più!



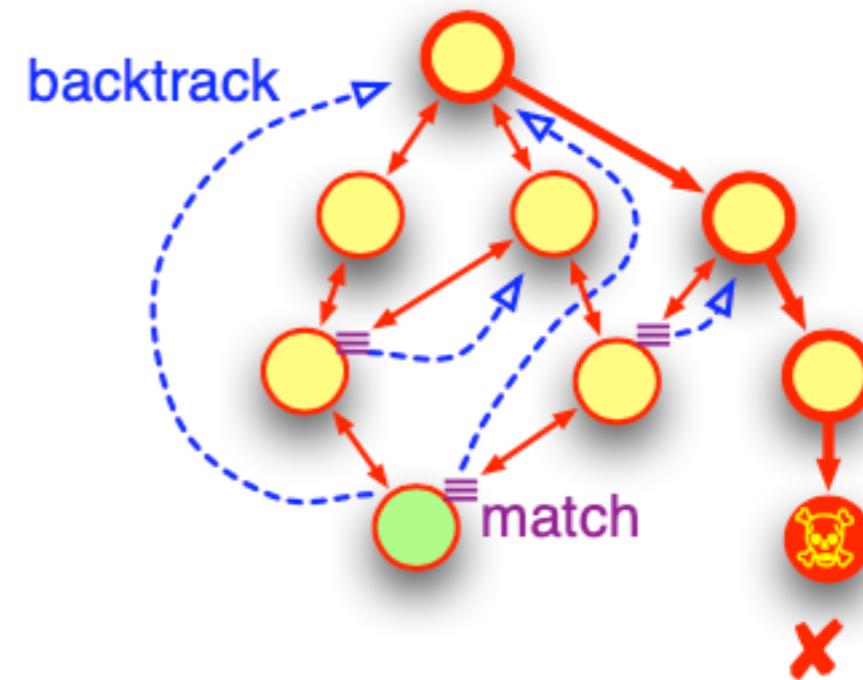
# Model Checking in JPF

- Il programma viene effettivamente eseguito
  - `jpf <la tua classe> <parametri>`
    - molto simile a "`java <la tua classe> <parametri>`"
  - Esegui in modo che tutti i possibili scenari siano esplorati
    - Interleaving di Threads
    - Valori non deterministici (random)
  - L'input concreto viene fornito
  - Uno stato è effettivamente uno stato concreto, che consiste in
    - Valori concreti nella memoria heap/stack

# Model Checking vs Testing



Testing esplora un singolo cammino d'esecuzione

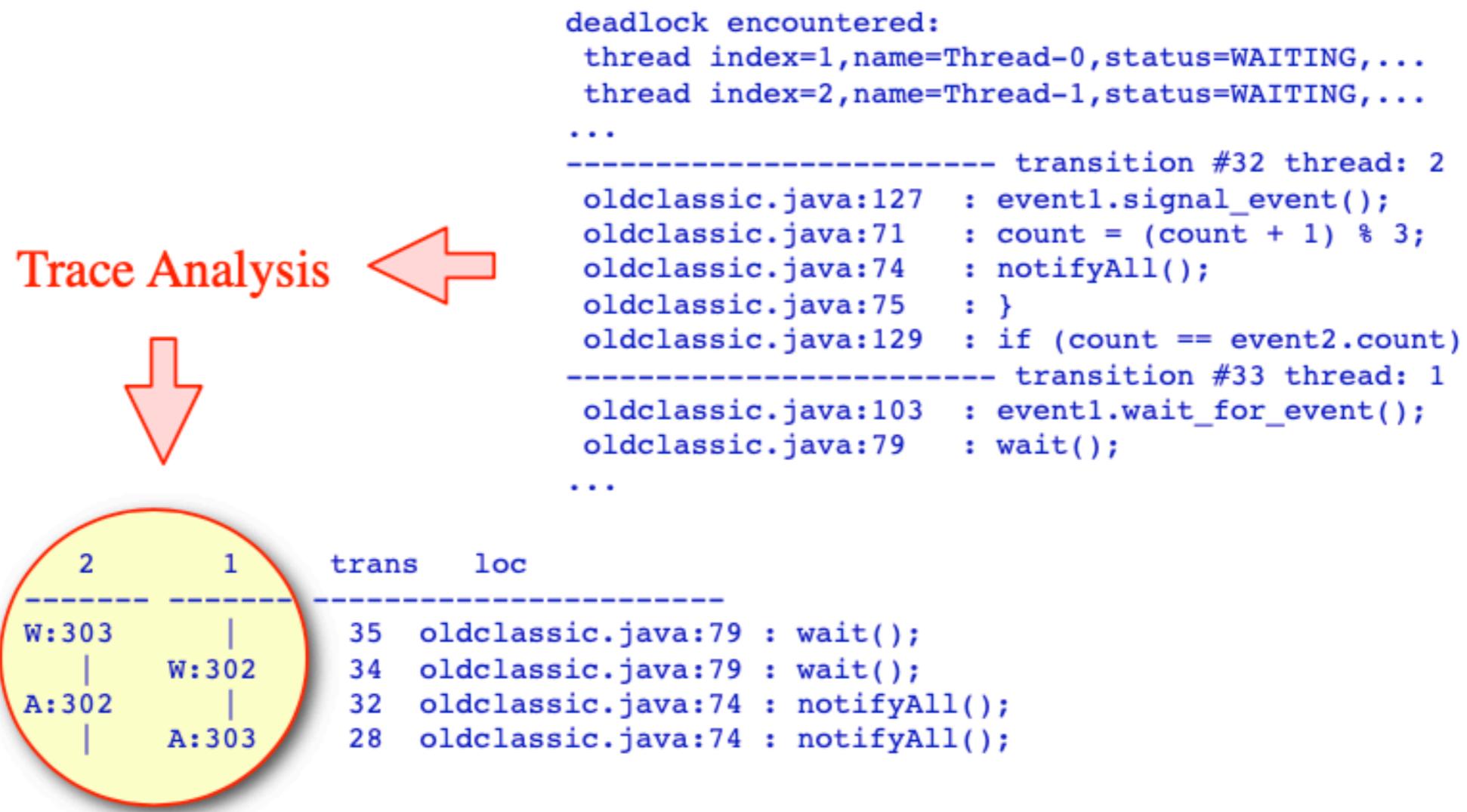


Model Checking esplora tutti i possibili cammini

- Model Checking è un rigoroso metodo formale
- Model Checking produce tracce (esecuzioni) quando trova un errore

# Model Checking vs Debugging

- JPF trova automaticamente errori (che poi si può andare successivamente a debuggare)
- Tracce di programmi - JPF ricorda l'intera esecuzione che ha portato all'errore
- Analisi automatica della traccia porta alla spiegazione dell'errore



# Esempio (non determinismo)

```
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);

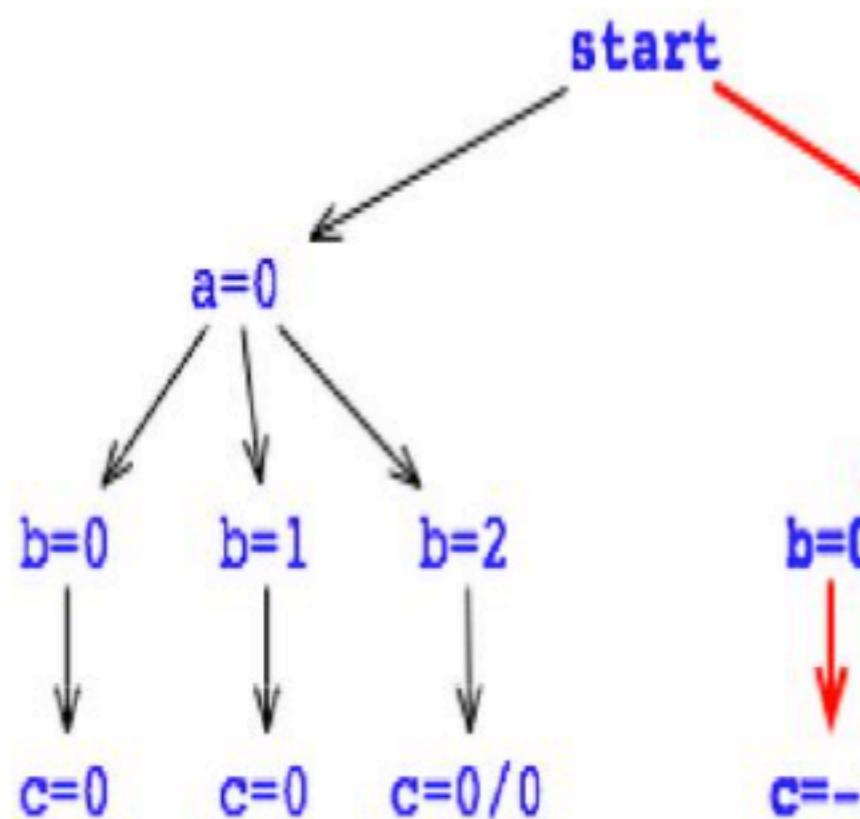
        //... lots of code here

        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
>
```

# Esempio (cont'd)



- ① Random random = new Random()
- ② int a = random.nextInt(2)
- ③ int b = random.nextInt(3)
- ④ int c = a/(b+a -2)

- Una esecuzione corrisponde a un cammino

# Esempio (cont'd)

```
> 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  
>
```

# Esempio (cont'd)

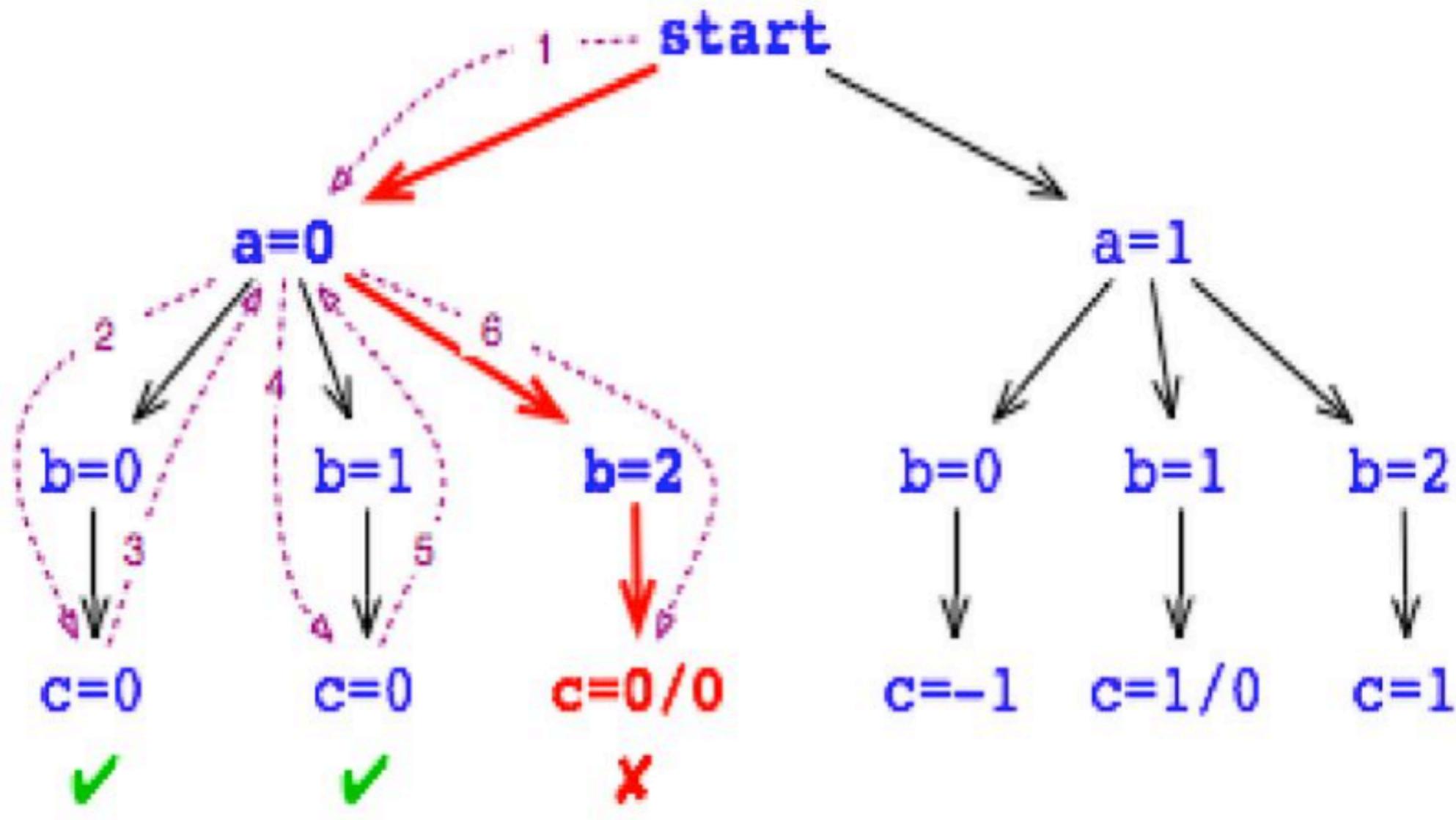
```
> bin/jpf +vm.enumerate_random=true Rand
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
===== system under test
application: /Users/pcmehlitz/tmp/Rand.java

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

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

- Specificando "+vm.enumerate\_random=true" stiamo dicendo a JPF di considerare tutti i possibili valori per espressioni (2) e (3)

# Esempio (cont'd)



- JPF esplora più possibili esecuzioni, dato lo stesso input concreto

# Esempio (data race)

```
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 started
10
10

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

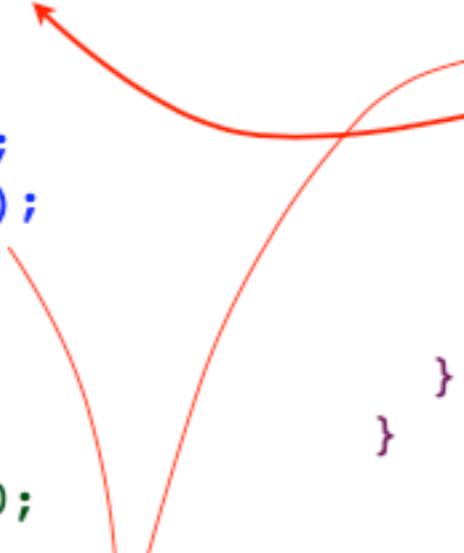
=====
===== trace #1
----- transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
    [282 insn w/o sources]
        Racer.java:15          : Racer racer = new Racer();
        Racer.java:1           : public class Racer implements Runnable {
            [1 insn w/o sources]
   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

# Esempio (deadlock)

```
class FirstTask extends Thread {    class SecondTask extends Thread {  
    Event event1, event2;  
    int count = 0;  
    ...  
    public void run () {  
        count = event1.count;  
  
        while (true) {  
            if (count == event1.count) {  
                event1.wait_for_event();  
            }  
  
            count = event1.count;  
            event2.signal_event();  
        }  
    }  
  
    class Event {  
        int count = 0;  
  
        public synchronized void signal_event () {  
            count++; notifyAll();  
        }  
        public synchronized void wait_for_event () {  
            ... wait(); ...  
        }  
    }  
}
```



# Esempio (cont'd)

```
JavaPathfinder v5.x - (C) RIACS/NASA Ames Research Center

===== system under test
application: oldclassic.java

===== search started: 2/11/10 11:10 AM
1
2
1
2
1
...
.

===== error #1
gov.nasa.jpf.jvm.NotDeadlockedProperty
deadlock encountered:
    thread index=0,name=main,status=TERMINATED,this=null,target=null, ...
    thread index=1,name=Thread-0,status=WAITING,this=FirstTask@295,lockCount=1, ...
    thread index=2,name=Thread-1,status=WAITING,this=SecondTask@322,lockCount=1, ...
...
```

# Esempio (cont'd)

```
...
===== trace #1
----- transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
[2843 insn w/o sources]
oldclassic.java:47           : Event      new_event1 = new Event();
...
----- transition #29 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>Thread-0,Thread-1}
oldclassic.java:102          : if (count == event1.count) {
oldclassic.java:103          : event1.wait_for_event();
----- transition #30 thread: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {Thread-0,>Thread-1}
oldclassic.java:129          : if (count == event2.count) {
oldclassic.java:133          : count = event2.count;
...
oldclassic.java:126          : System.out.println(" 2");
oldclassic.java:127          : event1.signal_event();
...
oldclassic.java:71           : count = (count + 1) % 3;
oldclassic.java:74           : notifyAll(); ←
oldclassic.java:75           : }
oldclassic.java:129          : if (count == event2.count) {
----- transition #33 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>Thread-0,Thread-1}
oldclassic.java:103          : event1.wait_for_event();
oldclassic.java:79           : wait(); ←
----- transition #34 thread: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>Thread-1}
oldclassic.java:129          : if (count == event2.count) {
oldclassic.java:130          : event2.wait_for_event();
oldclassic.java:79           : wait();
```

Segnale perso

# Esempio (overflow)

- Java non solleva eccezioni in caso di overflow
- Tedioso da controllare.. Solitamente ignorato

**JPF configuration**

```
vmInsnFactory.class =
    .numeric.NumericInstructionFactory
...
2103933699
2123371282
2142808865
...
=====
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmaticException: integer overflow: 2142808865+19437583 = -2132720848
    at Overflow.notSoObvious(Overflow.java:18)
...
```

**System under Test**

**Report**

```
void notSoObvious(int x){
    int a = x*50;
    int b = 19437583;
    int c = a;

    for (int k=0; k<100; k++){
        c += b;
        System.out.println(c);
    }
    ...
    notSoObvious( 21474836);
```

# Due capacità essenziali

- Backtracking
  - JPF può restaurare stati appartenenti a esecuzioni passate, per poter controllare se esistono scelte ancora da esplorare
    - Mentre questo può essere teoricamente ottenuto eseguendo nuovamente il programma dall'inizio, Backtracking è una meccanica molto più efficiente se il salvataggio (storage) degli stati è ottimizzato
- State matching
  - JPF controlla se ogni nuovo stato generato è stato già visto in precedenza, in questo caso la ricerca sul cammino corrente può essere interrotta. JPF può tornare indietro (backtrack) al più vicino non esplorato non deterministico punto di scelta
    - Heap e Thread-Stack sono fotografati (snapshots)

# Esercizio

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

thread_1(void)      /* initialize p, q, and r */
{
    p = &x;
    q = &y;
    z = &r;
}

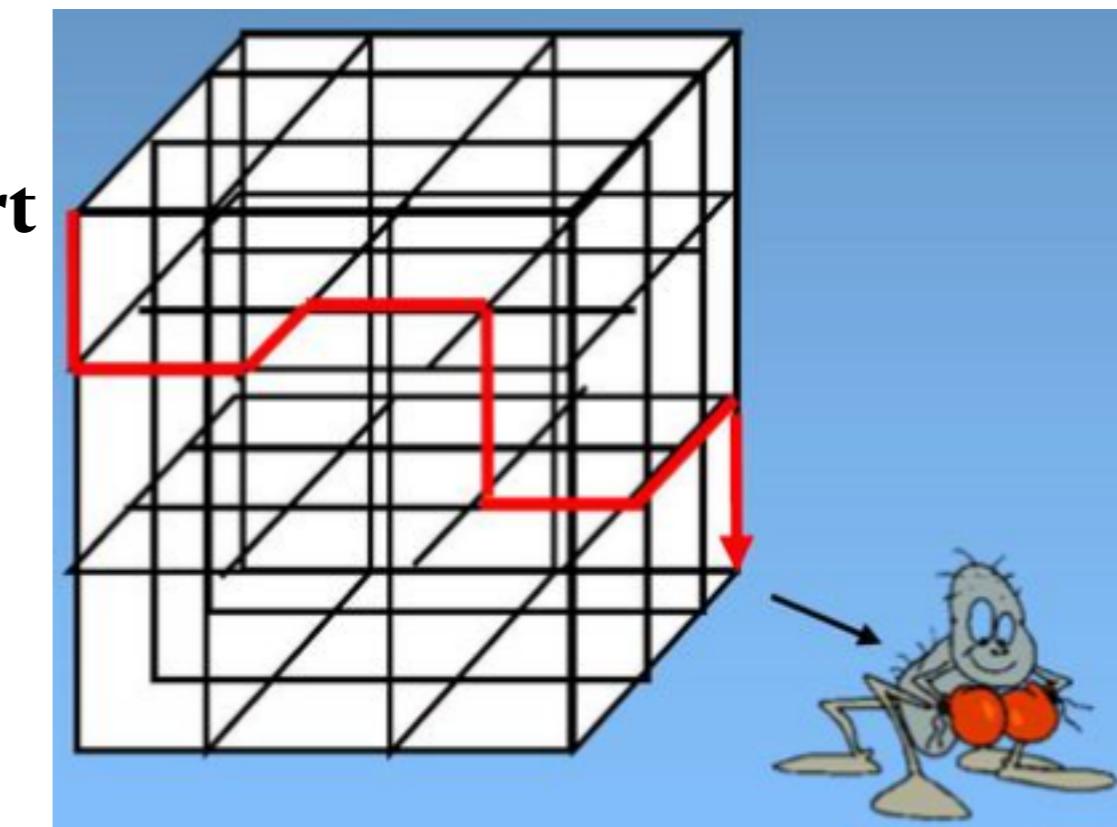
thread_2(void)      /* swap contents of x and y */
{
    r = *p;
    *p = *q;
    *q = r;
}

thread_3(void)      /* access z via a and p */
{
    a = &p;
    *a = z;
    **a = 12;
}
```

3 threads asincroni che accedono dati condivisi  
3 linee di codice ciascuno  
Quante esecuzioni di test sono necessarie per controllare che non ci sia corruzione nei dati?

# Esercizio (cont'd)

- Il numero di possibili thread interleaving è...
  - $(9! / 6! * 3!) * (6! / 3! * 3!) * (3! / 3!) = \mathbf{1680 \text{ possibili esecuzioni}}$
  - Posizionare 3 insiemi di 3 tokens in 9 slots
- Sono tutte queste esecuzioni valide?
- Possiamo controllarle tutte? Dovremmo controllarle tutte?
- In un classico sistema di test, quante di queste verrebbero normalmente controllate?
- **State Explosion!!**



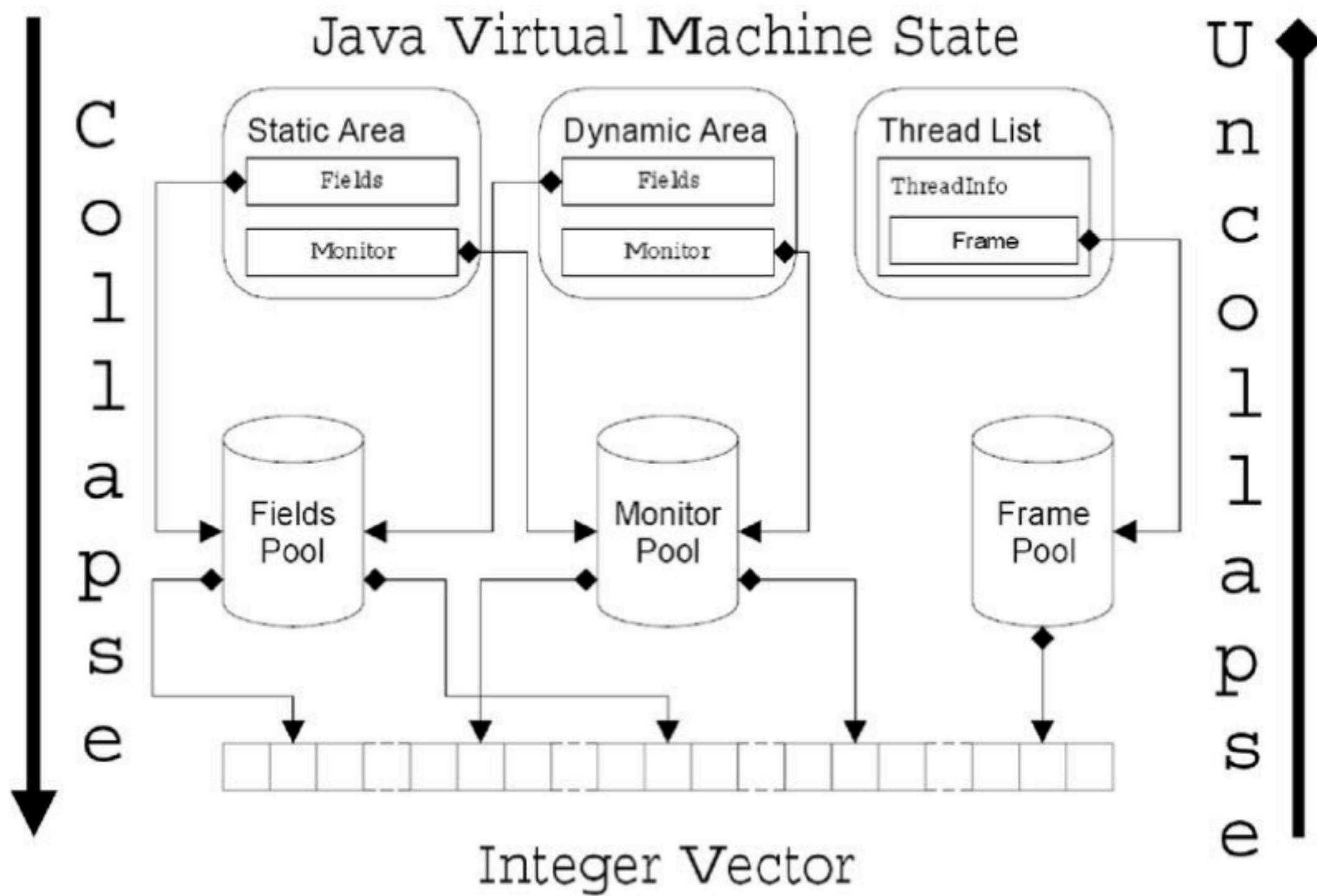
# La soluzione di JPF

- Strategia di ricerca configurabile
  - Direzionare la ricerca in modo che gli errori possano essere trovati più velocemente
    - Uno strumento di debugging invece di un sistema di prova
    - Utente può facilmente sviluppare la sua strategia
- Host VM execution
  - Delegare l'esecuzione alla sottostante macchina virtuale host (no tracking degli stati)
- Ridurre memoria per salvataggio degli stati
  - State collapsing (collassare gli stati)
    - Premessa: solo una piccola parte dello stato viene modificata in una transizione (es. un singolo stack frame)
    - Divisione di uno stato in componenti; usare hash table per indicizzare una specifica componente

# Uno stato in JPF

- Contiene informazioni riguardo ogni singolo thread (es. il stack frame corrente)
- Contiene i campi statici e dinamici delle classi (lock inclusi)
- Tutto questo viene salvato in un array di interi

# State collapsing



# Alternativa.. State reduction

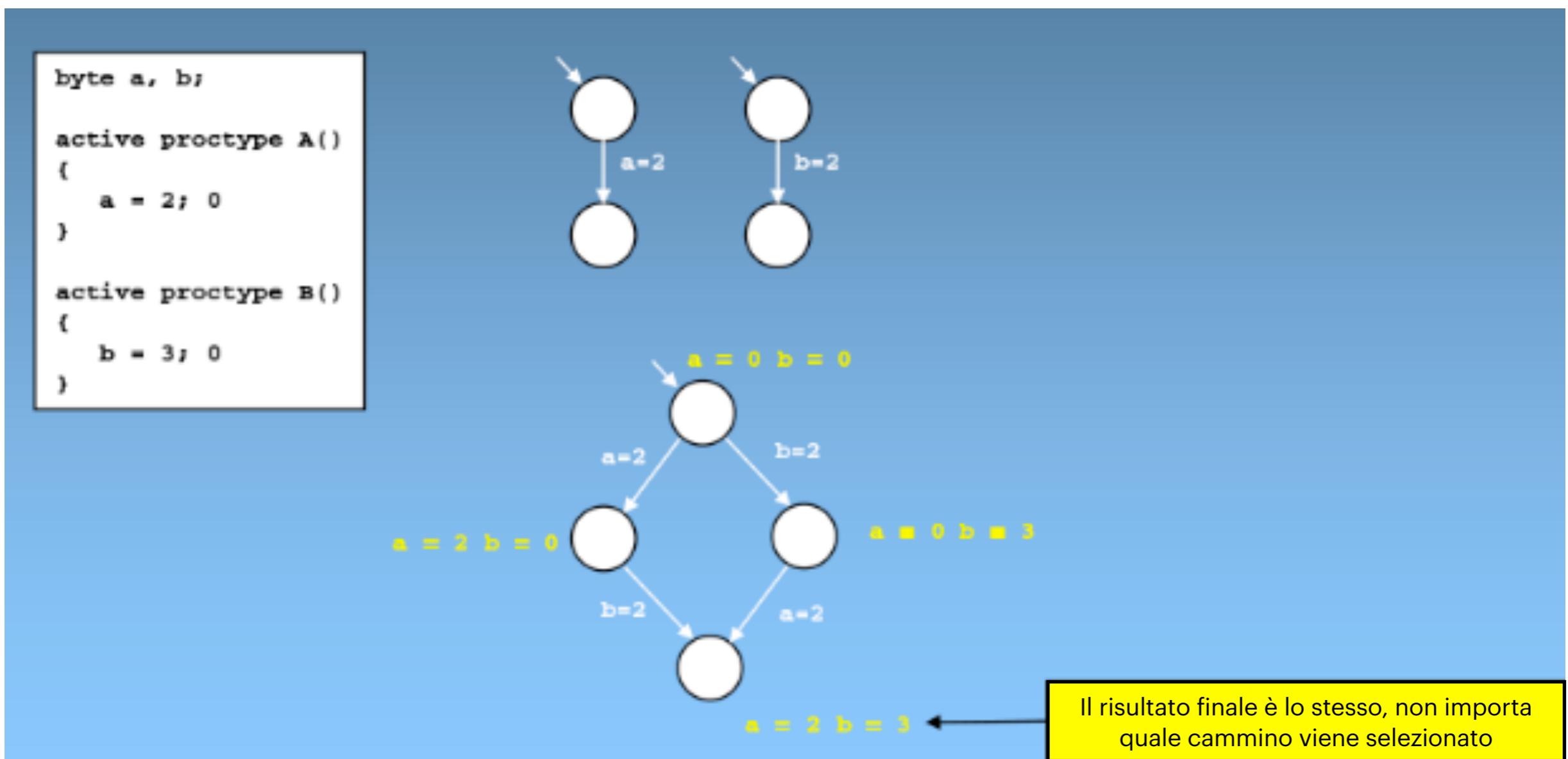
- Ortogonale (utilizzato in JPF)
  - State Abstraction (astrazione degli stati)
  - Partial Order Reduction

# Partial Order Reduction (POR)

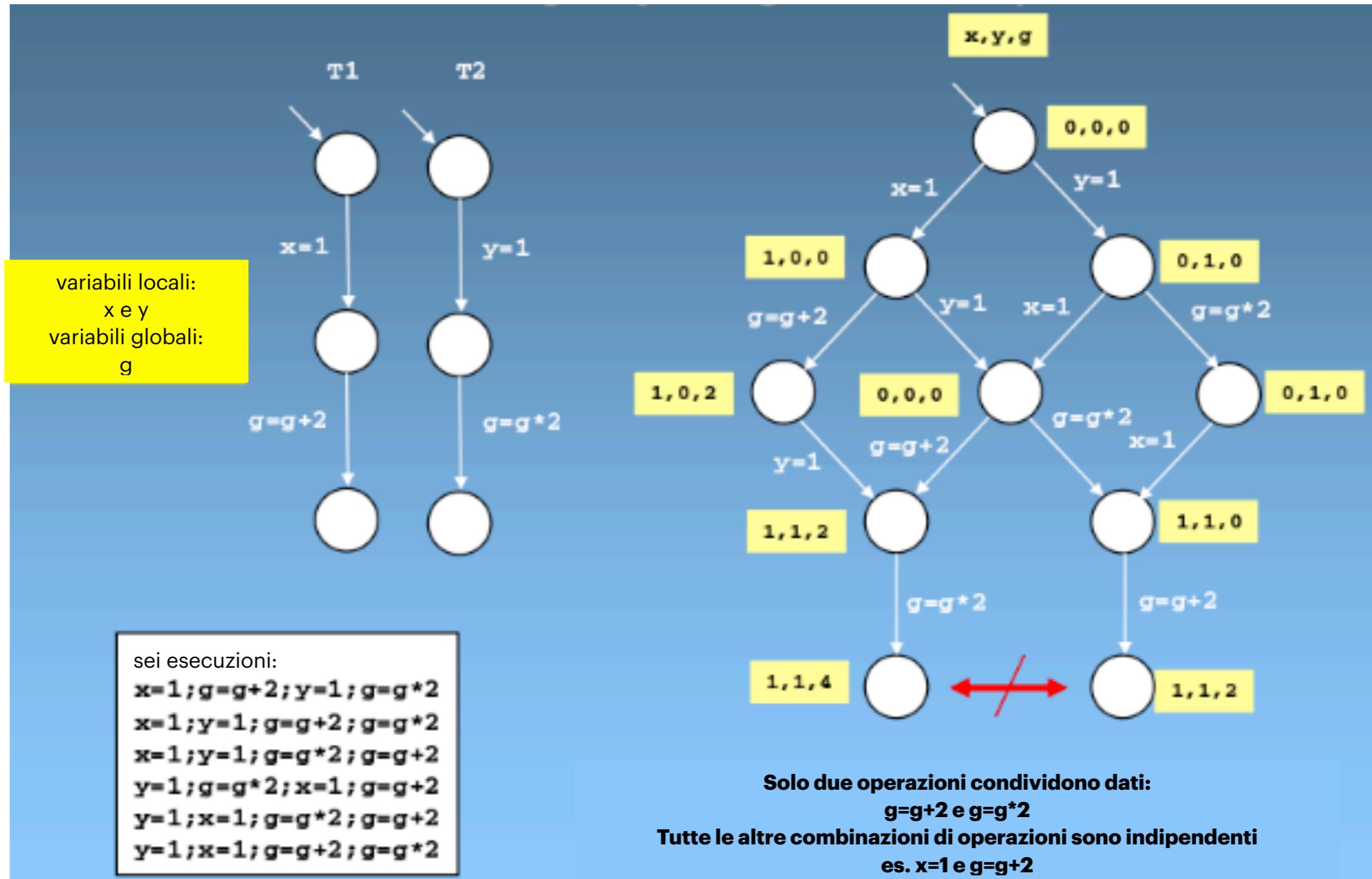
- JPF usa on-the-fly partial order reduction algoritmi per ridurre lo spazio degli stati tramite l'identificazione di insiemi di stati di azioni concorrenti
- On-the-fly significa che JPF effettua tale operazioni mentre il codice è in esecuzione
- La transizione tra stati è determinata dal tipo di istruzioni in JPF

# Partial Order Reduction (POR)

- L'interleaving considerando totale asincronia delle azioni può essere alcune volte ridondante



# Partial Order Reduction (Example)



# Partial Order Reduction (Example)

independent pairs:

- $x=1 \leftrightarrow y=1$
- $x=1 \leftrightarrow g=g^*2$
- $y=1 \leftrightarrow g=g+2$

2 gruppi di 3 esecuzioni equivalenti:

~~$x=1; g=g+2; y=1; g=g^*2$~~

~~$x=1; y=1; g=g+2; g=g^*2$~~

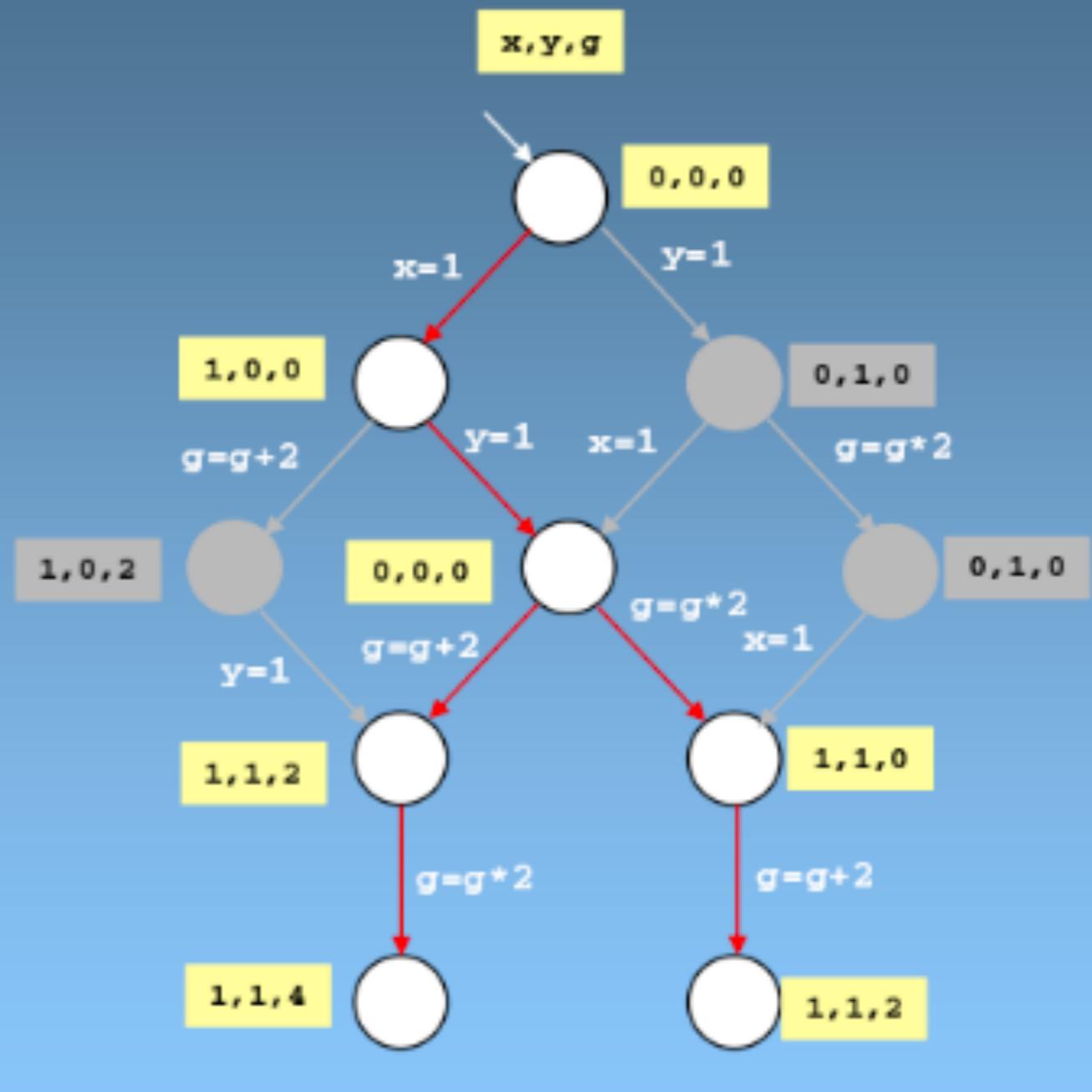
~~$y=1; x=1; g=g+2; g=g^*2$~~



~~$x=1; y=1; g=g^*2; g=g+2$~~

~~$y=1; x=1; g=g^*2; g=g+2$~~

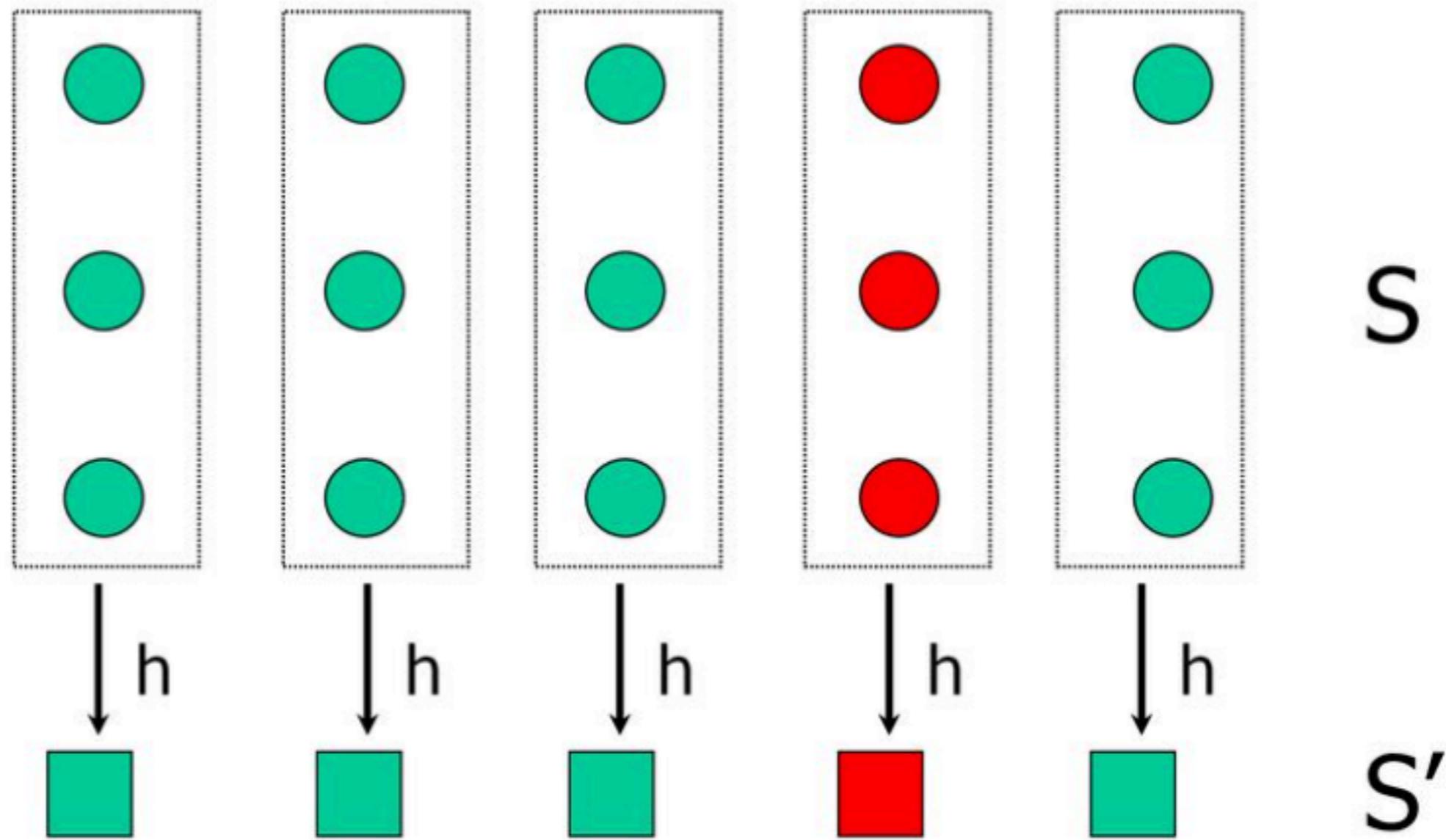
~~$y=1; g=g^*2; x=1; g=g+2$~~



# Astrazione

- Eliminare i dettagli irrilevanti per la proprietà sotto analisi
- Ottenere semplici modelli a stati finiti che siano sufficienti a verificare la proprietà
- Svantaggio:
  - Perdita di precisione: **False positives/negatives**

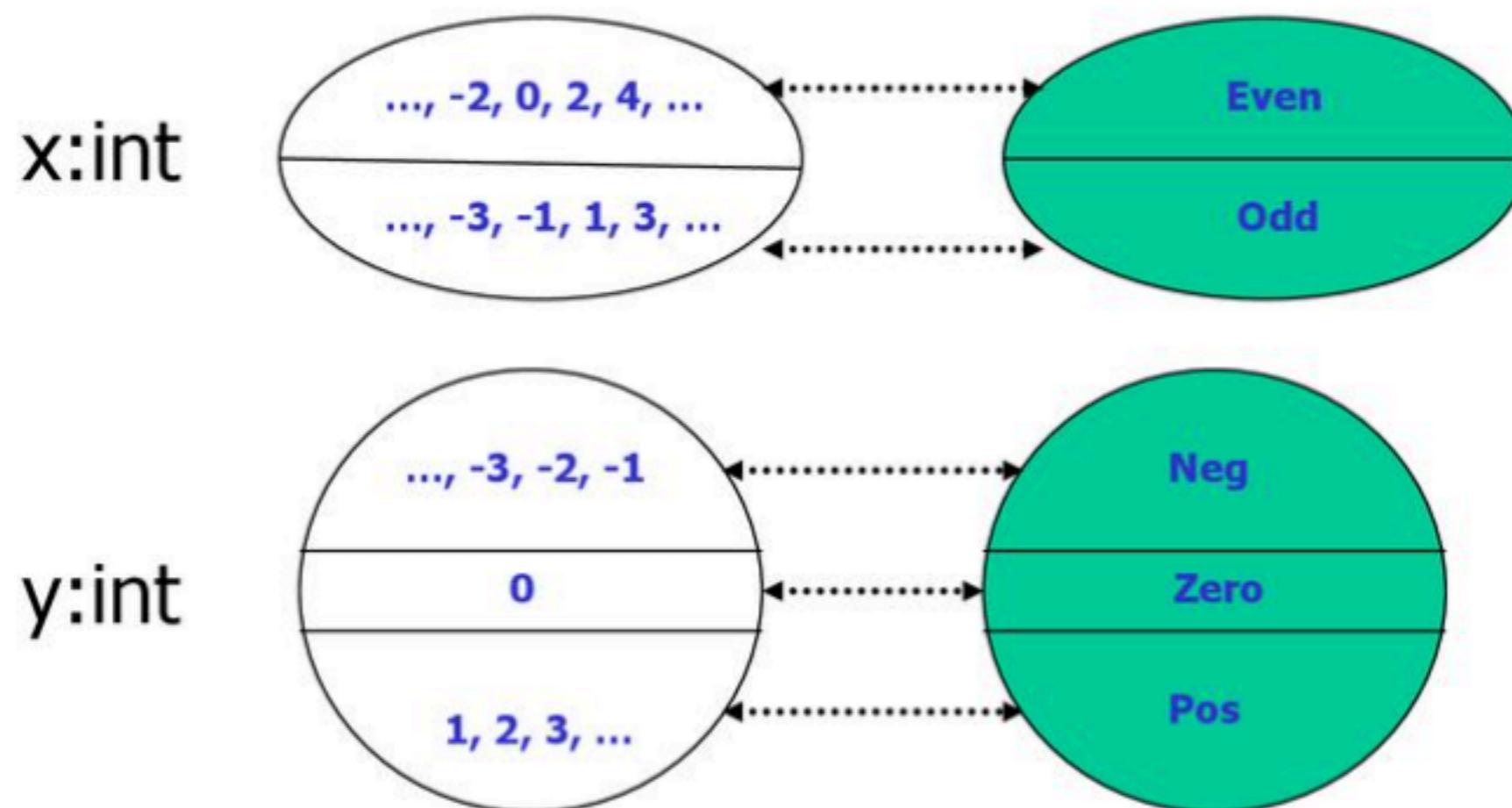
# Astrazione dei dati



Abstraction Function  $h$  : from  $S$  to  $S'$

# Astrazione dei dati (esempio)

- L'astrazione procede considerando le componenti. Le variabili sono considerate componenti.



# Come possiamo astrarre il comportamento?

- Dominio astratto A
  - Astrazione dei valori concreti a quelli appartenenti ad A
  - Calcolare poi le transizioni nel dominio astratto

# Astrazione del tipo di dato

Code

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



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

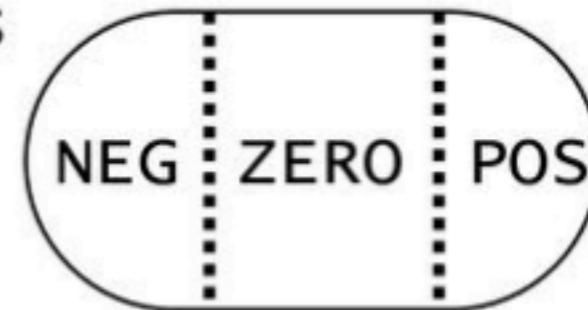
Abstract Data domain

int



(n<0) : NEG  
(n==0) : ZERO  
(n>0) : POS

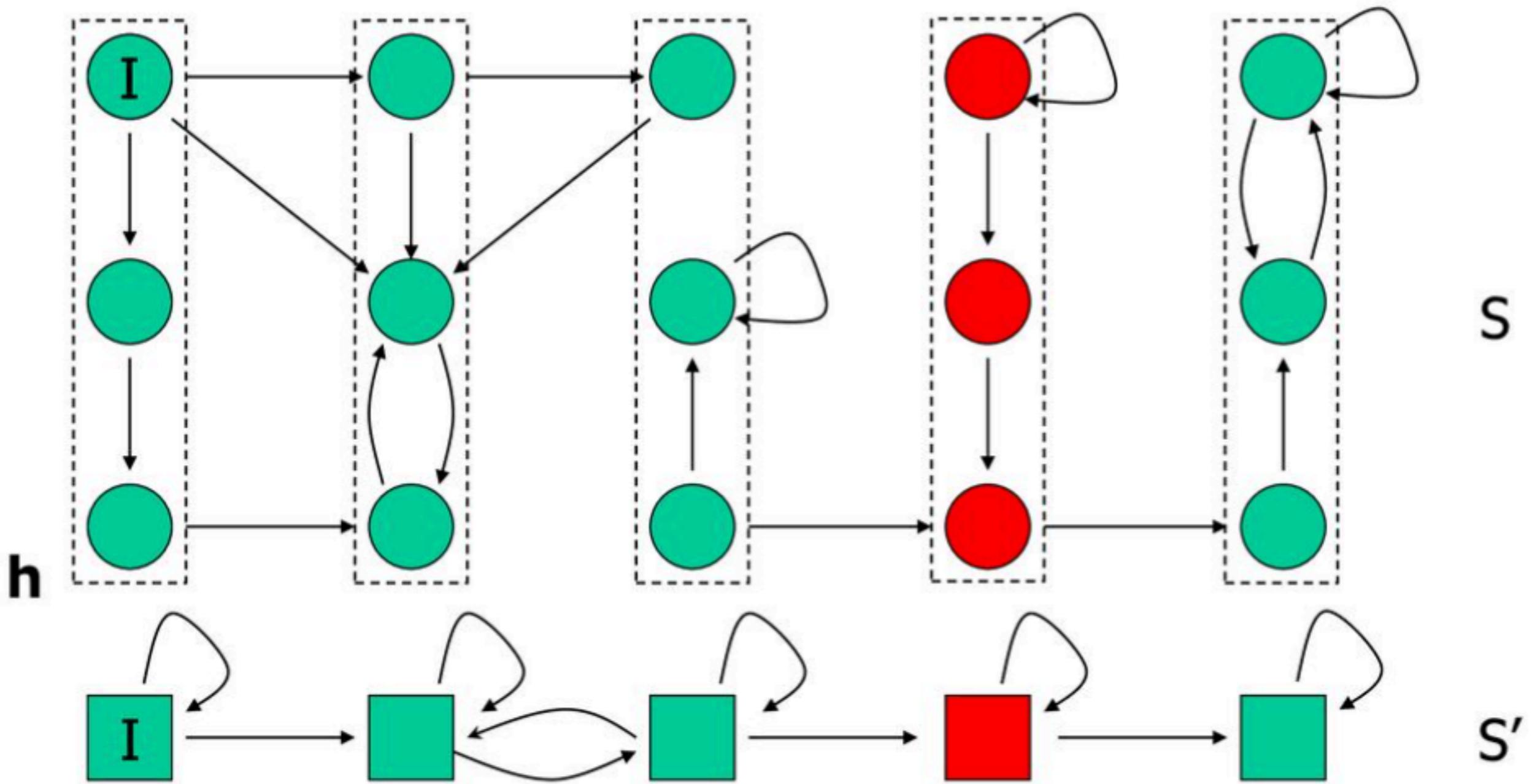
signs



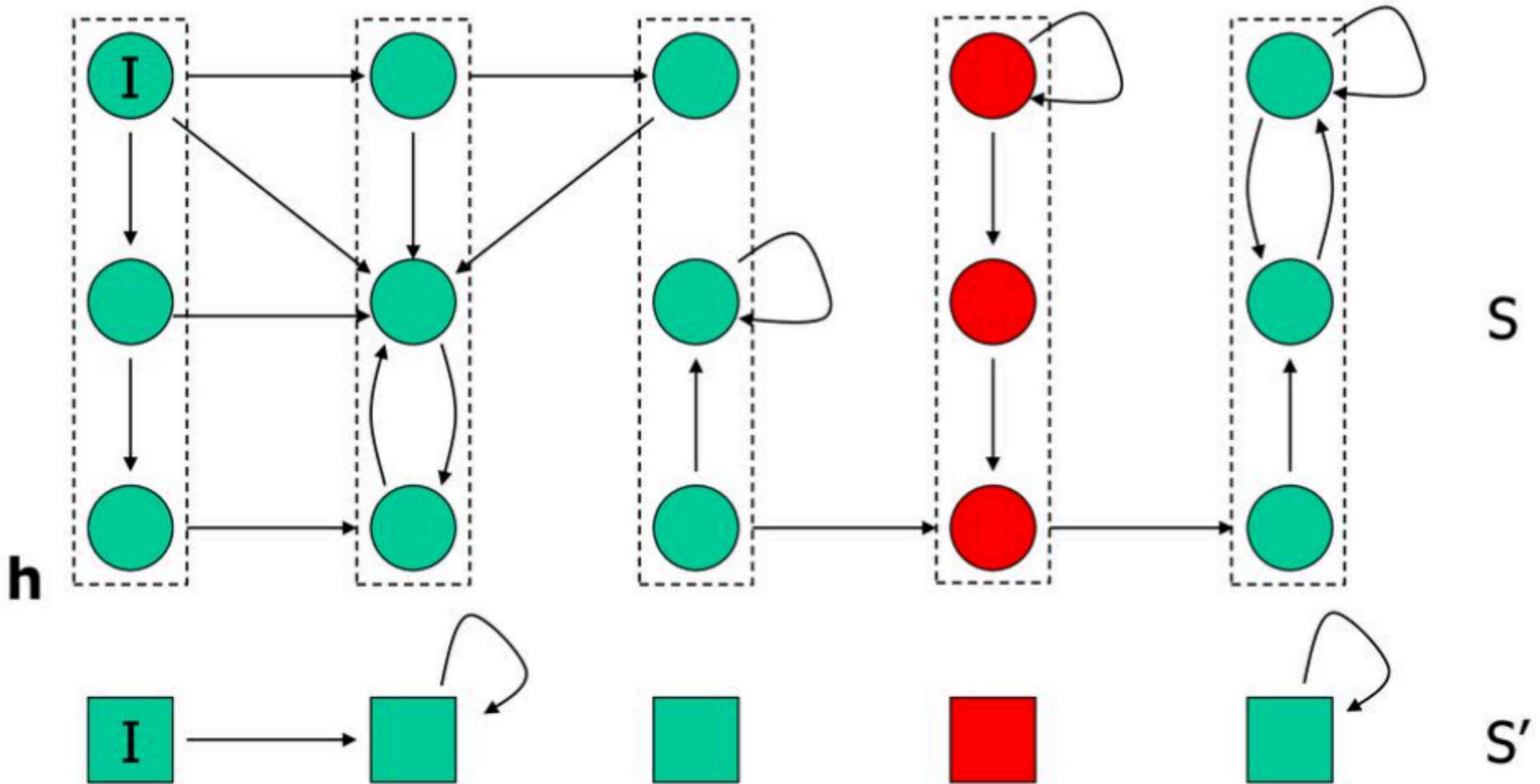
# Astrazione Esistenziale/Universale

- Esistenziale
  - Considerare una transizione da uno stato astratto se **almeno uno** stato concreto corrispondente possiede tale transizione
  - Il modello astratto  $M'$  simula il modello concreto  $M$
- Universale
  - Considerare una transizione da uno astratto se **tutti i** corrispondenti stati concreti possiedono tale transizione

# Astrazione Esistenziale (Over-Approximation)



# Astrazione Universale (Under-Approximation)

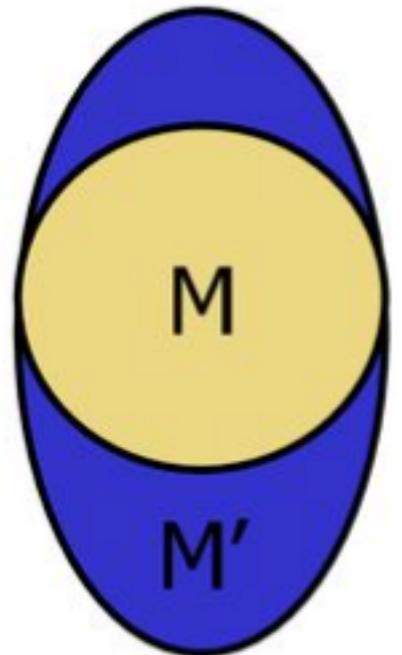


# Garanzie dall'astrazione

- Assumendo che  $M'$  sia un'astrazione di  $M$ 
  - **Strong preservation**
    - una proprietà  $P$  è verificata in  $M'$  se e solo se  $P$  è verificata in  $M$
  - **Weak preservation**
    - se una proprietà  $P$  è verificata in  $M'$  allora  $P$  è verificata in  $M$

# Garanzie dall'astrazione esistenziale

- Sia  $\phi$  una proprietà che deve valere per tutti i cammini
- $M'$  un'astrazione esistenziale di  $M$
- Teorema di preservazione



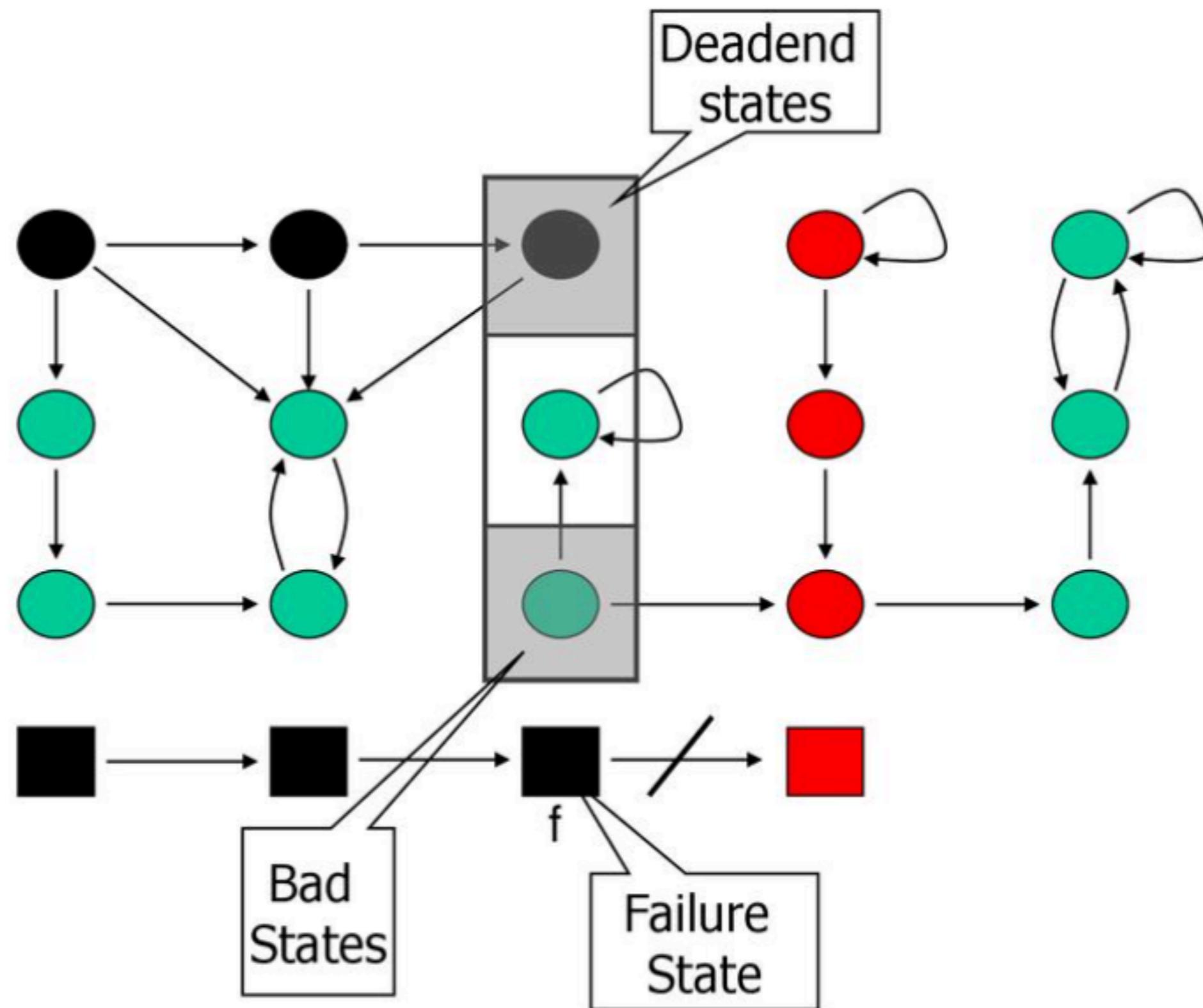
$$M' \models \varphi \rightarrow M \models \varphi$$

- L'inverso non vale

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

- Il controesempio può essere spurio (possibile nel modello astratto, ma non nel modello concreto)

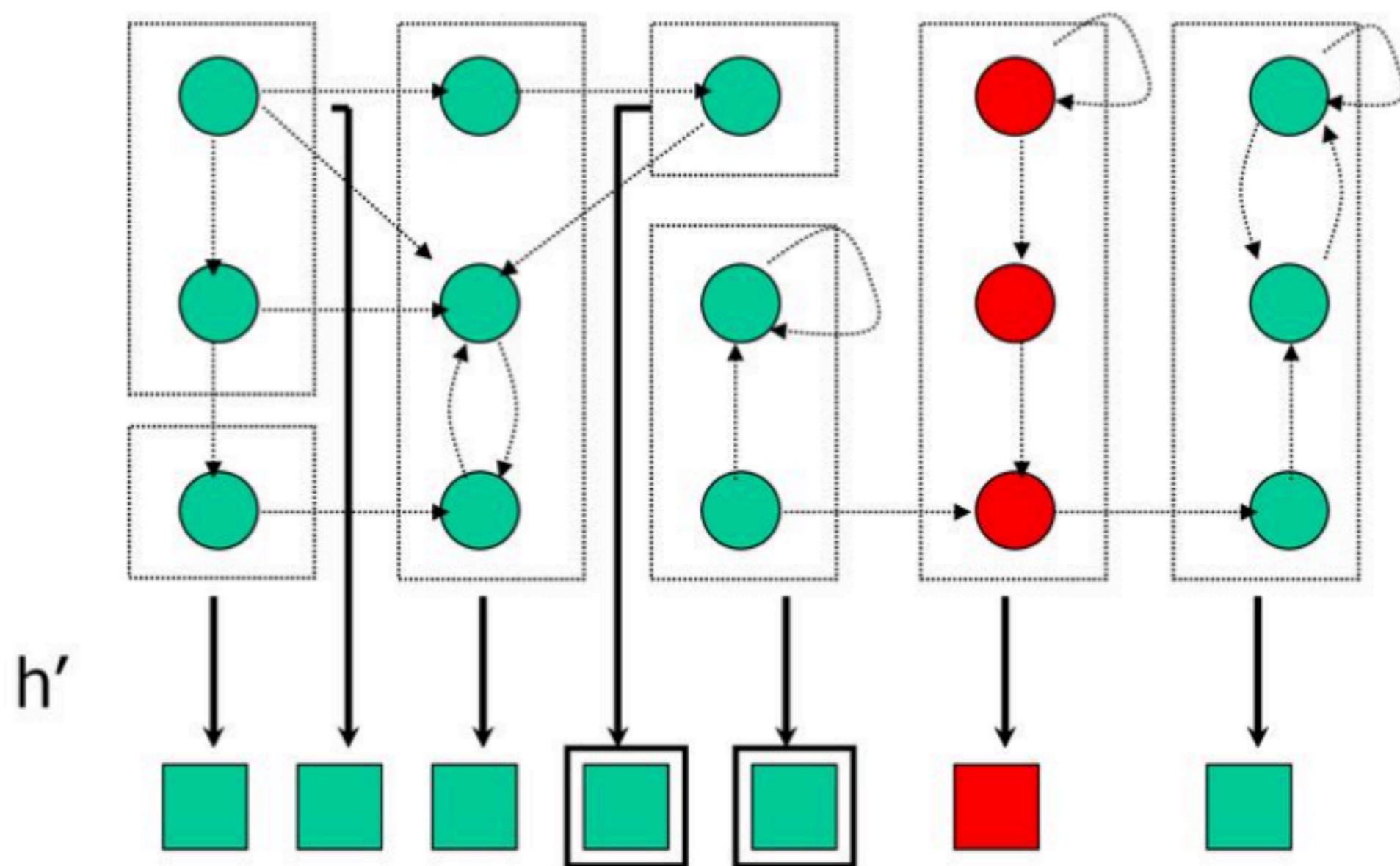
# Controesempio spurio in Over-Approximation



# Raffinamento

- Problema: Dead End e Bad States sono in qualche stato astratto
- Soluzione: Rifinire la funzione di astrazione
- L'insieme dei Dead End e Bad states dovrebbe essere separato in differenti stati astratti

# Raffinamento



Refinement :  $h'$

# Astrazione/Raffinamento automatici

- Buone astrazioni sono difficili da ottenere
  - Dobbiamo automatizzare sia Astrazione che Raffinamento
- Counterexample-Guided AR (CEGAR)
  - Creare modello astratto  $M'$  (da modello concreto  $M$ )
  - Effettuare model checking della proprietà  $P$  su  $M'$
  - Se la proprietà è verificata da  $M'$ , allora  $M$  soddisfa  $P$  (teorema di preservazione)
  - Altrimenti, controllare se il controesempio è spurio
  - Raffinare lo spazio degli stati astratti utilizzando il risultato dell'analisi sul controesempio
  - Ripetere

# Counterexample-Guided Abstract-Refinement (CEGAR)

