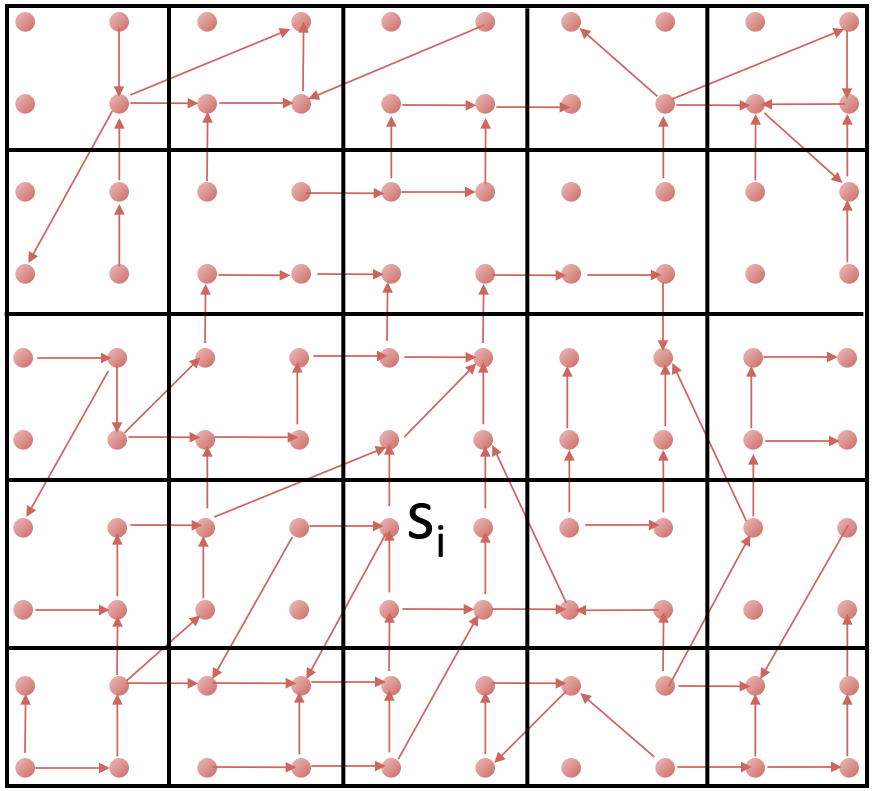
Predicate abstraction



Group concrete states that satisfy a certain property together.

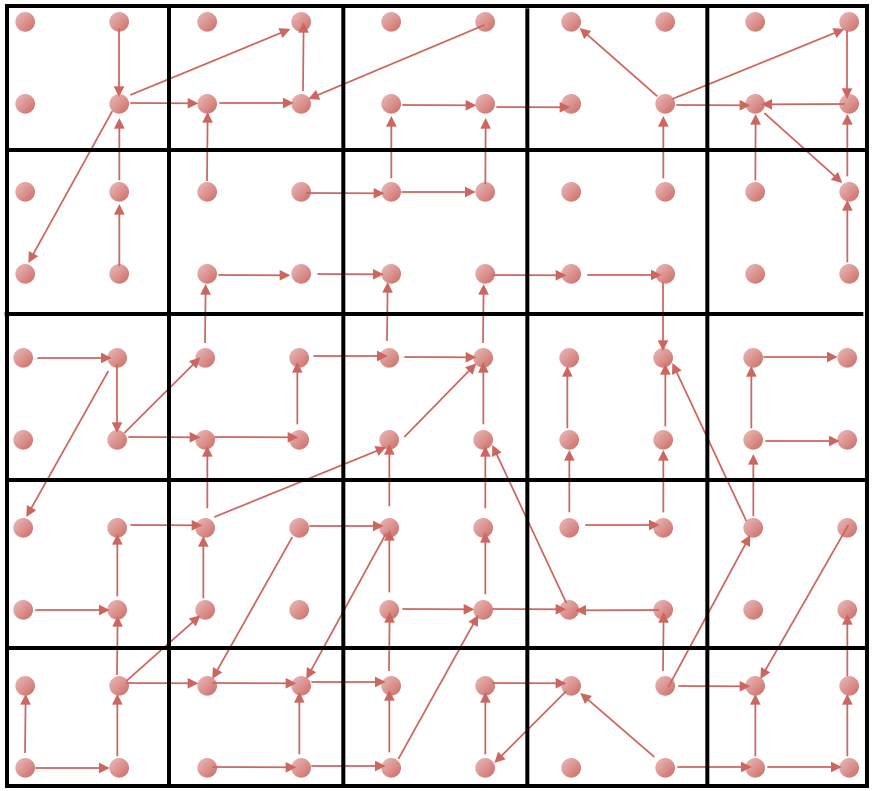
→ finitely many abstract states, labeled by predicates

Each such concrete state $\mathbf{s}_i$ consists of

- program counter
- state of variables

Hence, one node in the CFG can correspond to many different program states.

Predicate abstraction



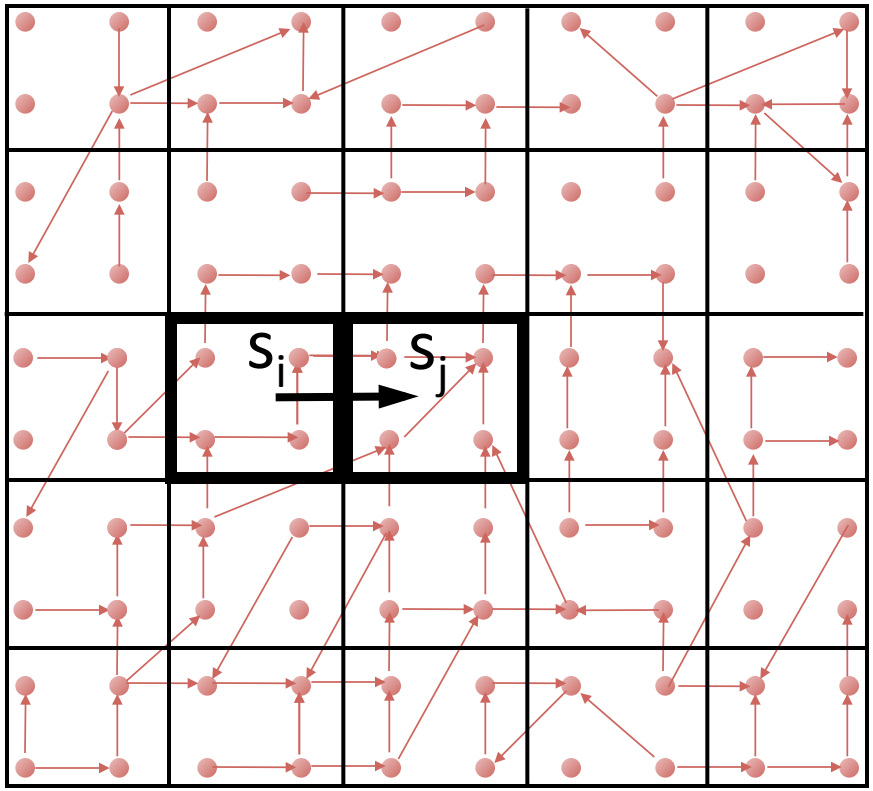
We are given the concrete relation with transitions $(s_i, s_j) \in r$, i.e whenever we have an edge in the CFG.

Using some abstraction function β we get corresponding abstract states

$a_i = \beta(s_i)$ and $a_j = \beta(s_j)$

and we merge those states whose predicates are the same.

Predicate abstraction



$a_i = \beta(s_i)$ and $a_j = \beta(s_j)$

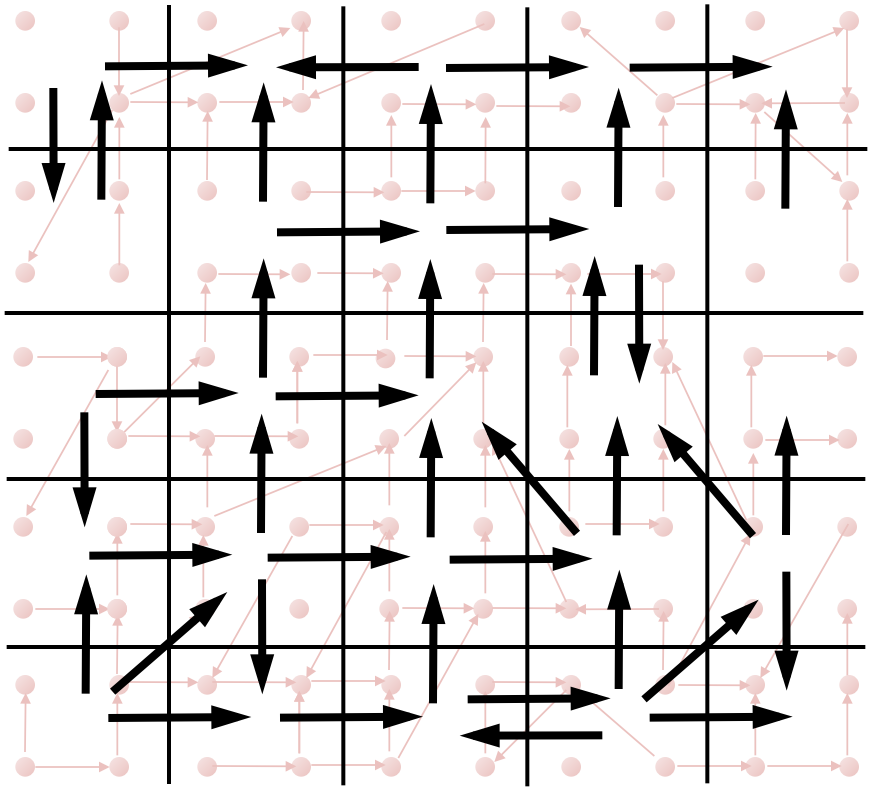
Then, if

$(s_i, s_j) \in r$

we require

$(a_i, a_j) \in a$.

Predicate abstraction



$a_i = \beta(s_i)$ and $a_j = \beta(s_j)$

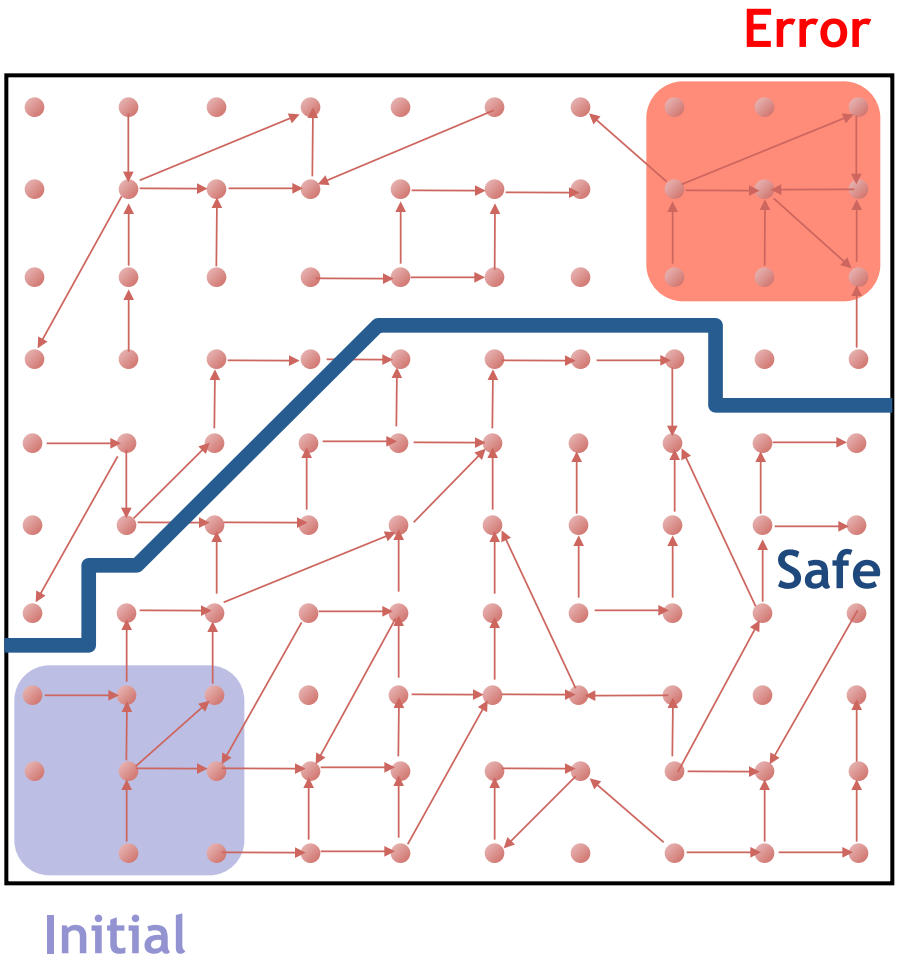
Then, if

$(s_i, s_j) \in r$

we require

$(a_i, a_j) \in a$.

Predicate abstraction

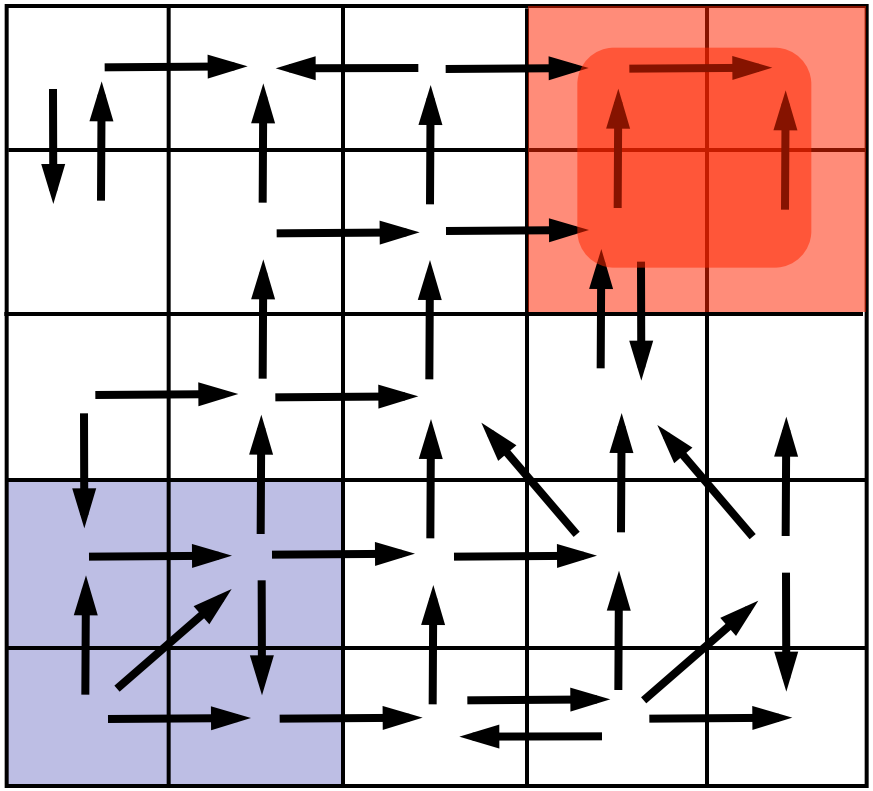


Error states are bad states where the property to check does not hold.

Reachability question:

Is there a **path** from an **initial** to an **error** state ?

Predicate abstraction

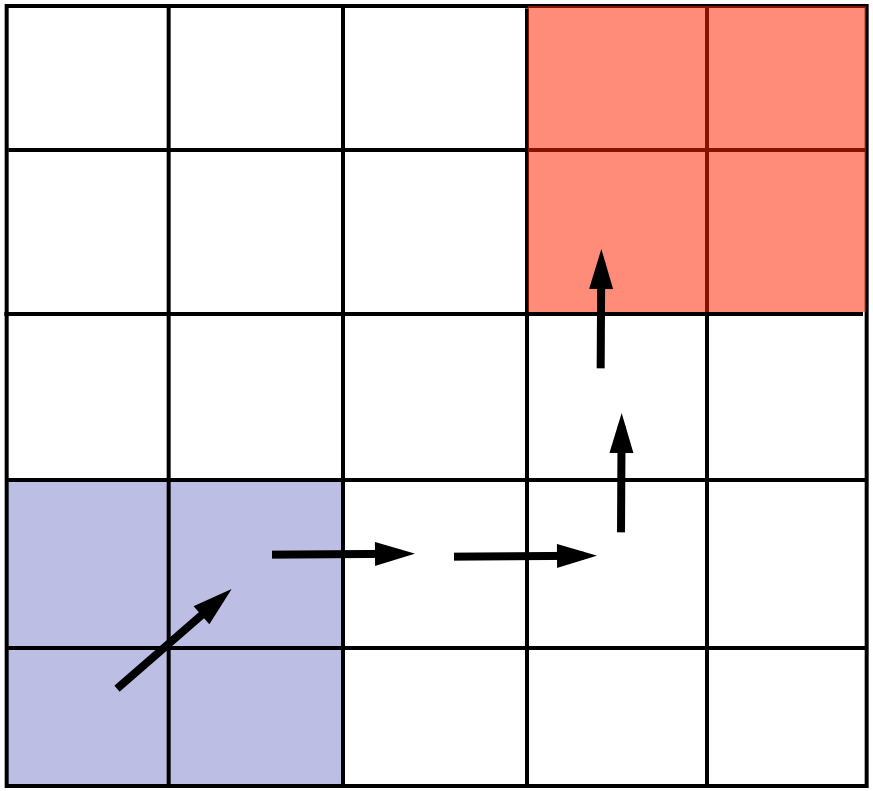


Is there a **path** from an **initial** to an **error** state ?

We are guaranteed to not get any false negatives:

if a state is unreachable in abstraction, it is unreachable in the concrete state space.

False positives



Suppose we find a path to some error state.

Have we found a true bug in the program?

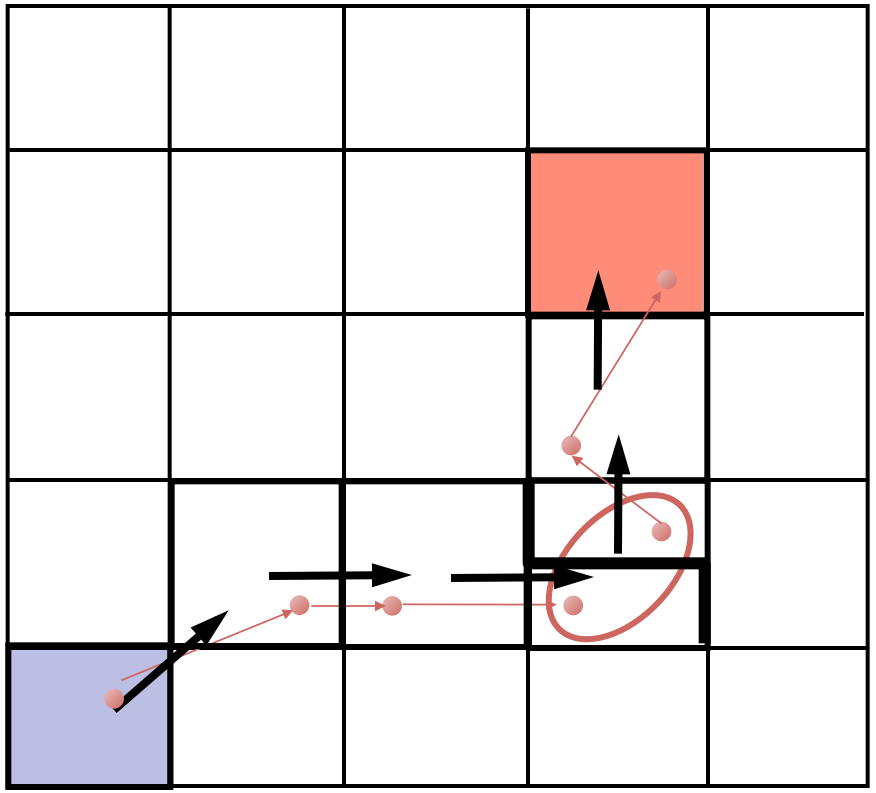
Maybe, or we just found a spurious counterexample.

How to check:

- take the concrete path through the program and construct the formula describing its relation
- feed this formula to a theorem prover
  - path feasible: true bug found, report and finish
  - path infeasible: no bug, refine abstraction

Note: how we get the concrete path will become obvious later.

Counter-example guided refinement



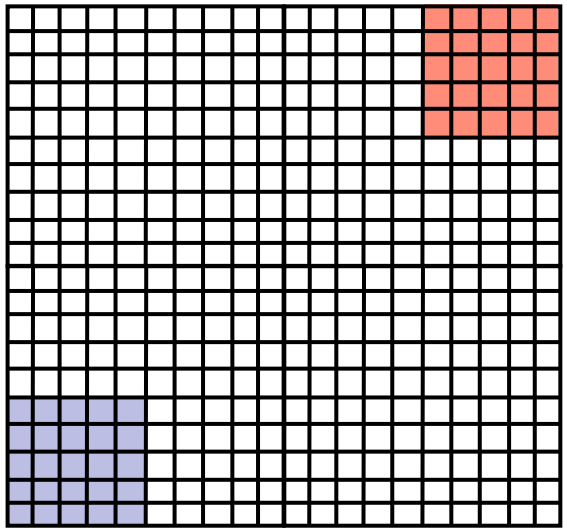
If path is infeasible, add more predicates to distinguish paths and rule out this particular one.

Idea: use infeasible path to generate predicates such that when added, this path will not appear any more.

Repeat until

- find a true counterexample
- system is proven safe
- timeout

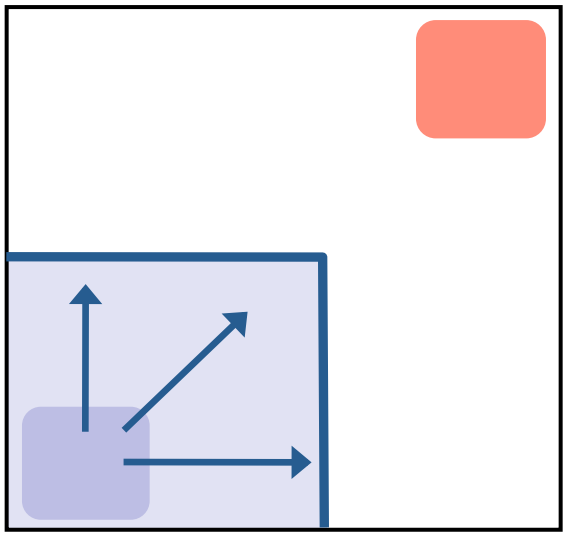
Lazy abstraction



Abstraction is expensive:

abstract states is finite, but still too large:

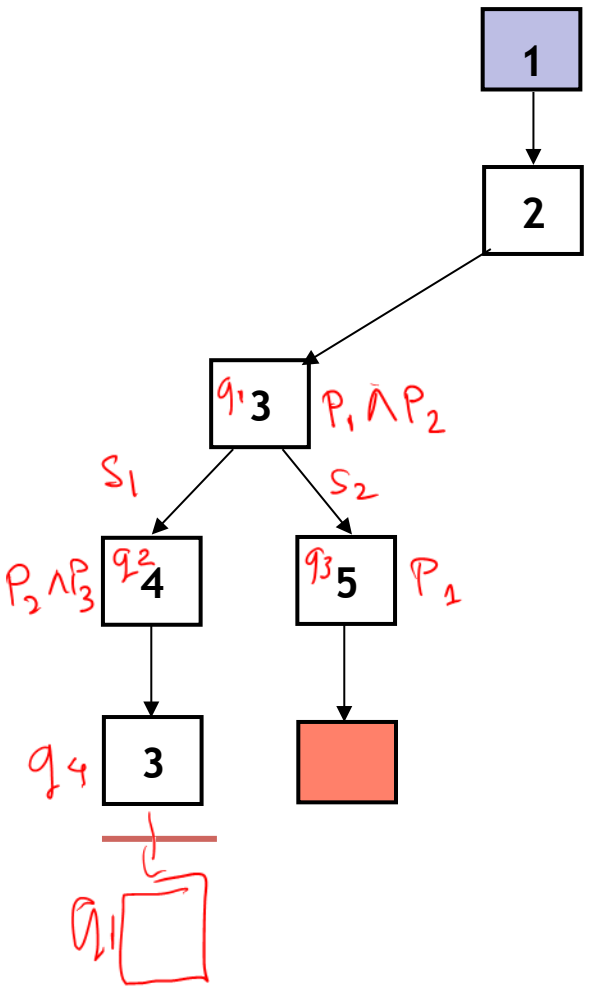
$$2^{\#\ predicates}$$



Observation:

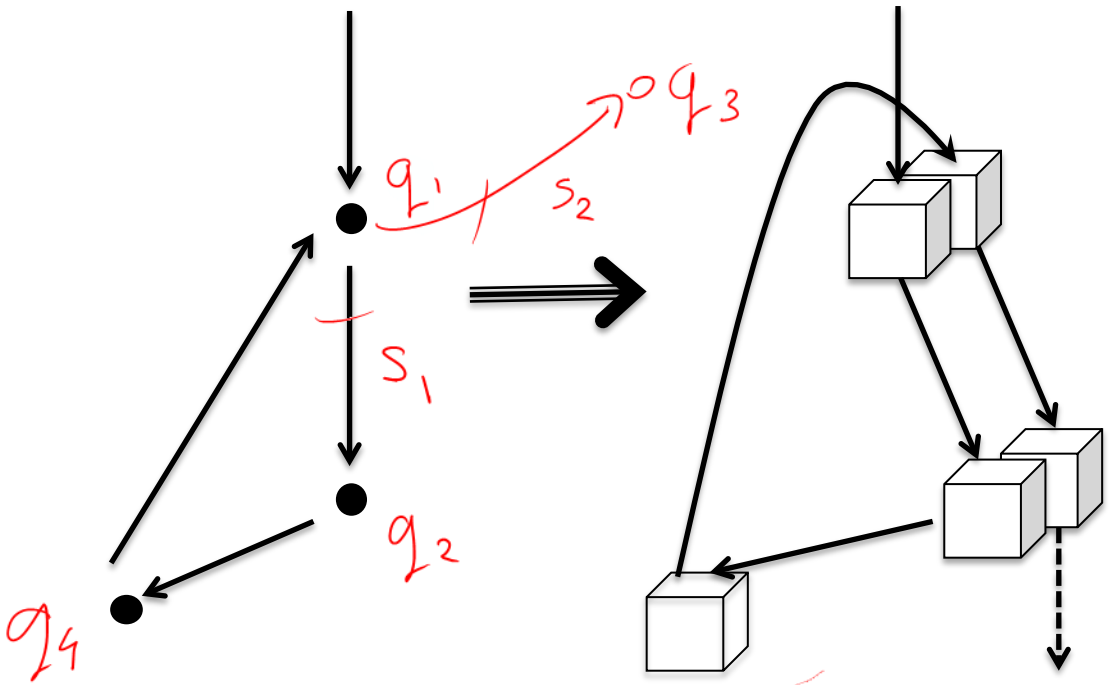
- not all predicates are needed everywhere
- only a fraction of states is reachable

Abstract reachability tree



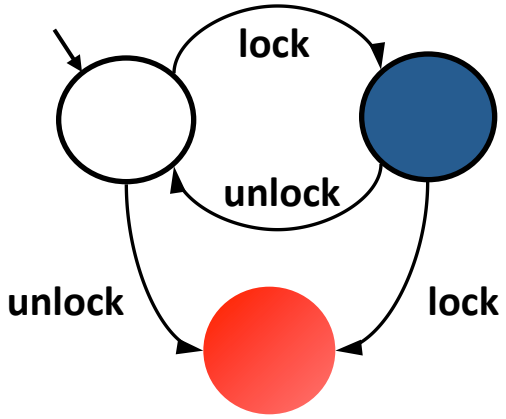
Unroll the CFG:

- pick a tree node
- add children
- if we revisit a state already seen, cut off



Example

```
Example ( ) {
1: do{
     lock();
     old = new;
       q = q->next;
2:    if (q != NULL){
3:        q->data = new;
          unlock();
        new ++;
     }
4: } while(new != old);
5:  unlock ();
   return;
}
```

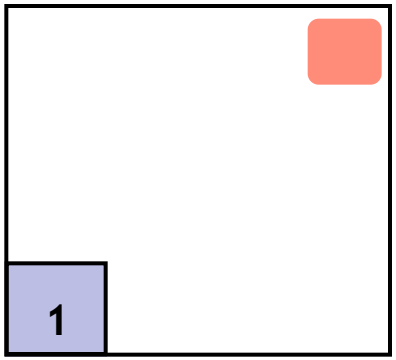


"An attempt to re-acquire an acquired lock or release a released lock will cause a deadlock."

Calls to **lock** and **unlock** must **alternate**.

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
    }
4:}while(new != old);
5: unlock ();
}
```

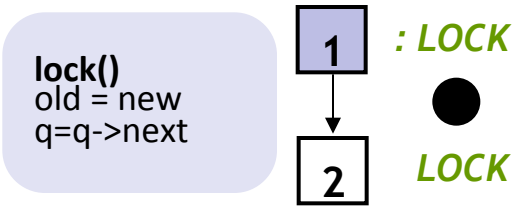


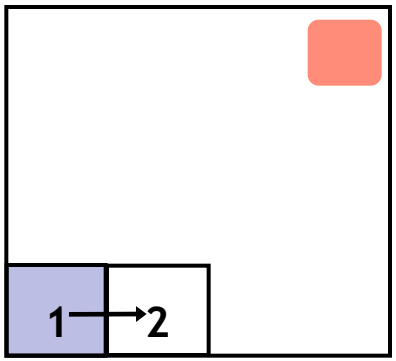
Predicates: *LOCK*

Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



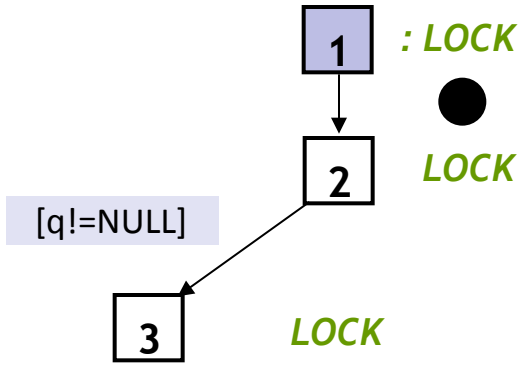


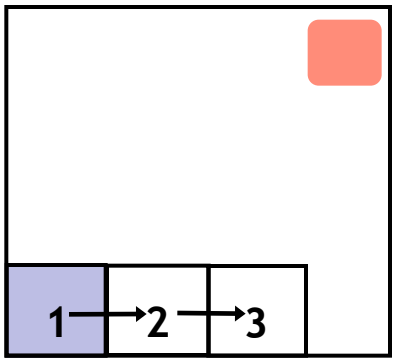
Predicates: *LOCK*

Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



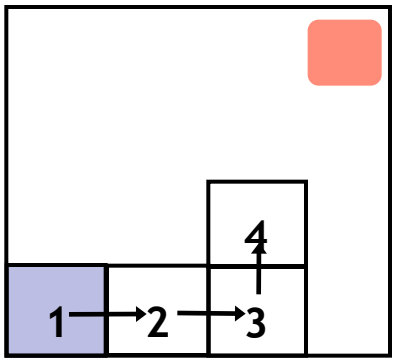


Predicates: *LOCK*

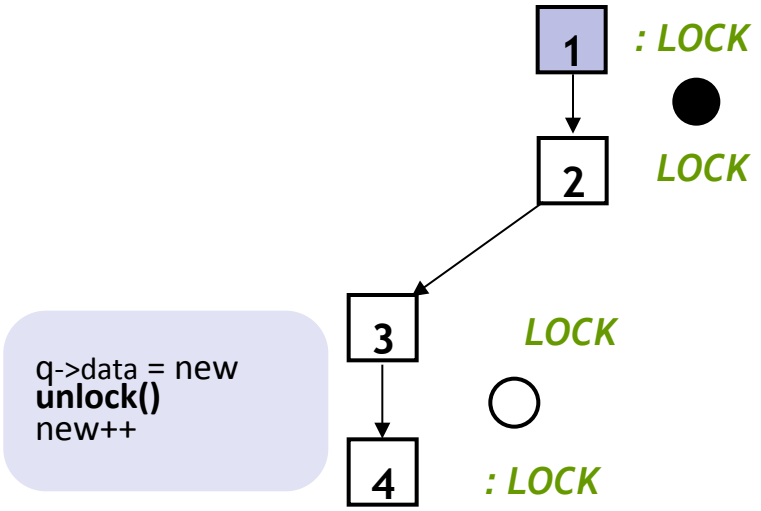
Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



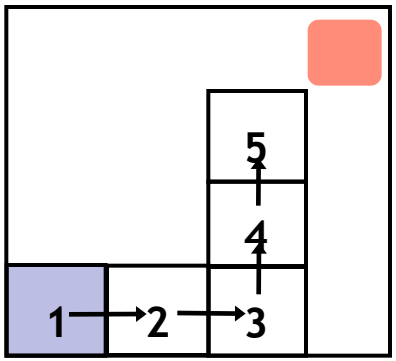
Predicates: *LOCK*



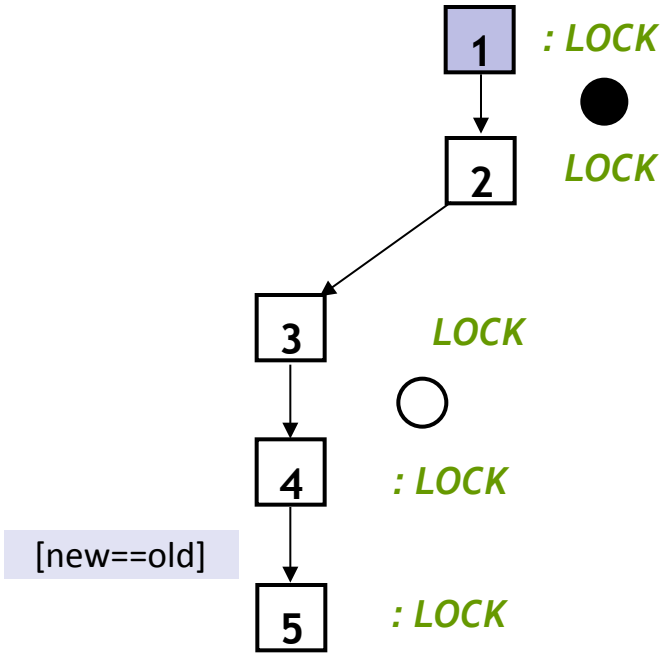
Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



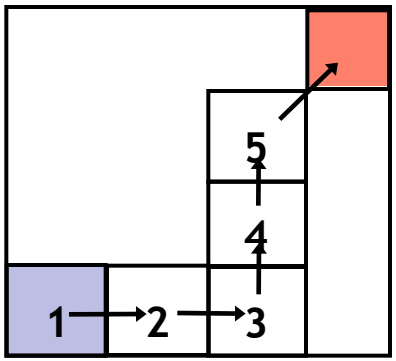
Predicates: *LOCK*



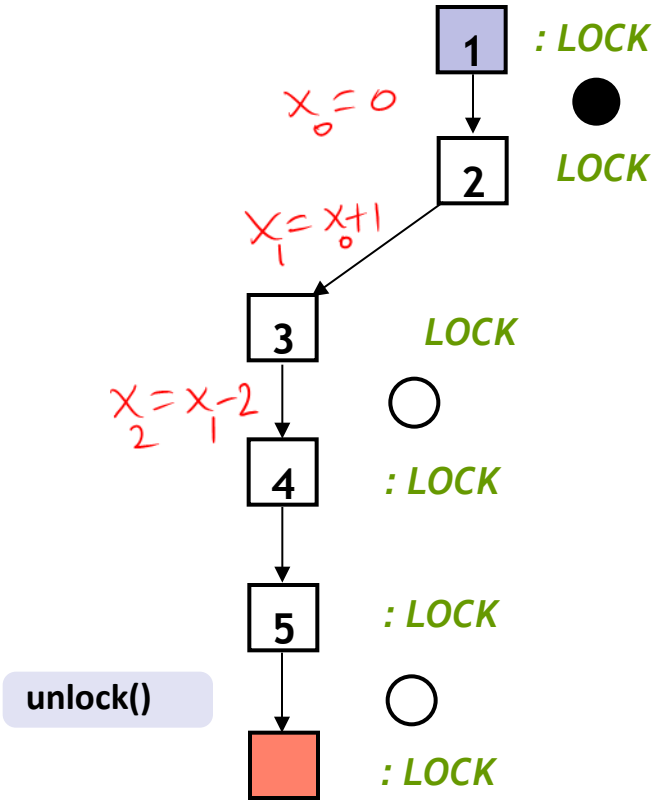
Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



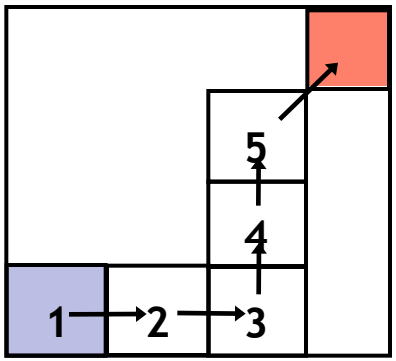
Predicates: *LOCK*



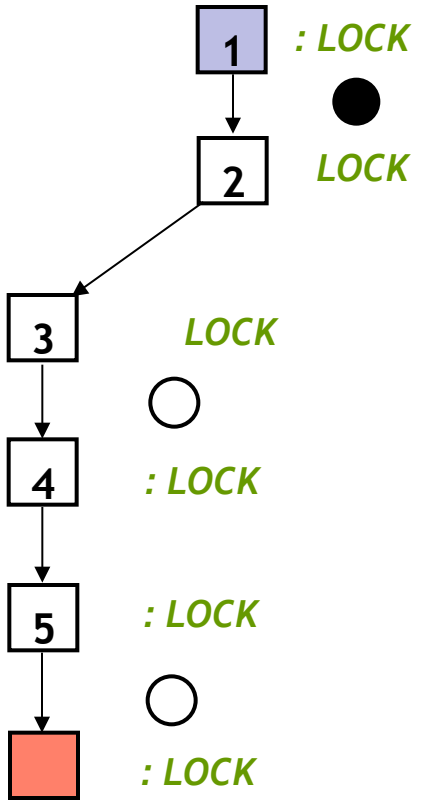
Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



Predicates: *LOCK*



lock()
old = new
q=q->next

[q!=NULL]

q->data = new
unlock()
new++

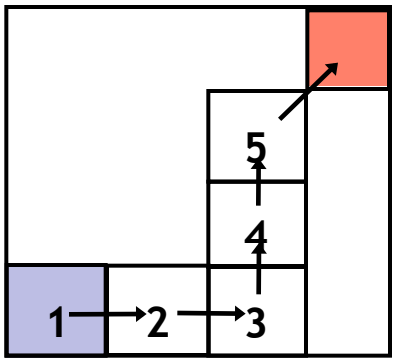
[new==old]

unlock()

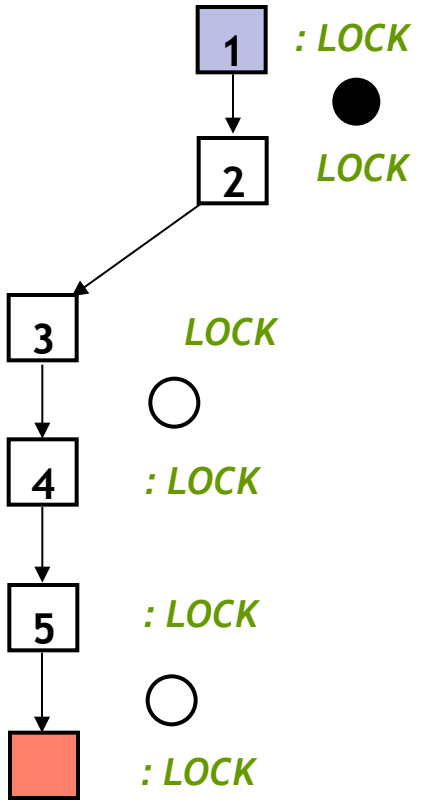
Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



Predicates: *LOCK*



Reachability Tree

old = new

new++

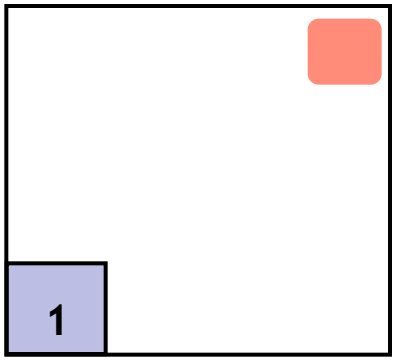
[new==old]

Inconsistent

new == old

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```

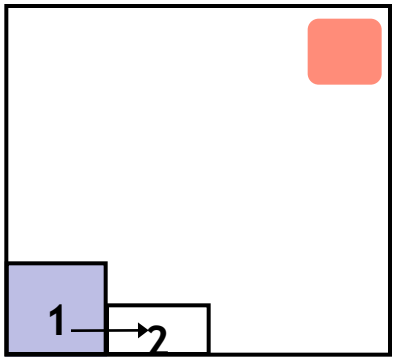


Predicates: *LOCK, new==old*

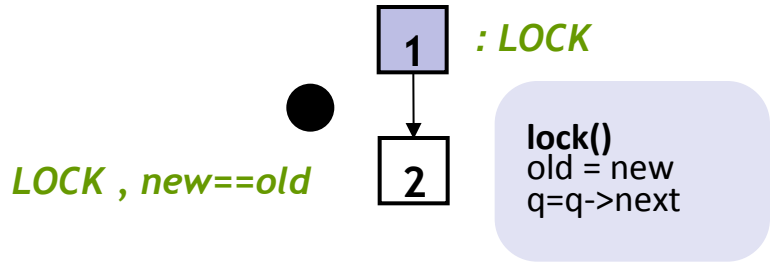
Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



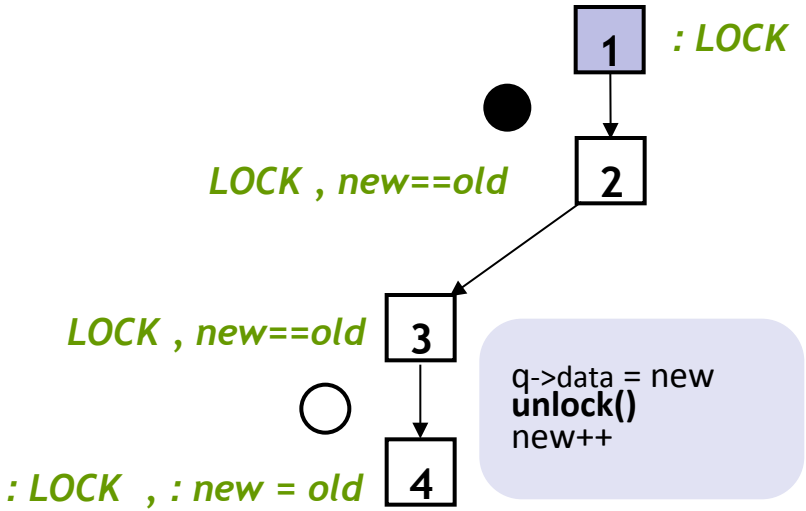
Predicates: *LOCK, new==old*

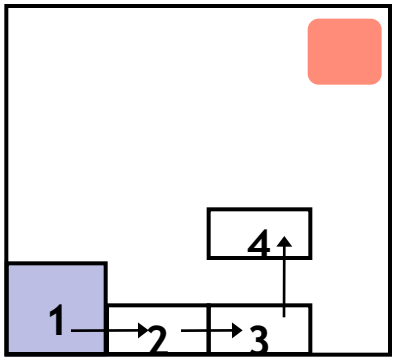


Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



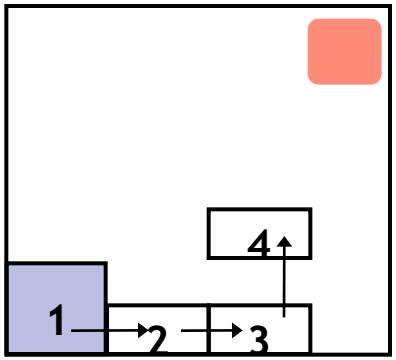


Predicates: *LOCK, new==old*

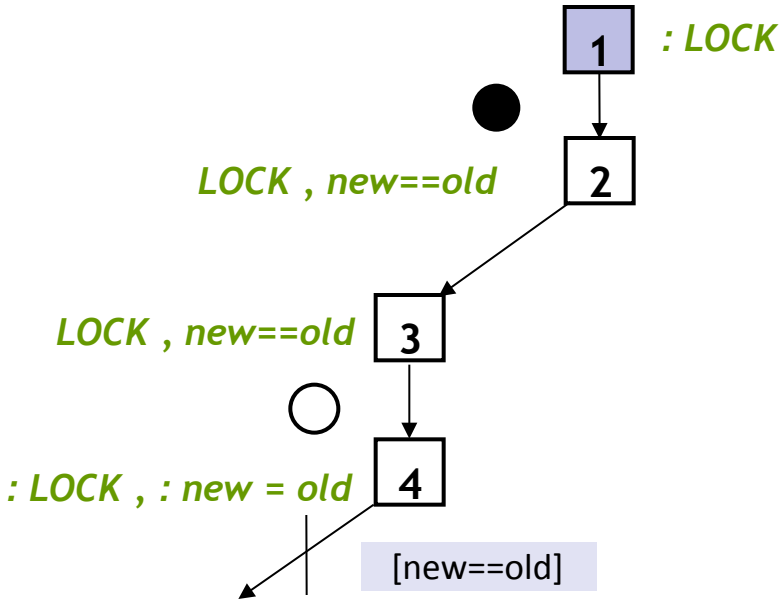
Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



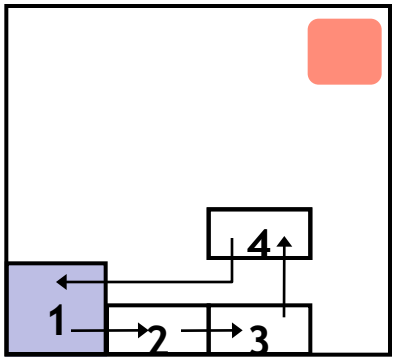
Predicates: *LOCK, new==old*

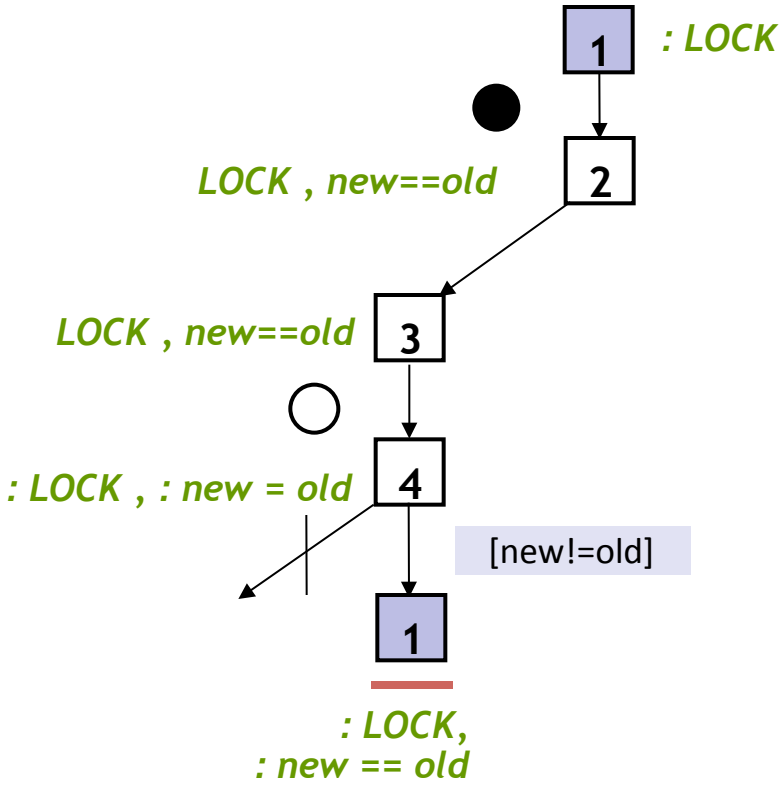


Reachability Tree

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```

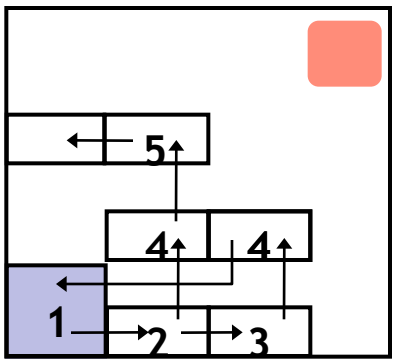




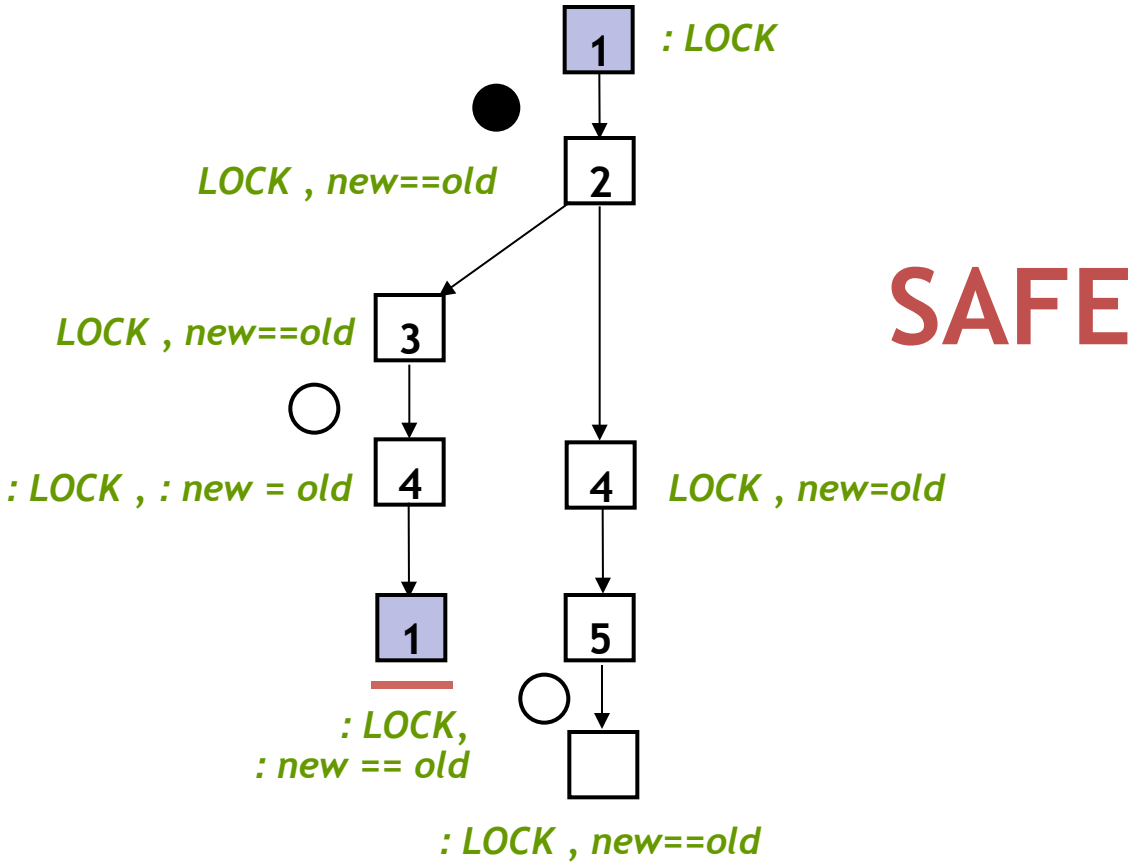
Predicates: *LOCK, new==old*

Example

```
Example ( ) {
1: do{
    lock();
    old = new;
    q = q->next;
2:  if (q != NULL){
3:    q->data = new;
      unlock();
      new ++;
    }
4:}while(new != old);
5: unlock ();
}
```



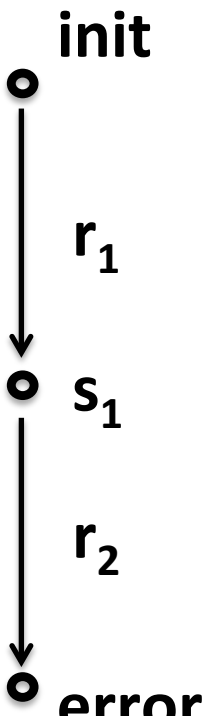
Predicates: *LOCK, new==old*



SAFE

Reachability Tree

What kind of predicates are needed?



Suppose our path consists of states s_1, $s_{2, \ldots,}$ s_n.

What we want are predicates P_i (corresponding to s_i), such that

P_1 -> P_2 -> … P_{n-1} and

P_{n-1} and P_n are inconsistent.

→ the path has been ruled out.

Note: it is always sound to pick predicates at random!