CSCI 740 - Programming Language Theory

Lecture 36
Software Model Checking
Instructor: Hossein Hojjat

December 1, 2017

Model Checking (so far)

The promise of model checking

- Exhaustive exploration of the state space of a program
- Push-button verification of arbitrary temporal logic formulas
- Dramatic performance improvements from state-space reduction techniques (e.g. partial order reduction)

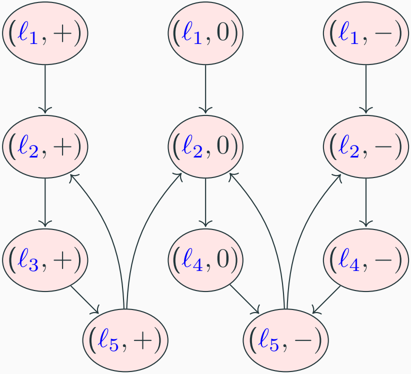
But

- It only works for programs with finite state space

Abstraction to the Rescue

- We can abstract the infinite state space into a finite one
- Every abstract state corresponds to an infinite set of states
- Is this the same thing as abstract interpretation?

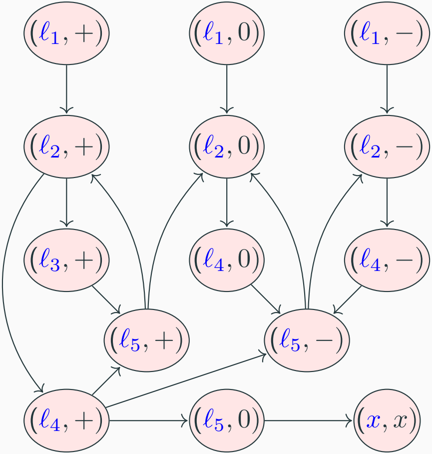
```
       void main(){
         int x = *;(ℓ1)
         while(*){
(ℓ2)       if(x>0)
(ℓ3)         x = 2*x;
           else
(ℓ4)         x = x-1;
(ℓ5)       x = abs(*)/x;
         }
       }
```



Abstraction to the Rescue

- Abstractions usually have to be tailored to the program and property of interest
- Imprecision on the abstraction can lead to spurious paths

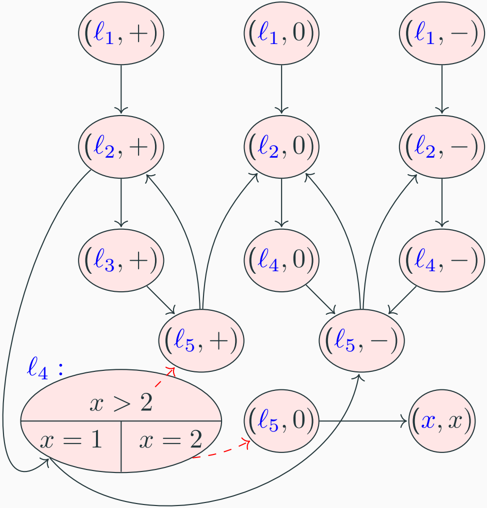
```
     void main(){
       int x = *;(ℓ1)
       while(*){
(ℓ2)     if(x>1)
(ℓ3)       x = 2*x;
         else
(ℓ4)       x = x−2;
(ℓ5)     x = abs(*)/x;
       }
     }
```



Spurious Path under Microscope

- Abstractions usually have to be tailored to the program and property of interest
- Imprecision on the abstraction can lead to spurious paths

```
      void main(){
        int x = *;(ℓ1)
        while(*){
(ℓ2)      if(x>1)
(ℓ3)        x = 2*x;
          else
(ℓ4)        x = x−2;
(ℓ5)      x = abs(*)/x;
        }
      }
```



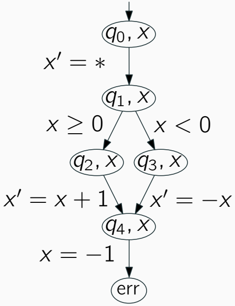
Abstraction Refinement

- We need a simple way to come up with abstractions
- Our abstractions must be flexible
- We need to be able to refine them on demand
- This is how we identify spurious paths and eliminate them

Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: $(p_1(s), \cdots, p_n(s))$
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

int x = *;
if (x $\geq$ 0) **then**
x++;
else x = $-$x
assert(x $\neq$ -1);

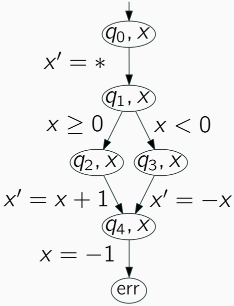


state: (x: Int)

Predicate Abstraction

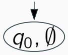
- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: $(p_1(s), \cdots, p_n(s))$
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

int x = *;
if (x $\geq$ 0) **then**
x++;
  else x = $-$x
assert(x $\neq$ -1);



$p_1 \equiv (x \geq 0)$

$p_2 \equiv (x < 0)$



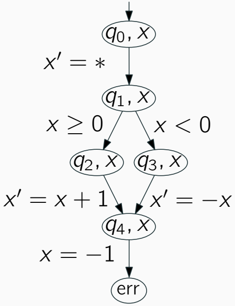
state: (x: Int)

state: (p_1: Bool,p_2: Bool)
$p_1(x) \equiv (x \geq 0)$,$p_2(x) \equiv (x < 0)$

Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: $(p_1(s), \cdots, p_n(s))$
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

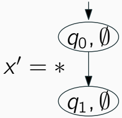
int x = *;
if (x $\geq$ 0) **then**
x++;
  else x = $-$x
assert(x $\neq$ -1);



state: (x: Int)

$p_1 \equiv (x \geq 0)$

$p_2 \equiv (x < 0)$

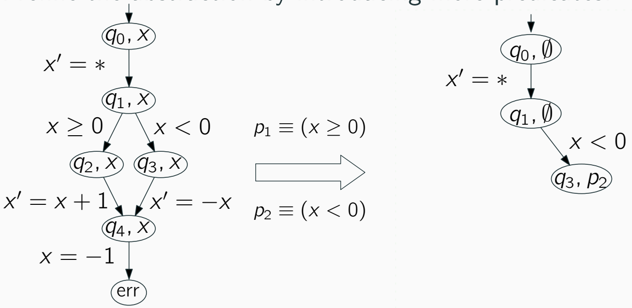


state: (p_1: Bool,p_2: Bool)
$p_1(x) \equiv (x \geq 0)$,$p_2(x) \equiv (x < 0)$

Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: $(p_1(s), \cdots, p_n(s))$
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

int x = *;
if (x $\geq$ 0) **then**
x++;
else x = $-$x
assert(x $\neq$ -1);



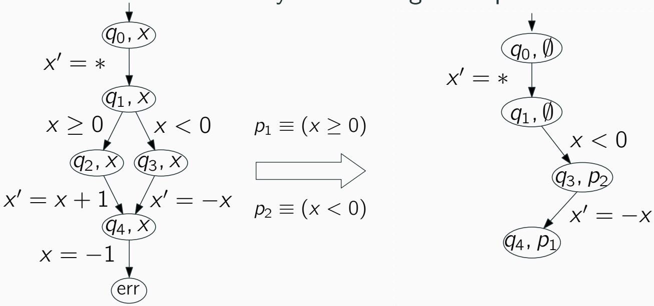
state: (x: Int)

state: (p_1: Bool,p_2: Bool)
$p_1(x) \equiv (x \geq 0)$,$p_2(x) \equiv (x < 0)$

Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: $(p_1(s), \cdots, p_n(s))$
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

int x = *;
if (x $\geq$ 0) **then**
x++;
  else x = $-$x
assert(x $\neq$ -1);



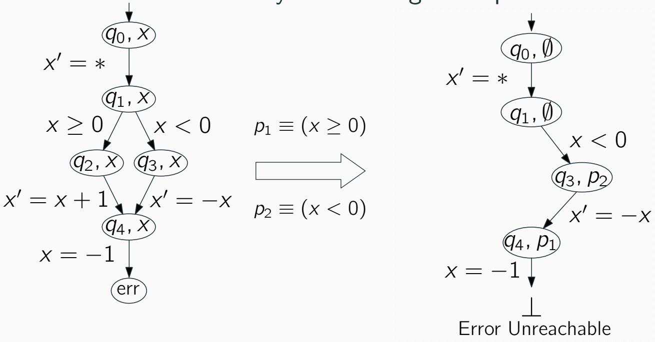
state: (x: Int)

state: (p_1: Bool,p_2: Bool)
$p_1(x) \equiv (x \geq 0)$,$p_2(x) \equiv (x < 0)$

Predicate Abstraction

- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: $(p_1(s), \cdots, p_n(s))$
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

int x = *;
if (x $\geq$ 0) **then**
x++;
else x = $-$x
assert(x $\neq$ -1);



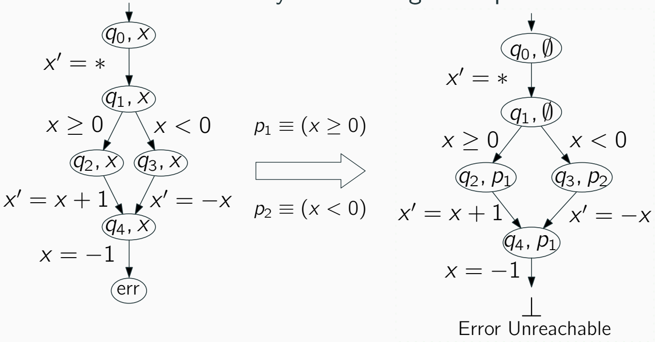
state: (x: Int)

state: (p_1: Bool,p_2: Bool)
$p_1(x) \equiv (x \geq 0)$,$p_2(x) \equiv (x < 0)$

Predicate Abstraction

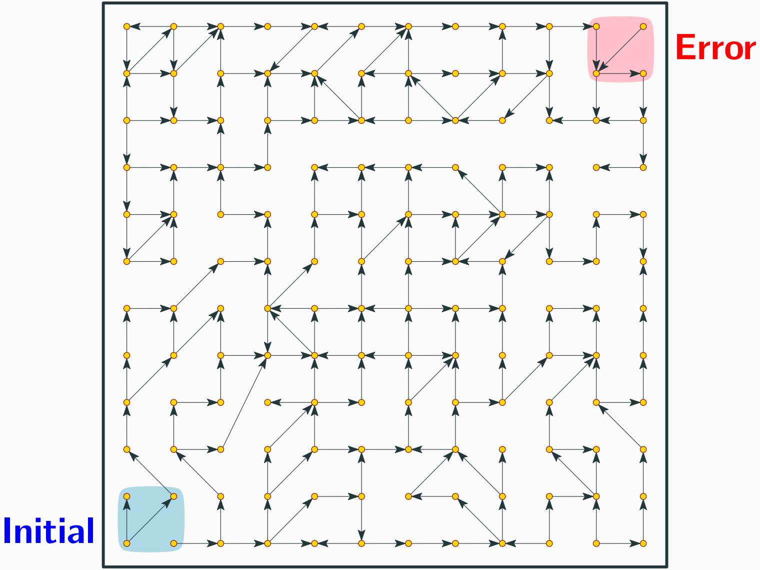
- Software has too many state variables (practically infinite space)
- Predicate Abstraction [Graf/Saïdi 97]: Only keep track of predicates on data: $(p_1(s), \cdots, p_n(s))$
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

int x = *;
if (x $\geq$ 0) **then**
x++;
else x = $-$x
assert(x $\neq$ -1);

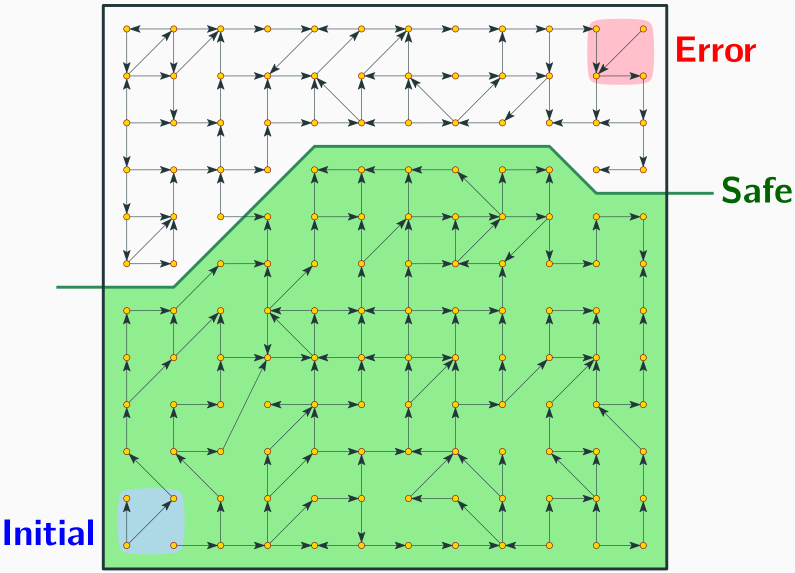


state: (x: Int)

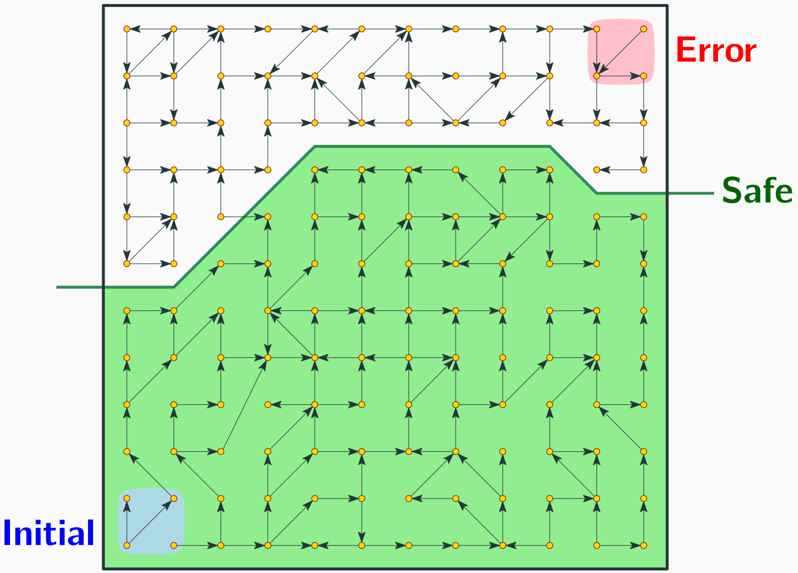
state: (p_1: Bool,p_2: Bool)
$p_1(x) \equiv (x \geq 0)$,$p_2(x) \equiv (x < 0)$



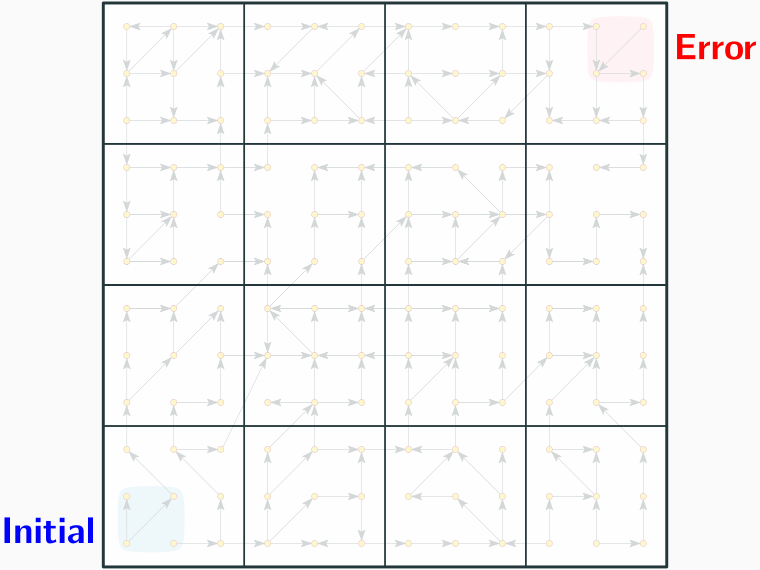
- Safety verification: Prove no path from initial to final state.
- Problem: Huge (infinite) state graph



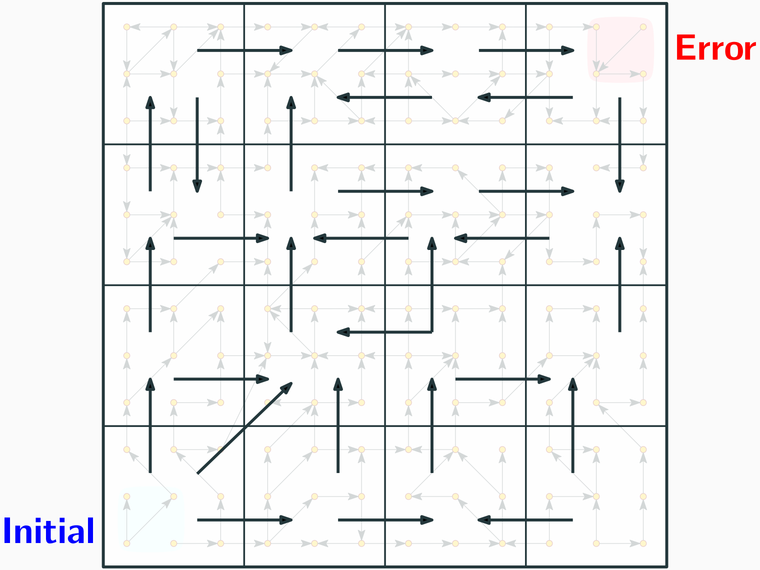
- Safety verification: Prove no path from initial to final state.
- Problem: Huge (infinite) state graph



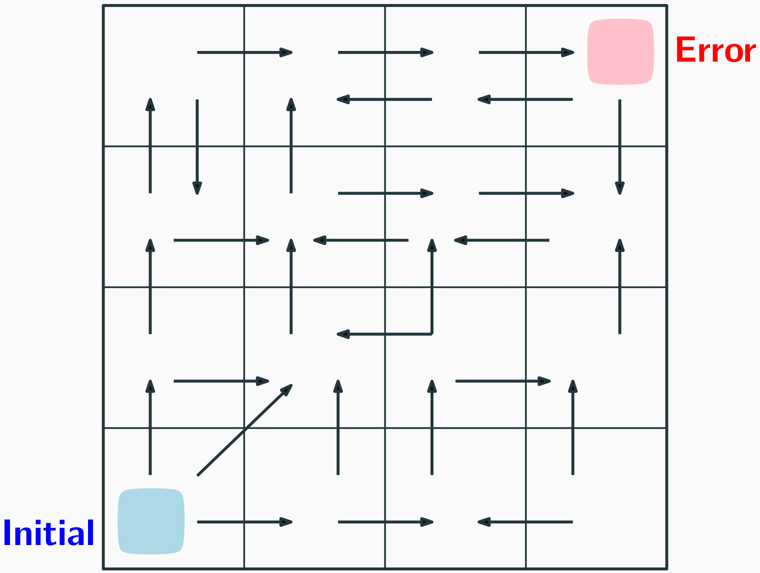
Predicate Abstraction: Merge states satisfying same predicates to one abstract state



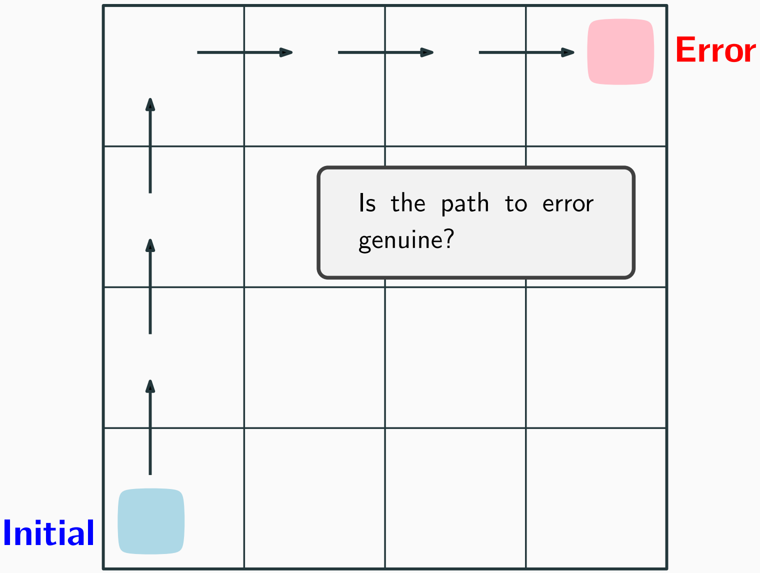
Predicate Abstraction: Merge states satisfying same predicates to one abstract state



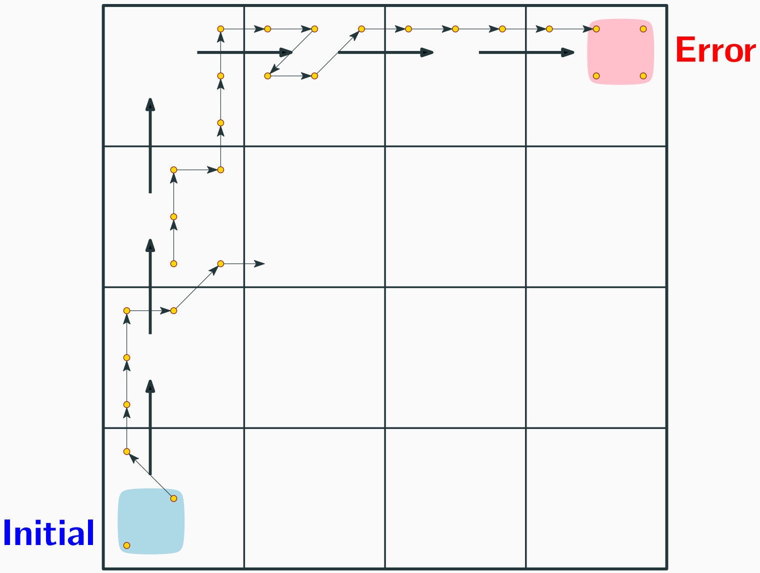
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



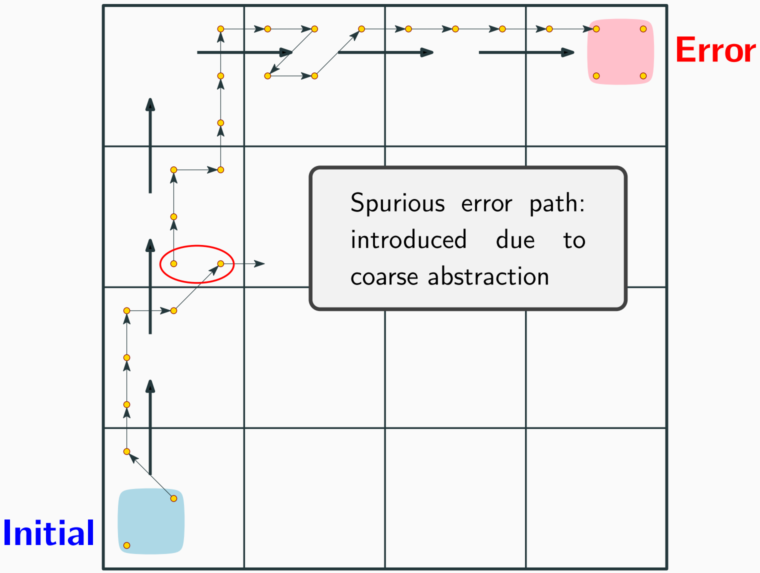
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



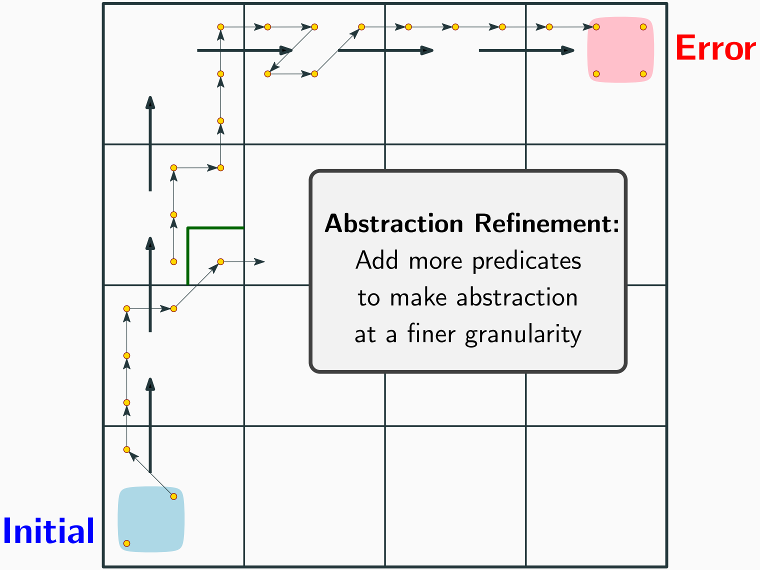
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



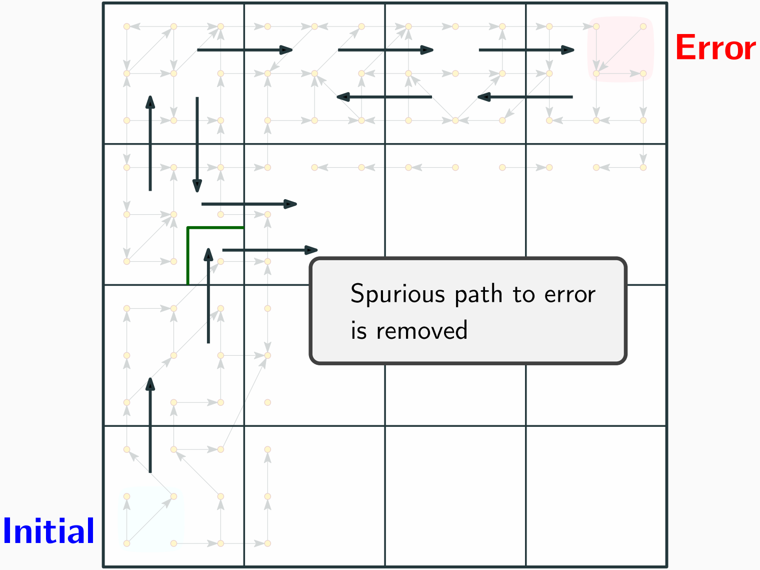
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



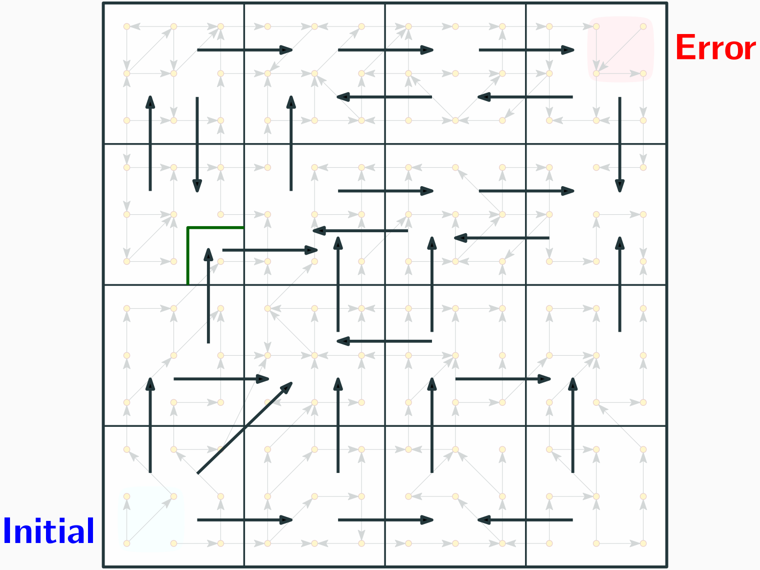
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



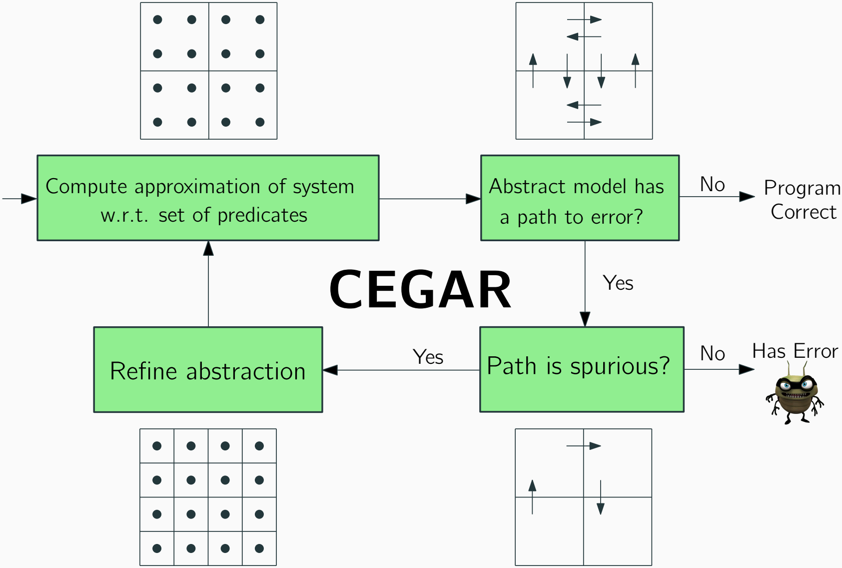
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



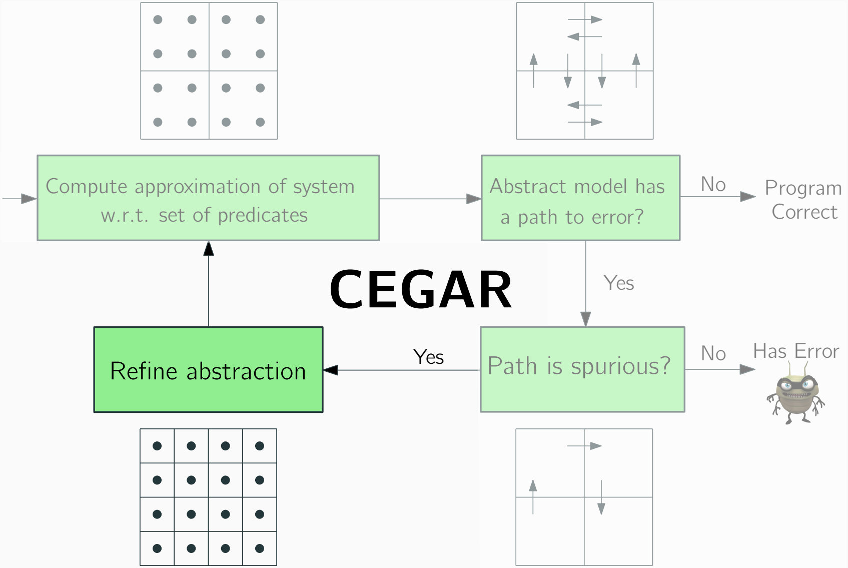
Continue searching in the new abstract state space.

CounterExample-Guided Abstraction Refinement (CEGAR)

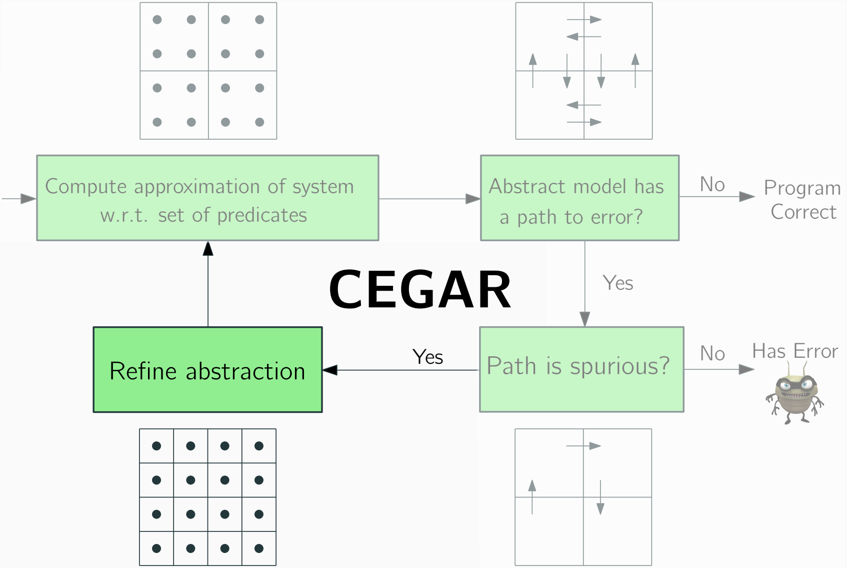
CounterExample-Guided Abstraction Refinement



CounterExample-Guided Abstraction Refinement

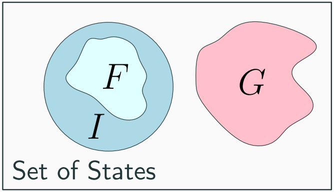


CounterExample-Guided Abstraction Refinement



Abstraction refinement: Craig interpolation

Craig Interpolation

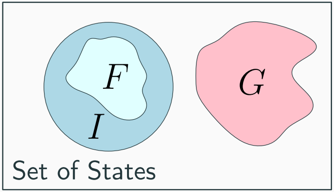


Craig interpolant for an **inconsistent** pair of formulae (F, G) is a formula I s.t.

1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \land G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

Interpolant summarizes the reason two formulae are inconsistent in their shared language

Craig Interpolation



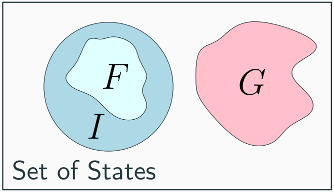
Craig interpolant for an **inconsistent** pair of formulae (F, G) is a formula I s.t.

1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

Interpolant summarizes the reason two formulae are inconsistent in their shared language

$$(A \wedge B \quad , \quad \neg B)$$

$A, B \in \mathbb{B}$



Craig interpolant for an **inconsistent** pair of formulae (F, G) is a formula I s.t.

1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

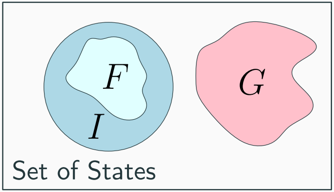
Interpolant summarizes the reason two formulae are inconsistent in their shared language

common variables

$$(A \wedge B \quad , \quad \neg B)$$

$$A, B \in \mathbb{B}$$

Craig Interpolation



Craig interpolant for an **inconsistent** pair of formulae (F, G) is a formula I s.t.

1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \land G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

Interpolant summarizes the reason two formulae are inconsistent in their shared language

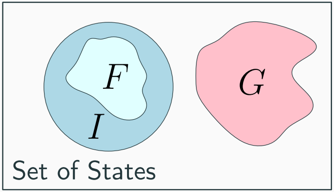
common variables

$$(A \land B \quad , \quad \neg B)$$

$$I = B$$

$A, B \in \mathbb{B}$

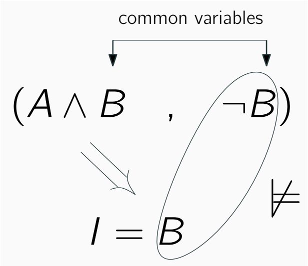
Craig Interpolation



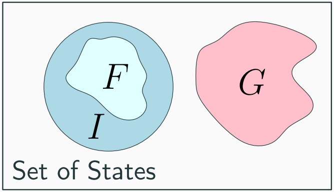
Craig interpolant for an **inconsistent** pair of formulae (F, G) is a formula I s.t.

1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

Interpolant summarizes the reason two formulae are inconsistent in their shared language



$A, B \in \mathbb{B}$



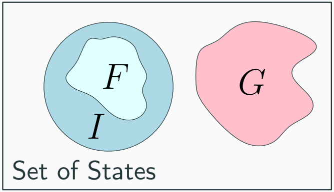
Craig interpolant for an **inconsistent** pair of formulae (F, G) is a formula I s.t.

1. $F \to \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

Interpolant summarizes the reason two formulae are inconsistent in their shared language

- Craig's Theorem [1957]:
  - First-order logic has the interpolation property
  - If $F \wedge G$ is unsatisfiable, then a Craig interpolant exists
- For certain theories (like linear arithmetic) interpolant can be derived from a refutation of $F \wedge G$ in polynomial time

Craig Interpolation



Craig interpolant for an **inconsistent** pair of formulae (F, G) is a formula I s.t.

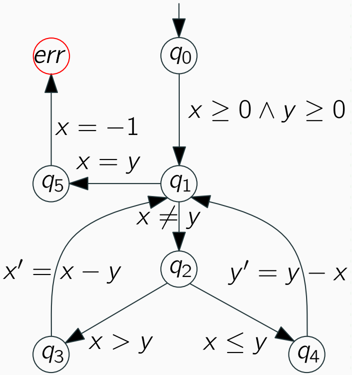
1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

Interpolant summarizes the reason two formulae are inconsistent in their shared language

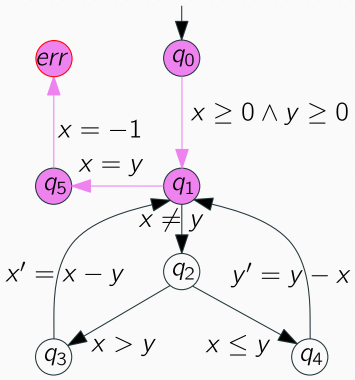
Let $\Gamma = \{F_1, F_2, \cdots, F_n\}$ be a set of formulae such that $\bigwedge \Gamma \equiv false$
An Inductive Interpolant Sequence for Γ is a set $\{I_0, I_1, \cdots, I_n\}$ such that:

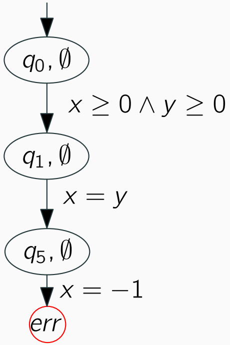
1. $I_0 =$ *true* and $I_n =$ *false*
2. For every $0 \leq \mathrm{j} < n$: $I_{\mathrm{j}} \wedge F_{\mathrm{j}+1} \rightarrow I_{\mathrm{j}+1}$
3. For every $0 < \mathrm{j} < n$: $\mathcal{L}(I_{\mathrm{j}}) \subseteq \mathcal{L}(F_1, \cdots, F_{\mathrm{j}}) \cap \mathcal{L}(F_{\mathrm{j}+1}, \cdots, F_n)$

CEGAR: Example

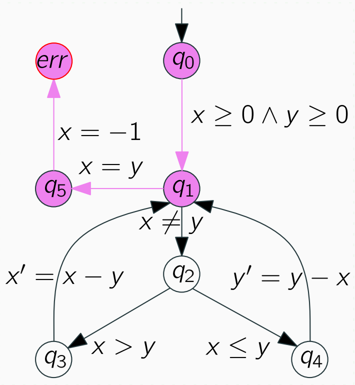


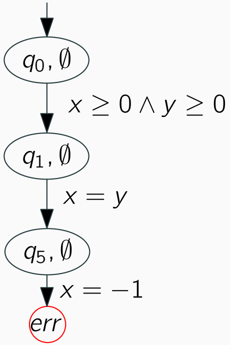
CEGAR: Example





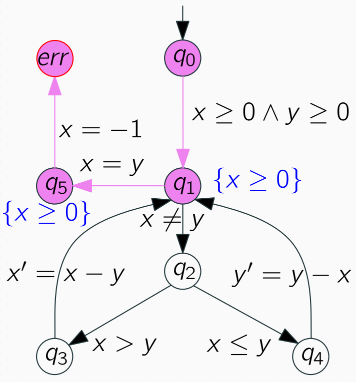
CEGAR: Example

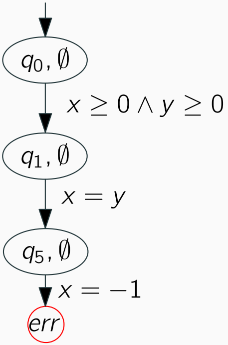




- $\Big(\underbrace{(x \geq 0 \wedge y \geq 0)}_{A_1} \wedge \underbrace{(x = y)}_{A_2} \wedge \underbrace{(x = -1)}_{A_3}\Big) =$ *false*
- Interpolant: $\{x \geq 0, x \geq 0\}$

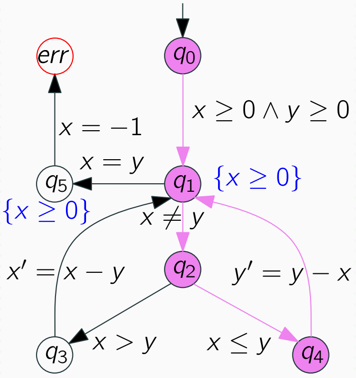
CEGAR: Example

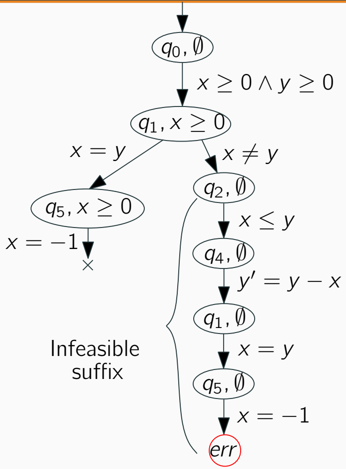




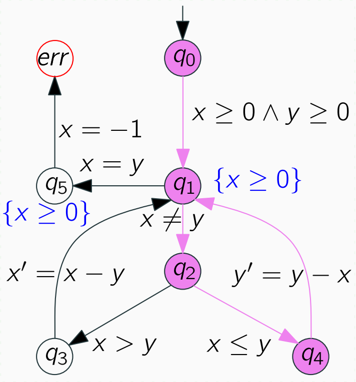
- $\Big(\underbrace{(x \geq 0 \wedge y \geq 0)}_{A_1} \wedge \underbrace{(x = y)}_{A_2} \wedge \underbrace{(x = -1)}_{A_3}\Big) =$ *false*
- Interpolant: $\{x \geq 0, x \geq 0\}$

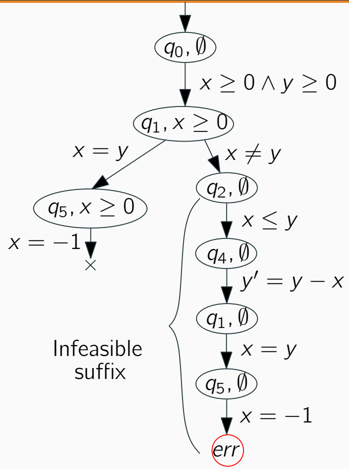
CEGAR: Example





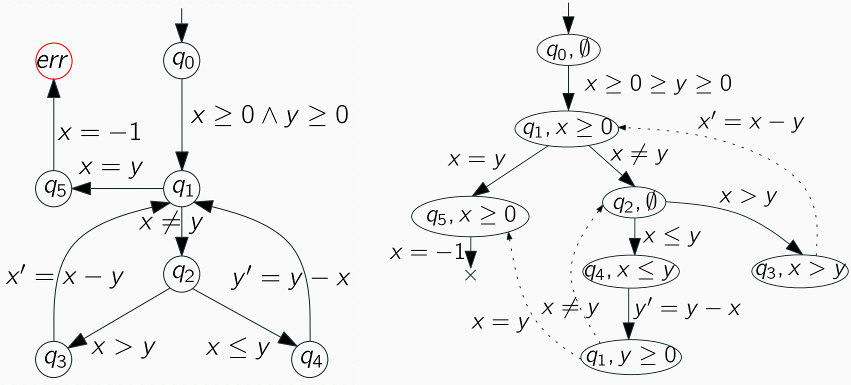
CEGAR: Example





- $\left(\underbrace{(x \leq y)}_{A_1} \wedge \underbrace{(y_1 = y - x)}_{A_2} \wedge \underbrace{(x = y_1)}_{A_3} \wedge \underbrace{(x = -1)}_{A_4}\right) = \textit{false}$
- Interpolant: $\{x \leq y, y_1 \geq 0, x \geq 0\}$

CEGAR: Example



Abstract Reachability Graph:

Unfolding the program in the abstract space