Introduction to Formal Methods

Lecture 26

Software Model Checking

Hossein Hojjat & Fatemeh Ghassemi

December 25, 2018

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)

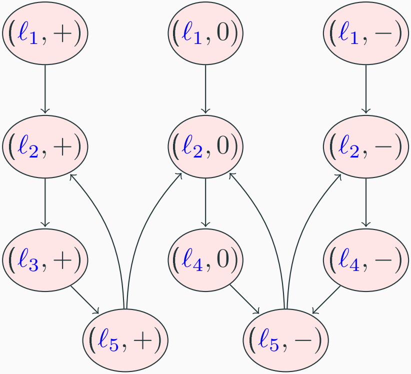
But

- 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

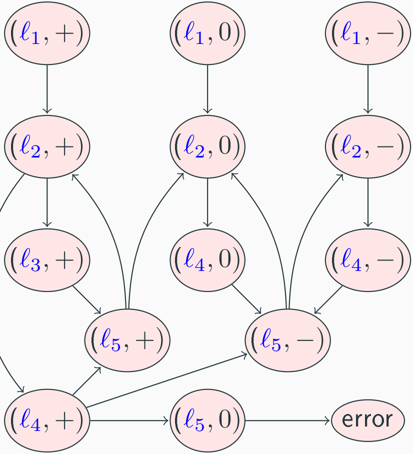
```
       void main(){
         int x = *;(ℓ1)
         while(*){
(ℓ2)       if(x>0)
(ℓ3)         x = 2*x;
           else
(ℓ4)         x = x-1;
(ℓ5)       x = abs(*)/x;
         }
       }
```



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

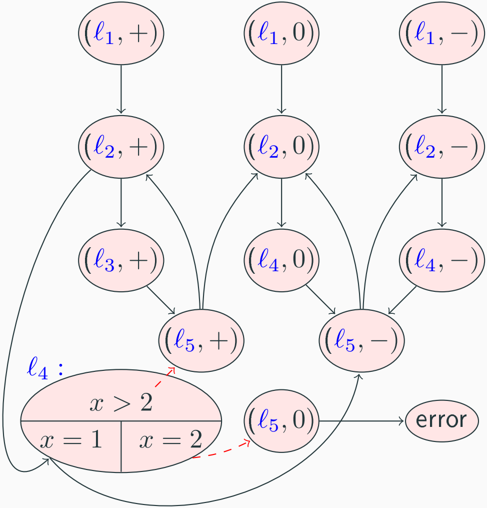
```
      void main(){
        int x = *;(ℓ1)
        while(*){
(ℓ2)      if(x>1)
(ℓ3)        x = 2*x;
          else
(ℓ4)        x = x-2;
(ℓ5)      x = abs(*)/x;
        }
      }
```



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: Only keep track of predicates on data
$$\mathcal{P} = \{p_1(\vec{x}), \cdots, p_n(\vec{x})\}$$
Graf and Saïdi: "Construction of abstract state graphs with PVS", CAV 1997
- Transition function can be computed by a theorem prover
- **Big idea:** We can refine the abstraction by introducing more predicates!

Abstraction

Example

- Let $\mathcal{P} = \{x > y, x = 2\}$
- What is the abstraction after executing the following piece of code?

$\{\textit{true}\}$

```
int x,y;
x = y + 1;
```

- $\forall x, y, x', y' \in \mathbb{Z}.(x' = y + 1) \wedge (y' = y) \rightarrow (x' > y')$ (valid)
- $\forall x, y, x', y' \in \mathbb{Z}.(x' = y + 1) \wedge (y' = y) \rightarrow (x' = 2)$ (not valid)

The abstraction is $\{(x > y)\}$

Abstraction

Example

- Let $\mathcal{P} = \{x < 2\}$
- What is the abstraction after executing the following piece of code?

$$\{x = 2\}$$

```
if (x > 2) x = x - 1;
```

- $\forall x, x'.(x = 2) \wedge (x > 2) \wedge (x' = x - 1) \rightarrow (x' < 2)$
  $\equiv \forall x, x'.\mathit{false} \rightarrow (x' < 2)$ (valid)

- The abstraction in this case :$\{(x < 2)\}$

Abstract Reachability Tree

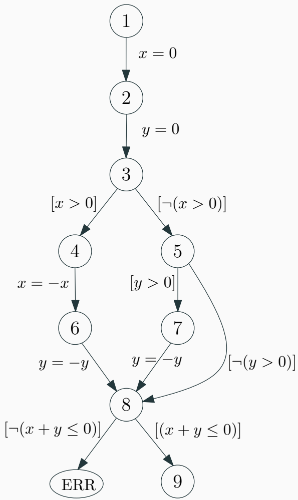
- Abstract state (l, ψ)
  - l: location in control flow graph
  - ψ: predicate abstraction

- Abstract reachability tree (ART) is a tree $G = (V_{\mathcal{A}}, \rightarrow, I)$
- $V_{\mathcal{A}}$ is a set of abstract states
- $\rightarrow \subseteq V_{\mathcal{A}} \times V_{\mathcal{A}}$ is the transition relation
  - Let c be the command between l_i and l_j in the CFG
  - $((l_i, \psi), (l_j, \phi)) \in \rightarrow \quad \text{iff} \quad \forall \vec{v}, \vec{v'}.\psi(\vec{v}) \wedge c(\vec{v}, \vec{v'}) \rightarrow \phi(\vec{v'})$
- $I \in V_{\mathcal{A}}$ is the initial abstract state

- (l_i, ψ) is leaf if there exists another node (l_j, ϕ) in the tree such that $\models \psi \rightarrow \phi$

ART Example

Control Flow Graph:

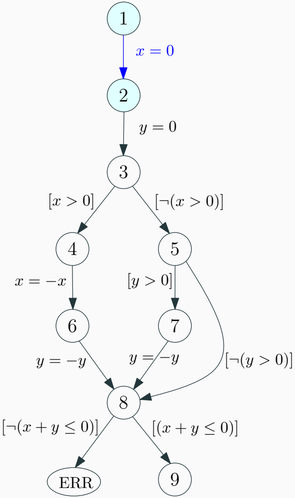


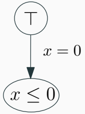
Prove $G(pc \neq \ell_5)$:

```
ℓ1 :   x = 0;
ℓ2 :   y = 0;
       if (x > 0) {
ℓ3 :     x = -x
ℓ4 :     y = -y
       } else {
         if (y > 0) y = -y
       }
       if(x + y <= 0)
ℓ5 :       ERR
       else
       // ...
```

ART Example

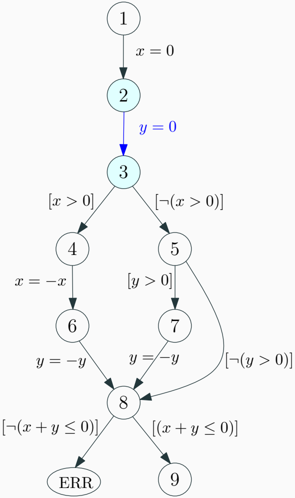
$$\mathcal{P} = \{\perp, (x <= 0), (x+y <= 0), (x-y <= 0)\}$$

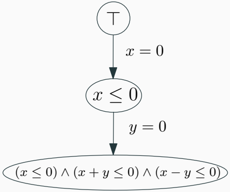




ART Example

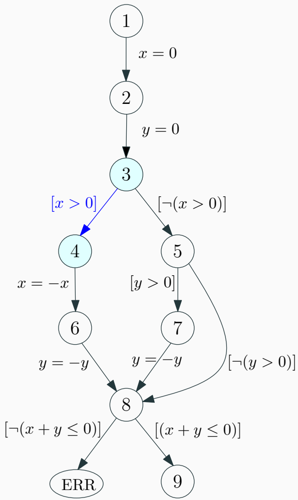
$$\mathcal{P} = \{\perp, (x <= 0), (x+y <= 0), (x-y <= 0)\}$$

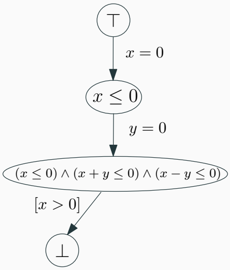




ART Example

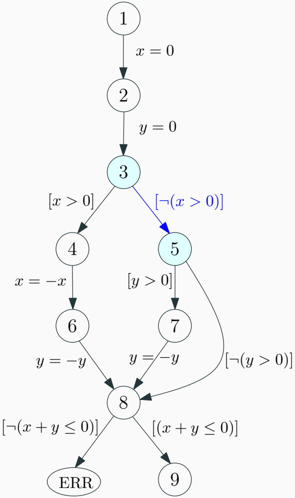
$$\mathcal{P} = \{\bot, (x <= 0), (x+y <= 0), (x-y <= 0)\}$$

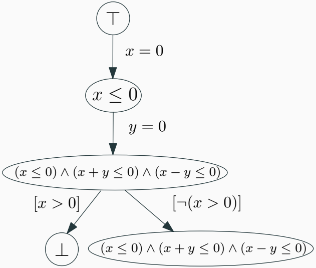




ART Example

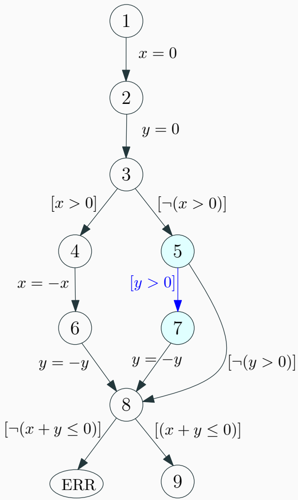
$$\mathcal{P} = \{\perp, (x <= 0), (x+y <= 0), (x-y <= 0)\}$$

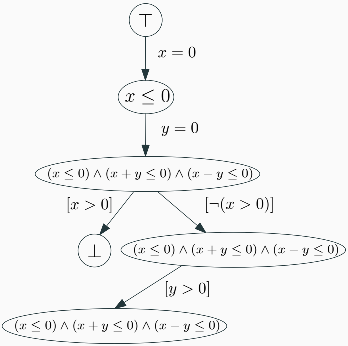




ART Example

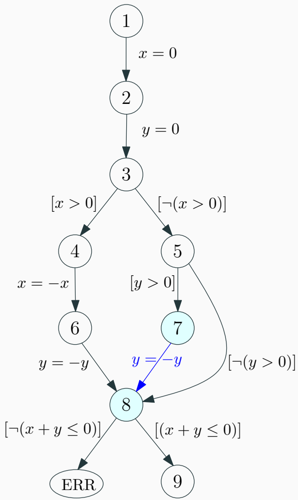
$$\mathcal{P} = \{\bot, (x <= 0), (x+y <= 0), (x-y <= 0)\}$$

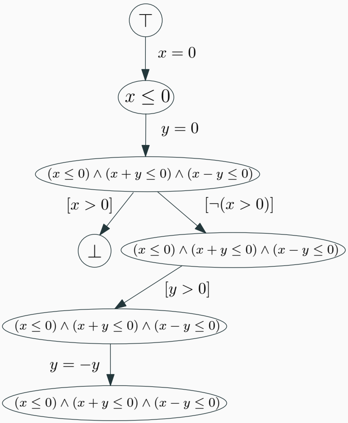




ART Example

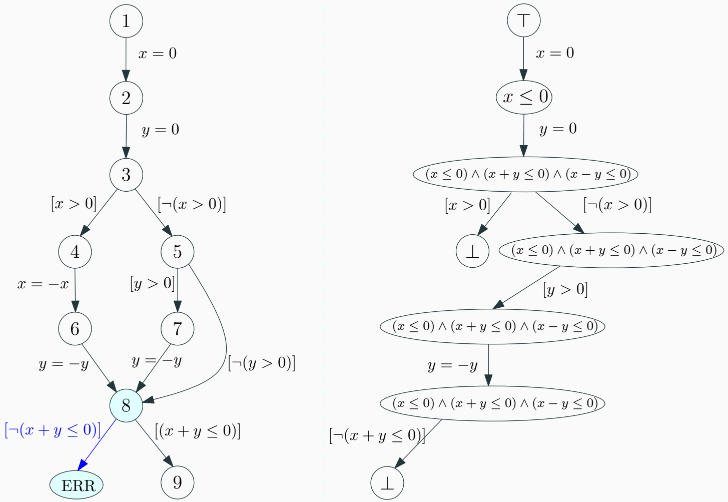
$$\mathcal{P} = \{\perp, (x <= 0), (x+y <= 0), (x-y <= 0)\}$$





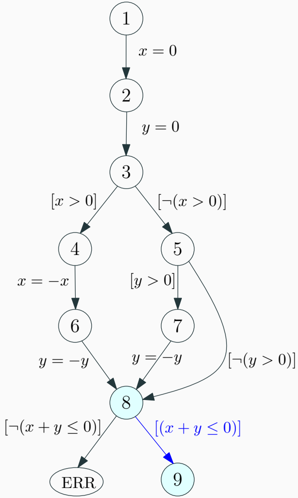
ART Example

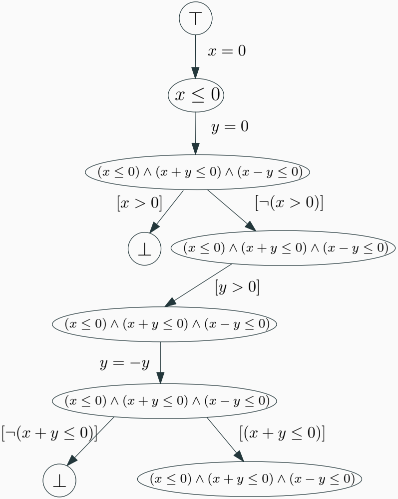
$$\mathcal{P} = \{\bot, (x <= 0), (x+y <= 0), (x-y <= 0)\}$$



ART Example

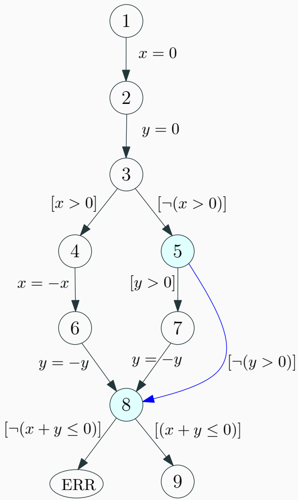
$$\mathcal{P} = \{\perp, (x <= 0), (x{+}y <= 0), (x{-}y <= 0)\}$$

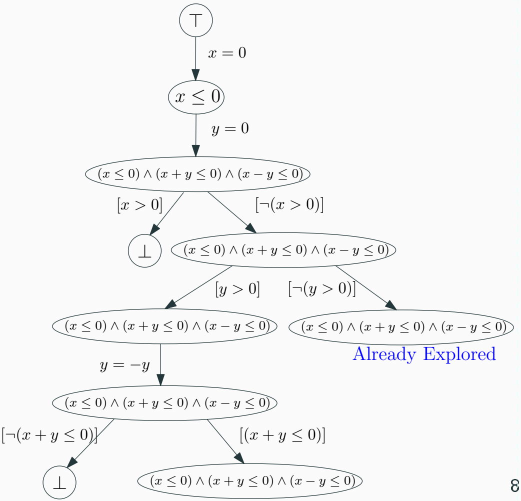




ART Example

$$\mathcal{P} = \{\bot, (x <= 0), (x+y <= 0), (x-y <= 0)\}$$



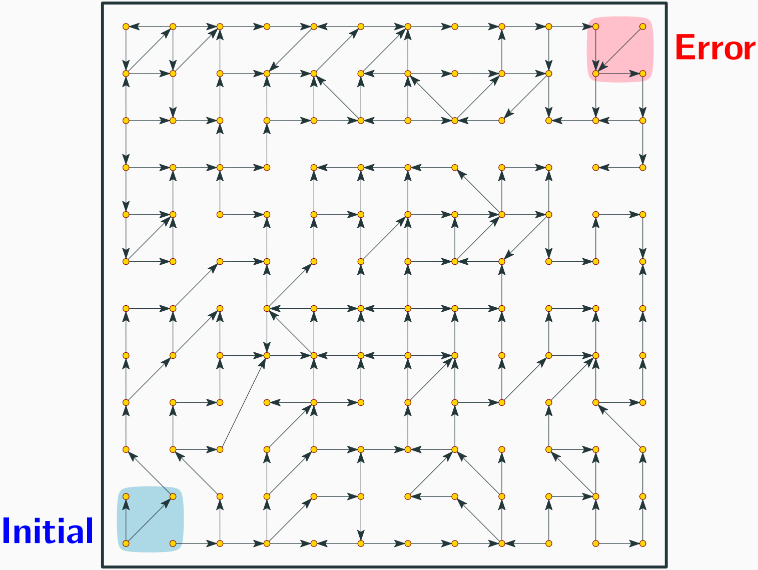


CEGAR

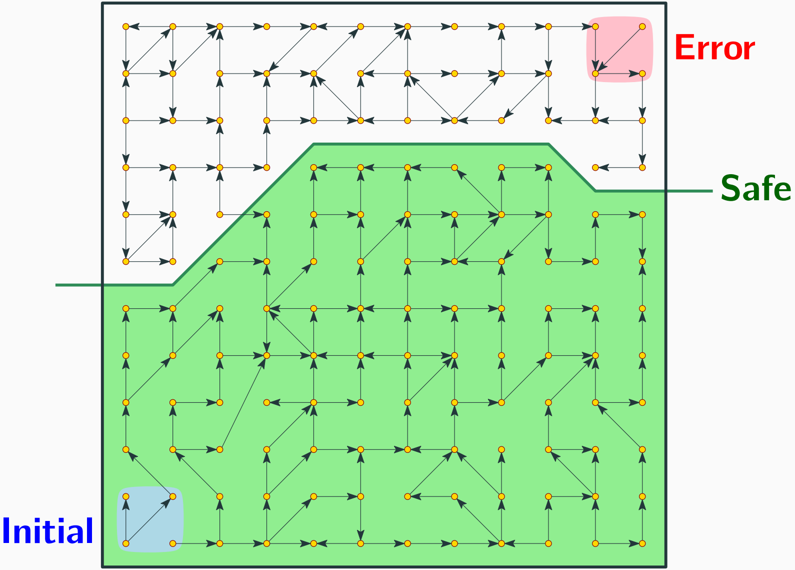
- During manual verification we add information on demand
  - When a proof fails we analyze the reason
  - We add pre-/post-conditions as necessary

- Idea: Use the same idea within automatic predicate abstraction
  - When a proof fails let the verifier analyze the reason
  - Ask it to refine the abstraction as necessary

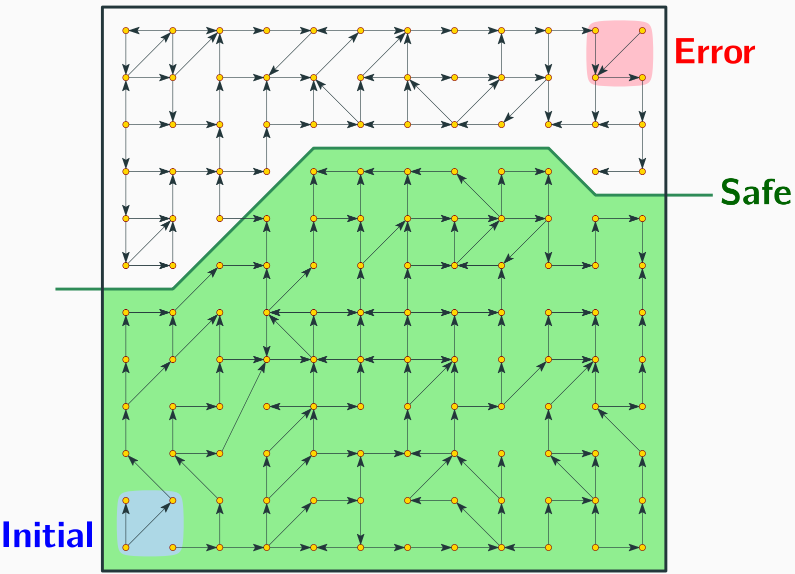
CEGAR: CounterExample-Guided Abstraction Refinement



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

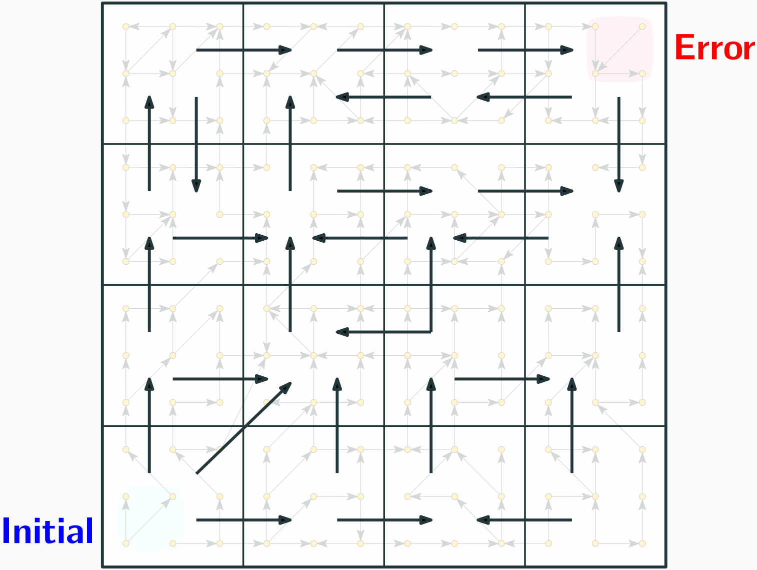


- Safety verification: Prove no path from initial to final state.
- Problem: Huge (infinite) state graph

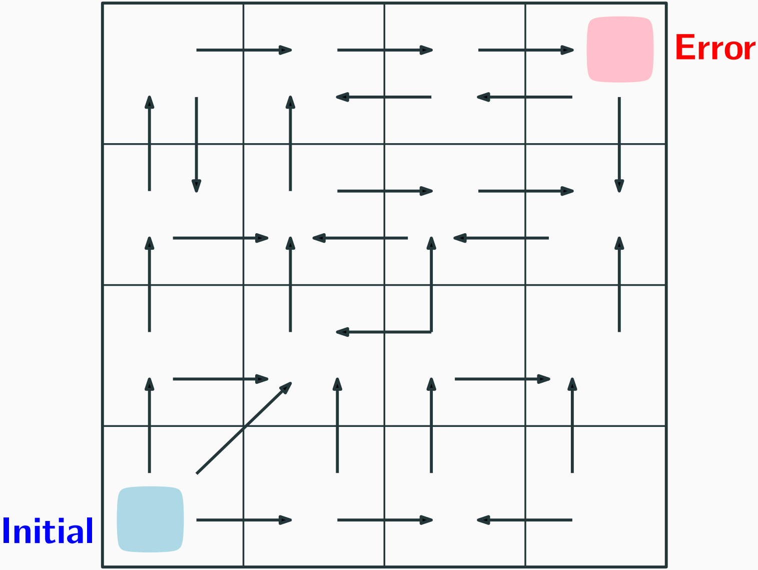


Predicate Abstraction: Merge states satisfying same predicates to one abstract state

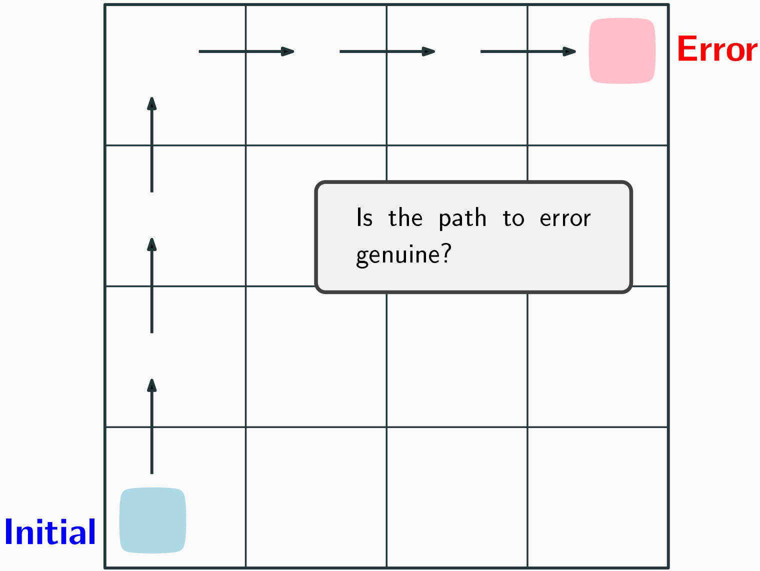
Predicate Abstraction: Merge states satisfying same predicates to one abstract state



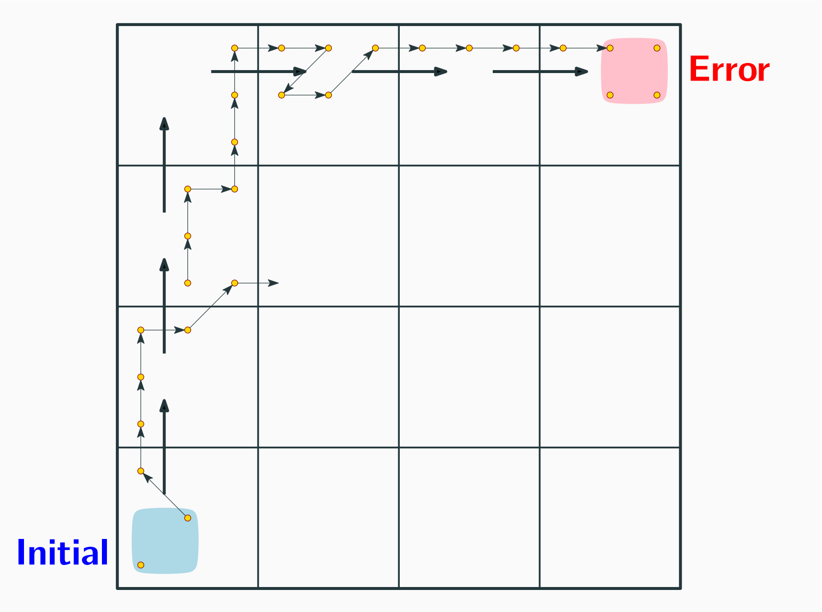
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



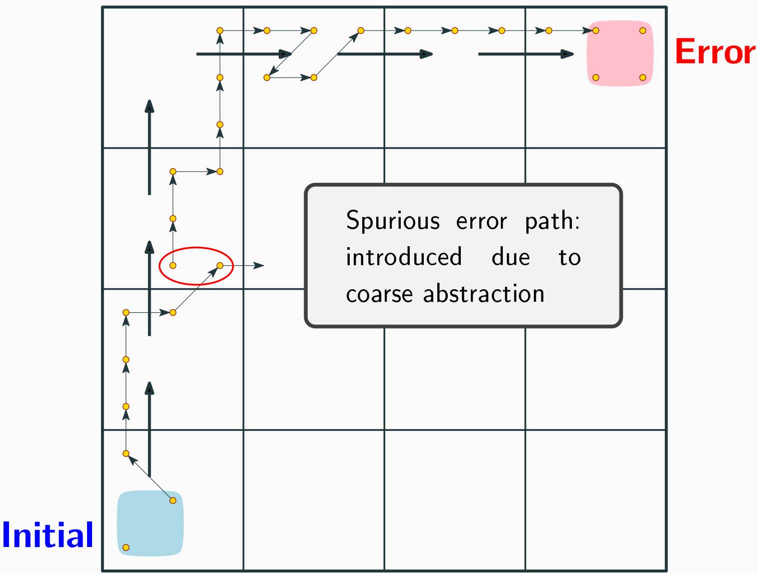
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



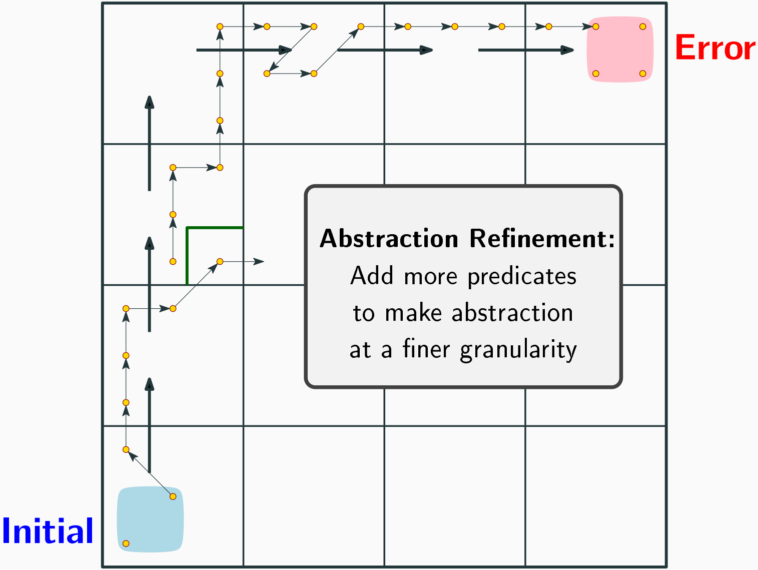
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



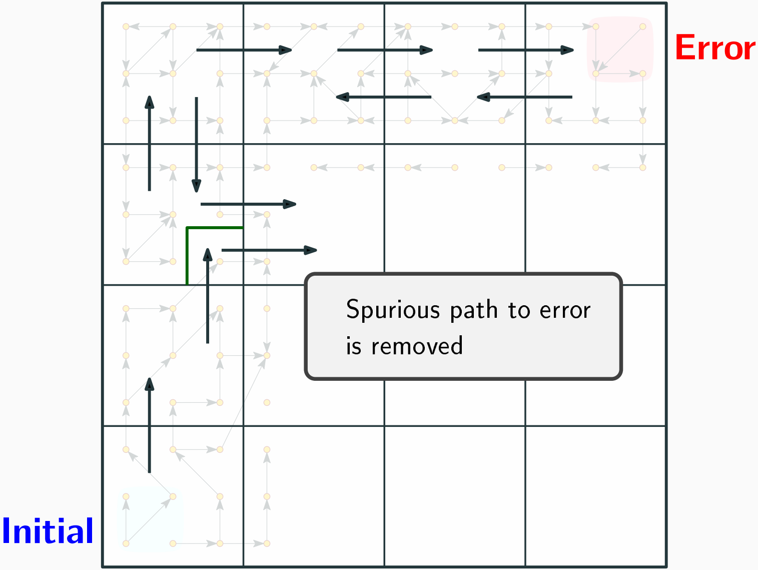
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



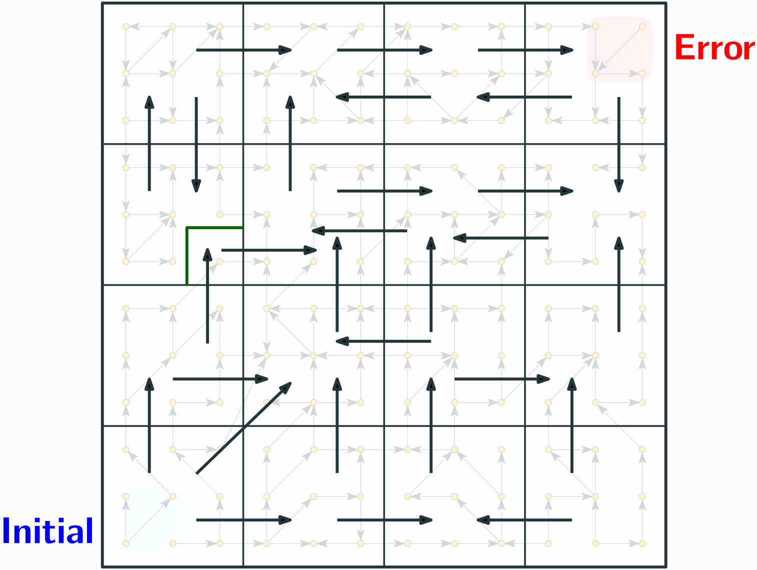
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



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



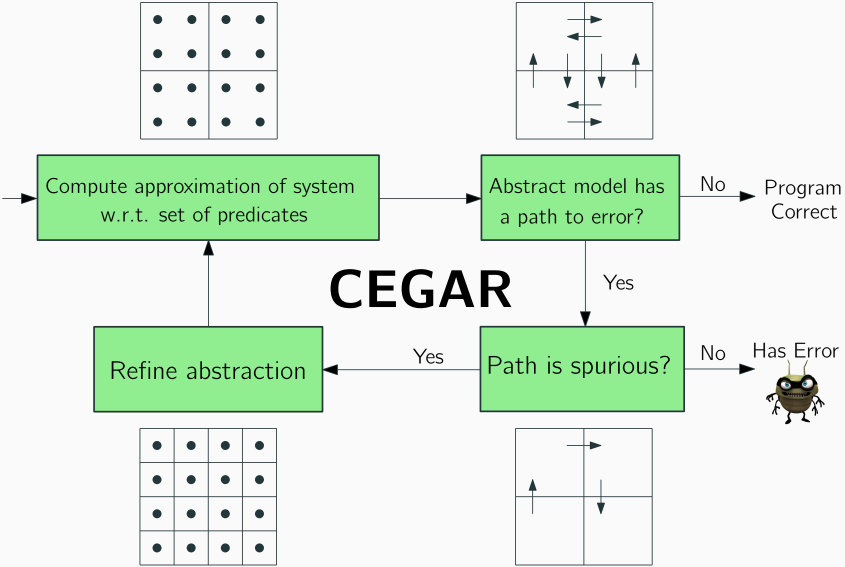
Over-Approximation: Make a transition from an abstract state if at least one corresponding concrete state has the transition.



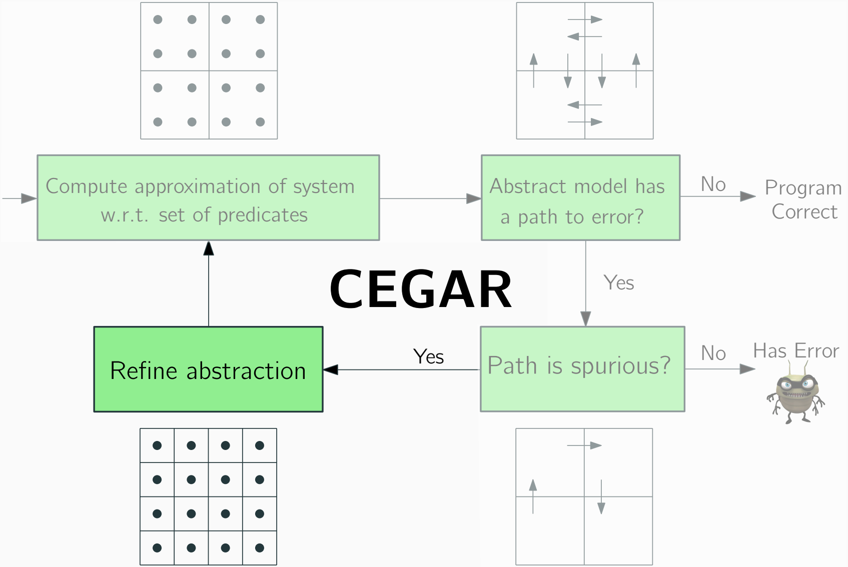
Continue searching in the new abstract state space.

Counter**E**xample-**G**uided **A**bstraction **R**efinement (CEGAR)

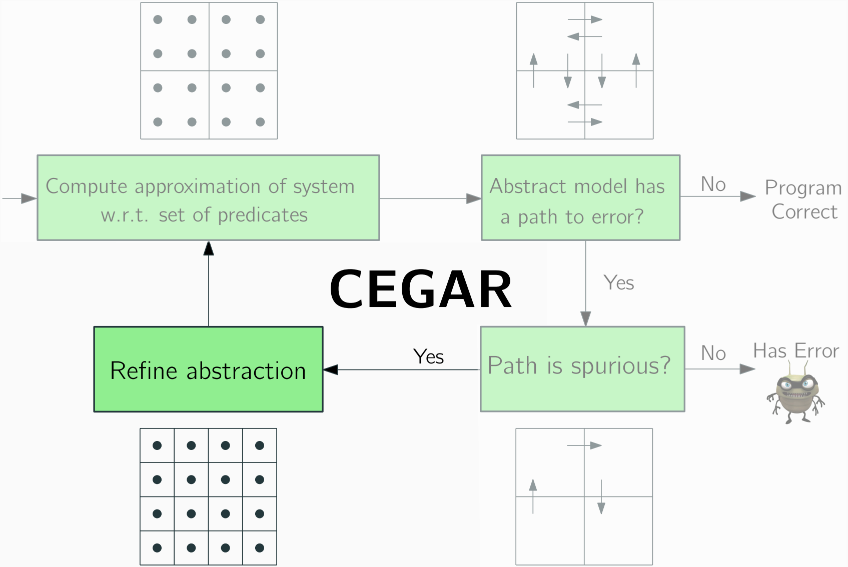
CounterExample-Guided Abstraction Refinement



CounterExample-Guided Abstraction Refinement

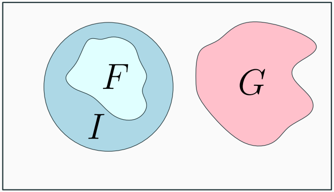


CounterExample-Guided Abstraction Refinement



Abstraction refinement: Craig interpolation

Craig Interpolation



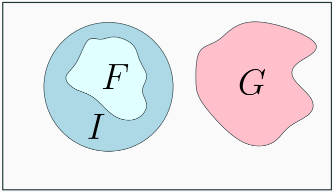
Set of States

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

Craig Interpolation



Set of States

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

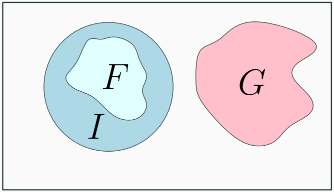
1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

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 Interpolation



Set of States

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

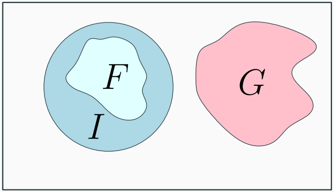
1. $F \to \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

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

common variables

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

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



Set of States

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

common variables

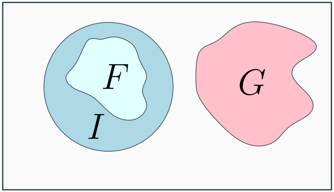
$$(A \wedge B \quad , \quad \neg B)$$

$$\Downarrow$$

$$I = B$$

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

Craig Interpolation

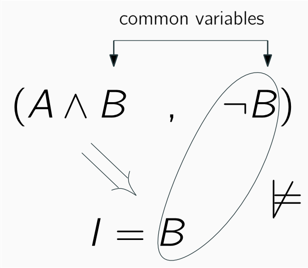


Set of States

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

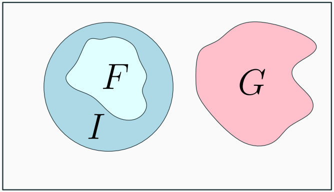
1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

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



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

Craig Interpolation



Set of States

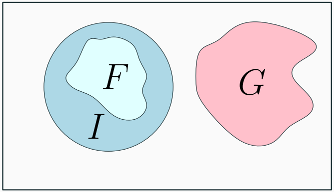
Craig interpolant for an **inconsistent** pair of formulae (F, G) is a formula I s.t.

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

- 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



Set of States

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

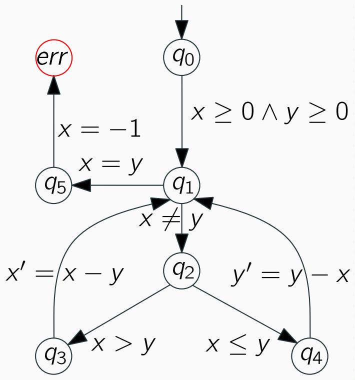
1. $F \rightarrow \mathrm{I}$,
2. $\mathrm{I} \wedge G$ is unsatisfiable,
3. I refers only to the common variables of F and G.

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

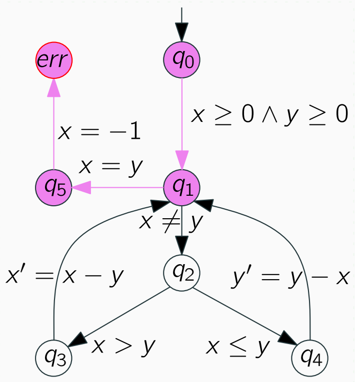
Let $\Gamma = \{F_1, F_2, \cdots, F_n\}$ be a set of formulae such that $\bigwedge \Gamma \equiv false$
An Inductive Interpolant Sequence for Γ is a set $\{I_0, I_1, \cdots, I_n\}$ such that:

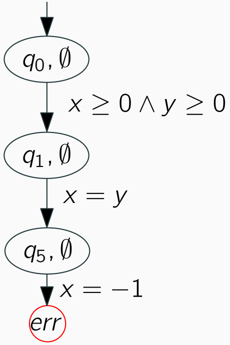
1. $I_0 =$ *true* and $I_n =$ *false*
2. For every $0 \leq \mathrm{j} < n$: $I_\mathrm{j} \wedge F_{\mathrm{j}+1} \rightarrow I_{\mathrm{j}+1}$
3. For every $0 < \mathrm{j} < n$: $\mathcal{L}(I_\mathrm{j}) \subseteq \mathcal{L}(F_1, \cdots, F_\mathrm{j}) \cap \mathcal{L}(F_{\mathrm{j}+1}, \cdots, F_n)$

CEGAR: Example

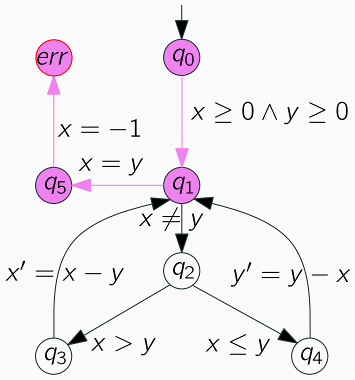


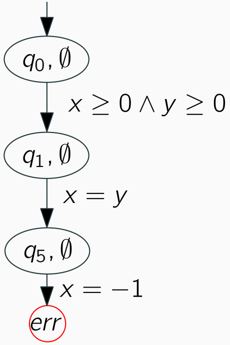
CEGAR: Example





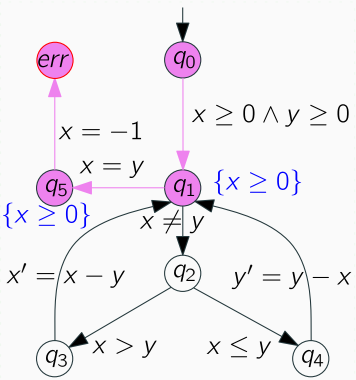
CEGAR: Example

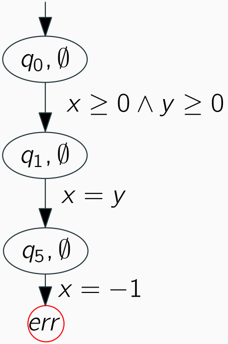




- $\Big(\underbrace{(x \geq 0 \wedge y \geq 0)}_{A_1} \wedge \underbrace{(x = y)}_{A_2} \wedge \underbrace{(x = -1)}_{A_3}\Big) = \mathit{false}$
- Interpolant: $\{x \geq 0, x \geq 0\}$

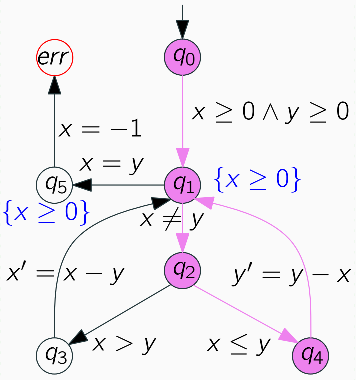
CEGAR: Example

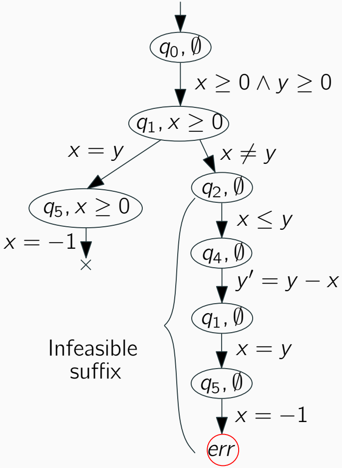




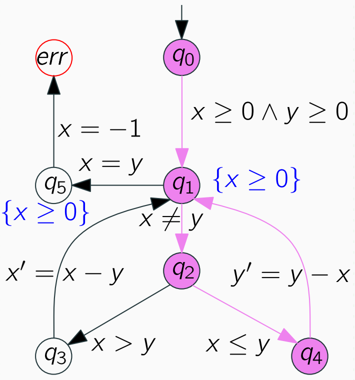
- $\Big(\underbrace{(x \geq 0 \wedge y \geq 0)}_{A_1} \wedge \underbrace{(x = y)}_{A_2} \wedge \underbrace{(x = -1)}_{A_3}\Big) = \mathit{false}$
- Interpolant: $\{x \geq 0, x \geq 0\}$

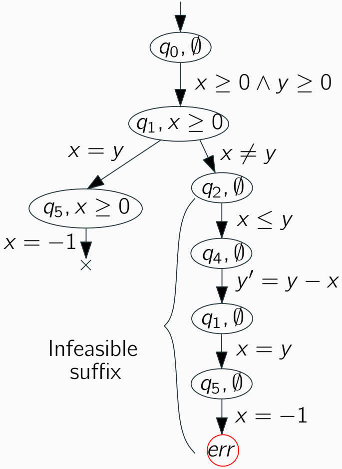
CEGAR: Example





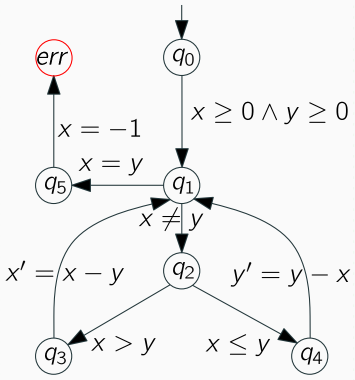
CEGAR: Example

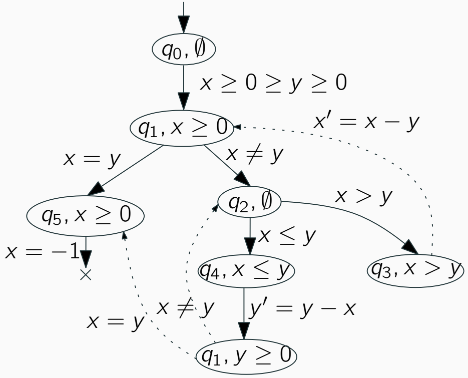




- $\left(\underbrace{(x \leq y)}_{A_1} \wedge \underbrace{(y_1 = y - x)}_{A_2} \wedge \underbrace{(x = y_1)}_{A_3} \wedge \underbrace{(x = -1)}_{A_4}\right) = \mathit{false}$
- Interpolant: $\{x \leq y, y_1 \geq 0, x \geq 0\}$

CEGAR: Example





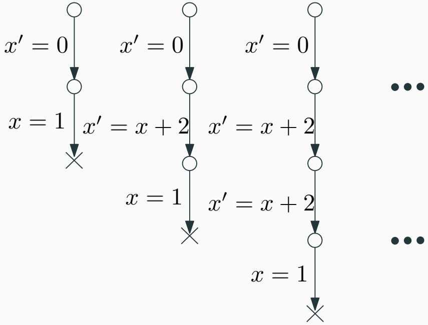
Abstract Reachability Graph:

Unfolding the program in the abstract space

Divergence

```
x = 0
while (⋆) {
   x = x + 2
}
assert (x ≠ 1)
```

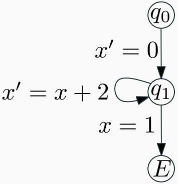
Spurious paths



- Classical predicate abstraction eliminates spurious counter-examples one by one
- Predicate abstraction divergence [Jhala & McMillan 2006]
- The acceleration technique solves this problem

Solution by Acceleration

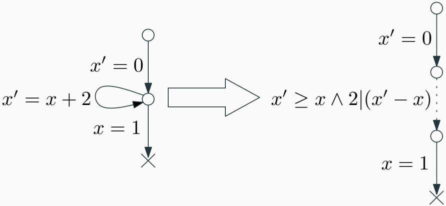
```
x = 0
while (⋆) {
   x = x + 2
}
assert (x ≠ 1)
```



- Compute transitive closure of loop relation $R^* = \bigvee_{i=0}^{\infty} R^i$
- $(x' = x + 2)^* \;=\; (x' = x) \vee (x' = x + 2) \vee (x' = x + 4) \vee \cdots$
  $\;=\; (x' \geq x) \wedge 2 | (x' - x)$

UNSAT

$x = 0 \wedge$
$x' \geq x \wedge$
$2 | (x' - x) \wedge x' = 1$



Acceleration

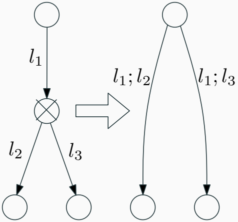
- Acceleration of Integer arithmetic are Presburger definable for
    1. Octagonal relation [Bozga et al. 2006]
        - Conjunctions of terms $x \pm y \leq c, c \in \mathbb{Z}$
    2. Finite Monoid Affine Relations [Finkel & Leroux 2002]
        - $T \equiv (x' = A \otimes \mathbf{x} + \mathbf{b}) \wedge \phi(x)$
        - where $A \in \mathbb{Z}^{\mathbb{N} \times \mathbb{N}}$, $b \in \mathbb{Z}$, ϕ is a Presburger formula, and $\{A, A^2, \cdots\}$ is finite

- Many programs do not fit into any of these categories
- Acceleration itself is not sufficient for proving some programs

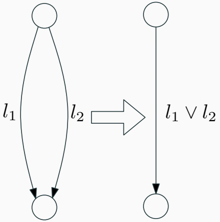
Static Acceleration

- Use acceleration to prevent divergence in predicate abstraction
- First try: use acceleration statically
- Introduce loop elimination rule in large block encoding

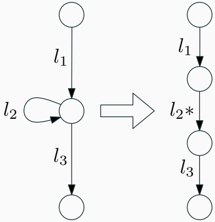
Sequence



Choice



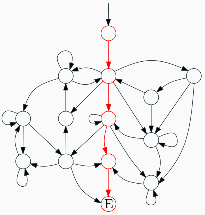
Loop Elimination



Static Acceleration

Drawback

- Large block encoding with static acceleration combines all labels together
- Potentially generates very big formulas
- There might be a short path to error
- We'd like to have on-demand acceleration

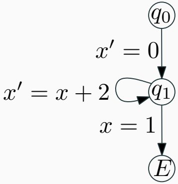


Drawback

- Loop relation may not be precisely accelerable
- How can we use approximate acceleration when precise is impossible?

Accelerating Counter-examples

```
x = 0
while (⋆) {
   x = x + 2
}
assert (x ≠ 1)
```



spurious counter-examples

$$q_0 \xrightarrow{x'=0} q_1 \xrightarrow{x=1} \mathrm{E}$$

$$q_0 \xrightarrow{x'=0} q_1 \xrightarrow{x'=x+2} q_1 \xrightarrow{x=1} \mathrm{E}$$

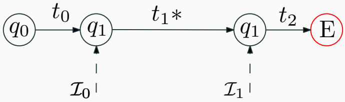
$$q_0 \xrightarrow{x'=0} q_1 \xrightarrow{x'=x+2} q_1 \xrightarrow{x'=x+2} q_1 \xrightarrow{x=1} \mathrm{E}$$

$$\vdots \qquad\qquad \vdots$$

- Generalize when a pattern is repeated for a number of times (delay)

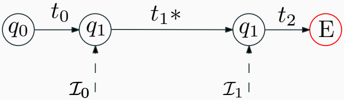
$$q_0 \xrightarrow{x'=0} q_1 \xrightarrow{(x'=x+2)*} q_1 \xrightarrow{x=1} \mathrm{E}$$

Accelerating Interpolants [Hojjat et al. 2012]



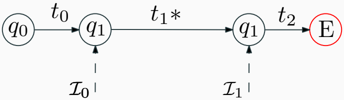
- $\langle \mathcal{I}_0, \mathcal{I}_1 \rangle$ are the interpolants for the sequence $\langle t_0, t_1*, t_2 \rangle$
- They are not necessarily inductive

Accelerating Interpolants [Hojjat et al. 2012]



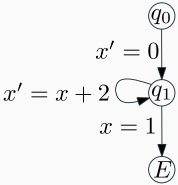
- $\langle \mathcal{I}_0, \mathcal{I}_1 \rangle$ are the interpolants for the sequence $\langle t_0, t_1*, t_2 \rangle$
- They are not necessarily inductive
- Consider interpolant images instead of original interpolants
    - $\langle \mathcal{I}_0, sp(\mathcal{I}_0, t_1*) \rangle$
    - $\langle wp(t_1*, \mathcal{I}_1), \mathcal{I}_1 \rangle$

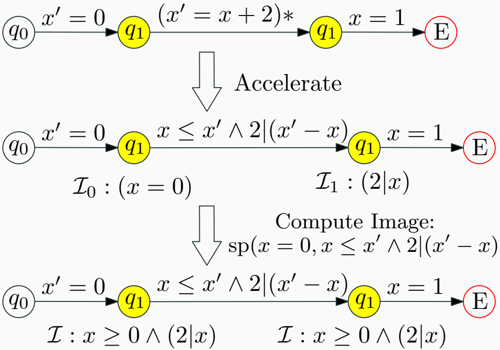
Accelerating Interpolants [Hojjat et al. 2012]



- $\langle \mathcal{I}_0, \mathcal{I}_1 \rangle$ are the interpolants for the sequence $\langle t_0, t_1*, t_2 \rangle$
- They are not necessarily inductive
- Consider interpolant images instead of original interpolants
  - $\langle \mathcal{I}_0, sp(\mathcal{I}_0, t_1*) \rangle$
  - $\langle wp(t_1*, \mathcal{I}_1), \mathcal{I}_1 \rangle$
- **Theorem.** Any of the above sequences are inductive interpolants

```
x = 0
while (⋆) {
  x = x + 2
}
assert (x ≠ 1)
```





Experiments

Model	Time [s]			
	Flata	Pred. Abs.	Static Acc.	Dynamic Acc.
Examples from [Jhala & McMillan 2006]				
anubhav (C)	0.6	1.5	1.8	1.5
copy1 (E)	1.7	8.1	1.2	3.5
cousot (C)	0.5	-	-	4.3
loop1 (C)	0.4	2.1	0.9	2.1
loop (C)	0.4	0.3	0.9	0.3
scan (E)	2.4	-	1.0	2.9
string_concat1 (E)	4.4	-	3.2	5.0
string_concat (E)	4.1	-	2.5	4.2
string_copy (E)	3.7	-	1.5	3.6
substring1 (E)	0.3	1.6	23.9	1.5
substring (E)	1.8	0.6	1.6	0.6

Experiments

Model	Time [s]			
	Flata	Pred. Abs.	Static Acc.	Dynamic Acc.
Examples from David Monniaux				
boustrophedon (C)	-	-	-	12.2
gopan (C)	0.5	-	-	6.7
halbwachs (C)	-	-	1.6	8.2
rate_limiter (C)	-	7.2	2.7	7.1
Examples from Array Programs				
rotation_vc.1 (C)	0.7	2.0	6.3	1.9
rotation_vc.2 (C)	1.3	2.1	202.2	2.1
rotation_vc.3 (C)	1.2	0.3	181.5	0.3
rotation_vc.1 (E)	1.1	1.4	14.9	1.4
split_vc.1 (C)	4.2	2.7	-	2.7
split_vc.2 (C)	2.8	2.1	-	2.1
split_vc.3 (C)	2.9	0.5	-	0.5