Automated Program Verification and Testing 15414/15614 Fall 2016 Lecture 26: Counterexamples & Abstraction Refinement

Matt Fredrikson
mfredrik@cs.cmu.edu

December 6, 2016

Abstraction (Review)

Key Idea: Approximate system so that a given property is preserved

Abstraction (Review)

Key Idea: Approximate system so that a given property is preserved

More precisely, given KS M and ϕ, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

Abstraction (Review)

Key Idea: Approximate system so that a given property is preserved

More precisely, given KS M and ϕ, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We'll see how to build a **conservative overapproximation** of M

Abstraction (Review)

Key Idea: Approximate system so that a given property is preserved

More precisely, given KS M and ϕ, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We'll see how to build a **conservative overapproximation** of M

- Every trace of M is also a trace of $\hat{M}$

Abstraction (Review)

Key Idea: Approximate system so that a given property is preserved

More precisely, given KS M and ϕ, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We'll see how to build a **conservative overapproximation** of M

- Every trace of M is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in M

Abstraction (Review)

Key Idea: Approximate system so that a given property is preserved

More precisely, given KS M and ϕ, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We'll see how to build a **conservative overapproximation** of M

- Every trace of M is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in M

This preserves safety properties: if $\hat{M}$ verifies, so will M

Abstraction (Review)

Key Idea: Approximate system so that a given property is preserved

More precisely, given KS M and ϕ, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We'll see how to build a **conservative overapproximation** of M

- Every trace of M is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in M

This preserves safety properties: if $\hat{M}$ verifies, so will M

But it might introduce **spurious counterexamples**

Predicate Abstraction (Review)

How do we know which abstraction to use?

Predicate Abstraction (Review)

How do we know which abstraction to use?

Idea: Only track **predicates** on program's data state

- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

Predicate Abstraction (Review)

How do we know which abstraction to use?

Idea: Only track **predicates** on program's data state

- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We're given: set of predicates $E = \{\phi_1, \ldots, \phi_n\}$

Define **abstraction function** $\alpha : \text{Env} \mapsto \{0, 1\}^n$:

$$\alpha((\ell, \sigma)) = (\ell, (\phi_1(\sigma), \ldots, \phi_n(\sigma)))$$

Predicate Abstraction (Review)

How do we know which abstraction to use?

Idea: Only track **predicates** on program's data state

- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We're given: set of predicates $E = \{\phi_1, \ldots, \phi_n\}$

Define **abstraction function** $\alpha : \text{Env} \mapsto \{0, 1\}^n$:

$$\alpha((\ell, \sigma)) = (\ell, (\phi_1(\sigma), \ldots, \phi_n(\sigma)))$$

Intuitively: α ranges over conjunctions of $\phi_i, \neg\phi_i$

Predicate Abstraction (Review)

How do we know which abstraction to use?

Idea: Only track **predicates** on program's data state

- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We're given: set of predicates $E = \{\phi_1, \ldots, \phi_n\}$

Define **abstraction function** $\alpha : \mathsf{Env} \mapsto \{0,1\}^n$:

$$\alpha((\ell, \sigma)) = (\ell, (\phi_1(\sigma), \ldots, \phi_n(\sigma)))$$

Intuitively: α ranges over conjunctions of $\phi_i, \neg\phi_i$

The states in our abstraction will be: $S = \mathsf{Loc} \times \{0,1\}^m$

Existential Abstraction (Review)

Existential Abstraction (Review)

Important: We want an over-approximation that gives us:

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

Existential Abstraction (Review)

Important: We want an over-approximation that gives us:

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We'll define an **existential abstraction**:

$$\begin{aligned}(\hat{s}_1, \hat{s}_2) \in \hat{R} \Leftrightarrow \quad & \exists s_1, s_2 . R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2 \\ \hat{s} \in \hat{I} \Leftrightarrow \quad & \exists s . s \in I \wedge h(s) = \hat{s}\end{aligned}$$

A transition is in the abstraction $\hat{M}$ if and only if:

1. There **exist** corresponding states (s_1, s_2) in M,
2. where s_1, s_2 are the endpoints of a transition in M

Existential Abstraction (Review)

Important: We want an over-approximation that gives us:

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We'll define an **existential abstraction**:

$$\begin{aligned}(\hat{s}_1, \hat{s}_2) \in \hat{R} \Leftrightarrow & \quad \exists s_1, s_2 . R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2 \\ \hat{s} \in \hat{I} \Leftrightarrow & \quad \exists s . s \in I \wedge h(s) = \hat{s}\end{aligned}$$

A transition is in the abstraction $\hat{M}$ if and only if:

1. There **exist** corresponding states (s_1, s_2) in M,
2. where s_1, s_2 are the endpoints of a transition in M

Why is this conservative?

Intuition: Existential Abstraction

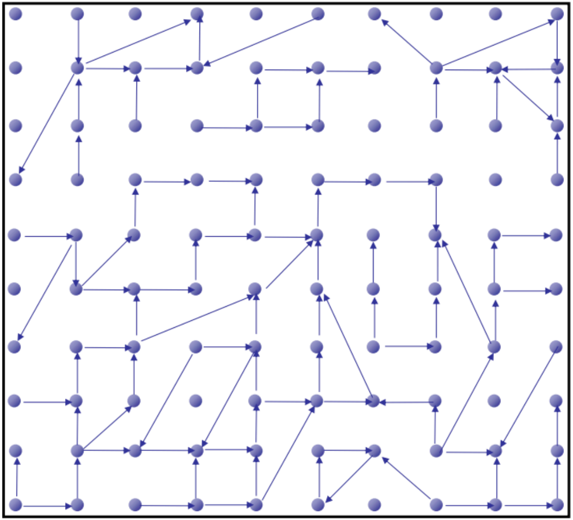


Image Credit: Tom Henzinger, Ranjit Jhala, Rupak Majumdar

Intuition: Existential Abstraction

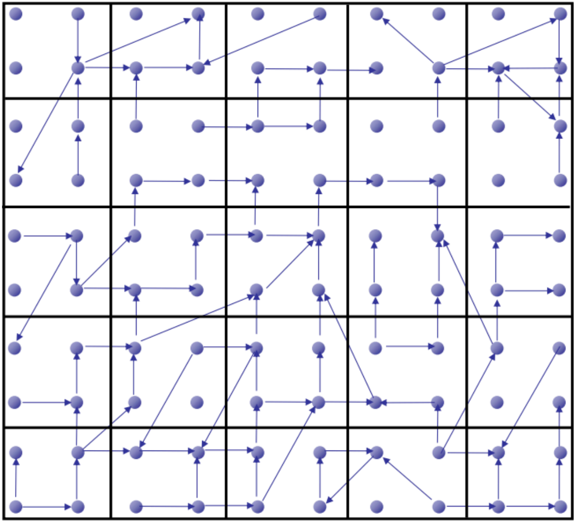


Image Credit: Tom Henzinger, Ranjit Jhala, Rupak Majumdar

Intuition: Existential Abstraction

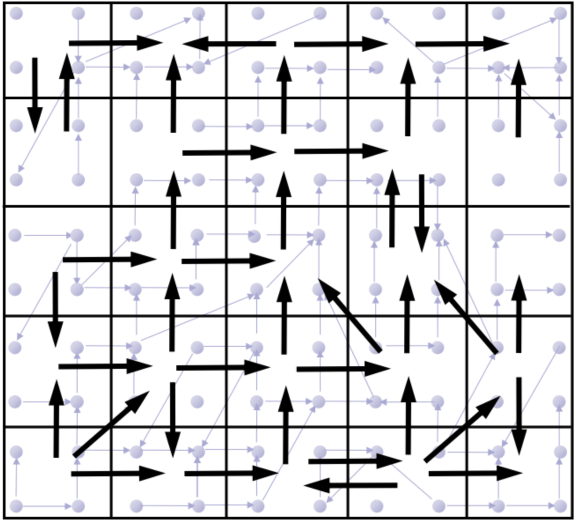


Image Credit: Tom Henzinger, Ranjit Jhala, Rupak Majumdar

Computing Program Approximations

The key issue: how do we compute **transitions**

Computing Program Approximations

The key issue: how do we compute **transitions**

Recall our construction of KS from program graphs:

$$\frac{(\ell_1, b, \ell_2) \in T \qquad \langle b, \sigma_1 \rangle \Downarrow_b \textit{true} \qquad \langle C(\ell_1), \sigma_1 \rangle \Downarrow \sigma_2}{([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R}$$

Computing Program Approximations

The key issue: how do we compute **transitions**

Recall our construction of KS from program graphs:

$$\frac{(\ell_1, b, \ell_2) \in T \qquad \langle b, \sigma_1 \rangle \Downarrow_b \textit{true} \qquad \langle C(\ell_1), \sigma_1 \rangle \Downarrow \sigma_2}{([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R}$$

We don't have concrete states σ to work with anymore

Computing Program Approximations

The key issue: how do we compute **transitions**

Recall our construction of KS from program graphs:

$$\frac{(\ell_1, b, \ell_2) \in T \qquad \langle b, \sigma_1 \rangle \Downarrow_b \textit{true} \qquad \langle C(\ell_1), \sigma_1 \rangle \Downarrow \sigma_2}{([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R}$$

We don't have concrete states σ to work with anymore

Just predicates.

Computing Program Approximations

The key issue: how do we compute **transitions**

Recall our construction of KS from program graphs:

$$\frac{(\ell_1, b, \ell_2) \in T \qquad \langle b, \sigma_1 \rangle \Downarrow_b \textit{true} \qquad \langle C(\ell_1), \sigma_1 \rangle \Downarrow \sigma_2}{([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R}$$

We don't have concrete states σ to work with anymore

Just predicates. **Idea**: Use predicate transformers

Strengthening Predicates (Review)

Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\mathsf{Pred}(\phi, E)$:

- The **weakest** DNF over E,
- that is at least as strong as ϕ,
- where each clause has n literals

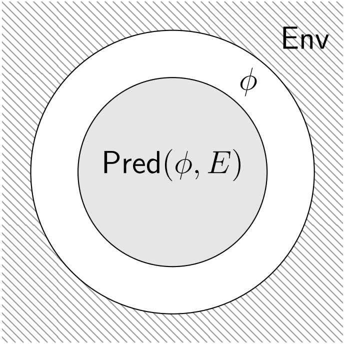
Notice: $\mathsf{Pred}(\phi, E) \Rightarrow \phi$

Compute this by querying SMT solver

- What's the complexity of this?
- $O(2^n)$
- Need to query each:

$$p_1 \wedge \cdots \wedge p_n \Rightarrow \phi$$

where p_i is ϕ_i or $\neg\phi_i$



Computing Transitions via Strengthening

For assignments $x := e$:

Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $\mathsf{wp}(x := e, \phi)$, $\mathsf{wp}(x := e, \neg\phi)$

Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $\mathsf{wp}(x := e, \phi)$, $\mathsf{wp}(x := e, \neg\phi)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$

Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $\mathsf{wp}(x := e, \phi)$, $\mathsf{wp}(x := e, \neg\phi)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If state implies $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, draw an edge to ϕ

Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $\mathsf{wp}(x := e, \phi)$, $\mathsf{wp}(x := e, \neg\phi)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If state implies $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, draw an edge to ϕ
4. If state implies $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$, draw an edge to $\neg\phi$

Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $\mathsf{wp}(x := e, \phi)$, $\mathsf{wp}(x := e, \neg\phi)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If state implies $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, draw an edge to ϕ
4. If state implies $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$, draw an edge to $\neg\phi$
5. If neither implication holds, draw an edge to both

Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $\mathsf{wp}(x := e, \phi)$, $\mathsf{wp}(x := e, \neg\phi)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If state implies $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, draw an edge to ϕ
4. If state implies $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$, draw an edge to $\neg\phi$
5. If neither implication holds, draw an edge to both

$$\begin{array}{ll} \ell_0: & x := x + 1 \\ \ell_1: & \textbf{skip} \end{array}$$

$$E = \{\underbrace{x = y}_{p_0}\}$$

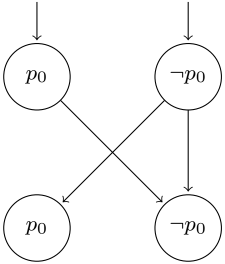
Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $\mathsf{wp}(x := e, \phi)$, $\mathsf{wp}(x := e, \neg\phi)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If state implies $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, draw an edge to ϕ
4. If state implies $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$, draw an edge to $\neg\phi$
5. If neither implication holds, draw an edge to both

$$\begin{aligned} \ell_0 &: \quad x := x + 1 \\ \ell_1 &: \quad \textbf{skip} \end{aligned}$$

$$E = \{\underbrace{x = y}_{p_0}\}$$



Computing Transitions via Strengthening

For assumptions **assume** ϕ:

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\texttt{Pred}(\neg\phi, E)$

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\mathsf{Pred}(\neg\phi, E)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\mathsf{Pred}(\neg\phi, E)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If next state implies $\neg\mathsf{Pred}(\neg\phi, E)$, draw an edge to it

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\mathsf{Pred}(\neg\phi, E)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If next state implies $\neg\mathsf{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\mathsf{Pred}(\phi, E)$, draw an edge to it

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\mathsf{Pred}(\neg\phi, E)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If next state implies $\neg\mathsf{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\mathsf{Pred}(\phi, E)$, draw an edge to it

ℓ_0 : **assume** $x = 1$
ℓ_1 : **skip**

$$E = \{\underbrace{x = y}_{p_0}\}$$

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\mathsf{Pred}(\neg\phi, E)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If next state implies $\neg\mathsf{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\mathsf{Pred}(\phi, E)$, draw an edge to it

ℓ_0 : **assume** $x = 1$
ℓ_1 : **skip**

$$E = \{\underbrace{x = y}_{p_0}\}$$

$\mathsf{Pred}(\neg(x = 1), \{x = y\}) =$

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\mathsf{Pred}(\neg\phi, E)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If next state implies $\neg\mathsf{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\mathsf{Pred}(\phi, E)$, draw an edge to it

ℓ_0 : **assume** $x = 1$
ℓ_1 : **skip**

$$E = \{\underbrace{x = y}_{p_0}\}$$

$\mathsf{Pred}(\neg(x = 1), \{x = y\}) =$ *false*

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\text{Pred}(\neg\phi, E)$
2. Strengthen them: $\text{Pred}(\text{wp}(x := e, \phi), E)$, $\text{Pred}(\neg\text{wp}(x := e, \phi), E)$
3. If next state implies $\neg\text{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\text{Pred}(\phi, E)$, draw an edge to it

ℓ_0 : **assume** $x = 1$
ℓ_1 : **skip**

$$E = \{\underbrace{x = y}_{p_0}\}$$

$\text{Pred}(\neg(x = 1), \{x = y\})$ = *false*

$\text{Pred}(x = 1, \{x = y\})$ =

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

1. *Weaken* ϕ: $\neg\mathsf{Pred}(\neg\phi, E)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If next state implies $\neg\mathsf{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\mathsf{Pred}(\phi, E)$, draw an edge to it

ℓ_0 : **assume** $x = 1$
ℓ_1 : **skip**

$$E = \{\underbrace{x = y}_{p_0}\}$$

$\mathsf{Pred}(\neg(x = 1), \{x = y\})$ = *false*

$\mathsf{Pred}(x = 1, \{x = y\})$ = *false*

Computing Transitions via Strengthening

For assumptions **assume** ϕ:

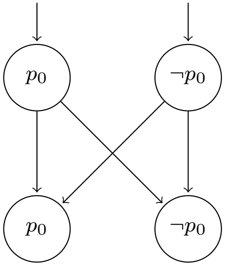
1. *Weaken* ϕ: $\neg\mathsf{Pred}(\neg\phi, E)$
2. Strengthen them: $\mathsf{Pred}(\mathsf{wp}(x := e, \phi), E)$, $\mathsf{Pred}(\neg\mathsf{wp}(x := e, \phi), E)$
3. If next state implies $\neg\mathsf{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\mathsf{Pred}(\phi, E)$, draw an edge to it

ℓ_0 : **assume** $x = 1$
ℓ_1 : **skip**

$$E = \{\underbrace{x = y}_{p_0}\}$$

$\mathsf{Pred}(\neg(x = 1), \{x = y\})$ = *false*

$\mathsf{Pred}(x = 1, \{x = y\})$ = *false*



Example: Predicate Abstraction

$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$

Suppose we check:

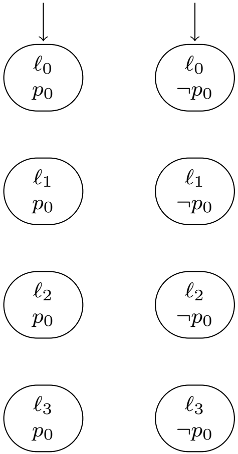
$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$

Example: Predicate Abstraction

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$



Suppose we check:

$$\mathbf{G}\ (\neg\ell_0 \rightarrow 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$

Example: Predicate Abstraction

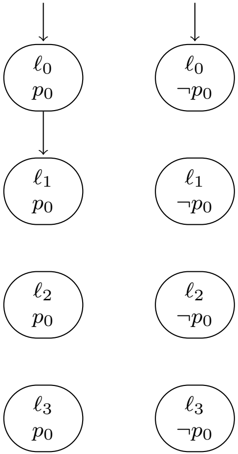
$$\begin{aligned}
\ell_0 &: \quad i := 1; \\
\ell_1 &: \quad \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2 &: \quad\quad i := i - 1; \\
\ell_3 &: \quad\quad x := x + 1; \\
&\quad\ \}
\end{aligned}$$

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$



Example: Predicate Abstraction

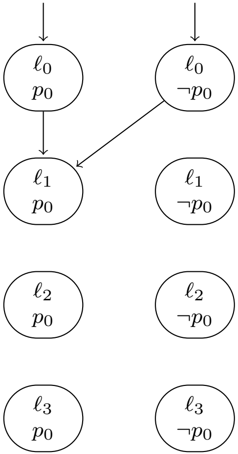
$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \to 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$



Example: Predicate Abstraction

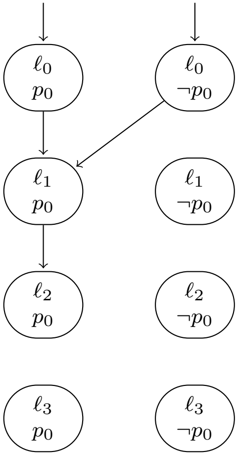
$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \to 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$



Example: Predicate Abstraction

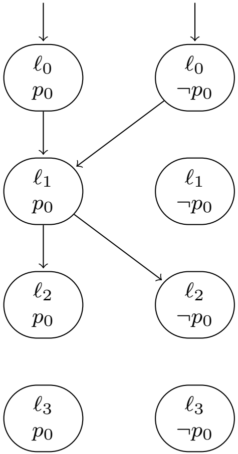
$$
\begin{aligned}
\ell_0 : &\quad i := 1; \\
\ell_1 : &\quad \textbf{while}(0 \leq x < 1)\ \{ \\
\ell_2 : &\quad\ \ i := i - 1; \\
\ell_3 : &\quad\ \ x := x + 1; \\
&\quad \}
\end{aligned}
$$

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \leq i)$$

Using:

$$E = \{\underbrace{0 \leq i}_{p_0}\}$$



Example: Predicate Abstraction

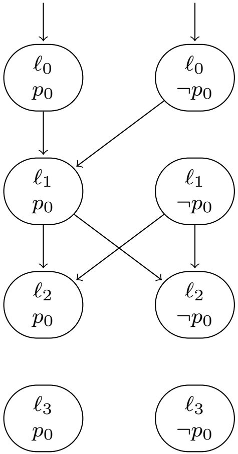
$\ell_0 :$ $i := 1;$
$\ell_1 :$ **while**$(0 \leq x < 1)$ $\{$
$\ell_2 :$ $i := i - 1;$
$\ell_3 :$ $x := x + 1;$
$\}$

Suppose we check:

$$\mathbf{G}\ (\neg\ell_0 \rightarrow 0 \leq i)$$

Using:

$$E = \{\underbrace{0 \leq i}_{p_0}\}$$



Example: Predicate Abstraction

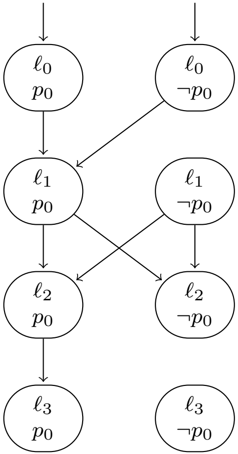
ℓ_0 : $i := 1;$
ℓ_1 : **while**$(0 \leq x < 1)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \leq i)$$

Using:

$$E = \{\underbrace{0 \leq i}_{p_0}\}$$



Example: Predicate Abstraction

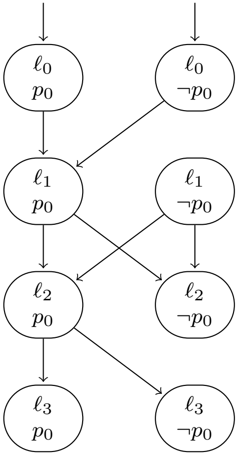
$$\begin{aligned}
\ell_0 :&\quad i := 1; \\
\ell_1 :&\quad \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2 :&\quad\ \ i := i - 1; \\
\ell_3 :&\quad\ \ x := x + 1; \\
&\quad \}
\end{aligned}$$

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \to 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$



Example: Predicate Abstraction

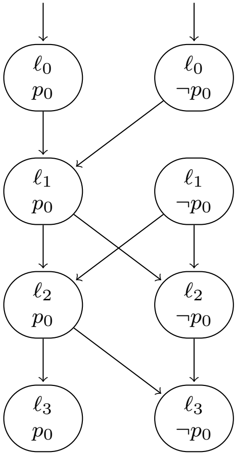
$$
\begin{aligned}
\ell_0 :&\quad i := 1; \\
\ell_1 :&\quad \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2 :&\quad\ \ i := i - 1; \\
\ell_3 :&\quad\ \ x := x + 1; \\
&\quad \}
\end{aligned}
$$

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \to 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$



Example: Predicate Abstraction

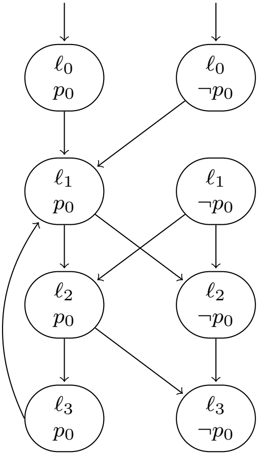
$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$



Example: Predicate Abstraction

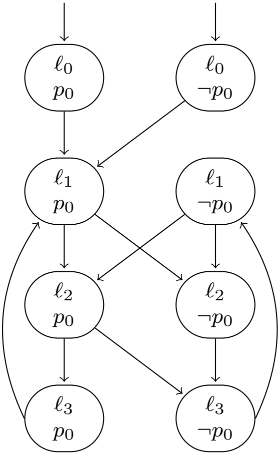
$$
\begin{aligned}
\ell_0 :&\quad i := 1; \\
\ell_1 :&\quad \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2 :&\quad\ \ i := i - 1; \\
\ell_3 :&\quad\ \ x := x + 1; \\
&\quad \}
\end{aligned}
$$

Suppose we check:

$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$

Using:

$$E = \{\underbrace{0 \le i}_{p_0}\}$$

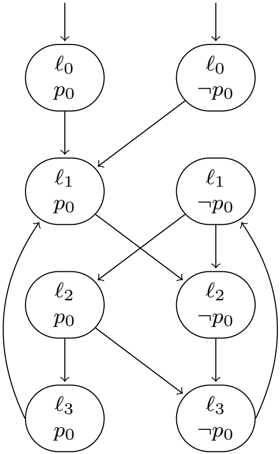


Example: Predicate Abstraction

$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$

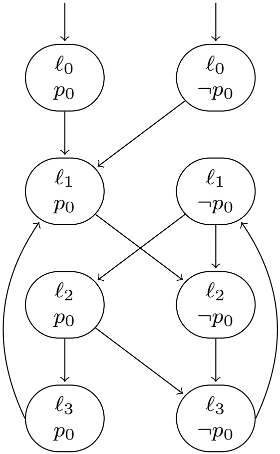
Does the property hold?

$$\textbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$



Example: Predicate Abstraction

$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$



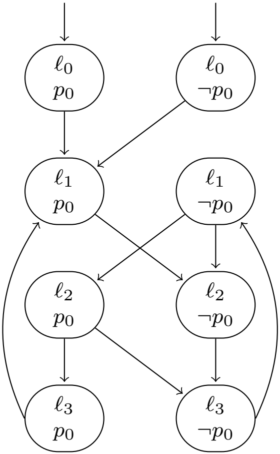
Does the property hold?

$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \leq i)$$

No.

Example: Predicate Abstraction

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$



Does the property hold?

$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$

No. What's a counterexample?

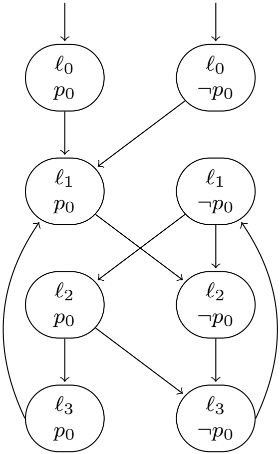
Example: Predicate Abstraction

$$\begin{array}{ll} \ell_0 : & i := 1; \\ \ell_1 : & \textbf{while}(0 \le x < 1)\ \{ \\ \ell_2 : & \quad i := i - 1; \\ \ell_3 : & \quad x := x + 1; \\ & \} \end{array}$$

Does the property hold?

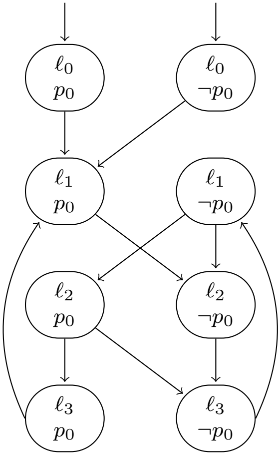
$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$

No. What's a counterexample?



Example: Predicate Abstraction

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$



Does the property hold?

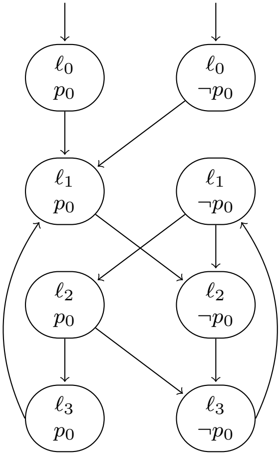
$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$

No. What's a counterexample?

$$(\ell_0, p_0)$$

Example: Predicate Abstraction

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$



Does the property hold?

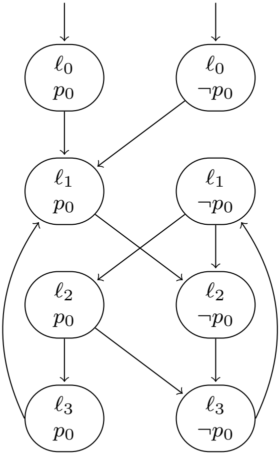
$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \le i)$$

No. What's a counterexample?

$$(\ell_0, p_0)$$
$$(\ell_1, p_0)$$

Example: Predicate Abstraction

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$



Does the property hold?

$$\mathbf{G}\ (\neg \ell_0 \rightarrow 0 \leq i)$$

No. What's a counterexample?

$$
\begin{array}{l}
(\ell_0, p_0) \\
(\ell_1, p_0) \\
(\ell_2, \neg p_0)
\end{array}
$$

Spurious Counterexamples

$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$

Consider the KS path:

$$\begin{array}{l} (\ell_0, p_0) \\ (\ell_1, p_0) \\ (\ell_2, \neg p_0) \end{array}$$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Spurious Counterexamples

$$
\begin{array}{ll}
\ell_0 : & i := 1; \\
\ell_1 : & \textbf{while}(0 \leq x < 1) \; \{ \\
\ell_2 : & \quad i := i - 1; \\
\ell_3 : & \quad x := x + 1; \\
 & \}
\end{array}
$$

Consider the KS path:

$$
\begin{array}{l}
(\ell_0, p_0) \\
(\ell_1, p_0) \\
(\ell_2, \neg p_0)
\end{array}
$$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Consider the corresponding program path

Spurious Counterexamples

ℓ_0 : $i := 1;$
ℓ_1 : **while**$(0 \leq x < 1)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}

ℓ_0 : $i := 1;$

Consider the KS path:

(ℓ_0, p_0)
(ℓ_1, p_0)
$(\ell_2, \neg p_0)$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Consider the corresponding program path

Spurious Counterexamples

$\ell_0 :\quad i := 1;$
$\ell_1 :\quad \textbf{while}(0 \leq x < 1)\ \{$
$\ell_2 :\quad\quad i := i - 1;$
$\ell_3 :\quad\quad x := x + 1;$
$\quad\quad \}$

$\ell_0 :\quad i := 1;$
$\ell_1 :\quad \textbf{assume}(0 \leq x < 1)$

Consider the KS path:

$$(\ell_0, p_0)$$
$$(\ell_1, p_0)$$
$$(\ell_2, \neg p_0)$$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Consider the corresponding program path

Spurious Counterexamples

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$

$$
\begin{array}{ll}
 & \{0 \leq i\} \\
\ell_0: & i := 1; \\
\ell_1: & \textbf{assume}(0 \leq x < 1)
\end{array}
$$

Consider the KS path:

$$
\begin{array}{l}
(\ell_0, p_0) \\
(\ell_1, p_0) \\
(\ell_2, \neg p_0)
\end{array}
$$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Consider the corresponding program path

Spurious Counterexamples

$$
\begin{array}{ll}
\ell_0 : & i := 1; \\
\ell_1 : & \textbf{while}(0 \leq x < 1)\ \{ \\
\ell_2 : & \quad i := i - 1; \\
\ell_3 : & \quad x := x + 1; \\
 & \}
\end{array}
$$

$$
\begin{array}{ll}
 & \{0 \leq i\} \\
\ell_0 : & i := 1; \\
\ell_1 : & \textbf{assume}(0 \leq x < 1) \\
 & \{i < 0\}
\end{array}
$$

Consider the KS path:

$$
\begin{array}{l}
(\ell_0, p_0) \\
(\ell_1, p_0) \\
(\ell_2, \neg p_0)
\end{array}
$$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Consider the corresponding program path

Spurious Counterexamples

$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$

$$\begin{array}{ll} & \{0 \leq i\} \\ \ell_0: & i := 1; \\ \ell_1: & \textbf{assume}(0 \leq x < 1) \\ & \{i < 0\} \end{array}$$

Consider the KS path:

$$(\ell_0, p_0)$$
$$(\ell_1, p_0)$$
$$(\ell_2, \neg p_0)$$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Is this a valid Hoare triple?

Consider the corresponding program path

Spurious Counterexamples

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$

$$
\begin{array}{ll}
 & \{0 \leq i\} \\
\ell_0: & i := 1; \\
\ell_1: & \textbf{assume}(0 \leq x < 1) \\
 & \{i < 0\}
\end{array}
$$

Consider the KS path:

$$(\ell_0, p_0)$$
$$(\ell_1, p_0)$$
$$(\ell_2, \neg p_0)$$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Is this a valid Hoare triple?

1. $\{0 \leq i\}\ i := 1\ \{0 \leq i\}$

Consider the corresponding program path

Spurious Counterexamples

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$

$$
\begin{array}{ll}
 & \{0 \le i\} \\
\ell_0: & i := 1; \\
\ell_1: & \textbf{assume}(0 \le x < 1) \\
 & \{i < 0\}
\end{array}
$$

Consider the KS path:

$$
\begin{array}{c}
(\ell_0, p_0) \\
(\ell_1, p_0) \\
(\ell_2, \neg p_0)
\end{array}
$$

(recall that $p_0 \Leftrightarrow 0 \le i$)

Is this a valid Hoare triple?

1. $\{0 \le i\}\ i := 1\ \{0 \le i\}$ **Yes**

Consider the corresponding program path

Spurious Counterexamples

$$
\begin{array}{ll}
\ell_0 : & i := 1; \\
\ell_1 : & \textbf{while}(0 \leq x < 1) \; \{ \\
\ell_2 : & \quad i := i - 1; \\
\ell_3 : & \quad x := x + 1; \\
 & \}
\end{array}
$$

$$
\begin{array}{ll}
 & \{0 \leq i\} \\
\ell_0 : & i := 1; \\
\ell_1 : & \textbf{assume}(0 \leq x < 1) \\
 & \{i < 0\}
\end{array}
$$

Consider the KS path:

$$(\ell_0, p_0)$$
$$(\ell_1, p_0)$$
$$(\ell_2, \neg p_0)$$

(recall that $p_0 \Leftrightarrow 0 \leq i$)

Is this a valid Hoare triple?

1. $\{0 \leq i\}\ i := 1\ \{0 \leq i\}$ **Yes**
2. $\{0 \leq i\}$ **assume**$(0 \leq x < 1)\ \{0 > i\}$

Consider the corresponding program path

Spurious Counterexamples

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \le x < 1) \, \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$

$$
\begin{array}{ll}
 & \{0 \le i\} \\
\ell_0: & i := 1; \\
\ell_1: & \textbf{assume}(0 \le x < 1) \\
 & \{i < 0\}
\end{array}
$$

Consider the KS path:

$$
\begin{array}{c}
(\ell_0, p_0) \\
(\ell_1, p_0) \\
(\ell_2, \neg p_0)
\end{array}
$$

(recall that $p_0 \Leftrightarrow 0 \le i$)

Is this a valid Hoare triple?

1. $\{0 \le i\}\ i := 1\ \{0 \le i\}$ **Yes**
2. $\{0 \le i\}$ **assume**$(0 \le x < 1)\ \{0 > i\}$ **No**

Consider the corresponding program path

Spurious Counterexamples

$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$

$$\begin{array}{ll} & \{0 \le i\} \\ \ell_0: & i := 1; \\ \ell_1: & \textbf{assume}(0 \le x < 1) \\ & \{i < 0\} \end{array}$$

Consider the KS path:

$$(\ell_0, p_0)$$
$$(\ell_1, p_0)$$
$$(\ell_2, \neg p_0)$$

(recall that $p_0 \Leftrightarrow 0 \le i$)

Is this a valid Hoare triple?

1. $\{0 \le i\}\ i := 1\ \{0 \le i\}$ **Yes**
2. $\{0 \le i\}$ **assume**$(0 \le x < 1)\ \{0 > i\}$ **No**

Consider the corresponding program path

This is how we know that the counterexample is spurious

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same "cause"

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)

- E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)

- E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Main technique behind all modern software model checking

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)

- E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Main technique behind all modern software model checking

1. Start with a simple, automatic abstraction

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)

- E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Main technique behind all modern software model checking

1. Start with a simple, automatic abstraction
2. Search for counterexamples

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)

- E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Main technique behind all modern software model checking

1. Start with a simple, automatic abstraction
2. Search for counterexamples
3. Refine spurious counterexamples, building model on-demand

Abstraction Refinement

We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)

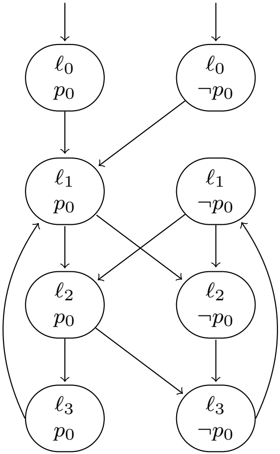
- E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Main technique behind all modern software model checking

1. Start with a simple, automatic abstraction
2. Search for counterexamples
3. Refine spurious counterexamples, building model on-demand
4. Continue until real counterexample, or property holds

Cause and Refinement

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
& \}
\end{array}
$$



What caused this?

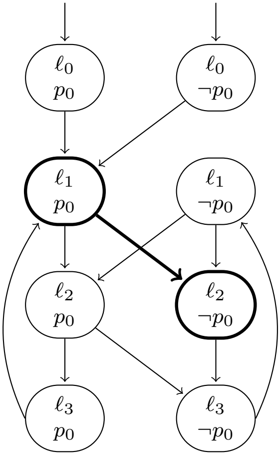
$$(\ell_0, p_0)\ (\ell_1, p_0)\ (\ell_2, \neg p_0)$$

Cause and Refinement

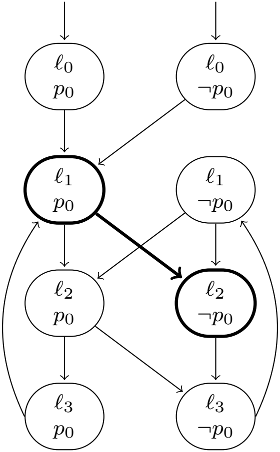
$$
\begin{aligned}
\ell_0 : &\quad i := 1; \\
\ell_1 : &\quad \textbf{while}(0 \le x < 1)\ \{ \\
\ell_2 : &\quad\quad i := i - 1; \\
\ell_3 : &\quad\quad x := x + 1; \\
&\quad \}
\end{aligned}
$$

What caused this?

$$(\ell_0, p_0)\ (\ell_1, p_0)\ (\ell_2, \neg p_0)$$



$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$



What caused this?

$$(\ell_0, p_0)\ (\ell_1, p_0)\ (\ell_2, \neg p_0)$$

We had $\neg$Pred$(0 \leq x < 1, \{p_0\})$ = *true*

Cause and Refinement

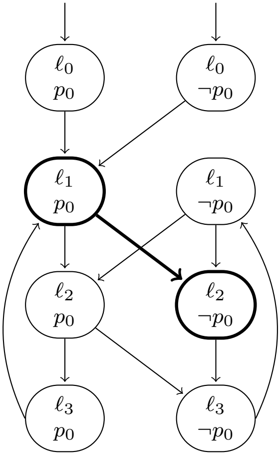
$$
\begin{aligned}
\ell_0 :&\quad i := 1;\\
\ell_1 :&\quad \textbf{while}(0 \le x < 1)\ \{\\
\ell_2 :&\quad\ \ i := i - 1;\\
\ell_3 :&\quad\ \ x := x + 1;\\
&\quad \}
\end{aligned}
$$

What caused this?

$$(\ell_0, p_0)\ (\ell_1, p_0)\ (\ell_2, \neg p_0)$$

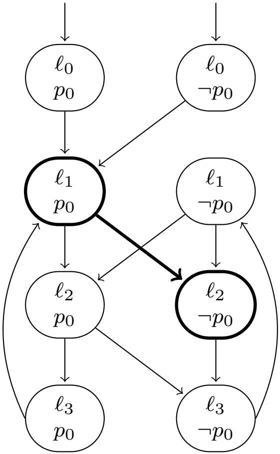
We had $\neg\text{Pred}(0 \le x < 1, \{p_0\})$ = *true*

...and $\neg p_0 \Rightarrow$ *true*



Cause and Refinement

$$\begin{array}{ll} \ell_0: & i := 1; \\ \ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\ \ell_2: & \quad i := i - 1; \\ \ell_3: & \quad x := x + 1; \\ & \} \end{array}$$



What caused this?

$$(\ell_0, p_0)\ (\ell_1, p_0)\ (\ell_2, \neg p_0)$$

We had $\neg$Pred$(0 \leq x < 1, \{p_0\})$ = *true*

...and $\neg p_0 \Rightarrow$ *true*

How do we fix it?

$$
\begin{array}{ll}
\ell_0: & i := 1; \\
\ell_1: & \textbf{while}(0 \leq x < 1)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \}
\end{array}
$$

What caused this?

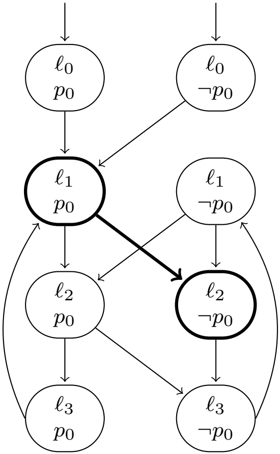
$$(\ell_0, p_0)\ (\ell_1, p_0)\ (\ell_2, \neg p_0)$$

We had $\neg\text{Pred}(0 \leq x < 1, \{p_0\})$ = *true*

...and $\neg p_0 \Rightarrow$ *true*

How do we fix it?

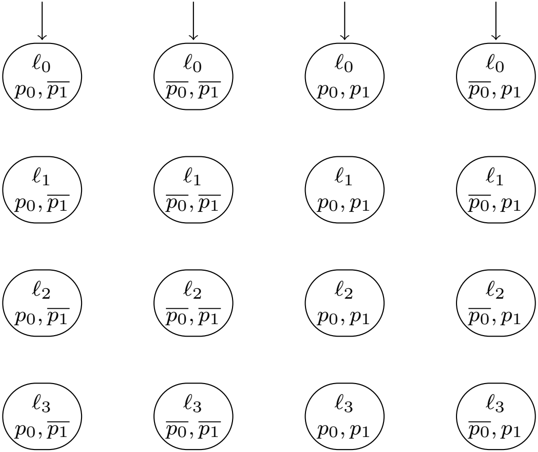
$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}\}$$



Example: Abstraction Refinement

ℓ_0 : $i := 1;$
ℓ_1 : **while**$(0 \leq x < 1)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}

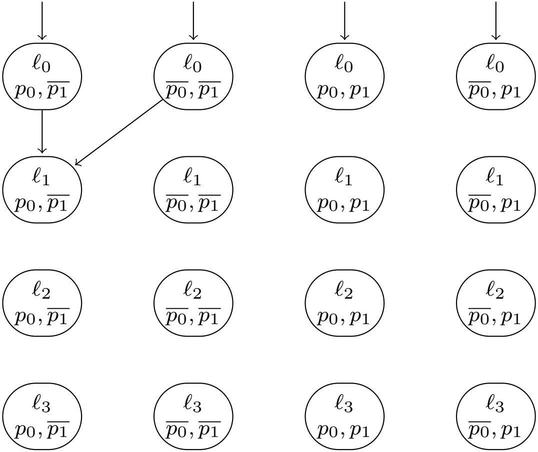
$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}\}$$



Example: Abstraction Refinement

ℓ_0 : $i := 1;$
ℓ_1 : **while** $(0 \le x < 1)$ {
ℓ_2 : $\quad i := i - 1;$
ℓ_3 : $\quad x := x + 1;$
}

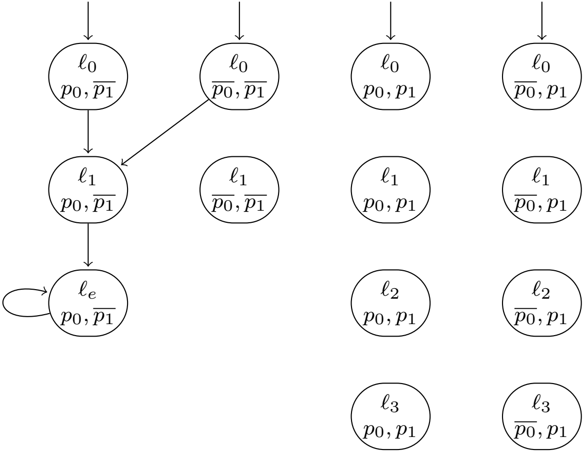
$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}\}$$



Example: Abstraction Refinement

ℓ_0 : $i := 1$;
ℓ_1 : **while**$(0 \le x < 1)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e : **skip**

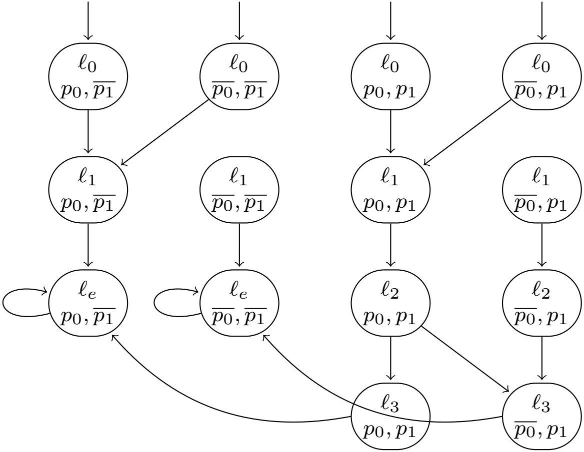
$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}\}$$



Example: Abstraction Refinement

ℓ_0 :$i := 1$;
ℓ_1 :**while**$(0 \leq x < 1)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e :**skip**

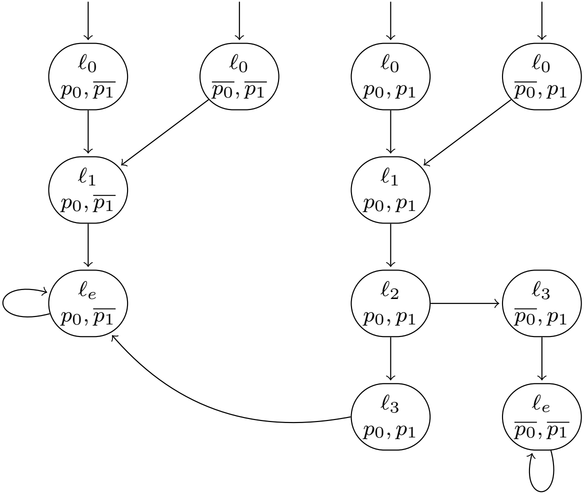
$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}\}$$



Example: Abstraction Refinement

ℓ_0 : $i := 1$;
ℓ_1 : **while**$(0 \leq x < 1)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}\}$$

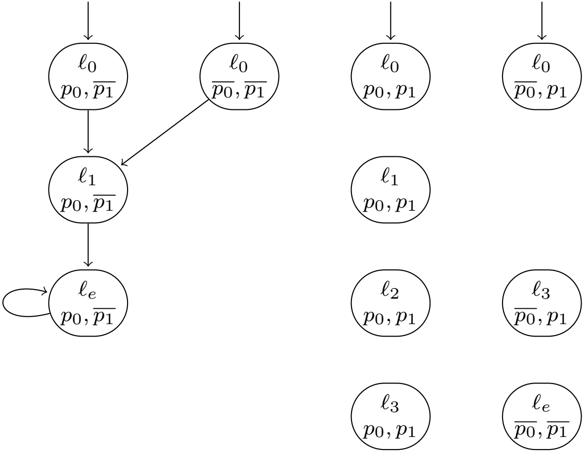


Example: Abstraction Refinement

$\ell_0 : i := 1;$
$\ell_1 :$ **while** $(0 \leq x < 1)$ {
$\ell_2 : \quad i := i - 1;$
$\ell_3 : \quad x := x + 1;$
}
$\ell_e :$ **skip**

$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}\}$$

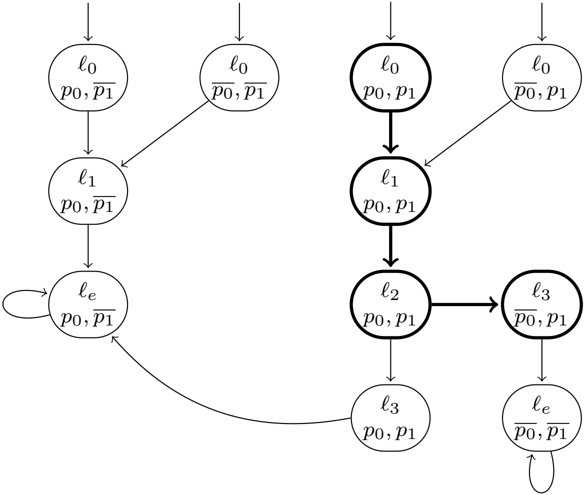
Is there a counterexample?



Example: Abstraction Refinement

ℓ_0 : $i := 1;$
ℓ_1 : **while** $(0 \le x < 1)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}\}$$

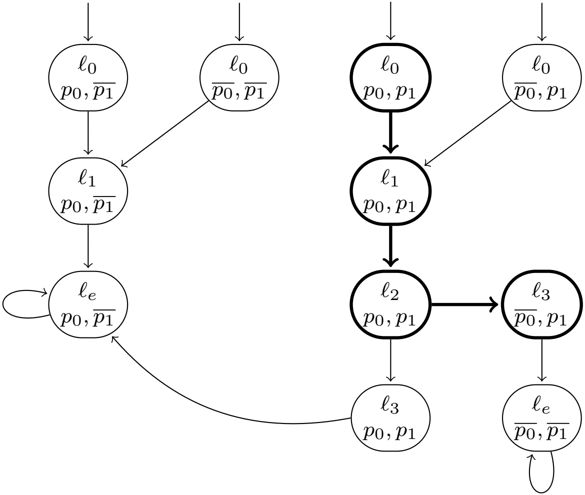


ℓ_0 : $i := 1$;
ℓ_1 : **while** $(0 \le x < 1)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}\}$$

Is this valid?

$\{0 \le i \land 0 \le x < 1\}$
$i := 1$;
assume $(0 \le x < 1)$
$i := i - 1$;
$\{i < 0 \land 0 \le x < 1\}$



Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant

Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant

We should really add this to our abtraction set E

Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant

We should really add this to our abtraction set E

This is turning into a lot of work!

Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant

We should really add this to our abtraction set E

This is turning into a lot of work!

- Now we have 8 initial states...

Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant

We should really add this to our abtraction set E

This is turning into a lot of work!

- Now we have 8 initial states...
- *#loc* $\times 2^{|E|}$ states in general

Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant

We should really add this to our abtraction set E

This is turning into a lot of work!

- Now we have 8 initial states...
- *#loc* $\times 2^{|E|}$ states in general
- There must be a better way!

Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant

We should really add this to our abtraction set E

This is turning into a lot of work!

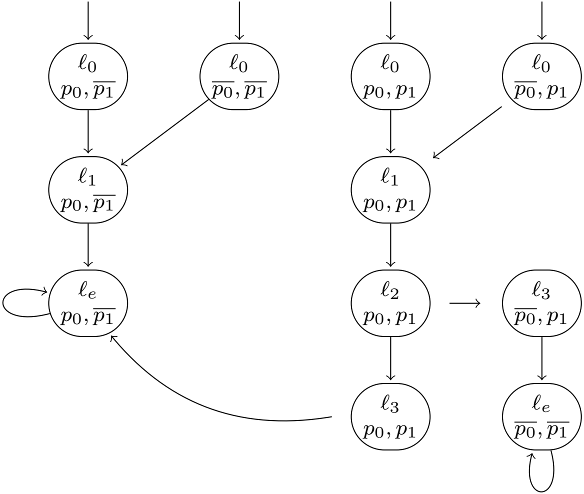
- Now we have 8 initial states...
- *#loc* $\times 2^{|E|}$ states in general
- There must be a better way!

Idea: Don't refine error-free parts of the abstraction

Example: Lazy Abstraction

ℓ_0 : $i := 1;$
ℓ_1 : **while**$(0 \le x < 1)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}\}$$



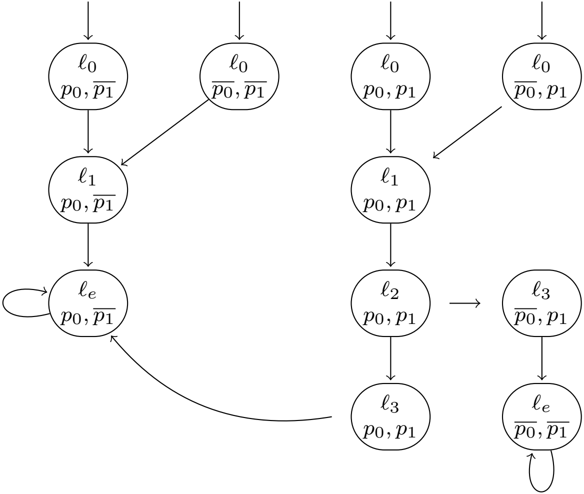
Example: Lazy Abstraction

ℓ_0 :i := 1;
ℓ_1 :**while**$(0 \leq x < 1)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e :**skip**

$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}\}$$

Don't need to update left side with

$$p_2 \Leftrightarrow i = 1$$



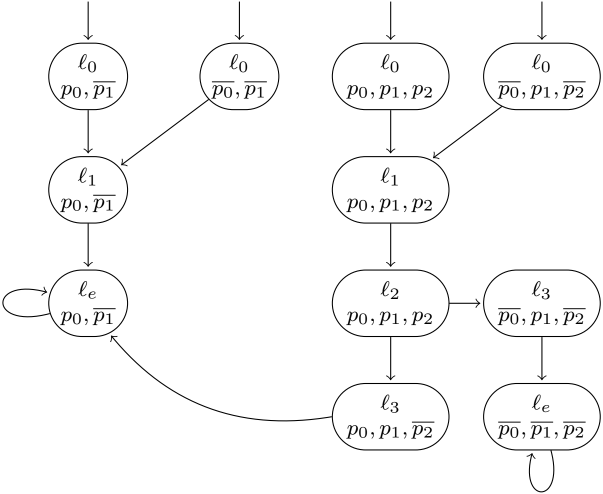
Example: Lazy Abstraction

ℓ_0 : $i := 1;$
ℓ_1 : **while**$(0 \le x < 1)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}\}$$

Don't need to update left side with

$$p_2 \Leftrightarrow i = 1$$



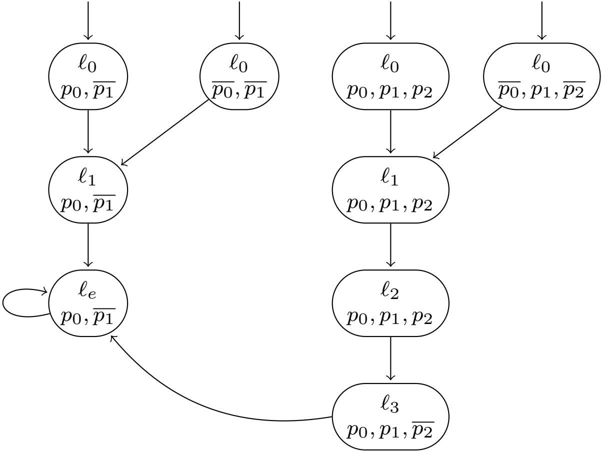
Example: Lazy Abstraction

ℓ_0 : $i := 1;$
ℓ_1 : **while** $(0 \leq x < 1)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}\}$$

Don't need to update left side with

$$p_2 \Leftrightarrow i = 1$$



Example: Lazy Abstraction

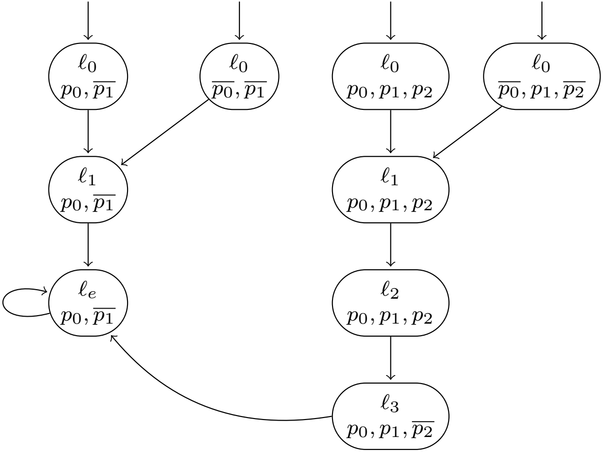
ℓ_0 : $i := 1$;
ℓ_1 : **while**$(0 \le x < 1)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}\}$$

Don't need to update left side with

$$p_2 \Leftrightarrow i = 1$$

Now there's no counterexample



More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree

More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree

1. Pick an abstract initial state

More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree

1. Pick an abstract initial state
2. Add children by computing abstract transitions

More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree

1. Pick an abstract initial state
2. Add children by computing abstract transitions
3. Only refine subtrees that could contain errors

More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree

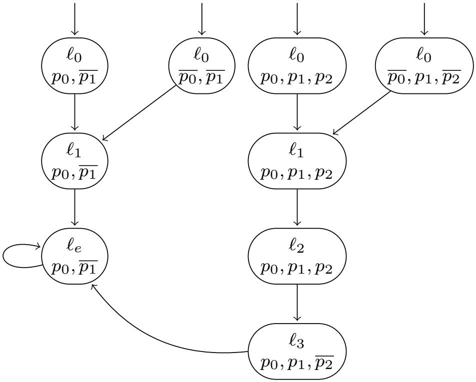
1. Pick an abstract initial state
2. Add children by computing abstract transitions
3. Only refine subtrees that could contain errors

In practice, this approach gives drastic performance improvements

Proofs from Abstractions

ℓ_0 : $i := 1$;
ℓ_1 : **while**$(0 \leq x < 1)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e : **skip**

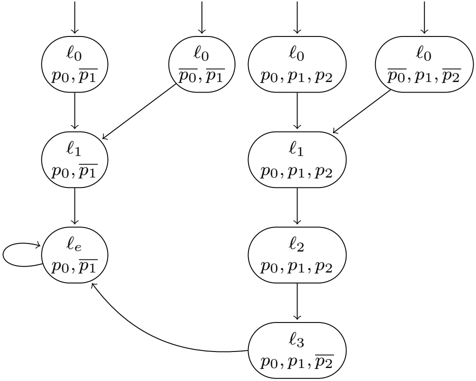
$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$



Proofs from Abstractions

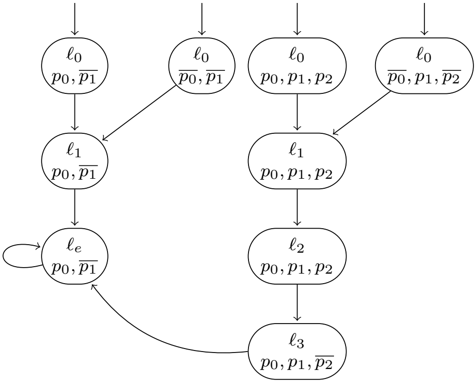
$\{\mathit{true}\}$
ℓ_0 : $i := 1$;
ℓ_1 : **while** $(0 \le x < 1)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$



$$\{\textit{true}\}$$
$$\ell_0 : i := 1;$$
$$\{0 \le i \wedge i = 1\}$$
$$\ell_1 : \textbf{while}(0 \le x < 1)\ \{$$
$$\ell_2 : \ \ i := i - 1;$$
$$\ell_3 : \ \ x := x + 1;$$
$$\}$$
$$\ell_e : \textbf{skip}$$

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$



Proofs from Abstractions

$\{\textit{true}\}$

$\ell_0 : i := 1;$

$\{0 \leq i \wedge i = 1\}$

$\ell_1 : \textbf{while}(0 \leq x < 1)\ \{$

$\{0 \leq i \wedge 0 \leq x < 1 \wedge i = 1\}$

$\ell_2 : \ i := i - 1;$

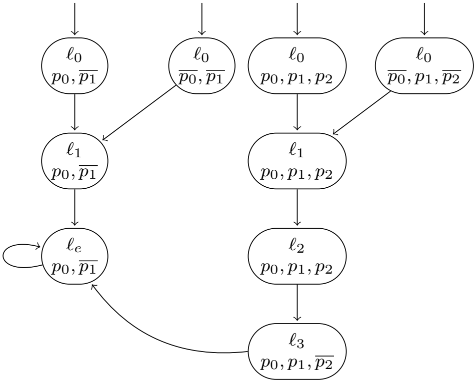
$\{0 \leq i \wedge 0 \leq x < 1\}$

$\ell_3 : \ x := x + 1;$

$\}$

$\ell_e : \textbf{skip}$

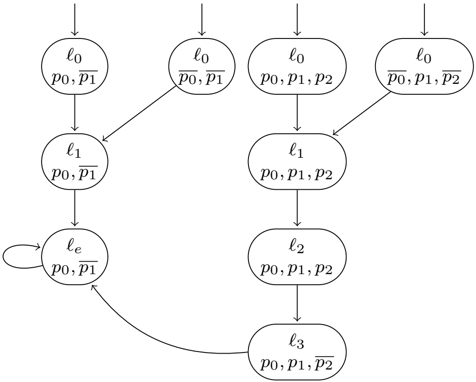
$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$



Proofs from Abstractions

$\{\textit{true}\}$
$\ell_0 : i := 1;$
$\{0 \leq i \wedge i = 1\}$
$\ell_1 :$ **while**$(0 \leq x < 1)$ $\{$
$\{0 \leq i \wedge 0 \leq x < 1 \wedge i = 1\}$
$\ell_2 : \ i := i - 1;$
$\{0 \leq i \wedge 0 \leq x < 1\}$
$\ell_3 : \ x := x + 1;$
$\{0 \leq i \wedge \neg(0 \leq x < 1)\}$
$\}$
$\ell_e :$ **skip**

$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$



Proofs from Abstractions

$\{\textit{true}\}$
$\ell_0 : i := 1;$
$\{0 \le i \land i = 1\}$
$\ell_1 : \textbf{while}(0 \le x < 1)\ \{$
$\{0 \le i \land 0 \le x < 1 \land i = 1\}$
$\ell_2 : \ i := i - 1;$
$\{0 \le i \land 0 \le x < 1 \land i \ne 1\}$
$\ell_3 : \ x := x + 1;$
$\{0 \le i \land \neg(0 \le x < 1) \land i \ne 1\}$
$\}$
$\ell_e : \textbf{skip}$

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$

Proofs from Abstractions

$\{true\}$
$\ell_0 : i := 1;$
$\{0 \leq i \wedge i = 1\}$
$\ell_1 :$ **while** $(0 \leq x < 1)$ {
$\{0 \leq i \wedge 0 \leq x < 1 \wedge i = 1\}$
$\ell_2 : \; i := i - 1;$
$\{0 \leq i \wedge 0 \leq x < 1 \wedge i \neq 1\}$
$\ell_3 : \; x := x + 1;$
$\{0 \leq i \wedge \neg(0 \leq x < 1) \wedge i \neq 1\}$
}
$\ell_e :$ **skip**

$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$

These annotations are sufficient to prove the property

Proofs from Abstractions

$\{\textit{true}\}$
$\ell_0 : i := 1;$
$\{0 \le i \wedge i = 1\}$
$\ell_1 : \textbf{while}(0 \le x < 1)\ \{$
$\{0 \le i \wedge 0 \le x < 1 \wedge i = 1\}$
$\ell_2 :\ i := i - 1;$
$\{0 \le i \wedge 0 \le x < 1 \wedge i \ne 1\}$
$\ell_3 :\ x := x + 1;$
$\{0 \le i \wedge \neg(0 \le x < 1) \wedge i \ne 1\}$
$\}$
$\ell_e : \textbf{skip}$

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$

These annotations are sufficient to prove the property

Suppose we wanted to verify

$$\{\textit{true}\}\ \textit{Prog}\ \{0 \le i\}$$

Proofs from Abstractions

$\{true\}$
ℓ_0 : $i := 1;$
$\{0 \le i \wedge i = 1\}$
ℓ_1 : **while**$(0 \le x < 1)$ {
$\{0 \le i \wedge 0 \le x < 1 \wedge i = 1\}$
ℓ_2 : $i := i - 1;$
$\{0 \le i \wedge 0 \le x < 1 \wedge i \ne 1\}$
ℓ_3 : $x := x + 1;$
$\{0 \le i \wedge \neg(0 \le x < 1) \wedge i \ne 1\}$
}
ℓ_e : **skip**

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$

These annotations are sufficient to prove the property

Suppose we wanted to verify

$$\{true\}\ Prog\ \{0 \le i\}$$

What is our loop invariant?

Proofs from Abstractions

$\{\textit{true}\}$
$\ell_0 : i := 1;$
$\{0 \leq i \wedge i = 1\}$
$\ell_1 : \textbf{while}(0 \leq x < 1)\ \{$
$\{0 \leq i \wedge 0 \leq x < 1 \wedge i = 1\}$
$\ell_2 : \ i := i - 1;$
$\{0 \leq i \wedge 0 \leq x < 1 \wedge i \neq 1\}$
$\ell_3 : \ x := x + 1;$
$\{0 \leq i \wedge \neg(0 \leq x < 1) \wedge i \neq 1\}$
$\}$
$\ell_e : \textbf{skip}$

$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$

These annotations are sufficient to prove the property

Suppose we wanted to verify

$$\{\textit{true}\}\ \textit{Prog}\ \{0 \leq i\}$$

What is our loop invariant?

$$
\begin{array}{ll}
 & (0 \leq i \wedge i = 1) \\
\vee & (0 \leq i \wedge 0 \leq x < 1 \wedge i = 1) \\
\vee & (0 \leq i \wedge 0 \leq x < 1 \wedge i \neq 1) \\
\vee & (0 \leq i \wedge \neg(0 \leq x < 1) \wedge i \neq 1)
\end{array}
$$

Proofs from Abstractions

$\{\textit{true}\}$
$\ell_0 : i := 1;$
$\{0 \leq i \wedge i = 1\}$
$\ell_1 : \textbf{while}(0 \leq x < 1) \ \{$
$\{0 \leq i \wedge 0 \leq x < 1 \wedge i = 1\}$
$\ell_2 : \ i := i - 1;$
$\{0 \leq i \wedge 0 \leq x < 1 \wedge i \neq 1\}$
$\ell_3 : \ x := x + 1;$
$\{0 \leq i \wedge \neg(0 \leq x < 1) \wedge i \neq 1\}$
$\}$
$\ell_e : \textbf{skip}$

$$E = \{\underbrace{0 \leq i}_{p_0}, \underbrace{0 \leq x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$

These annotations are sufficient to prove the property

Suppose we wanted to verify

$$\{\textit{true}\} \ \textit{Prog} \ \{0 \leq i\}$$

What is our loop invariant?

$$\begin{array}{ll} & (0 \leq i \wedge i = 1) \\ \vee & (0 \leq i \wedge 0 \leq x < 1 \wedge i = 1) \\ \vee & (0 \leq i \wedge 0 \leq x < 1 \wedge i \neq 1) \\ \vee & (0 \leq i \wedge \neg(0 \leq x < 1) \wedge i \neq 1) \\ \Leftrightarrow & \\ 0 \leq i & \end{array}$$

Proofs from Abstractions

$$
\begin{array}{ll}
 & \{\textit{true}\} \\
\ell_0: & i := 1; \\
 & \{0 \le i \land i = 1\} \\
\ell_1: & \textbf{while}(0 \le x < 1)\ \{ \\
 & \{0 \le i \land 0 \le x < 1 \land i = 1\} \\
\ell_2: & \quad i := i - 1; \\
 & \{0 \le i \land 0 \le x < 1 \land i \ne 1\} \\
\ell_3: & \quad x := x + 1; \\
 & \{0 \le i \land \neg(0 \le x < 1) \land i \ne 1\} \\
 & \} \\
\ell_e: & \textbf{skip}
\end{array}
$$

$$E = \{\underbrace{0 \le i}_{p_0}, \underbrace{0 \le x < 1}_{p_1}, \underbrace{i = 1}_{p_2}\}$$

These annotations are sufficient to prove the property

Suppose we wanted to verify

$$\{\textit{true}\}\ \textit{Prog}\ \{0 \le i\}$$

What is our loop invariant?

$$
\begin{array}{ll}
 & (0 \le i \land i = 1) \\
\lor & (0 \le i \land 0 \le x < 1 \land i = 1) \\
\lor & (0 \le i \land 0 \le x < 1 \land i \ne 1) \\
\lor & (0 \le i \land \neg(0 \le x < 1) \land i \ne 1) \\
\Leftrightarrow & \\
0 \le i &
\end{array}
$$

CEGAR automatically constructs deductive proofs!

Limitations

Suppose we wanted to verify:

$$
\begin{array}{ll}
 & \{\textit{true}\} \\
\ell_0 : & i := 10; \\
\ell_1 : & \textbf{while}(0 \leq x < 10)\ \{ \\
\ell_2 : & \quad i := i - 1; \\
\ell_3 : & \quad x := x + 1; \\
 & \} \\
\ell_e : & \textbf{skip} \\
 & \{0 \leq i\}
\end{array}
$$

Limitations

Suppose we wanted to verify:

How would we do it by hand?

$$
\begin{array}{ll}
 & \{\textit{true}\} \\
\ell_0: & i := 10; \\
\ell_1: & \textbf{while}(0 \leq x < 10)\ \{ \\
\ell_2: & \quad i := i - 1; \\
\ell_3: & \quad x := x + 1; \\
 & \} \\
\ell_e: & \textbf{skip} \\
 & \{0 \leq i\}
\end{array}
$$

Limitations

Suppose we wanted to verify:

$$\begin{aligned}
&\{\textit{true}\}\\
\ell_0 :& i := 10;\\
\ell_1 :& \textbf{while}(0 \le x < 10)\ \{\\
\ell_2 :& \quad i := i - 1;\\
\ell_3 :& \quad x := x + 1;\\
&\}\\
\ell_e :& \textbf{skip}\\
&\{0 \le i\}
\end{aligned}$$

How would we do it by hand?

- Find the invariant $0 \le i - x$

How would CEGAR do it?

Limitations

Suppose we wanted to verify:

$\{\textit{true}\}$
ℓ_0 : $i := 10$;
ℓ_1 : **while**$(0 \leq x < 10)$ {
ℓ_2 : $i := i - 1$;
ℓ_3 : $x := x + 1$;
}
ℓ_e : **skip**
$\{0 \leq i\}$

How would we do it by hand?

- Find the invariant $0 \leq i - x$

How would CEGAR do it?

- Find $i = 10, x = 9$

Limitations

Suppose we wanted to verify:

$$
\begin{aligned}
& \{\textit{true}\} \\
\ell_0 :& i := 10; \\
\ell_1 :& \textbf{while}(0 \le x < 10)\ \{ \\
\ell_2 :& \quad i := i - 1; \\
\ell_3 :& \quad x := x + 1; \\
& \} \\
\ell_e :& \textbf{skip} \\
& \{0 \le i\}
\end{aligned}
$$

How would we do it by hand?

- Find the invariant $0 \le i - x$

How would CEGAR do it?

- Find $i = 10, x = 9$
- Find $i = 10, x = 8$

Limitations

Suppose we wanted to verify:

$\{\mathit{true}\}$
$\ell_0 : i := 10;$
$\ell_1 : \textbf{while}(0 \le x < 10)\ \{$
$\ell_2 : \quad i := i - 1;$
$\ell_3 : \quad x := x + 1;$
$\}$
$\ell_e : \textbf{skip}$
$\{0 \le i\}$

How would we do it by hand?

- Find the invariant $0 \le i - x$

How would CEGAR do it?

- Find $i = 10, x = 9$
- Find $i = 10, x = 8$
- ...

Limitations

Suppose we wanted to verify:

$\{\textit{true}\}$
$\ell_0 : i := 10;$
$\ell_1 : \textbf{while}(0 \leq x < 10)\ \{$
$\ell_2 : \quad i := i - 1;$
$\ell_3 : \quad x := x + 1;$
$\}$
$\ell_e : \textbf{skip}$
$\{0 \leq i\}$

How would we do it by hand?

- Find the invariant $0 \leq i - x$

How would CEGAR do it?

- Find $i = 10, x = 9$
- Find $i = 10, x = 8$
- ...
- Find $i = 9$

Limitations

Suppose we wanted to verify:

$\{true\}$
ℓ_0 : $i := 10;$
ℓ_1 : **while** $(0 \leq x < 10)$ $\{$
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
$\}$
ℓ_e : **skip**
$\{0 \leq i\}$

How would we do it by hand?

- Find the invariant $0 \leq i - x$

How would CEGAR do it?

- Find $i = 10, x = 9$
- Find $i = 10, x = 8$
- ...
- Find $i = 9$
- ...

Limitations

Suppose we wanted to verify:

$\{\textit{true}\}$
$\ell_0 : i := 10;$
$\ell_1 : \textbf{while}(0 \leq x < 10)\ \{$
$\ell_2 : \quad i := i - 1;$
$\ell_3 : \quad x := x + 1;$
$\}$
$\ell_e : \textbf{skip}$
$\{0 \leq i\}$

How would we do it by hand?

- Find the invariant $0 \leq i - x$

How would CEGAR do it?

- Find $i = 10, x = 9$
- Find $i = 10, x = 8$
- …
- Find $i = 9$
- …

Finding the right predicates early is crucial

Learning Predicates

Before, we found new predicates by intuition

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Key tool: SMT solver

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Key tool: SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate ϕ_{path}

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Key tool: SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate ϕ_{path}
- ϕ_{path} is *sat* iff $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ not spurious

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Key tool: SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate ϕ_{path}
- ϕ_{path} is *sat* iff $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ not spurious
- If ϕ_{path} *unsat*, extract predicates from “witness”

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Key tool: SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate ϕ_{path}
- ϕ_{path} is *sat* iff $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ not spurious
- If ϕ_{path} *unsat*, extract predicates from "witness"

Intuitively,

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Key tool: SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate ϕ_{path}
- ϕ_{path} is *sat* iff $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ not spurious
- If ϕ_{path} *unsat*, extract predicates from "witness"

Intuitively,

- ϕ_{path} simulates executing the counterexample path

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Key tool: SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate ϕ_{path}
- ϕ_{path} is *sat* iff $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ not spurious
- If ϕ_{path} *unsat*, extract predicates from "witness"

Intuitively,

- ϕ_{path} simulates executing the counterexample path
- If execution completes without error, path is valid counterexample

Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

Key tool: SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate ϕ_{path}
- ϕ_{path} is *sat* iff $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ not spurious
- If ϕ_{path} *unsat*, extract predicates from "witness"

Intuitively,

- ϕ_{path} simulates executing the counterexample path
- If execution completes without error, path is valid counterexample
- Otherwise, take an observation that explains why the path won't execute

SSA Form

To build ϕ_{path}, we'll put path in **static single-assignment** (SSA) form

SSA Form

To build ϕ_{path}, we'll put path in **static single-assignment** (SSA) form

Assume we're given a path with only assign, **assume**, **assert**

SSA Form

To build ϕ_{path}, we'll put path in **static single-assignment** (SSA) form

Assume we're given a path with only assign, **assume**, **assert**

Each variable is only assigned once:

SSA Form

To build ϕ_{path}, we'll put path in **static single-assignment** (SSA) form

Assume we're given a path with only assign, **assume**, **assert**

Each variable is only assigned once:

1. Attach subscripts to vars, starting at 0

SSA Form

To build ϕ_{path}, we'll put path in **static single-assignment** (SSA) form

Assume we're given a path with only assign, **assume**, **assert**

Each variable is only assigned once:

1. Attach subscripts to vars, starting at 0
2. Each time a variable is assigned, increment its subscript

SSA Form

To build ϕ_{path}, we'll put path in **static single-assignment** (SSA) form

Assume we're given a path with only assign, **assume**, **assert**

Each variable is only assigned once:

1. Attach subscripts to vars, starting at 0
2. Each time a variable is assigned, increment its subscript
3. All reads of the variable use the must recent subscript

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions

assert $0 \leq i$
$i := 1$
assert $0 \leq i$
assume $0 \leq x < 1$
assert $\neg(0 \leq i)$
$i := i - 1$

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions
2. Convert the path into SSA form

assert $0 \leq i$
$i := 1$
assert $0 \leq i$
assume $0 \leq x < 1$
assert $\neg(0 \leq i)$
$i := i - 1$

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions
2. Convert the path into SSA form

assert $0 \leq i$	**assert** $0 \leq i_0$
$i := 1$	$i_1 := 1$
assert $0 \leq i$	**assert** $0 \leq i_1$
assume $0 \leq x < 1$	**assume** $0 \leq x_0 < 1$
assert $\neg(0 \leq i)$	**assert** $\neg(0 \leq i_1)$
$i := i - 1$	$i_0 := i_1 - 1$

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions
2. Convert the path into SSA form
3. Replace assignments with **assume** over equality

assert $0 \leq i$
$i := 1$
assert $0 \leq i$
assume $0 \leq x < 1$
assert $\neg(0 \leq i)$
$i := i - 1$

assert $0 \leq i_0$
$i_1 := 1$
assert $0 \leq i_1$
assume $0 \leq x_0 < 1$
assert $\neg(0 \leq i_1)$
$i_0 := i_1 - 1$

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions
2. Convert the path into SSA form
3. Replace assignments with **assume** over equality

assert $0 \leq i$	**assert** $0 \leq i_0$	**assert** $0 \leq i_0$
$i := 1$	$i_1 := 1$	**assume** $i_1 = 1$
assert $0 \leq i$	**assert** $0 \leq i_1$	**assert** $0 \leq i_1$
assume $0 \leq x < 1$	**assume** $0 \leq x_0 < 1$	**assume** $0 \leq x_0 < 1$
assert $\neg(0 \leq i)$	**assert** $\neg(0 \leq i_1)$	**assert** $\neg(0 \leq i_1)$
$i := i - 1$	$i_0 := i_1 - 1$	**assume** $i_2 = i_1 - 1$

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions
2. Convert the path into SSA form
3. Replace assignments with **assume** over equality
4. Compute weakest precondition of path wrt. *true*

assert $0 \leq i$
$i := 1$
assert $0 \leq i$
assume $0 \leq x < 1$
assert $\neg(0 \leq i)$
$i := i - 1$

assert $0 \leq i_0$
$i_1 := 1$
assert $0 \leq i_1$
assume $0 \leq x_0 < 1$
assert $\neg(0 \leq i_1)$
$i_0 := i_1 - 1$

assert $0 \leq i_0$
assume $i_1 = 1$
assert $0 \leq i_1$
assume $0 \leq x_0 < 1$
assert $\neg(0 \leq i_1)$
assume $i_2 = i_1 - 1$

Building Path Formulas

We're given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions
2. Convert the path into SSA form
3. Replace assignments with **assume** over equality
4. Compute weakest precondition of path wrt. *true*

assert $0 \leq i$	**assert** $0 \leq i_0$	**assert** $0 \leq i_0$
$i := 1$	$i_1 := 1$	**assume** $i_1 = 1$
assert $0 \leq i$	**assert** $0 \leq i_1$	**assert** $0 \leq i_1$
assume $0 \leq x < 1$	**assume** $0 \leq x_0 < 1$	**assume** $0 \leq x_0 < 1$
assert $\neg(0 \leq i)$	**assert** $\neg(0 \leq i_1)$	**assert** $\neg(0 \leq i_1)$
$i := i - 1$	$i_0 := i_1 - 1$	**assume** $i_2 = i_1 - 1$

wp$(\ldots,$ *true*$) = 0 \leq i_0 \wedge i_1 = 1 \wedge 0 \leq i_1 \wedge 0 \leq x_0 < 1 \wedge \neg(0 \leq i_1) \wedge i_2 = i_1 - 1$

Path Validity

We have a counterexample, path formula pair

Path Validity

We have a counterexample, path formula pair

$(i := 1, \textit{true})$
$(\textbf{assume } 0 \leq x < 1, 0 \leq i)$
$(i := i - 1, \neg(0 \leq i))$

Path Validity

We have a counterexample, path formula pair

$(i := 1, \textit{true})$
$(\textbf{assume } 0 \leq x < 1, 0 \leq i)$
$(i := i - 1, \neg(0 \leq i))$

$$
\begin{aligned}
& 0 \leq i_0 \wedge \\
& i_1 = 1 \wedge \\
& 0 \leq i_1 \wedge \\
& 0 \leq x_0 < 1 \wedge \\
& \neg(0 \leq i_1) \wedge \\
& i_2 = i_1 - 1
\end{aligned}
$$

Path Validity

We have a counterexample, path formula pair

$(i := 1, \textit{true})$
$(\textbf{assume } 0 \leq x < 1, 0 \leq i)$
$(i := i - 1, \neg(0 \leq i))$

$$
\begin{array}{l}
0 \leq i_0 \wedge \\
i_1 = 1 \wedge \\
0 \leq i_1 \wedge \\
0 \leq x_0 < 1 \wedge \\
\neg(0 \leq i_1) \wedge \\
i_2 = i_1 - 1
\end{array}
$$

Is the path formula satisfiable?

No. We already knew this path was invalid

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable
- Dropping any element of C^* makes it satisfiable

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable
- Dropping any element of C^* makes it satisfiable

To generate: For each literal l in C:

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable
- Dropping any element of C^* makes it satisfiable

To generate: For each literal l in C:

1. Drop l from C to build C'

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable
- Dropping any element of C^* makes it satisfiable

To generate: For each literal l in C:

1. Drop l from C to build C'
2. If C' is still unsatisfiable, then let $C := C'$

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable
- Dropping any element of C^* makes it satisfiable

To generate: For each literal l in C:

1. Drop l from C to build C'
2. If C' is still unsatisfiable, then let $C := C'$
3. Otherwise, keep original C

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable
- Dropping any element of C^* makes it satisfiable

To generate: For each literal l in C:

1. Drop l from C to build C'
2. If C' is still unsatisfiable, then let $C := C'$
3. Otherwise, keep original C

We'll modify this slightly:

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable
- Dropping any element of C^* makes it satisfiable

To generate: For each literal l in C:

1. Drop l from C to build C'
2. If C' is still unsatisfiable, then let $C := C'$
3. Otherwise, keep original C

We'll modify this slightly:

1. First, enumerate every $l, l' \in C$ where $l \neq l'$

Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An **unsatisfiable core** C^* is a subset of C:

- C^* is still unsatisfiable
- Dropping any element of C^* makes it satisfiable

To generate: For each literal l in C:

1. Drop l from C to build C'
2. If C' is still unsatisfiable, then let $C := C'$
3. Otherwise, keep original C

We'll modify this slightly:

1. First, enumerate every $l, l' \in C$ where $l \neq l'$
2. If $l \Rightarrow l'$, then remove l'

Example: Learning New Predicates

Initial formula:

$$
\begin{array}{ll}
0 \leq i_0 & \wedge \\
i_1 = 1 & \wedge \\
0 \leq i_1 & \wedge \\
0 \leq x_0 < 1 & \wedge \\
\neg(0 \leq i_1) & \wedge \\
i_2 = i_1 - 1 &
\end{array}
$$

Example: Learning New Predicates

Initial formula:

$$\begin{array}{ll} 0 \leq i_0 & \wedge \\ i_1 = 1 & \wedge \\ 0 \leq i_1 & \wedge \\ 0 \leq x_0 < 1 & \wedge \\ \neg(0 \leq i_1) & \wedge \\ i_2 = i_1 - 1 & \end{array}$$

1: Remove $0 \leq i_1$ $(i_1 = 1 \Rightarrow 0 \leq i_1)$

$$\begin{array}{ll} 0 \leq i_0 & \wedge \\ i_1 = 1 & \wedge \\ 0 \leq x_0 < 1 & \wedge \\ \neg(0 \leq i_1) & \wedge \\ i_2 = i_1 - 1 & \end{array}$$

Example: Learning New Predicates

Initial formula:

$$
\begin{aligned}
0 \leq i_0 &\quad \wedge \\
i_1 = 1 &\quad \wedge \\
0 \leq i_1 &\quad \wedge \\
0 \leq x_0 < 1 &\quad \wedge \\
\neg(0 \leq i_1) &\quad \wedge \\
i_2 = i_1 - 1 &
\end{aligned}
$$

1: Remove $0 \leq i_1$ ($i_1 = 1 \Rightarrow 0 \leq i_1$)

$$
\begin{aligned}
0 \leq i_0 &\quad \wedge \\
i_1 = 1 &\quad \wedge \\
0 \leq x_0 < 1 &\quad \wedge \\
\neg(0 \leq i_1) &\quad \wedge \\
i_2 = i_1 - 1 &
\end{aligned}
$$

2: Remove $0 \leq i_0$

$$
\begin{aligned}
i_1 = 1 &\quad \wedge \\
0 \leq x_0 < 1 &\quad \wedge \\
\neg(0 \leq i_1) &\quad \wedge \\
i_2 = i_1 - 1 &
\end{aligned}
$$

Example: Learning New Predicates

Initial formula:

$$\begin{aligned} & 0 \leq i_0 && \wedge \\ & i_1 = 1 && \wedge \\ & 0 \leq i_1 && \wedge \\ & 0 \leq x_0 < 1 && \wedge \\ & \neg(0 \leq i_1) && \wedge \\ & i_2 = i_1 - 1 \end{aligned}$$

1: Remove $0 \leq i_1$ ($i_1 = 1 \Rightarrow 0 \leq i_1$)

$$\begin{aligned} & 0 \leq i_0 && \wedge \\ & i_1 = 1 && \wedge \\ & 0 \leq x_0 < 1 && \wedge \\ & \neg(0 \leq i_1) && \wedge \\ & i_2 = i_1 - 1 \end{aligned}$$

2: Remove $0 \leq i_0$

$$\begin{aligned} & i_1 = 1 && \wedge \\ & 0 \leq x_0 < 1 && \wedge \\ & \neg(0 \leq i_1) && \wedge \\ & i_2 = i_1 - 1 \end{aligned}$$

3: Remove $0 \leq x_0 < 1$

$$\begin{aligned} & i_1 = 1 && \wedge \\ & \neg(0 \leq i_1) && \wedge \\ & i_2 = i_1 - 1 \end{aligned}$$

Example: Learning New Predicates

Initial formula:

$$\begin{array}{ll} 0 \leq i_0 & \wedge \\ i_1 = 1 & \wedge \\ 0 \leq i_1 & \wedge \\ 0 \leq x_0 < 1 & \wedge \\ \neg(0 \leq i_1) & \wedge \\ i_2 = i_1 - 1 & \end{array}$$

1: Remove $0 \leq i_1$ $(i_1 = 1 \Rightarrow 0 \leq i_1)$

$$\begin{array}{ll} 0 \leq i_0 & \wedge \\ i_1 = 1 & \wedge \\ 0 \leq x_0 < 1 & \wedge \\ \neg(0 \leq i_1) & \wedge \\ i_2 = i_1 - 1 & \end{array}$$

2: Remove $0 \leq i_0$

$$\begin{array}{ll} i_1 = 1 & \wedge \\ 0 \leq x_0 < 1 & \wedge \\ \neg(0 \leq i_1) & \wedge \\ i_2 = i_1 - 1 & \end{array}$$

3: Remove $0 \leq x_0 < 1$

$$\begin{array}{ll} i_1 = 1 & \wedge \\ \neg(0 \leq i_1) & \wedge \\ i_2 = i_1 - 1 & \end{array}$$

4: Remove $i_2 = i_1 - 1$

$$\begin{array}{ll} i_1 = 1 & \wedge \\ \neg(0 \leq i_1) & \wedge \end{array}$$

Example: Learning New Predicates

Initial formula:

$$\begin{aligned} &0 \le i_0 && \wedge \\ &i_1 = 1 && \wedge \\ &0 \le i_1 && \wedge \\ &0 \le x_0 < 1 && \wedge \\ &\neg(0 \le i_1) && \wedge \\ &i_2 = i_1 - 1 \end{aligned}$$

1: Remove $0 \le i_1$ $(i_1 = 1 \Rightarrow 0 \le i_1)$

$$\begin{aligned} &0 \le i_0 && \wedge \\ &i_1 = 1 && \wedge \\ &0 \le x_0 < 1 && \wedge \\ &\neg(0 \le i_1) && \wedge \\ &i_2 = i_1 - 1 \end{aligned}$$

2: Remove $0 \le i_0$

$$\begin{aligned} &i_1 = 1 && \wedge \\ &0 \le x_0 < 1 && \wedge \\ &\neg(0 \le i_1) && \wedge \\ &i_2 = i_1 - 1 \end{aligned}$$

3: Remove $0 \le x_0 < 1$

$$\begin{aligned} &i_1 = 1 && \wedge \\ &\neg(0 \le i_1) && \wedge \\ &i_2 = i_1 - 1 \end{aligned}$$

4: Remove $i_2 = i_1 - 1$

$$\begin{aligned} &i_1 = 1 && \wedge \\ &\neg(0 \le i_1) && \wedge \end{aligned}$$

$i_1 = 1$ wasn't previously in our set, so we refine by adding it

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the "counterexample" $(\ell_1, \textit{true}), \ldots, (\ell_n, \textit{true}), (\textbf{skip}, \neg\phi)$

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the "counterexample" $(\ell_1, \textit{true}), \ldots, (\ell_n, \textit{true}), (\textbf{skip}, \neg\phi)$
4. Generate the path formula ϕ_{path}

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the "counterexample" $(\ell_1, \textit{true}), \ldots, (\ell_n, \textit{true}), (\textbf{skip}, \neg\phi)$
4. Generate the path formula ϕ_{path}
5. Check ϕ_{path} for satisfiability

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the "counterexample" $(\ell_1, \textit{true}), \ldots, (\ell_n, \textit{true}), (\textbf{skip}, \neg\phi)$
4. Generate the path formula ϕ_{path}
5. Check ϕ_{path} for satisfiability

The results tell us everything:

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the "counterexample" $(\ell_1, \textit{true}), \ldots, (\ell_n, \textit{true}), (\textbf{skip}, \neg\phi)$
4. Generate the path formula ϕ_{path}
5. Check ϕ_{path} for satisfiability

The results tell us everything:

- If *unsat*, there's no way to execute $\ell_1, \ldots, \ell_n$ satisfying $\neg\phi$

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the "counterexample" $(\ell_1, \textit{true}), \ldots, (\ell_n, \textit{true}), (\textbf{skip}, \neg\phi)$
4. Generate the path formula ϕ_{path}
5. Check ϕ_{path} for satisfiability

The results tell us everything:

- If *unsat*, there's no way to execute $\ell_1, \ldots, \ell_n$ satisfying $\neg\phi$
- If *sat*, then this path is a valid counterexample

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the "counterexample" $(\ell_1, \textit{true}), \ldots, (\ell_n, \textit{true}), (\textbf{skip}, \neg\phi)$
4. Generate the path formula ϕ_{path}
5. Check ϕ_{path} for satisfiability

The results tell us everything:

- If *unsat*, there's no way to execute $\ell_1, \ldots, \ell_n$ satisfying $\neg\phi$
- If *sat*, then this path is a valid counterexample

sat assignment to initial SSA variables is an input to the program

Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property **G** ϕ,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the "counterexample" $(\ell_1, \textit{true}), \ldots, (\ell_n, \textit{true}), (\textbf{skip}, \neg\phi)$
4. Generate the path formula ϕ_{path}
5. Check ϕ_{path} for satisfiability

The results tell us everything:

- If *unsat*, there's no way to execute $\ell_1, \ldots, \ell_n$ satisfying $\neg\phi$
- If *sat*, then this path is a valid counterexample

sat assignment to initial SSA variables is an input to the program

- When run on these inputs, the property will be violated

Bounded Model Checking: Example

ℓ_0 : $i := 1;$
ℓ_1 : **while**$(0 \leq x < 2)$ $\{$
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
$\}$
ℓ_e : **assert**$(0 \leq i)$

Bounded Model Checking: Example

ℓ_0 :$i := 1;$
ℓ_1 :**while**$(0 \leq x < 2)$ $\{$
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
$\}$
ℓ_e :**assert**$(0 \leq i)$

We suspect the path:

$i := 1;$
assume$(0 \leq x < 2)$
$i := i - 1;$
$x := x + 1;$
assume$(0 \leq x < 2)$
$i := i - 1;$
$x := x + 1;$
assume$(\neg(0 \leq x < 2))$
assert$(0 \leq i)$

Bounded Model Checking: Example

ℓ_0 : $i := 1;$
ℓ_1 : **while**$(0 \leq x < 2)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}
ℓ_e : **assert**$(0 \leq i)$

We suspect the path:

$i := 1;$
assume$(0 \leq x < 2)$
$i := i - 1;$
$x := x + 1;$
assume$(0 \leq x < 2)$
$i := i - 1;$
$x := x + 1;$
assume$(\neg(0 \leq x < 2))$
assert$(0 \leq i)$

After SSA, assumption encoding:

assume $i_1 = 1;$
assume $0 \leq x_0 < 2;$
assume $i_2 = i_1 - 1;$
assume $x_1 = x_0 + 1;$
assume $0 \leq x_1 < 2;$
assume $i_3 = i_2 - 1;$
assume $x_2 = x_1 + 1;$
assume $\neg(0 \leq x_2 < 2);$
assert $0 \leq i_3;$

Bounded Model Checking: Example

$$\begin{aligned}
&\ell_0 : i := 1;\\
&\ell_1 : \textbf{while}(0 \le x < 2)\ \{\\
&\ell_2 : \quad i := i - 1;\\
&\ell_3 : \quad x := x + 1;\\
&\quad\ \}\\
&\ell_e : \textbf{assert}(0 \le i)
\end{aligned}$$

We suspect the path:

$$\begin{aligned}
&i := 1;\\
&\textbf{assume}(0 \le x < 2)\\
&i := i - 1;\\
&x := x + 1;\\
&\textbf{assume}(0 \le x < 2)\\
&i := i - 1;\\
&x := x + 1;\\
&\textbf{assume}(\neg(0 \le x < 2))\\
&\textbf{assert}(0 \le i)
\end{aligned}$$

Path formula:

$$\begin{aligned}
&i_1 = 1 && \wedge\\
&0 \le x_0 < 2 && \wedge\\
&i_2 = i_1 - 1 && \wedge\\
&x_1 = x_0 + 1 && \wedge\\
&0 \le x_1 < 2 && \wedge\\
&i_3 = i_2 - 1 && \wedge\\
&x_2 = x_1 + 1 && \wedge\\
&\neg(0 \le x_2 < 2) && \wedge\\
&0 \le i_3
\end{aligned}$$

Bounded Model Checking: Example

ℓ_0 : $i := 1;$
ℓ_1 : **while** $(0 \le x < 2)$ {
ℓ_2 : $i := i - 1;$
ℓ_3 : $x := x + 1;$
}
ℓ_e : **assert** $(0 \le i)$

We suspect the path:

$i := 1;$
assume $(0 \le x < 2)$
$i := i - 1;$
$x := x + 1;$
assume $(0 \le x < 2)$
$i := i - 1;$
$x := x + 1;$
assume $(\neg(0 \le x < 2))$
assert $(0 \le i)$

Path formula:

$$
\begin{array}{ll}
i_1 = 1 & \wedge \\
0 \le x_0 < 2 & \wedge \\
i_2 = i_1 - 1 & \wedge \\
x_1 = x_0 + 1 & \wedge \\
0 \le x_1 < 2 & \wedge \\
i_3 = i_2 - 1 & \wedge \\
x_2 = x_1 + 1 & \wedge \\
\neg(0 \le x_2 < 2) & \wedge \\
0 \le i_3 &
\end{array}
$$

Is this satisfiable?

Bounded Model Checking: Example

$$
\begin{aligned}
\ell_0 &: i := 1; \\
\ell_1 &: \textbf{while}(0 \le x < 2)\ \{ \\
\ell_2 &: \quad i := i - 1; \\
\ell_3 &: \quad x := x + 1; \\
&\quad \} \\
\ell_e &: \textbf{assert}(0 \le i)
\end{aligned}
$$

We suspect the path:

$$
\begin{aligned}
& i := 1; \\
& \textbf{assume}(0 \le x < 2) \\
& i := i - 1; \\
& x := x + 1; \\
& \textbf{assume}(0 \le x < 2) \\
& i := i - 1; \\
& x := x + 1; \\
& \textbf{assume}(\neg(0 \le x < 2)) \\
& \textbf{assert}(0 \le i)
\end{aligned}
$$

Path formula:

$$
\begin{array}{ll}
i_1 = 1 & \wedge \\
0 \le x_0 < 2 & \wedge \\
i_2 = i_1 - 1 & \wedge \\
x_1 = x_0 + 1 & \wedge \\
0 \le x_1 < 2 & \wedge \\
i_3 = i_2 - 1 & \wedge \\
x_2 = x_1 + 1 & \wedge \\
\neg(0 \le x_2 < 2) & \wedge \\
0 \le i_3 &
\end{array}
$$

Is this satisfiable?

$i_1 = 1, x_0 = 0, i_2 = 0, x_1 = 1, i_3 = -1, x_2 = 2$

Bounded Model Checking: Example

$$\begin{aligned}
\ell_0 &: i := 1; \\
\ell_1 &: \textbf{while}(0 \leq x < 2)\ \{ \\
\ell_2 &: \quad i := i - 1; \\
\ell_3 &: \quad x := x + 1; \\
&\quad \} \\
\ell_e &: \textbf{assert}(0 \leq i)
\end{aligned}$$

We suspect the path:

$$\begin{aligned}
&i := 1; \\
&\textbf{assume}(0 \leq x < 2) \\
&i := i - 1; \\
&x := x + 1; \\
&\textbf{assume}(0 \leq x < 2) \\
&i := i - 1; \\
&x := x + 1; \\
&\textbf{assume}(\neg(0 \leq x < 2)) \\
&\textbf{assert}(0 \leq i)
\end{aligned}$$

Path formula:

$$\begin{aligned}
&i_1 = 1 && \wedge \\
&0 \leq x_0 < 2 && \wedge \\
&i_2 = i_1 - 1 && \wedge \\
&x_1 = x_0 + 1 && \wedge \\
&0 \leq x_1 < 2 && \wedge \\
&i_3 = i_2 - 1 && \wedge \\
&x_2 = x_1 + 1 && \wedge \\
&\neg(0 \leq x_2 < 2) && \wedge \\
&0 \leq i_3
\end{aligned}$$

Is this satisfiable?

$$i_1 = 1, x_0 = 0, i_2 = 0, x_1 = 1, i_3 = -1, x_2 = 2$$

We can use $x = 0$ as an initial test case

Next Lecture

Go over homeworks

Next Lecture

Go over homeworks

Review for the final

Next Lecture

Go over homeworks

Review for the final

Last homework due on Friday evening, 11:59

- No late days!
- University policy...