COL750: Foundations of Automatic Verification (Jan-May 2023)

Lectures 25 & 26 (Predicate Abstraction & CEGAR)[1]

Kumar Madhukar

madhukar@cse.iitd.ac.in

Apr 17th and 20th

[1]Most of the slides in this deck are taken directly from Daniel Kroening's slides from a tutorial on Predicate Abstraction that he gave at SRI. His slides are an excellent resource on this topic, and can be found here: https://fm.csl.sri.com/SSFT12/predabs-SSFT12.pdf

Abstraction

- reduce the size of the model by removing irrelevant details
- predicate abstraction – only track predicates on data (remove data variables)
- reduces state-space (from possibly an infinite set of states to a finite set of states given by the 0/1 values of the predicates)
- can be very effective for control-flow dominated properties
- but the question of relevance is a difficult one – how does one know what's relevant and what is not!

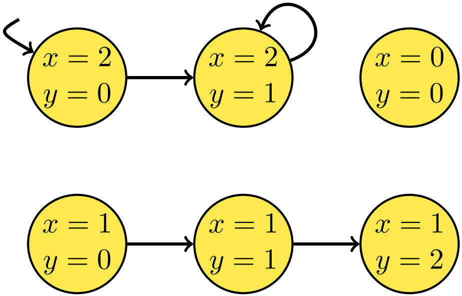
Sign abstraction

```
int x;

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

else if (x > 0)  x = x * 20;

else   // if (x < 0)
  x = x * -10;

assert (x > 0)
```

- we can prove the assertion by simply tracking the sign of x
- i.e., whether x is positive, negative, or zero (which can be thought of as three predicates on x)

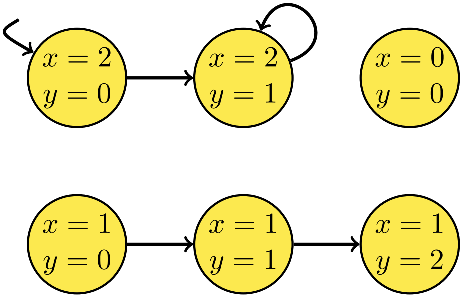
Predicate Abstraction: the Basic Idea

Concrete states over variables x, y:



Predicate Abstraction: the Basic Idea

Concrete states over variables x, y:

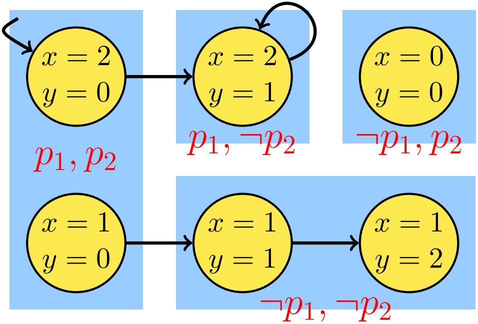


Predicates:

$$p_1 \iff x > y$$
$$p_2 \iff y = 0$$

Predicate Abstraction: the Basic Idea

Concrete states over variables x, y:

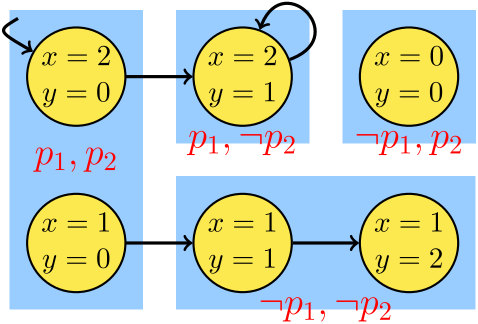


Predicates:

$$p_1 \iff x > y$$
$$p_2 \iff y = 0$$

Predicate Abstraction: the Basic Idea

Concrete states over variables x, y:



Predicates:

$$p_1 \iff x > y$$
$$p_2 \iff y = 0$$

Abstract Transitions?

Existential Abstraction[1]

Definition (Existential Abstraction)

A model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{T})$ is an *existential abstraction* of $M = (S, S_0, T)$ with respect to $\alpha : S \to \hat{S}$ iff

- $\exists s \in S_0 . \, \alpha(s) = \hat{s} \quad \Rightarrow \quad \hat{s} \in \hat{S}_0$ and
- $\exists (s, s') \in T . \, \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}' \quad \Rightarrow \quad (\hat{s}, \hat{s}') \in \hat{T}$.

[1] Clarke, Grumberg, Long: *Model Checking and Abstraction*, ACM TOPLAS, 1994

Minimal Existential Abstractions

There are obviously many choices for an existential abstraction for a given α.

Definition (Minimal Existential Abstraction)

A model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{T})$ is the *minimal existential abstraction* of $M = (S, S_0, T)$ with respect to $\alpha : S \to \hat{S}$ iff

- $\exists s \in S_0 . \alpha(s) = \hat{s} \quad \iff \quad \hat{s} \in \hat{S}_0$ and
- $\exists (s, s') \in T . \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}' \quad \iff \quad (\hat{s}, \hat{s}') \in \hat{T}$.

This is the most precise existential abstraction.

Existential Abstraction

We write $\alpha(\pi)$ for the abstraction of a path $\pi = s_0, s_1, \ldots$:

$$\alpha(\pi) \;=\; \alpha(s_0), \alpha(s_1), \ldots$$

Existential Abstraction

We write $\alpha(\pi)$ for the abstraction of a path $\pi = s_0, s_1, \ldots$:

$$\alpha(\pi) \; = \; \alpha(s_0), \alpha(s_1), \ldots$$

Lemma

Let $\hat{M}$ be an existential abstraction of M. The abstraction of every path (trace) π in M is a path (trace) in $\hat{M}$.

$$\pi \in M \quad \Rightarrow \quad \alpha(\pi) \in \hat{M}$$

Proof by induction.
We say that $\hat{M}$ overapproximates M.

Abstracting Properties

Reminder: we are using

- a set of atomic propositions (predicates) A, and
- a state-labelling function $L : S \to \mathscr{P}(A)$

in order to define the meaning of propositions in our properties.

Abstracting Properties

We define an abstract version of it as follows:

- First of all, the negations are pushed into the atomic propositions.
  E.g., we will have

$$x = 0 \quad \in A$$

and

$$x \neq 0 \quad \in A$$

Abstracting Properties

- An abstract state $\hat{s}$ is labelled with $a \in A$ iff all of the corresponding concrete states are labelled with a.

$$a \in \hat{L}(\hat{s}) \quad \Longleftrightarrow \quad \forall s | \alpha(s) = \hat{s}.\, a \in L(s)$$

- This also means that an abstract state may have neither the label $x = 0$ nor the label $x \neq 0$ – this may happen if it concretizes to concrete states with different labels!

Conservative Abstraction

The keystone is that existential abstraction is conservative for certain properties:

Theorem (Clarke/Grumberg/Long 1994)

Let ϕ be a $\forall$CTL formula where all negations are pushed into the atomic propositions, and let $\hat{M}$ be an existential abstraction of M. If ϕ holds on $\hat{M}$, then it also holds on M.*

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

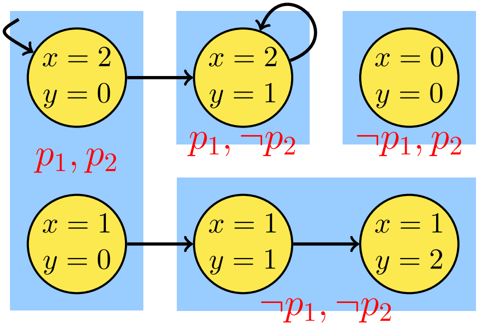
We say that an existential abstraction is conservative for $\forall$CTL* properties. The same result can be obtained for LTL properties.

The proof uses the lemma and is by induction on the structure of ϕ. The converse usually does not hold.

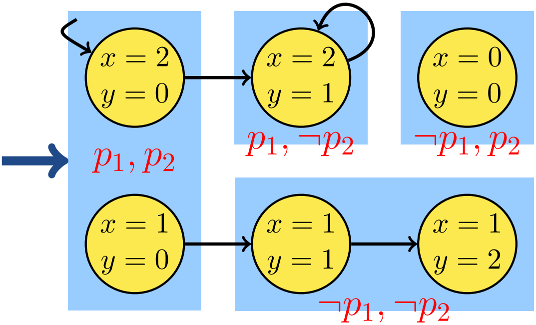
Conservative Abstraction

We hope: computing $\hat{M}$ and checking $\hat{M} \models \phi$ is easier than checking $M \models \phi$.

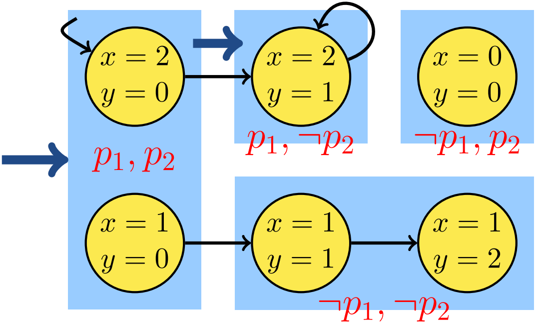
Back to the Example



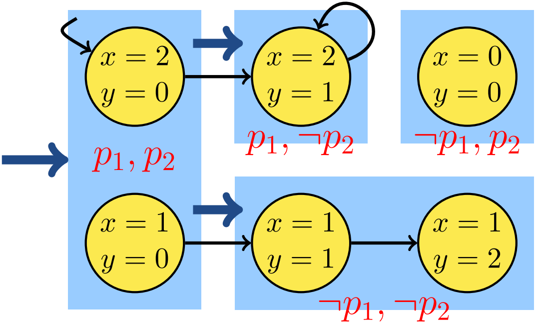
Back to the Example



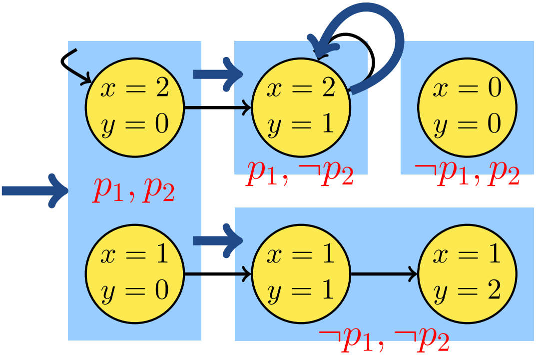
Back to the Example



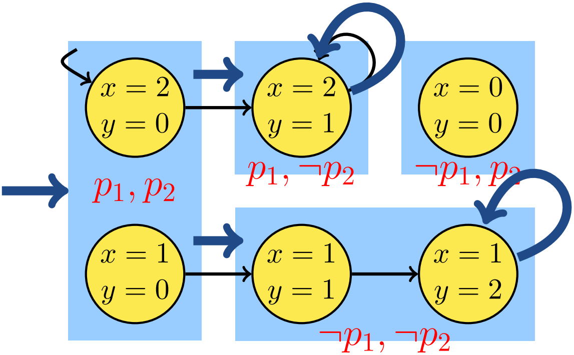
Back to the Example



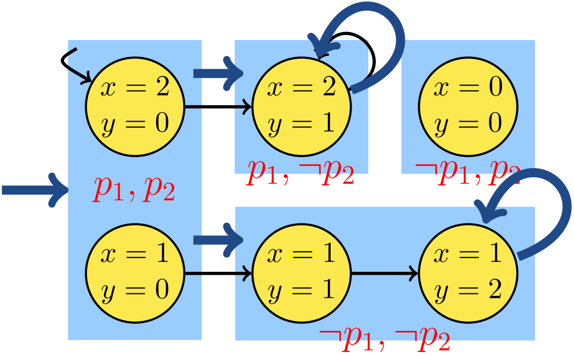
Back to the Example



Back to the Example



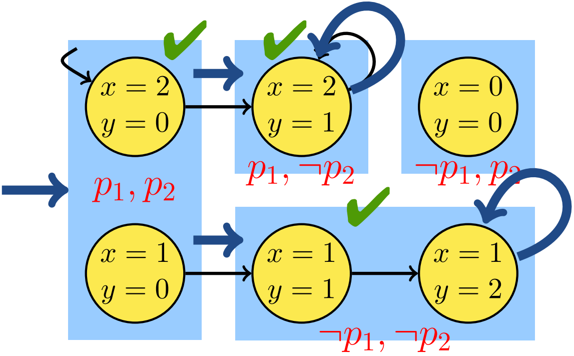
Let's try a Property



Property:

$$x > y \lor y \neq 0 \quad \iff \quad p_1 \lor \neg p_2$$

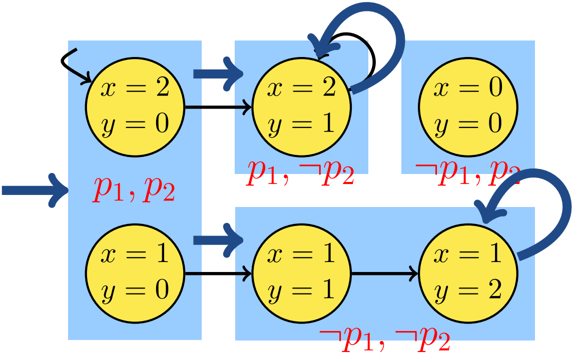
Let's try a Property



Property:

$$x > y \lor y \neq 0 \quad \iff \quad p_1 \lor \neg p_2$$

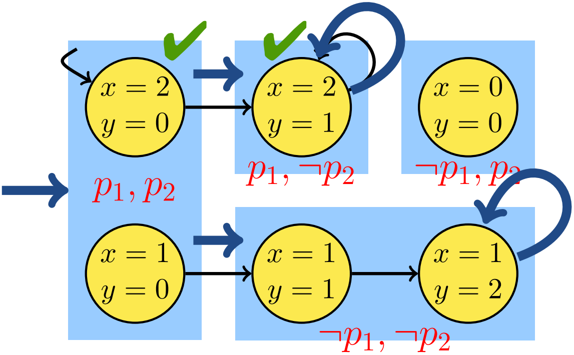
Another Property



Property:

$x > y \quad \Longleftrightarrow \quad p_1$

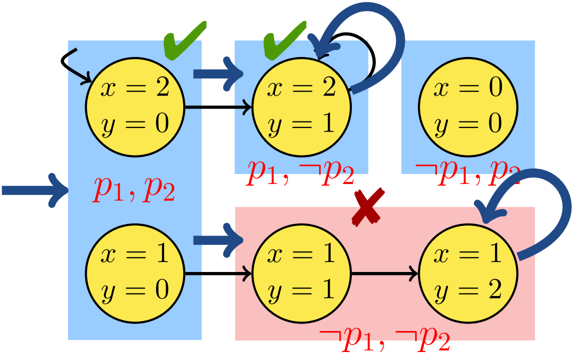
Another Property



Property:

$x > y \iff p_1$

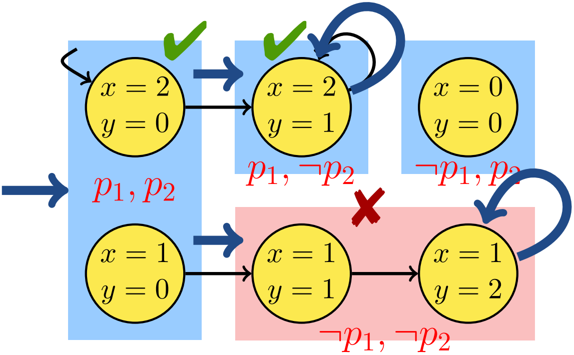
Another Property



Property:

$x > y \quad \Longleftrightarrow \quad p_1$

Another Property

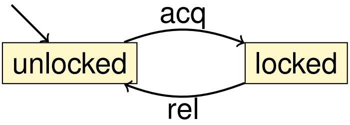


Property:

$x > y \iff p_1$

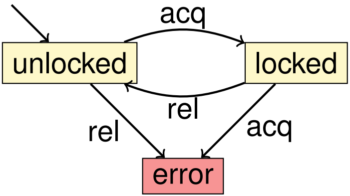
But: the counterexample is spurious

SLIC Example




```
state {
  enum {Locked, Unlocked}
     s = Unlocked;
}

KeAcquireSpinLock.entry {
  if (s==Locked) abort;
  else s = Locked;
}

KeReleaseSpinLock.entry {
  if (s==Unlocked) abort;
  else s = Unlocked;
}
```

SLIC Example




```
state {
  enum {Locked,Unlocked}
     s = Unlocked;
}

KeAcquireSpinLock.entry {
   if (s==Locked) abort;
   else s = Locked;
}

KeReleaseSpinLock.entry {
   if (s==Unlocked) abort;
   else s = Unlocked;
}
```

Refinement Example

```
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while(nPackets != nPacketsOld);

KeReleaseSpinLock();
```

Refinement Example

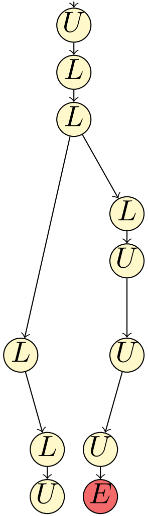
Does this code obey the locking rule?

```
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while(nPackets != nPacketsOld);

KeReleaseSpinLock();
```

Refinement Example

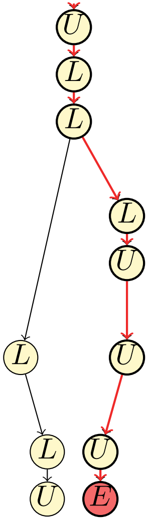
```
do {
    KeAcquireSpinLock();

    if (*) {

        KeReleaseSpinLock();

    }
} while(*);

KeReleaseSpinLock();
```

Refinement Example



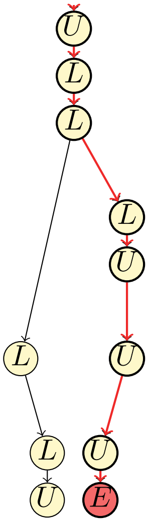

```
do {
    KeAcquireSpinLock();

    if (*) {

        KeReleaseSpinLock();

    }
} while(*);

KeReleaseSpinLock();
```

Refinement Example




```
do {
    KeAcquireSpinLock();

    if (*) {

        KeReleaseSpinLock();

    }
} while(*);

KeReleaseSpinLock();
```

Refinement Example



do {

KeAcquireSpinLock ();

if (∗) {

KeReleaseSpinLock ();

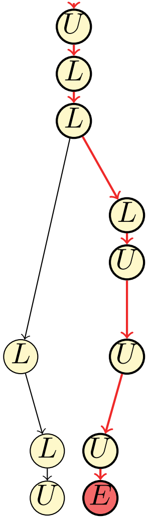
}

} **while**(∗);

KeReleaseSpinLock ();

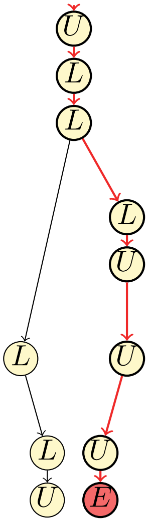
Is this path concretizable?

Refinement Example




```
do {
    KeAcquireSpinLock ();
    nPacketsOld = nPackets;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock ();
        nPackets++;
    }
} while(nPackets != nPacketsOld);

KeReleaseSpinLock ();
```

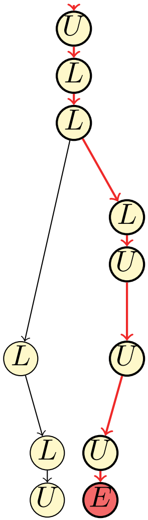
Refinement Example




```
do {
        KeAcquireSpinLock ();
        nPacketsOld = nPackets;
        if (request) {
                request = request−>Next;
                KeReleaseSpinLock ();
                nPackets++;
        }
} while(nPackets != nPacketsOld);

KeReleaseSpinLock ();
```

This path is spurious!

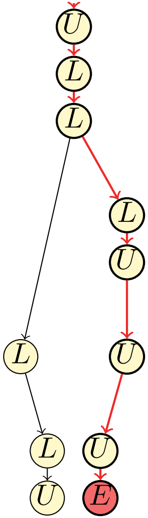
Refinement Example




```
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while(nPackets != nPacketsOld);

KeReleaseSpinLock();
```

Let's add the predicate
nPacketsOld==nPackets

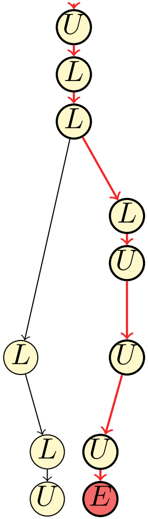
Refinement Example




```
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;          b=true;
    if(request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while(nPackets != nPacketsOld);

KeReleaseSpinLock();
```

Let's add the predicate nPacketsOld==nPackets

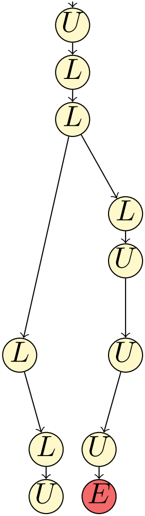
Refinement Example




```
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;          b=true;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;                  b=b?false:*;
    }
} while(nPackets != nPacketsOld);   !b

KeReleaseSpinLock();
```

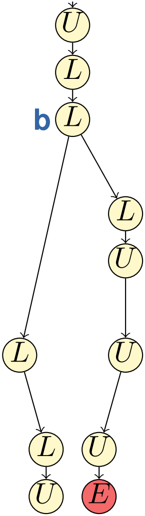
Let's add the predicate nPacketsOld==nPackets

Refinement Example



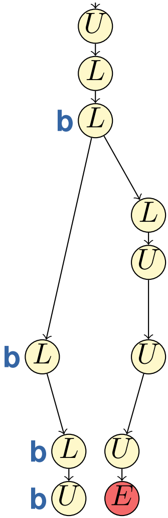

```
do {
    KeAcquireSpinLock();
    b=true;
    if (*) {

        KeReleaseSpinLock();
        b=b?false:*;
    }
} while( !b );

KeReleaseSpinLock();
```

Refinement Example



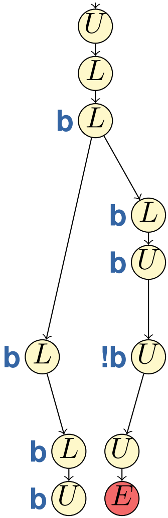

```
do {
      KeAcquireSpinLock ();
      b=true;
      if (∗)  {

            KeReleaseSpinLock ();
            b=b?false:∗;
      }
} while( !b );

KeReleaseSpinLock ();
```

Refinement Example



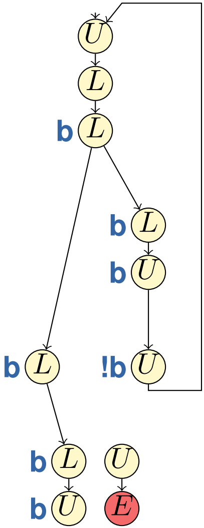

```
do {
    KeAcquireSpinLock ();
    b=true;
    if (*) {

        KeReleaseSpinLock ();
        b=b?false:*;
    }
} while( !b );

KeReleaseSpinLock ();
```

Refinement Example



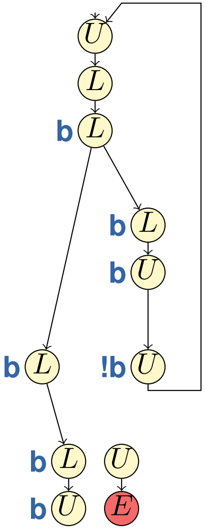

```
do {
    KeAcquireSpinLock ();
    b=true;
    if (*) {

        KeReleaseSpinLock ();
        b=b?false:*;
    }
} while( !b );

KeReleaseSpinLock ();
```

Refinement Example




```
do {
    KeAcquireSpinLock ();
    b=true;
    if (∗) {

        KeReleaseSpinLock ();
        b=b?false:∗;
    }
} while( !b );

KeReleaseSpinLock ();
```

Refinement Example



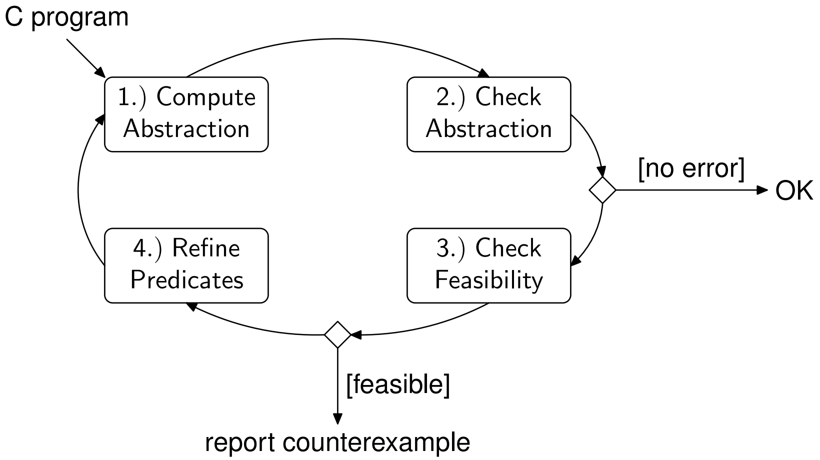

```
do {
    KeAcquireSpinLock();
    b=true;
    if (∗) {

        KeReleaseSpinLock();
        b=b?false:∗;
    }
} while( !b );

KeReleaseSpinLock();
```

The property holds!

Counterexample-guided Abstraction Refinement

- ”CEGAR”
- An iterative method to compute a sufficiently precise abstraction
- Initially applied in the context of hardware [Kurshan]

CEGAR Overview



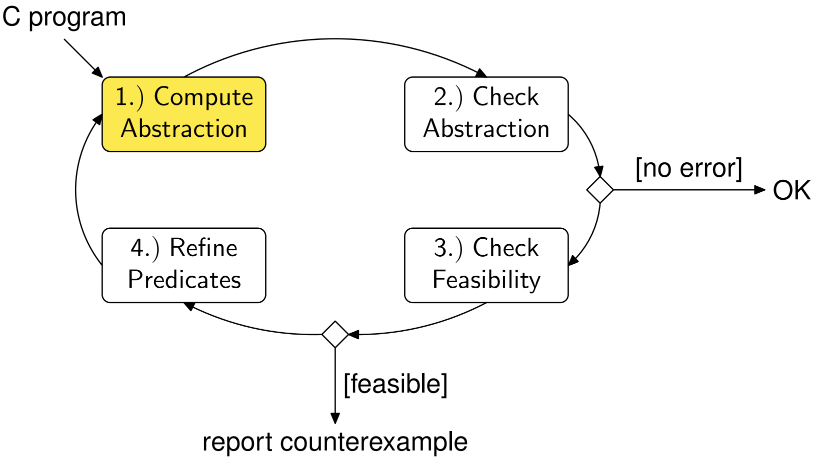
Counterexample-guided Abstraction Refinement

Claims:

1. This never returns a false error.
2. This never returns a false proof.

3. This is complete for finite-state models.
4. But: no termination guarantee in case of infinite-state systems

Computing Existential Abstractions of Programs



Computing Existential Abstractions of Programs

```
int main() {
  int i;

  i=0;

  while(even(i))
    i++;
}
```

C Program

Computing Existential Abstractions of Programs

```
int main() {
  int i;

  i=0;

  while(even(i))
    i++;
}
```

+

$$p_1 \iff i = 0$$
$$p_2 \iff even(i)$$

C Program   Predicates

Computing Existential Abstractions of Programs

```
int main() {
  int i;

  i=0;

  while(even(i))
    i++;
}
```

C Program

+ $p_1 \iff i = 0$
$p_2 \iff even(i)$

Predicates

→

```
void main() {
  bool p1, p2;

  p1=TRUE;
  p2=TRUE;

  while(p2) {
    p1= p1 ? FALSE : *;
    p2= !p2;
  }
}
```

Boolean Program

Computing Existential Abstractions of Programs

```
int main() {
  int i;

  i=0;

  while(even(i))
    i++;
}
```

+ $p_1 \iff i = 0$
$p_2 \iff even(i)$

→

```
void main() {
  bool p1, p2;

  p1=TRUE;
  p2=TRUE;

  while(p2) {
    p1= p1 ? FALSE : *;
    p2= !p2;
  }
}
```

C Program | Predicates | Boolean Program

Minimal?

Predicate Images

Reminder:

$$Image(X) = \{s' \in S \mid \exists s \in X.\, T(s, s')\}$$

We need

$$\widehat{Image}(\hat{X}) = \{\hat{s}' \in \hat{S} \mid \exists \hat{s} \in \hat{X}.\, \hat{T}(\hat{s}, \hat{s}')\}$$

$\widehat{Image}(\hat{X})$ is equivalent to

$$\{\hat{s}, \hat{s}' \in \hat{S}^2 \mid \exists s, s' \in S^2.\, \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}' \wedge T(s, s')\}$$

This is called the predicate image of T.

Enumeration

- Let's take existential abstraction seriously
- Basic idea: with n predicates, there are $2^n \cdot 2^n$ possible abstract transitions
- Let's just check them!

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

```
i++;
```

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

```
i++;
```

→

T

$$i' = i + 1$$

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

```
i++;
```

→

T

$$i' = i + 1$$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

```
i++;
```

→

T

$$i' = i + 1$$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

$\xrightarrow{?}$

p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Enumeration: Example

UNIVERSITY OF OXFORD

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

```
i++;
```

→

T

$$i' = i + 1$$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

$\xrightarrow{?}$

p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Query to Solver

$$i \neq 1 \wedge i \neq 2 \wedge \overline{\text{even}(i)} \wedge$$
$$i' = i + 1 \wedge$$
$$i' \neq 1 \wedge i' \neq 2 \wedge \overline{\text{even}(i')}$$

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block		T
i++;	→	$i' = i + 1$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

—✗→

p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Query to Solver

$$i \neq 1 \wedge i \neq 2 \wedge \overline{\text{even}(i)} \wedge$$
$$i' = i + 1 \wedge$$
$$i' \neq 1 \wedge i' \neq 2 \wedge \overline{\text{even}(i')}$$

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

```
i++;
```

→

T

$$i' = i + 1$$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

→ ?

p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Query to Solver

$$i \neq 1 \wedge i \neq 2 \wedge \overline{\text{even}(i)} \wedge$$
$$i' = i + 1 \wedge$$
$$i' \neq 1 \wedge i' \neq 2 \wedge \text{even}(i')$$

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

```
i++;
```

→

T

$$i' = i + 1$$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

✔ (000 → 001)

p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Query to Solver

$$i \neq 1 \wedge i \neq 2 \wedge \overline{\text{even}(i)} \wedge$$
$$i' = i + 1 \wedge$$
$$i' \neq 1 \wedge i' \neq 2 \wedge \text{even}(i')$$

Enumeration: Example

Predicates

$$p_1 \iff i = 1$$
$$p_2 \iff i = 2$$
$$p_3 \iff \text{even}(i)$$

Basic Block

```
i++;
```

→

T

$$i' = i + 1$$

p_1	p_2	p_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

p'_1	p'_2	p'_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Query to Solver

...and so on ...

Predicate Images

- ✘ Computing the minimal existential abstraction can be way too slow

- Use an over-approximation instead
  - ✔ Fast(er) to compute
  - ✘ But has additional transitions

- Examples:
  - Cartesian approximation (SLAM)
  - FastAbs (SLAM)
  - Lazy abstraction (Blast)
  - Predicate partitioning (VCEGAR)

Using *wp* to generate Boolean Programs

- given a set of predicates $\mathcal{P}$, our aim is to replace the assignments in our program by assignments to boolean variables (corresponding to the predicates)

- given a statement s and a boolean b corresponding to a predicate p
  - if $wp(s, p)$ is *true* before s, then b should be assigned *true*
  - if $wp(s, \neg p)$ is *true* before s, then b should be assigned *false*
  - otherwise, b should be set to *unknown*

- but the exact *wp* may not be expressible as conjunction of (some of) our predicates (or their negations)

- compute the weakest cubes P_t and P_f over $\mathcal{P}$ such that $P_t \rightarrow wp(s, p)$ and $P_f \rightarrow wp(s, \neg p)$

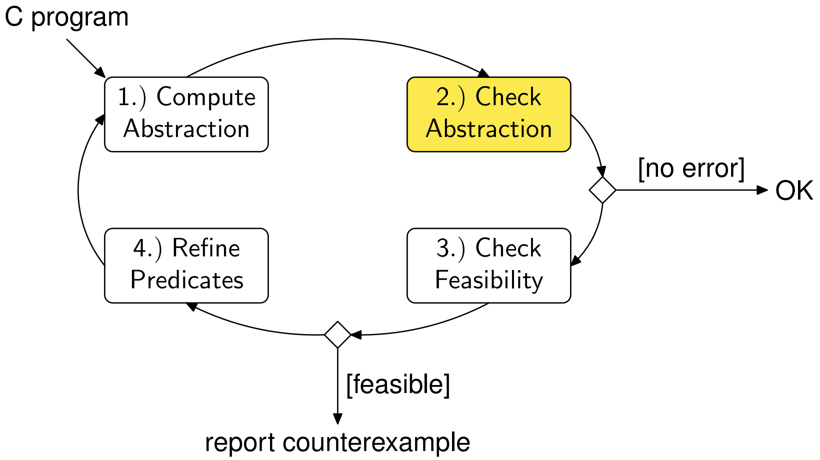
Using *wp* to generate Boolean Programs

Here's how the statement can then be modelled in Boolean Program:

```
if (Pt)  b := true
else if (Pf)  b := false
else  b := *
```

Exercise: Model the statement `x := y` as a statement in a Boolean Program using variables b_1, b_2, b_3 corresponding respectively to the predicates $x > 5$, $x < 5$, and $y = 5$.

Checking the Abstract Model

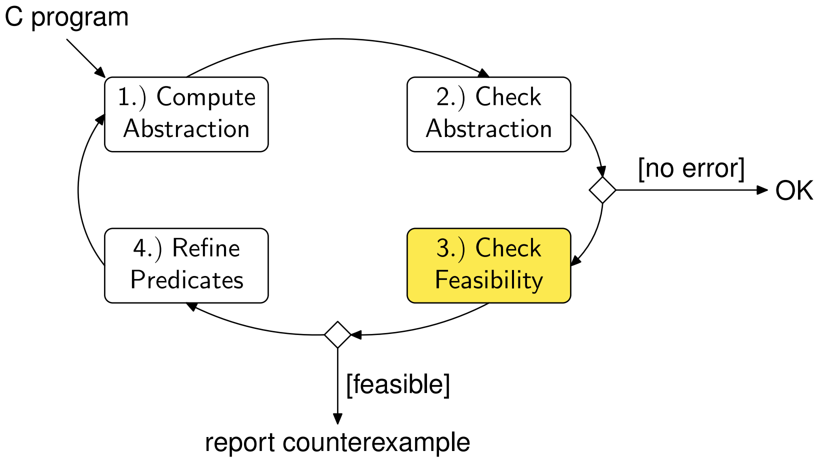


Checking the Abstract Model

- No more integers!
- But:
  - All control flow constructs, including function calls
  - (more) non-determinism

✔ BDD-based model checking now scales

Simulating the Counterexample



Example Simulation

```
int main() {
    int x, y;
    y=1;
    x=1;
    if (y>x)
        y--;
    else
        y++;
    assert(y>x);
}
```

Predicate:

y>x

```
main() {
    bool b0; // y>x
    b0=*;
    b0=*;
    if (b0)
        b0=*;
    else
        b0=*;
    assert(b0);
}
```

Example Simulation

```
int main() {
    int x, y;
    y=1;
    x=1;
    if (y>x)
        y−−;
    else
        y++;
    assert(y>x);
}
```

Predicate:
y>x

```
main() {
    bool b0; // y>x
    b0=*;
    b0=*;
    if (b0)
        b0=*;
    else
        b0=*;
    assert(b0);
}
```

Example Simulation

```
int main() {
    int x, y;
    y=1;
    x=1;
    if (y>x)
        y--;
    else
        y++;
    assert(y>x);
}
```

Example Simulation

```
int main() {
    int x, y;
    y=1;
    x=1;
    if (y>x)
        y−−;
    else
        y++;
    assert(y>x);
}
```

We now do a path test, so convert to SSA.

Example Simulation

int main() {

  int x, y;

  $y_1=1;$

  $x_1=1;$

  if $(y_1>x_1)$

    $y_2=y_1-1;$

  else

    y++;

  assert$(y_2>x_1)$;

}

Example Simulation

int main() {

  int x, y;

  y_1=1;

  x_1=1;

  if (y_1>x_1)

    y_2=y_1−1;

  else

    y++;

  assert(y_2>x_1);

}

$$
\begin{aligned}
& y_1 = 1 \quad \wedge \\
& x_1 = 1 \quad \wedge \\
& y_1 > x_1 \quad \wedge \\
& y_2 = y_1 - 1 \quad \wedge
\end{aligned}
$$

$$\neg(y_2 > x_0)$$

Example Simulation

int main() {

int x, y;

y_1=1;

x_1=1;

if ($y_1 > x_1$)

y_2=$y_1 - 1$;

else

y++;

assert($y_2 > x_1$);

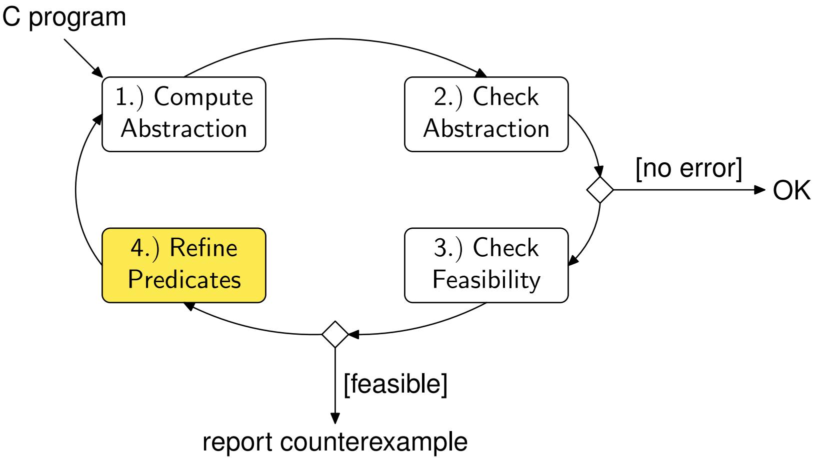
}

$$y_1 = 1 \quad \wedge$$
$$x_1 = 1 \quad \wedge$$
$$y_1 > x_1 \quad \wedge$$
$$y_2 = y_1 - 1 \quad \wedge$$

$$\neg(y_2 > x_0)$$

This is UNSAT, so $\hat{\pi}$ is spurious.

Refining the Abstraction



Manual Proof!

```
int main() {
    int x, y;
    y=1;

    x=1;

    if (y>x)
        y--;
    else

        y++;

    assert(y>x);
}
```

Manual Proof!

```
int main() {
    int x, y;
    y=1;
    {y = 1}
    x=1;

    if (y>x)
        y−−;
    else

        y++;

    assert(y>x);
}
```

Manual Proof!

```
int main() {
    int x, y;
    y=1;
    {y = 1}
    x=1;
    {x = 1 ∧ y = 1}
    if (y>x)
        y−−;
    else


        y++;


    assert(y>x);
}
```

Manual Proof!

```
int main() {
    int x, y;
    y=1;
```

$\{y = 1\}$

```
    x=1;
```

$\{x = 1 \wedge y = 1\}$

```
    if (y>x)
        y−−;
    else
```

$\{x = 1 \wedge y = 1 \wedge \neg y > x\}$

```
        y++;


    assert(y>x);
}
```

Manual Proof!

```
int main() {
    int x, y;
    y=1;
    {y = 1}
    x=1;
    {x = 1 ∧ y = 1}
    if (y>x)
        y−−;
    else
        {x = 1 ∧ y = 1 ∧ ¬y > x}
        y++;
    {x = 1 ∧ y = 2 ∧ y > x}
    assert(y>x);
}
```

This proof uses strongest post-conditions

An Alternative Proof

```
int main() {
    int x, y;
    y=1;

    x=1;

    if (y>x)
        y--;
    else

        y++;

    assert(y>x);
}
```

An Alternative Proof

```
int main() {
    int x, y;
    y=1;

    x=1;

    if (y>x)
        y−−;
    else

        y++;
    {y > x}
    assert(y>x);
}
```

An Alternative Proof

```
int main() {
    int x, y;
    y=1;

    x=1;

    if (y>x)
        y−−;
    else
        {y + 1 > x}
        y++;
    {y > x}
    assert(y>x);
}
```

An Alternative Proof

```
int main() {
    int x, y;
    y=1;


    x=1;
```
$\{\neg y > x \Rightarrow y + 1 > x\}$
```
    if (y>x)
        y−−;
    else
```
$\{y + 1 > x\}$
```
        y++;
```
$\{y > x\}$
```
    assert(y>x);
}
```

An Alternative Proof

```
int main() {
    int x, y;
    y=1;
```

$\{\neg y > 1 \Rightarrow y+1 > 1\}$

```
    x=1;
```

$\{\neg y > x \Rightarrow y+1 > x\}$

```
    if (y>x)
        y−−;
    else
```

$\{y+1 > x\}$

```
        y++;
```

$\{y > x\}$

```
    assert(y>x);
}
```

An Alternative Proof

```
int main() {
    int x, y;

    y=1;
```
$\{\neg y > 1 \Rightarrow y + 1 > 1\}$
```
    x=1;
```
$\{\neg y > x \Rightarrow y + 1 > x\}$
```
    if (y>x)
        y−−;
    else
```
$\{y + 1 > x\}$
```
        y++;
```
$\{y > x\}$
```
    assert(y>x);
}
```

We are using weakest pre-conditions here

$$wp(x\texttt{:=}E, P) = P[x/E]$$

$$wp(S\texttt{;}T, Q) = wp(S, wp(T, Q))$$

$$wp(\texttt{if}(c)\ A\ \texttt{else}\ B, P) = (B \Rightarrow wp(A, P)) \wedge (\neg B \Rightarrow wp(B, P))$$

The proof for the ”true” branch is missing

Refinement Algorithms

Using WP

1. Start with failed guard G
2. Compute $wp(G)$ along the path

Using SP

1. Start at beginning
2. Compute $sp(\ldots)$ along the path

- Both methods eliminate the trace
- Advantages/disadvantages?

Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

$$x_1 = 10 \quad \wedge \quad y_1 = x_1 + 10 \quad \wedge \quad y_2 = y_1 + 10 \quad \wedge \quad y_2 \neq 30$$

Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

$$x_1 = 10 \quad \wedge \quad y_1 = x_1 + 10 \quad \wedge \quad y_2 = y_1 + 10 \quad \wedge \quad y_2 \neq 30$$

$\Rightarrow x_1 = 10$

Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

$$x_1 = 10 \quad \wedge \quad y_1 = x_1 + 10 \quad \wedge \quad y_2 = y_1 + 10 \quad \wedge \quad y_2 \neq 30$$

$$\Rightarrow x_1 = 10 \qquad \Rightarrow y_1 = 20$$

Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

$$x_1 = 10 \quad \wedge \quad y_1 = x_1 + 10 \quad \wedge \quad y_2 = y_1 + 10 \quad \wedge \quad y_2 \neq 30$$

$$\Rightarrow x_1 = 10 \qquad \Rightarrow y_1 = 20 \qquad \Rightarrow y_2 = 30$$

Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

$$x_1 = 10 \quad \wedge \quad y_1 = x_1 + 10 \quad \wedge \quad y_2 = y_1 + 10 \quad \wedge \quad y_2 \neq 30$$

$$\Rightarrow x_1 = 10 \qquad \Rightarrow y_1 = 20 \qquad \Rightarrow y_2 = 30 \qquad \Rightarrow \text{false}$$

Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

$$\overbrace{x_1 = 10}^{A_1} \quad \wedge \quad \overbrace{y_1 = x_1 + 10}^{A_2} \quad \wedge \quad \overbrace{y_2 = y_1 + 10}^{A_3} \quad \wedge \quad \overbrace{y_2 \neq 30}^{A_4}$$

$\Rightarrow x_1 = 10$ $\quad \Rightarrow y_1 = 20$ $\quad \Rightarrow y_2 = 30$ $\quad \Rightarrow$ false

Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

$$\overbrace{x_1 = 10}^{A_1} \wedge \overbrace{y_1 = x_1 + 10}^{A_2} \wedge \overbrace{y_2 = y_1 + 10}^{A_3} \wedge \overbrace{y_2 \neq 30}^{A_4}$$

$$\Rightarrow \underbrace{x_1 = 10}_{A'_1} \quad \Rightarrow \underbrace{y_1 = 20}_{A'_2} \quad \Rightarrow \underbrace{y_2 = 30}_{A'_3} \quad \Rightarrow \underbrace{\text{false}}_{A'_4}$$

Predicate Refinement for Paths

For a path with n steps:

$$A_1 \;\vert\; A_2 \;\vert\; A_3 \;\vert\; \cdots \;\vert\; A_n$$

$$\text{true} \;\Rightarrow A'_1 \;\Rightarrow A'_2 \;\Rightarrow A'_3 \;\Rightarrow A'_{n-1} \;\Rightarrow \text{false}$$

Predicate Refinement for Paths

For a path with n steps:

A_1	A_2	A_3	$\cdots$	A_n

true $\Rightarrow A'_1$ $\Rightarrow A'_2$ $\Rightarrow A'_3$ $\Rightarrow A'_{n-1}$ $\Rightarrow$ false

- Given $A_1, \ldots, A_n$ with $\bigwedge_i A_i =$ false
- $A'_0 =$ true and $A'_n =$ false
- $(A'_{i-1} \wedge A_i) \Rightarrow A'_i$ for $i \in \{1, \ldots, n\}$

Predicate Refinement for Paths

For a path with n steps:

A_1	A_2	A_3	$\cdots$	A_n

true $\Rightarrow A'_1$ $\Rightarrow A'_2$ $\Rightarrow A'_3$ $\Rightarrow A'_{n-1}$ $\Rightarrow$ false

- Given $A_1, \ldots, A_n$ with $\bigwedge_i A_i =$ false
- $A'_0 =$ true and $A'_n =$ false
- $(A'_{i-1} \wedge A_i) \Rightarrow A'_i$ for $i \in \{1, \ldots, n\}$
- Finally, $\mathit{Vars}(A'_i) \subseteq (\mathit{Vars}(A_1 \ldots A_i) \cap \mathit{Vars}(A_{i+1} \ldots A_n))$

Predicate Refinement for Paths

Special case $n = 2$:

- $A \wedge B =$ false
- $A \Rightarrow A'$
- $A' \wedge B =$ false
- $Vars(A') \subseteq (Vars(A) \cap Vars(B))$

Predicate Refinement for Paths

Special case $n = 2$:

- $A \wedge B =$ false
- $A \Rightarrow A'$
- $A' \wedge B =$ false
- $Vars(A') \subseteq (Vars(A) \cap Vars(B))$

W. Craig's Interpolation theorem (1957): such an A' exists for any first-order, inconsistent A and B.

Predicate Refinement with Craig Interpolants

- ✔ For propositional logic, a propositional Craig Interpolant can be extracted from a resolution proof ($\rightarrow$ SAT!) in linear time

- ✔ Interpolating solvers available for linear arithmetic over the reals and integer difference logic with uninterpreted functions

- ✘ Not possible for every fragment of FOL:

$$x = 2y \quad \text{and} \quad x = 2z + 1 \qquad \text{with } x, y, z \in \mathbb{Z}$$

Predicate Refinement with Craig Interpolants

✔ For propositional logic, a propositional Craig Interpolant can be extracted from a resolution proof ($\rightarrow$ SAT!) in linear time

✔ Interpolating solvers available for linear arithmetic over the reals and integer difference logic with uninterpreted functions

✘ Not possible for every fragment of FOL:

$$x = 2y \quad \text{and} \quad x = 2z + 1 \qquad \text{with } x, y, z \in \mathbb{Z}$$

The interpolant is "x is even"

Example

```
x = 0; y = 0;

while (*)
  x++; y++;

while (*)
  x--; y--;

assert (x >= 0 || y <= 0)
```

Set of predicates $\mathcal{P}$: $\{x \geq 0, y \leq 0, x \geq 1\}$

Abstract trace

In the following trace, $\langle b_1, b_2, b_3 \rangle$ denotes an abstract state corresponding to the boolean values b_1, b_2, b_3 for predicate $x \geq 0, y \leq 0, x \geq 1$ resp.

$$\langle 0,0,0 \rangle - - - (x := 0; y := 0;) - -- > \langle 1,1,0 \rangle$$
$$\langle 1,1,0 \rangle - - - (x + +; y + +;) - -- > \langle 1,0,1 \rangle$$
$$\langle 1,0,1 \rangle - - - (x - -; y - -;) - -- > \langle 1,0,0 \rangle$$
$$\langle 1,0,0 \rangle - - - (x - -; y - -;) - -- > \langle 0,0,0 \rangle$$

The trace leads to a bad (assertion-violating) state: $\langle 0,0,0 \rangle$.

Feasibility check

You may collect the assignments and the constraints along the counterexample path, and check feasibility using a SAT solver (e.g. Z3)

```
(declare-const x0 Int)
(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)
(declare-const y0 Int)
(declare-const y1 Int)
(declare-const y2 Int)
(declare-const y3 Int)

(assert (and (= 0 x0) (= 0 y0)))
(assert (and (= x1 (+ x0 1)) (= y1 (+ y0 1))))
(assert (and (= x2 (- x1 1)) (= y2 (- y1 1))))
(assert (and (= x3 (- x2 1)) (= y3 (- y2 1))))
(assert (and (< x3 0) (> y3 0)))

(check-sat)
```

Z3 returns unsat showing that the counterexample is infeasible (exercise: use z3 to check that this is indeed the case)

Refinement

- one can obtain (sequence) interpolants from unsatisfiability proofs and use them as predicates (an example shown below in red)

```
true
(assert (and (= 0 x0) (= 0 y0)))
y0 <= 0
(assert (and (= x1 (+ x0 1)) (= y1 (+ y0 1))))
y1 <= 1
(assert (and (= x2 (- x1 1)) (= y2 (- y1 1))))
y2 <= 0
(assert (and (= x3 (- x2 1)) (= y3 (- y2 1))))
y3 <= 0
(assert (and (< x3 0) (> y3 0)))
false
```

- suppose we pick `y <= 1` as the fourth predicate in our set of predicates

Eliminates Spurious Counterexample

$\langle 0, 0, 0, 0 \rangle - - - (x := 0; y := 0;) - -- > \langle 1, 1, 0, 1 \rangle$
$\langle 1, 1, 0, 1 \rangle - - - (x + +; y + +;) - -- > \langle 1, 0, 1, 1 \rangle$
$\langle 1, 0, 1, 1 \rangle - - - (x - -; y - -;) - -- > \langle 1, 1, 0, 1 \rangle$
$\langle 1, 1, 0, 1 \rangle - - - (x - -; y - -;) - -- > \langle 0, 1, 0, 1 \rangle$

The same sequence of statements now lead to $\langle 0, 1, 0, 1 \rangle$ which is not an assertion-violating state

Goodness of predicates/refinement

- while adding `y <= 1` did eliminate the spurious counterexample that we had, it is not avery useful refinement
- one can get another (spurious) counterexample by going through two iterations of the increment-loop
- we can, again, eliminate that by adding the predicate `y <= 2`, but longer (spurious) counterexamples will keep coming
- the predicates `y <= 1` and `y <= 2` – they are good enough to eliminate the counterexample at hand, but too specific to eliminate other spurious counterexamples
- a more general predicate, e.g. `y <= x`, can help refine all spurious counterexamples at once (but obtaining such predicates may be a challenge)

Thank you!