Predicate Abstraction: A Tutorial

Predicate Abstraction

Daniel Kroening

May 28 2012

Outline

Introduction

Existential Abstraction

Predicate Abstraction for Software

Counterexample-Guided Abstraction Refinement

Computing Existential Abstractions of Programs

Checking the Abstract Model

Simulating the Counterexample

Refining the Abstraction

Model Checking with Predicate Abstraction

- A heavy-weight formal analysis technique
- Recent successes in software verification, e.g., SLAM at Microsoft
- The abstraction reduces the size of the model by removing irrelevant detail
- The abstract model is then small enough for an analysis with a BDD-based Model Checker
- Idea: only track predicates on data, and remove data variables from model
- Mostly works with control-flow dominated properties

Reminder Abstract Interpretation

Abstract Domain

Approximate representation of
sets of concrete values

$$S \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \hat{S}$$

Predicate Abstraction as Abstract Domain

- We are given a set of predicates over S, denoted by $\Pi_1, \ldots, \Pi_n$.

- An abstract state is a valuation of the predicates:

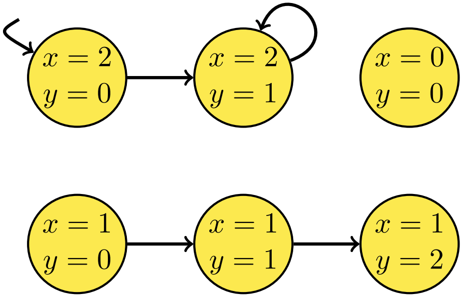
$$\hat{S} = \mathbb{B}^n$$

- The abstraction function:

$$\alpha(s) = \langle \Pi_1(s), \ldots, \Pi_n(s) \rangle$$

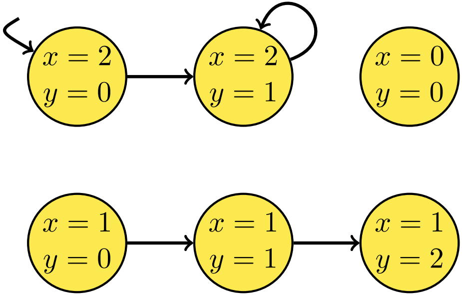
Predicate Abstraction: the Basic Idea

Concrete states over variables x, y:



Predicate Abstraction: the Basic Idea

Concrete states over variables x, y:

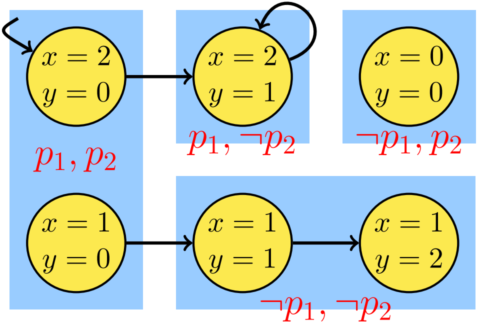


Predicates:

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

Predicate Abstraction: the Basic Idea

Concrete states over variables x, y:

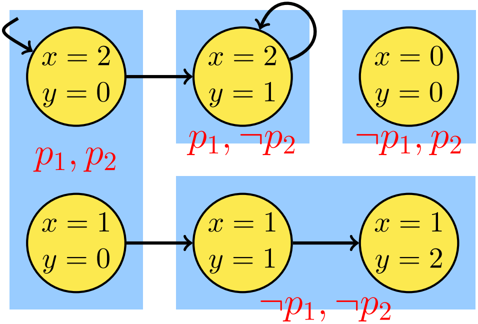


Predicates:

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

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

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

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.*

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

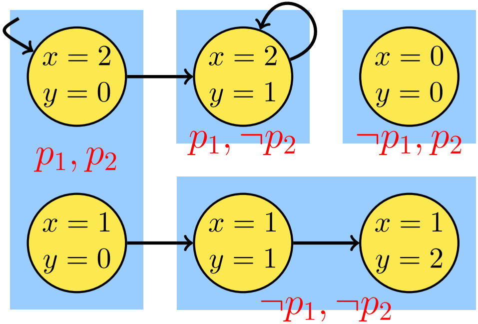
We say that an existential abstraction is conservative for $\forall$CTL* properties. The same result can be obtained for LTL properties.

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

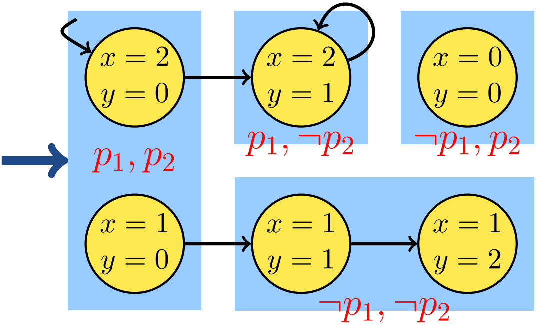
Conservative Abstraction

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

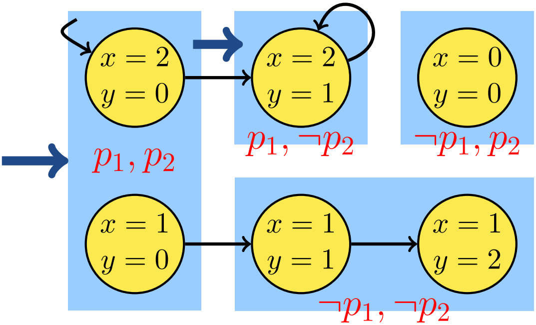
Back to the Example



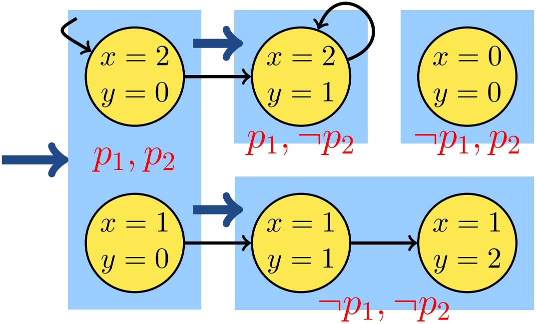
Back to the Example



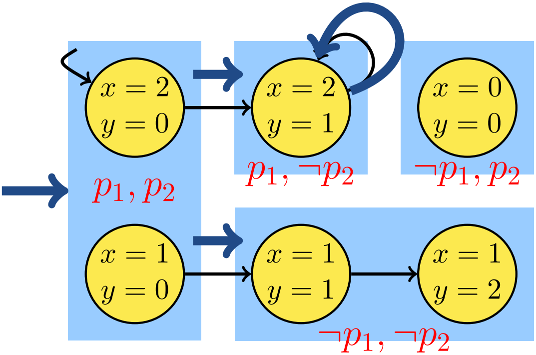
Back to the Example



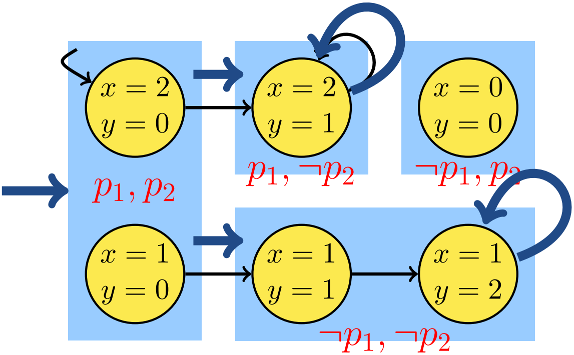
Back to the Example



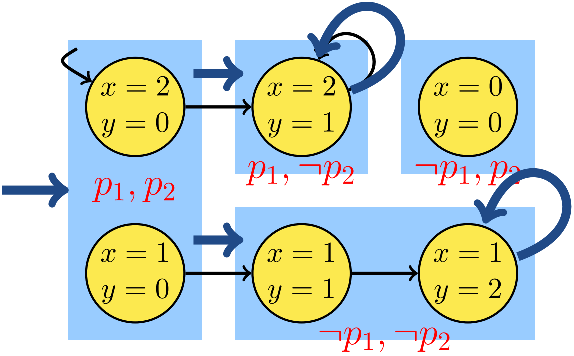
Back to the Example



Back to the Example



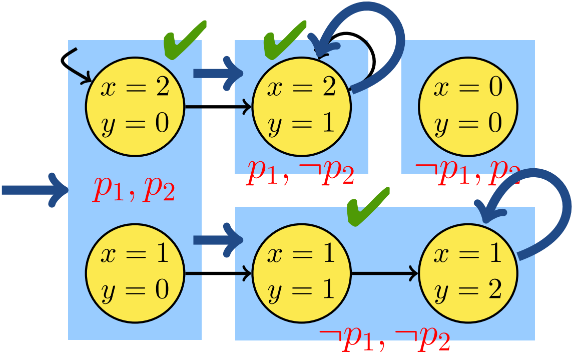
Let's try a Property



Property:

$$x > y \vee y \neq 0 \iff p_1 \vee \neg p_2$$

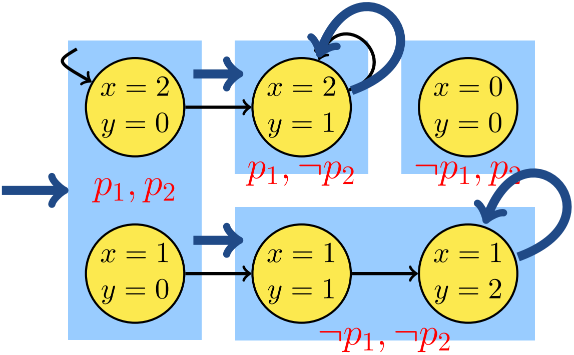
Let's try a Property



Property:

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

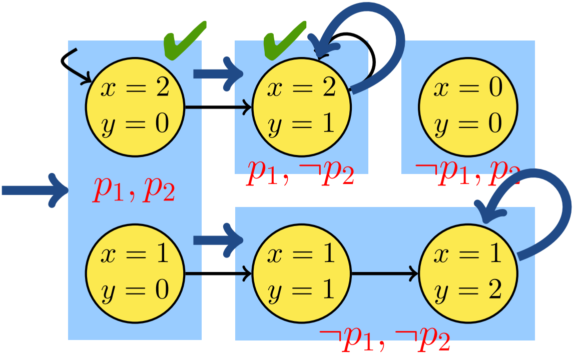
Another Property



Property:

$x > y \iff p_1$

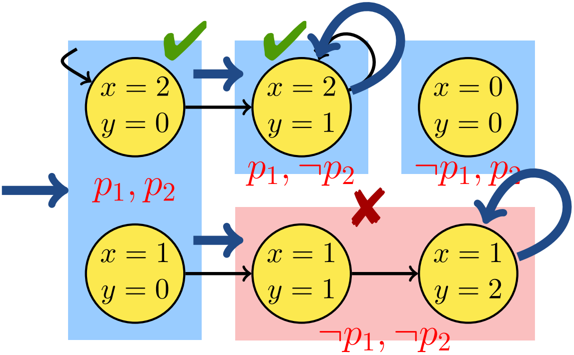
Another Property



Property:

$x > y \iff p_1$

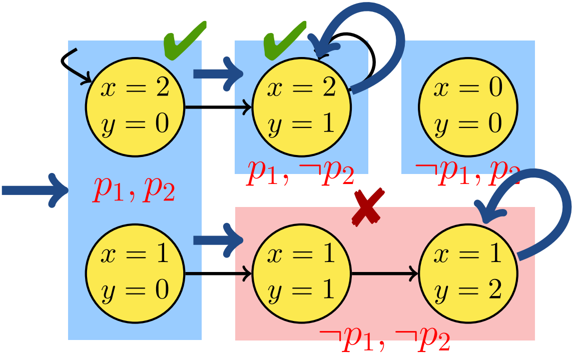
Another Property



Property:

$$x > y \quad \iff \quad p_1$$

Another Property



Property:

$x > y \iff p_1$

But: the counterexample is spurious

SLAM

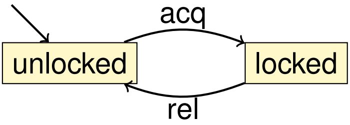
- Microsoft blames most Windows crashes on third party device drivers
- The Windows device driver API is quite complicated
- Drivers are low level C code
- SLAM: Tool to automatically check device drivers for certain errors
- SLAM is shipped with Device Driver Development Kit
- Full detail available at `http://research.microsoft.com/slam/`

SLIC

- Finite state language for defining properties
  - Monitors behavior of C code
  - Temporal safety properties (security automata)
  - familiar C syntax

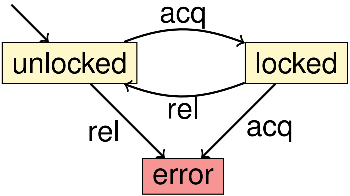
- Suitable for expressing control-dominated properties
  - e.g., proper sequence of events
  - can track data values

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

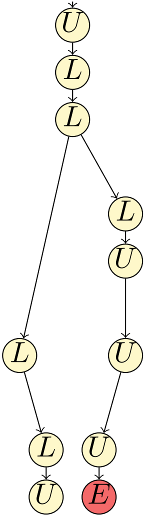
Does this code obey the locking rule?

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

KeReleaseSpinLock();
```

Refinement Example

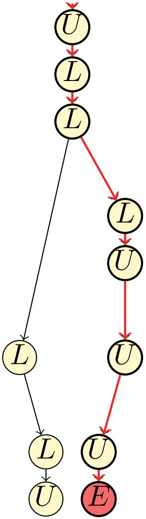
```
do {
    KeAcquireSpinLock();

    if (*) {

        KeReleaseSpinLock();

    }
} while(*);

KeReleaseSpinLock();
```

Refinement Example



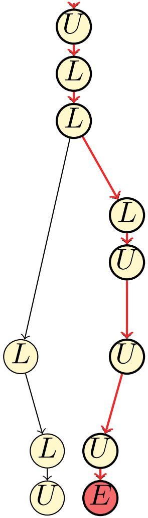

```
do {
    KeAcquireSpinLock();

    if (*) {

        KeReleaseSpinLock();

    }
} while(*);

KeReleaseSpinLock();
```

Refinement Example




```
do {
    KeAcquireSpinLock();

    if (*) {

        KeReleaseSpinLock();

    }
} while(*);

KeReleaseSpinLock();
```

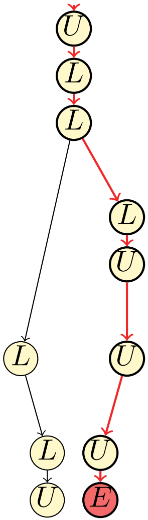
Refinement Example




```
do {
    KeAcquireSpinLock();

    if (*) {

        KeReleaseSpinLock();

    }
} while(*);

KeReleaseSpinLock();
```

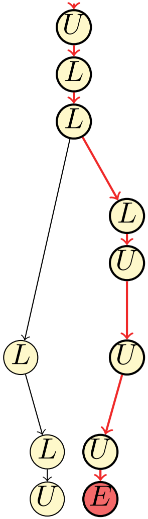
Is this path concretizable?

Refinement Example




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

KeReleaseSpinLock();
```

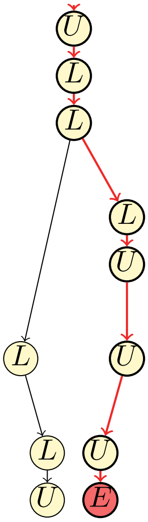
Refinement Example




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

KeReleaseSpinLock();
```

This path is spurious!

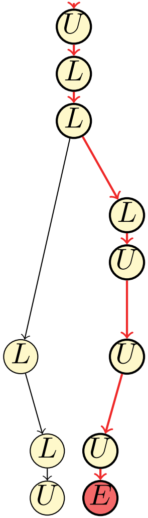
Refinement Example




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

KeReleaseSpinLock();
```

Let's add the predicate nPacketsOld==nPackets

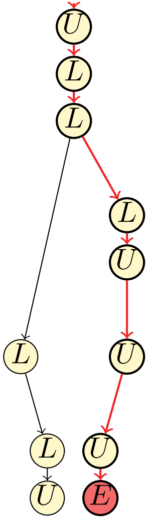
Refinement Example




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

KeReleaseSpinLock ();
```

Let's add the predicate nPacketsOld==nPackets

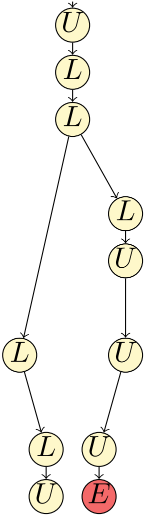
Refinement Example




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

KeReleaseSpinLock();
```

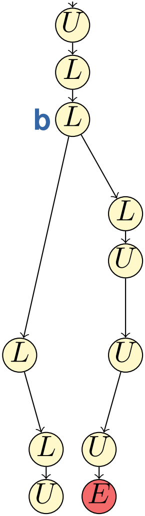
Let's add the predicate nPacketsOld==nPackets

Refinement Example



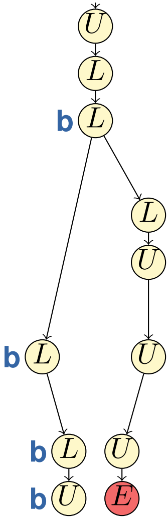

```
do {
    KeAcquireSpinLock ();
    b=true;
    if (∗) {

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

KeReleaseSpinLock ();
```

Refinement Example



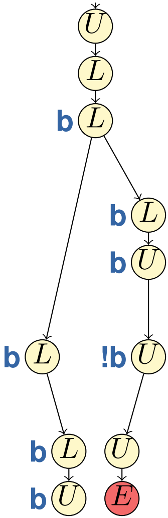

```
do {
    KeAcquireSpinLock();
    b=true;
    if (*) {

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

KeReleaseSpinLock();
```

Refinement Example



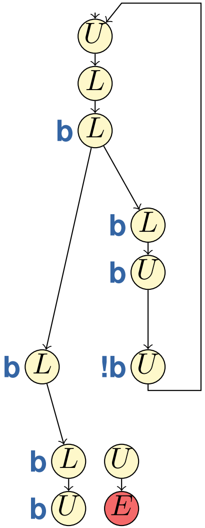

```
do {
    KeAcquireSpinLock();
    b=true;
    if (*) {

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

KeReleaseSpinLock();
```

Refinement Example



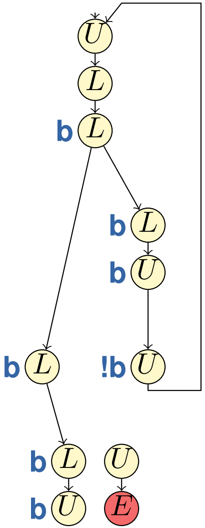

```
do {
    KeAcquireSpinLock();
    b=true;
    if (*) {

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

KeReleaseSpinLock();
```

Refinement Example




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

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

KeReleaseSpinLock();
```

Refinement Example



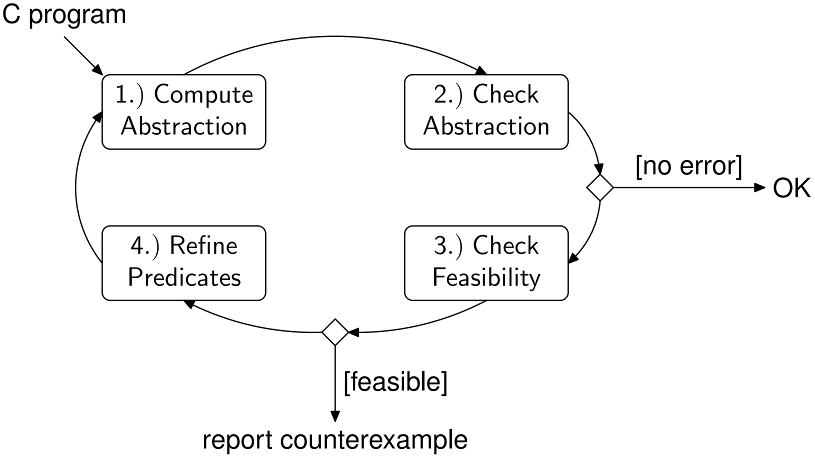

```
do {
    KeAcquireSpinLock ();
    b=true;
    if (∗) {

        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



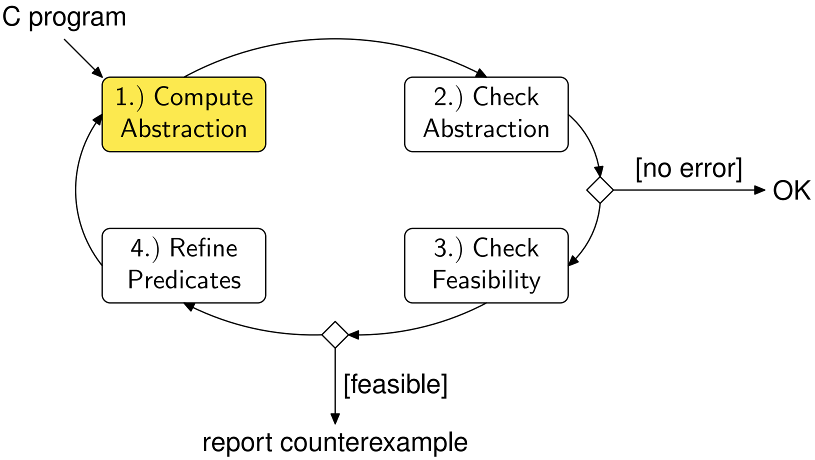
Counterexample-guided Abstraction Refinement

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++;
}
```

+

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

→

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

  p1=TRUE;
  p2=TRUE;

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

C Program | Predicates | Boolean Program

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

Minimal?

Predicate Images

Reminder:

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

We need

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

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

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

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

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

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

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

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

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

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

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

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 ...

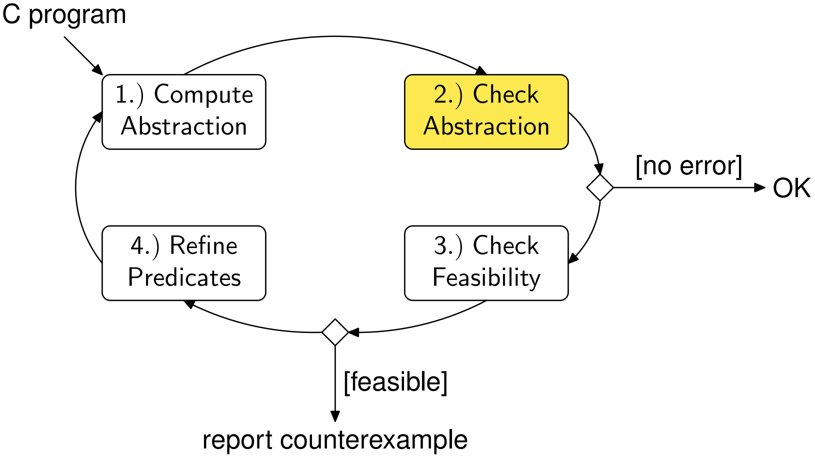
Predicate Images

✘ 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)

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

Finite-State Model Checkers: SMV

① Variables

```
VAR b0_argc_ge_1: boolean;                -- argc >= 1
VAR b1_argc_le_2147483646: boolean;       -- argc <= 2147483646
VAR b2: boolean;                          -- argv[argc] == NULL
VAR b3_nmemb_ge_r: boolean;               -- nmemb >= r
VAR b4: boolean;                          -- p1 == &array[0]
VAR b5_i_ge_8: boolean;                   -- i >= 8
VAR b6_i_ge_s: boolean;                   -- i >= s
VAR b7: boolean;                          -- 1 + i >= 8
VAR b8: boolean;                          -- 1 + i >= s
VAR b9_s_gt_0: boolean;                   -- s > 0
VAR b10_s_gt_1: boolean;                  -- s > 1
...
```

Finite-State Model Checkers: SMV

② Control Flow

```
-- program counter: 56 is the "terminating" PC
VAR PC: 0..56;
ASSIGN init(PC):=0; -- initial PC

ASSIGN next(PC):=case
     PC=0: 1; -- other
     PC=1: 2; -- other
     . . .
     PC=19: case  -- goto (with guard)
       guard19: 26;
       1: 20;
     esac;
     . . .
```

Finite-State Model Checkers: SMV

③ Data

```
TRANS (PC=0) -> next(b0_argc_ge_1)=b0_argc_ge_1
              & next(b1_argc_le_213646)=b1_argc_le_21646
              & next(b2)=b2
              & (!b30 | b36)
              & (!b17 | !b30 | b42)
              & (!b30 | !b42 | b48)
              & (!b17 | !b30 | !b42 | b54)
              & (!b54 | b60)

TRANS (PC=1) -> next(b0_argc_ge_1)=b0_argc_ge_1
              & next(b1_argc_le_214646)=b1_argc_le_214746
              & next(b2)=b2
              & next(b3_nmemb_ge_r)=b3_nmemb_ge_r
              & next(b4)=b4
              & next(b5_i_ge_8)=b5_i_ge_8
              & next(b6_i_ge_s)=b6_i_ge_s
              ...
```

Finite-State Model Checkers: SMV

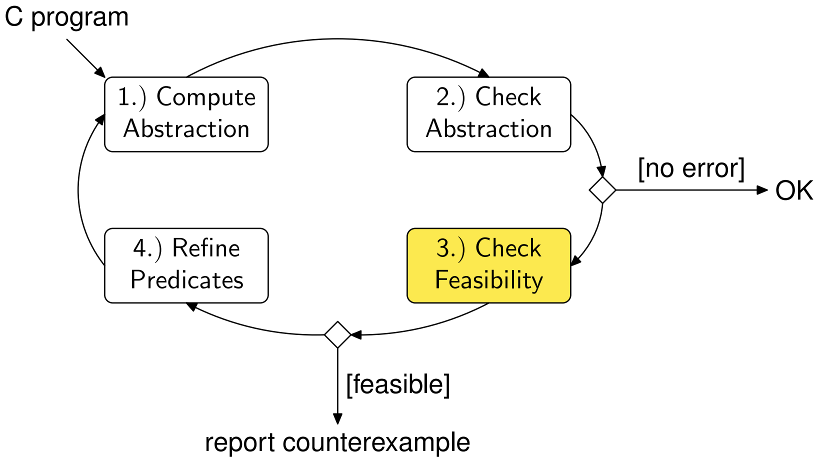
④ Property

```
-- the specification

-- file main.c line 20 column 12
-- function c::very_buggy_function
SPEC AG ((PC=51) -> !b23)
```

Finite-State Model Checkers: SMV

- If the property holds, we can terminate
- If the property fails, SMV generates a counterexample with an assignment for all variables, including the PC

Simulating the Counterexample



Lazy Abstraction

- The progress guarantee is only valid if the minimal existential abstraction is used.

- Thus, distinguish spurious transitions from spurious prefixes.

- Refine spurious transitions separately to obtain minimal existential abstraction
- SLAM: `Constrain`

Lazy Abstraction

- One more observation:
  each iteration only causes only minor changes in the abstract model

- Thus, use "incremental Model Checker", which retains the set of reachable states between iterations (BLAST)

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;
    y1=1;
    x1=1;
    if (y1>x1)
        y2=y1−1;
    else
        y++;
    assert(y2>x1);
}
```

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

$$\neg(y_2 > x_0)$$

Example Simulation

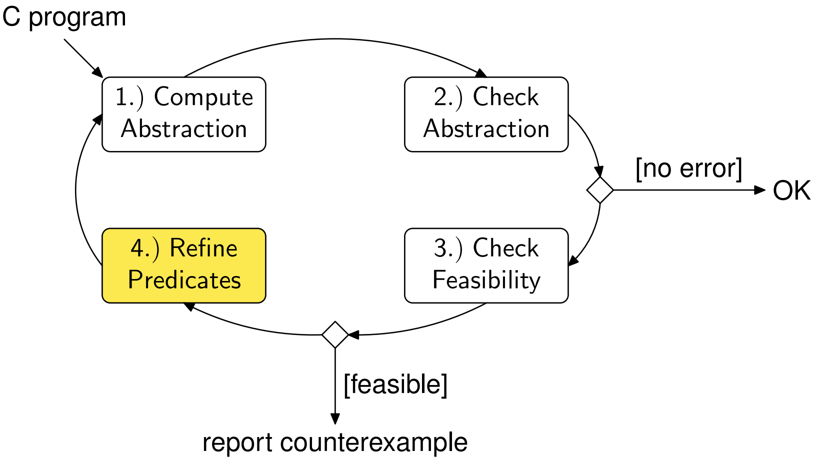
```
int main() {
    int x, y;
    y1=1;
    x1=1;
    if (y1>x1)
        y2=y1−1;
    else
        y++;
    assert(y2>x1);
}
```

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

$$\neg(y_2 > x_0)$$

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 ∧ y = 1}
    if (y>x)
        y−−;
    else
        {x = 1 ∧ y = 1 ∧ ¬y > x}
        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++;
```

$\{x = 1 \wedge y = 2 \wedge y > x\}$

```
    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

<u>Using WP</u>

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

<u>Using SP</u>

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:

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

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

Predicate Refinement for Paths

Recall the decision problem we build for simulating paths:

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

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

Predicate Refinement for Paths

For a path with n steps:

$$A_1 \quad | \quad A_2 \quad | \quad A_3 \quad | \quad \cdots \quad | \quad A_n$$

$$\text{true} \quad \Rightarrow A_1' \quad \Rightarrow A_2' \quad \Rightarrow A_3' \quad \Rightarrow A_{n-1}' \quad \Rightarrow \text{false}$$

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:

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

  The interpolant is "x is even"

Craig Interpolation for Linear Inequalities

UNIVERSITY OF OXFORD

$$\frac{0 \leq x \quad 0 \leq y}{0 \leq c_1 x + c_2 y} \quad \text{with } 0 \leq c_1, c_2$$

- "Cutting-planes"
- Naturally arise in Fourier-Motzkin or Simplex

Example

$$A = (0 \leq x - y) \wedge (0 \leq y - z - 1)$$

$$B = (0 \leq z - x)$$

Example

$A = (0 \leq x - y) \wedge (0 \leq y - z - 1)$

$B = (0 \leq z - x)$

$0 \leq y - z - 1$

$0 \leq z - x$

Example

UNIVERSITY OF OXFORD

$$A = (0 \leq x - y) \wedge (0 \leq y - z - 1)$$

$$B = (0 \leq z - x)$$

$$0 \leq y - z - 1 \qquad 0 \leq z - x$$

$$0 \leq y - x - 1$$

Example

$$A = (0 \leq x - y) \wedge (0 \leq y - z - 1)$$

$$B = (0 \leq z - x)$$

$$0 \leq y - z - 1 \qquad 0 \leq z - x$$

$$0 \leq y - x - 1$$

$$0 \leq x - y$$

$$0 \leq -1$$

Example

$$A = (0 \le x - y) \wedge (0 \le y - z - 1) \qquad B = (0 \le z - x)$$

$$0 \le y - z - 1 \qquad 0 \le z - x$$

$$0 \le y - x - 1$$

$$0 \le x - y$$

$$0 \le -1$$

$$0 \le y - z - 1 \qquad 0 \le \mathbf{0}$$

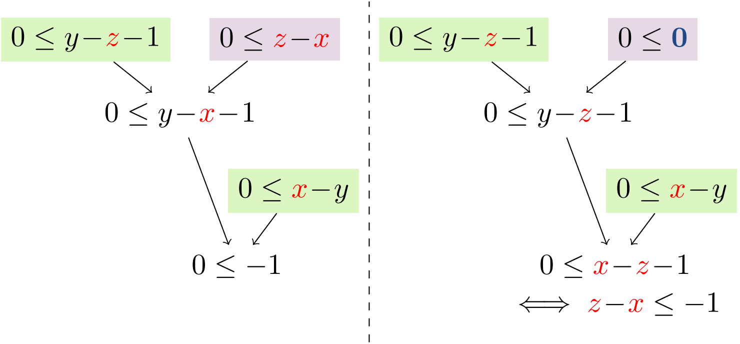
$$0 \le y - z - 1$$

$$0 \le x - y$$

$$0 \le x - z - 1$$

Example

$A = (0 \le x - y) \wedge (0 \le y - z - 1)$ $\quad$ $B = (0 \le z - x)$



Example

$A = (0 \leq x - y) \wedge (0 \leq y - z - 1)$ $\quad$ $B = (0 \leq z - x)$

$$\frac{\dfrac{0 \leq y - z - 1 \qquad 0 \leq z - x}{0 \leq y - x - 1} \qquad 0 \leq x - y}{0 \leq -1}$$

$$\frac{\dfrac{0 \leq y - z - 1 \qquad 0 \leq 0}{0 \leq y - z - 1} \qquad 0 \leq x - y}{0 \leq x - z - 1}$$

$$\iff \quad z - x \leq -1$$

Just sum the inequalities from A, and you get an interpolant!

Approximating Loop Invariants: SP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The SP refinement results in

$$sp(\text{x=y=0}, \text{true}) \quad = \quad x = 0 \wedge y = 0$$

Approximating Loop Invariants: SP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The SP refinement results in

$$
\begin{aligned}
sp(\text{x=y=0}, \text{true}) &= x = 0 \land y = 0 \\
sp(\text{x++; y++}, \ldots) &= x = 1 \land y = 1
\end{aligned}
$$

Approximating Loop Invariants: SP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The SP refinement results in

$$
\begin{aligned}
sp(\text{x=y=0}, \text{true}) &= x = 0 \wedge y = 0 \\
sp(\text{x++; y++}, \ldots) &= x = 1 \wedge y = 1 \\
sp(\text{x++; y++}, \ldots) &= x = 2 \wedge y = 2
\end{aligned}
$$

Approximating Loop Invariants: SP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The SP refinement results in

$$
\begin{aligned}
sp(\text{x=y=0}, \text{true}) &= x = 0 \wedge y = 0 \\
sp(\text{x++; y++}, \ldots) &= x = 1 \wedge y = 1 \\
sp(\text{x++; y++}, \ldots) &= x = 2 \wedge y = 2 \\
sp(\text{x++; y++}, \ldots) &= x = 3 \wedge y = 3 \\
\ldots
\end{aligned}
$$

✘ 10 iterations required to prove the property.

✘ It won't work if we replace 10 by n.

Approximating Loop Invariants: WP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The WP refinement results in

$$wp(\texttt{x==10}, y \neq 10) \quad = \quad y \neq 10 \wedge x = 10$$

Approximating Loop Invariants: WP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The WP refinement results in

$$wp(\texttt{x==10}, y \neq 10) \quad = \quad y \neq 10 \wedge x = 10$$
$$wp(\texttt{x++; y++}, \ldots) \quad = \quad y \neq 9 \wedge x = 9$$

Approximating Loop Invariants: WP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The WP refinement results in

$$
\begin{aligned}
wp(\texttt{x==10}, y \neq 10) &= y \neq 10 \wedge x = 10 \\
wp(\texttt{x++; y++}, \ldots) &= y \neq 9 \wedge x = 9 \\
wp(\texttt{x++; y++}, \ldots) &= y \neq 8 \wedge x = 8
\end{aligned}
$$

Approximating Loop Invariants: WP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The WP refinement results in

$$
\begin{aligned}
wp(\texttt{x==10}, y \neq 10) &= y \neq 10 \wedge x = 10 \\
wp(\texttt{x++; y++}, \ldots) &= y \neq 9 \wedge x = 9 \\
wp(\texttt{x++; y++}, \ldots) &= y \neq 8 \wedge x = 8 \\
wp(\texttt{x++; y++}, \ldots) &= y \neq 7 \wedge x = 7
\end{aligned}
$$

Approximating Loop Invariants: WP

```
int x, y;

x=y=0;

while(x!=10) {
  x++;
  y++;
}

assert(y==10);
```

The WP refinement results in

$$
\begin{aligned}
wp(\texttt{x==10}, y \neq 10) &= y \neq 10 \wedge x = 10 \\
wp(\texttt{x++; y++}, \ldots) &= y \neq 9 \wedge x = 9 \\
wp(\texttt{x++; y++}, \ldots) &= y \neq 8 \wedge x = 8 \\
wp(\texttt{x++; y++}, \ldots) &= y \neq 7 \wedge x = 7 \\
\ldots
\end{aligned}
$$

✘ Also requires 10 iterations.

✘ It won't work if we replace 10 by n.

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

$$x_1 = 0$$
$$y_1 = 0$$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

	1st It.
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

	1st It.	2nd It.
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$	$x_2 \neq 10$ $x_3 = x_2 + 1$ $y_3 = y_2 + 1$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

	1st It.	2nd It.	3rd It.
	$x_1 \neq 10$	$x_2 \neq 10$	$x_3 \neq 10$
$x_1 = 0$	$x_2 = x_1 + 1$	$x_3 = x_2 + 1$	$x_4 = x_3 + 1$
$y_1 = 0$	$y_2 = y_1 + 1$	$y_3 = y_2 + 1$	$y_4 = y_3 + 1$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$	$x_2 \neq 10$ $x_3 = x_2 + 1$ $y_3 = y_2 + 1$	$x_3 \neq 10$ $x_4 = x_3 + 1$ $y_4 = y_3 + 1$	$x_4 = 10$ $y_4 \neq 10$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$	$x_2 \neq 10$ $x_3 = x_2 + 1$ $y_3 = y_2 + 1$	$x_3 \neq 10$ $x_4 = x_3 + 1$ $y_4 = y_3 + 1$	$x_4 = 10$ $y_4 \neq 10$

$$x_1 = 0$$
$$y_1 = 0$$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$	$x_2 \neq 10$ $x_3 = x_2 + 1$ $y_3 = y_2 + 1$	$x_3 \neq 10$ $x_4 = x_3 + 1$ $y_4 = y_3 + 1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$ $\quad$ $x_2 = 1$

$y_1 = 0$ $\quad$ $y_2 = 1$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$	$x_2 \neq 10$ $x_3 = x_2 + 1$ $y_3 = y_2 + 1$	$x_3 \neq 10$ $x_4 = x_3 + 1$ $y_4 = y_3 + 1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$, $y_1 = 0$ $\qquad$ $x_2 = 1$, $y_2 = 1$ $\qquad$ $x_3 = 2$, $y_3 = 2$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1{+}1$ $y_2 = y_1{+}1$	$x_2 \neq 10$ $x_3 = x_2{+}1$ $y_3 = y_2{+}1$	$x_3 \neq 10$ $x_4 = x_3{+}1$ $y_4 = y_3{+}1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$, $y_1 = 0$ | $x_2 = 1$, $y_2 = 1$ | $x_3 = 2$, $y_3 = 2$ | $x_4 = 3$, $y_4 = 3$

What do we really need?

Consider an SSA-unwinding with 3 loop iterations:

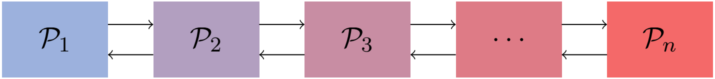
	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1+1$ $y_2 = y_1+1$	$x_2 \neq 10$ $x_3 = x_2+1$ $y_3 = y_2+1$	$x_3 \neq 10$ $x_4 = x_3+1$ $y_4 = y_3+1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$ $y_1 = 0$	$x_2 = 1$ $y_2 = 1$	$x_3 = 2$ $y_3 = 2$	$x_4 = 3$ $y_4 = 3$

✘ *This proof will produce the same predicates as SP.*

Split Provers

Idea:



- Each prover $\mathcal{P}_i$ only knows A_i, but they exchange facts
- We require that each prover only exchanges facts with common symbols
- Plus, we restrict the facts exchanged to some language $\mathcal{L}$

Back to the Example

Restriction to language $\mathcal{L}$ = "no new constants":

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$	$x_1 \neq 10$	$x_2 \neq 10$	$x_3 \neq 10$	$x_4 = 10$
$y_1 = 0$	$x_2 = x_1 + 1$	$x_3 = x_2 + 1$	$x_4 = x_3 + 1$	$y_4 \neq 10$
	$y_2 = y_1 + 1$	$y_3 = y_2 + 1$	$y_4 = y_3 + 1$	

Back to the Example

Restriction to language $\mathcal{L}$ = "no new constants":

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1{+}1$ $y_2 = y_1{+}1$	$x_2 \neq 10$ $x_3 = x_2{+}1$ $y_3 = y_2{+}1$	$x_3 \neq 10$ $x_4 = x_3{+}1$ $y_4 = y_3{+}1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$
$y_1 = 0$

Back to the Example

Restriction to language $\mathcal{L}$ = "no new constants":

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$	$x_2 \neq 10$ $x_3 = x_2 + 1$ $y_3 = y_2 + 1$	$x_3 \neq 10$ $x_4 = x_3 + 1$ $y_4 = y_3 + 1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$ $\quad x_2 = 1$

$y_1 = 0$ $\quad y_2 = 1$

Back to the Example

Restriction to language $\mathcal{L}$ = "no new constants":

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1+1$ $y_2 = y_1+1$	$x_2 \neq 10$ $x_3 = x_2+1$ $y_3 = y_2+1$	$x_3 \neq 10$ $x_4 = x_3+1$ $y_4 = y_3+1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$ $\quad x_2 = 1$ $\quad x_3 = 2$

$y_1 = 0$ $\quad y_2 = 1$ $\quad y_3 = 2$

Back to the Example

Restriction to language $\mathcal{L}$ = "no new constants":

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$	$x_2 \neq 10$ $x_3 = x_2 + 1$ $y_3 = y_2 + 1$	$x_3 \neq 10$ $x_4 = x_3 + 1$ $y_4 = y_3 + 1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$, $y_1 = 0$ $x_2 = 1$, $y_2 = 1$ $x_3 = 2$, $y_3 = 2$ ✗

Back to the Example

Restriction to language $\mathcal{L}$ = "no new constants":

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1{+}1$ $y_2 = y_1{+}1$	$x_2 \neq 10$ $x_3 = x_2{+}1$ $y_3 = y_2{+}1$	$x_3 \neq 10$ $x_4 = x_3{+}1$ $y_4 = y_3{+}1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$, $y_1 = 0$ $\quad$ $x_2 = 1$, $y_2 = 1$ $\quad$ $x_3 = y_3$

Back to the Example

Restriction to language $\mathcal{L}$ = "no new constants":

	1st It.	2nd It.	3rd It.	Assertion
$x_1 = 0$ $y_1 = 0$	$x_1 \neq 10$ $x_2 = x_1 + 1$ $y_2 = y_1 + 1$	$x_2 \neq 10$ $x_3 = x_2 + 1$ $y_3 = y_2 + 1$	$x_3 \neq 10$ $x_4 = x_3 + 1$ $y_4 = y_3 + 1$	$x_4 = 10$ $y_4 \neq 10$

$x_1 = 0$, $y_1 = 0$ $\qquad$ $x_2 = 1$, $y_2 = 1$ $\qquad$ $x_3 = y_3$ $\qquad$ $x_4 = y_4$

Invariants from Restricted Proofs

✔ The language restriction forces the solver to generalize!

- Algorithm:

  - If the proof fails, increase $\mathcal{L}$!
  - If we fail to get a sufficiently strong invariant, increase n.

✔ This does work if we replace 10 by n!

Invariants from Restricted Proofs

✔ The language restriction forces the solver to generalize!

- Algorithm:
  - If the proof fails, increase $\mathcal{L}$!
  - If we fail to get a sufficiently strong invariant, increase n.

✔ This does work if we replace 10 by n!

? Which $\mathcal{L}_1, \mathcal{L}_2, \ldots$ is complete for which programs?