Program Verification

CS60030 FORMAL SYSTEMS

PALLAB DASGUPTA,
FNAE, FASc,
A K Singh Distinguished Professor in AI,
Dept of Computer Science & Engineering
Indian Institute of Technology Kharagpur
Email: pallab@cse.iitkgp.ac.in
Web: http://cse.iitkgp.ac.in/~pallab

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

FMSAFE
FORMAL METHODS FOR SAFETY CRITICAL SYSTEMS

Software Verification

Is a software program free from bugs?

- What kind of bugs?
  - Lint checking – Divide by zero, Variable values going out of range
  - User specified bugs – Assertions

Challenges:

- Real valued variables
  - Huge state space if we have to consider all values
- Size of the program is much smaller than the number of paths to be explored
  - Branchings, Loops

We need to extract an abstract state machine from a program

Abstraction: Sound versus Complete

- **Sound Abstraction**

  If the abstraction shows no bugs, then the original program also doesn’t have bugs

- **Complete Abstraction**

  If the abstraction shows a bug, then the original program has a bug

Due to undecidability of static analysis problems, we cant have a general procedure that is both sound and complete.

Techniques

Abstract Static Analysis

- Abstract interpretation
- Numerical abstract domains

Software Model Checking

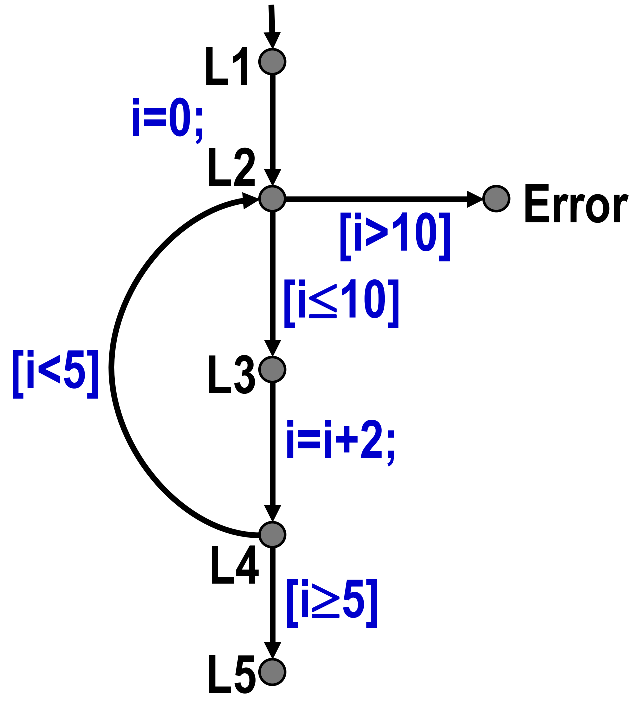
- Explicit and symbolic model checking
- Predicate abstraction and abstraction refinement

Example

Sample program:

```
int i=0
do {
        assert( i <= 10);
        i = i+2;
} while (i < 5);
```

Control Flow Graph (CFG):



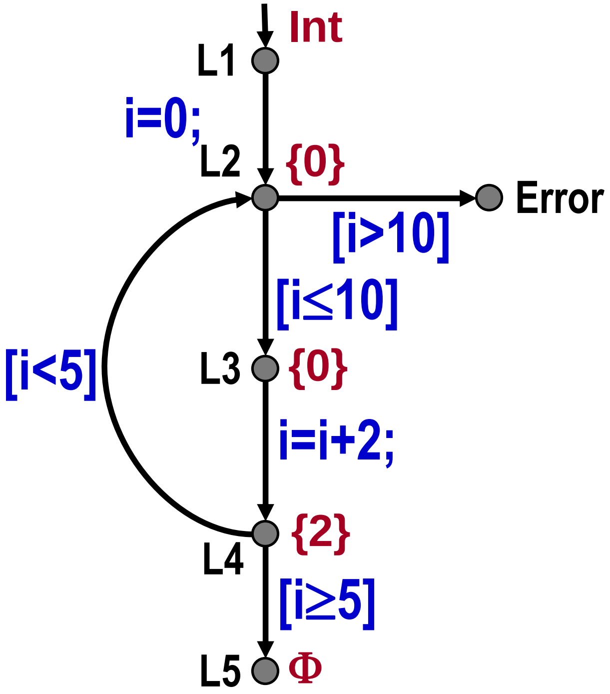
Concrete Interpretation

Sample program:

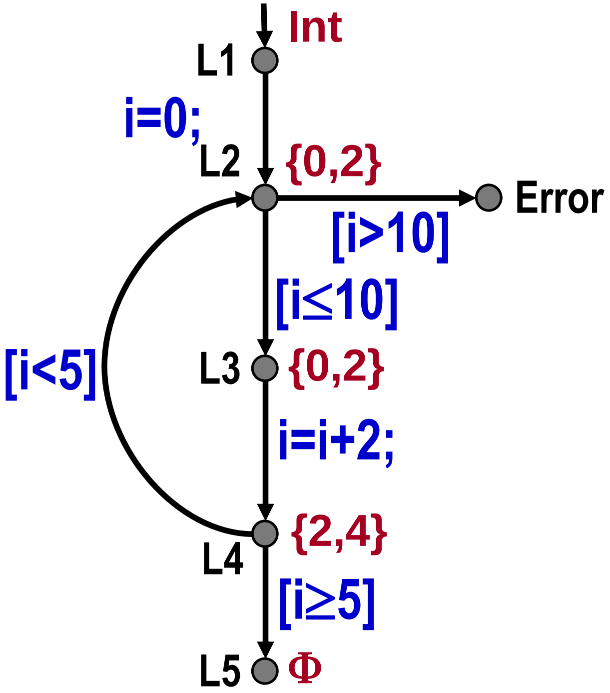
```
int i=0
do {
              assert( i <= 10);
              i = i+2;
}             while (i < 5);
```

Philosophy:

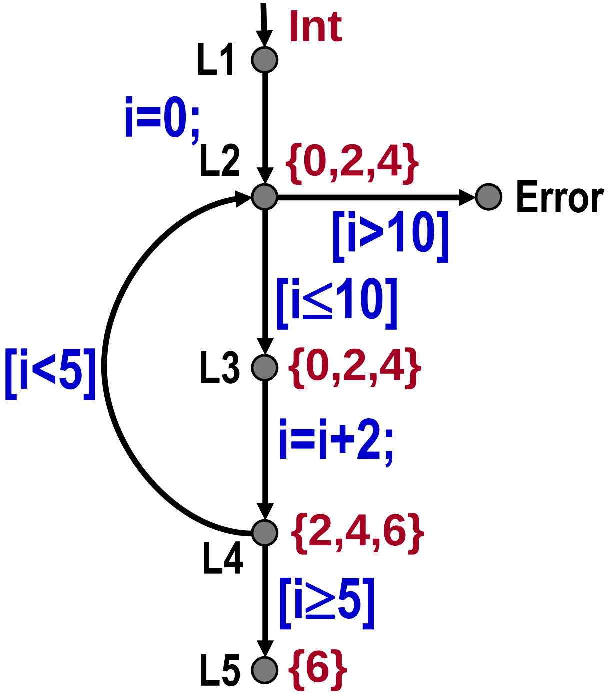
Collect the set of possible values of i until a fixed point is reached



Iteration-1



Iteration-2



Iteration-3

Abstract Interpretation

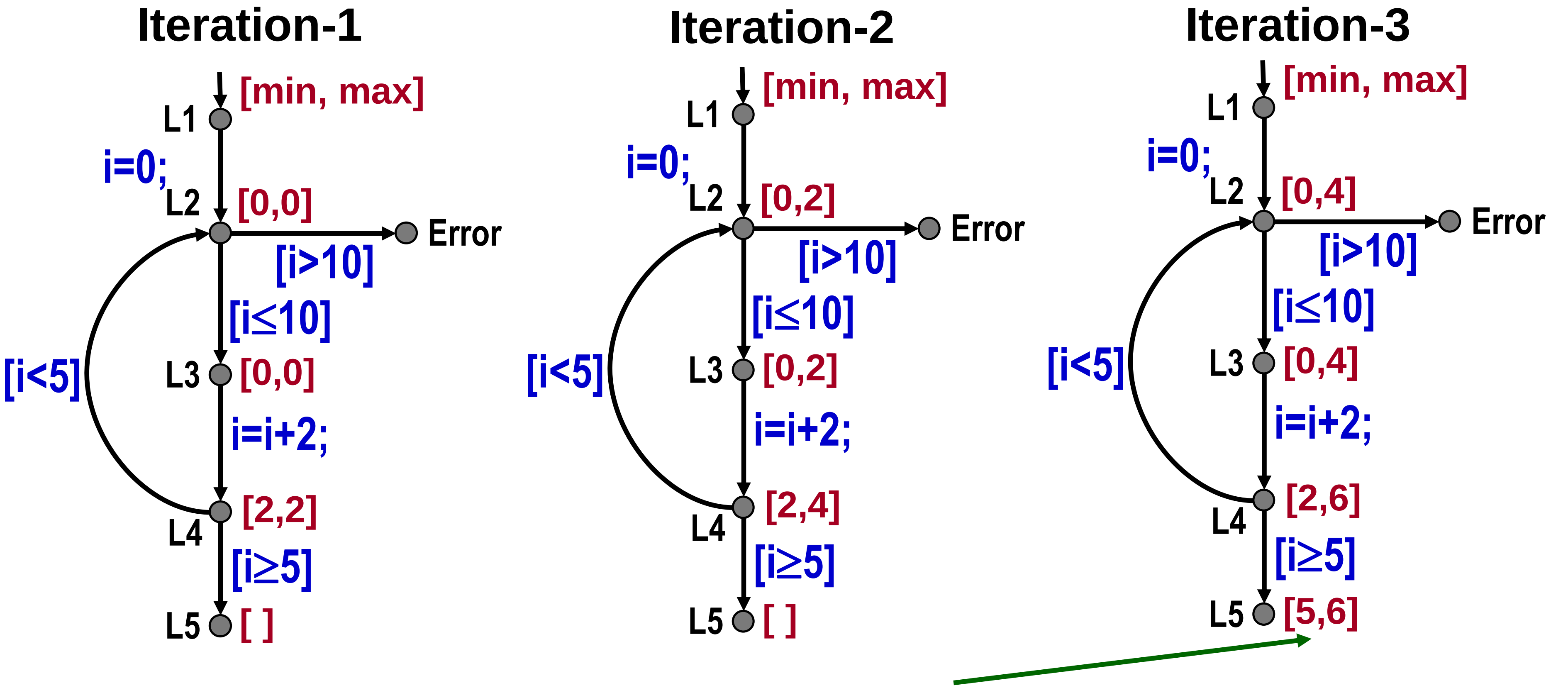
Sample program:

```
int i=0
do {
        assert( i <= 10);
        i = i+2;
}       while (i < 5);
```

Philosophy:

Use an abstract domain instead of value sets

Example: We may use value intervals instead of value sets



Actually, the value 5 is not possible here

Numerical Abstract Domains

The class of invariants that can be computed, and hence the properties that can be proved, varies with the expressive power of a domain

- An abstract domain can be more *precise* than another
- The information loss between different domains may be incomparable

Examples:

- The domain of *Signs* has three values: {Pos, Neg, Zero}
- *Intervals* are more expressive than signs. Signs can be modeled as [min,0], [0,0], and [0,max]
- The domain of *Parities* abstracts values as Even and Odd
- *Signs* or *Intervals* cannot be compared with *Parities.*

Predicate Abstraction

- A sound approximation R’ of the transition relation R is constructed using predicates over program variables
- A predicate P partitions the states of a program into two classes: one in which P evaluates to true and one in which it evaluates to false
  - Each class is an *abstract state*
  - Let A and B be abstract states. A transition is defined from A to B if there is a state in A with a transition to a state in B
  - This construction yields an existential abstraction of a program, which is sound for reachability properties
  - The abstract program corresponding to R’ is represented by a *Boolean program*, one with only Boolean data types, and the same control flow constructs as C programs

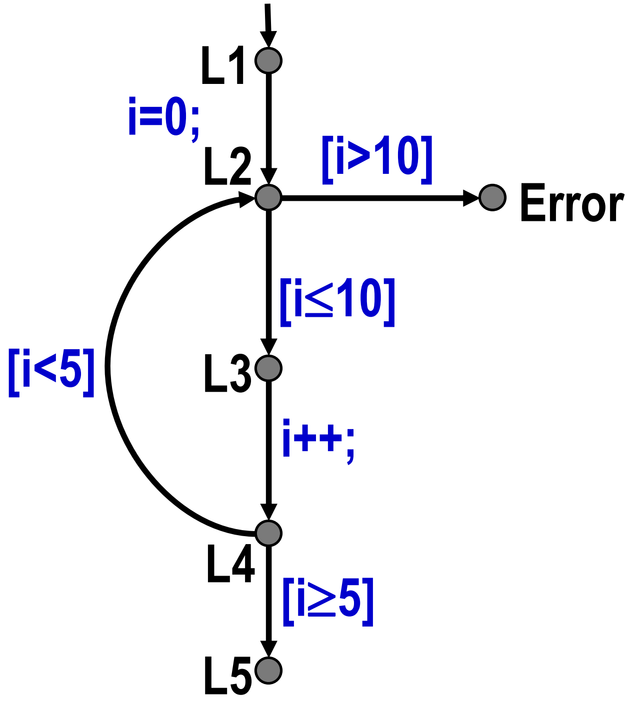
Predicate Abstraction

Sample program:

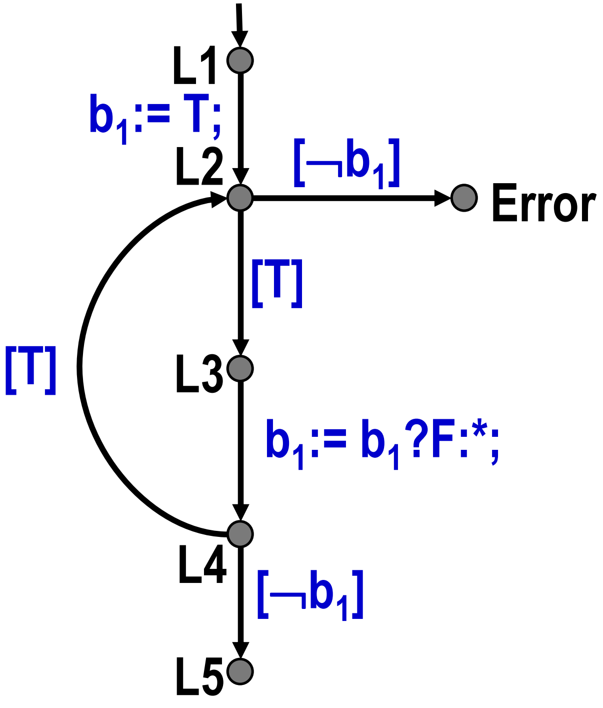
```
int i=0
do {
        assert( i <= 10);
        i++;
}  while (i < 5);
```

Abstraction-1 uses the predicate (i=0) (represented by the variable b_1)

CFG of program



Abstraction-1



In Abstraction-1 the Error location is reachable, but the counter-example cant be reconstructed in the real program

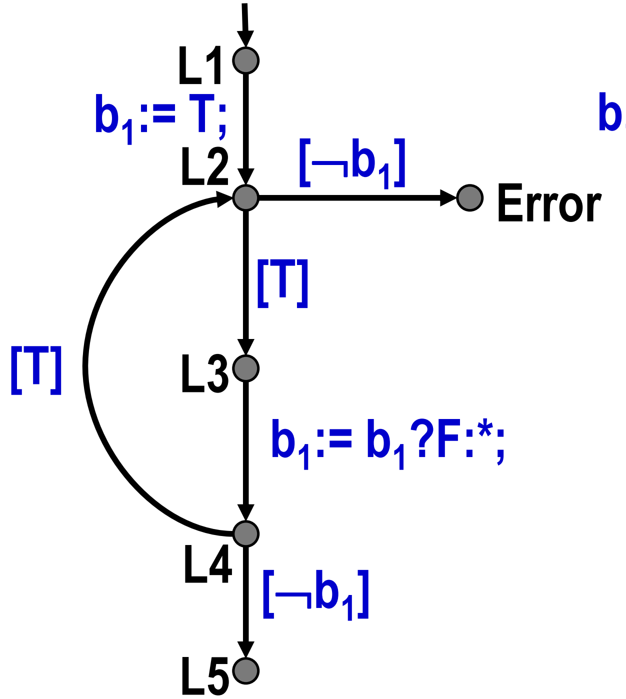
Predicate Abstraction

Sample program:

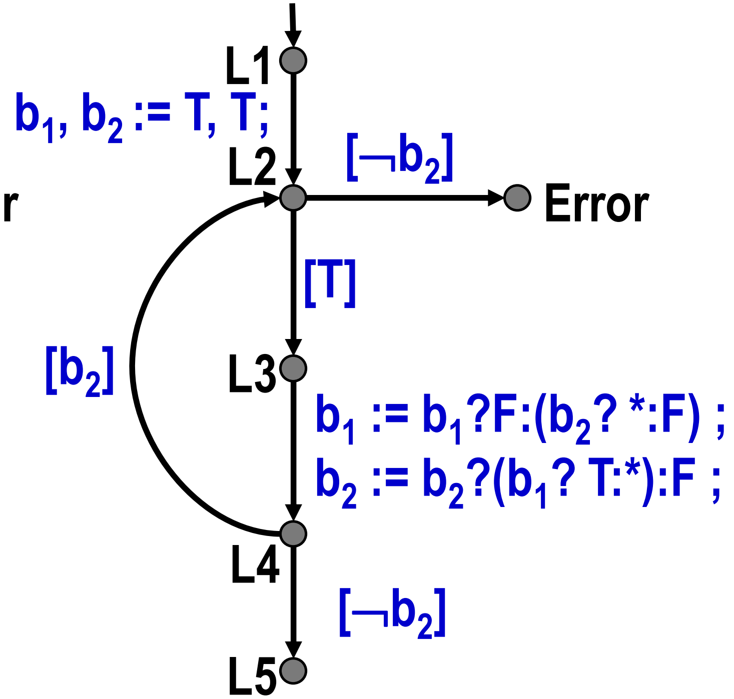
```
int i=0
do {
        assert( i <= 10);
        i++;
}  while (i < 5);
```

Abstraction-2 refines Abstraction-1 using the additional predicate (i<5) (represented by the variable b_2)

Abstraction-1



Abstraction-2



In Abstraction-2 the location L2 is reached with b_2 every time. Hence the Error location is unreachable.

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 details
- 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

Source of these slides: D. Kroening: SSFT12 – Predicate Abstraction: A Tutorial

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

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:

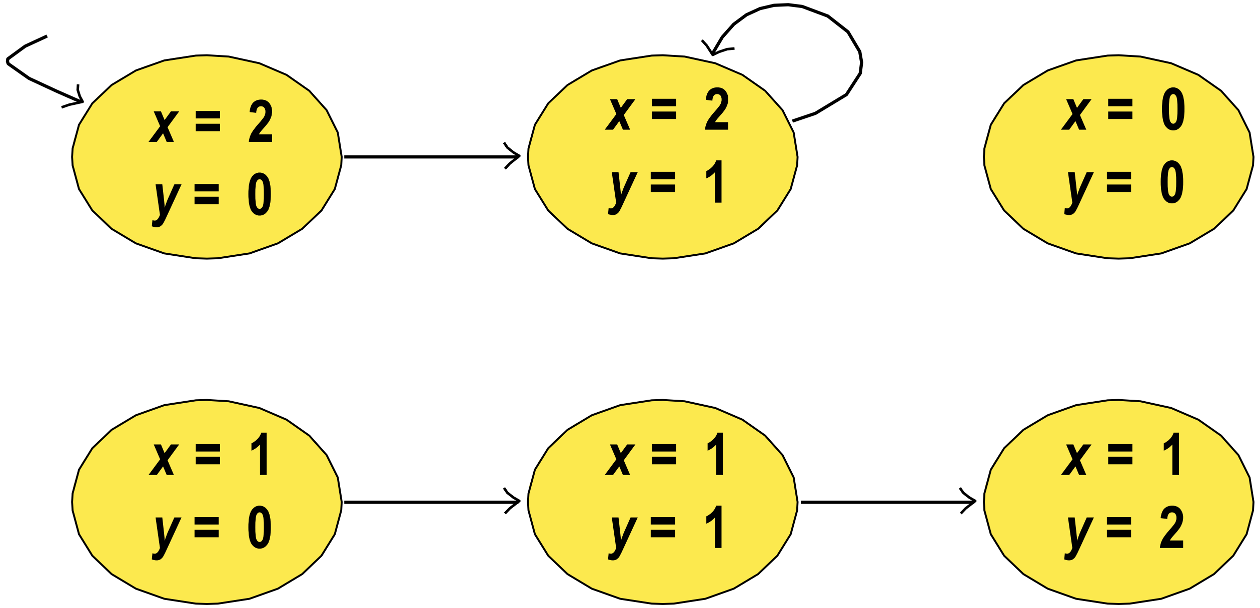
$$\hat{S} = \mathrm{B}^n$$

- The abstraction function:

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

Predicate Abstraction: the Basic Idea

Concrete states over variables *x*, *y*:



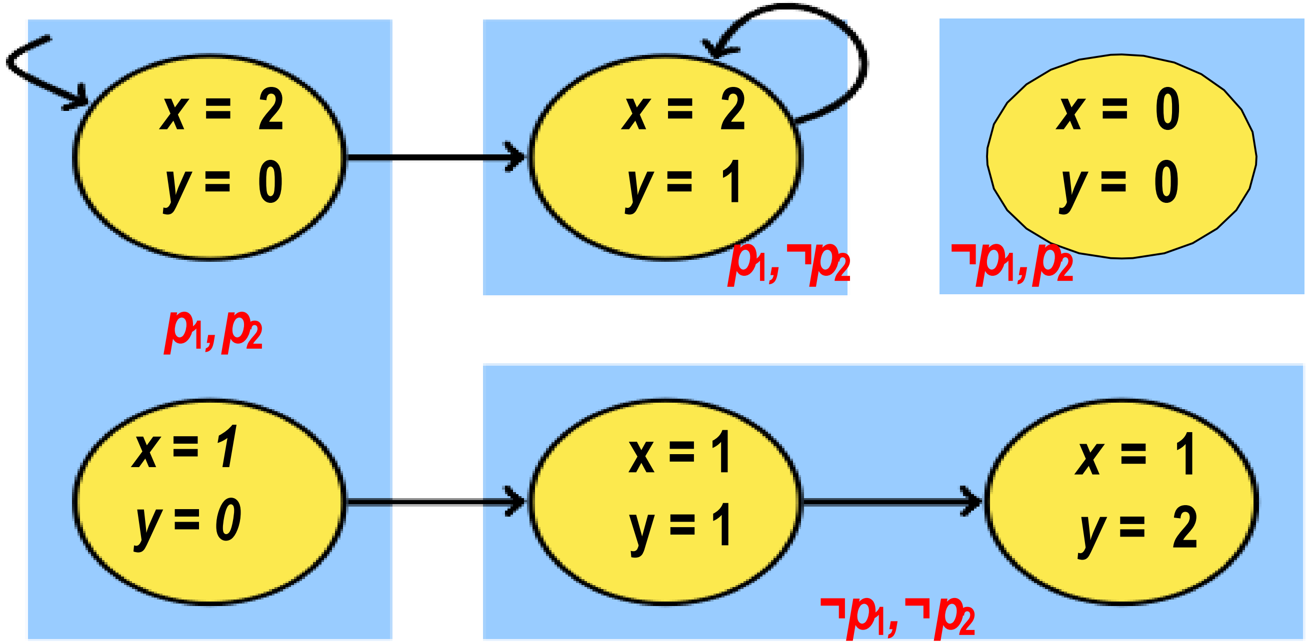
Predicates:

p1 ⟺ $x > y$

p2 ⟺ $y = 0$

Predicate Abstraction: The Basic Idea

Concrete states over variables *x*, *y*:



Predicates:

$p_1 \Longleftrightarrow x > y$

$p_2 \Longleftrightarrow y = 0$

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 $\alpha : S \rightarrow \hat{S}$ iff

- $\exists\, s \in S_0.\, \alpha(s) = \hat{s} \Rightarrow \hat{s} \in \hat{S}_0$ and
- $\exists\, (s, s^t) \in T.\, \alpha(s) = \hat{s} \wedge \alpha(s^t) = \hat{s}^t \Rightarrow (\hat{s}, \hat{s}^t) \in \hat{T}$.

---

[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 $\alpha : S \rightarrow \hat{S}$ iff

- $\exists\, s \in S_0.\ \alpha(s) = \hat{s} \quad \Leftrightarrow \quad \hat{s} \in \hat{S}_0$ and
- $\exists\, (s, s^t) \in T.\ \alpha(s) = \hat{s} \wedge \alpha(s^t) = \hat{s}^t \quad \Leftrightarrow \quad (\hat{s}, \hat{s}^t) \in \hat{T}$.

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 $L : S \rightarrow P\ (A)$

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
  $x = 0 \in A$ and $x \neq 0 \in A$

Abstracting Properties

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

$$a \in \hat{L}(\hat{s}) \Leftrightarrow \forall s \mid \alpha(s) = \hat{s} . a \in L(s)$$

- 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)

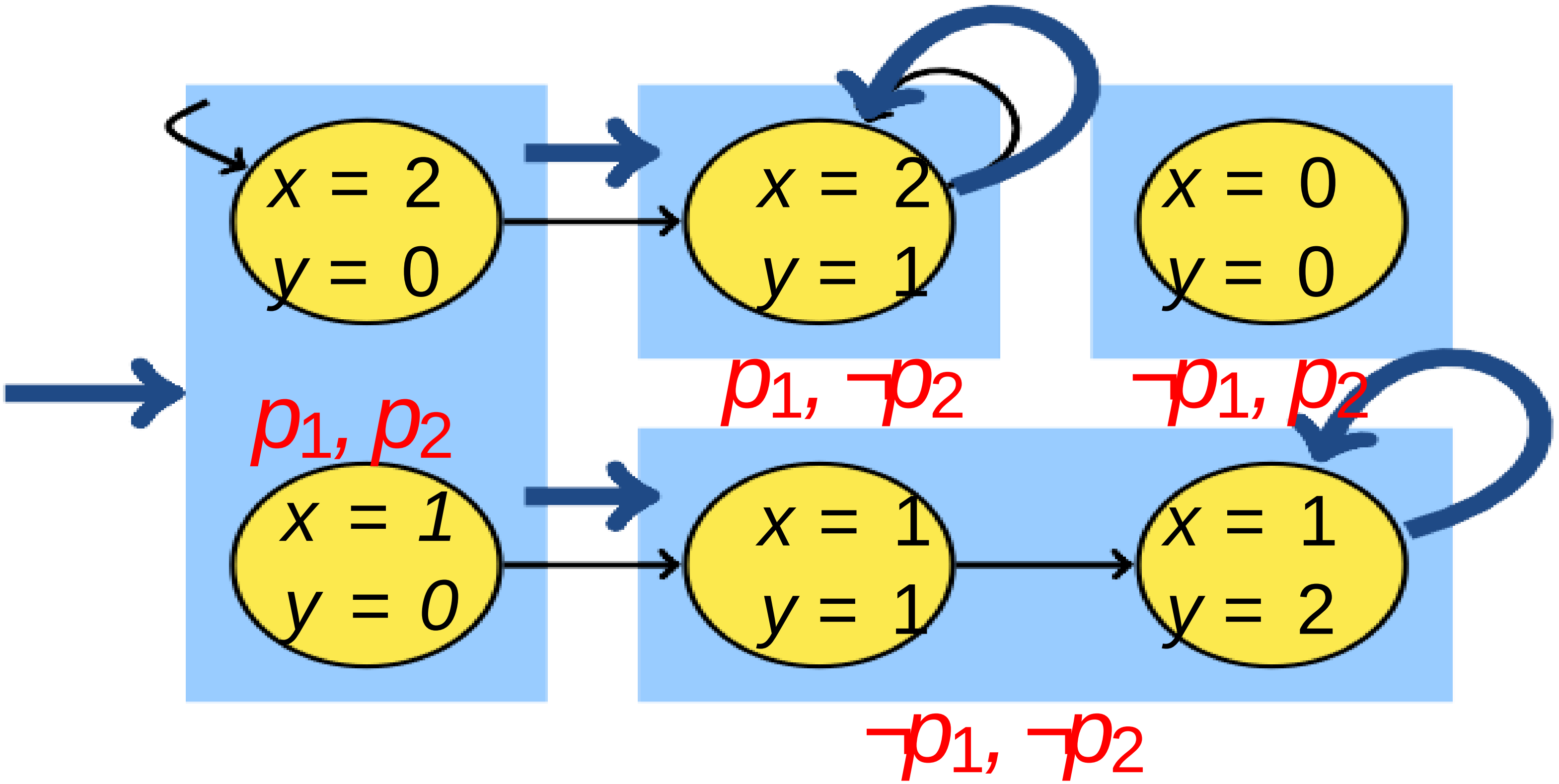
Let φ be a $\forall$CTL formula where all negations are pushed into the atomic propositions, and let $\hat{M}$ be an existential abstraction of M. If φ holds on $\hat{M}$, then it also holds on M.*

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

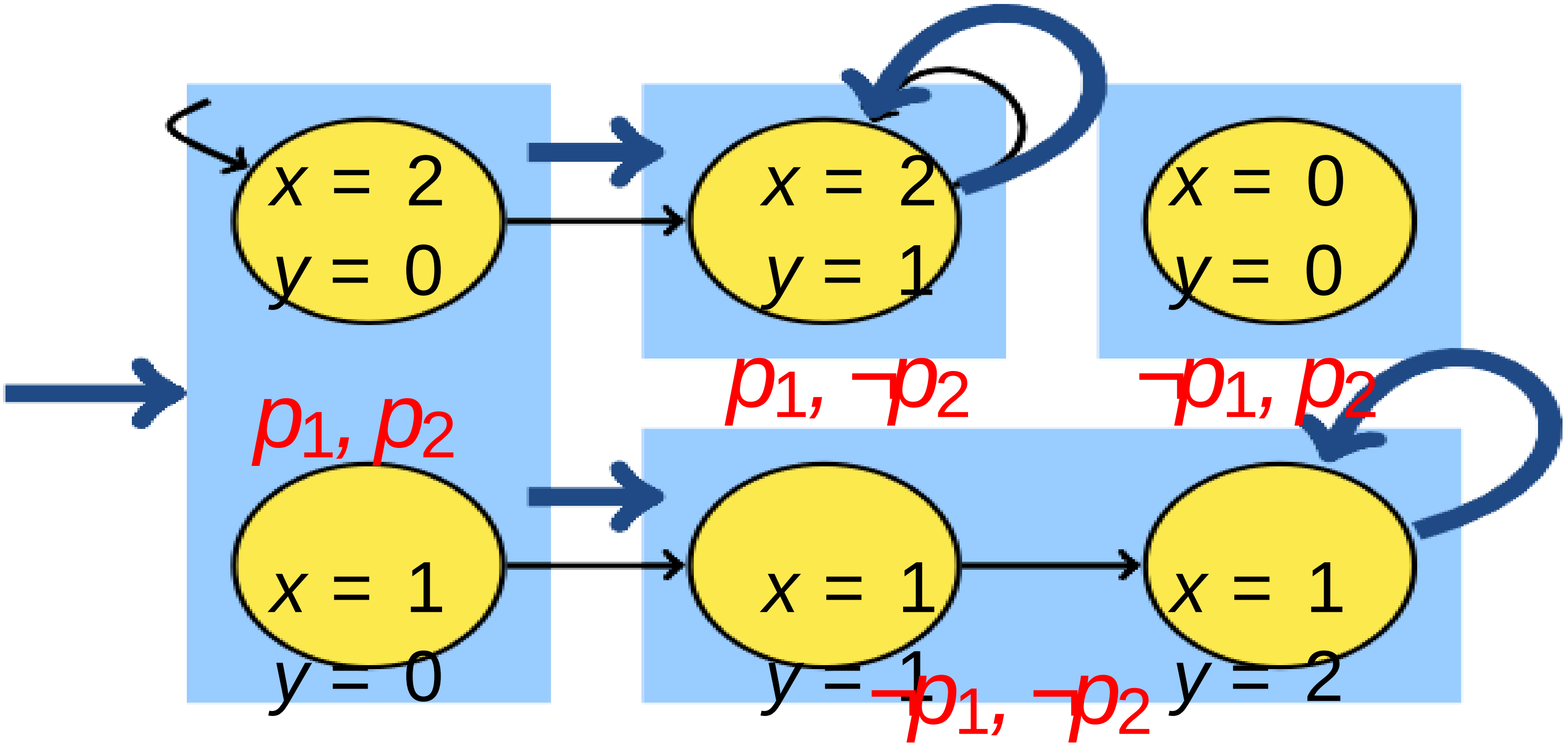
We say that an existential abstraction is conservative for $\forall$CTL* properties. The same result can be obtained for LTL properties.

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

Back to the Example



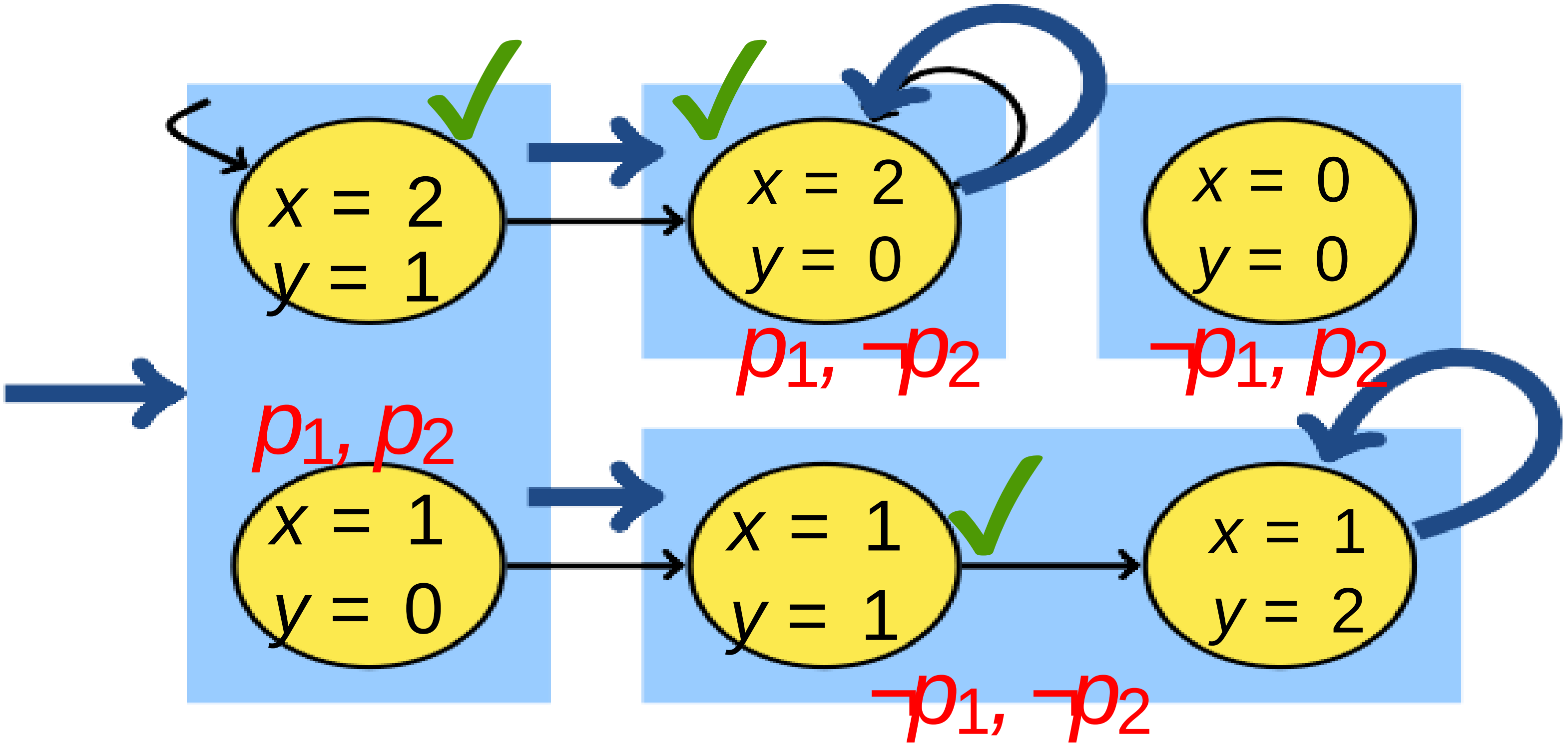
Let’s try a Property



Property:

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

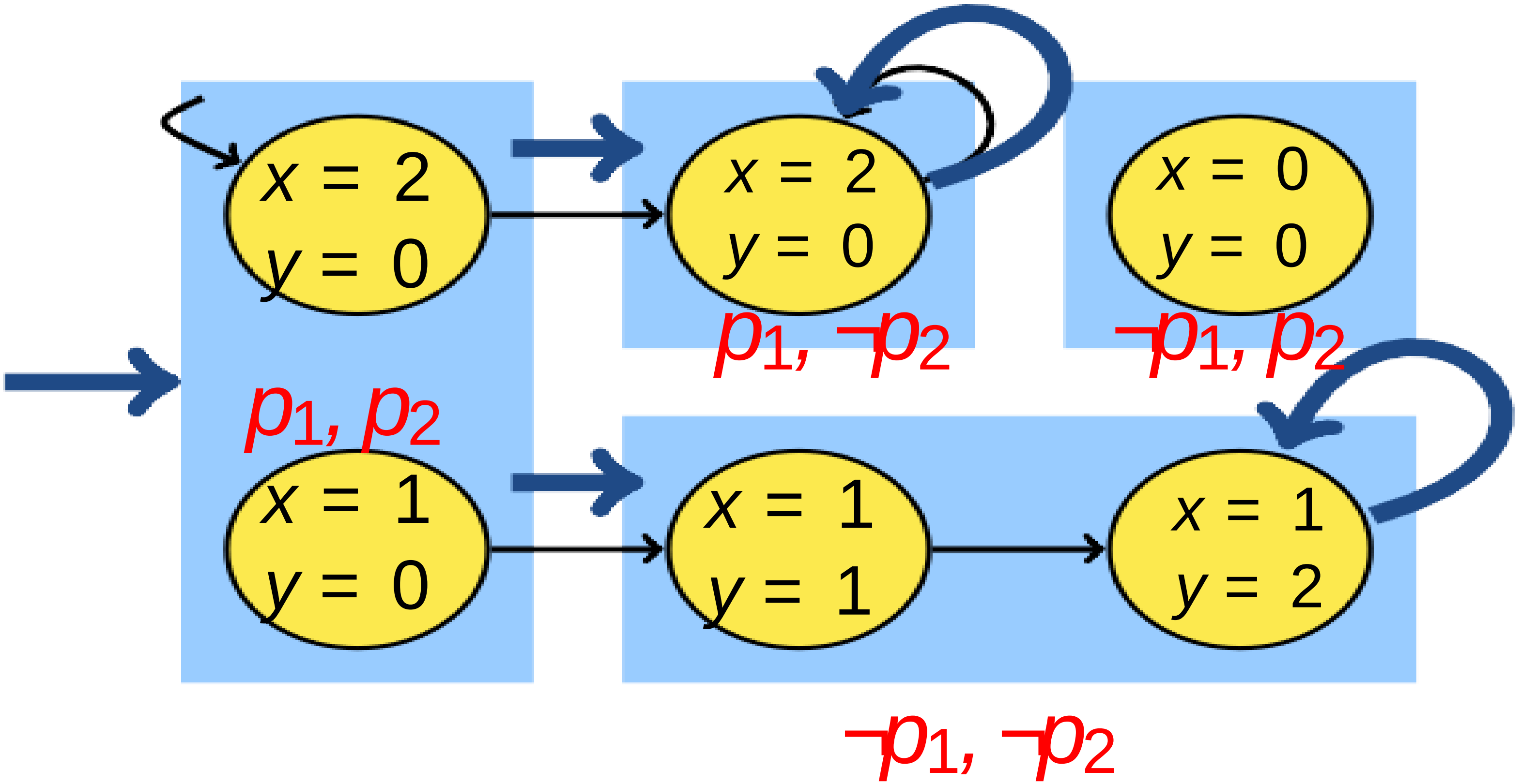
Let’s try a Property



Property:

$$x > y \vee y \neq 0 \quad \Longleftrightarrow \quad p_1 \vee \neg p_2$$

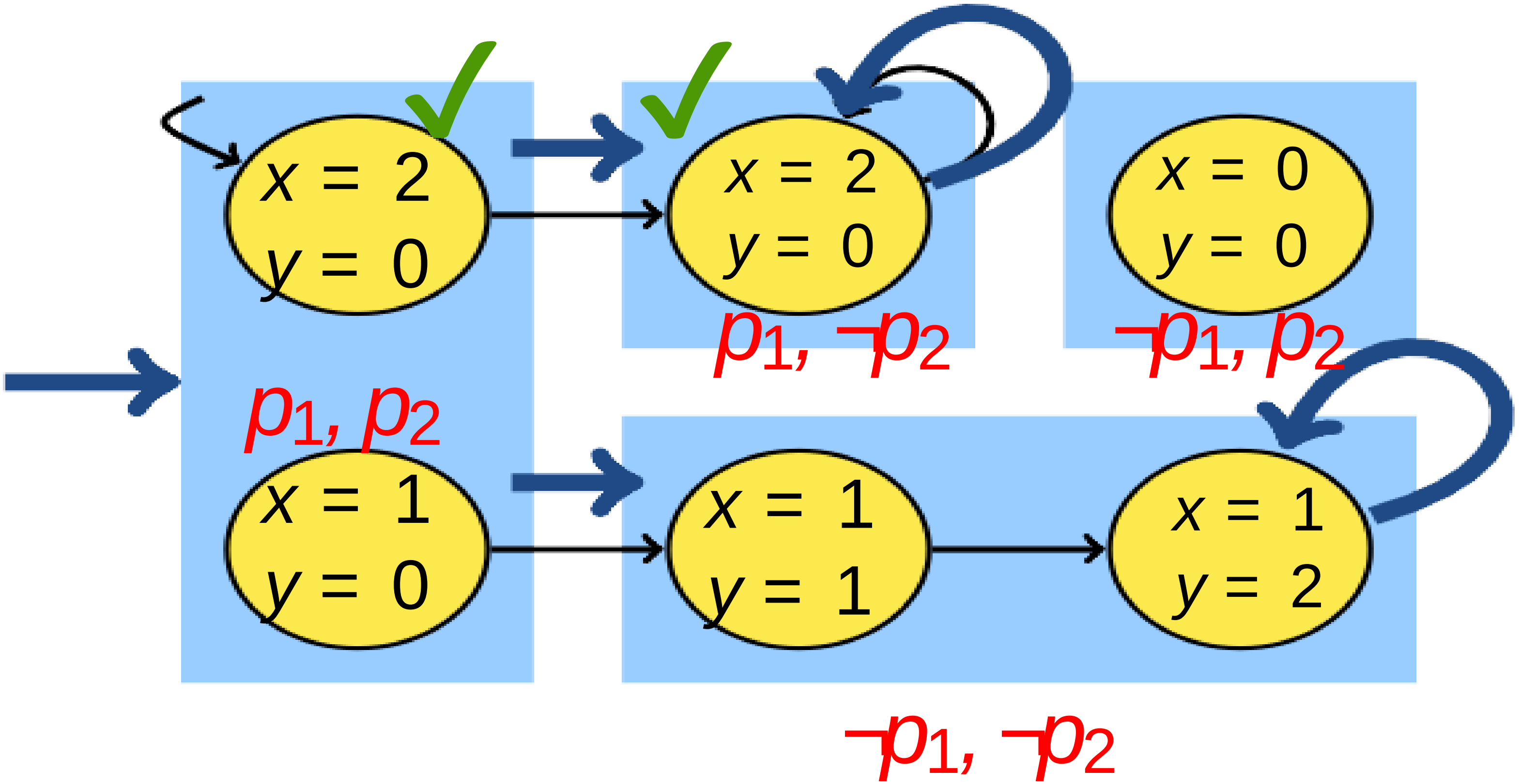
Another Property



Property:

$x > y \iff p_1$

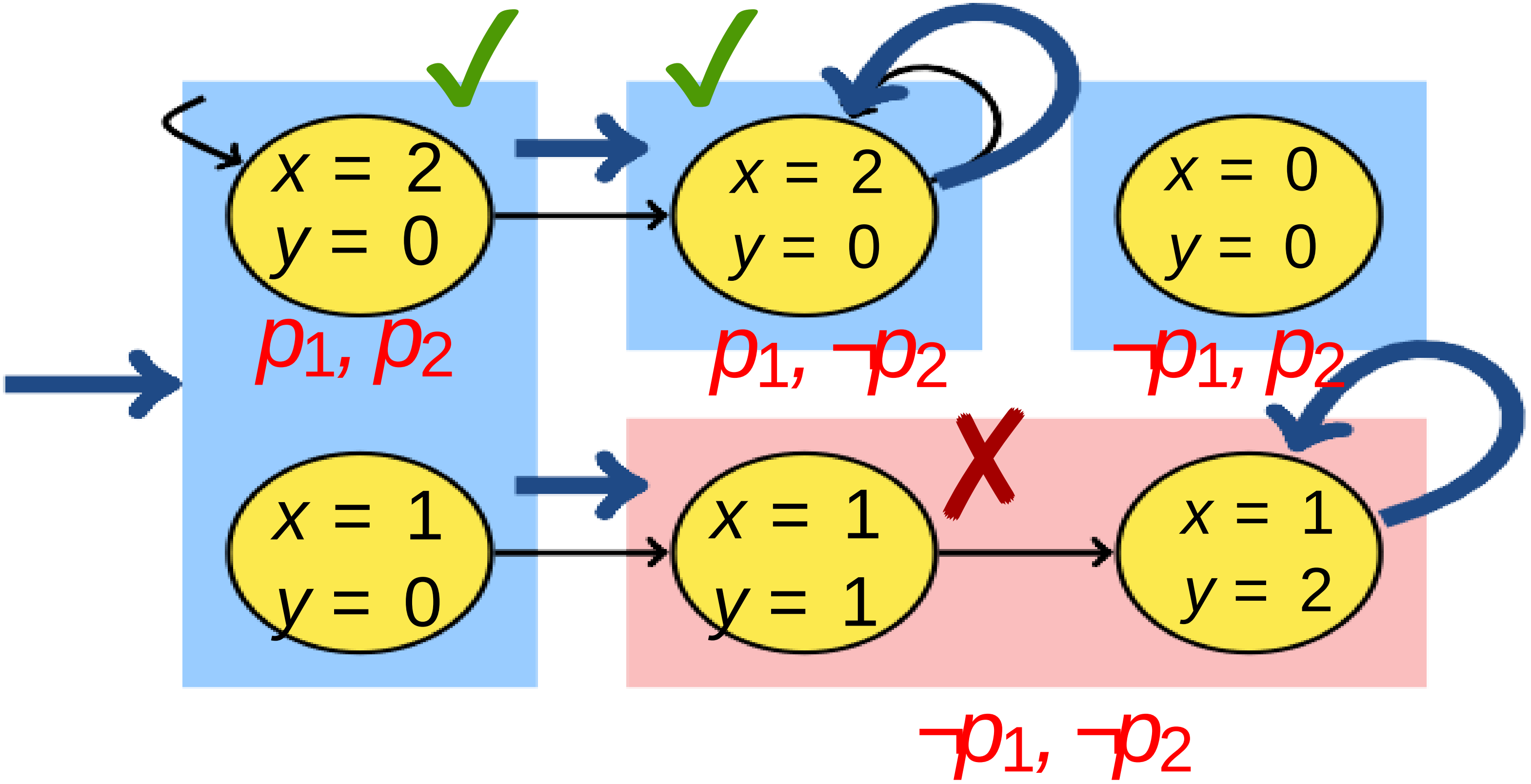
Another Property



Property:

$x > y \iff p_1$

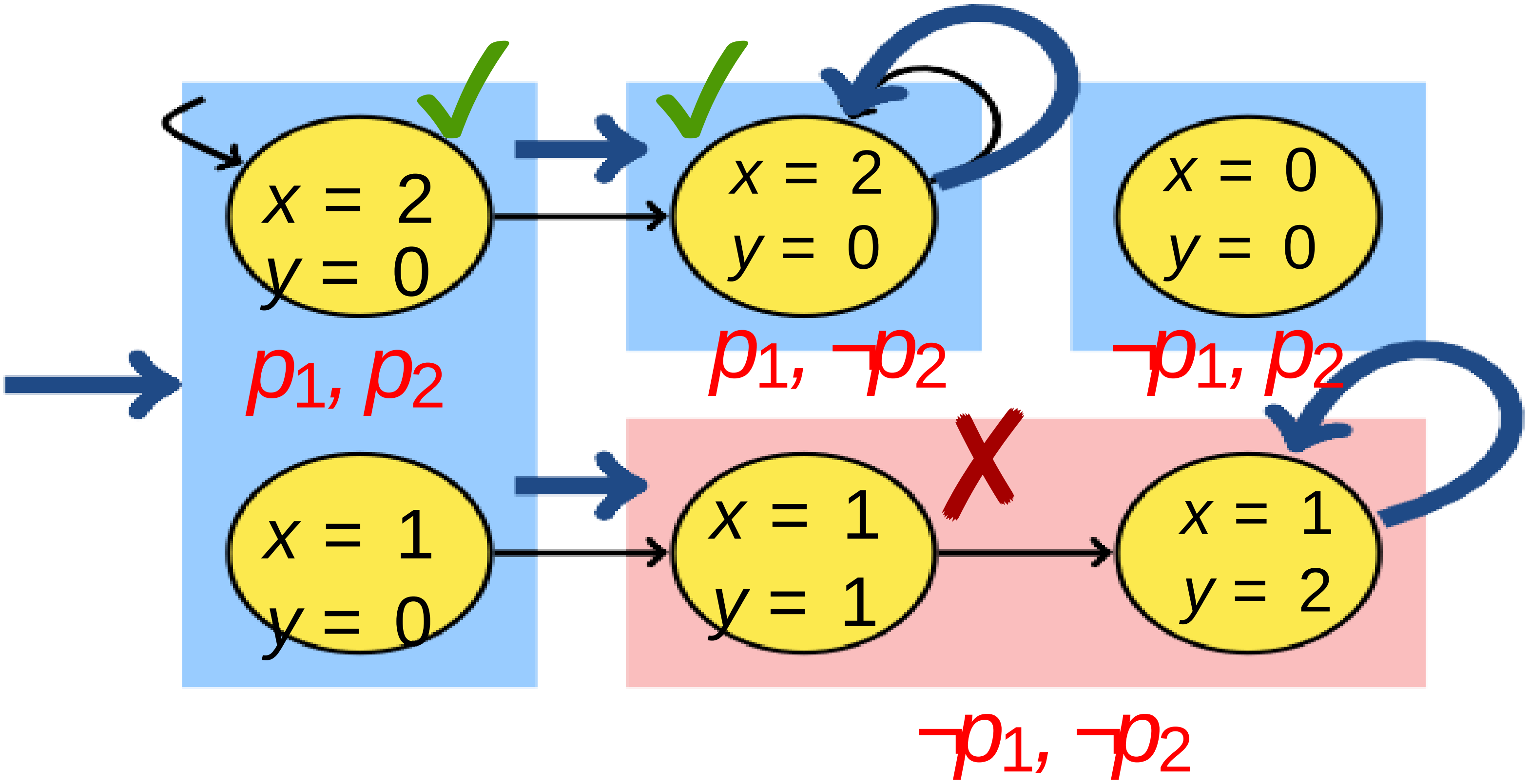
Another Property



Property:

$x > y \iff p_1$

Another Property



Property:

$x > y \iff p_1$

But: the counterexample is spurious

SLAM

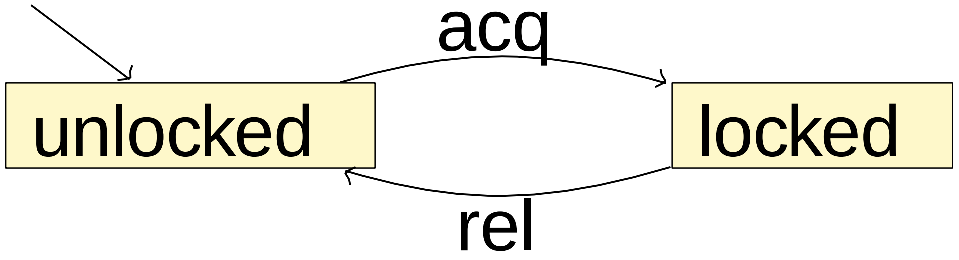
- Microsoft blames most Windows crashes on third party device drivers
- The Windows device driver API is quite complicated
- Drivers are low level C code
- SLAM: Tool to automatically check device drivers for certain errors
- SLAM is shipped with Device Driver Development Kit
- Full detail available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

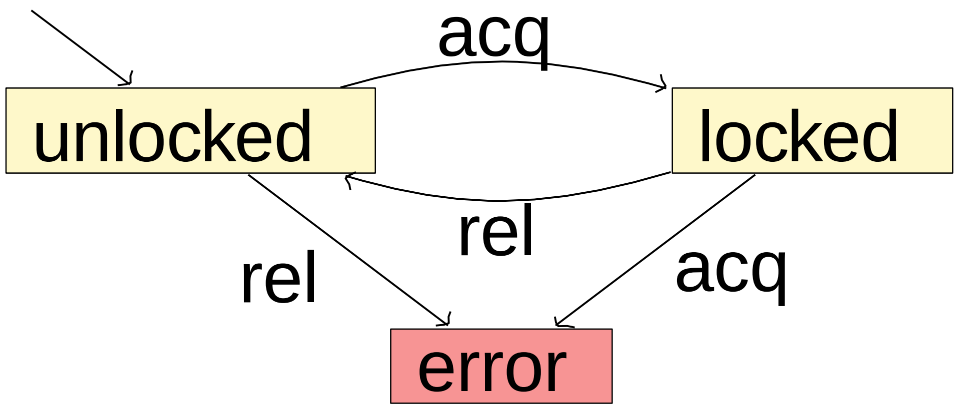
- Suitable for expressing control-dominated properties
    - e.g., proper sequence of events
    - can track data values

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

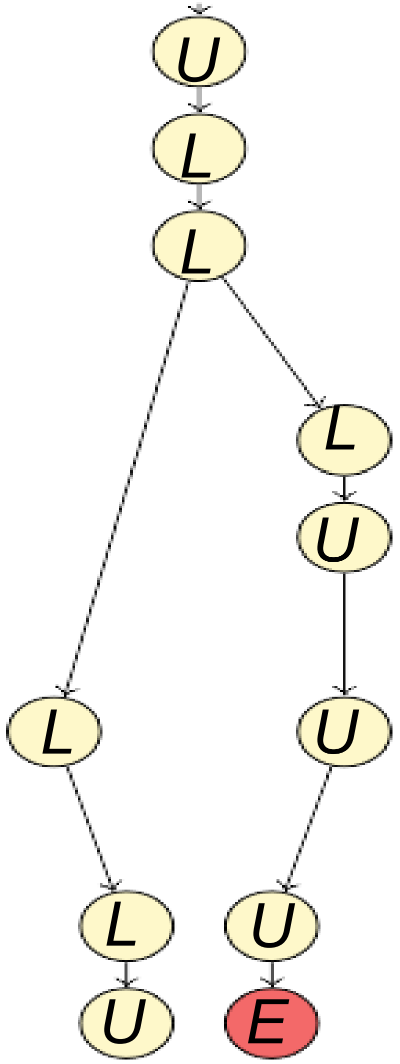
Does this code obey the locking rule?

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

Refinement Example

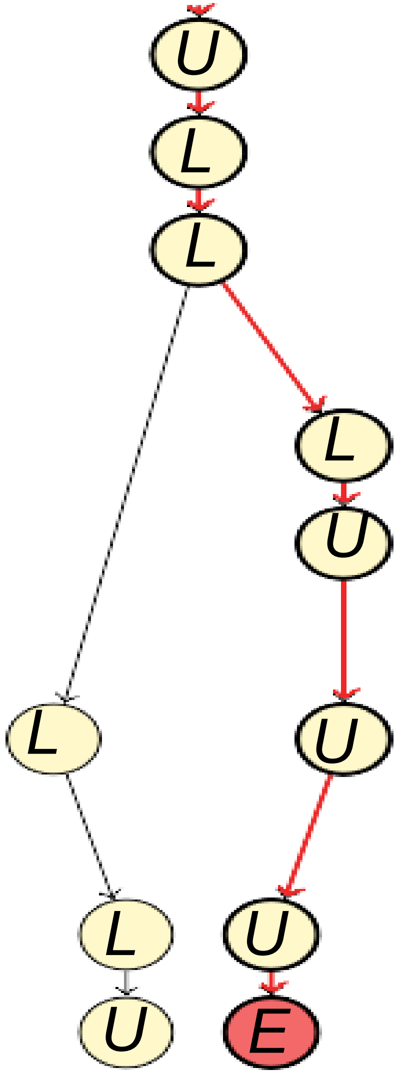
```
do {
    KeAcquireSpinLock();

    if (*)  {

        KeReleaseSpinLock();

    }
} while(*);
KeReleaseSpinLock();
```

Refinement Example



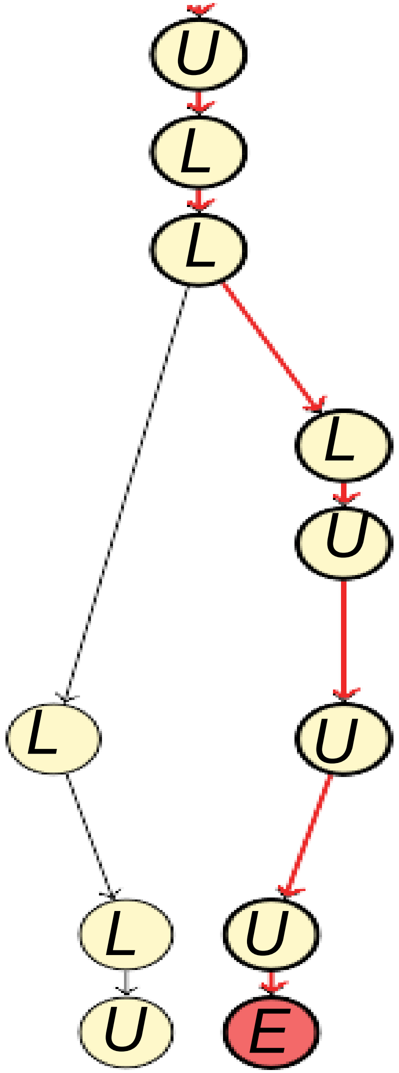
```
do {
        KeAcquireSpinLock();

        if (*)  {

             KeReleaseSpinLock();
             }

        } while(*);


        KeReleaseSpinLock();
```

Refinement Example



```
do {
        KeAcquireSpinLock();

        if (*) {

            KeReleaseSpinLock();
              }

        } while(*);


        KeReleaseSpinLock();
```

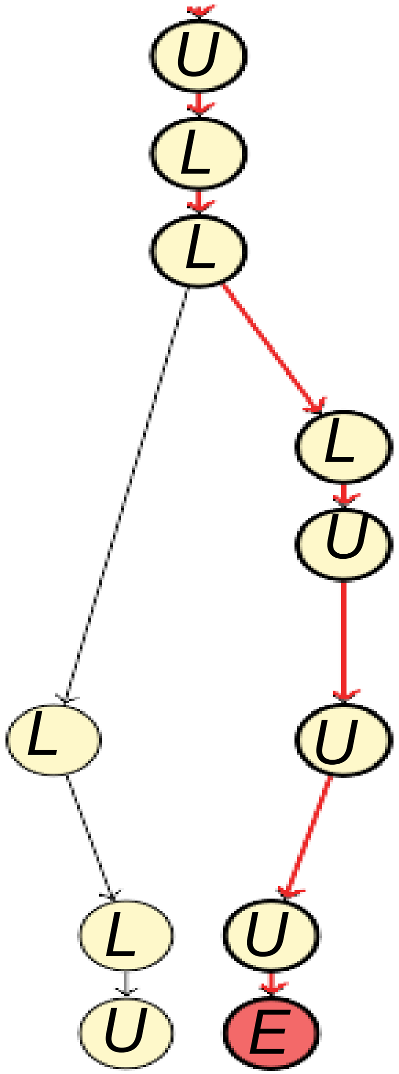
Refinement Example



```
do {
    KeAcquireSpinLock();

    if (*) {

        KeReleaseSpinLock();
    }

} while(*);


KeReleaseSpinLock();
```

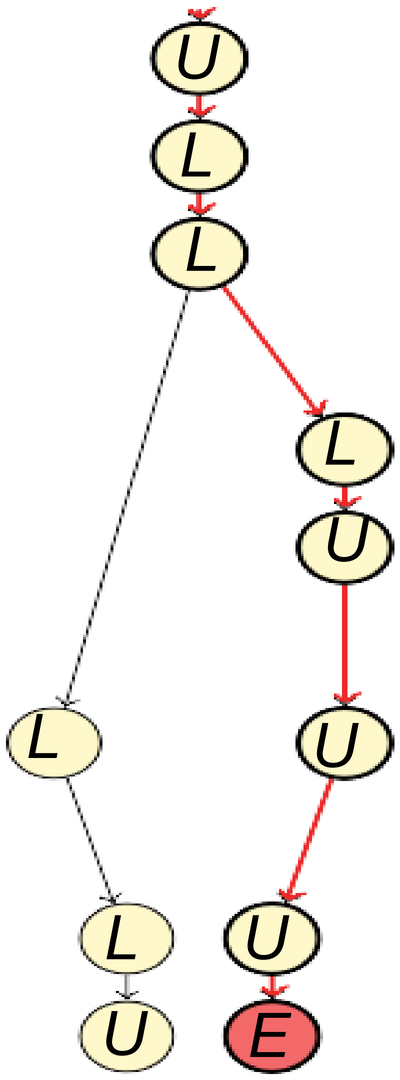
Is this path concretizable?

Refinement Example



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

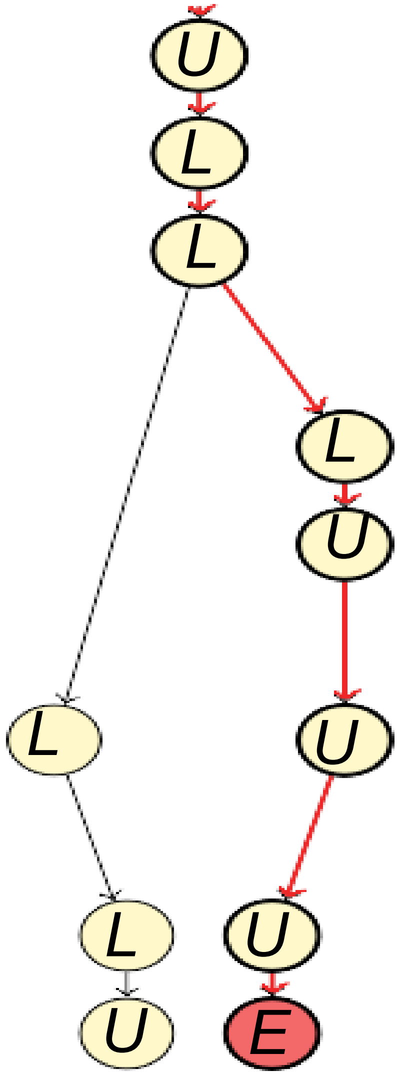
Refinement Example



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

KeReleaseSpinLock();
```

This path is spurious!

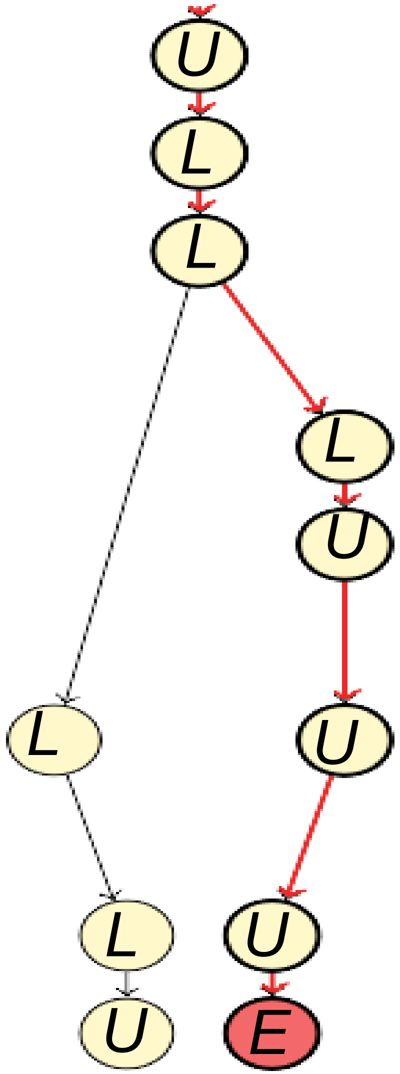
Refinement Example



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

KeReleaseSpinLock();
```

Let’s add the predicate nPacketsOld==nPackets

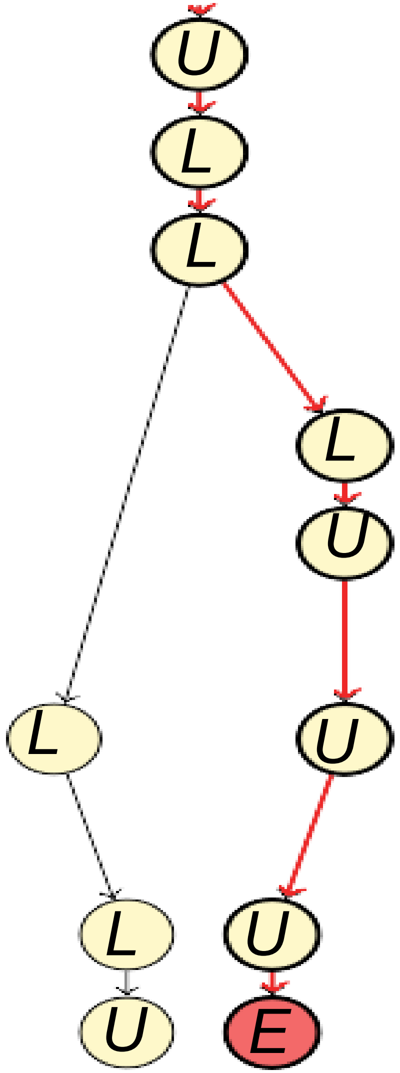
Refinement Example



```
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;

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

KeReleaseSpinLock();
```

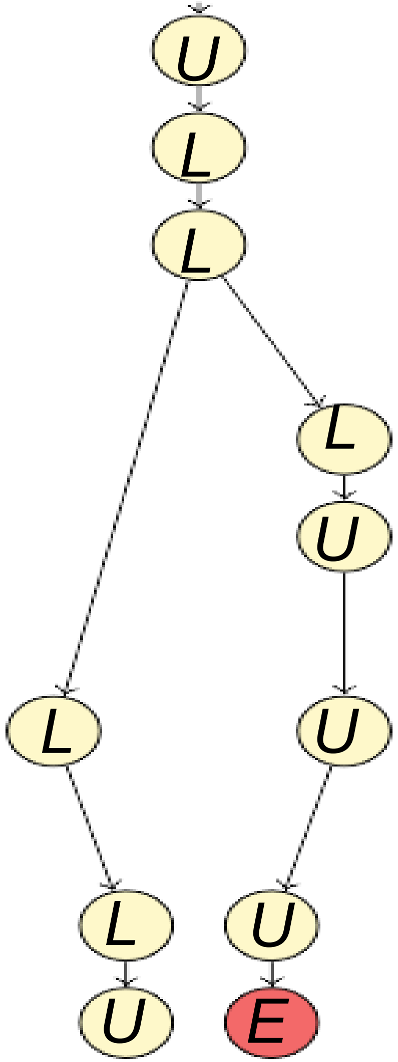
b=true;

b=b?false:*;

!b

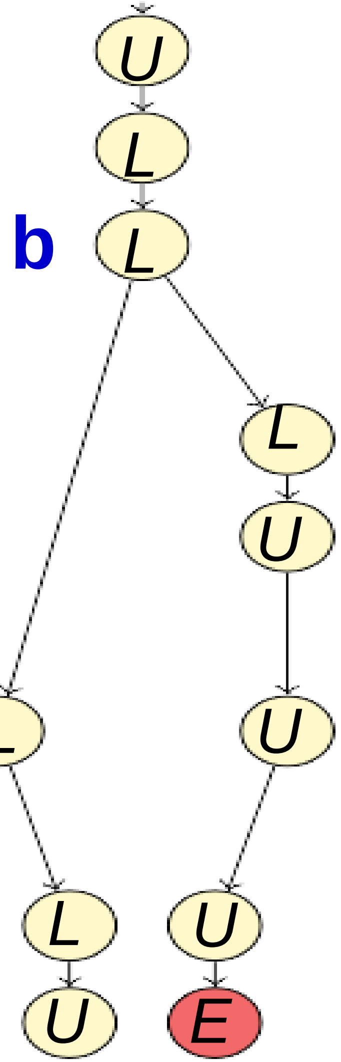
Let’s add the predicate
nPacketsOld==nPackets

Refinement Example



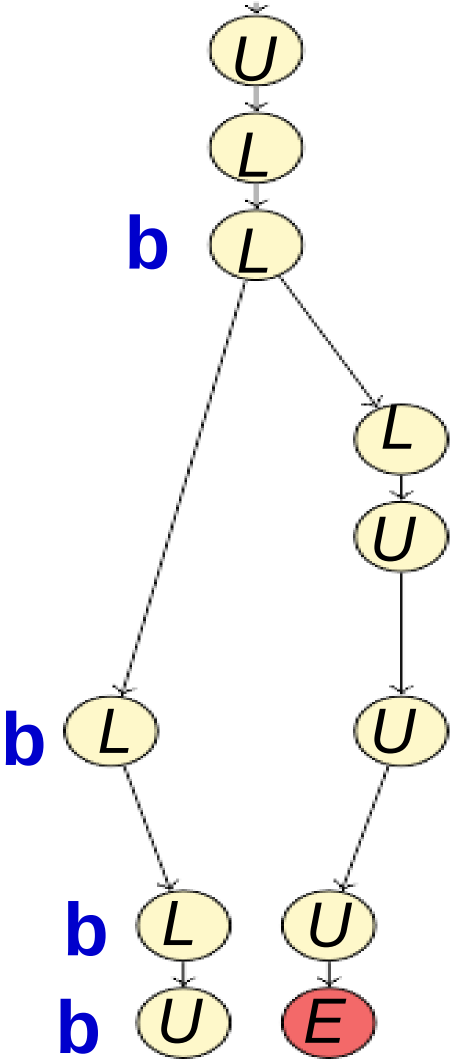
```
do {

        KeAcquireSpinLock();
         b=true;

        if (*)  {
                KeReleaseSpinLock();

                 b=b?false:*;
        }

} while(!b);

KeReleaseSpinLock();
```

Refinement Example



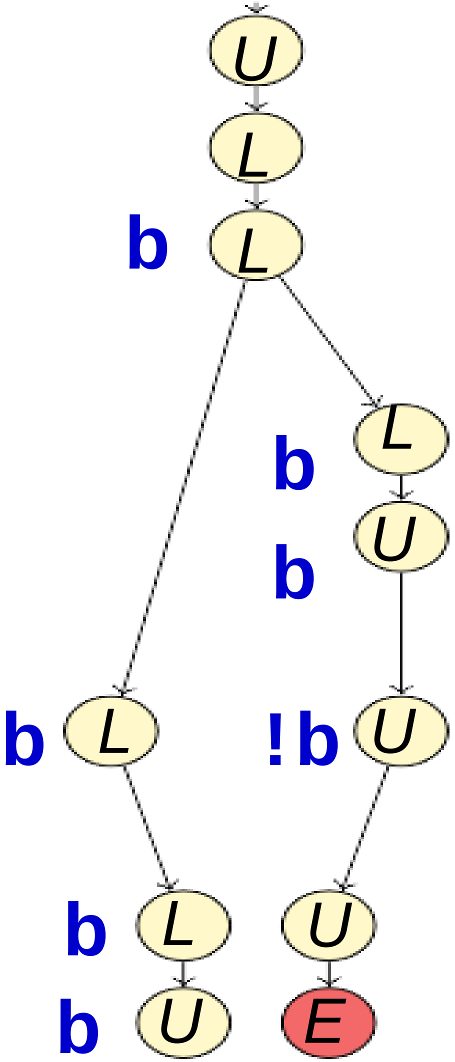
```
do {
        KeAcquireSpinLock();
        b=true;

        if (*) {
                KeReleaseSpinLock();

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

KeReleaseSpinLock();
```

Refinement Example



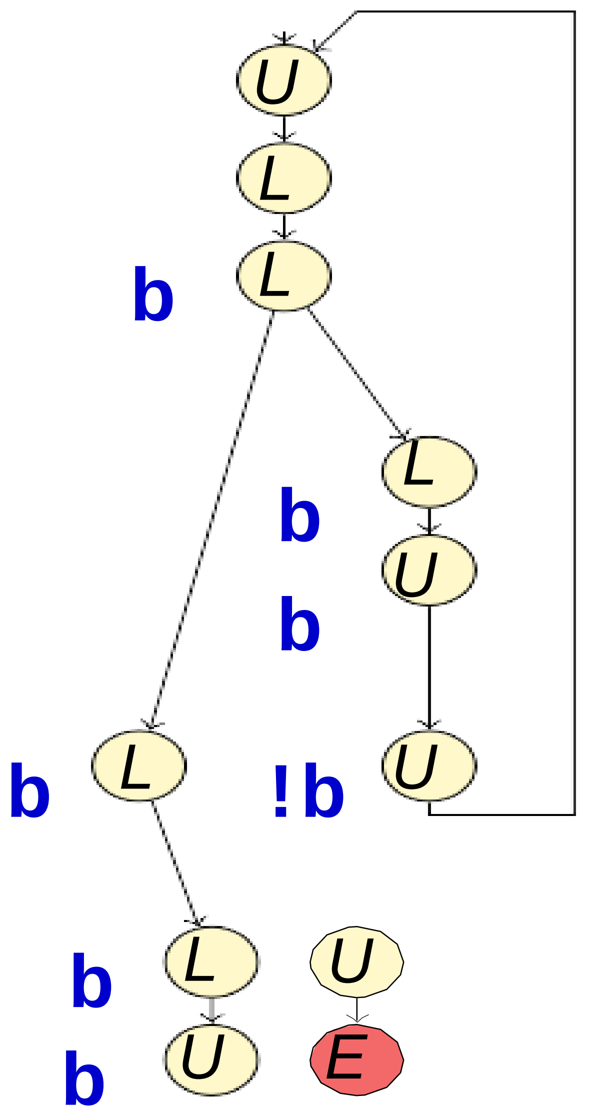
```
do {
        KeAcquireSpinLock();
         b=true;

        if (*)  {
                KeReleaseSpinLock();

                 b=b?false:*;
        }

} while( !b );

KeReleaseSpinLock();
```

Refinement Example



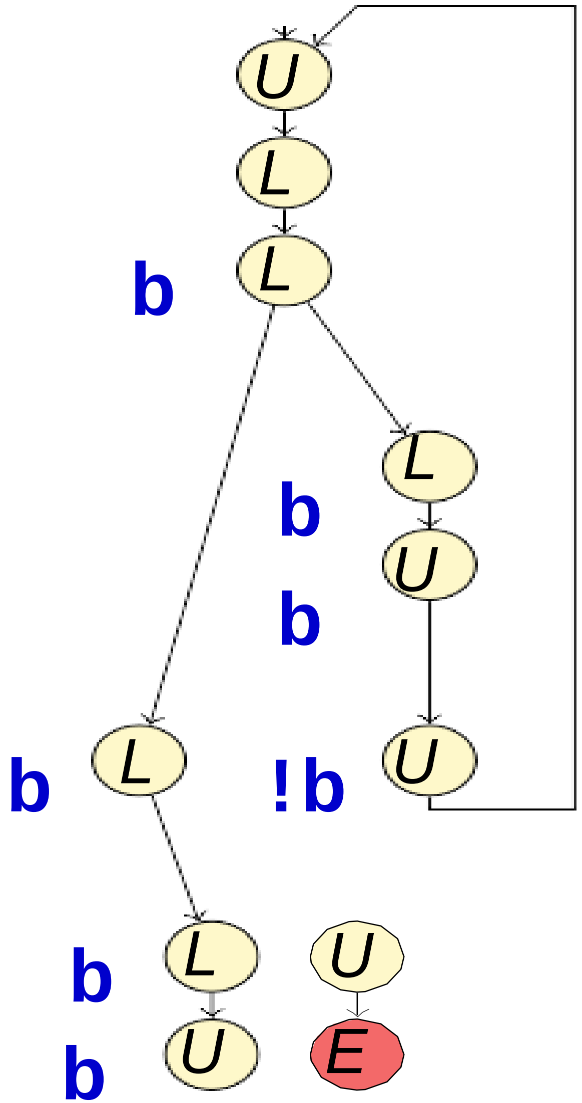
```
do {

        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



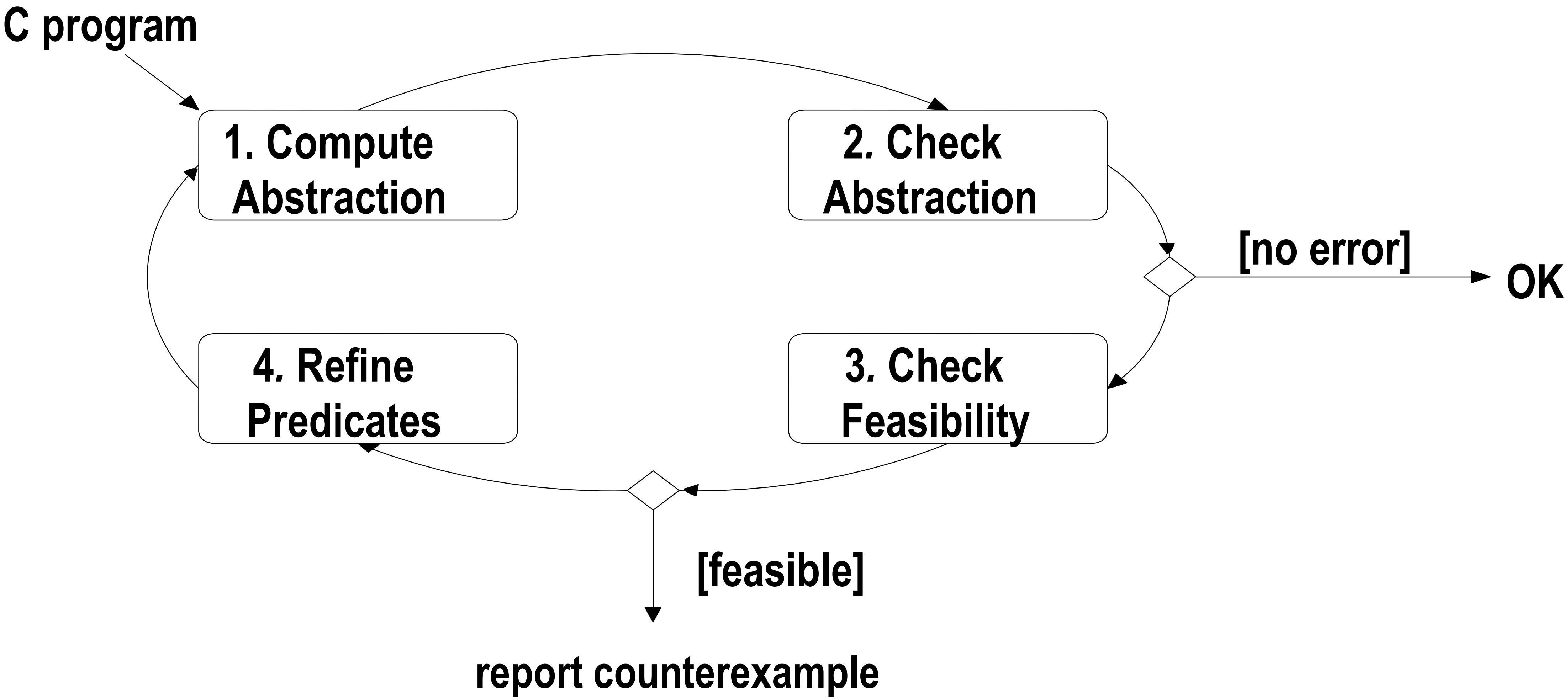
```
do {

        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



C program

1. Compute Abstraction

2. Check Abstraction

[no error]

OK

3. Check Feasibility

4. Refine Predicates

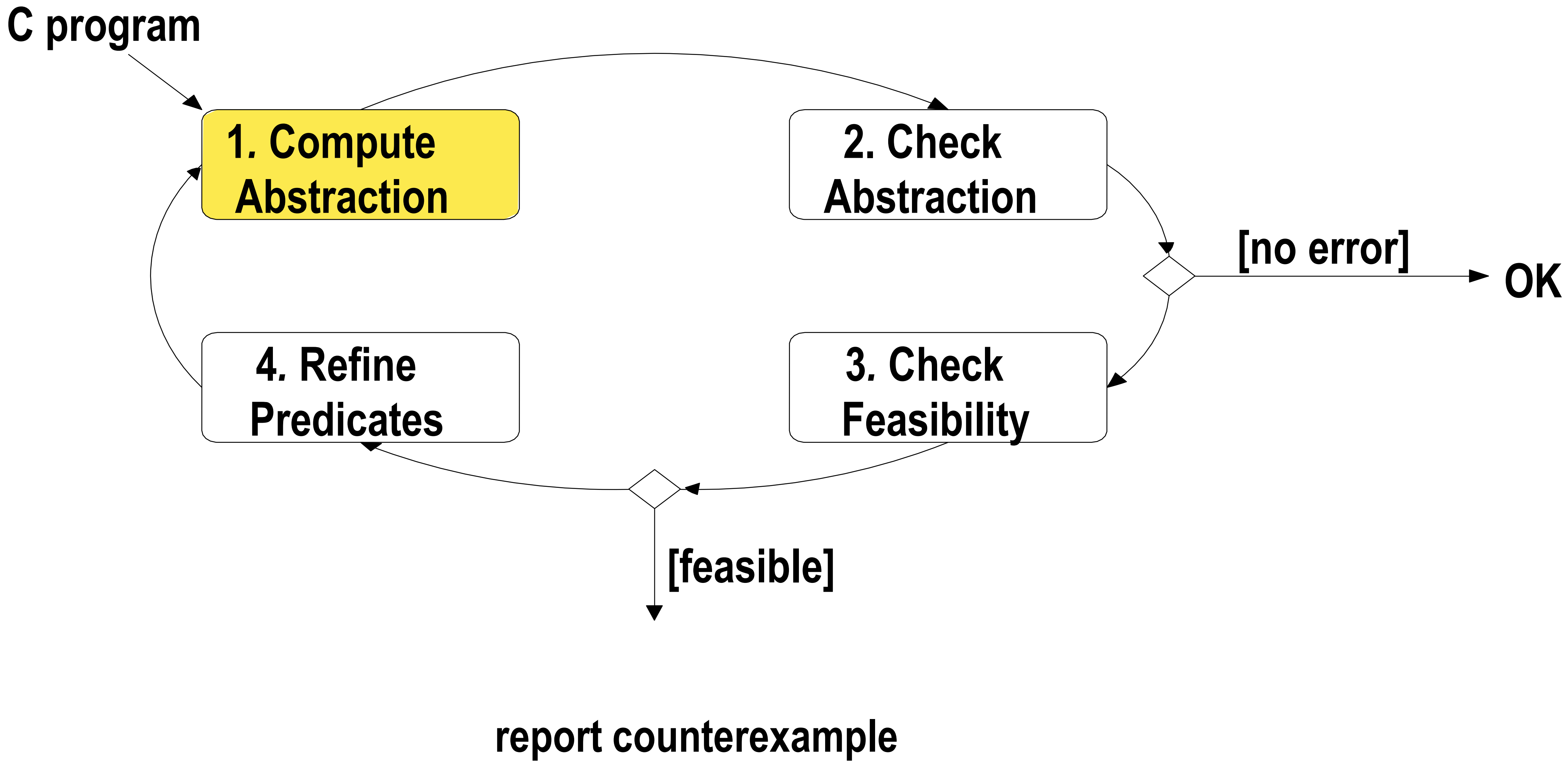
[feasible]

report counterexample

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



C program

1. Compute Abstraction

2. Check Abstraction

[no error]

OK

3. Check Feasibility

4. Refine Predicates

[feasible]

report counterexample

Computing Existential Abstractions of Programs

```
int main ( )  {
    int i ;

    i = 0 ;

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

C Program

+

$p_1 \Leftrightarrow i=0$

$p_2 \Leftrightarrow even(i)$

Predicates

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

    p1=TRUE ;
    p2=TRUE ;

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

Boolean Program

Minimal?

Predicate Images

Reminder:

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

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

$p_1 \iff i = 1$

$p_2 \iff i = 2$

$p_3 \iff \text{even}(i)$

Basic Block

i++;

$\rightarrow$

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

$\cancel{\xrightarrow{?}}$

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

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

Enumeration: Example

Predicates

$p_1 \iff i = 1$

$p_2 \iff i = 2$

$p_3 \iff \text{even}(i)$

Basic Block

`i++;`

$\rightarrow$

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 . . .

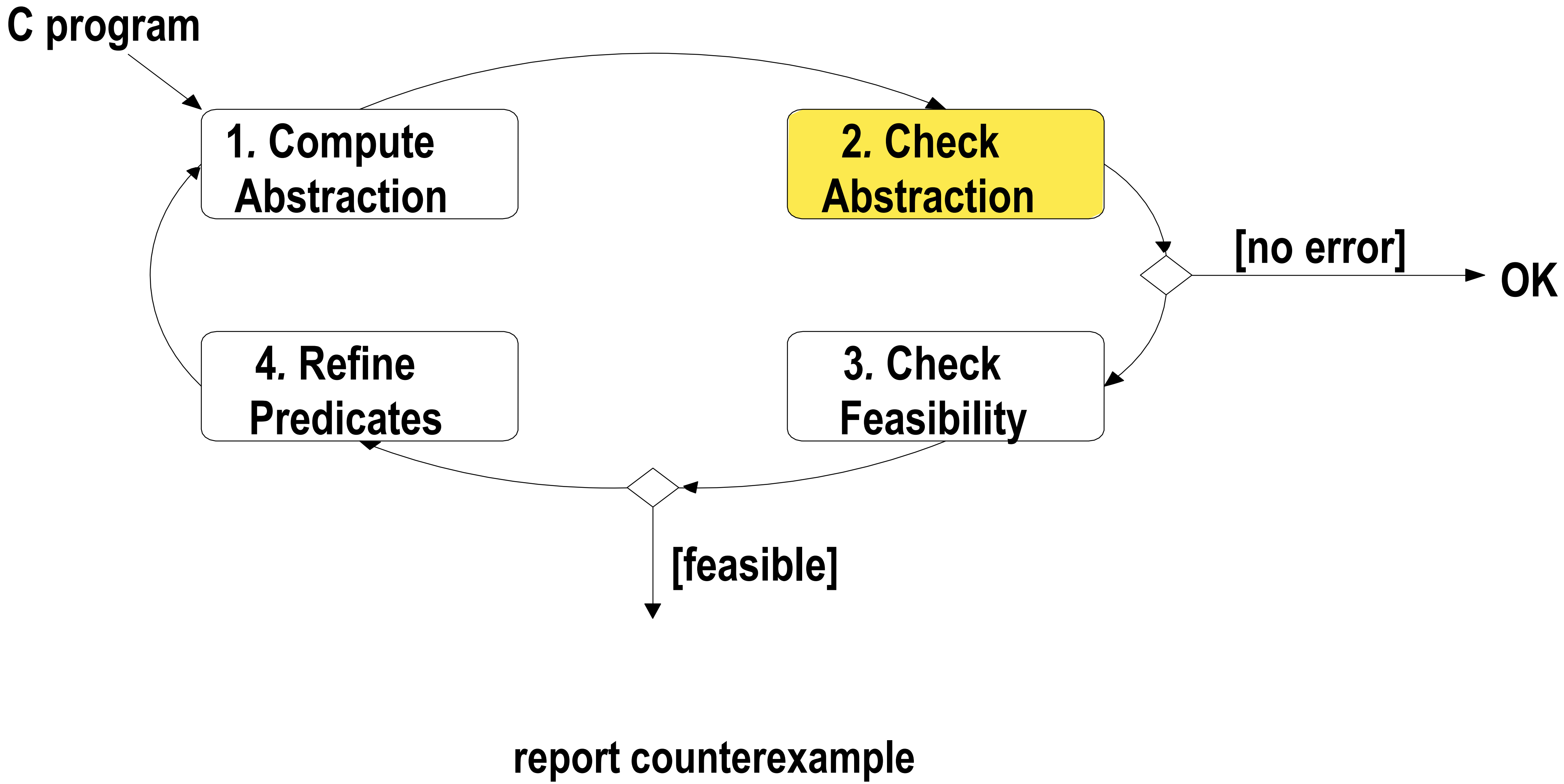
Predicate Images

☒ 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



C program

1. Compute Abstraction

2. Check Abstraction

[no error]

OK

3. Check Feasibility

4. Refine Predicates

[feasible]

report counterexample

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_g e_8 :   boolean ;                  --   i >=  8
VAR  b6_i_g e_s :   boolean ;                  --   i >=  s
VAR  b7 :   boolean ;                          --  1 +  i  >=  8
VAR  b8 :   boolean ;                          --  1 +  i  >=  s
VAR  b9_s_g t_0 :   boolean ;                  --  s > 0
VAR  b10_s_g t_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)  |  b42)
                 & (!b17  |  !b30  |  b48)
                 & (!b30  |  !b42  |  !b42  |  b54)
                 & (!b17  |  !b30
                 & (!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

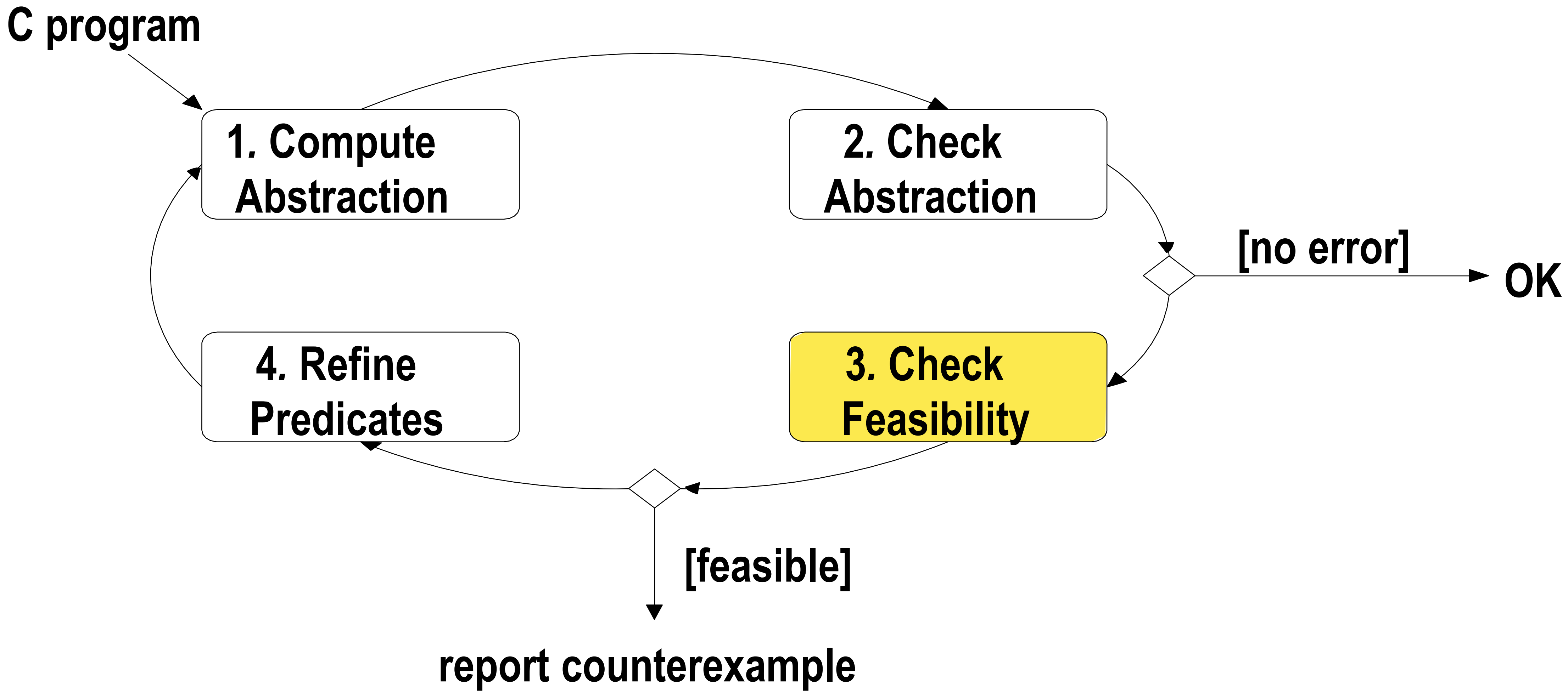
④ Property

```
--  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

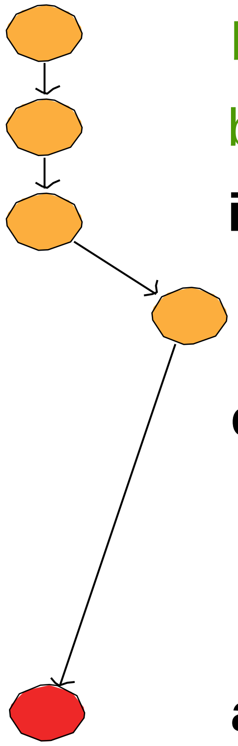
```
main() {
     bool b0;  //  y> x
     b0=*;
     b0=*;
     if (b0)
          b0=*;
     else
          b0=*;
      assert(b0);
}
```

Example Simulation

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

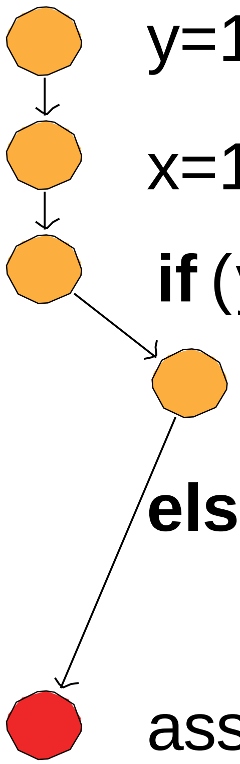
Predicate:

y> x



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

    assert(b0);
}
```

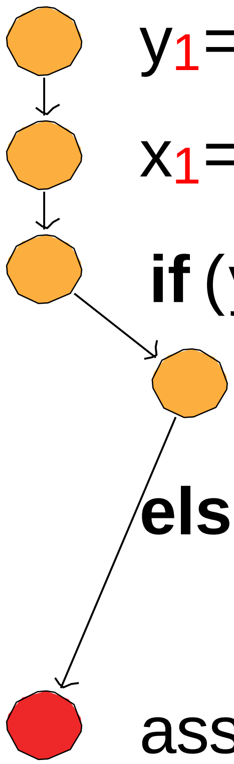
Example Simulation



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

We now do a path test, so convert to Static Single Assignment (SSA).

Example Simulation



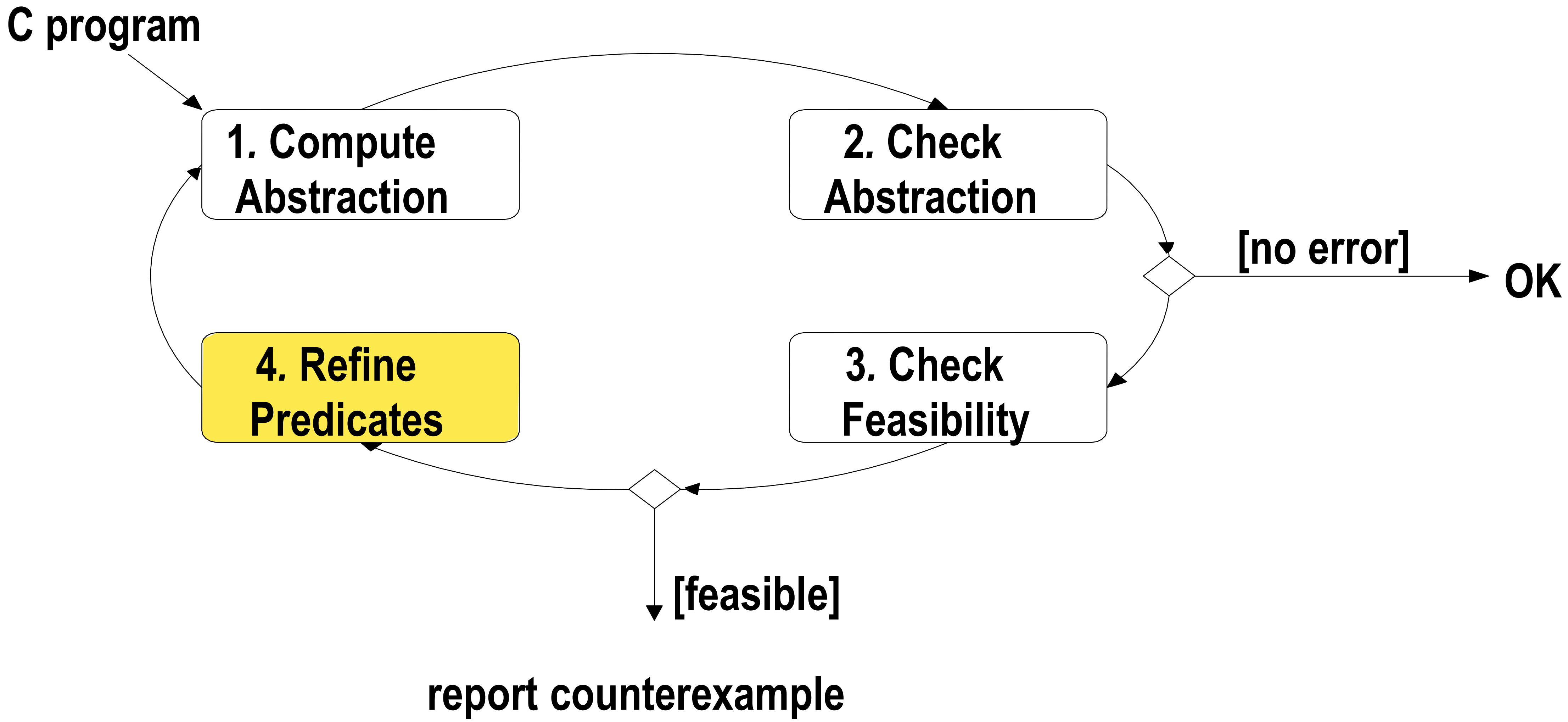
```
int  main() {
        int  x,  y;
        $y_1$=1;
        $x_1$=1;
        if ($y_1$>$x_1$)
            $y_2$=$y_1$−1;
        else
            y++;
        assert($y_2$>$x_1$);
}
```

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

$$\neg(y_2 > x_1)$$

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

Refining the Abstraction



C program

1. Compute Abstraction

2. Check Abstraction

[no error]

OK

3. Check Feasibility

4. Refine Predicates

[feasible]

report counterexample

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;

        {¬y > 1 ⇒ y + 1 > 1}

        x=1;

        {¬y > x ⇒ y + 1 > x}

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

We are using weakest pre-conditions here

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

$$wp(S\,;T,Q) = wp(S, wp(T,Q))$$

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

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 the beginning
2. Compute sp(…) along the path

- Both methods eliminate the trace
- Advantages / Disadvantages?

Approximating Loop Invariants: SP

```
int x, y;

x=y=0;

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

assert(y==10);
```

The SP refinement results in

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

$$sp(\text{x++; y++}, \ldots) \quad = \quad x = 1 \wedge y = 1$$

$$sp(\text{x++; y++}, \ldots) \quad = \quad x = 2 \wedge y = 2$$

$$sp(\text{x++; y++}, \ldots) \quad = \quad x = 3 \wedge y = 3$$

...

✓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

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

✓ Also requires 10 iterations.
✓ It won't work if we replace 10 by n.

What do we really need?

```
int x, y;

x=y=0;

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

assert(y==10);
```

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.*

What do we really need?

```
int x, y;

x=y=0;

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

assert(y==10);
```

Suppose we add a restriction = “no new constants”:

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

$x_1 = 0$
$y_1 = 0$

$x_2 = 1$
$y_2 = 1$

~~$x_3 = 2$~~
~~$y_3 = 2$~~

$x_3 = y3$
(loop invariant)

$x_4 = y4$
(loop invariant)

✔ The language restriction forces the solver to generalize!