# Software Verification and Abstraction

Rupak Majumdar

UCLA

Max
Planck
Institute
for
Software Systems

Kaiserslautern · Saarbrücken // Germany

# Lecture 1: Model Checking
# Basic Concepts

Rupak Majumdar

Model checking, generally interpreted:

Automatic algorithmic techniques for system verification which operate on a system model (semantics)

# Somewhat General View

Model checking, narrowly interpreted:

Decision procedures for checking if a given Kripke structure is a model for a given formula of a modal logic (CTL or LTL).

Our view includes

- Dataflow analysis in compilers
- Symbolic execution based methods

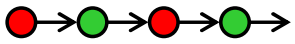
Our view excludes

- Language design for ensuring properties
- Proof calculi and interactive theorem proving

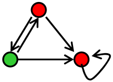
There are many different model checking algorithms, depending on

- The system model
- The specification formalism

# Discrete Systems Theory



**Trajectory:** dynamic evolution of state
sequence of states



**Model:** generates a set of trajectories
transition graph

**Property:** assigns boolean values to trajectories
temporal logic formula

"red and green alternate"

**Algorithm:** compute values of the trajectories generated by a model

# Paradigmatic Example: Mutual Exclusion

```
loop
  out:  x1 := 1; last := 1
  req:  await  x2 = 0  or  last = 2
  in:    x1 := 0
end loop.
```

P1

||

```
loop
  out:  x2 := 1; last := 2
  req:  await  x1 = 0  or  last = 1
  in:    x2 := 0
end loop.
```

P2

Property: It is never the case that P1 and P2 are both at `in’

# System Modeling

- Various factors influence choice of model
  - State based vs event based
  - Concurrency model

- While the choice of system model is important for ease of modeling in a given situation,

the only thing that is important for model checking is that the system model can be translated into some form of state-transition graph.

- So: Will not focus much on syntactic constructs

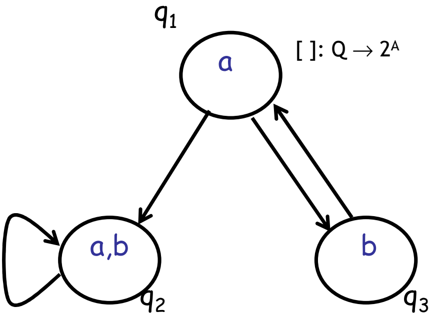
# Syntax: Finite State Programs

- Parallel composition of C programs, without function calls
- Each variable has a finite range

- We'll write such programs as guarded commands

# Semantics: State Transition Graph

| | | |
|---|---|---|
| $Q$ | set of states | $\{q_1,q_2,q_3\}$ |
| $q_0$ | initial state | $q_1$ |
| $\rightarrow \subseteq Q \times Q$ | transition relation | $q_1 \rightarrow q_2$ |
| $A$ | set of atomic observations | $\{a,b\}$ |
| $[\ ]: Q \rightarrow 2^A$ | observation function | $[q_1] = \{a\}$ |



# Important Restriction

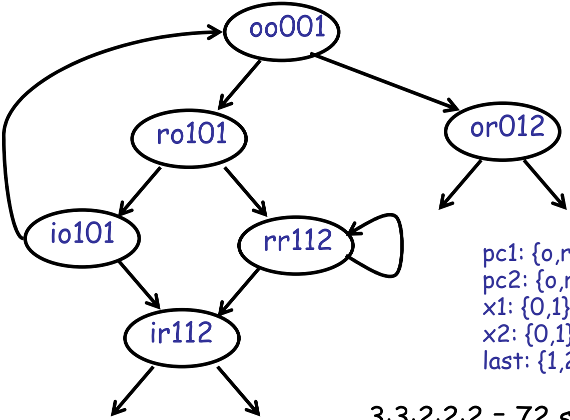
Until notified, restrict attention to
finite-state transition systems

Q is finite

# Example: Mutual Exclusion

```
loop                                          ||  loop
  out:  x1 := 1; last := 1                          out:  x2 := 1; last := 2
  req:  await  x2 = 0  or  last = 2                 req:  await  x1 = 0  or  last = 1
  in:    x1 := 0                                    in:    x2 := 0
end loop.                                         end loop.

     P1                                                  P2
```



pc1: {o,r,I}
pc2: {o,r,i}
x1: {0,1}
x2: {0,1}
last: {1,2}

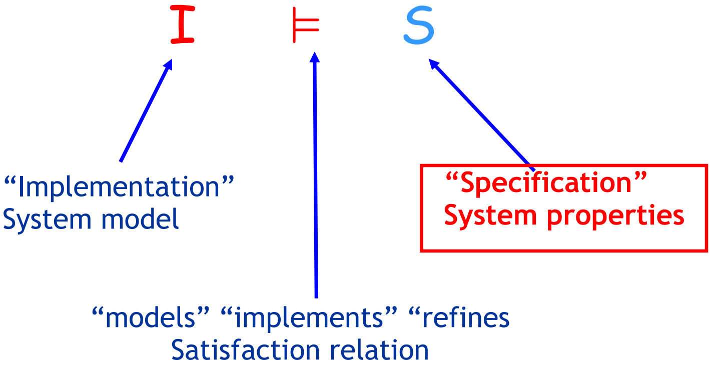
3·3·2·2·2 = 72 states

# State Explosion Problem

The translation from a system description to a state-transition graph usually involves an exponential blow-up !!!

e.g., n boolean variables $\Rightarrow$ $2^n$ states

# System Verification Problem



$I \models S$

“Implementation”
System model

“models” “implements” “refines
Satisfaction relation

**“Specification”**
**System properties**

# System Properties

Some orthogonal dimensions in choosing specification formalisms

1 operational vs. declarative:

automata vs. logic

2 may vs. must:

branching vs. linear time

2 prohibiting bad vs. desiring good behavior:

safety vs. liveness

The three decisions are orthogonal, and they lead to substantially different model-checking problems

# Safety vs Liveness

- Safety: Something “bad” will never happen
  - Program does not produce bad result

    “partial correctness”
  - *Example*: *Mutual exclusion*

- Liveness: Something “good” eventually happens
  - The program produces a result “termination”
  - *Example*: *A process wanting to go to the critical section eventually gets in*

# Safety vs Liveness Contd.

- Safety: those properties whose violation always has a finite witness
  - “if something bad happens on an infinite run, then it happens already on some finite prefix” --- Can be checked on finite runs
- Liveness: those properties whose violation never has a finite witness
  - “no matter what happens along a finite run, something good could still happen later” --- Must be checked on infinite runs

# Two Remarks

1. The vast majority of properties to be verified are safety

2. While nobody will ever observe the violation of a true liveness property, liveness is a useful abstraction that turns complicated safety into simple liveness

Accordingly, we focus on safety for most of the lectures

# Safety Model Checking

- Requirement: The system should always stay within some safe region

- Input: A state transition graph
- Input: A set of good states “invariants”

- Output: “Safe” if all executions maintain the invariant, “Unsafe” otherwise (and a trace)

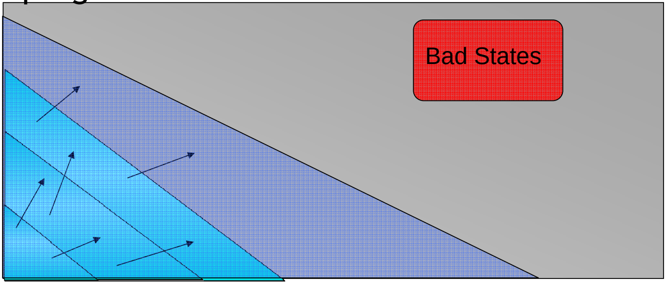
# From Safety to Reachability

- Input: A state transition graph
- Input: A set of bad states

- Output: “Safe” if there is no run from an initial state to any bad state, “Unsafe” otherwise (and a trace)

# Model Checking Algorithm

- Graph Search
  - Linear time in the size of the graph
  - Exponential time in the size of the program



# Enumerative Model Checking

- Provide access to each state
- For each state, provide access to neighboring states

- Implement classical graph algorithms
  - Depth-first or breadth-first search
  - Starting from initial states and searching forward for bad states
  - Or starting from bad states and searching backward for initial states

# State Space Explosion

- Biggest problem is state space explosion
  - N bits ⇒ $2^N$ states
- Many heuristics
  - Search on-the-fly,
  - partial order and symmetry reduction
  - Do not store dead variables

- Many successful implementations
    - Spin, Murphi, Verisoft, … [Protocol verification]

# Symbolic Model Checking

- Idea: Work with sets of states, rather than individual states

```
Given: Transition graph G, target states $\sigma^T$
begin
 -  $\sigma^R$ = set of Initial states
 - repeat forever
     if $\sigma^R \cap \sigma^T \neq \emptyset$ then return “yes”
     if $Post(\sigma^R) \subseteq \sigma^R$ then return “no”
     $\sigma^R := \sigma^R \cup Post(\sigma^R)$
   end
```

Here, $Post(\sigma) = \{s' \mid \exists s \in \sigma.\ s \rightarrow s'\}$

# Encoding Sets through Formulas

- Idea: Represent sets of states symbolically, using constraints
- E.g., $1 \leq x \leq 100$ represents the 100 states $x = 1$, $x = 2$, …, $x = 100$
- Represent both sets of initial states and transition relation implicitly

# Representing States as *Formulas*

| | |
|---|---|
| $[F]$<br>states satisfying $F$ $\{s \mid s \models F\}$ | $F$<br>FO fmla over prog. vars |
| $[F_1] \cap [F_2]$ | $F_1 \wedge F_2$ |
| $[F_1] \cup [F_2]$ | $F_1 \vee F_2$ |
| $\overline{[F]}$ | $\neg F$ |
| $[F_1] \subseteq [F_2]$ | $F_1$ implies $F_2$ |

i.e. $F_1 \wedge \neg F_2$ unsatisfiable

# Symbolic Transition Graph

- A transition graph
  - A Formula Init(x) representing initial states
  - A Formula TR(x,x’) representing the transition relation

- Example: C program

| | |
|---|---|
| x:=e | TR(x,x’): $loc=pc \wedge loc'=pc' \wedge x' = e \wedge \{ y'=y \mid y \neq x \}$ |
| Assume(p) | TR(x,x’): $loc=pc \wedge loc'=pc' \wedge p$ |

# Symbolic Transition Graph

- Operations:
  - Post(X) $= \{s' \mid \exists s \in X.\ s \to s'\}$
    $= \exists s.\ X(s) \wedge TR(s,s')$
  - Pre(X) $= \{s \mid \exists s' \in X.\ s \to s'\}$
    $= \exists s'.\ TR(s,s') \wedge X(s')$
- Can implement using formula manipulations

# Symbolic Model Checking

```
Given: Transition graph G, target states $\sigma^T$
begin
 –  $\sigma^R$ = Formula representing set of Initial states
 –  repeat forever
      if $\sigma^R \wedge \sigma^T$ is satisfiable then return “yes”
      if $Post(\sigma^R) \Rightarrow \sigma^R$ then return “no”
      $\sigma^R := \sigma^R \vee Post(\sigma^R)$
    end
```

Here, $\mathrm{Post}(\sigma)(s') = \exists s.\ \sigma(s) \wedge \mathrm{TR}(s,s')$

Can be implemented using decision procedures for the language of formulas

# Finite State Systems

- Symbolic representation in propositional logic
- State described by n bits X
- A region is a propositional formula with free variables in X

- Can implement symbolic operations using propositional formula manipulations

# Example: Mutual Exclusion

```
loop                                        ||  loop
  out:  x1 := 1; last := 1                        out:  x2 := 1; last := 2
  req:  await  x2 = 0  or  last = 2               req:  await  x1 = 0  or  last = 1
  in:    x1 := 0                                  in:    x2 := 0
end loop.                                       end loop.
```

Symbolic representation has variables

$$pc1, pc2, x1, x2, last$$

Initial states:

$$pc1=out \ \wedge pc2=out \wedge x1=0 \wedge x2=0$$

No constraint on last

Transition relation:

$$pc1=out \wedge x1'=1 \wedge last'=1 \wedge pc2'=pc2 \wedge x2'=x2$$

$$\vee \ \ldots$$

# Additional Desirable Properties

- All operations must be efficient in practice
- Should maintain compactness whenever possible
- Canonical representations

- Representing initial states and transition relation from the program description should be efficient

# Binary Decision Diagrams

- Efficient representations of boolean functions

[Bryant86]

- Share commonalities
- Ordered BDDs:
  - Fix a linear ordering of the variables in X
  - BDD = DAG, with nodes labeled with boolean variables
  - Each variable occurs 0 or 1 times along a path
  - Paths in the DAG encode assignments to variables

- Extremely successful in hardware verification

# More on Safety Properties

- Not all safety properties can be written as invariants on the program state space
- For example, if correctness depends on the order of events
  - Locks can be acquired and released in alternation, it is an error to acquire/release a lock twice in succession without an intermediate release / acquire

# Monitors

- Write the ordering of events as an automaton (called the monitor)
- Take the product of the system with the monitor
  - The monitor tracks the sequence of events
  - It goes to a special “bad” state if a bad sequence occurs
- Now we can express the property as an invariant: the monitor state is never bad

# Symbolic Search

- Guaranteed to terminate for finite state systems
- And can be applied to infinite state systems as well
  - Although without guarantees of termination in general
  - Application to infinite state requires richer languages for formulas and associated decision procedures

# What about Software?

- Can construct an infinite state transition system from a program

- States: The state of the program
  - (stack, heap, pc location)

- Transitions: $q \to q'$ iff in the operational semantics, there is a transition of the program from $q$ to $q'$

- Initial state: Initial state of the program

# Termination

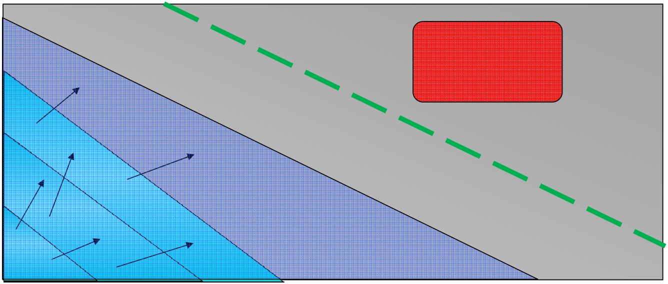
- Each operation can be computed

- But iterating Pre or Post operations may not terminate

- What do we do now?

# Observation

- Often, we do not need the exact set of reachable states
  - We need a set of states that separates the reachable states from the bad states



# One Possibility

- User gives an estimate (inductive invariant)

  A set of states Inv such that

  - Init $\subseteq$ Inv - Inv $\cap$ bad = $\emptyset$ - Post(Inv) $\subseteq$ Inv

* Can show that this implies system is safe (How?)

* Given Inv, and decision procedures, this procedure is guaranteed to terminate

- This is the idea of classical loop invariants
  - Problem: In general, it can be hard to manually construct Inv

# Before we proceed

- What is the sign of the following product:

  - 12433454628 * 94329545771 ?

# Idea

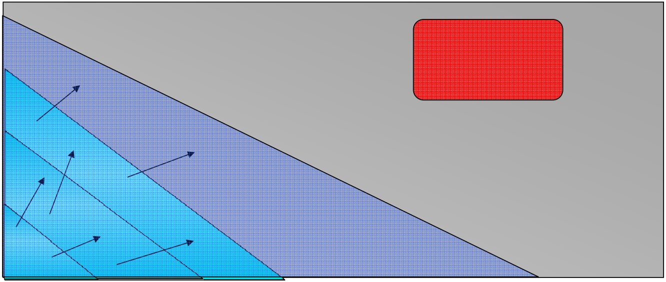
- One can “abstract” the behavior of the system, and yet reason about certain aspects of the program

- Abstraction:

  -ve * +ve = -ve

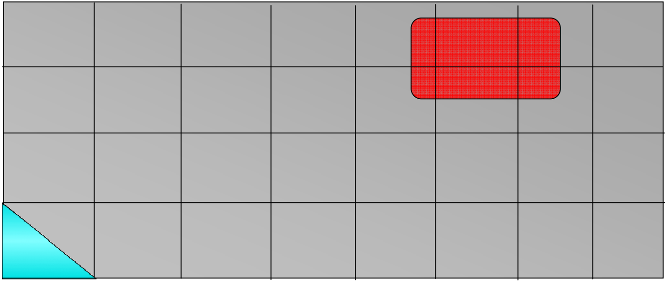
# Model Checking Algorithm

- Graph Search



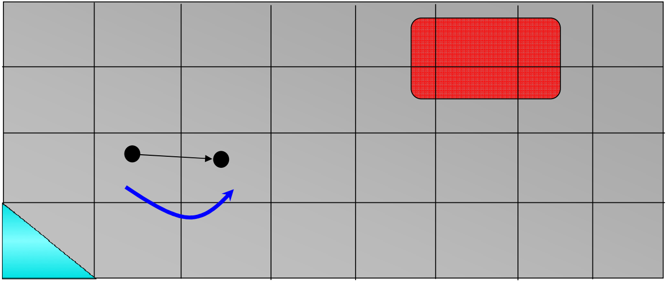
# Abstract Interpretation

- The state transition graph is large/infinite
- Suppose we put a finite grid on top



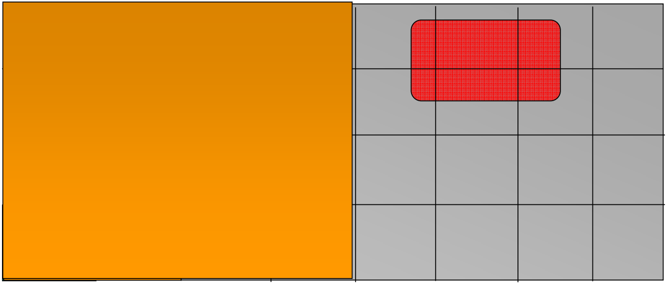
# Existential Abstraction

- Every time s → s’, we put [s] → [s’]
- This allows more behaviors



# Abstract Model Checking

- Search the abstract graph until fixpoint
  - Can be much smaller than original graph
  - Can be finite, when original is infinite



# Simulation Relations

- A relation $\preceq \subseteq Q \times Q$ is a simulation relation if $s \preceq s'$ implies
  - Observation(s) = Observation(s')
  - For all t such that $s \rightarrow t$

    there exists t' such that $s' \rightarrow t'$
    and $s' \preceq t'$

Formally captures notion of “more behaviors”
Implies containment of reachable behaviors

# Main Theorem

- $s \preceq [s]$ is a simulation relation
- If an error is unreachable in Abs(G) then it is unreachable in G
- Plan:

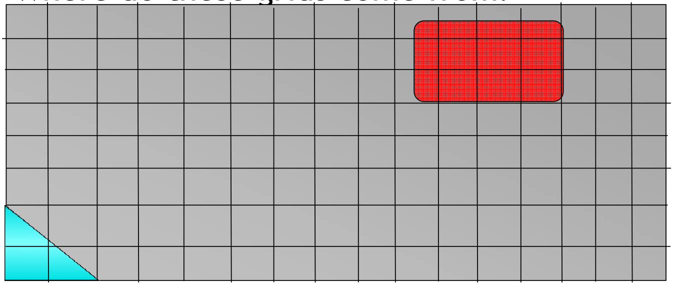
1. Find a suitable grid to make the graph finite state
2. Run the finite-state model checking algorithm on this abstract graph
3. If abstract graph is safe, say “safe” and stop

## What if the Abstract Graph says Unsafe?

- The error may or may not be reachable in the actual system
  - Stop and say “Don’t know”

# What if the Abstract Graph says Unsafe?

- Or, put a finer grid on the state space
- And try again
  - The set of abstract reachable states is smaller
  - Where do these grids come from?



# Grids: Predicate Abstraction

- Suppose we fix a set of facts about program variables
  - E.g., old = new, lock = 0, lock = 1

- Grid: Two states of the program are equivalent if they agree on the values of all predicates
  - N predicates = $2^N$ abstract states

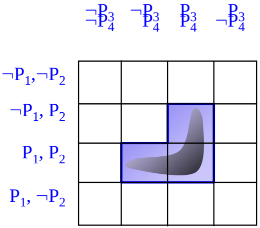
- How do we compute the grid from the program?

# Predicate Abstraction

Region Representation: formulas over predicates

$P_1 : x = y$ $P_2 : z = t + y$

$P_3 : x \leq z+1$ $P_4 : *u = x$

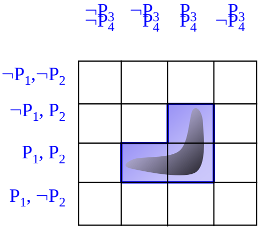


Karnaugh Map

Set of states

**Abstract Set: $P_1P_2P_4 \vee \neg P_1 P_2 P_3 P_4$**

# Predicate Abstraction



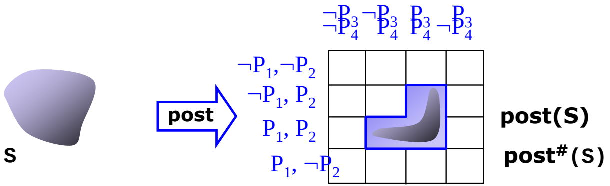
Karnaugh Map

$P_1 : x = y$ $P_2 : z = t + y$

$P_3 : x \leq z+1$ $P_4 : *u = x$

- Box: abstract variable valuation
- BoxCover(S): Set of boxes covering S
- Theorem prover used to compute BoxCover

# Post$^\#$, Pre

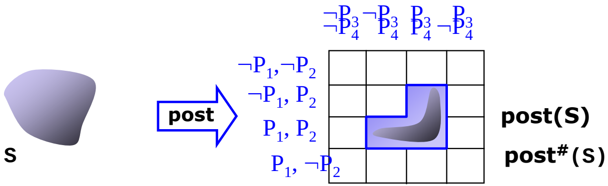


- pre(S,op) = { s | $\exists$s'$\in$S. s $\rightarrow^{op}$ s'} (Weakest Precondition)
- post(S,op) = { s | $\exists$s'$\in$S. s' $\rightarrow^{op}$ s} (Strongest Postcondition)

- Abstract Operators: post$^\#$

  post(S,op) $\subseteq$ post$^\#$(S,op)

# Computing Post$^{\#}$



- For each predicate p, check if
  - S$\Rightarrow$ Pre(p, op) then have a conjunct p
  - S$\Rightarrow$ Pre(¬ p, op) then have a conjunct ¬ p
  - Else have no conjunct corresponding to p
- Use a theorem prover for these queries

# Example

- I have predicates
  - lock=0, new=old, lock=1
- My current region is lock = 0 $\wedge$ new= old
- Consider the assignment new = new+1

- What is abstract post?

# Example

- WP(new:=new+1, lock=0) is lock=0
- WP(new:=new+1, lock=1) is lock=1
- WP(new:=new+1, new=old) is new+1=old

- $\text{lock}=0 \wedge \text{new}=\text{old} \Rightarrow \text{lock} = 0$ YES
- $\text{lock}=0 \wedge \text{new}=\text{old} \Rightarrow \text{lock} \neq 0$ NO
- $\text{lock}=0 \wedge \text{new}=\text{old} \Rightarrow \text{lock} = 1$ NO
- $\text{lock}=0 \wedge \text{new}=\text{old} \Rightarrow \text{lock} \neq 1$ YES
- $\text{lock}=0 \wedge \text{new}=\text{old} \Rightarrow \text{new}+1=\text{old}$ NO
- $\text{lock}=0 \wedge \text{new}=\text{old} \Rightarrow \text{new}+1 \neq \text{old}$ YES

- So post is $\text{lock} = 0 \wedge \text{lock} \neq 1 \wedge \text{new} \neq \text{old}$

# Symbolic Search with Predicates

Symbolic representation:
Boolean formulas of (fixed set of) predicates

- Boolean operations: easy
- Emptiness check: Decision procedures

- Post: The abstract post computation algorithm

- Can now implement symbolic reachability search!

# Big Question

- Who gives us these predicates?

- Answer 1: The user
    - Manual abstractions
        - Given a program and property, the user figures out what are the interesting predicates

    - Dataflow analysis
        - For “generic” properties, come up with a family of predicates that are likely to be sufficient for most programs

# Abstract Interpretation

- Abstract model checking is formalized through abstract interpretation
  - Formalizes and unifies semantics-based program analysis

# More Approximations

- Many program dataflow analyses do not perform exact reachability analysis on the abstract state space

- Instead, use the structure of the control flow graph to further approximate the result

# Example: Flow Sensitive Analysis

- For each control flow node, keep track of the set of reachable states (along any program path) to that node
  - Information may be lost at merge points by abstracting $\vee$ by something coarser
- Assumption: All paths of the control flow graph can be executed
  - Ignore conditional statements

# Flow Insensitive Analysis

- Even more approximate
- Disregard the order of operations in the program!

- Much faster analysis than abstract model checking
  - But results are much cruder of course!
  - Can still be useful: e.g., primary way to perform alias analysis

When I run a model checker, it goes to compute the result and never comes back. When I run a dataflow analysis, it comes back immediately and says “Don’t know”!

– Patrick Cousot

# Lecture 2:
# Software Model Checking
# and
# Counterexample-Guided Refinement

Rupak Majumdar

# Recap

- Model checking is an algorithmic technique to verify properties of systems

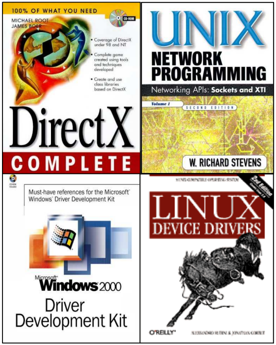
- In conjunction with abstractions, can be effective in proving subtle properties

- Today: Consider the problem of abstract model checking of (sequential) software implementations

# Setting: Property Checking

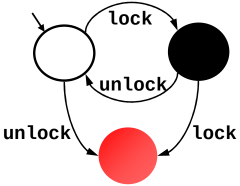
- Programmer gives partial specifications
- Code checked for consistency w/ spec
- Different from program correctness
  - Specifications are not complete
  - Is there a complete spec for Word ? Emacs ?

# Interface Usage Rules



- Rules in documentation
  - Order of operations & data access
  - Resource management
  - Incomplete, unenforced, wordy
- Violated rules ⇒ bad behavior
  - System crash or deadlock
  - Unexpected exceptions
  - Failed runtime checks

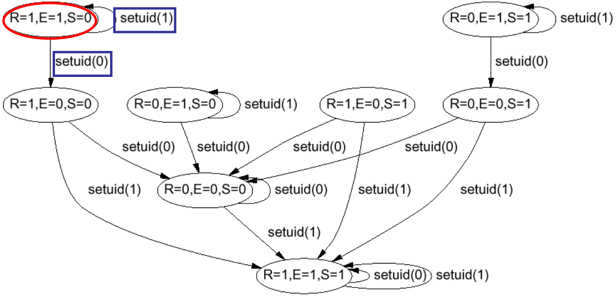
# Property 1: Double Locking



*"An attempt to re-acquire an acquired lock or release a released lock will cause a **deadlock**."*

Calls to lock and unlock must **alternate**.

# Property 2: Drop Root Privilege

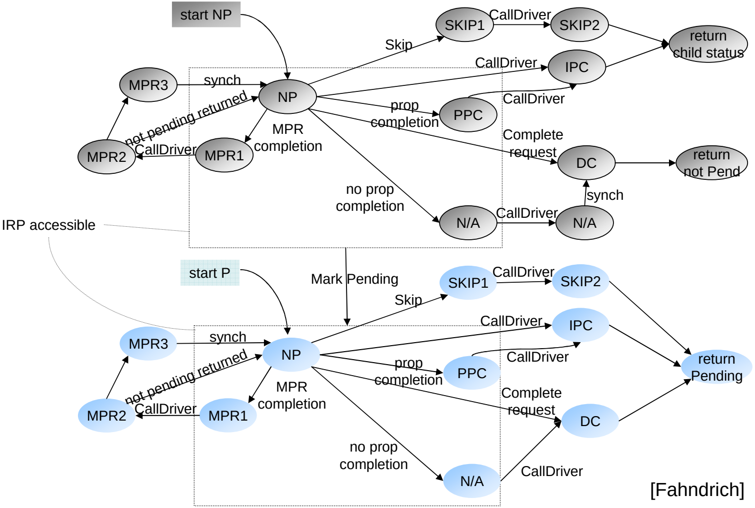


[Chen-Dean-Wagner ’02]

*“User applications must not run with root privilege”*

When **execv** is called, must have **suid ≠ 0**

# Property 3 : IRP Handler



[Fahndrich]

# Does a given usage rule hold?

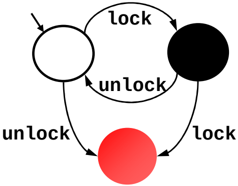
- Undecidable!
  - Equivalent to the halting problem

- Restricted computable versions are prohibitively expensive (PSPACE)

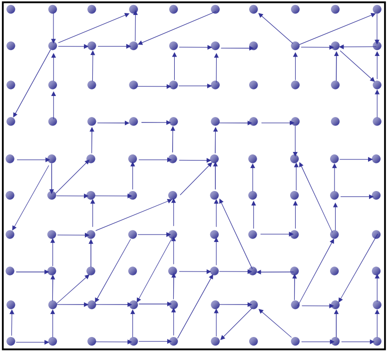
- Why bother ?
  - Just because a problem is undecidable, it doesn’t go away!

# Example

```
Example ( ) {
1: do{
         lock();
         old = new;
         q = q->next;
2:       if (q != NULL){
3:           q->data = new;
             unlock();
             new ++;
         }
4: } while(new != old);
5:  unlock ();
    return;
}
```



# What a program *really* is…



***State***

***Transition***

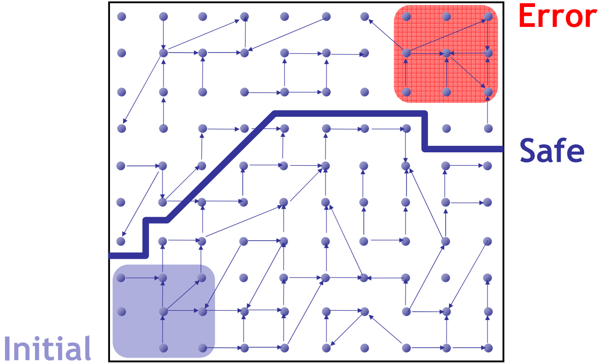
|  |  |
|---|---|
| *pc* | ☐ 3 |
| lock | ☐ ● |
| old | ☐ 5 |
| new | ☐ 5 |
| q | ☐ 0x133a |

```
3: unlock();
   new++;
4:} …
```

|  |  |
|---|---|
| *pc* | ☐ 4 |
| lock | ☐ ○ |
| old | ☐ 5 |
| new | ☐ 6 |
| q | ☐ 0x133a |

```
Example ( ) {
1: do{
      lock();
      old = new;
          q = q->next;
2:    if (q != NULL){
3:             q->data = new;
               unlock();
         new ++;
      }
4: } while(new != old);
5:  unlock ();
    return;}
```

# The Safety Verification Problem

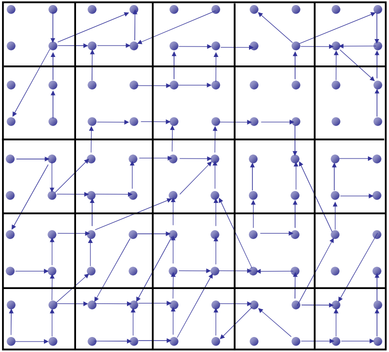


Is there a **path** from an **initial** to an **error** state ?

**Problem: Infinite** state graph

**Solution** : **Set** of states $\simeq$ logical **formula**

# Idea 1: Predicate Abstraction



- **Predicates** on program state:

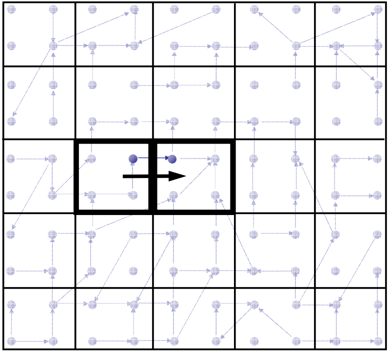
  *lock*

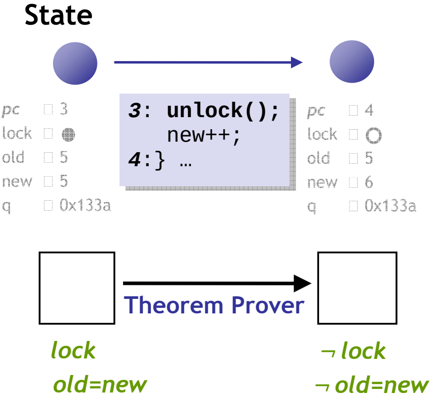
  *old = new*

- States satisfying **same** predicates are **equivalent**
  - **Merged** into one **abstract state**

- #abstract states is **finite**

# Abstract States and Transitions





**State**

*pc* ☐ 3
lock ☐ ●
old ☐ 5
new ☐ 5
q ☐ 0x133a

```
3: unlock();
   new++;
4:} …
```

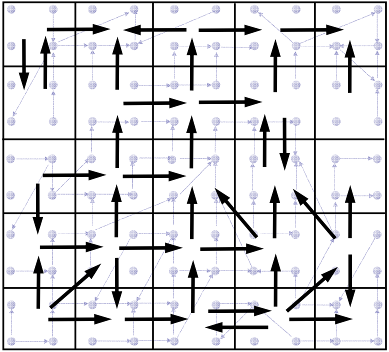
*pc* ☐ 4
lock ☐ ○
old ☐ 5
new ☐ 6
q ☐ 0x133a

**Theorem Prover**

*lock*
*old=new*

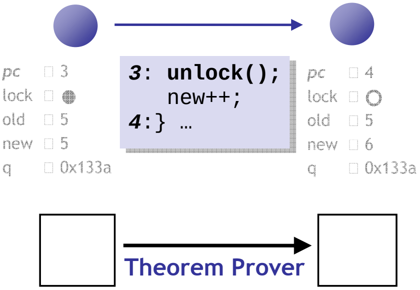
*¬ lock*
*¬ old=new*

# Abstraction



**Existential Lifting**

**State**



pc ☐ 3
lock ☐ ●
old ☐ 5
new ☐ 5
q ☐ 0x133a

```
3: unlock();
   new++;
4:} …
```

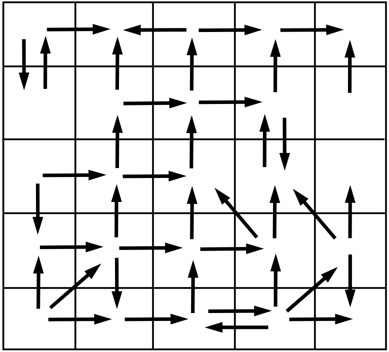
pc ☐ 4
lock ☐ ○
old ☐ 5
new ☐ 6
q ☐ 0x133a

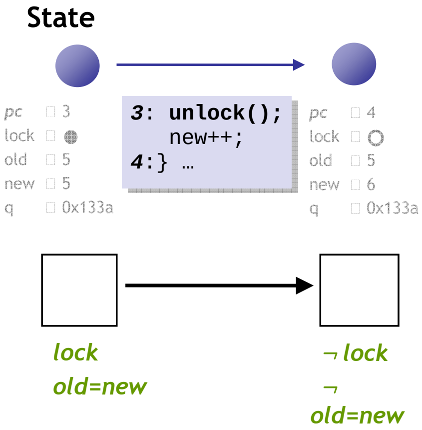
**Theorem Prover**

*lock*
*old=new*

¬ *lock*
¬ *old=new*

# Abstraction





**State**

pc ☐ 3
lock ☐ ●
old ☐ 5
new ☐ 5
q ☐ 0x133a

```
3: unlock();
   new++;
4:} …
```

pc ☐ 4
lock ☐ ○
old ☐ 5
new ☐ 6
q ☐ 0x133a

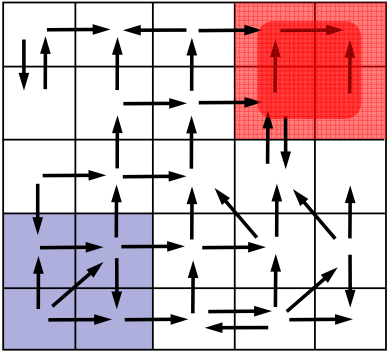
*lock*

*old=new*

¬ *lock*

¬ *old=new*

# Analyze Abstraction



Analyze finite graph

**Over** Approximate:

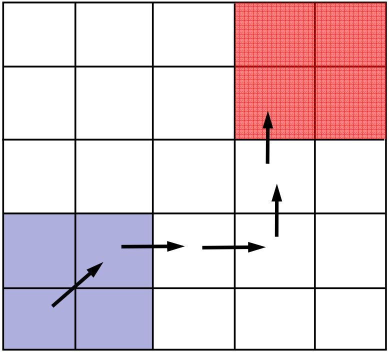
Safe $\Rightarrow$ System Safe

No **false negatives**

**Problem**

Spurious **counterexamples**

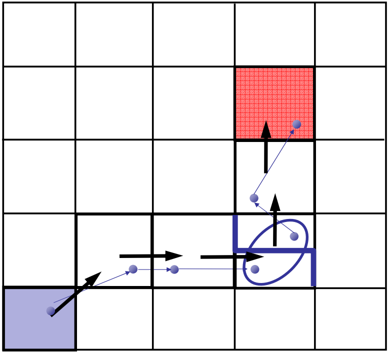
# Idea 2: Counterex.-Guided Refinement



**Solution**

Use spurious **counterexamples** to **refine** abstraction !

# Idea 2: Counterex.-Guided Refinement



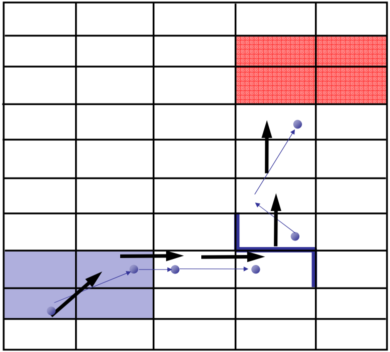
## Solution

Use spurious **counterexamples** to **refine** abstraction

1. **Add predicates** to distinguish states across **cut**
2. Build **refined** abstraction

Imprecision due to **merge**

# Iterative Abstraction-Refinement



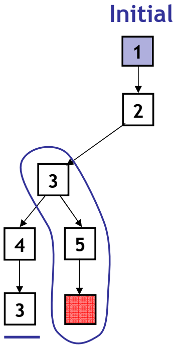
[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

**Solution**

Use spurious **counterexamples** to **refine** abstraction

1. Add predicates to distinguish states across **cut**
2. Build **refined** abstraction
   -eliminates counterexample
3. **Repeat** search
   Till real counterexample or system proved safe

# Key Idea: Reachability Tree



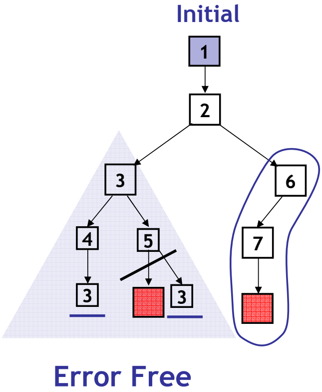
## Unroll Abstraction

1. Pick tree-node **(=abs. state)**
2. Add children **(=abs. successors)**
3. On **re-visiting** abs. state, **cut-off**

## Find min infeasible suffix

- Learn new predicates
- Rebuild subtree with new preds.

# Key Idea: Reachability Tree



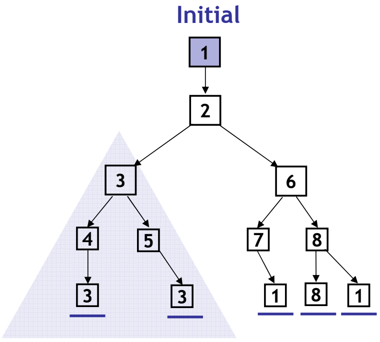
## Unroll Abstraction

1. Pick tree-node **(=abs. state)**
2. Add children **(=abs. successors)**
3. On **re-visiting** abs. state, **cut-off**

## Find min infeasible suffix

- Learn new predicates
- Rebuild subtree with new preds.

# Key Idea: Reachability Tree



## Unroll

1. Pick tree-node **(=abs. state)**
2. Add children **(=abs. successors)**
3. On **re-visiting** abs. state, **cut-off**

## Find min spurious suffix

- Learn new predicates
- Rebuild subtree with new preds.

**Error Free**

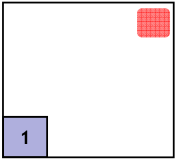
**SAFE**

**S1:** Only Abstract Reachable States

**S2:** Don't refine error-free regions

# Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```

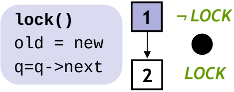


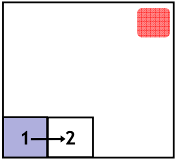
**Predicates:** *LOCK*

**Reachability Tree**

# Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5: unlock ();
}
```



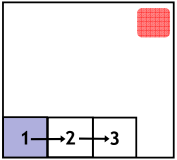


**Predicates:** *LOCK*

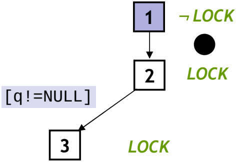
**Reachability Tree**

# Build-and-Search

```
Example ( ) {
1: do{
       lock();
       old = new;
       q = q->next;
2:     if (q != NULL){
3:        q->data = new;
          unlock();
          new ++;
       }
4:}while(new != old);
5: unlock ();
}
```



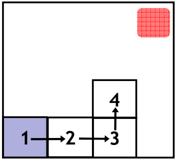
**Predicates:** *LOCK*



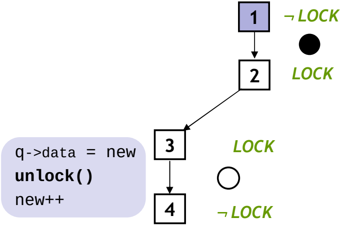
**Reachability Tree**

# Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:      q->data = new;
        unlock();
        new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



Predicates: *LOCK*

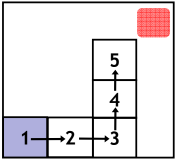


```
q->data = new
unlock()
new++
```

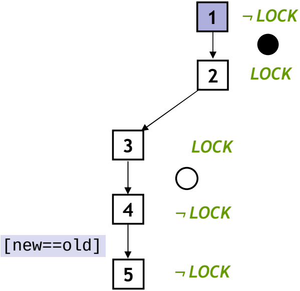
**Reachability Tree**

# Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



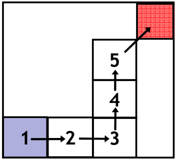
**Predicates:** *LOCK*



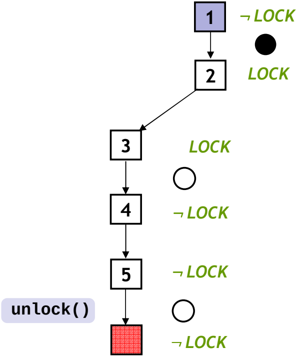
**Reachability Tree**

# Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



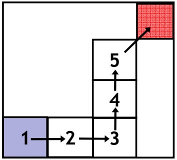
**Predicates:** *LOCK*



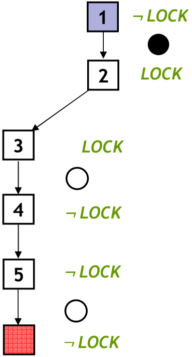
**Reachability Tree**

# Analyze Counterexample

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



**Predicates:** *LOCK*



```
lock()
old = new
q=q->next
```

`[q!=NULL]`

```
q->data = new
unlock()
new++
```

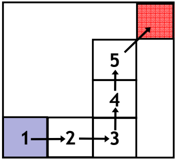
`[new==old]`

`unlock()`

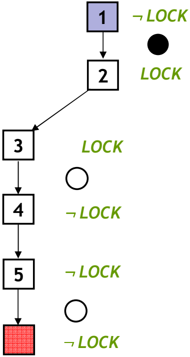
## Reachability Tree

# Analyze Counterexample

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



**Predicates:** *LOCK*



**old = new**

**new++**

**[new==old]**

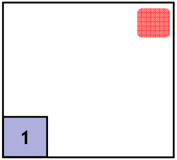
Inconsistent

*new == old*

## Reachability Tree

# Repeat Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```

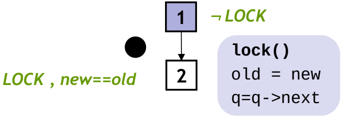


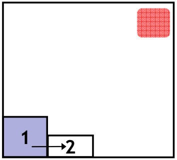
**Predicates:** *LOCK, new==old*

**Reachability Tree**

# Repeat Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5: unlock ();
}
```



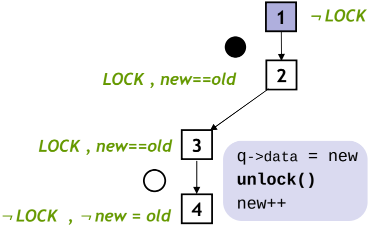


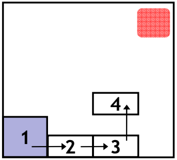
**Predicates:** *LOCK, new==old*

**Reachability Tree**

# Repeat Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:      q->data = new;
        unlock();
        new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



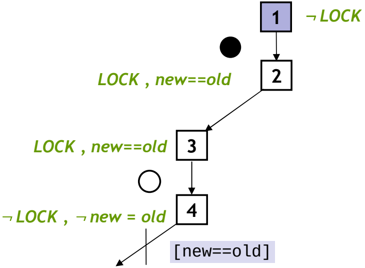


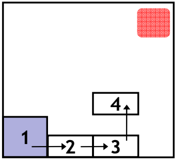
**Predicates:** *LOCK, new==old*

**Reachability Tree**

# Repeat Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



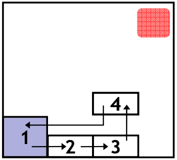


**Predicates:** *LOCK, new==old*

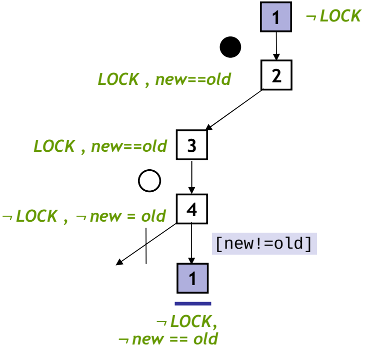
**Reachability Tree**

# Repeat Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



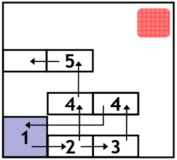
**Predicates:** *LOCK, new==old*



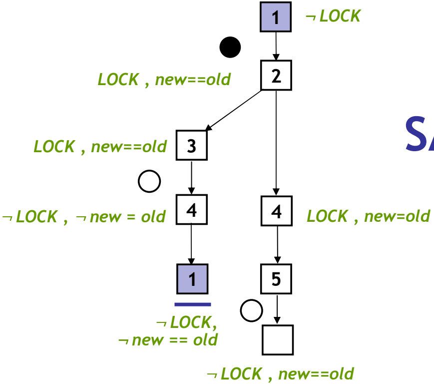
## Reachability Tree

# Repeat Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```



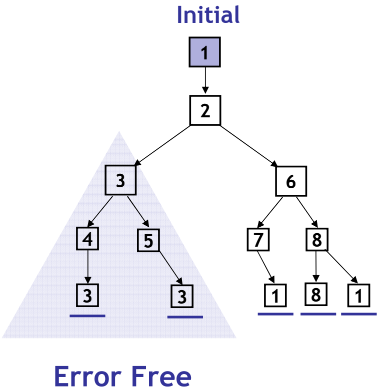
**Predicates:** *LOCK, new==old*



**SAFE**

**Reachability Tree**

# Key Idea: Reachability Tree



## Unroll

1. Pick tree-node (=abs. state)
2. Add children (=abs. successors)
3. On re-visiting abs. state, cut-off

## Find min spurious suffix

- Learn new predicates
- Rebuild subtree with new preds.

**SAFE**

**S1:** Only Abstract Reachable States

**S2:** Don't refine error-free regions

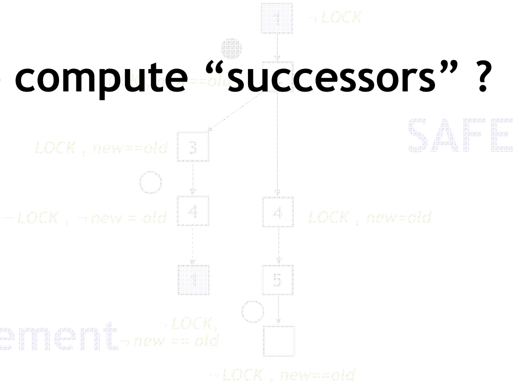
# Technical Details

- Q: How to compute “successors”?
- Q: How to find predicates ? [Interpolation]
- Q: How to analyze (recursive) procedures? [Context-free reachability]

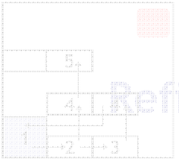
# Technical Details

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5: unlock ();
}
```

**Q.** **How to compute “successors” ?**



SAFE



Refinement

Predicates: LOCK, new==old

# #Predicates grows with program size

```
     while(1){
T ●  1: if (p1) lock() ;
F       if (p1) unlock() ;
                …
T ●  2: if (p2) lock() ;
        if (p2) unlock() ;
                …
     n: if (pn) lock() ;
        if (pn) unlock() ;
     }
```

Tracking **lock** not enough

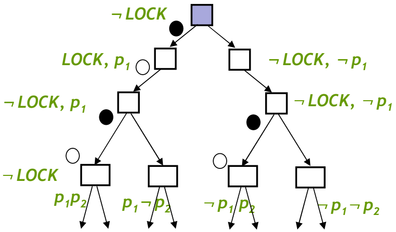
**Problem:**

$p_1, \ldots, p_n$ needed for verification

Exponential reachable abstract states

# #Predicates grows with program size

```
while(1){
1: if (p1) lock() ;
   if (p1) unlock() ;
              …
2: if (p2) lock() ;
   if (p2) unlock() ;
              …
n: if (pn) lock() ;
   if (pn) unlock() ;
}
```



$2^n$ **Abstract States**

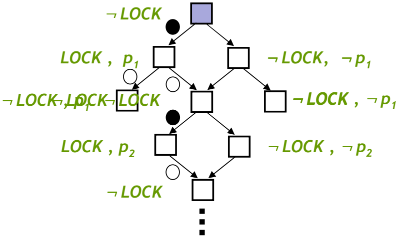
**Problem:**

$p_1,\ldots,p_n$ needed for verification

Exponential reachable abstract states

# Predicates useful *locally*

```
      while(1){
p1    1: if (p1) lock() ;
         if (p1) unlock() ;
                  …
p2    2: if (p2) lock() ;
         if (p2) unlock() ;
                  …
pn    n: if (pn) lock() ;
         if (pn) unlock() ;
      }
```



**2n Abstract States**

**Solution:** Use predicates **only where needed**

Using **Counterexamples:**

**Q1.** Find **predicates**

**Q2.** Find **where** predicates are needed

# Counterexample Traces

```
lock()
old = new
q=q->next
```

```
[q!=NULL]
```

```
q->data = new
unlock()
new++
```

```
[new==old]
```

```
unlock()
```

Trace

$lock_1 = 1$
$old_1 = new_0$
$q_1 = q_0\text{->}next$

$assume(q_1 \text{ != } NULL)$

$(q_1 \text{->} data)_1 = new_0$
$lock_2 = 0$
$new_1 = new_0 + 1$

$assume(new_1 = old_1)$

$assert(lock_2 = 1)$

SSA Trace

$lock_1 = 1 \wedge$
$old_1 = new_0 \wedge$
$q_1 = q_0\text{->}next \wedge$

$q_1 \text{ != } NULL \wedge$

$(q_1 \text{->} data)_1 = new_0 \wedge$
$lock_2 = 0 \wedge$
$new_1 = new_0 + 1 \wedge$

$new_1 = old_1$

Trace Feasibility Formula

Thm: Trace is **feasible** ⇔ TF is **satisfiable**

# Proof of Unsatisfiability

$\texttt{lock}_1 = 1 \wedge$

$\boxed{\texttt{old}_1 = \texttt{new}_0} \wedge$

$\texttt{q}_1 = \texttt{q}_0\texttt{->next} \wedge$

$\texttt{q}_1 \texttt{ != NULL} \wedge$

$(\texttt{q}_1 \texttt{ -> data})_1 = \texttt{new}_0 \wedge$

$\texttt{lock}_2 = 0 \wedge$

$\boxed{\texttt{new}_1 = \texttt{new}_0 \texttt{ +1}} \wedge$

$\boxed{\texttt{new}_1\texttt{=}\texttt{old}_1}$

Trace Feasibility Formula

$$\cfrac{\cfrac{old_1 = new_0 \qquad new_1 = new_0 + 1}{new_1 = old_1 + 1} \qquad new_1 = old_1}{\emptyset}$$

Proof of Unsatisfiability

Predicates: old=new, new=new+1, new=old

Add: old=new

[HenzingerJhalaM.Sutre02]

# Counterexample Traces: Take 2

```
1: x = ctr;
2: ctr = ctr + 1;
3: y = ctr;
4: if (x = i-1){
5:   if (y != i){
        ERROR: }
   }
```

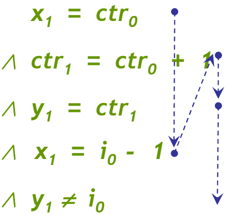
```
1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)
```

$y = x + 1$

# Trace Formulas

| | | |
|---|---|---|
| *1*: `x = ctr` | *1*: $\texttt{x}_1 = \texttt{ctr}_0$ | $x_1 = ctr_0$ |
| *2*: `ctr = ctr+1` | *2*: $\texttt{ctr}_1 = \texttt{ctr}_0\texttt{+1}$ | $\wedge\ ctr_1 = ctr_0 + 1$ |
| *3*: `y = ctr` | *3*: $\texttt{y}_1 = \texttt{ctr}_1$ | $\wedge\ y_1 = ctr_1$ |
| *4*: `assume(x=i-1)` | *4*: $\texttt{assume(x}_1\texttt{=i}_0\texttt{-1)}$ | $\wedge\ x_1 = i_0 - 1$ |
| *5*: `assume(y≠i)` | *5*: $\texttt{assume(y}_1 \neq \texttt{i}_0\texttt{)}$ | $\wedge\ y_1 \neq i_0$ |
| Trace | SSA Trace | Trace Feasibility Formula |

# Proof of Unsatisfiability



$x_1 = ctr_0$

$\wedge\ ctr_1 = ctr_0 + 1$

$\wedge\ y_1 = ctr_1$

$\wedge\ x_1 = i_0 - 1$

$\wedge\ y_1 \neq i_0$

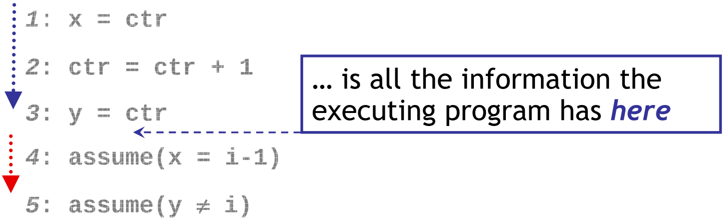
Trace Formula

$$\frac{\frac{\frac{\frac{x_1 = ctr_0 \quad x_1 = i_0 - 1}{ctr_0 = i_0 - 1} \quad ctr_1 = ctr_0 + 1}{ctr_1 = i_0} \quad y_1 = ctr_1}{y_1 = i_0} \quad y_1 \neq i_0}{\emptyset}$$

Proof of Unsatisfiability

# The Present State…

Trace



```
1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)
```

… is all the information the executing program has ***here***

State…

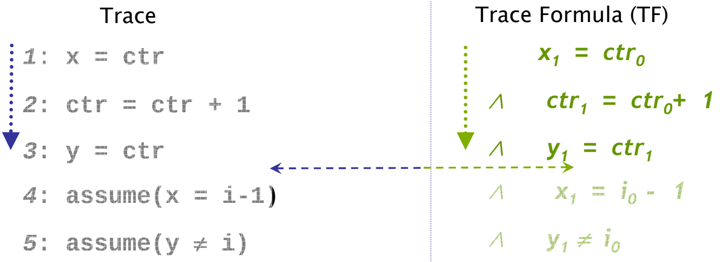
1. … after executing trace ***past (prefix)***
2. … knows ***present values*** of variables
3. … makes trace ***future (suffix)*** infeasible

At $\boldsymbol{pc_4}$, which predicate on ***present state*** shows infeasibility of ***future*** ?

# What Predicate is needed ?

| Trace | Trace Formula (TF) |
|---|---|
| ***1***: `x = ctr` | $x_1 = ctr_0$ |
| ***2***: `ctr = ctr + 1` | $\wedge \quad ctr_1 = ctr_0 + 1$ |
| ***3***: `y = ctr` | $\wedge \quad y_1 = ctr_1$ |
| ***4***: `assume(x = i-1)` | $\wedge \quad x_1 = i_0 - 1$ |
| ***5***: `assume(y ≠ i)` | $\wedge \quad y_1 \neq i_0$ |

# What Predicate is needed ?



Trace

```
1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)
```

Trace Formula (TF)

$$x_1 = ctr_0$$
$$\wedge \quad ctr_1 = ctr_0 + 1$$
$$\wedge \quad y_1 = ctr_1$$
$$\wedge \quad x_1 = i_0 - 1$$
$$\wedge \quad y_1 \neq i_0$$

**Relevant Information**

1. … after executing trace **prefix**

**Predicate …**

… implied by TF **prefix**

# What Predicate is needed ?

| Trace | Trace Formula (TF) |
|---|---|
| `1: x = ctr` | $x_1 = ctr_0$ |
| `2: ctr = ctr + 1` | $\wedge \quad ctr_1 = ctr_0 + 1$ |
| `3: y = ctr` | $\wedge \quad y_1 = ctr_1$ |
| `4: assume(x = i-1)` | $\wedge \quad x_1 = i_0 - 1$ |
| `5: assume(y ≠ i)` | $\wedge \quad y_1 \neq i_0$ |

| **Relevant Information** | **Predicate …** |
|---|---|
| 1. … after executing trace **prefix** | … implied by TF **prefix** |
| 2. … has **present values** of variables | … on **common** variables |

# What Predicate is needed ?

Trace

```
1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)
```

**Relevant Information**

1. … after executing trace **prefix**
2. … has **present values** of variables
3. … makes trace **suffix** infeasible

Trace Formula (TF)

$$
\begin{aligned}
& x_1 = ctr_0 \\
\wedge\ & ctr_1 = ctr_0 + 1 \\
\wedge\ & y_1 = ctr_1 \\
\wedge\ & x_1 = i_0 - 1 \\
\wedge\ & y_1 \neq i_0
\end{aligned}
$$

**Predicate …**

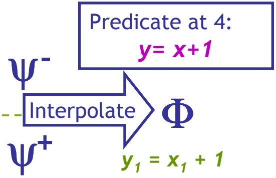
… implied by TF **prefix**

… on **common** variables

… & TF **suffix** is **unsatisfiable**

# Interpolant = Predicate !

| Trace | Trace Formula |
|---|---|
| `1: x = ctr` | $x_1 = ctr_0$ |
| `2: ctr = ctr + 1` | $\wedge \quad ctr_1 = ctr_0 + 1$ |
| `3: y = ctr` | $\wedge \quad y_1 = ctr_1$ |
| `4: assume(x = i-1)` | $\wedge \quad x_1 = i_0 - 1$ |
| `5: assume(y ≠ i)` | $\wedge \quad y_1 \neq i_0$ |



Predicate at 4:
$y = x+1$

$y_1 = x_1 + 1$

**Craig Interpolant**
[Craig 57]

**Computable from Proof of Unsat**

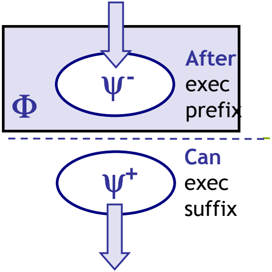
Predicate …

… implied by TF prefix

… on common variables

… & TF suffix is unsatisfiable

# Another interpretation …



Trace Formula

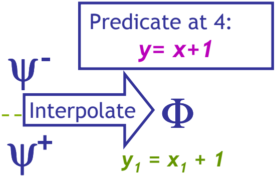
$x_1 = ctr_0$

$\wedge \quad ctr_1 = ctr_0 + 1$

$\wedge \quad y_1 = ctr_1$

$\wedge \quad x_1 = i_0 - 1$

$\wedge \quad y_1 \neq i_0$



Predicate at 4: $y = x+1$

$y_1 = x_1 + 1$

Unsat = Empty Intersection = Trace Infeasible

Interpolant $\Phi$ =

Overapproximation of states after prefix
that cannot execute suffix

# Main Questions

Q. How to find good predicates ?
Where to track each predicate?

**Q:** How to compute interpolants?
(And do they always exist?)

# Another Proof of Unsatisfiability

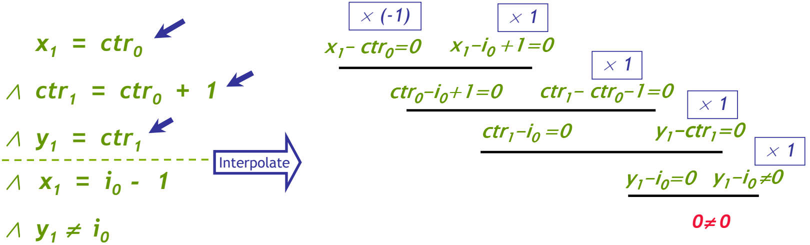
$$\cfrac{\cfrac{\cfrac{\cfrac{x_1 = ctr_0 \qquad x_1 = i_0 - 1}{ctr_0 = i_0 - 1} \qquad ctr_1 = ctr_0 + 1}{ctr_1 = i_0} \qquad y_1 = ctr_1}{y_1 = i_0} \qquad y_1 \neq i_0}{\emptyset}$$

Proof of Unsatisfiability

$$\cfrac{\cfrac{\cfrac{\cfrac{\overset{\boxed{\times\ (-1)}}{x_1 - ctr_0 = 0} \qquad \overset{\boxed{\times\ 1}}{x_1 - i_0 + 1 = 0}}{ctr_0 - i_0 + 1 = 0} \qquad \overset{\boxed{\times\ 1}}{ctr_1 - ctr_0 - 1 = 0}}{ctr_1 - i_0 = 0} \qquad \overset{\boxed{\times\ 1}}{y_1 - ctr_1 = 0}}{y_1 - i_0 = 0} \qquad \overset{\boxed{\times\ 1}}{y_1 - i_0 \neq 0}}{0 \neq 0}$$

Rewritten Proof

# Interpolant from Rewritten Proof ?



$x_1 = ctr_0$

$\wedge\ ctr_1 = ctr_0 + 1$

$\wedge\ y_1 = ctr_1$

$\wedge\ x_1 = i_0 - 1$

$\wedge\ y_1 \neq i_0$

Trace Formula

$$\dfrac{\boxed{\times (-1)}\ x_1 - ctr_0 = 0 \qquad \boxed{\times 1}\ x_1 - i_0 + 1 = 0}{ctr_0 - i_0 + 1 = 0}$$

$$\dfrac{ctr_0 - i_0 + 1 = 0 \qquad \boxed{\times 1}\ ctr_1 - ctr_0 - 1 = 0}{ctr_1 - i_0 = 0}$$

$$\dfrac{ctr_1 - i_0 = 0 \qquad \boxed{\times 1}\ y_1 - ctr_1 = 0}{y_1 - i_0 = 0}$$

$$\dfrac{y_1 - i_0 = 0 \qquad \boxed{\times 1}\ y_1 - i_0 \neq 0}{0 \neq 0}$$

Rewritten Proof

# Interpolant from Rewritten Proof ?

$x_1 = ctr_0$

$\wedge\ ctr_1 = ctr_0 + 1$

$\wedge\ y_1 = ctr_1$

- - - - - - - - - - - - - - - Interpolate ⇒

$\wedge\ x_1 = i_0 - 1$

$\wedge\ y_1 \neq i_0$

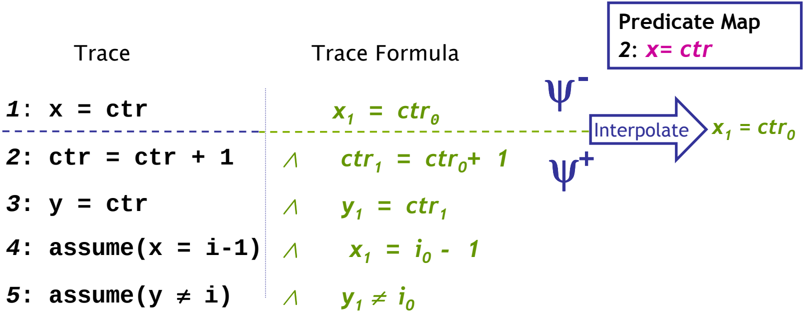
Trace Formula

$$\begin{array}{rl} x_1 - ctr_0 = 0 & \boxed{\times\ (-1)} \\ ctr_1 - ctr_0 - 1 = 0 & \boxed{\times\ 1} \\ y_1 - ctr_1 = 0 & \boxed{\times\ 1} \\ \hline \boxed{y_1 - x_1 - 1 = 0} & \end{array}$$

$\boxed{y_1 = x_1 + 1}$

Interpolant !

# Building Predicate Maps



**Predicate Map**
2: $x = ctr$

| Trace | Trace Formula |
|---|---|
| `1: x = ctr` | $x_1 = ctr_0$ |
| `2: ctr = ctr + 1` | $\wedge \quad ctr_1 = ctr_0 + 1$ |
| `3: y = ctr` | $\wedge \quad y_1 = ctr_1$ |
| `4: assume(x = i-1)` | $\wedge \quad x_1 = i_0 - 1$ |
| `5: assume(y ≠ i)` | $\wedge \quad y_1 \neq i_0$ |

$\psi^-$

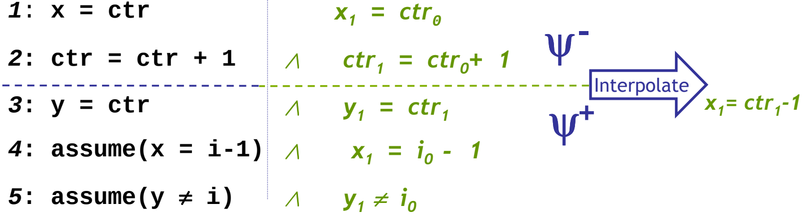
Interpolate → $x_1 = ctr_0$

$\psi^+$

- Cut + Interpolate at **each** point
- Pred. Map: $pc_i$ □ Interpolant from cut i

# Building Predicate Maps

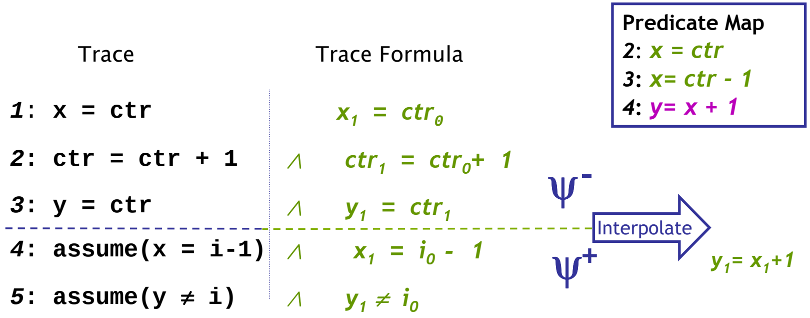
**Predicate Map**
*2*: $x = ctr$
*3*: $x= ctr-1$



| Trace | Trace Formula |
|---|---|
| `1: x = ctr` | $x_1 = ctr_0$ |
| `2: ctr = ctr + 1` | $\wedge \quad ctr_1 = ctr_0 + 1$ |
| `3: y = ctr` | $\wedge \quad y_1 = ctr_1$ |
| `4: assume(x = i-1)` | $\wedge \quad x_1 = i_0 - 1$ |
| `5: assume(y ≠ i)` | $\wedge \quad y_1 \neq i_0$ |

- Cut + Interpolate at **each** point
- Pred. Map: $pc_i$ □ Interpolant from cut i

# Building Predicate Maps



| Trace | Trace Formula |
|---|---|
| **1**: `x = ctr` | $x_1 = ctr_0$ |
| **2**: `ctr = ctr + 1` | $\wedge \; ctr_1 = ctr_0 + 1$ |
| **3**: `y = ctr` | $\wedge \; y_1 = ctr_1$ |
| **4**: `assume(x = i-1)` | $\wedge \; x_1 = i_0 - 1$ |
| **5**: `assume(y ≠ i)` | $\wedge \; y_1 \neq i_0$ |

$\psi^-$

$\psi^+$

Interpolate

$y_1 = x_1 + 1$

**Predicate Map**

**2**: $x = ctr$

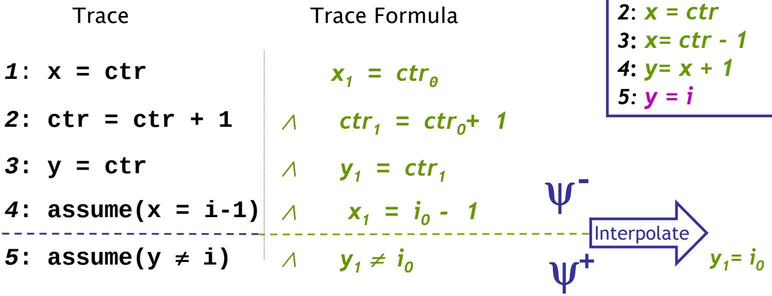
**3**: $x = ctr - 1$

**4**: $y = x + 1$

- Cut + Interpolate at **each** point
- Pred. Map: $pc_i$ □ Interpolant from cut i

# Building Predicate Maps

**Predicate Map**
*2*: $x = ctr$
*3*: $x = ctr - 1$
*4*: $y = x + 1$
*5*: $y = i$



| Trace | Trace Formula |
|---|---|
| `1: x = ctr` | $x_1 = ctr_0$ |
| `2: ctr = ctr + 1` | $\wedge\ ctr_1 = ctr_0 + 1$ |
| `3: y = ctr` | $\wedge\ y_1 = ctr_1$ |
| `4: assume(x = i-1)` | $\wedge\ x_1 = i_0 - 1$ |
| `5: assume(y ≠ i)` | $\wedge\ y_1 \neq i_0$ |

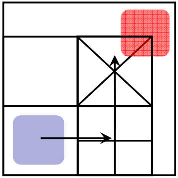
- Cut + Interpolate at **each** point
- Pred. Map: $pc_i$ □ Interpolant from cut i

# Local Predicate Use

Use predicates **needed** at **location**

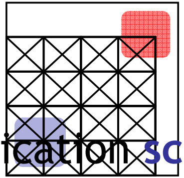
- #Preds. grows with program size
- #**Preds per location** small

**Predicate Map**
*2*: *x = ctr*
*3*: *x= ctr - 1*
*4*: *y= x + 1*
*5*: *y = i*



Local Predicate use

Ex: 2n states



Verification **scales …**

Global Predicate use

Ex: $2^n$ states

# Question: When Do Interpolants Exist?

- Craig's Theorem guarantees existence for first order logic
- But we are interpreting formulas over theories (arithmetic, theories of data structures)

# The Good News

- Interpolants always exist for recursively enumerable theories
    - The proof is a simple application of compactness
- So: interpolants exist for Presburger arithmetic, sets with cardinality constraints, theory of lists, (quantifier-free) theory of arrays, multisets, …

# The Bad News

- “The proof is a simple application of compactness”
  - May be algorithmically inefficient
  - Daunting engineering task to construct interpolating decision procedure for each individual theory

# An Alternate Path: Reduction

- Want to compile formulas in a new theory to formulas in an old theory such that interpolation in the old theory imply interpolation in the new theory

- T reduces to R: can compile formulas in theory T to formulas in theory R
  - And use decision procedures for R to answer decision questions for T

---

- Technically: Given theories T and R, with $R \subseteq T$, a reduction is a computable map $\mu$ from T formulas to R formulas such that for any T-formula $\phi$:
  - $\phi$ and $\mu(\phi)$ are T-equivalent
  - $\phi$ is T-satisfiable iff $\mu(\phi)$ is R-satisfiable

# Example: Theory of Sets

Theory of sets reduces to theory of equality with uninterpreted functions

| | |
|---|---|
| $x = y$ | $\forall e.\ e \in x \Leftrightarrow e \in y$ |
| $x = \emptyset$ | $\forall e. e \notin x$ |
| $x = U$ | $\forall e. e \in x$ |
| $x = \{e\}$ | $e \in x \wedge \forall e'. e' \in x \Rightarrow e = e'$ |
| $x = y \cup z$ | $\forall e. e \in x \Leftrightarrow e \in y \vee e \in z$ |
| $x = y \cap z$ | $\forall e. e \in x \Leftrightarrow e \in y \wedge e \in z$ |

# Example: Theory of Multisets

Theory of multisets reduces to the combination theory of equality with uninterpreted functions and linear arithmetic

| | |
|---|---|
| $x = y$ | $\forall e.\ count(x,e) = count(y,e)$ |
| $x=\emptyset$ | $\forall e. count(x,e) = 0$ |
| $x=[(e,n)]$ | $count(x,e)=max(0,n)$<br>$\wedge \forall e'. e' \neq e \Rightarrow count(x,e')=0$ |
| $x=y \uplus z$ | $\forall e. count(x,e)= count(y,e)+count(z,e)$ |
| $x=y \cup z$ | $\forall e.\ count(x,e) = max(count(y,e), count(z,e))$ |
| $x=y \cap z$ | $\forall e.\ count(x,e) = min(count(y,e), count(z,e))$ |

# Reduction and Interpolation

$\Psi^-$ and $\Psi^+$ in Theory T

Reduction from T to R

$\Phi^-$ and $\Phi^+$ in Theory R

Interpolate in R

Interpolant $\alpha$ in
Theory R as well as T

Eliminate quantifiers in T or R

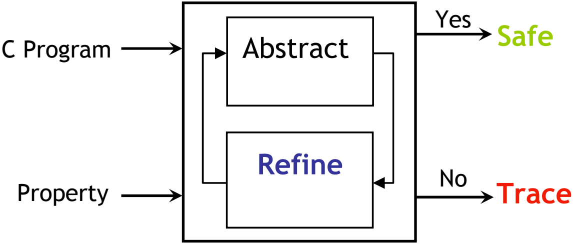
Quantifier-free
interpolant

[KapurM.Zarba06]

# Reduction Theorem

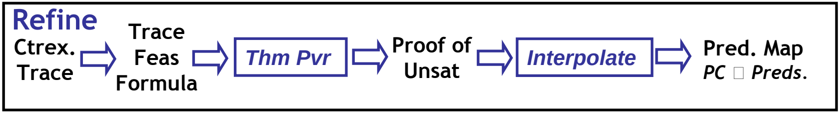
- Interpolants for the theory of arrays, sets, and multisets can be computed by reduction to the combination theory of linear arithmetic and equality with uninterpreted functions
  - We already have interpolating decision procedures for this latter theory

# Lazy Abstraction



**Problem:** #Preds grows w/ Program Size

**Solution:** Localize pred. use, find where preds. needed



# Refinement Failure: Unrolling Loops

```
x = 0; y = 50;
while ( x<100 ) {
  if ( x>=50 ) y = y+1;
  x = x+1;
}
assert( y==100 );
```

- counterexample:
  x=0; y=50; x>=100; y==100
  refinement: x==0
- counterexample:
  x=0; y=50; x<100; x=x+1; x>=100; y==100
  refinement: x==1
- counterexample:
  x=0; y=50; x<100; x=x+1; x<100; x=x+1;
  x>=100; y==100
  refinement: x==2
- ...

# Refinement Failure: Unfolding Arrays

```
for (i=0; i<n; i++) {
  a[i]=i;
}
for (j=0; j<n; j++) {
  assert( a[j]==j );
}
```

- counterexample:

  i=0; i<n; a[i]=i; i++; i>=n;

  j=0; j<n; a[j]!=j

  refinement: a[0]==0
- counterexample:

  i=0; i<n; a[i]=i; i++; i<n; a[i]=i; i++; i>=n;

  j=0; j<n; a[j]==j; j++; j<n; a[j]!=j

  refinement: a[1]==1
- ...

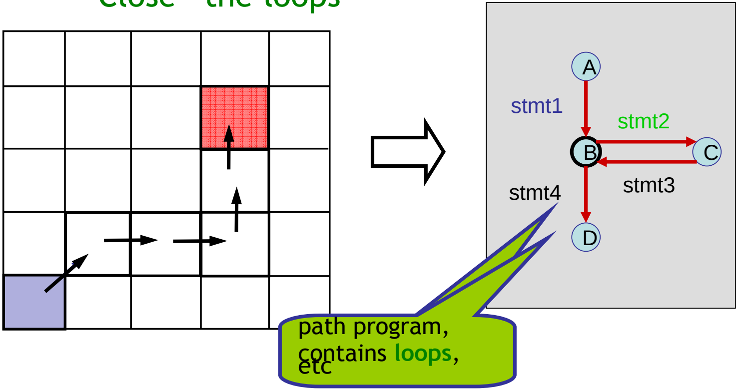
# What went Wrong?

- Consider all unrolled counterexamples at once
  - Convergence of abstraction discovery
- Inspect families of counterexamples of unbounded length
  - Justification for unbounded universal quantification
- Looking at one counterexample path at a time is too weak [JhalaMcMillan05,JhalaMcMillan06]

[BeyerHenzingerM.Rybalchenko07]

# Path Programs

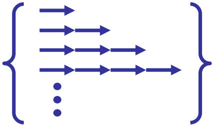
- Treat counterexamples as programs
  - “Close” the loops



path program, contains **loops**, etc

# Meaning of Path Programs

Path program ´ (Possibly unbounded) sets of counterexamples:



- Unbounded counterexamples

- Property-determined fragment of original program
  - Can be analyzed independently to find good abstractions

# Path Invariants

- Invariant for path programs ´ path invariant

- Abstraction refinement using path invariants

  - Elimination of **all** counterexamples **within** path program

  - Justification for **unbounded** quantification

# Invariant Generation

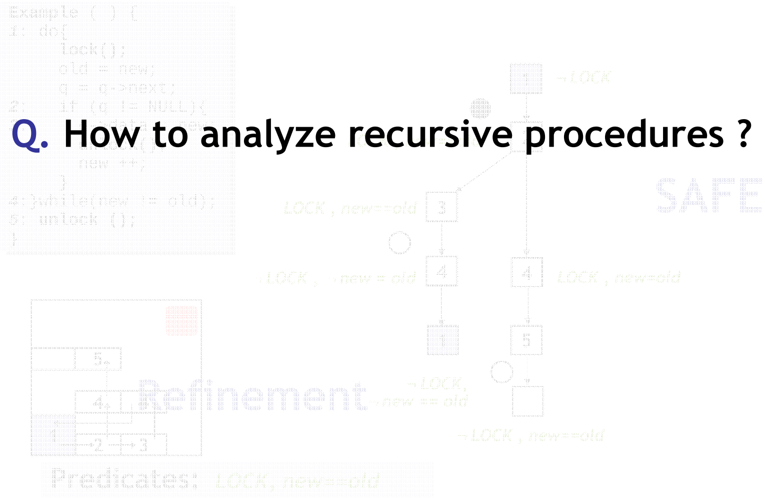
- Given a path program, with a designated error location, find an invariant that demonstrates error is not reachable
  - Can scale: Reduced obligation to program fragment
  - Outer model checking loop integrates path invariants into program invariant

- Can use any technique
- We use constraint-based invariant generation [SankaranarayananSipmaManna04,BeyerHenzingerM.Rybalchenko07]

# Lecture 3:
# Technical Extensions
# and
# Termination

Rupak Majumdar

# Technical Details



**Q.** **How to analyze recursive procedures ?**

# An example

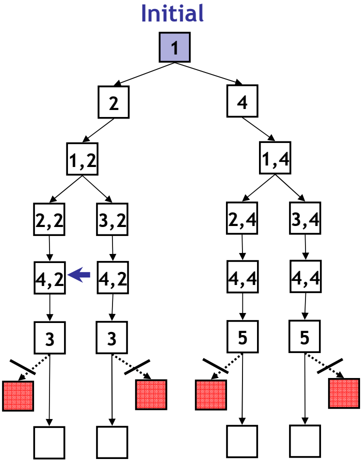
```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```

# Inline Calls in Reach Tree

```
main(){
    □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

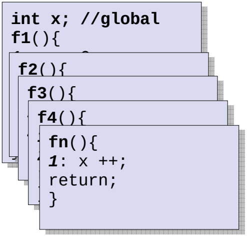
```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



# Inline Calls in Reach Tree

**Problem**

- Repeated analysis for “inc”
- Exploding call contexts



```
int x; //global
f1(){
```

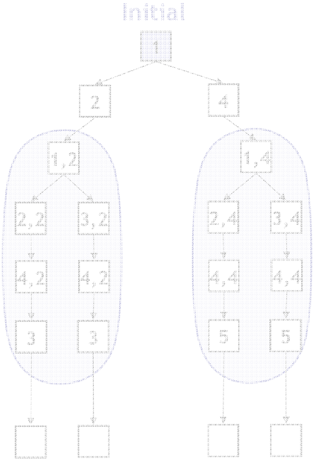
```
f2(){
```

```
f3(){
```

```
f4(){
```

```
fn(){
1: x ++;
return;
}
```

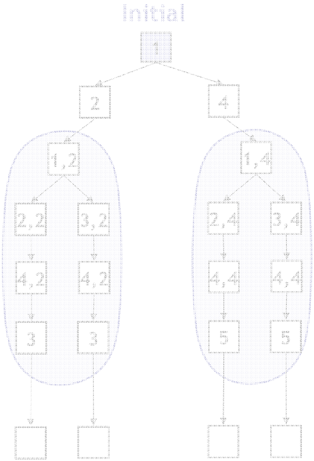
$2^n$ nodes in Reach Tree



# Inline Calls in Reach Tree

**Problem**

- Repeated analysis for “inc”
- Exploding call contexts
- Cyclic call graph (Recursion)
    - Infinite Tree!



# Solution : Procedure Summaries

Summaries: Input/Output behavior

- Plug summaries in at each callsite

  … instead of inlining entire procedure

[Sharir-Pnueli 81, Reps-Horwitz-Sagiv 95]

- Summary = set of *(F □ F')*
  - *F* : Precondition formula describing input state
  - *F'* : Postcondition formula describing output state

# Solution : Procedure Summaries

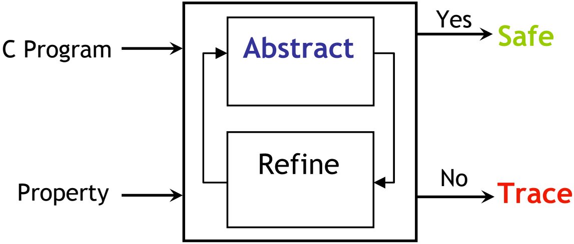
```
inc(int a, int sign){
1: if (sign){
2:   rv = a+1;
   } else {
3:   rv = a-1;
   }
4: return rv;
}
```

- $(\neg\, sign=0 \;\Box\; rv > a)$
- $(sign = 0 \;\Box\; rv < a)$

- Summary = set of $(F \;\Box\; F')$
  - $F$ : Precondition formula describing input state
  - $F'$ : Postcondition formula describing output state

**Q. How to compute, use summaries ?**

## Lazy Abstraction + Procedure Summaries


C Program

Property

**Abstract**

Refine

Yes

**Safe**

No

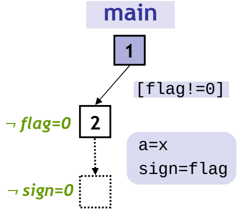
**Trace**

**Q. How to compute, use summaries ?**

# Abstraction with Summaries

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



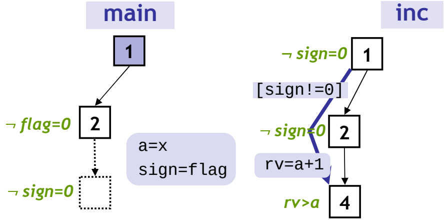
**Predicates:** $flag=0$ , $y>x$ , $y<z$

$sign=0$ , $rv>a$ , $rv<a$

# Abstraction with Summaries

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



**Predicates:** *flag=0 , y>x , y<z*

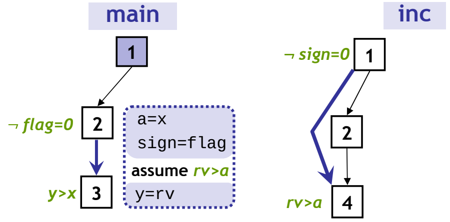
*sign=0 , rv>a , rv<a*

**Summary:** *(¬ sign=0 □ rv>a),*

# Summary Successor

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



**Predicates:** *flag=0 , y>x , y<z*

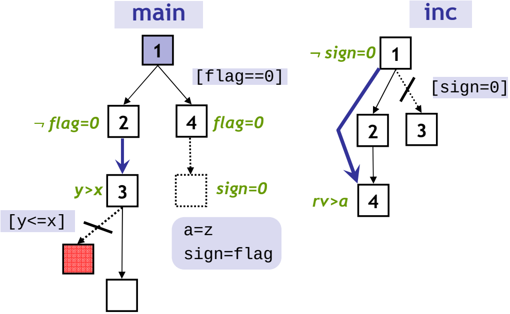
*sign=0 , rv>a , rv<a*

**Summary:** *(¬ sign=0 □ rv>a),*

# Abstraction with Summaries

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



**Predicates:** $flag=0$ , $y>x$ , $y<z$

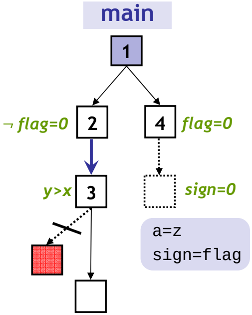
$sign=0$ , $rv>a$ , $rv<a$

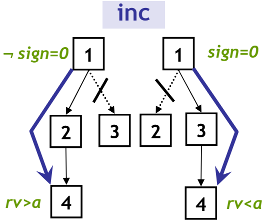
**Summary:** $(\neg sign=0$ □ $rv>a)$,

# Abstraction with Summaries

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```





**Predicates:** *flag=0 , y>x , y<z*

*sign=0 , rv>a , rv<a*

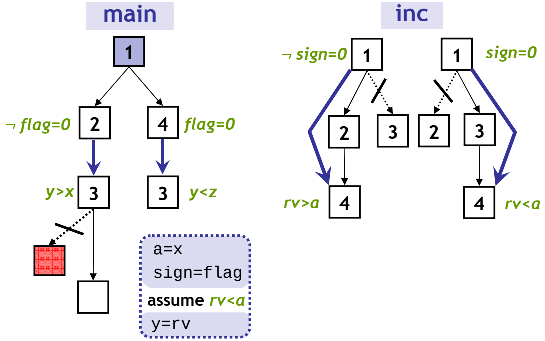
**Summary:** *(¬ sign=0 □ rv>a),*

*(sign=0 □ rv<a)*

# Summary Successor

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



**Predicates:** *flag=0 , y>x , y<z*

*sign=0 , rv>a , rv<a*

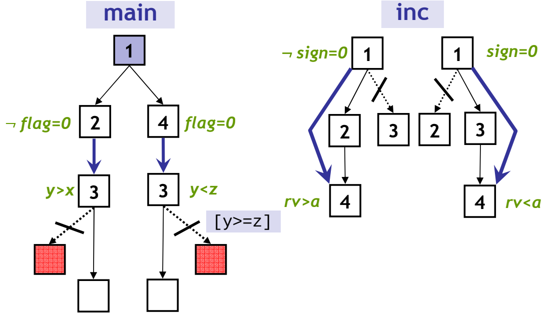
**Summary:** *(¬ sign=0 □ rv>a),*

*(sign=0 □ rv<a)*

# Abstraction with Summaries

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
   □
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



**Predicates:** *flag=0 , y>x , y<z*

*sign=0 , rv>a , rv<a*

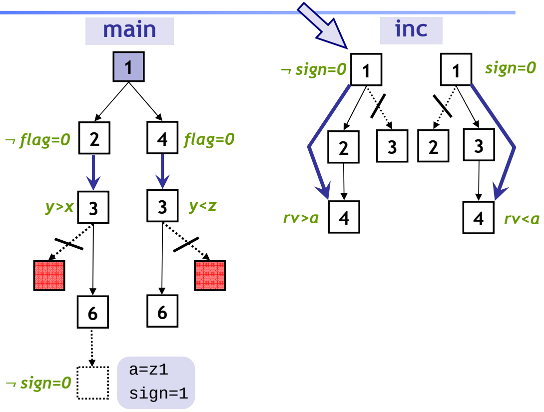
**Summary:** *(¬ sign=0 □ rv>a),*

*(sign=0 □ rv<a)*

# Another Call …

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
6: y1 = inc(z1,1);
7: if (y1<=z1) ERROR;
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



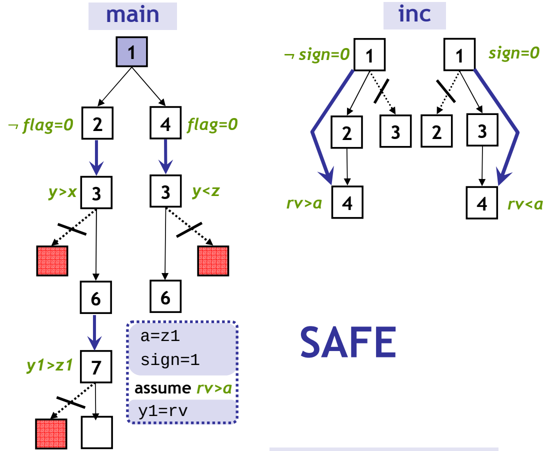
**Predicates:** *flag=0 ,y>x,y<z, y1>z1*
*sign=0 , rv>a , rv<a*

**Summary:** *(¬ sign=0 □ rv>a),*
*(sign=0 □ rv<a)*

# Another Call …

```
main(){
   □
1: if (flag){
2:    y = inc(x,flag);
3:    if (y<=x) ERROR;
   } else {
4:    y = inc(z,flag);
5:    if (y>=z) ERROR;
   }
6: y1 = inc(z1,1);
7: if (y1<=z1) ERROR;
return;
}
```

```
inc(int a, int sign){
1: if (sign){
2:    rv = a+1;
   } else {
3:    rv = a-1;
   }
4: return rv;
}
```



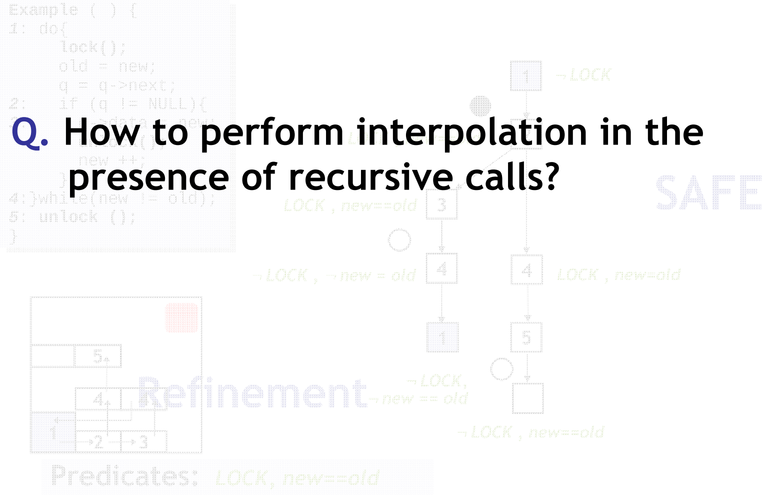
a=z1
sign=1
assume rv>a
y1=rv

**SAFE**

**Predicates:** *flag=0 ,y>x,y<z, y1>z1*
*sign=0 , rv>a , rv<a*

**Summary:** *(¬ sign=0 □ rv>a),*
*(sign=0 □ rv<a)*

# Technical Details

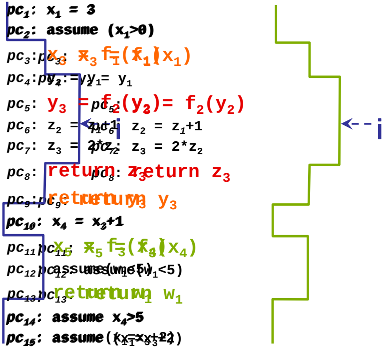


**Q.** How to perform interpolation in the presence of recursive calls?

# Traces with Procedure Calls

Trace

Trace Formula



$pc_1$: $x_1 = 3$

$pc_2$: assume $(x_1>0)$

$pc_3$: $x_3 = f_1(x_1)$

$pc_4$: $y_2 = y_1$

$pc_5$: $y_3 = f_2(y_2)$

$pc_6$: $z_2 = z_1+1$

$pc_7$: $z_3 = 2*z_2$

$pc_8$: return $z_3$

$pc_9$: return $y_3$

$pc_{10}$: $x_4 = x_3+1$

$pc_{11}$: $x_5 = f_3(x_4)$

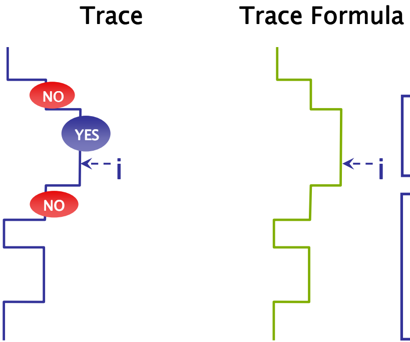
$pc_{12}$: assume$(w_1<5)$

$pc_{13}$: return $w_1$

$pc_{14}$: assume $x_4>5$

$pc_{15}$: assume$(x_1=x_3+2)$

# Interprocedural Analysis



Find predicate needed at point i

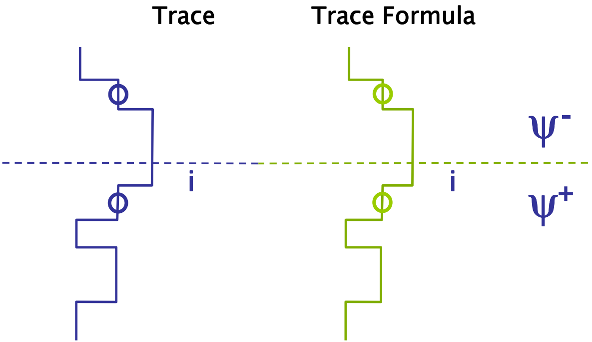
**Require at each point i :**

**Scoped** predicates

**YES**: Variables **visible** at i

**NO**: Caller's local variables

# Problems with Cutting

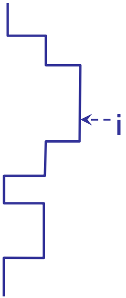


**Caller variables** common to $\psi^-$ and $\psi^+$

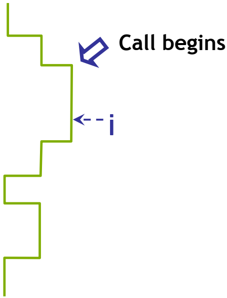
- Unsuitable interpolant: not well-scoped

# Scoped Cuts

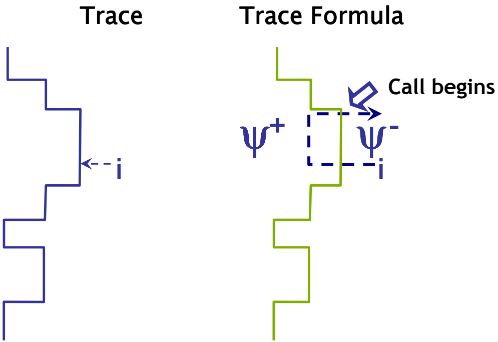
**Trace**



**Trace Formula**

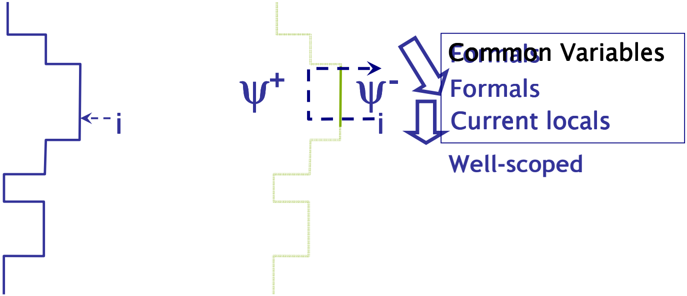


# Scoped Cuts



Predicate at $pc_i$ = Interpolant from cut i

# Common Variables



**Common Variables**

Formals

Current locals

**Well-scoped**

Predicate at $pc_i$ = Interpolant from i–cut

# When does a Program Terminate?

- Iff its reachable transition relation is well-founded

- Reachable transition relation =

  TR(x,x’) $\cap$ Reach(x)$\times$Reach(x’) =

  Restriction of the transition relation to the set of reachable states

# Well-Founded Relation

- A binary relation > is *well-founded* if there is no infinite descending sequence

- No s0, s1, s2,... such that

  s0 > s1 > s2 > ...

Example: > on natural numbers

But not > on integers

# Idea: Rank Functions

- Fix a set X, and > a wf relation on X
- Suppose I can map each reachable state s of the transition graph to a *rank* r(s)$\in$X s.t.

  *s $\rightarrow$ s' implies r(s) > r(s')*

Then the system must terminate

The converse is also true

# Example

```
Input x, n
While(x <= n) x++;
```

Terminates, using (roughly) the rank function n-x

Does it, really?

# Disjunctive Rank Functions

- In many cases, finding a single wf relation can be difficult

- Suppose I can find wf relations T1,…,Tk such that RTR $\subseteq$ T1 $\cup$ … $\cup$ Tk

- Does the program terminate?
  - Not in general (Why?)

# Disjunctive Well-foundedness

If T1…Tk are wf relations and

$R^+ \subseteq T1 \cup \ldots \cup Tk$

Then: R is well-founded

Such R is called *disjunctively well-founded*

# Disjunctive Well-foundedness

*P terminates if TR ∩ Reach×Reach is disjunctively well-founded*

Useful: Can consider individual portions of the program independent of other parts

# Incremental Termination

```
T = emptyset
While TR+ not included in T:
  invariant: T is a finite union of wf relations
  find abstract counterexample to wf
  if concretely feasible
    does not terminate
  otherwise find wf relation T’
  $T = T \cup T’$
```

# Counterexample to Termination

- Lasso = Stem + Cycle
  - Represents infinite execution

  Stem Cycle Cycle …

Needs rank-finding technique to find a wf relation showing lasso cannot be executed arbitrarily (Heuristics exist)

# Reduction to Safety

- How to check if $R^+ \subseteq T$ for the reachable transition relation?

- Can reduce check to safety
- Run program parallel with a monitor for T
  - runs in parallel with the program
  - inspects pairs of states wrt. T
  - goes to error if observes $(s, s') \notin T$
  - Use non-determinism to perform check

# Reduction to Safety: Idea

```
selected := ⊥
phase := SELECT

while True {
    switch (phase) {
       SELECT: if ( nondet() ) {
                   selected := current
                   phase := CHECK
               }
       CHECK: if ( (selected, current) ∉ T ) { ERROR: }
    }
}
```

# Terminator

- Input: program written in C

- Language features supported
  - nested loops, gotos
  - aliasing
  - (mutually) recursive function calls

- Output:
  - termination proof: transition invariant
  - counterexample: lasso = stem + cycle

- Scalability: (on drivers from WinDDK)