# Model Checking and Predicate Abstraction

CSE 501

Spring 15

With slides from Emina Torlak (507)

# Course Outline

- Static analysis
- Language design
- Program Verification
- Dynamic analysis
  - Model checking ← We are here
  - Concolic testing
- New compilers

# Understanding programs

- Static analysis
  - Abstract interpretation
  - Formal verification

- Dynamic analysis
  - Daikon: “Likely” invariants
  - Model checking
  - Testing

# Why dynamic analysis?

- Static analysis is imprecise
  - Branches, loops, gotos, …

- Formal verification is hard
  - How to find invariants?

- Implementing ynamic analysis
  - Strawman: run program enough number of times and check

  - Better: define metrics to make sure that all (i.e., sufficient) number of paths are covered

  - Even better: abstract the program into a finite set of states (i.e., a model), run the abstracted program enough number of times and check
    - Hence, *model checking*

# What is model checking

- An automated technique for verifying that a finite state system satisfies a given property.

- M, s ⊨ P
  - M: model of the system
  - s: state of the system
  - P: logic formula that specifies the property of interest

# What is model checking

- M, s ⊨ P

- What are the possible outcomes?
  - Checker returns **false** with a counter-example that violates P
  - Checker returns **true**
    - What does that mean?

# Model checking vs verification

- Model checking
  - Fully automatic checking of properties in less expressive logics (e.g., temporal)

- Verification
  - Semi-automatic or bounded automatic checking of properties in expressive logics (e.g., FOL)

# Model checking vs testing

- Model checking:
  - – If checker terminates, then program guaranteed to satisfy P
  - – What if it doesn't?

- Testing
  - – If tests finish and no counter-examples found, then P is satisfied *with respect to the set of test cases covered*

# Model checking: a history of logics

- 1960s:
  - Modal logics (Kripke)
  - Temporal logic (Arthur Prior)
- 1980-90s:
  - Using linear temporal logic for concurrent programs (Pnueli)
  - Explicit state model checking (Emerson & Clarke)
  - Symbolic model checking (McMillan)
  - Temporal logic of actions (Lamport)
- 1996:
  - Pnueli wins the Turing award "for seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification."
- 2007:
  - Clarke, Emerson and Sifakis jointly win the Turing award "for their role in developing model checking into a highly effective verification technology that is widely adopted in the hardware and software industries."

# Model checkers

- SPIN
- SMV
- BLAST
- Java Pathfinder
- TLA+

# How does it work

# Kripke structures

- Kripke structure is a tuple M = <S, $S_0$, R, L>
  - S is a finite set of states
  - $S_0 \subseteq S$ is the set of initial states.
  - $R \subseteq S \times S$ is the transition relation, which must be *total.*
  - L: S → $2^{AP}$ is a function that labels each state with a set of *atomic propositions* that are true in that state.
- A **path** in M is a (potentially infinite) sequence of states $\pi = s_0 s_1 \ldots$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R$.

# Modeling systems

```
// x==1, y==1
x := (x + y) % 2
```

$S \equiv (x = 0 \lor x = 1) \land (y = 0 \lor y = 1)$

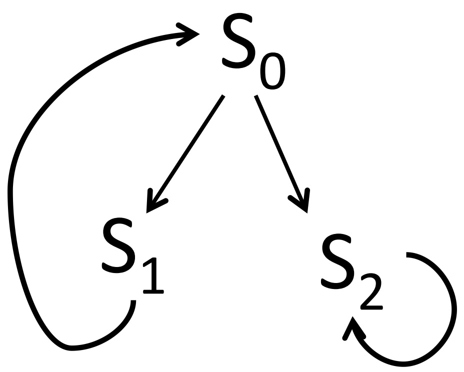
$S_0 \equiv (x=1) \land (y=1)$

$R(x, y, x', y') \equiv (x' = (x + y) \% 2) \land (y' = y)$

- Variables range over a finite domain
- Can use FOL to describe the initial states and transition relation
- Extract Kripke structure from FOL description

# Expressing properties

# Expressing properties in temporal logic

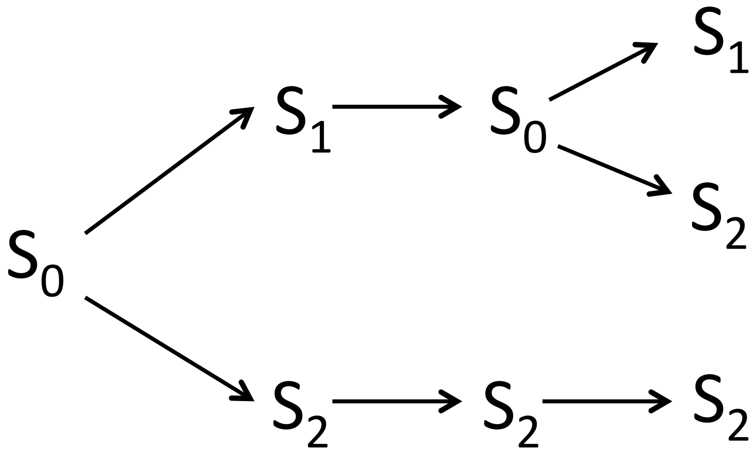


- Linear time: properties of computation **paths**

$S_0 \rightarrow S_1 \rightarrow S_0 \rightarrow S_1$

$S_0 \rightarrow S_2 \rightarrow S_2 \rightarrow S_2$

- Branching time: properties of computation **trees**



# Computation tree logic (CTL*)

- Path quantifiers describe the branching structure of the computation tree
  - **A** (for all paths)
  - **E** (there exists a path)
- Temporal operators
  - $\mathbf{X_p}$ (p holds “next time”)
  - $\mathbf{F_p}$ (p holds “eventually”)
  - $\mathbf{G_p}$ (p holds “always”)
  - **p U q** (p holds “until” q holds)

# Syntax of CTL*

- State formulas
  - Atomic propositions: a ∈ AP
  - ¬f, f ∧ g, f ∨ g, where f and g are state formulas
  - **A**p and **E**p**,** where p is a path formula

- Path formulas
  - f, where f is a state formula
  - ¬p, p ∧ p, p ∨ q, where p and q are path formulas
  - **X**p, **F**p, **G**p, p **U** q, where p and q are path formulas

# Semantics of CTL*

- State formulas
  - M, s $\models$ a iff a $\in$ L(s)
  - M, s $\models$ **A**p iff M, π $\models$ p for all paths π that start at s
  - M, s $\models$ **E**p iff M, π $\models$ p for some path π that starts at s
- Path formulas ($\pi^k$ is suffix of π starting at $s_k$)
  - M, π $\models$ f iff M, s $\models$ f and s is the first state of π
  - M, π $\models$ **X**p iff M, $\pi^1 \models$ p
  - M, π $\models$ **F**p iff M, $\pi^k \models$ p for some k ≥ 0
  - M, π $\models$ **G**p iff M, $\pi^k \models$ p for all k ≥ 0

# CTL and LTL

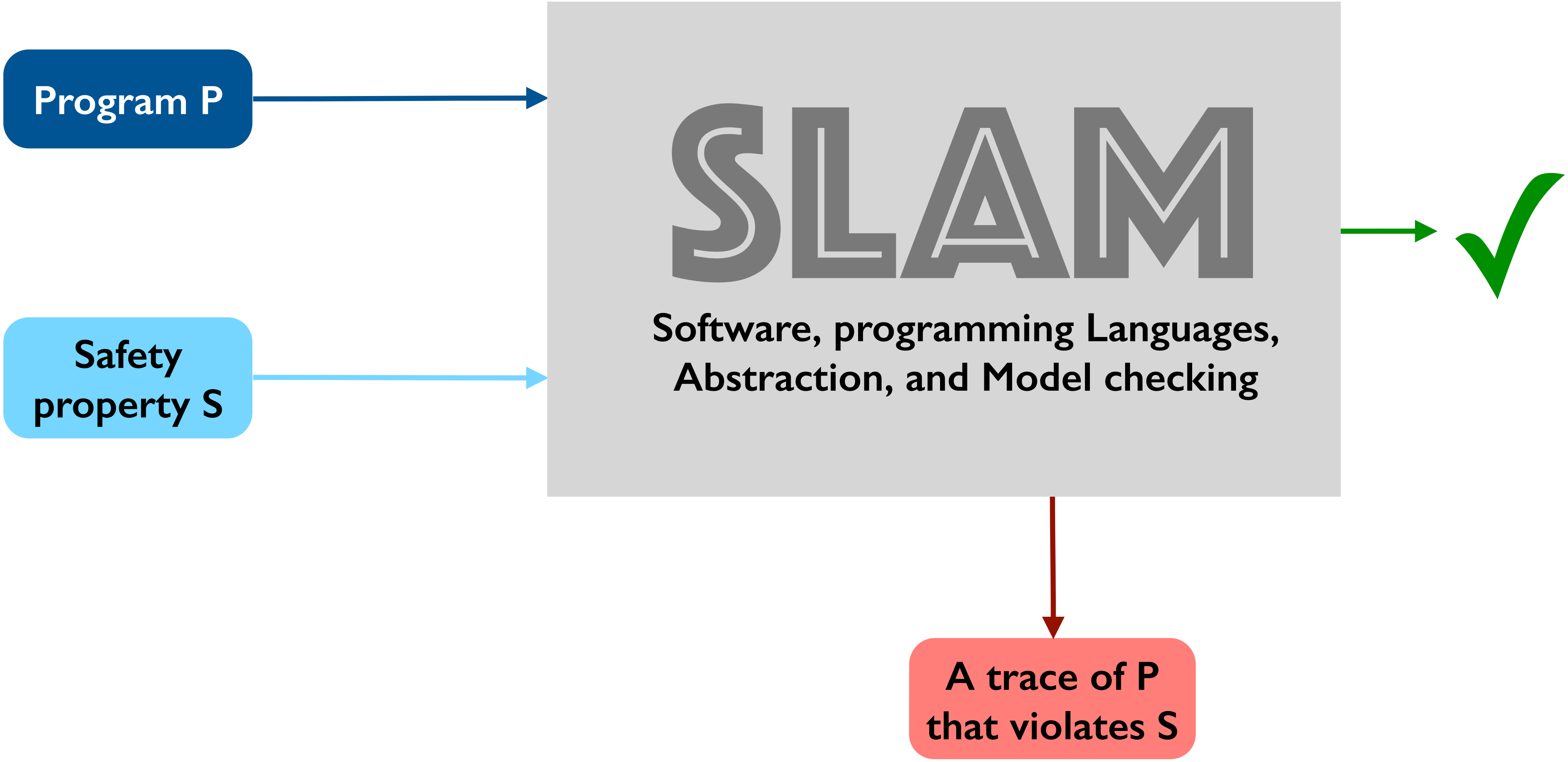
- Both are subsets of CTL*
- CTL:
  - Fragment of CTL* in which each temporal operator is prefixed with a path quantifier.
  - **AG(EF** p): From any state, it is possible to get to a state where p holds.
- LTL:
  - Fragment of CTL* with formulas of the form **Ap,** where p contains no path quantifiers.
  - **A(FG** p): Along every path, there is some state from which p will hold forever.

# Complexity of checking M, s ⊨ P

- Polynomial Time for CTL
  - – Best known algorithm: O(|M| * |P|)
- PSPACE-complete for LTL
  - – Best known algorithm: O(|M| * 2|P|)
- PSPACE-complete for CTL*
  - – Best known algorithm: O(|M| * 2|P|)

# Example checker: SLAM

# The SLAM process



Program P

Safety property S

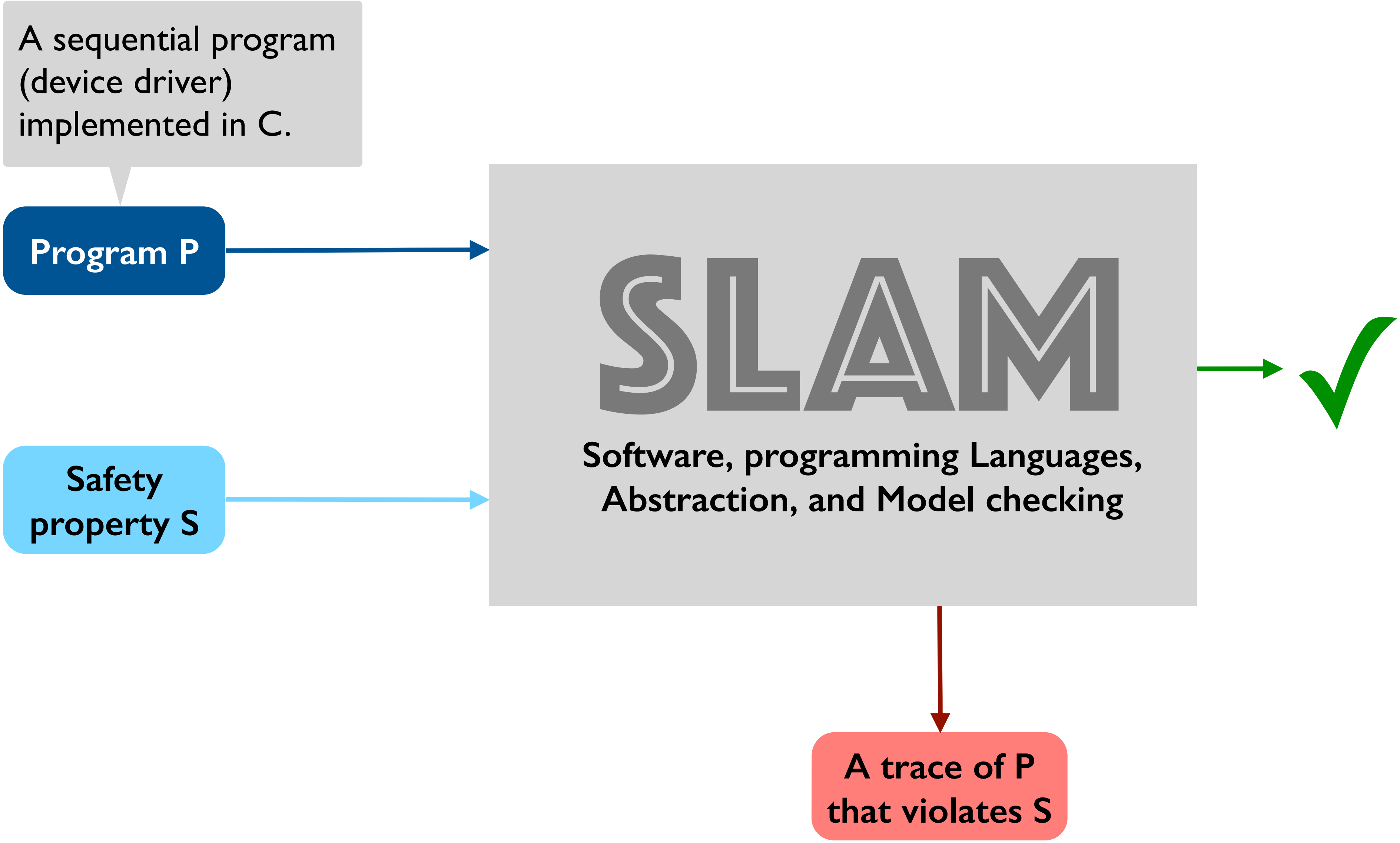
SLAM

Software, programming Languages, Abstraction, and Model checking

✓

A trace of P that violates S

# The SLAM process



A sequential program (device driver) implemented in C.

**Program P**

**Safety property S**

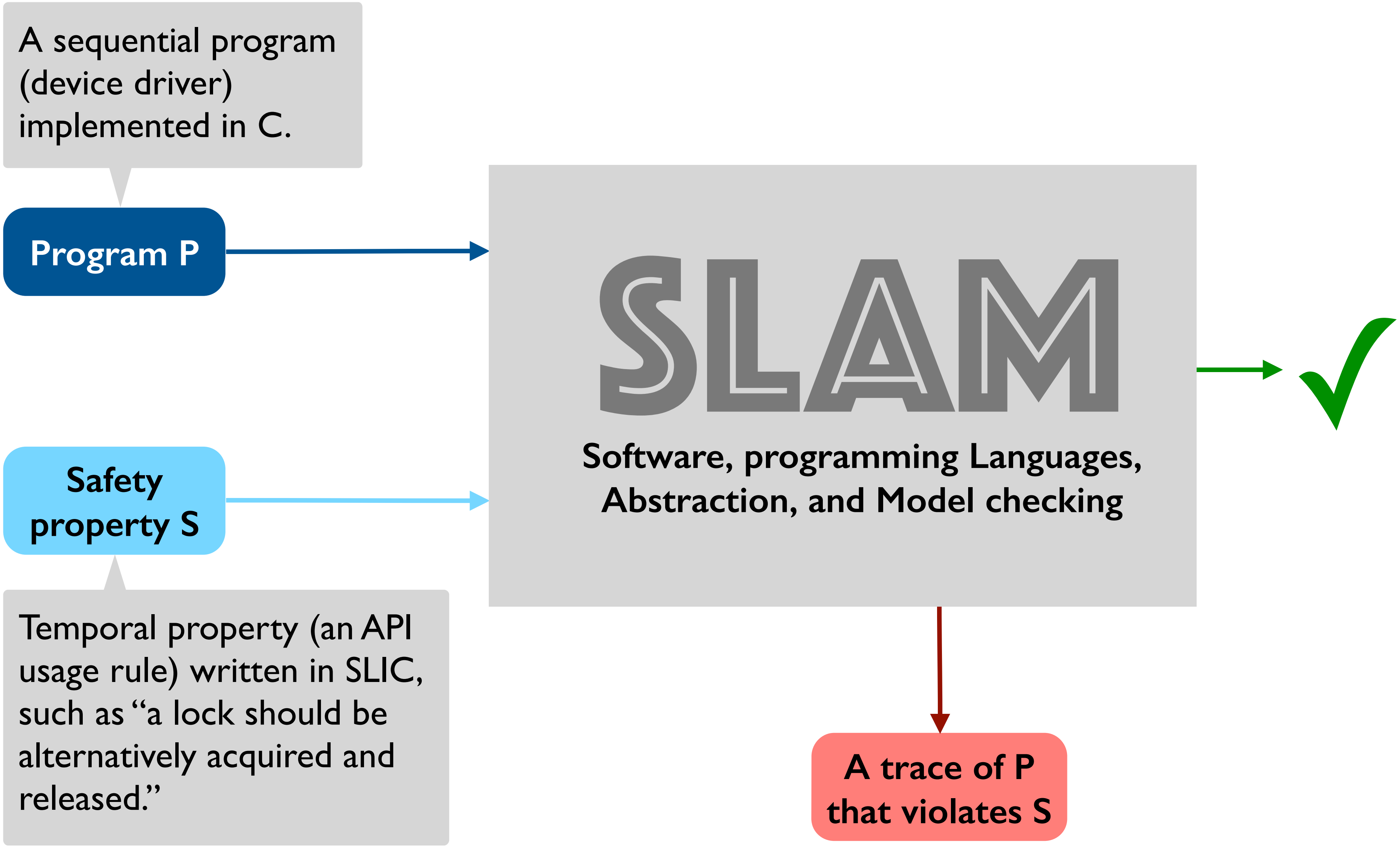
**SLAM**

**Software, programming Languages, Abstraction, and Model checking**

✓

**A trace of P that violates S**

# The SLAM process



A sequential program (device driver) implemented in C.

**Program P**

**Safety property S**

Temporal property (an API usage rule) written in SLIC, such as “a lock should be alternatively acquired and released.”

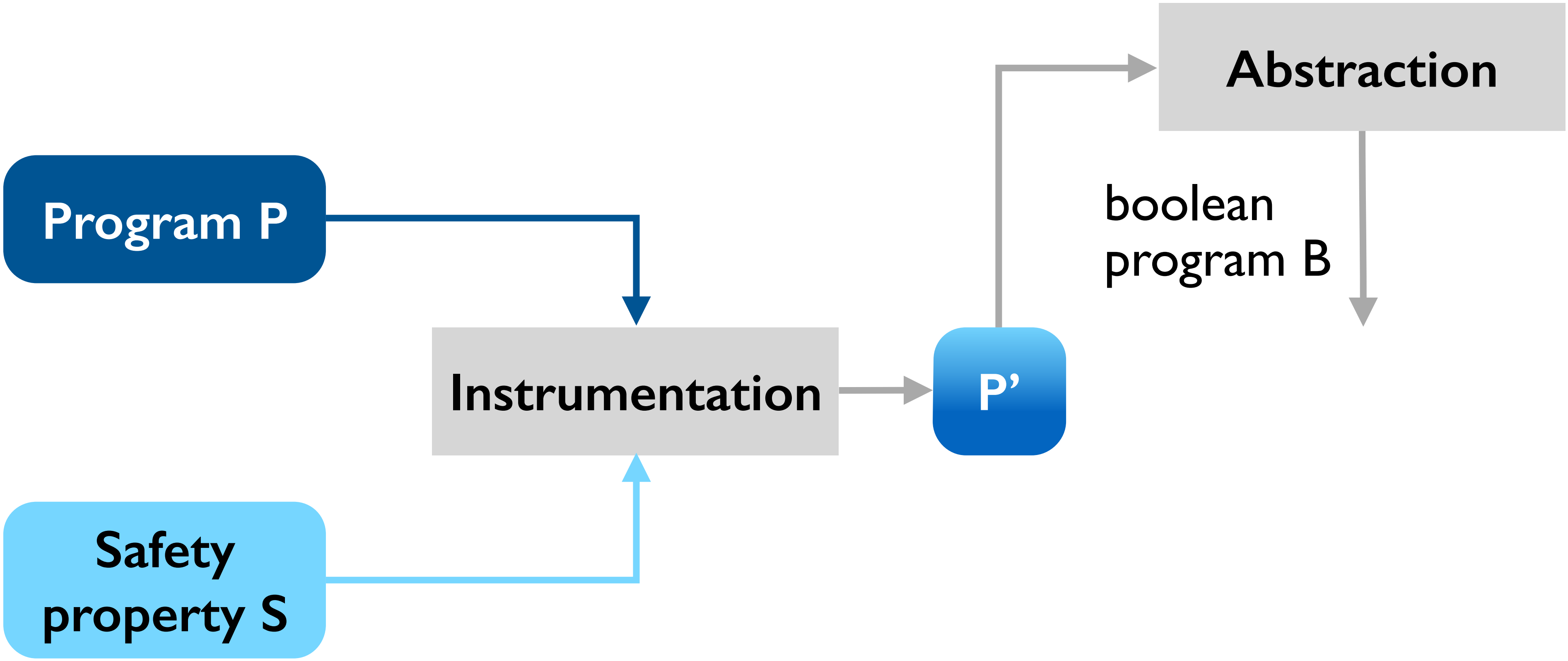
**SLAM**

**Software, programming Languages, Abstraction, and Model checking**

✓

**A trace of P that violates S**

# The SLAM process



Program P

Safety property S

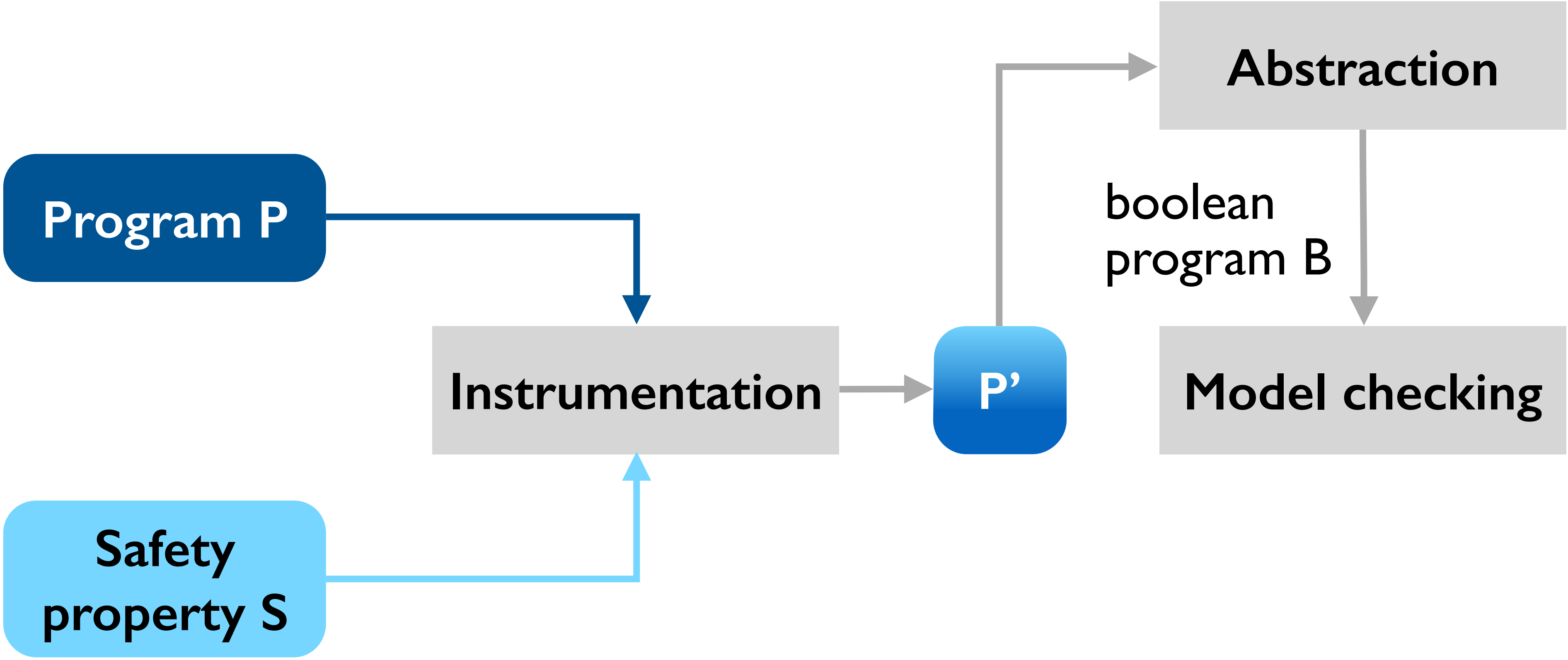
Instrumentation

P'

Abstraction

boolean program B

# The SLAM process



Program P

Safety property S

Instrumentation

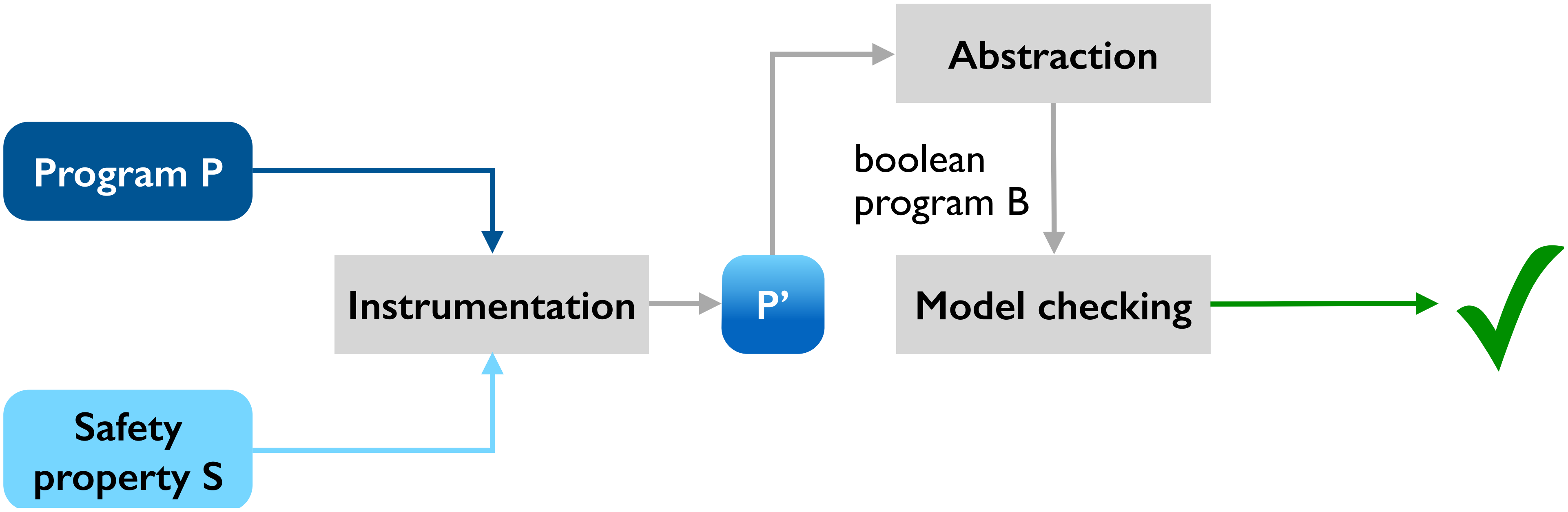
P’

Abstraction

boolean program B

Model checking

# The SLAM process



Program P

Safety property S

Instrumentation

P’

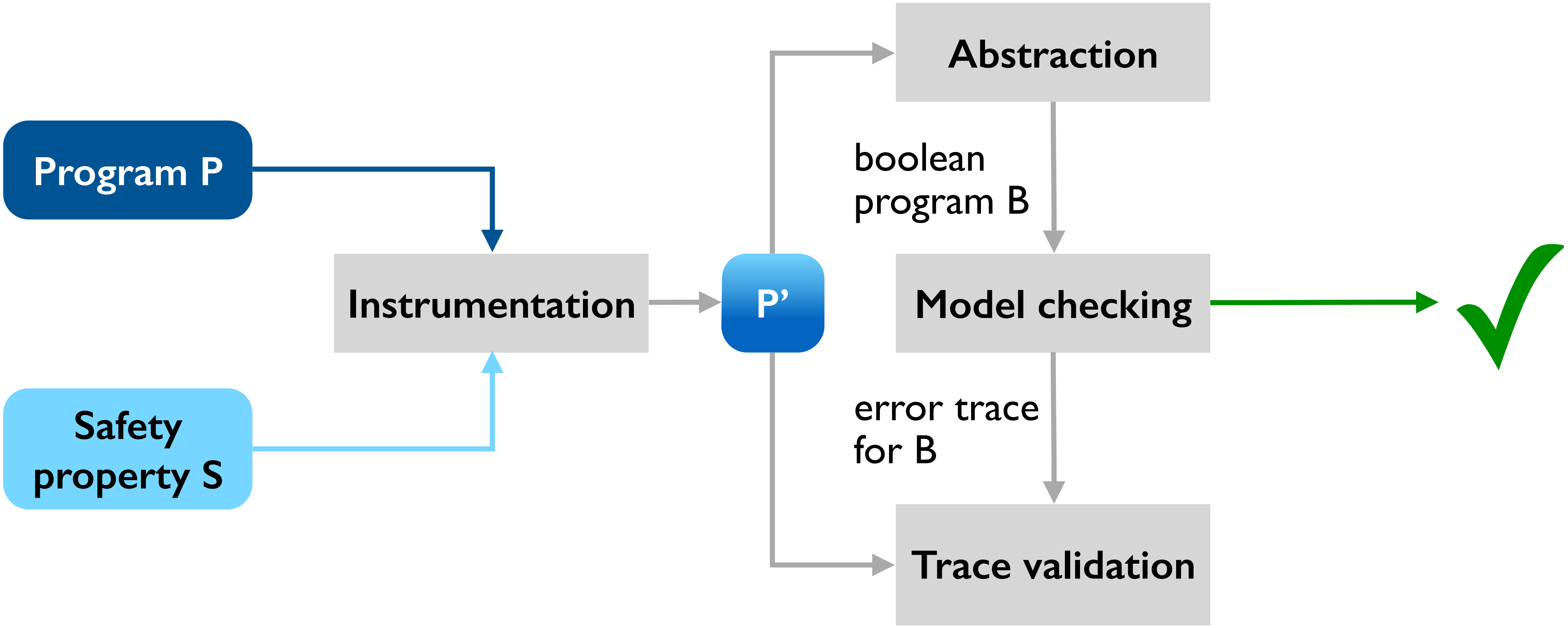
Abstraction

boolean program B

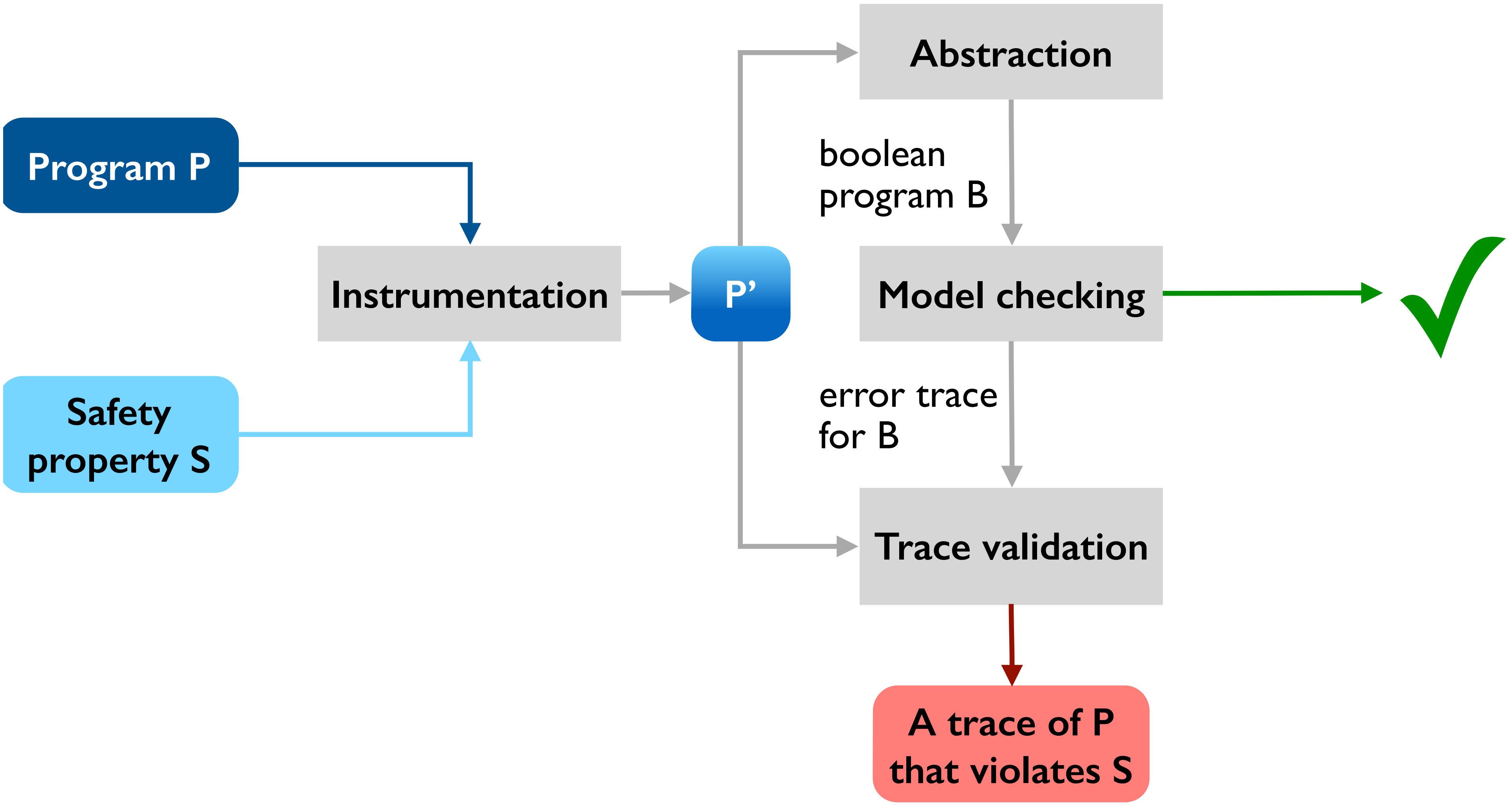
Model checking

✓

# The SLAM process



# The SLAM process



**Program P**

**Safety property S**

**Instrumentation**

**P’**

**Abstraction**

boolean program B

**Model checking**

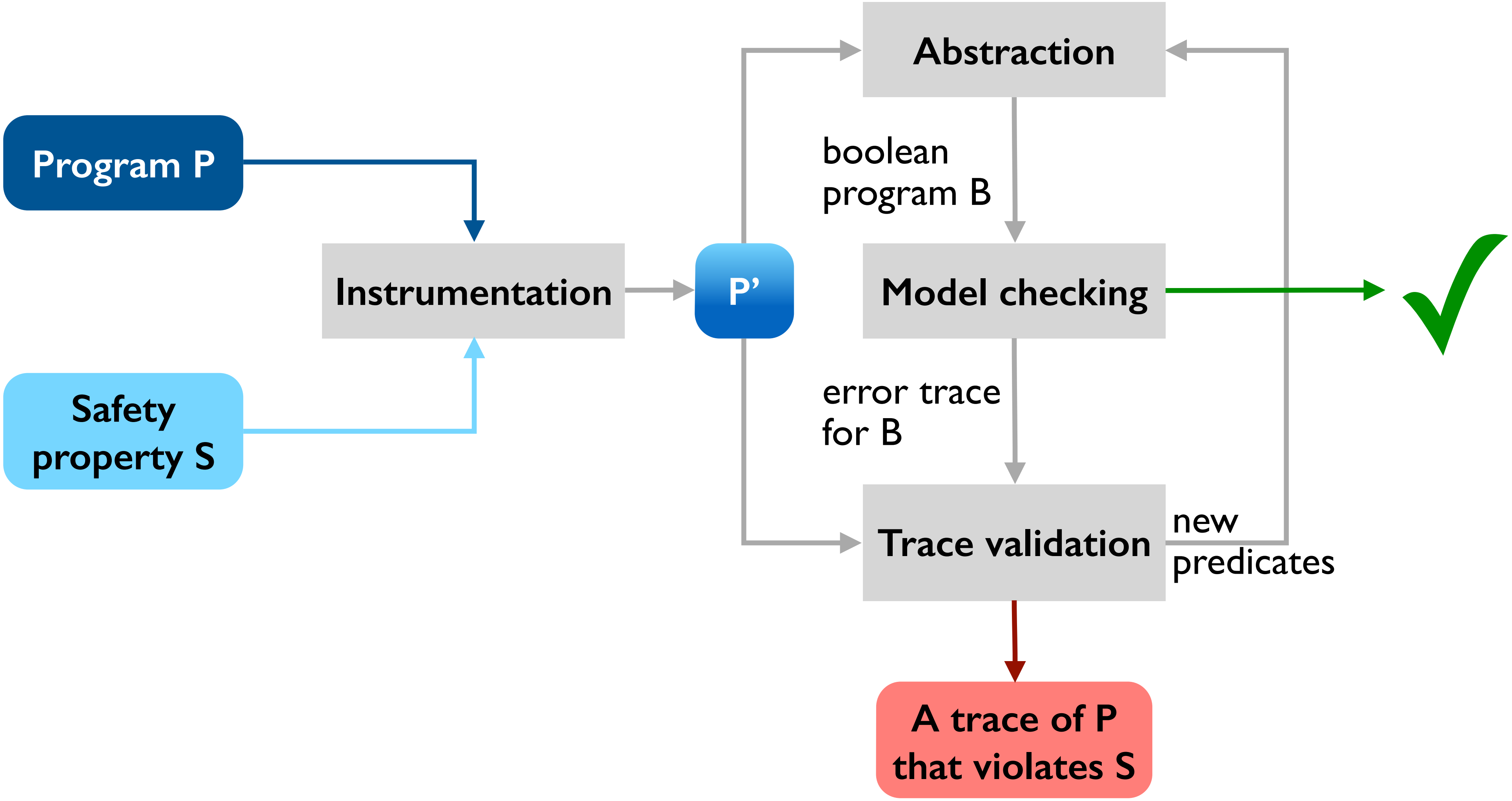
✓

error trace for B

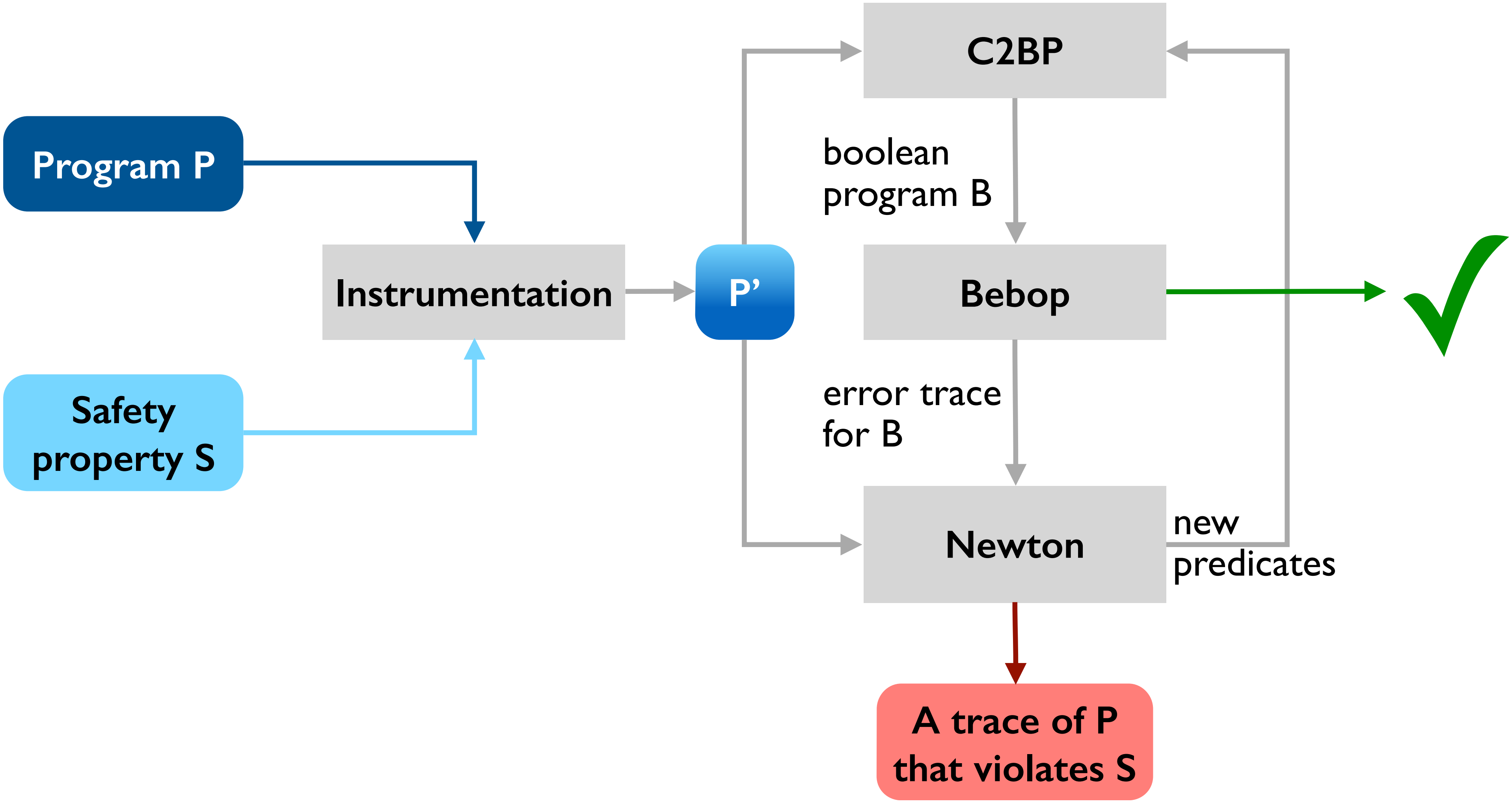
**Trace validation**

**A trace of P that violates S**

# The SLAM process



# The SLAM process



Program P

Safety property S

Instrumentation

P’

C2BP

boolean program B

Bebop

✓

error trace for B

Newton

new predicates

A trace of P that violates S

# Predicate Abstraction in BLAST

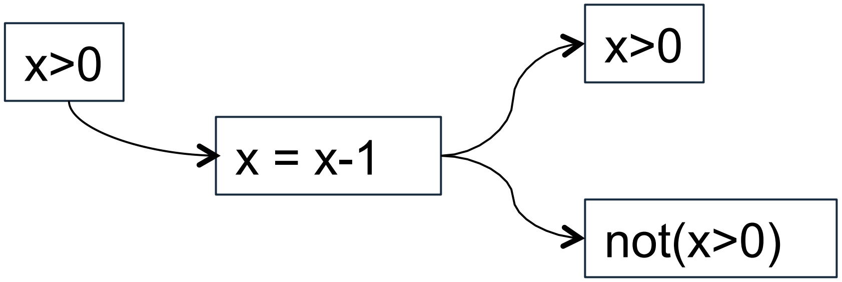
# Predicate Abstraction for M, s $\models$ P

- We need a simple way to come up with abstractions

- Our abstractions must be flexible
  - We need to be able to refine them on demand
  - This is how we identify spurious paths and eliminate them

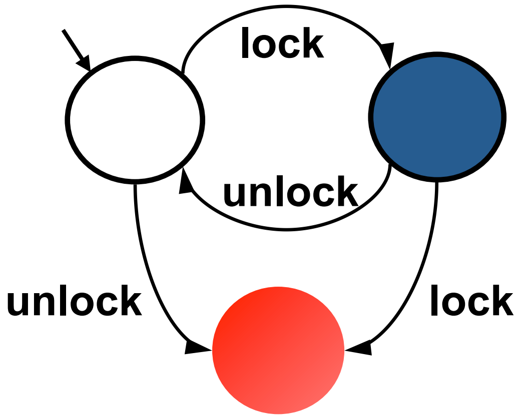
# Predicate Abstraction for M, s ⊨ P

- Abstract state s defined by a set of predicates
  - Examples: x > 0, p.next ≠ null, p.next.val > 0
- Transition function can be computed by a theorem prover
- Big idea:
  - We can refine the abstraction by introducing more predicates!



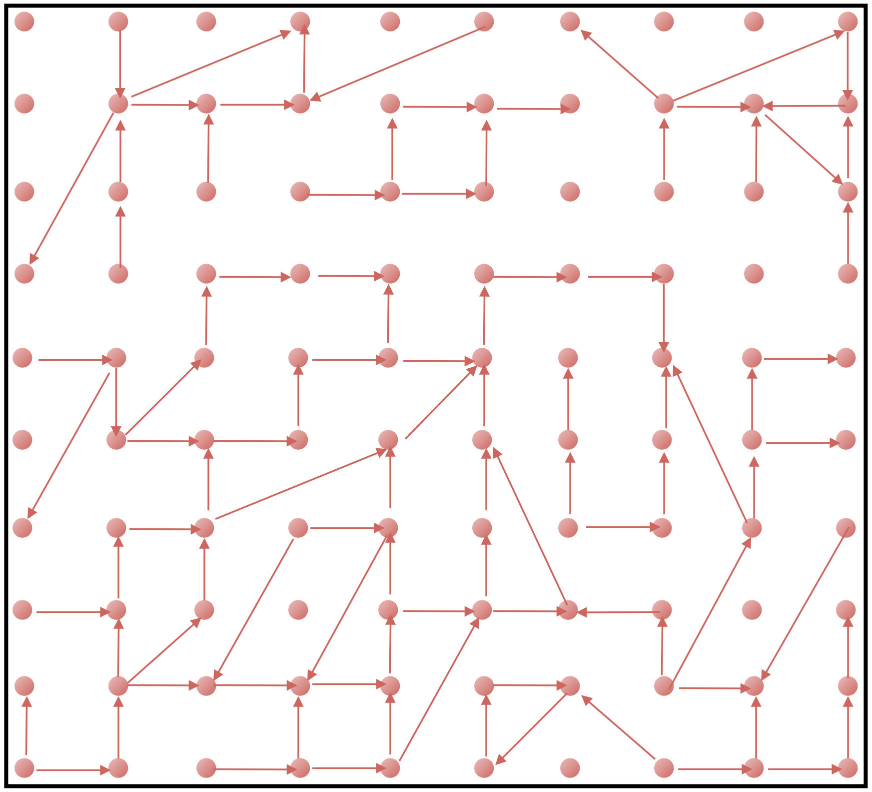
# Example

```
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;
}
```



Henzhinger, Jhala, Majumdar

# What a program *really* is…



**State**

**Transition**

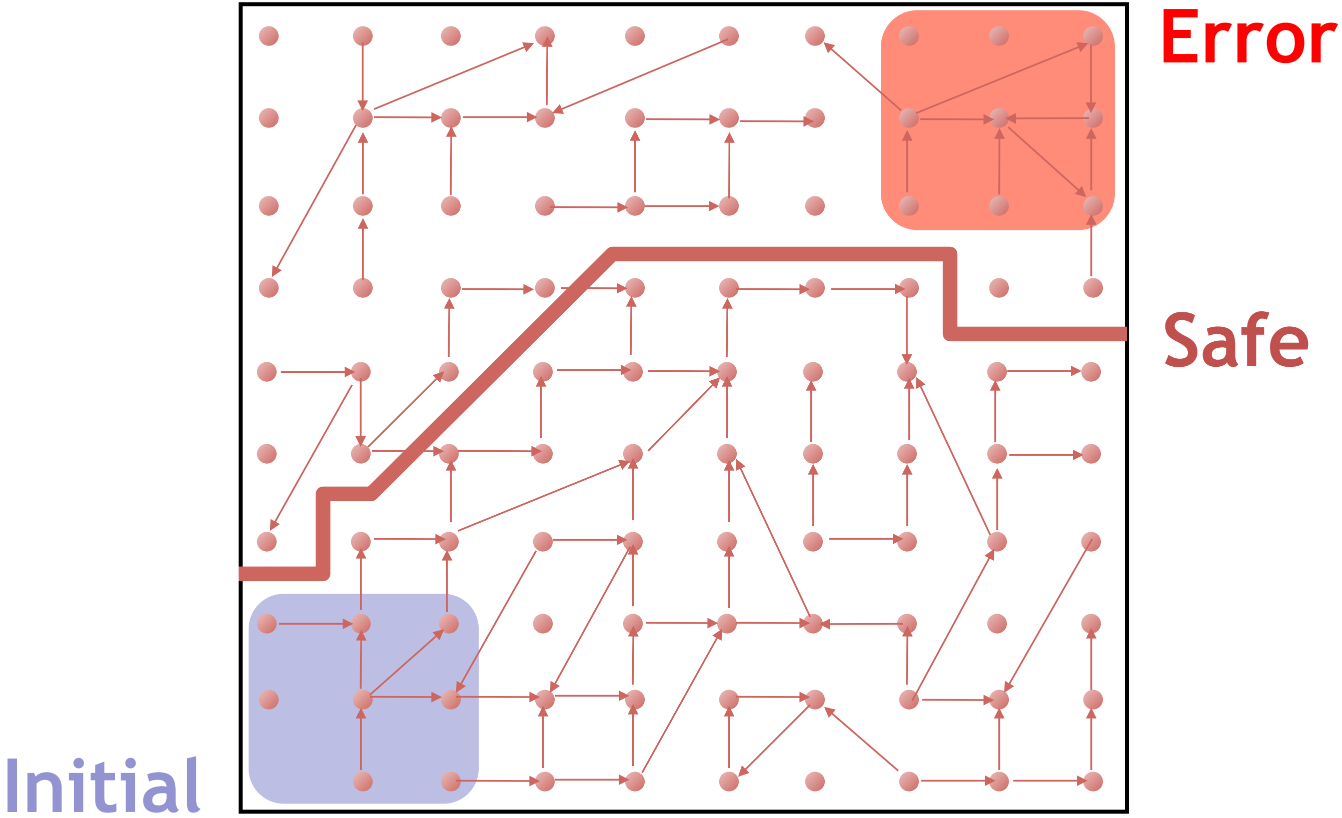
*pc* ↦ 3
lock ↦ ●
old ↦ 5
new ↦ 5
q ↦ 0x133a

```
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

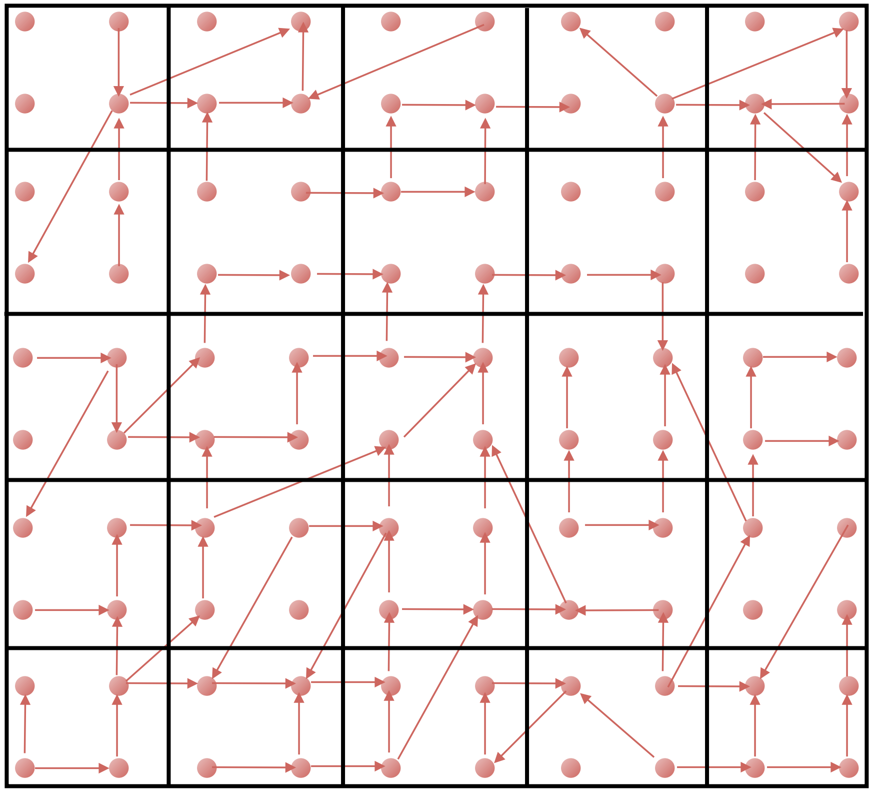


Is there a **path** from an **initial** to an **error** state ?

**Problem:** **Infinite** state graph

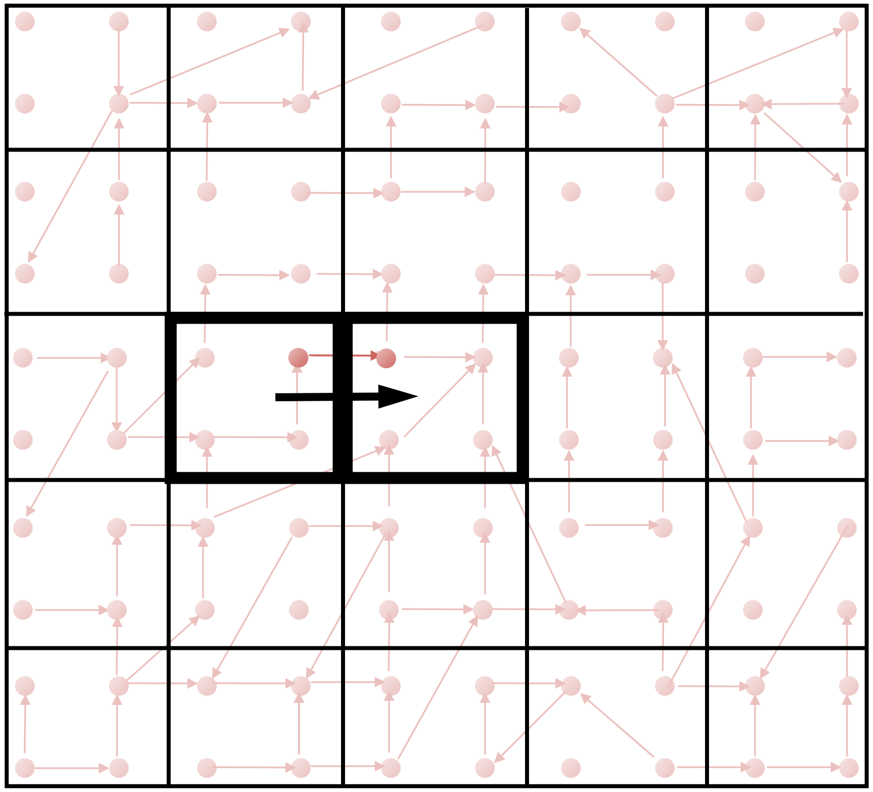
**Solution** : **Set** of states = logical **formula**

# Idea 1: Predicate Abstraction

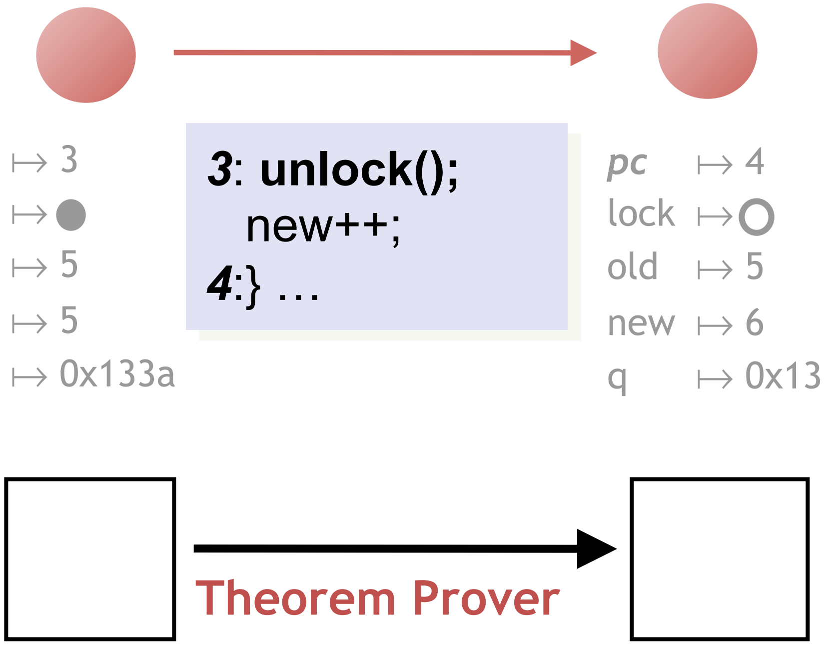


- **Predicates** on program state:
  *lock*
  *old = new*
- States satisfying **same** predicates are **equivalent**
  - **Merged** into one **abstract state**
- # abstract states is **finite**

# Abstract States and Transitions



**State**



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

```
3: unlock();
   new++;
4:} …
```

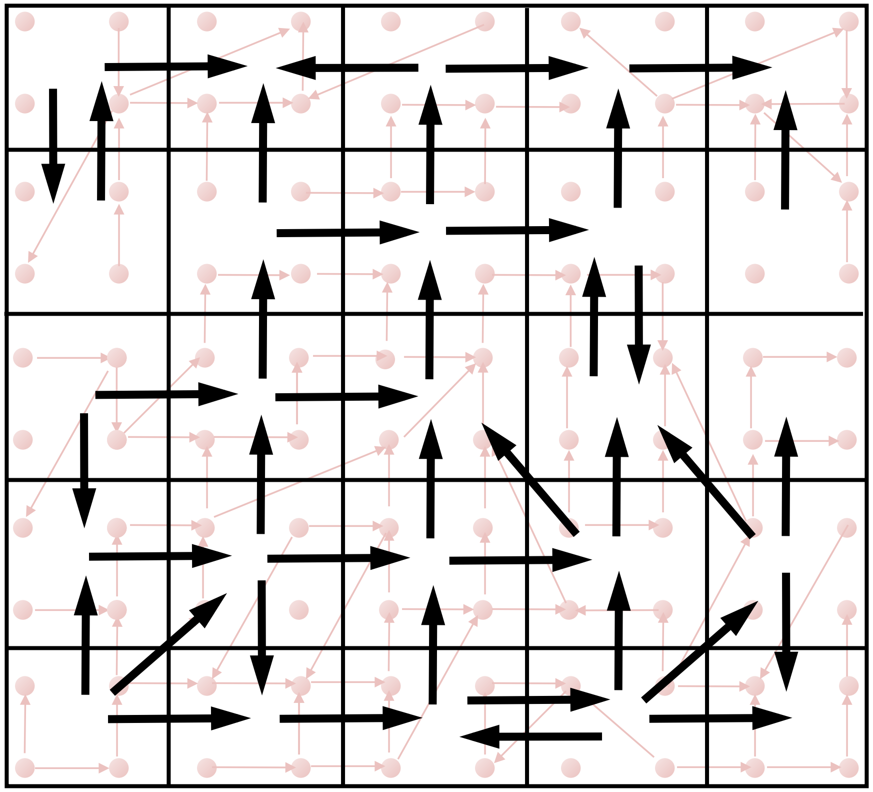
*pc* ↦ 4
lock ↦ ○
old ↦ 5
new ↦ 6
q ↦ 0x133a

Theorem Prover

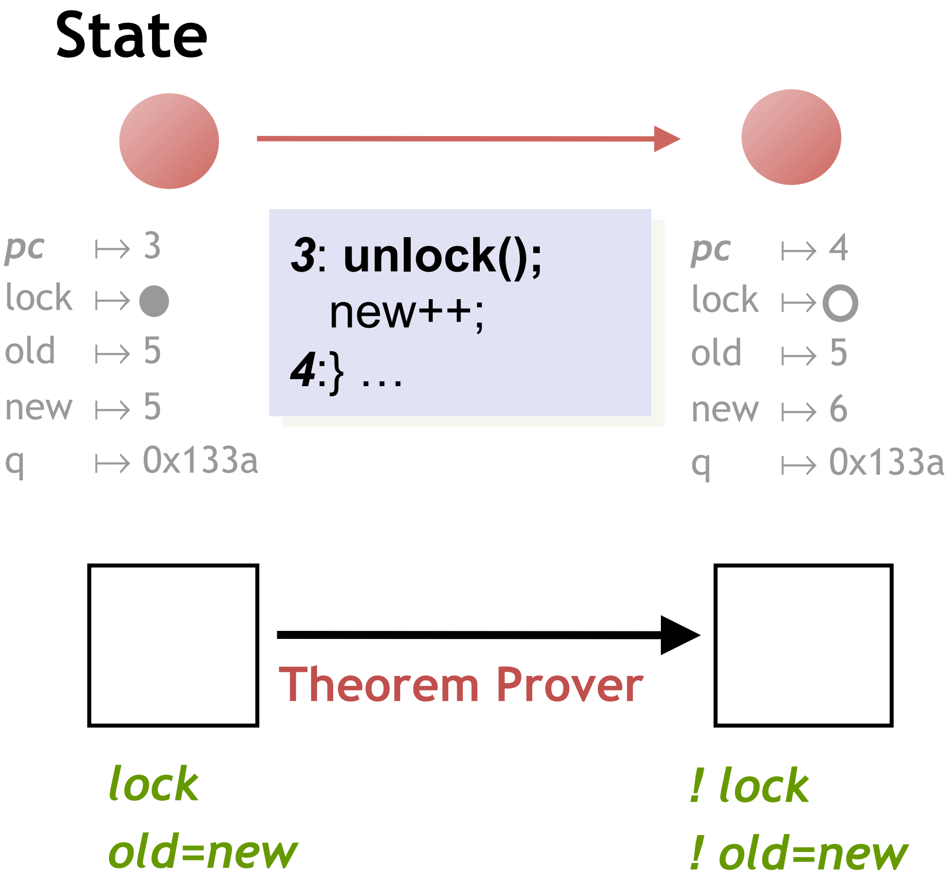
*lock*
*old=new*

*!lock*
*! old=new*

# Abstraction



Existential Lifting



**State**

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

```
3: unlock();
   new++;
4:} …
```

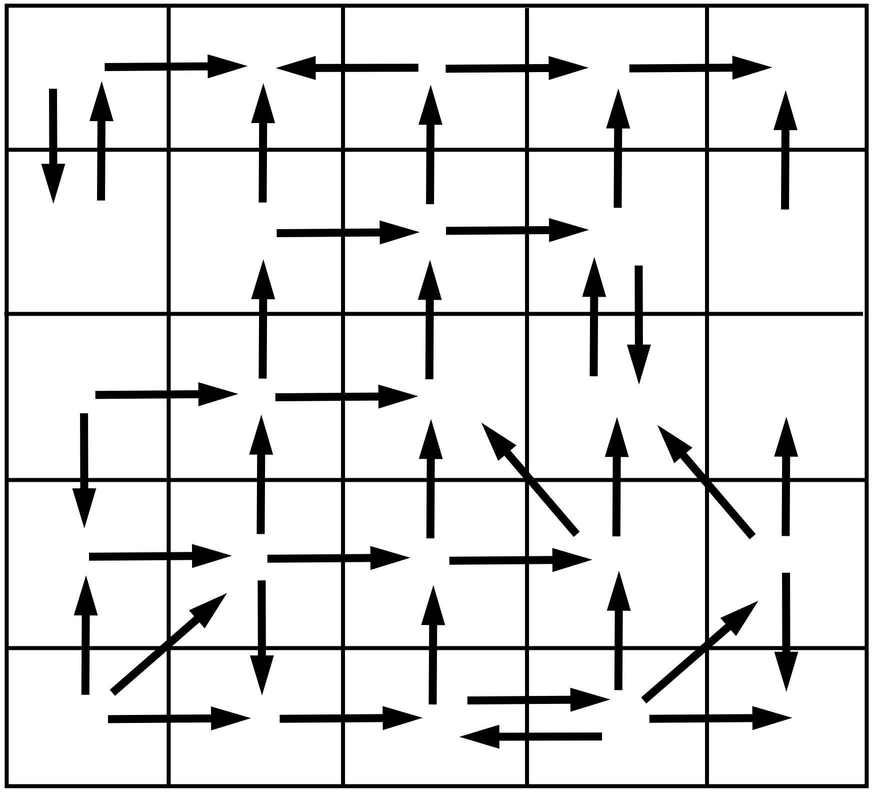
*pc* ↦ 4
lock ↦ ○
old ↦ 5
new ↦ 6
q ↦ 0x133a

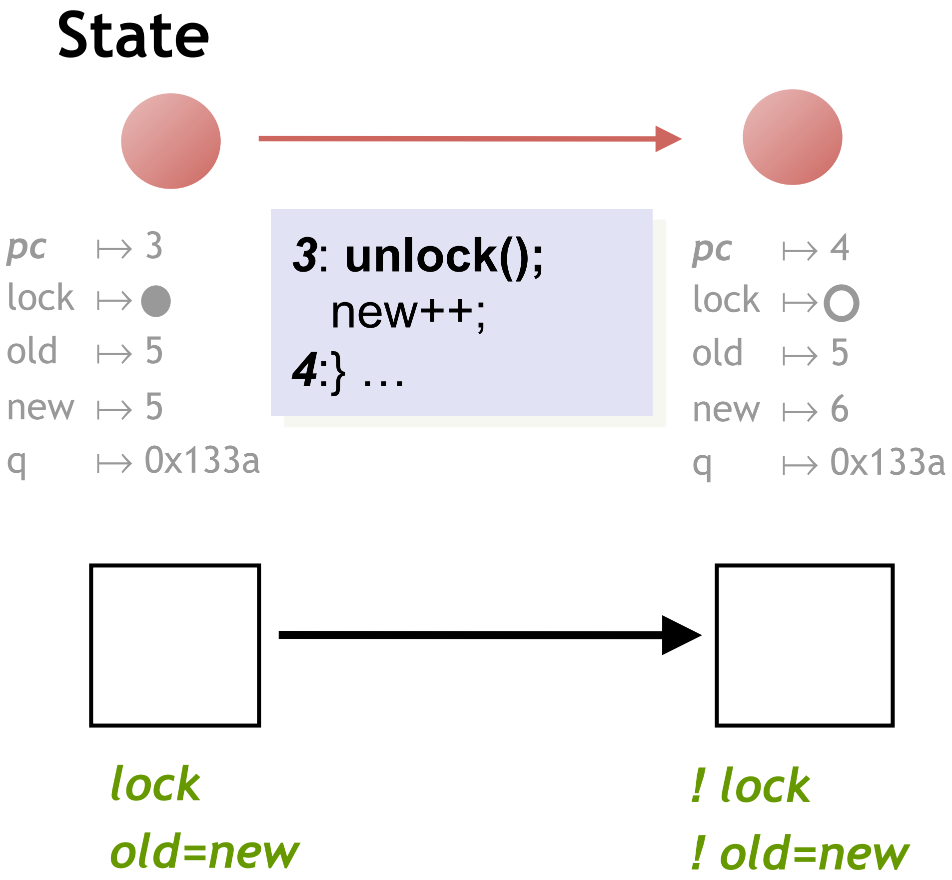
Theorem Prover

*lock*
*old=new*

*! lock*
*! old=new*

# Abstraction





**State**

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

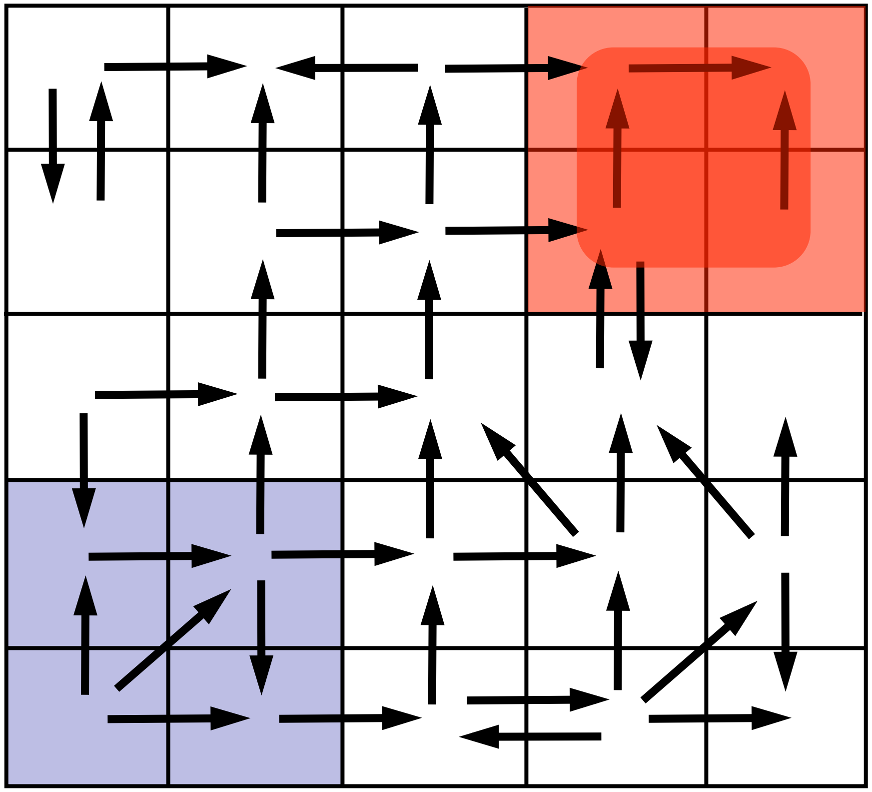
```
3: unlock();
   new++;
4:} …
```

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

*lock*
*old=new*

*! lock*
*! old=new*

# Analyze Abstraction



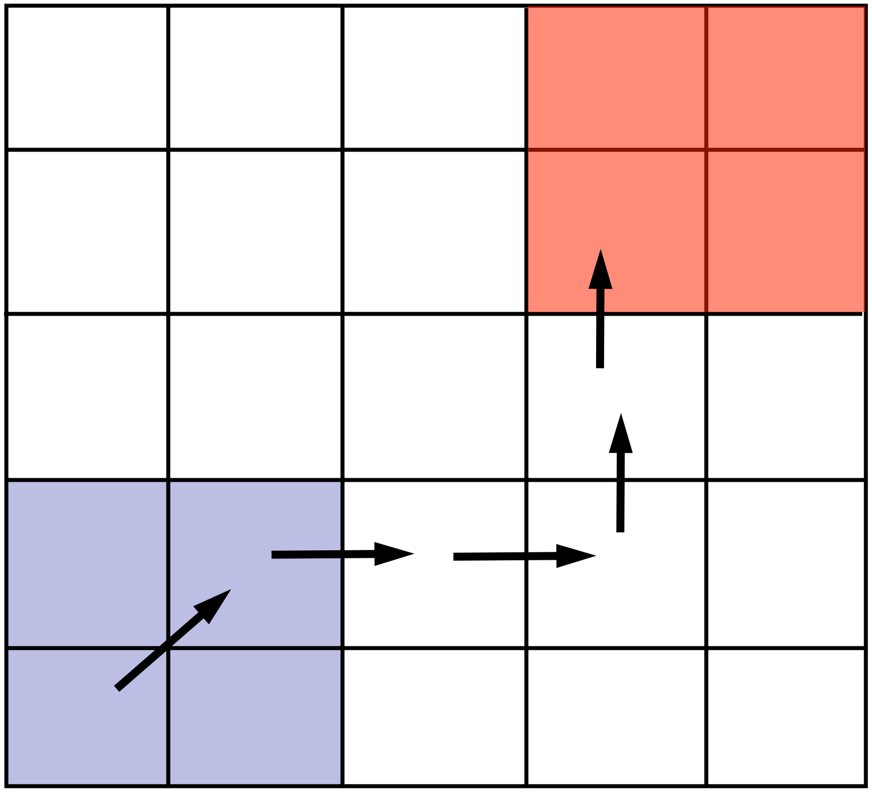
Analyze finite graph

No **false negatives**

**Problem**

Spurious **counterexamples**

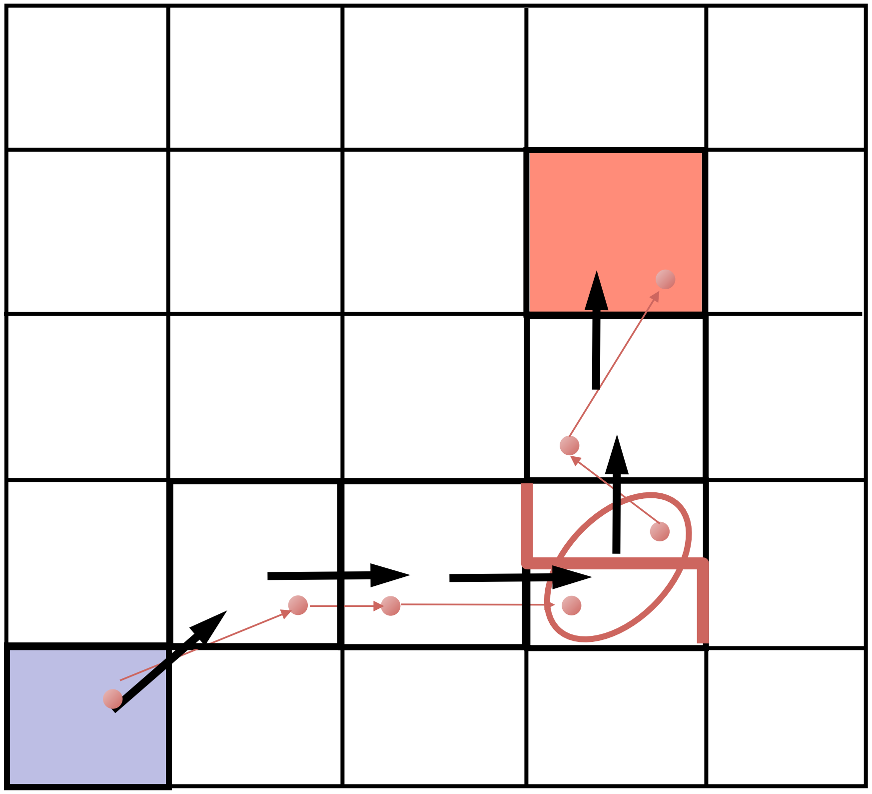
# Idea 2: Counterex.-Guided Refinement



**Solution**

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

# Idea 2: Counterex.-Guided Refinement



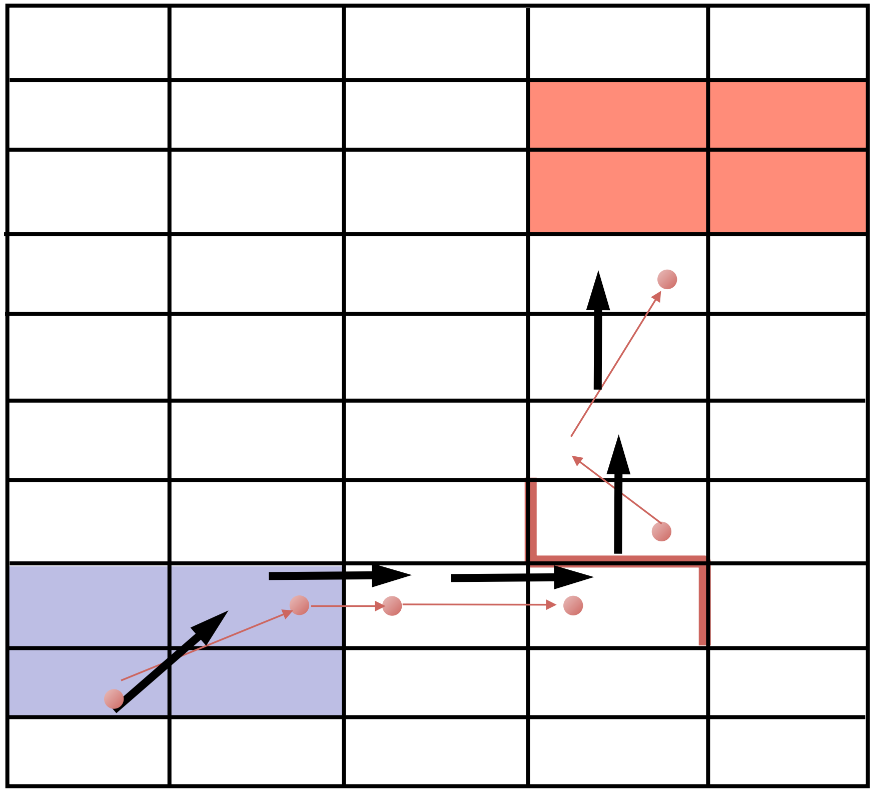
**Solution**

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



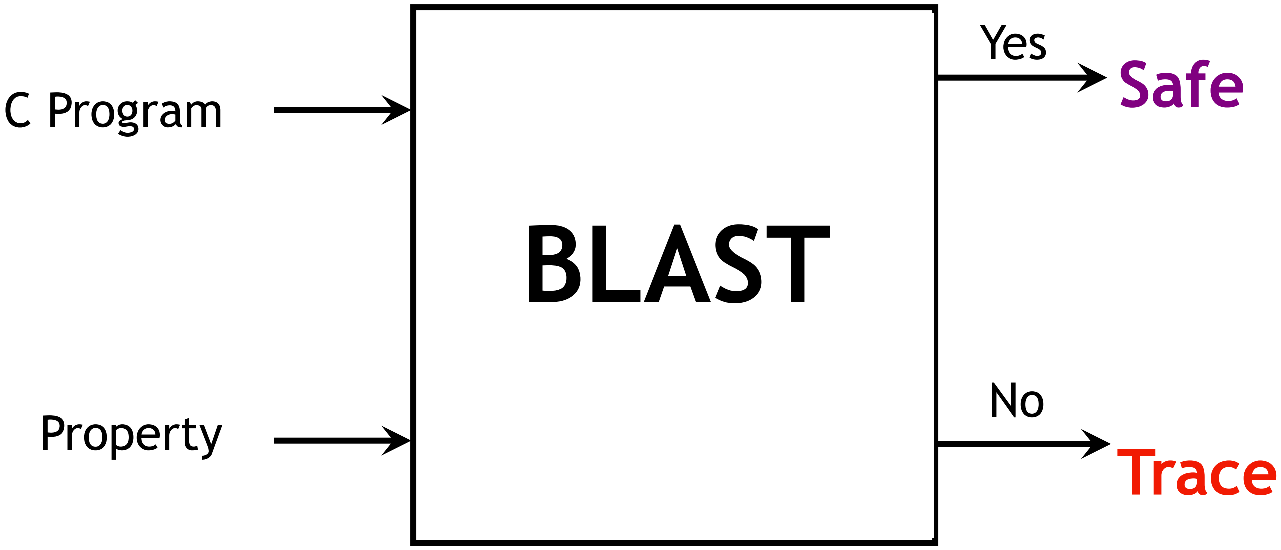
[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

**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

# Lazy Abstraction



C Program

Property

**BLAST**

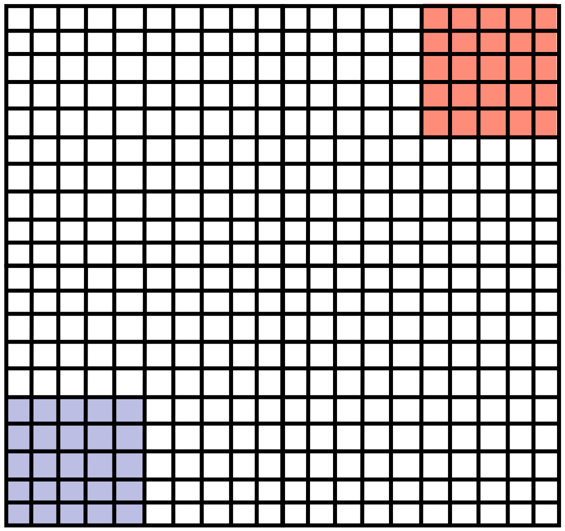
Yes

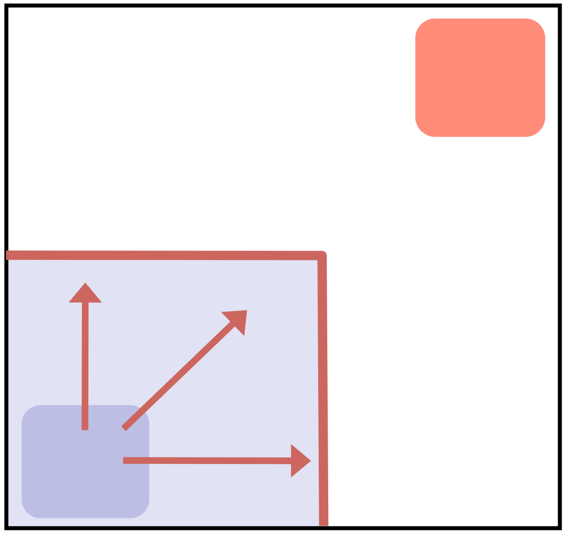
**Safe**

No

**Trace**

# **Problem:** Abstraction is Expensive





**Reachable**

## **Problem**

#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

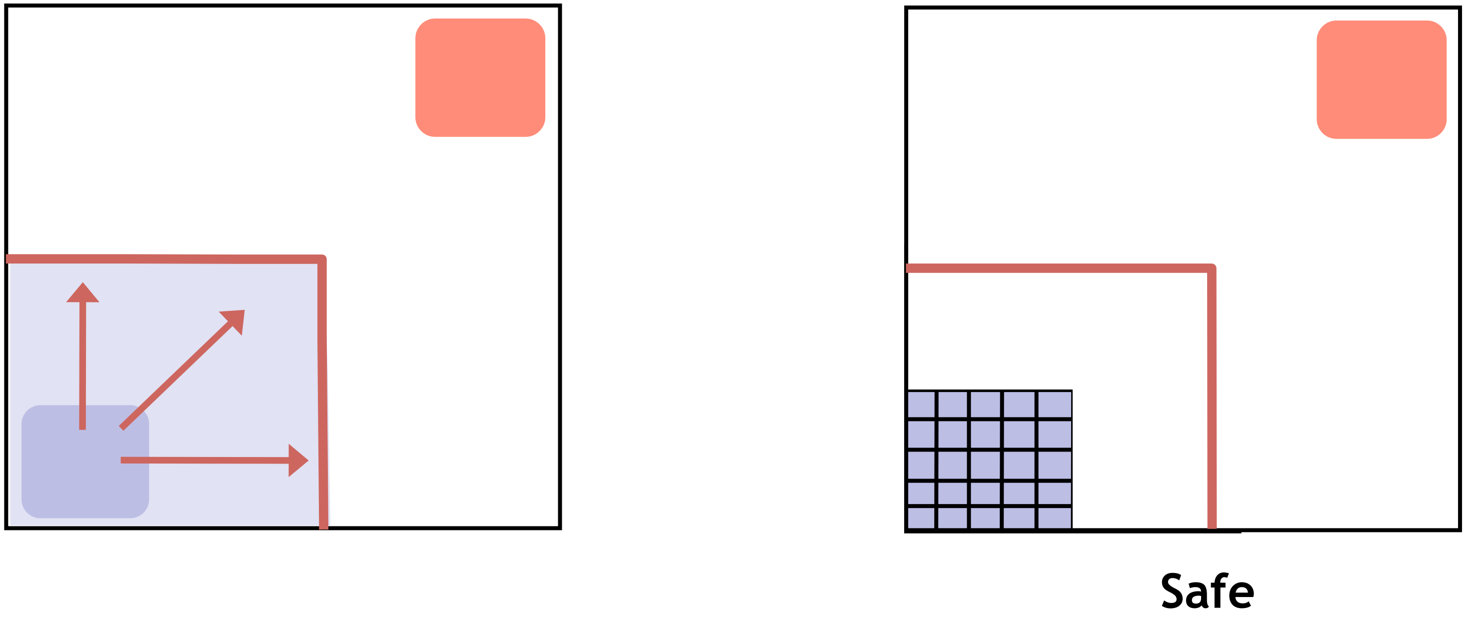
## **Observe**

Fraction of state space reachable

#Preds ~ 100’s, #States ~ $2^{100}$ ,

#Reach ~ 1000’s

# Solution1: Only Abstract Reachable States



## Problem

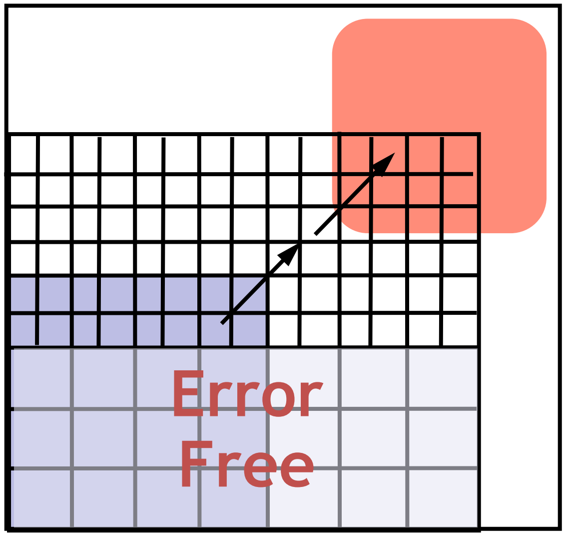
#abstract states = $2^{\#\text{predicates}}$

Exponential Thm. Prover queries

## Solution

Build abstraction during search

# Solution2: Don’t Refine Error-Free Regions



**Problem**

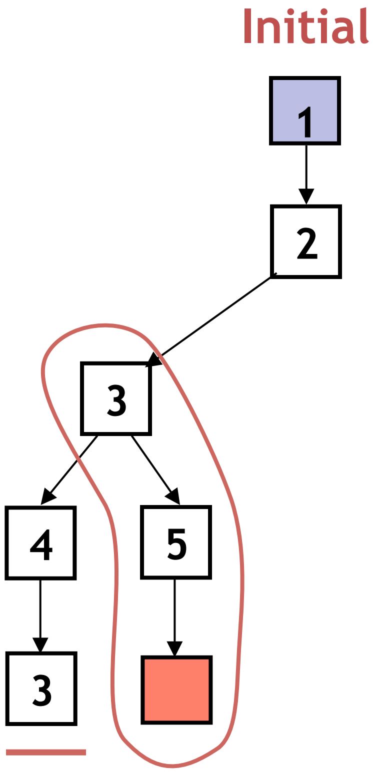
#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

**Solution**

Don’t refine error-free regions

# **Key Idea:** Reachability Tree



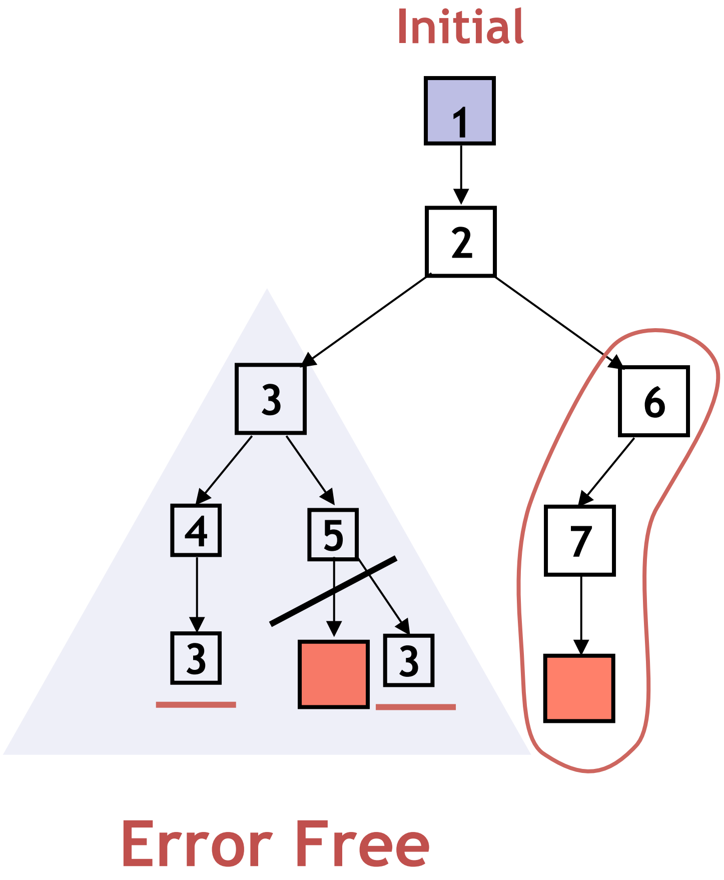
## Unroll Abstraction

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



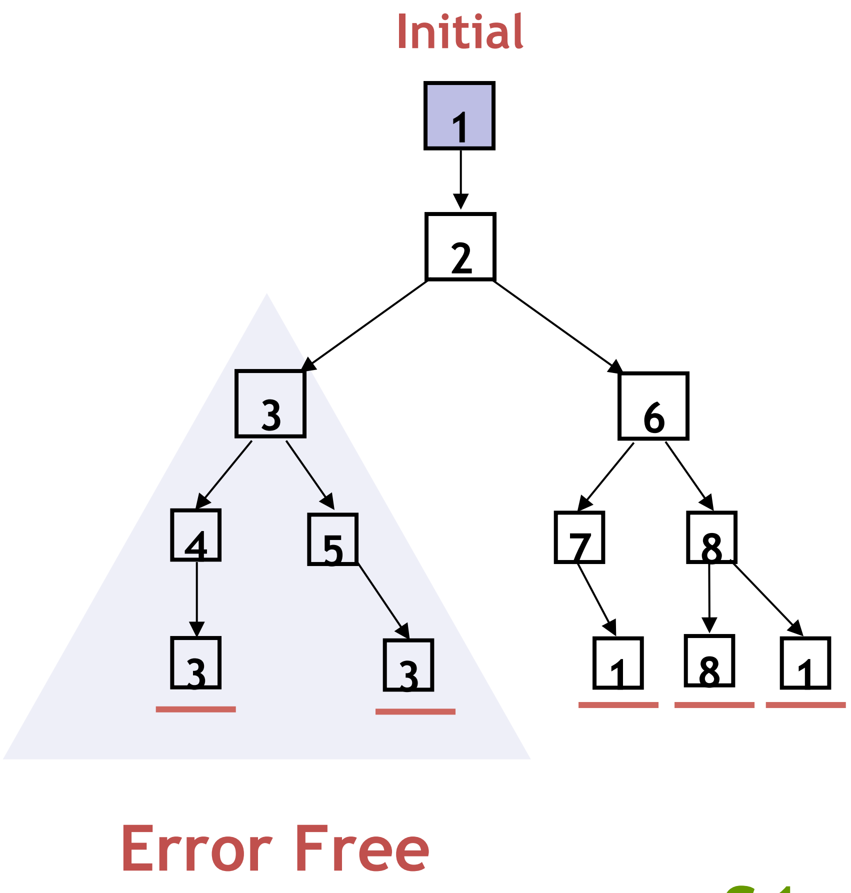
## Unroll Abstraction

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.

**SAFE**

**S1:** Only Abstract Reachable States

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