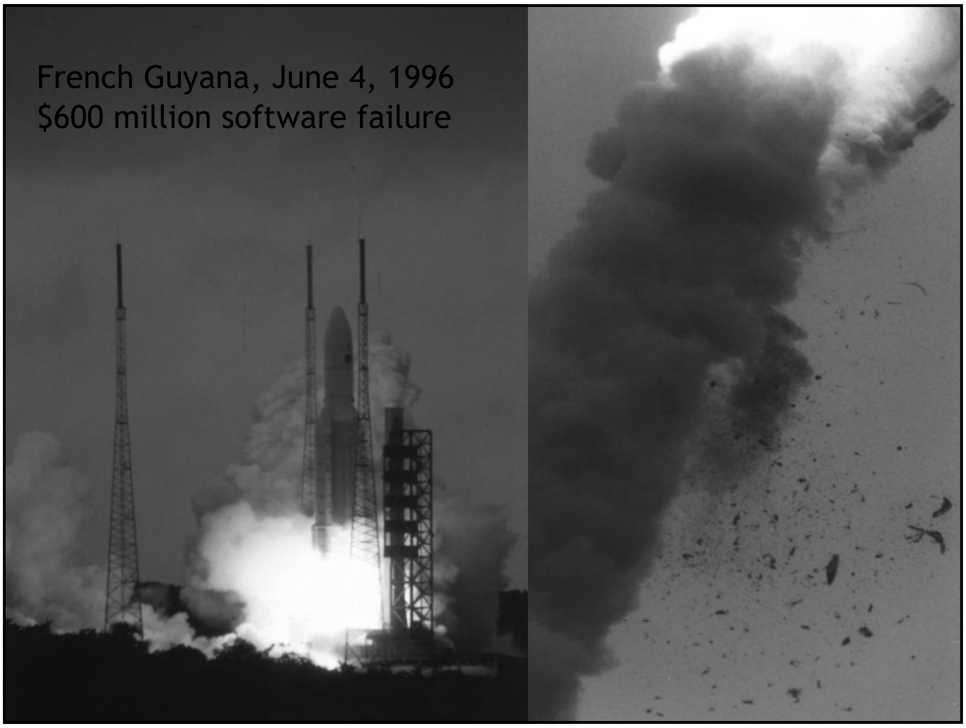
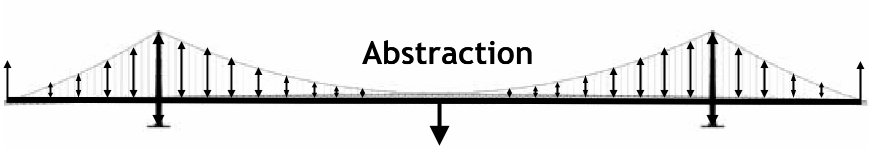
Scalable Program Verification by Lazy Abstraction

Ranjit Jhala
U.C. Berkeley


French Guyana, June 4, 1996
$600 million software failure


Something Reliable

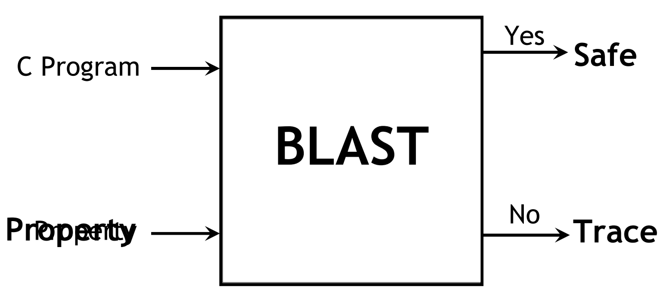
Why don’t Bridges Crash ?



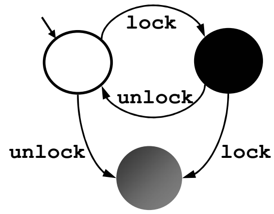
Building Blocks	Bridges Mechanics	Programs Logic
1. Relevant facts*	Mass, Tensile Strength	?
2. Model	Free Body Diagram	?
3. Analysis	Solve Equations	?

* w.r.t. **property** of interest

Contributions



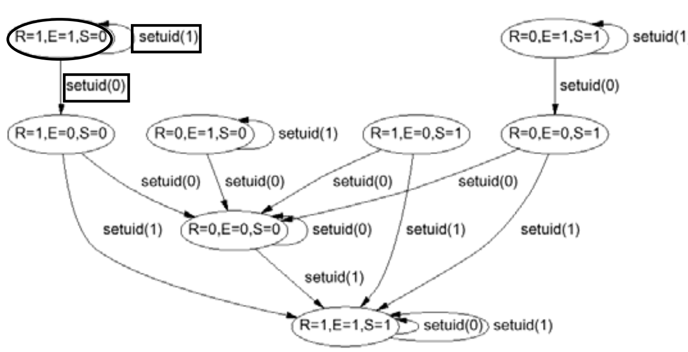
Property 1: Double Locking



*"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

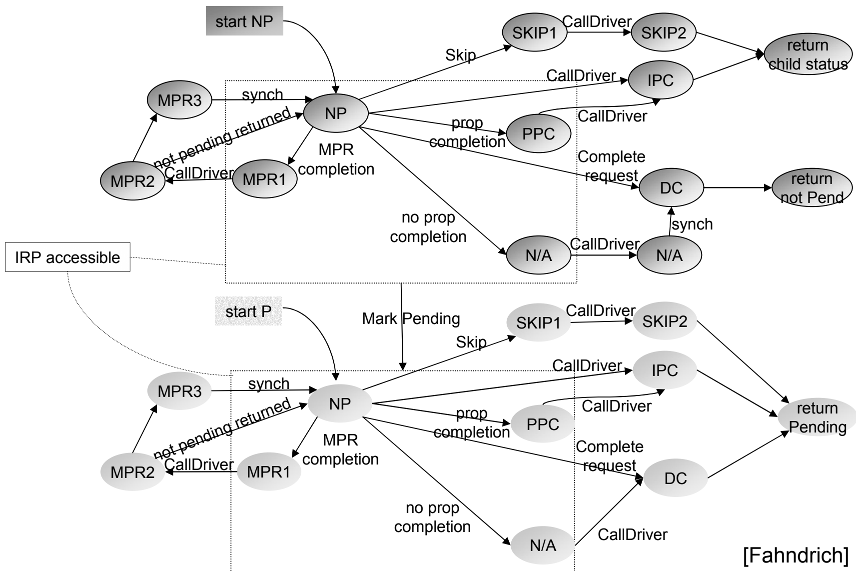


[Chen-Dean-Wagner '02]

"User applications must not run with root privilege"

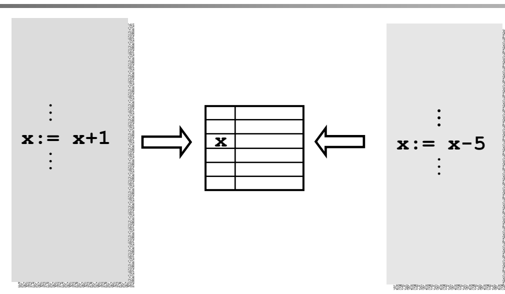
When **execv** is called, must have **suid** ≠ **0**

Property 3 : IRP Handler



[Fahndrich]

Property 4: Data Races

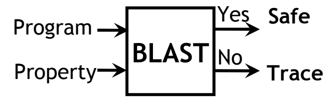


A **data race** on **x** is a state where:

1. **Two** threads can access **x**
2. One of the accesses is a **write**

There should be **no races** on shared variables

Contributions



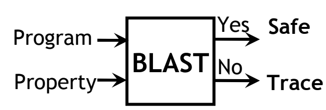
Sequential Programs

Counterex.-Guided **Abstraction-Refinement**
For large programs, complex properties
New Algorithms: **Abstraction [POPL 02],Refinement [POPL 04]**

Property 1: **Double Locking** (Linux/Windows Drivers)
Property 2: **Drop Root Privilege** (Linux Daemons ~59kloc)
- Precise: No false Errors

Property 3: **IRP Handler** (NT Drivers ~130Kloc)
- Large Programs

Contributions



Multithreaded Programs

New **models** for thread interactions
New algorithms to compute models and
Verify multithreaded programs **[CAV 03] [PLDI 04]**

Property 4: **Data Races**

- Linux/Windows Drivers
- Sensor Network Apps. (TinyOS/NesC) ~10kloc
- Arbitrarily many threads
- Any synchronization mechanisms
- Real counterexamples, Safety Proofs

Plan

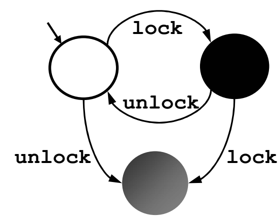
1. **C.G. Abstraction-Refinement**

2. Lazy Abstraction
   - Sequential Programs
   - Multithreaded Programs

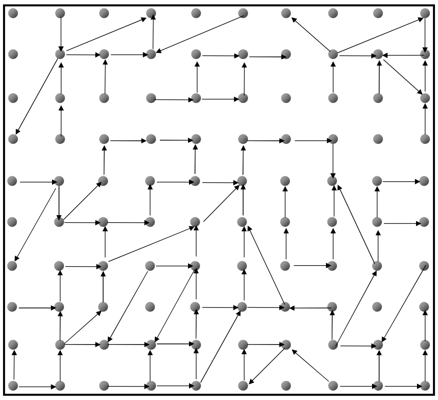
3. Future Work

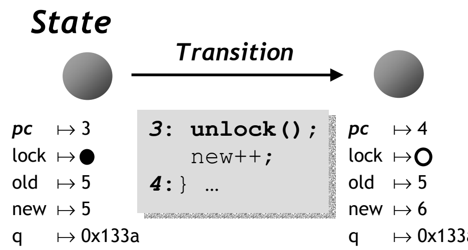
Example

```
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;
}
```



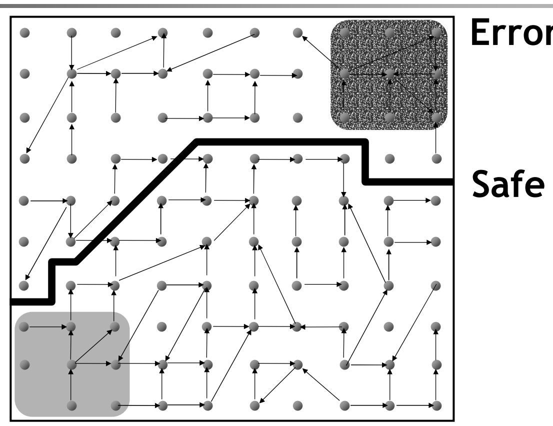
What a program *really* is…






```
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

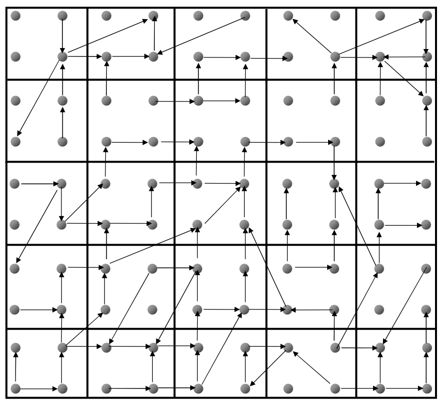


Is there a **path** from an **initial** to an **error** state ?

Problem: Infinite state graph

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

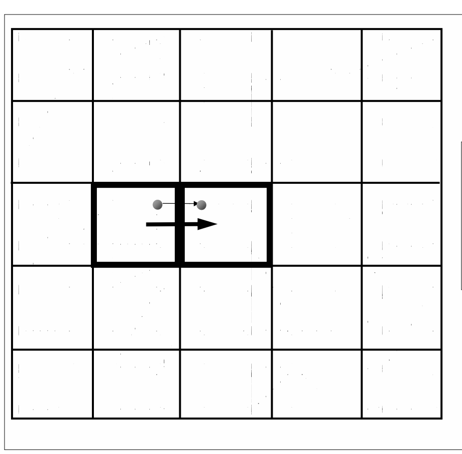
Idea 1: Predicate Abstraction

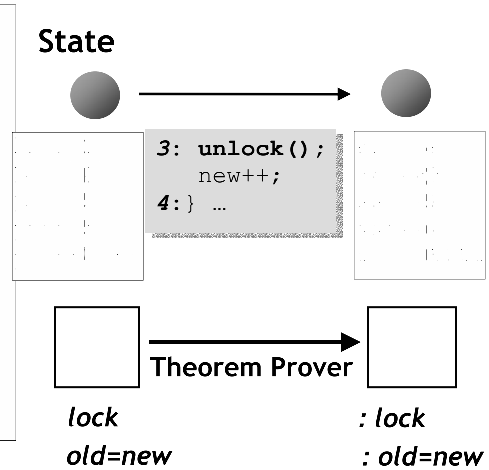


- **Predicates** on program state:
  lock
  old = new
- States satisfying **same** predicates are **equivalent**
  - **Merged** into one **abstract state**
- #abstract states is **finite**

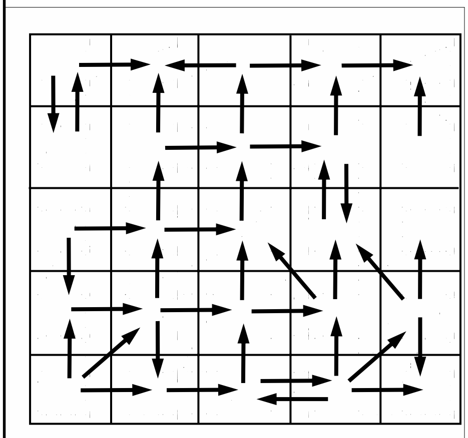
[Graf-Saidi 97]

Abstract States and Transitions



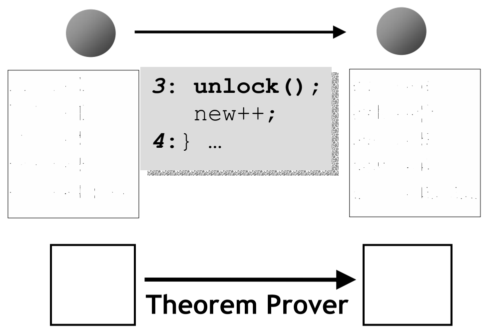


Abstraction



Existential Lifting

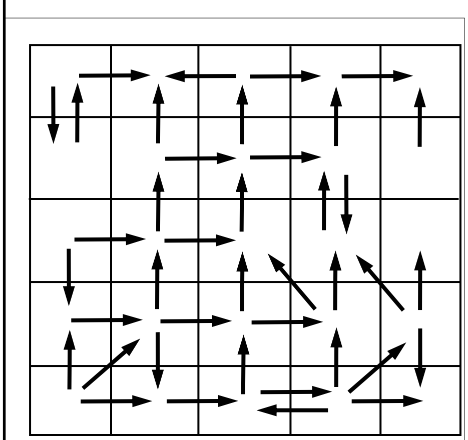
State



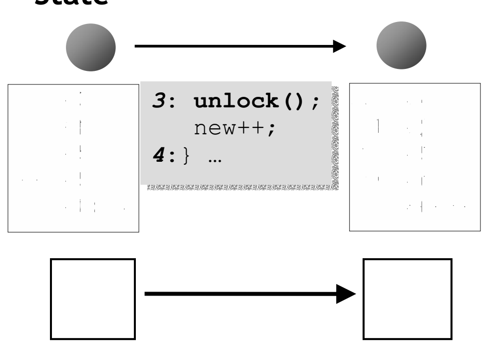
lock   *: lock*

old=new   *: old=new*

Abstraction



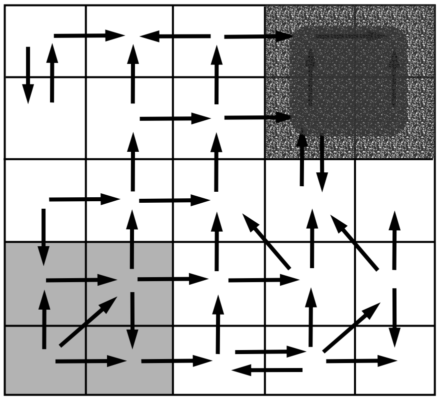
State



lock   *: lock*

old=new   *: old=new*

Analyze Abstraction



Analyze finite graph

Over Approximate:

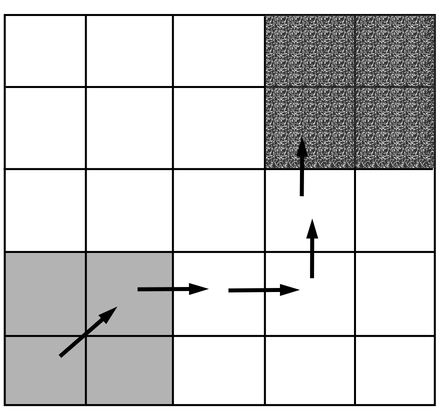
Safe) System Safe

No **false negatives**

Problem

Spurious **counterexamples**

Idea 2: Counterex.-Guided Refinement

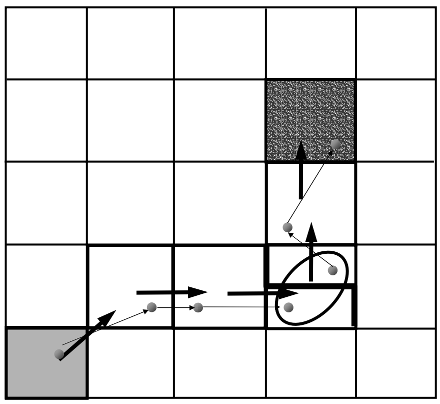


Solution

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

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

Idea 2: Counterex.-Guided Refinement



Solution

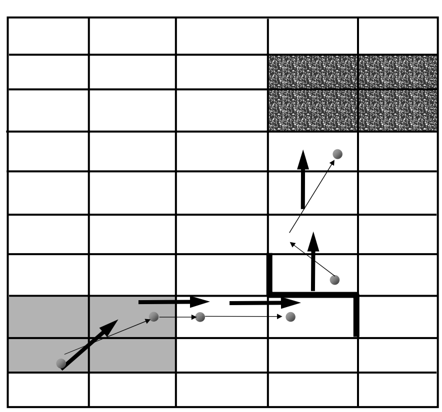
Use spurious **counterexamples** to **refine** abstraction

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

Imprecision due to merge

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

Iterative Abstraction-Refinement



Solution

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

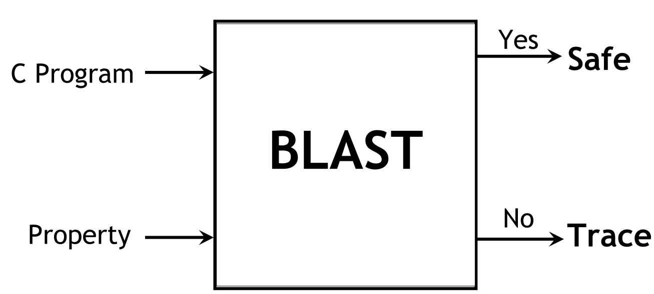
1. Add predicates to distinguish states across **cut**
2. Build **refined** abstraction
   - -eliminates counterexample
3. **Repeat** search
   Till real counterexample or system proved safe

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

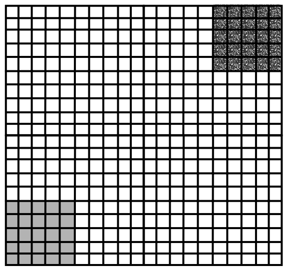
Plan

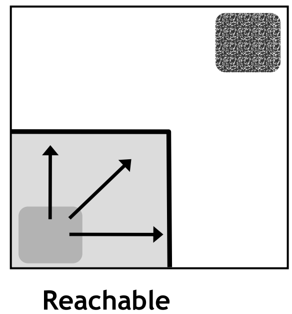
1. C.G. Abstraction-Refinement
2. Lazy Abstraction
   - **Sequential Programs** [POPL 02] [POPL04]
   - Multithreaded Programs
3. Future Work

Scaling Sequential Verification



Problem: Abstraction is Expensive





Problem

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

Exponential Thm. Prover queries

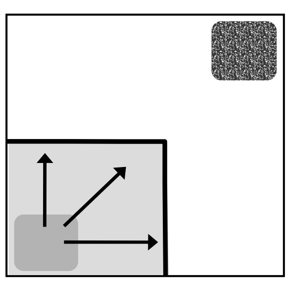
Observe

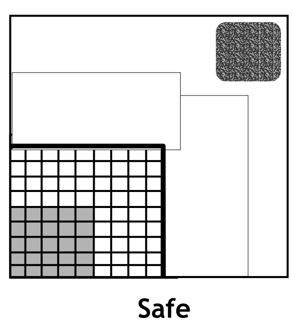
Fraction of state space reachable

#Preds ~ 100's, #States ~ 2^{100} ,

#Reach ~ 1000's

Solution1: Only Abstract Reachable States





Problem

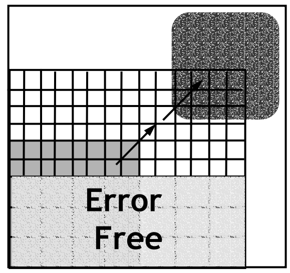
#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Solution

Build abstraction **during** search

Solution2: Don't Refine Error-Free Regions



Problem

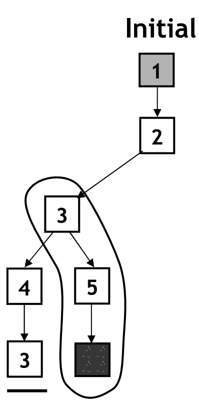
#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Solution

Don't refine error-free regions

Key Idea: Reachability Tree



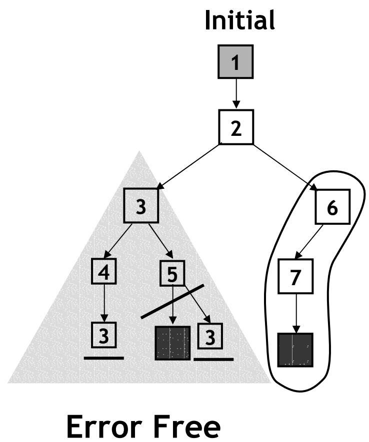
Unroll Abstraction

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



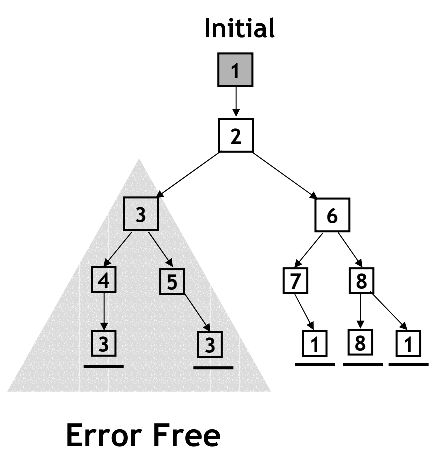
Unroll Abstraction

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.

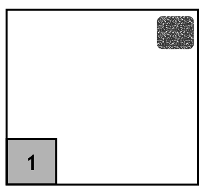
SAFE

S1: Only Abstract Reachable States

S2: Don't refine error-free regions

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 ();
}
```

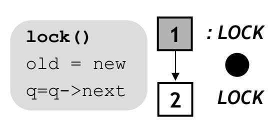


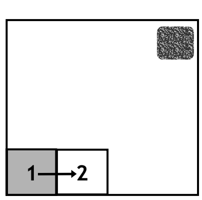
Predicates: *LOCK*

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 ();
}
```



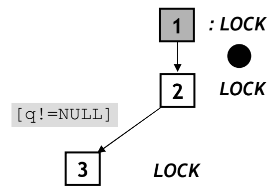


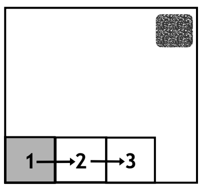
Predicates: *LOCK*

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 ();
}
```



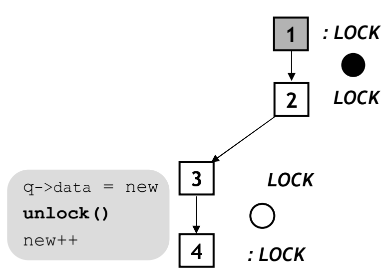


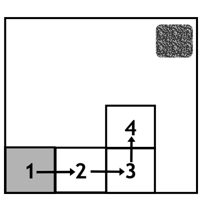
Reachability Tree

Predicates: *LOCK*

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 ();
}
```



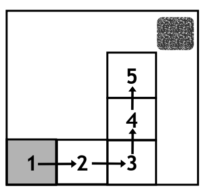


Reachability Tree

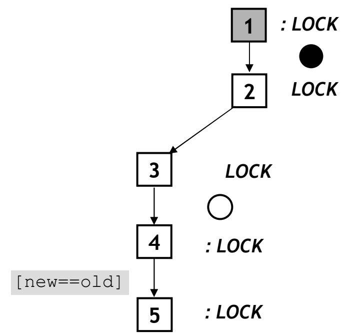
Predicates: *LOCK*

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 ();
}
```



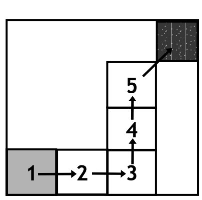
Predicates: *LOCK*



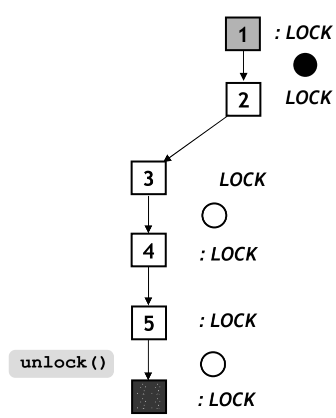
Reachability Tree

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 ();
}
```



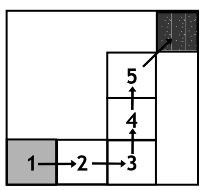
Predicates: *LOCK*



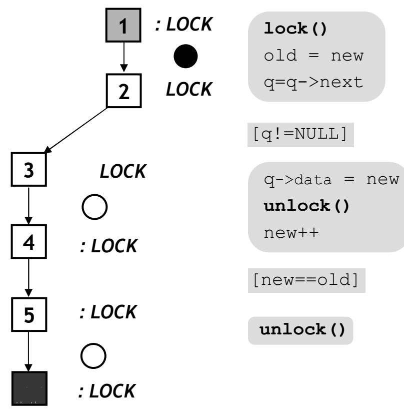
Reachability Tree

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 ();
}
```



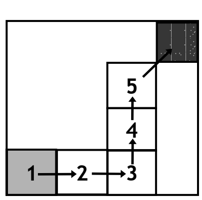
Predicates: *LOCK*



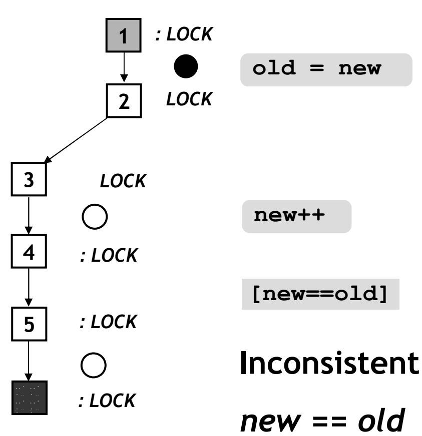
Reachability Tree

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 ();
}
```



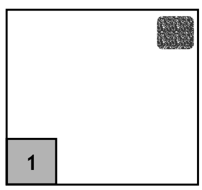
Predicates: *LOCK*



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 ();
}
```

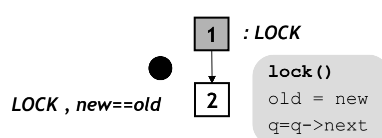


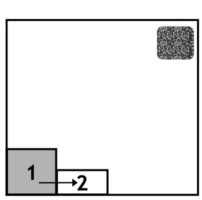
Reachability Tree

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

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 ();
}
```



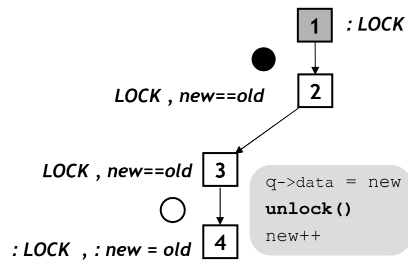


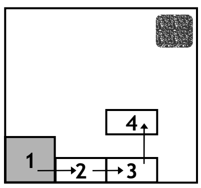
Reachability Tree

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

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 ();
}
```



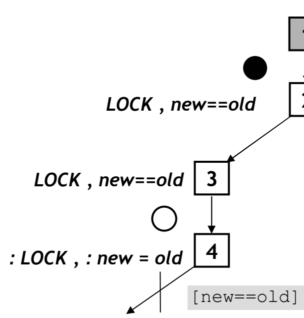


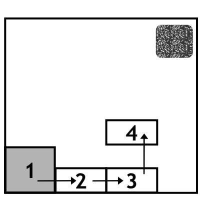
Predicates: *LOCK*, *new==old*

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 ();
}
```



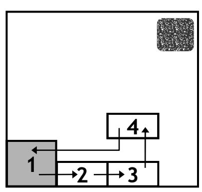


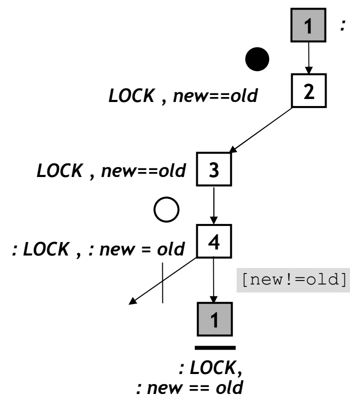
Predicates: *LOCK*, *new==old*

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 ();
}
```



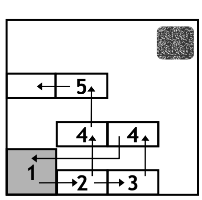


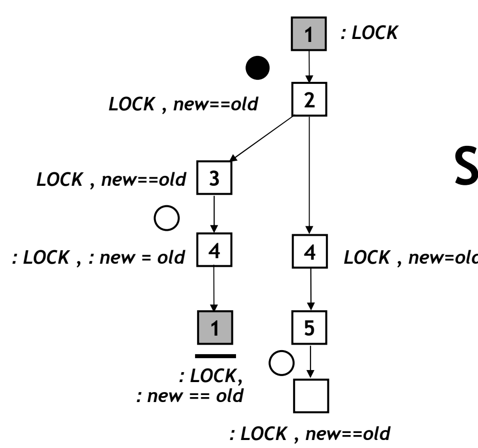
Reachability Tree

Predicates: *LOCK, new==old*

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 ();
}
```



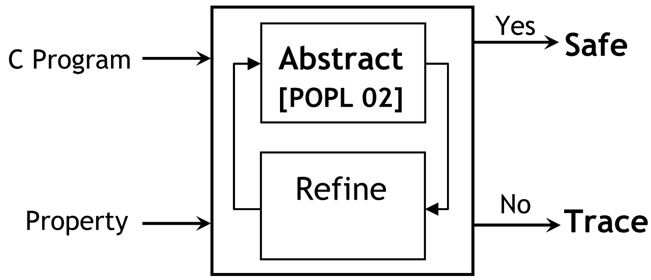


SAFE

Reachability Tree

Predicates: *LOCK, new==old*

Scaling Sequential Verification



Problem: Abstraction is Expensive

Solution: 1. Abstract reachable states,
2. Avoid refining error-free regions

Key Idea: Reachability Tree

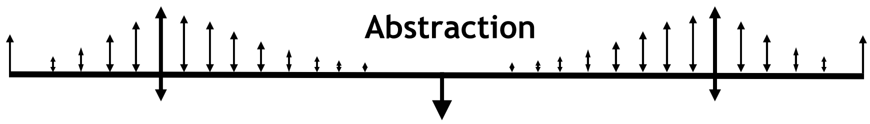
Results

Property3:
IRP Handler
Win NT DDK

Program	***Lines****	***Previous Time(mins)***	***Time (mins)***	***Predicates*** ***Total***	***Average***
`kbfiltr`	12k	1	3	72	6.5
`floppy`	17k	7	25	240	7.7
`diskprf`	14k	5	13	140	10
`cdaudio`	18k	20	23	256	7.8
`parport`	61k	***DNF***	***74***	753	8.1
`parclss`	138k	***DNF***	***77***	382	7.2

* Pre-processed

Analyzing Programs



Programs

Building Blocks	Logic
1. Relevant facts*	Predicates
2. Model	Reach Tree
3. Analysis	Search

* w.r.t. **property** of interest

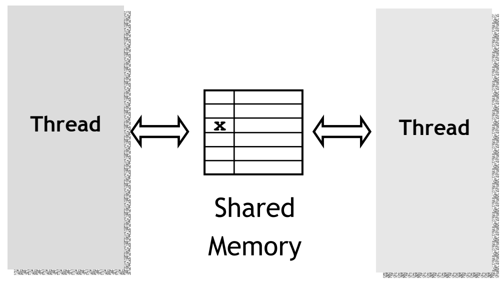
Plan

1. C.G. Abstraction-Refinement

2. Lazy Abstraction
   - Sequential Programs **[POPL 02, POPL 04]**
   - **Multithreaded Programs**

3. Future Work

Multithreaded Programs

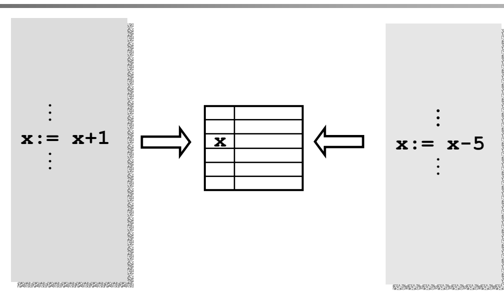


Curse of **Interleaving**

- Non-deterministic scheduling
- Exponentially many behaviors
- Errors are hard to detect, reproduce, eliminate

Testing exercises a tiny fraction of possible behaviours

Data Races

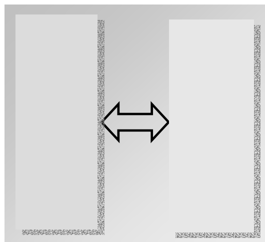


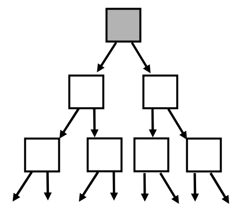
A data race on **x** is a state where:

- **Two** threads can access **x**
- One of the accesses is a **write**

Unpredictable, undesirable program

Brute Force Approach





Model Checking: Explore (abstract) State Space

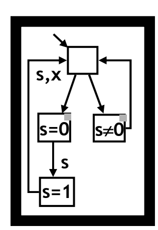
- The **curse** of **Interleavings**
- #Control Combinations = m.n
  - 250,000 if 500 lines/thread, ignoring predicates
  - 3,4,5,…,k threads ? Unbounded threads ?

A Thread-Modular Approach

Key Idea: Summarize each thread

- **Interactions** with others **w.r.t. property**

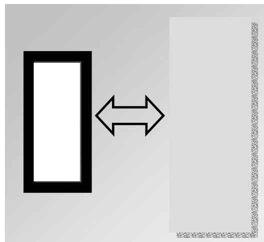
```
while(1){
 atomic{
  old := s;
  if(s==0){
    s :=1;
  }
 }
 :
 if(old==0){
   x++;
   s :=0;}
}
```



[PLDI 04]

Automaton on predicates on **global** variables

A Thread-Modular Approach

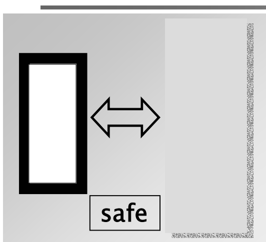


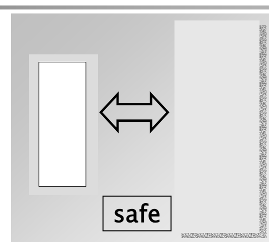
Analysis Time
» Thread £ Summary

Problem: Find Summary which:

1. **[Scalability]** is **small**
2. **[Verification]** has **all behaviors** of thread

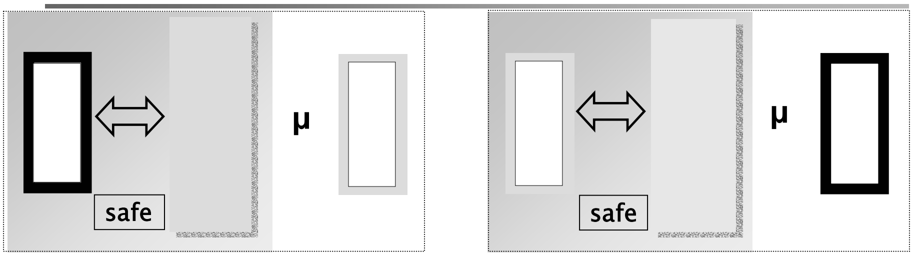
Verify (Thread || Other's Summary)



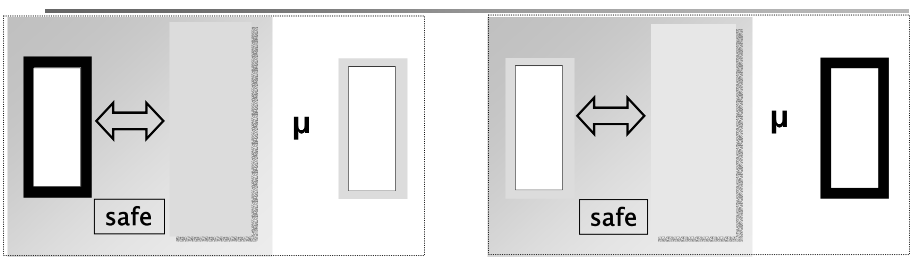


- Control Combinations: Thread £ Summary
  - Small (if summary is small)

Check that Summaries are Valid

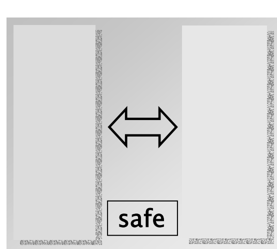


Thread-Modular Verification



Assume-Guarantee

[Owicki-Gries 73]
[Jones 83] [Stark 85]
[Abadi-Lamport 93]
[Alur-Henzinger 96]
[McMillan 97]
[Flanagan-Qadeer 01]



Q: Finding Summaries ?

Data Races in NesC Programs [PLDI 04]

- PL for Networked Embedded Systems [Gay et al. 03]
  - TinyOS Sensor Networks Applications
- **Interrupts** fire **events**, which fire other events or post **tasks** which run asynchronously
- Race-freedom important
  - Non-trivial synchronization idioms
  - Flow-based analysis
- Compiled to C

Case Study: ***sense.nc*** [PLDI 04]

```
atomic{
 old:= state;
 if(state==0){
    state:=1;
 }
}
⋮
if(old==0){
  x++;
     ⋮
```

Interrupt 1 fires
⋮
old := state
if (state ==0){
state := 1
⋮
If (old == 0){
about to write x

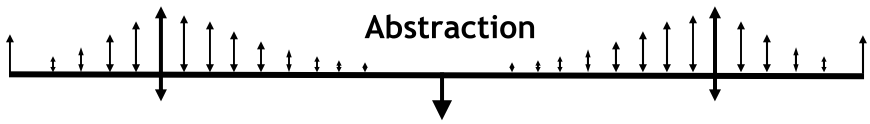
Interrupt 2 fires
⋮
state := 0

Interrupt 1 fires
⋮
old := state
if (state ==0){
state := 1
⋮
If (old == 0){
about to write x

Interrupt 1 handler **disables** interrupt 2

BLAST finds information
- proves no races

Analyzing Programs



Building Blocks	Programs Logic	Multithreaded
1. Relevant facts*	Predicates	Predicates
2. Model	Reach Tree	Summary
3. Analysis	Search	Thread-Modular

* w.r.t. **property** of interest