Static Analysis of Concurrent Programs

Mooly Sagiv

Why Concurrent Programs

- Partial execution order
- More efficient implementations
  - Multicore machines
- Distributed environments

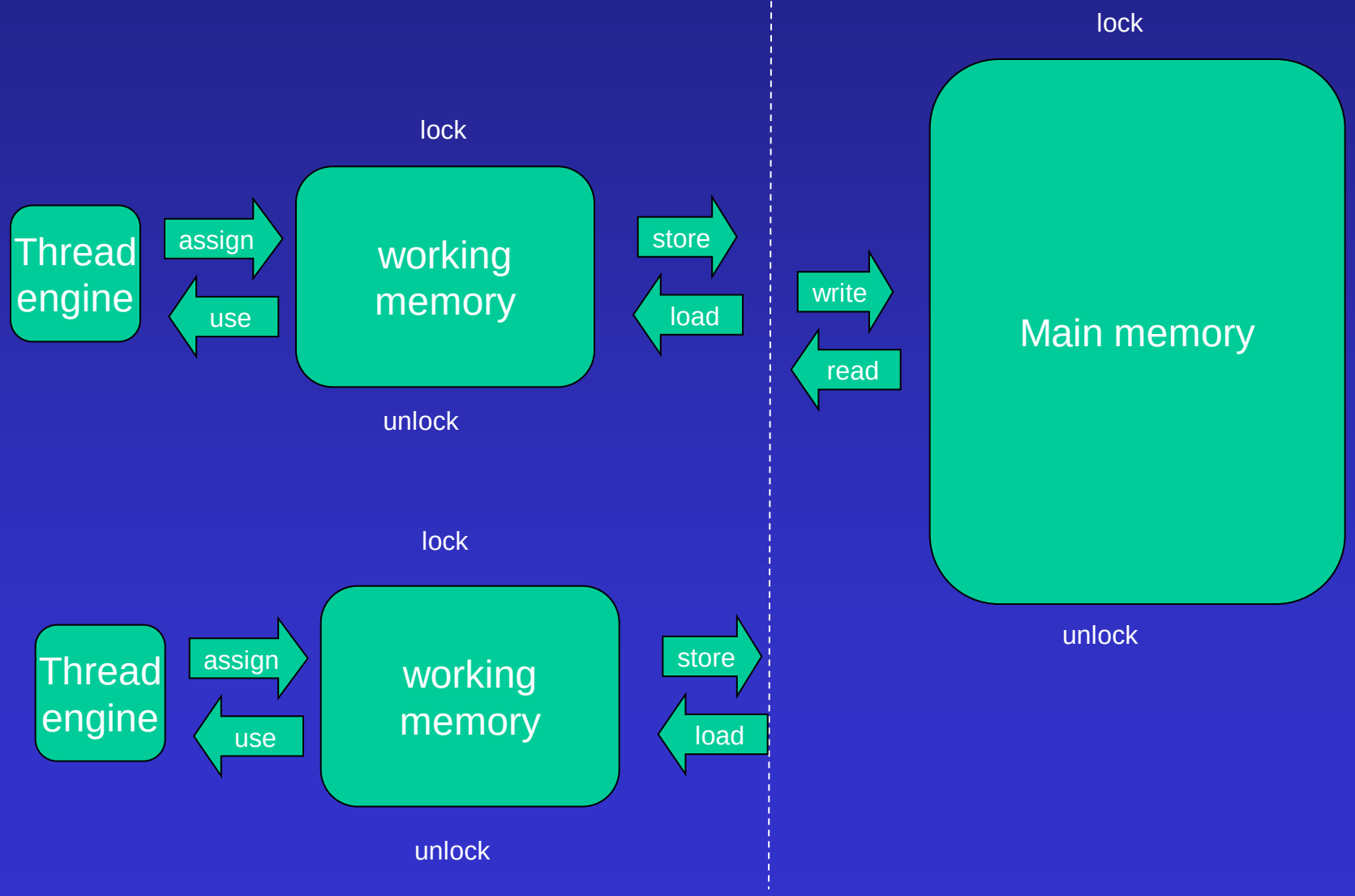
Challenges in Concurrent Programs

- Hard to define the semantics
- Hard to debug
  - Assertion checking /partial does not suffice
  - Many potential executions
  - Testing is very hard
  - Reproducing bugs is hard
- Hard to verify

Java Concurrency

- JVM supports threads of execution
  - Many threads can be executed “simultaneously”
  - Communicate via shared memory
  - Synchronize activities using locks
  - Both threads and locks are objects
- Standard methods for efficient transfer of control
  - wait
  - notify
  - …
- Special rules about allowed orders of executions

Java Memory Model



Java code fragment

$[p.x \mapsto 0, p.y \mapsto 0]$

```
synchronized(p) {
    p.y = 2;
  }
a = p.x;
b = p.y;
c = p.y;
```

```
synchronized(p) {
    p.y = 3;
    p.y = 100;
  }
p.x = 1;
```

Correctness Criteria

- Partial correctness (assertion checking)
  - No runtime errors
  - No memory leaks
- Absence of deadlocks
- Termination
- Absence of dataraces
- Linearizability

Deadlocks

- Happens when two or more competing threads are waiting for the other to finish, and thus neither ever does

```
lock(l1);
lock(l2);
…
unlock(l2);
unlock(l1);
```

```
lock(l2);
lock(l1);
…
unlock(l1);
unlock(l2);
```

Dataraces

- Occurs when two threads access the same resource
  - At least one of the accesses are write
  - No mechanism to prevent order conflicts

A Simple Example

x := y;

y := y +1;

Java code fragment

$[p.x \mapsto 0, p.y \mapsto 0]$

```
synchronized(p) {
    p.y = 2;
  }
a = p.x;
b = p.y;
c = p.y;
```

```
synchronized(p) {
    p.y = 3;
    p.y = 100;
  }
p.x = 1;
```

No Datarace

```
lock(l);
v := v+1;
unlock(l);
```

```
lock(l);
v := v +1;
unlock(l);
```

Concurrency and Alias Analysis

Different
locks

sync(e1) {

e2.f =

}

sync(e3) {

… = e4.f

}

Same Object

Programs and Properties

- Concurrent programs
- Unbounded number of threads
  - parametric systems
- Unbounded number of objects
- Pointers and destructive updates
- Memory safety
  - Absence of null dereferences
  - Absence of memory leaks
- Preservation of data structure invariants
- Linearizability
- User-specified invariants

A Generic Framework for Verifying Concurrent Programs

Eran Yahav

POPL’01

Main Ideas

- An interleaving semantics for Java
  - A program configuration encodes
    - global store
    - program-location of every thread
    - status of locks and threads
- First-order logical structures used to represent program configurations
- Use TVLA to obtain a static analyzer
- Works well for tiny programs

Configurations

- Predicates model properties of interest
  - $eq(v_1,v_2)$
  - is_T(v)
  - { at[lab](t) : lab $\in$ Labels }
  - { rv[fld](o_1,o_2) : fld $\in$ Fields }
  - heldBy(l,t), blocked(t,l), waiting(t,l)
- Can use the framework with different predicates

Structural Operational Semantics

- An action consists of:
  - precondition formula
  - update formulae
- Precondition formula may use a free variable t_s for "currently scheduled" thread
- Semantics is non-deterministic

Structural Operational Semantics - actions

lock(v)	
precondition	$\neg\exists t \neq t_s$: rval[v](t_s,l) $\wedge$ held_by(l,t)
predicate update	held_by'(l_1,t_1) = held_by(l_1,t_1) $\vee$ (l_1 = l $\wedge$ t_1 = t_s) blocked' (t_1,l_1) = blocked(t_1,l_1) $\wedge$ (($l_1 \neq$ l) $\vee$ ($t_1 \neq t_s$))

Safety Properties

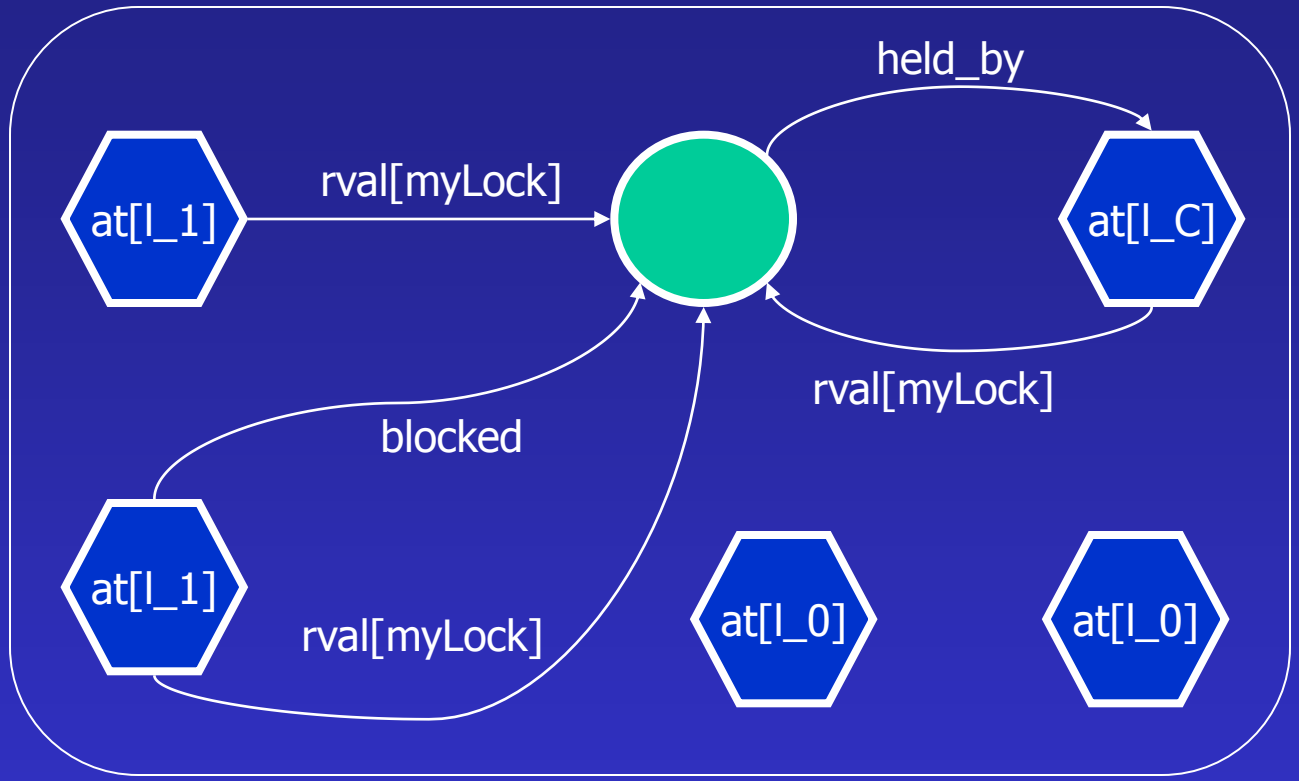
- Configuration-local property as logical formula
- Example: mutual exclusion

$$\forall t_1,t_2: (t_1 \neq t_2) \rightarrow \neg(at[l_{crit}](t_1) \wedge at[l_{crit}](t_2))$$

- Example: no total deadlock

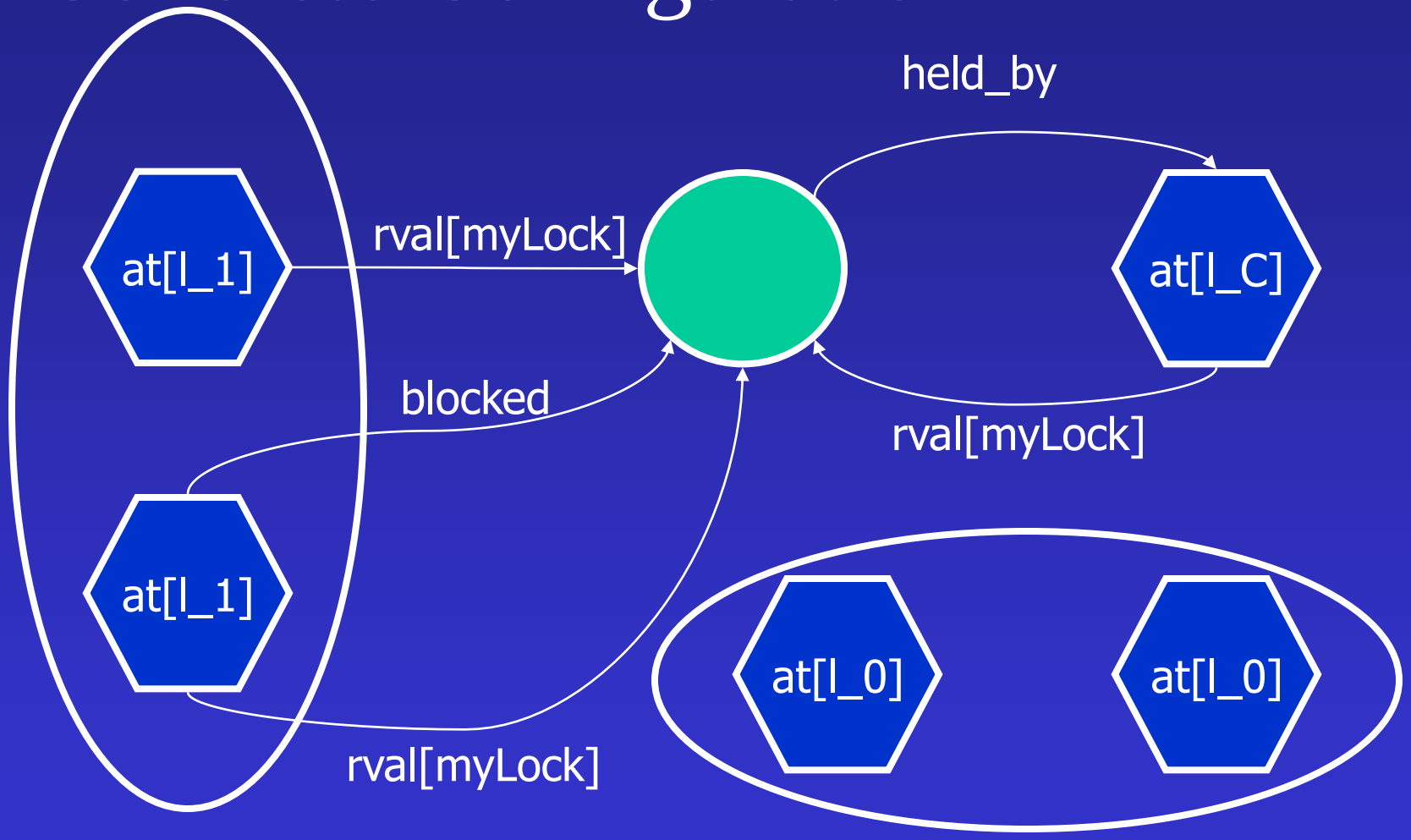
$$\exists t, \forall l_b : is\_thread(t) \wedge \neg blocked(t, l_b)$$

Configurations – Example

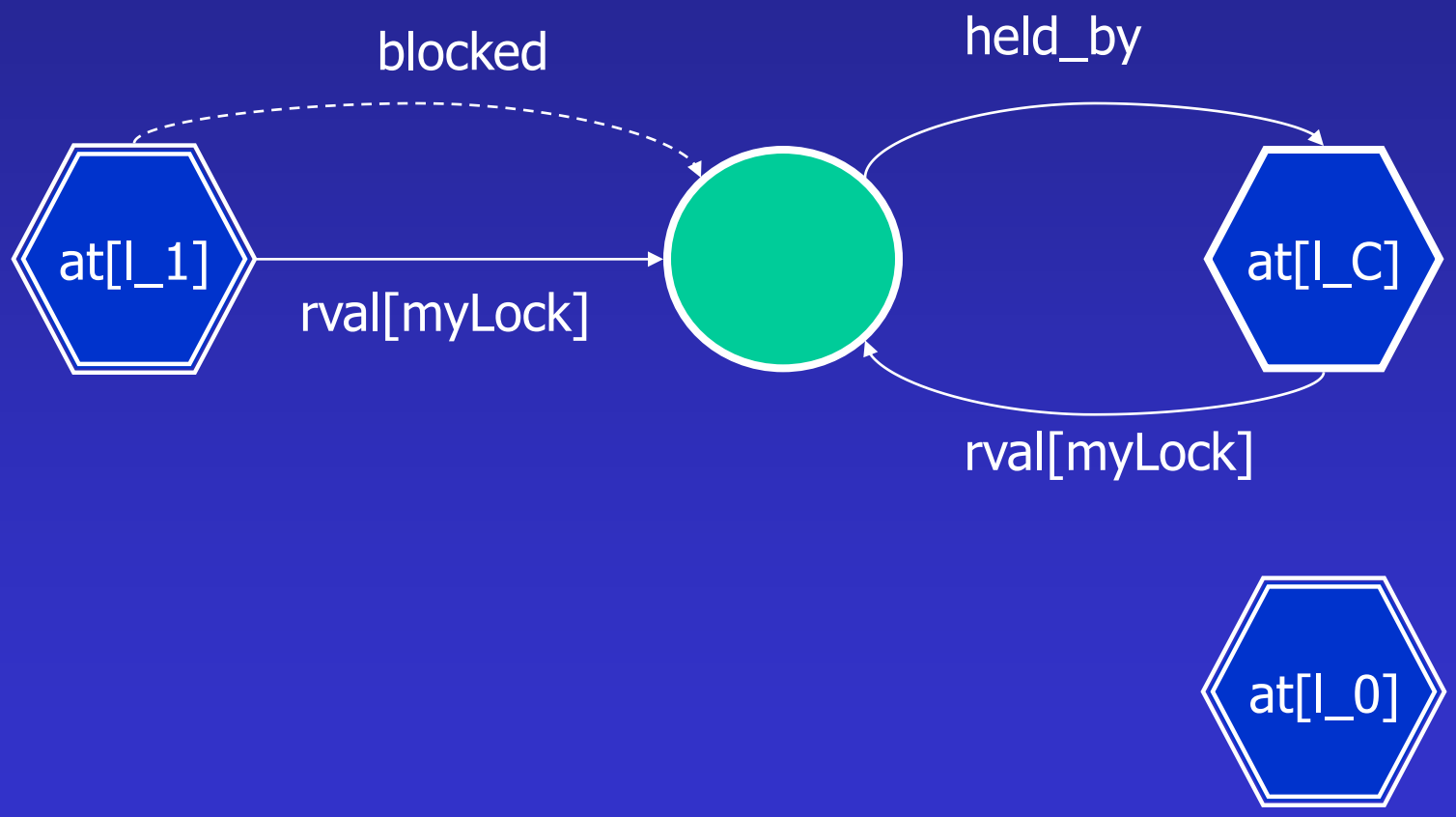



```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }
```

Concrete Configuration

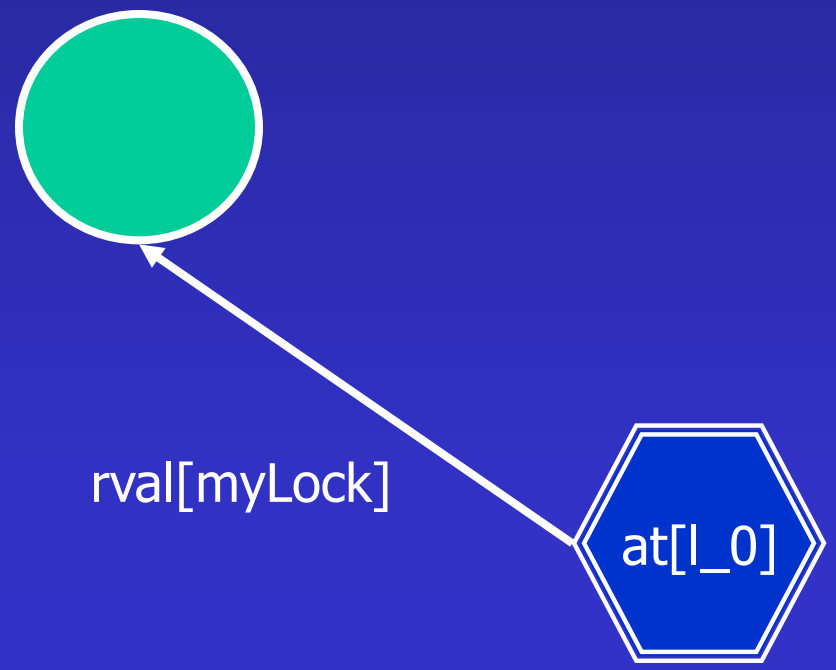


Abstract Configuration



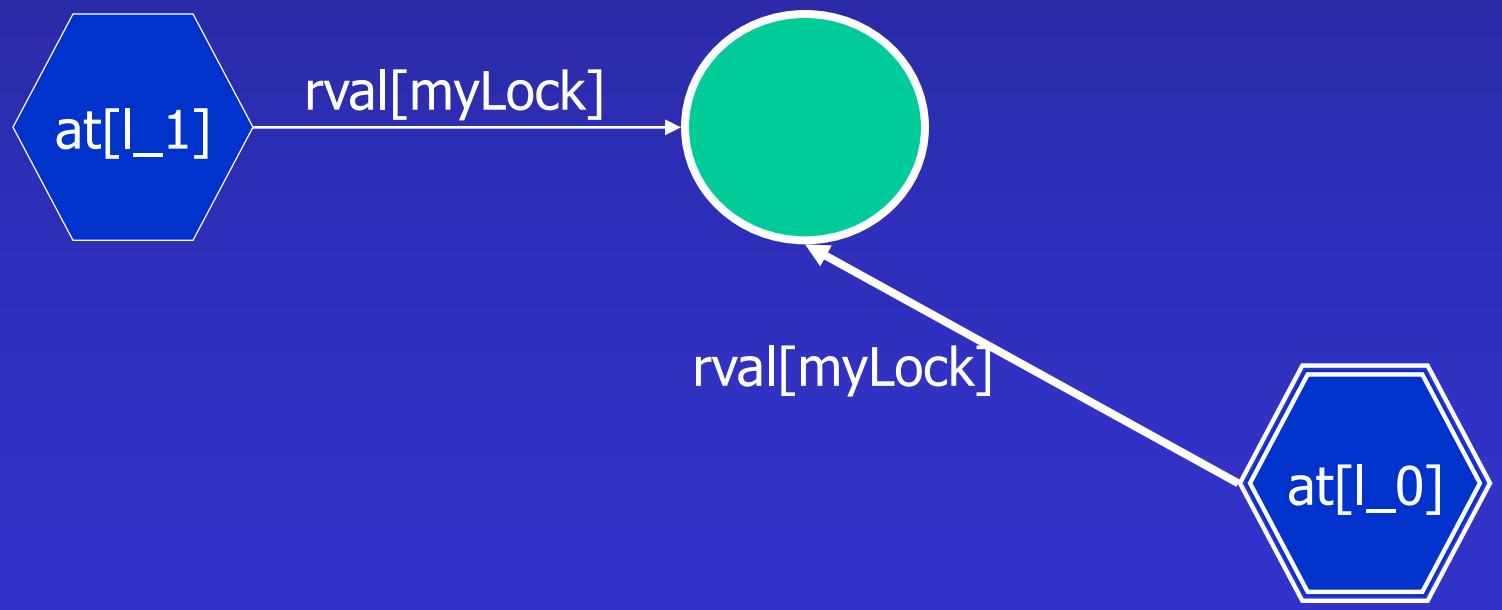
Abstract Interpretation

```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }
```



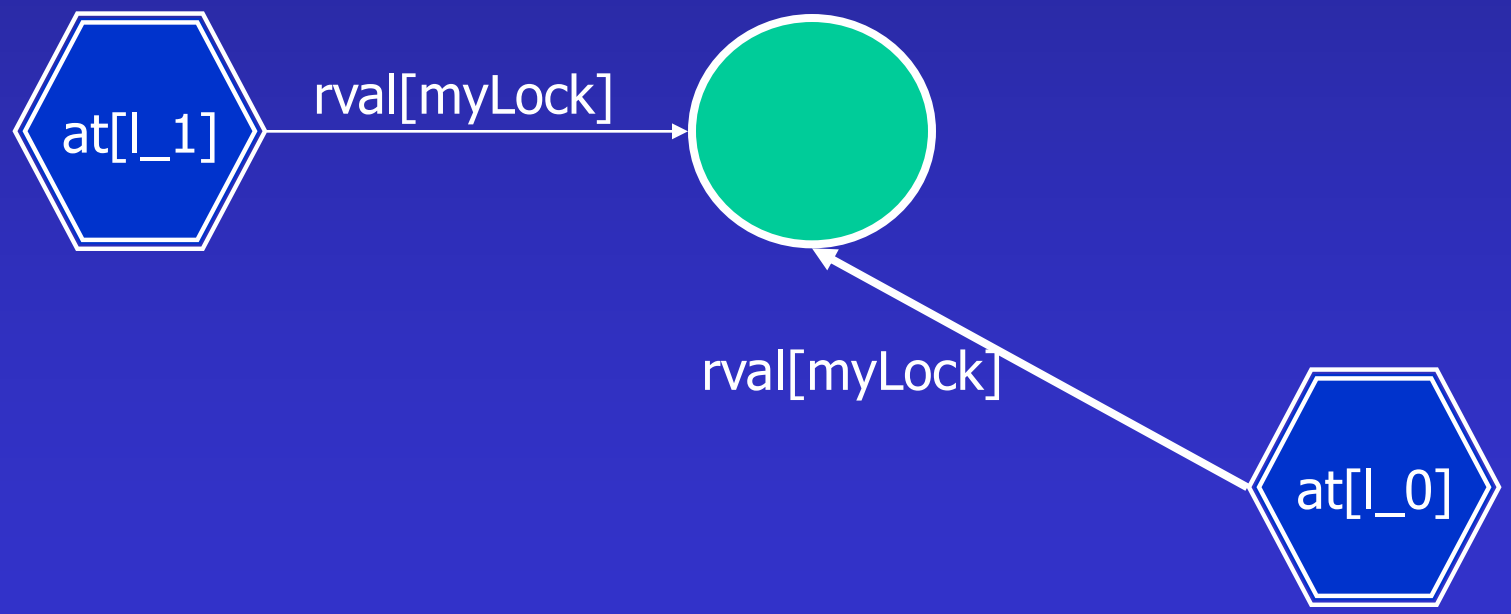
Abstract Interpretation

```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }
```



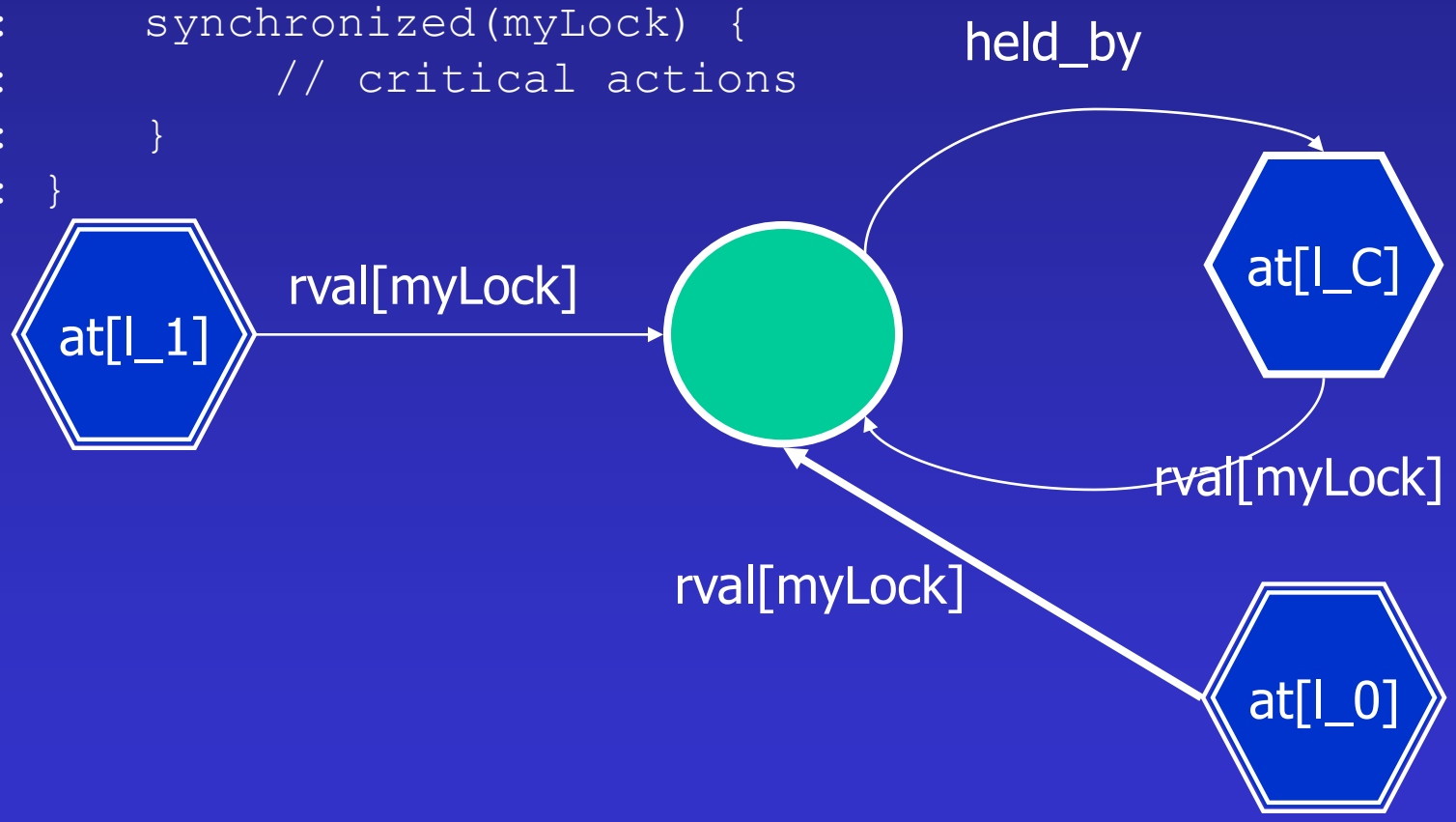
Abstract Interpretation

```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }
```



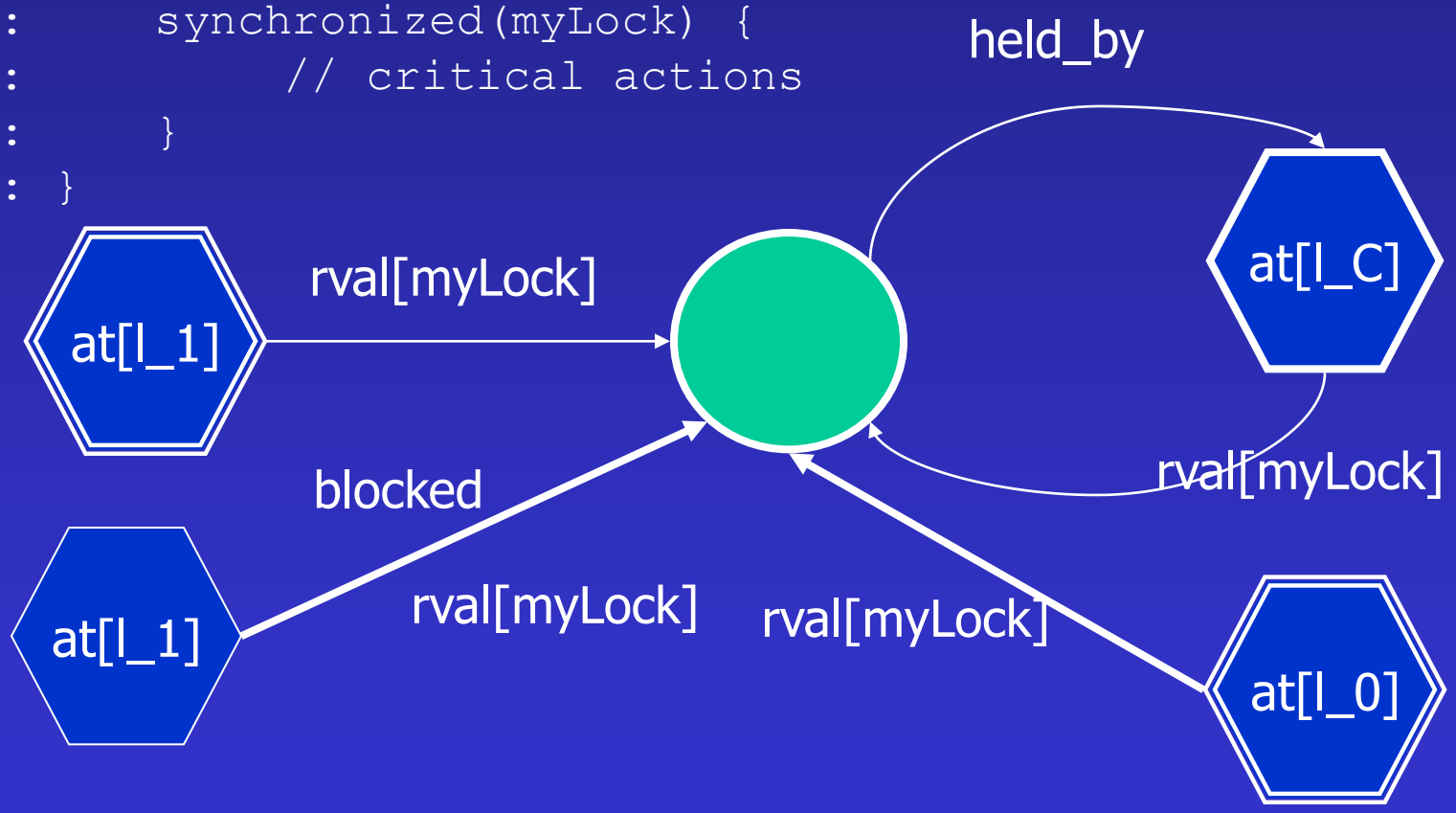
Abstract Interpretation

```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }
```

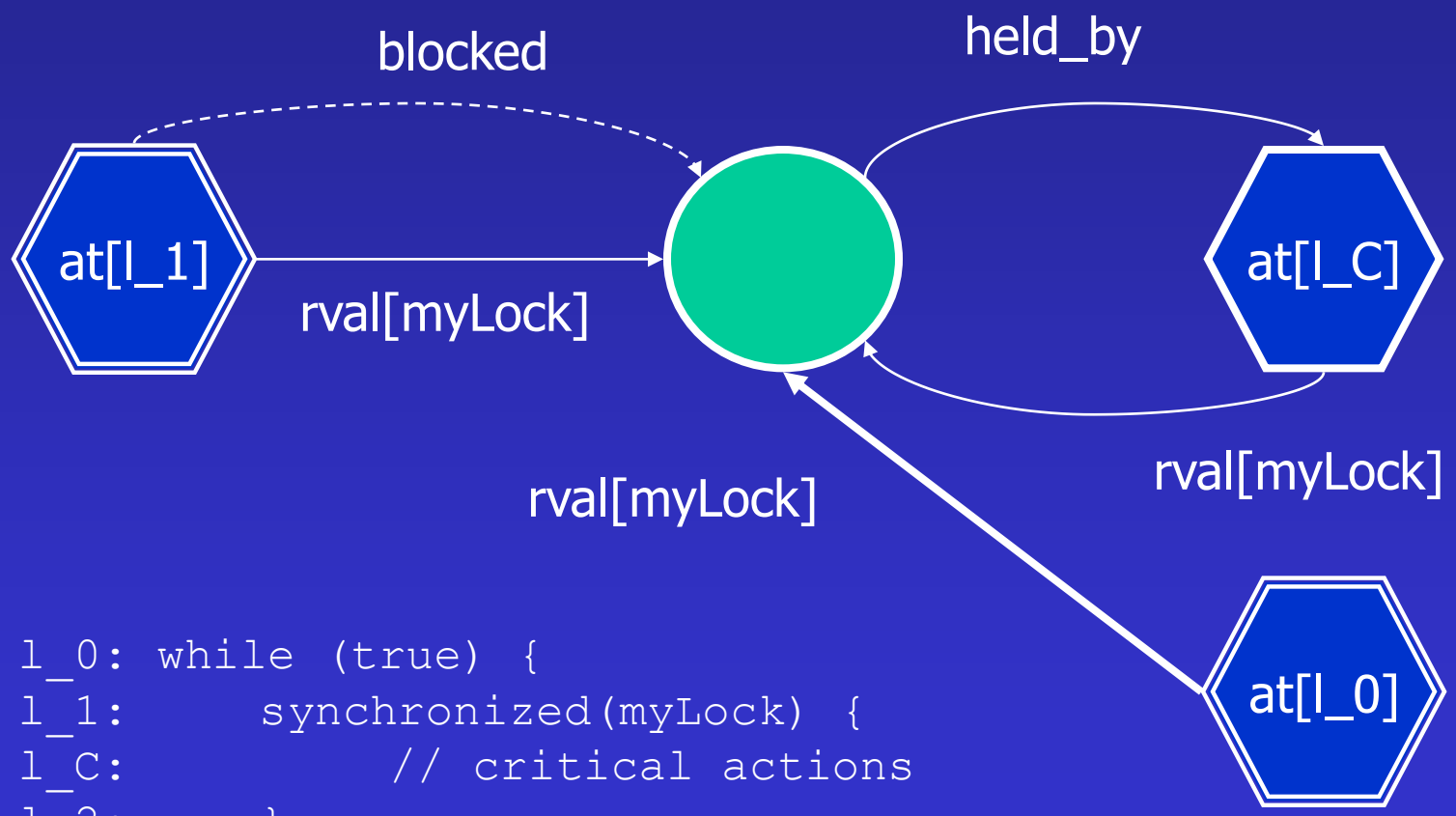


Abstract Interpretation

```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }
```



Abstract Interpretation




```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }
```

Examples Verified

Program	Property
twoLock Q	No interference No memory leaks Partial correctness
Producer/consumer	No interference No memory leaks
Apprentice Challenge	Counter increasing
Dining philosophers with resource ordering	Absence of deadlock
Mutex	Mutual exclusion
Web Server	No interference

Scalability

- The number of abstract configurations is doubly exponential in the program size
- Interleaving semantics quickly leads to state explosion

Techniques for Scalability

- Courser abstractions
  - Refinement
- Program restrictions
- Limited semantics
- Specialization
- Unsoundness

Flow Insensitive Analysis

- Ignore control flow statements
- Obtain a sound solution w.r.t. all interleavings
- But can it be precise enough?
  - Initializations
  - Strong updates
  - Object sensitivity
  - Context sensitivity

Critical Section Example

```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }
```

Thread Modular Shape Analysis

Alexey Gotsman

PLDI’07

Microsoft Development Environment [design] - kbdclass.c [Read Only]

File Edit View Debug Tools Window Help

kbdclass.c

```
987   {
988       PIRP irp;
989       LIST_ENTRY listHead, *entry;
990       KIRQL irql;
991
992       InitializeListHead(&listHead);
993
994       KeAcquireSpinLock(&DeviceExtension->SpinLock, &irql);
995
996       do {
997           irp = KeyboardClassDequeueReadP
998           if (irp) {
999               irp->IoStatus.Status = STAT
1000              irp->IoStatus.Information =
1001
1002              InsertTailList (&listHead, &irp->Tail.Overlay.ListEntry);
1003          }
1004      } while (irp != NULL);
1005
1006      KeReleaseSpinLock(&DeviceExtension->SpinLock, irql);
1007
1008      //
1009      // Complete these irps outside of the spin lock
1010      //
1011      while (! IsListEmpty (&listHead)) {
1012          entry = RemoveHeadList (&listHead);
1013          irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);
```

Is this a well-formed cyclic doubly-linked list?

Ready Ln 971 Col 2 Ch 2 INS

Main Ideas

- Compile-time names for threads and locks
- Static association of the locks and parts of the heap
- Analyze the behavior of every thread separately with a resource invariant describing the state of others
  - Break the heap into two parts
- Overapproximate local and resource invariants
- Prove the absence of dataraces

A motivating example(unrealistic)

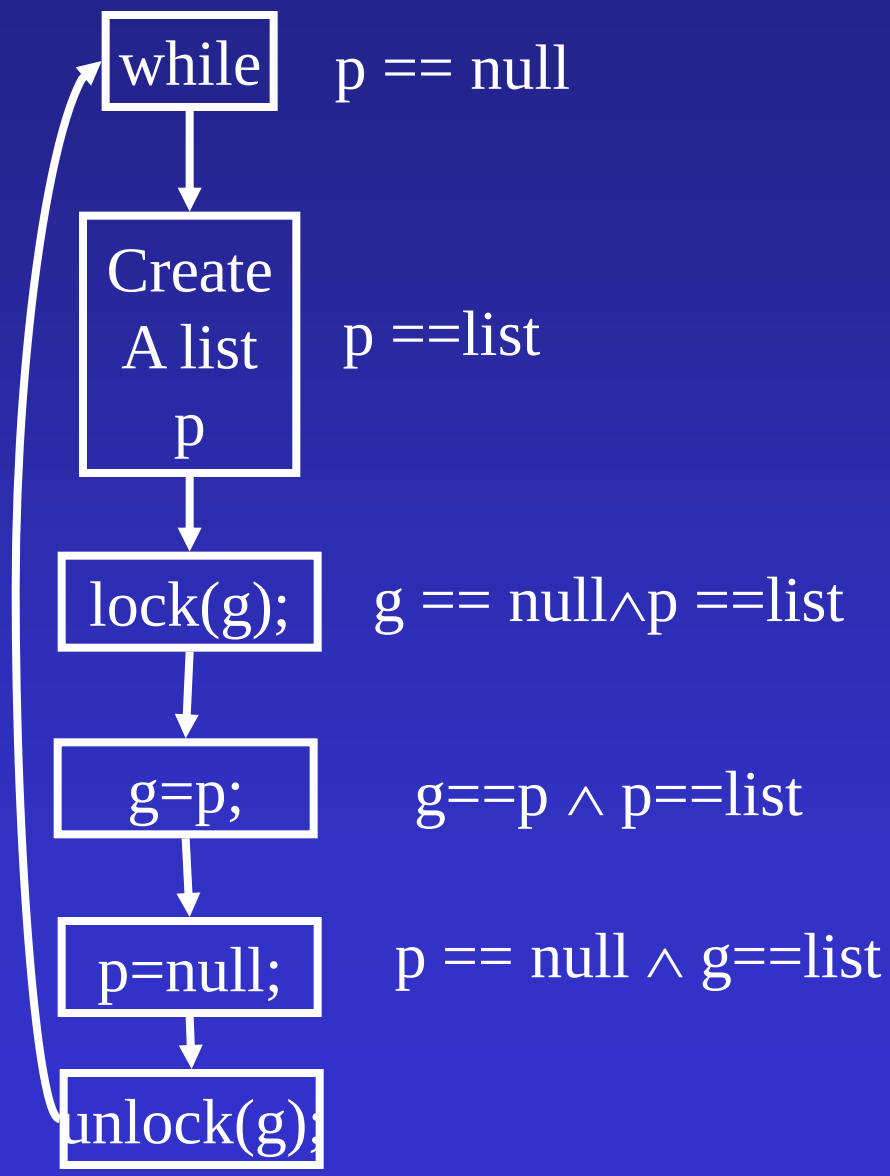
```
class List {
   public  List n;
    public int d;
            }
List g;
```

```
void producer(){
    List p;
    while (1) {
      …
     create a list in p
      synchronized (g} {
            g = p;
            p = null ;
      }
}
```

```
void consumer(){
    List c;
    while (1) {
      synchronized (g}    {
            c = g;
            g = null ;
  }
   …
  consume c;
}
```

A motivating example(producer)

g == null

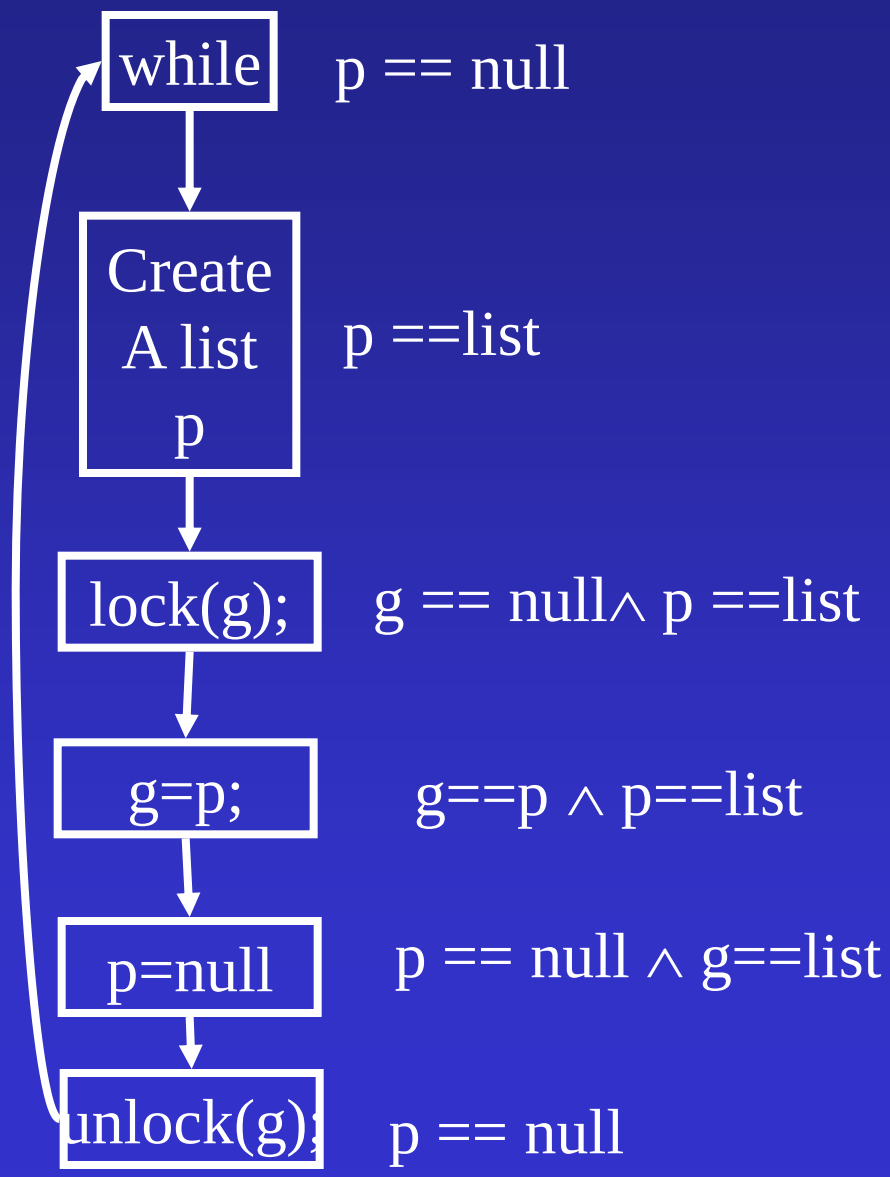


A motivating example(producer)

g == null

$\vee$

g==list

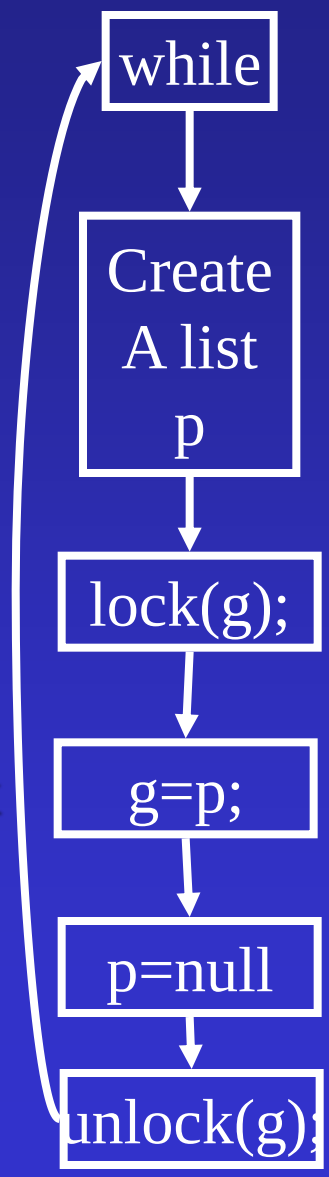


A motivating example(producer)

g == null

$\vee$

g==list



p == null

p ==list

(g == null $\vee$ g==list) $\wedge$ p ==list

g==p $\wedge$ p==list

p == null $\wedge$ g==list

p == null

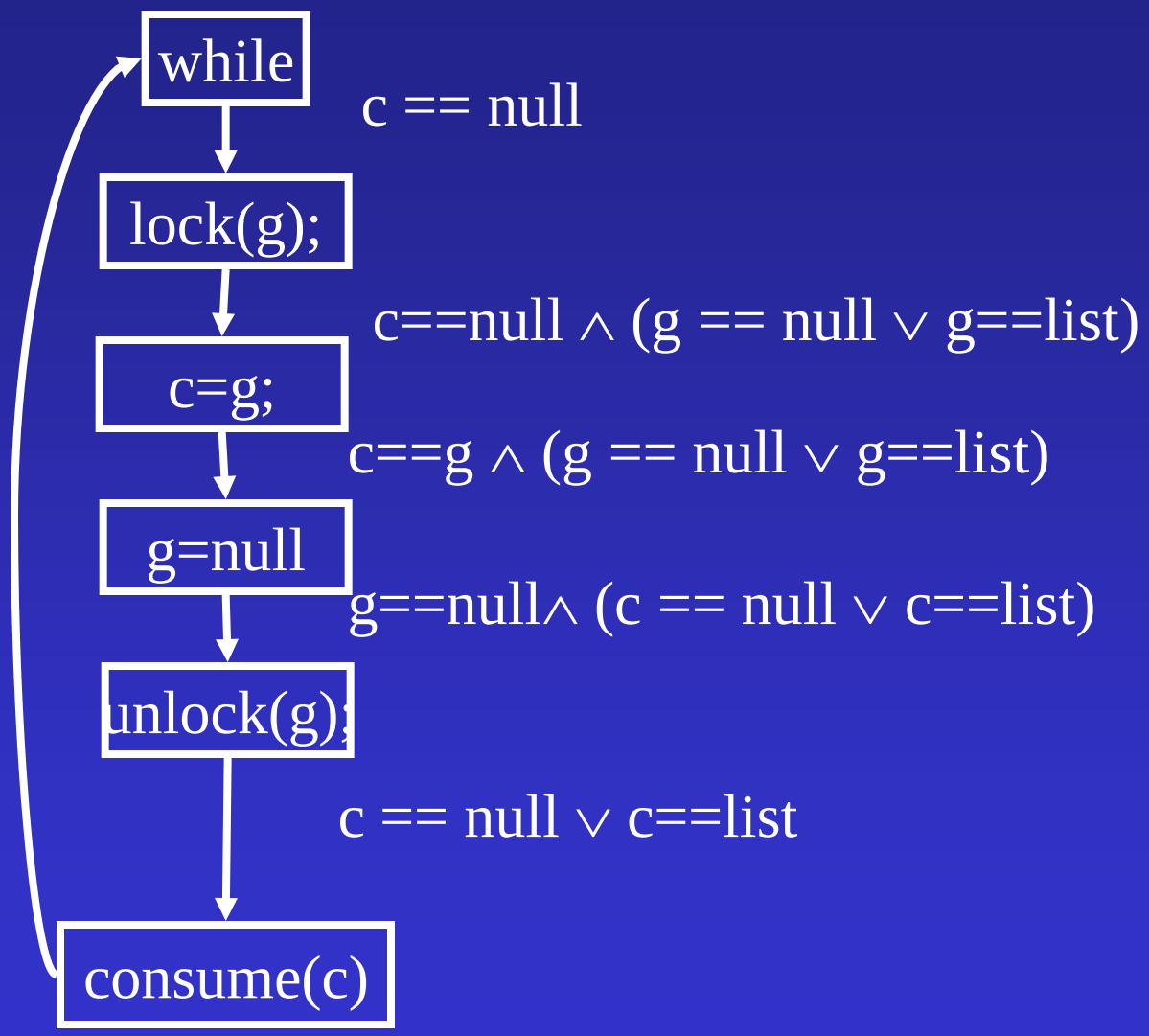
- Potential memory leak

A motivating example(consumer)

g == null

$\vee$

g==list



A Separation Domain

- $<D, \sqsubseteq, \sqcup, \bot, \top, e, * >$
- $<D, \sqsubseteq, \sqcup, \bot, \top>$ is the usual lattice
- $*: D \times D \rightarrow D$
- $<D, \sqsubseteq, e, * >$ is a partially ordered commutative monoid
  - * is associative and commutative
  - * has the unit element e
  - * is monotone

Locality

- A separated domain $<D, \sqsubseteq, \sqcup, \perp, \top, e, * >$
- A function $f: D \rightarrow D$ is local if for all u,v: $f(u * v) \sqsubseteq f(u) * v$

Relating Concrete and Abstract Separated Domains

- A separated concrete domain $<C, \sqsubseteq_C, \sqcup_C, \bot_C, \top_C, e_C, *_C>$
- A separated abstract domain $<A, \sqsubseteq_A, \sqcup_A, \bot_A, \top_A, e_A, *_A>$
- A concretization $\gamma: A \rightarrow C$ such that
  - γ is monotone
  - Abstract transformers over-approximate concrete ones
  - γ is a homomorphism between abstract and concrete monoids
    - $\forall a, b: \gamma(a *_A b) = \gamma(a) *_C \gamma(b)$

Decomposition Based Shape Analysis

Roman Manevich

Thread-Modular Analysis

- Abstract away the correlations between local states of different threads
  - No correlations between program counters
  - Cartesian Abstraction
- Information maintained
  - Correlations between the local state and global state of each thread
- "The quadratic cost of computing transformers can be greatly reduced…" [Flanagan & Qadeer SPIN, 2003]
- Naturally handles unbounded number of threads

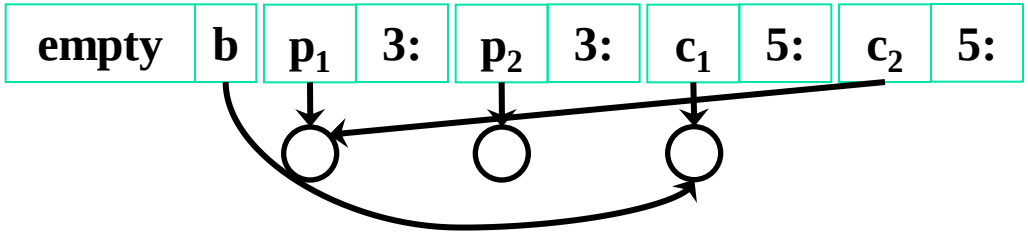
A Singleton Buffer

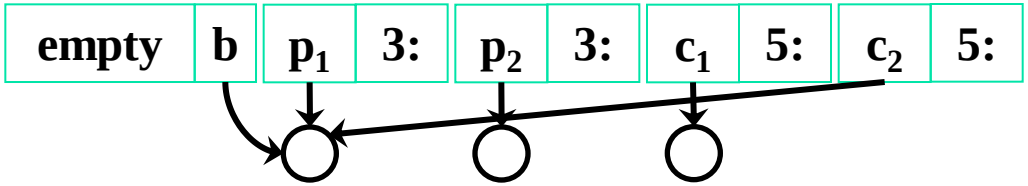
```
Boolean empty = true;
Object b = null;

produce() {
1: Object p = new();
2: await (empty) then {
        b = p;
        empty = false;
    }
3:
}

consume() {
Object c;
4: await (!empty) then {
        c = b;   b=null;
        empty = true;
    }
5: use(c);
6: dispose(c);
7:
}
```

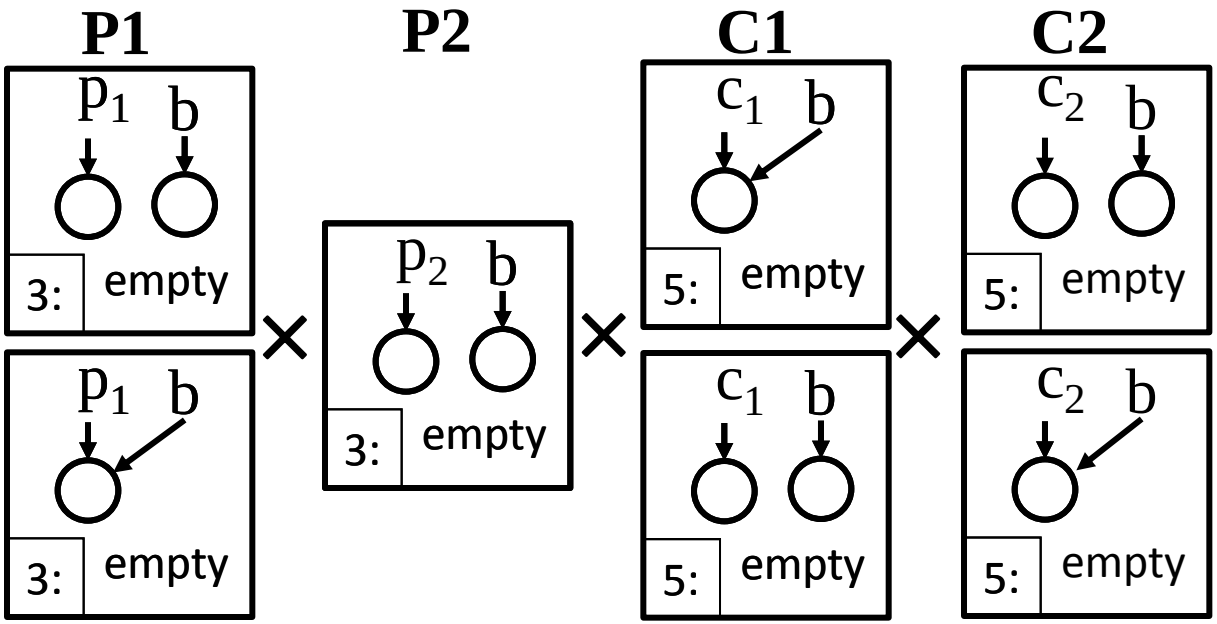
Safe Dereference

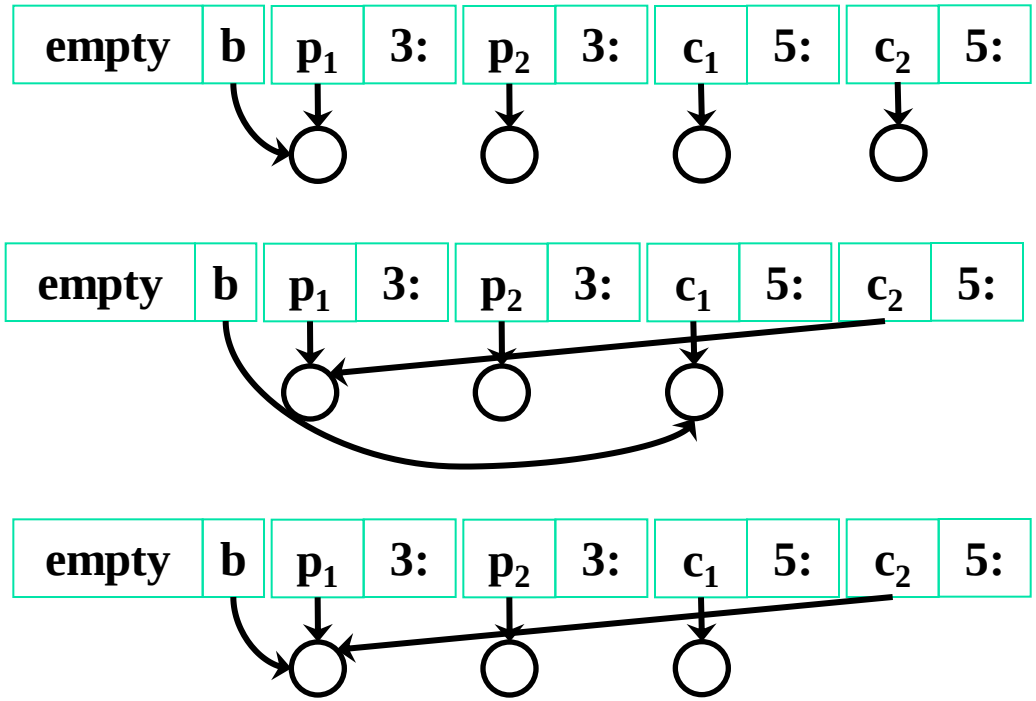
No Double free

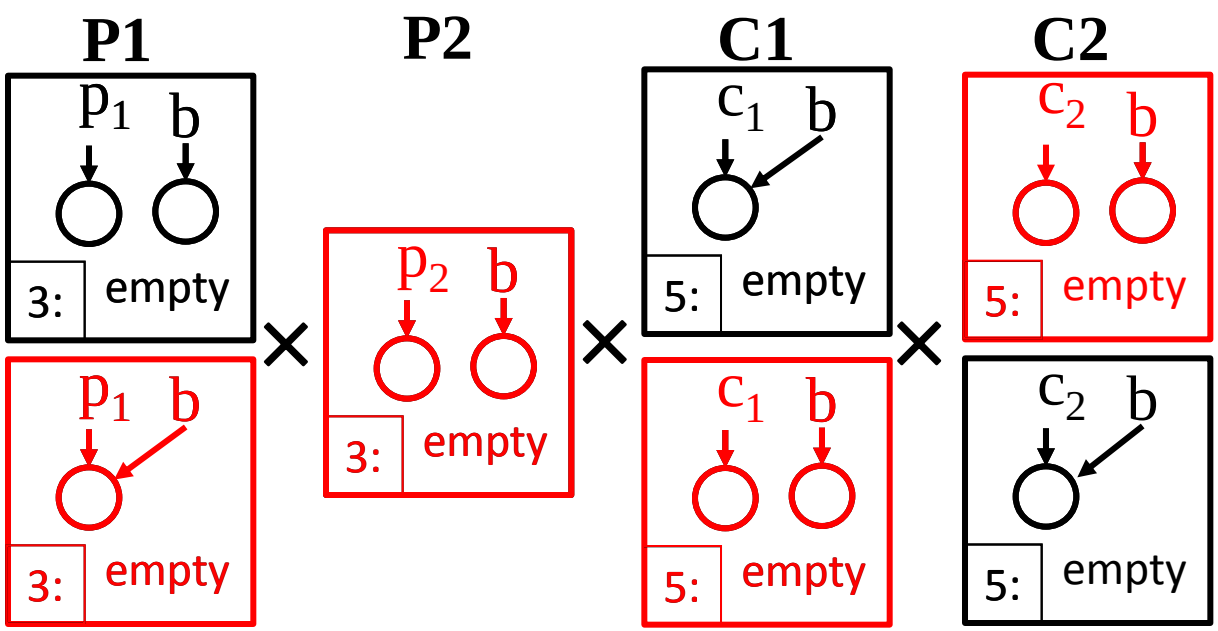
Thread-Modular Abstraction



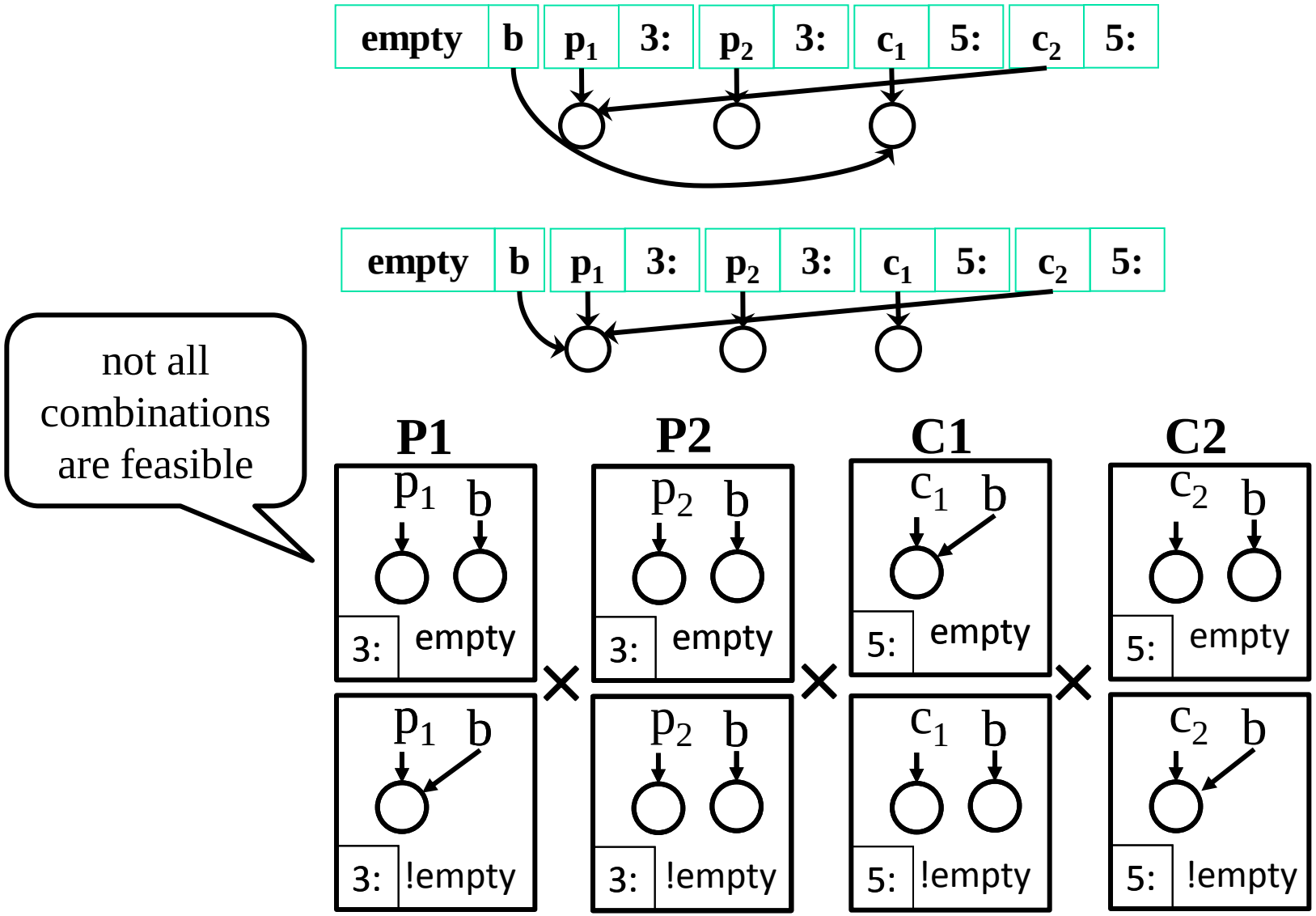




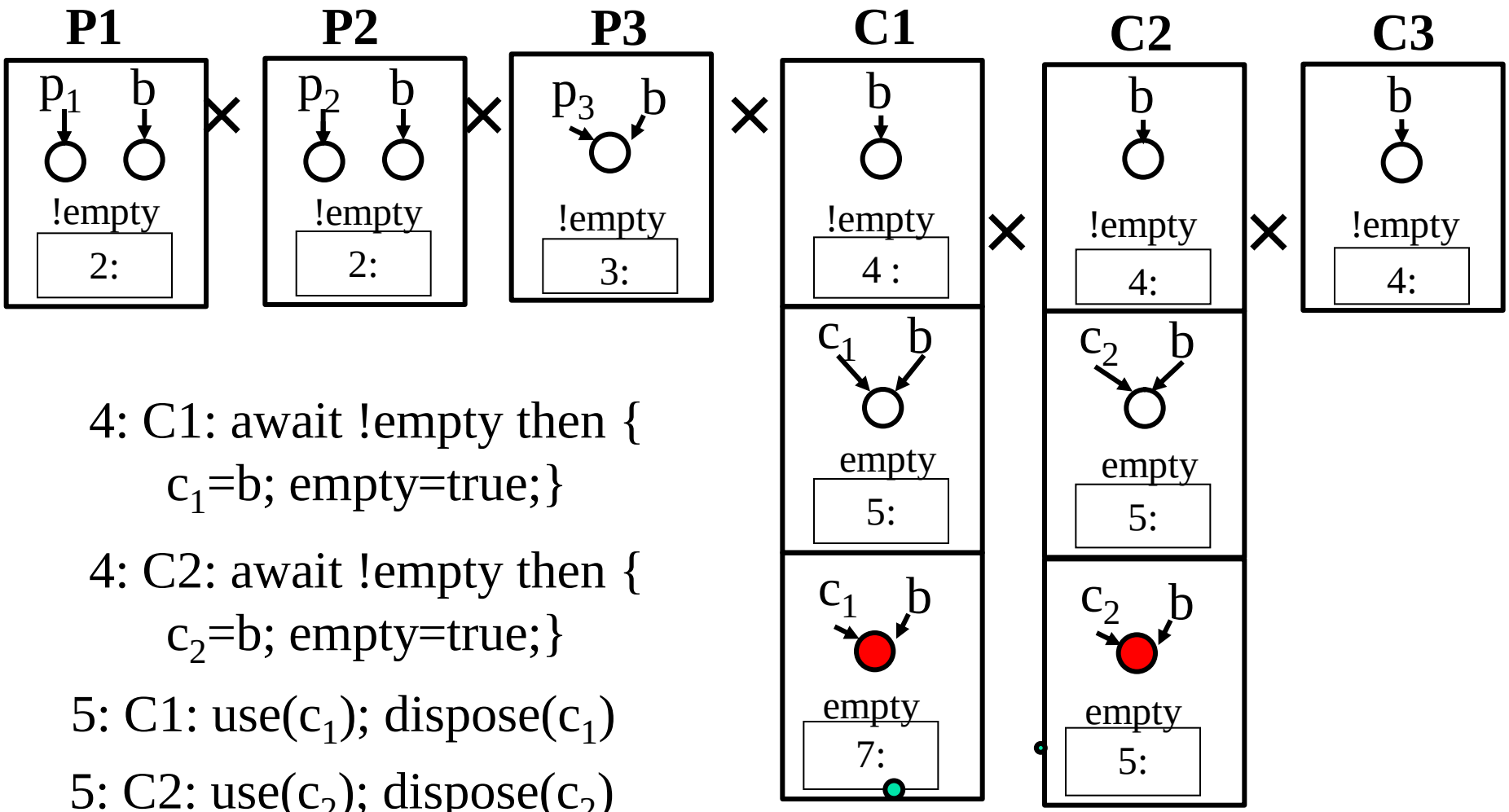

empty
b
p1
3:
p2
3:
c1
5:
c2
5:


P1
P2
C1
C2
p1
b
3:
empty
p2
c1
5:
c2

Thread-Modular Abstraction



Partial Abstract Interpretation



4: C1: await !empty then {
c_1=b; empty=true;}

4: C2: await !empty then {
c_2=b; empty=true;}

5: C1: use(c_1); dispose(c_1)

5: C2: use(c_2); dispose(c_2)

Potential Double Free!!!

A Singleton Buffer

```
Boolean empty = true;
Object b = null;

produce() {
1: Object p = new();
2: await (empty) then {
        b = p;
        empty = false;
    }
3:
}

consume() {
Object c;
4: await (!empty) then {
        c = b;   b=null;
        empty = true;
    }
5: use(c);
6: dispose(c);
7:
}
```

Safe Dereference

No Double free

Thread-Modular Analysis

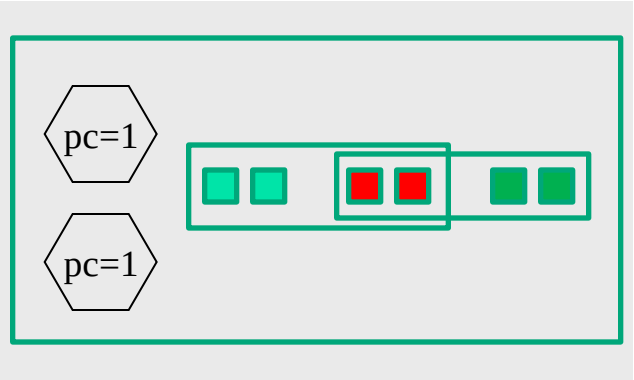
- Abstract away the correlations between local states of different threads
  - No correlations between program counters
  - Cartesian Abstraction
- Information maintained
  - Correlations between the local state of each thread and the globals
- Scales with the number of threads
- Handles unbounded number of threads
- But limited precision

Increasing Precision

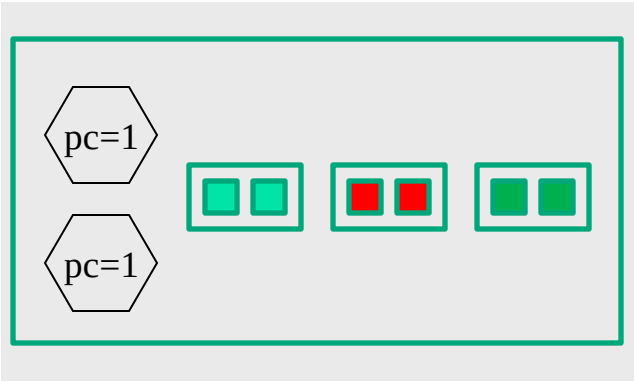
- Enforce program restrictions
  - Limited aliasing
  - Ownership relations
  - Limited concurrency
- Enhanced analysis
  - Global instrumentation
  - Separation Domains [Gotsman et. al. PLDI'07]
  - Semi-Thread Modular Analysis [Segalov et. al., TR, Berdine et. al. CAV'08]

Thread-Modular Analysis

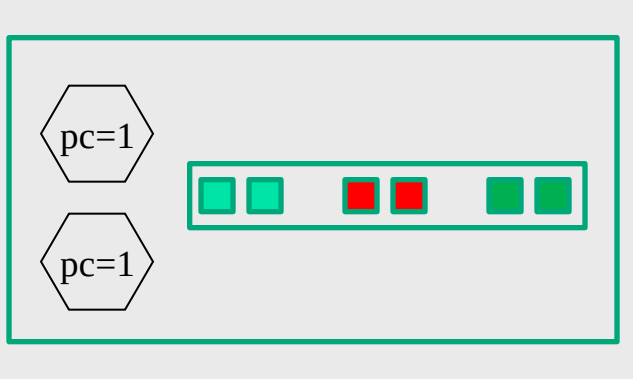
Non-disjoint resource invariants
[Roman's work]
Fine-grained concurrency



Separated resource invariants
[Gotsman et al., PLDI 07]
Coarse-grained concurrency



Single global resource invariant
[Flanagan & Qadeer, SPIN 03]



Thread Quantification for Concurrent Shape Analysis

J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, M. Sagiv

CAV'08

Semi-Thread-Modular Analysis

M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, M. Sagiv

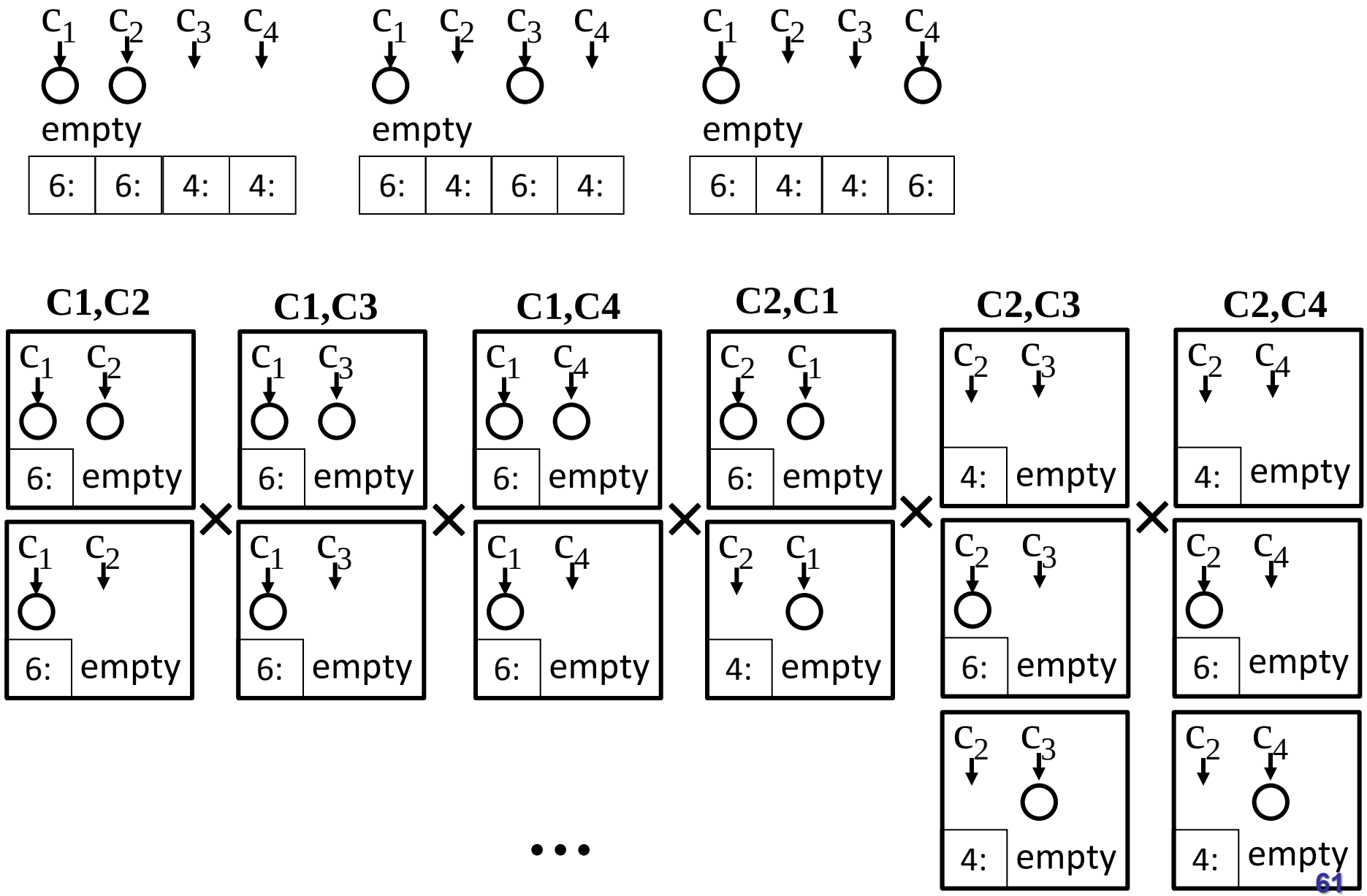
Main Results

- A refinement of thread-modular analysis
  - Not fully modular
- Precise enough to prove properties of fine-grained concurrent programs
  - Were not automatically proved before
- Two effective methods for efficiently computing transformers
  - Summarizing Effects
  - Summarizing Abstraction
  - On a concurrent set imp. speedup is x34!

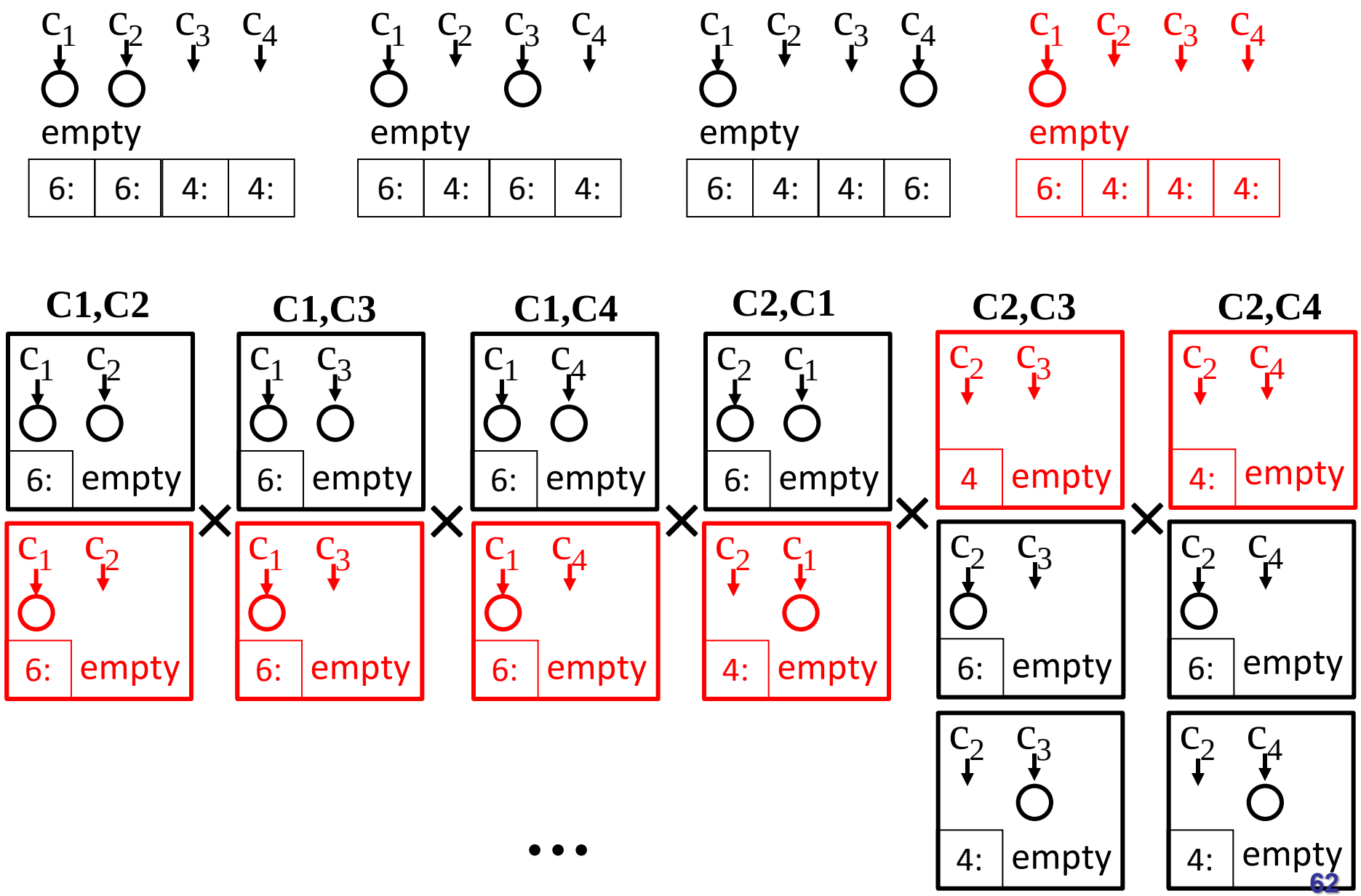
Semi-Thread-Modular Analysis

- Abstract away correlations between local states of more than two threads
- Information maintained
  - Correlations between the local state of each thread and the global state
  - May-correlations between local states of every pair of threads
    - Not necessarily symmetric

Semi-Thread-Modular Abstraction



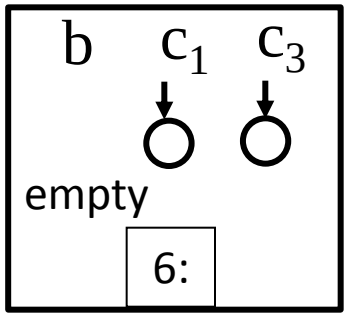
Semi-Thread-Modular Concretization



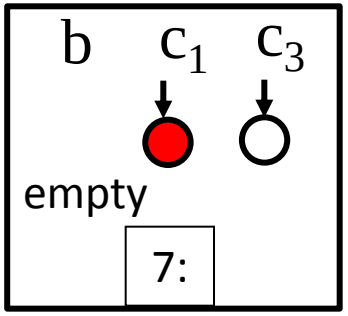
Worst-Case Complexity

- Full state analysis
  - Shared state – G, Local state – L_{tid}
  - State space = $\wp(G \times L_1 \times \ldots \times L_n)$
  - #states: $O(|G| \cdot |L|^n)$
- Thread-modular analysis
  - State space = $\wp(G \times L_1) \times \ldots \times \wp(G \times L_n)$
  - #states: $O(n \cdot |G| \cdot |L|)$
- Semi-thread-modular analysis
  - State space = $\wp(G \times L_1 \times L_2) \times \ldots \times \wp(G \times L_{n-1} \times L_n)$
  - #states: $O(n \cdot |G| \cdot |L|^2)$

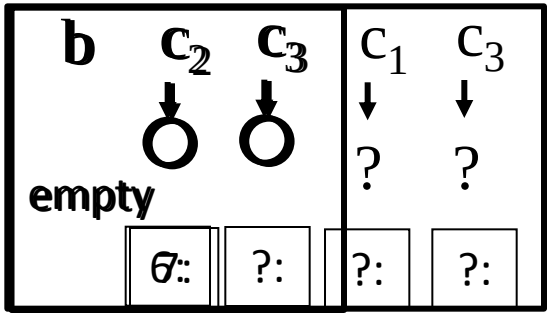
Point-wise Transformer dispose(c_1)



6: C1: dispose(c_1)



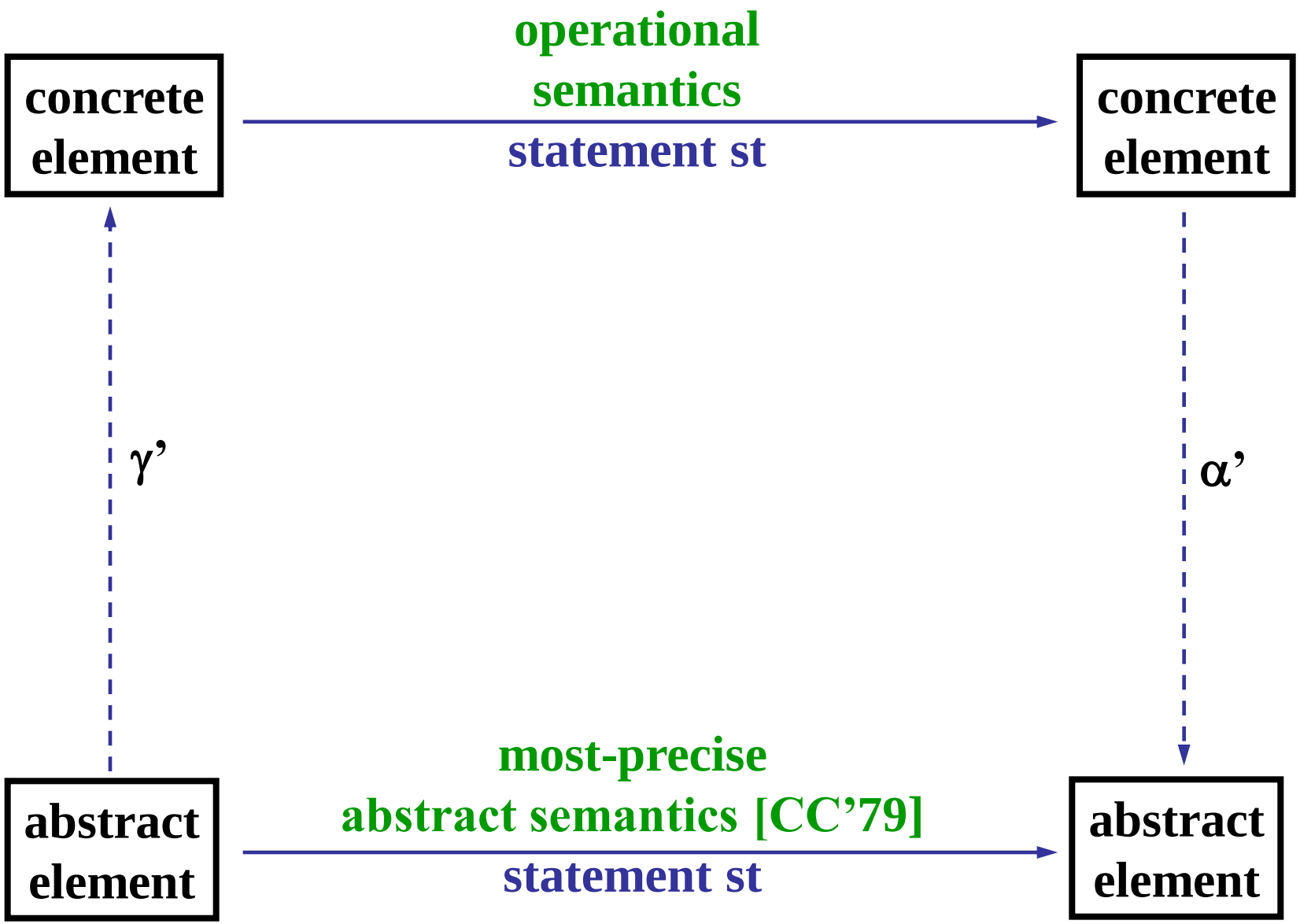
Point-wise Transformer dispose(c_1)



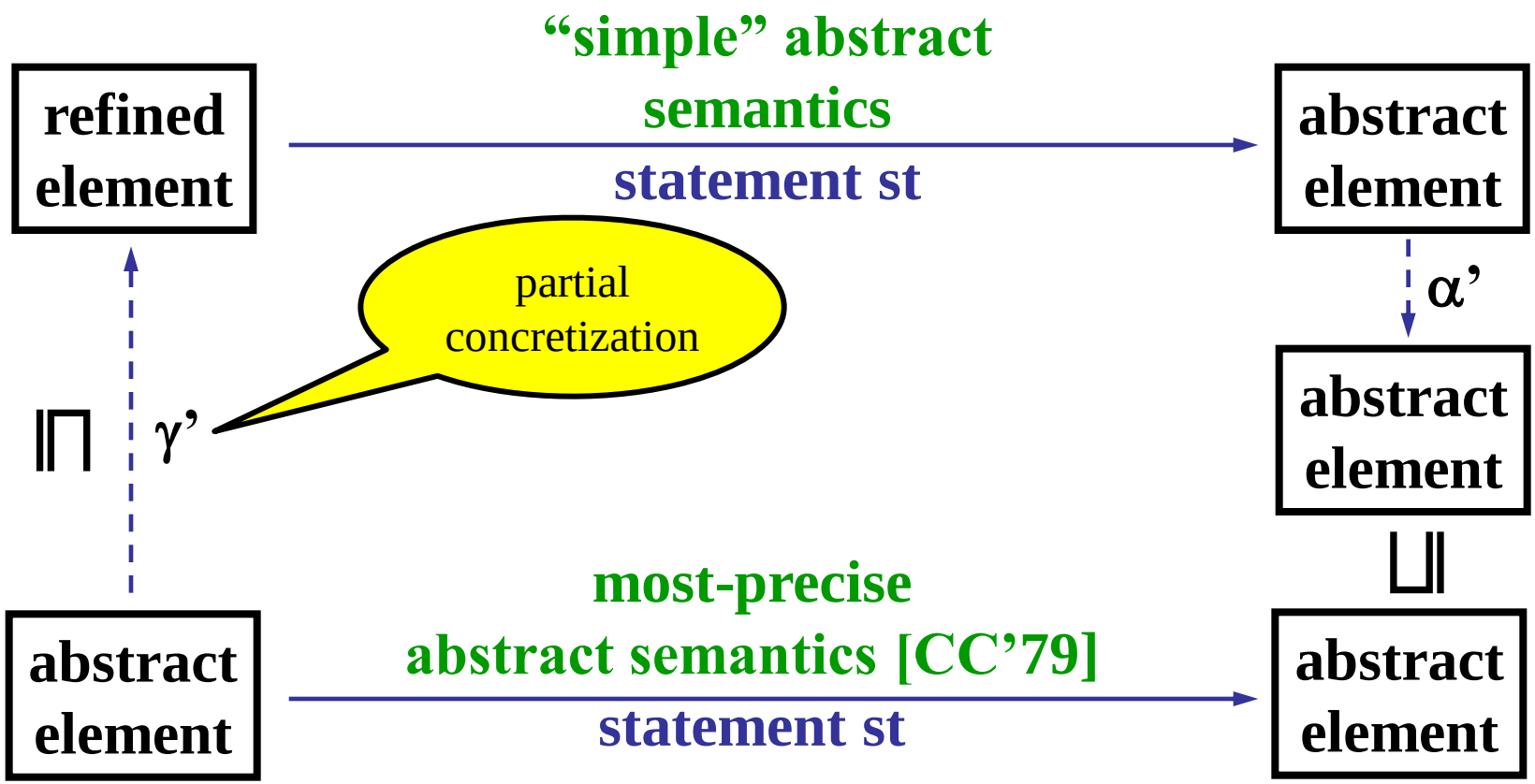
6: C1: dispose(c_1)

is this command safe
in this configuration?
missing information on c_1

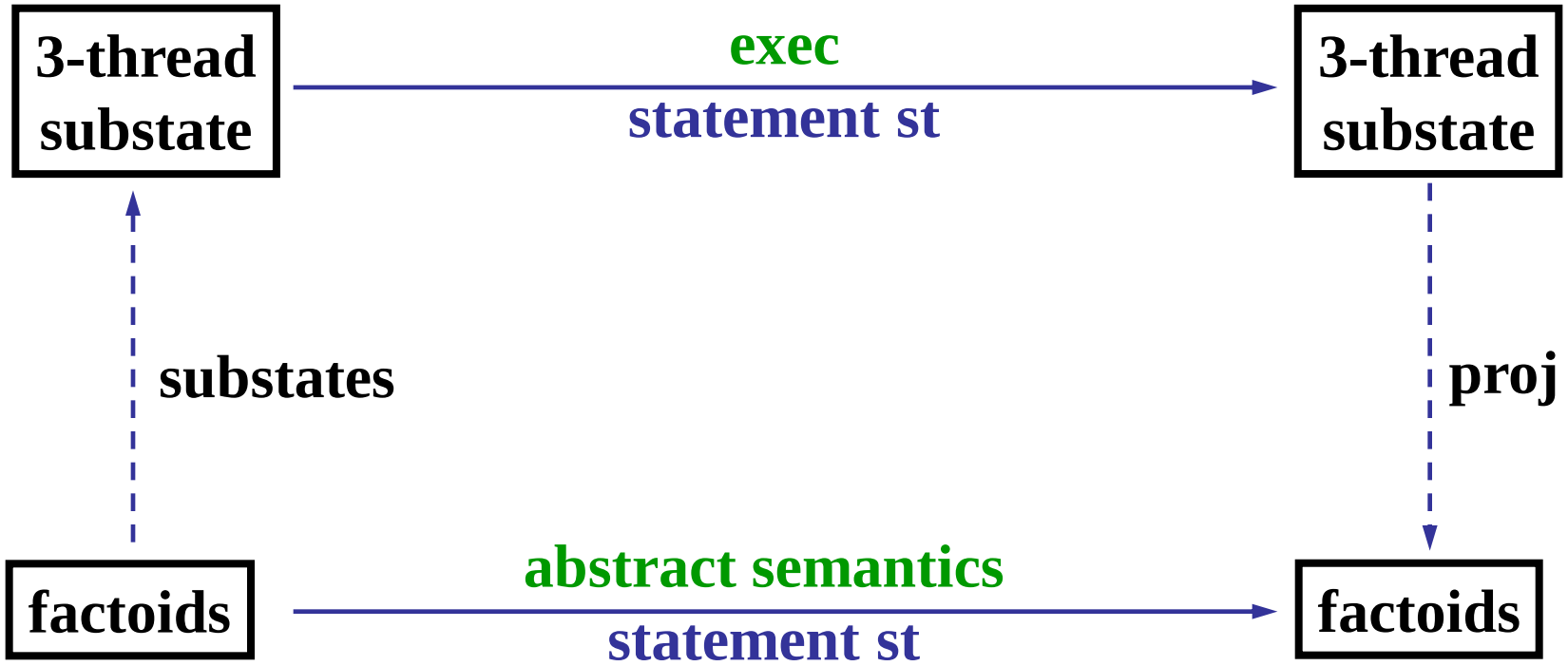
Most-Precise Transformer



Sound Transformer

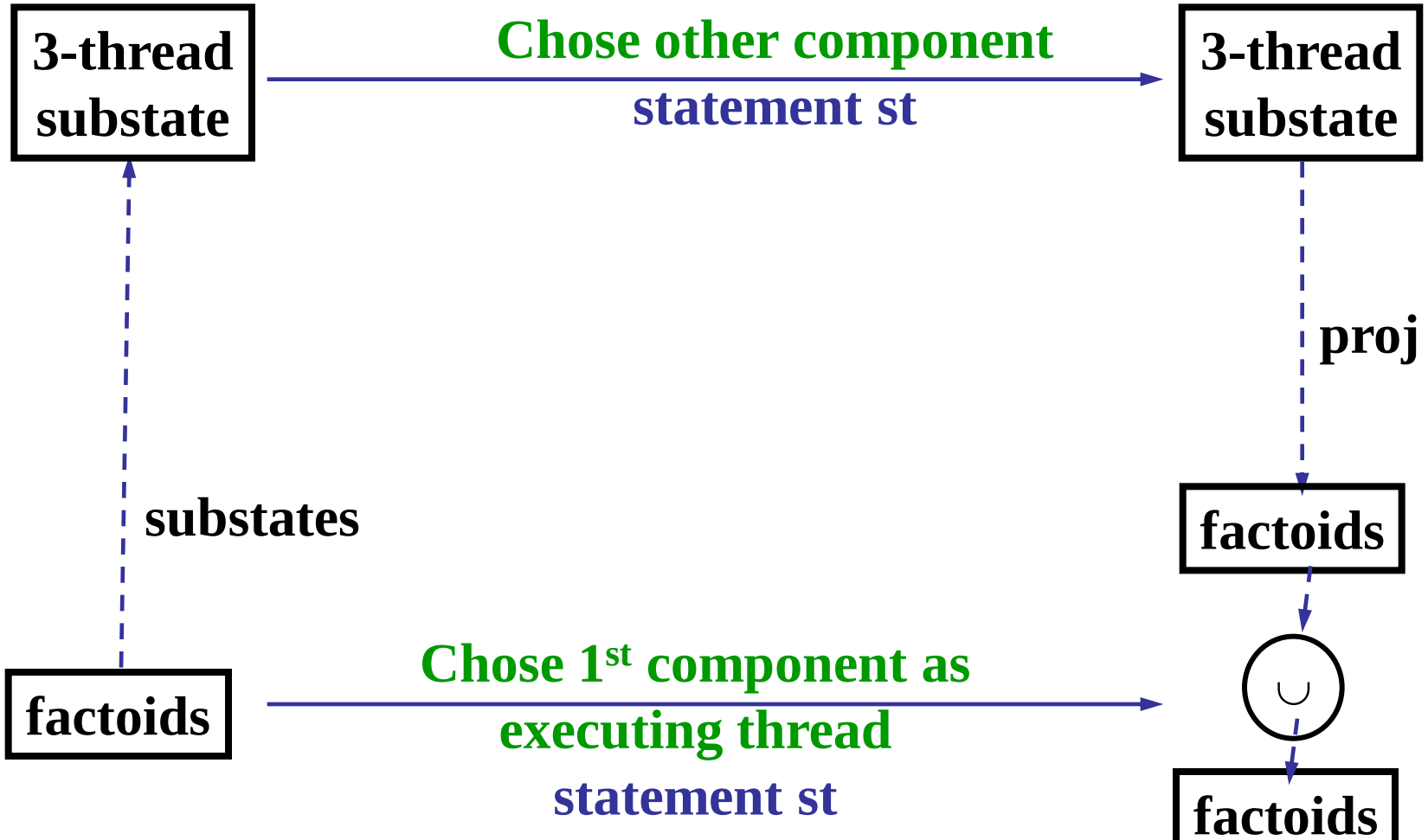


Partial Concretization Based Transformer

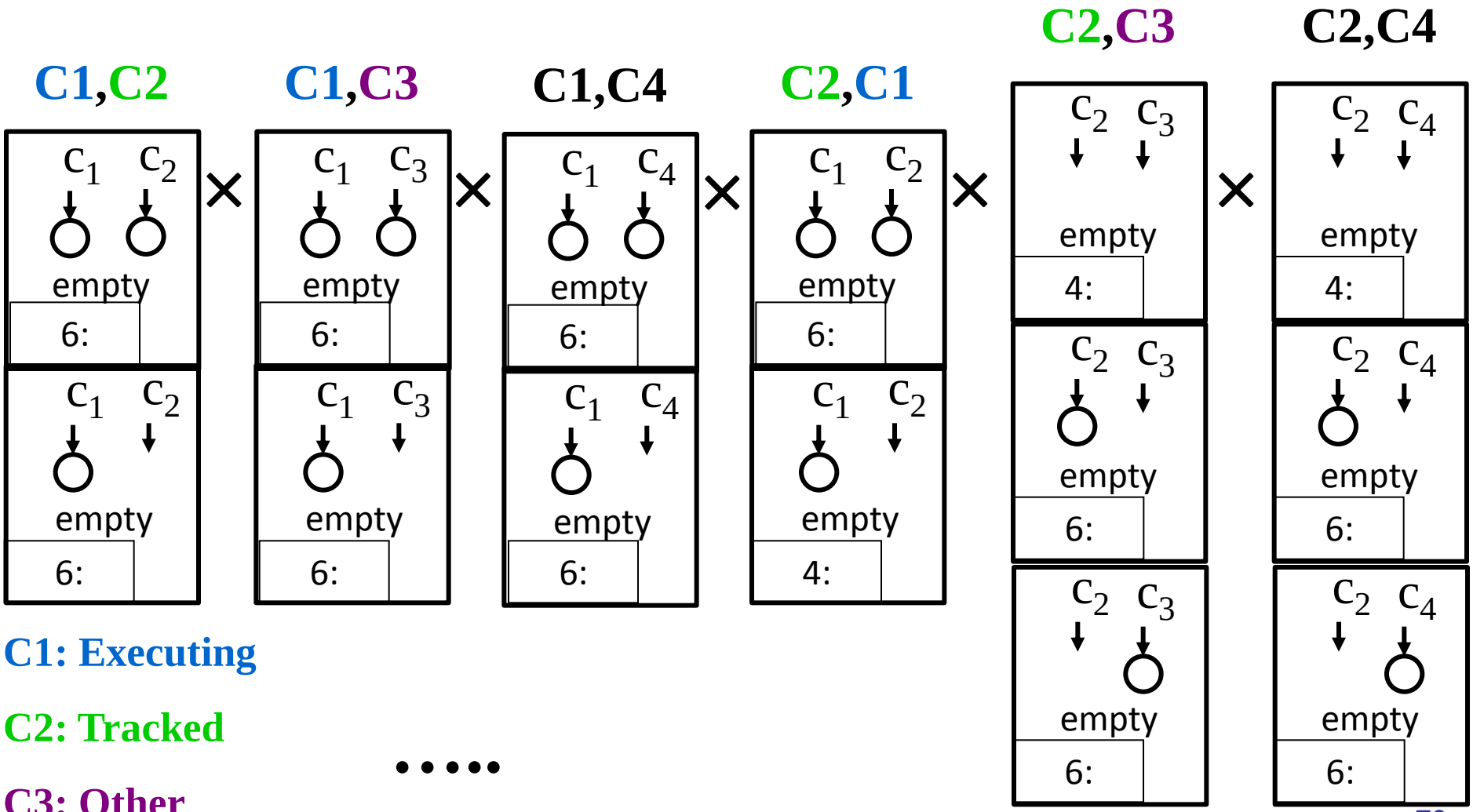


Transformer for Concurrent Systems

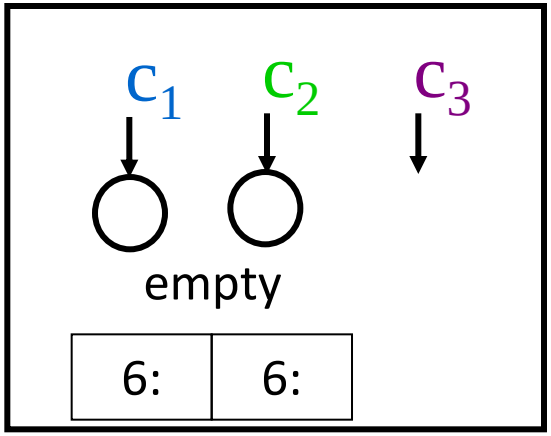
$$TR(F) = \{\langle l', g', o\rangle : \langle l, g, o\rangle \in F, \langle l, g\rangle \tau \langle l', g'\rangle\} \cup \left\{ \begin{array}{l} \langle l_2, g', o\rangle, \langle l_2, g', \alpha(l_1')\rangle : f_1, f_2, f_3, f_4 \in F: \\ \quad \langle l_1, g, l_2, o\rangle \in \text{substates}(f_1, f_2, f_3, f_4), \\ \quad \langle l_1, g\rangle \tau \langle l_1', g'\rangle \end{array} \right\}$$

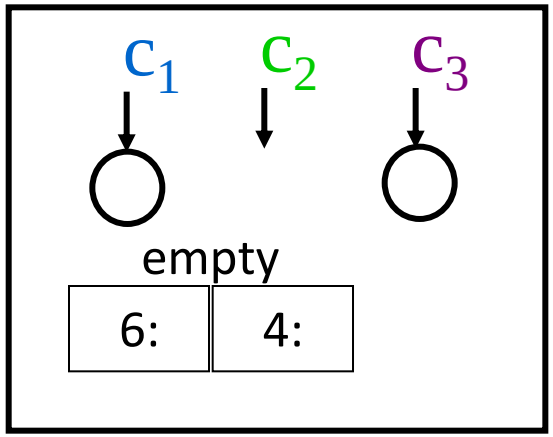


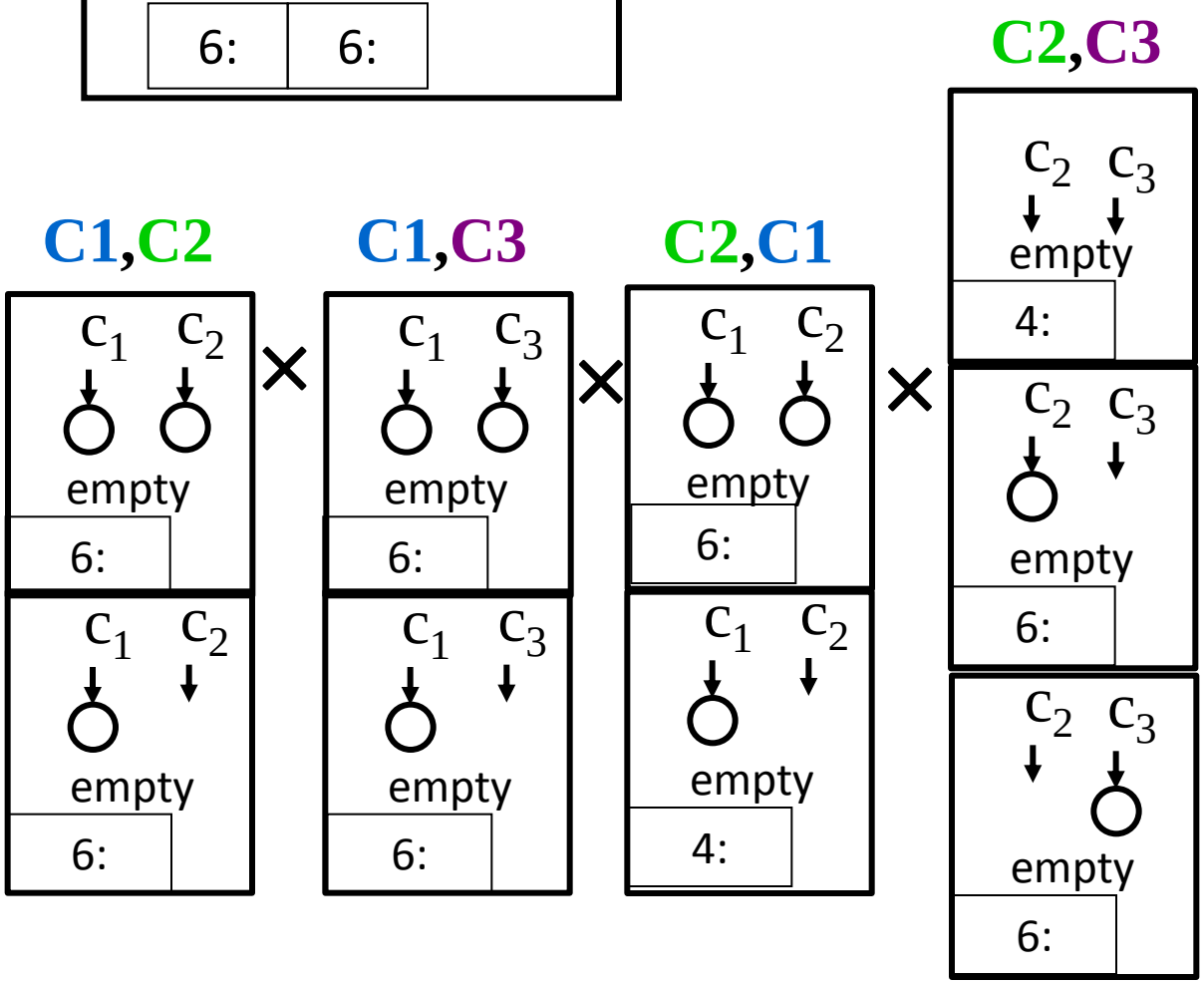
Partial Concretization



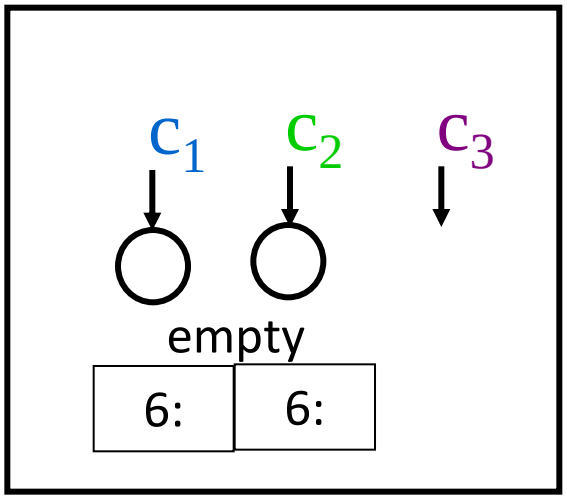
Partial Concretization(Substates)

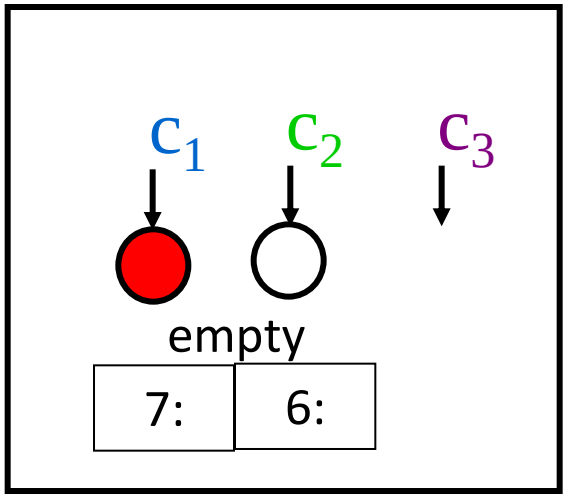


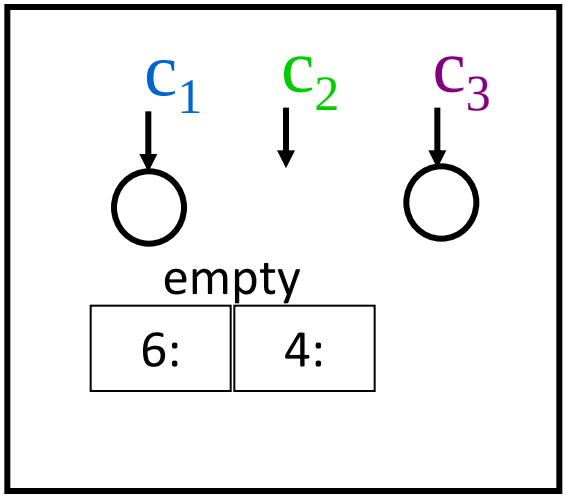


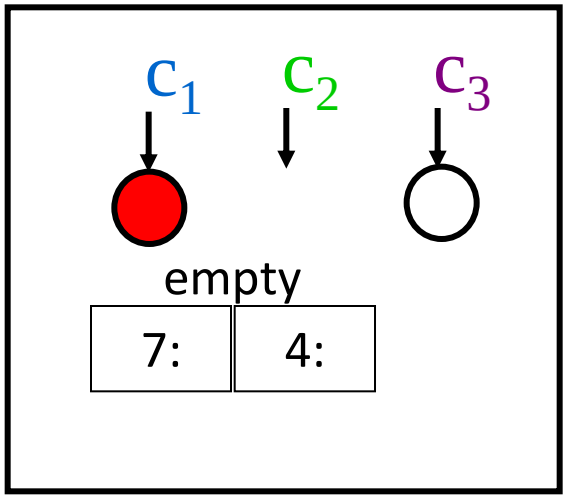


6: C1: dispose(c) (exec)

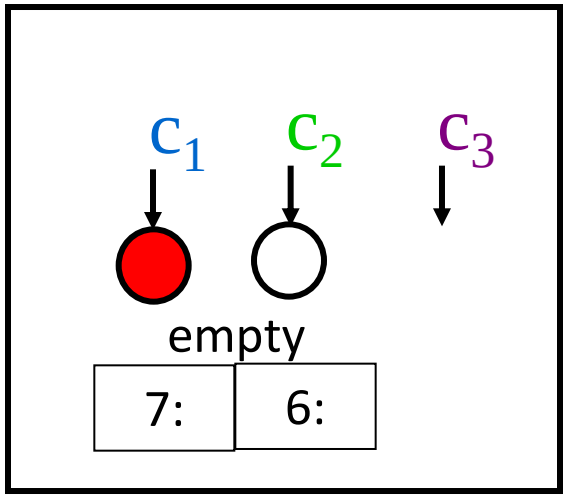


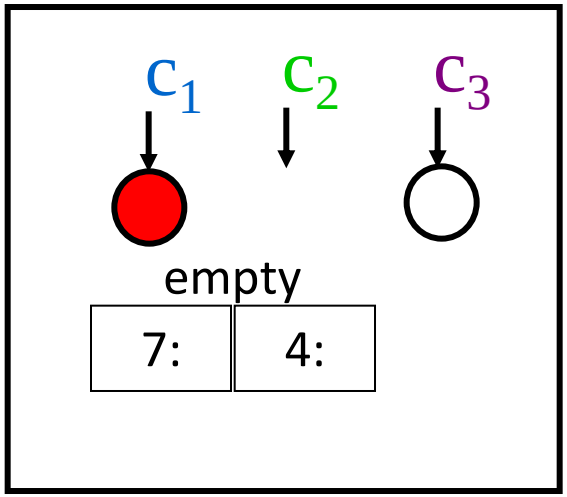


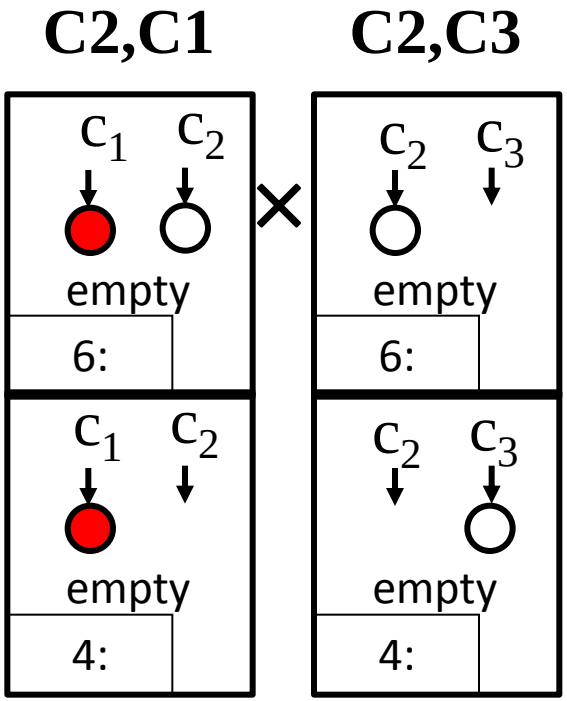




6: C1: dispose(c) (project)







Reducing Quadratic Factors

$$TR(F) = \{\langle l', g', o\rangle : \langle l, g, o\rangle \in F, \langle l, g\rangle \ \tau \ \langle l', g'\rangle\} \cup \left\{ \begin{array}{l} \langle l_2, g', o\rangle, \langle l_2, g', \alpha(l_1')\rangle : f_1, f_2, f_3, f_4 \in F: \\ \quad \langle l_1, g, l_2, o\rangle \in substates(f_1, f_2, f_3, f_4), \\ \quad \langle l_1, g\rangle \ \tau \ \langle l_1', g'\rangle \end{array} \right\}$$

- **Exploit redundancies in the action**
  - **Cannot affect locals of other threads**
  - **Use asymmetry between the two abstraction**
  - **Can prove no loss of information**
  - **Summarizing Effects**
- **Apply aggressive abstraction to the executing threads**
  - **Summarizing Abstraction**

Other Methods for Static Analysis of Concurrency

- Start by sequential analysis and then coarsen abstractions
- Rely/Guarantee reasoning
- …

Summary

- Static analysis of concurrency is interesting
- Thread modular analysis yields impressive results for well behaved coarse-grained concurrency
- Decompositions can be applied to scale abstractions even for fine-grained concurrency