SYNTHESIS OF SYNCHRONIZATION

Eran Yahav
Technion

Joint work with

Martin Vechev

Greta Yorsh

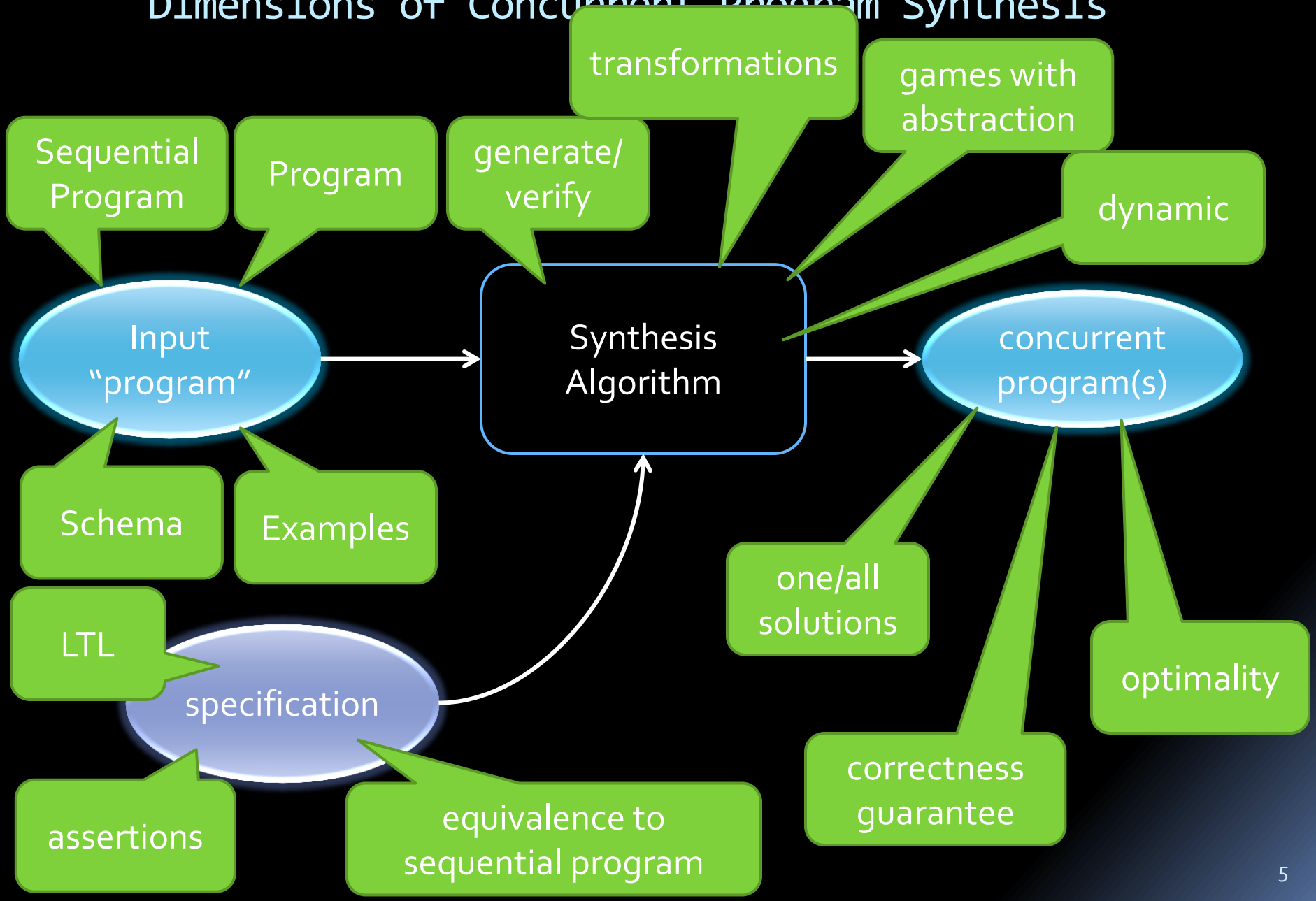
Michael Kuperstein

can we use computational power to simplify concurrent programming?

Some tasks are best done by machine, while others are best done by human insight; and a properly designed system will find the right balance.

– D. Knuth

Dimensions of Concurrent Program Synthesis



The verification problem is solved

for anything you can specify

--David Harel

Synthesis of Synchronization: Specifications

- concurrent data structures
  - should provide the illusion of sequential data structures (linearizability)
- synchronization primitives
  - simple specs (e.g., mutual exclusion)
- …

Examples

Domain	Input	Spec	Technique	Guarantee	Reference
concurrent data structures	schema	equiv to sequential	gen/verify	checked	[pldi'08]
concurrent gc	schema	gc safety	gen/verify	verified	[pldi'07]
concurrent gc	sequential program	gc safety	transformations	verified	[pldi'06]
Atomic Sections	program	safety	AGS	verified	[popl'10]
Weak Memory Models	program	safety or SC-equiv	AGS	checked, verified	[fmcad'10], [pldi'11]
Cooperative Concurrency	program	safety	AGS	verified	[tacas'09]
…					

Non-Blocking Concurrent Data Structures

- Extremely tricky to get right
- A new algorithm is a publishable result
- Very fragile, minor changes can have drastic effects
- Have to understand underlying memory model
- For performance, have to consider mem hierarchy
  - (e.g., Hopscoth Hashing / Herlihy, Shavit, Tzafrir)

- Tricky to adapt from existing implementations

Example: Concurrent Set Algorithm

```
bool add(int key)
{
 Entry *pred,*curr,*entry
 locate(pred,curr,key)
 k = (curr->key == key)
 if (k) return false
 entry = new Entry()
 entry->next = curr
 pred->next = entry
 return true
}
```

```
bool remove(int key)
{
 Entry *pred,*curr,*r
 locate(pred,curr,key)
 k = (curr->key ≠ key)
 if (k) return false
 r = curr->next
 pred->next = r
 return true
}
```

```
bool contains(int key)
{
 Entry *pred,*curr
 locate(pred,curr,key)
 k = (curr->key == key)
 if (k) return true
 if (¬k) return false
}
```

```
locate(int k) {
 pred = head
 curr = head->next
 while(curr->key < k)
 {
  pred=curr
  curr=curr->next
 }
}
```

Systematically derived with machine assistance
Correctness – machine checked
"Performance" – only uses CAS

```
bool a
 Entry *pred, *curr, *entry
restart:
 locate(pred,curr,key)
 k = (curr->key == key)
 if (k) return false
 entry = new Entry()
 entry->next = curr
 val=CAS(&pred->next,⟨curr.ptr,0⟩,⟨entry.ptr,0⟩)
 if (¬val) goto restart
 if (¬k) return true
 goto restart
}
```

```
 Entry *pred, *curr, *r
restart:
 locate(pred,curr,key)
 k = (curr->key ≠ key)
 if (k) return false
 r = curr->next
 lval=CAS(&curr->next, ⟨r.ptr,0⟩,⟨r.ptr,1⟩)
 if (¬lval) goto restart
 pval=CAS(&pred->next,⟨curr.ptr,0⟩,⟨r.ptr,0⟩)
 if (¬pval) goto restart
 pred->next = r
 if (¬k) return true
}
```

Memory Fences

- Fences are expensive
  - 10s-100s of cycles
  - collateral damage (e.g., prevent compiler optimizations)
- Required fences depend on memory model
- Where should I put fences?

Synthesis of Memory Fences

- There are ~3500 fences in the Linux Kernel
- Which fences are required on x86?
  - removal of one redundant fence in spin lock => 4% performance improvement **[Sewell]**

- Automatically synthesize fences required to satisfy a given specification

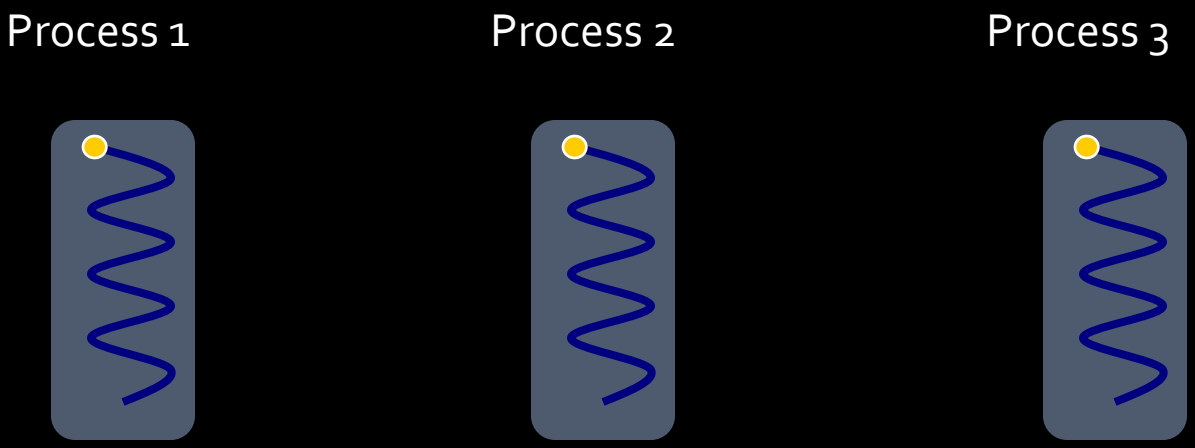
- Successfully synthesized required fences for
  - concurrent data structures
  - mutual exclusion primitives

Synthesizing locks from atomic sections

- Program annotated with atomic sections
- Use static information about the shape of the heap to enable automatic inference of locks
- New locking protocol: domination locking
- Automatic techniques for enforcing the protocol

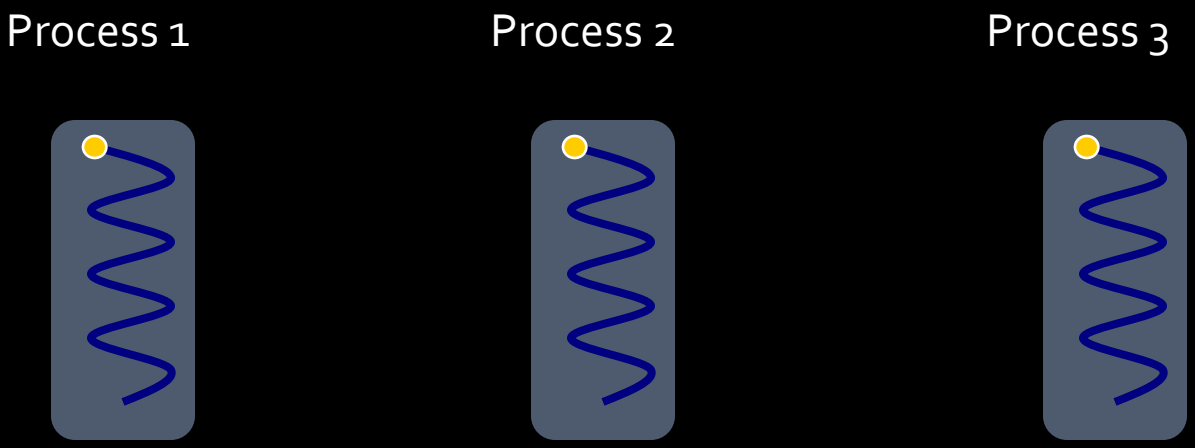
[Guy Gueta et. al. OOPSLA'11]

Challenge: Correct and Efficient Synchronization



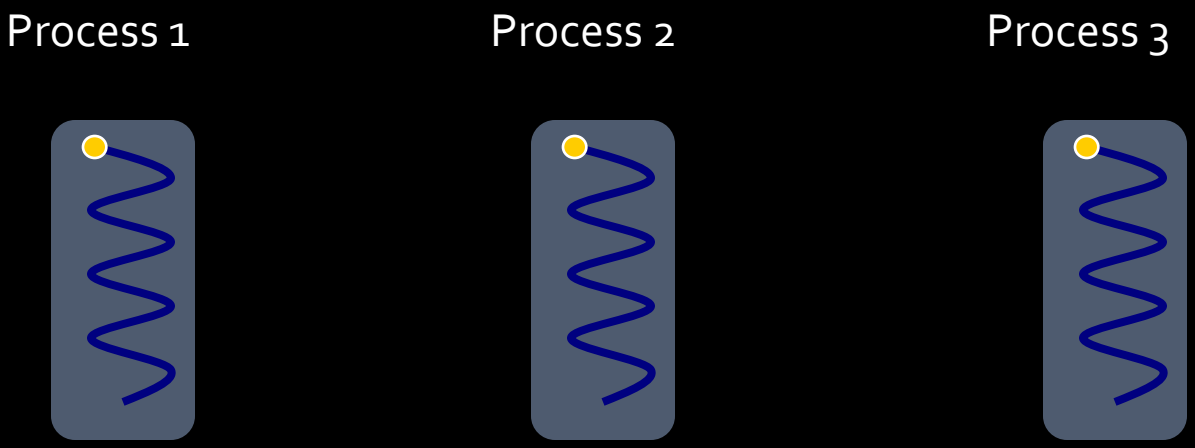
- Shared memory concurrent program
- No synchronization: often incorrect (but "efficient")
- Coarse-grained synchronization: easy to reason about, often inefficient
- Fine-grained synchronization: hard to reason about, programmer often gets this wrong

Challenge: Correct and Efficient Synchronization



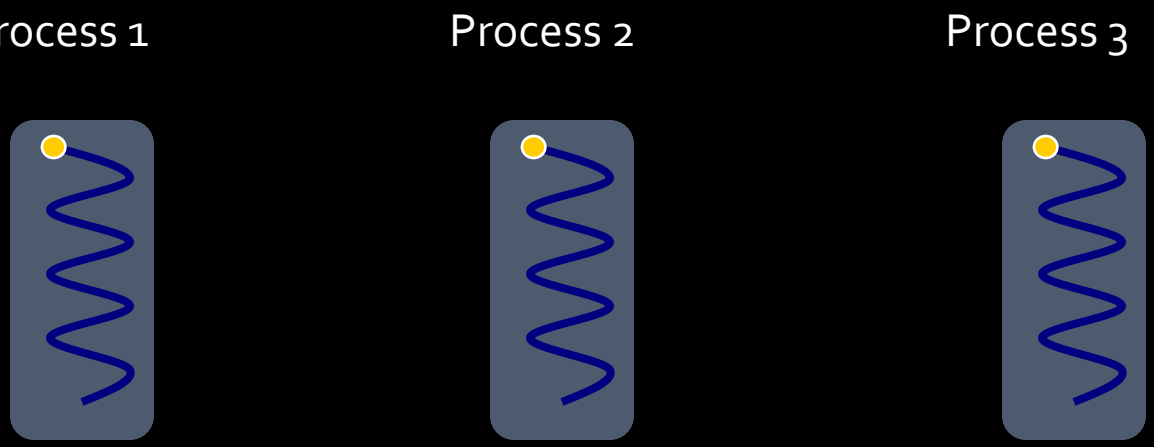
- Shared memory concurrent program
- No synchronization: often incorrect (but "efficient")
- Coarse-grained synchronization: easy to reason about, often inefficient
- Fine-grained synchronization: hard to reason about, programmer often gets this wrong

Challenge: Correct and Efficient Synchronization



- Shared memory concurrent program
- No synchronization: often incorrect (but "efficient")
- Coarse-grained synchronization: easy to reason about, often inefficient
- Fine-grained synchronization: hard to reason about, programmer often gets this wrong

Challenge

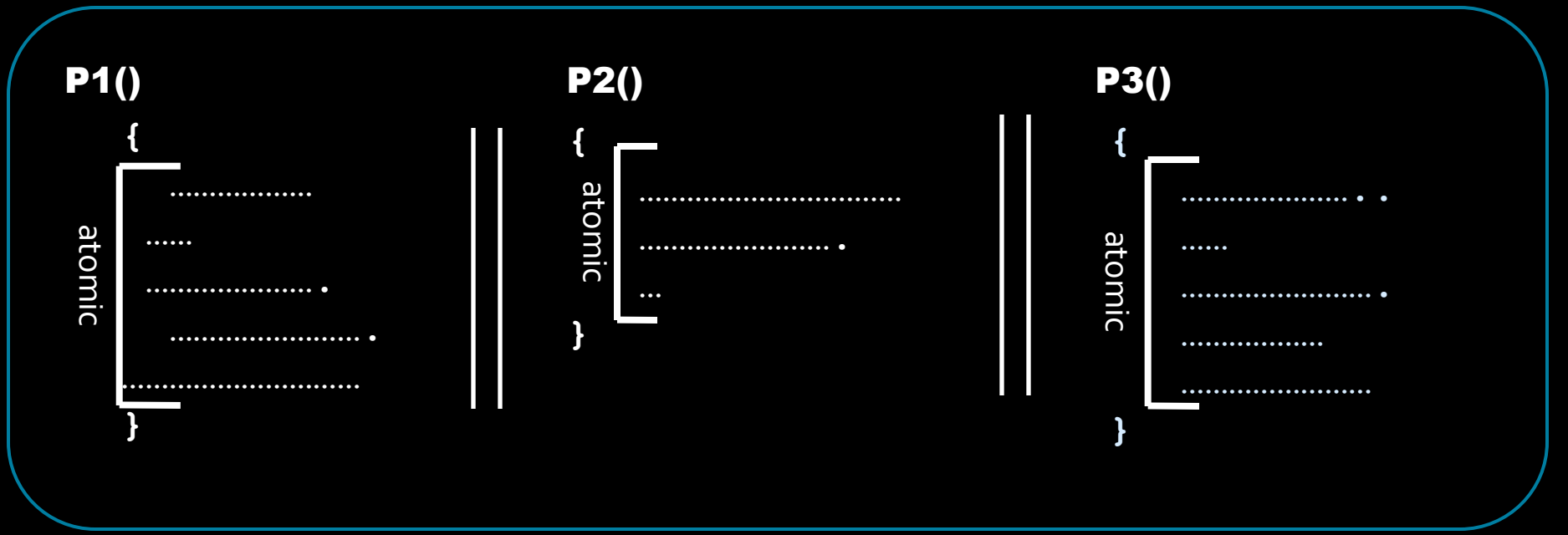


How to synchronize processes to achieve correctness and efficiency?

Synchronization Primitives

- Atomic sections
- Conditional critical region (CCR)
- Memory barriers (fences)
- CAS
- Semaphores
- Monitors
- Locks
- ….

Example: Correct and Efficient Synchronization with Atomic Sections



Safety Specification: S

Example: Correct and Efficient Synchronization with Atomic Sections

Safety Specification: S

Assist the programmer by automatically inferring correct and efficient synchronization

Challenge

- Find minimal synchronization that makes the program satisfy the specification
  - Avoid all bad interleavings while permitting as many good interleavings as possible

- Assumption: we can prove that serial executions satisfy the specification
  - Interested in bad behaviors due to concurrency

Sounds Suspiciously Like a Game?

- A (safety) game of program vs. scheduler
  - program's goal - avoid error states
- A winning strategy for the program avoids error states despite adversarial scheduler
- Implementation mechanisms
- Find a solution that corresponds to minimal synchronization
- Infinite-state systems

Synthesis of Synchronization with Abstract Interpretation

- Compute over-approximation of all possible program executions
- Add minimal synchronization to avoid (over-approximation of) bad interleavings

- Interplay between abstraction and synchronization
  - Finer abstraction may enable finer synchronization
  - Coarse synchronization may enable coarser abstraction

Crash Course on Abstract Interpretation

- verify that property holds on **all** executions
- challenges
  - programs with unbounded state
  - non trivial properties

☹ bad news: problem is undecidable

☺ good news: can use over-approximation

- Consider a superset of possible executions
- **sound**: a **yes** is a **yes**!
- **incomplete**: a **no** is a **maybe** …

Verification Challenge

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

Determine what states can arise during any execution

Challenge: set of states is unbounded

Abstract Interpretation

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

French Recipe

1) Abstraction
2) Transformers
3) Exploration

Determine what states can arise during any execu

Challenge: set of states is unbounded

Solution: compute a bounded representation of (a superset) of program states

1) Abstraction

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

- concrete state

ρ: Var → $\mathbb{Z}$

- abstract state

$\rho^{\#}$: Var → {+, o, -, ?}

x	y	i
3	1	7

x	y	i
3	2	6

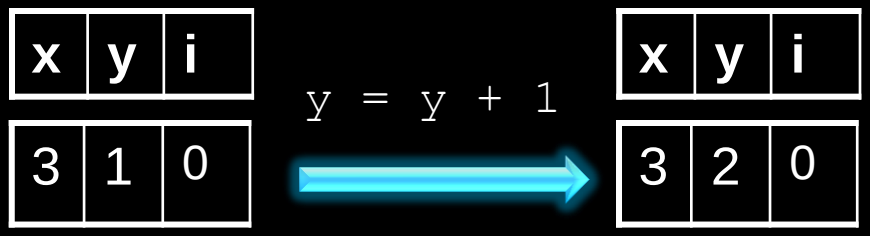
...

x	y	i
+	+	+

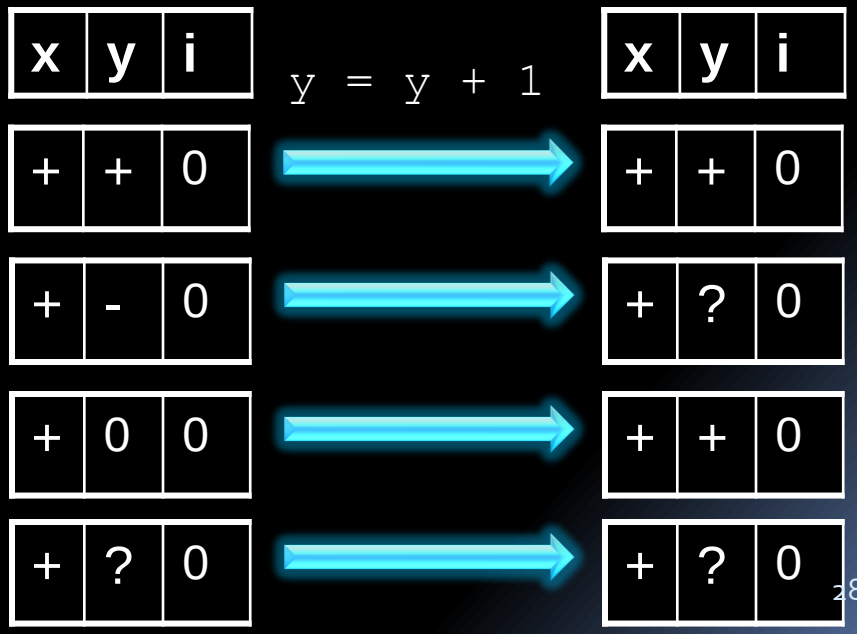
2) Transformers

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

- concrete transformer

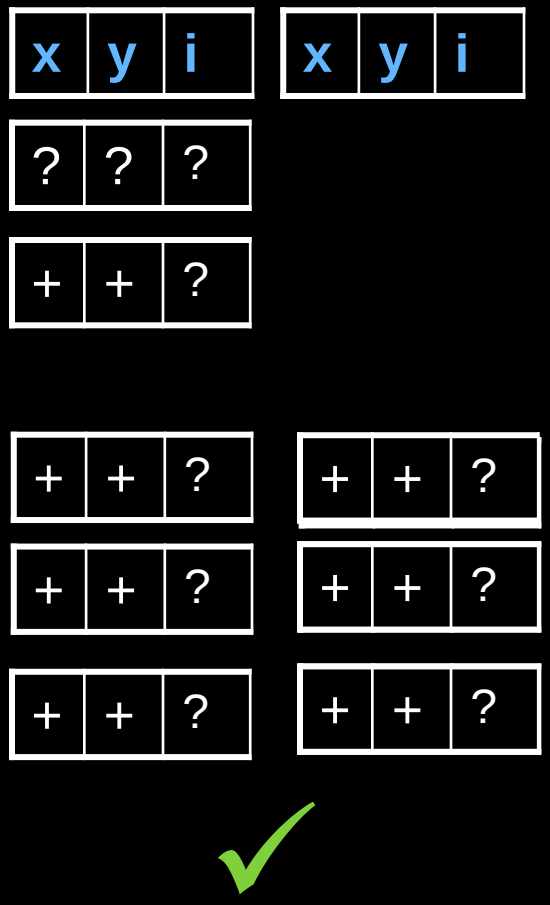


- abstract transformer



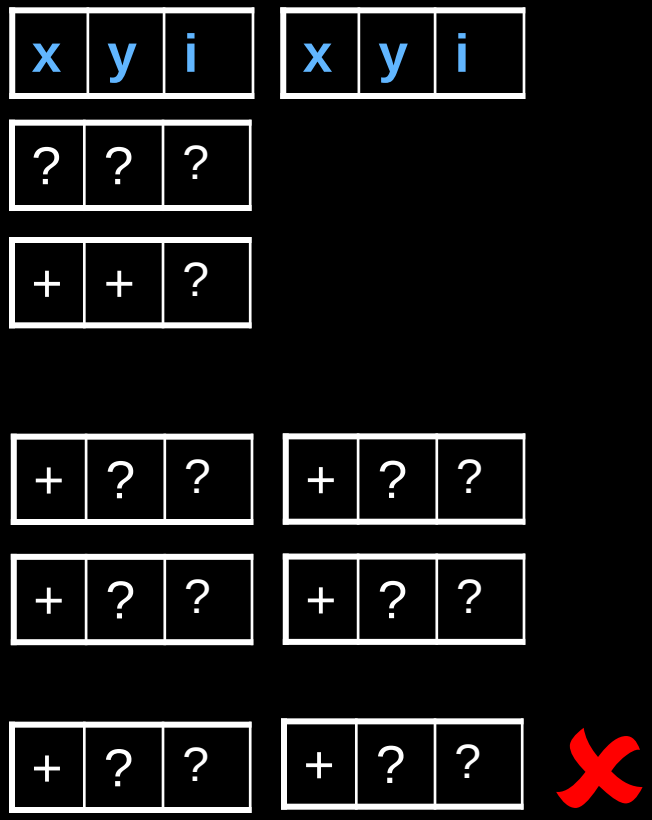
3) Exploration

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```



Incompleteness

```
main(int i) {
 int x=3,y=1;

 do {
  y = y - 2;
  y = y + 3;
 } while(--i > 0)
 assert 0 < x + y
}
```

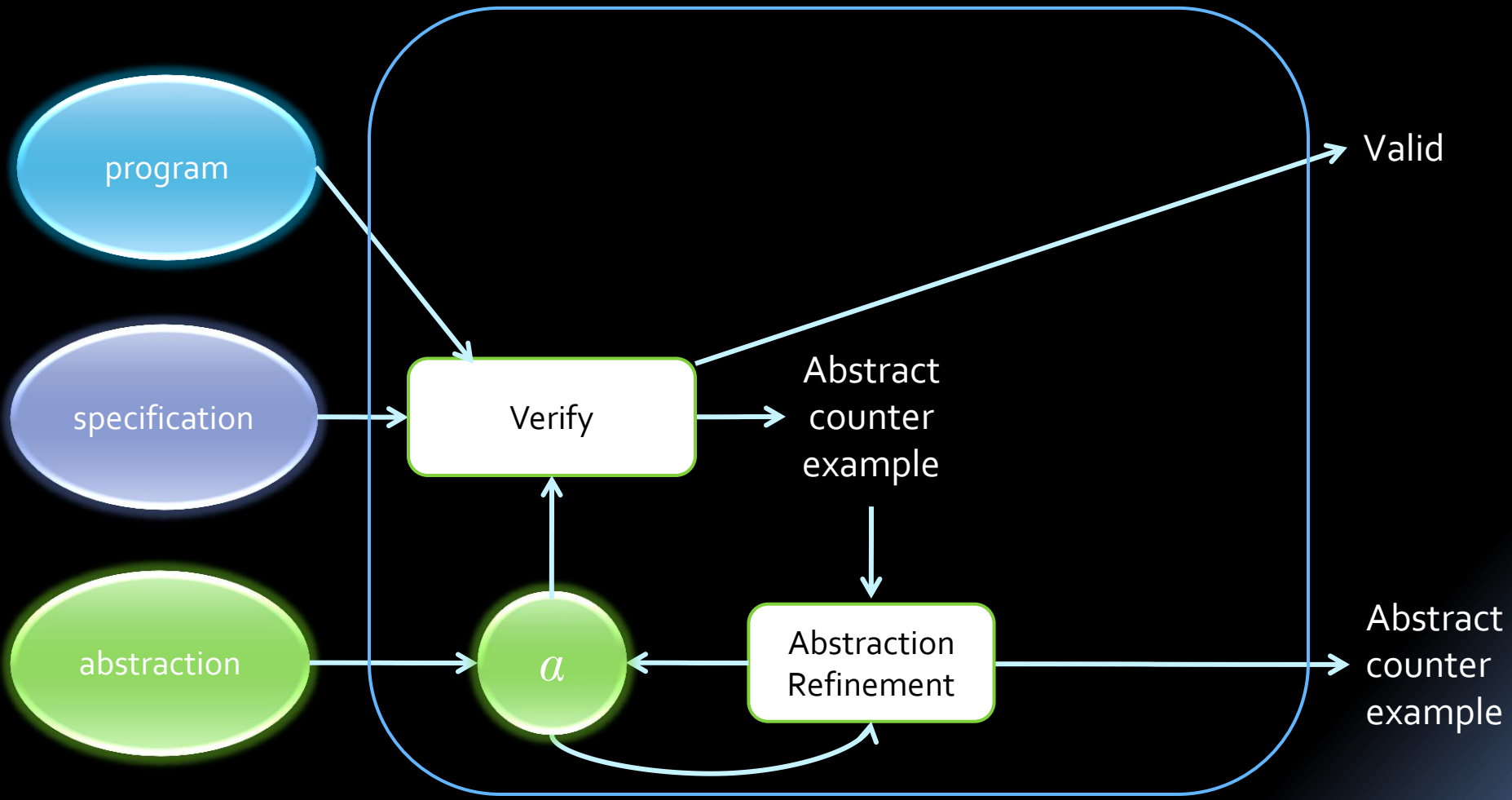


Parity Abstraction

```
while (x !=1 ) do {
   if (x % 2) == 0 {
      x := x / 2;
   } else {
      x := x * 3 + 1;
      assert (x %2 ==0);
   }
}
```

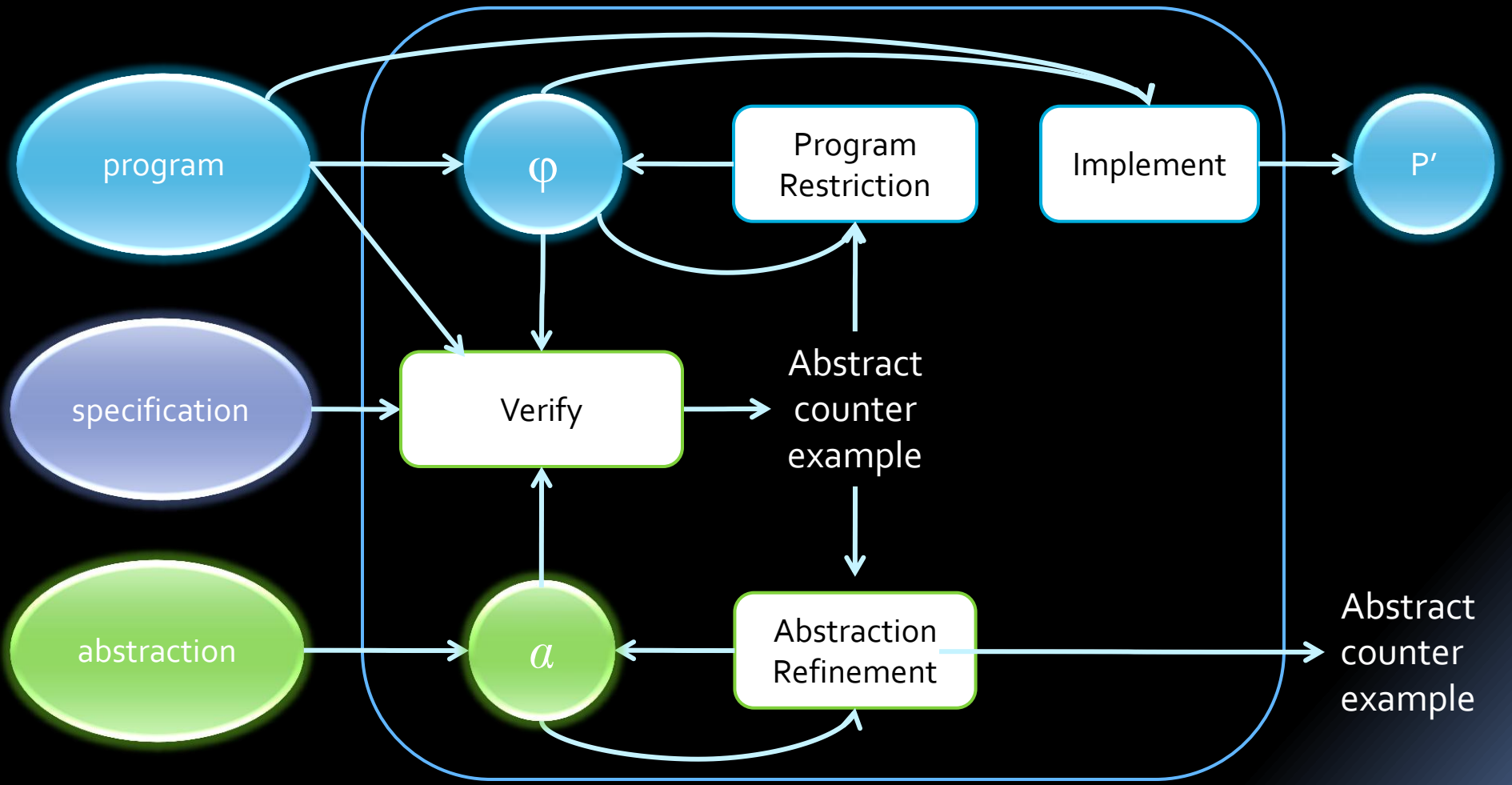
challenge: how to find “the right” abstraction

A Standard Approach: Abstraction Refinement



Change the **abstraction** to match the **program**

Our Approach: Abstraction-Guided Synthesis



Change the **program** to match the **abstraction**

AGS Trace-Based Algorithm – High Level

Input: Program P, Specification S, Abstraction α

Output: Program P′ satisfying S under α

```
φ = true
while(true) {
  BadTraces = {π | π ∈ ([[P]]_α ∩ [[φ]]) and π ∉ S }
  if (BadTraces is empty) return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    ψ = avoid(π)
    if (ψ ≠ false) φ = φ ∧ ψ
      else abort
  } else {
    α′ = refine(α, π)
    if (α′ ≠ α) α = α′
      else abort
  }
}
```

Avoid and Implement

- Desired output – program satisfying the spec
- Implementability guides the choice of constraint language
- Examples
  - Atomic sections [POPL'10]
  - Conditional critical regions (CCRs) [TACAS'09]
  - Memory fences (for weak memory models) [FMCAD'10 + PLDI'11]
  - …

Avoiding an interleaving with atomic sections

- Adding atomicity constraints
  - Atomicity predicate [l1,l2] – no context switch allowed between execution of statements at l1 and l2
- `avoid(π)`
  - A disjunction of all possible atomicity predicates that would prevent π

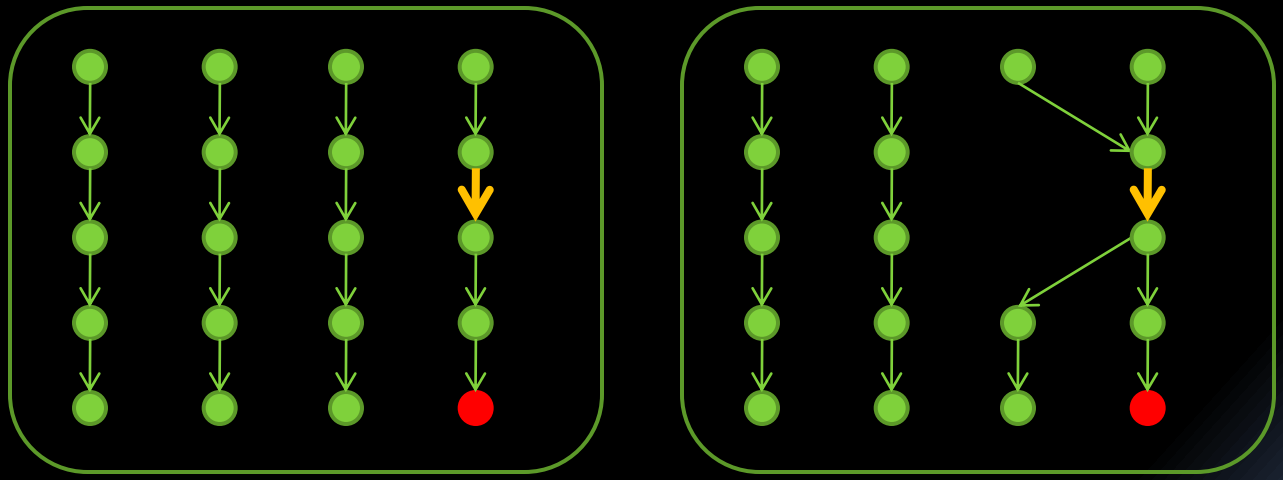
Thread A	Thread B
A1	B1
A2	B2

$\pi = A_1 B_1 A_2 B_2$

$\text{avoid}(\pi) = [A_1, A_2] \vee [B_1, B_2]$

Avoid and abstraction

- ψ = avoid(π)
- Enforcing ψ avoids any abstract trace π' such that $\pi' \not\models \psi$
- Potentially avoiding "good traces"
- Abstraction may affect our ability to avoid a smaller set of traces

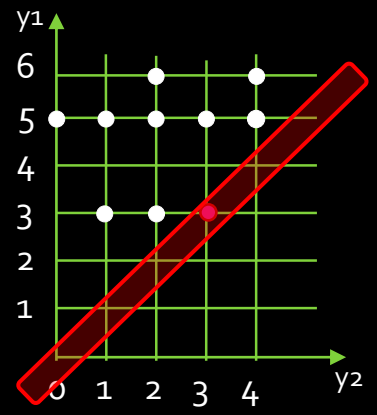


Example

T1	T2	T3
1: x += z	1: z++	1: y1 = f(x)
2: x += z	2: z++	2: y2 = x
		3: assert(y1 != y2)

```
f(x)  {
  if (x == 1) return 3
  else if (x == 2) return 6
  else return 5
}
```

Example: Concrete Values



Concrete values

x += z; x += z; z++;z++;y1=f(x);y2=x;assert ➔ y1=5,y2=0

z++; x+=z; y1=f(x); z++; x+=z; y2=x;assert ➔ y1=3,y2=3

⋮

T1

1: x += z
2: x += z

||

T2

1: z++
2: z++

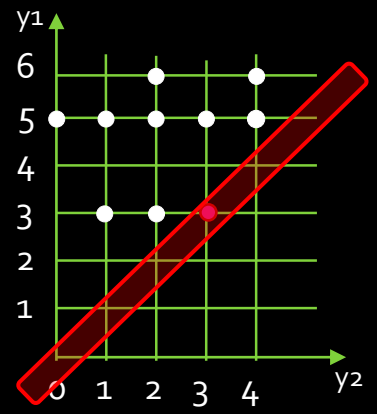
||

T3

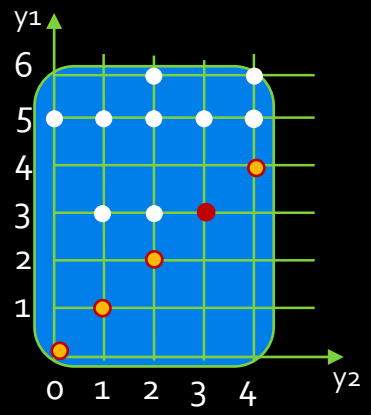
1: y1 = f(x)
2: y2 = x
3: assert(y1 != y2)

```
f(x)  {
  if (x == 1) return 3
  else if (x == 2) return 6
  else return 5
}
```

Example: Parity Abstraction



Concrete values



Parity abstraction (even/odd)

x += z; x += z; z++;z++;y1=f(x);y2=x;assert ➔ y1=Odd,y2=Even

T1	T2	T3
1: x += z	1: z++	1: y1 = f(x)
2: x += z	2: z++	2: y2 = x
		3: assert(y1 != y2)

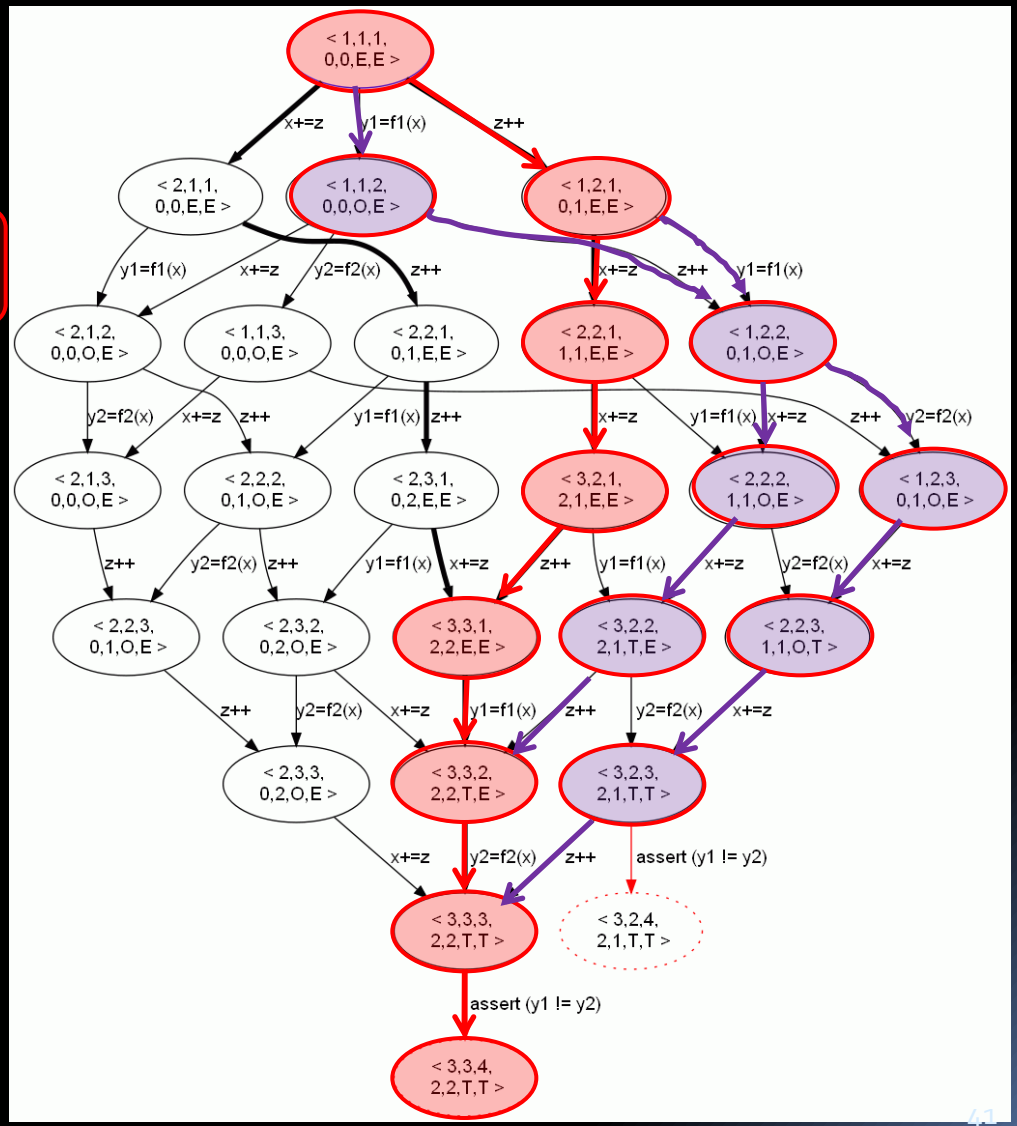
```
f(x)  {
  if (x == 1) return 3
  else if (x == 2) return 6
  else return 5
}
```

Example: Avoiding Bad Interleavings

```
φ = true
while(true) {
  BadTraces={π|π∈([[P]]_α ∩ [[φ]]) and
                π ∉ S }
  if (BadTraces is empty)
      return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    φ = φ ∧ avoid(π)
  } else {
    α = refine(α, π)
  }
}
```

avoid(π_1) = [z++,z++]

φ = true

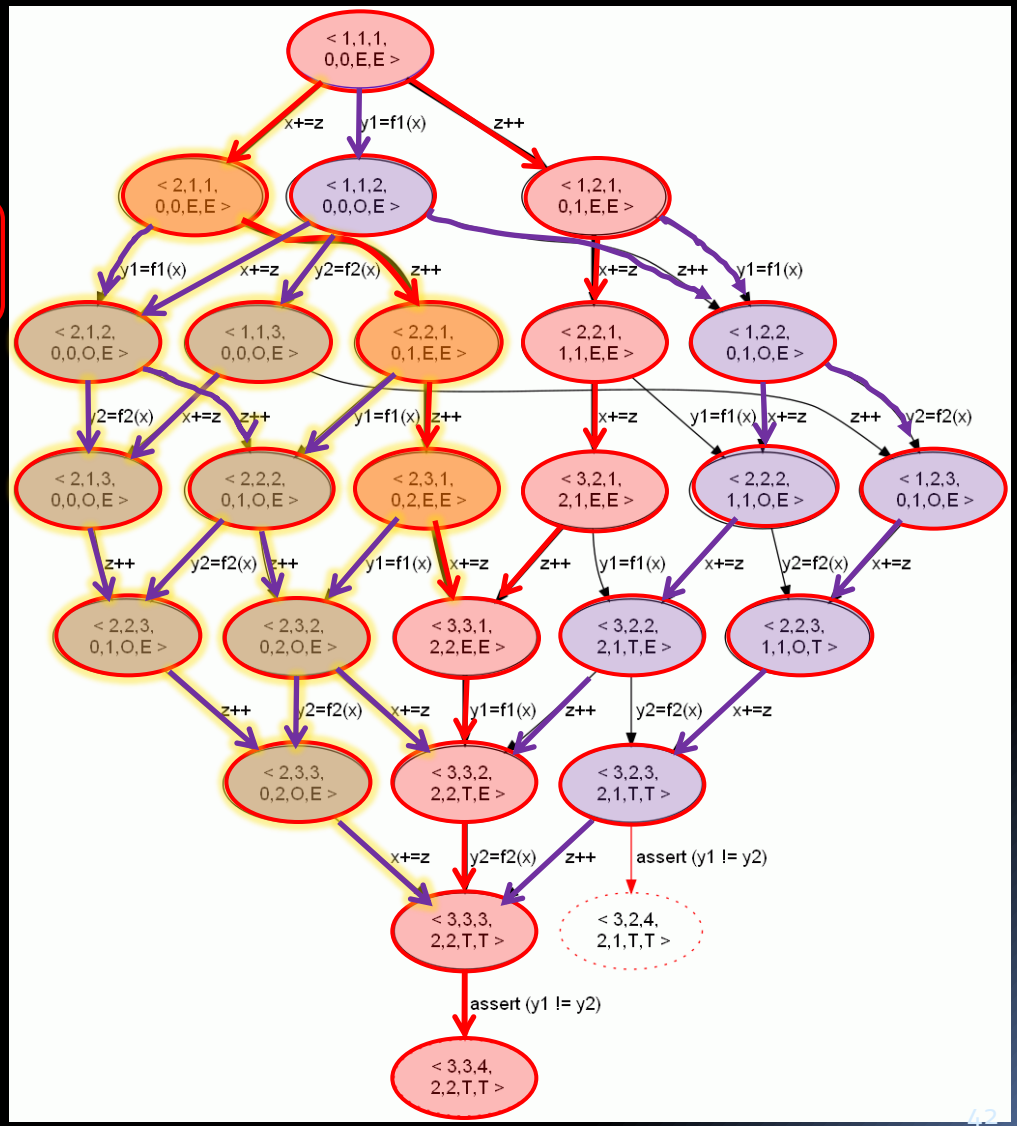


Example: Avoiding Bad Interleavings

```
φ = true
while(true) {
  BadTraces={π|π∈([[P]]_α ∩ [[φ]]) and
                π ∉ S }
  if (BadTraces is empty)
      return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    φ = φ ∧ avoid(π)
  } else {
    α = refine(α, π)
  }
}
```

avoid(π_2) =[x+=z,x+=z]

φ = [z++,z++]∧[x+=z,x+=z]



Example: Avoiding Bad Interleavings

```
φ = true
while(true) {
   BadTraces={π|π∈([[P]]_α ∩ [[φ]]) and
                π ⊭ S }
  if (BadTraces is empty)
      return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    φ = φ ∧ avoid(π)
  } else {
    α = refine(α, π)
  }
}
```

T1

1: x += z
2: x += z

T2

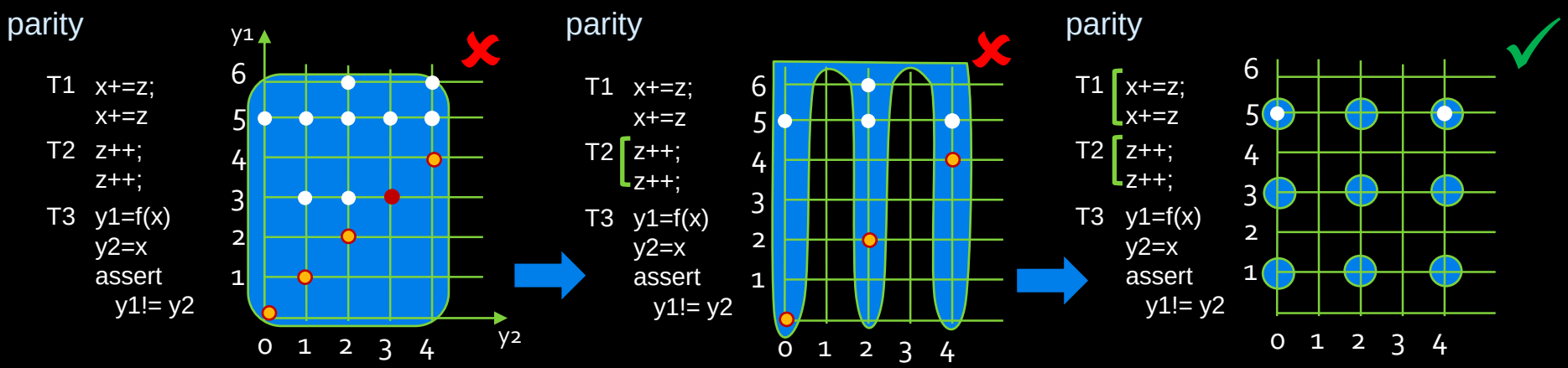
1: z++
2: z++

T3

1: y1 = f(x)
2: y2 = x
3: assert(y1 != y2)

φ = [z++,z++]∧[x+=z,x+=z]

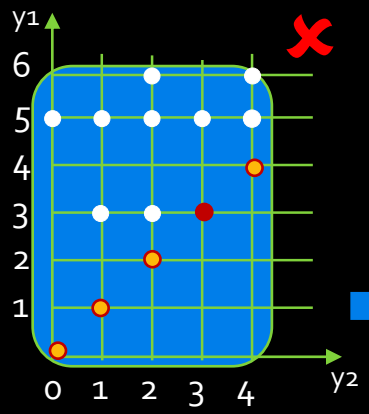
Example: Avoiding Bad Interleavings



But we can also refine the abstraction…

parity

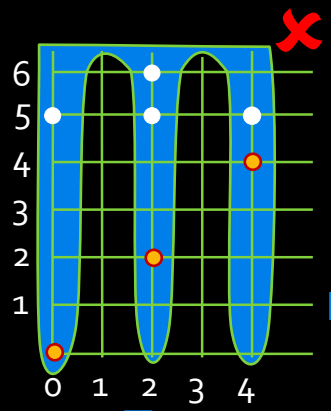
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(a)

parity

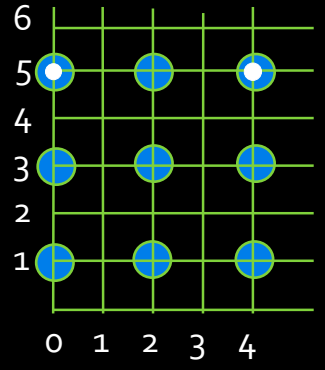
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(b)

parity

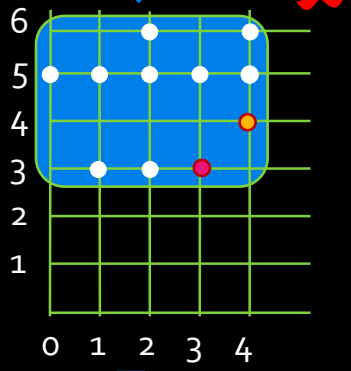
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(c)

interval

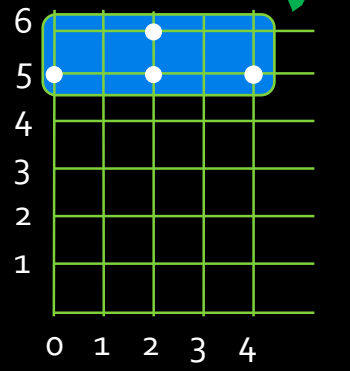
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(d)

interval

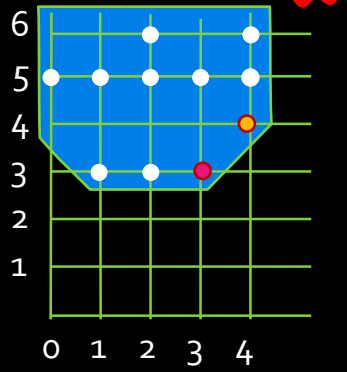
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(e)

octagon

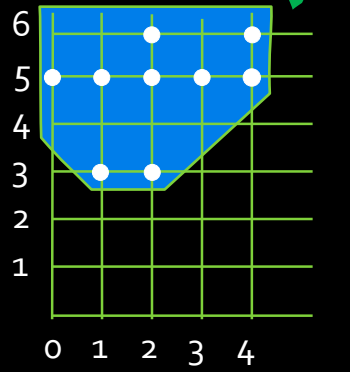
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(f)

octagon

T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(g)

Multiple Solutions

- Performance: smallest atomic sections
- Interval abstraction for our example produces the atomicity constraint:

  ([x+=z,x+=z] V [z++,z++])
  Λ ([y1=f(x),y2=x] V [x+=z,x+=z] V [z++,z++])

- Minimal satisfying assignments
  - Γ_1 = [z++,z++]
  - Γ_2 = [x+=z,x+=z]

AGS Trace-Baesd Algorithm – More Details

Input: Program P, Specification S

Output: Program P′ satisfying S

```
φ = true
while(true) {
  BadTraces = {π | π ∈ ([[P]]_α ∩ [[φ]]) and π ⊭ S }
  if (BadTraces is empty) return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    ψ = avoid(π)
    if (ψ ≠ false) φ = φ ∧
      else abort
  } else {
    α′ = refine(α, π)
    if (α′ ≠ α) α = α′
      else abort
  }
}
```

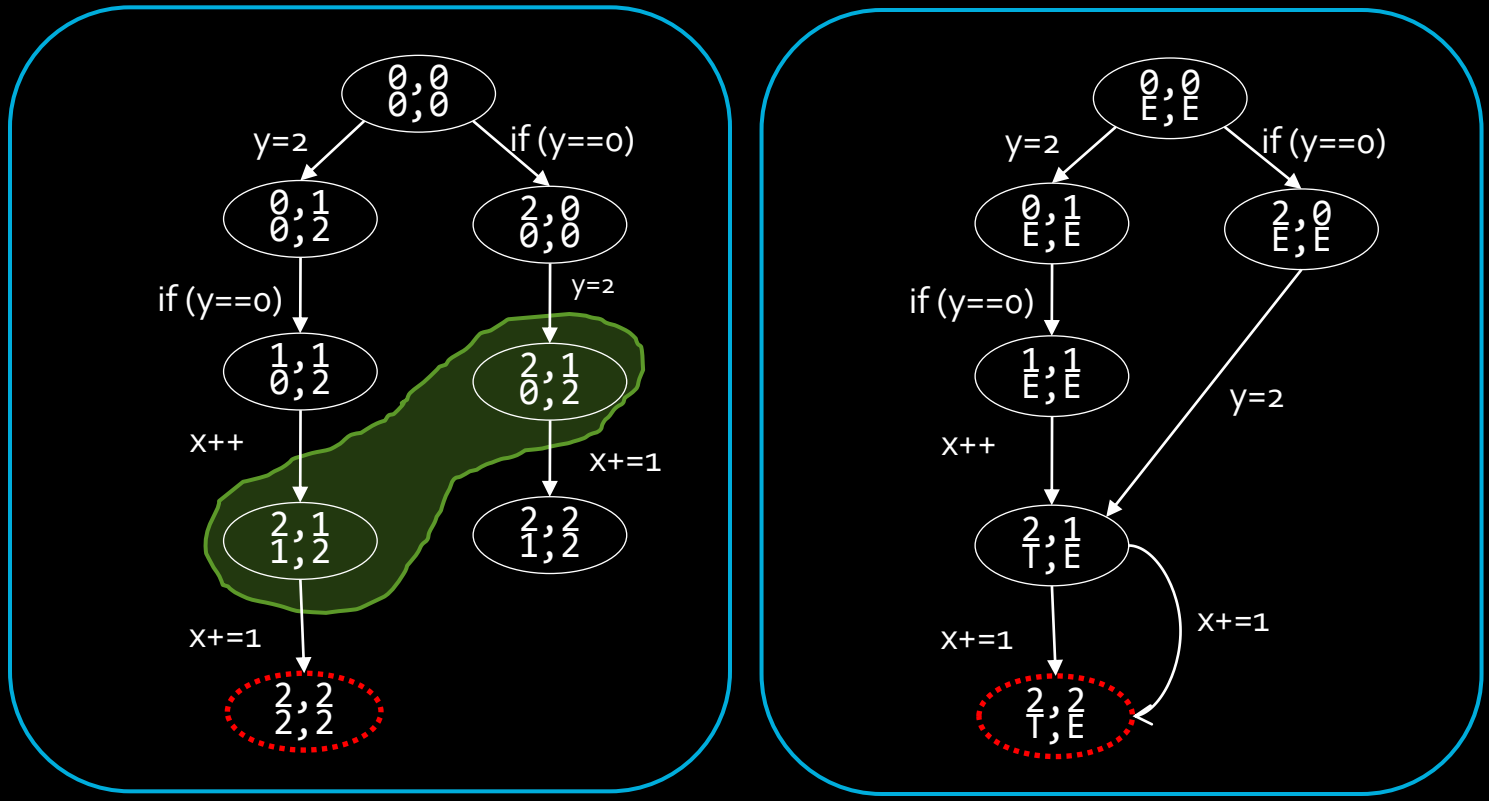
Forward Abstract Interpretation, taking φ into account for pruning infeasible interleavings

Order of selection matters

Backwar… …loration of invalid …vings using …n …riction

Up to this point did not commit to a synchronization mechanism

Choosing a trace to avoid



T1

0: if (y==0) goto L
1: x++
2: L:

||

T2

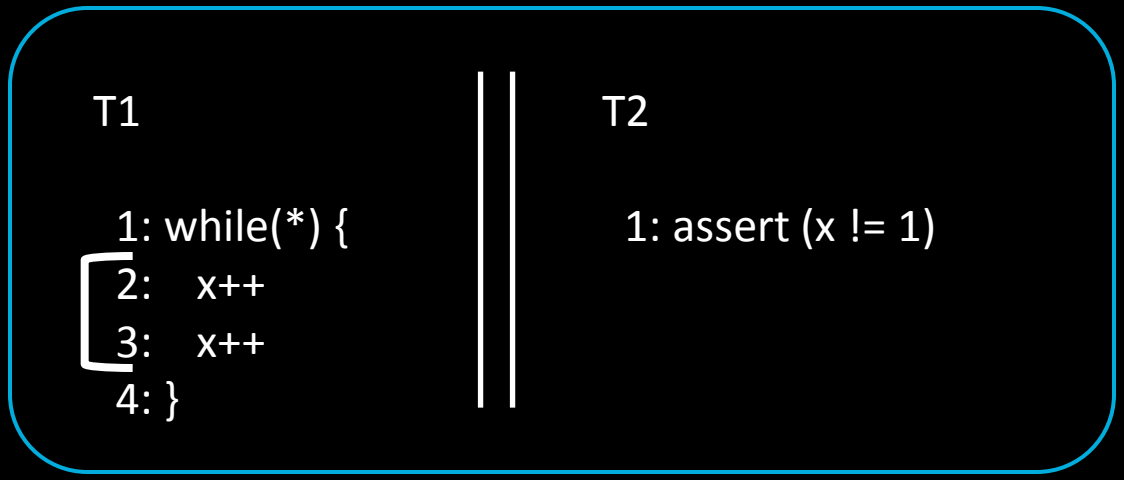
0: y=2
1: x+=1
2: **assert x !=y**

legend

pc1,pc2
x,y

Implementability

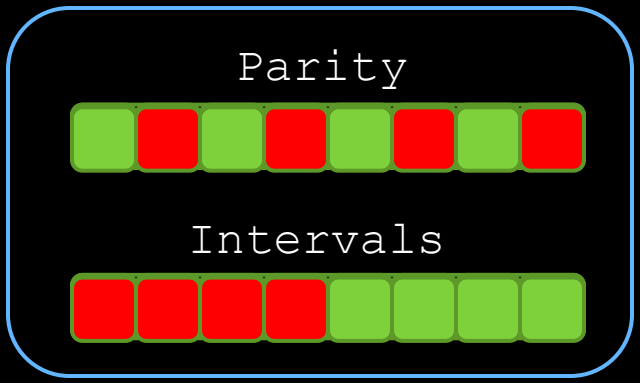
- Separation between schedule constraints and how they are realized
  - Can realize in program: atomic sections, locks,…
  - Can realize in scheduler: benevolent scheduler



- No program transformations (e.g., loop unrolling)
- Memoryless strategy

Atomic sections results

- If we can show disjoint access we can avoid synchronization
- Requires abstractions rich enough to capture access pattern to shared data



Program	Refine Steps	Avoid Steps
Double buffering	1	2
Defragmentation	1	8
3D array update	2	23
Array Removal	1	17
Array Init	1	56

AGS with guarded commands

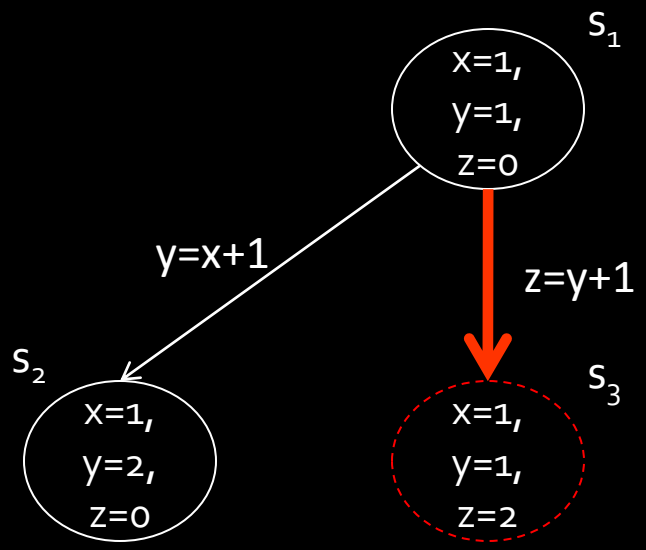
- Implementation mechanism:
  conditional critical region (CCR)

`guard → stmt`

- Constraint language:
  Boolean combinations of equalities over variables (x == c)

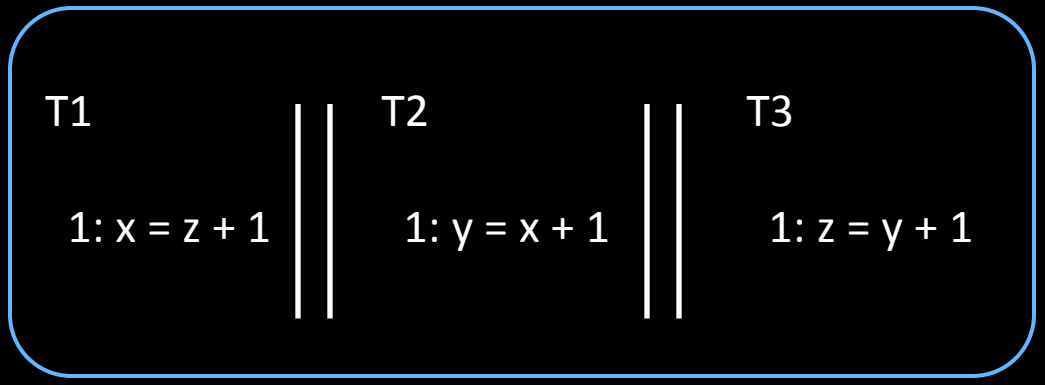
- Abstraction:
  what variables a guard can observe

Avoiding a transition using a guard



- Add guard to `z = y+1` to prevent execution from state s_1
- Guard for s_1 : $(x \neq 1 \vee y \neq 1 \vee z \neq 0)$
- Can affect other transitions where `z = y+1` is executed

Example: full observability



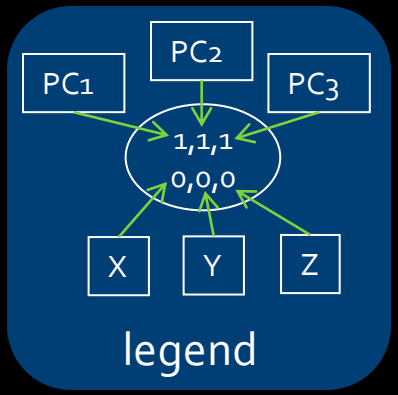
Specification:

- ¬(y = 2 ∧ z = 1)
- No Stuck States

Abstraction:

{ x, y, z }

Build Transition System



1,1,1
0,0,0

x=z+1 | y=x+1 | z=y+1

e,1,1
1,0,0

1,e,1
0,1,0

1,1,e
0,0,1

y=x+1 | z=y+1 | z=y+1 | x=z+1 | x=z+1 | y=x+1

e,e,1
1,2,0

e,1,e
1,0,1

e,e,1
1,1,0

1,e,e
0,1,2

e,1,e
2,0,1

1,e,e
0,1,1

z=y+1 | y=x+1 | z=y+1 | x=z+1 | y=x+1 | x=z+1

e,e,e
1,2,3

e,e,e
1,2,1

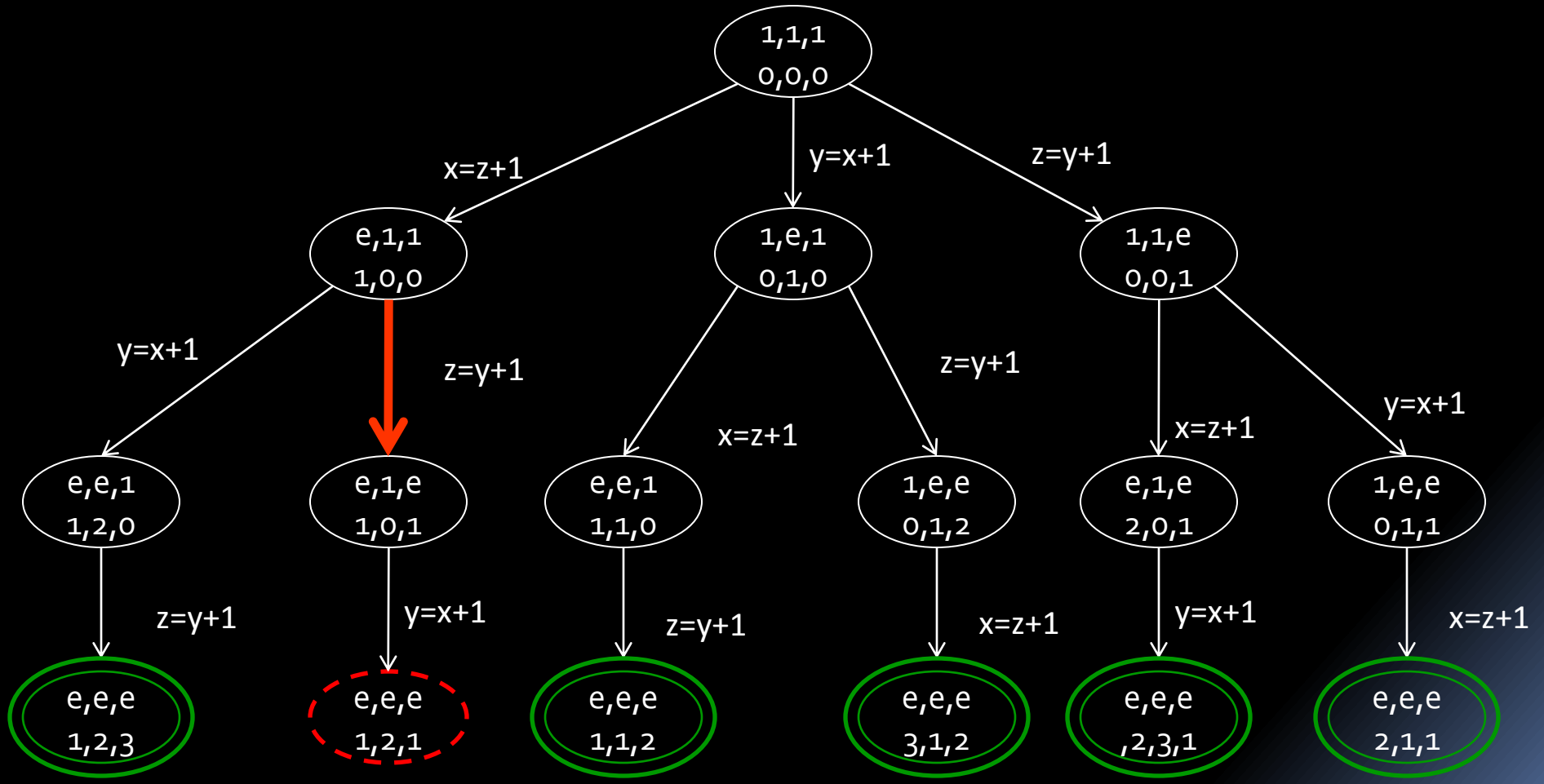
e,e,e
1,1,2

e,e,e
3,1,2

e,e,e
,2,3,1

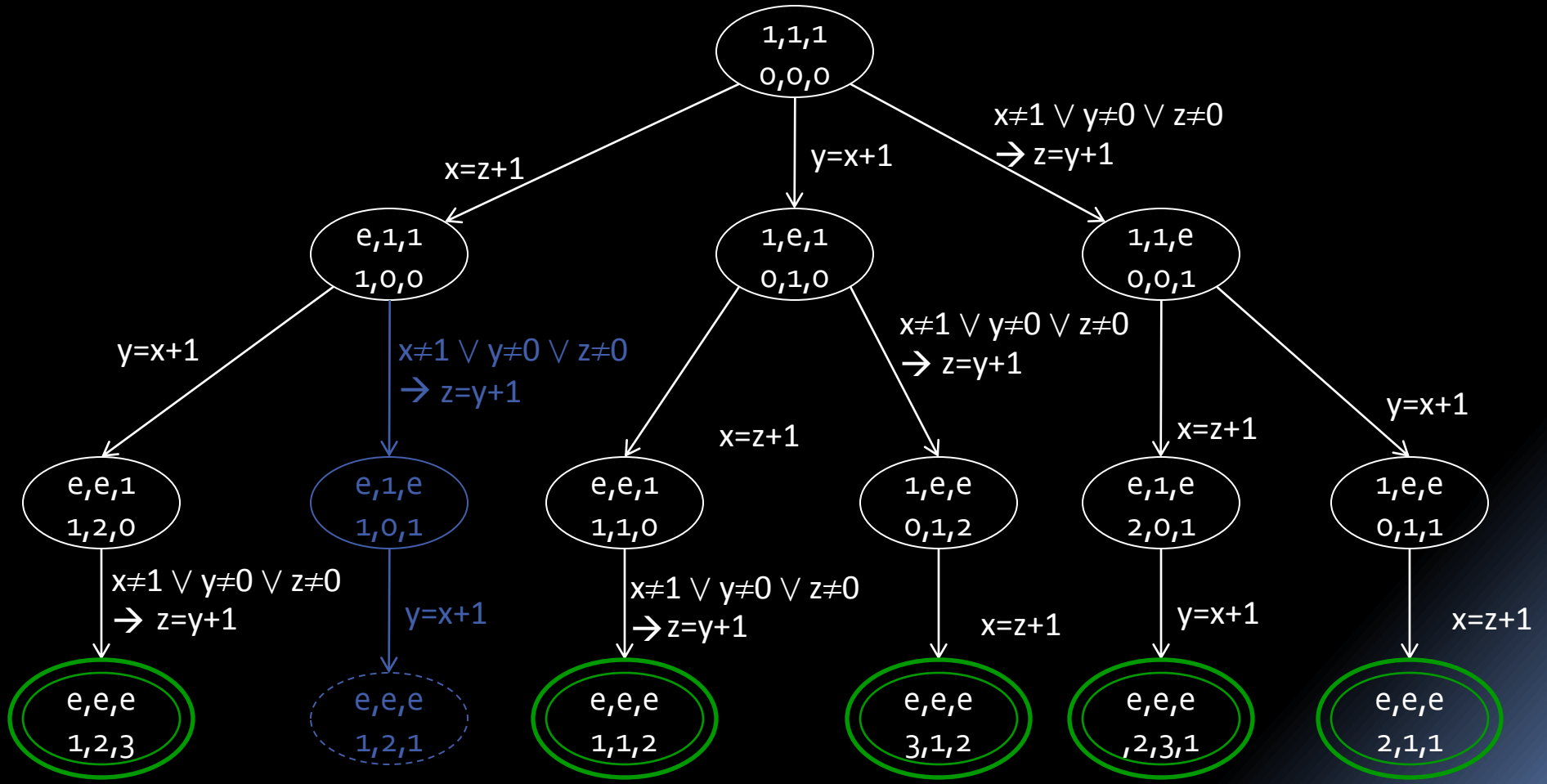
e,e,e
2,1,1

Avoid transition



Result is valid

Correct and Maximally Permissive ✓



Resulting program

T1 || T2 || T3

1: x = z + 1 || 1: y = x + 1 || 1: z = y + 1

Specification:

- ¬(y = 2 ∧ z = 1)
- No Stuck States

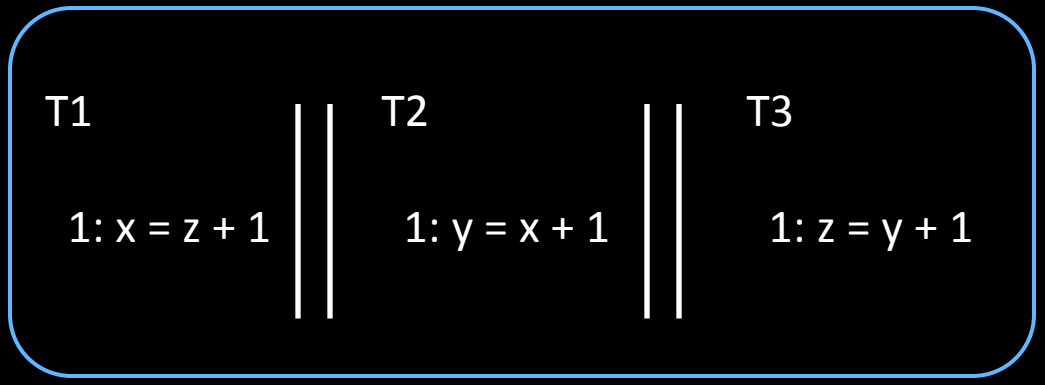
Abstraction:

{ x, y, z }

T1 || T2 || T3

1: x = z + 1 || 1: y = x + 1 || 1: (x ≠ 1 ∨ y ≠ 0 ∨ z ≠ 0) → z = y + 1

Example: limited observability



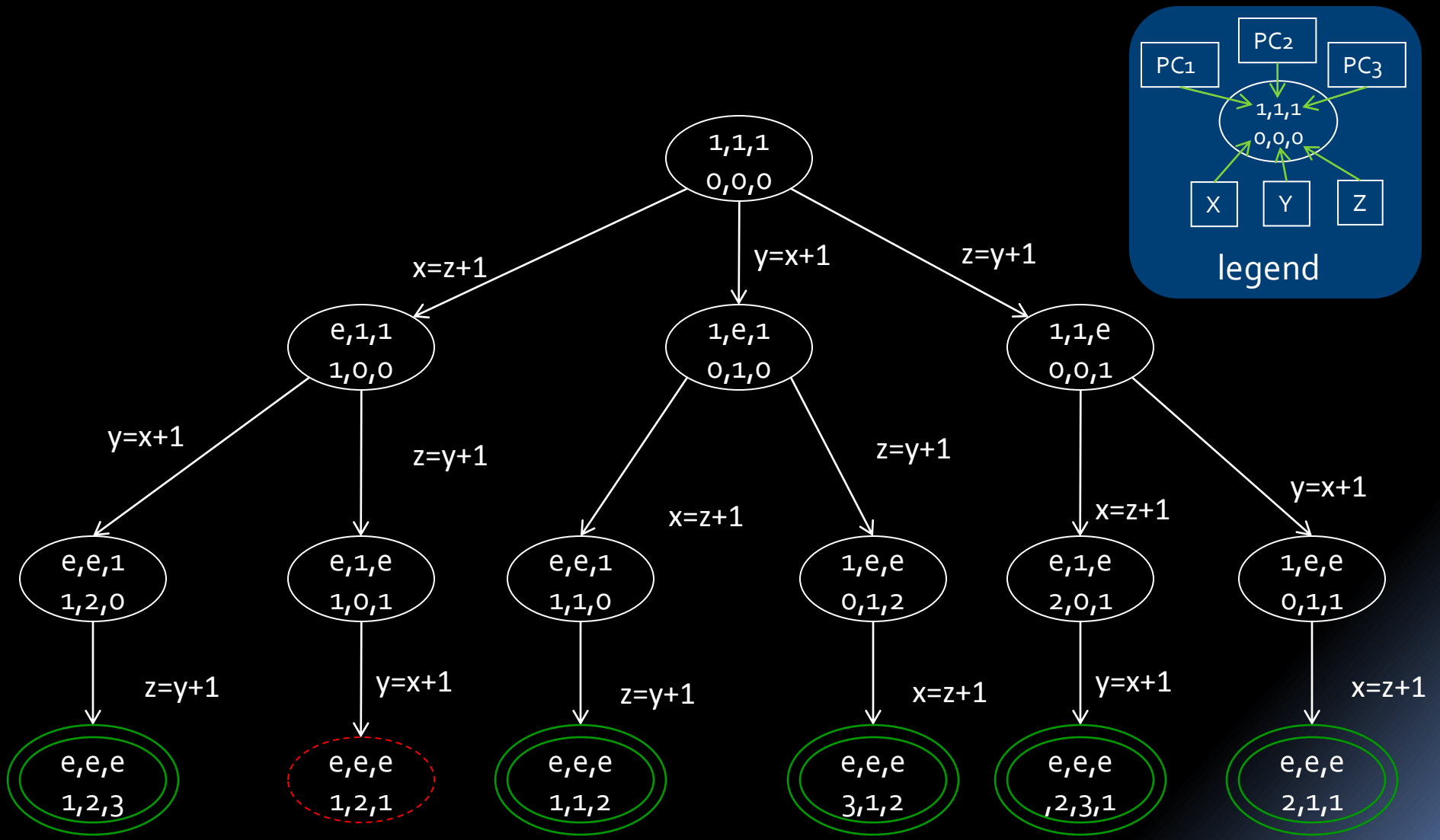
Specification:

- ¬(y = 2 ∧ z = 1)
- No Stuck States

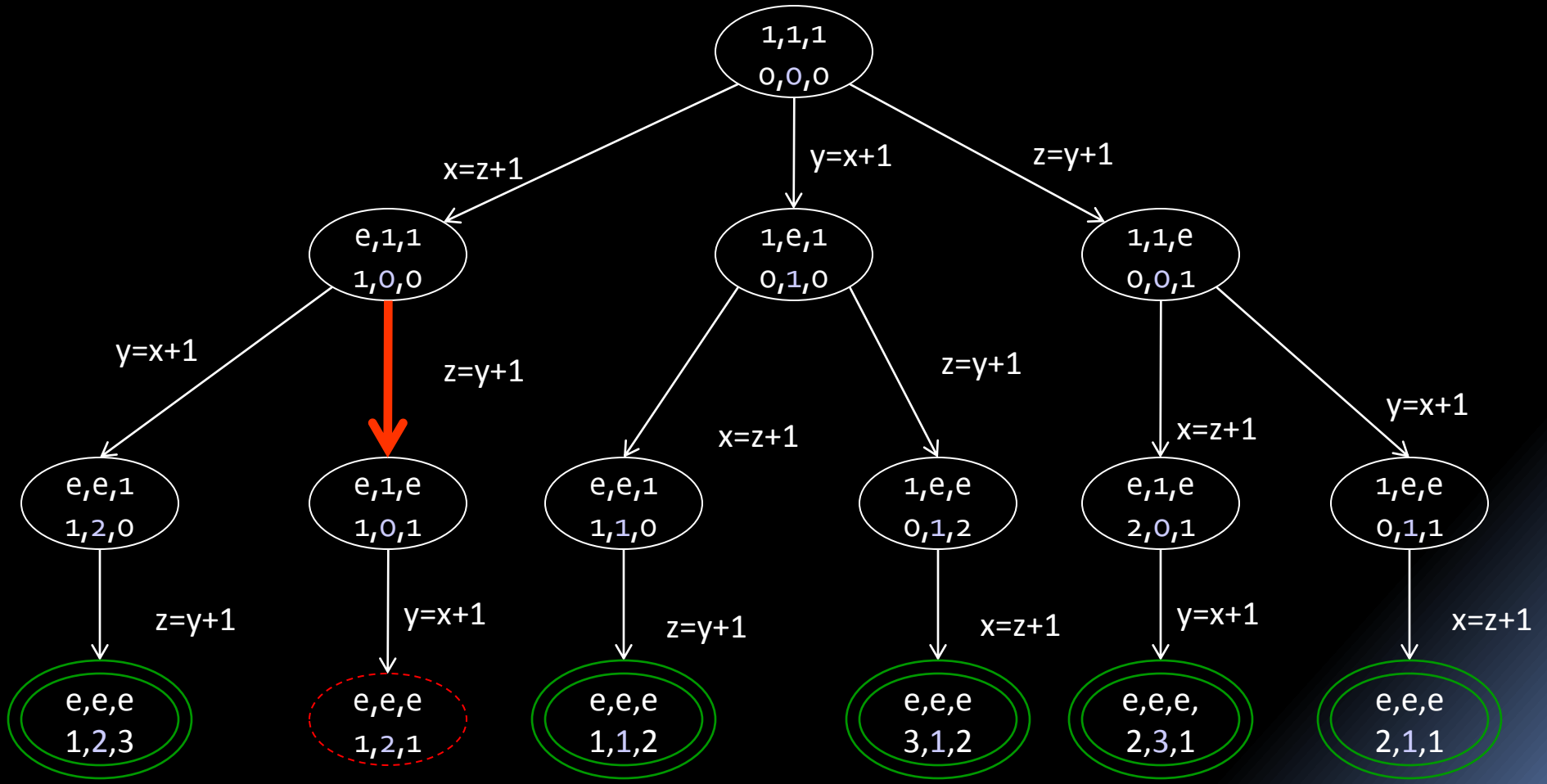
Abstraction:

{ x, z }

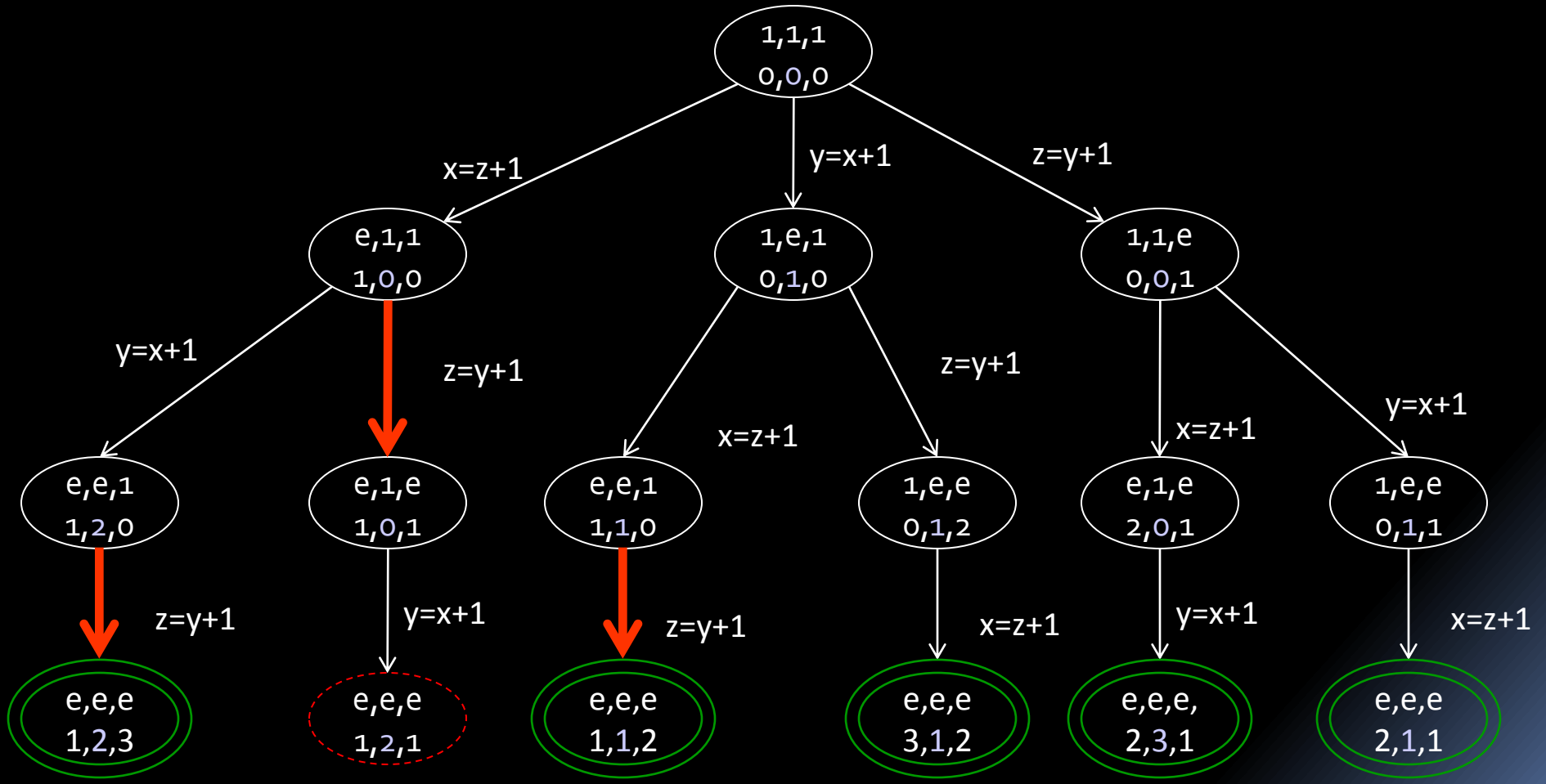
Build transition system



Avoid bad state

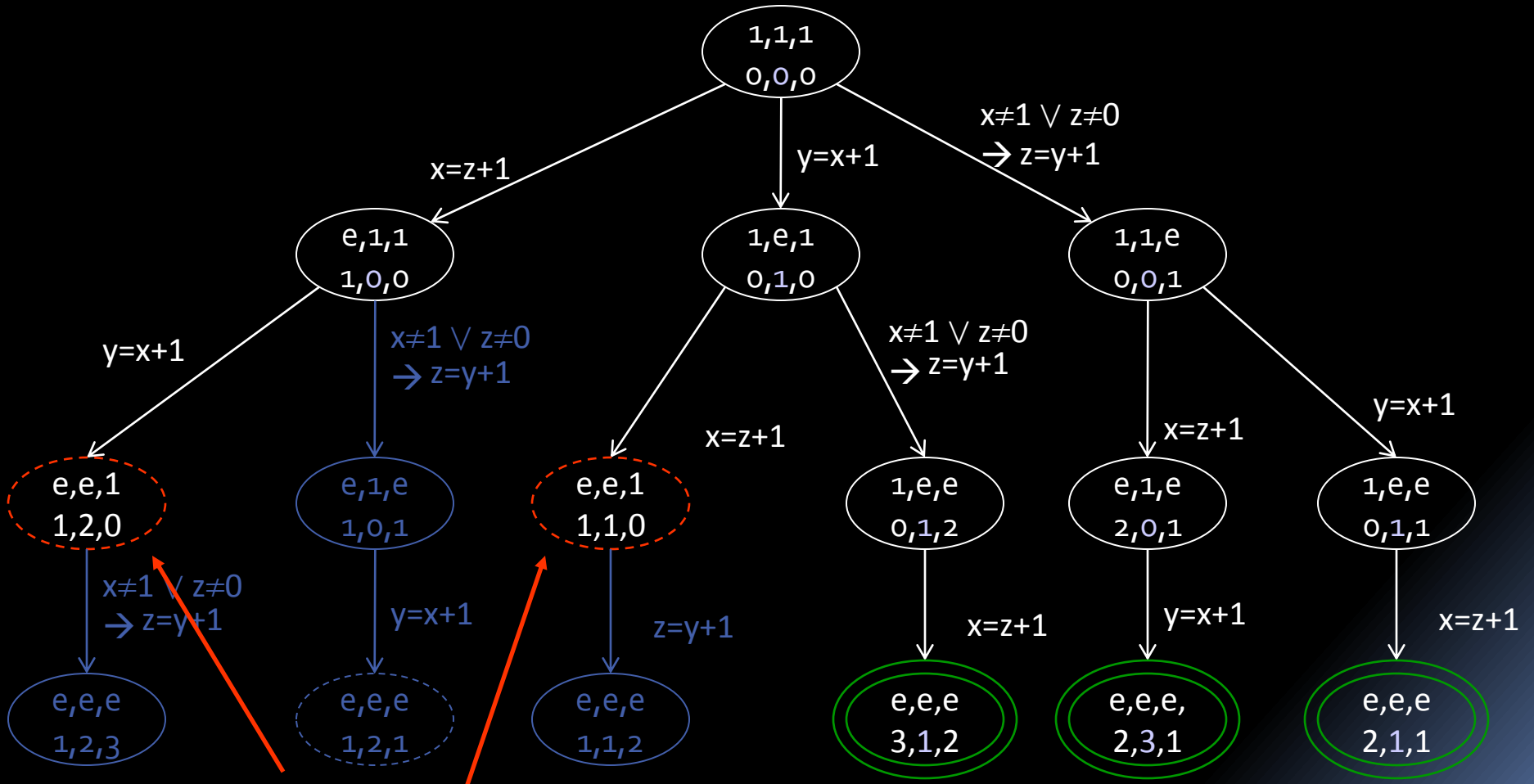


Select all equivalent transitions

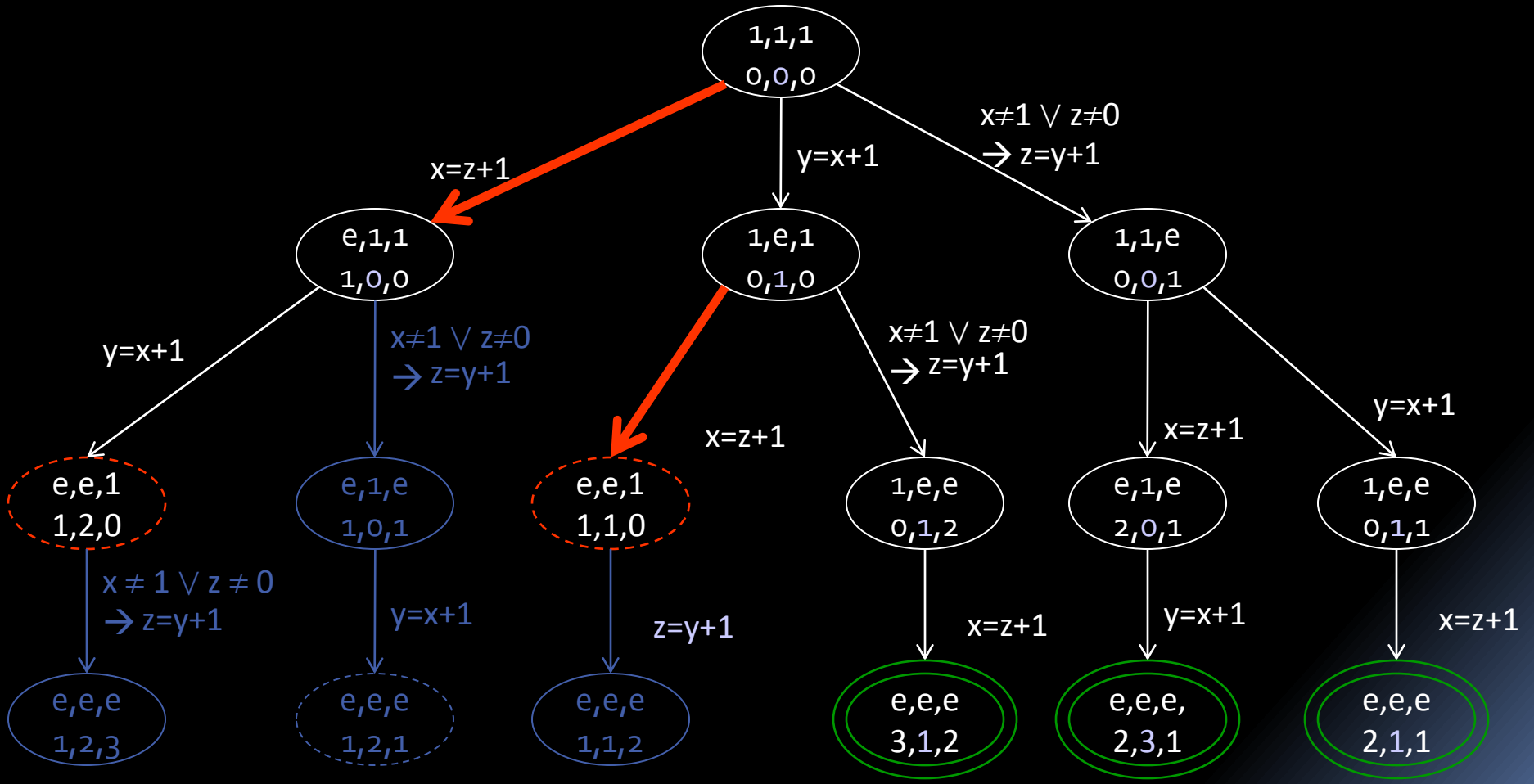


Implementability

Result has stuck states

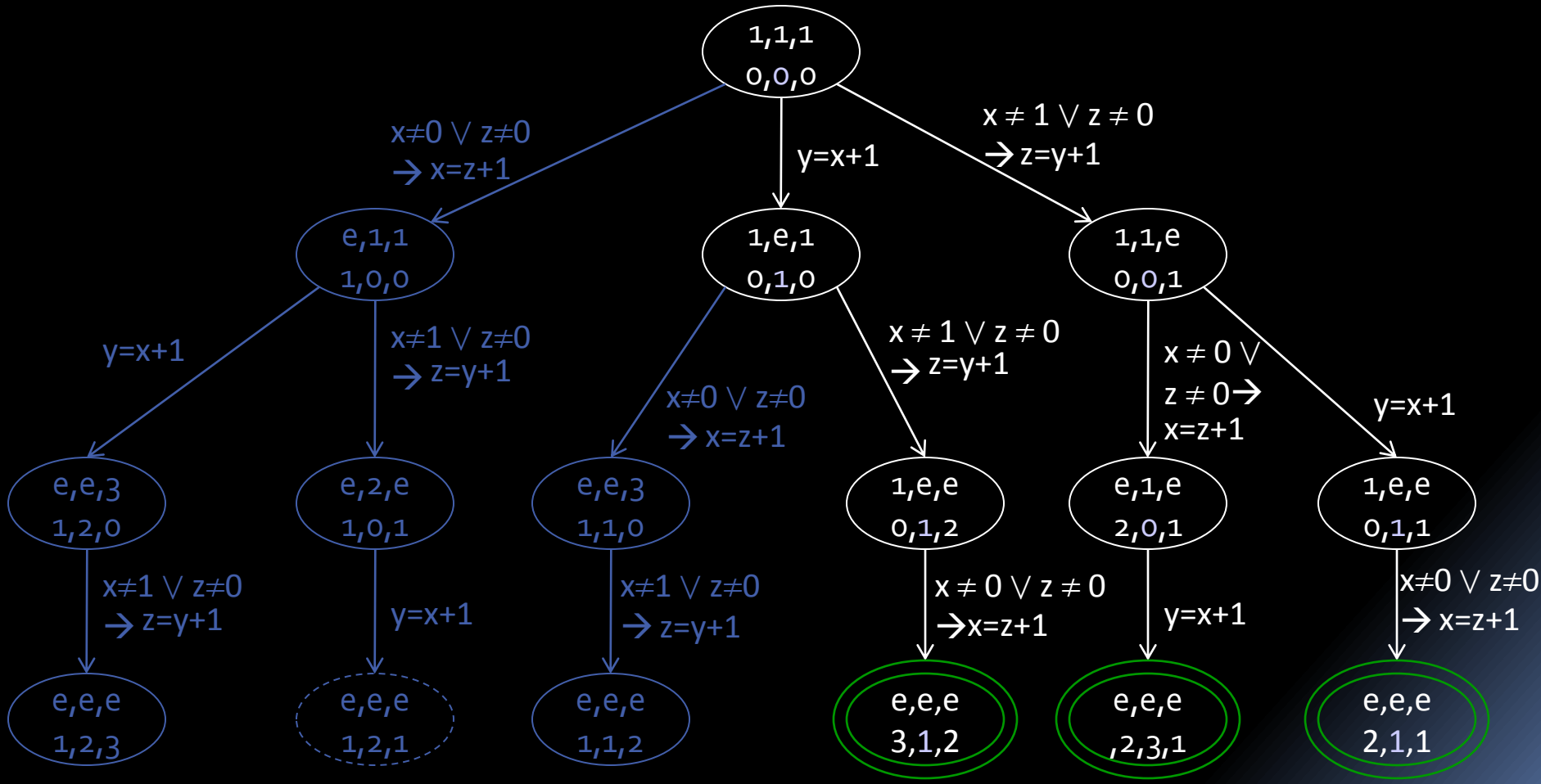


Select transition to remove



Result is valid

Correct and Maximally Permissive ✓



Resulting program

T1

1: x = z + 1

||

T2

1: y = x + 1

||

T3

1: z = y + 1

Specification:

- ! (y = 2 && z = 1)
- No Stuck States

Abstraction:

{ x, z }

T1

1: (x ≠ 0 ∨ z ≠ 0)→
x = z + 1

||

T2

1: y = x + 1

||

T3

1: (x ≠ 1 ∨ z ≠ 0) →
z = y + 1

{ x }

T1
x = z + 1
T2
y = x + 1
T3
z = y + 1

✗

T1
x = z + 1
T2
y = x + 1
T3
(x ≠ 1) → z = y + 1

✗

T1
(x ≠ 0) → x = z + 1
T2
y = x + 1
T3
(x ≠ 1) → z = y + 1

✗

{ x,z }

T1
x = z + 1
T2
y = x + 1
T3
z = y + 1

✗

T1
x = z + 1
T2
y = x + 1
T3
(x ≠ 1 ∨ z ≠ 0) → z = y + 1

✗

T1
(x ≠ 0 ∨ z ≠ 0) → x = z + 1
T2
y = x + 1
T3
(x ≠ 1 ∨ z ≠ 0) → z = y + 1

✓

{ x,y,z }

T1
x = z + 1
T2
y = x + 1
T3
z = y + 1

✗

T1
x = z + 1
T2
y = x + 1
T3
(x ≠ 1 ∨ y ≠ 0 ∨ z ≠ 0) →
z = y + 1

✓

AGS with memory fences

- Avoidable transition more tricky to define
  - Operational semantics of weak memory models
- Special abstraction required to deal with potentially unbounded store-buffers
  - Even for finite-state programs
- Informally "finer abstraction = fewer fences"

Some AGS instances

Avoid	Implementation Mechanism	Abstraction Space (examples)	Reference
Context switch	Atomic sections	Numerical abstractions	[POPL'10]
Transition	Conditional critical regions (CCRs)	Observability	[TACAS'09]
Intra-thread ordering	Memory fences	Partial-Coherence abstractions for Store buffers	[FMCAD'10] [PLDI'11]
inter-thread ordering	Synch barriers	Numerical abstractions	[in progress]
…			

Summary

- An algorithm for Abstraction-Guided Synthesis
  - Synthesize efficient and correct synchronization
  - Handles infinite-state systems based on abstract interpretation
  - Refine the abstraction and/or restrict program behavior
  - Interplay between abstraction and synchronization
- Quantitative Synthesis
  - Separate characterization of solution from choosing optimal solutions (e.g., smallest atomic sections)

Invited Questions

1. What about other synchronization mechanisms?
2. Why not start from the most constrained program and work downwards (relaxing constraints)?
3. Can't you solve the problem purely by brute force search of all atomic section placements?
4. Why are you enumerating traces? Can't you compute your solution via state-based semantics?
5. Why use only a single CEX at a time? Could use information about the whole (abstract) transition system?
6. How is this related to supervisor synthesis?