Lecture 09 – Synthesis of Synchronization

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

Previously

- Predicate abstraction
- Abstraction refinement
  - at a high level

Today

- Synthesis of Synchronization

Verification Challenge

$$P \overset{?}{\models} S$$

With abstraction α

$$P \overset{?}{\models}_{\alpha} S$$

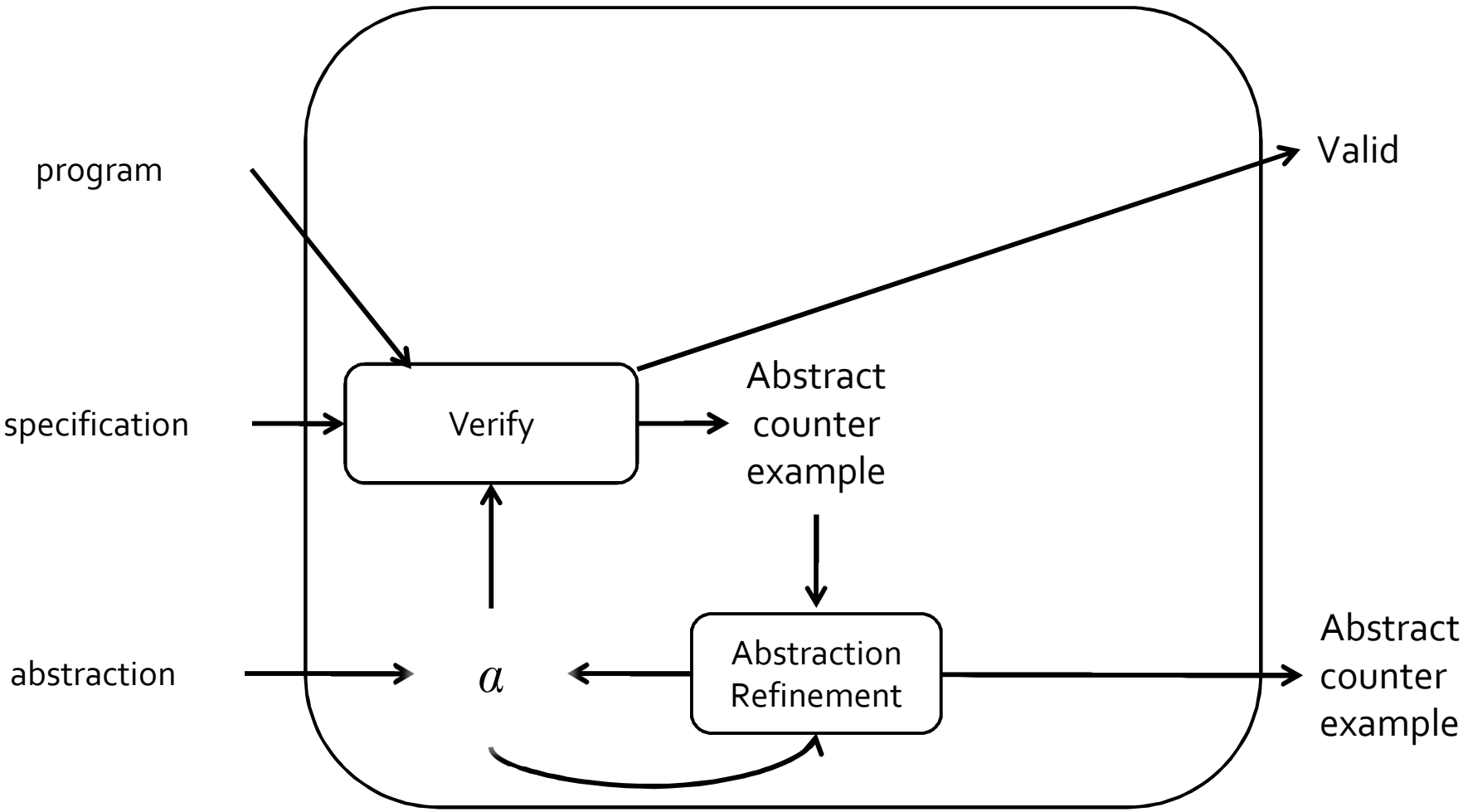
Now what?

$$P \not\models_{\alpha} S$$

$$P \models_{\alpha'} S$$

abstraction refinement

A Standard Approach: Abstraction Refinement



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

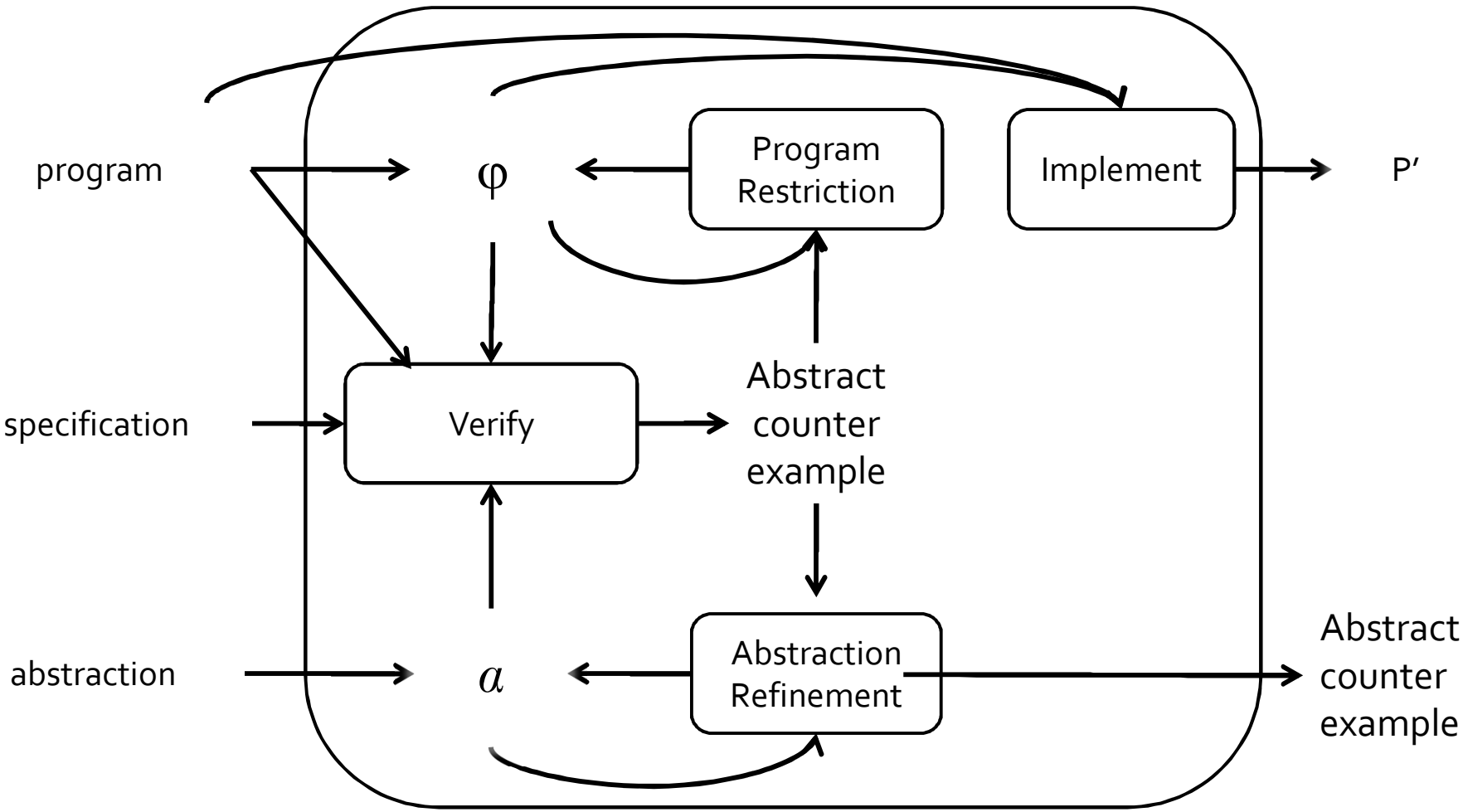
Alternatively…

$$P \not\models_{\alpha} S$$

$$P' \models_{\alpha} S$$

program modification

Abstraction-Guided Synthesis [VYY-POPL'10]



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

Alternatively…

$$P \not\models_{\alpha} S$$

$$P \models_{\alpha} S'$$

change specification (but to what?)

Focus on Program Modification

- Given a program P, specification S, and an abstraction α such that $P \not\models_\alpha S$
- find a modified program P′ such that $P' \models_\alpha S$

- how do you find such a program?
  - very hard in general
  - there is some hope when considering equivalent programs or restricted programs
    - changing a program to add new legal behaviors is hard
  - program restriction natural in concurrent programs
    - reducing permitted schedules

Example

```
int x, y, sign;
x = ?;
if (x<0) {
  sign = -1;
} else {
  sign = 1;
}
y = x/sign;
```

$x \mapsto [-\infty, \infty]$, $sign \mapsto \bot$

$x \mapsto [-\infty, -1]$, $sign \mapsto [-1, -1]$

$x \mapsto [0, \infty]$, $sign \mapsto [1, 1]$

$x \mapsto [-\infty, \infty]$, $sign \mapsto [-1, 1]$

$0 \in [-1,1]$
division by zero
seems possible

$P \not\models_\alpha S$

Specification S: no division by zero.

Abstract domain α: intervals

Example from "The Trace Partitioning Abstract Domain" Rival&Mauborgne

Now what?

- Can use a more refined abstract domain
- Disjunctive completion
- abstract value = set (disjunction) of intervals

```
int x, y, sign;
x = ?;
if (x<0) {
  sign = -1;
} else {
  sign = 1;
}
y = x/sign;
```

$x \mapsto \{[-\infty, \infty]\}$, $sign \mapsto \bot$

$x \mapsto \{[-\infty, -1]\}$, $sign \mapsto \{[-1, -1]\}$

$x \mapsto \{[0, \infty]\}$, sign [illegible] $, 1]\}$

$x \mapsto \{[-\infty, -1],[0, \infty]\}$, $sign \mapsto \{[-1, -1],[1,1]\}$

$0 \notin [-1, -1]$ and $0 \notin [1,1]$
division by zero
impossible

$P \models_{\alpha'} S$

Specification S: no division by zero.

Abstract domain α': set of intervals

Are we done?

- Exponential cost !
- Can we do better?

- Idea: track relationship between x and sign

$$\begin{cases} x < 0 \Rightarrow sign = -1 \\ x >= 0 \Rightarrow sign = 1 \end{cases}$$

- Problem: again, expensive
- Which relationships should I track?
- Domains such as polyhedra won't help (convex)

Trace partitioning abstract domain

- refine abstraction by adding some control history
- won't get into that
- further reading
  - Rival and Mauborgne. The trace partitioning abstract domain. TOPLAS'07.
  - Holley and Rosen. Qualified data flow problems. POPL'80

We can also change the program

```
int x, y, sign;
x = ?;
if (x<0) {
  sign = -1;
} else {
  sign = 1;
}
y = x/sign;
```

P

```
int x, y, sign;
x = ?;
if (x<0) {
  sign = -1;
  y = x/sign;
} else {
  sign = 1;
  y = x/sign;
}
```

P'

Modified Program

```
int x, y, sign;
x = ?;
if (x<0) {
  sign = -1;
  y = x/sign;
} else {
  sign = 1;
  y = x/sign;
}
```

$x \mapsto [-\infty, \infty],\ \text{sign} \mapsto \bot$

$x \mapsto [-\infty, -1],\ \text{sign} \mapsto [-1, -1]$

$0 \notin [-1, -1]$
division by zero impossible

$x \mapsto [0, \infty],\ \text{sign} \mapsto [1, 1]$

$0 \notin [1, 1]$
division by zero impossible

$x \mapsto [-\infty, \infty],\ \text{sign} \mapsto [-1, 1]$

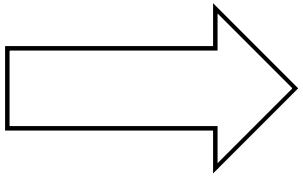
Specification S: no division by zero.

Abstract domain α: intervals

Another Example

```
while(e) {
  s;
}
...
```

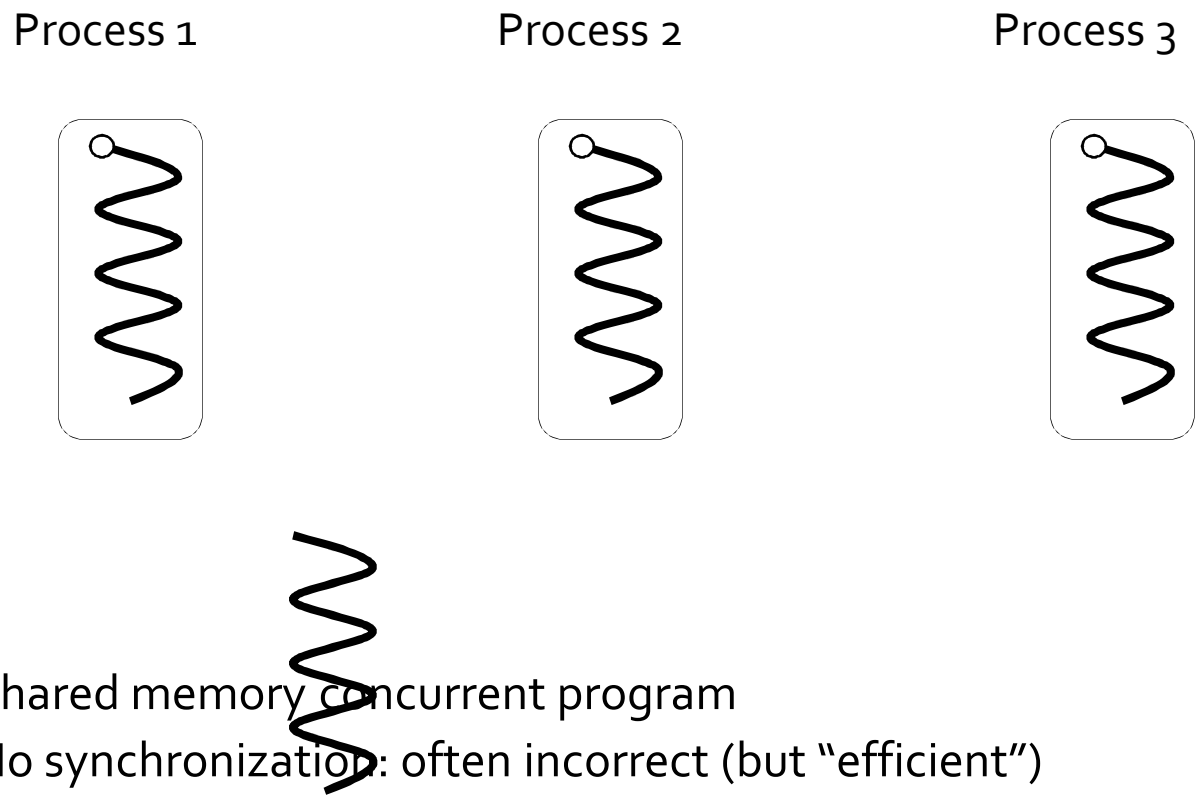
P



```
if (e) {
  s;
  while(e) {
    s;
  }
}
...
```

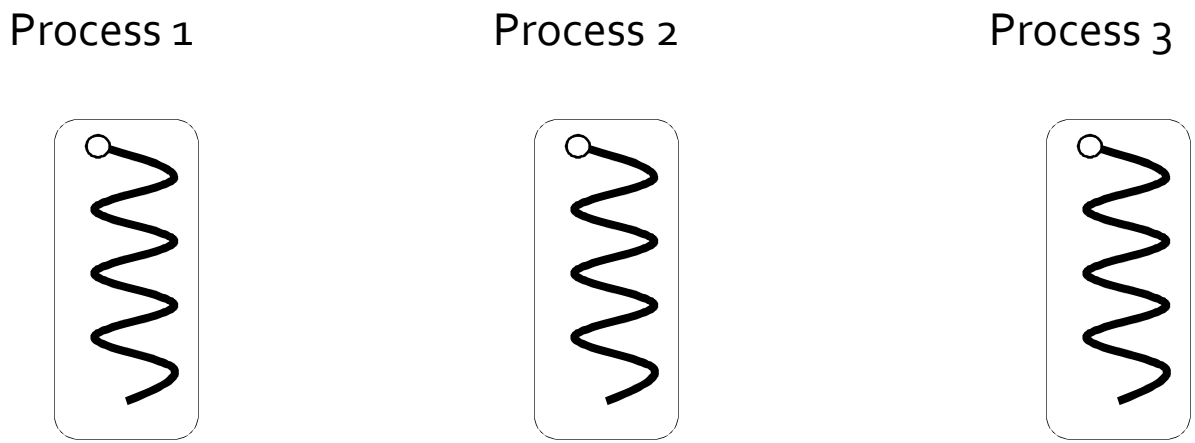
P′

Challenge: Correct and Efficient Synchronization



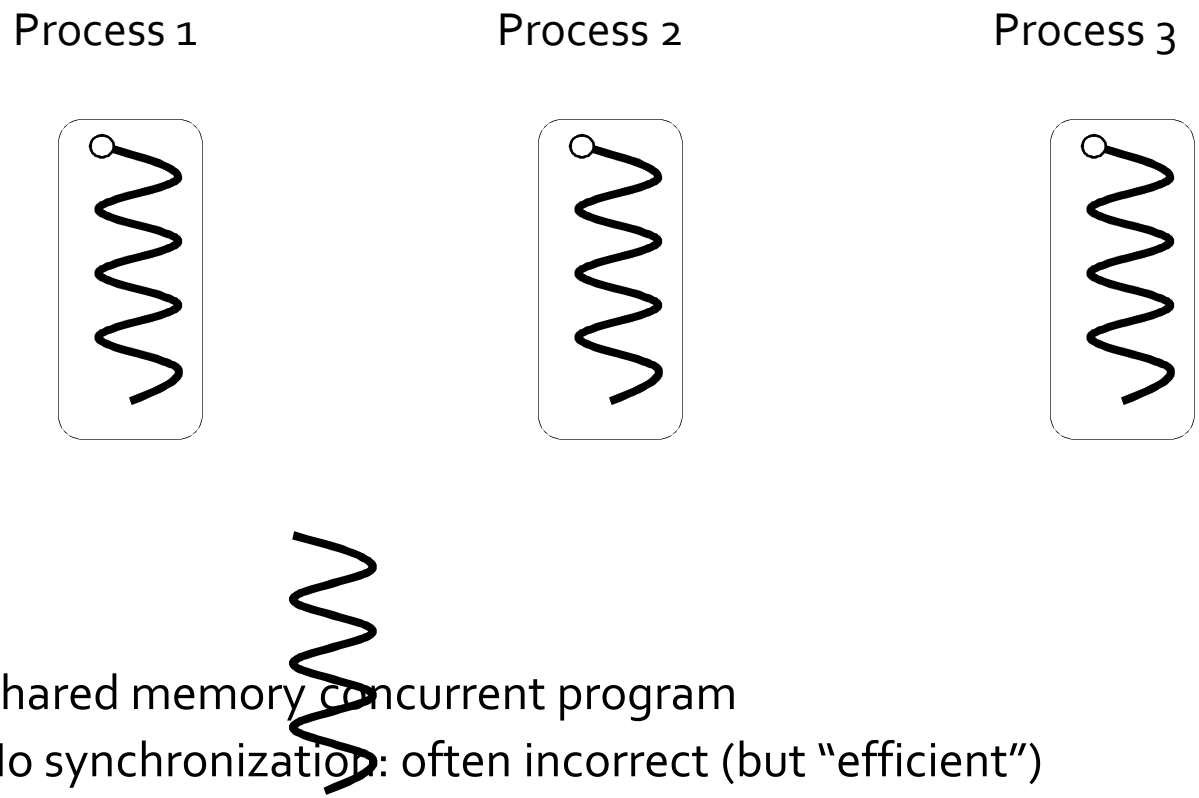
- Shared memory concurrent program
- No synchronization: often incorrect (but "efficient")
- Coarse-grained synchronization: easy to reason about, often inefficient
- Fine-grained synchronization: hard to reason about, programmer often gets this wrong

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

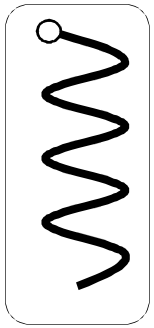
Challenge: Correct and Efficient Synchronization



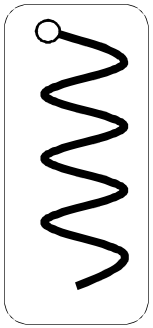
- Shared memory concurrent program
- No synchronization: often incorrect (but "efficient")
- Coarse-grained synchronization: easy to reason about, often inefficient
- Fine-grained synchronization: hard to reason about, programmer often gets this wrong

Challenge

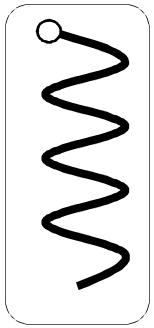
Process 1



Process 2



Process 3

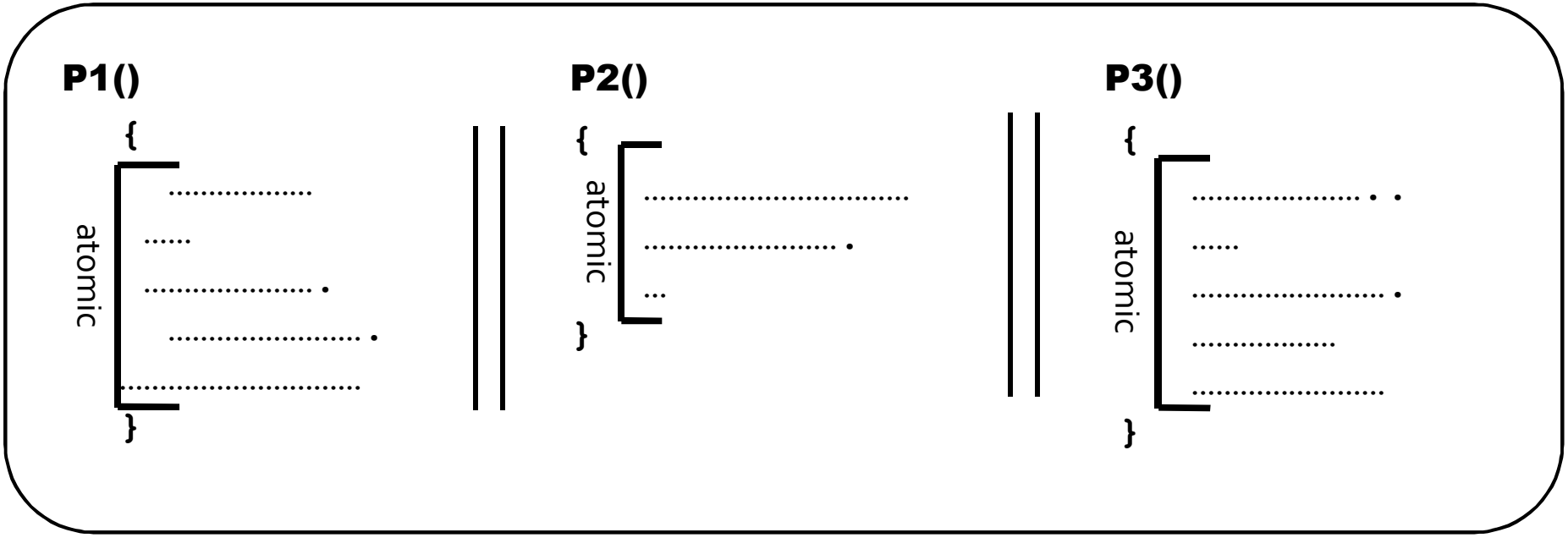


How to synchronize processes to achieve correctness and efficiency?

Synchronization Primitives

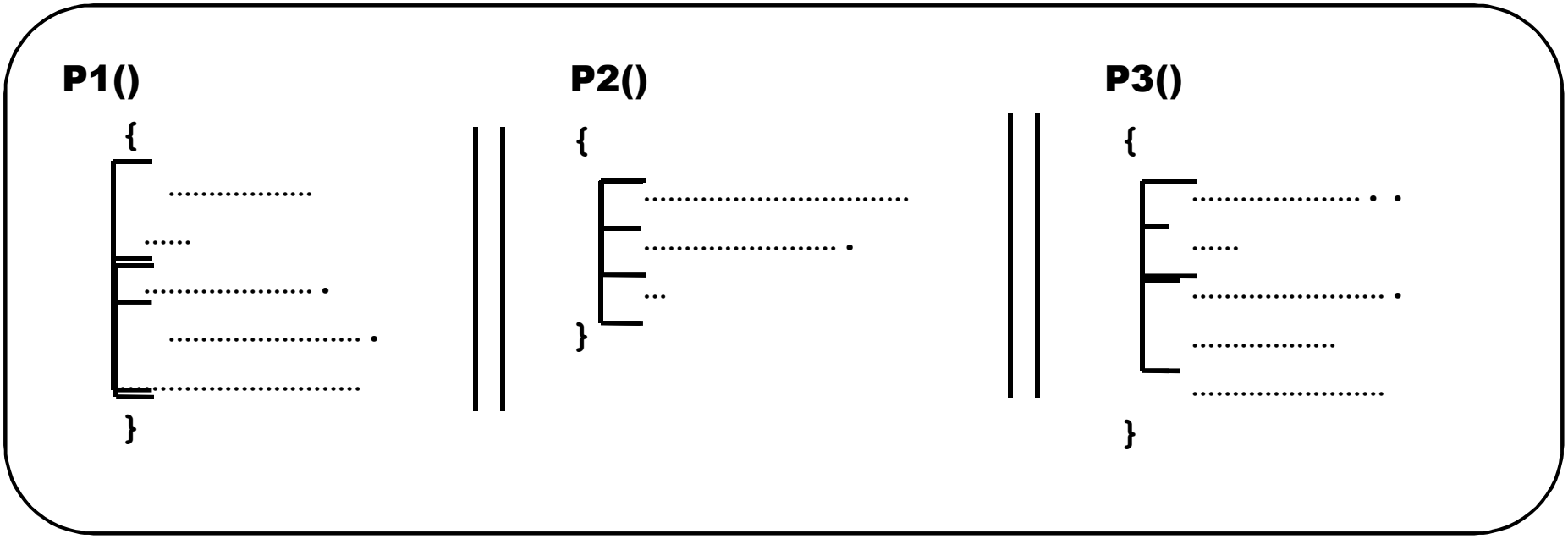
- Atomic sections
- Conditional critical region (CCR)
- Memory barriers (fences)
- CAS
- Semaphores
- Monitors
- Locks
- ....

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

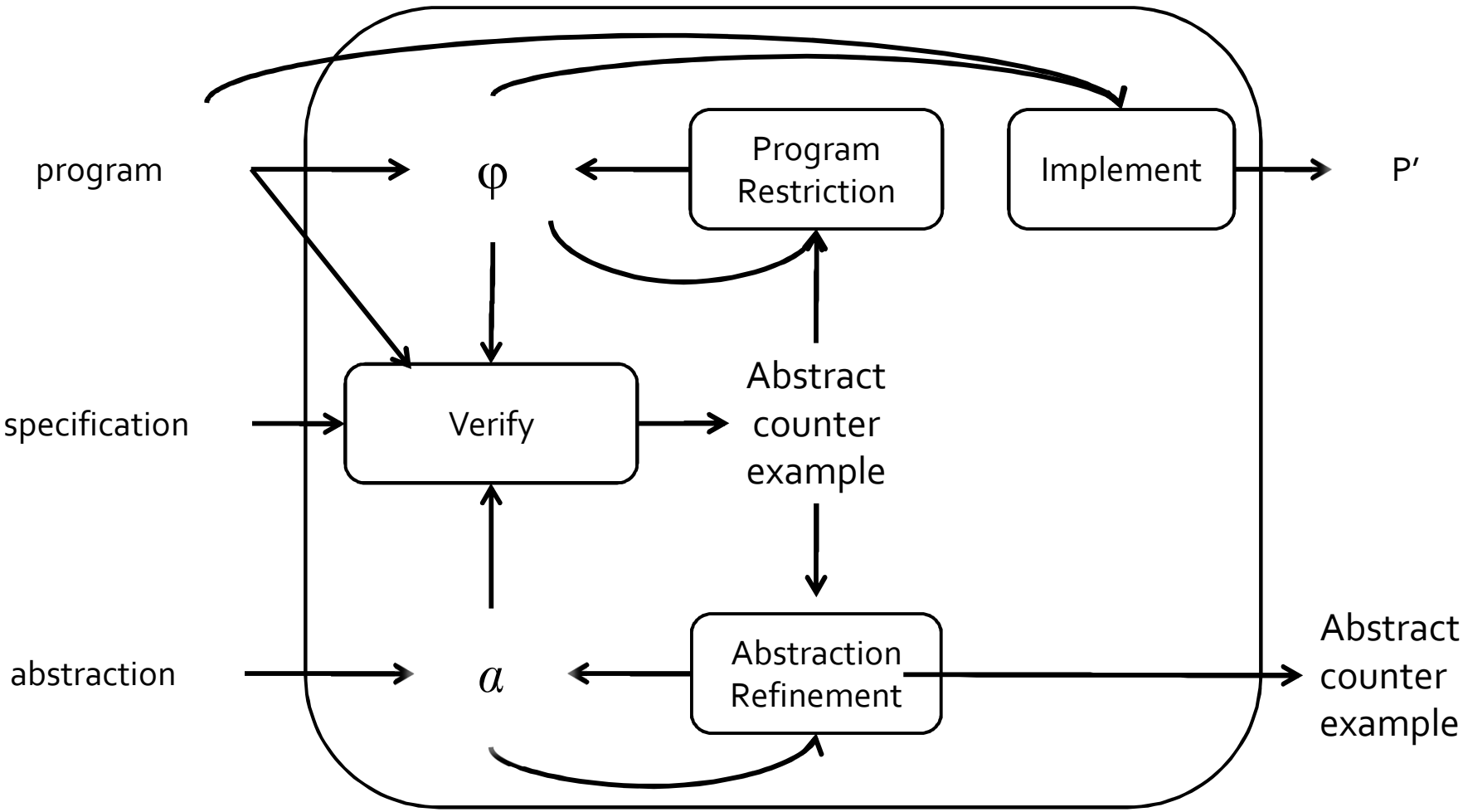
- Find minimal synchronization that makes the program satisfy the specification
  - Avoid all bad interleavings while permitting as many good interleavings as possible
- Assumption: we can prove that serial executions satisfy the specification
  - Interested in bad behaviors due to concurrency
- Handle infinite-state programs

Abstraction-Guided Synthesis of Synchronization

- Synthesis of synchronization via 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

Abstraction-Guided Synthesis



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

AGS 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 + abstractions in progress]
  - …

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

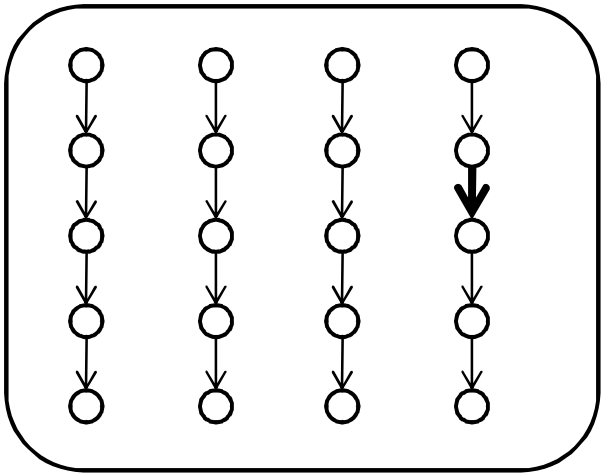
Thread A	Thread B
A1	B1
A2	B2

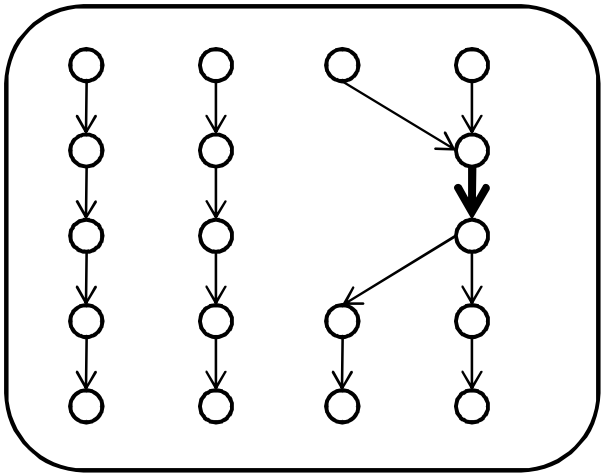
$\pi = A_1 B_1 A_2 B_2$

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

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



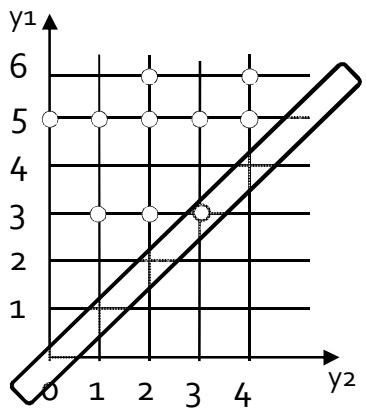


Example

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

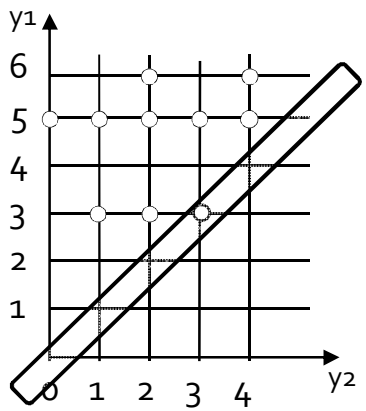
x += z; x += z; z++;z++;y1=f(x);y2=x;assert ➔ y1=5,y2=0

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

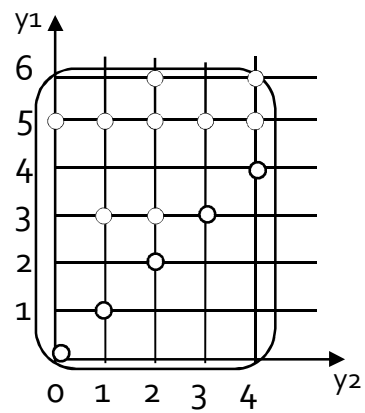
⋮

```
T1                T2                T3                    f(x)  {
                                                            if (x == 1) return 3
 1: x += z         1: z++            1: y1 = f(x)          else if (x == 2) return 6
 2: x += z         2: z++            2: y2 = x             else return 5
                                     3: assert(y1 != y2)   }
```

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

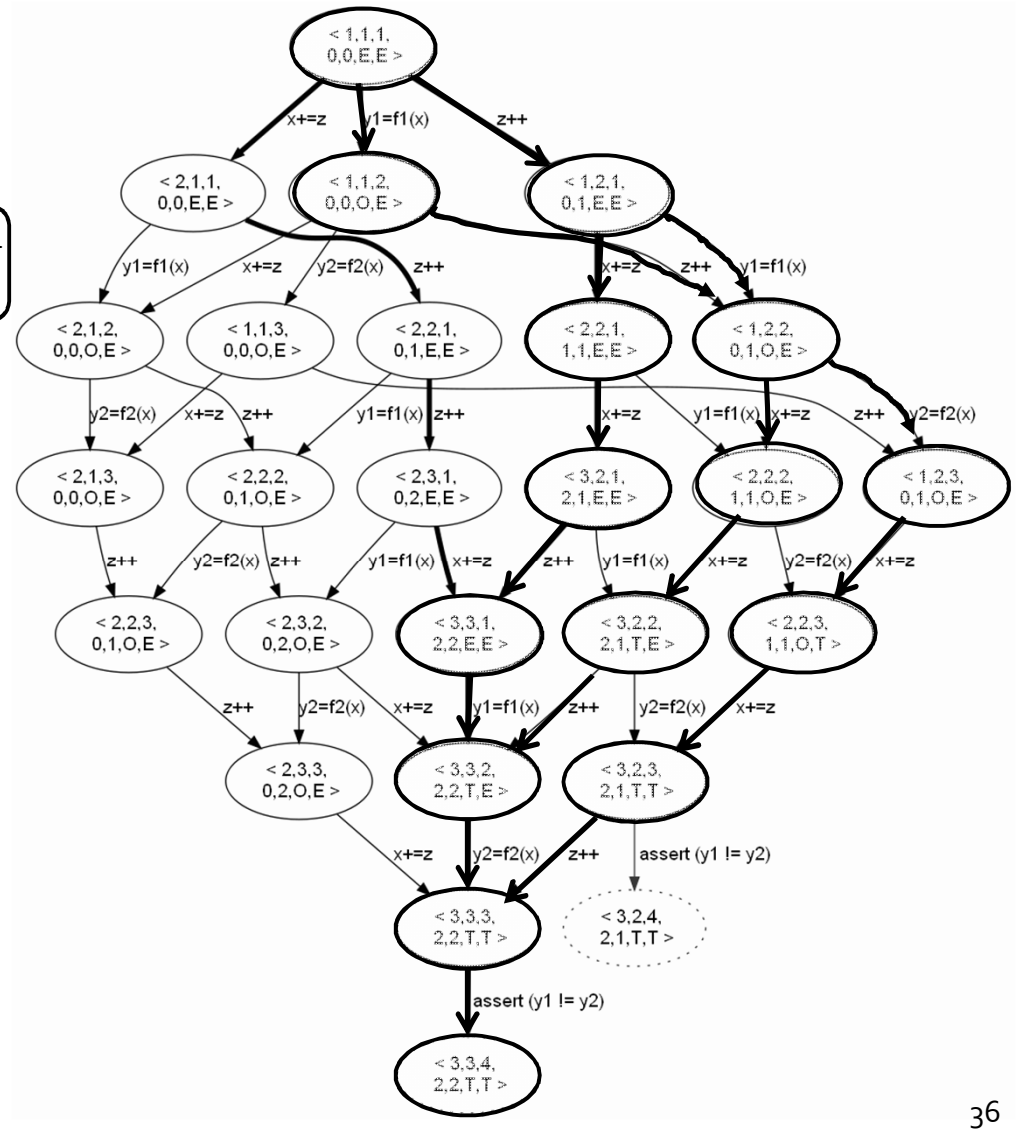
```
T1          ||  T2          ||  T3                      f(x)  {
                                                          if (x == 1) return 3
 1: x += z      1: z++          1: y1 = f(x)             else if (x == 2) return 6
 2: x += z      2: z++          2: y2 = x                else return 5
                                3: assert(y1 != y2)     }
```

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

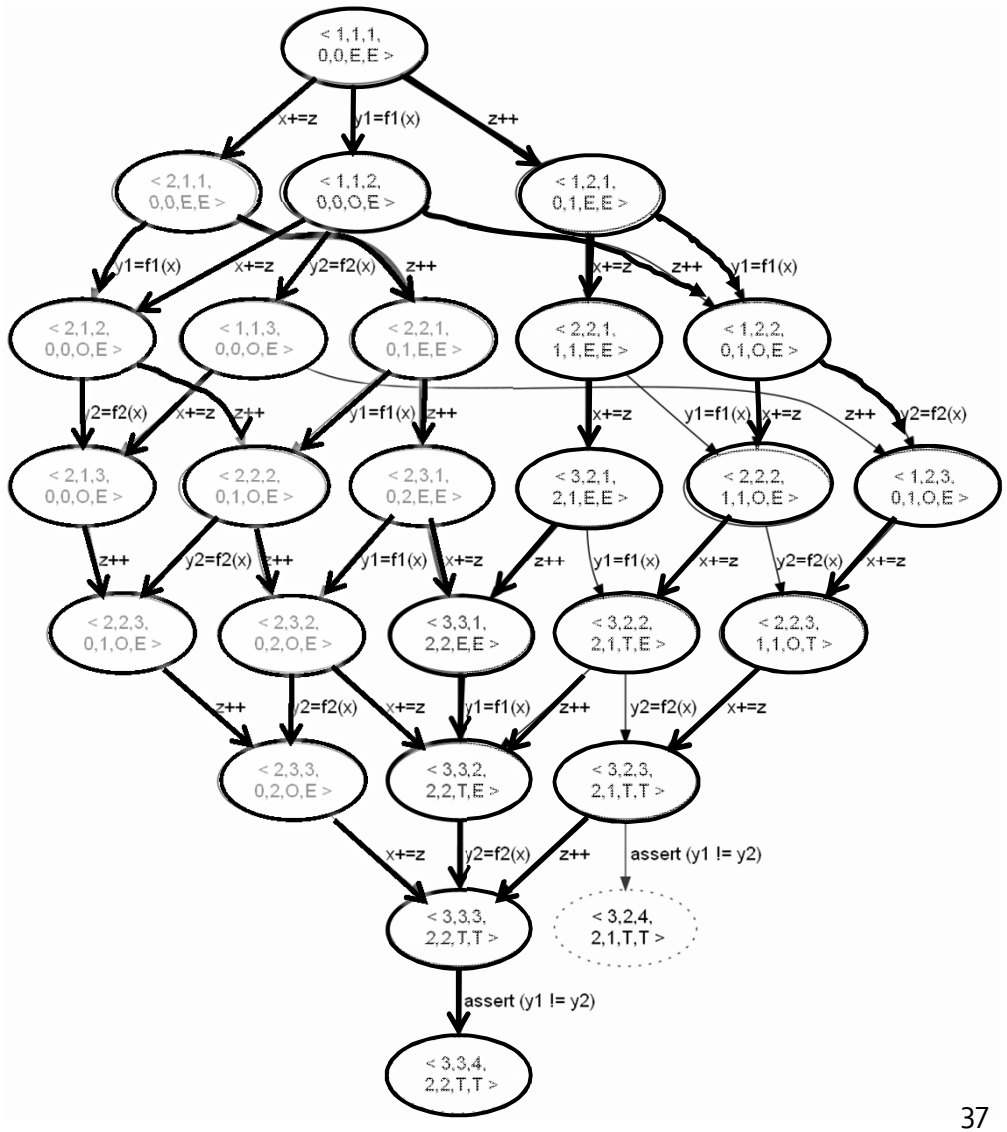


Example: Avoiding Bad Interleavings

```
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)
  }
}
```

φ = [z++,z++]∧[x+=z,x+=z]

T1

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

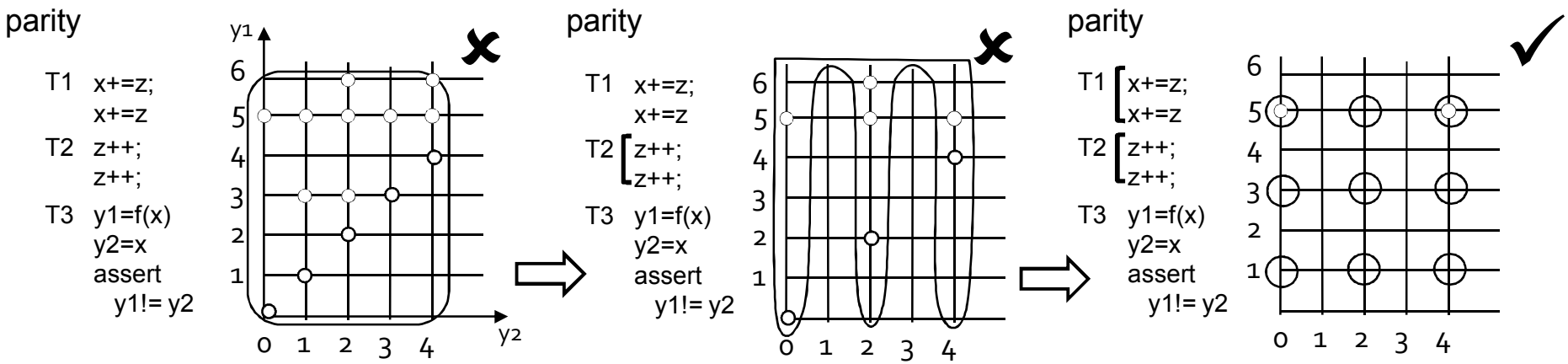
T2

1: z++
2: z++

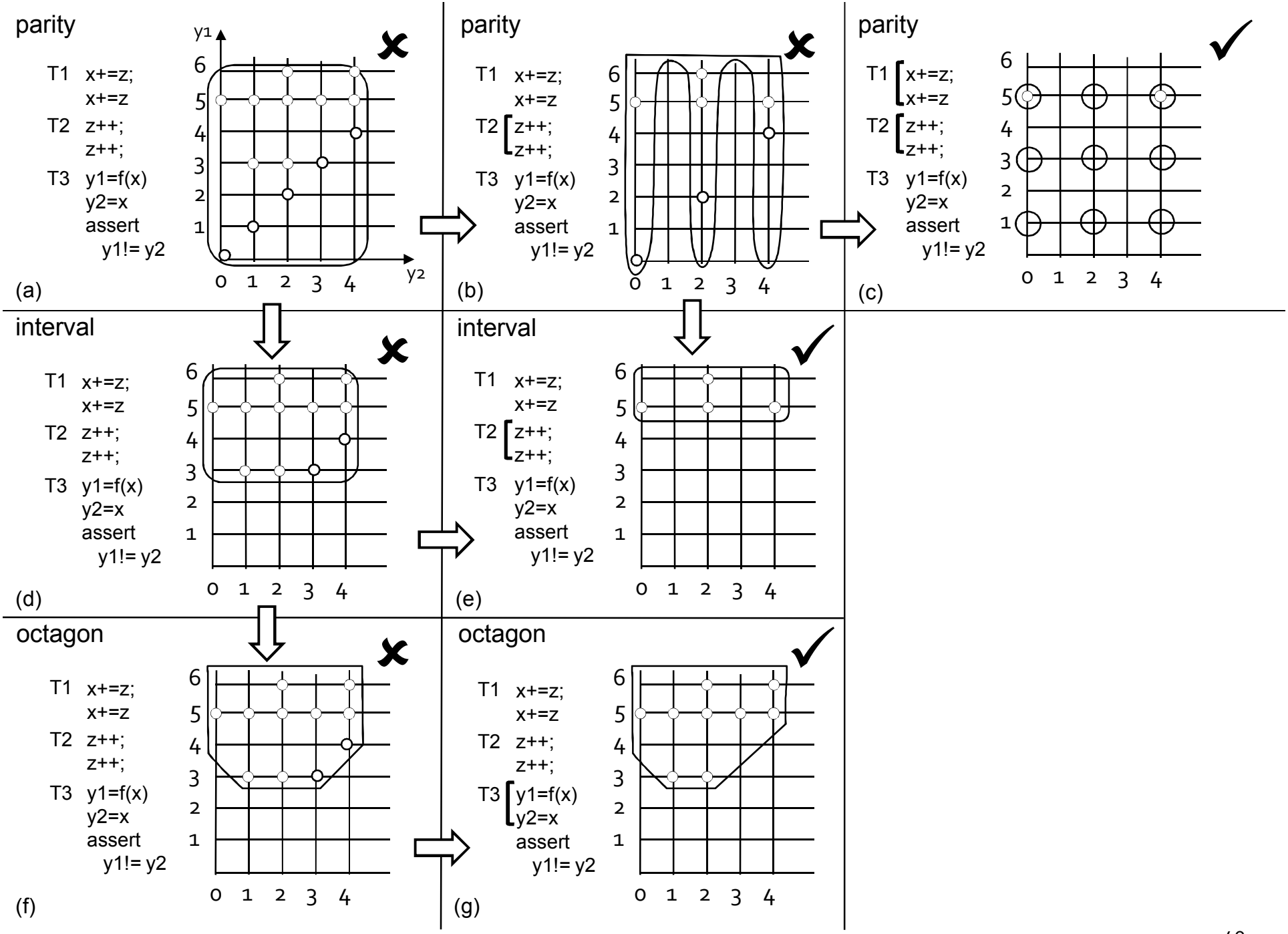
T3

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

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
y1
y2
(a)
(b)
(c)
interval
(d)
(e)
octagon
(f)
(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 Algorithm – More Details

Input: Program P, Specification S

Output: Program P′ satisfying S

```
φ = true
while(true) {
  BadTraces = {π | π ∈ ([[P]]_a ∩ [[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 {
    a′ = refine(a, π)
    if (a′≠ a) a = a′
      else abort
  }
}
```

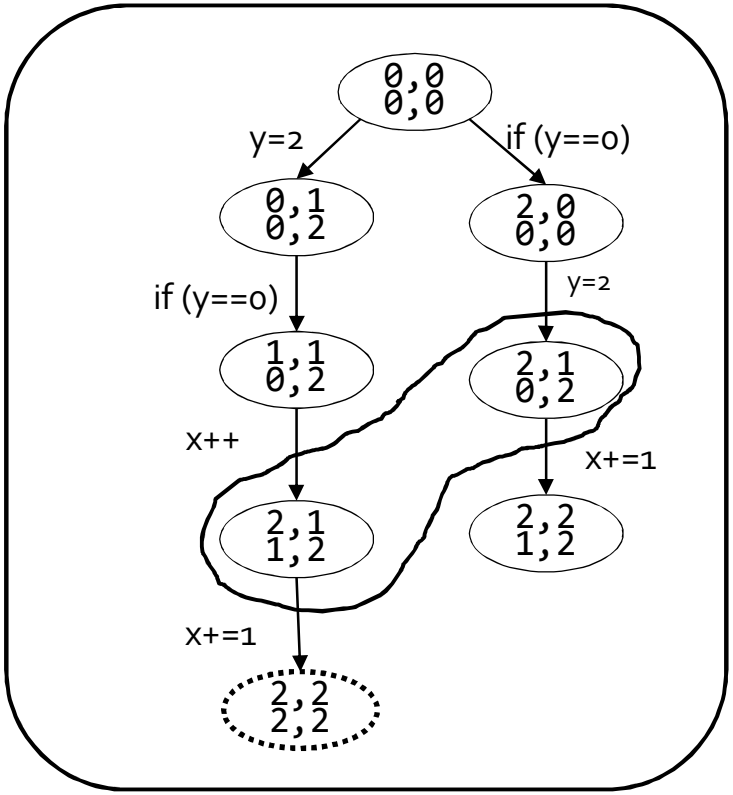
Forward Abstract Interpretation, taking φ into account for pruning infeasible interleavings

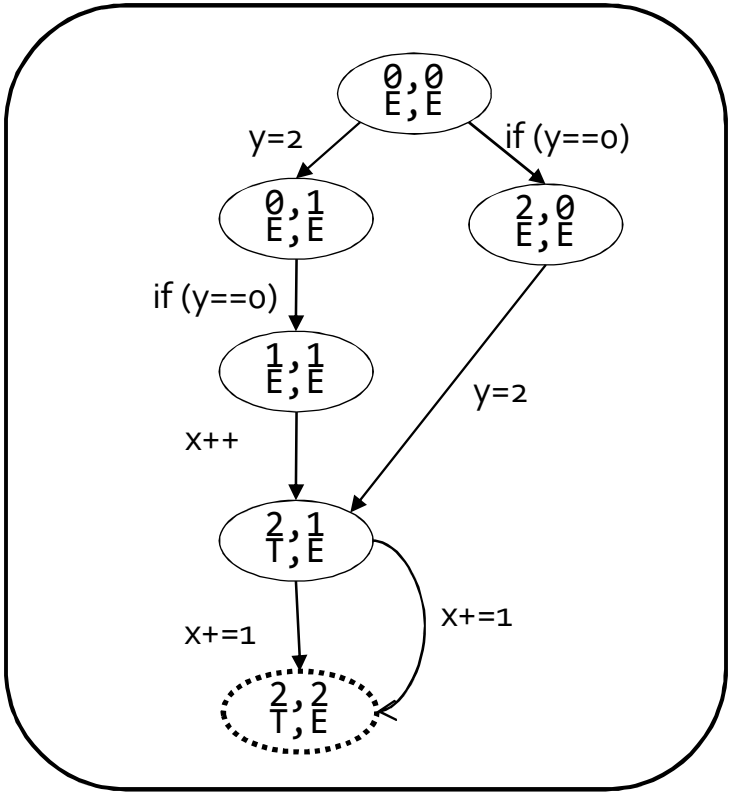
Order of selection matters

Backwar loration of invalid vings using

Up to this point did not commit to a synchronization mechanism

Choosing a trace to avoid





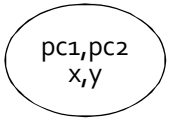
T1

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

T2

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

legend



Implementability

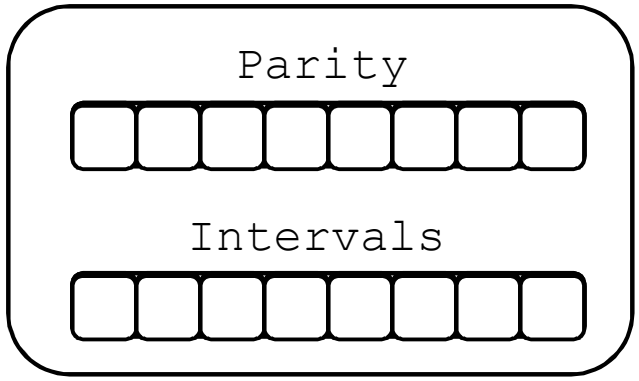
- Separation between schedule constraints and how they are realized
  - Can realize in program: atomic sections, locks,…
  - Can realize in scheduler: benevolent scheduler

T1	T2
1: while(*) {	1: assert (x != 1)
2: x++	
3: x++	
4: }	

- 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

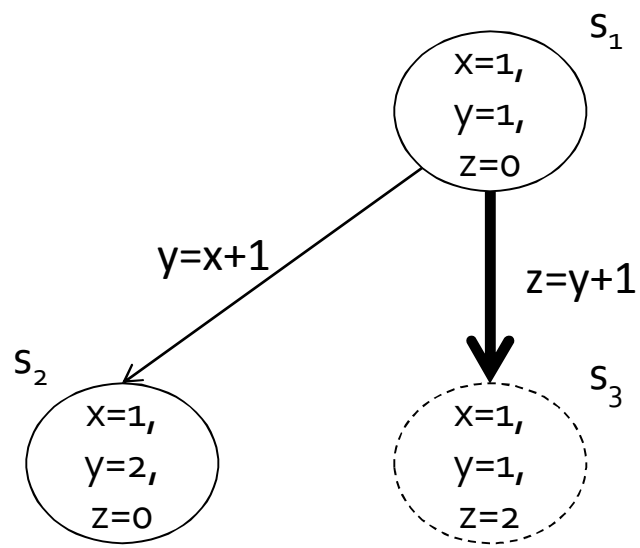
- Implementation mechanism:
  conditional critical region (CCR)

`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

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

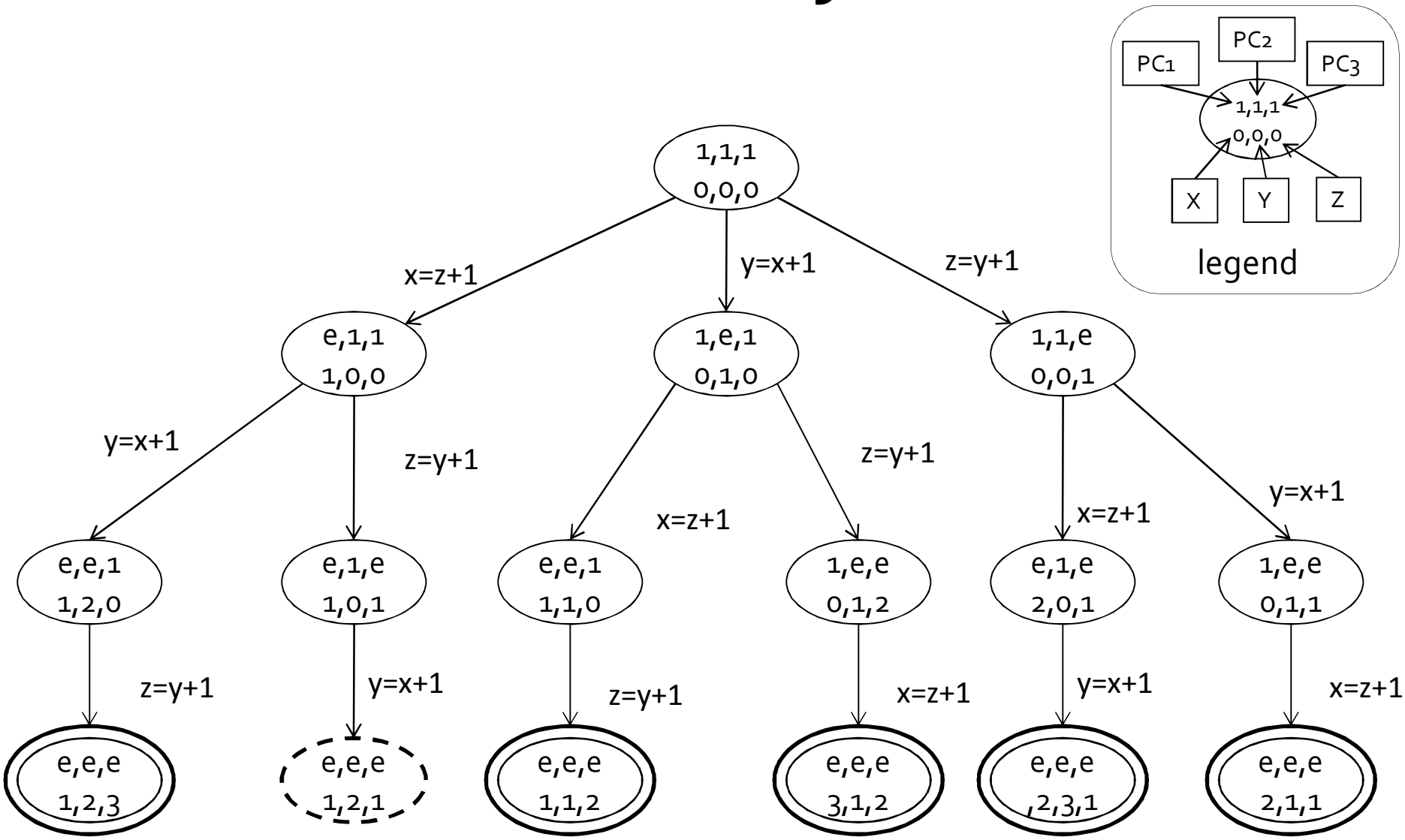
Specification:

- $\neg(y = 2 \wedge z = 1)$
- No Stuck States

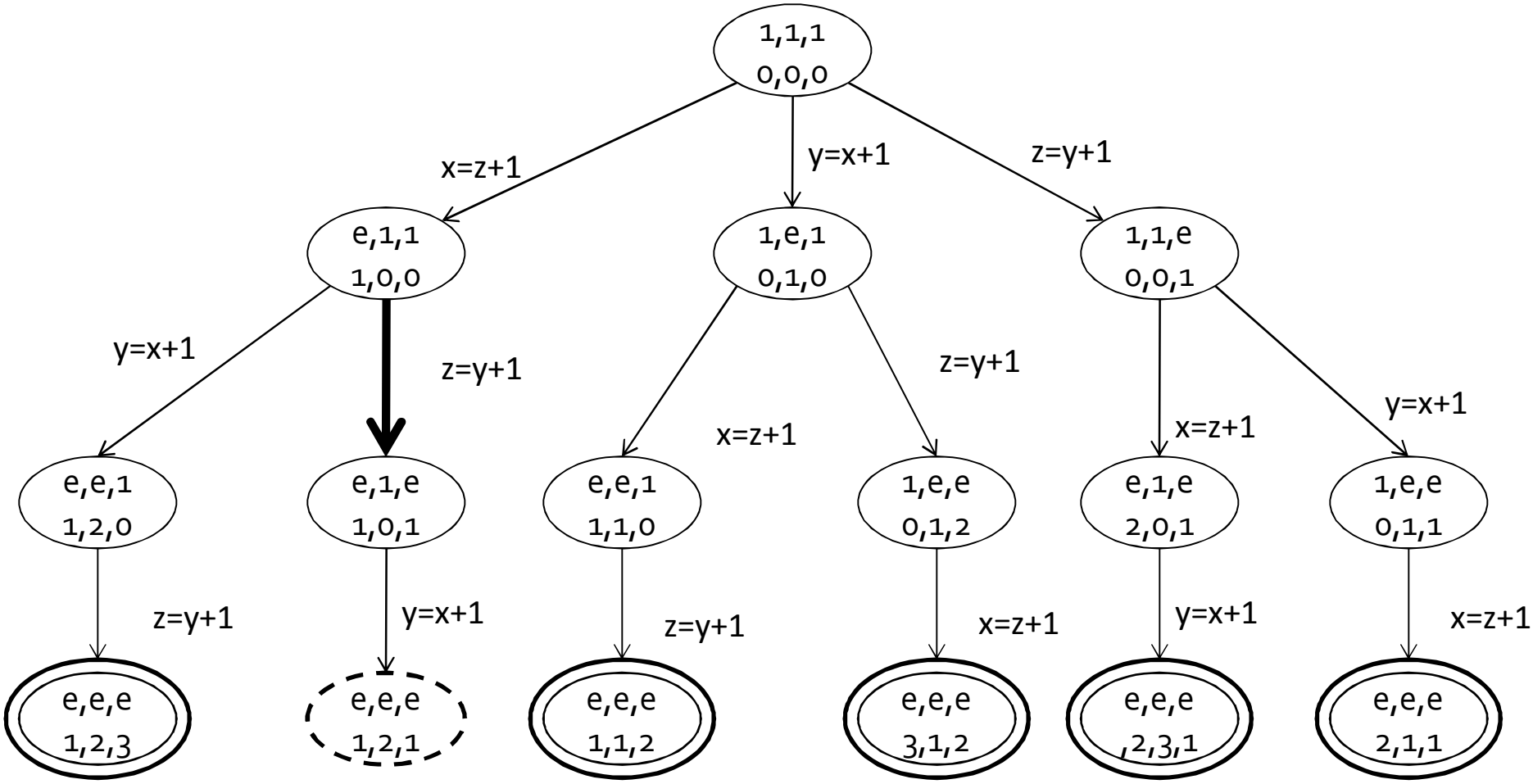
Abstraction:

$\{$ **x, y, z** $\}$

Build Transition System

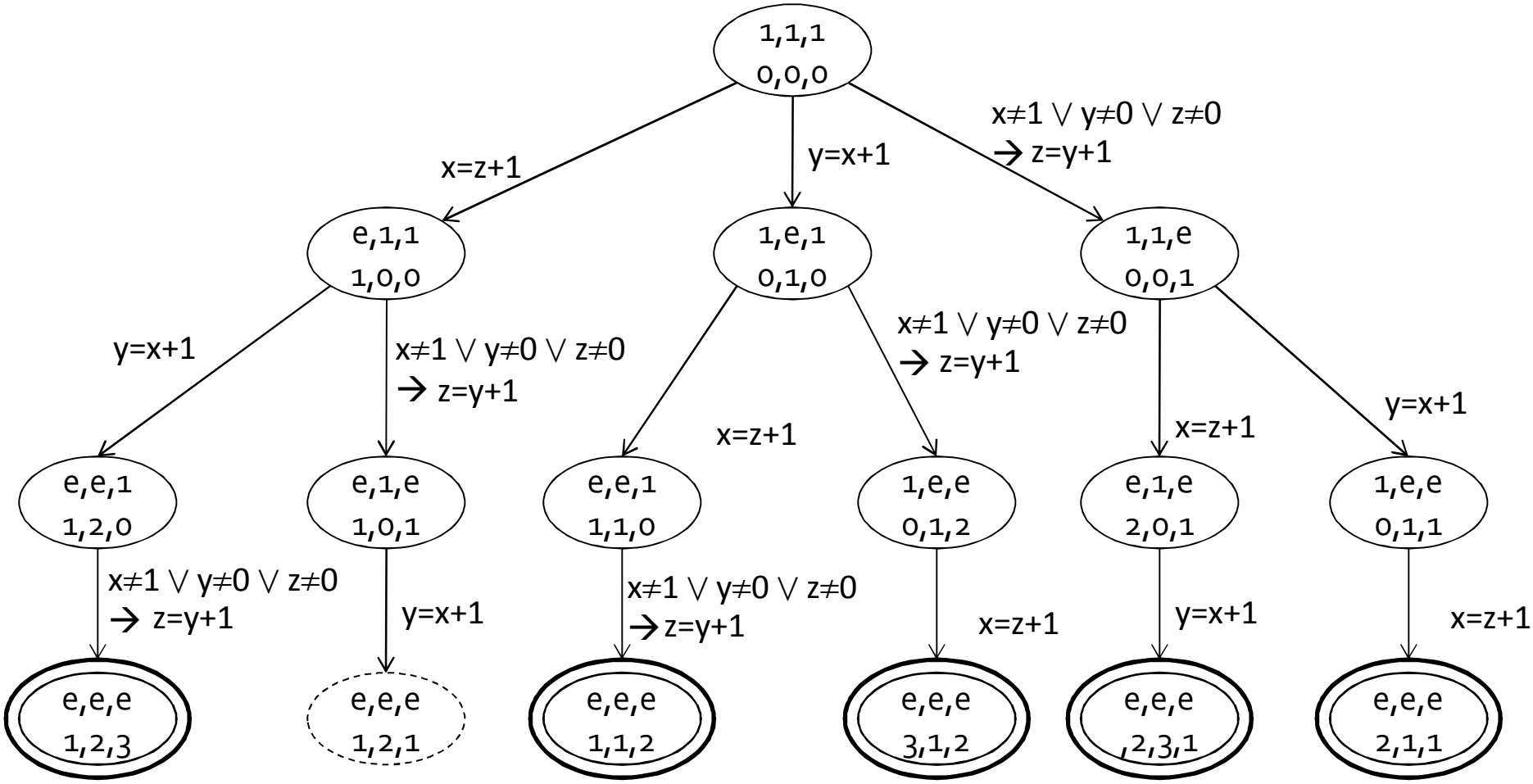


Avoid transition



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:

- $\neg(y = 2 \wedge z = 1)$
- No Stuck States

Abstraction:

$\{ \mathbf{x}, \mathbf{y}, \mathbf{z} \}$

T1	T2	T3
1: $x = z + 1$	1: $y = x + 1$	1: $(\mathbf{x} \neq \mathbf{1} \vee \mathbf{y} \neq \mathbf{0} \vee \mathbf{z} \neq \mathbf{0}) \rightarrow z = y + 1$

Example: limited observability

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

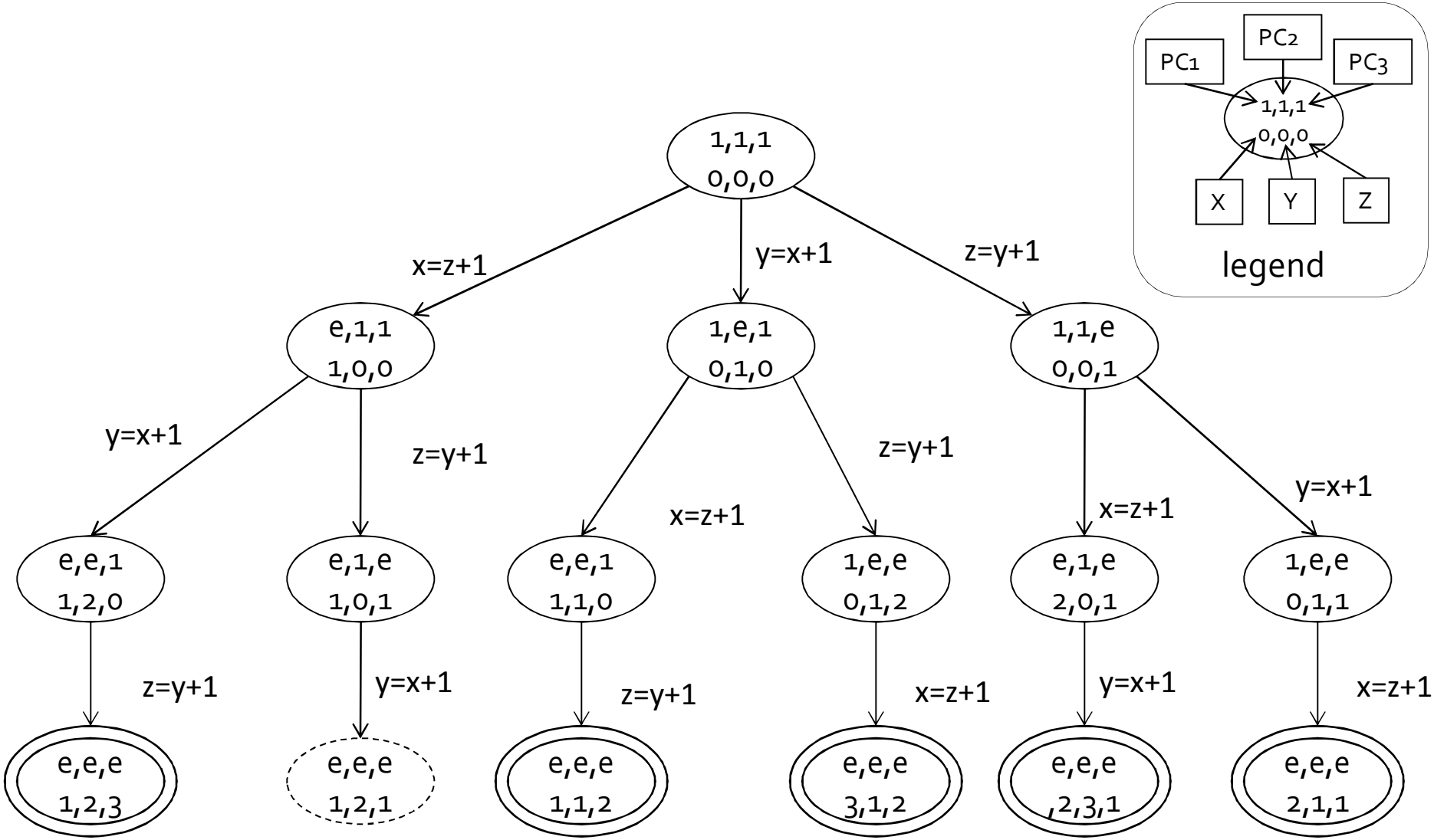
Specification:

- $\neg(y = 2 \wedge z = 1)$
- No Stuck States

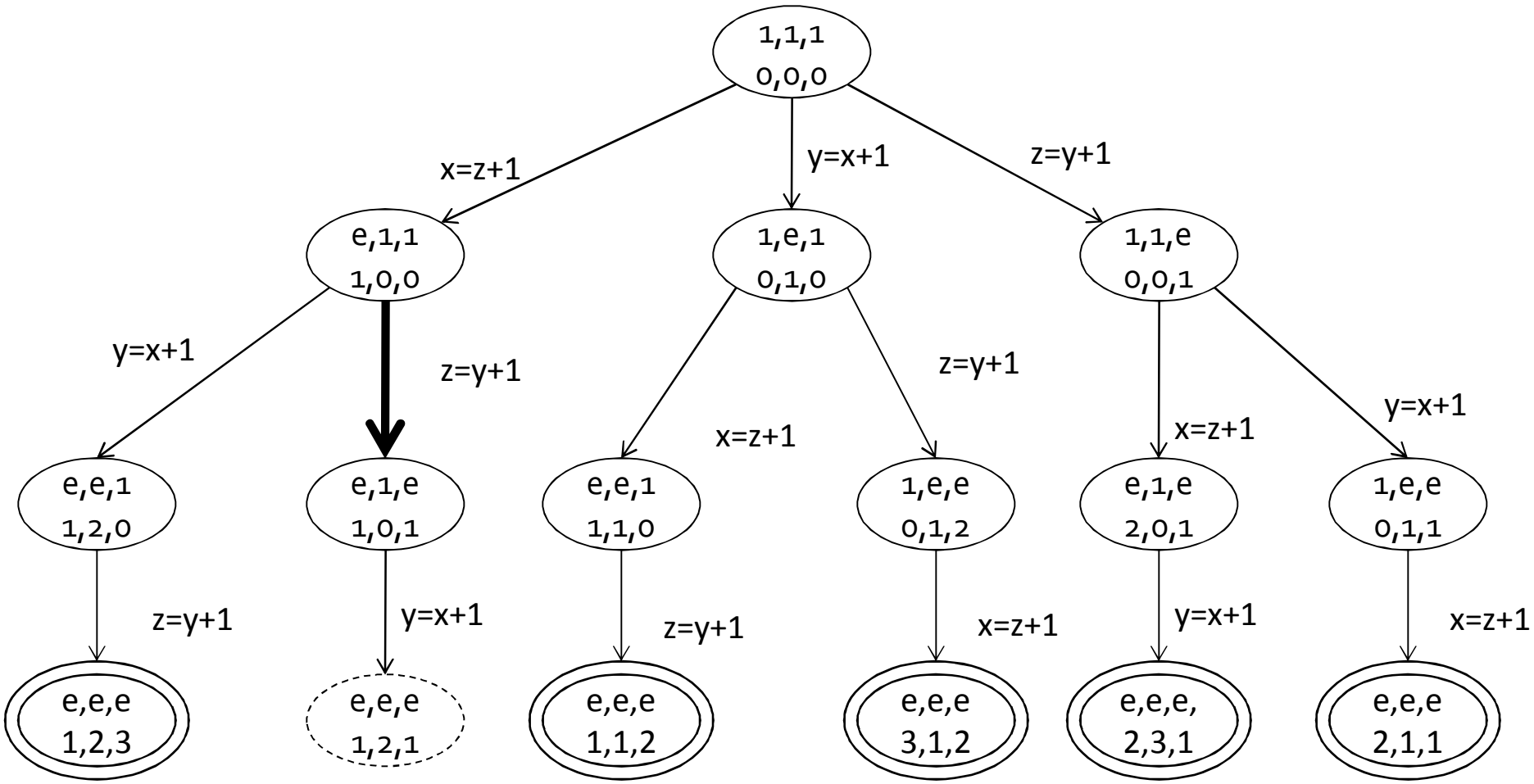
Abstraction:

{ **x, z** }

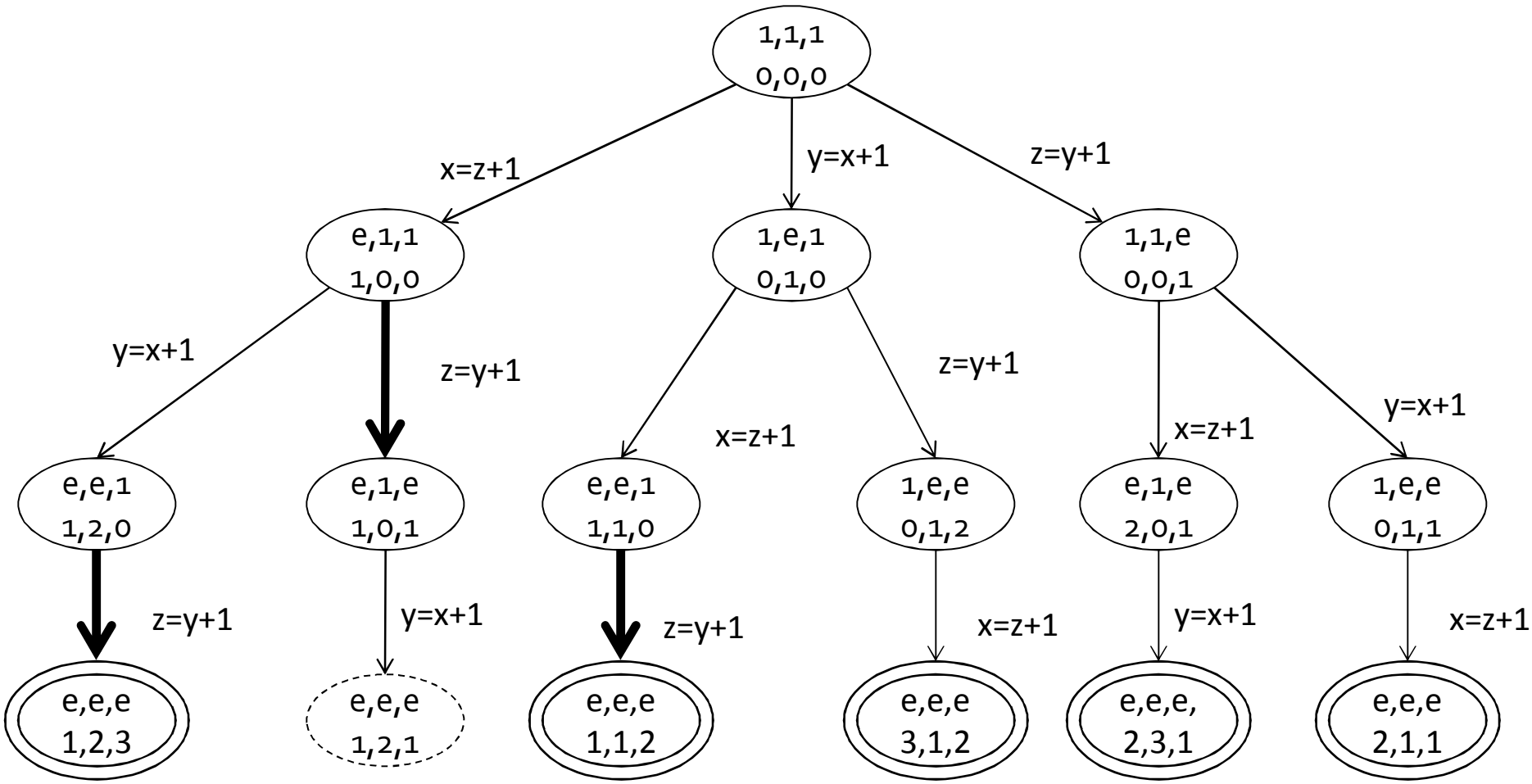
Build transition system



Avoid bad state

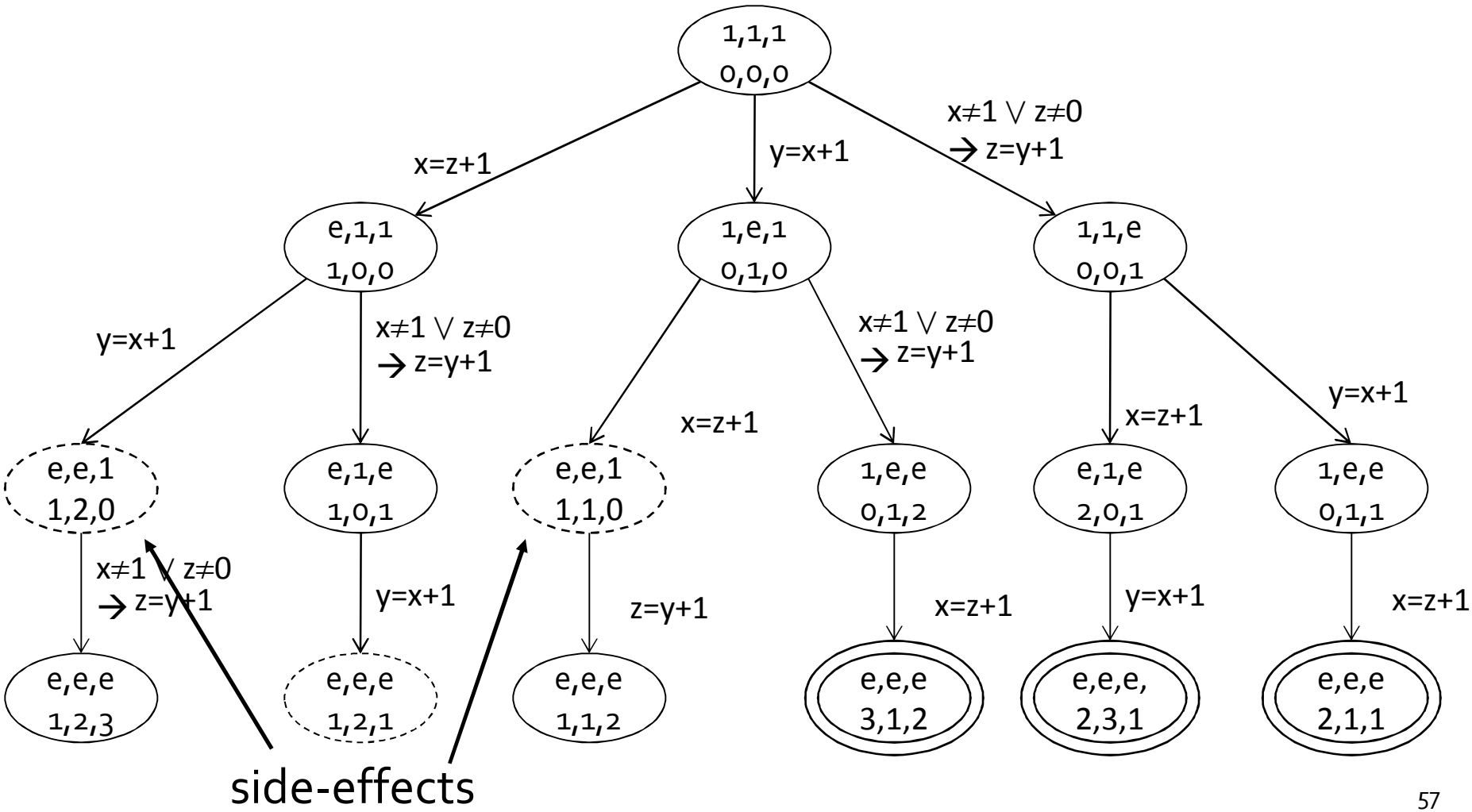


Select all equivalent transitions

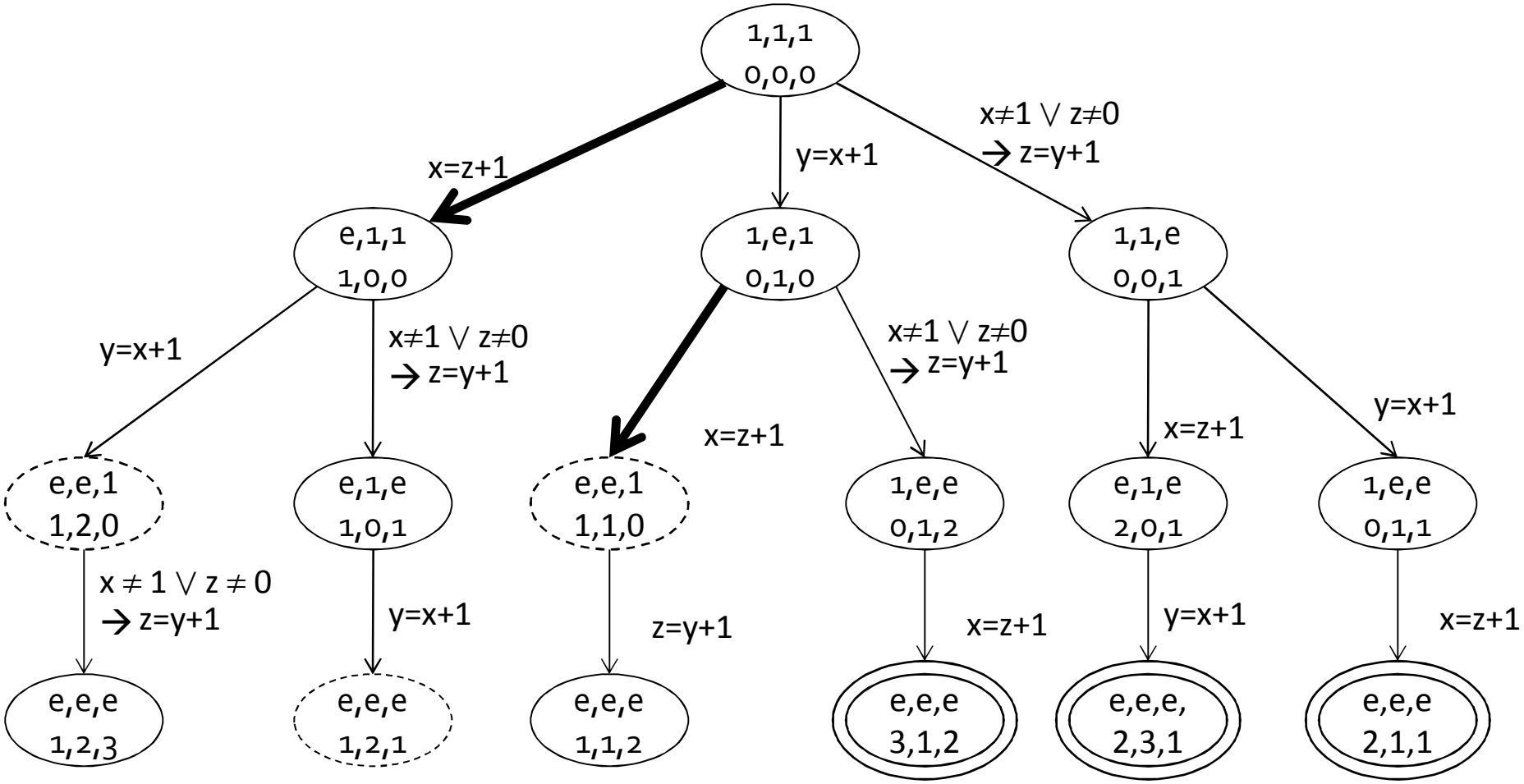


Implementability

Result has stuck states

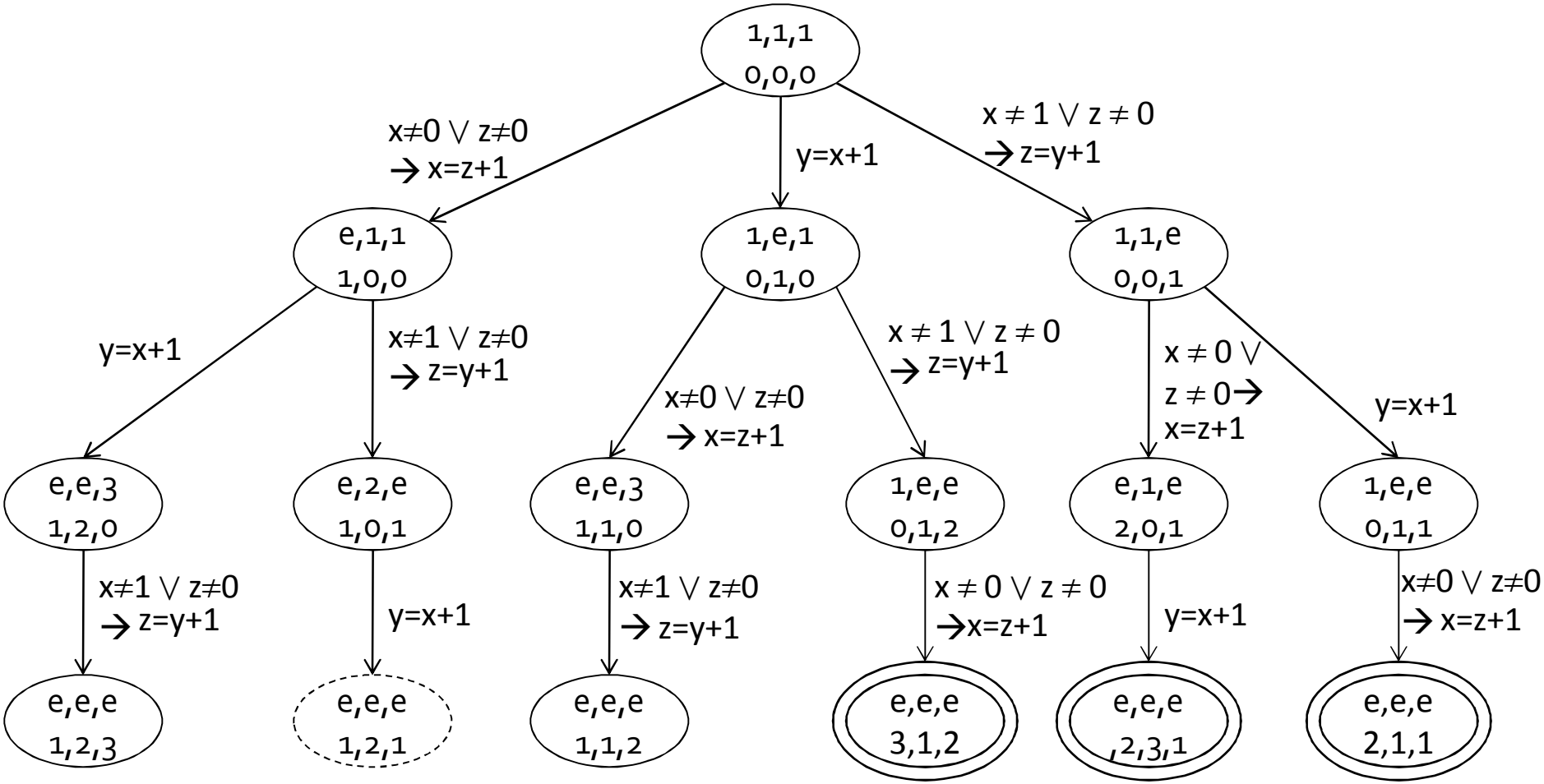


Select transition to remove



Result is valid

Correct and Maximally Permissive ✓



Resulting program

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

Specification:

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

Abstraction:

{ **x, z** }

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

Synthesis Results

- Mostly mutual exclusion primitives
- Different variations of the abstraction

Program	**FD k=0**	**FD k=1**	**PD k=0**	**PD k=1**	**PD k=2**
Sense0	✓	☺	✓	☺	☺
Pet0	☺	☺	✓	✓	☺
Dek0	☺	☺	✓	✓	☺
Lam0	T/O	T/O	✓	✓	T/O
Fast0	T/O	T/O	☹	T/O	T/O
Fast1a	✓	✓	☹	☺	☺
Fast1b	✓	✓	☹	✓	☺
Fast1c	T/O	T/O	☹	✓	☺

Chase-Lev Work-Stealing Queue

```
int take() {
   long b = bottom - 1;
   item_t * q = wsq;
   bottom = b
   fence("store-load");
   long t = top
   if (b < t) {
      bottom = t;
      return EMPTY;
   }
   task = q->ap[b % q->size];
   if (b > t)
      return task
   if (!CAS(&top, t, t+1))
      return EMPTY;
   bottom = t + 1;
   return task;
  }
```

```
void push(int task) {
   long b = bottom;
   long t = top;
   item_t * q = wsq;
   if (b - t >= q->size - 1) {
       wsq = expand();
       q = wsq;
   }
   q->ap[b % q->size] = task;
   fence("store-store");
   bottom = b + 1;
 }
```

```
int steal() {
   long t = top;
   fence("load-load");
   long b = bottom;
   fence("load-load");
   item_t * q = wsq;
   if (t >= b)
       return EMPTY;
   task = q->ap[t % q->size];
   fence("load-store");
   if (!CAS(&top, t, t+1))
       return ABORT;
   return task;
 }
```

```
item_t* expand() {
int  newsize = wsq->size * 2;
int* newitems = (int *) malloc(newsize*sizeof(int));
item_t *newq =  (item_t *)malloc(sizeof(item_t));
for (long i = top; i < bottom; i++) {
 newitems[i % newsize] = wsq->ap[i % wsq->size];
}
newq->size = newsize;
newq->ap = newitems;
fence("store-store");
wsq = newq;
return newq;
}
```

Specification: no lost items, no phantom items, memory safety

Results

	Initial	Client	$\|E\|$ Bound	RMO			PSO			TSO			SC	
				States	Edges	#C	States	Edges	#C	States	Edges	#C	States	Edges
MSN	empty	e|d	∞	1219	2671	2	455	743	1	228	316	0	146	180
	empty	e|e	∞	4934	12670	1	2678	6354	1	586	994	0	252	328
	empty	ee|dd	∞	24194	61514	3	7025	13689	2	1724	2512	0	1029	1325
	empty	ed|ed	∞	86574	242822	2	15450	35362	2	2476	3972	0	1538	2126
	empty	ed|de	∞	59119	167067	4	11023	24362	2	2570	4010	0	1541	2073
	empty	e|e|d	∞	233414	653094	3	51990	119050	2	9638	16822	0	4928	7632
Chase-Lev WSQ	empty	pppt(tpt|sss)	∞	386283	1030857	-	74533	256613	-	12348	20004	-	4961	6740
	empty	tttt(ptt|sss)	∞	1048498	2819355	-	124455	255390	-	6418	9380	-	3101	4069
	empty	pppt(ttp|sss)	∞	281314	878880	-	66960	241814	-	10564	16317	-	4199	5700
	empty	tttt(tpp|sss)	∞	1325858	4150650	-	361855	1080835	-	9878	13956	-	3473	4537
	empty	tttp(tptp|ss)	∞	280396	698398	-	29573	54696	-	9197	14499	-	4760	6455
"LIFO" WSQ	2/2	tp|ss	∞	2151	3190	2	882	1171	1	676	852	0	570	694
	2/2	tpt|ss	∞	9721	16668	2	3908	5811	1	2256	3116	0	1410	1786
	2/2	ptp|ss	∞	89884	195246	3	31289	64133	3	4045	5688	0	2317	3007
	2/2	ptt|ss	∞	85104	198353	4	29920	62020	3	4130	5987	0	2198	2866
	1/1	ptt|ss	∞	23913	48997	4	9976	18002	3	2353	3269	0	1314	1654
Dekker	-	-	1	1388	2702	2	1388	2702	2	489	674	1	388	490
	-	-	10	7504	14477	2	7504	14477	2	2560	3750	1	388	490
	-	-	20	13879	26422	2	13879	26422	2	4845	7115	1	388	490
	-	-	50	33004	62257	2	33004	62257	2	11770	17210	1	388	490
Treiber	empty	p|t	∞	71	93	2	71	93	2	43	48	0	36	38
	empty	pt|tp	∞	3054	6190	2	3041	6167	2	407	609	0	392	482
	empty	pp|tt	∞	1276	2250	2	1276	2250	2	325	407	0	270	323
VYSet	empty	ar|ra	10	4079	6247	2	4079	6247	2	1088	1308	0	1088	1308
	empty	aa|rr	10	20034	31623	2	20034	31623	2	1168	1411	0	1168	1411
	empty	ar|ar	10	6093	9905	2	6093	9905	2	1671	1968	0	1671	1968
	empty	aaa|rrr	10	41520	66533	2	41520	66533	2	3311	4072	0	3311	4072

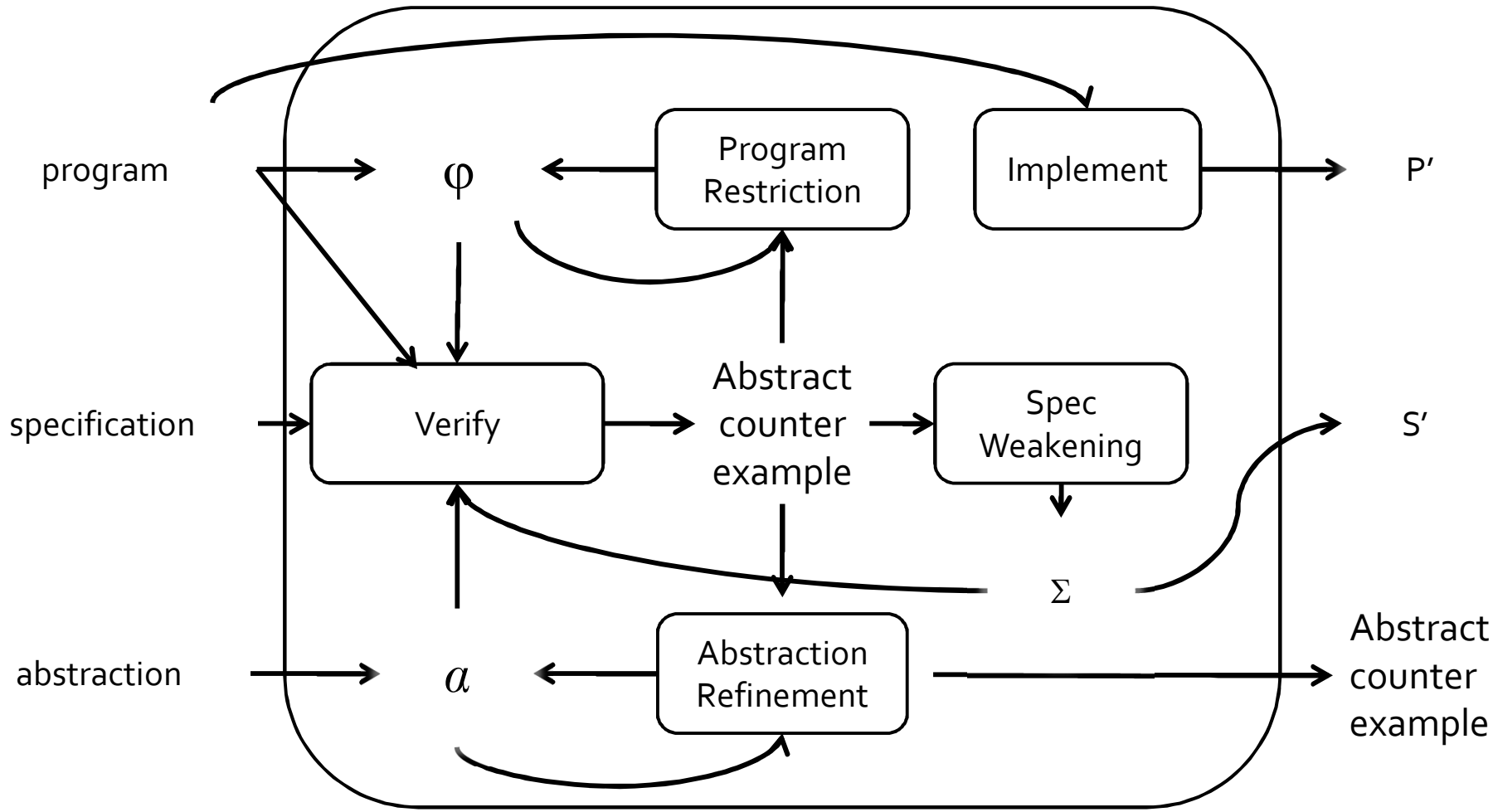
Performance

	Initial State	Client	$\|E\|$ Bnd	Time (sec.) total	build	form.	solve
MSN	empty	e|d	∞	0.83	0.45	0.14	0.23
	empty	e|e	∞	1.78	1.15	0.54	0.07
	empty	ee|dd	∞	5.21	2.87	2.12	0.21
	empty	ed|ed	∞	13.05	9.14	3.73	0.18
	empty	ed|de	∞	9.26	6.54	2.52	0.19
	empty	e|e|d	∞	31.43	23.03	8.2	0.18
Chase-Lev WSQ	empty	pppt(tpt|sss)	∞	97.22	56.1	41.11	-
	empty	tttt(ptt|sss)	∞	255.5	160.06	95.44	-
	empty	pppt(ttp|sss)	∞	90.28	45.17	45.11	-
	empty	tttt(tpp|sss)	∞	355.95	212.29	143.65	-
	empty	tttp(tptp|ss)	∞	37.98	31.34	6.63	-
"LIFO" WSQ	2/2	tp|ss	∞	0.69	0.5	0.14	0.05
	2/2	tpt|ss	∞	1.94	1.27	0.61	0.06
	2/2	ptp|ss	∞	11.41	8.27	3.05	0.09
	2/2	ptt|ss	∞	11.31	8.26	2.99	0.06
	1/1	ptt|ss	∞	4.07	2.57	1.44	0.06
Dekker	-	-	1	0.64	0.39	0.19	0.05
	-	-	10	2.13	1.15	0.92	0.05
	-	-	20	2.71	1.68	0.98	0.05
	-	-	50	5.99	4.46	1.46	0.05
Treiber	empty	p|t	∞	1	0.07	0.02	0
	empty	pt|tp	∞	1.02	0.76	0.26	0
	empty	pp|tt	∞	0.6	0.46	0.14	0
VYSet	empty	ar|ra	10	1.98	1.49	0.41	0.08
	empty	aa|rr	10	4.56	3.41	1.1	0.05
	empty	ar|ar	10	2.19	1.7	0.43	0.05
	empty	aaa|rrr	10	7.98	5.61	2.24	0.13

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]
Reordering	Memory fences	Partial-Coherence abstractions for Store buffers	[FMCAD'10] [PLDI'11]
	…		

General Setting Revisited



Change the specification to match the abstraction

Summary

- Modifying the program to fit an abstraction
  - examples inspired by the trace partitioning domain
  - (trace partitioning domain can capture these effects, and more, without changing the program)
- 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
  - Separate characterization of solution from choosing optimal solutions (e.g., smallest atomic sections)