MACHINE-ASSISTED CONCURRENT PROGRAMMING

Martin Vechev

ETH Zürich

(Lecture 1)

Based on material from:

- "Abstraction-Guided Synthesis of Synchronization", POPL'2010
- "Automatic Inference of Memory Fences", FMCAD'2010
- "Partial coherence abstractions", PLDI'2011

Machine-Assisted Programming

- let the machine perform certain programming tasks
  - not only verify, but synthesize code

- help programmers
  - code prototyping
  - figure out tricky implementation details
  - even discover new algorithms !

- emerging area: concurrency, automated education, end-user programming, code search, etc…

Machine-Assisted Programming

- these lectures focus on concurrency
  - but the ideas are more general

- concurrent programming is:
  - error prone
  - time consuming
  - can lead to sub-optimal results

Challenge 1: Algorithm Discovery

synthesize that:

```
bool add(int key) {

 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,0⟩,⟨entry,0⟩)
 if (¬val) goto restart
 return true

}
```

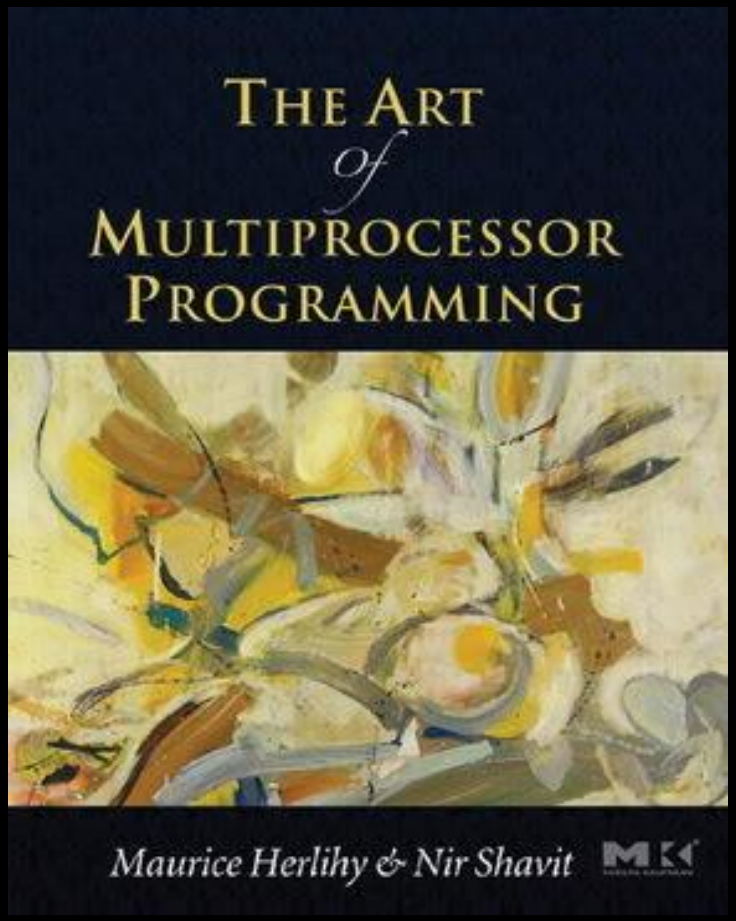
```
bool remove(int key) {

 Entry *pred,*c,*r

restart:

 locate(pred,c,key)
 k = (c->key ≠ key)
 if (k) return false
 <r,m> = c->next
 lval=CAS(&c->next, ⟨r,m⟩,⟨r,1⟩)
 if (¬lval) goto restart
 pval=CAS(&pred->next, ⟨c,0⟩,⟨r,0⟩)
 if (¬pval) goto restart
 return true

}
```

```
bool contains(int key) {

 Entry *pred,*curr

 locate(pred,curr,key)
 k = (curr->key == key)
 if (k) return false
 return true

}
```

concurrent set based on a sorted singly linked list

Challenge 1: Algorithm Discovery

many more examples here:



Challenge 2: System Programming

automatically discover all synchronization used in Linux:

- e.g. locks, fences, barriers,…

```
static void wa_urb_enqueue_b(struct wa_xfer *xfer)
{
        int result;
        unsigned long flags;
        struct urb *urb = xfer->urb;
        struct wahc *wa = xfer->wa;
        struct wusbhc *wusbhc = wa->wusb;
        struct wusb_dev *wusb_dev;
        unsigned done;

        result = rpipe_get_by_ep(wa, xfer->ep, urb, xfer->gfp);
        if (result < 0)
                goto error_rpipe_get;
        result = -ENODEV;
        /* FIXME: segmentation broken -- kills DWA */
        mutex_lock(&wusbhc->mutex);             /* get a WUSB dev */
        if (urb->dev == NULL) {
                mutex_unlock(&wusbhc->mutex);
                goto error_dev_gone;
        }
        wusb_dev = __wusb_dev_get_by_usb_dev(wusbhc, urb->dev);
        if (wusb_dev == NULL) {
                mutex_unlock(&wusbhc->mutex);
                goto error_dev_gone;
        }
        mutex_unlock(&wusbhc->mutex);

                ...
```

USB Driver

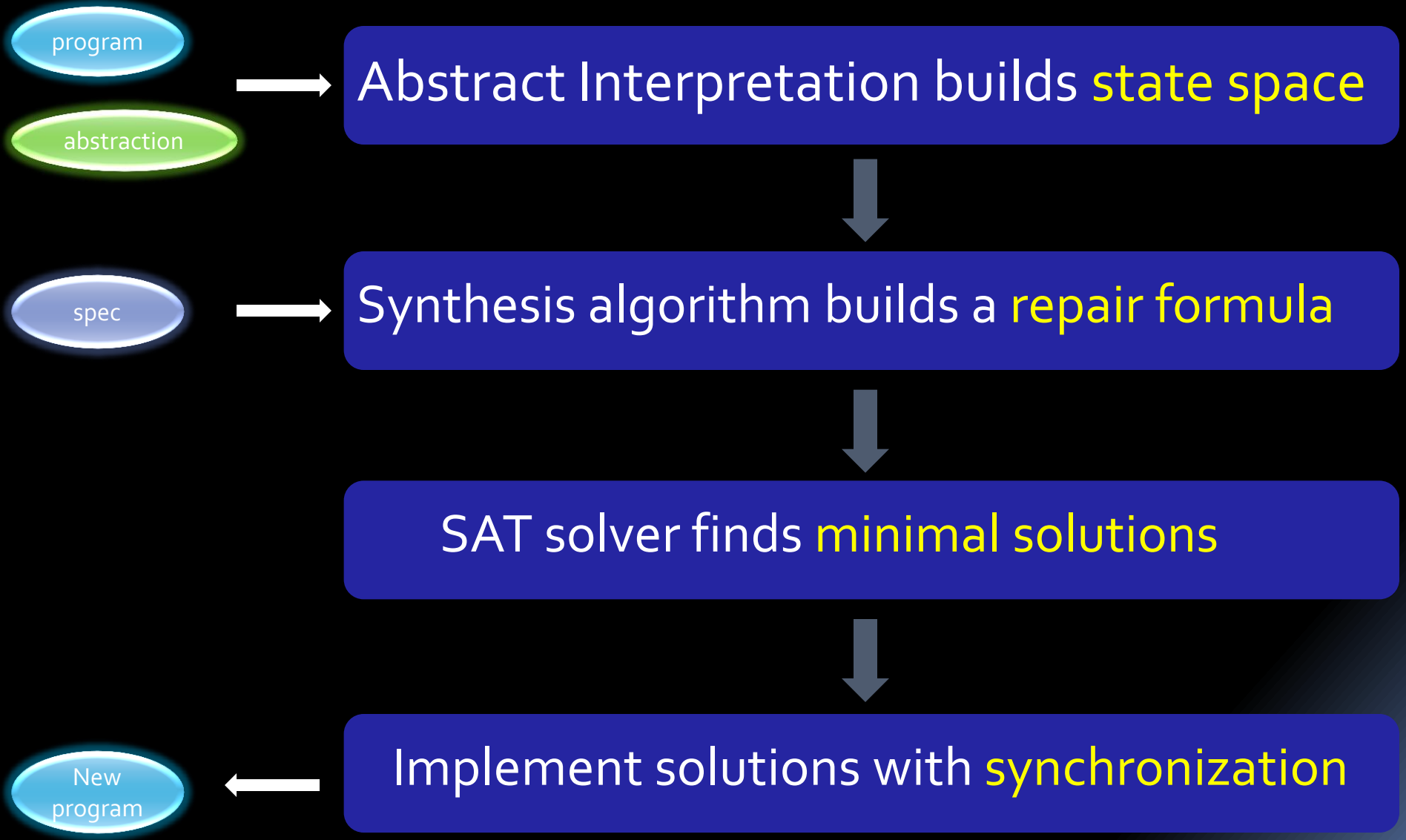
Starting Point

- synchronization is a central issue
- can we infer synchronization automatically ?
  - kinds of synchronization ?
  - scale of programs ?
  - verification techniques ?

These lectures

- specific technical concept
  - program analysis + discovery of synchronization
- applied to different settings
  - atomicity / conditional critical regions
  - weak memory models (e.g. AMD/Intel x86)
  - deterministic programming (e.g. GPUs)

Flow


program
abstraction
Abstract Interpretation builds state space
spec
Synthesis algorithm builds a repair formula
SAT solver finds minimal solutions
Implement solutions with synchronization
New program

Lecture Objectives

- understand how it works
- understand limitations / what does not work yet
- be able to apply the ideas to a new domain

Plan

- Setting
- Program analysis by abstract interpretation
- Synthesis based on abstract interpretation
- Analysis + synthesis for weak memory models

Setting

The general case is as follows: given a program P and a specification S, does program P satisfy specification S ?

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

Setting

If program P is infinite-state, and we want to answer the question automatically, we need to over-approximate P

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

Setting

to over-approximate P, we need an abstraction α

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

Setting

given a program P, a specification S, and an abstraction α, does program P satisfy specification S ?

$$P_\alpha \stackrel{?}{\models} S$$

Setting

a basic question in program analysis

$$P_{\alpha} \stackrel{?}{\models} S$$

Setting

but what if P does not satisfy S under abstraction α?

$$P_\alpha \not\models S$$

Setting

refine abstraction α to a new abstraction α'

$$P_{\alpha'} \models S$$

Setting

modify program P to a new program P′
how ?

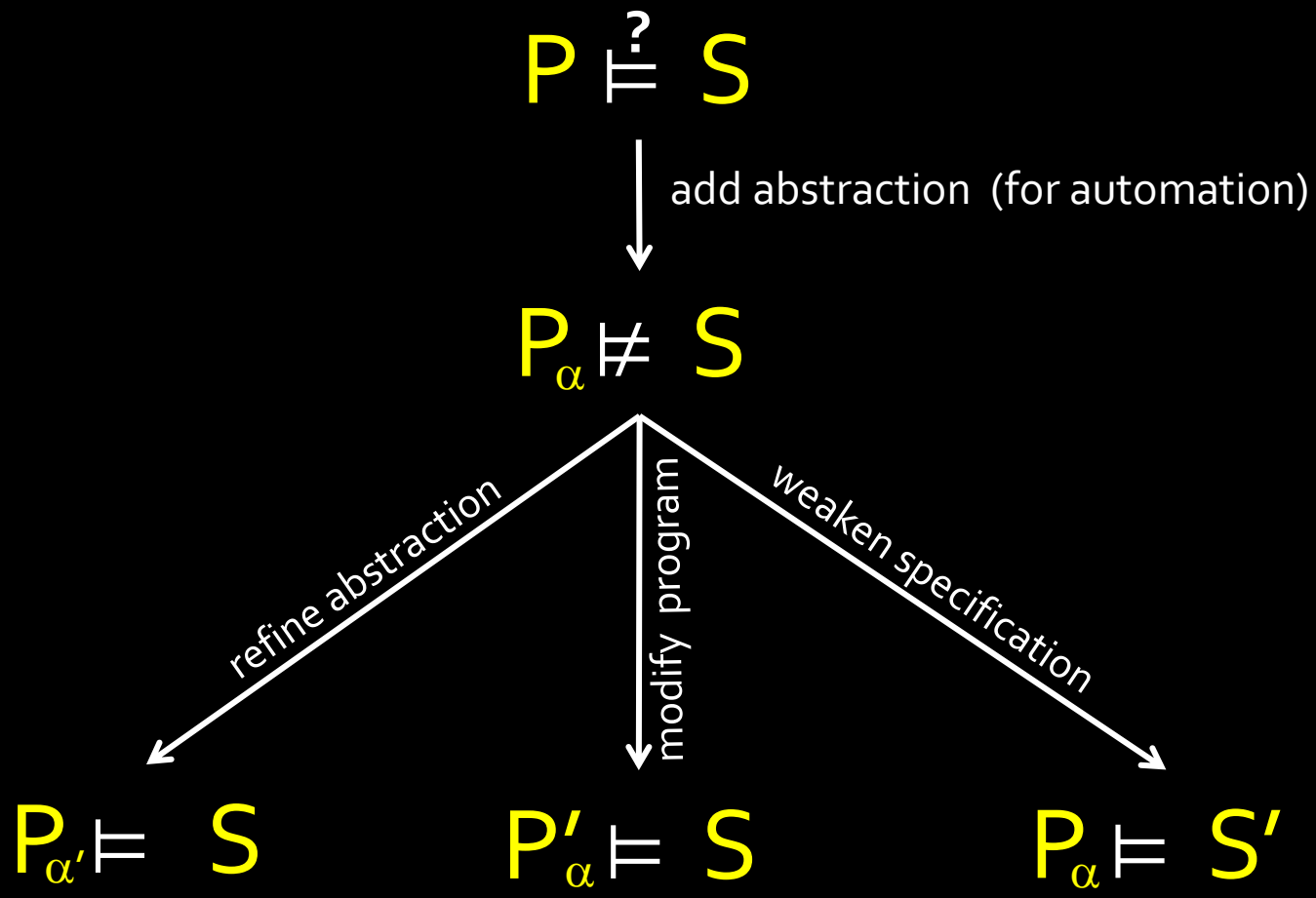
$$P' \models_{\alpha} S$$

Setting

weaken specification S to S′
unclear to what, true is also a solution

$$P_{\alpha} \models S'$$

Setting



can also combine steps

Plan

- Setting
- Program analysis by abstract interpretation
- Synthesis based on abstract interpretation
- Analysis + synthesis for weak memory models

Concrete Trace Semantics

- Trace semantics are the set of all finite program traces starting from initial configurations

$$[\![P]\!] = \{ c_0 \cdot c_1 \cdot \ldots \cdot c_{n-1} \mid n \geq 1 \wedge c_0 \in I \wedge \forall i \in [0, n-2]: c_i \rightarrow c_{i+1} \}$$

- $c_i \rightarrow c_{i+1}$ is a transition from configuration c_i to c_{i+1}
- Note that traces need not end in final configurations
- Traces are of finite length, but the number of initial configurations can be infinite. Hence, an infinite number of traces: computation is non-feasible

Properties

To check if P satisfies a property, check that all sequences in $[\![P]\!]$ satisfy the property

Problem: $[\![P]\!]$ can be infinite (or very large)

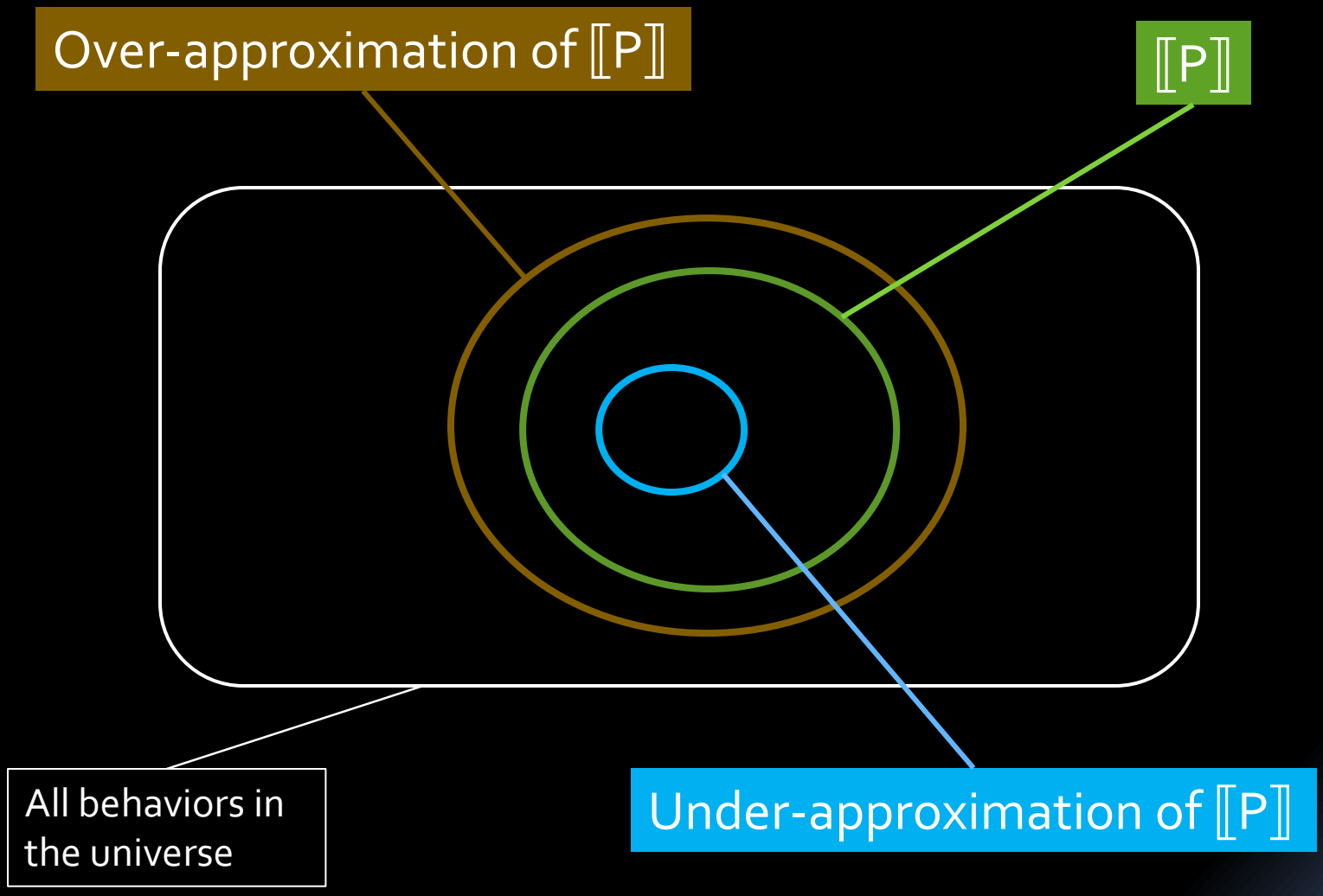
Program Analysis: 2 approaches

$[\![P]\!]$ is generally not computable, so we can:

- Under-approximate
  - Pick a subset of $[\![P]\!]$
- Over-approximate
  - Discover a superset of $[\![P]\!]$

Both approaches are useful in practice

Program Analysis: 2 approaches



Static Program Analysis: Over-approximate ⟦P⟧

Automatically over-approximate ⟦P⟧ using a bounded representation $⟦P⟧^{\#}$. Then, prove properties on $⟦P⟧^{\#}$

\+ property that holds in $⟦P⟧^{\#}$ is guaranteed to hold in ⟦P⟧

\- $⟦P⟧^{\#}$ may include behaviors that do not exist in ⟦P⟧. So the property may still hold for ⟦P⟧, even when it does not hold in $⟦P⟧^{\#}$

The Beginning of Abstraction

Lets start with the trace semantics

Computing an Abstraction of ⟦P⟧

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

Property

$$\langle 1, \{ i \mapsto 42, x \mapsto 9, y \mapsto 3 \} \rangle \xrightarrow{x := 5^1} \langle 2, \{ i \mapsto 42, x \mapsto 5, y \mapsto 3 \} \rangle \xrightarrow{y := 7^2}$$

$$\langle 3, \{ i \mapsto 42, x \mapsto 5, y \mapsto 7 \} \rangle \xrightarrow{i \geq 0^3} \langle 4, \{ i \mapsto 42, x \mapsto 5, y \mapsto 7 \} \rangle \xrightarrow{y := y+1^4}$$

$$\langle 5, \{ i \mapsto 42, x \mapsto 5, y \mapsto 8 \} \rangle \xrightarrow{i := i-1^5} \langle 6, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle \xrightarrow{goto\ 3^6}$$

$$\langle 3, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle \xrightarrow{i \geq 0^3} \langle 4, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle \xrightarrow{y := y+1^4}$$

$$\langle 5, \{ i \mapsto 41, x \mapsto 5, y \mapsto 9 \} \rangle \xrightarrow{i := i-1^5} \langle 6, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle \xrightarrow{goto\ 3^6}$$

$$\langle 3, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle \xrightarrow{i \geq 0^3} \langle 4, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle \rightarrow \ldots$$

Computing an Abstraction of ⟦P⟧

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

Property

The property refers to program states and not program traces

Enough to keep track of program states and not of program traces

This suggests our first abstraction

Computing an Abstraction of ⟦P⟧

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

$$
\begin{aligned}
&[\![P]\!]_s = \\
&\{ \\
&\quad \langle 1, \{ i \mapsto 42, x \mapsto 9, y \mapsto 3 \} \rangle, \langle 2, \{ i \mapsto 42, x \mapsto 5, y \mapsto 3 \} \rangle, \\
&\quad \langle 3, \{ i \mapsto 42, x \mapsto 5, y \mapsto 7 \} \rangle, \langle 4, \{ i \mapsto 42, x \mapsto 5, y \mapsto 7 \} \rangle, \\
&\quad \langle 5, \{ i \mapsto 42, x \mapsto 5, y \mapsto 8 \} \rangle, \langle 6, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle, \\
&\quad \langle 3, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle, \langle 4, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle, \\
&\quad \langle 5, \{ i \mapsto 41, x \mapsto 5, y \mapsto 9 \} \rangle, \langle 6, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle, \\
&\quad \langle 3, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle, \langle 4, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle, \\
&\quad \ldots\ldots\ldots \\
&\}
\end{aligned}
$$

- The set $[\![P]\!]_s$ contains all reachable states, but it lost the relationship between states
- The set $[\![P]\!]_s$ is infinite

Computing an Abstraction of ⟦P⟧

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

$$\llbracket P \rrbracket_s = \{ \langle 1, \{ i \mapsto 42, x \mapsto 9, y \mapsto 3 \} \rangle, \langle 2, \{ i \mapsto 42, x \mapsto 5, y \mapsto 3 \} \rangle, \langle 3, \{ i \mapsto 42, x \mapsto 5, y \mapsto 7 \} \rangle, \langle 4, \{ i \mapsto 42, x \mapsto 5, y \mapsto 7 \} \rangle, \langle 5, \{ i \mapsto 42, x \mapsto 5, y \mapsto 8 \} \rangle, \langle 6, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle, \langle 3, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle, \langle 4, \{ i \mapsto 41, x \mapsto 5, y \mapsto 8 \} \rangle, \langle 5, \{ i \mapsto 41, x \mapsto 5, y \mapsto 9 \} \rangle, \langle 6, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle, \langle 3, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle, \langle 4, \{ i \mapsto 40, x \mapsto 5, y \mapsto 9 \} \rangle, \ldots\ldots\ldots \}$$

- The set $\llbracket P \rrbracket_s$ is infinite
- Instead of keeping all values for a variable, abstract its values and keep only the sign of the value, that is whether it is +,- or zero.

Computing an Abstraction of ⟦P⟧

Recipe:

1. select/define an abstract domain
   - selected based on the property you want to prove about P

2. define abstract semantics for the language w.r.t. to the domain
   - prove sound w.r.t concrete semantics
   - involves defining abstract transformers

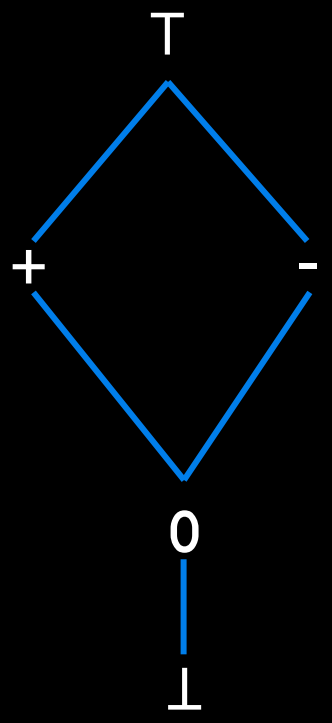
3. iterate abstract transformers over the abstract domain
   - until we reach a fixed point

the fixed point is the over-approximation (abstraction) of ⟦P⟧

Step 1: Select abstract domain

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

Abstract domain here is a cross-product of sign-domain for each variable at each label



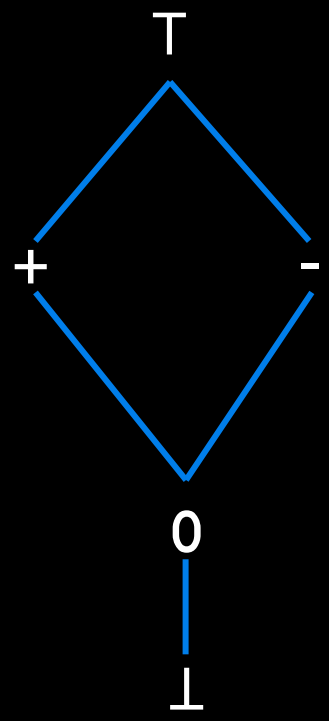
What does + mean ?

Step 1: Select abstract domain

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
2	+	⊥	T

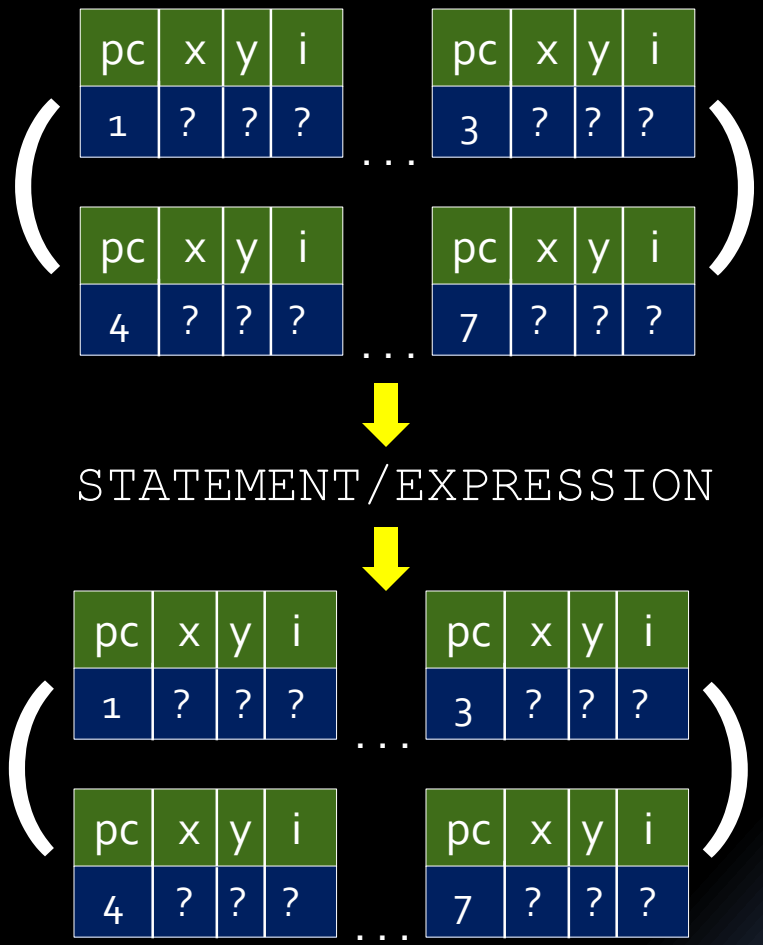
Example Abstract Element



An abstract program state is a map from labels to variables to elements in the domain

Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



An abstract transformer describes the effect of statement and expression evaluation on an abstract state

Important Point

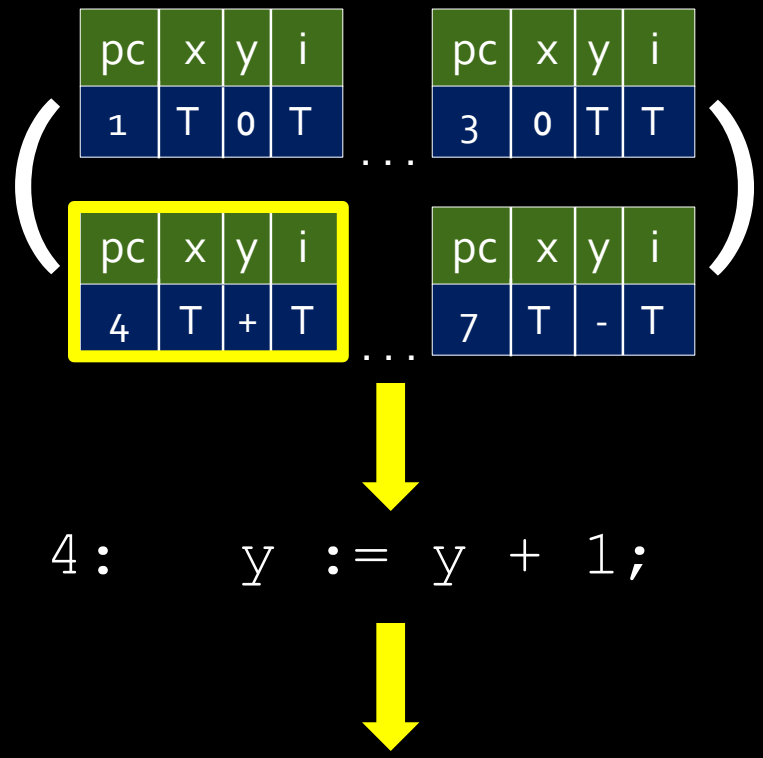
It is important to remember that abstract transformers are defined per programming language once and for all, and not per-program !

That is, they essentially define the new (abstract) operational semantics of the language.

This means that any program in the programming language can use the same transformers.

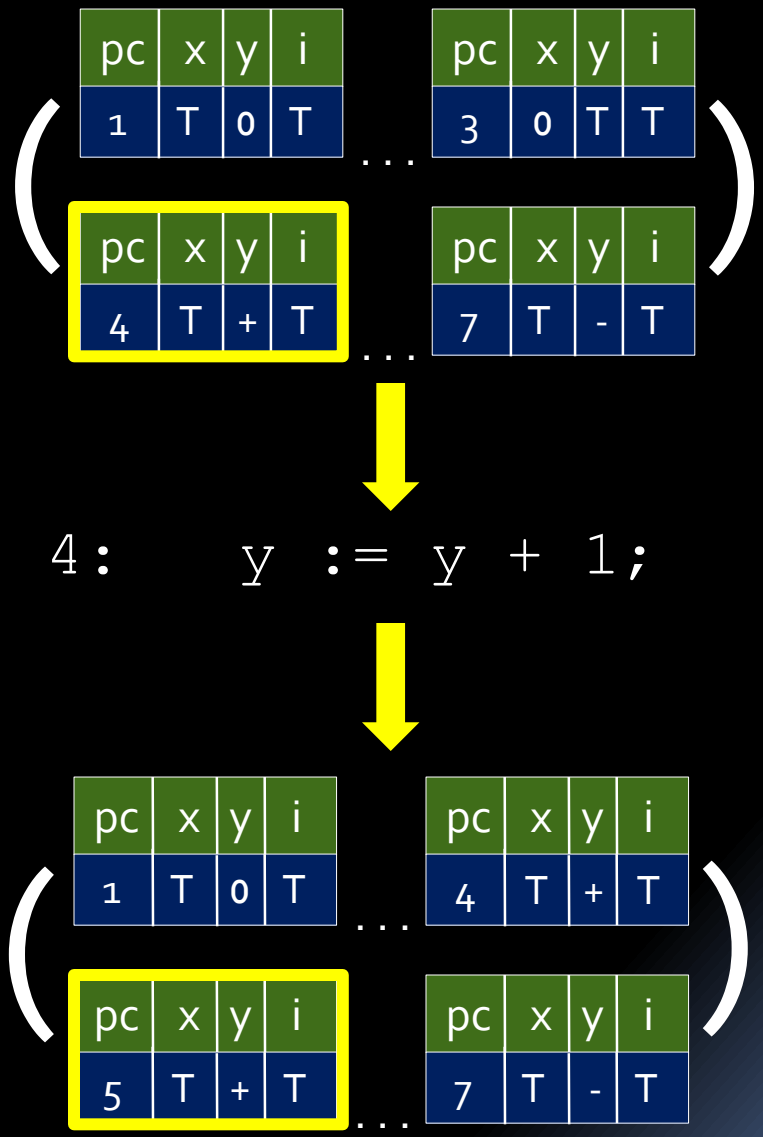
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



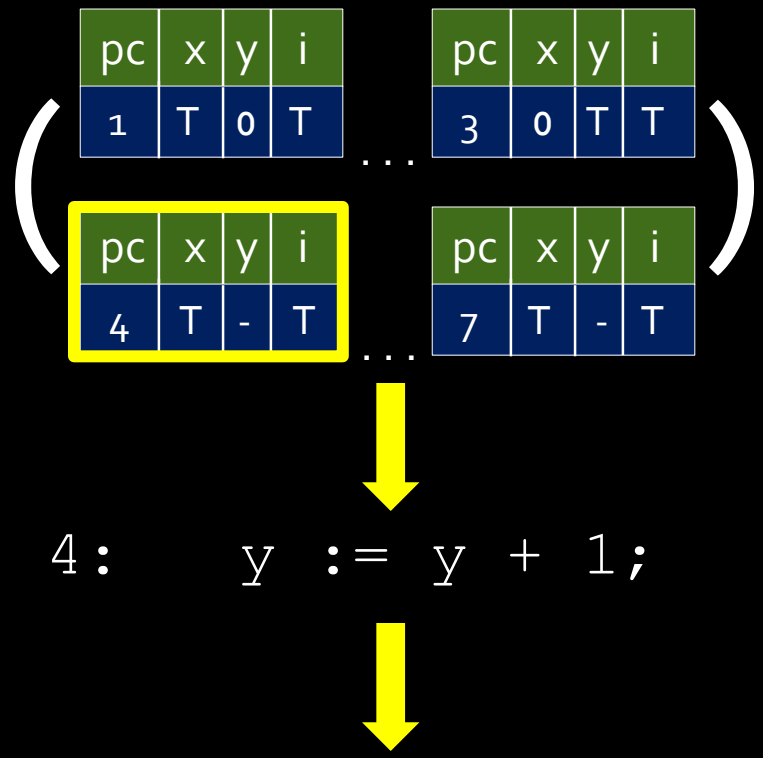
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



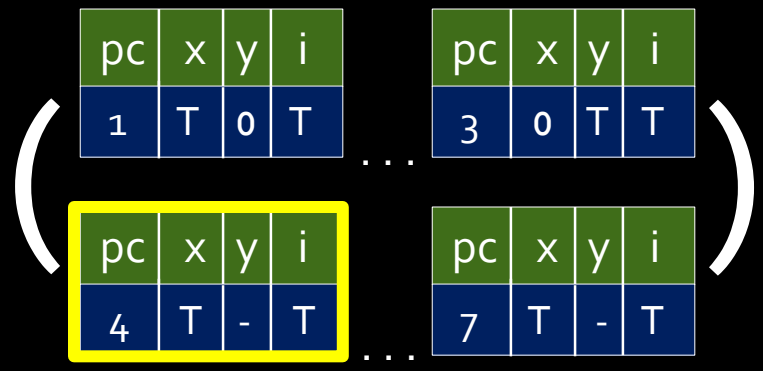
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



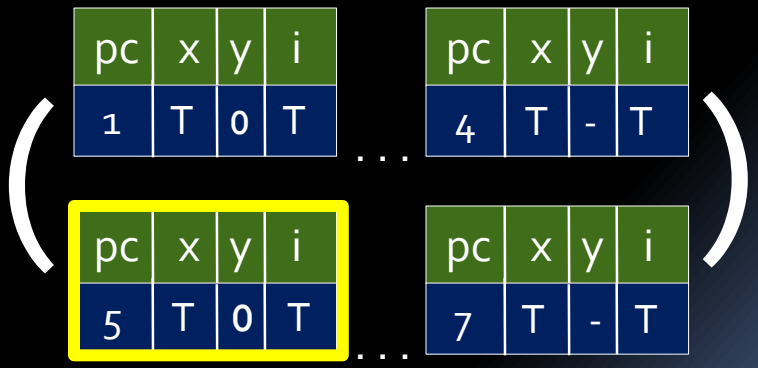
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



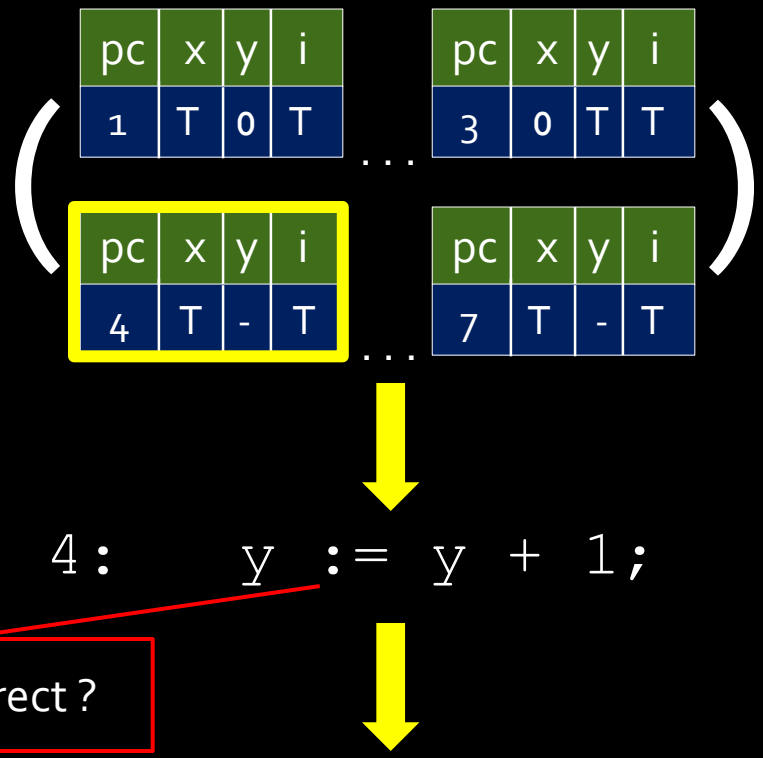
4: y := y + 1;

is this correct ?



Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



What does correct mean ?

Transformer Correctness

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

A correct abstract transformer should always produce results that are a superset of what a concrete transformer (statement) would produce

Unsound Transformer

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

This abstract state:

pc	x	y	i
4	T	-	T

represents infinitely many concrete states including:

pc	x	y	i
4	1	-3	2

If we perform `y := y + 1` to this concrete state, we get:

pc	x	y	i
5	1	-2	2

However, the abstract transformer produces an abstract state:

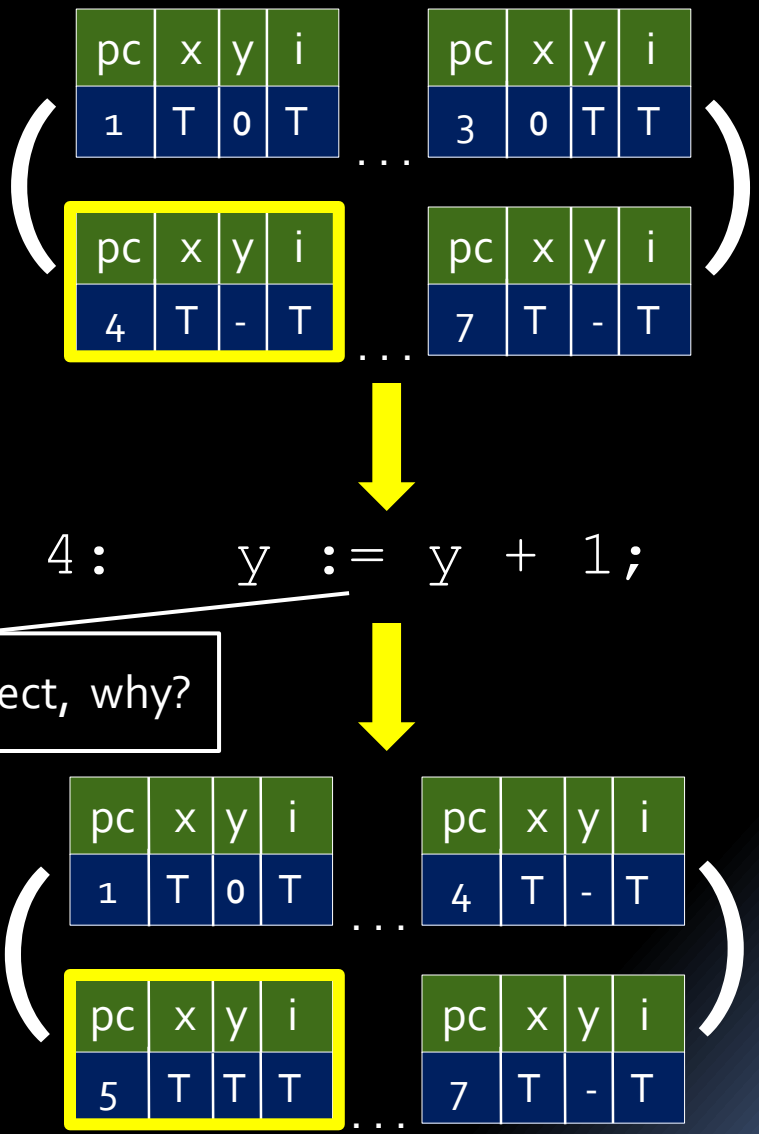
pc	x	y	i
5	T	0	T

This abstract state does not represent any state where y = -2

The abstract transformer is unsound ! ☹

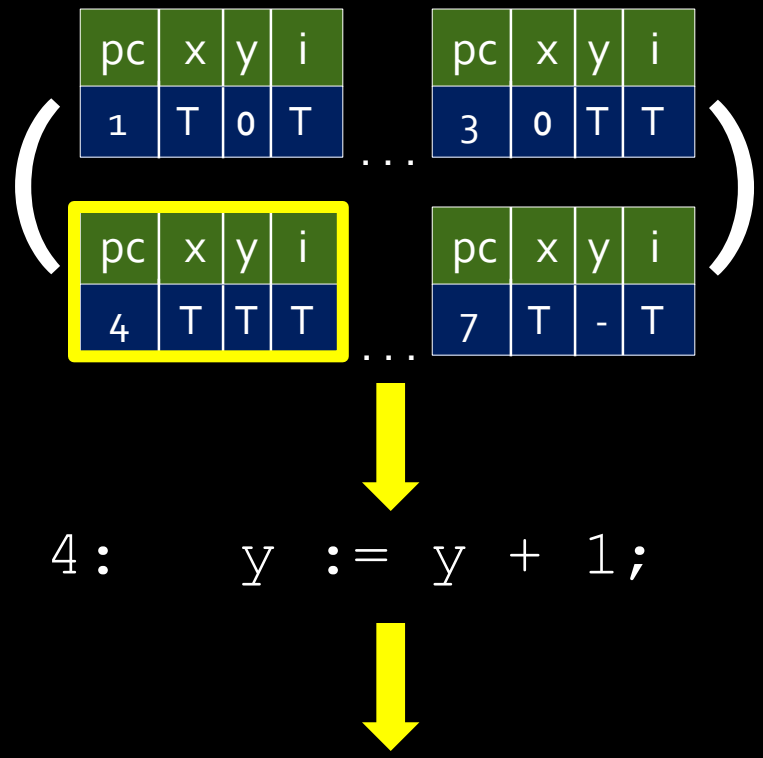
How about this ?

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



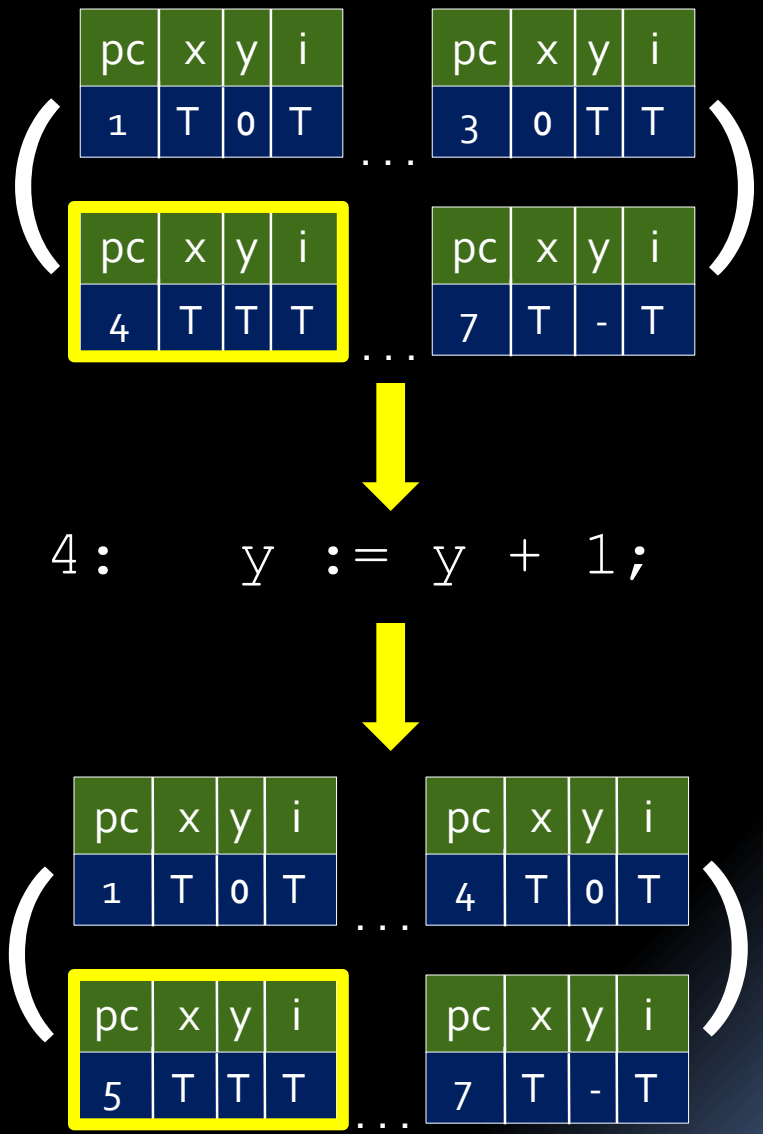
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



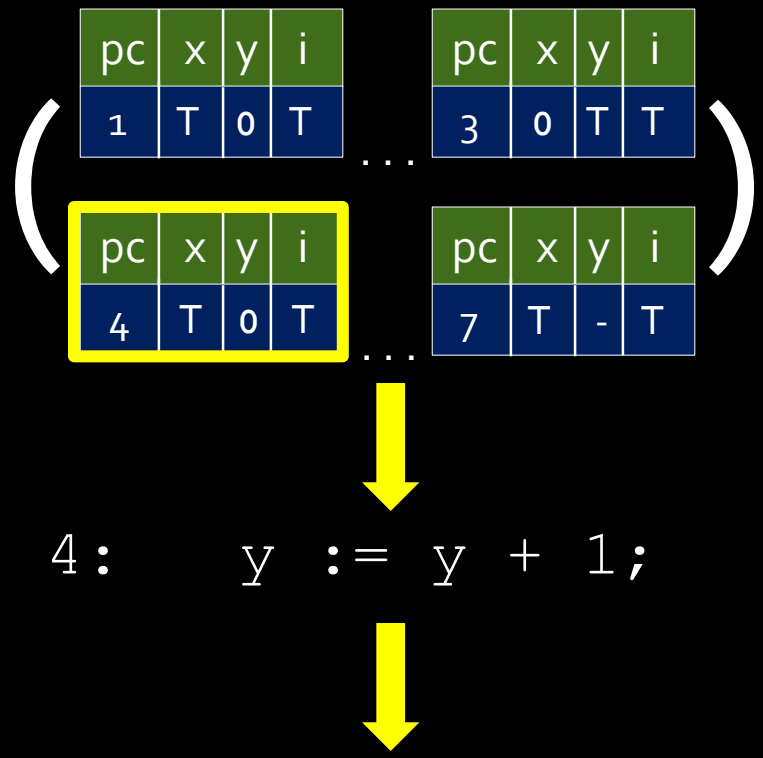
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



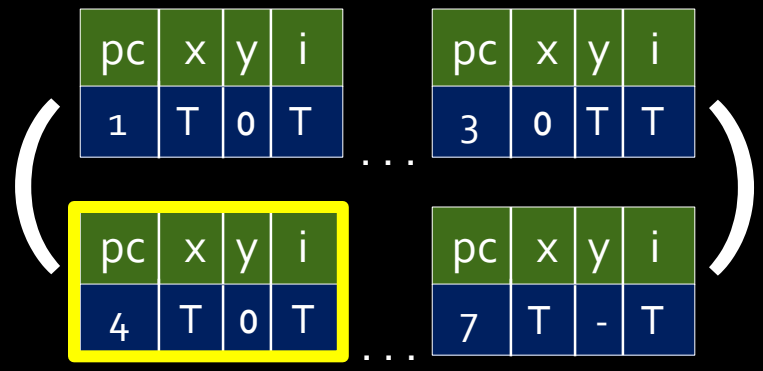
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

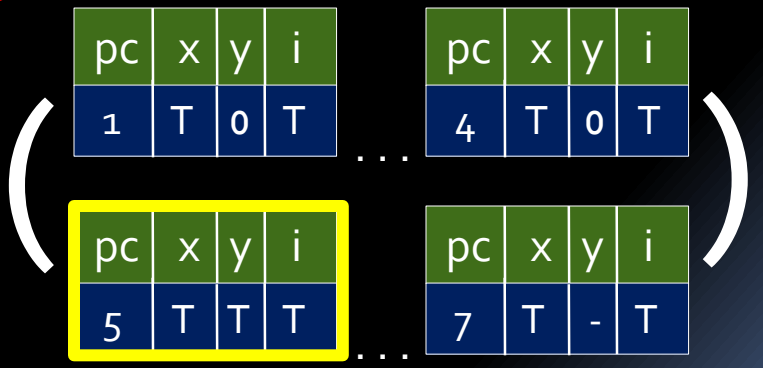


Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



4: y := y + 1;



Is this sound ? Yes
Is this precise ? No !

Imprecise Transformer

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

This abstract state:

pc	x	y	i
4	T	0	T

represents infinitely many concrete states where y is always **0** , including:

pc	x	y	i
4	1	0	2

If we perform `y := y + 1` on any of these concrete states, we will always get states where y is always positive, such as:

pc	x	y	i
5	1	1	2

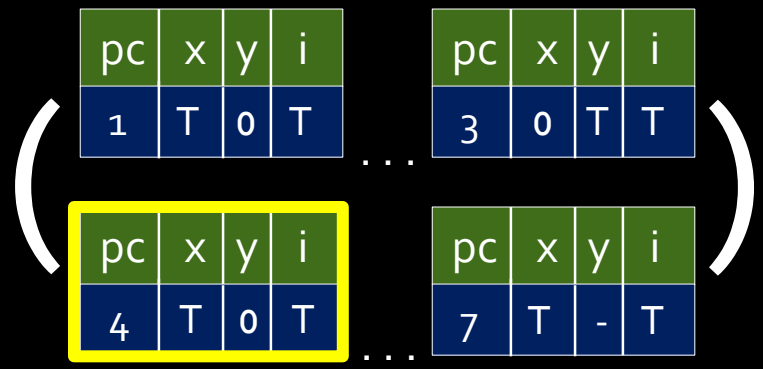
However, the abstract transformer produces an abstract state where y can be any value, such as:

pc	x	y	i
5	T	T	T

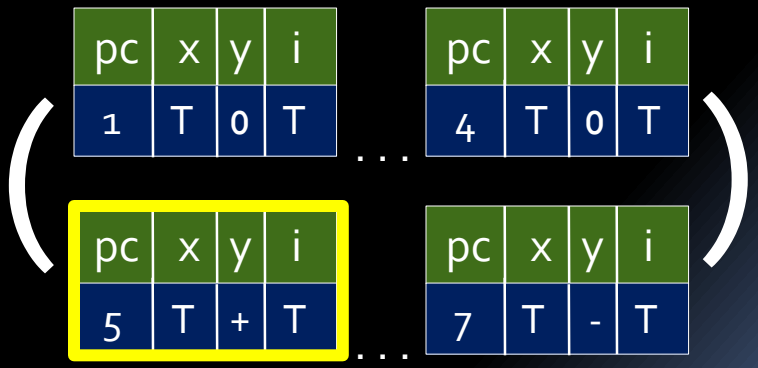
The abstract transformer is imprecise ! ☹

How about this ?

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



4: y := y + 1;



Is this sound ? Yes
Is this precise ? Yes !
Why ?

Abstract Transformers

It is easy to be sound and imprecise: simply output T

It is desirable to be both sound and precise.

If we lose precision, it needs to be clear why:

- sometimes, computing the most precise transformer (also called the best transformer) is impossible
- for efficiency reasons, we may sacrifice some precision

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

Start with the least abstract element

pc	x	y	i
1	⊥	⊥	⊥

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

Let us see a few iterations next

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	⊥

pc	x	y	i
2	⊥	⊥	⊥

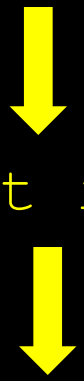
pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥



int i

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	⊥

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

int i

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	⊥	⊥	⊥

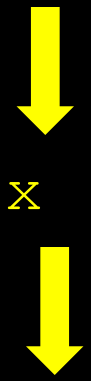
pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥



1: int x :=5;

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

1: int x :=5;

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

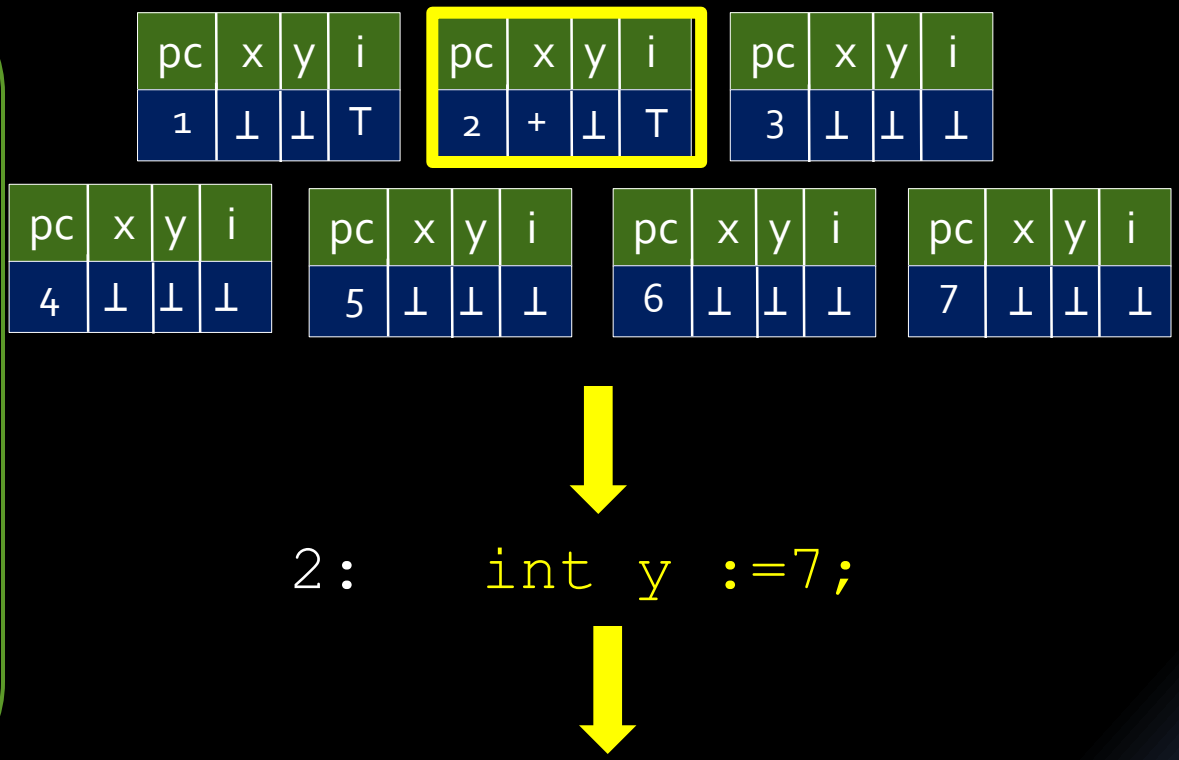
pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

2: int y :=7;

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

3: if (i ≥ 0)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

3: if (i ≥ 0)

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	+	+	-

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

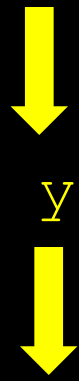
pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	+	+	-



4: y := y + 1;

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	+	+	-

4: y := y + 1;

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	+	+	-

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

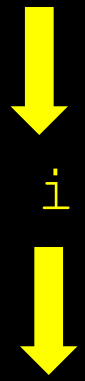
pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	+	+	-



5: i := i - 1;

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	+	+	-

5: i := i - 1;

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

6: goto 3;

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

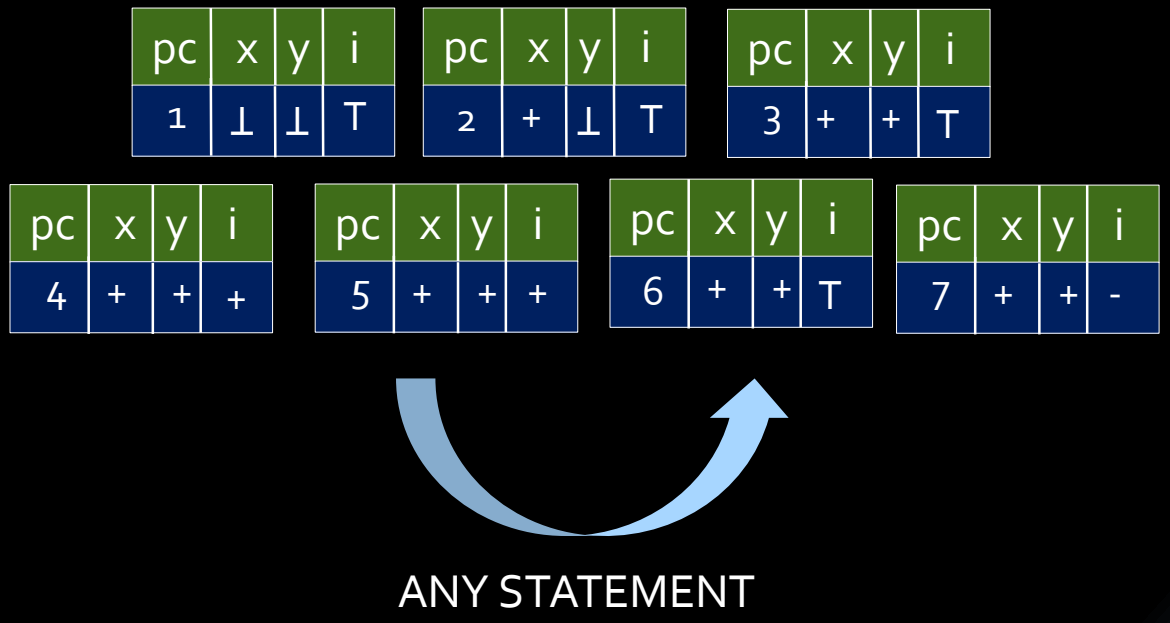
pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

Step 3: Reached a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



No matter what statement we execute from this state, we reach that state

Step 4: Check property

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

$P \models (0 \leq x + y)$ ✓

$P \models_{sign} (0 \leq x + y)$ ✓

sign domain precise enough to prove property

Lets change the property

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert
   0 ≤ x - y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

$P \not\models (0 \leq x - y)$ ✗

$P_{sign} \not\models (0 \leq x - y)$ ✗

sign domain precise enough to disprove spec

Lets change the property again

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert
   0 ≤ y - x
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

$P \models (0 \leq y - x)$ ✓

$P \not\models_{\text{sign}} (0 \leq y - x)$ ✗

sign domain too imprecise
to prove spec !

Abstraction

- more abstract domains:
  - In exercises: parity, intervals,
  - octagon, polyhedra, heap,…
  - custom abstractions

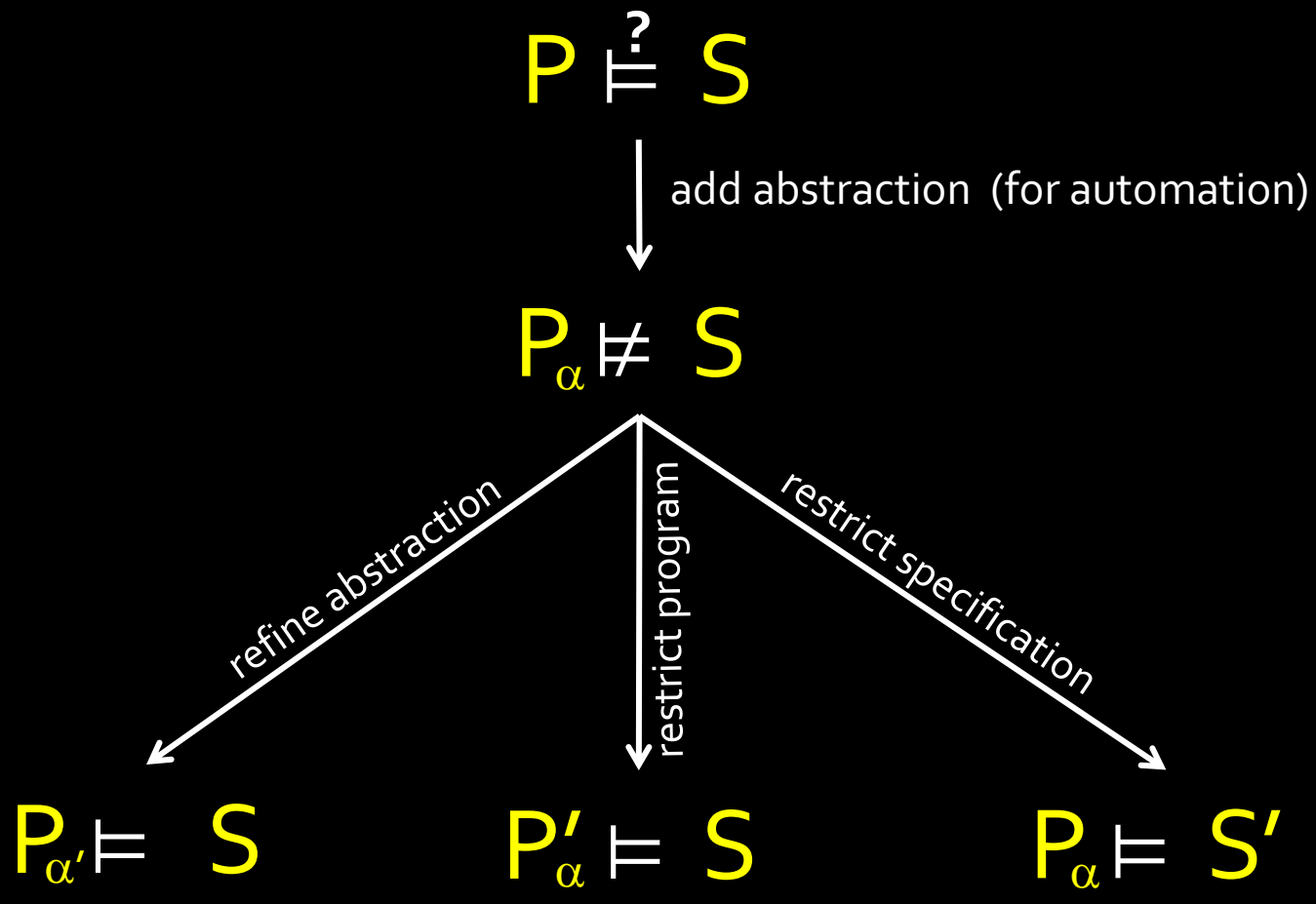
Questions that should bother you

- How do we describe abstract domains mathematically ?
- How do we discover best/sound abstract transformers ? What does best mean ?
- What is the function that we iterate to a fixed point ?
- How do we ensure termination of the analysis ?

Plan

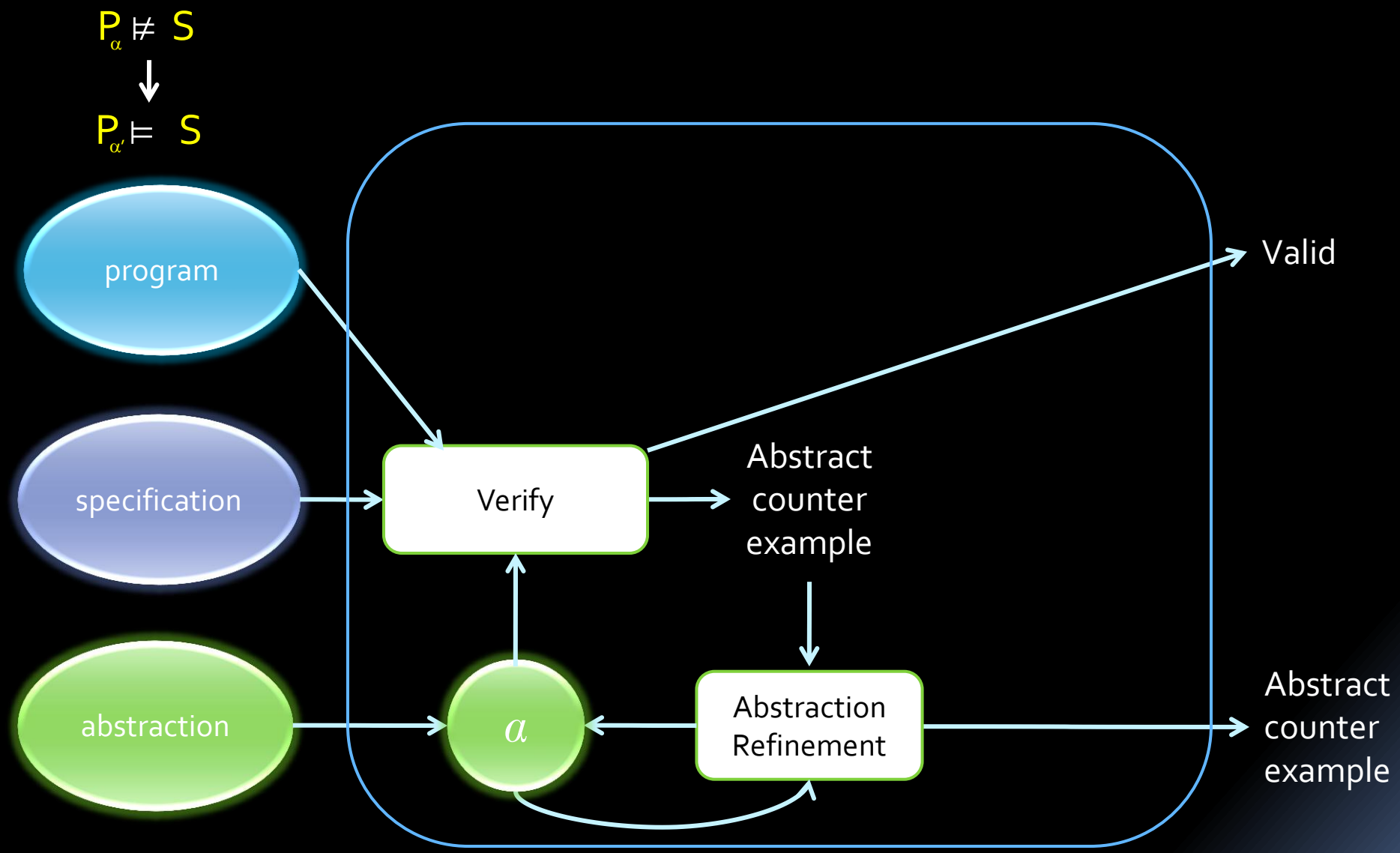
- Setting
- Program analysis by abstract interpretation
- Synthesis based on abstract interpretation
- Analysis + synthesis for weak memory models

Recall our Setting

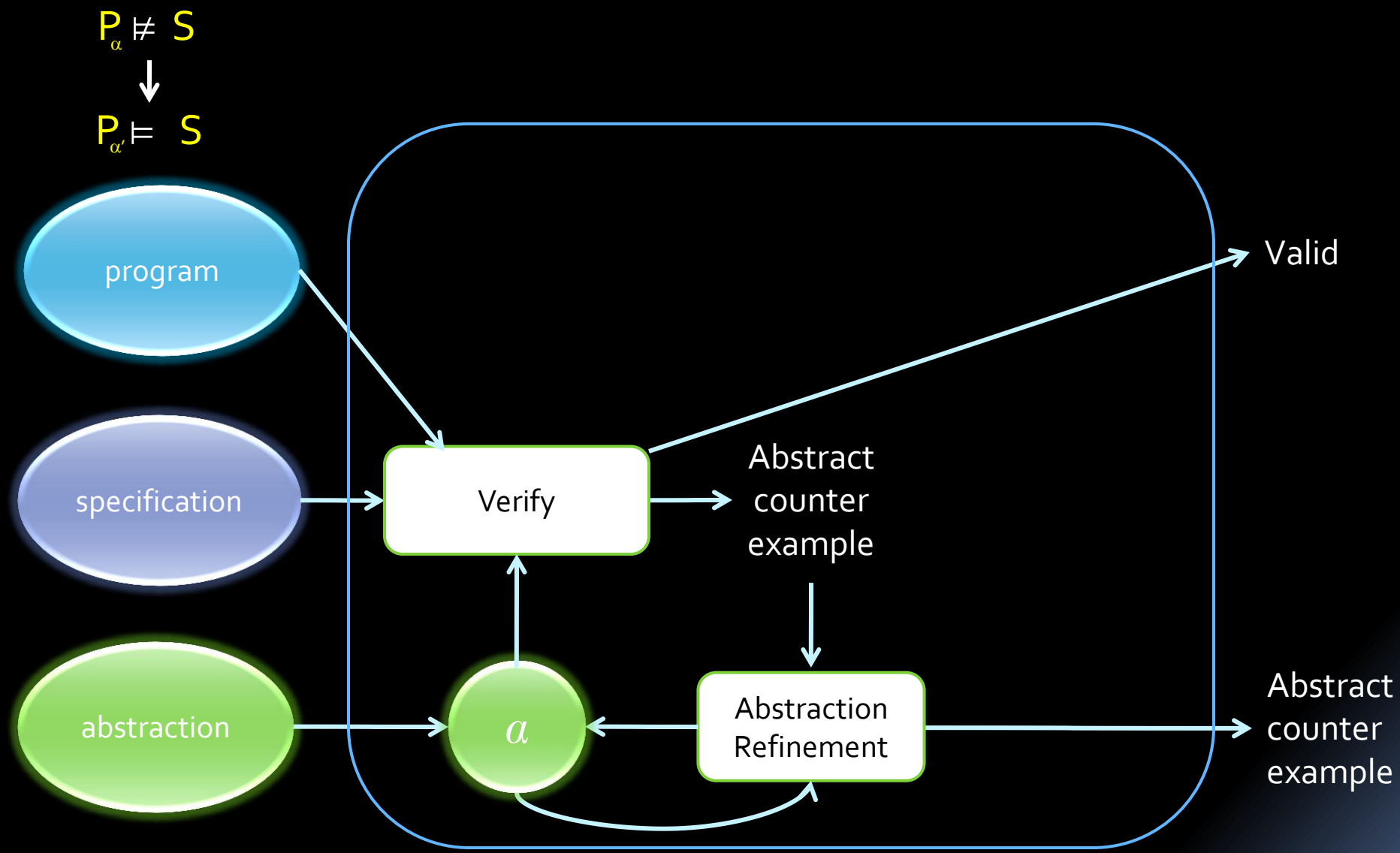


can also combine steps

Refine Abstraction

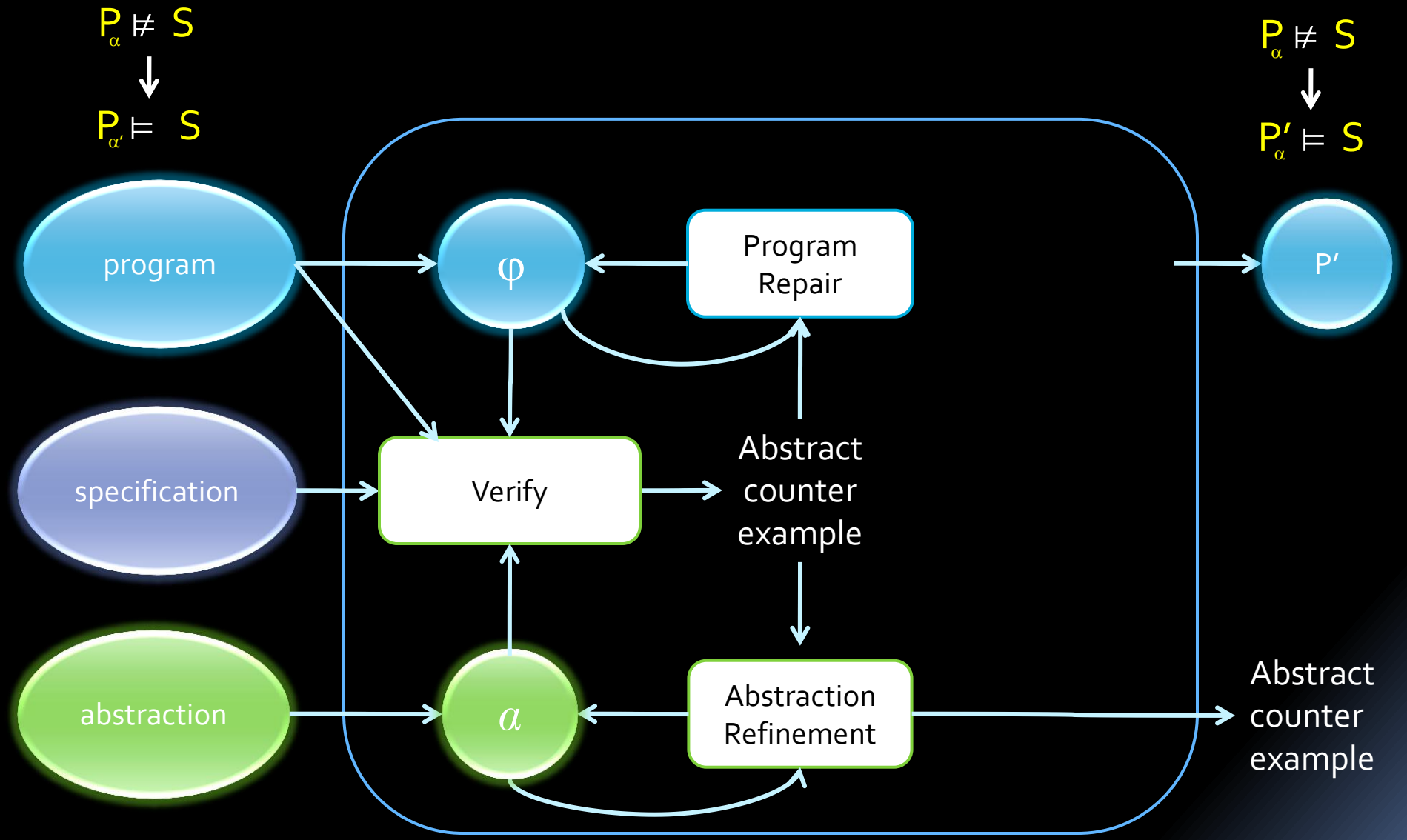


Refine Abstraction

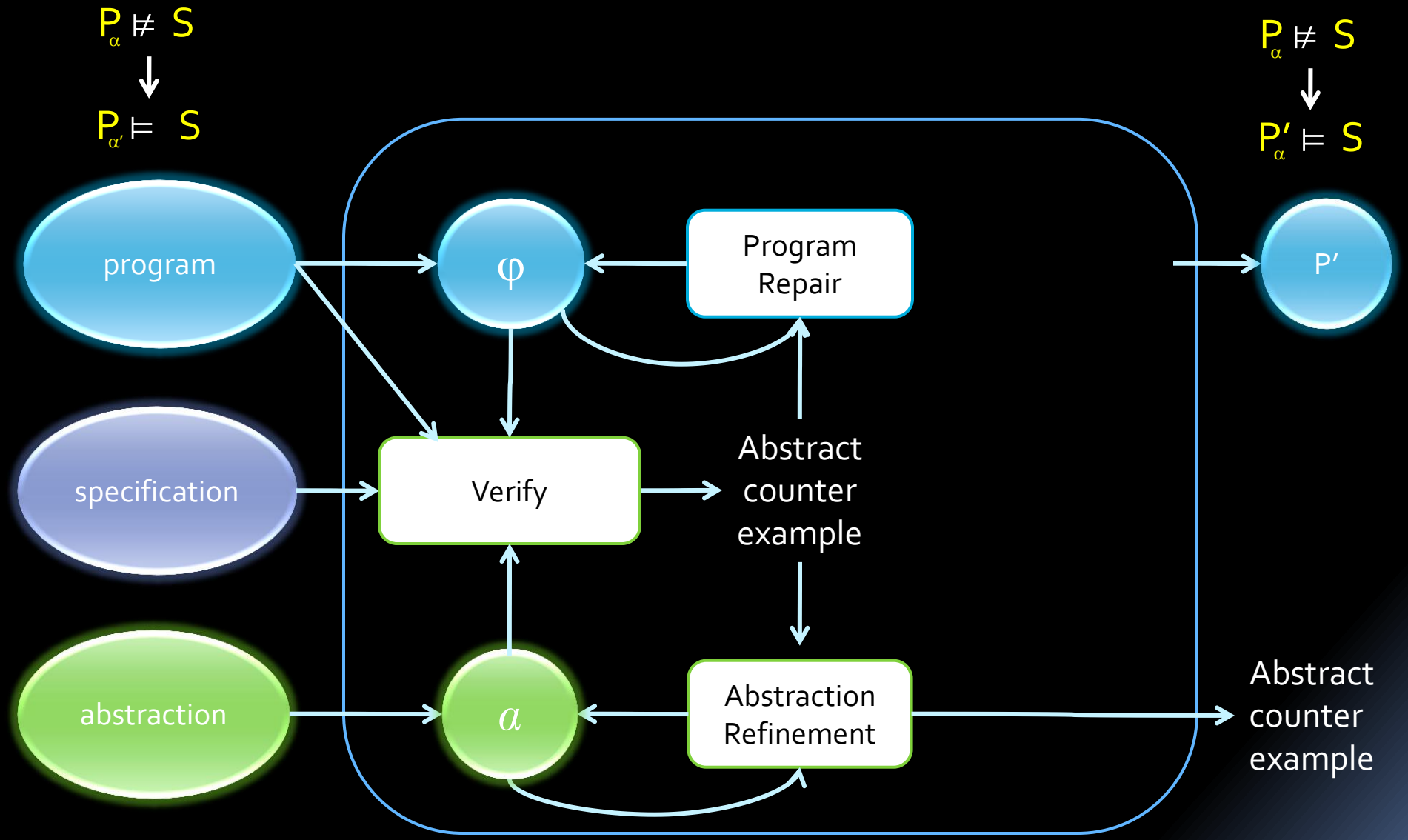


Change the **abstraction** to match the **program**

Refine Abstraction or Repair Program

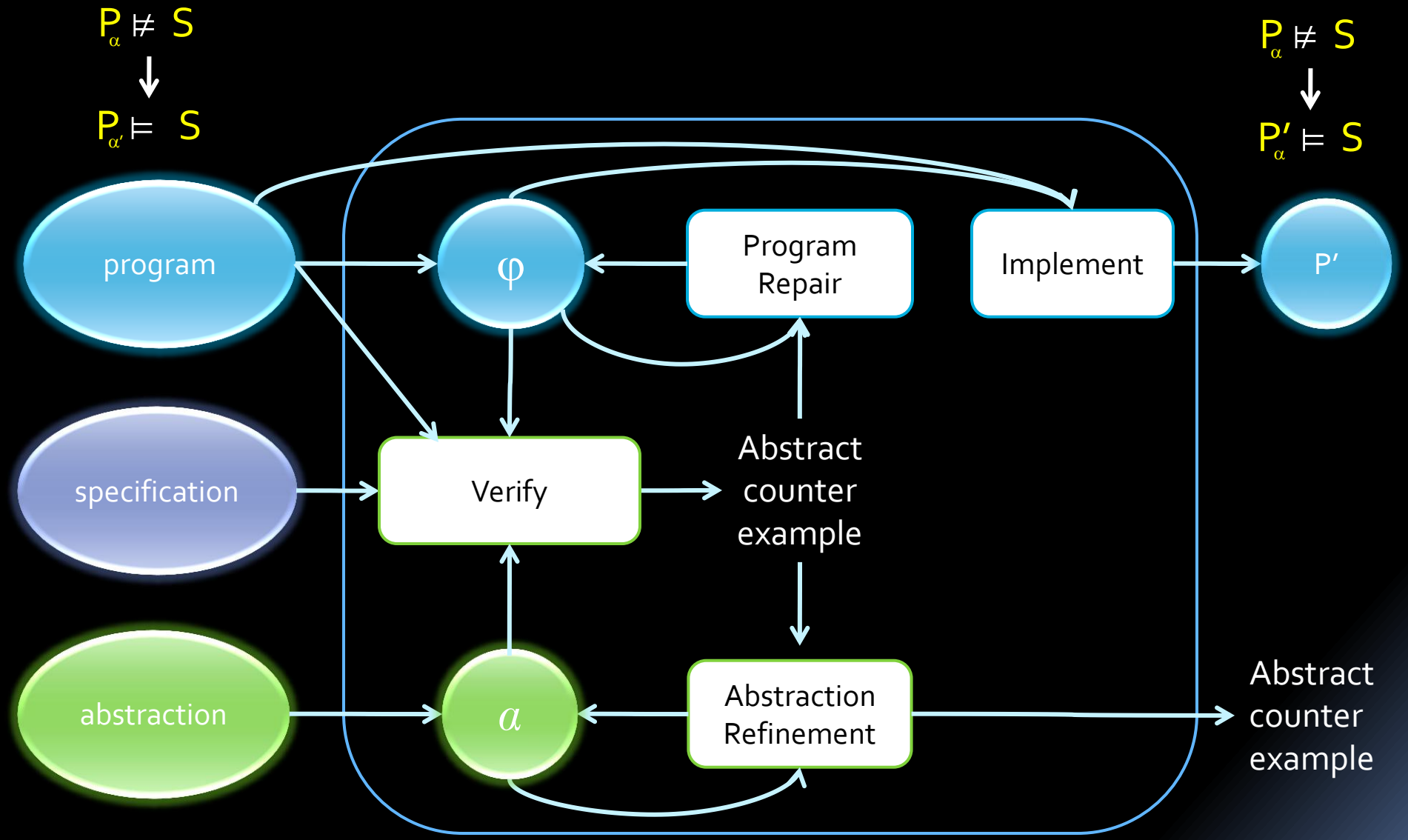


Refine Abstraction or Repair Program



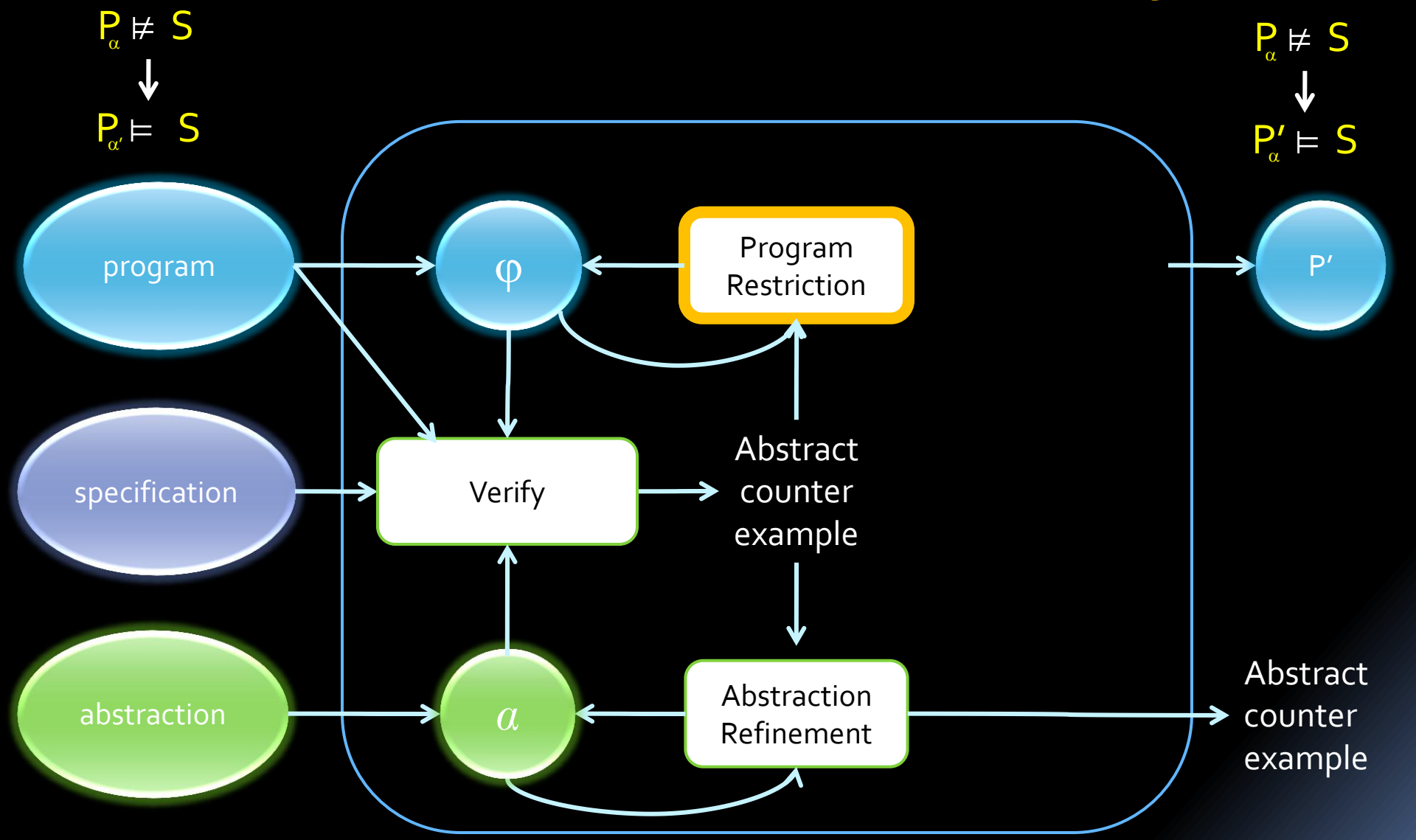
Change the **program** to match the **abstraction**

Refine Abstraction or Repair Program

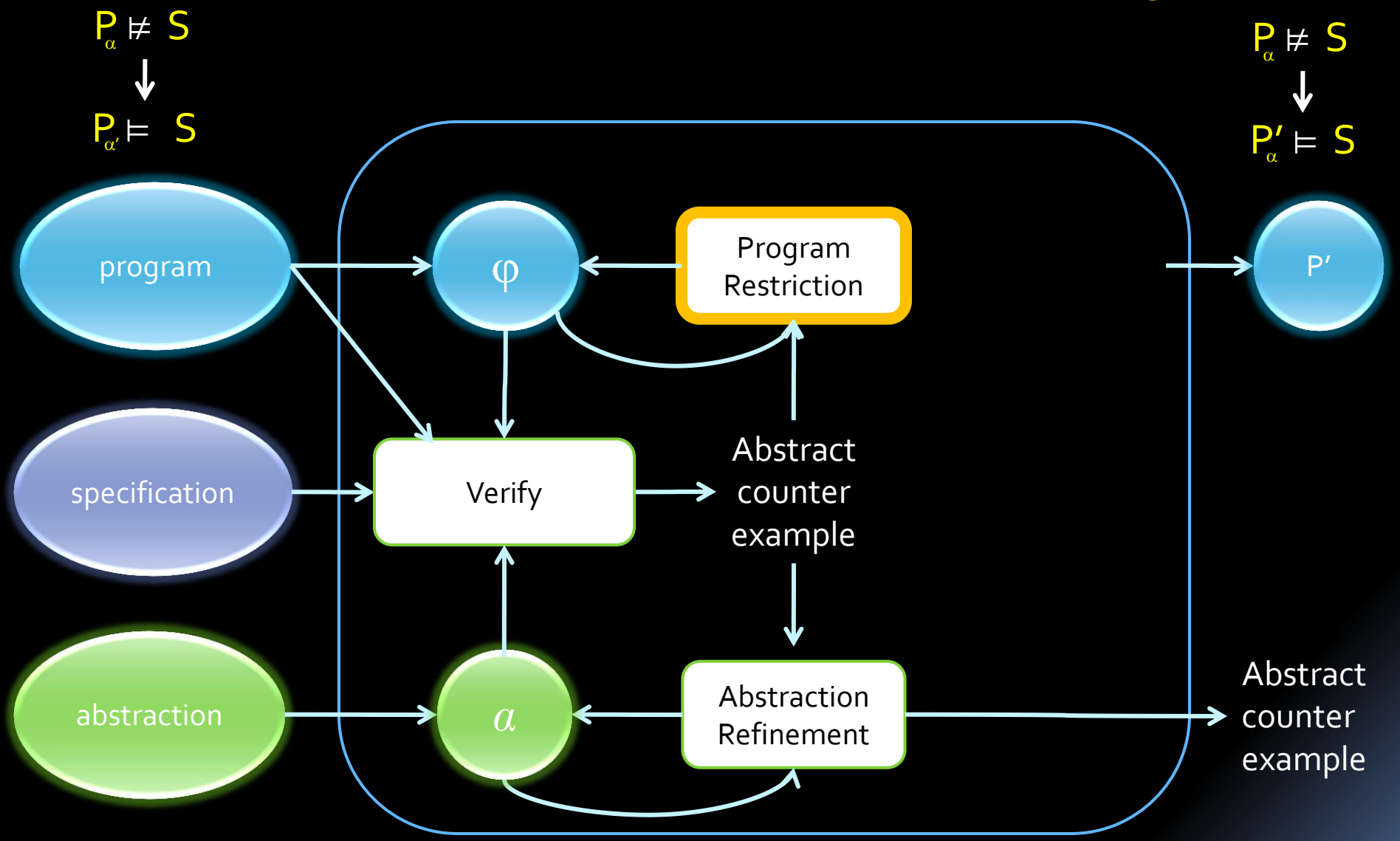


Change the **program** to match the **abstraction**

Instantiate for Concurrency

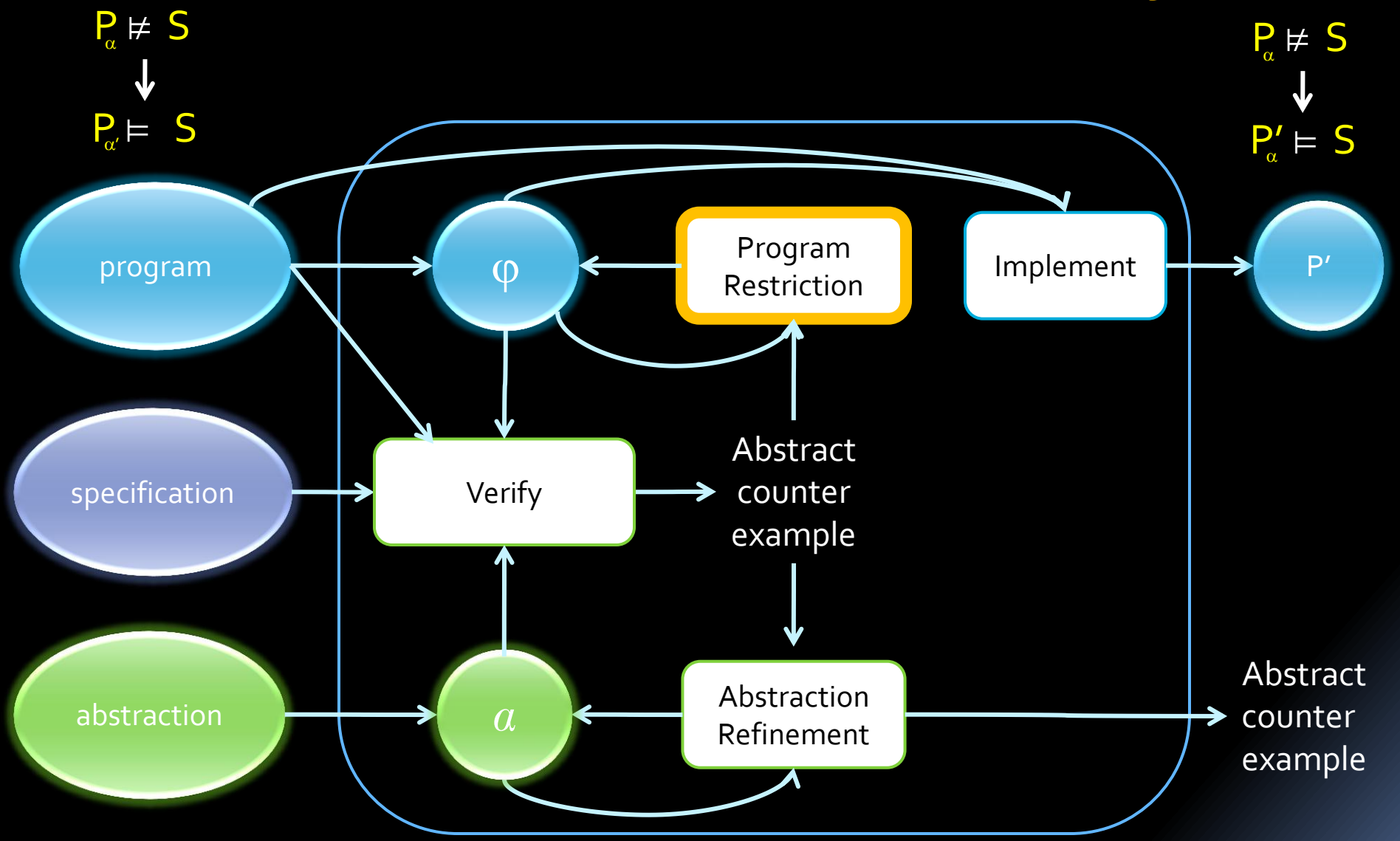


Instantiate for Concurrency



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

Instantiate for Concurrency



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

Instantiate for Concurrency

Restrict the program by introducing synchronization

How to synchronize processes to achieve correctness and efficiency?

Synchronization Primitives

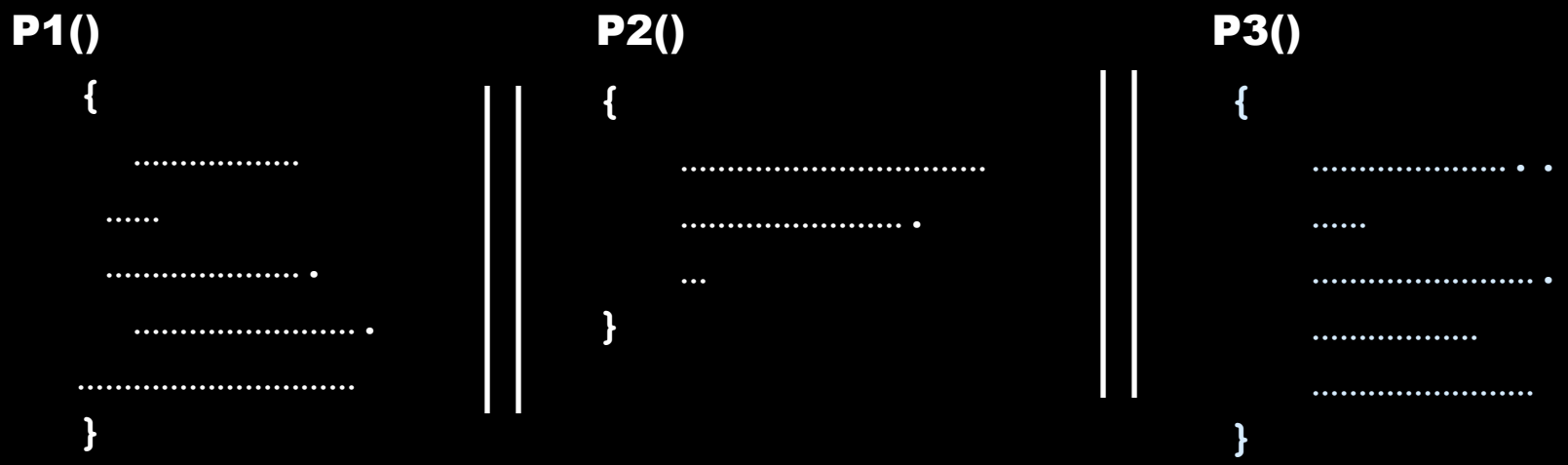
- Atomic sections
- Conditional critical region (CCR)
- Memory barriers (fences)
- CAS
- Semaphores
- Monitors
- Locks
- ….

Synchronization Primitives

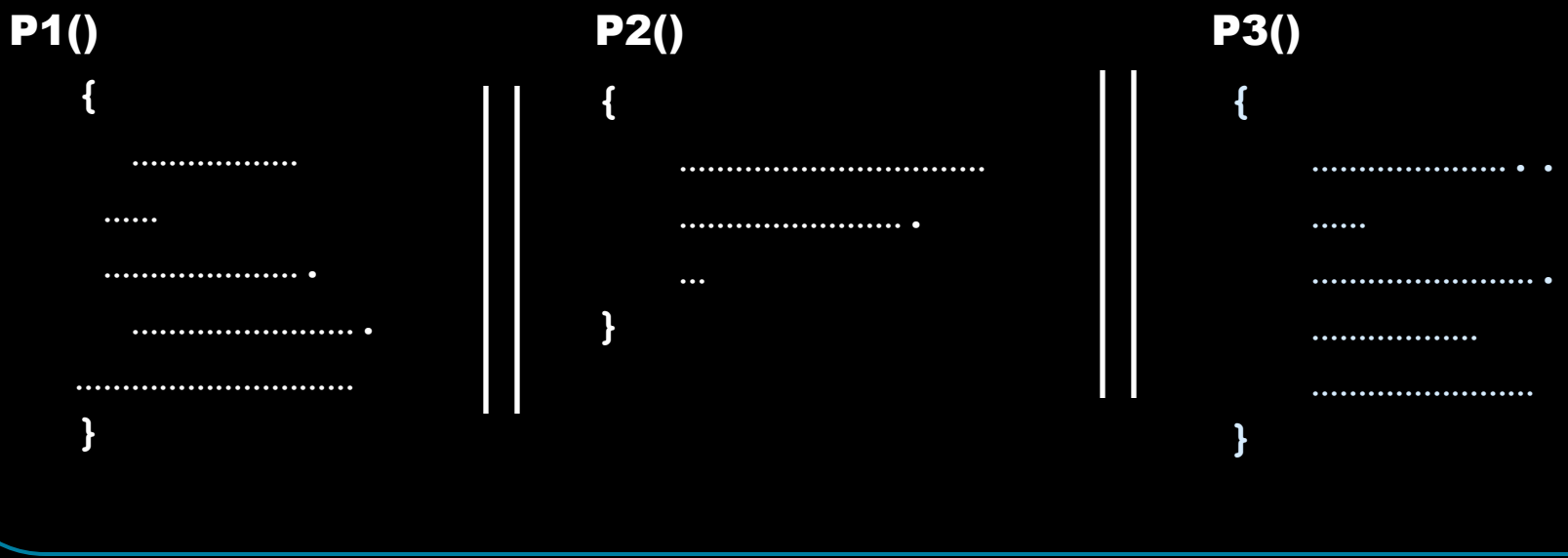
- Atomic sections
- Conditional critical region (CCR)
- Memory barriers (fences)
- CAS
- Semaphores
- Monitors
- Locks
- ….

Example: Correct and Efficient Synchronization with Atomic Sections

Example: Correct and Efficient Synchronization with Atomic Sections

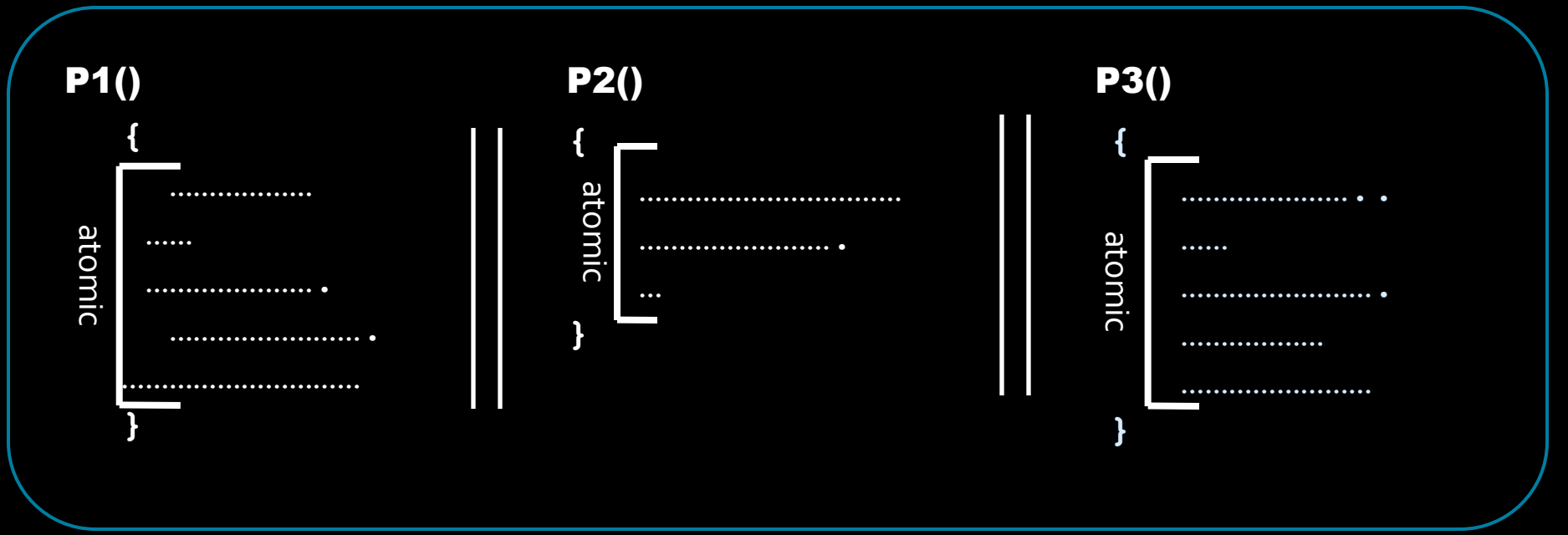


Example: Correct and Efficient Synchronization with Atomic Sections



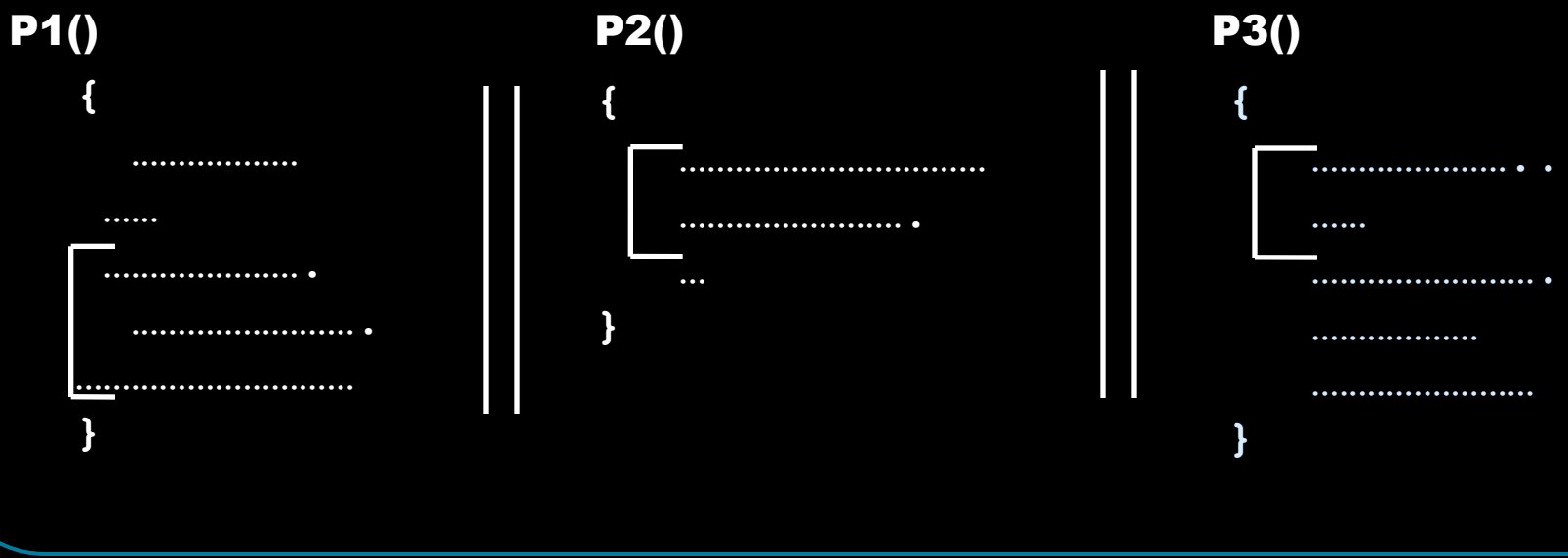
Safety Specification: S

Example: Correct and Efficient Synchronization with Atomic Sections



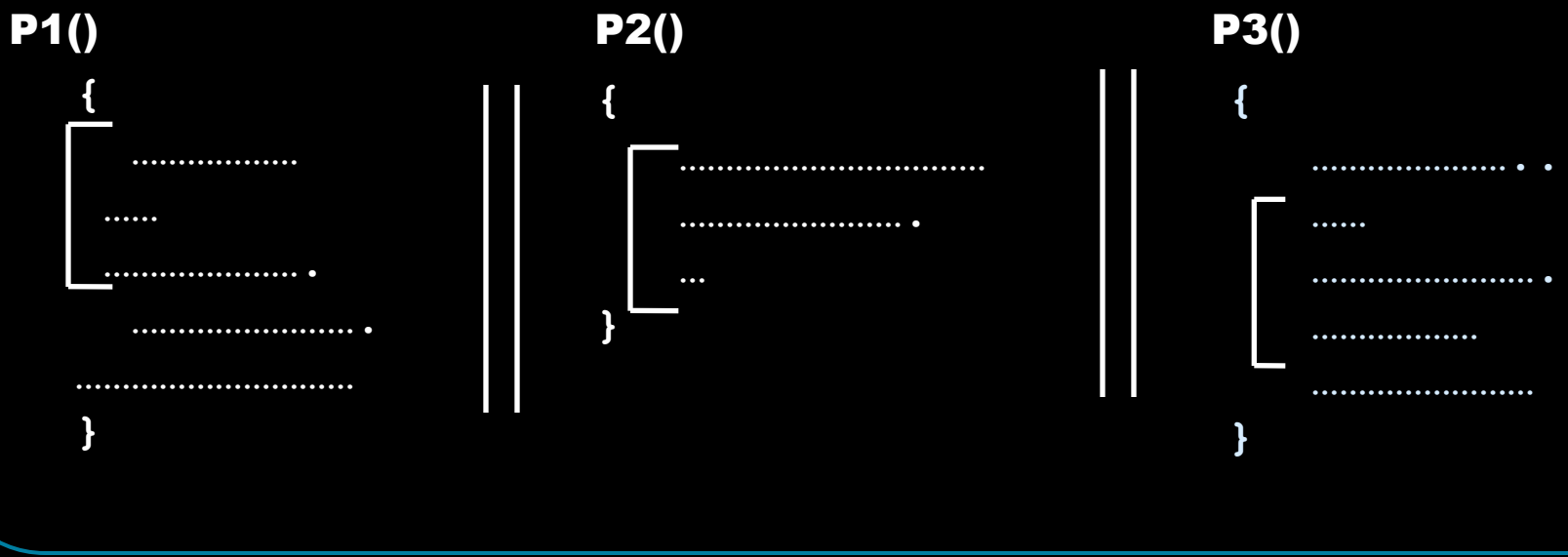
Safety Specification: S

Example: Correct and Efficient Synchronization with Atomic Sections



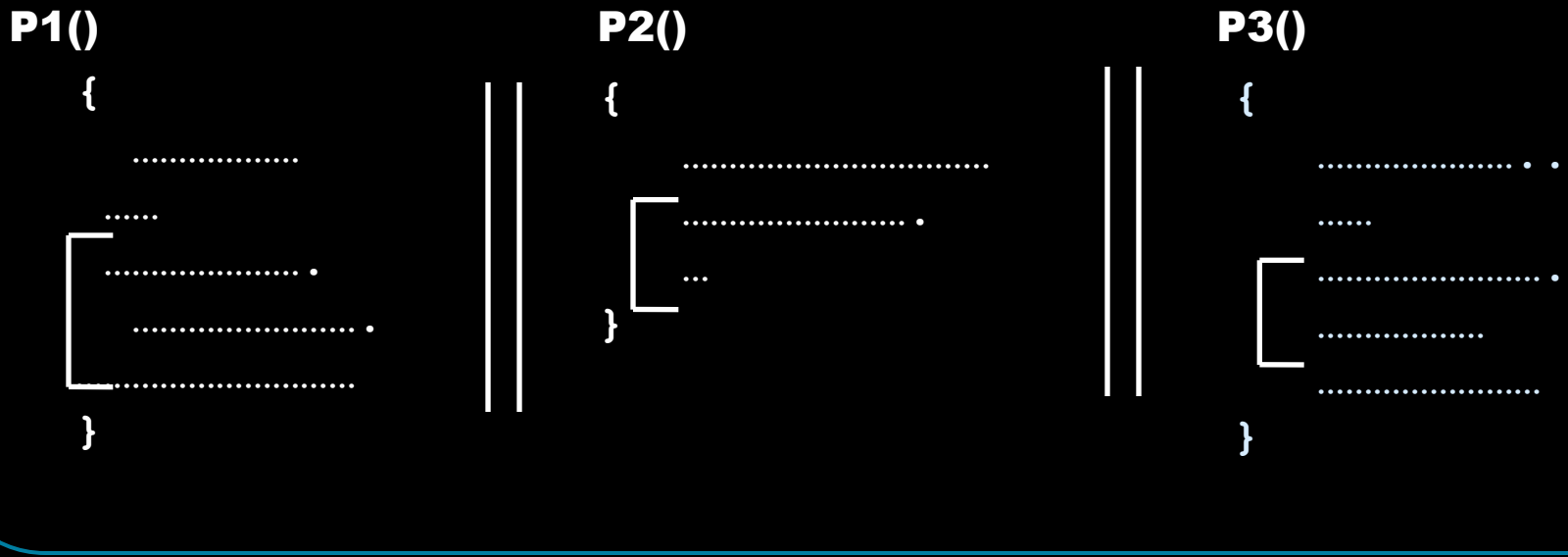
Safety Specification: S

Example: Correct and Efficient Synchronization with Atomic Sections



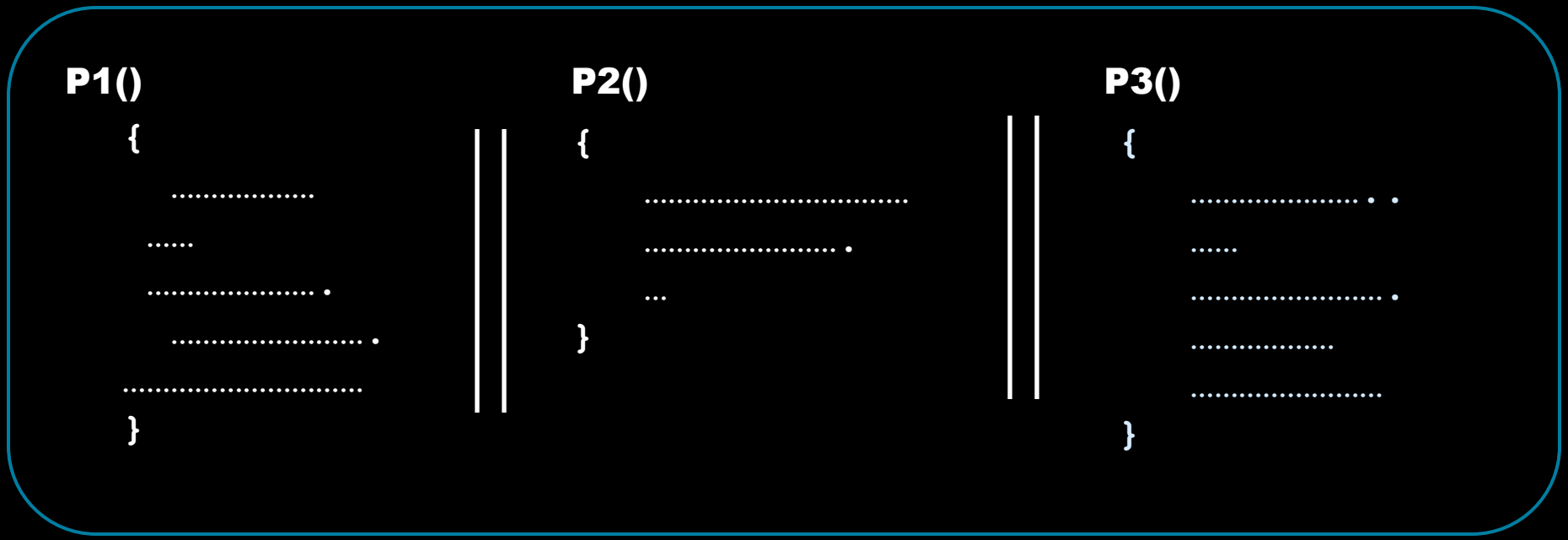
Safety Specification: S

Example: Correct and Efficient Synchronization with Atomic Sections



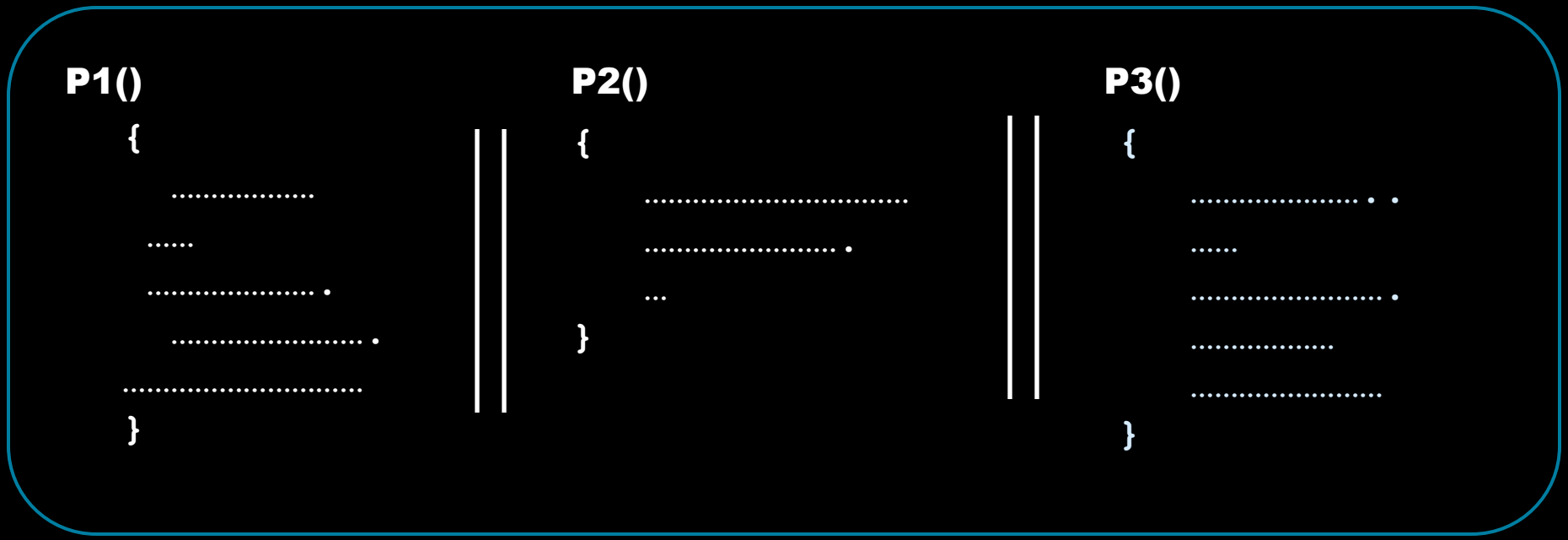
Safety Specification: S

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

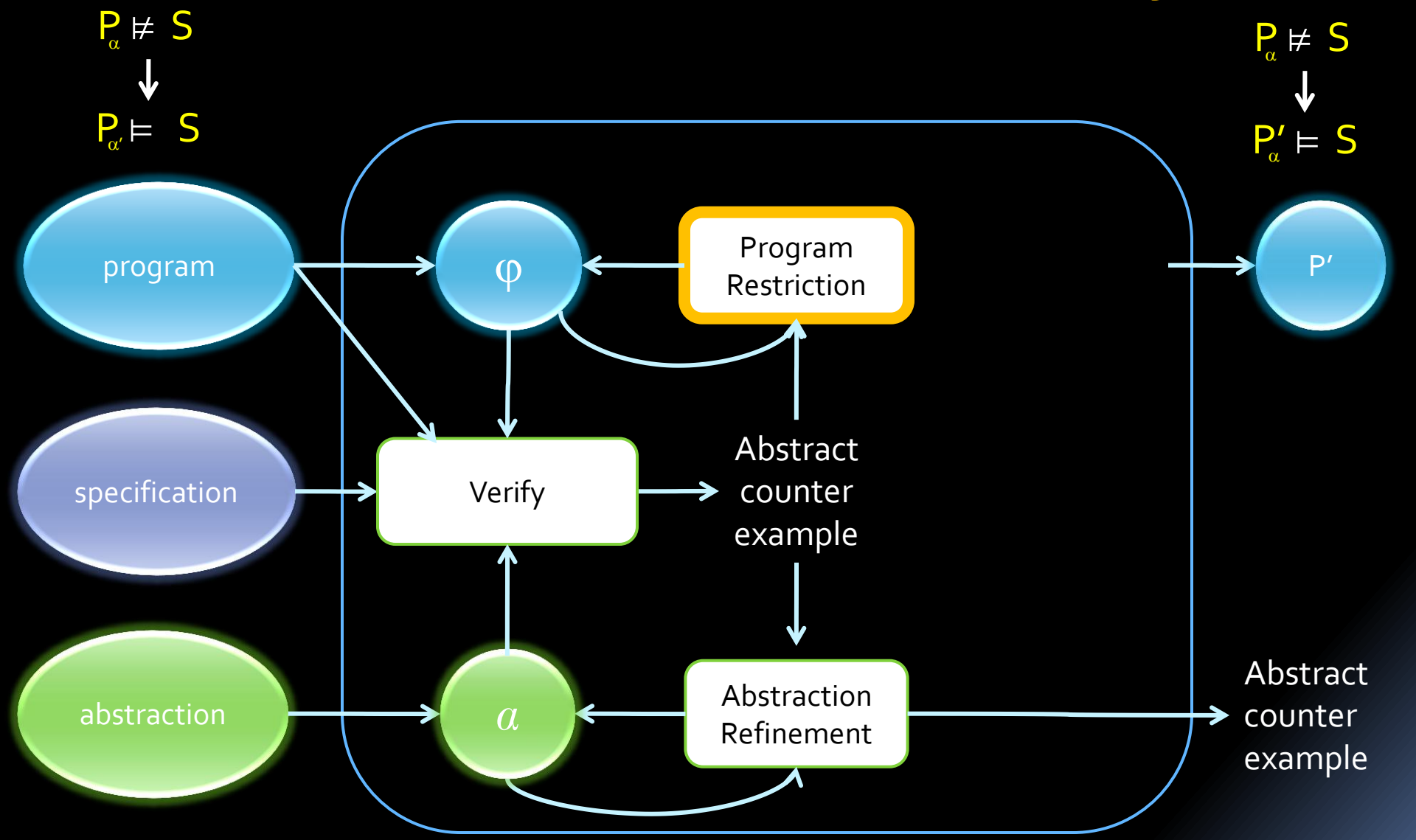
- Find minimal synchronization that makes the program satisfy the specification
  - Avoid all bad interleavings while permitting as many good interleavings as possible
- Assumption: we can prove that serial executions satisfy the specification
  - Interested in bad behaviors due to concurrency
- Handle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

Abstraction-Guided Synthesis of Synchronization

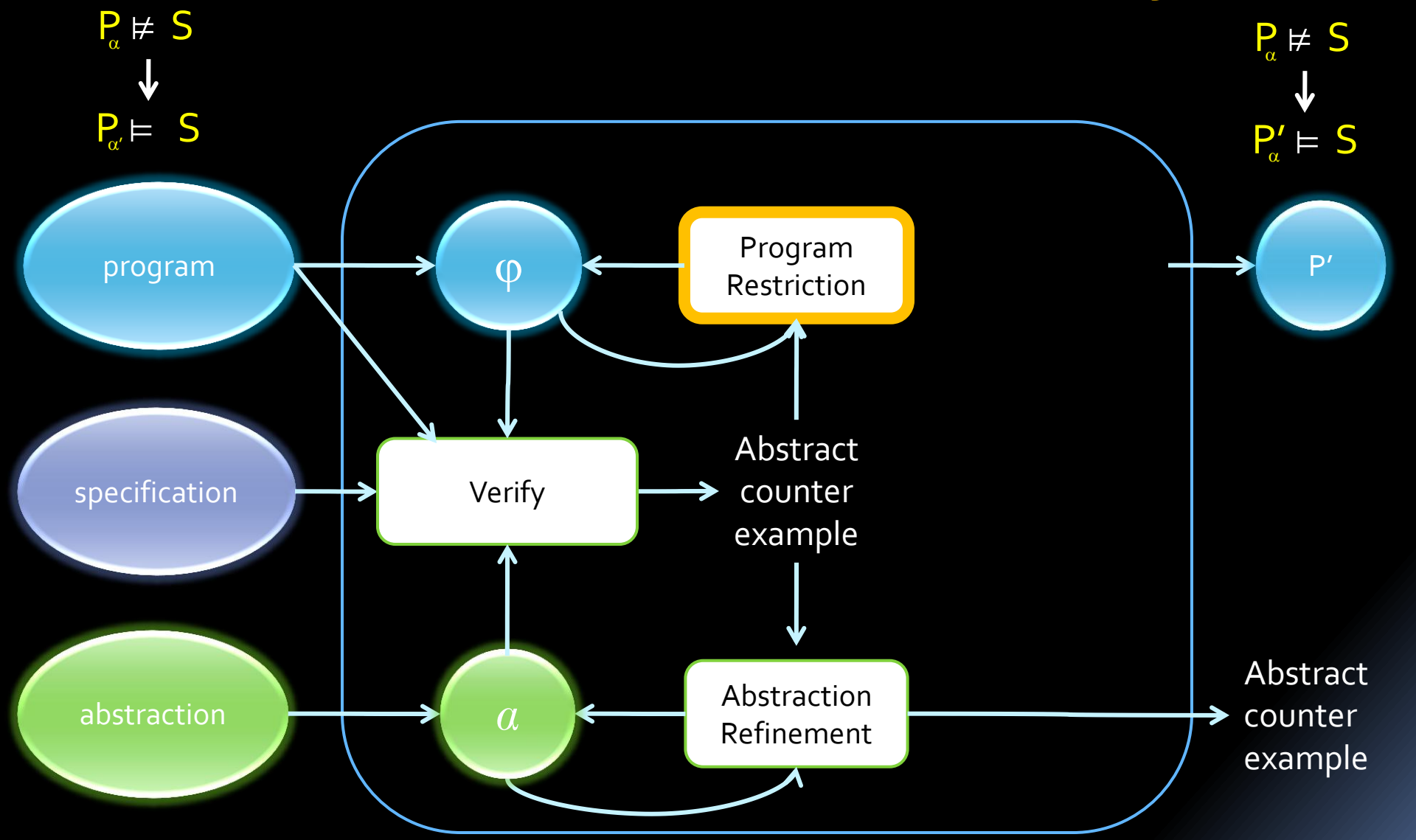
- Synthesis of synchronization via abstract interpretation
  - Compute over-approximation of all possible program executions
  - Add minimal synchronization to avoid (over-approximation of) bad interleavings

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

Instantiate for Concurrency

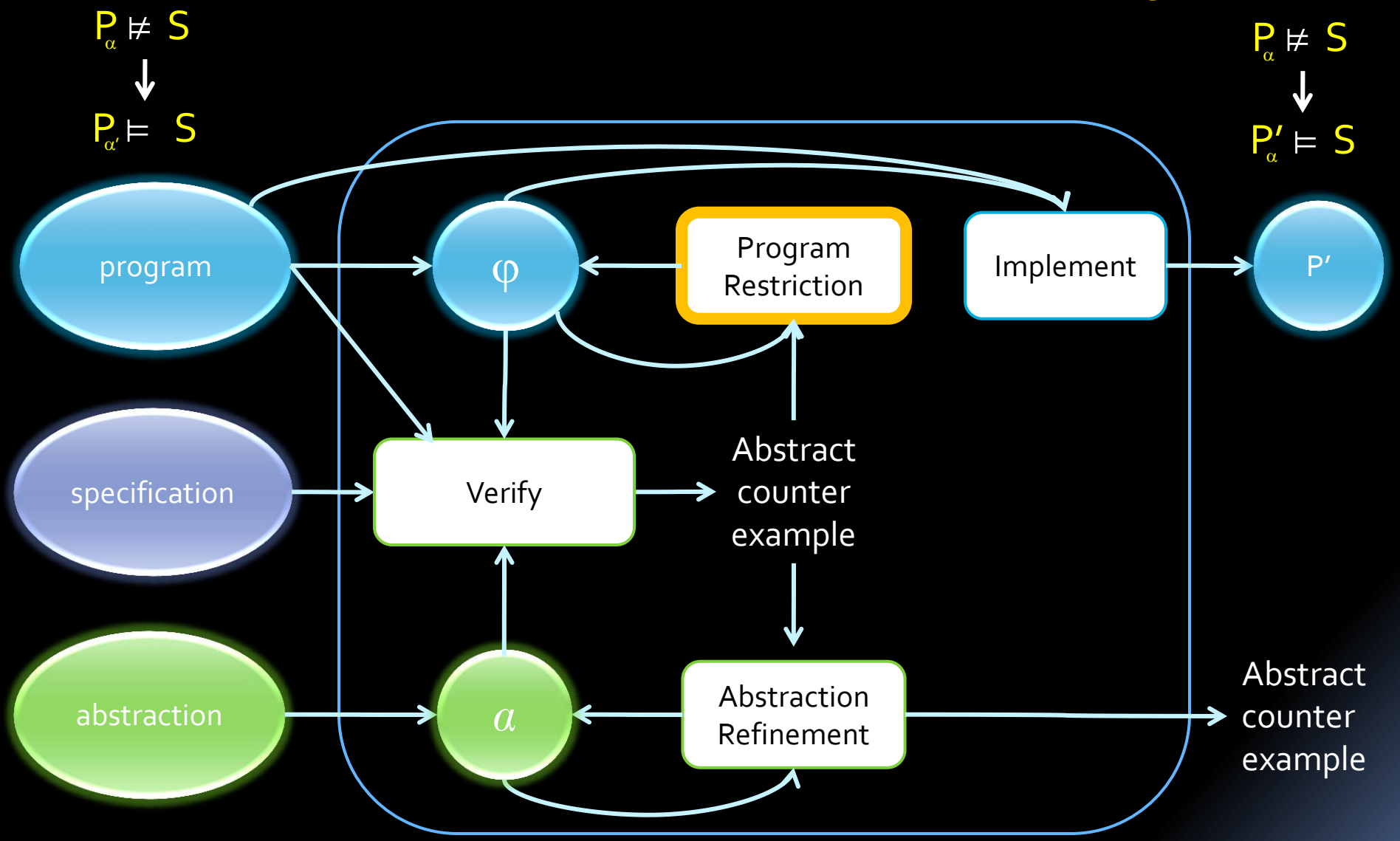


Instantiate for Concurrency



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

Instantiate for Concurrency



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
  }
}
```

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
  }
}
```

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
  }
}
```

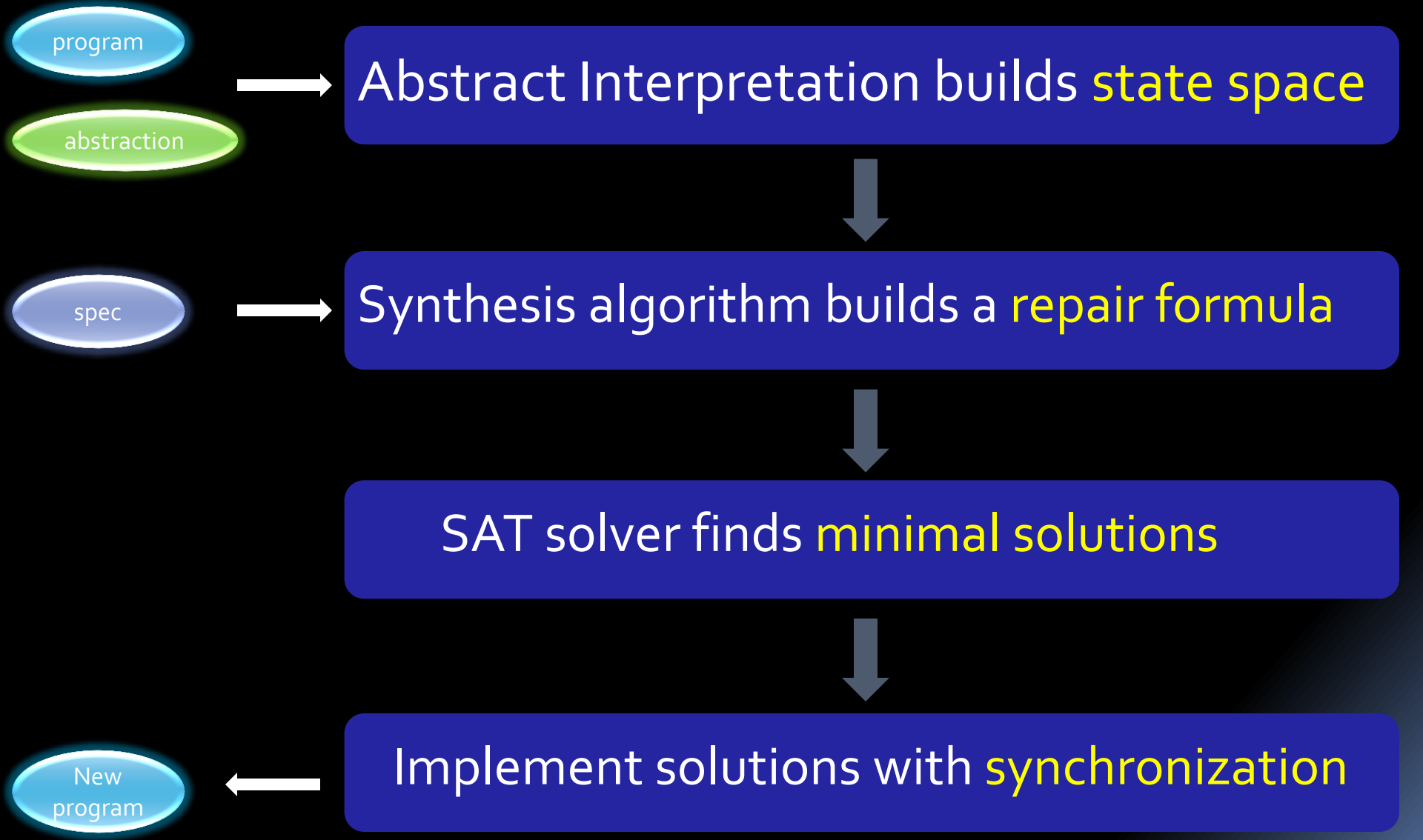
AGS Algorithm – High Level

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
  }
}
```

Flow of Synthesis



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

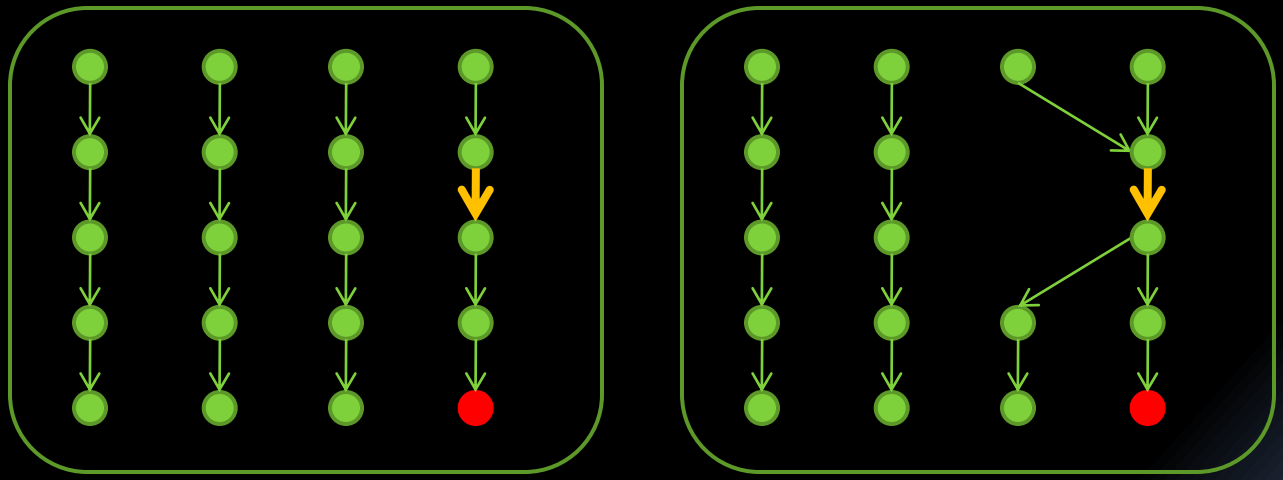
Thread A	Thread B
A1	B1
A2	B2

$\pi = A_1 B_1 A_2 B_2$

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

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

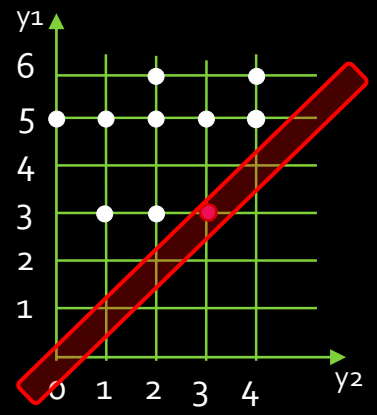


Example

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

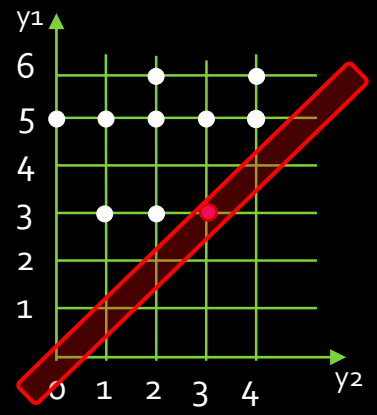


Concrete values

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

T1

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

||

T2

1: z++
2: z++

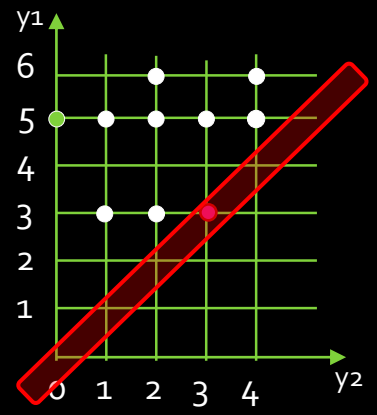
||

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: Concrete Values



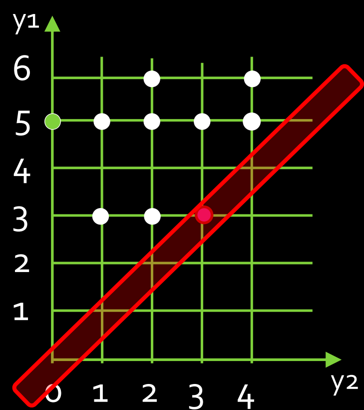
Concrete values

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

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



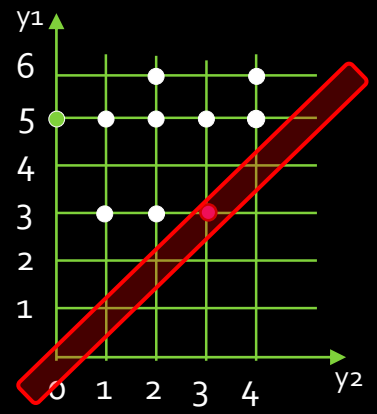
Concrete values

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
  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++

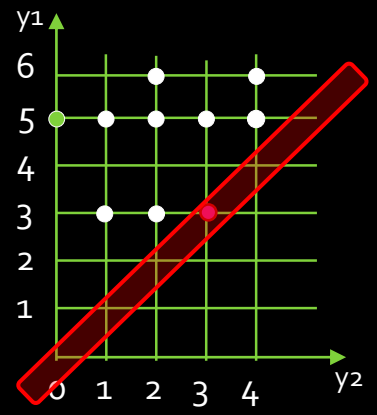
||

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: 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++

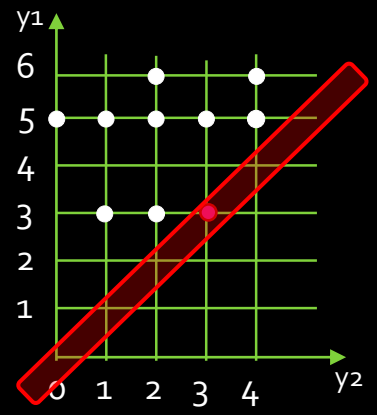
||

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

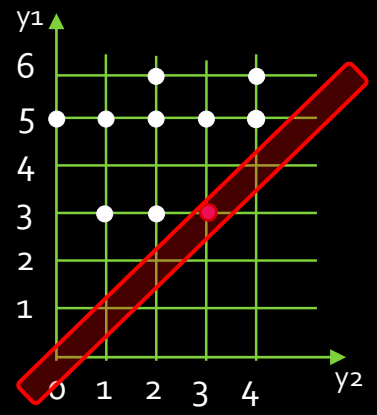


Concrete values

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: Parity Abstraction



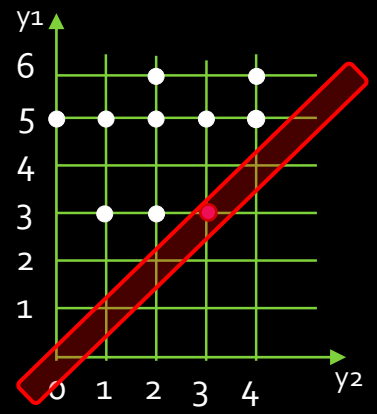
Concrete values

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

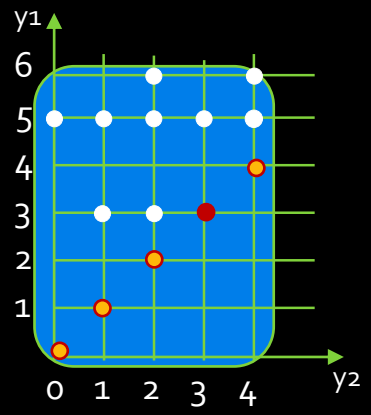
T1	T2	T3
1: x += z 2: x += z	1: z++ 2: z++	1: y1 = f(x) 2: y2 = x 3: assert(y1 != y2)

```
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)
  }
}
```

φ = true

Example: Avoiding Bad Interleavings

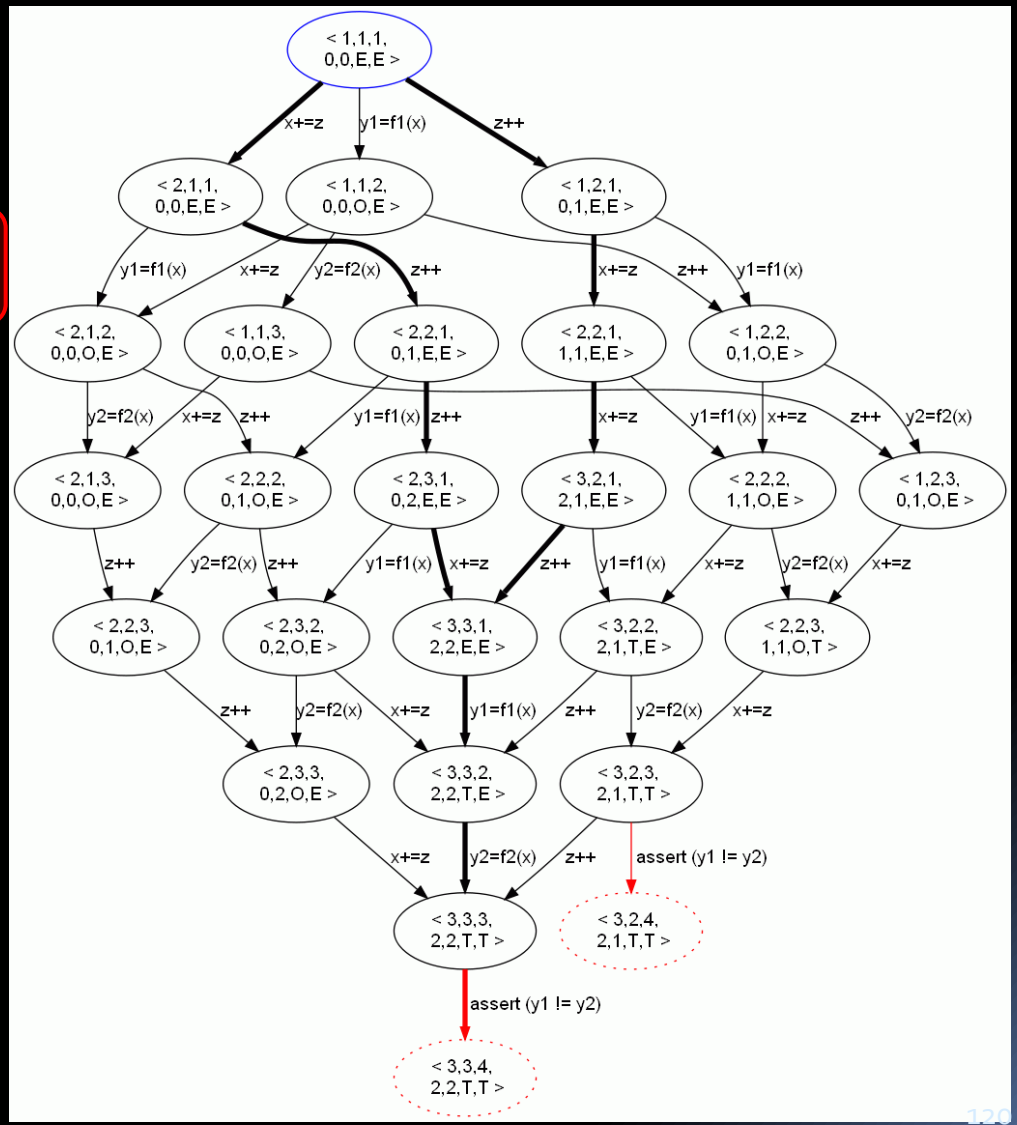
```
φ = true
while(true) {
   BadTraces={π|π∈([[P]]_α ∩ [[φ]]) and
                π ∉ S }
   if (BadTraces is empty)
       return implement(P,φ)
   select π ∈ BadTraces
   if (?) {
     φ = φ ∧ avoid(π)
   } else {
     α = refine(α, π)
   }
}
```

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)
  }
}
```

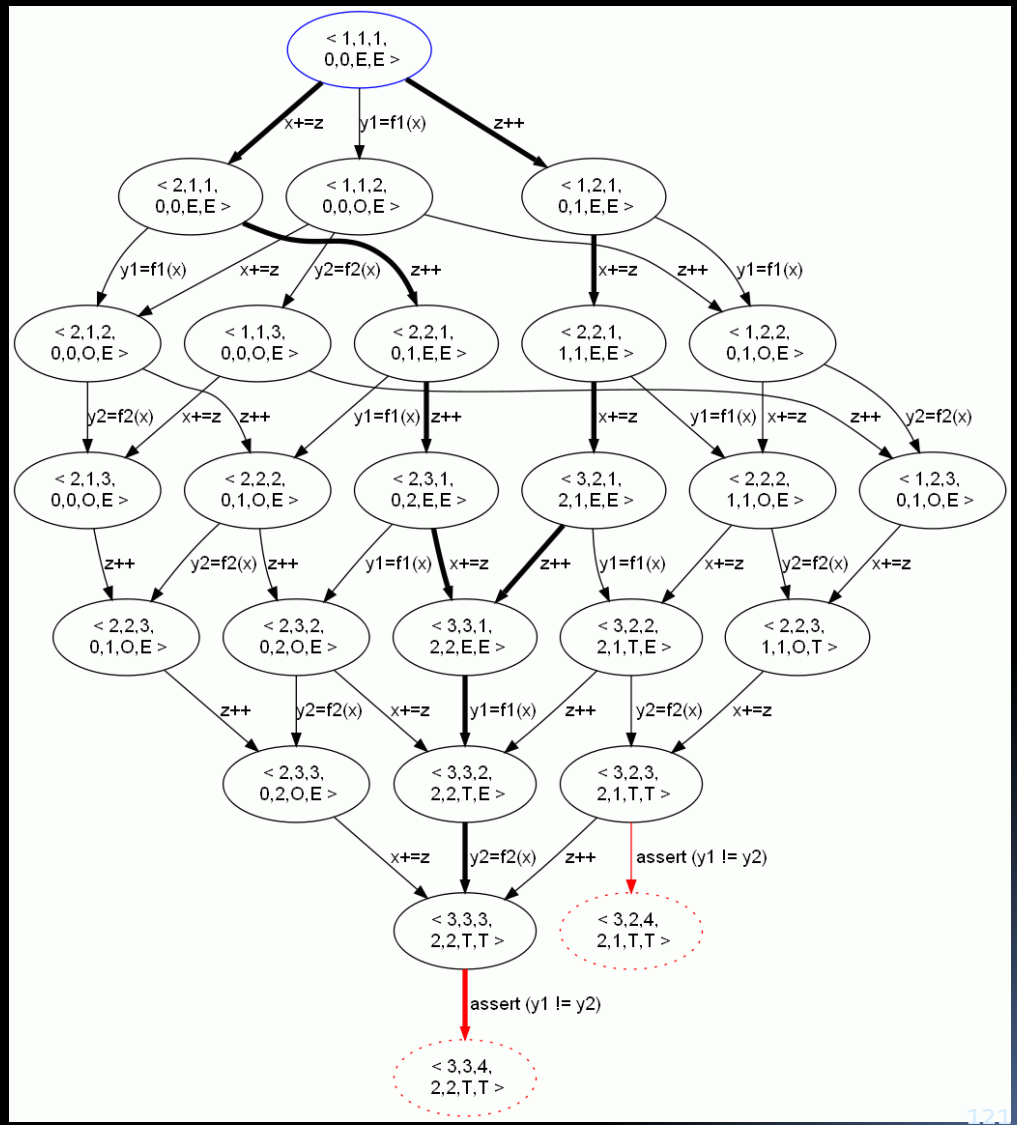
φ = true



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)
  }
}
```

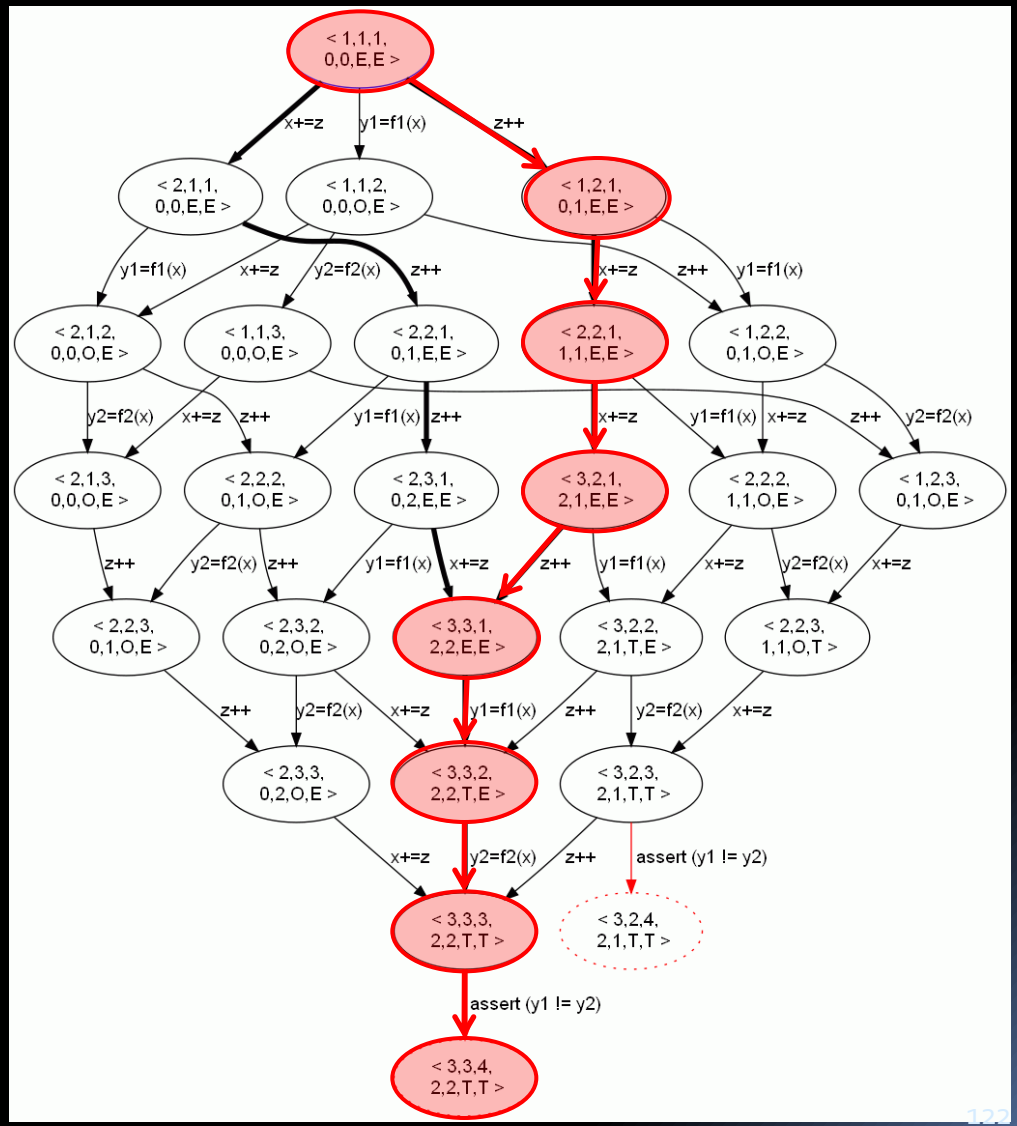
φ = true



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)
  }
}
```

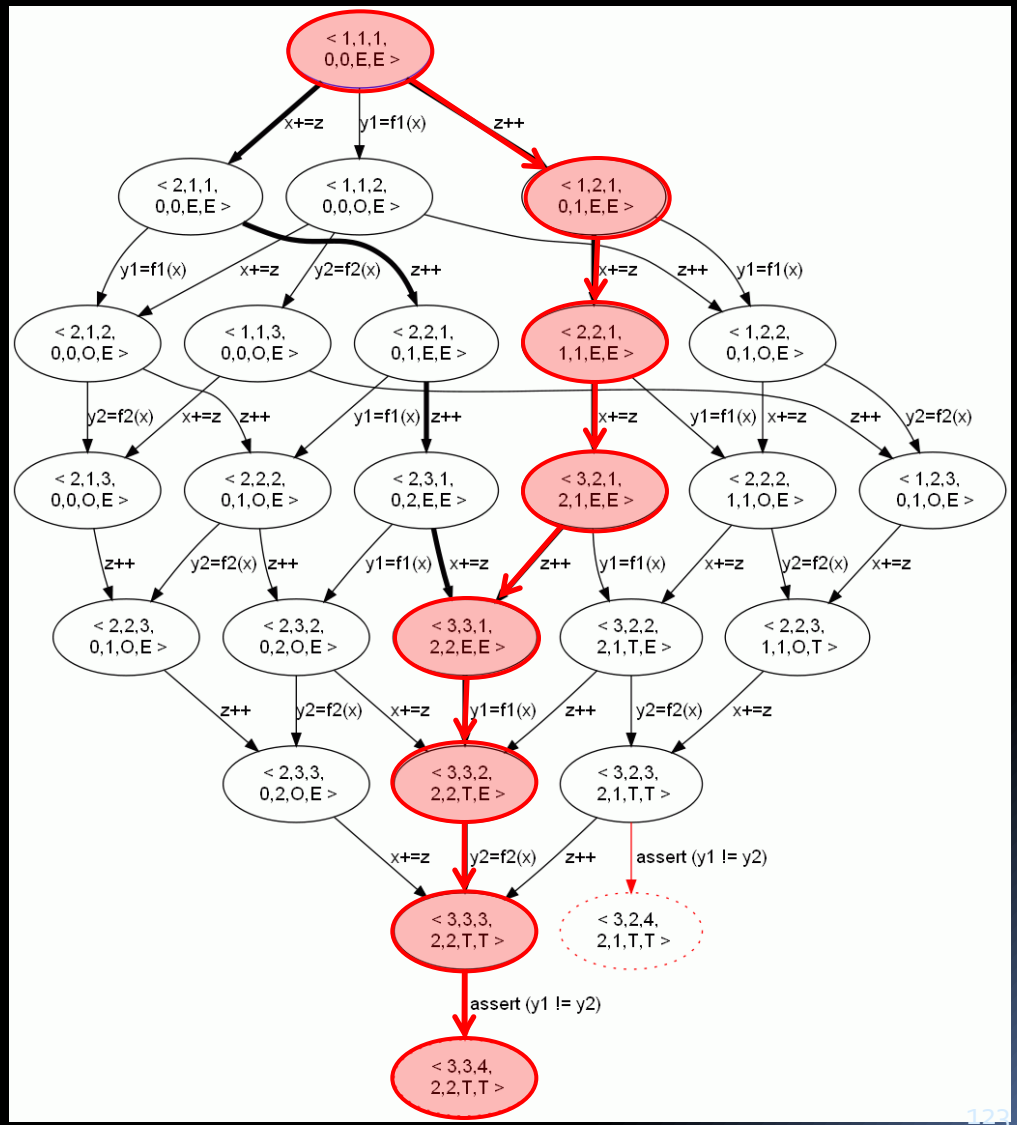
φ = true



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)
  }
}
```

φ = true

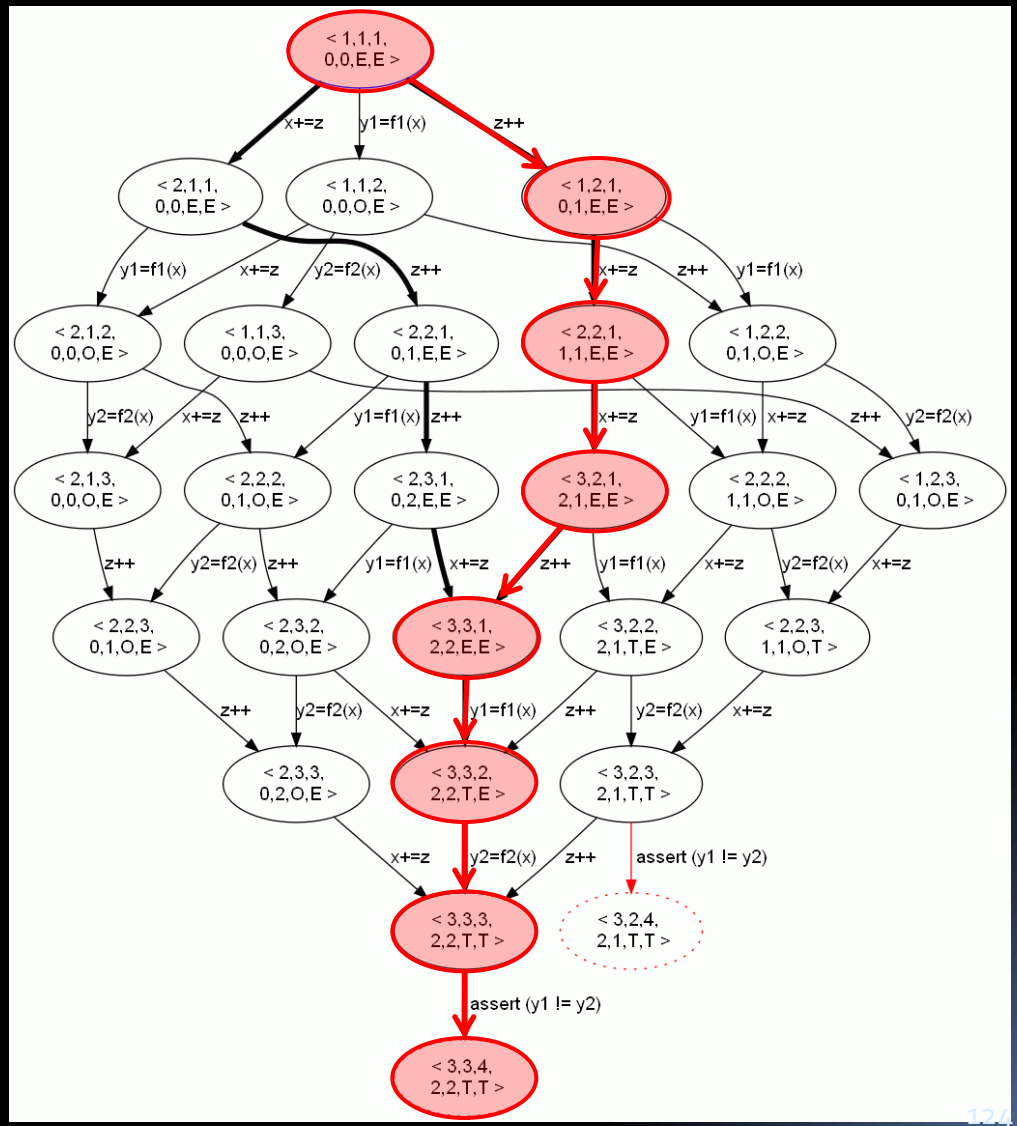


Example: Avoiding Bad Interleavings

```
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_1) = [z++,z++]

φ = true

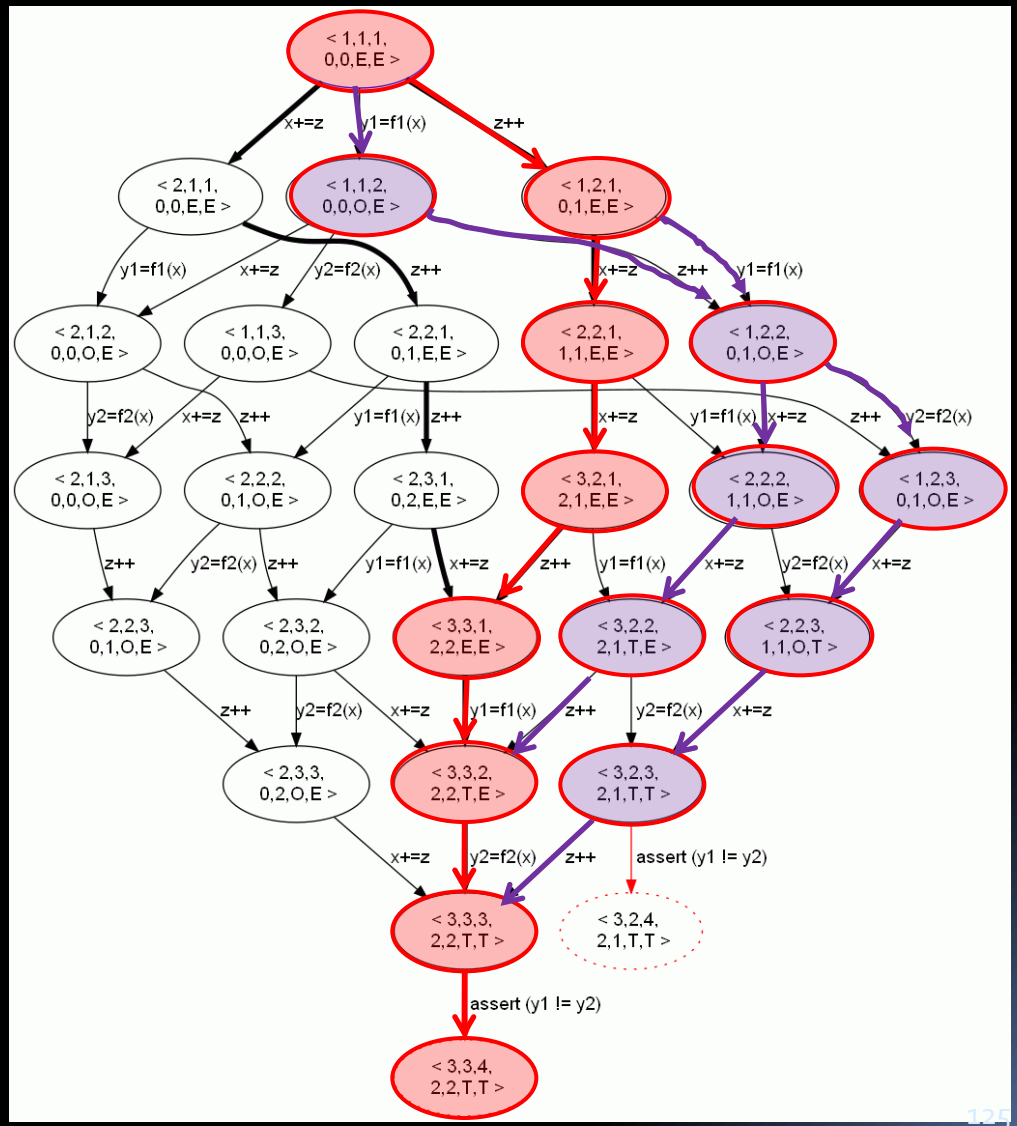


Example: Avoiding Bad Interleavings

```
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_1) = [z++,z++]

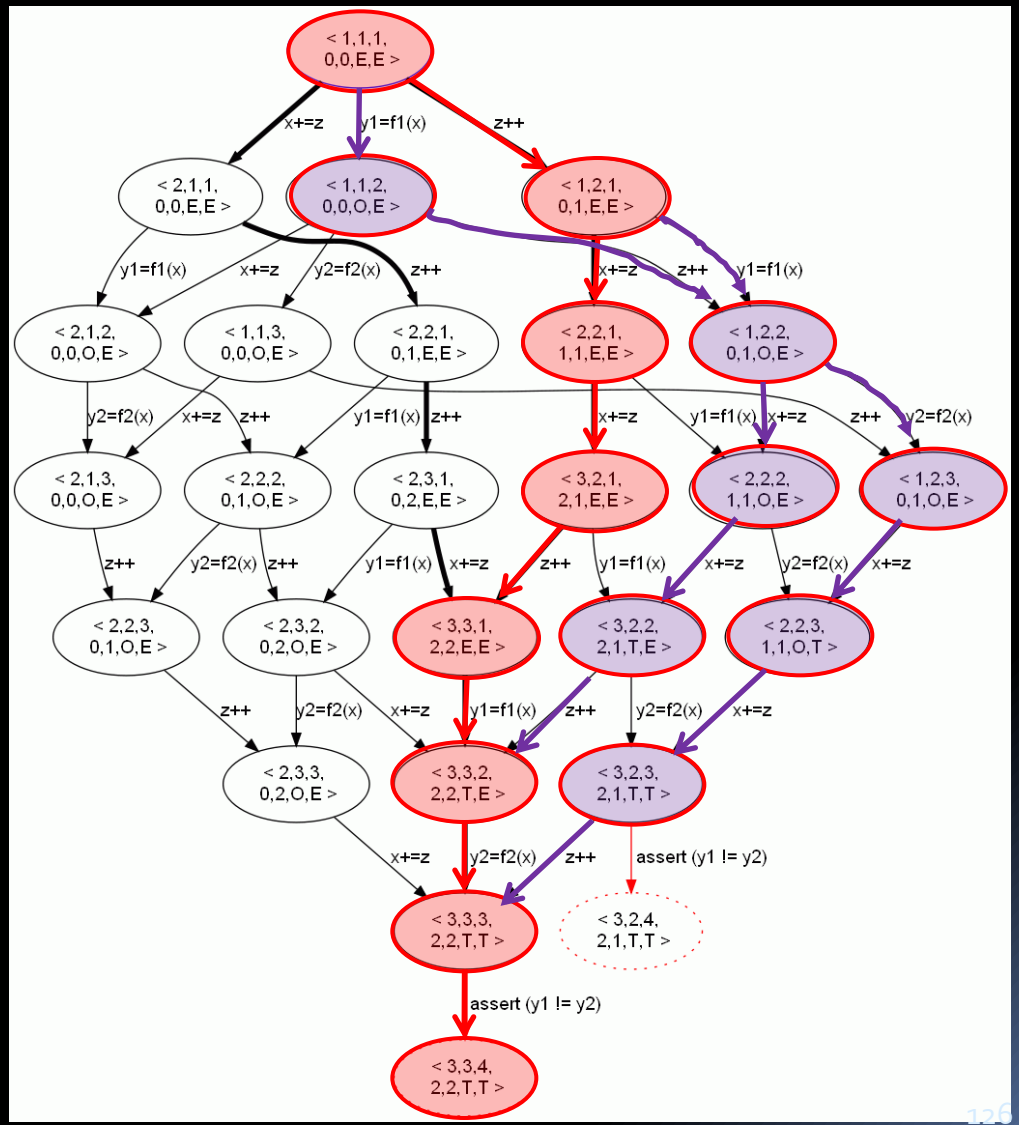
φ = [z++,z++]



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)
  }
}
```

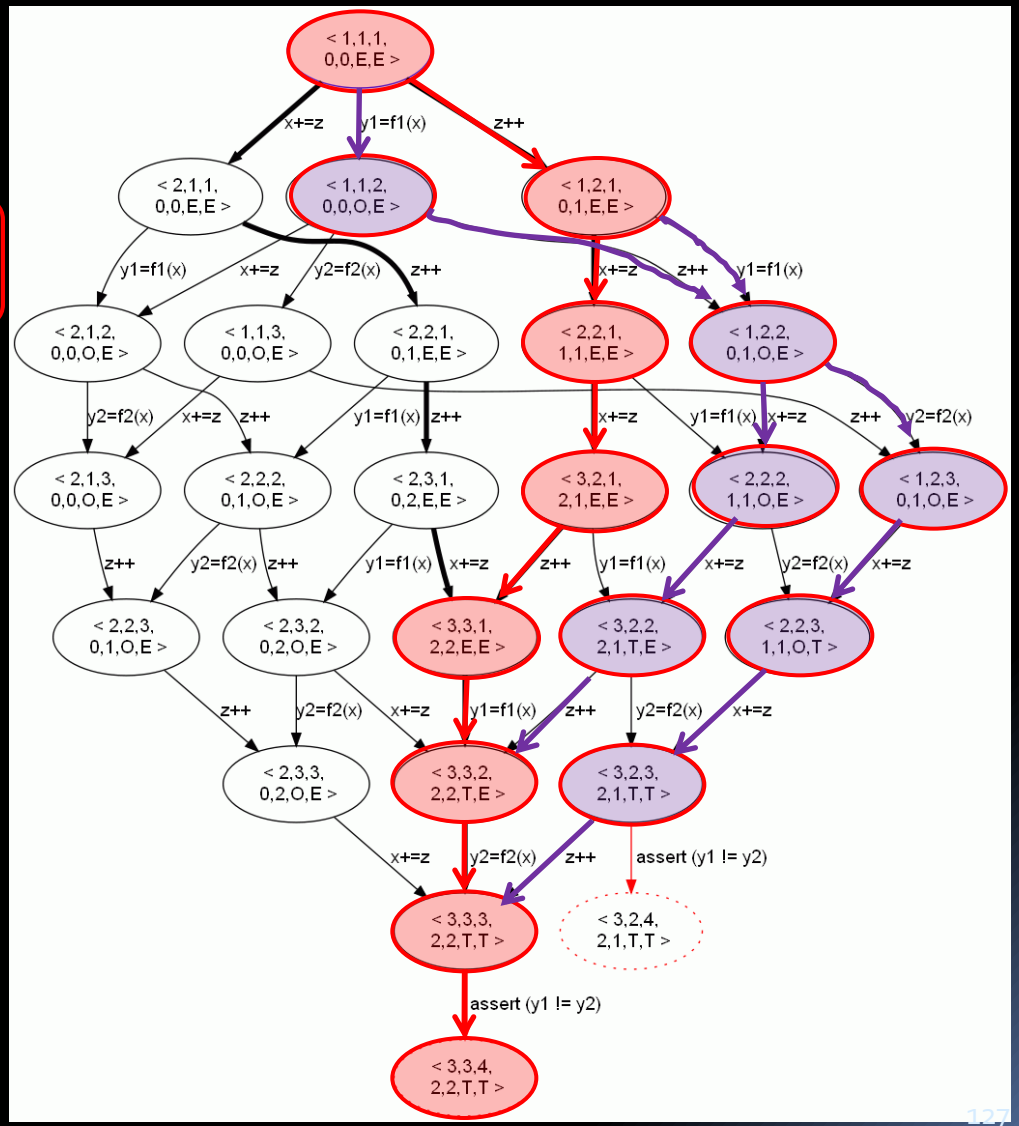
φ = [z++,z++]



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)
  }
}
```

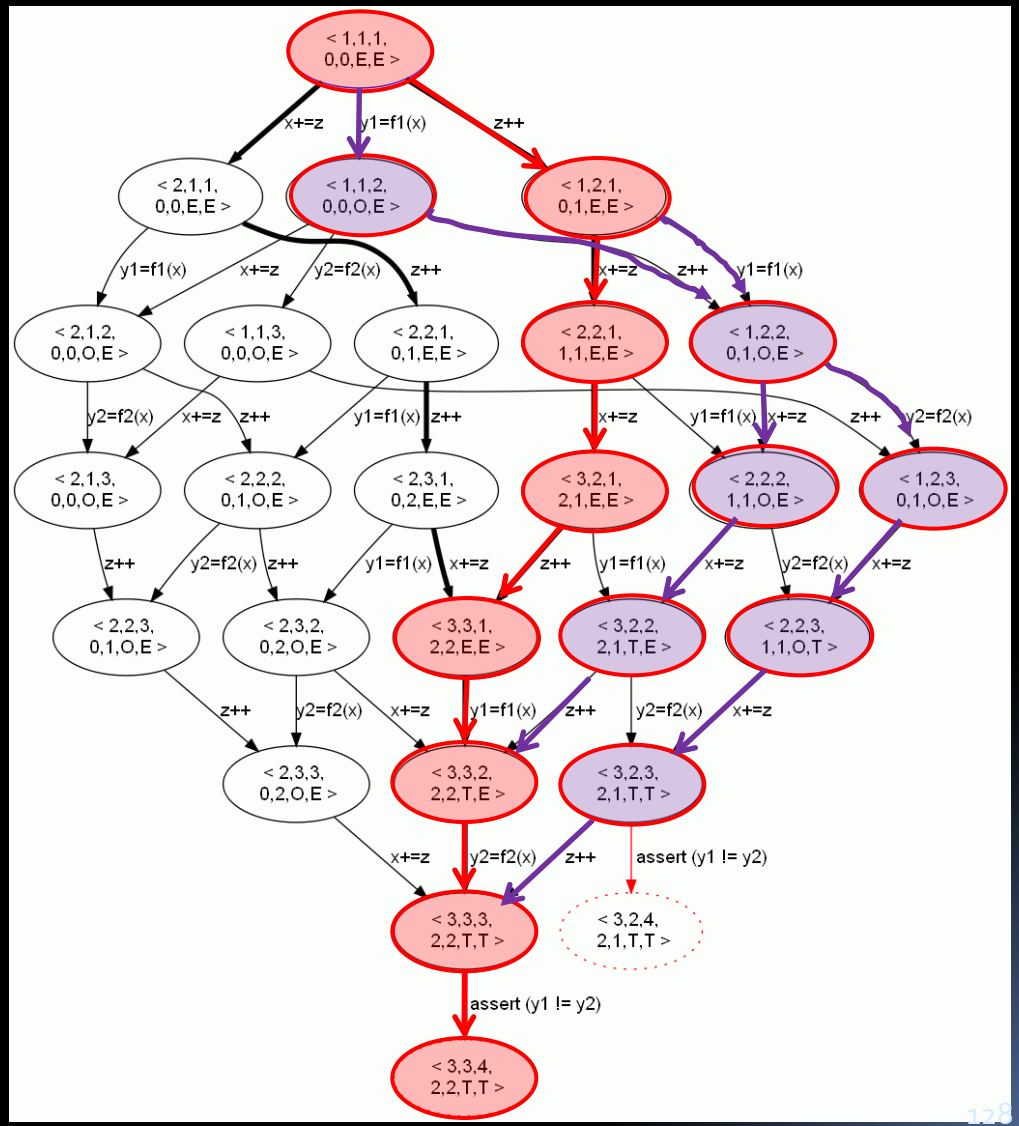
φ = [z++,z++]



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)
  }
}
```

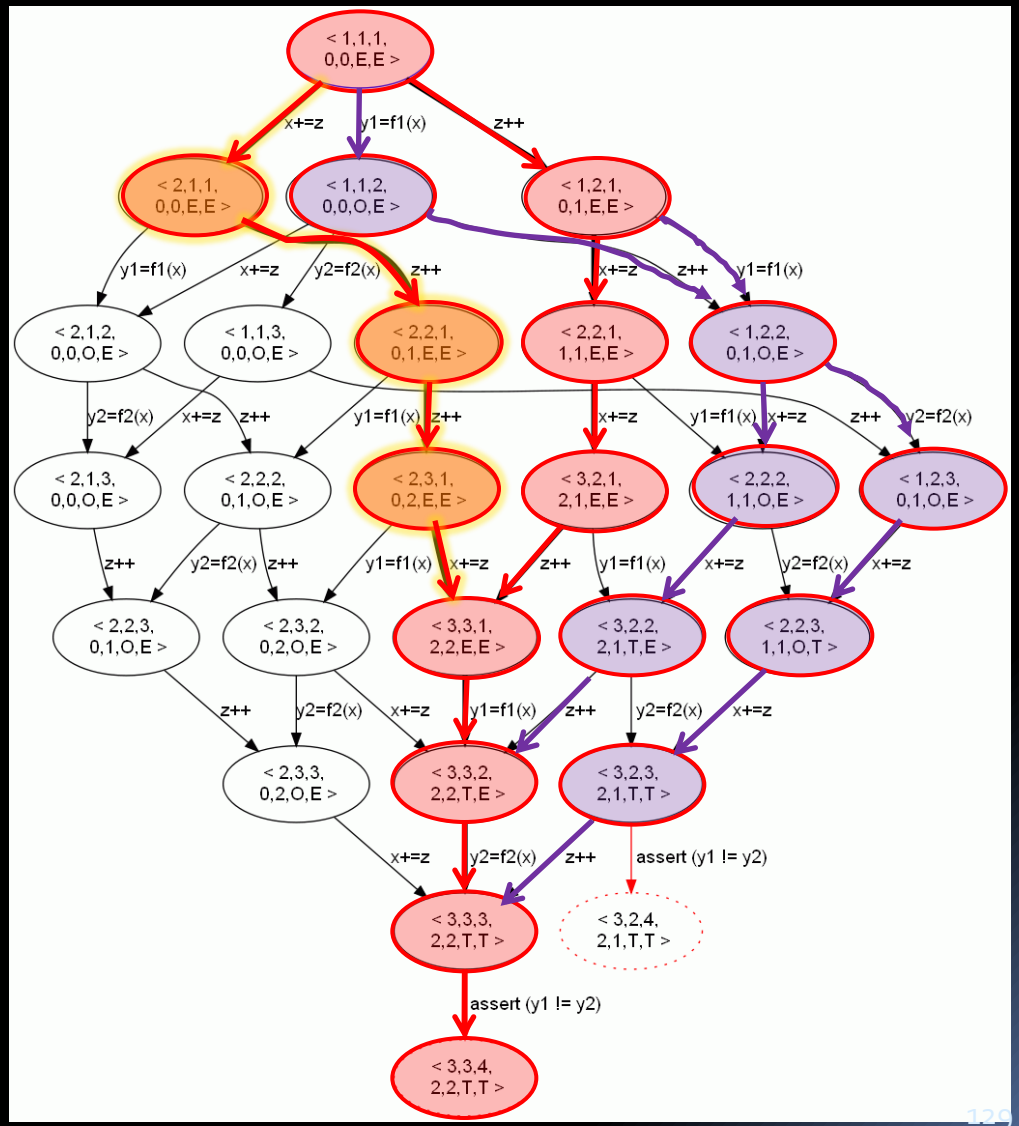
φ = [z++,z++]



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++]

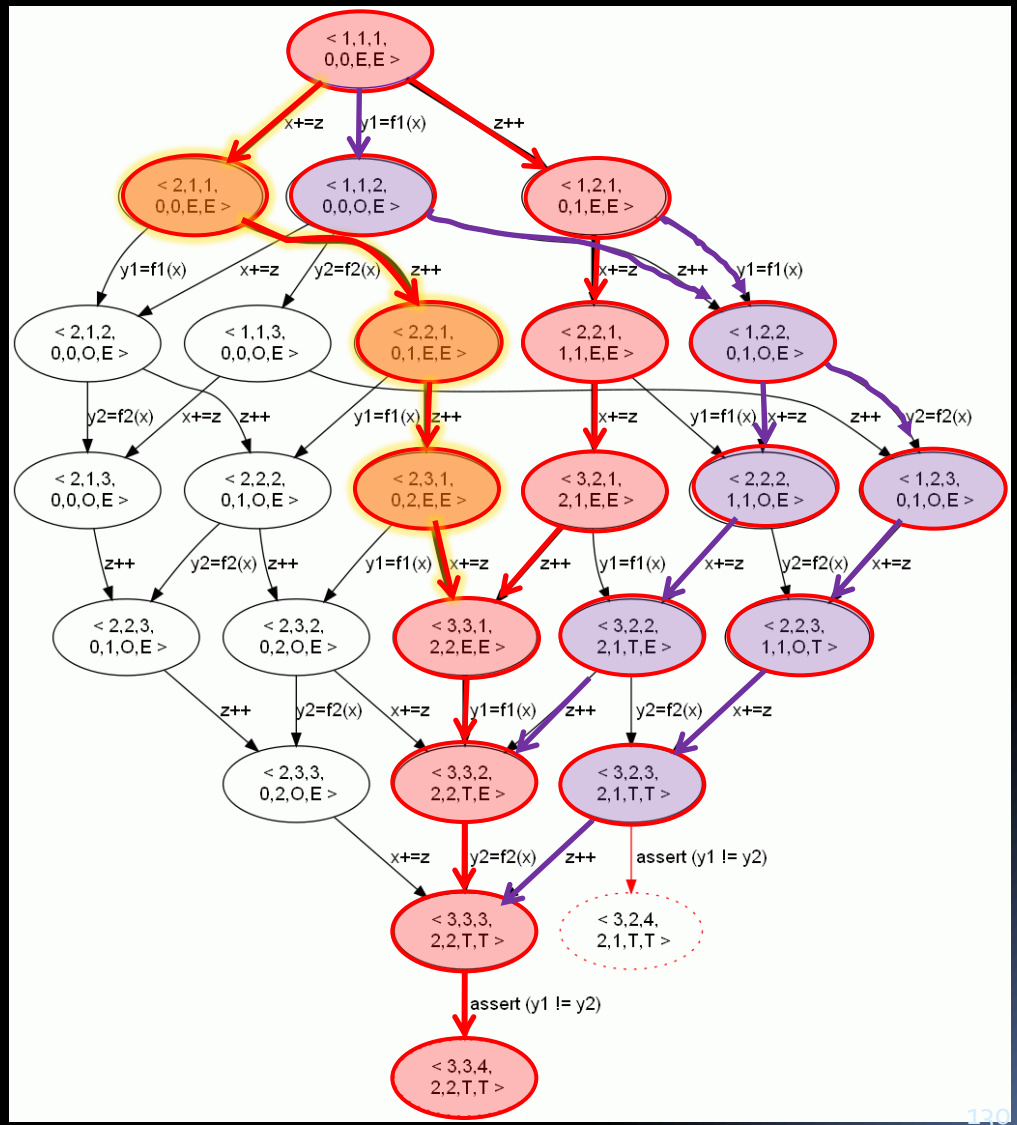


Example: Avoiding Bad Interleavings

```
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++]

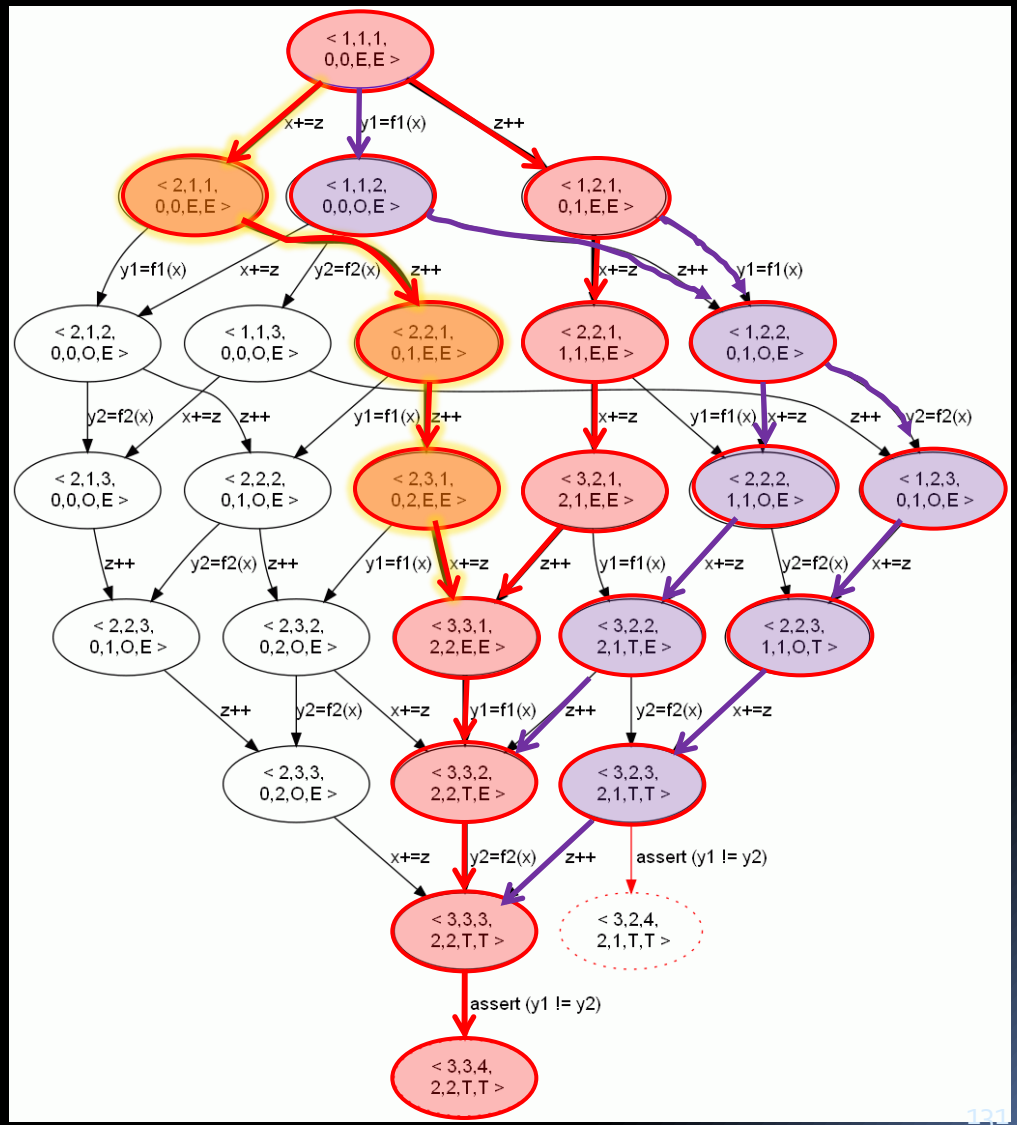


Example: Avoiding Bad Interleavings

```
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]

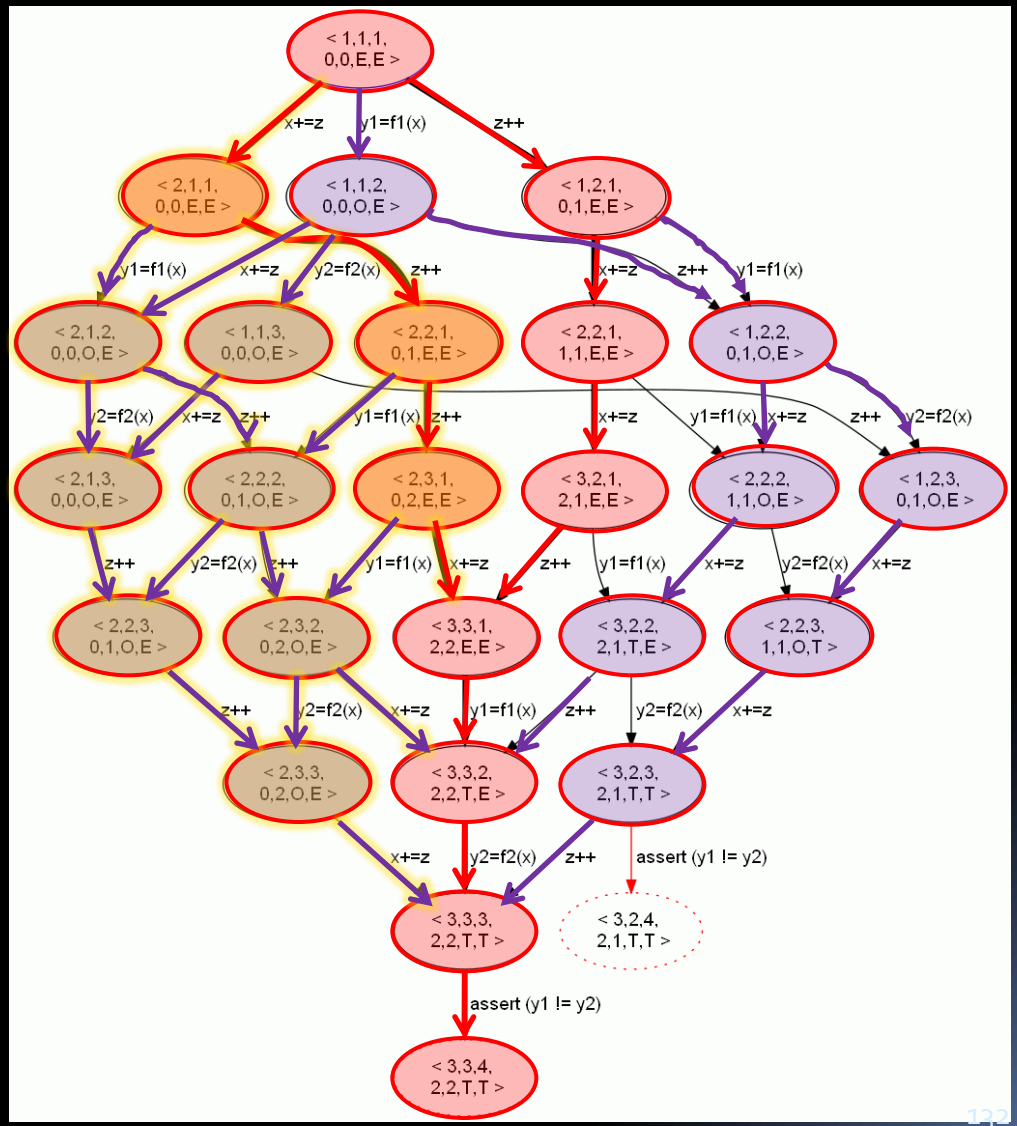


Example: Avoiding Bad Interleavings

```
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)
  }
}
```

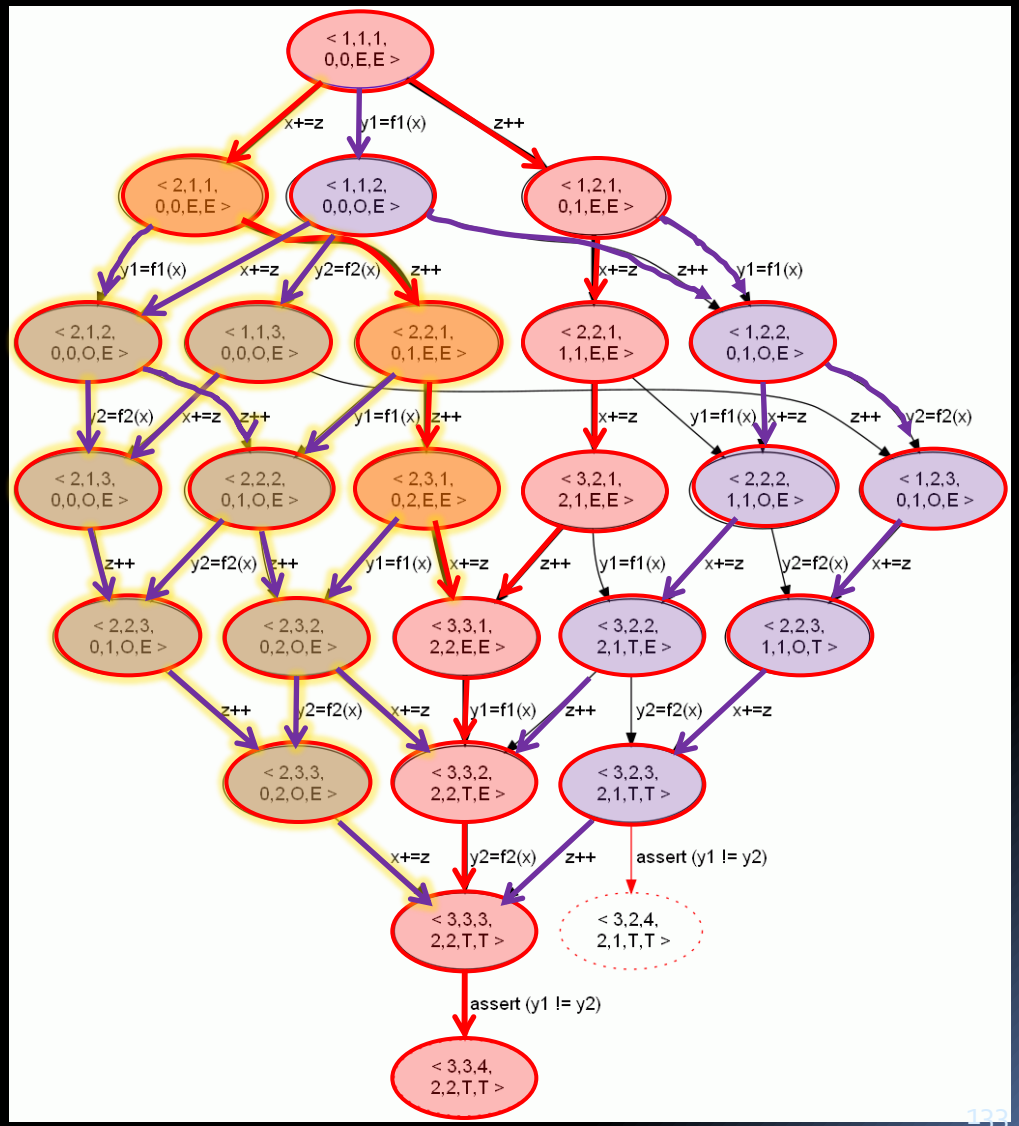
avoid(π_2) =[x+=z,x+=z]

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]

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]

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]

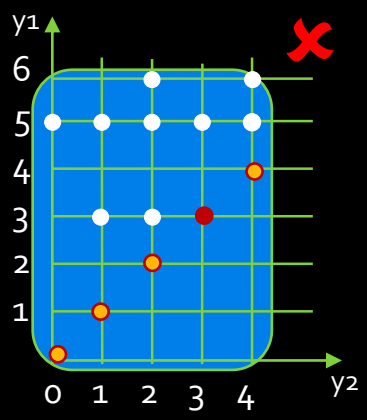
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2



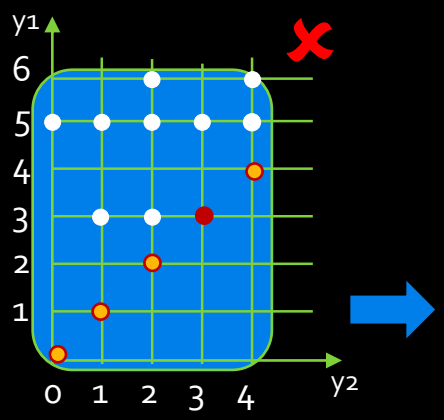
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2



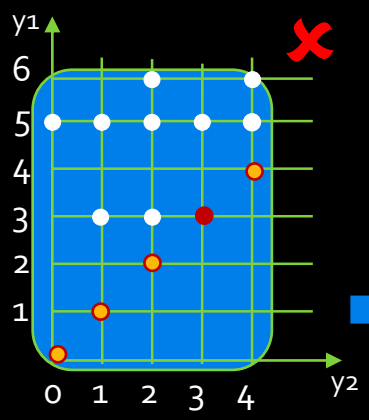
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2

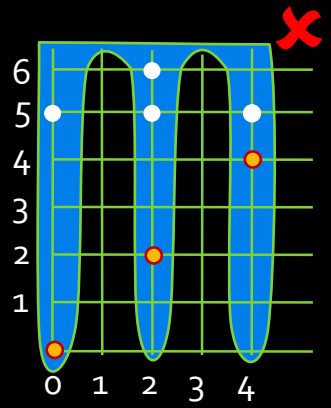


parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2



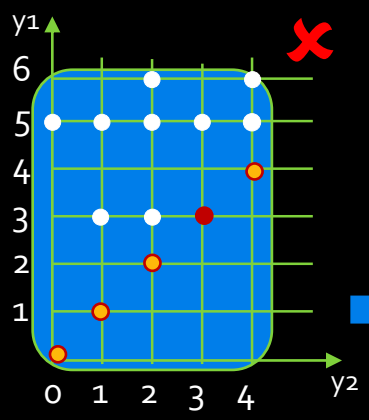
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2

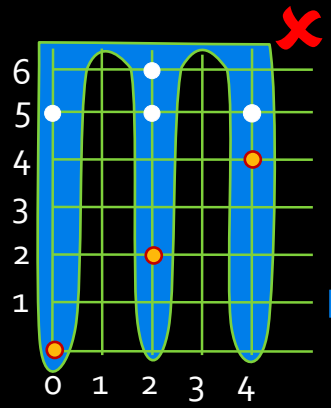


parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2



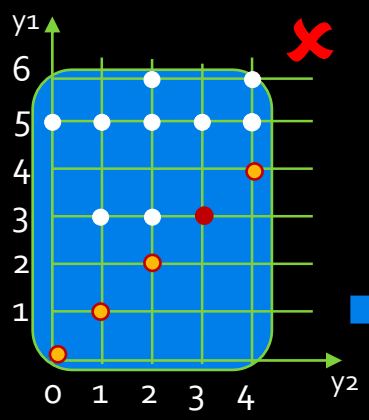
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2

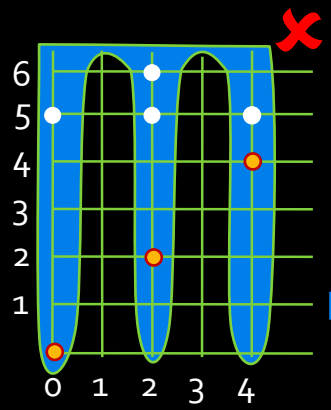


parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2

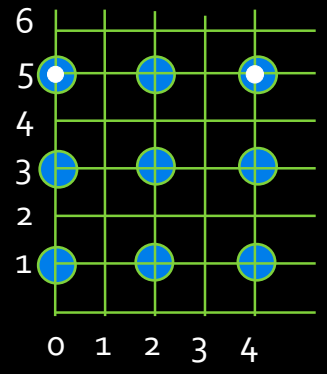


parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2



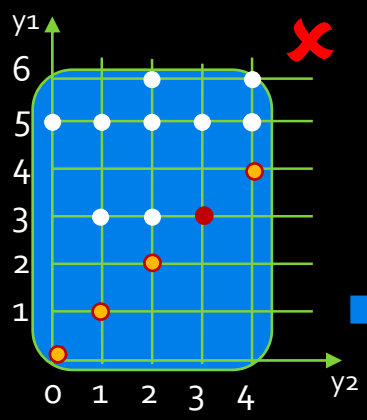
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2

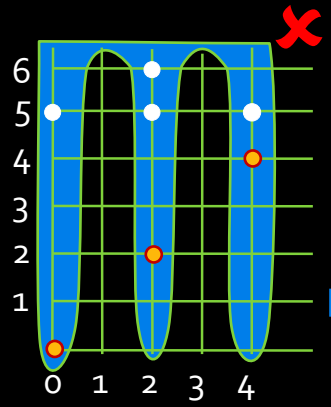


parity

T1 x+=z;
x+=z

T2 [z++;
z++;

T3 y1=f(x)
y2=x
assert
y1!= y2

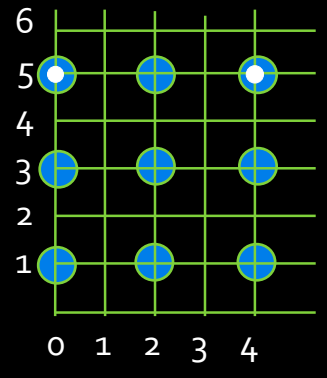


parity

T1 [x+=z;
x+=z

T2 [z++;
z++;

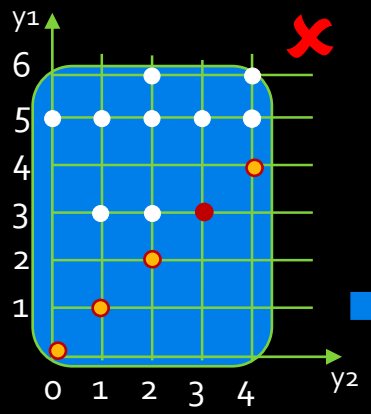
T3 y1=f(x)
y2=x
assert
y1!= y2



But we can also refine the abstraction…

parity

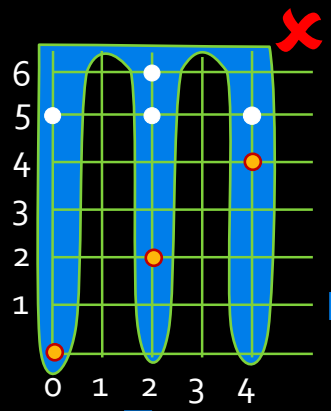
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(a)

parity

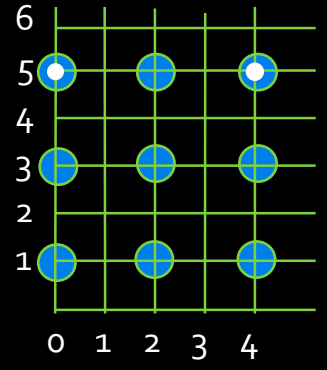
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(b)

parity

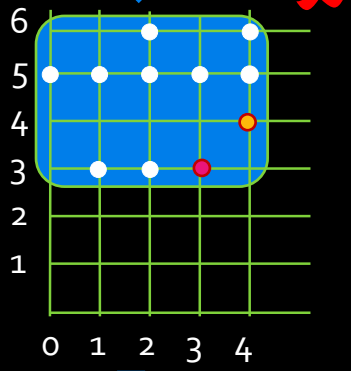
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(c)

interval

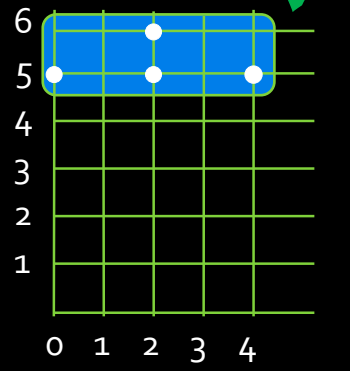
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(d)

interval

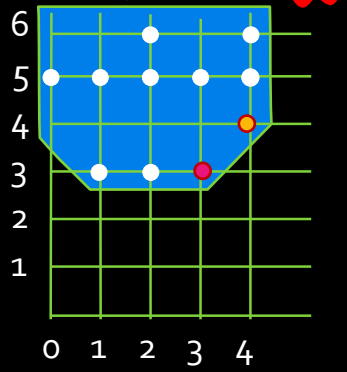
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(e)

octagon

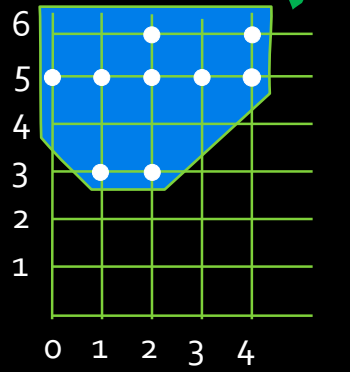
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(f)

octagon

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



(g)

AGS Algorithm – More Details

Input: Program P, Specification S, Abstraction α

Output: Program P′ satisfying S under α

Order of selection matters

```
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
  }
}
```

AGS Algorithm – More Details

Input: Program P, Specification S

Output: Program P′ satisfying S

Forward Abstract Interpretation, taking φ into account for pruning infeasible 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 (?) {
    ψ = avoid(π)
    if (ψ ≠ false) φ = φ ∧ ψ
      else abort
  } else {
    α′ = refine(α, π)
    if (α′ ≠ α) α = α′
      else abort
  }
}
```

AGS Algorithm – More Details

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
  }
}
```

Backward exploration of invalid Interleavings using φ to prune infeasible interleavings.

AGS Algorithm – More Details

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 ∧
      else abort
  } else {
    α′ = refine(α, π)
    if (α′ ≠ α) α = α′
      else abort
  }
}
```

Choosing between abstraction refinement and program restriction
- not always possible to refine/avoid
- may try and backtrack

AGS Algorithm – More Details

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
  }
}
```

Up to this point did not commit to a synchronization mechanism