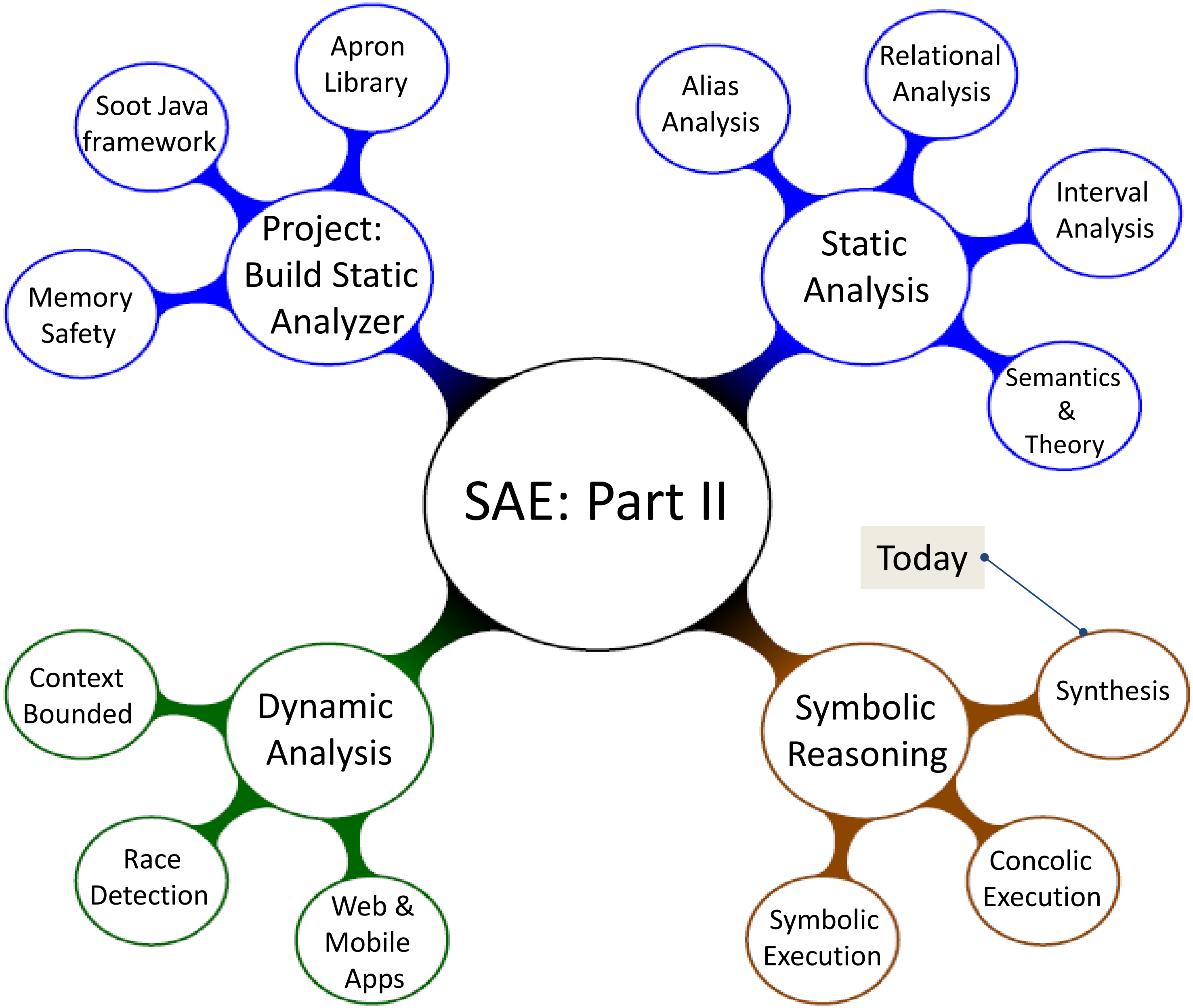
Software Architecture and Engineering: Part II

ETH Zurich, Spring 2014
Prof. Martin Vechev


SAE: Part II
Project: Build Static Analyzer
Soot Java framework
Apron Library
Memory Safety
Static Analysis
Alias Analysis
Relational Analysis
Interval Analysis
Semantics & Theory
Today
Dynamic Analysis
Context Bounded
Race Detection
Web & Mobile Apps
Symbolic Reasoning
Synthesis
Concolic Execution
Symbolic Execution

Setting

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

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

Setting

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

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

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 ?

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

Setting

a basic question in program analysis

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

Setting

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

$$P_{\alpha} \nvDash S$$

Setting

refine abstraction α to a new abstraction α'

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

Setting

modify program P to a new program P'

how ?

$$P'_\alpha \models S$$

Setting

weaken specification S to S'

unclear to what, true is also a solution

$$P_\alpha \models S'$$

Setting

$$\begin{tikzcd}
& P \overset{?}{\models} S \arrow[d, "\text{add abstraction (for automation)}"] & \\
& P_\alpha \not\models S \arrow[dl, "\text{refine abstraction}"'] \arrow[d, "\text{modify program}"] \arrow[dr, "\text{weaken specification}"] & \\
P_{\alpha'} \models S & P'_\alpha \models S & P_\alpha \models S'
\end{tikzcd}$$

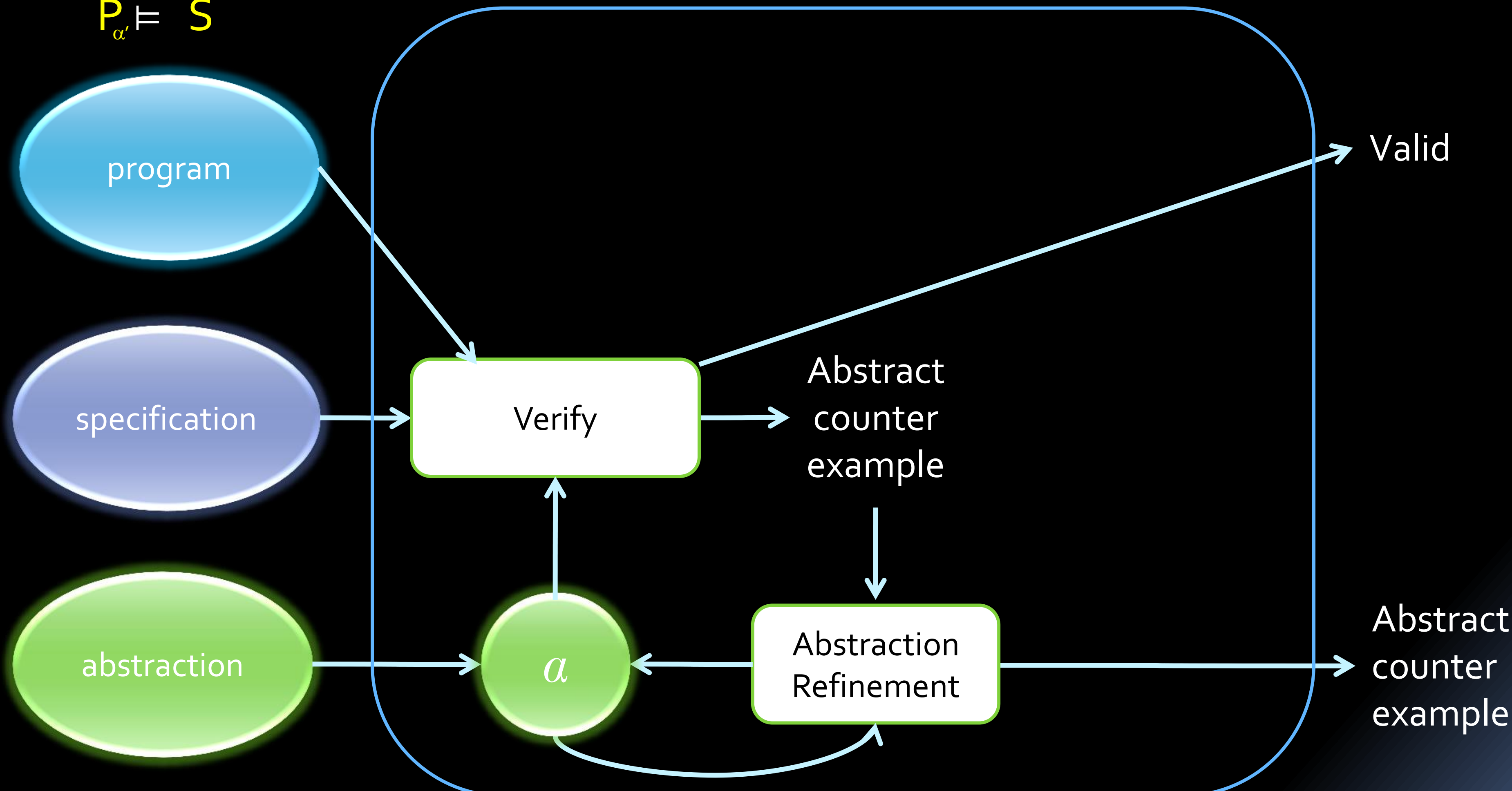
can also combine steps

Refine Abstraction

$P_\alpha \nvDash S$

$\downarrow$

$P_{\alpha'} \vDash S$

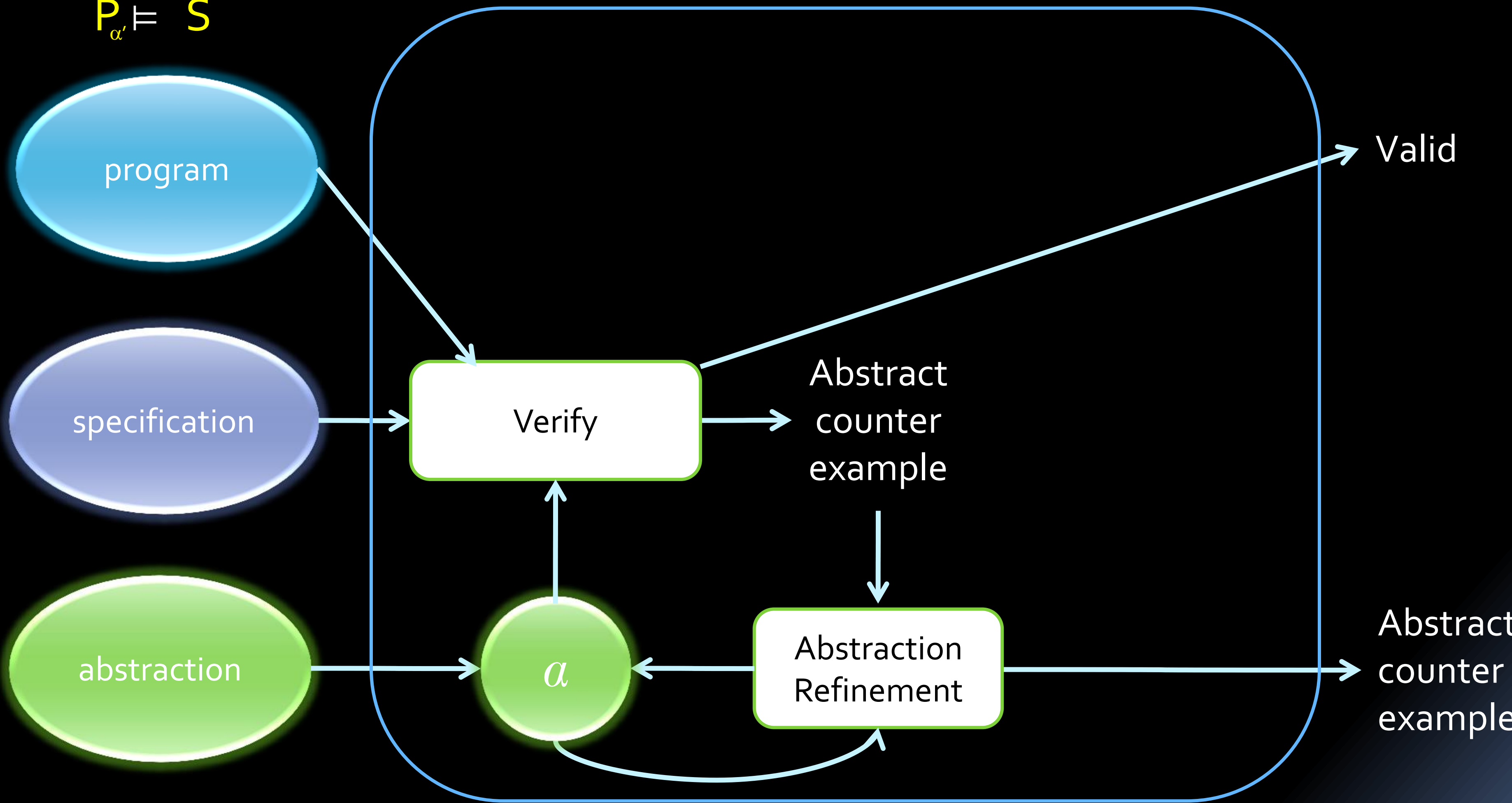


Refine Abstraction

$P_{\alpha} \nvDash S$

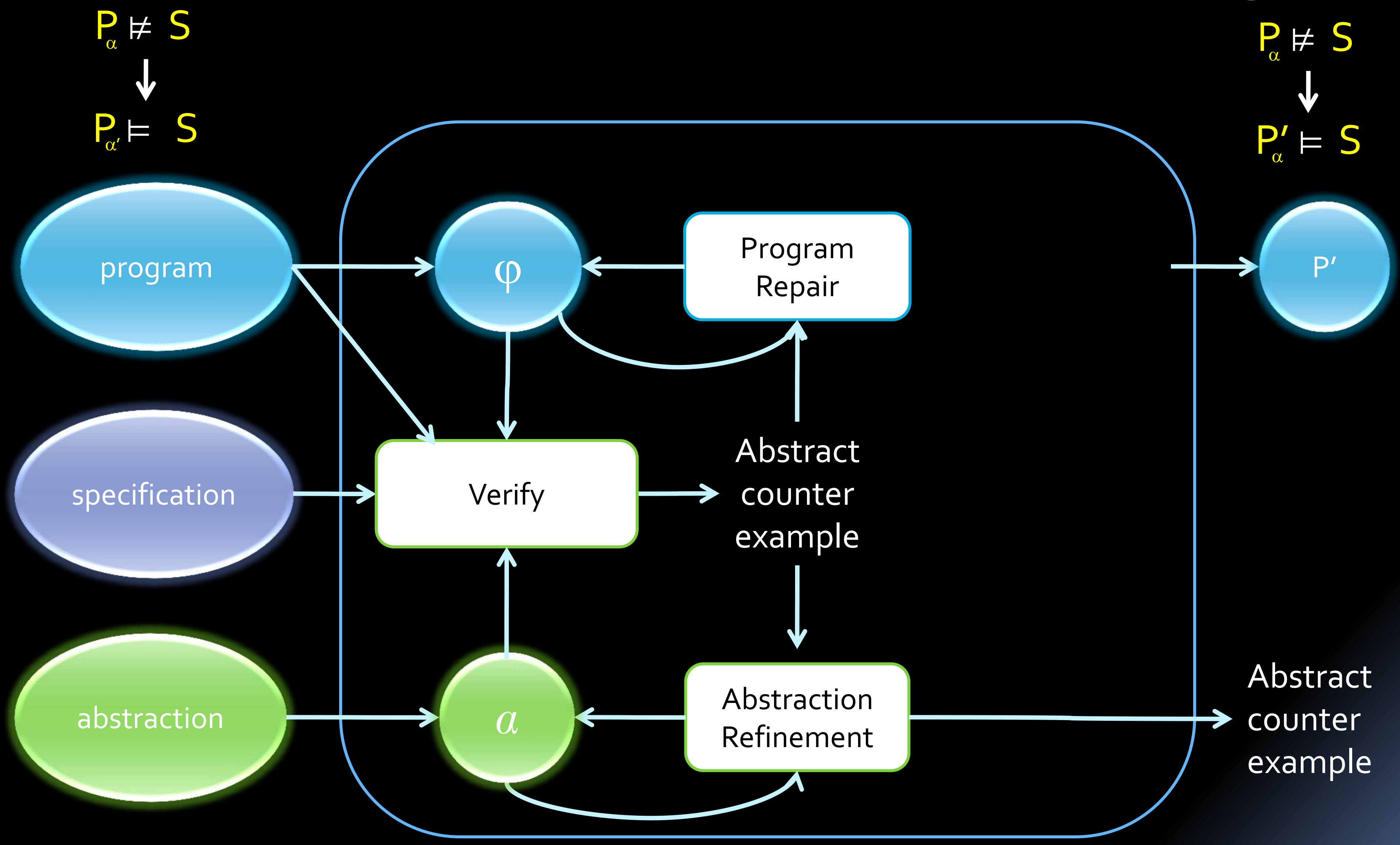
$\downarrow$

$P_{\alpha'} \vDash S$

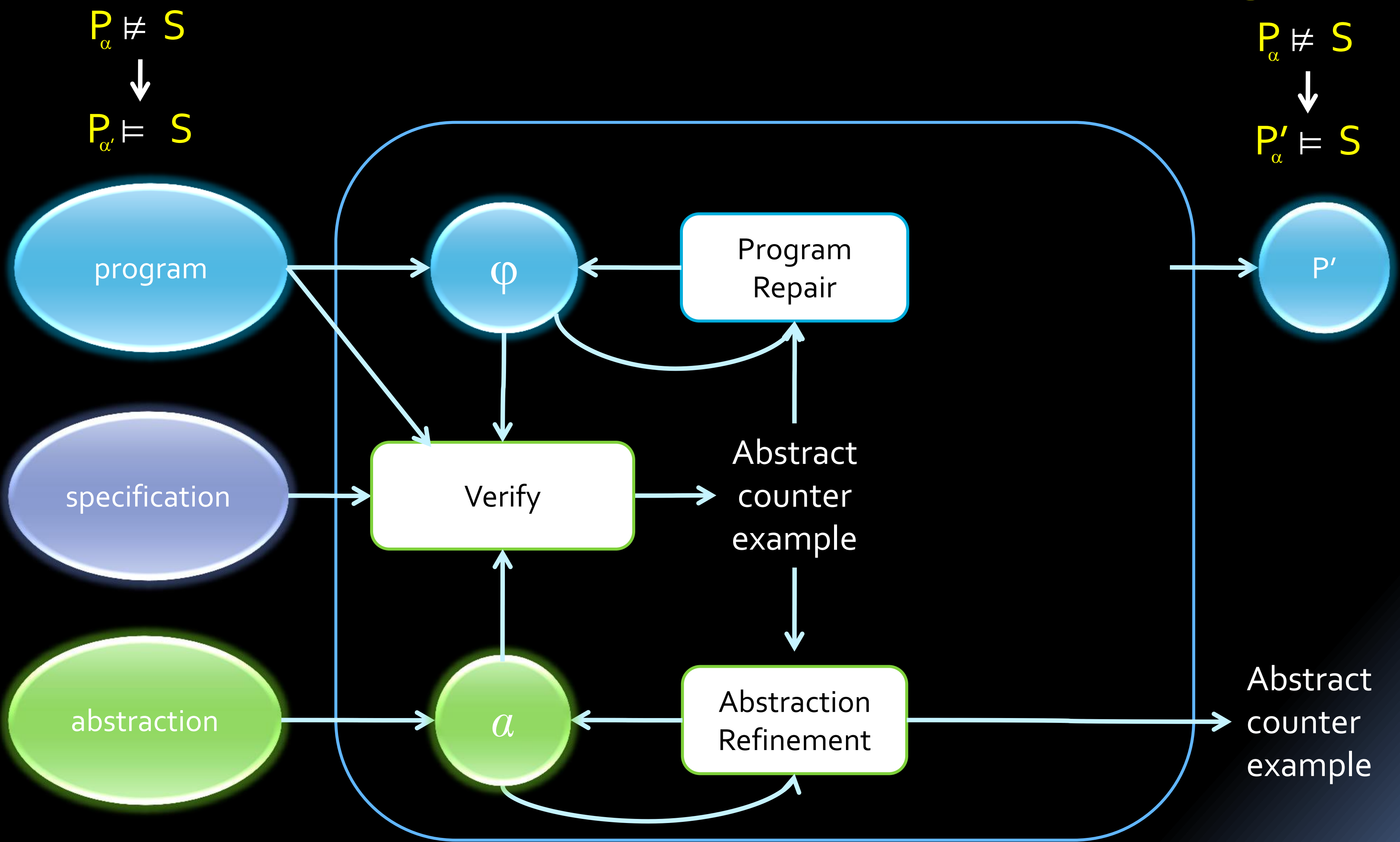


Change the **abstraction** to match the **program**

Refine Abstraction or Repair Program

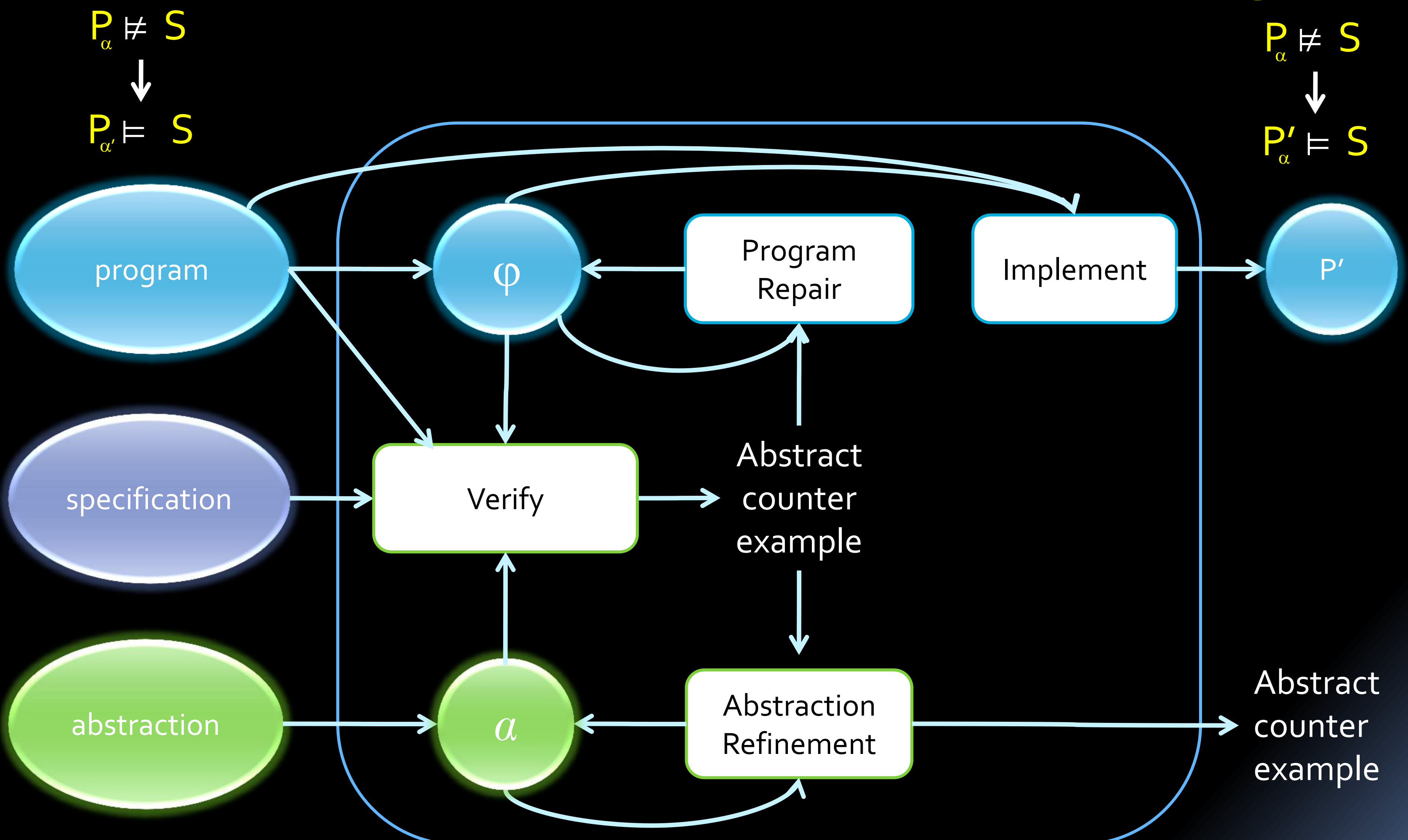


Refine Abstraction or Repair Program



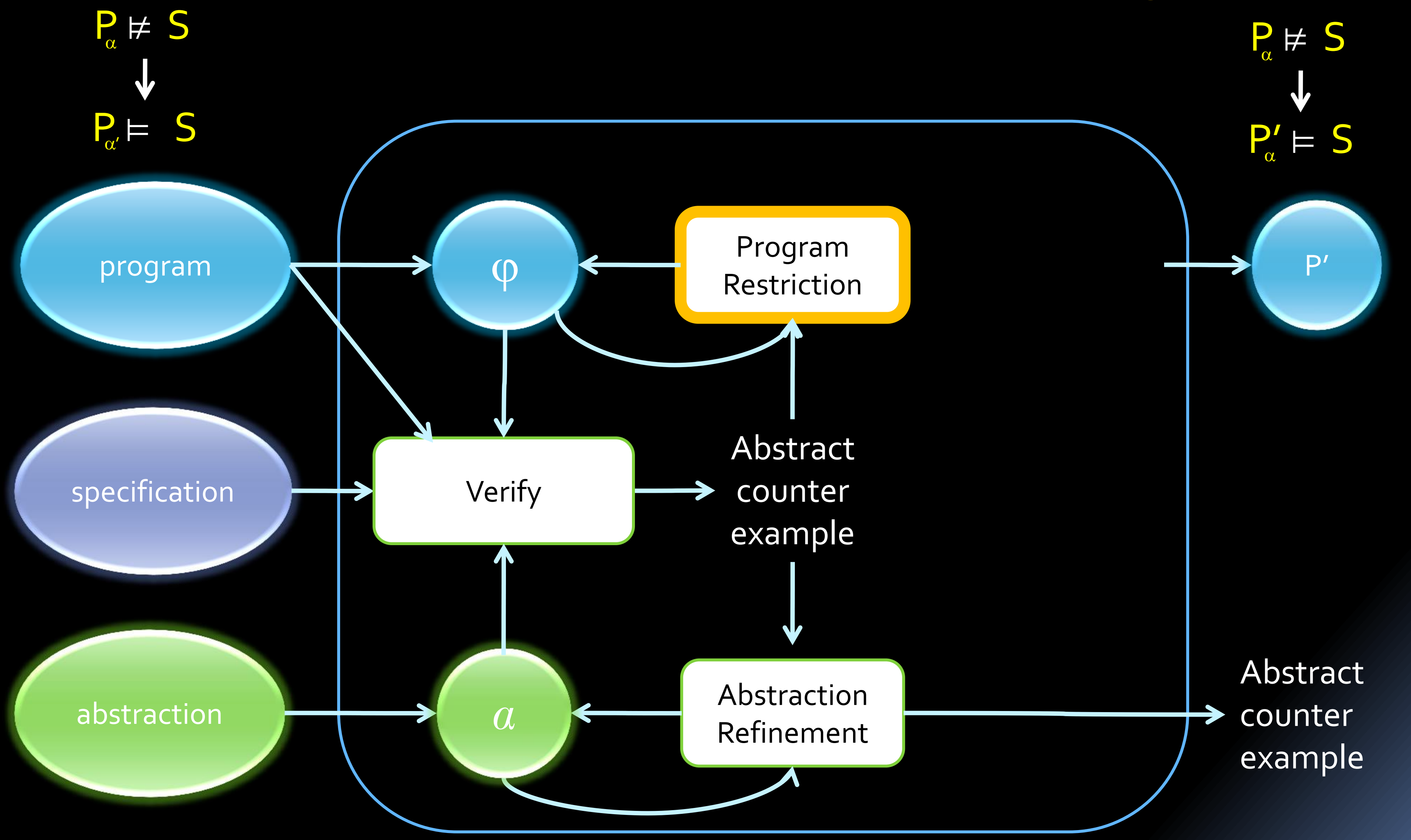
Change the **program** to match the **abstraction**

Refine Abstraction or Repair Program

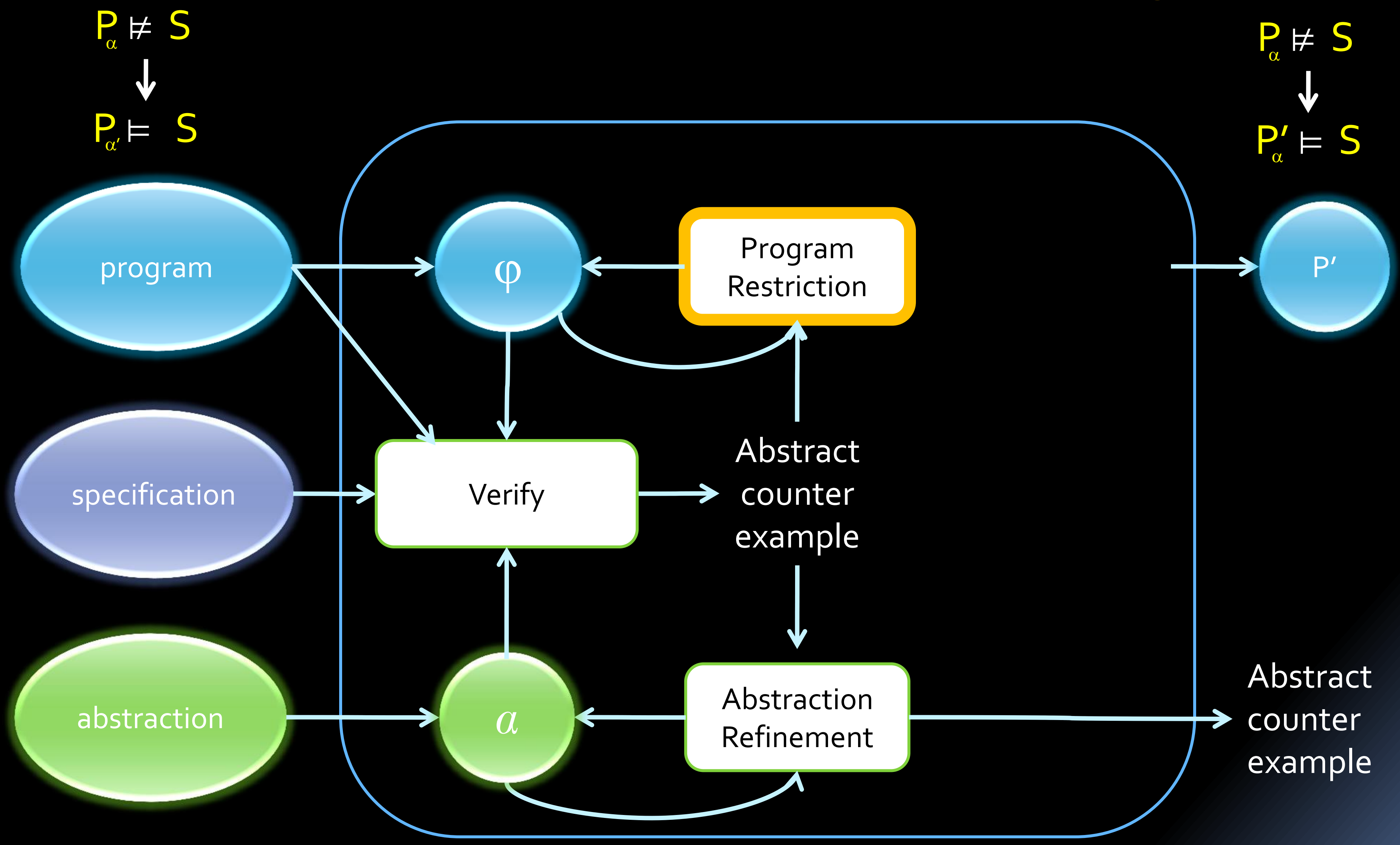


Change the **program** to match the **abstraction**

Instantiate for Concurrency

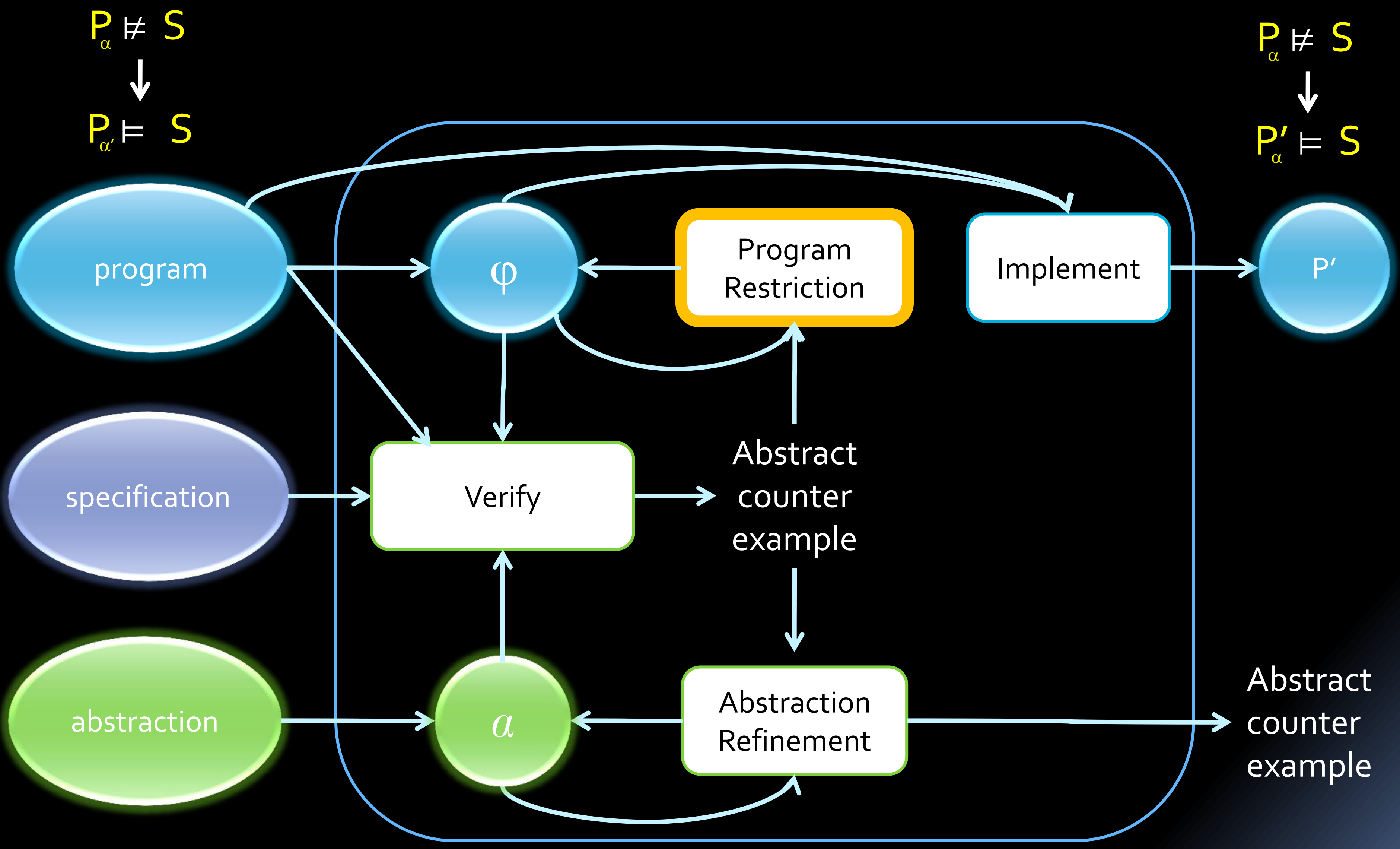


Instantiate for Concurrency



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

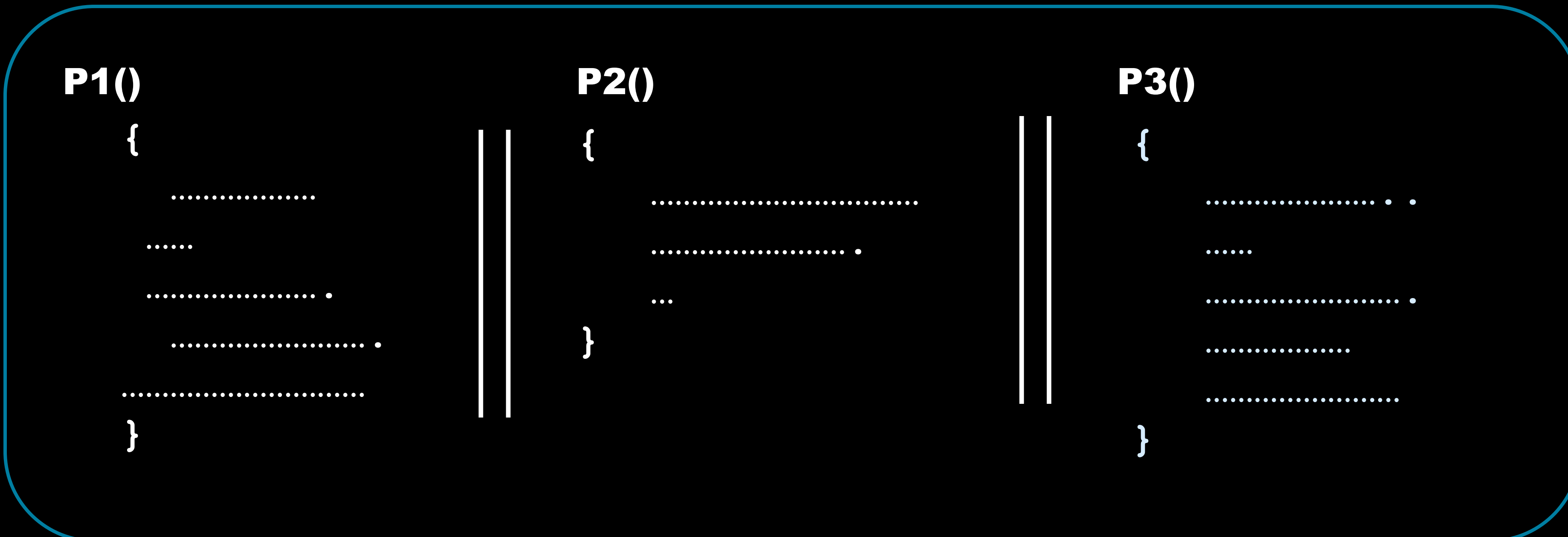
- Atomic sections
- Conditional critical region (CCR)
- Memory barriers (fences)
- CAS
- Semaphores
- Monitors
- Locks
- ....

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

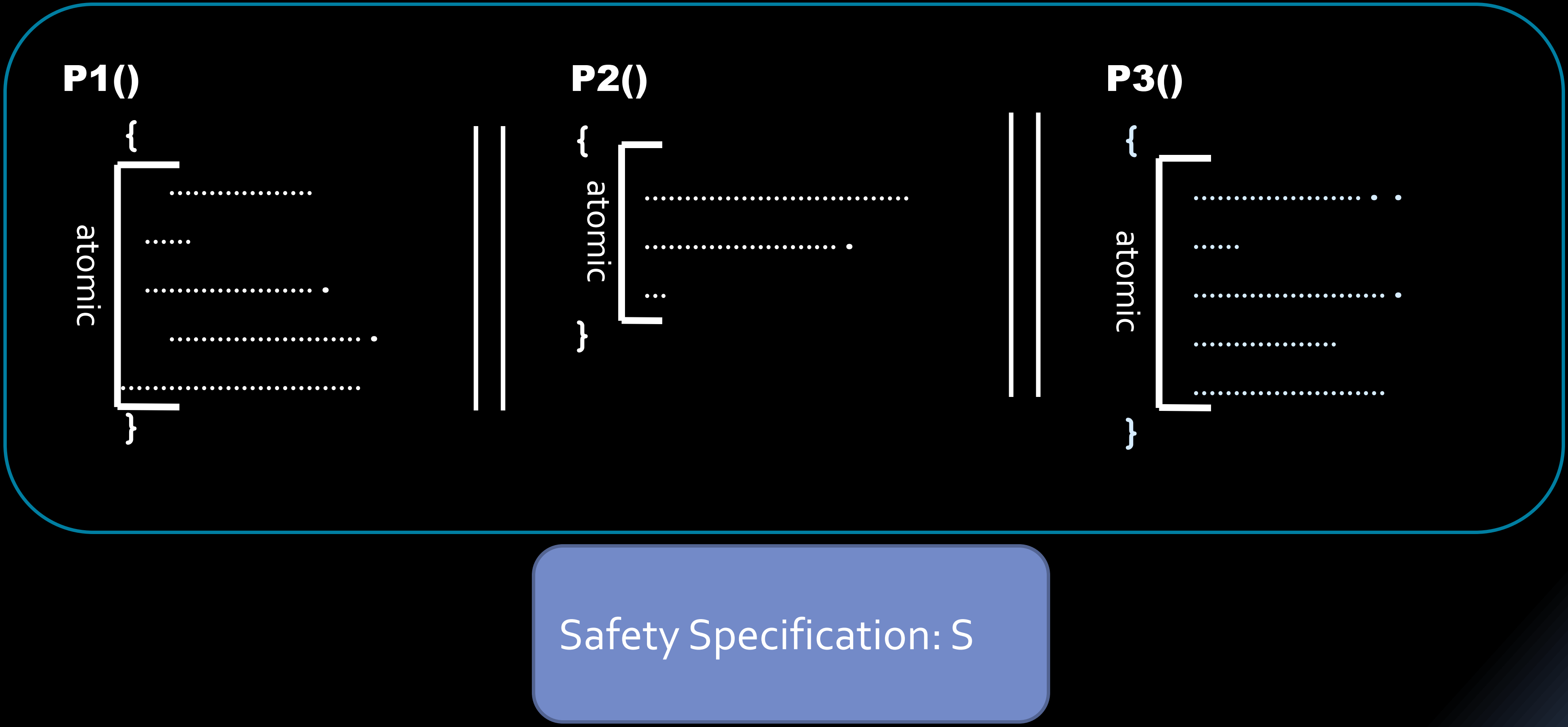


Example: Correct and Efficient Synchronization with Atomic Sections

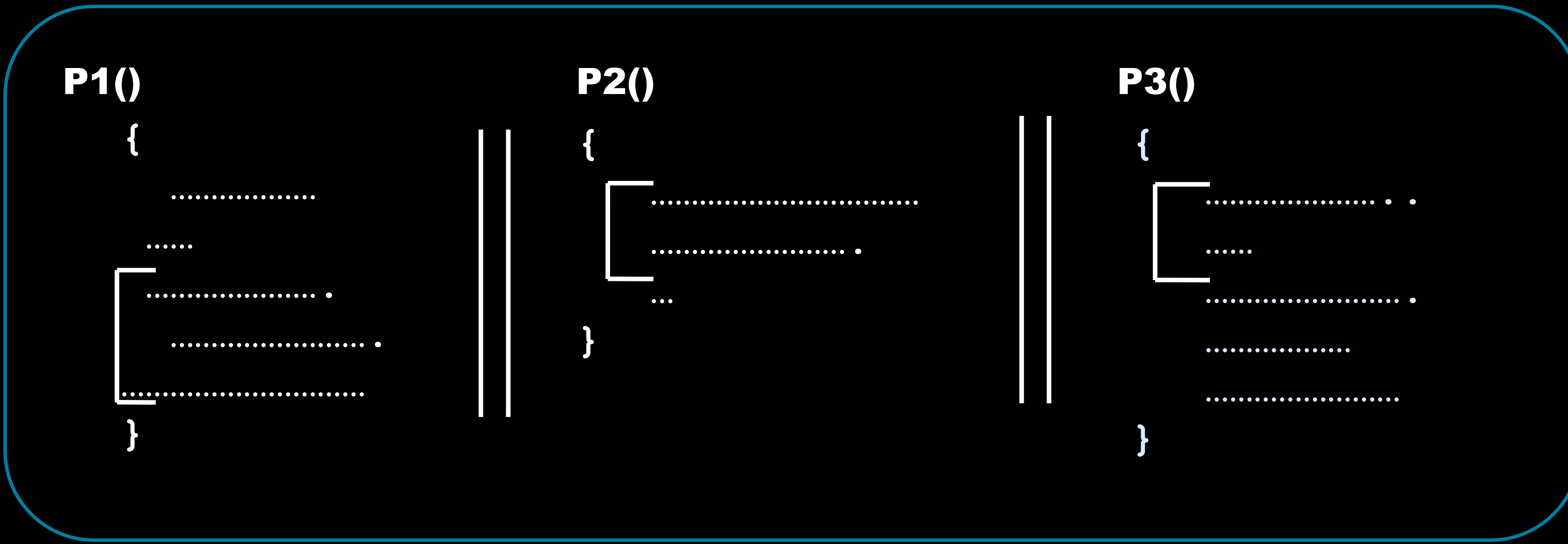
```
P1()                ||  P2()                        ||  P3()
  {                 ||  {                           ||   {
     ..................        ................................       ..................  . .
   ......           ||         ......................  .        ......
   ..................... .     ...                          ...................... .
     ........................ .                             ..................
  ..............................                            ........................
  }                 ||  }                           ||   }
```

Safety Specification: S

Example: Correct and Efficient Synchronization with Atomic Sections

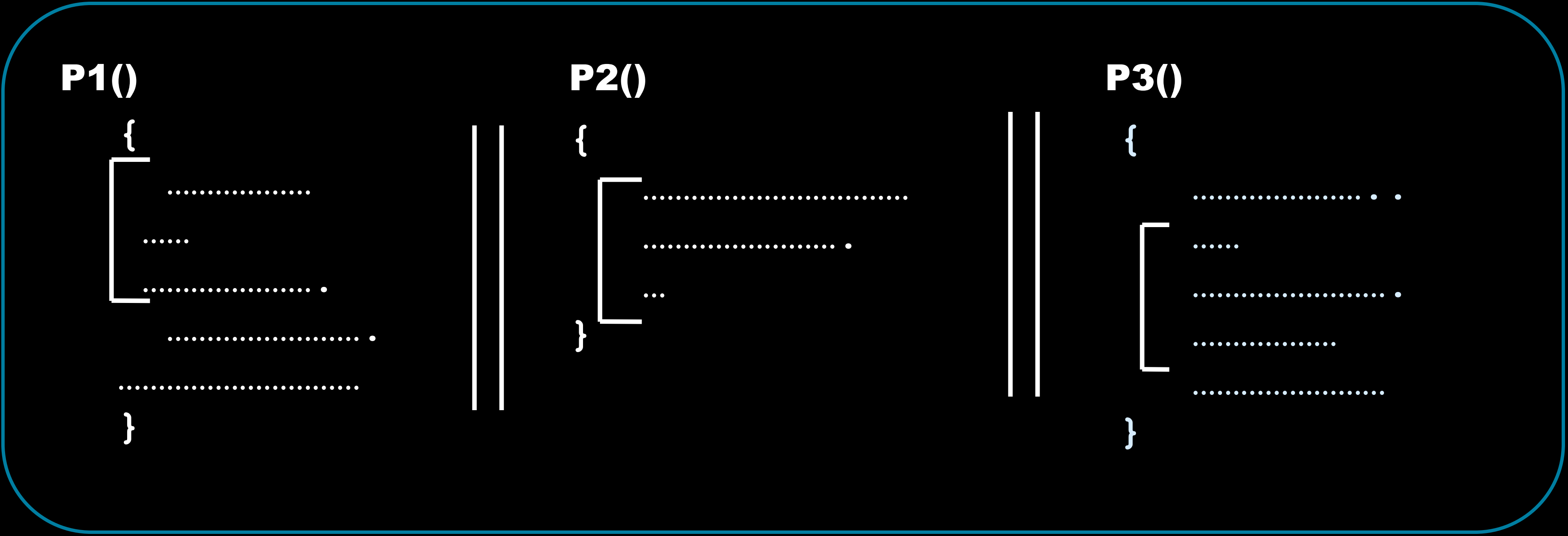


Example: Correct and Efficient Synchronization with Atomic Sections



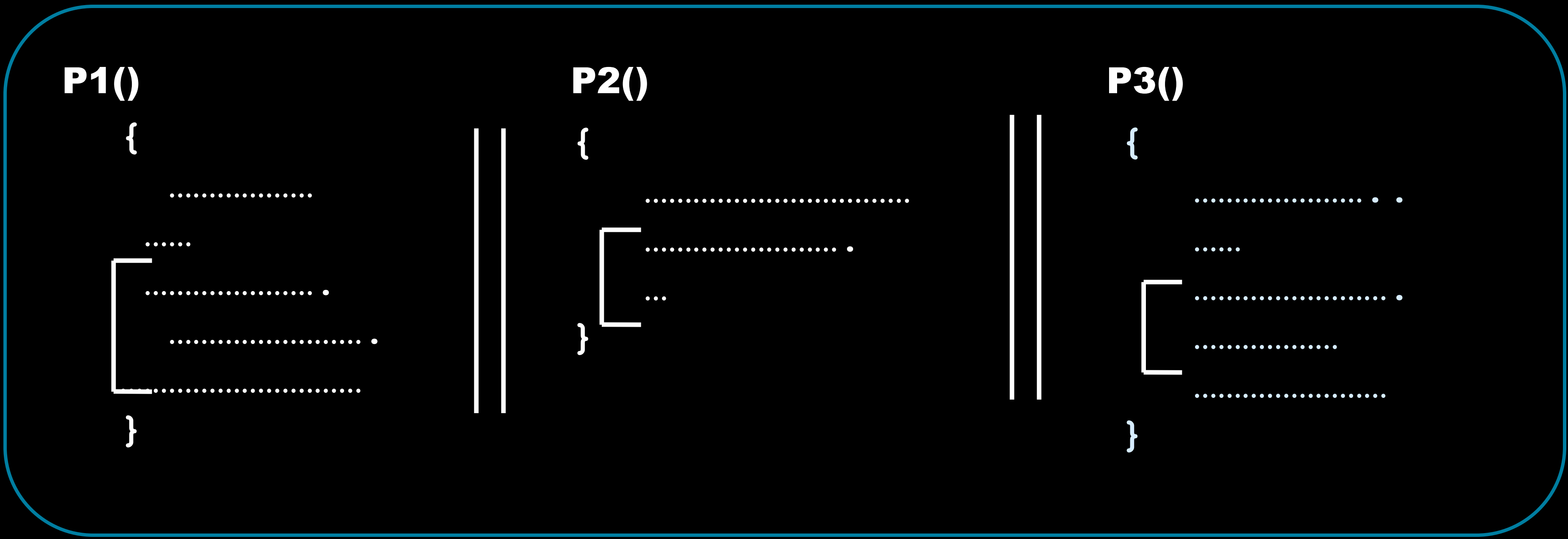
Safety Specification: S

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

```
P1()            ||   P2()                           ||   P3()
 {                    {                                   {
   ..................      ................................      ...................  .  .
  ......                   ...................... .              ......
  .................... .   ...                                   ....................... .
   ........................ .  }                                 .................
 .............................                                   .......................
 }                                                                }
```

Safety Specification: S

Example: Correct and Efficient Synchronization with Atomic Sections

```
P1()                              P2()                                 P3()
  {                               {                                      {
     ..................                ................................       ....................  .  .
   ......                              ....................... .              ......
   ..................... .             ...                                    ....................... .
     ........................ .   }                                           ..................
 ...............................                                              ........................
  }                                                                      }
```

Safety Specification: S

Assist the programmer by automatically inferring
correct and efficient synchronization

Challenge

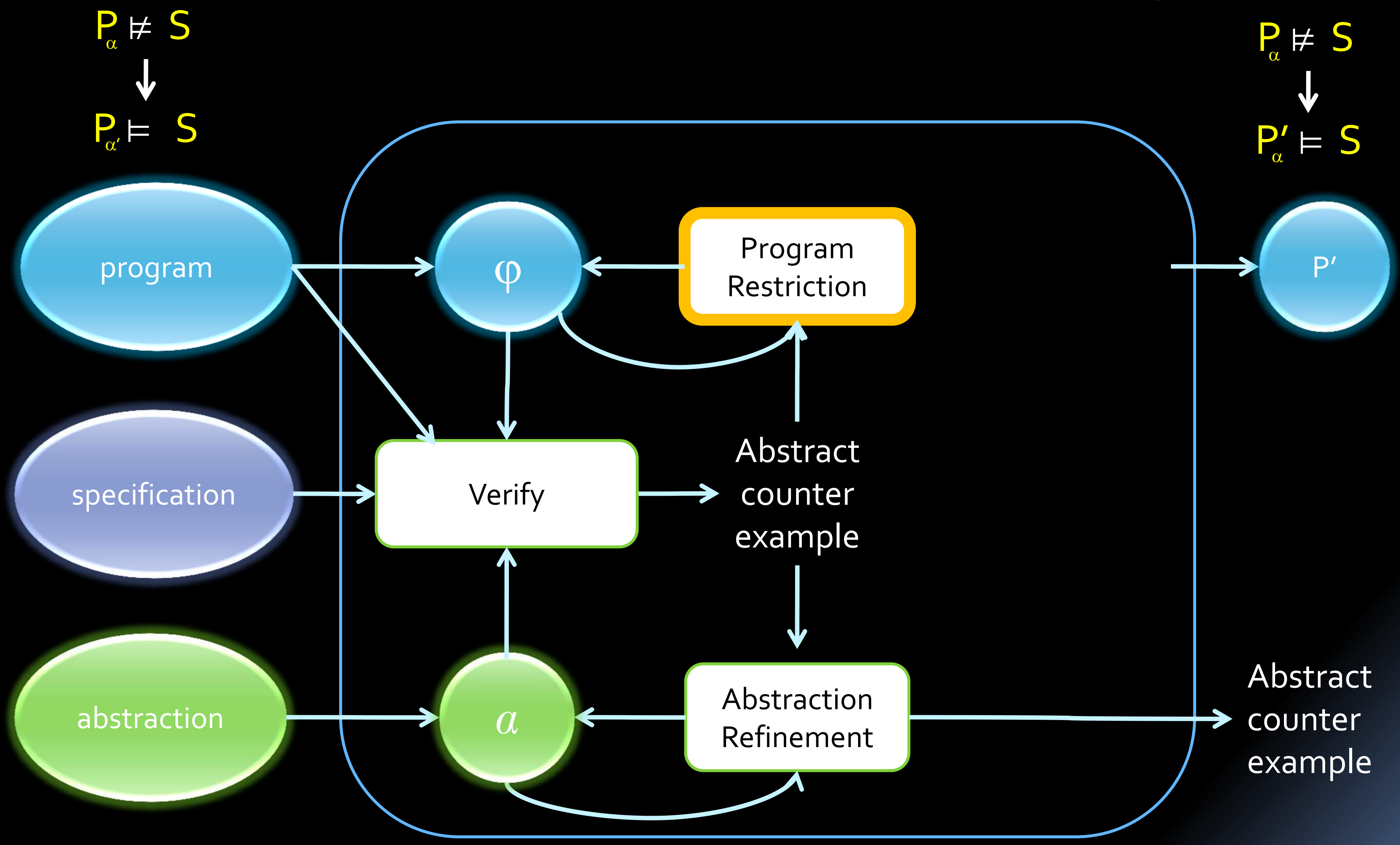
- Find minimal synchronization that makes the program satisfy the specification
  - Avoid all bad interleavings while permitting as many good interleavings as possible
- Assumption: we can prove that serial executions satisfy the specification
  - Interested in bad behaviors due to concurrency
- Handle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

Abstraction-Guided Synthesis of Synchronization

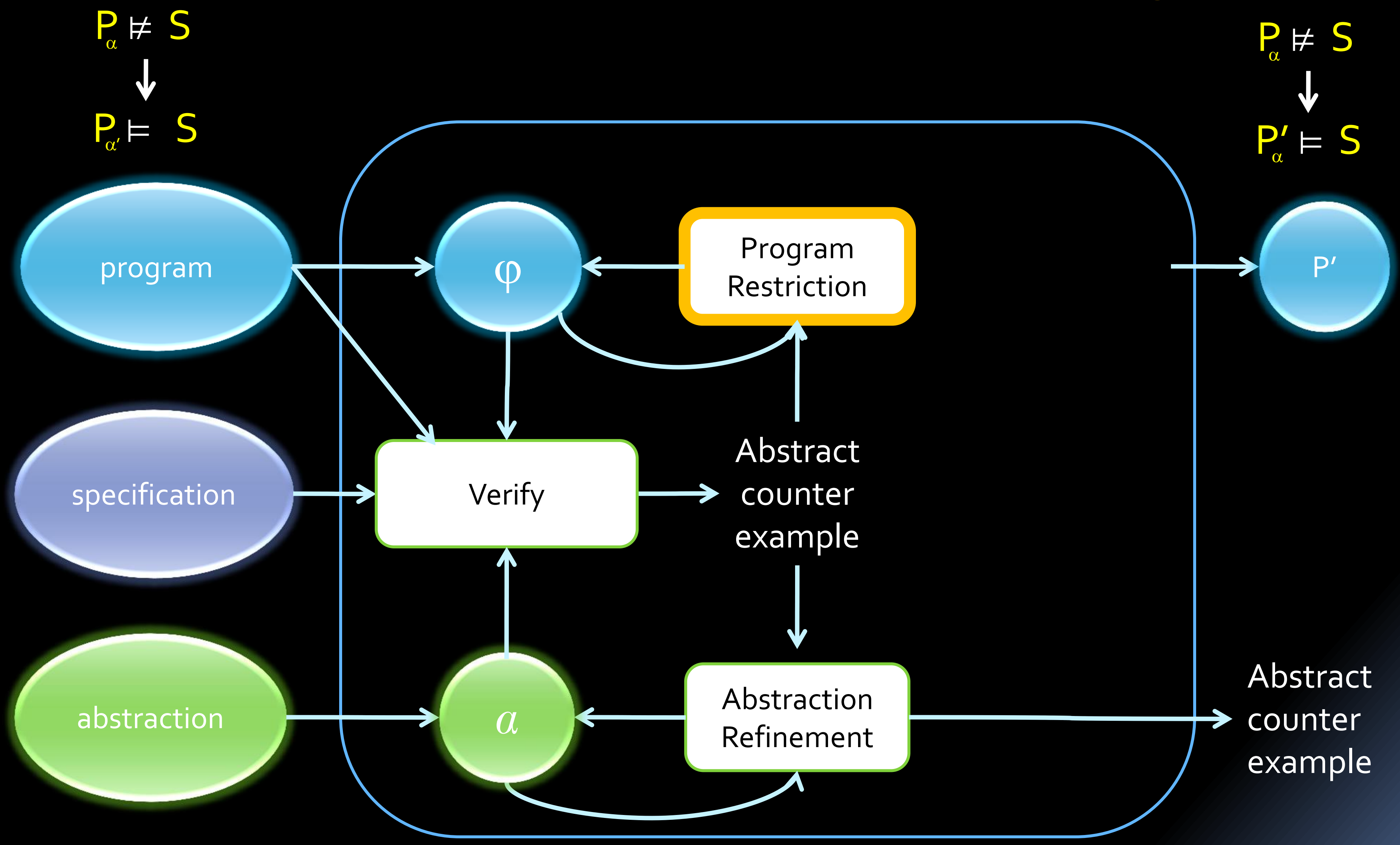
- Synthesis of synchronization via abstract interpretation
  - Compute over-approximation of all possible program executions
  - Add minimal synchronization to avoid (over-approximation of) bad interleavings

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

Instantiate for Concurrency

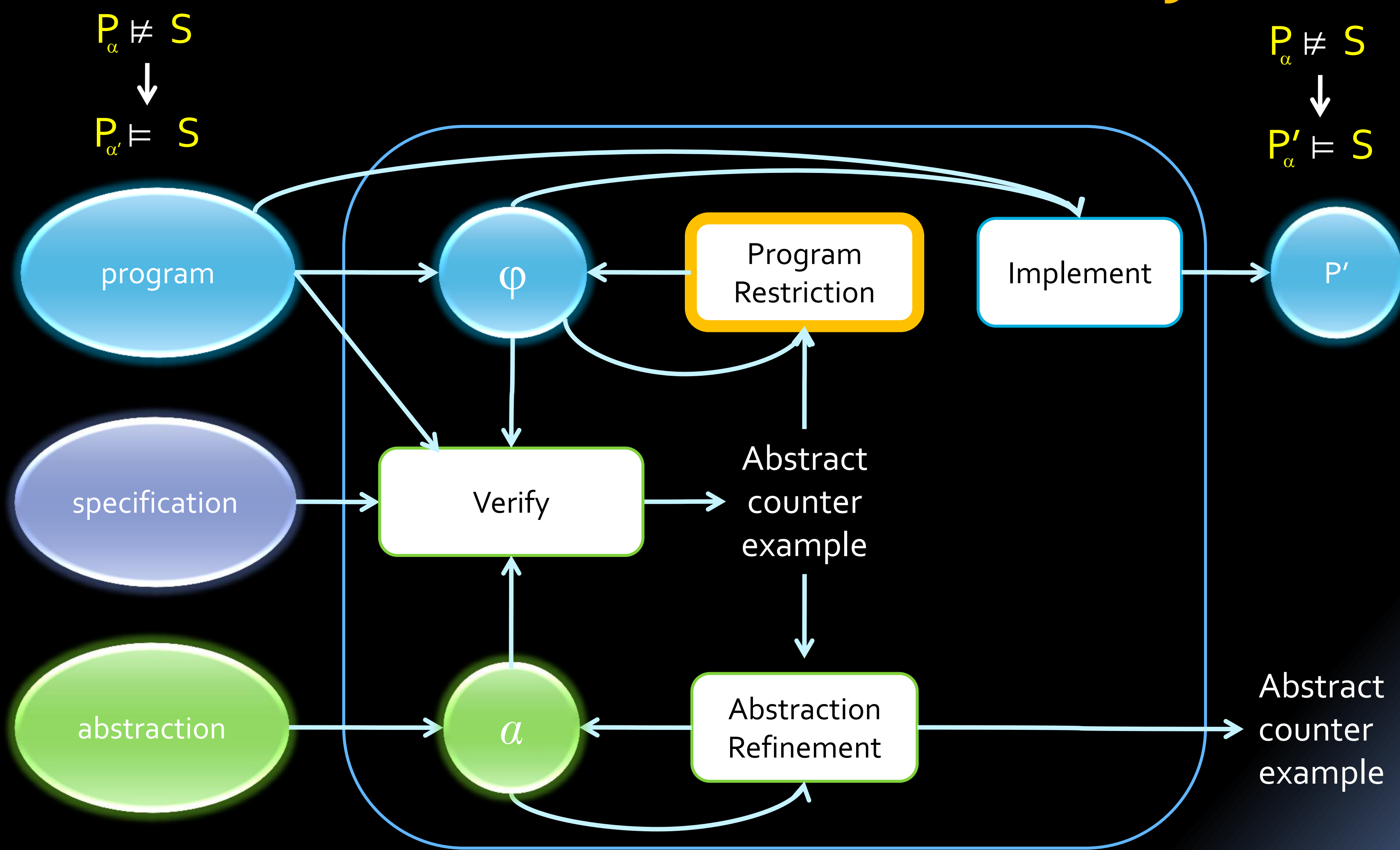


Instantiate for Concurrency



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
  }
}
```

Avoid interleaving with atomics

- 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

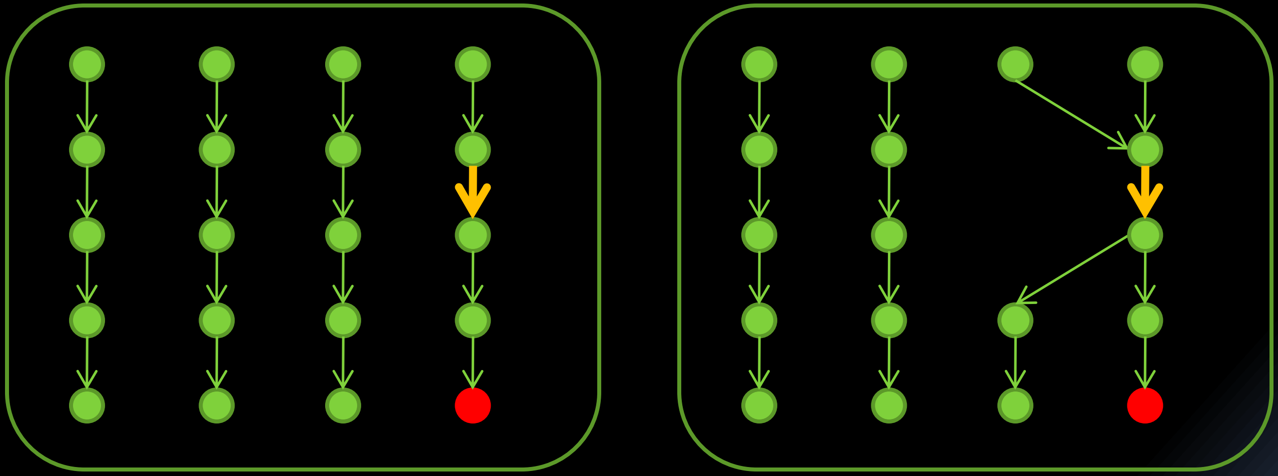
Thread A	||	Thread B
A1		B1
A2		B2

$\pi = A_1 B_1 A_2 B_2$

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

Avoid and abstraction

- ψ = avoid(π)
- Enforcing ψ avoids any abstract trace π' such that $\pi' \nvDash \psi$
- Potentially avoiding "good traces"
- Abstraction may affect our ability to avoid a smaller set of traces



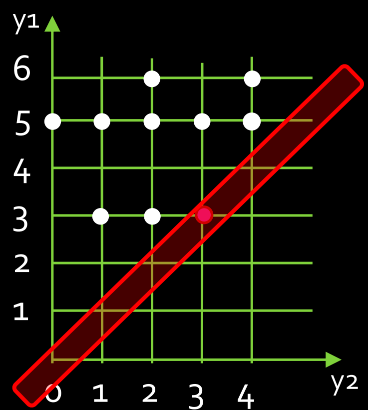
Example

T1 || T2 || T3

```
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

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

||

T2

```
1: z++
2: z++
```

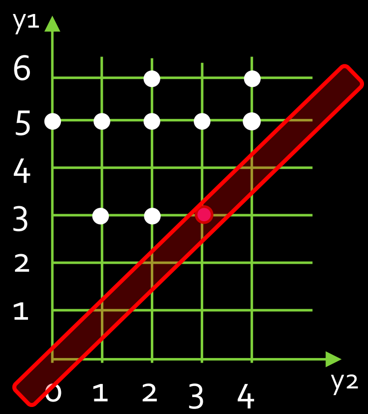
||

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

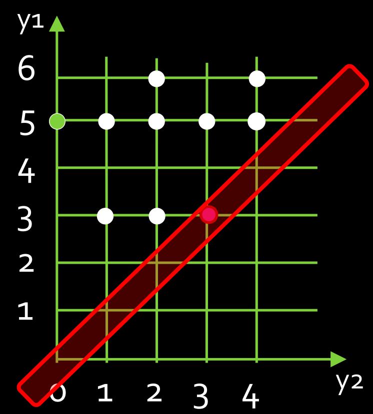


Concrete values

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

T1		T2		T3	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: Concrete Values

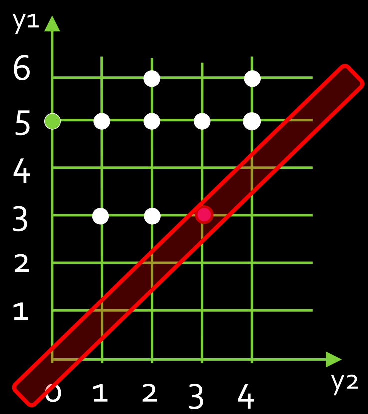


Concrete values

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

```
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: Concrete Values



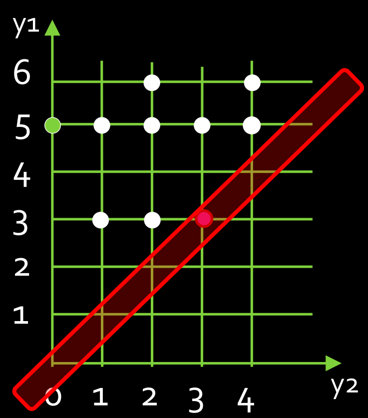
Concrete values

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: Concrete Values



Concrete values

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

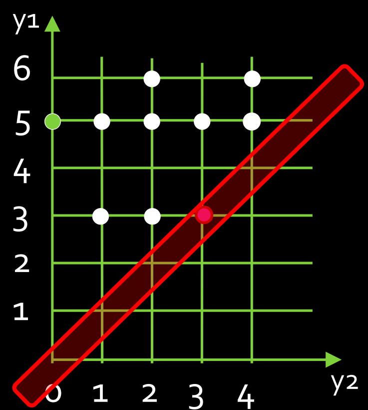
z++; x+=z; y1=f(x); z++; x+=z; y2=x;assert ➔ y1=3,y2=3

```
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

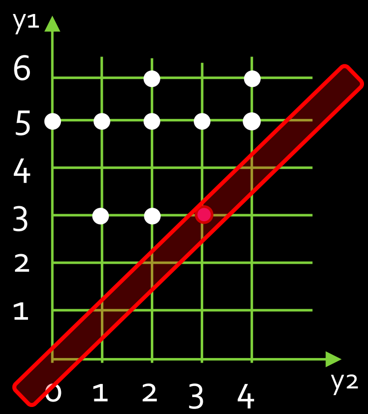
x += z; x += z; z++;z++;y1=f(x);y2=x;assert → y1=5,y2=0

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

⋮

```
T1              ||   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)   }
```

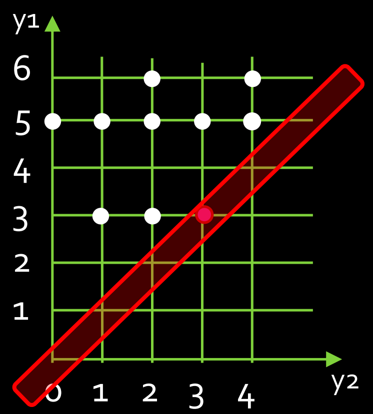
Example: Parity Abstraction



Concrete values

```
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



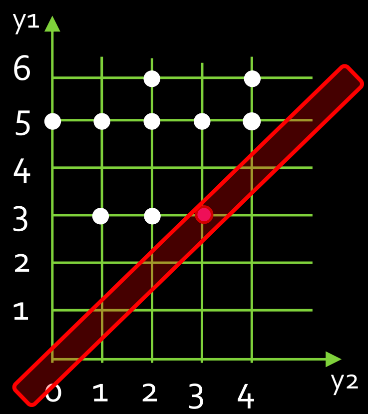
Concrete values

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

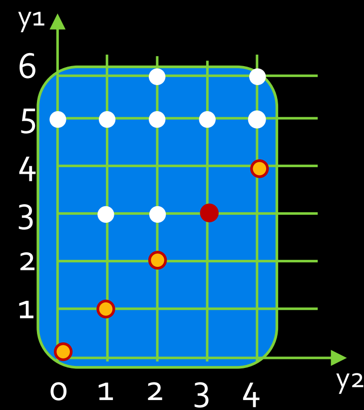
```
T1                 T2               T3
  1: x += z   ||    1: z++    ||     1: y1 = f(x)
  2: x += z   ||    2: z++    ||     2: y2 = x
                                     3: assert(y1 != y2)
```

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

Example: 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
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: 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

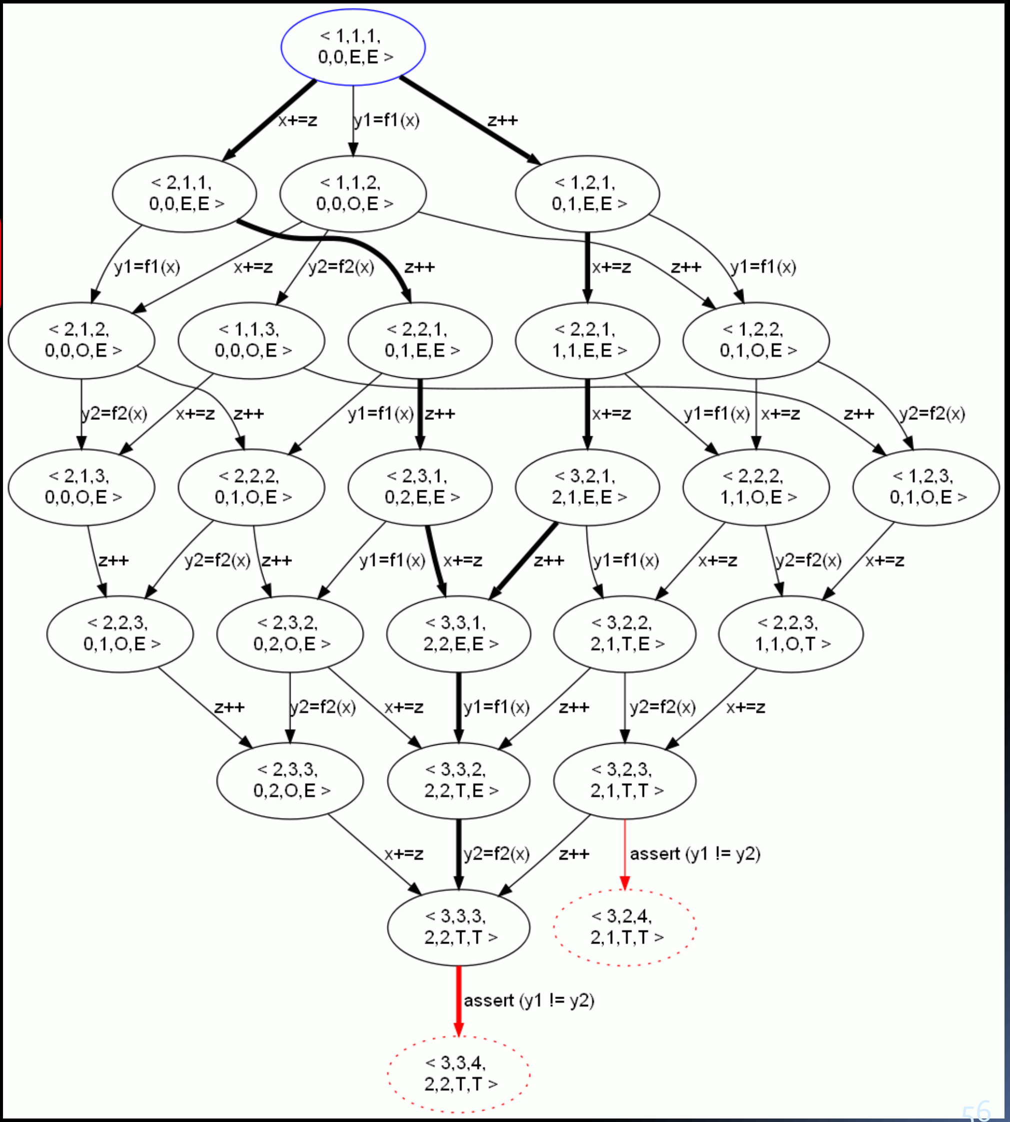
Example: Avoiding Bad Interleavings

```
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

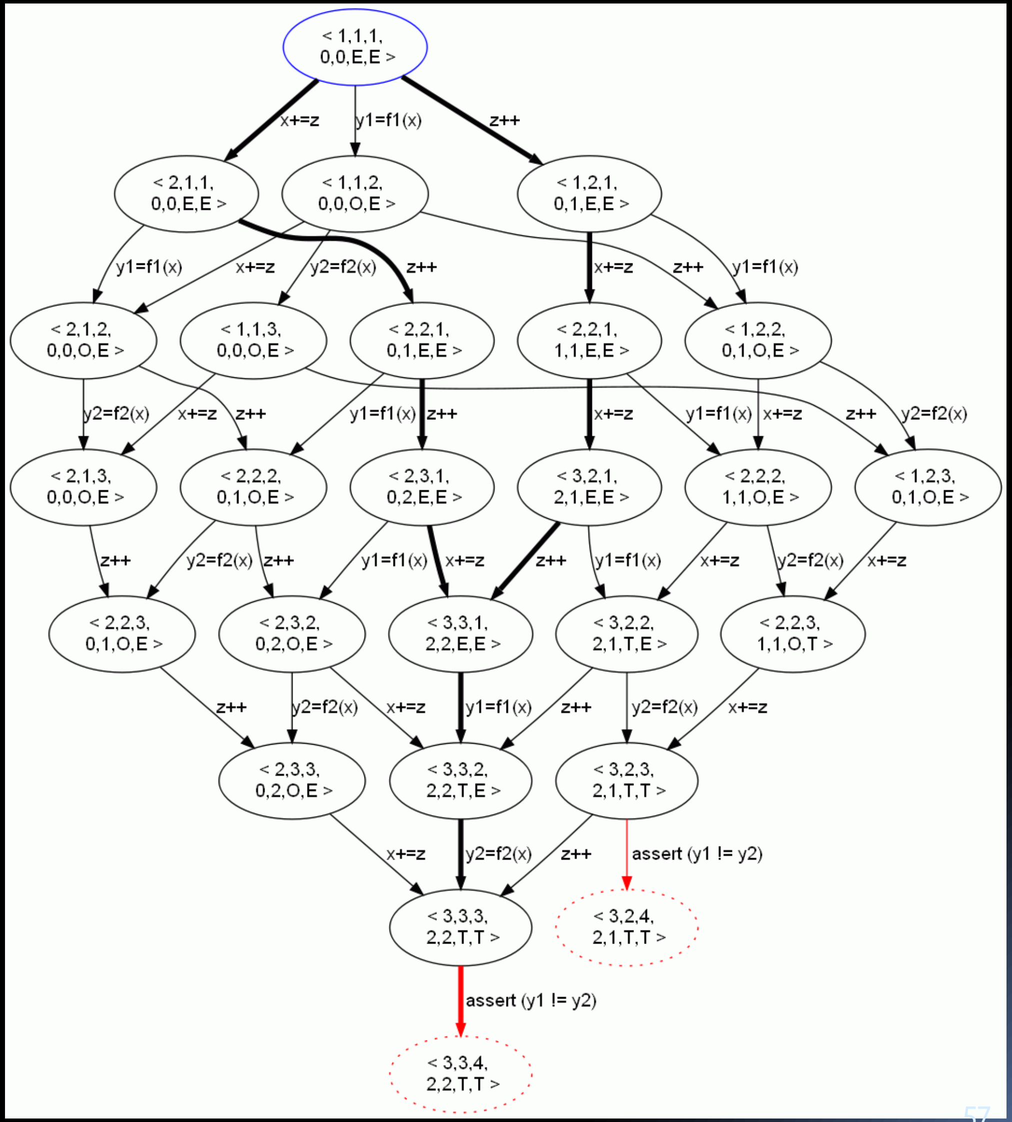
```
φ = true
while(true) {
  BadTraces={π|π∈(⟦P⟧_α ∩ ⟦φ⟧) and
          π ⊭ S }
  if (BadTraces is empty)
      return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    φ = φ ∧ avoid(π)
  } else {
    α = refine(α, π)
  }
}
```



φ = true

Example: Avoiding Bad Interleavings

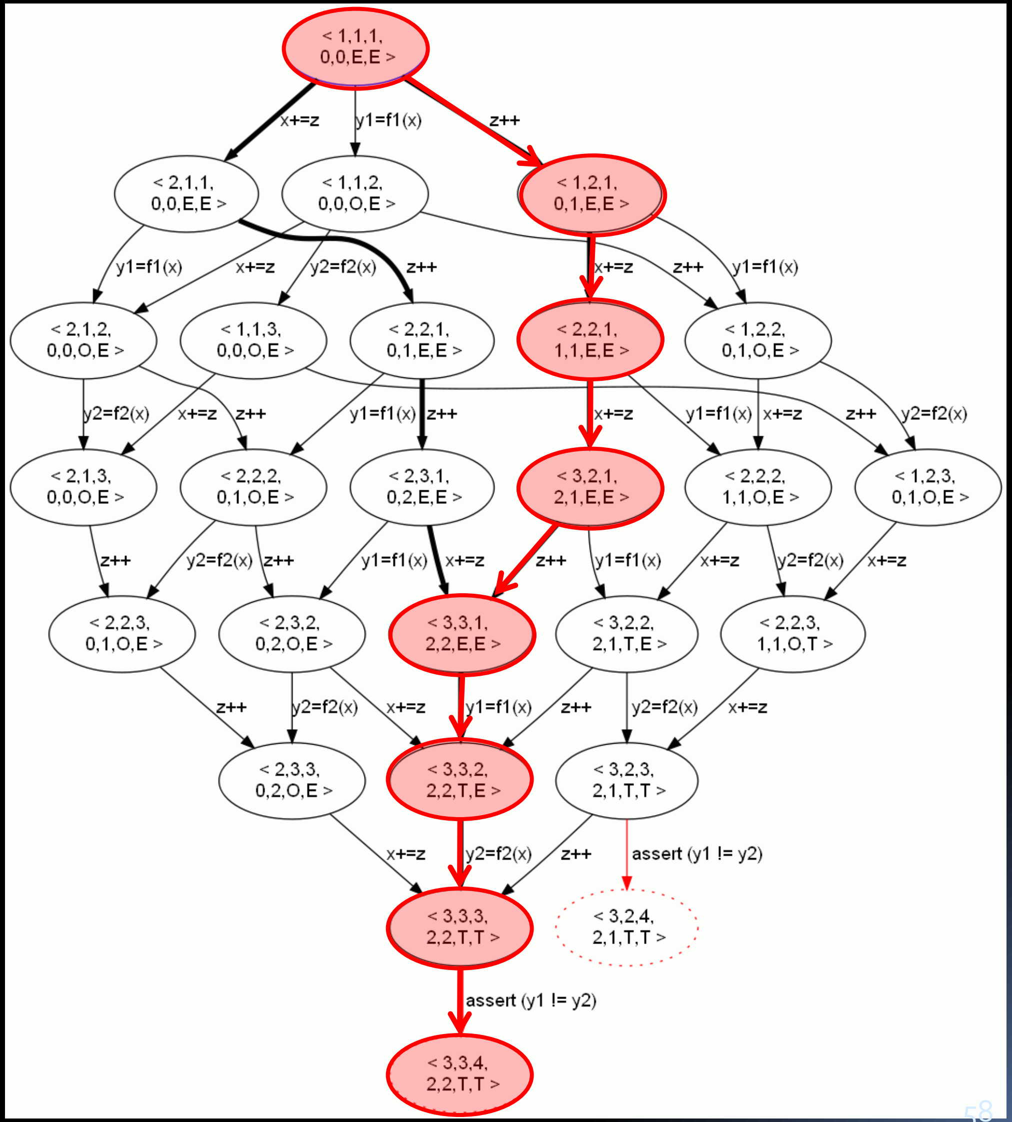
```
φ = true
while(true) {
   BadTraces={π|π∈(⟦P⟧_α ∩ ⟦φ⟧) and
              π ⊭ S }
  if (BadTraces is empty)
      return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    φ = φ ∧ avoid(π)
  } else {
    α = refine(α, π)
  }
}
```



φ = true

Example: Avoiding Bad Interleavings

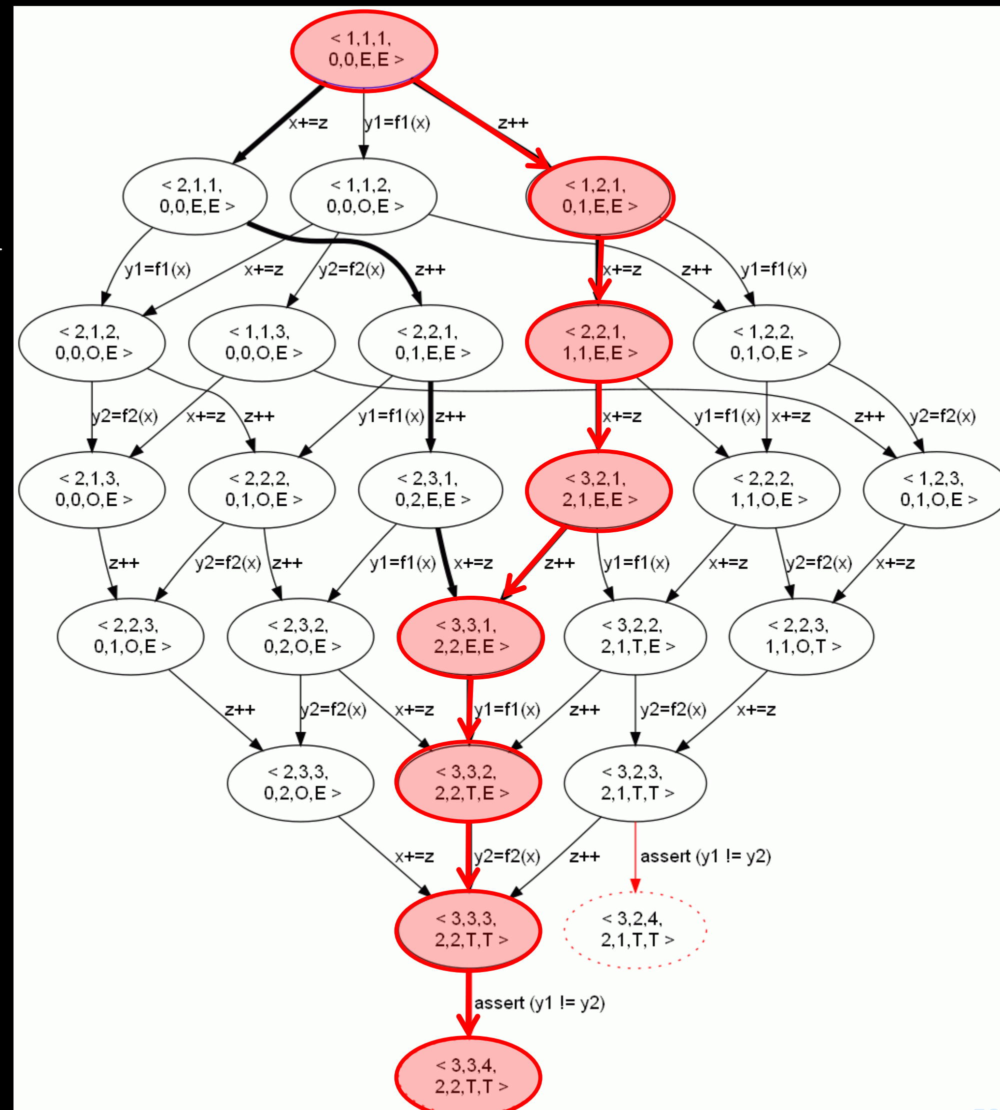
```
φ = true
while(true) {
   BadTraces={π|π∈(⟦P⟧α ∩ ⟦φ⟧) and
                π ⊭ S }
  if (BadTraces is empty)
      return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    φ = φ ∧ avoid(π)
  } else {
    α = refine(α, π)
  }
}
```



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)
   }
}
```



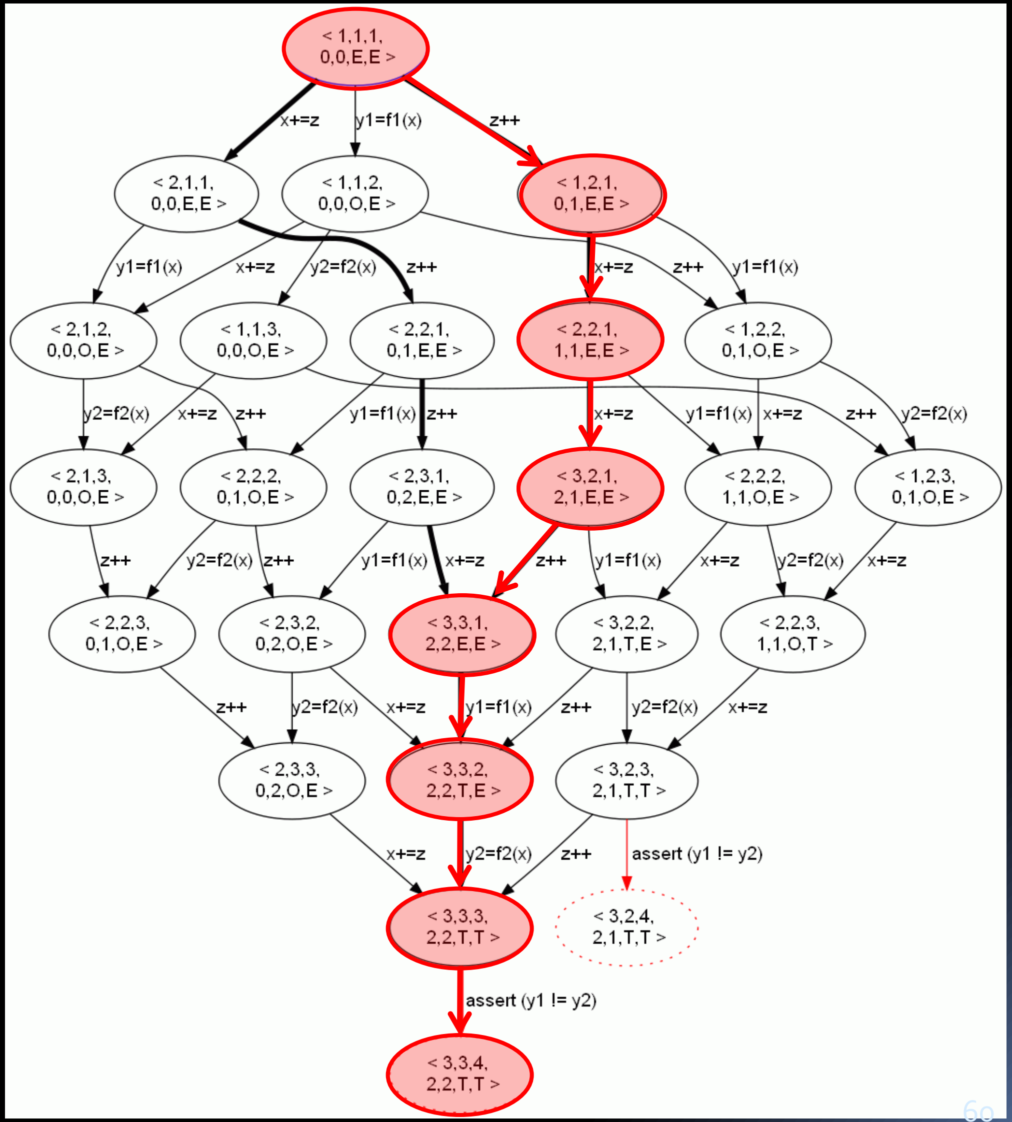
φ = true

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

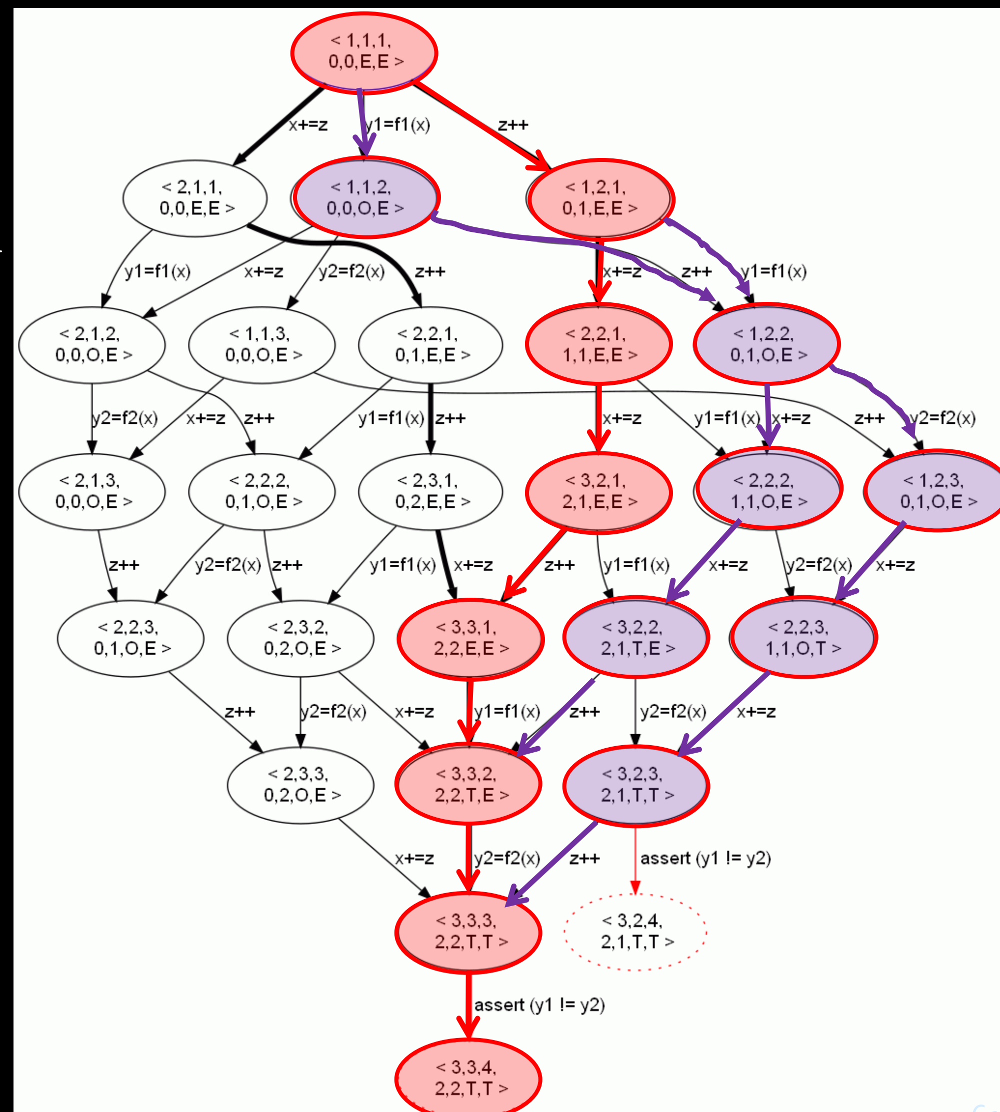


Example: Avoiding Bad Interleavings

```
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)
   }
}
```

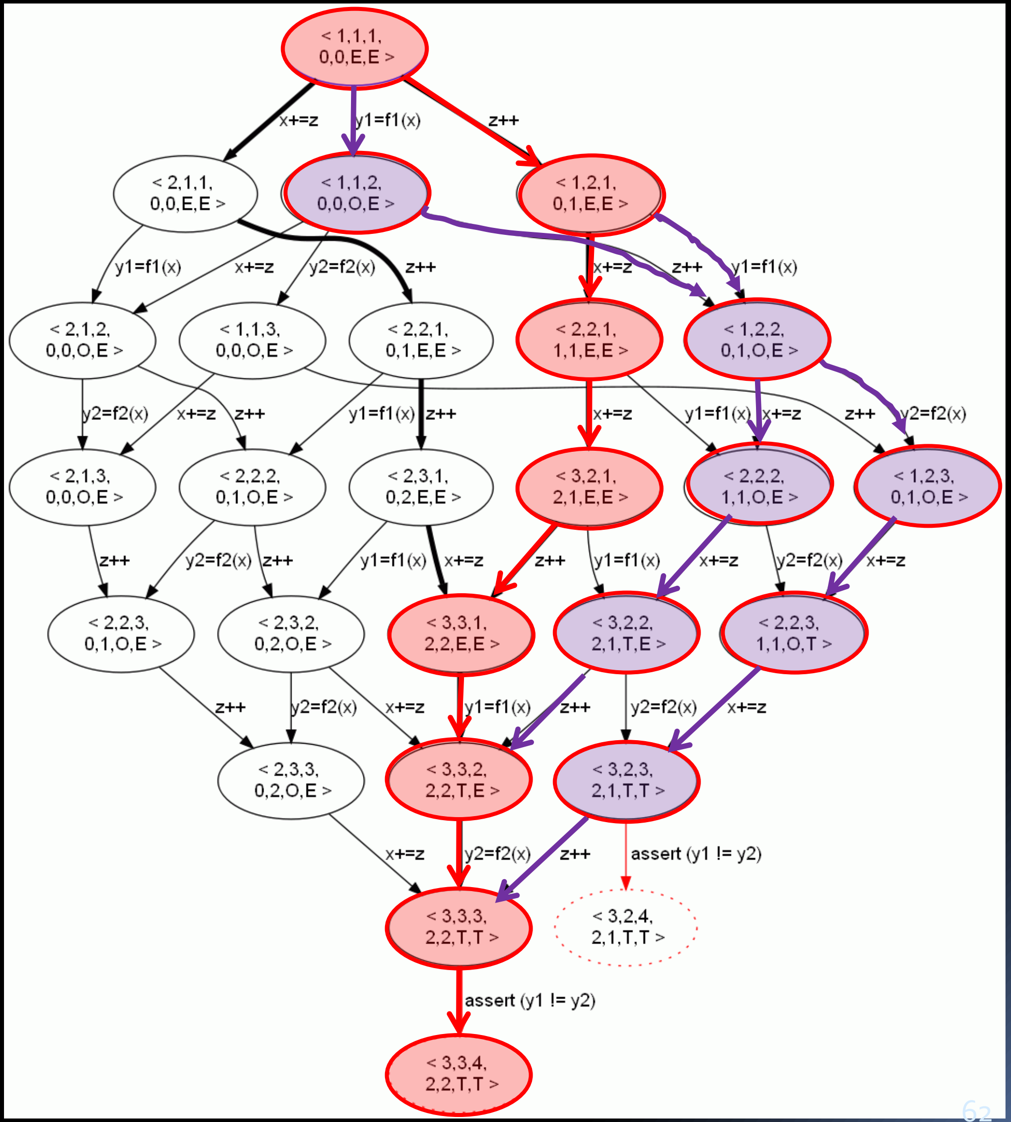
avoid(π_1) = [z++,z++]

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)
  }
}
```

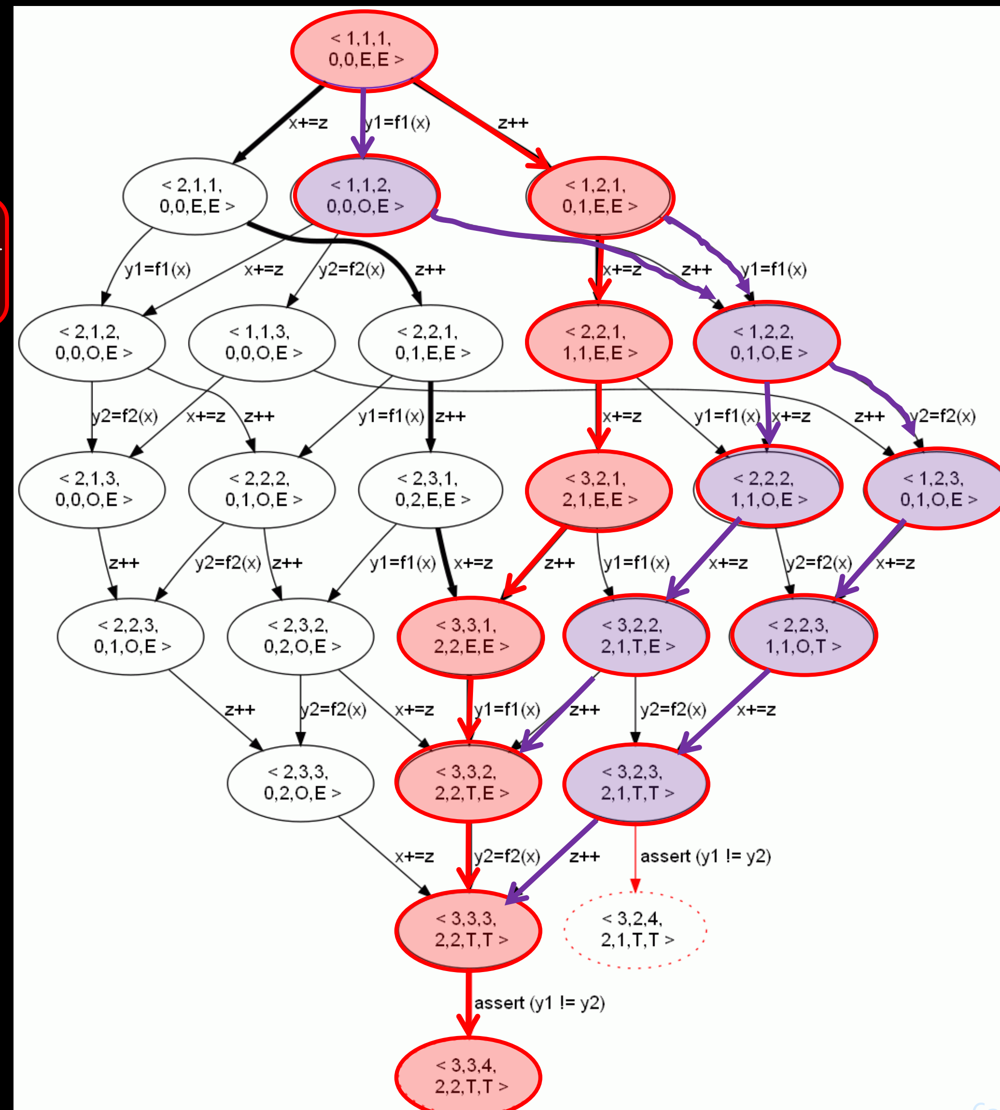


φ = [z++,z++]

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)
   }
}
```

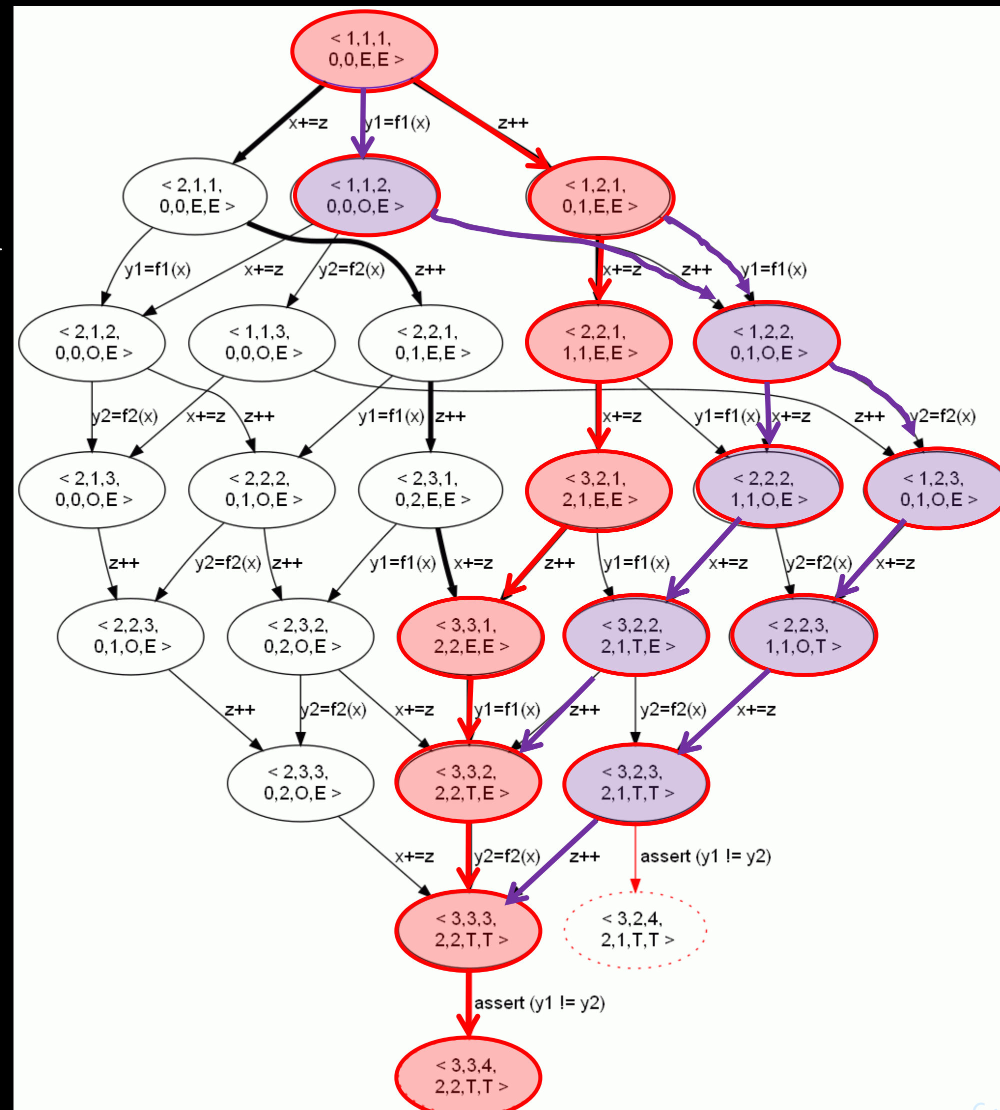
φ = [z++,z++]



Example: Avoiding Bad Interleavings

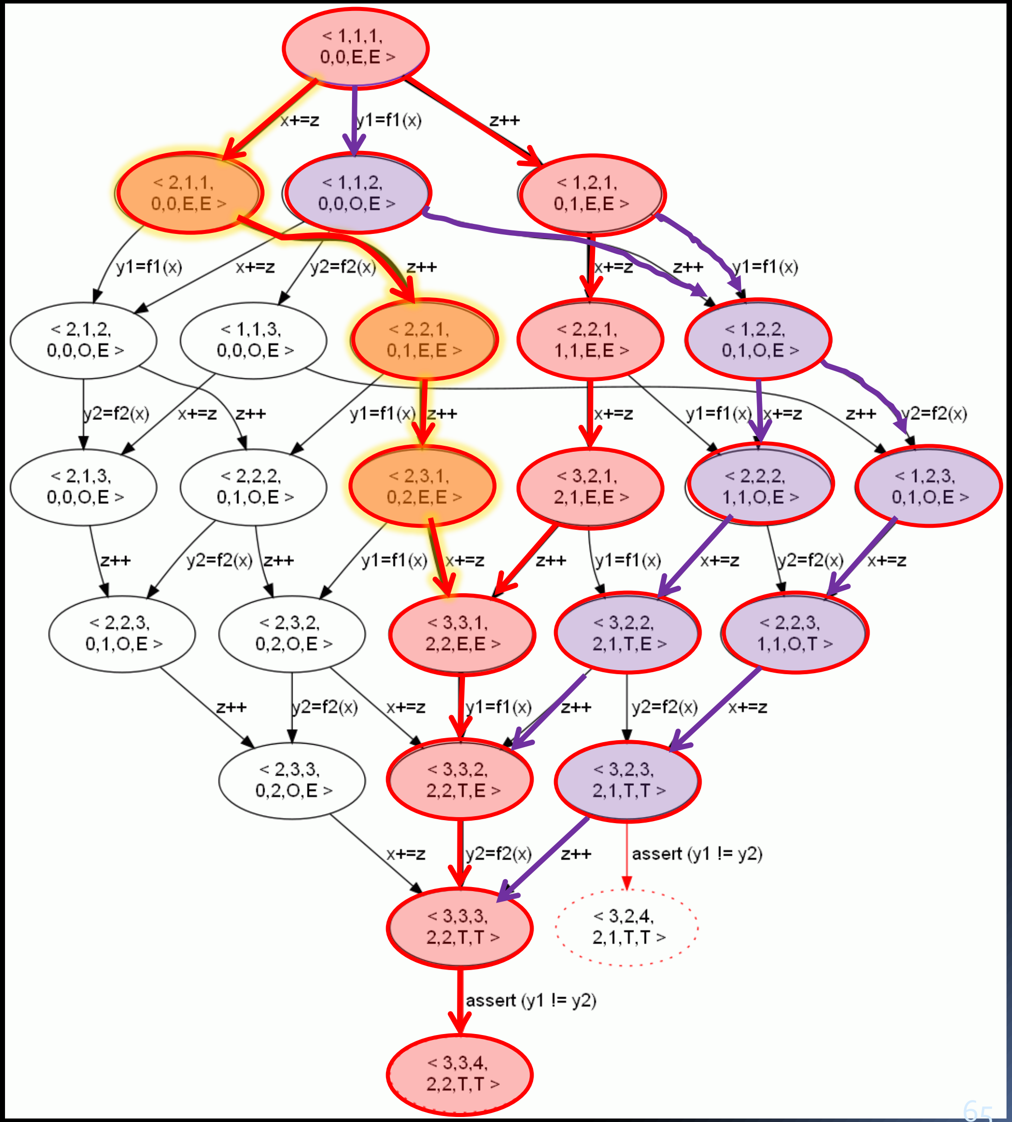
```
φ = true
while(true) {
   BadTraces={π|π∈(⟦P⟧α ∩ ⟦φ⟧) and
               π ⊭ S }
  if (BadTraces is empty)
      return implement(P,φ)
  select π ∈ BadTraces
  if (?) {
    φ = φ ∧ avoid(π)
  } else {
    α = refine(α, π)
  }
}
```

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)
  }
}
```



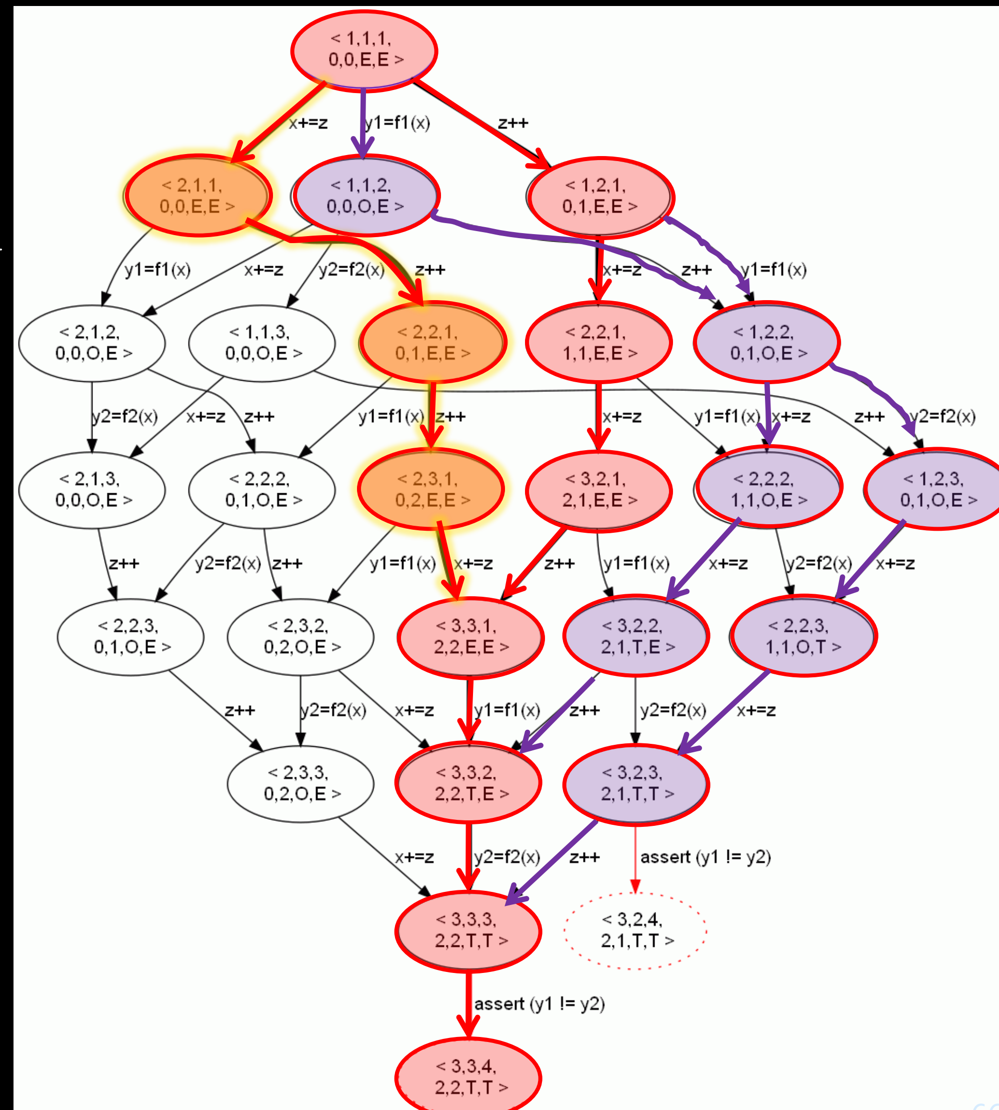
φ = [z++,z++]

Example: Avoiding Bad Interleavings

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

avoid(π_2) = [x+=z, x+=z]

φ = [z++, z++]



Example: Avoiding Bad Interleavings

φ = true

while(true) {

  BadTraces={π|$\pi \in (⟦P⟧_a \cap ⟦\varphi⟧)$ and

          $\pi \not\models S$ }

  if (BadTraces is empty)

    return implement(P,φ)

  select $\pi \in$ BadTraces

  if (?) {

    $\boxed{\varphi = \varphi \wedge \text{avoid}(\pi)}$

  } else {

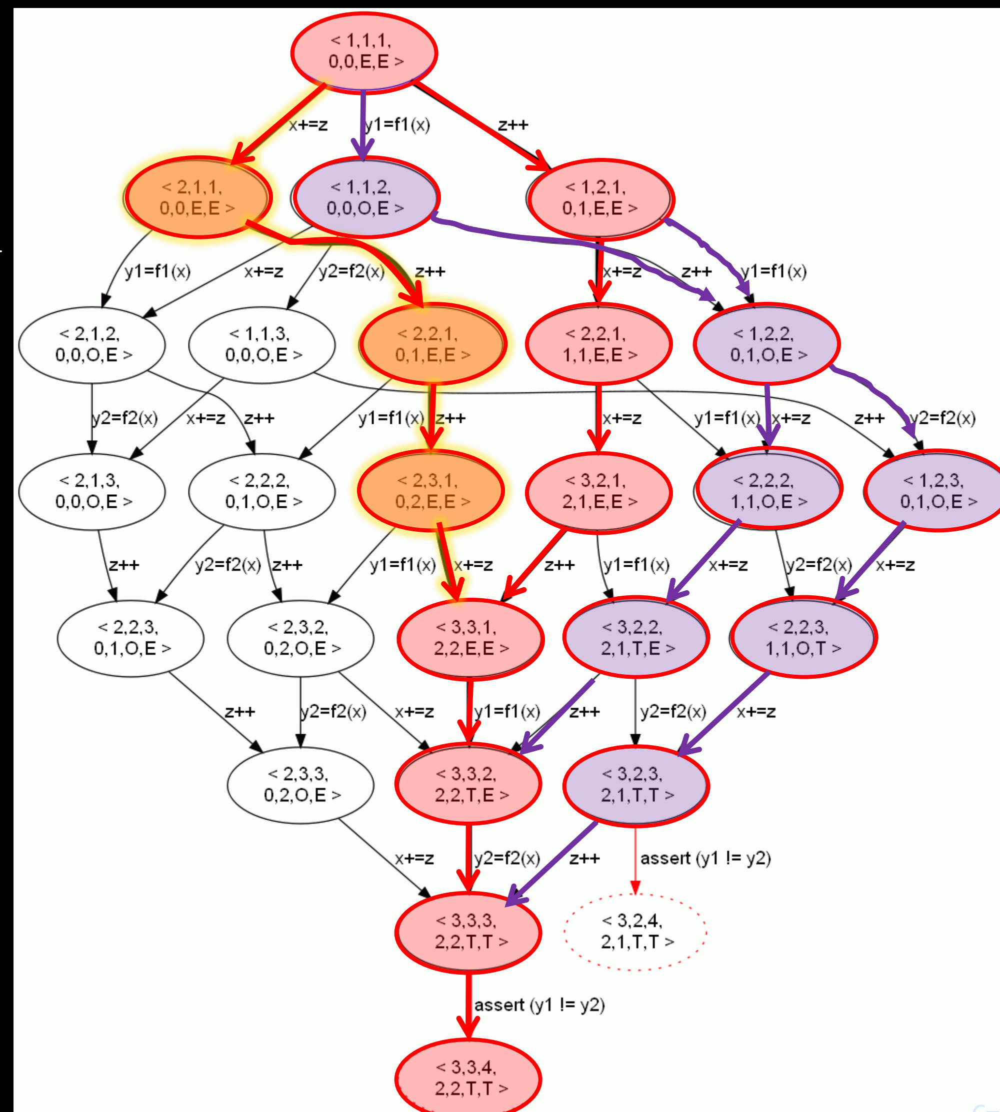
    a = refine(a, π)

  }

}

avoid(π_2) = [x+=z,x+=z]

φ = [z++,z++]$\wedge$[x+=z,x+=z]

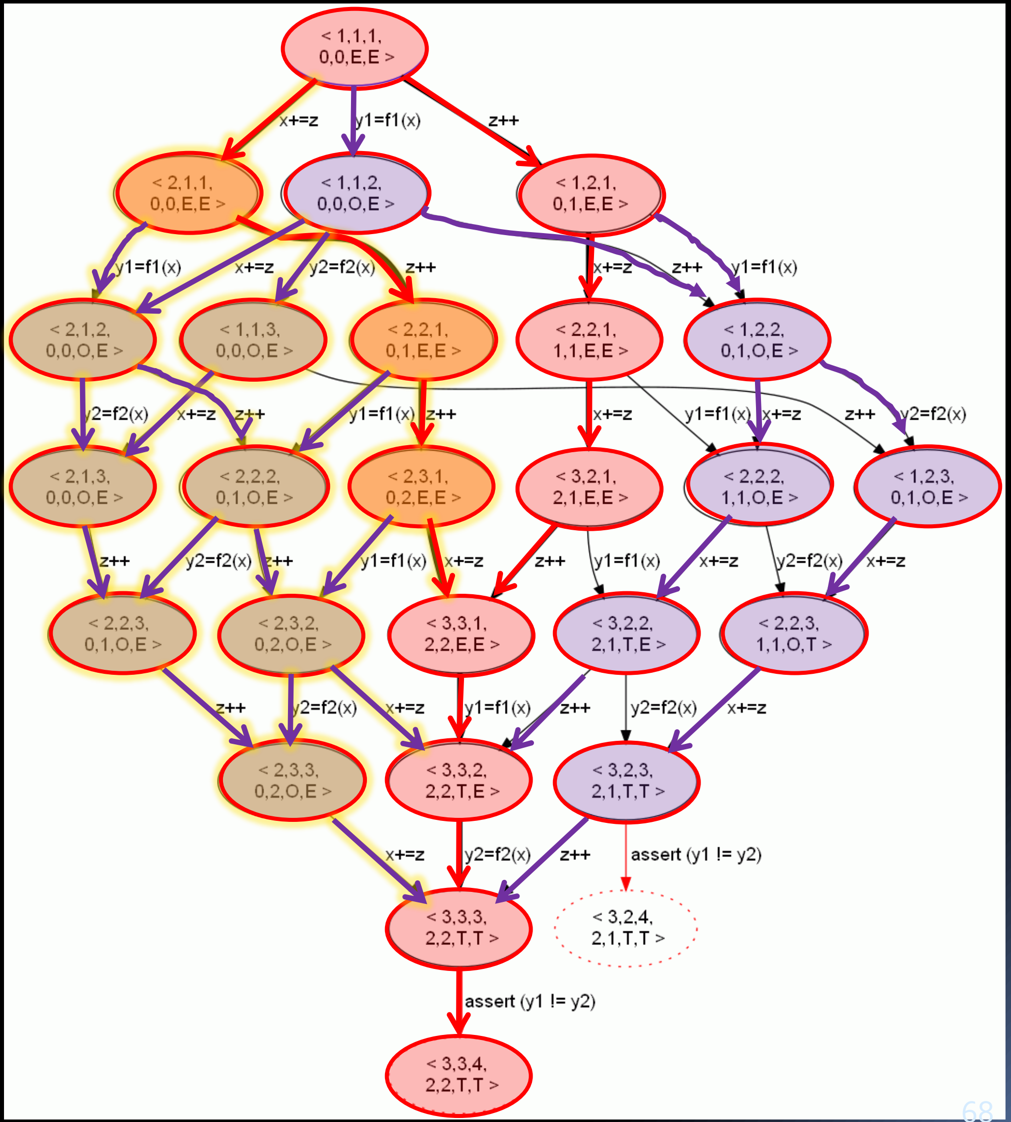


Example: Avoiding Bad Interleavings

```
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 (?) {
     φ = φ ∧ avoid(π)
   } else {
     a = refine(a, π)
   }
}
```

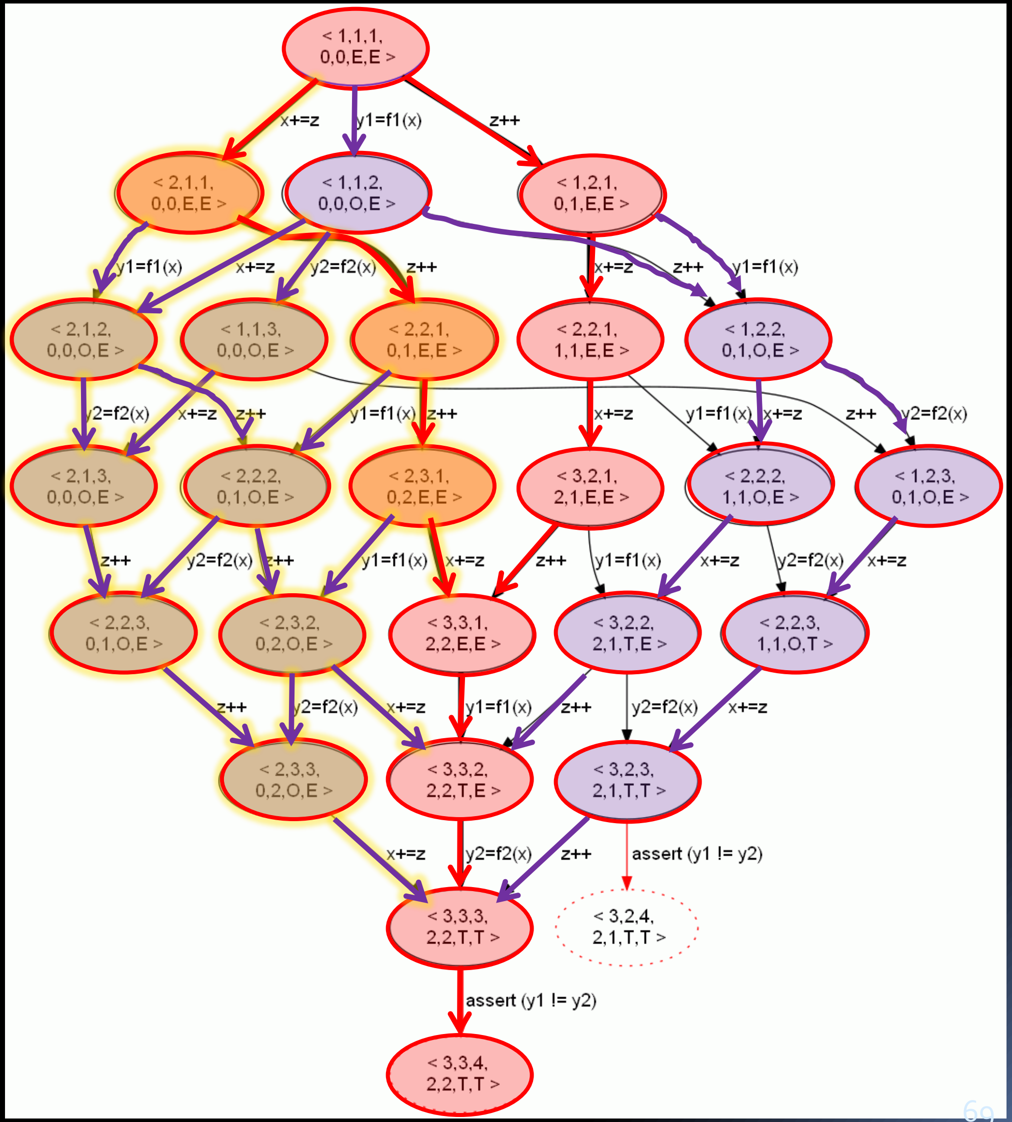
avoid(π_2) $= [x{+}{=}z, x{+}{=}z]$

$\varphi = [z{+}{+}, z{+}{+}] \wedge [x{+}{=}z, x{+}{=}z]$



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]

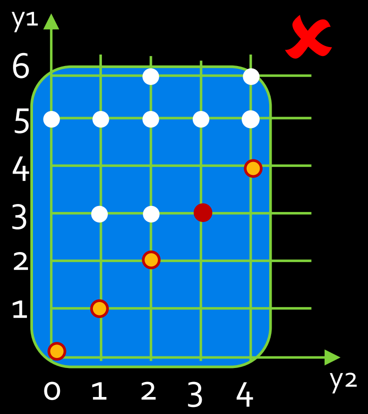
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

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



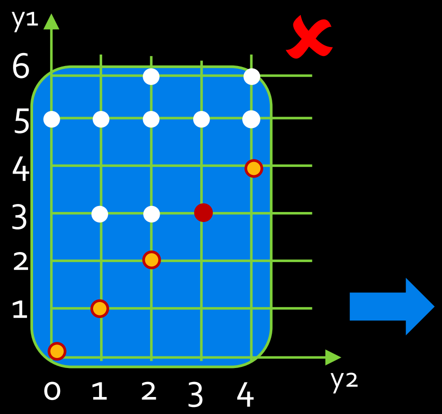
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

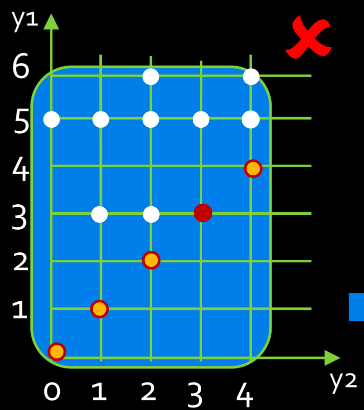
T3 y1=f(x)
y2=x
assert
y1!= y2



Example: Avoiding Bad Interleavings

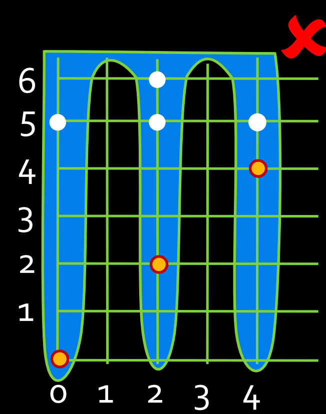
parity

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



parity

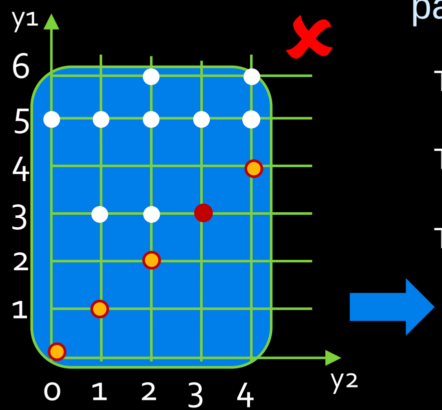
```
T1  x+=z;
    x+=z
T2 [z++;
    z++;
T3  y1=f(x)
    y2=x
    assert
      y1!= y2
```



Example: Avoiding Bad Interleavings

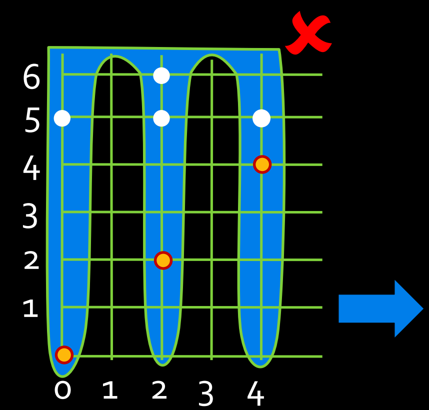
parity

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



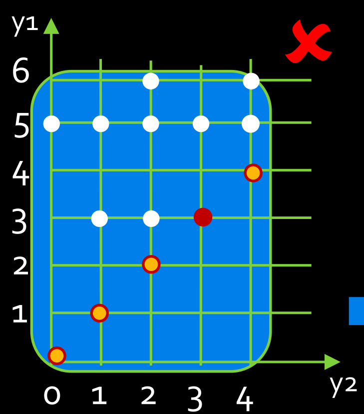
Example: Avoiding Bad Interleavings

parity

T1 x+=z;
x+=z

T2 z++;
z++;

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

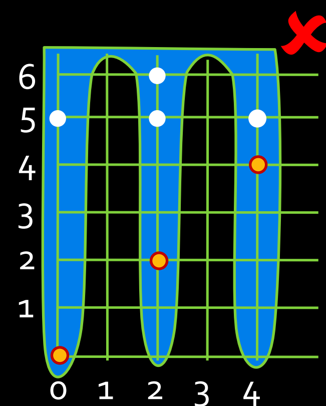


parity

T1 x+=z;
x+=z

T2 [z++;
z++;

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

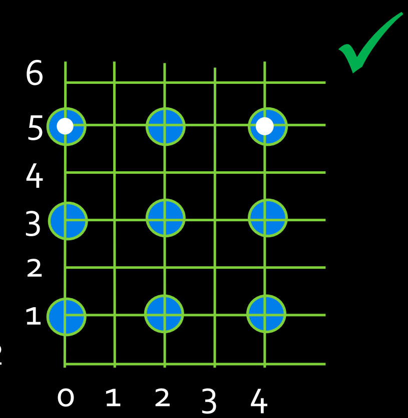


parity

T1 [x+=z;
x+=z

T2 [z++;
z++;

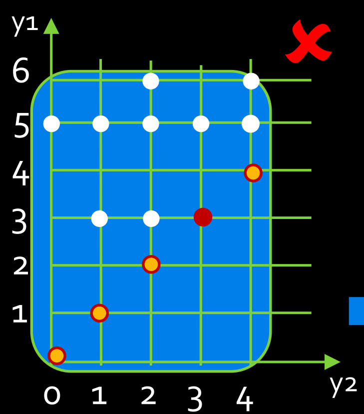
T3 y1=f(x)
y2=x
assert
y1!= y2



Example: Avoiding Bad Interleavings

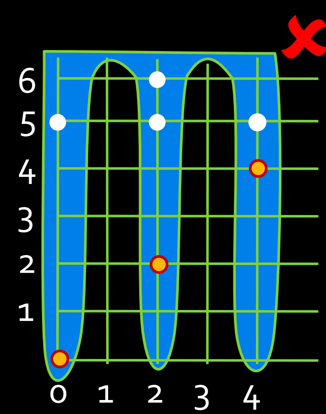
parity

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



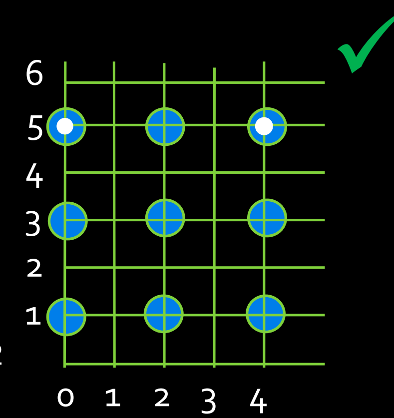
parity

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



parity

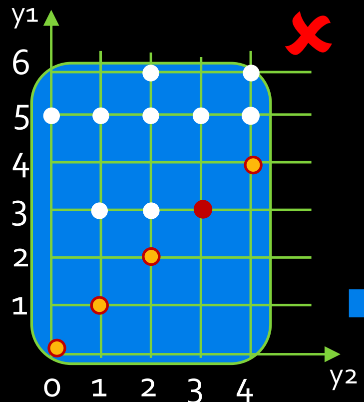
```
T1 [x+=z;
    x+=z
T2 [z++;
    z++;
T3  y1=f(x)
    y2=x
    assert
      y1!= y2
```



But we can also refine the abstraction…

parity

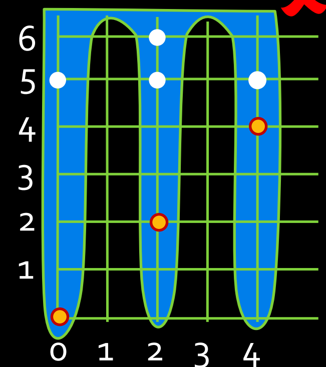
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(a)

parity

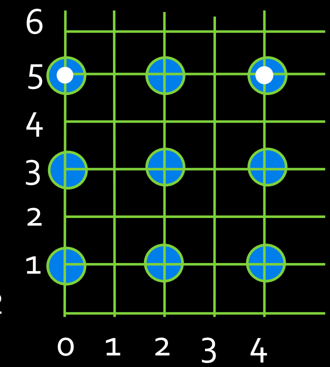
T1 x+=z;
x+=z
T2 [z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(b)

parity

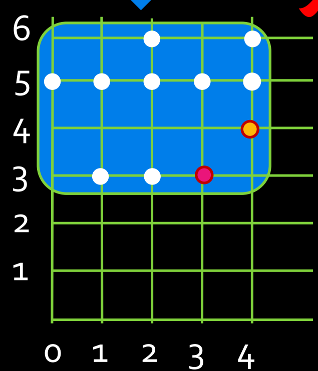
T1 [x+=z;
x+=z
T2 [z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(c)

interval

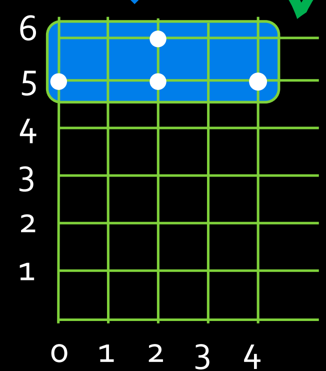
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(d)

interval

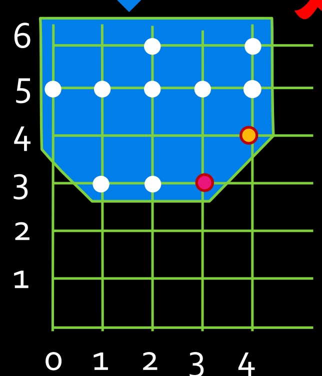
T1 x+=z;
x+=z
T2 [z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(e)

octagon

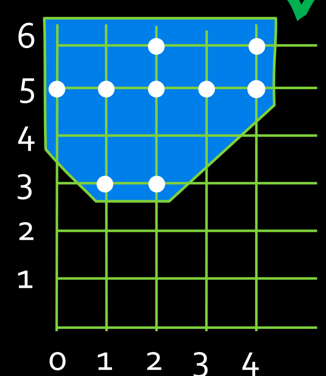
T1 x+=z;
x+=z
T2 z++;
z++;
T3 y1=f(x)
y2=x
assert
y1!= y2



(f)

octagon

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



(g)

Multiple Solutions

- Performance: smallest atomic sections

- Interval abstraction for our example produces the atomicity constraint:

  ([x+=z,x+=z] ∨ [z++,z++])
  ∧ ([y1=f(x),y2=x] ∨ [x+=z,x+=z] ∨ [z++,z++])

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

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

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)