

AAA616: Program Analysis

Lecture 3 – Concolic Testing*

Hakjoo Oh
2026 Spring

Motivation

- Writing and maintaining tests is tedious and error-prone
- Idea: [Automated Test Generation](#)
 - Generate a regression test suite
 - Execute all reachable statements
 - Catch any assertion violations

Existing Approach 1

- Random Testing
 - Generate random inputs
 - Execute the program on those (concrete) inputs
- Problem
 - Probability of catching error can be astronomically small

```
void testme (int x) {  
    if (x == 94389) {  
        ERROR  
    }  
}
```

Probability of ERROR:

$$1/2^{32} = 0.000000023 \%$$

Existing Approach 2

- Symbolic Execution
 - Use symbolic values for inputs
 - Execute program symbolically on symbolic input values
 - Collect symbolic path constraints
 - Use theorem prover to check if a branch can be taken
- Problem
 - Incomplete theorem prover
 - Limited scalability

Symbolic Execution

Execution Tree

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
1   z := double (y);  
2  
   if (z==x) {  
3       if (x>y+10) {  
4           Crash  
5       } else { 5 }  
6   }
```

1 $x: \alpha, y: \beta$
pc: *true*

Symbolic Execution

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
1   z := double (y);
```

```
2  
   if (z==x) {
```

```
3       if (x>y+10) {
```

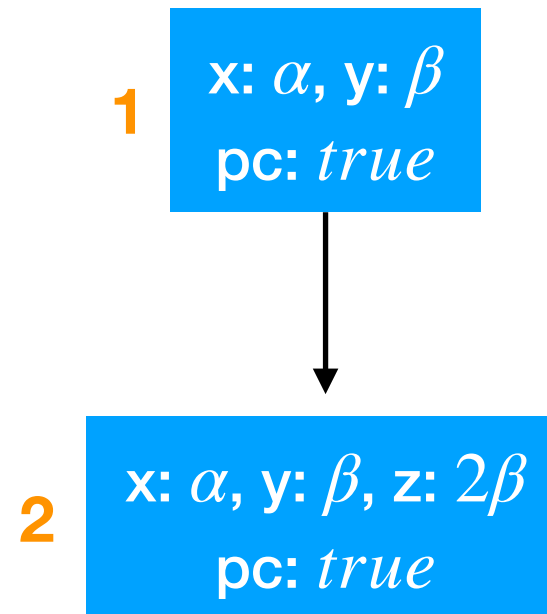
```
4         Crash
```

```
       } else { 5 }
```

```
   }
```

```
6 }
```

Execution Tree



Symbolic Execution

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
1   z := double (y);
```

```
2  
   if (z==x) {
```

```
3       if (x>y+10) {
```

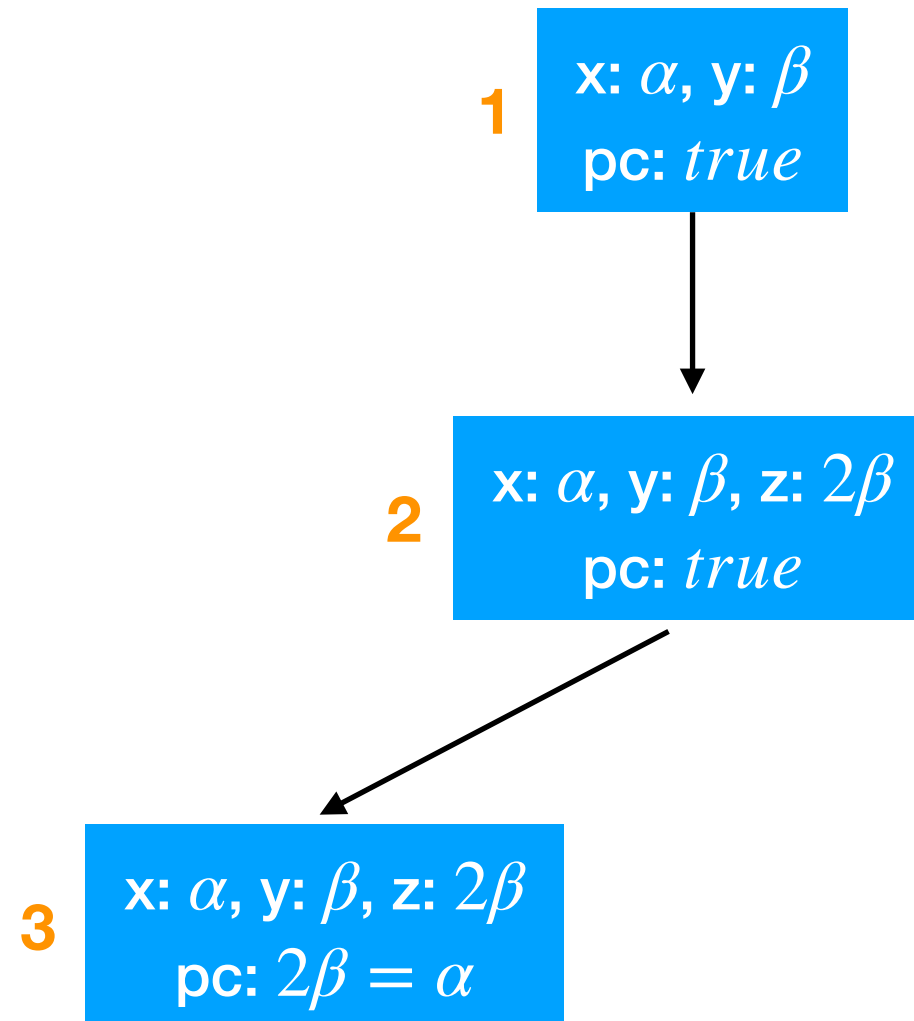
```
4         Crash
```

```
       } else { 5 }
```

```
   }
```

```
6 }
```

Execution Tree



Symbolic Execution

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
1   z := double (y);
```

```
2   if (z==x) {
```

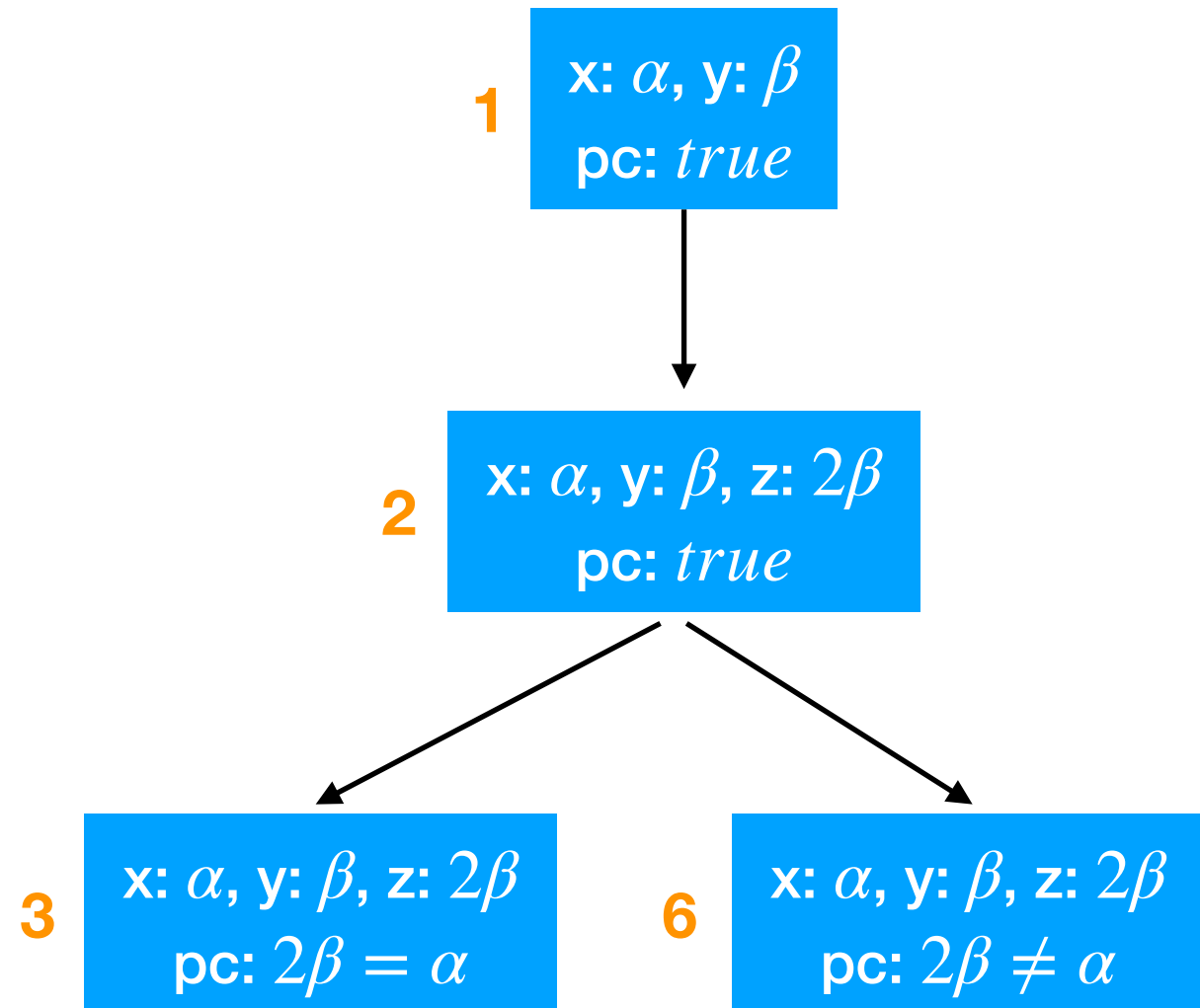
```
3       if (x>y+10) {
```

```
4         Crash
```

```
5       } else {
```

```
6   }
```

Execution Tree



Symbolic Execution

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
1   z := double (y);
```

```
2   if (z==x) {
```

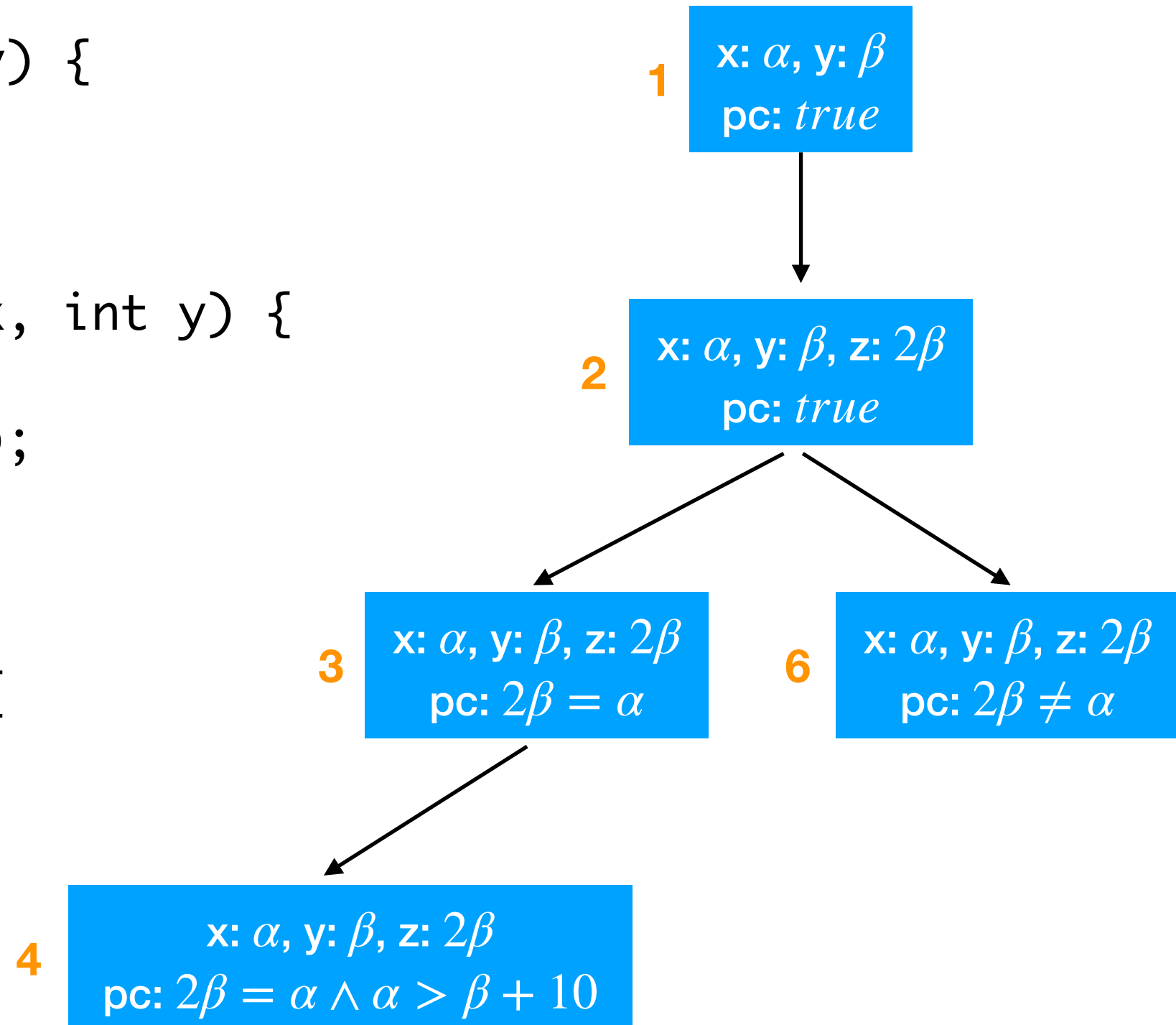
```
3       if (x>y+10) {
```

```
4       Crash
```

```
5       } else {
```

```
6   }
```

Execution Tree



Symbolic Execution

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
1  z := double (y);
```

```
2  if (z==x) {
```

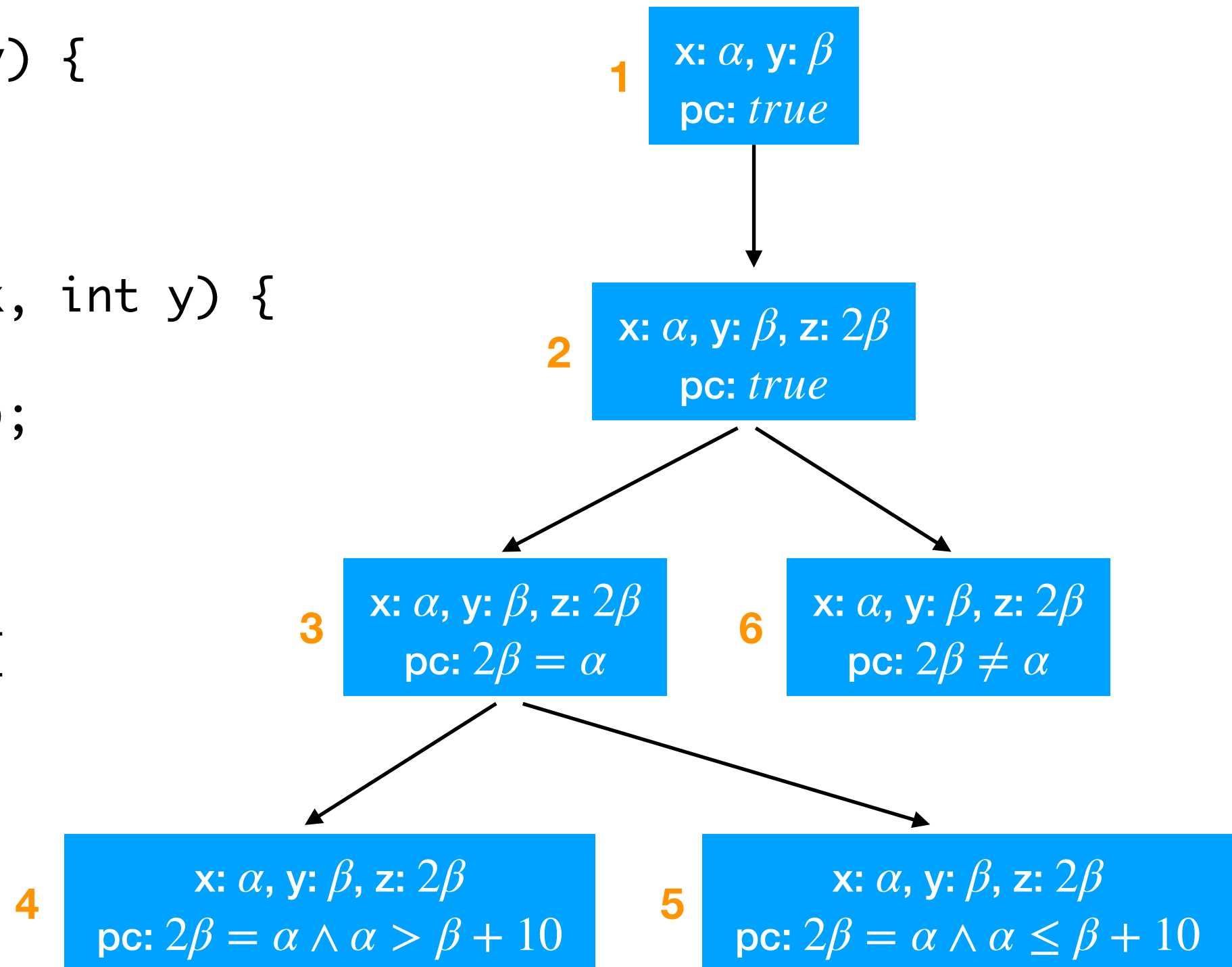
```
3      if (x>y+10) {
```

```
4      Crash
```

```
5      } else {
```

```
6  }
```

Execution Tree



Limitation of Symbolic Execution

```
int foo (int v) {  
    return secure_hash(v);  
}  
  
void testme(int x, int y) {  
  
    z := foo (y);  
  
    if (z==x) {  
  
        if (x>y+10) {  
            Crash  
        } else {  
        }  
    }  
}
```

Concolic Testing

- Approach
 - Store program state **concretely** and **symbolically**
 - Solve constraints to guide execution at branch points
 - Explore all execution paths of the unit tested
 - Use concrete values to simplify symbolic constraints
- Example of **hybrid analysis**
 - Collaboratively combines dynamic and static analysis

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    ← z := double (y);
```

```
    if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else { }
```

```
    }
```

```
}
```

Concrete
State

x=22, y=7

Symbolic
State

x=α, y=β
true

1st iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    z := double (y);
```

```
    ← if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else {    }
```

```
    }
```

```
}
```

Concrete
State

x=22, y=7,
z=14

Symbolic
State

x=α, y=β, z=2*β
true

1st iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
    z := double (y);  
    if (z==x) {  
        if (x>y+10) {  
            Crash  
        } else {  
        }  
    }  
}
```

Concrete
State

Symbolic
State

x=22, y=7,
z=14

x=a, y= β , z=2* β
2* $\beta \neq a$

1st iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    z := double (y);
```

```
    if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else {    }
```

```
    }
```

```
}
```

Concrete
State

Symbolic
State

- Constraint: $2*\beta = \alpha$
- Solution: $\alpha=2, \beta=1$

$x=22, y=7,$
 $z=14$

$x=\alpha, y=\beta, z=2*\beta$
 $2*\beta \neq \alpha$

1st iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

←

```
    z := double (y);
```

```
    if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else {    }
```

```
    }
```

```
}
```

Concrete
State

$x=2, y=1$

Symbolic
State

$x=\alpha, y=\beta$
true

2nd iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    z := double (y);
```

```
    ← if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else {    }
```

```
    }
```

```
}
```

Concrete
State

x=2, y=1,
z=2

Symbolic
State

x=α, y=β, z=2*β
true

2nd iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    z := double (y);
```

```
    if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else {    }
```

```
    }
```

```
}
```

Concrete
State

$x=2, y=1,$
 $z=2$

Symbolic
State

$x=\alpha, y=\beta, z=2*\beta$
 $2*\beta = \alpha$

2nd iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    z := double (y);
```

```
    if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else {    }
```

```
    }
```

```
}
```

Concrete
State

Symbolic
State

x=2, y=1,
z=2

$x=\alpha, y=\beta, z=2*\beta$

$2*\beta = \alpha \wedge$

$\alpha \leq \beta+10$

2nd iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
    z := double (y);  
    if (z==x) {  
        if (x>y+10) {  
            Crash  
        } else {  
            }  
        }  
    }  
}
```

Concrete
State

Symbolic
State

- Constraint: $2*\beta = \alpha \wedge \alpha > \beta+10$
- Solution: $\alpha=30, \beta=15$

$x=2, y=1,$
 $z=2$

$x=\alpha, y=\beta, z=2*\beta$
 $2*\beta = \alpha \wedge$
 $\alpha \leq \beta+10$

2nd iteration

Concolic Testing

	Concrete State	Symbolic State
<pre>int double (int v) { return 2*v; }</pre>		
<pre>void testme(int x, int y) { ← z := double (y); if (z==x) { if (x>y+10) { Crash } else { } } }</pre>	x=30, y=15	x=α, y=β true
3rd iteration		

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    z := double (y);
```

```
    ← if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else {    }
```

```
    }
```

```
}
```

Concrete
State

x=30, y=15,
z=30

Symbolic
State

x=α, y=β, z=2*β
true

3rd iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    z := double (y);
```

```
    if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else {    }
```

```
    }
```

```
}
```

Concrete
State

x=30, y=15,
z=30

Symbolic
State

x=α, y=β, z=2*β
2*β = α

3rd iteration

Concolic Testing

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

```
    z := double (y);
```

```
    if (z==x) {
```

```
        if (x>y+10) {
```

Crash

```
        } else { }
```

```
    }
```

```
}
```

Concrete
State

Symbolic
State

crashing input

x=30, y=15,
z=30

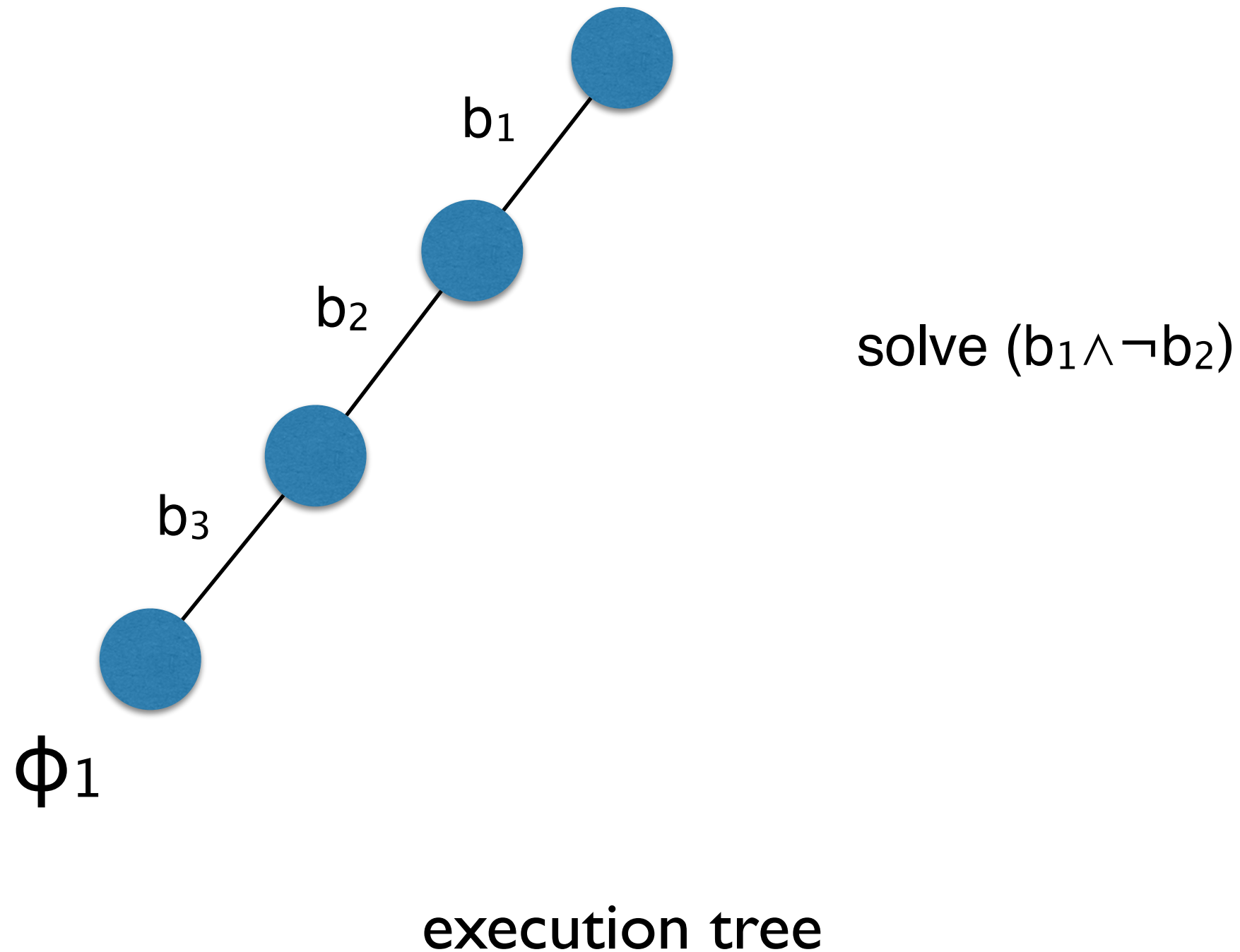
$x=\alpha, y=\beta, z=2*\beta$

$2*\beta = \alpha \wedge$

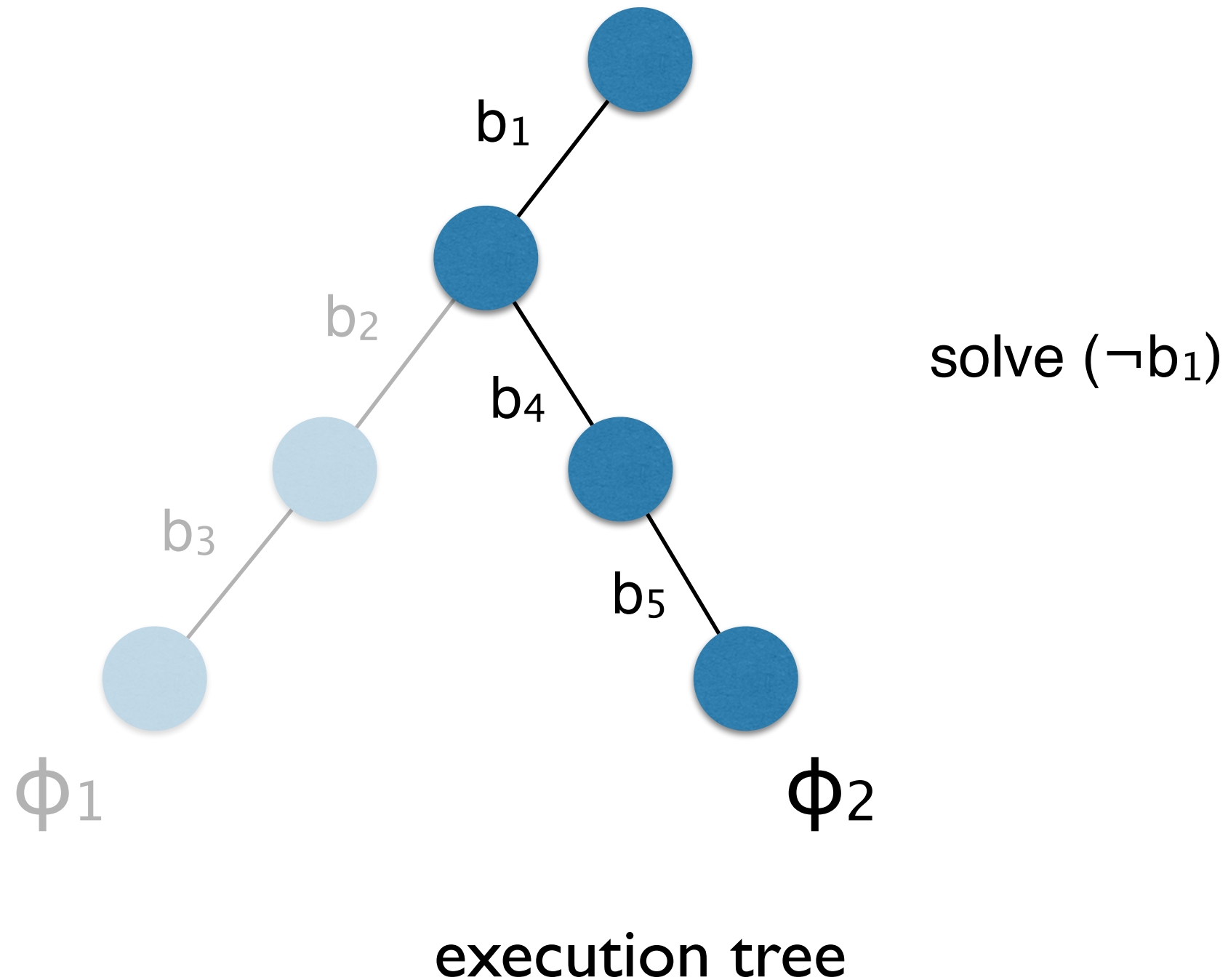
$\alpha > \beta+10$

3rd iteration

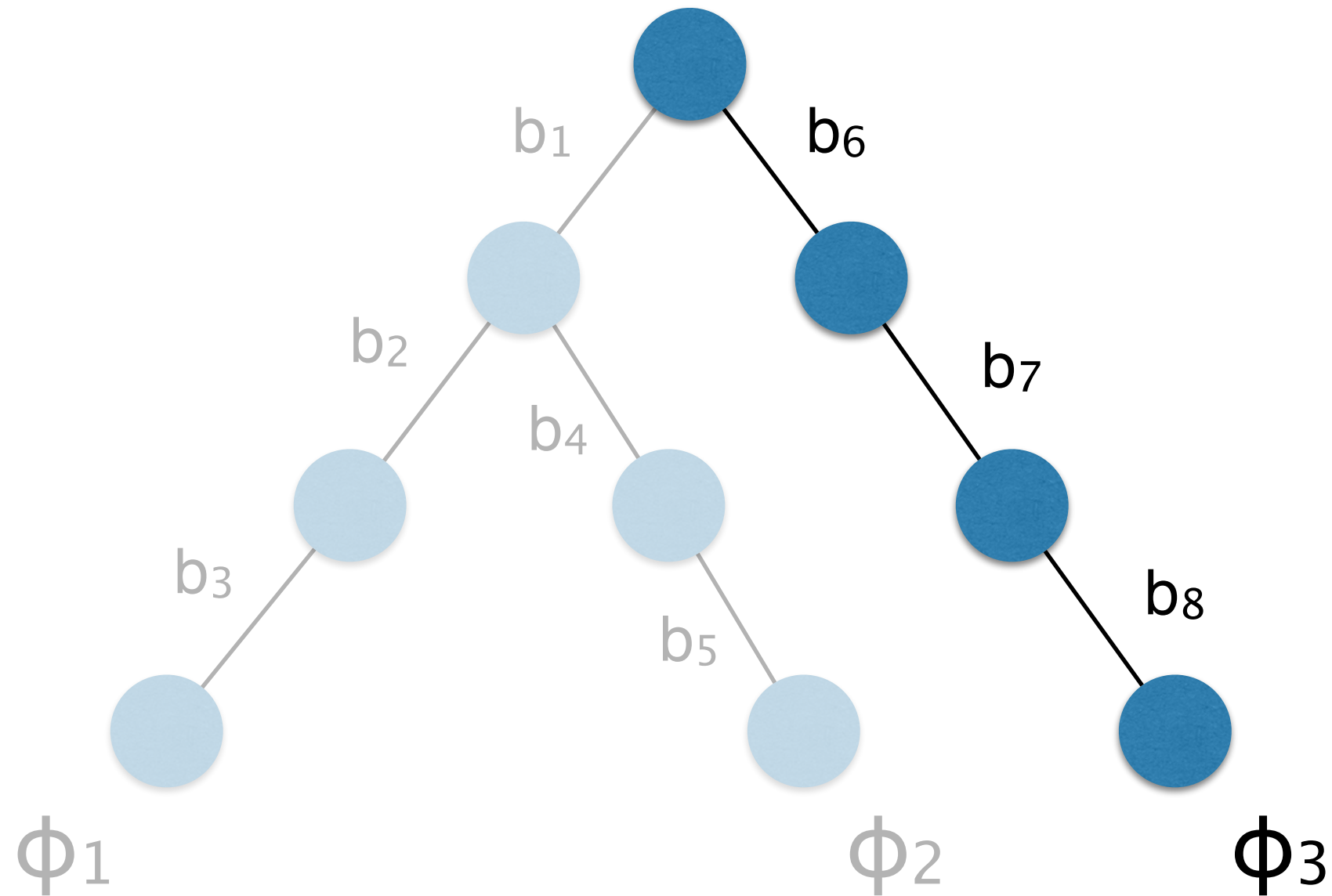
Concolic Testing Algorithm



Concolic Testing Algorithm



Concolic Testing Algorithm



execution tree

Concolic Testing Algorithm

Input : Program P , initial input vector v_0 , budget N

Output: The number of branches covered

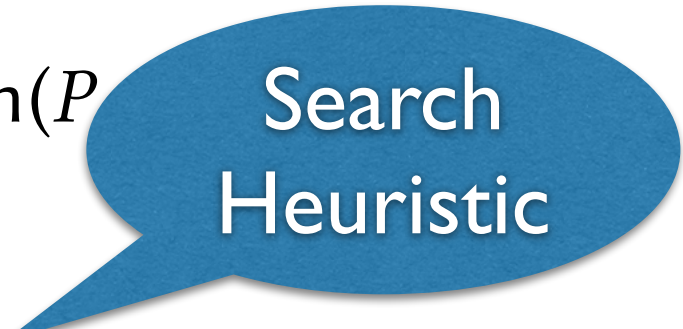
```
1:  $T \leftarrow \langle \rangle$ 
2:  $v \leftarrow v_0$ 
3: for  $m = 1$  to  $N$  do
4:    $\Phi_m \leftarrow \text{RunProgram}(P, v)$ 
5:    $T \leftarrow T \cdot \Phi_m$ 
6:   repeat
7:      $(\Phi, \phi_i) \leftarrow \text{Choose}(T)$        $(\Phi = \phi_1 \wedge \dots \wedge \phi_n)$ 
8:   until  $\text{SAT}(\bigwedge_{j < i} \phi_j \wedge \neg \phi_i)$ 
9:    $v \leftarrow \text{model}(\bigwedge_{j < i} \phi_j \wedge \neg \phi_i)$ 
10: end for
11: return  $|\text{Branches}(T)|$ 
```

Concolic Testing Algorithm

Input : Program P , initial input vector v_0 , budget N

Output: The number of branches covered

```
1:  $T \leftarrow \langle \rangle$ 
2:  $v \leftarrow v_0$ 
3: for  $m = 1$  to  $N$  do
4:    $\Phi_m \leftarrow \text{RunProgram}(P, v)$ 
5:    $T \leftarrow T \cdot \Phi_m$ 
6:   repeat
7:      $(\Phi, \phi_i) \leftarrow \text{Choose}(T)$        $(\Phi = \phi_1 \wedge \dots \wedge \phi_n)$ 
8:   until  $\text{SAT}(\bigwedge_{j < i} \phi_j \wedge \neg \phi_i)$ 
9:    $v \leftarrow \text{model}(\bigwedge_{j < i} \phi_j \wedge \neg \phi_i)$ 
10: end for
11: return  $|\text{Branches}(T)|$ 
```



Advantage of Concolic Testing

	Concrete State	Symbolic State
<pre>int foo (int v) { return hash(v); }</pre>		
<pre>void testme(int x, int y) { ← z := foo (y); if (z==x) { if (x>y+10) { Crash } else { } } }</pre>	$x=22, y=7$	$x=\alpha, y=\beta$ true
1st iteration		

Advantage of Concolic Testing

	Concrete State	Symbolic State
<pre>int foo (int v) { return hash(v); }</pre>		
<pre>void testme(int x, int y) { z := foo (y); ← if (z==x) { if (x>y+10) { Crash } else { } } }</pre>	x=22, y=7, z=601...129	x=α, y=β, z=hash(β) true
1st iteration		

Advantage of Concolic Testing

	Concrete State	Symbolic State
<pre>int foo (int v) { return hash(v); } void testme(int x, int y) { z := foo (y); if (z==x) { if (x>y+10) { Crash } else { } } }</pre>	$x=22, y=7,$ $z=601\dots129$	$x=\alpha, y=\beta,$ $z=\text{hash}(\beta)$ $\text{hash}(\beta) \neq \alpha$
	1st iteration	

Advantage of Concolic Testing

```
int foo (int v) {  
    return hash(v);  
}
```

```
void testme(int x, int y) {  
    z := foo (y);  
    if (z==x) {  
        if (x>y+10) {  
            Crash  
        } else {  
        }  
    }  
}
```

Concrete
State

Symbolic
State

- Constraint: $\text{hash}(\beta) = \alpha$
- Replace β by 7: $601\dots129 = \alpha$
- Solution: $\alpha=601\dots129, \beta=7$

$x=22, y=7,$
 $z=601\dots129$

$x=\alpha, y=\beta,$
 $z=\text{hash}(\beta)$
 $\text{hash}(\beta) \neq \alpha$

1st iteration

Advantage of Concolic Testing

	Concrete State	Symbolic State
<pre>int foo (int v) { return hash(v); }</pre>		
<pre>void testme(int x, int y) { ← z := foo (y); if (z==x) { if (x>y+10) { Crash } else { } } }</pre>	$x=601\dots129$ $y=7$	$x=\alpha, y=\beta$ true
2nd iteration		

Advantage of Concolic Testing

	Concrete State	Symbolic State
<pre>int foo (int v) { return hash(v); }</pre>		
<pre>void testme(int x, int y) { z := foo (y); ← if (z==x) { if (x>y+10) { Crash } else { } } }</pre>	<pre>x=601...129 y=7 z=601...129</pre>	<pre>x=α, y=β, z=hash(β) true</pre>
2nd iteration		

Advantage of Concolic Testing

	Concrete State	Symbolic State
<pre>int foo (int v) { return hash(v); }</pre>		
<pre>void testme(int x, int y) { z := foo (y); if (z==x) { ← if (x>y+10) { Crash } else { } } }</pre>	$x=601\dots129$ $y=7$ $z=601\dots129$	$x=\alpha, y=\beta,$ $z=\text{hash}(\beta)$ $\text{hash}(\beta) = \alpha$
2nd iteration		

Advantage of Concolic Testing

	Concrete State	Symbolic State
<pre>int foo (int v) { return hash(v); } void testme(int x, int y) { z := foo (y); if (z==x) { if (x>y+10) { Crash } else { } } }</pre>	$x=601\dots129$ $y=7$ $z=601\dots129$	$x=\alpha, y=\beta,$ $z=\text{hash}(\beta)$ $\text{hash}(\beta) = \alpha \wedge$ $\alpha > \beta+10$
2nd iteration		

Limitation of Concolic Testing

	Concrete State	Symbolic State
<pre>int foo (int v) { return secure_hash(v); }</pre>		
<pre>void testme(int x, int y) { ← if (x != y) { if (foo(x) == foo(y)) { Crash } } }</pre>	$x=22, y=7$	$x=\alpha, y=\beta$ true
1st iteration		

Limitation of Concolic Testing

Concrete
State

Symbolic
State

```
int foo (int v) {  
    return secure_hash(v);  
}
```

```
void testme(int x, int y) {
```

```
    if (x != y) {
```



```
        if (foo(x) == foo(y)) {
```

Crash

```
        }
```

```
    }
```

```
}
```

$x=22, y=7$

$x=a, y=\beta$
 $a \neq \beta$

1st iteration

Limitation of Concolic Testing

Concrete
State

Symbolic
State

```
int foo (int v) {  
    return secure_hash(v);  
}
```

```
void testme(int x, int y) {
```

```
    if (x != y) {
```

```
        if (foo(x) == foo(y)) {
```

Crash

```
        }
```

```
    }
```

```
}
```

$x=22, y=7$

$x=\alpha, y=\beta$

$\alpha \neq \beta \wedge$

$\text{hash}(\alpha) \neq \text{hash}(\beta)$

1st iteration

Limitation of Concolic Testing

Concrete
State

Symbolic
State

```
int foo (int v) {  
    return secure_hash(v);  
}
```

```
void testme(  
    if (x != y)
```

```
    if (foo(x) == foo(y)) {  
        Crash  
    }  
}
```

```
}
```

$x=22, y=7$

$x=\alpha, y=\beta$

$\alpha \neq \beta \wedge$

$\text{hash}(\alpha) \neq \text{hash}(\beta)$

- Constraint: $\alpha \neq \beta \wedge \text{hash}(\alpha) = \text{hash}(\beta)$

- Replace α, β by 22,7: $22 \neq 7 \wedge 438\dots861 = 601\dots129$

- Unsatisfiable!

1st iteration

Limitation of Concolic Testing

Concrete
State

Symbolic
State

```
int foo (int v) {
    return secure_hash(v);
}
```

```
void testme()
```

```
if (x != y)
```

```
    if (foo(x) == foo(y)) {
```

Crash

false negative

```
    }
```

```
}
```

```
}
```

$x=22, y=7$

$x=\alpha, y=\beta$

$\alpha \neq \beta \wedge$

$\text{hash}(\alpha) \neq \text{hash}(\beta)$

- Constraint: $\alpha \neq \beta \wedge \text{hash}(\alpha) = \text{hash}(\beta)$

- Replace α, β by 22,7: $22 \neq 7 \wedge 438\dots861 = 601\dots129$

- Unsatisfiable!

1st iteration

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { ←────────────────────────────────── int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	x=1	x=a true
1st iteration		

Testing Loops

```
void testme(int x) {
```

```
int A[] = { 5, 7, 9 };
```



```
int i = 0;
```

```
while (i < 3) {
    if (A[i] == x) break;
    i++;
}
```

```
return i;
```

}

Concrete State

$$x=1$$
$$A = \{5, 7, 9\}$$

Symbolic State

$$x = d$$

true

1st iteration

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
    ←  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

x=1, i=0,
A = {5,7,9}

Symbolic
State

x=a

true

1st iteration

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=1, i=0,$ $A = \{5,7,9\}$ 	$x=a$ $true$
1st iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=1, i=0,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a$
1st iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>		
		
	$x=1, i=1,$ $A = \{5,7,9\}$	$x=a$ $5 \neq a$
	1st iteration	

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=1, i=1,$ $A = \{5,7,9\}$ 1st iteration	$x=a$ $5 \neq a$

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=1, i=1,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a \wedge 7 \neq a$
	1st iteration	

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

Symbolic
State

$x=1, i=2,$
 $A = \{5, 7, 9\}$

$x=a$

$5 \neq a \wedge 7 \neq a$

1st iteration

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=1, i=2,$ $A = \{5,7,9\}$ 1st iteration	$x=a$ $5 \neq a \wedge 7 \neq a$

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=1, i=2,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a \wedge 7 \neq a \wedge$ $9 \neq a$
	1st iteration	

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

$x=1, i=3,$
 $A = \{5,7,9\}$

Symbolic
State

$x=a$

$5 \neq a \wedge 7 \neq a \wedge$
 $9 \neq a$

1st iteration

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

Symbolic
State

$x=1, i=3,$
 $A = \{5,7,9\}$

$x=a$

$5 \neq a \wedge 7 \neq a \wedge$

$9 \neq a$

1st iteration

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

Symbolic
State

- Constraint: $5 \neq a \wedge 7 \neq a \wedge 9 = a$
- Solution: $a=9$

$x=1, i=3,$
 $A = \{5,7,9\}$

$x=a$

$5 \neq a \wedge 7 \neq a \wedge$
 $9 \neq a$

1st iteration

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { ←────────────────────────────────── int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	x=9	x=a true
2nd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; ← int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=9,$ $A = \{5,7,9\}$	$x=a$ true
2nd iteration		

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
    ←  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

x=9, i=0,
A = {5,7,9}

Symbolic
State

x=a

true

2nd iteration

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=9, i=0,$ $A = \{5,7,9\}$ ←	$x=a$ $true$
2nd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=9, i=0,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a$
2nd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>		
		
	$x=9, i=1,$ $A = \{5,7,9\}$	$x=a$ $5 \neq a$
	2nd iteration	

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=9, i=1,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a$
2nd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=9, i=1,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a \wedge 7 \neq a$
2nd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=9, i=2,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a \wedge 7 \neq a$
2nd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=9, i=2,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a \wedge 7 \neq a$
2nd iteration		

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

Symbolic
State

$x=9, i=2,$
 $A = \{5,7,9\}$

$x=a$
 $5 \neq a \wedge 7 \neq a \wedge$
 $9=a$



2nd iteration

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

Symbolic
State

- Constraint: $5 \neq a \wedge 7 = a$
- Solution: $a=7$

$x=9, i=2,$
 $A = \{5,7,9\}$

$x=a$
 $5 \neq a \wedge 7 \neq a \wedge$
 $9=a$

2nd iteration

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { ←────────────────────────────────── int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=7$	$x=a$ true
3rd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; ← int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=7,$ $A = \{5,7,9\}$	$x=a$ true
3rd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; ← while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=7, i=0,$ $A = \{5,7,9\}$	$x=a$ true
3rd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=7, i=0,$ $A = \{5,7,9\}$ 3rd iteration	$x=a$ $true$

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=7, i=0,$ $A = \{5,7,9\}$ ←	$x=a$ $5 \neq a$
3rd iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	← $x=7, i=1,$ $A = \{5,7,9\}$ 3rd iteration	$x=a$ $5 \neq a$

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=7, i=1,$ $A = \{5,7,9\}$ 3rd iteration	$x=a$ $5 \neq a$

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

Symbolic
State

$x=7, i=1,$
 $A = \{5,7,9\}$

$x=a$

$5 \neq a \wedge 7 = a$

3rd iteration

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

Symbolic
State

- Constraint: $5 \neq a$
- Solution: $a=5$

$x=7, i=1,$
 $A = \{5,7,9\}$

$x=a$

$5 \neq a \wedge 7 = a$

3rd iteration

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { ←────────────────────────────────── int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	x=5	x=a true
4th iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; ← int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=5,$ $A = \{5,7,9\}$	$x=a$ true
4th iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; ← while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	$x=5, i=0,$ $A = \{5,7,9\}$	$x=a$ true
4th iteration		

Testing Loops

	Concrete State	Symbolic State
<pre>void testme(int x) { int A[] = { 5, 7, 9 }; int i = 0; while (i < 3) { if (A[i] == x) break; i++; } return i; }</pre>	x=5, i=0, A = {5,7,9} ←	x=a true
4th iteration		

Testing Loops

```
void testme(int x) {  
    int A[] = { 5, 7, 9 };  
  
    int i = 0;  
  
    while (i < 3) {  
        if (A[i] == x) break;  
        i++;  
    }  
  
    return i;  
}
```

Concrete
State

Symbolic
State

x=7, i=0,
A = {5,7,9}

x=a
5=a

4th iteration

Testing Data Structures

Concrete
State

Symbolic
State

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0) ←  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;  
}
```

x=236
p=NULL

x=α, p=β
true

1st iteration

Testing Data Structures

Concrete
State

Symbolic
State

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0)  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;  
}
```

x=236
p=NULL

x=a, p= β
a > 0

1st iteration

Testing Data Structures

Concrete
State

Symbolic
State

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0)  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;   
}
```

x=236
p=NULL

$x = \alpha, p = \beta$
 $\alpha > 0 \wedge$
 $\beta = \text{NULL}$

1st iteration

Testing Data Structures

	Concrete State	Symbolic State
<pre>typedef struct cell { int data; struct cell *next; } cell; int foo(int v) { return 2*v; } void testme(int x, cell *p) { if (x > 0) if (p != NULL) if (foo(x) == p->data) if (p->next == p) Crash return 0; }</pre>	- Constraint: $\alpha > 0 \wedge \beta \neq \text{NULL}$ - Solution: $\alpha = 236, \beta =$ 634 NULL $x=236$ $p=\text{NULL}$	$x=\alpha, p=\beta$ $\alpha > 0 \wedge$ $\beta = \text{NULL}$
	1st iteration	

Testing Data Structures

Concrete
State

Symbolic
State

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0)  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;  
}
```

x=236
p=

634	NULL
-----	------

$x = \alpha$, $p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
true

2nd iteration

Testing Data Structures

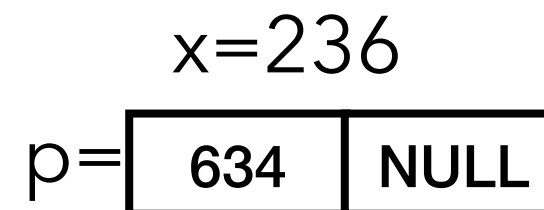
Concrete
State

Symbolic
State

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0)  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;  
}
```



$x = \alpha$, $p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
 $\alpha > 0$

2nd iteration

Testing Data Structures

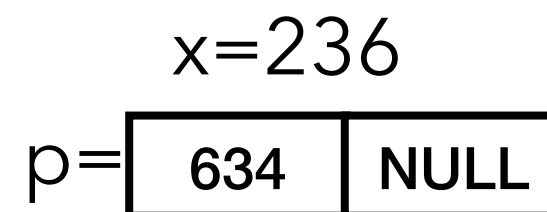
Concrete
State

Symbolic
State

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0)  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;  
}
```



$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
 $\alpha > 0 \wedge$
 $\beta \neq \text{NULL}$

2nd iteration

Testing Data Structures

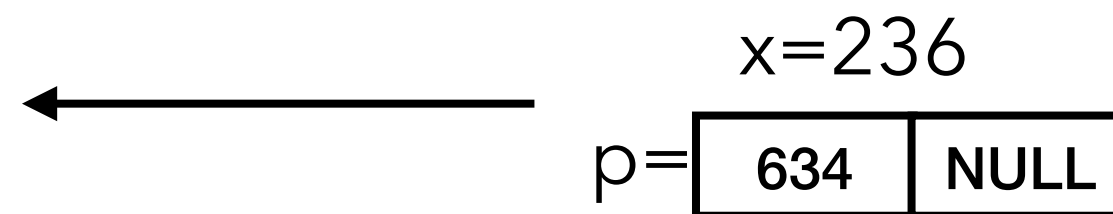
Concrete
State

Symbolic
State

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    Crash
    return 0;
}
```



2nd iteration

$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
 $\alpha > 0 \wedge$
 $\beta \neq \text{NULL} \wedge$
 $2 * \alpha + 1 \neq \gamma$

Testing Data Structures

Concrete
State

Symbolic
State

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return
```

- Constraint: $\alpha > 0 \wedge \beta \neq \text{NULL} \wedge 2 * \alpha + 1 = \gamma$
 - Solution: $\alpha = 1, \beta =$

3

NULL

```
void testme(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    Crash
    return 0;
}
```



x=236
 p=

634	NULL
-----	------

2nd iteration

$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
 $\alpha > 0 \wedge$
 $\beta \neq \text{NULL} \wedge$
 $2 * \alpha + 1 \neq \gamma$

Testing Data Structures

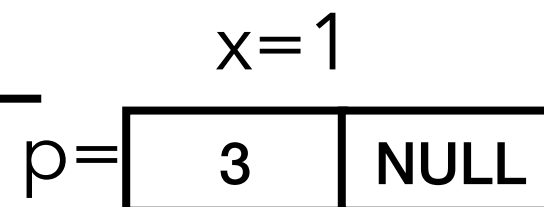
Concrete
State

Symbolic
State

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0)  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;  
}
```



$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
true

3rd iteration

Testing Data Structures

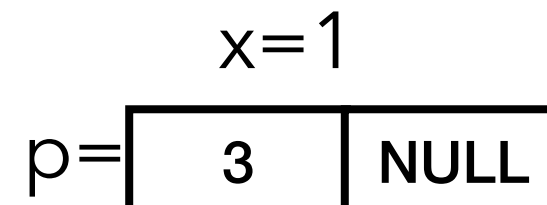
Concrete
State

Symbolic
State

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    Crash
    return 0;
}
```



$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
 $\alpha > 0 \wedge$
 $\beta \neq \text{NULL} \wedge$
 $2 * \alpha + 1 = \gamma$

3rd iteration

Testing Data Structures

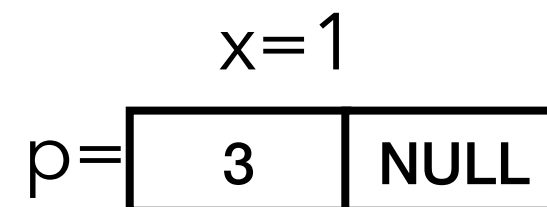
Concrete
State

Symbolic
State

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0)  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;  
}
```



3rd iteration

$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
 $\alpha > 0 \wedge$
 $\beta \neq \text{NULL} \wedge$
 $2 * \alpha + 1 = \gamma \wedge$
 $\delta \neq \beta$

Testing Data Structures

Concrete
State

Symbolic
State

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;
```

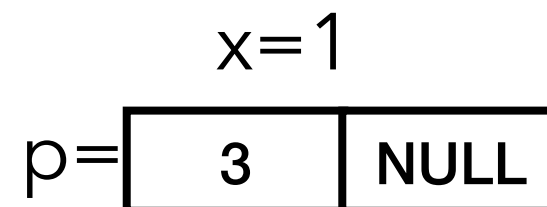
```
int foo(int v) { re
```

- Constraint: $\alpha > 0 \wedge \beta \neq \text{NULL} \wedge 2 * \alpha + 1 = \gamma \wedge \delta = \beta$

- Solution: $\alpha = 1, \beta =$



```
void testme(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    Crash
    return 0;
}
```



3rd iteration

$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
 $\alpha > 0 \wedge$
 $\beta \neq \text{NULL} \wedge$
 $2 * \alpha + 1 = \gamma \wedge$
 $\delta \neq \beta$

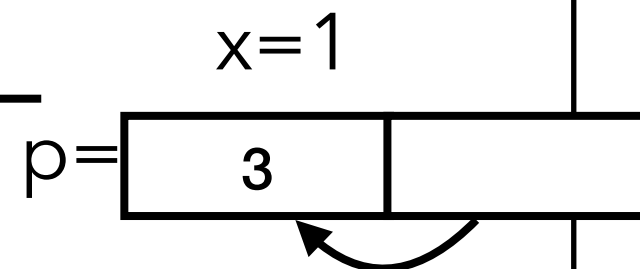
Testing Data Structures

```
typedef struct cell {  
    int data;  
    struct cell *next;  
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {  
    if (x > 0)  
        if (p != NULL)  
            if (foo(x) == p->data)  
                if (p->next == p)  
                    Crash  
    return 0;  
}
```

Concrete
State



Symbolic
State

$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
true

4th iteration

Testing Data Structures

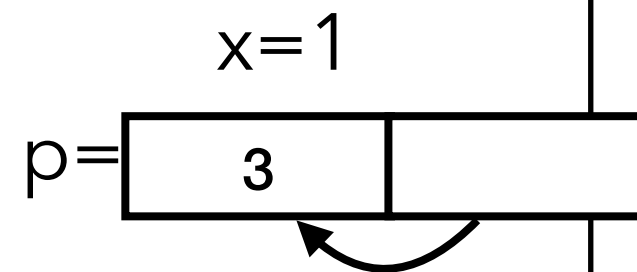
Concrete
State

Symbolic
State

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;
```

```
int foo(int v) { return 2*v + 1; }
```

```
void testme(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    Crash
    return 0;
}
```



$x = \alpha, p = \beta$
 $p \rightarrow \text{data} = \gamma$
 $p \rightarrow \text{next} = \delta$
 $\alpha > 0 \wedge$
 $\beta \neq \text{NULL} \wedge$
 $2 * \alpha + 1 = \gamma \wedge$
 $\delta = \beta$

4th iteration

Summary: Concolic Testing

- An [automated, white-box](#) approach to test generation
- Concrete and symbolic execution [cooperate](#) w/ each other
 - [Concrete execution guides symbolic execution](#), enabling it to overcome incompleteness of theorem prover
 - [Symbolic execution guides generation of concrete inputs](#), increasing program code coverage
- Further reading:
 - Automatically Generating Search Heuristics for Concolic Testing. ICSE 2018
 - Concolic Testing with Adaptively Changing Search Heuristics. ESEC/FSE 2019
 - SymTuner: Maximizing the Power of Symbolic Execution by Adaptively Tuning External Parameters. ICSE 2022