# COSE419: Software Verification

# Lecture 3 – Concolic Testing*

Hakjoo Oh
2024 Spring

*Most slides based on Mayur Naik's (https://software-analysis-class.org/)

# 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

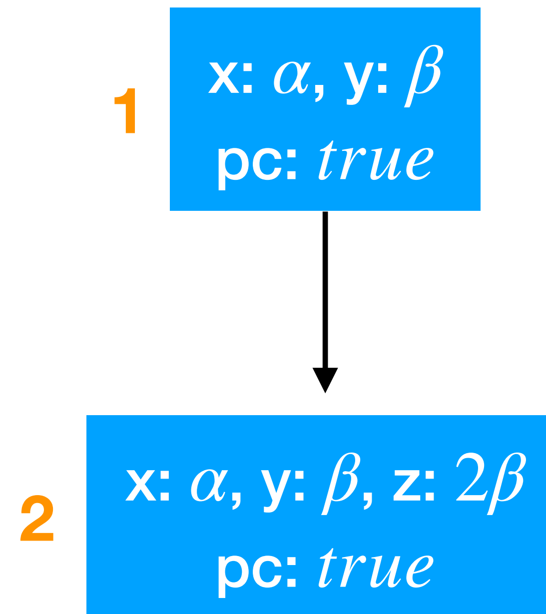
```
int double (int v) {
  return 2*v;
}

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

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

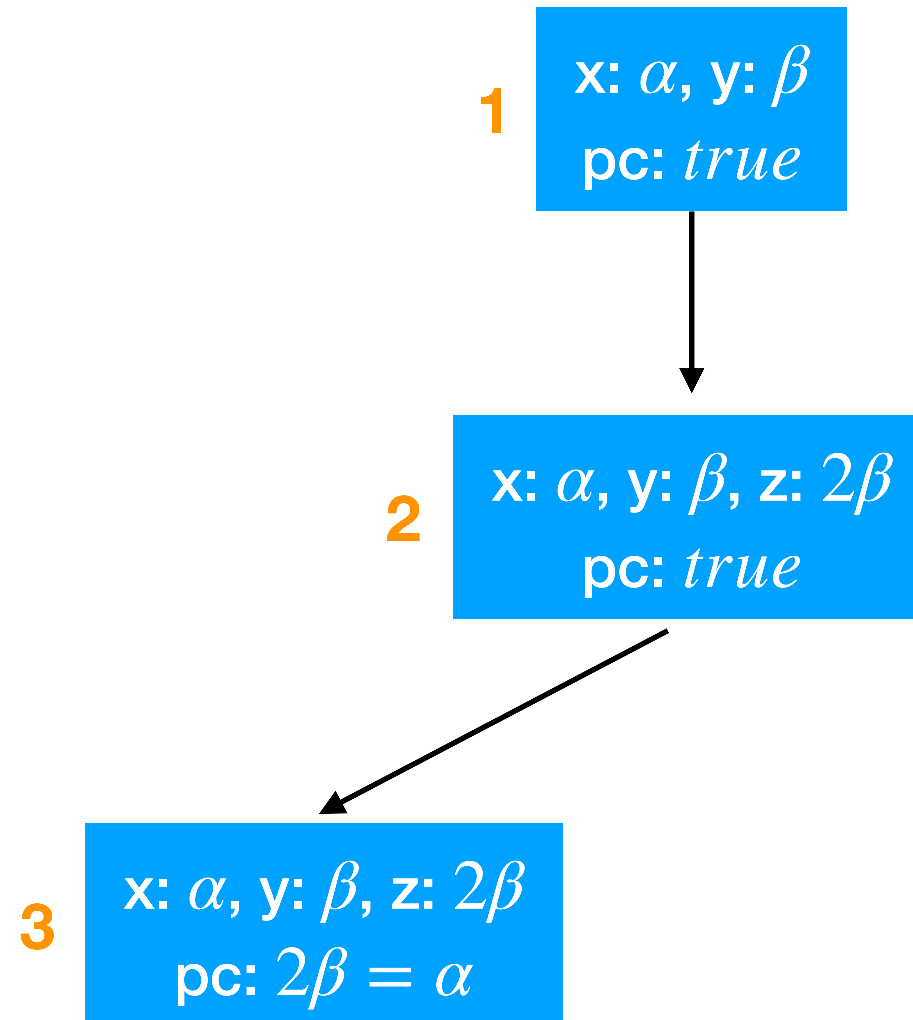
# Symbolic Execution

```
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 }
```



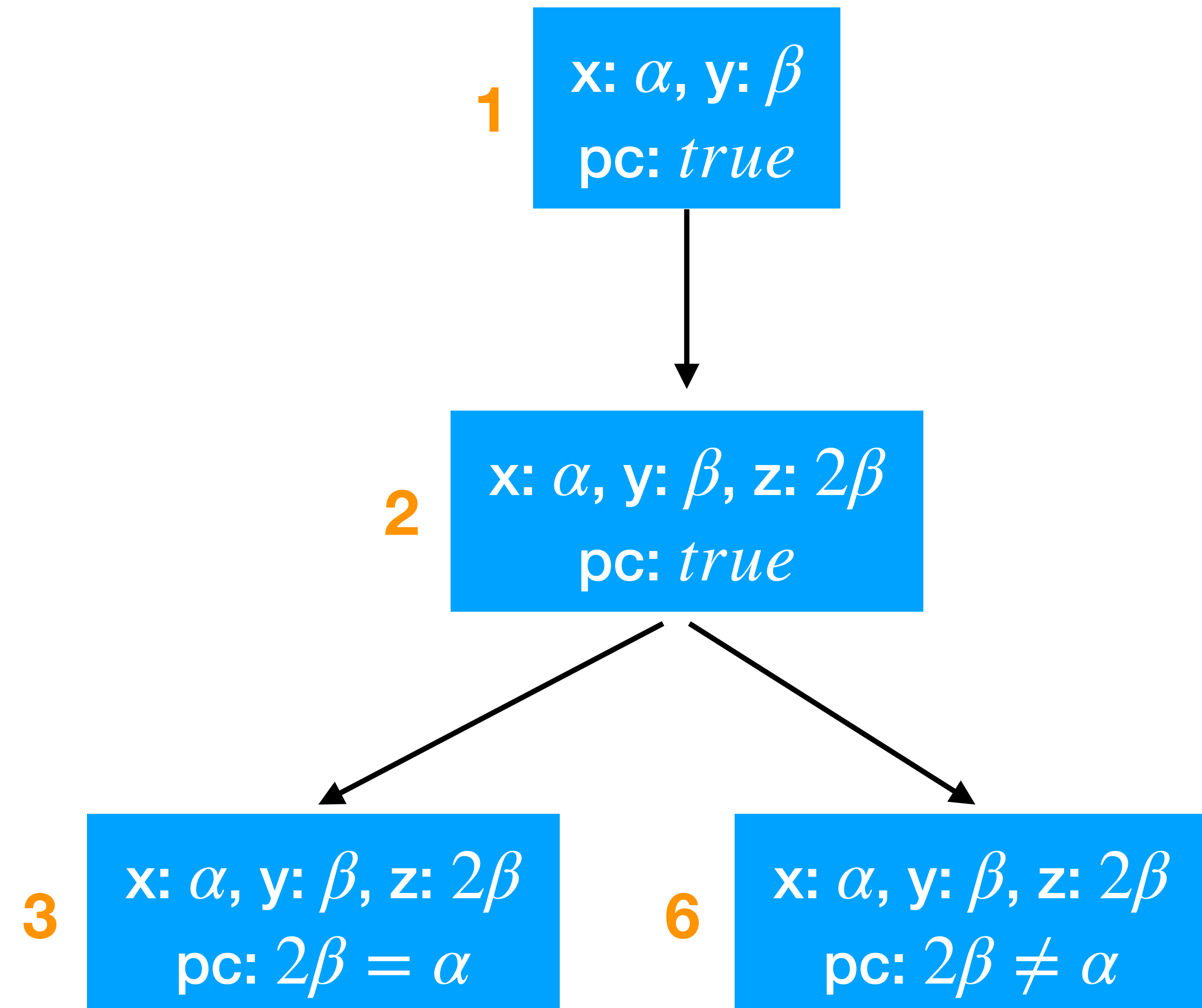
# Symbolic Execution

```
  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 }
```



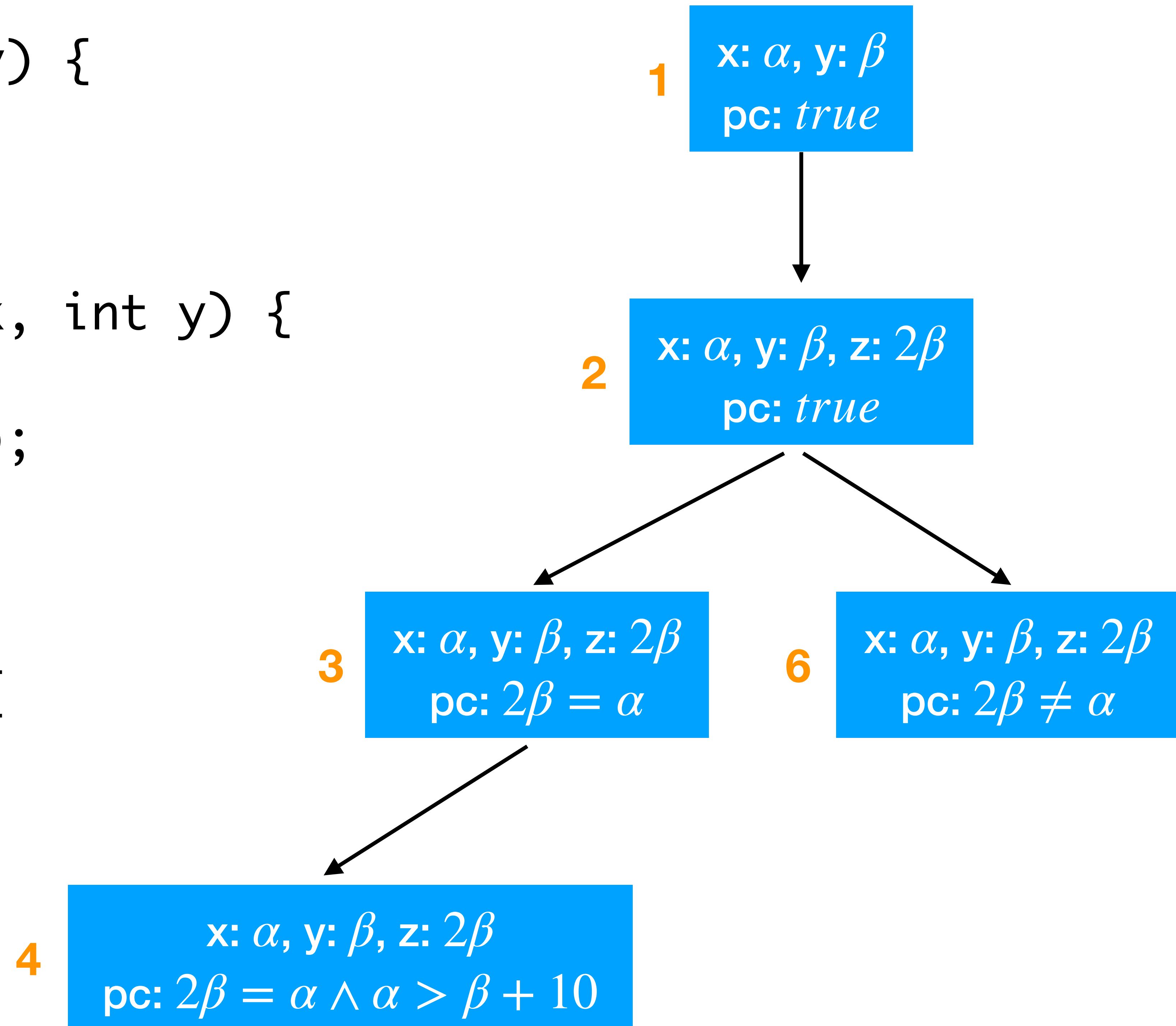
# Symbolic Execution

```
  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 }
```



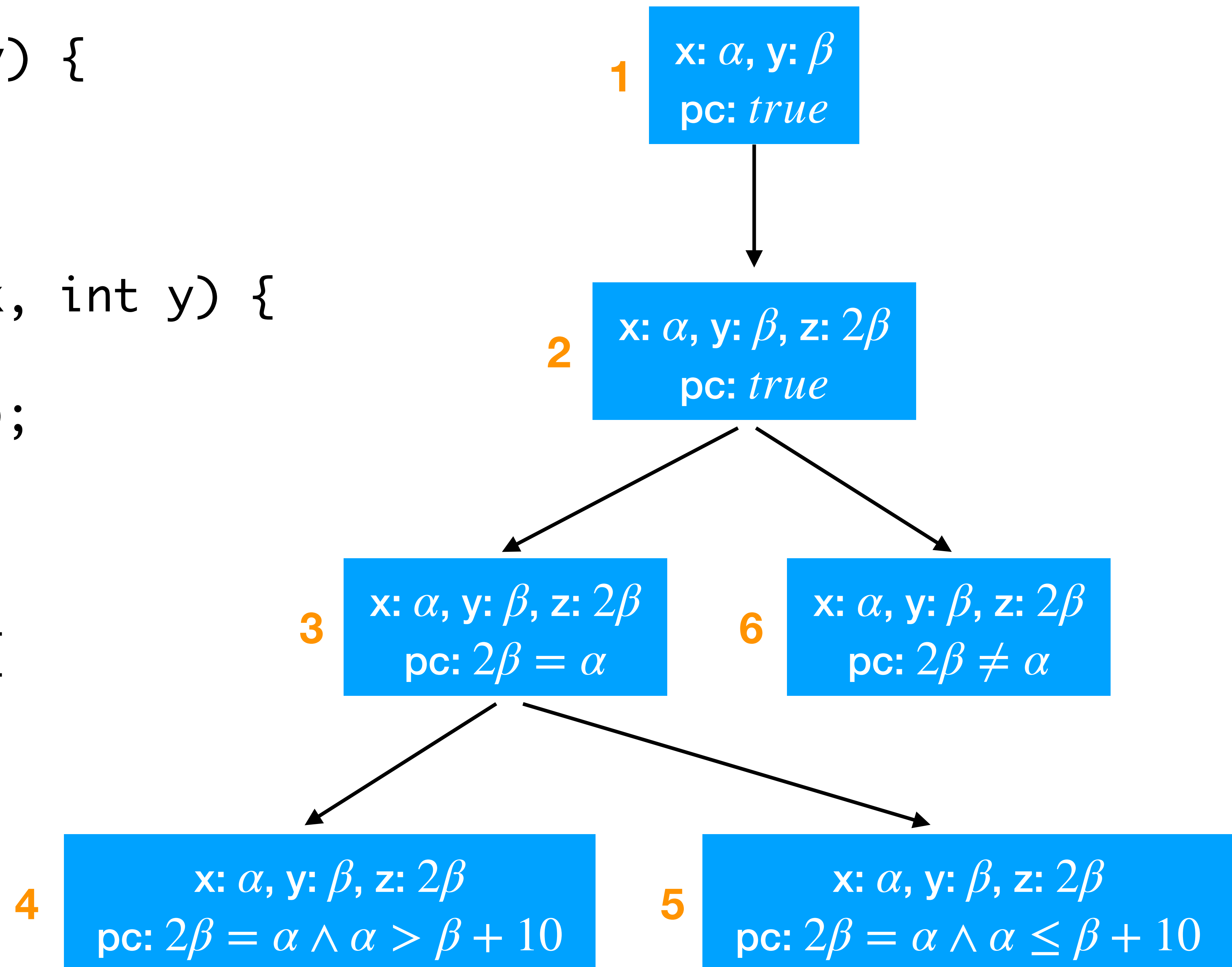
# Symbolic Execution

```
  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 }
```



# 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 }
```



# 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 | Symbolic State |
| --- | --- |
| x=22, y=7 | 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 | Symbolic State |
| --- | --- |
| x=22, y=7, z=14 | 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=α, y=β, z=2*β<br>$2*\beta \neq \alpha$ |

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 | Symbolic State |
|---|---|
| x=22, y=7, z=14 | x=α, y=β, z=2*β |
| | 2*β ≠ α |

- Constraint: 2*β = α
- Solution: α=2,β=1

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=2, y=1 | x=α, y=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 | Symbolic State |
|---|---|
| x=2, y=1, z=2 | $x=\alpha$, $y=\beta$, $z=2*\beta$ |
| | 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 | Symbolic State |
|---|---|
| x=2, y=1, z=2 | x=α, y=β, z=2*β<br>2*β = α |

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=α, y=β, z=2*β |
| | $2*\beta = \alpha \wedge$ $\alpha \leq \beta+10$ |

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 |
|---|---|
| x=2, y=1, z=2 | x=α, y=β, z=2*β |
| | $2*\beta = \alpha \wedge$ $\alpha \leq \beta+10$ |

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

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=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 | Symbolic State |
| --- | --- |
| x=30, y=15, z=30 | x=α, y=β, z=2*β<br>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

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

$2*\beta = \alpha$

3rd iteration

# Concolic Testing

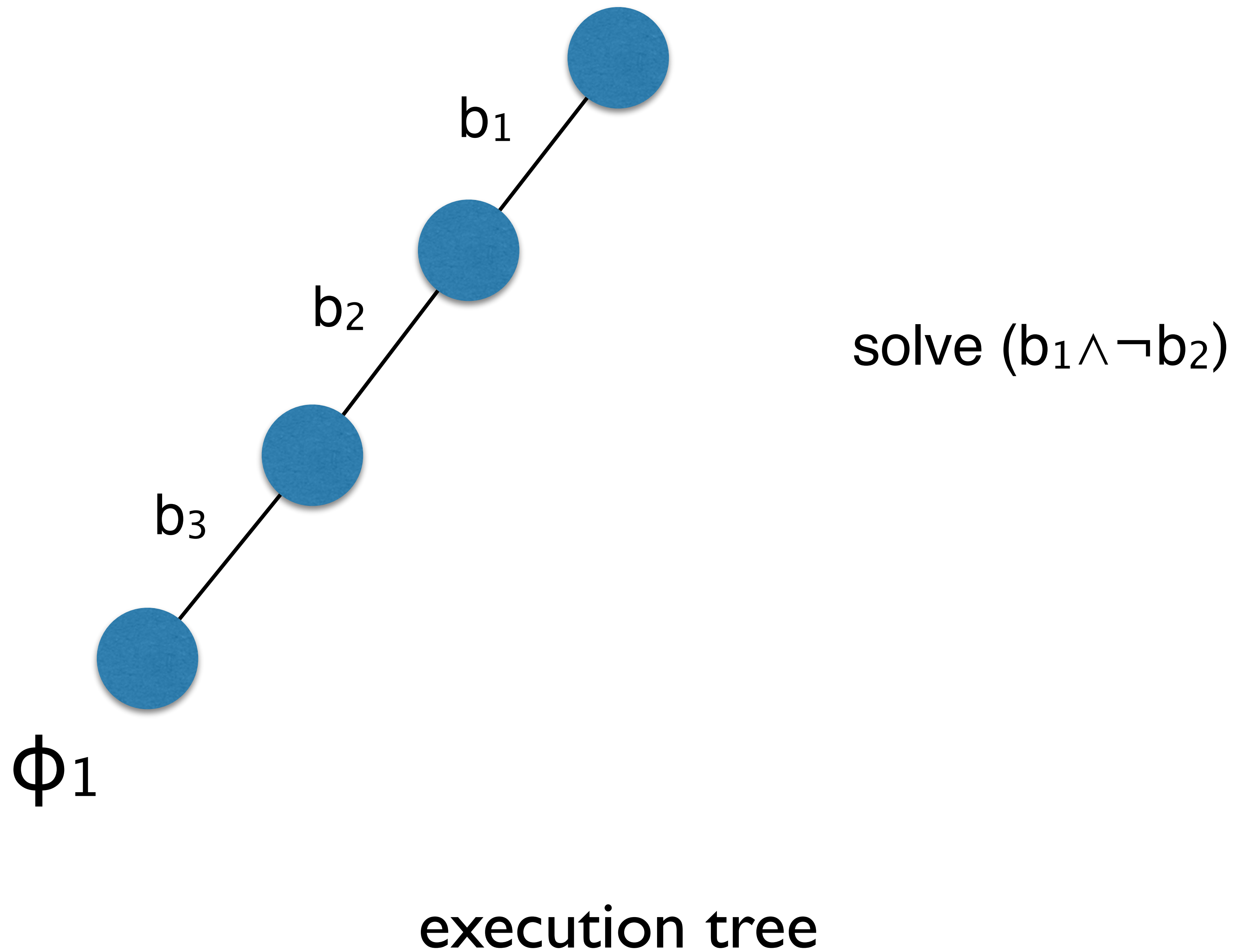
```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

| Concrete State | Symbolic State |
| --- | --- |
| x=30, y=15, z=30 | x=α, y=β, z=2*β |
| | $2*\beta = \alpha \wedge \alpha > \beta+10$ |

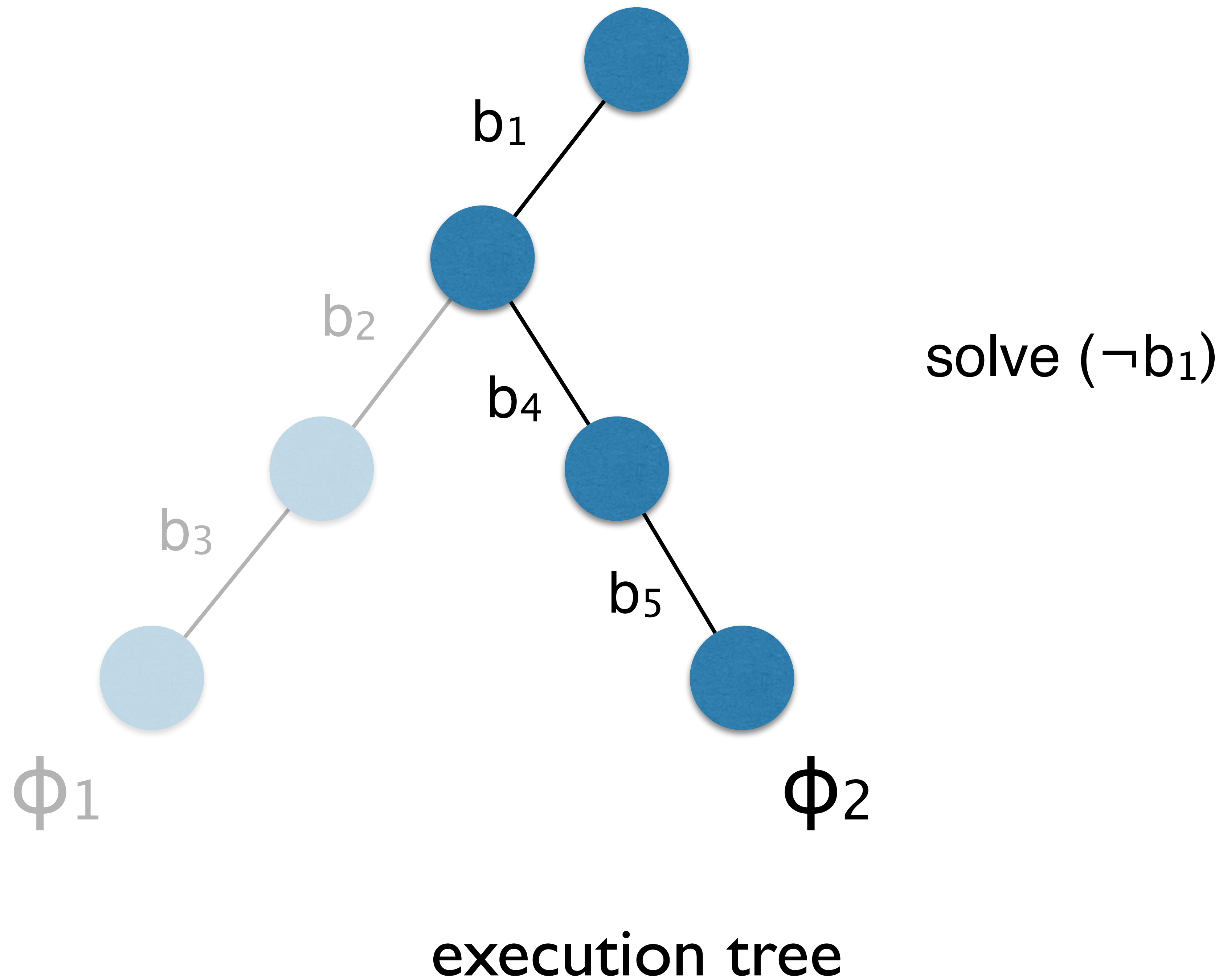
crashing input

3rd iteration

# Concolic Testing Algorithm

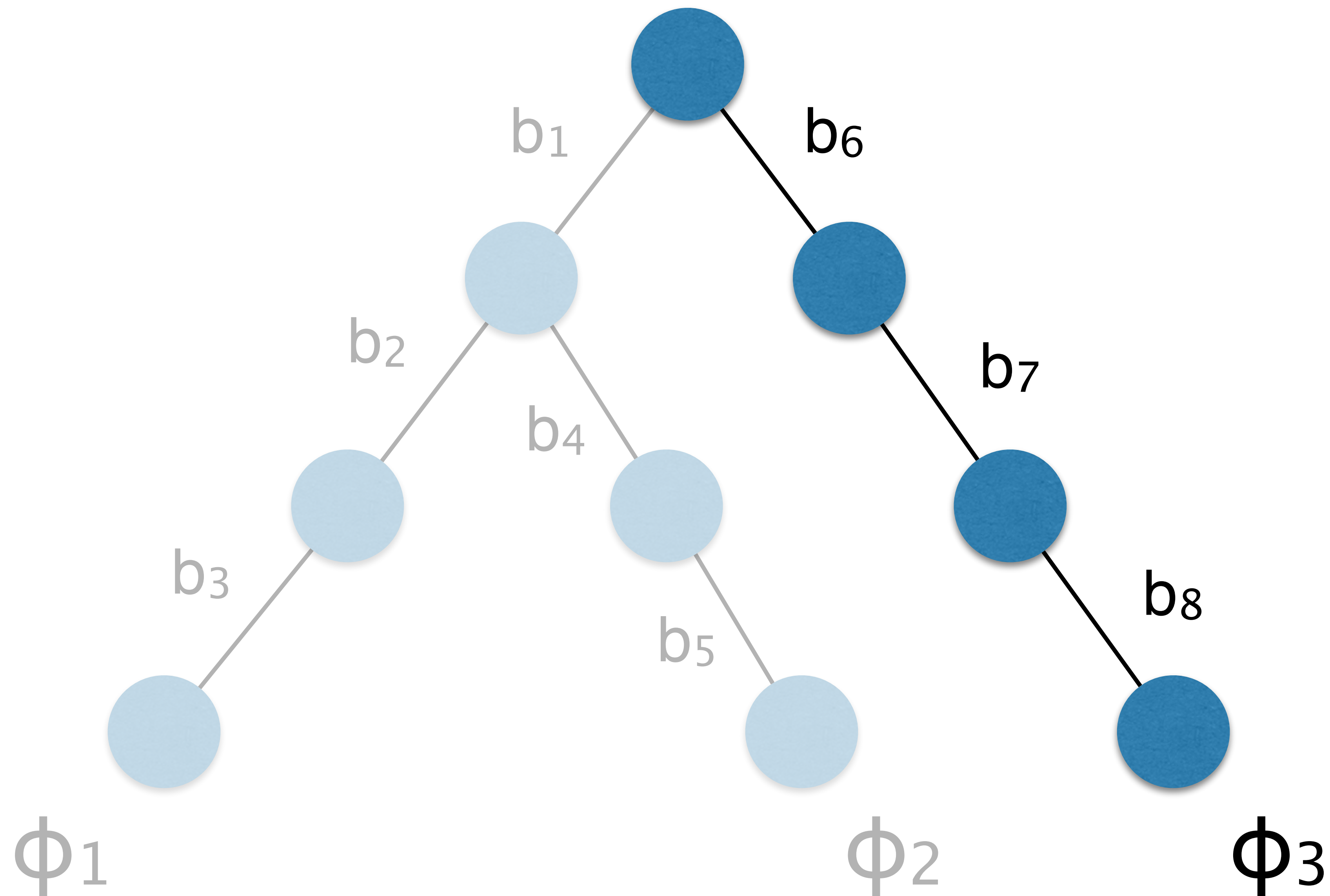


# Concolic Testing Algorithm




solve (¬b1)

# 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: $\quad \Phi_m \leftarrow \text{RunProgram}(P, v)$
5: $\quad T \leftarrow T \cdot \Phi_m$
6: $\quad$ **repeat**
7: $\quad\quad (\Phi, \phi_i) \leftarrow \text{Choose}(T) \qquad (\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$
8: $\quad$ **until** $\text{SAT}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
9: $\quad v \leftarrow \text{model}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
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: $\quad \Phi_m \leftarrow \text{RunProgram}(P$

5: $\quad T \leftarrow T \cdot \Phi_m$
6: $\quad$ **repeat**
7: $\quad\quad (\Phi, \phi_i) \leftarrow \text{Choose}(T) \qquad (\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$
8: $\quad$ **until** $\text{SAT}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
9: $\quad v \leftarrow \text{model}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
10: **end for**
11: **return** $|\text{Branches}(T)|$

# Advantage of Concolic Testing

```
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 |
| --- | --- |
| x=22, y=7 | x=α, y=β |
| | true |

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 |
|---|---|
| x=22, y=7, z=601...129 | x=α, y=β, z=hash(β) true |

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 |
|---|---|
| x=22, y=7, z=601…129 | x=α, y=β, z=hash(β) hash(β) ≠ α |

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 |
|---|---|
| x=22, y=7, z=601…129 | x=α, y=β, z=hash(β) hash(β) ≠ α |

- Constraint: hash(β) = α
- Replace β by 7: 601…129 = α
- Solution: α=601…129, β=7

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 |
| --- | --- |
| x=601...129<br>y=7 | x=α, y=β<br>true |

2nd 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 |
|---|---|
| x=601…129<br>y=7<br>z=601…129 | x=α, y=β,<br>z=hash(β)<br>true |

2nd 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 |
|---|---|
| x=601…129<br>y=7<br>z=601…129 | x=α, y=β,<br>z=hash(β)<br>hash(β) = α |

2nd 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 |
|---|---|
| x=601…129<br>y=7<br>z=601…129 | $x=\alpha, y=\beta,$<br>$z=hash(\beta)$<br>$hash(\beta) = \alpha \wedge$<br>$\alpha > \beta+10$ |

2nd iteration

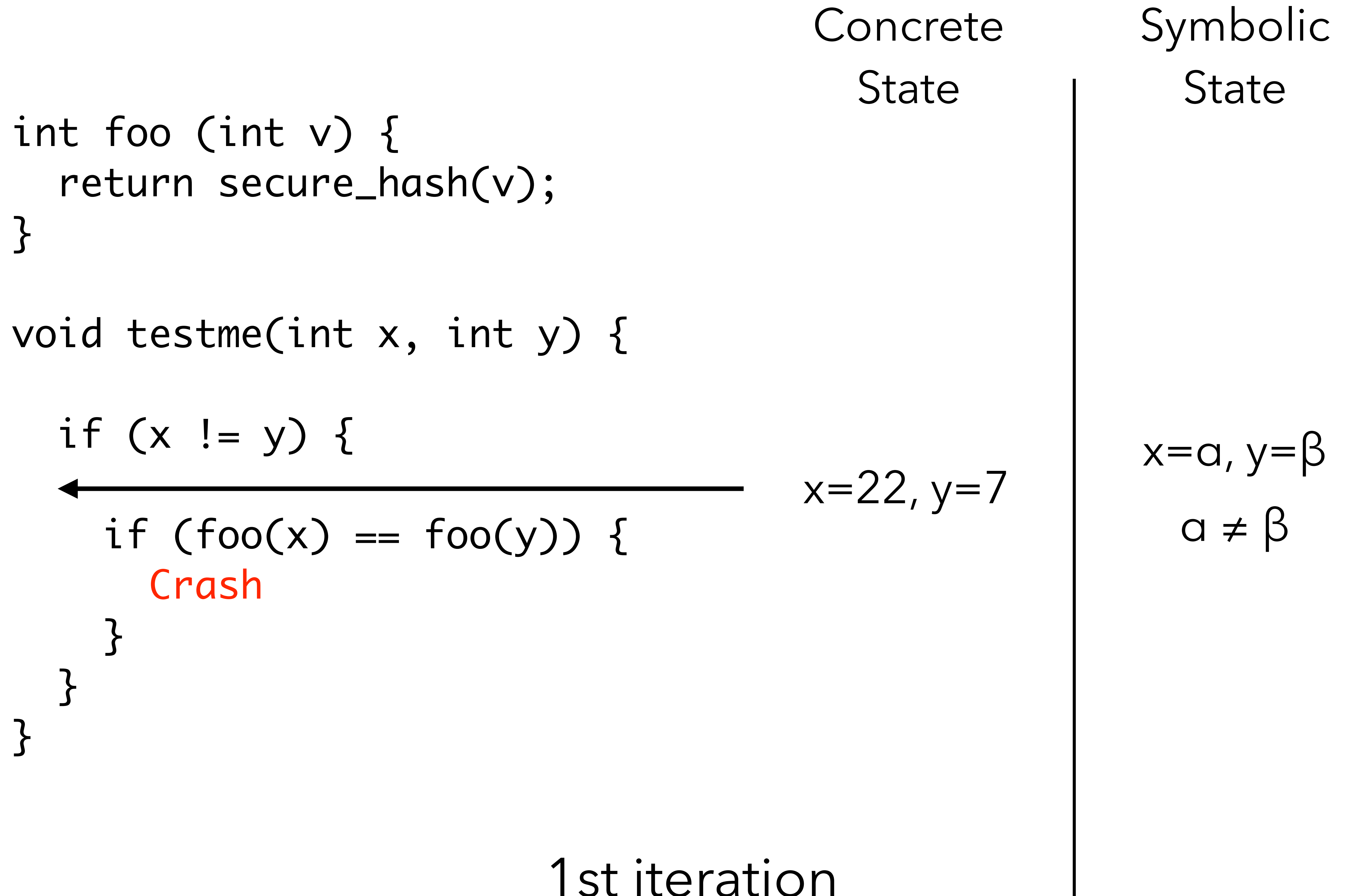
# Limitation of Concolic Testing

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

void testme(int x, int y) {

  if (x != y) {

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

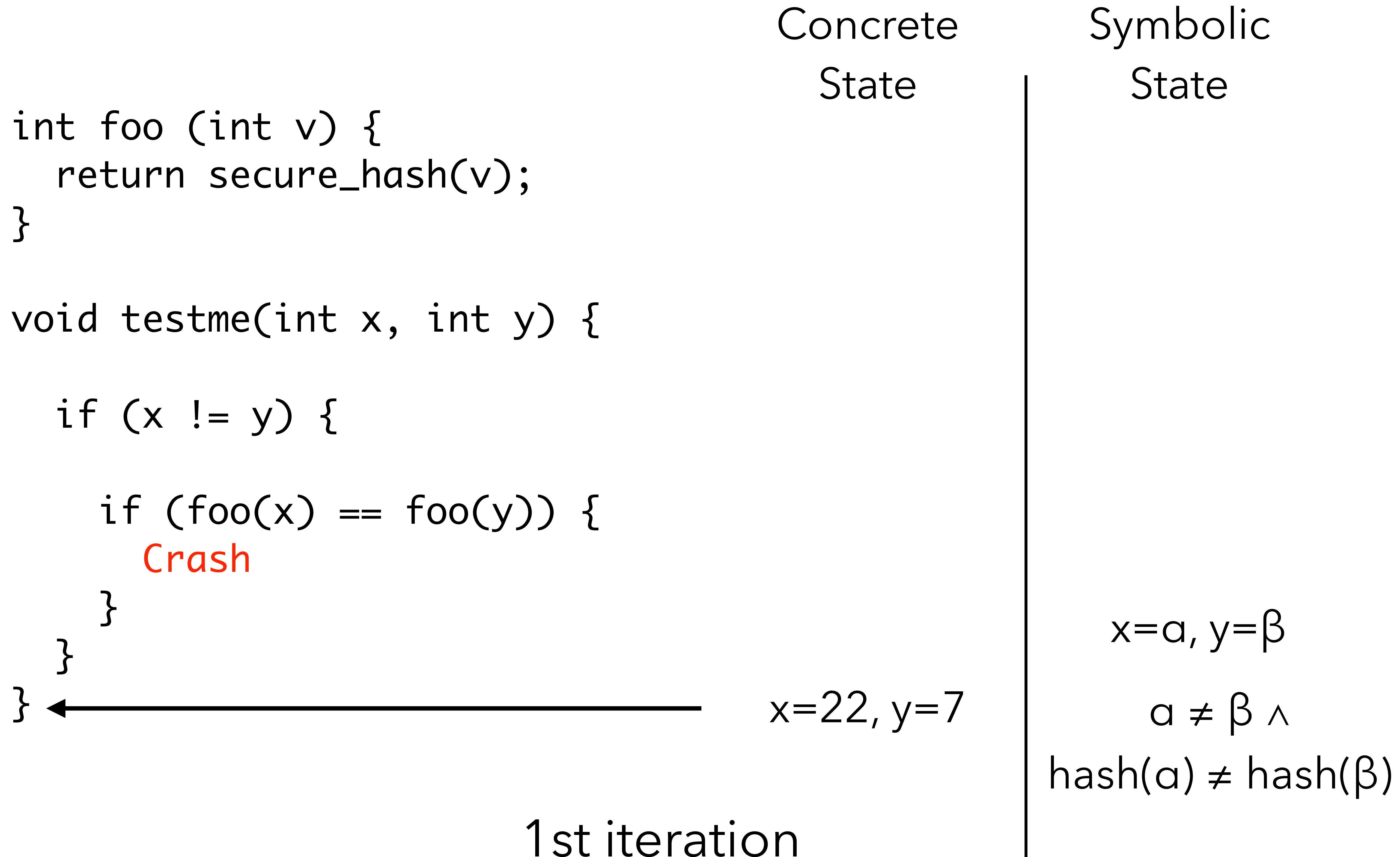
| Concrete State | Symbolic State |
| --- | --- |
| x=22, y=7 | x=α, y=β |
| | true |

1st iteration

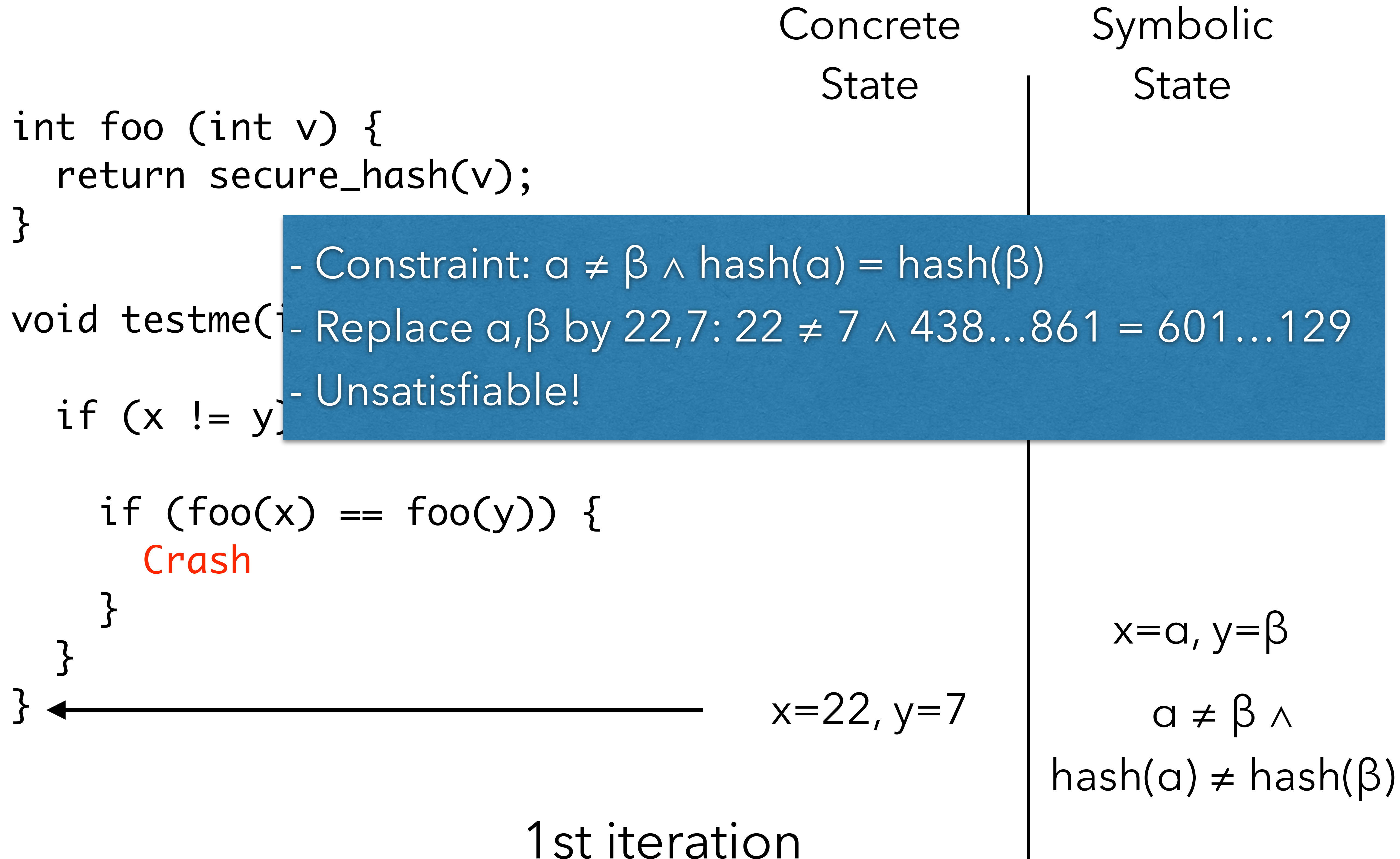
# Limitation of Concolic Testing



# Limitation of Concolic Testing



# Limitation of Concolic Testing



# Limitation of Concolic Testing

Concrete State | Symbolic State

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

void testme(i

  if (x != y)

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

- Constraint: $\alpha \neq \beta \wedge hash(\alpha) = hash(\beta)$
- Replace $\alpha,\beta$ by 22,7: $22 \neq 7 \wedge 438\ldots861 = 601\ldots129$
- Unsatisfiable!

false negative

Concrete: x=22, y=7

Symbolic: $x=\alpha, y=\beta$

$\alpha \neq \beta \wedge$
$hash(\alpha) \neq hash(\beta)$

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 | 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 | Symbolic State |
|---|---|
| x=1 | x=α |
| A = {5,7,9} | true |

1st iteration

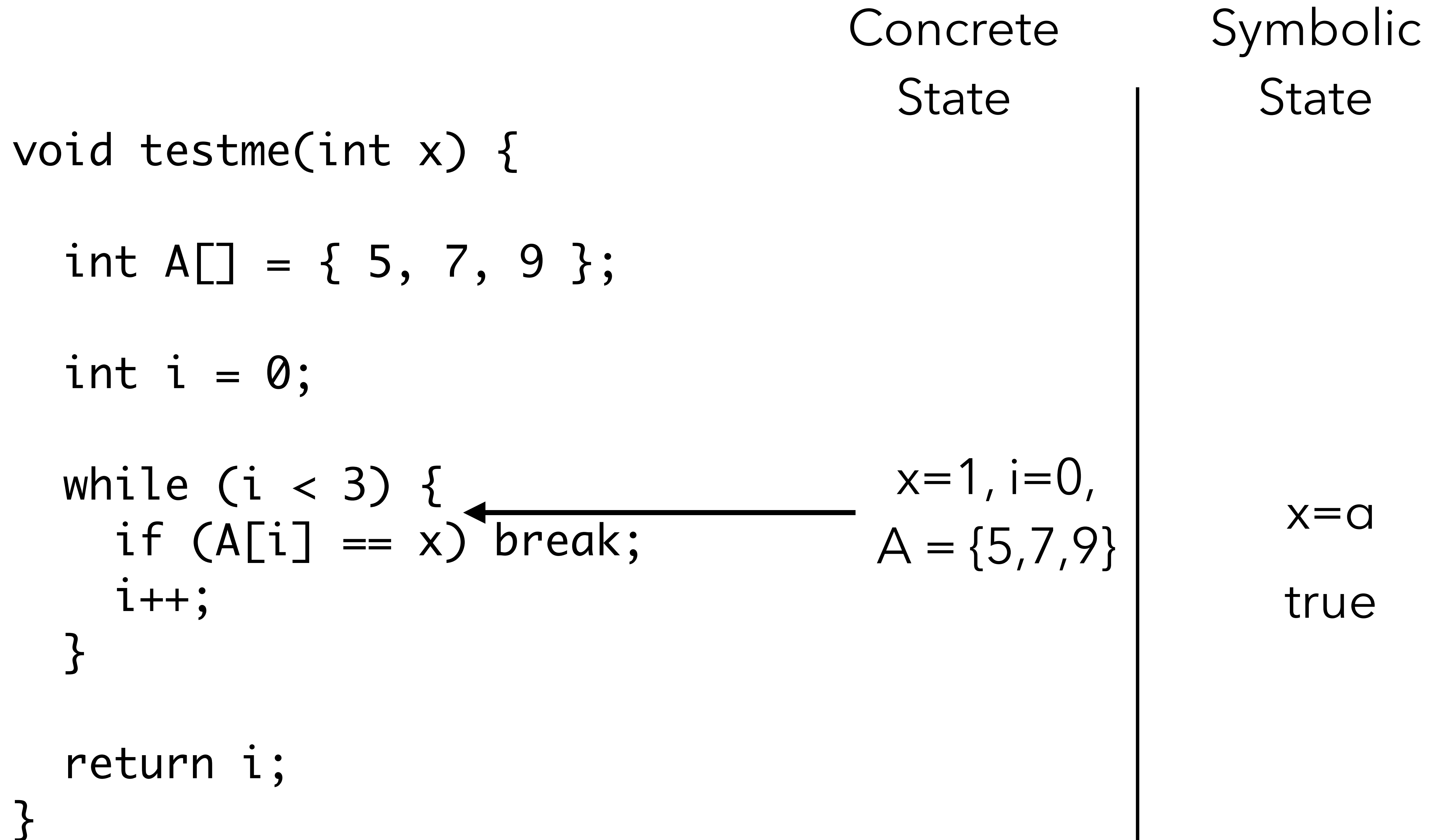
# Testing Loops

```
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 | Symbolic State |
|---|---|
| x=1, i=0, A = {5,7,9} | x=a<br>true |

1st iteration

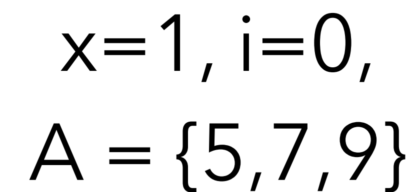
# Testing Loops



# Testing Loops

Concrete State | Symbolic State

```
void testme(int x) {

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

  int i = 0;

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

  return i;
}
```




$x = a$

$5 \neq a$

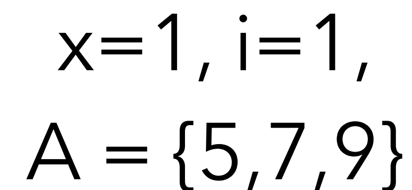
1st iteration

# Testing Loops

Concrete State

Symbolic State

```
void testme(int x) {

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

  int i = 0;

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

  return i;
}
```




x=a

$5 \neq a$

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;
}
```

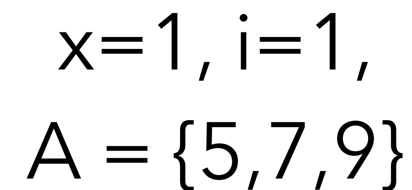
| Concrete State | Symbolic State |
| --- | --- |
| x=1, i=1, A = {5,7,9} | x=a<br>5≠a |

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=a

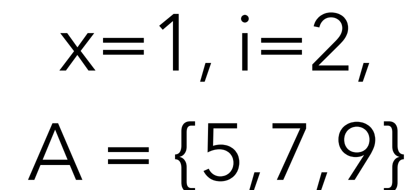
$5 \neq a \wedge 7 \neq a$

1st iteration

# Testing Loops

Concrete State | Symbolic State

```
void testme(int x) {

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

  int i = 0;

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

  return i;
}
```




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<br>$5 \neq a \wedge 7 \neq a$ |

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

x=1, i=2,
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 |
| --- | --- |
| 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 |
| --- | --- |
| x=1, i=3, A = {5,7,9} | x=a<br>$5 \neq a \wedge 7 \neq a \wedge 9 \neq a$ |

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

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=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 | Symbolic State |
| --- | --- |
| 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 | Symbolic State |
|---|---|
| x=9, i=0, A = {5,7,9} | x=a<br>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;
}
```

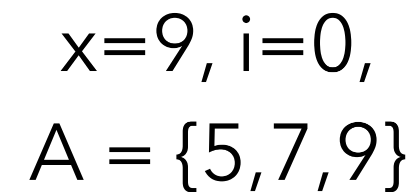
| Concrete State | Symbolic State |
| --- | --- |
| x=9, i=0, A = {5,7,9} | x=a<br>true |

2nd iteration

# Testing Loops

Concrete State | Symbolic State

```
void testme(int x) {

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

  int i = 0;

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

  return i;
}
```




x=a

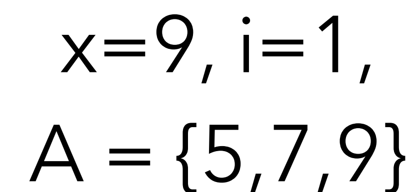
5≠a

2nd iteration

# Testing Loops

Concrete State | Symbolic State

```
void testme(int x) {

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

  int i = 0;

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

  return i;
}
```




x=a

5≠a

2nd iteration

# Testing Loops

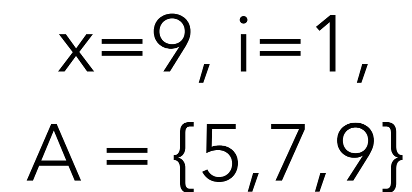
```
void testme(int x) {

  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=1, A = {5,7,9} | x=a<br>$5 \neq a$ |

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;
}
```

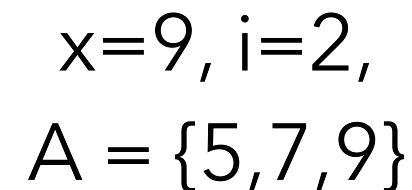
| Concrete State | Symbolic State |
| --- | --- |
|  | x=a<br>$5 \neq a \wedge 7 \neq a$ |

2nd iteration

# Testing Loops

Concrete State | Symbolic State

```
void testme(int x) {

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

  int i = 0;

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

  return i;
}
```




$x = a$

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

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<br>$5 \neq a \wedge 7 \neq a$ |

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,<br>A = {5,7,9} | x=a<br>$5 \neq a \wedge 7 \neq a \wedge 9 = a$ |

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 |
|---|---|
| x=9, i=2, A = {5,7,9} | x=a; $5 \neq a \wedge 7 \neq a \wedge 9 = a$ |

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

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=7 | x=a |
| | true |

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 |
| --- | --- |
| x=7, A = {5,7,9} | x=a true |

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 |
| --- | --- |
| x=7, i=0, A = {5,7,9} | x=a true |

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

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

Symbolic State

x=a

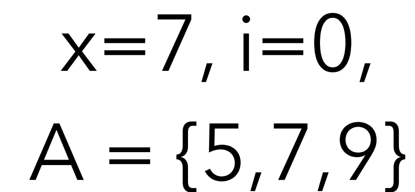
true

3rd iteration

# Testing Loops

Concrete State | Symbolic State

```
void testme(int x) {

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

  int i = 0;

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

  return i;
}
```




x=a

5≠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 |
|---|---|
| x=7, i=1, A = {5,7,9} | x=a<br>5≠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 |
|---|---|
| x=7, i=1, A = {5,7,9} | x=a<br>5≠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 |
|---|---|
| $x=7, i=2,$ $A = \{5,7,9\}$ | $x=a$ <br> $5 \neq a \wedge 7=a$ |

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 |
| --- | --- |
| x=7, i=2, A = {5,7,9} | x=a |
| | $5 \neq a \wedge 7 = a$ |

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

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 |
| --- | --- |
| x=5 | 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=5, A = {5,7,9} | x=a true |

4th iteration

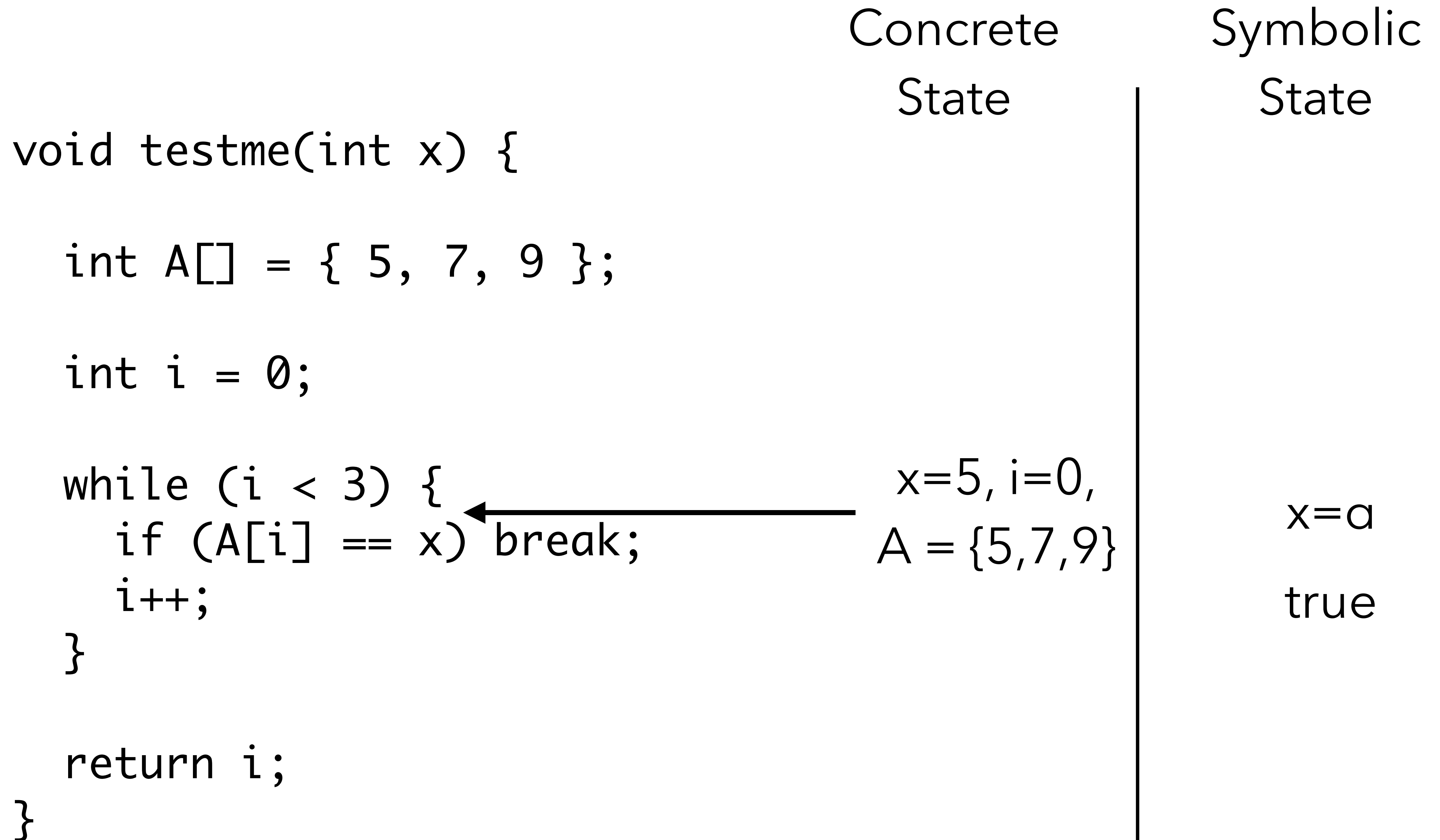
# Testing Loops

```
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=5, i=0, A = {5,7,9} | x=a |
| | true |

4th iteration

# Testing Loops



# 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=2, A = {5,7,9} | x=a<br>5=a |

4th iteration

# 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=236<br>p=NULL | x=α, p=β<br>true |

1st iteration

# 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=236 | x=α, p=β |
| p=NULL | α > 0 |

1st iteration

# 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=236<br>p=NULL | x=a, p=β<br>$a > 0 \wedge$<br>$\beta$ = NULL |

1st iteration

# 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=236 | x=α, p=β |
| p=NULL | α > 0 ∧ β = NULL |

- Constraint: $\alpha > 0 \wedge \beta \neq NULL$
- Solution: $\alpha = 236$, $\beta$ = [634 | NULL]

1st iteration

# 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=α, p=β |
| x=236 | $p\text{->}data = \gamma$ |
| p= [634 \| NULL] | $p\text{->}next = \delta$ |
| | true |

2nd iteration

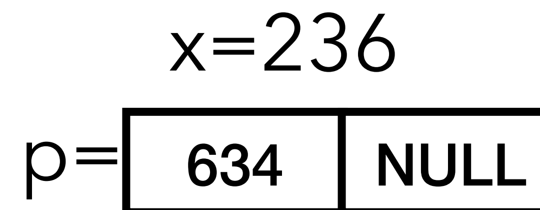
# Testing Data Structures

```
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

x=236

p=




Symbolic State

x=α, p=β

p->data = $\gamma$

p->next = $\delta$

α > 0

2nd iteration

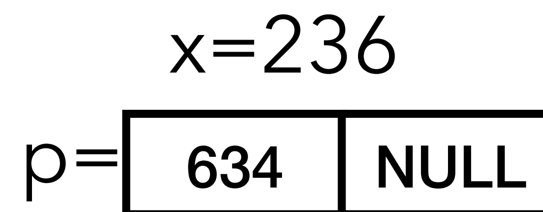
# Testing Data Structures

```
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

x=236

p=




Symbolic State

$x = \alpha$, $p = \beta$

$p\text{->}data = \gamma$

$p\text{->}next = \delta$

$\alpha > 0 \wedge$

$\beta \neq NULL$

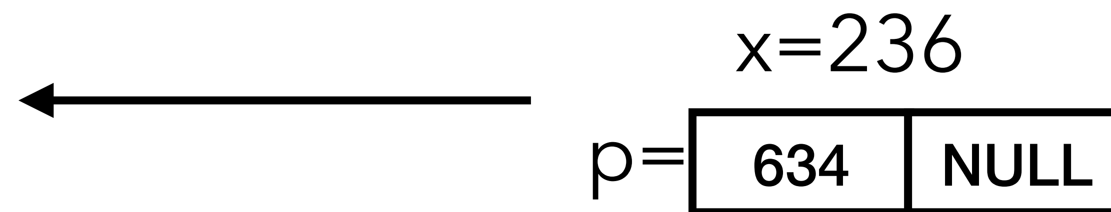
2nd iteration

# 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

x=236




Symbolic State

x=a, p=β

p->data = $\gamma$

p->next = $\delta$

a > 0 ∧

β ≠ NULL ∧

2*a+1 ≠ $\gamma$

2nd iteration

# Testing Data Structures

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

int foo(int v) { return

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

- Constraint: $\alpha>0 \wedge \beta\neq NULL \wedge 2*\alpha+1=\gamma$
- Solution: $\alpha = 1, \beta =$ [3 | NULL]

x=236

p= [634 | NULL]

$x=\alpha, p=\beta$

$p\text{->}data = \gamma$

$p\text{->}next = \delta$

$\alpha > 0 \wedge$

$\beta \neq NULL \wedge$

$2*\alpha+1 \neq \gamma$

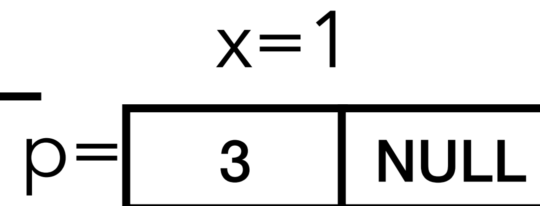
2nd iteration

# 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

x=1




Symbolic State

x=α, p=β

p->data = $\gamma$

p->next = $\delta$

true

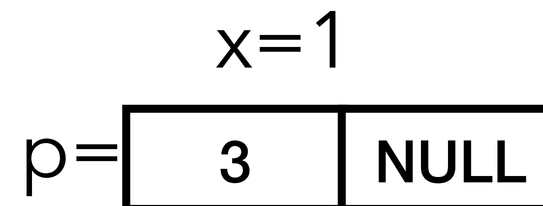
3rd iteration

# 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

x=1




Symbolic State

$x=\alpha, p=\beta$

$p\text{->}data = \gamma$

$p\text{->}next = \delta$

$\alpha > 0 \wedge$

$\beta \neq NULL \wedge$

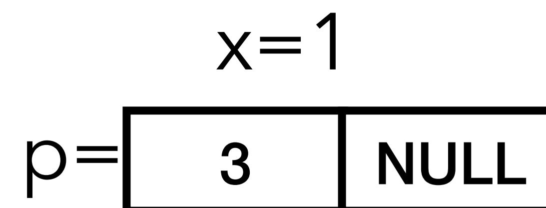
$2*\alpha+1 = \gamma$

3rd iteration

# 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=α, p=β |
| | p->data = $\gamma$ |
| | p->next = $\delta$ |
| x=1 | $\alpha > 0 \wedge$ |
| p= | $\beta \neq$ NULL $\wedge$ |
| | $2*\alpha+1 = \gamma \wedge$ |
| | $\delta \neq \beta$ |

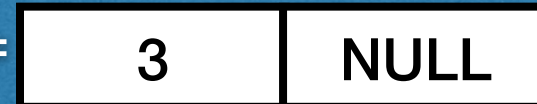



3rd iteration

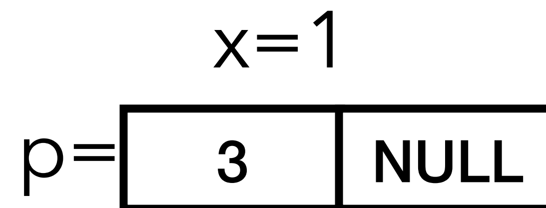
# Testing Data Structures

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

int foo(int v) { re

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

- Constraint: $a>0 \wedge \beta \neq NULL \wedge 2*a+1=\gamma \wedge \delta=\beta$
- Solution: $a = 1, \beta =$ 

$x=1$

p= 


$x=a, p=\beta$

$p\text{->}data = \gamma$

$p\text{->}next = \delta$

$a > 0 \wedge$

$\beta \neq NULL \wedge$

$2*a+1 = \gamma \wedge$

$\delta \neq \beta$

3rd iteration

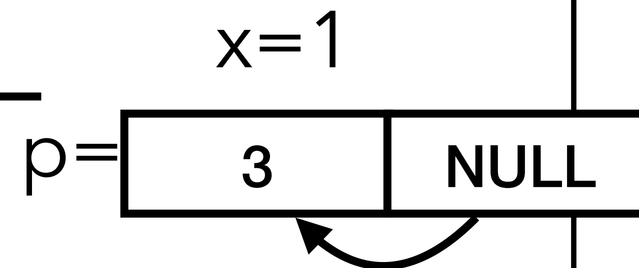
# Testing Data Structures

```
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

x=1

p=




Symbolic State

x=α, p=β

p->data = $\gamma$

p->next = $\delta$

true

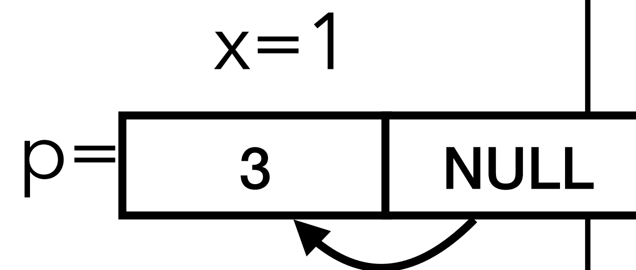
4th iteration

# 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

x=1




Symbolic State

$x = \alpha, p = \beta$

$p\text{->}data = \gamma$

$p\text{->}next = \delta$

$\alpha > 0 \wedge$

$\beta \neq NULL \wedge$

$2*\alpha+1 = \gamma \wedge$

$\delta = \beta$

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