EC4219: Software Engineering

Lecture 15 — Symbolic Execution (2)
More Complex Examples

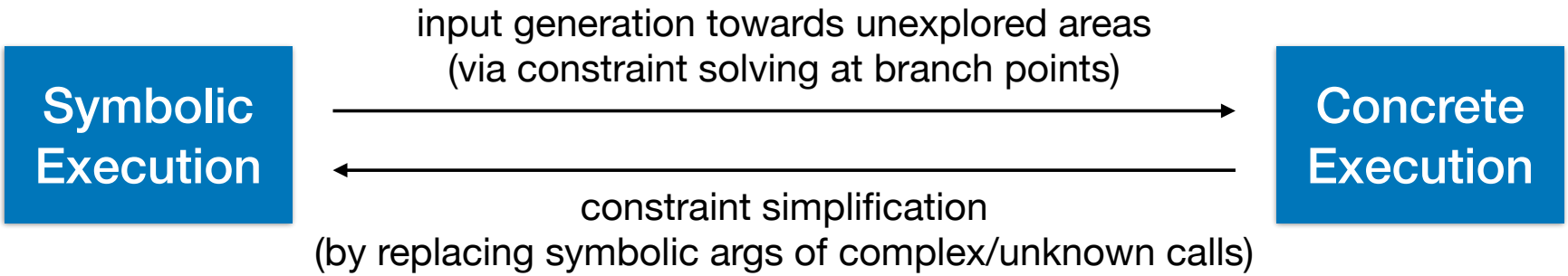
Sunbeom So
2024 Spring

Review

Symbolic execution is a testing approach for systemically finding SW bugs.

- Key idea: execute programs "symbolically" using "symbolic" inputs
- Challenge: path explosion + constraint solving

Concolic testing is a hybrid approach to mitigate the second challenge.



- Downside: false negative due to overly-constrained simplification
  - E.g., most hash functions are not collision-free thus F can be satisfiable. However, if $\alpha = 22$, $\beta = 7$, and $\mathbf{hash}(22) \neq \mathbf{hash}(7)$ from concrete execution, we incorrectly decide F is unsatisfiable.

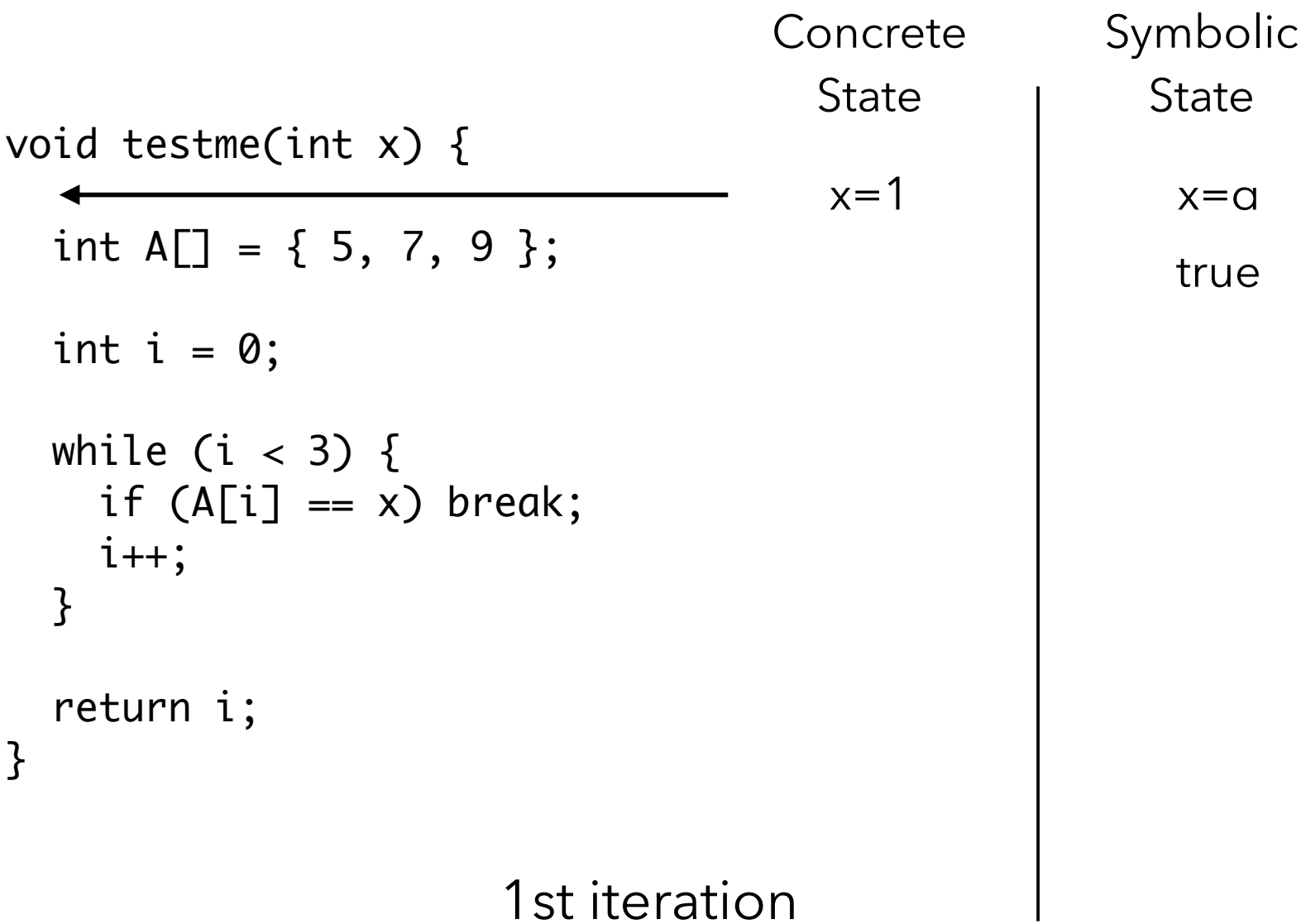
$$F : \alpha \neq \beta \wedge \mathbf{hash}(\alpha) = \mathbf{hash}(\beta)$$

This Lecture

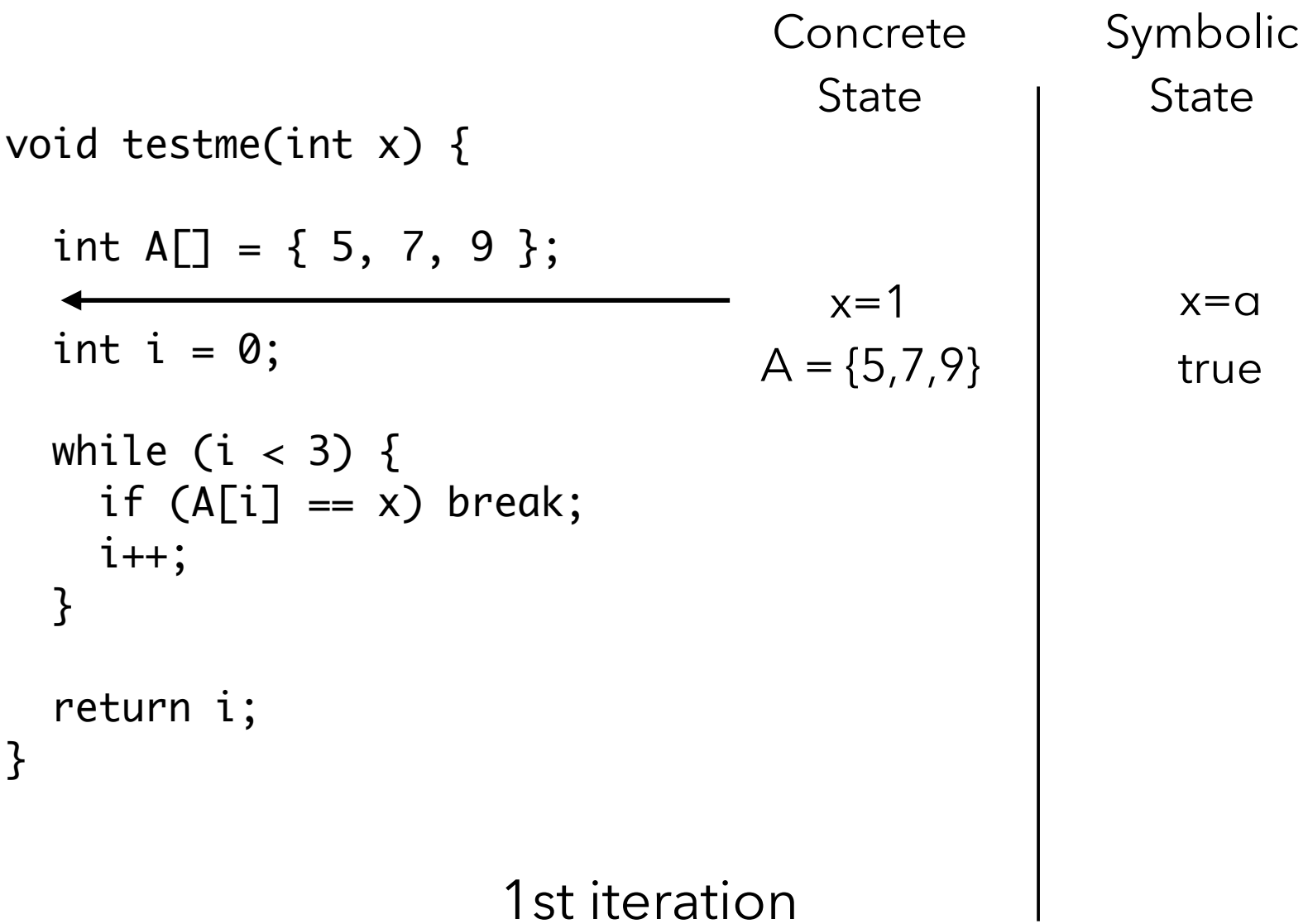
We will study how concolic testing works in more complex cases.

- Concolic testing of programs with loops
- Concolic testing of programs with data structures

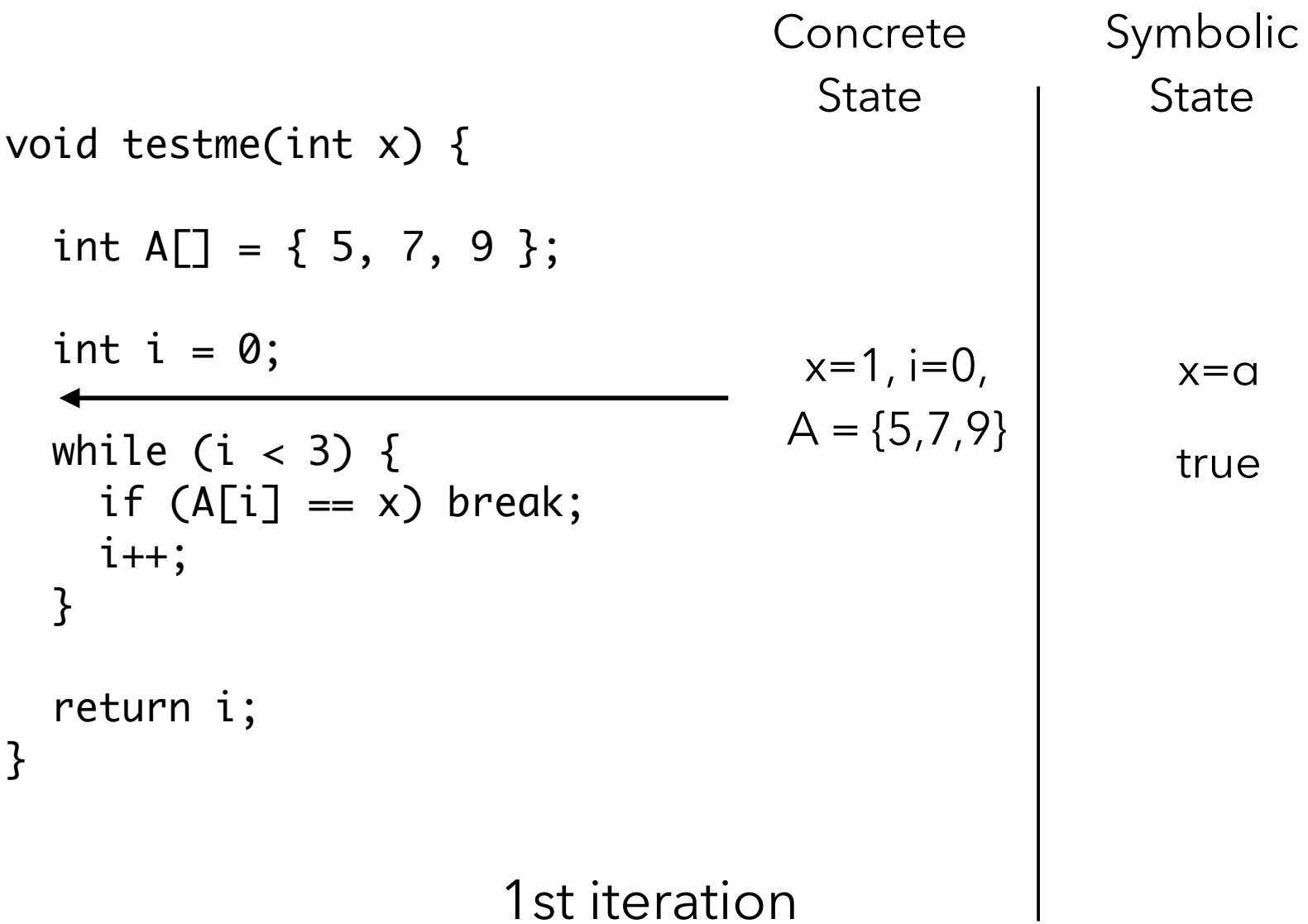
Example: Concolic Testing of Loops



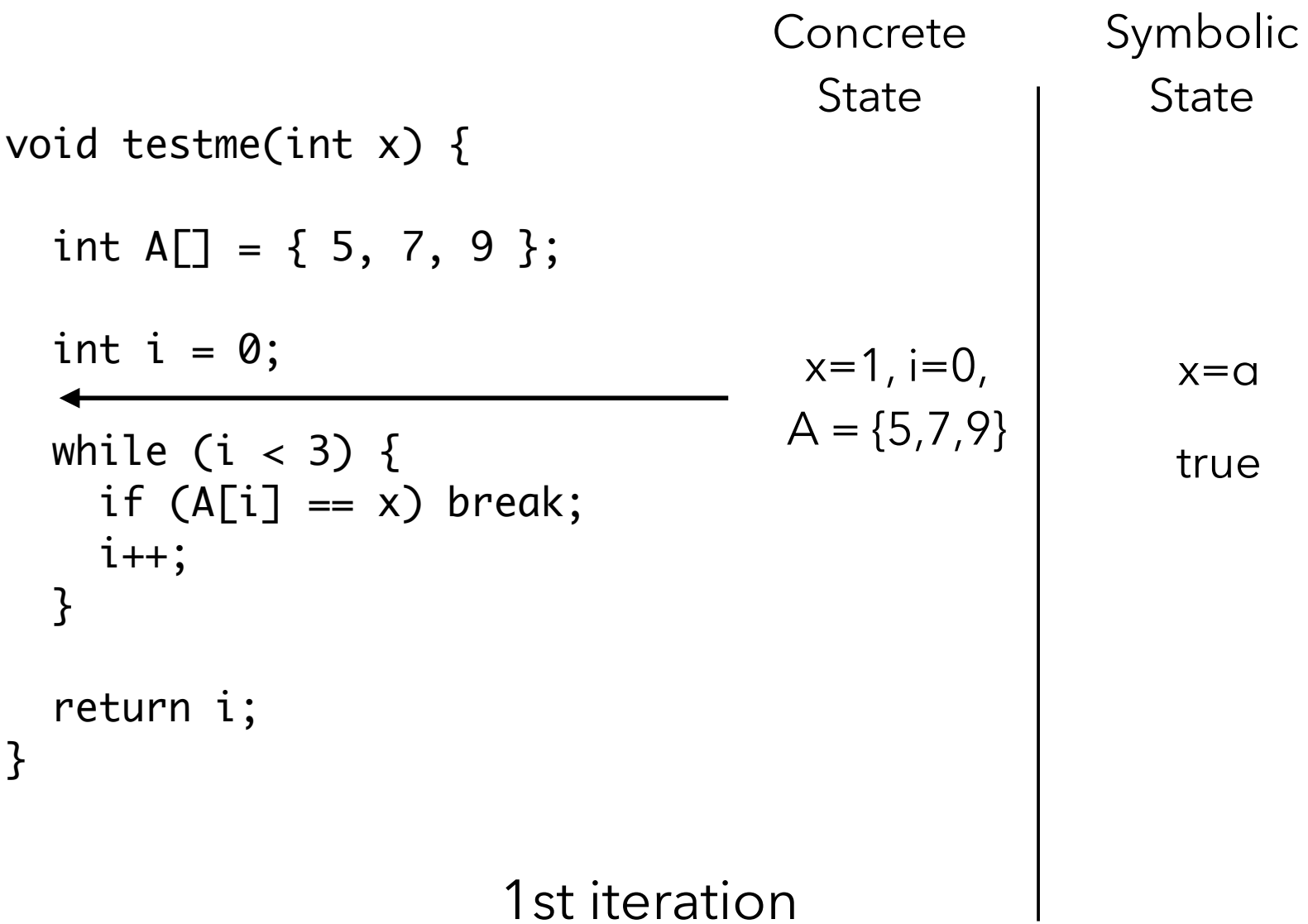
Example: Concolic Testing of Loops (Cont'd)



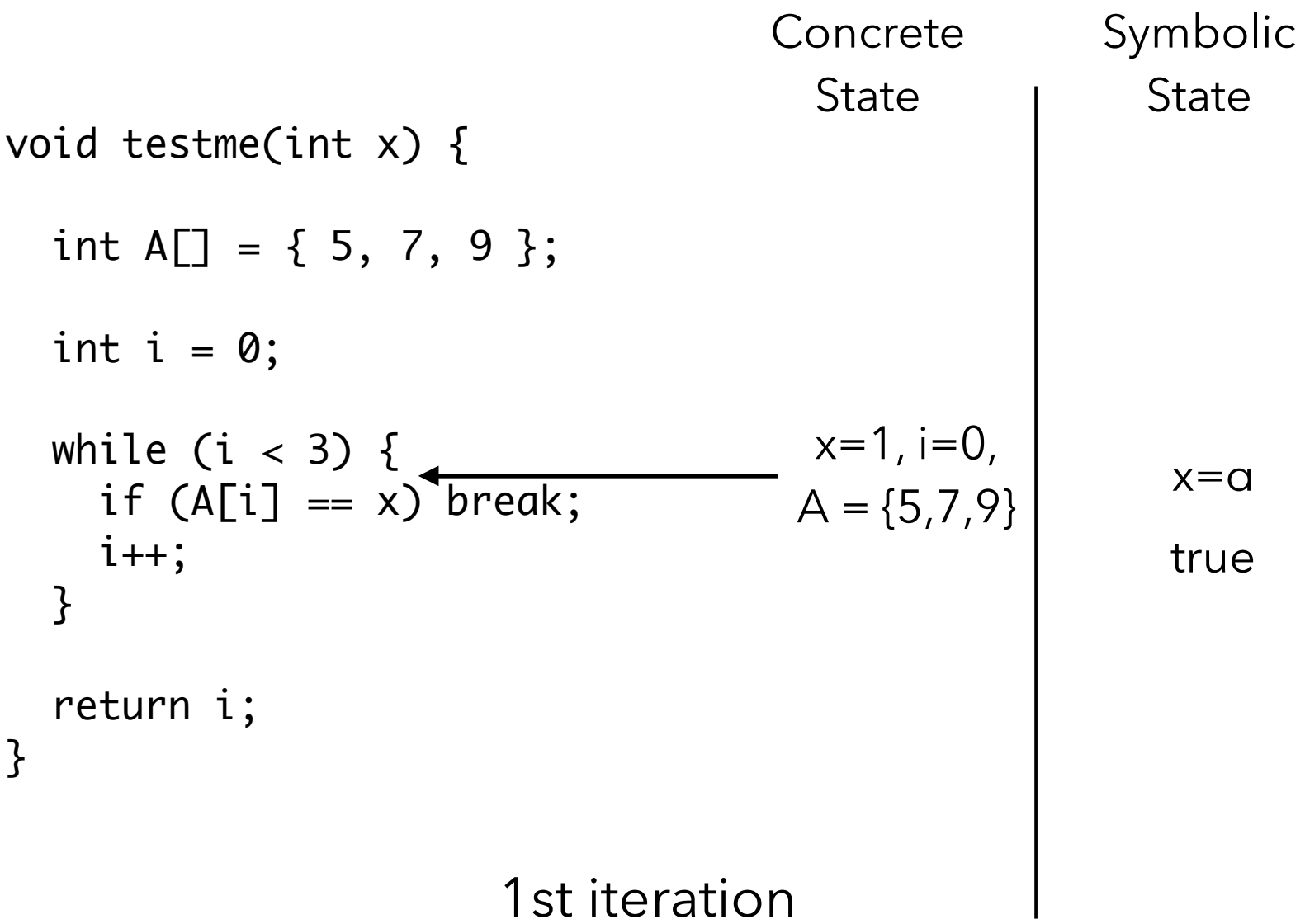
Example: Concolic Testing of Loops (Cont'd)



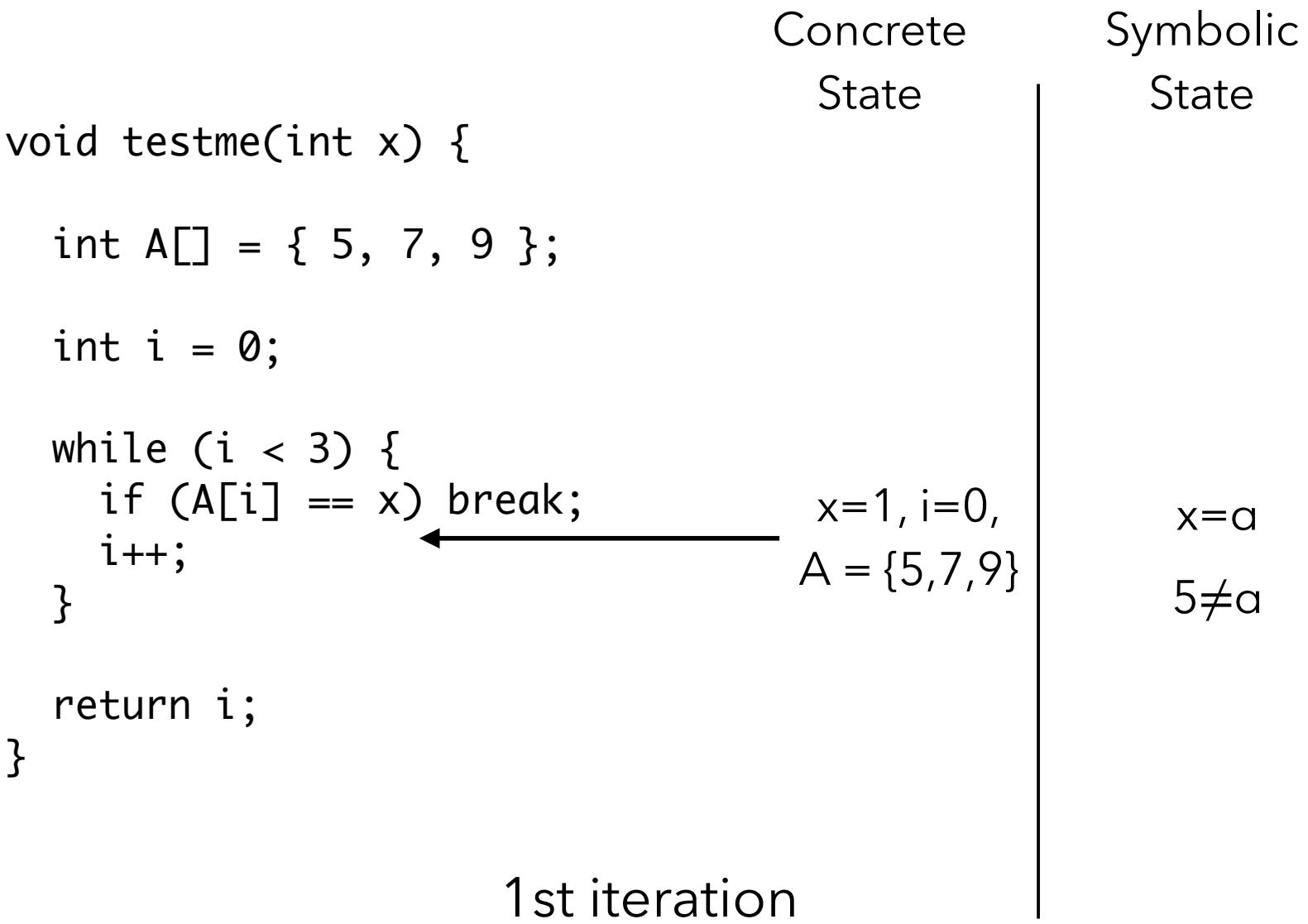
Example: Concolic Testing of Loops (Cont'd)



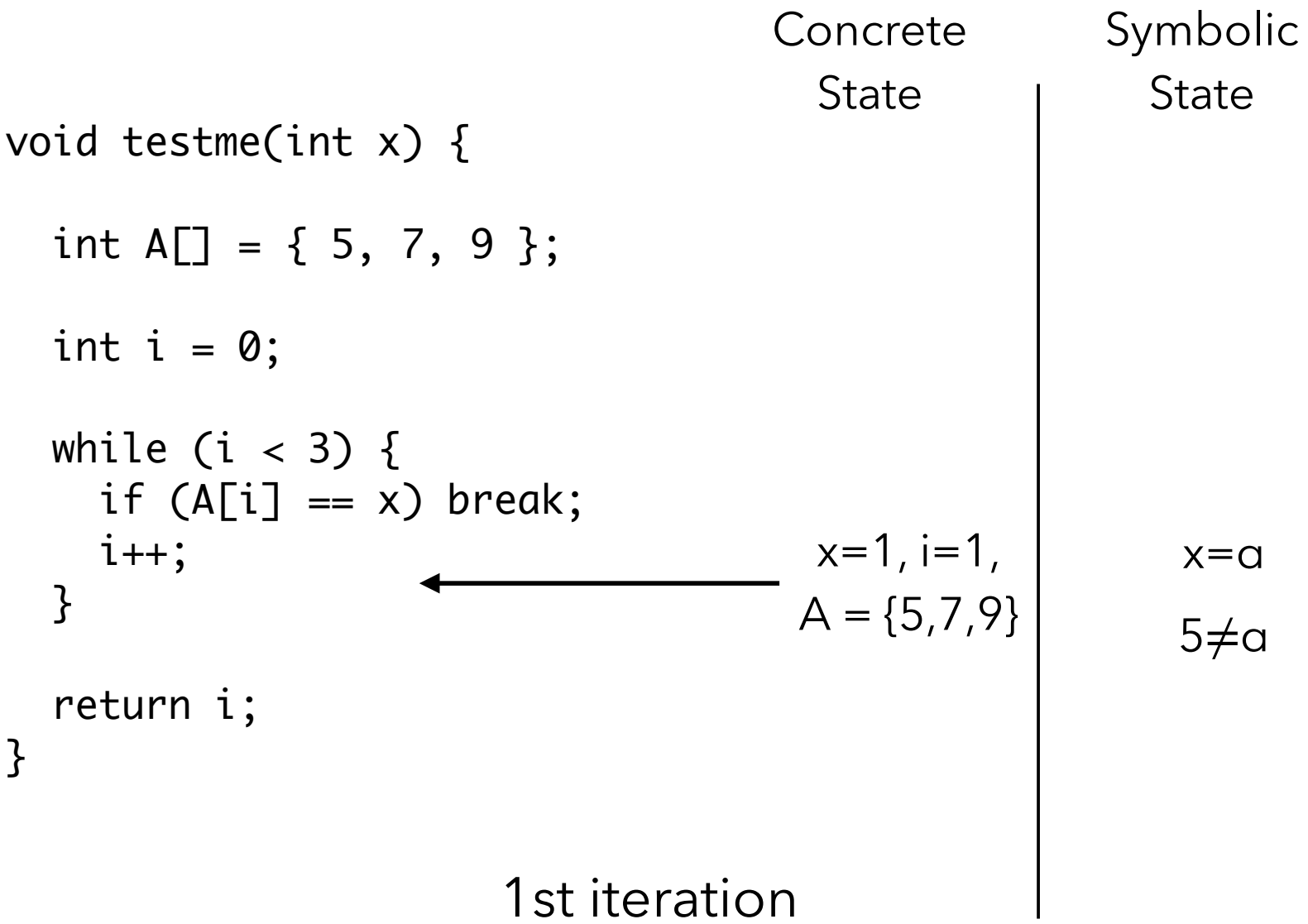
Example: Concolic Testing of Loops (Cont'd)



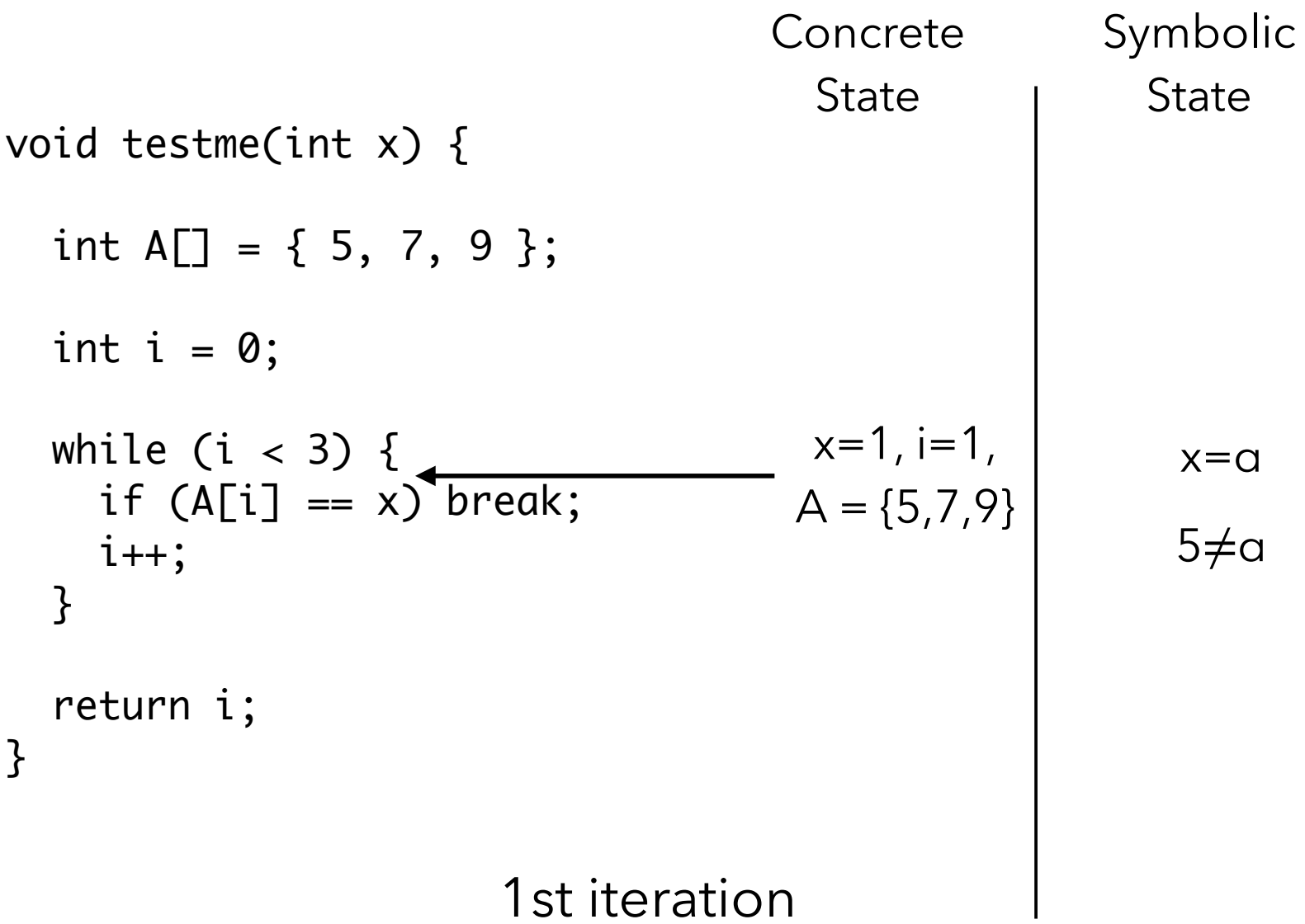
Example: Concolic Testing of Loops (Cont'd)



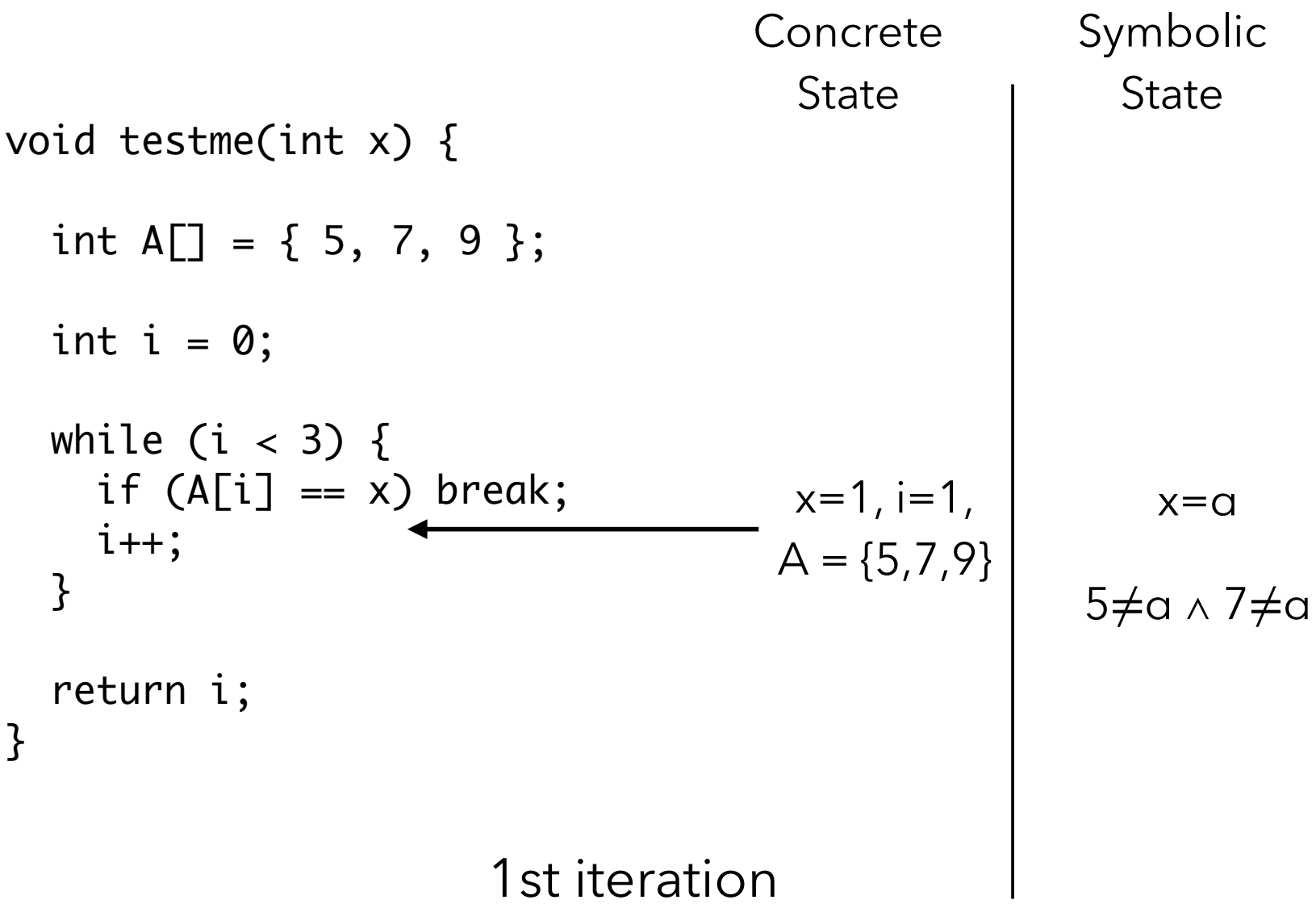
Example: Concolic Testing of Loops (Cont'd)



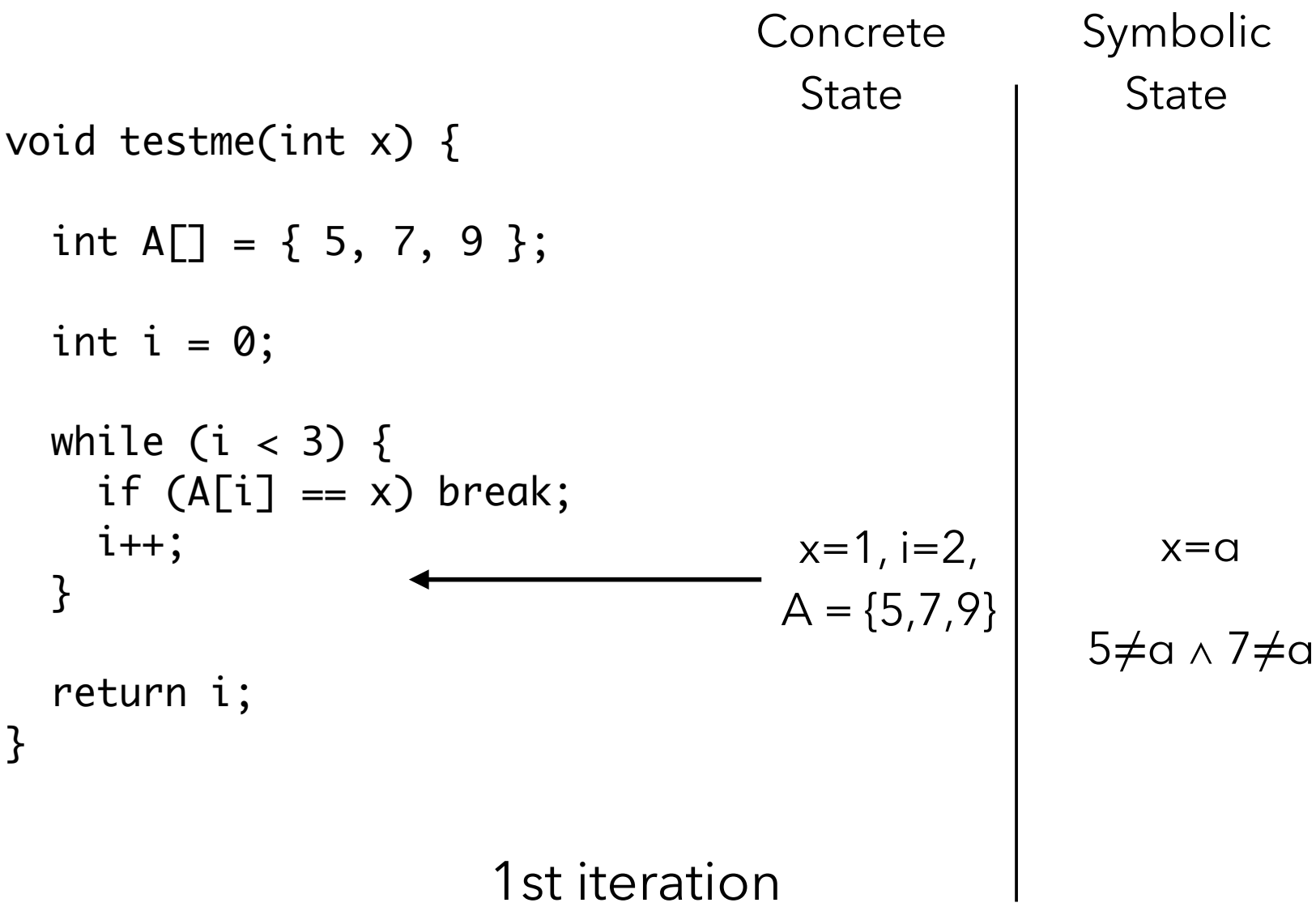
Example: Concolic Testing of Loops (Cont'd)



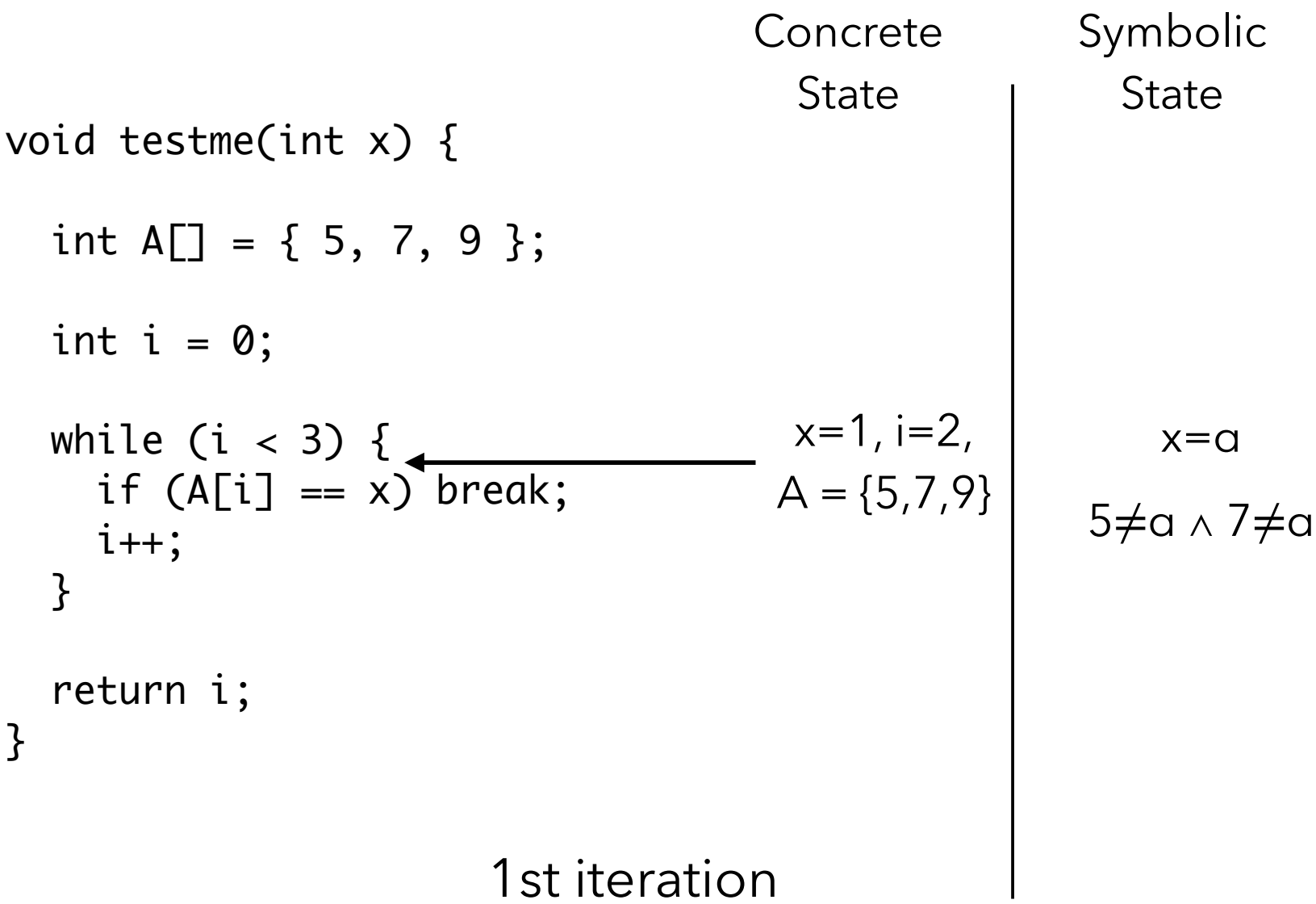
Example: Concolic Testing of Loops (Cont'd)



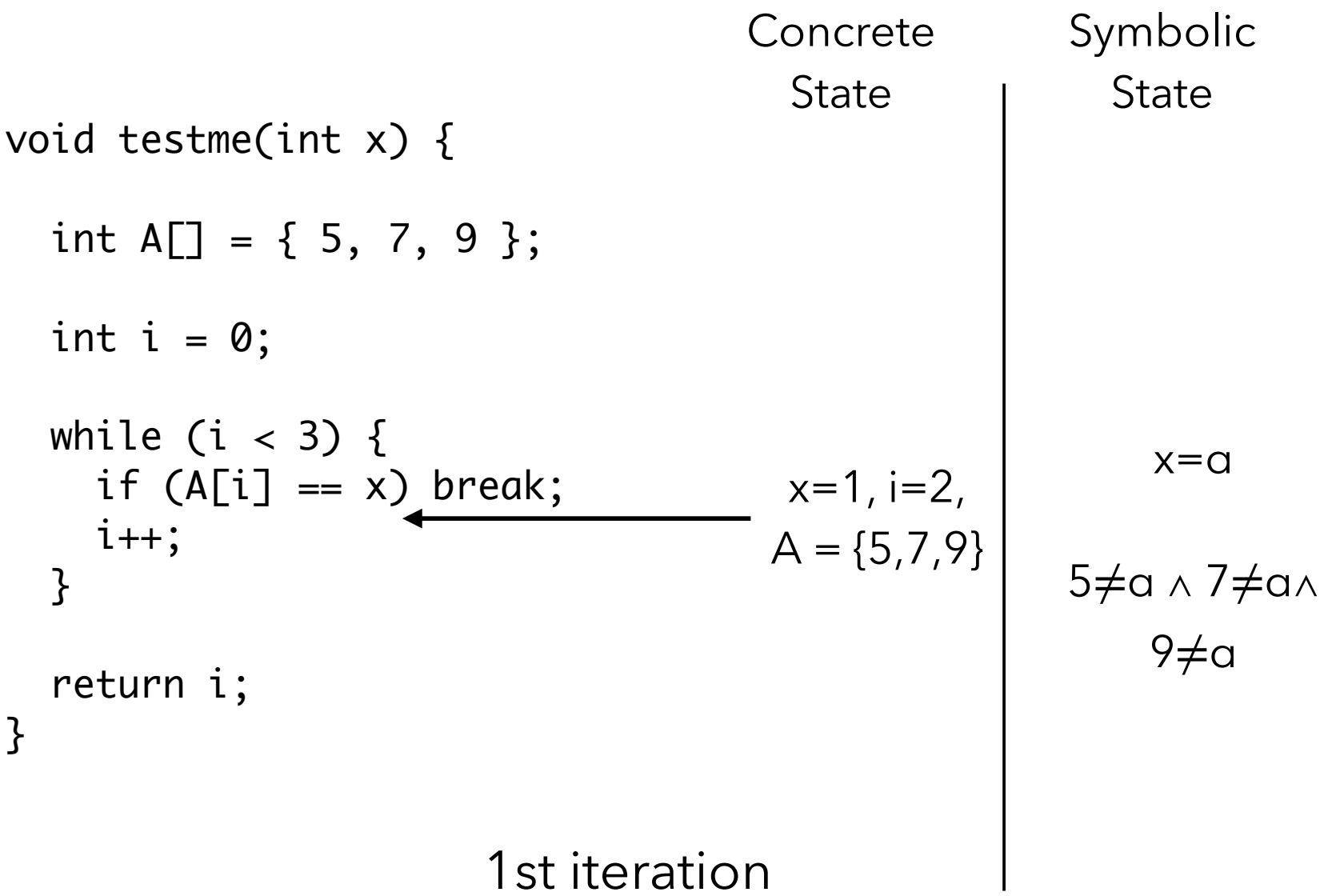
Example: Concolic Testing of Loops (Cont'd)



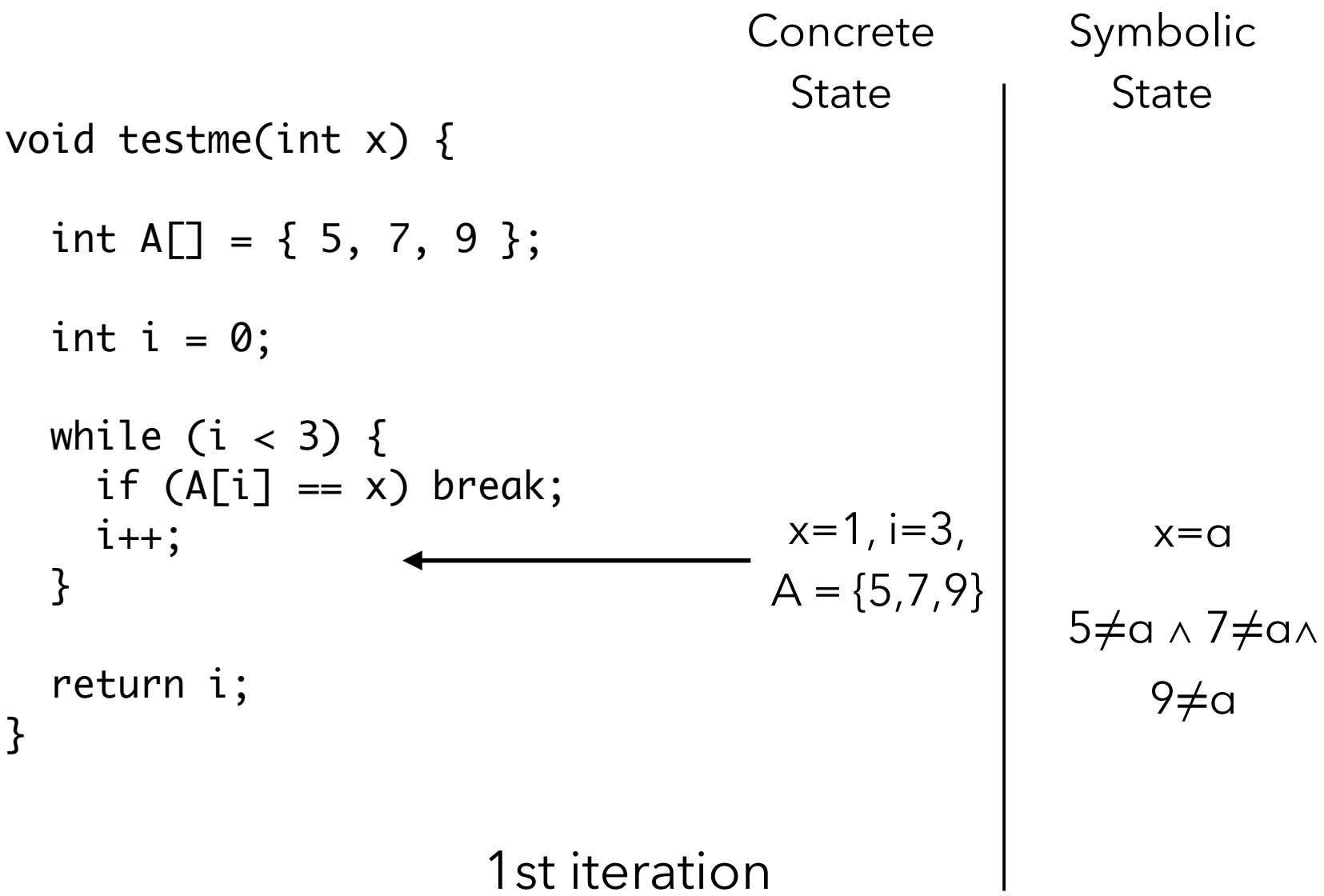
Example: Concolic Testing of Loops (Cont'd)



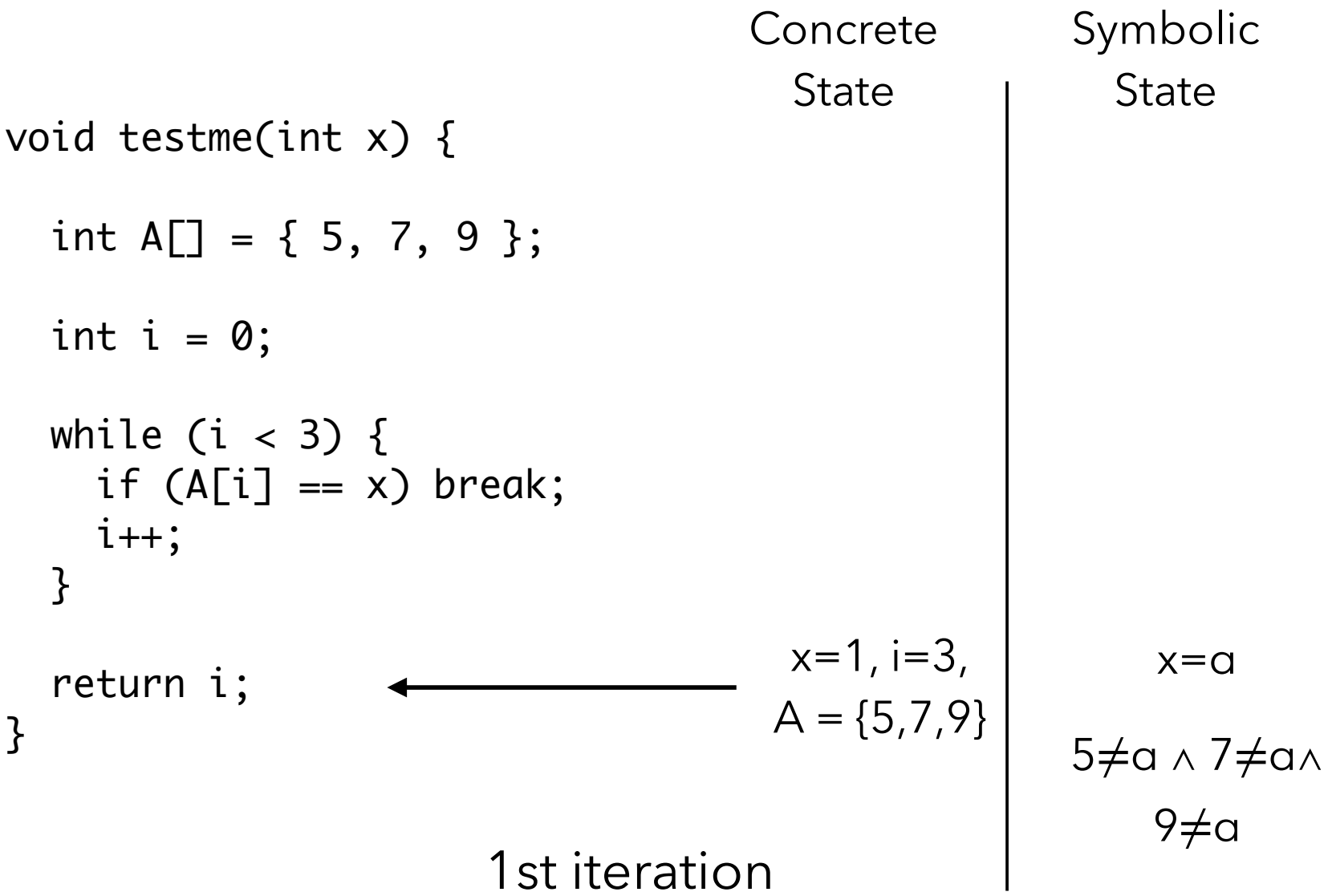
Example: Concolic Testing of Loops (Cont'd)



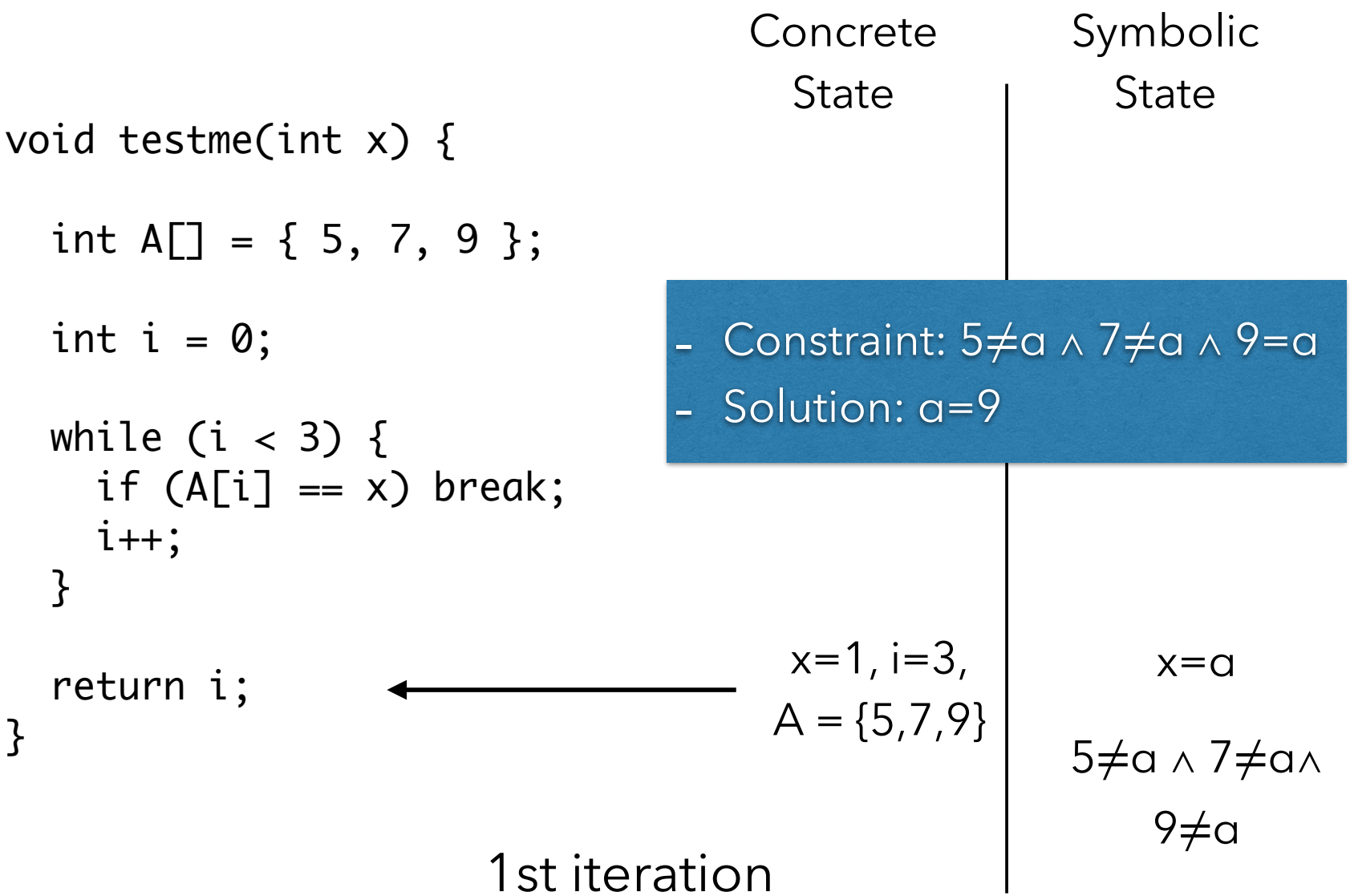
Example: Concolic Testing of Loops (Cont'd)



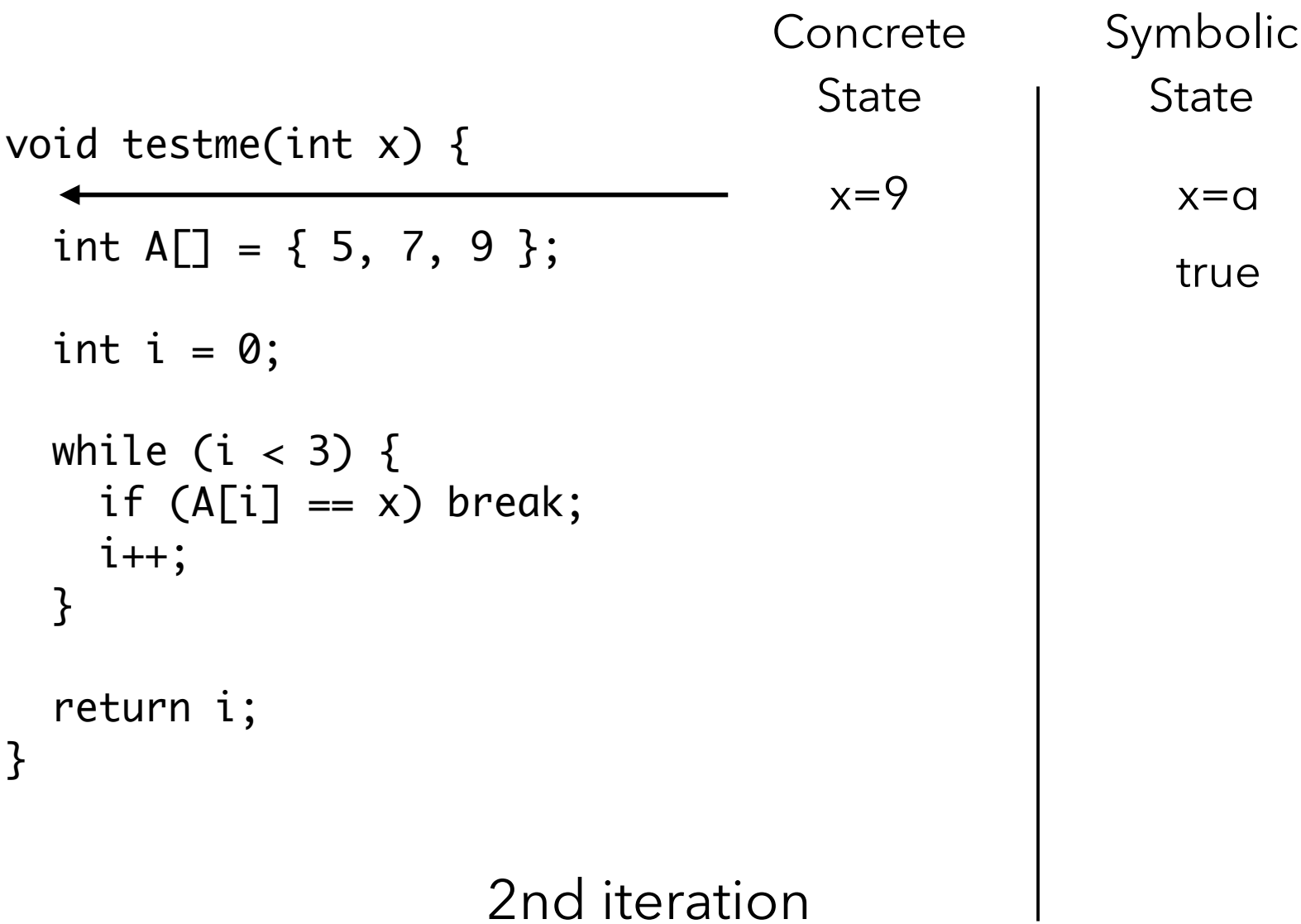
Example: Concolic Testing of Loops (Cont'd)



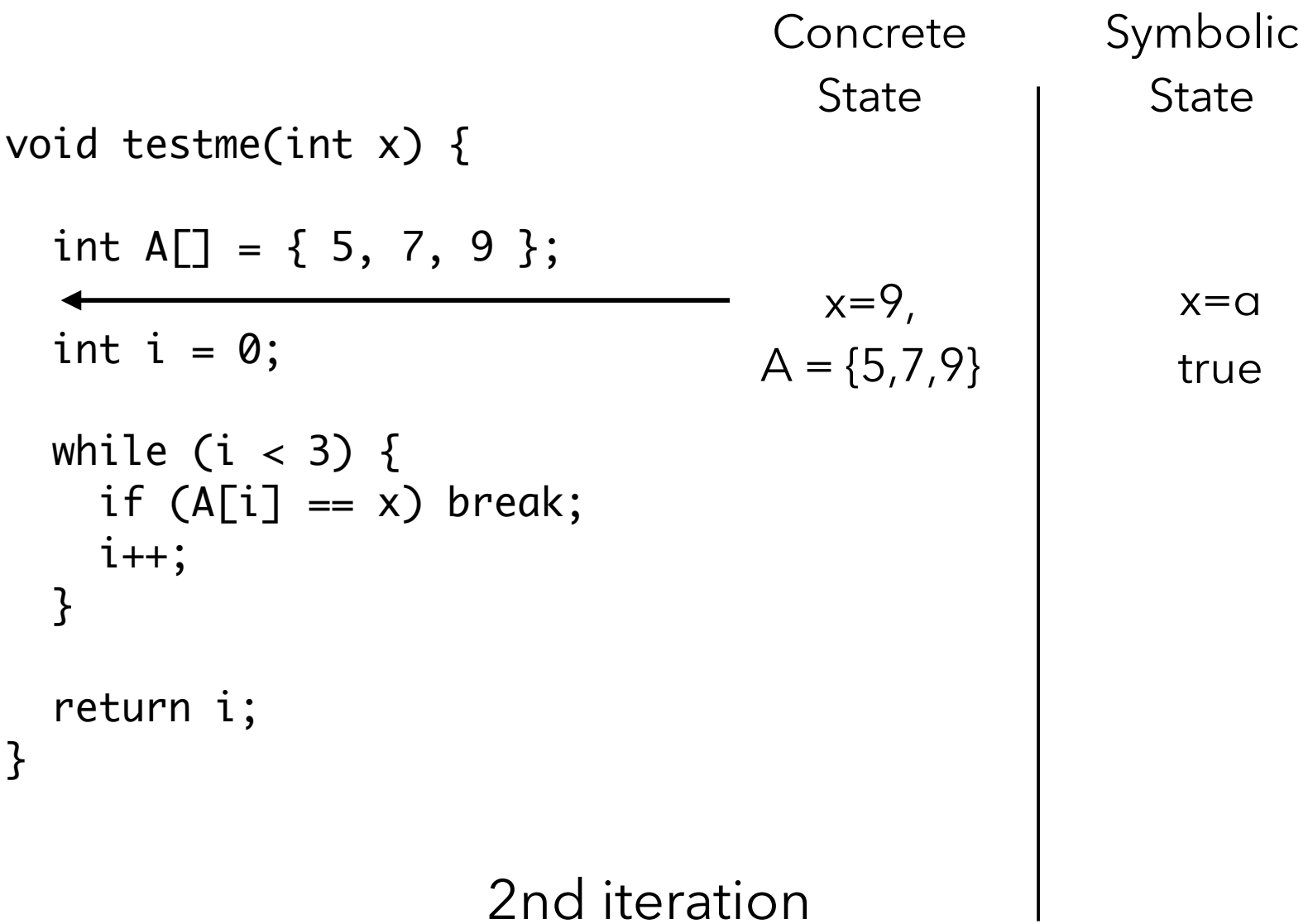
Example: Concolic Testing of Loops (Cont'd)



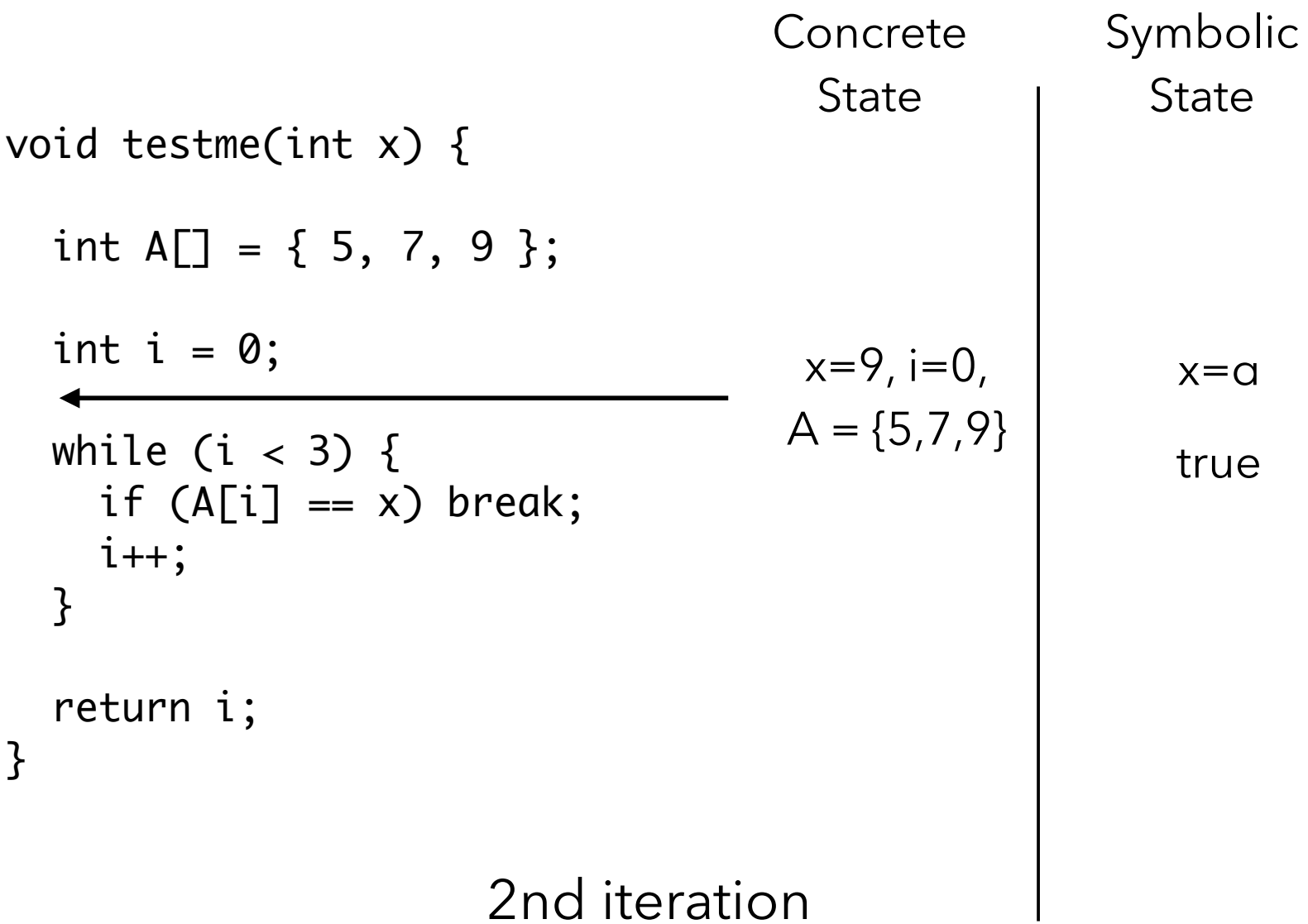
Example: Concolic Testing of Loops (Cont'd)



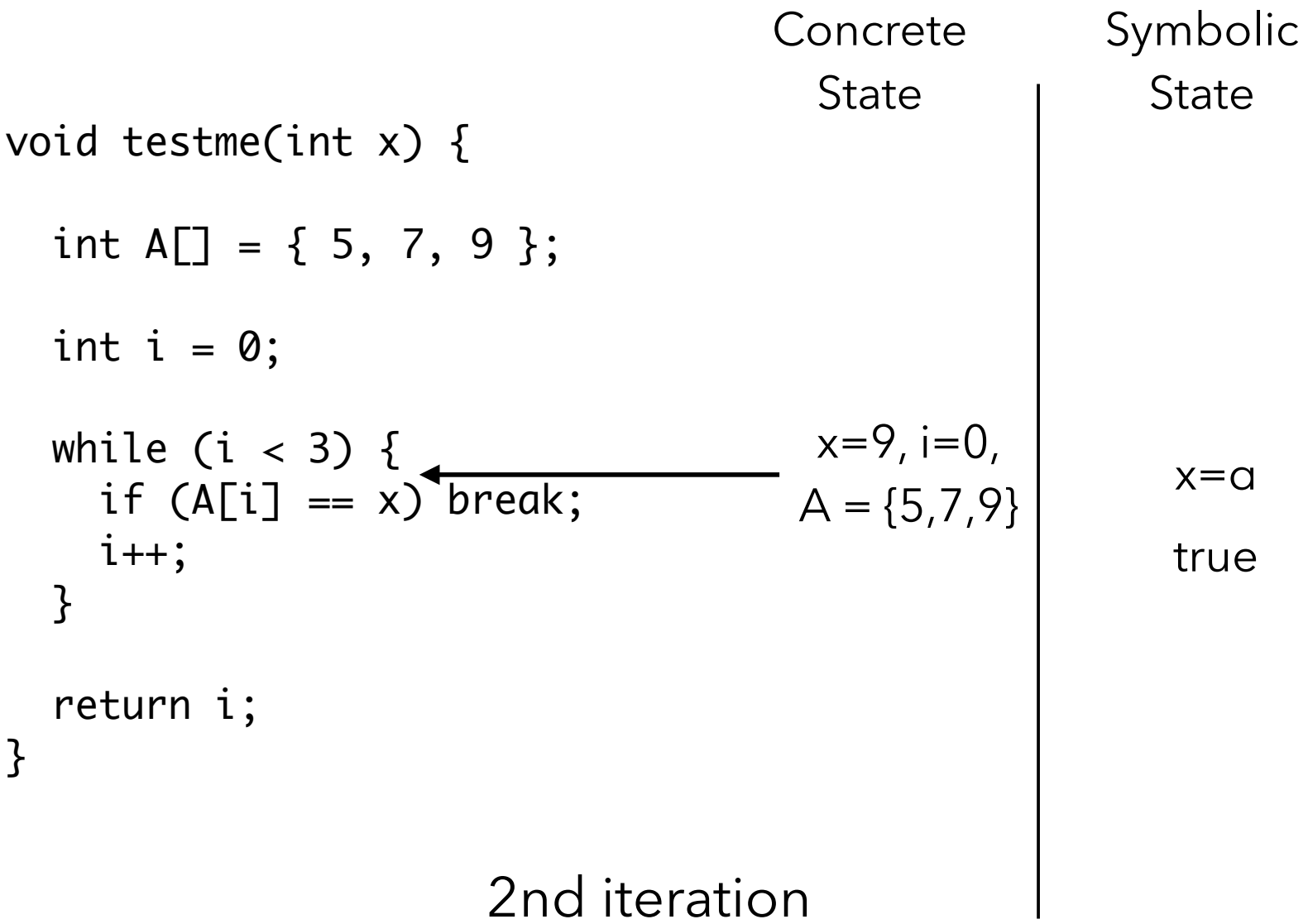
Example: Concolic Testing of Loops (Cont'd)



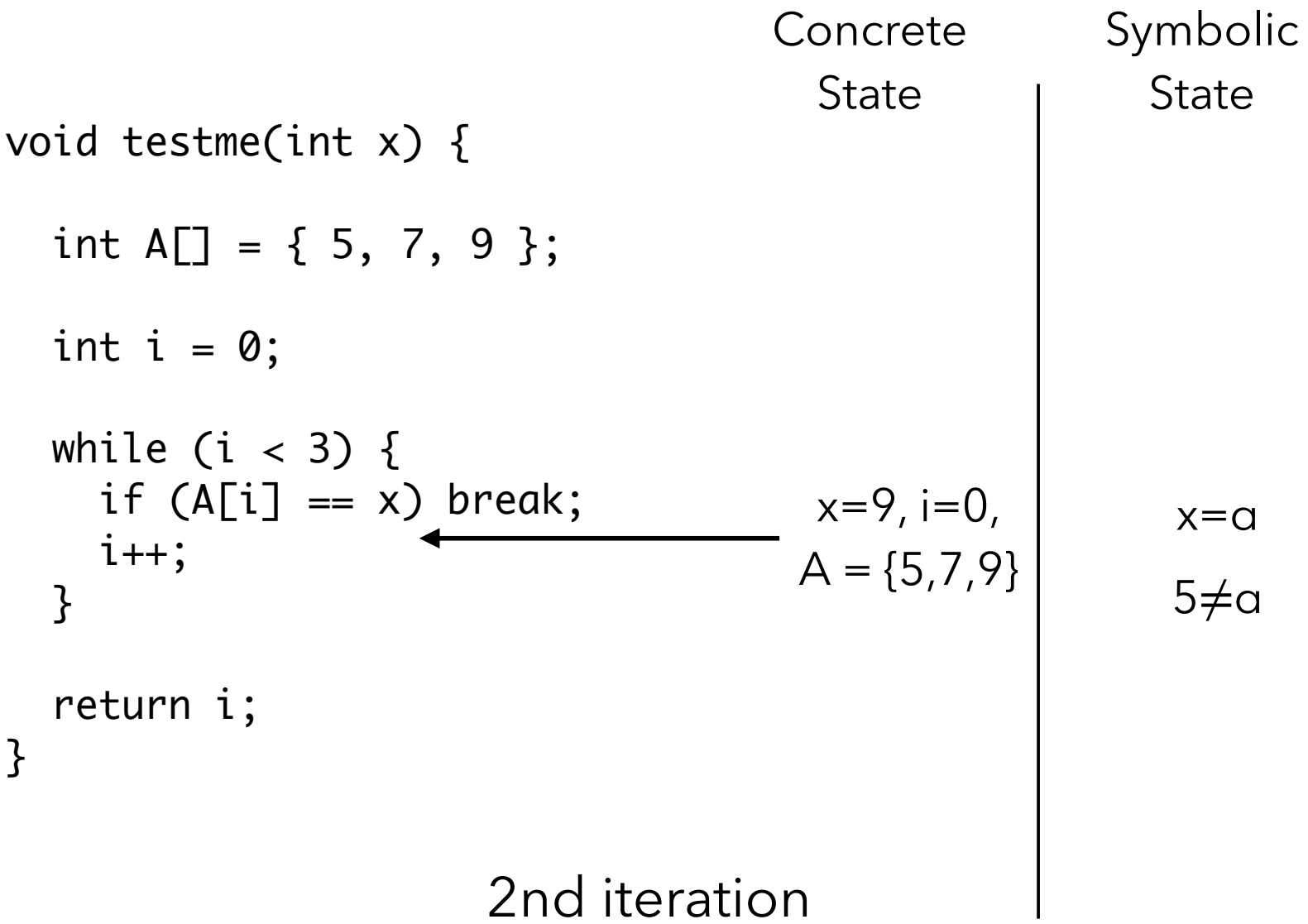
Example: Concolic Testing of Loops (Cont'd)



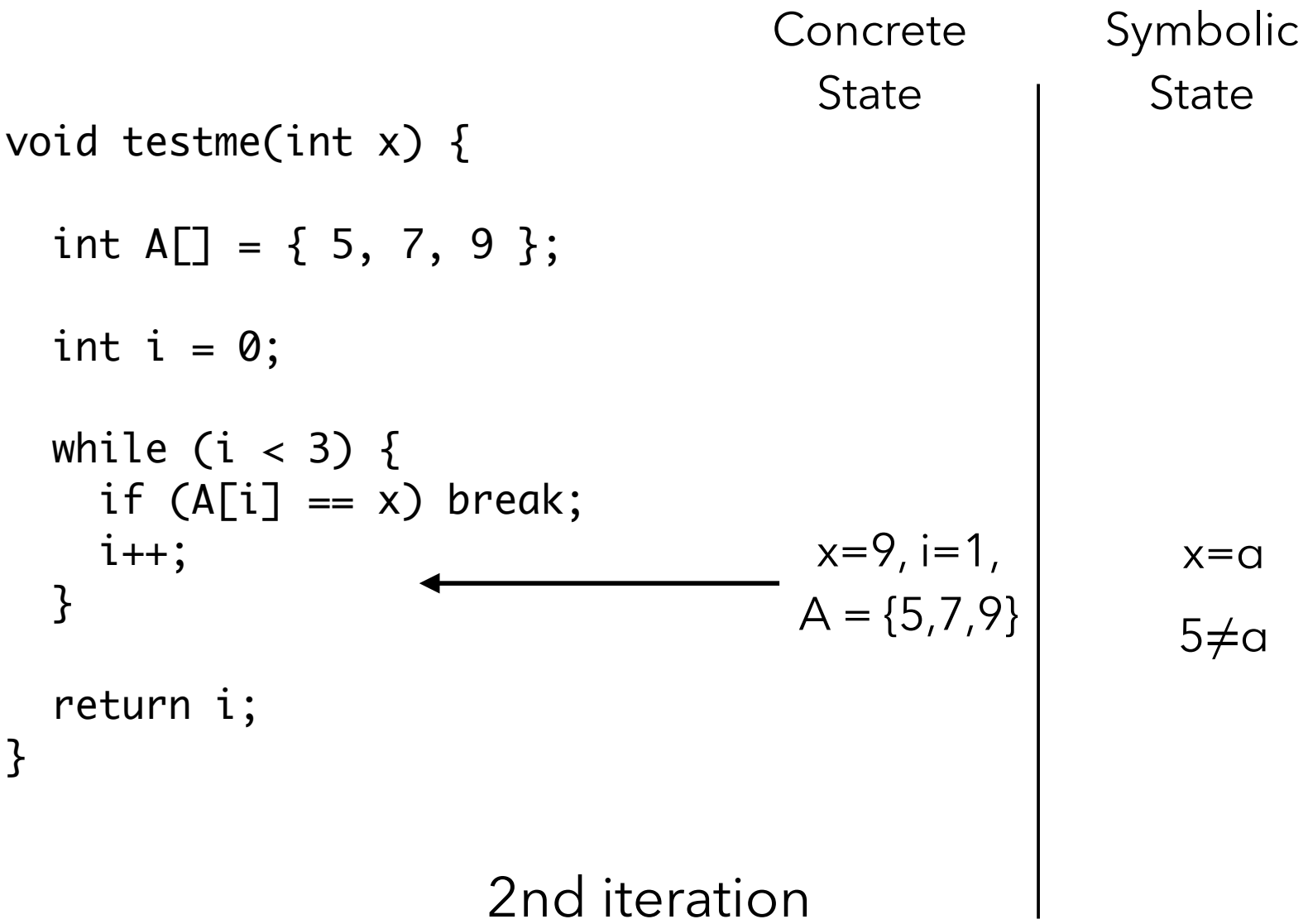
Example: Concolic Testing of Loops (Cont'd)



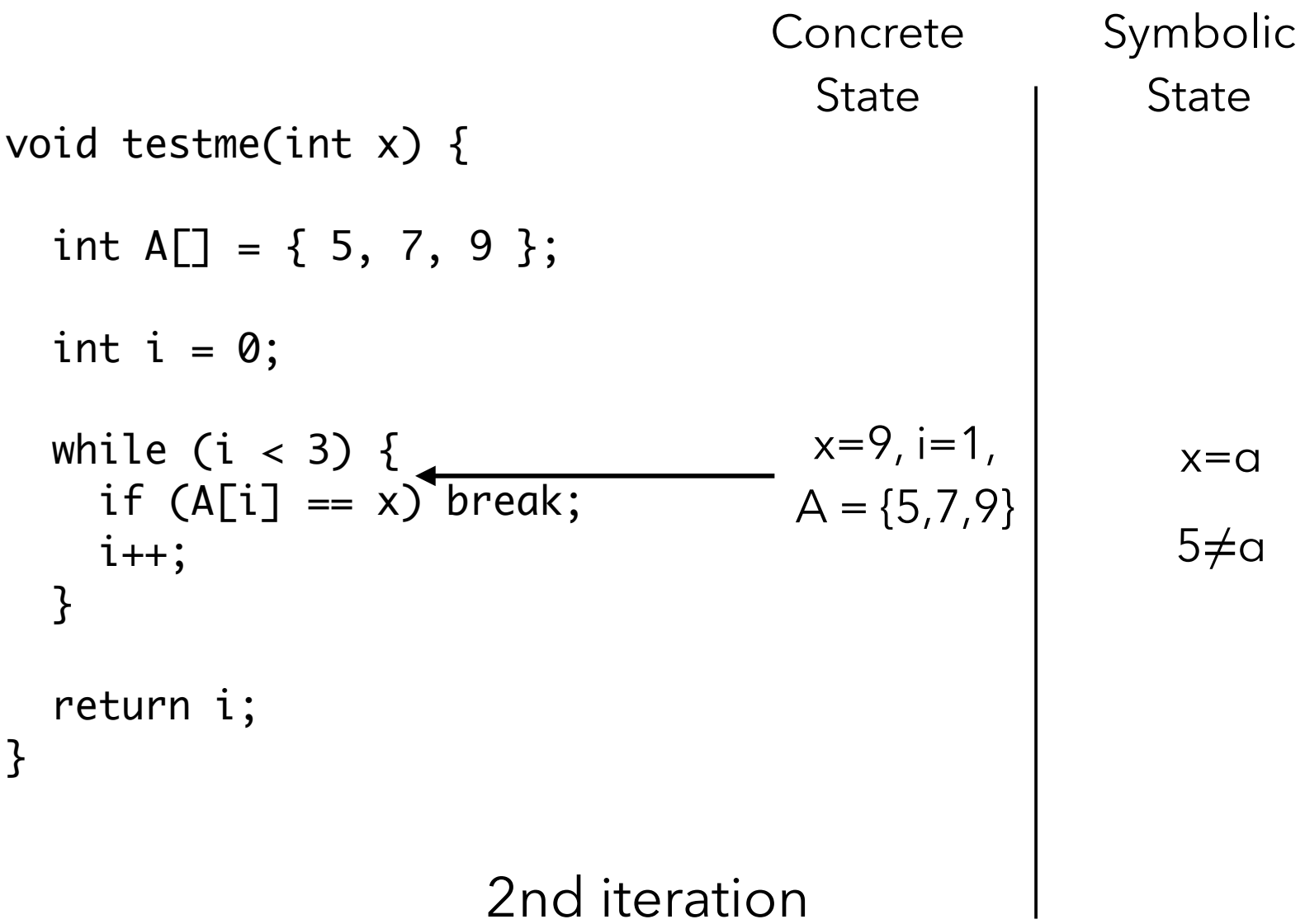
Example: Concolic Testing of Loops (Cont'd)



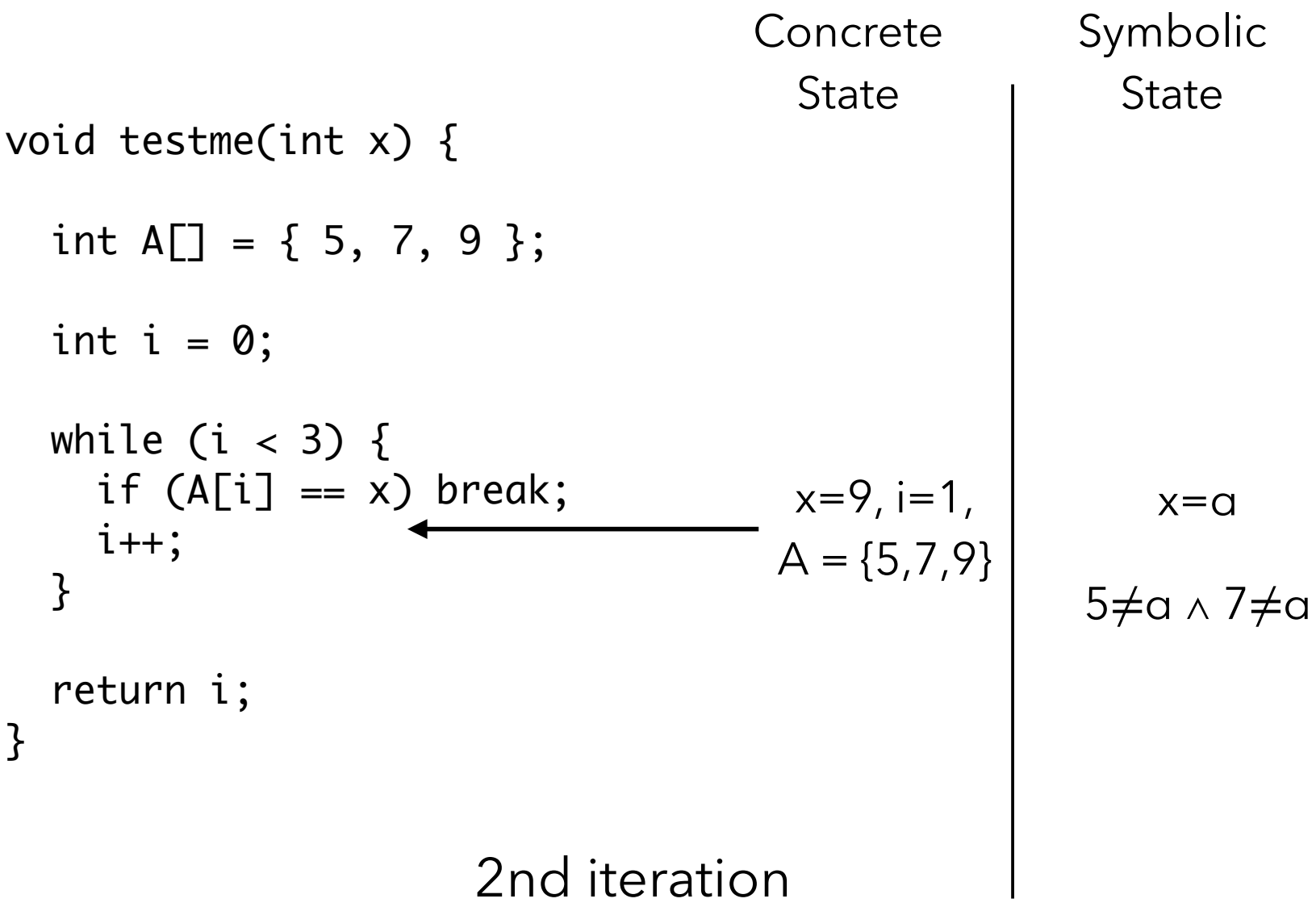
Example: Concolic Testing of Loops (Cont'd)



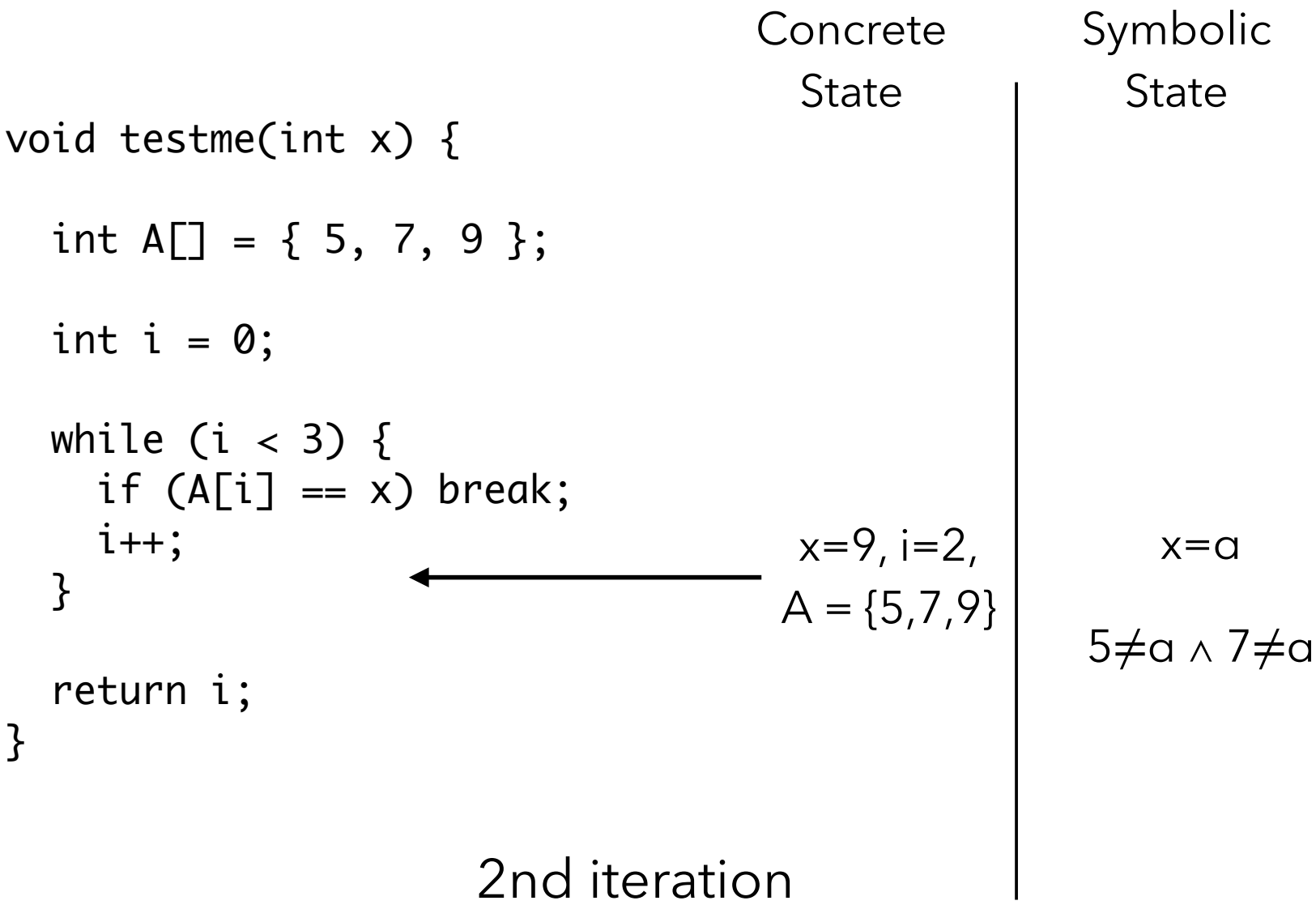
Example: Concolic Testing of Loops (Cont'd)



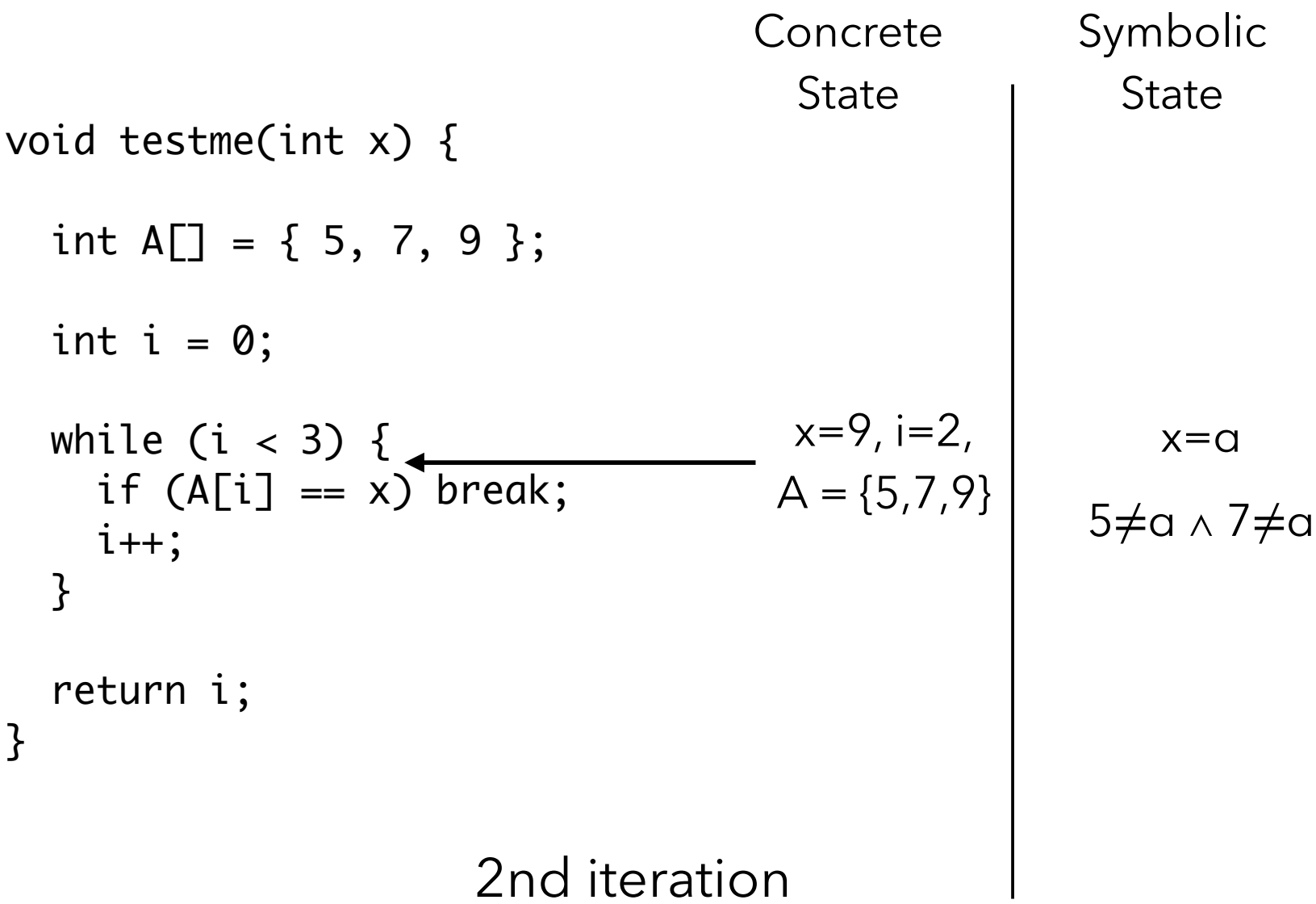
Example: Concolic Testing of Loops (Cont'd)



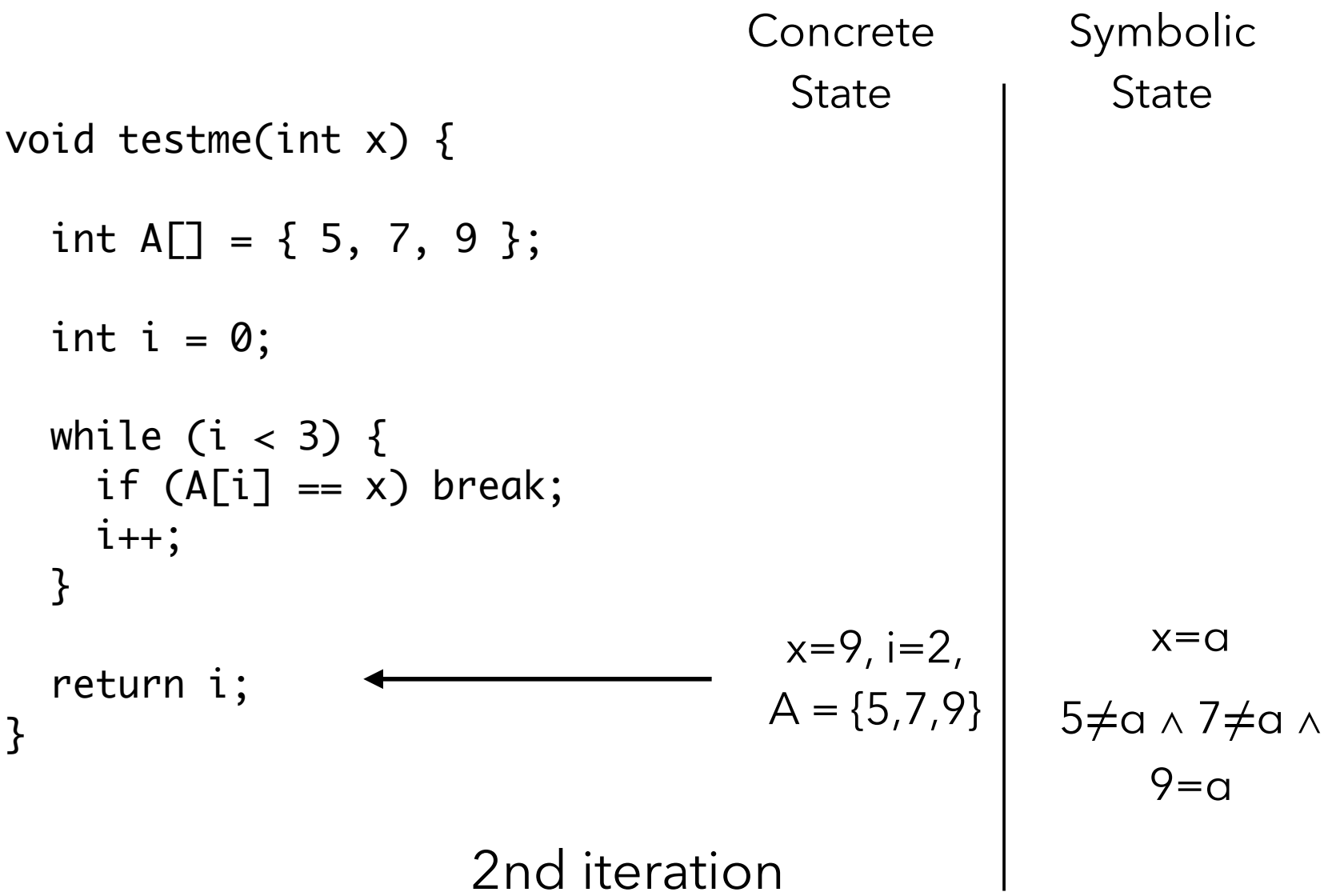
Example: Concolic Testing of Loops (Cont'd)



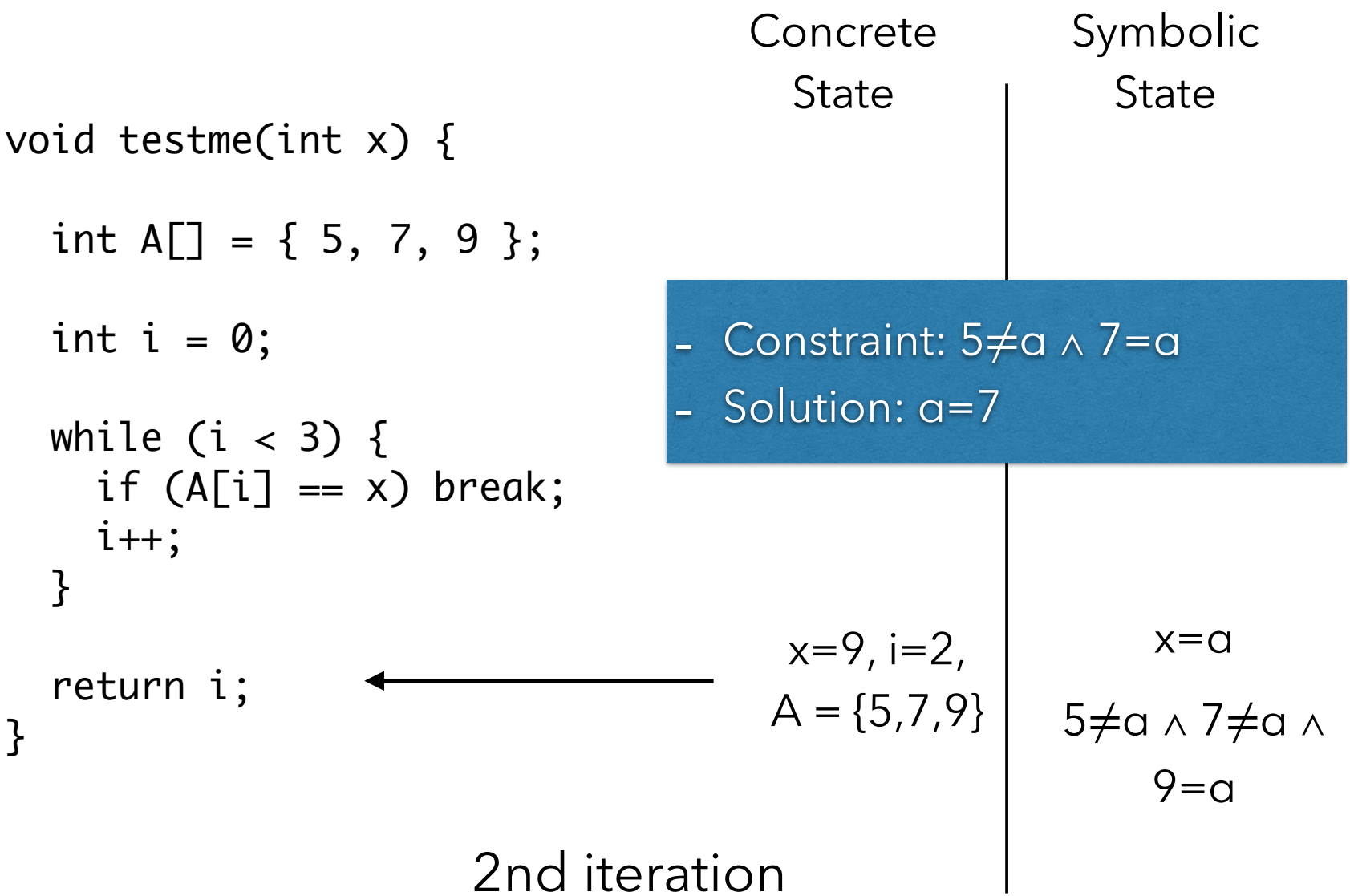
Example: Concolic Testing of Loops (Cont'd)



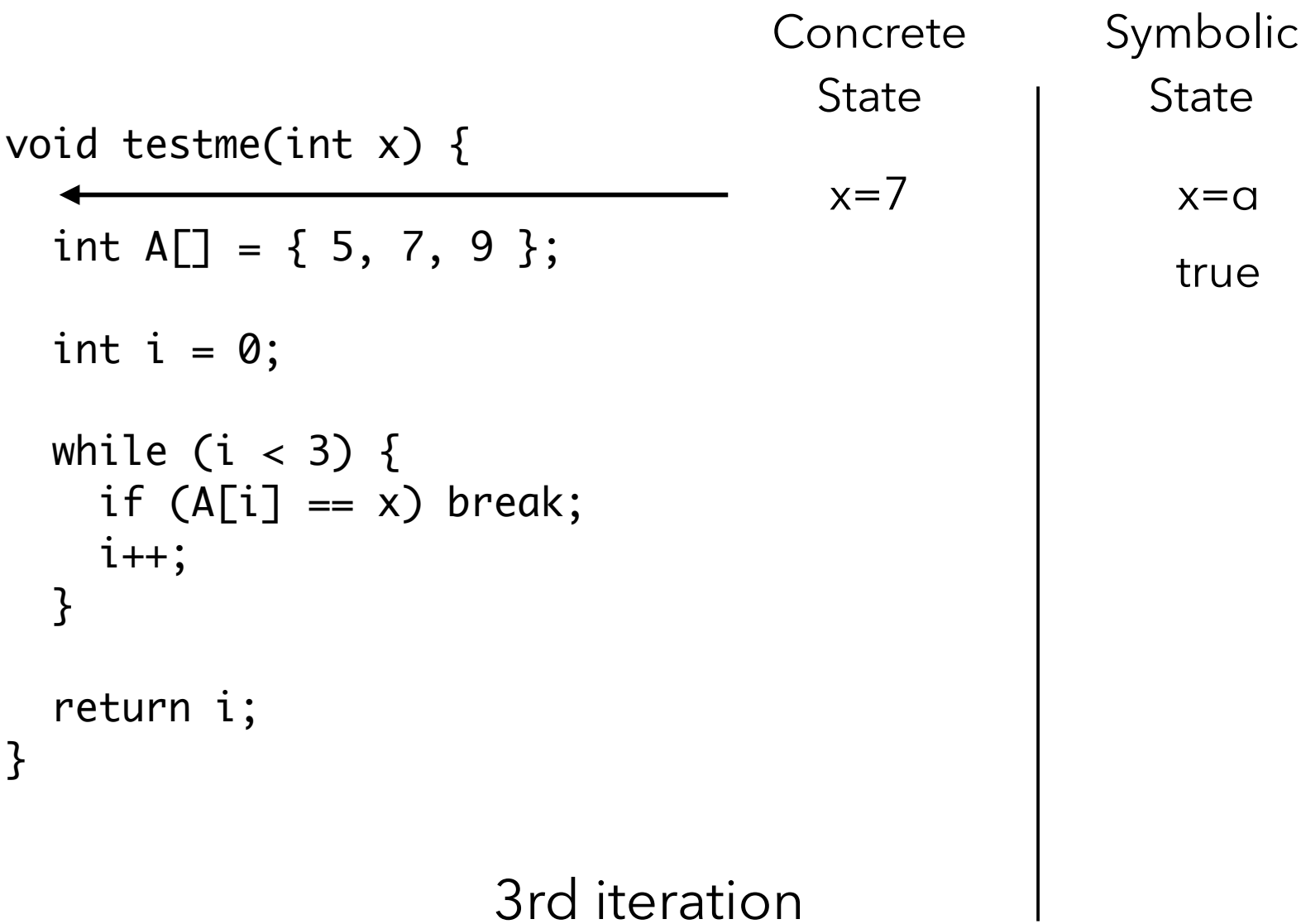
Example: Concolic Testing of Loops (Cont'd)



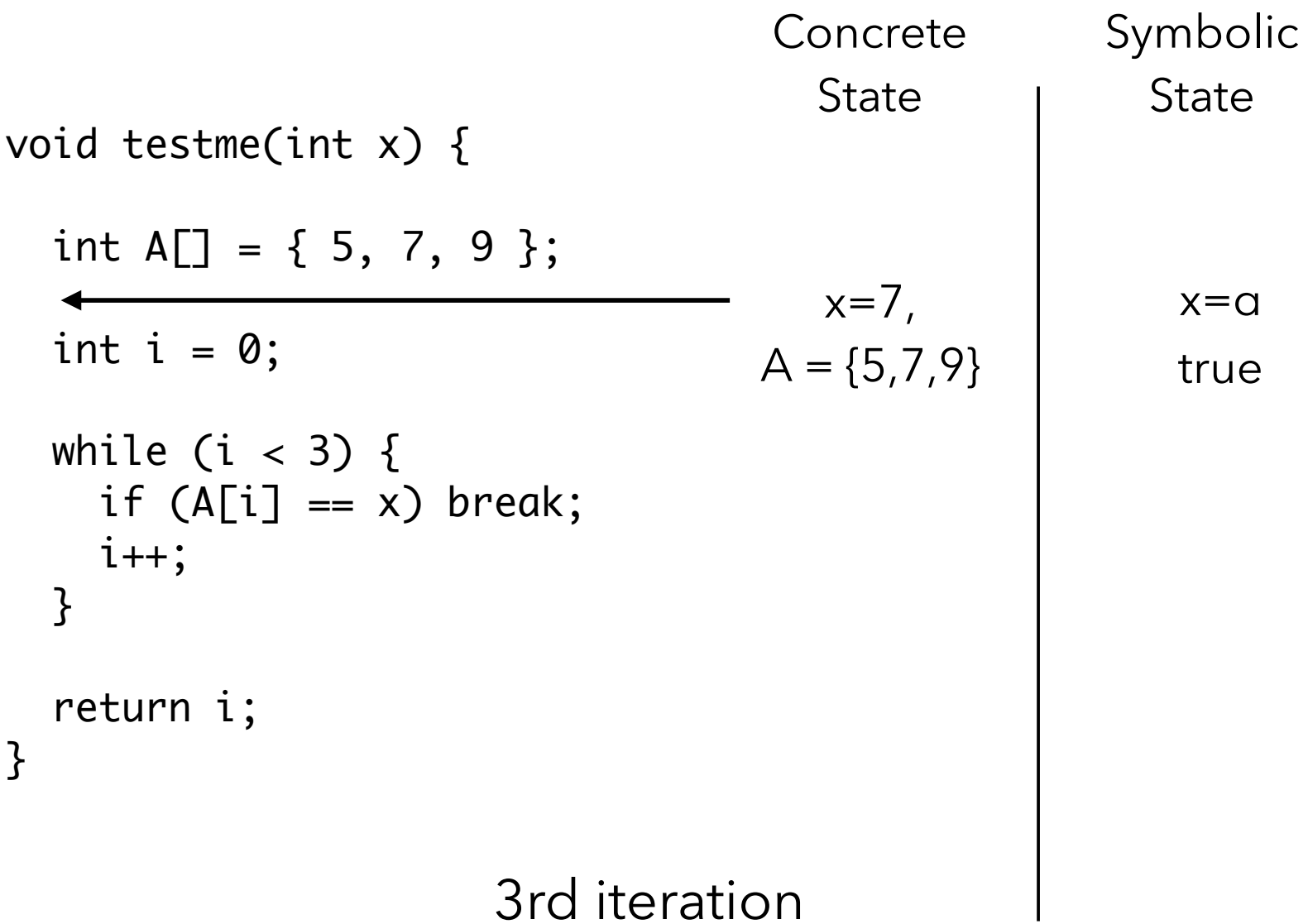
Example: Concolic Testing of Loops (Cont'd)



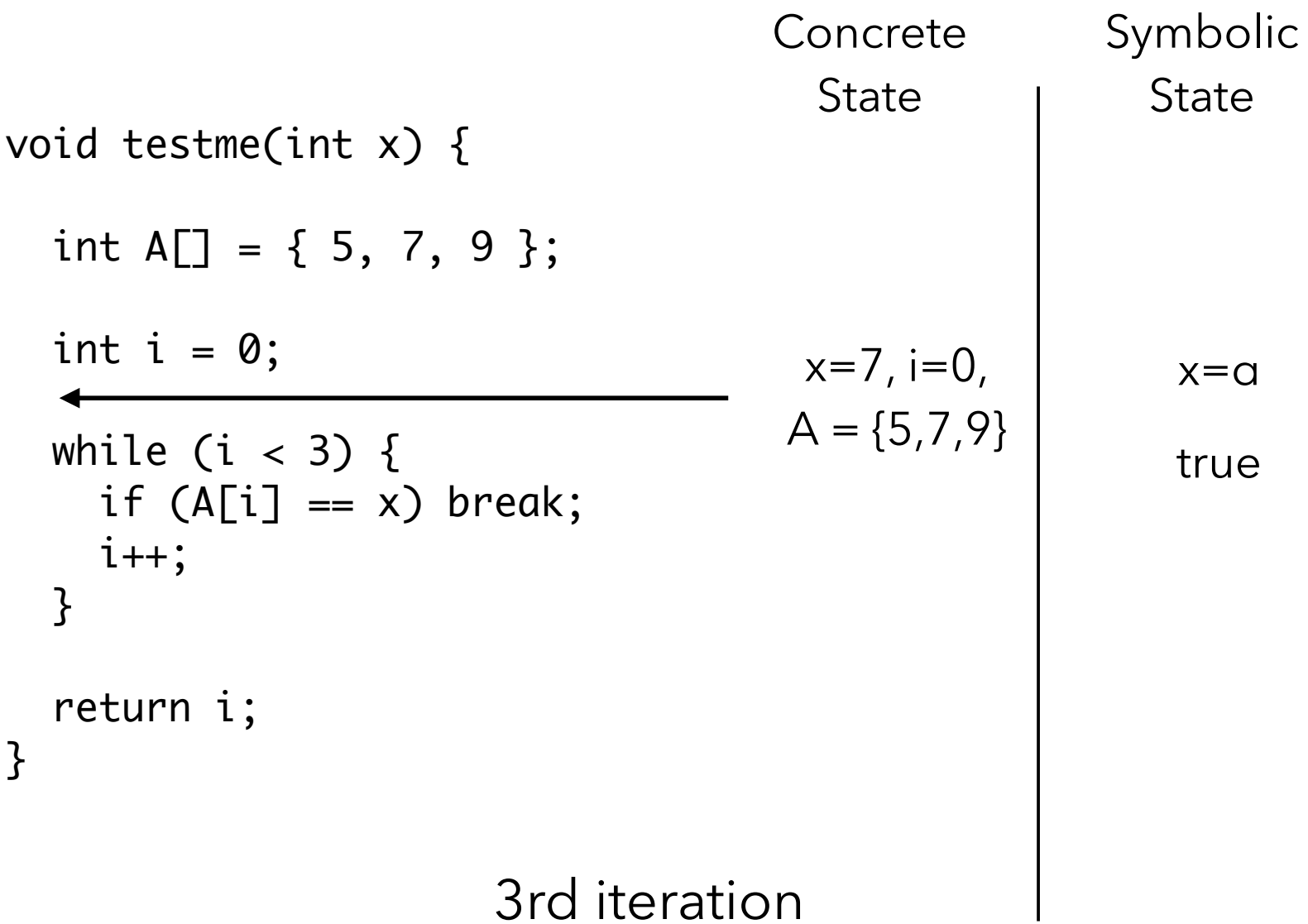
Example: Concolic Testing of Loops (Cont'd)



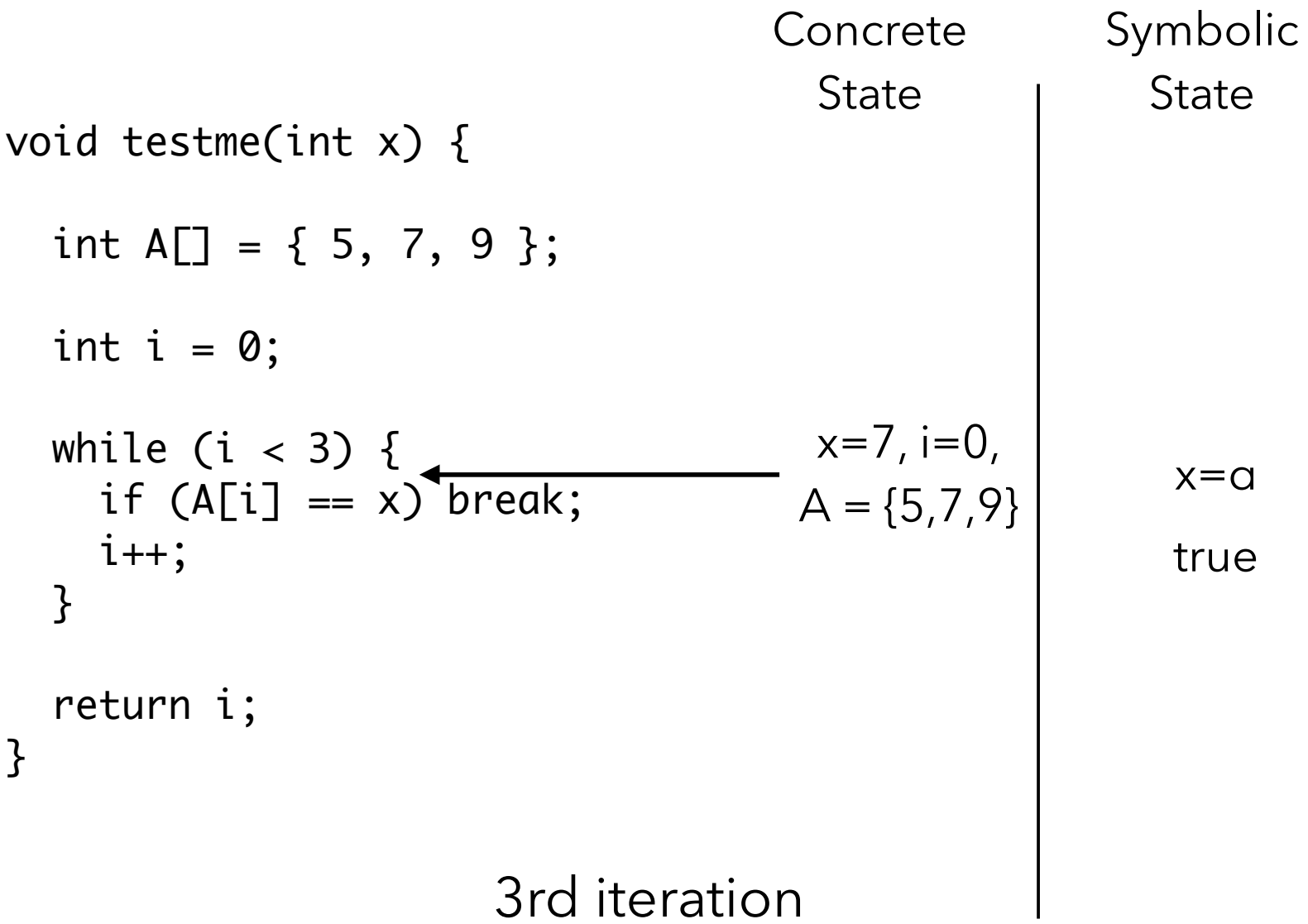
Example: Concolic Testing of Loops (Cont'd)



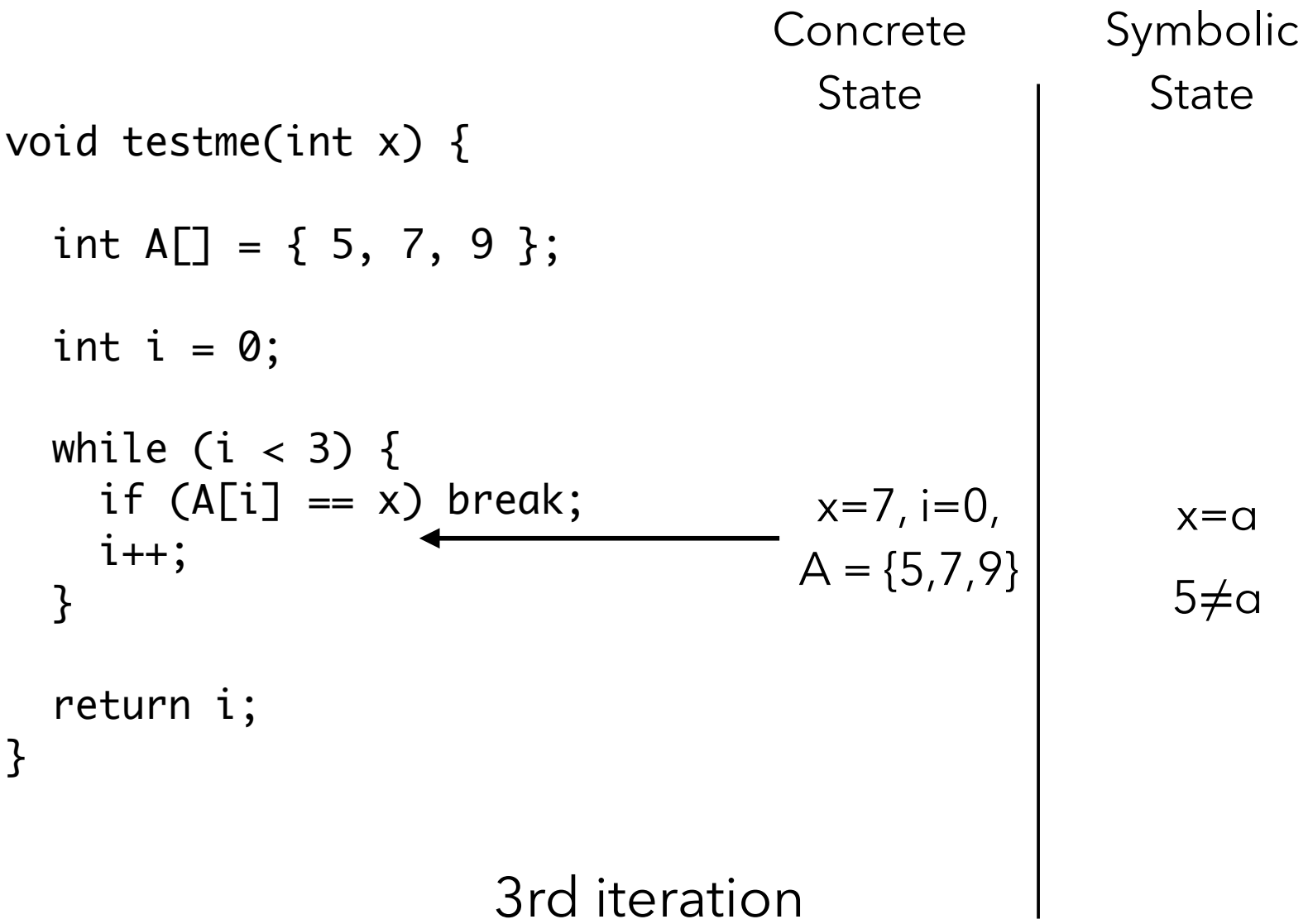
Example: Concolic Testing of Loops (Cont'd)



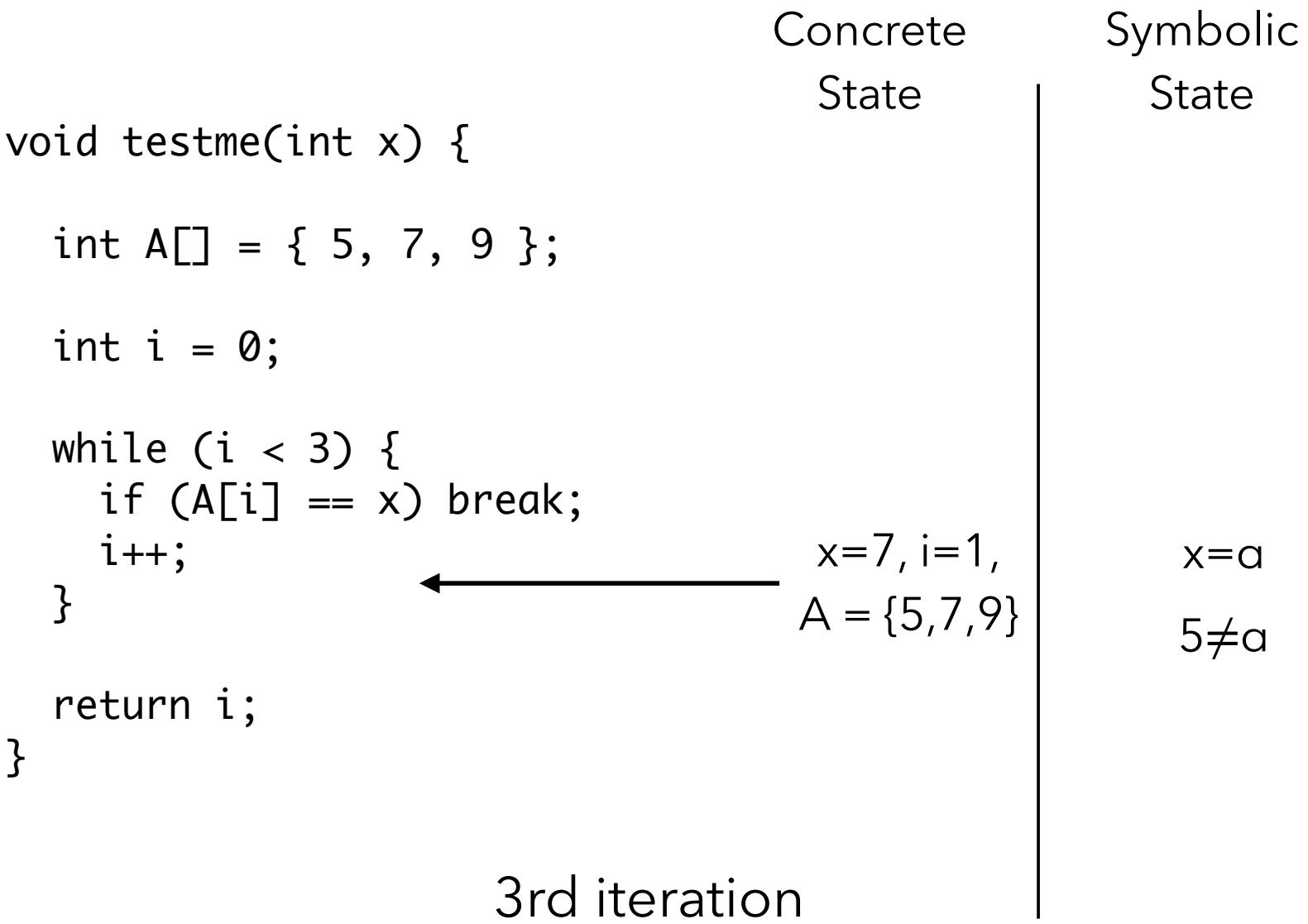
Example: Concolic Testing of Loops (Cont'd)



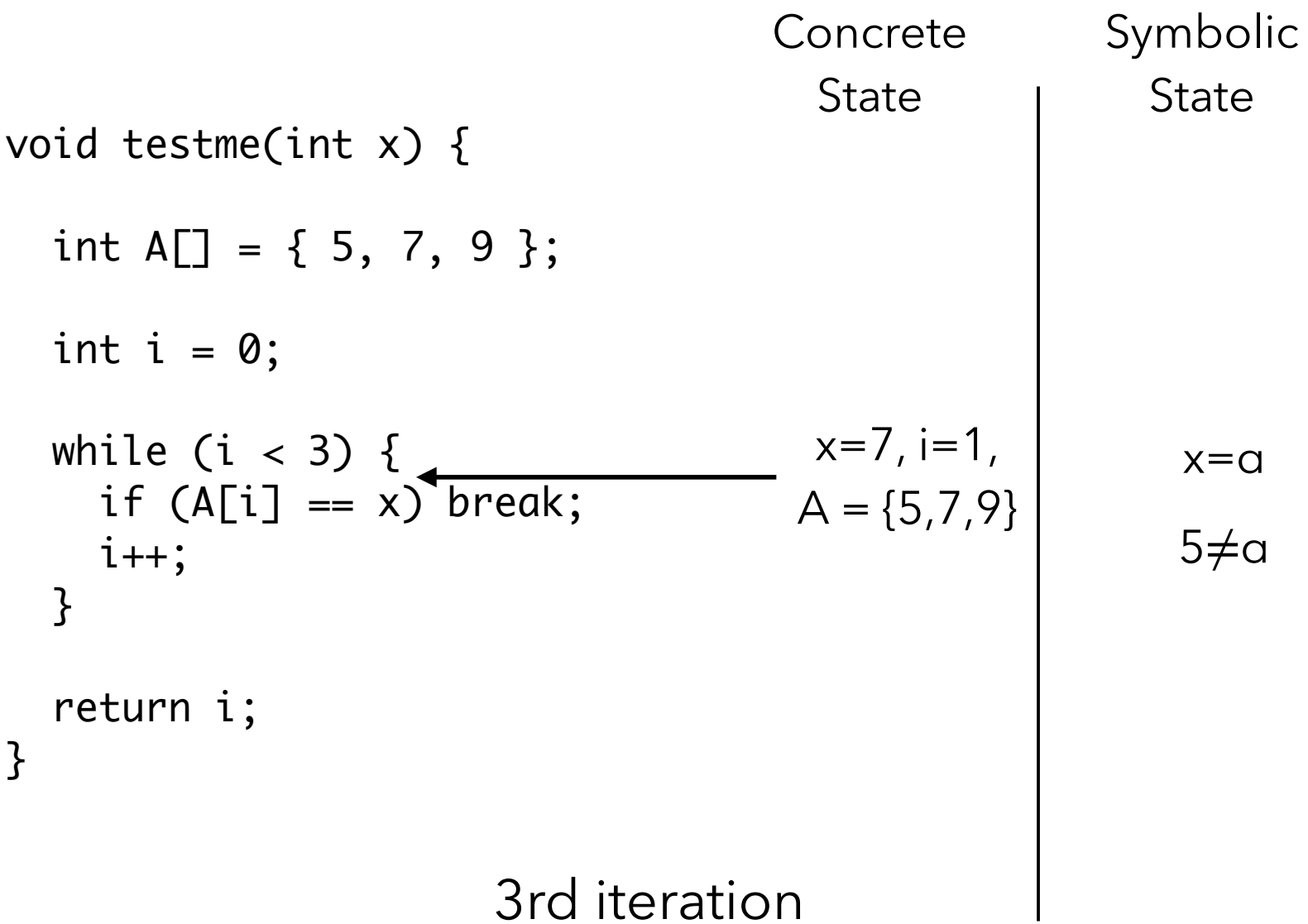
Example: Concolic Testing of Loops (Cont'd)



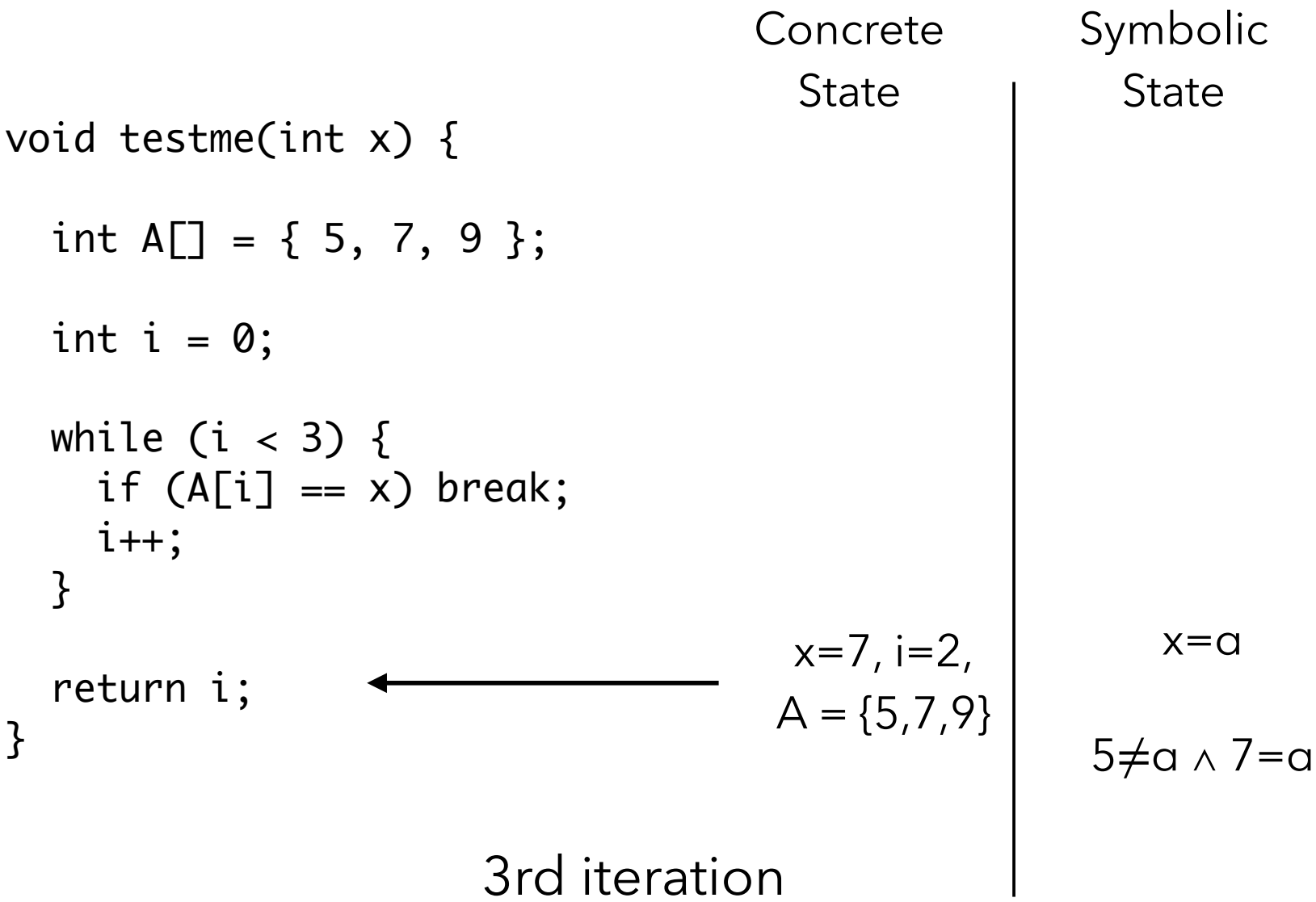
Example: Concolic Testing of Loops (Cont'd)



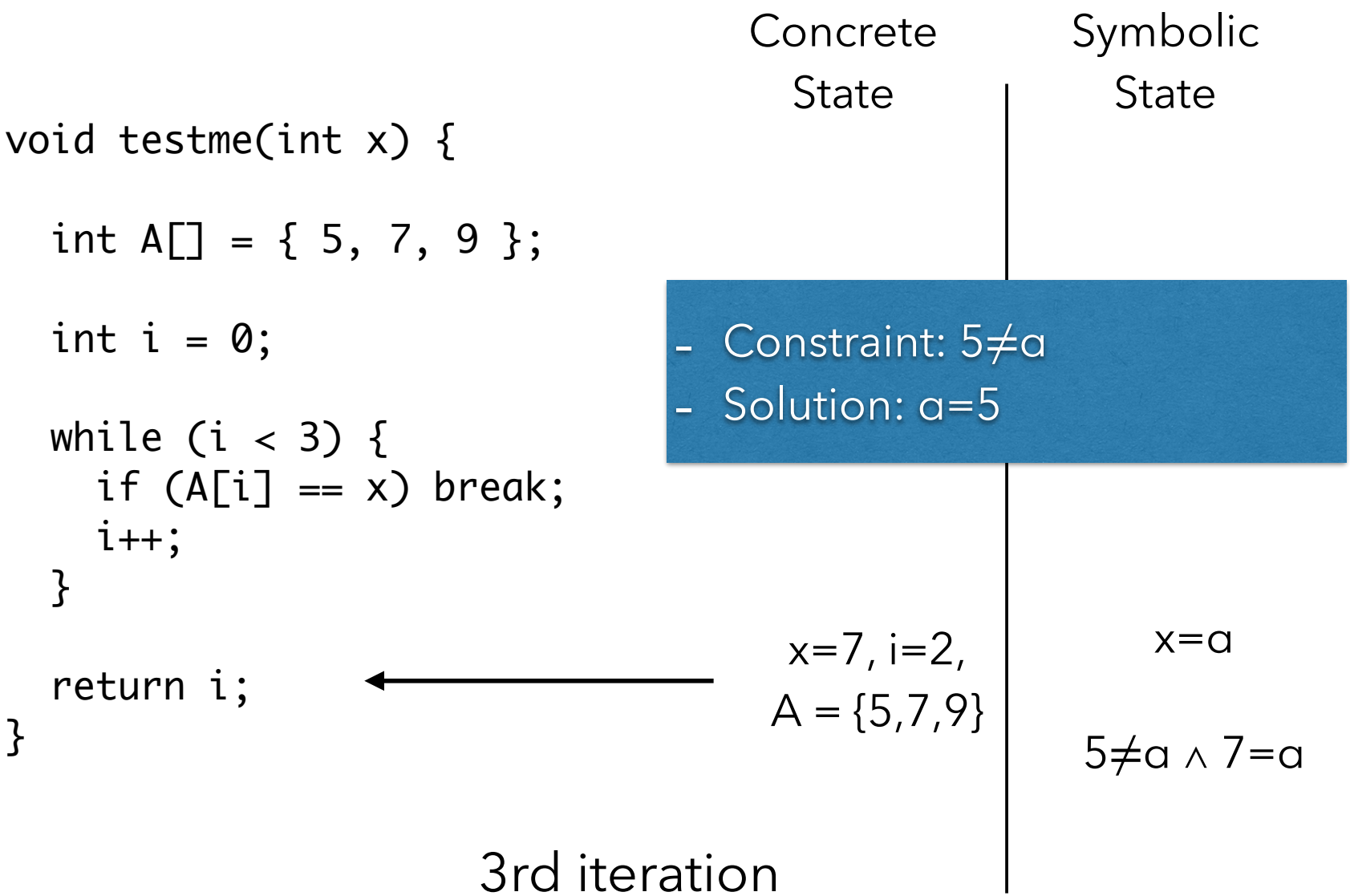
Example: Concolic Testing of Loops (Cont'd)



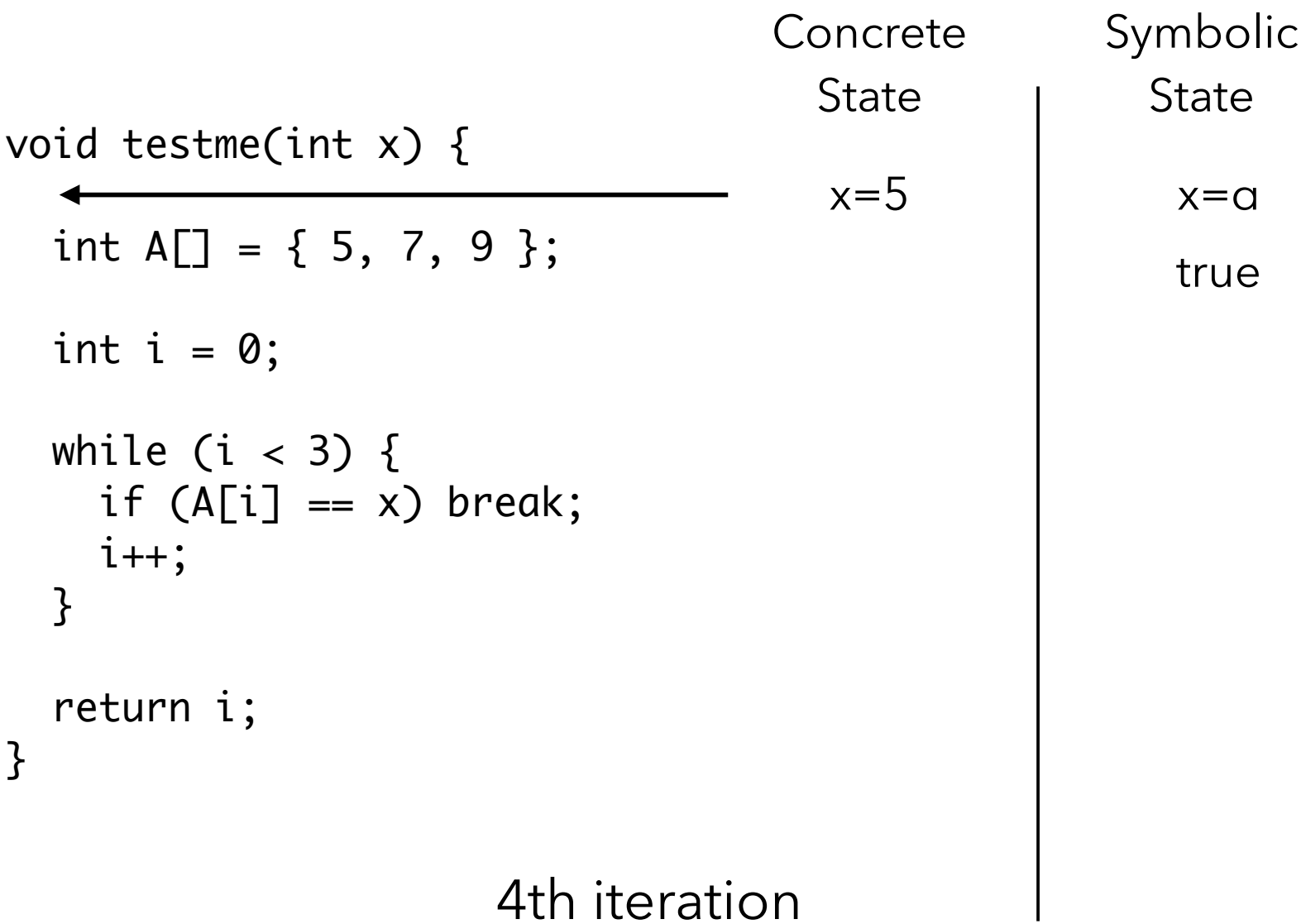
Example: Concolic Testing of Loops (Cont'd)



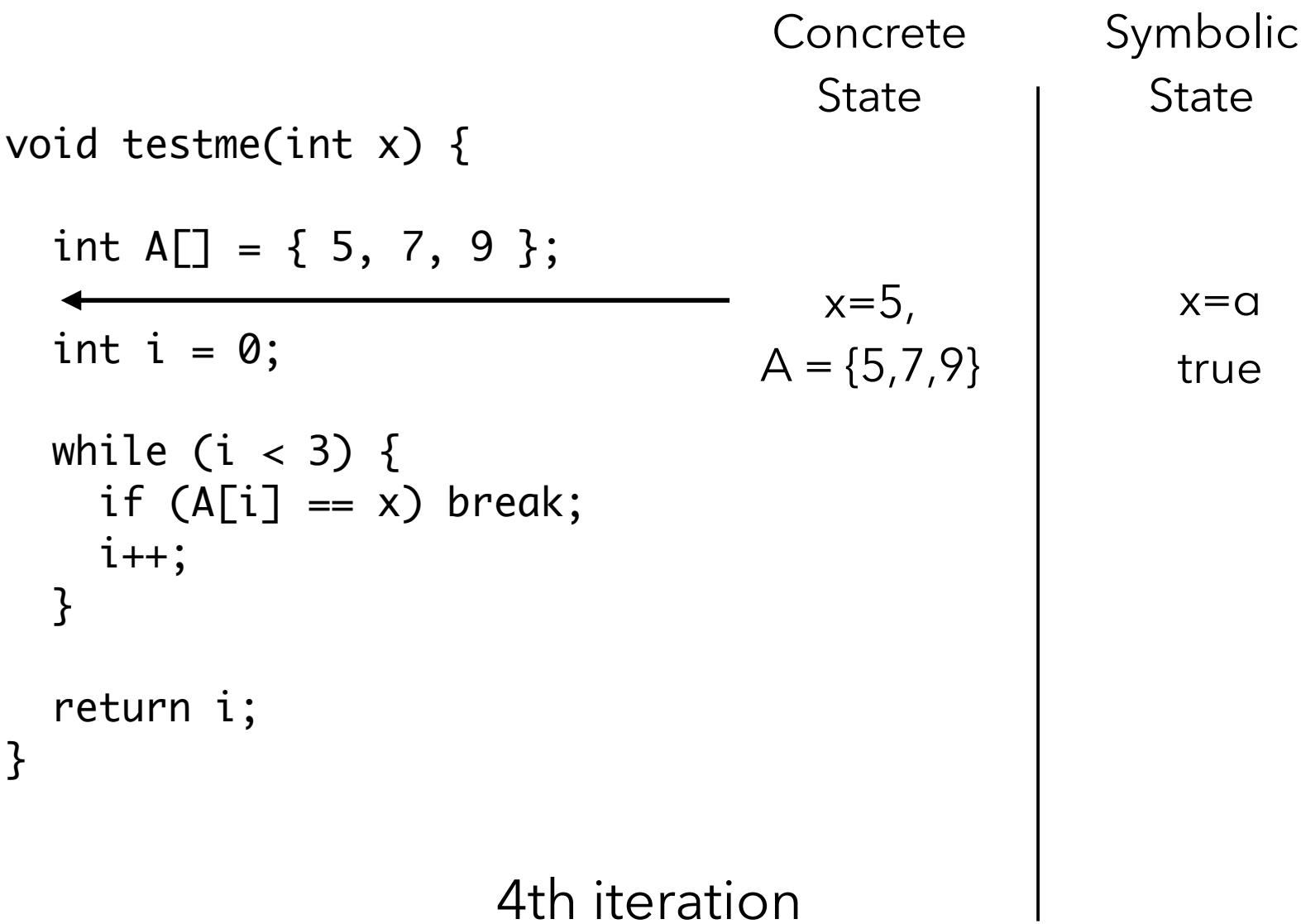
Example: Concolic Testing of Loops (Cont'd)



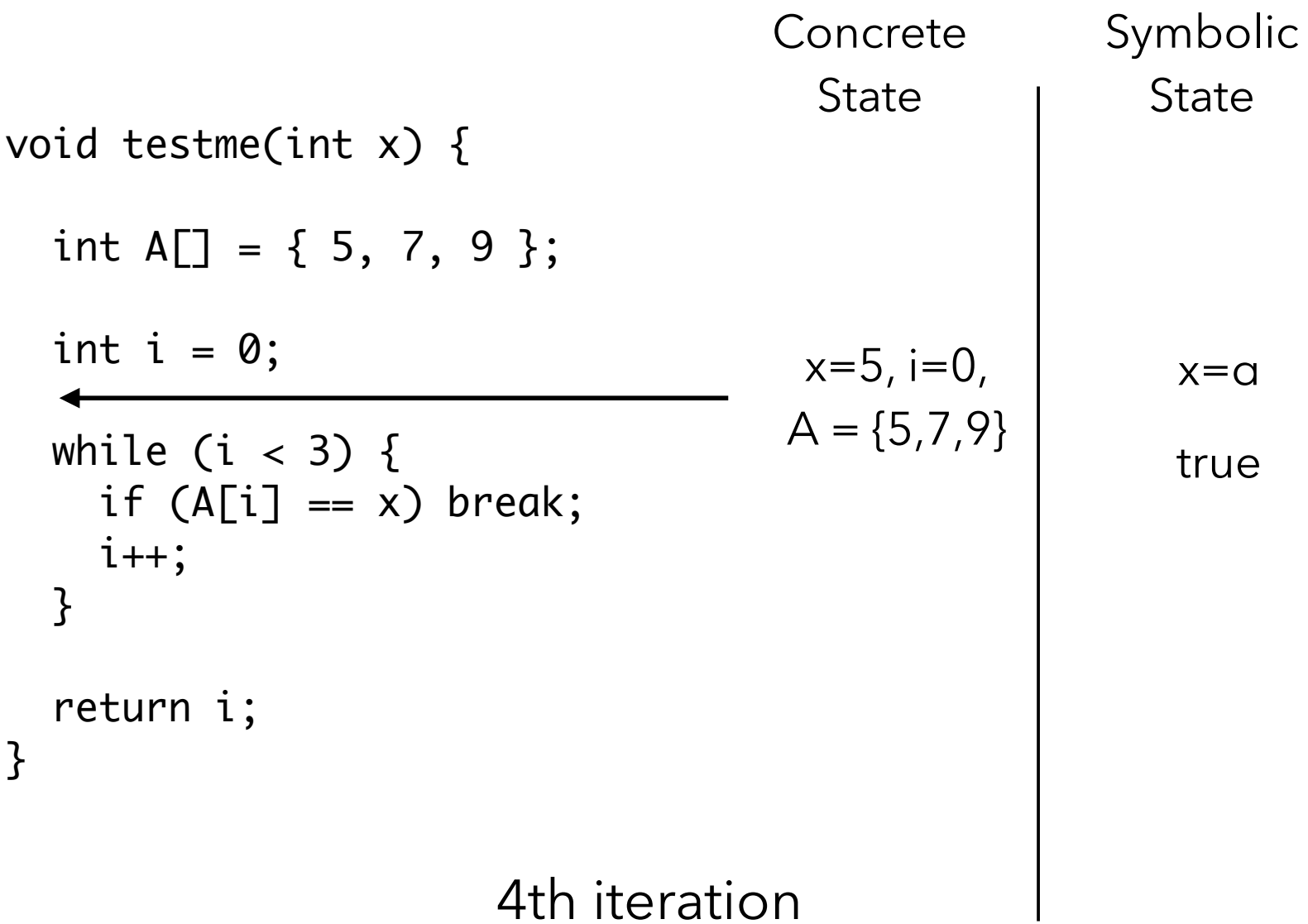
Example: Concolic Testing of Loops (Cont'd)



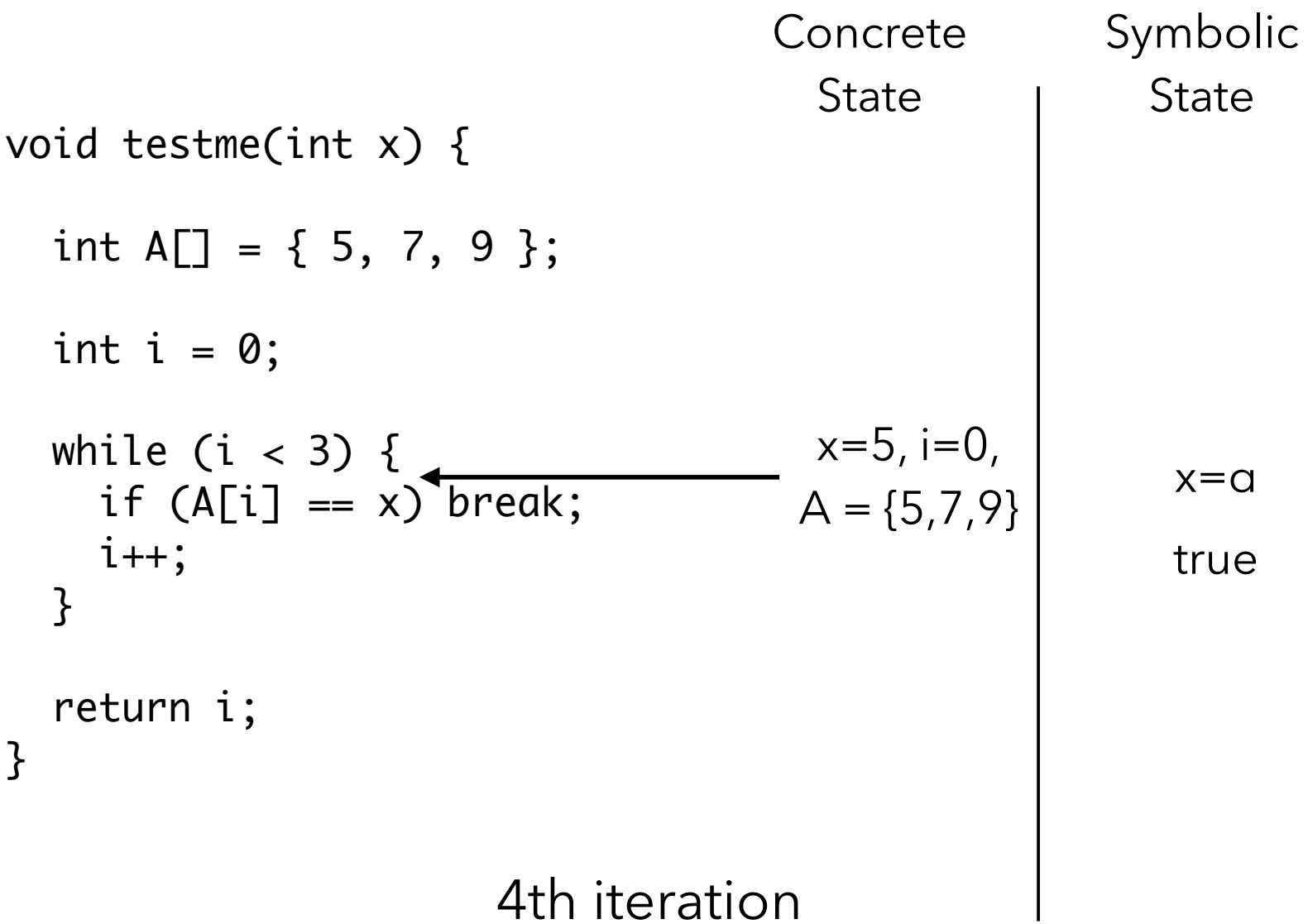
Example: Concolic Testing of Loops (Cont'd)



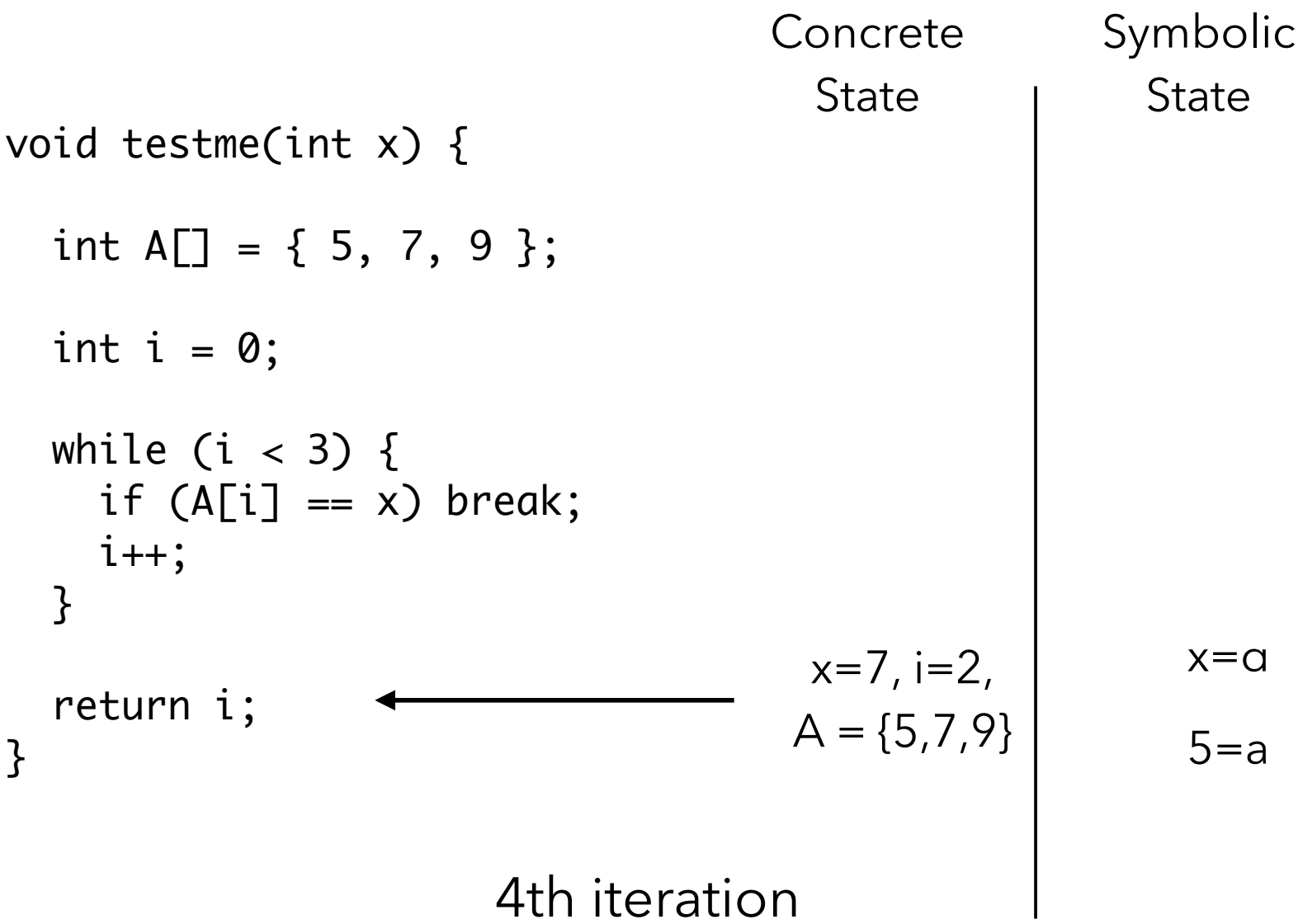
Example: Concolic Testing of Loops (Cont'd)



Example: Concolic Testing of Loops (Cont'd)



Example: Concolic Testing of Loops (Cont'd)



Takeaways

- In symbolic states, we do not need to keep track of constraints that do not have data-dependency on inputs.
  - Why? To decide the next branch points to explore, it is enough to solve path constraints that are data-dependant on inputs.
- Negating path conditions does not always increase code coverage.
  - In the previous example, concolic testing terminates after four iterations but we executed all branches in the second iteration.

Example: Concolic Testing of 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 p=NULL	x=α, p=β true

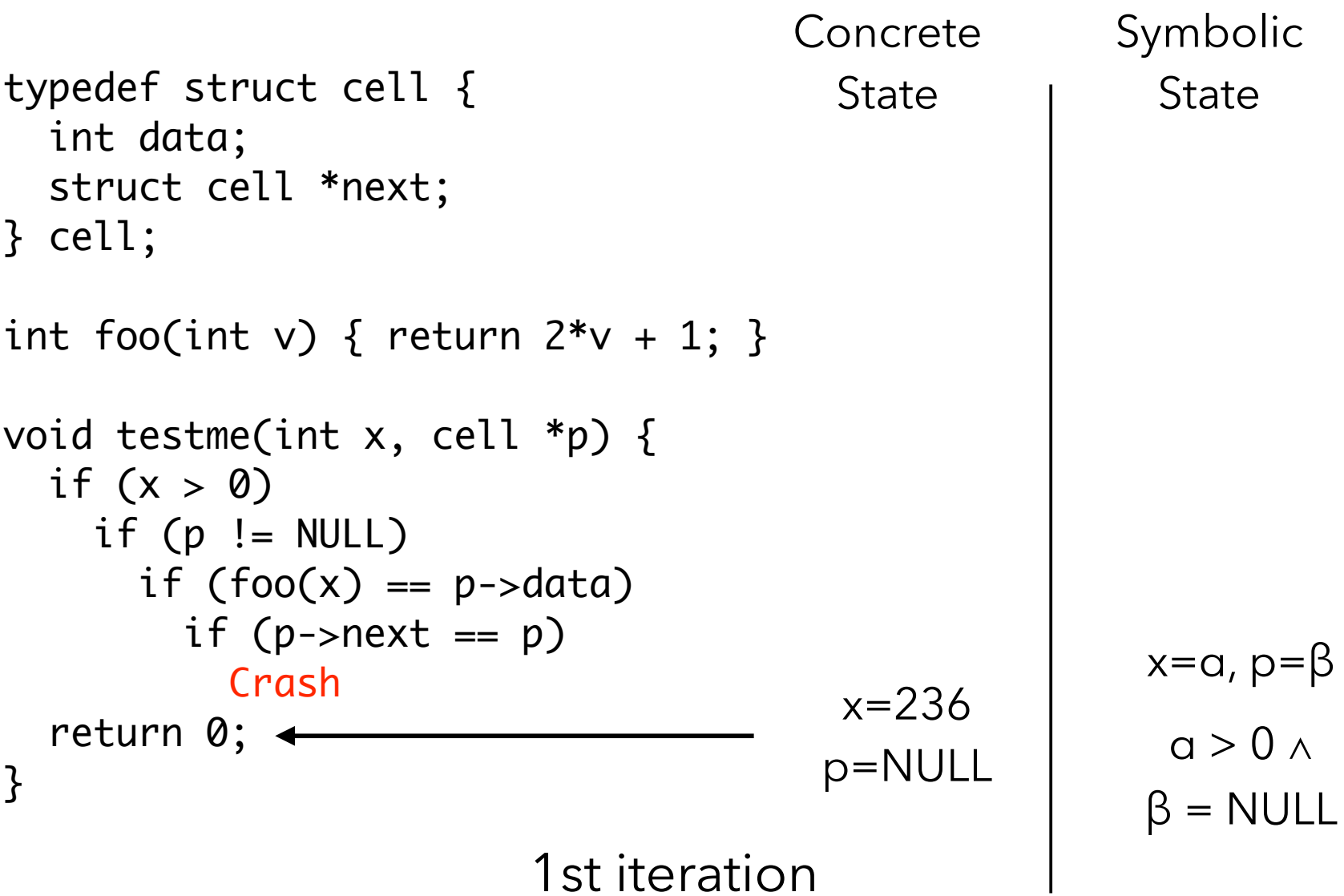
1st iteration

Example: Concolic Testing of Data Structures (Cont'd)

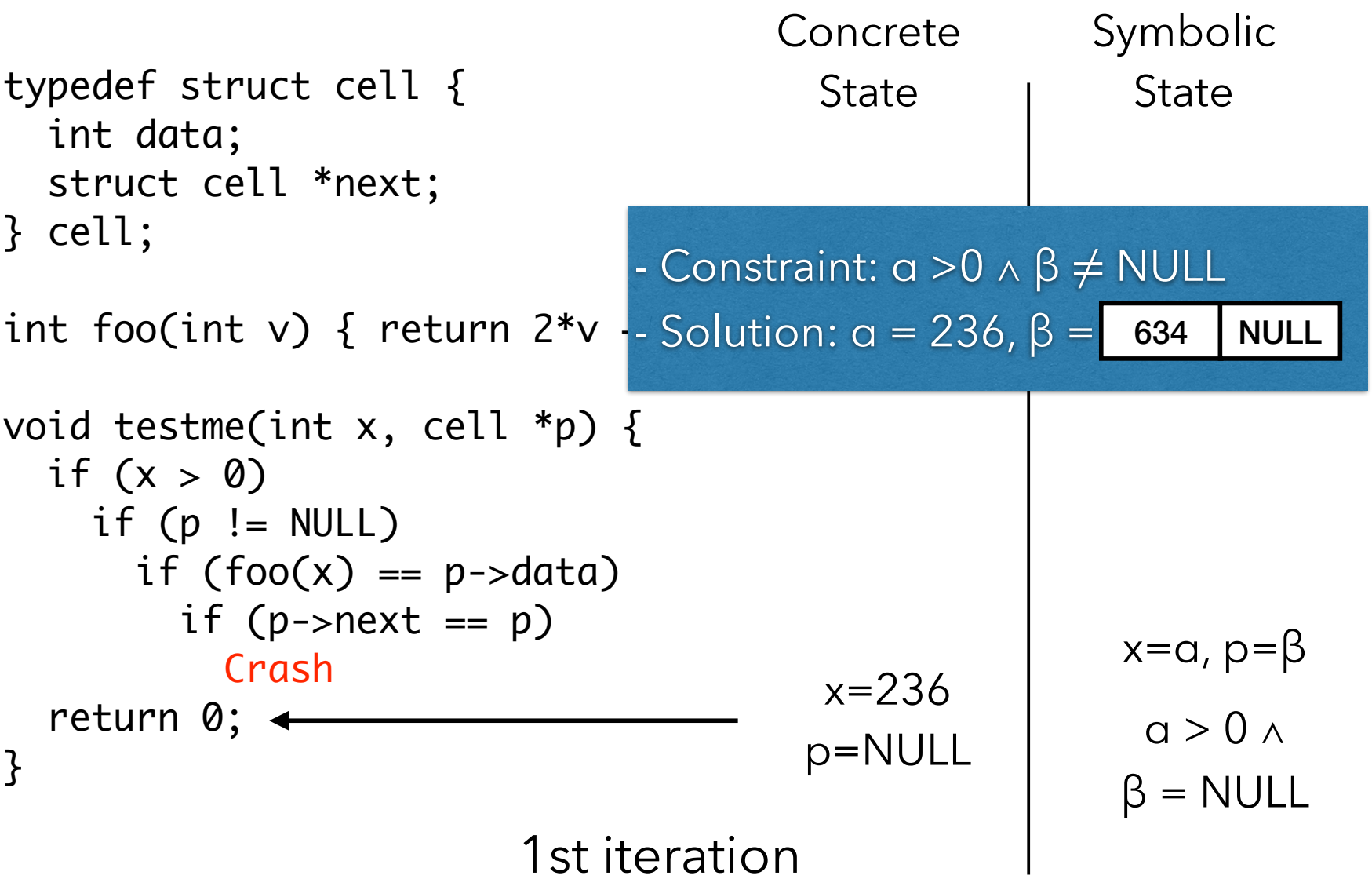
```
typedef struct cell {
  int data;
  struct cell *next;
} cell;

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
1st iteration	x=236 p=NULL	x=α, p=β α > 0

Example: Concolic Testing of Data Structures (Cont'd)

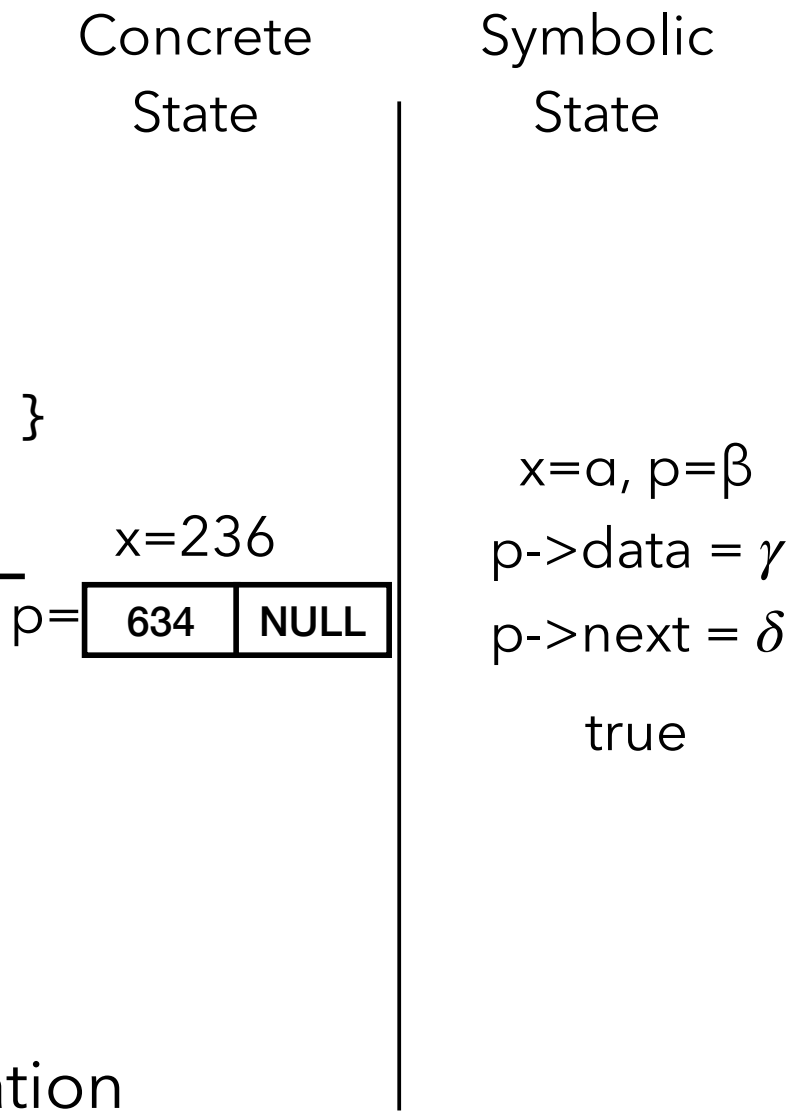


Example: Concolic Testing of Data Structures (Cont'd)

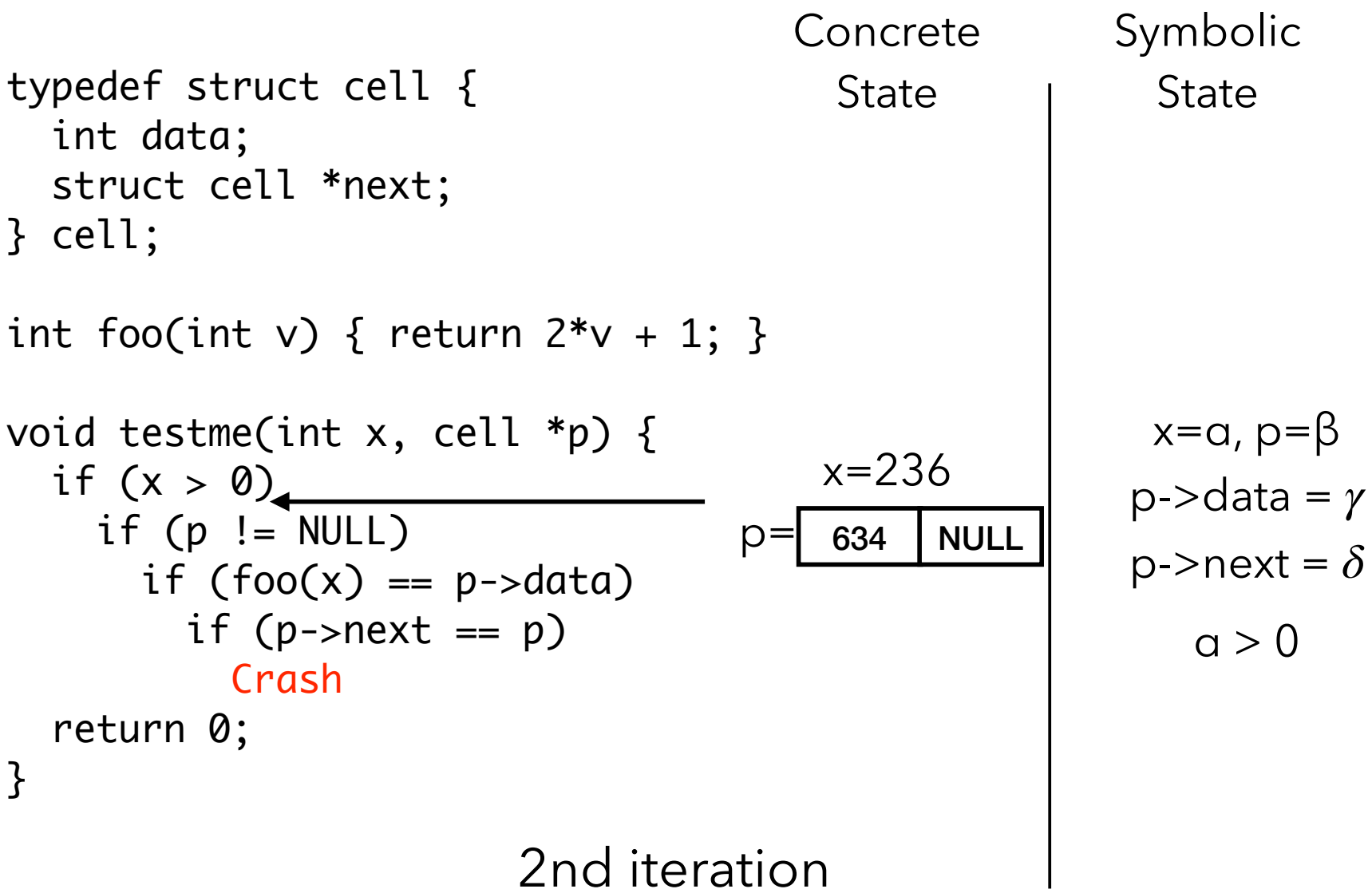


Example: Concolic Testing of Data Structures (Cont'd)

```
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;
}
```



Example: Concolic Testing of Data Structures (Cont'd)

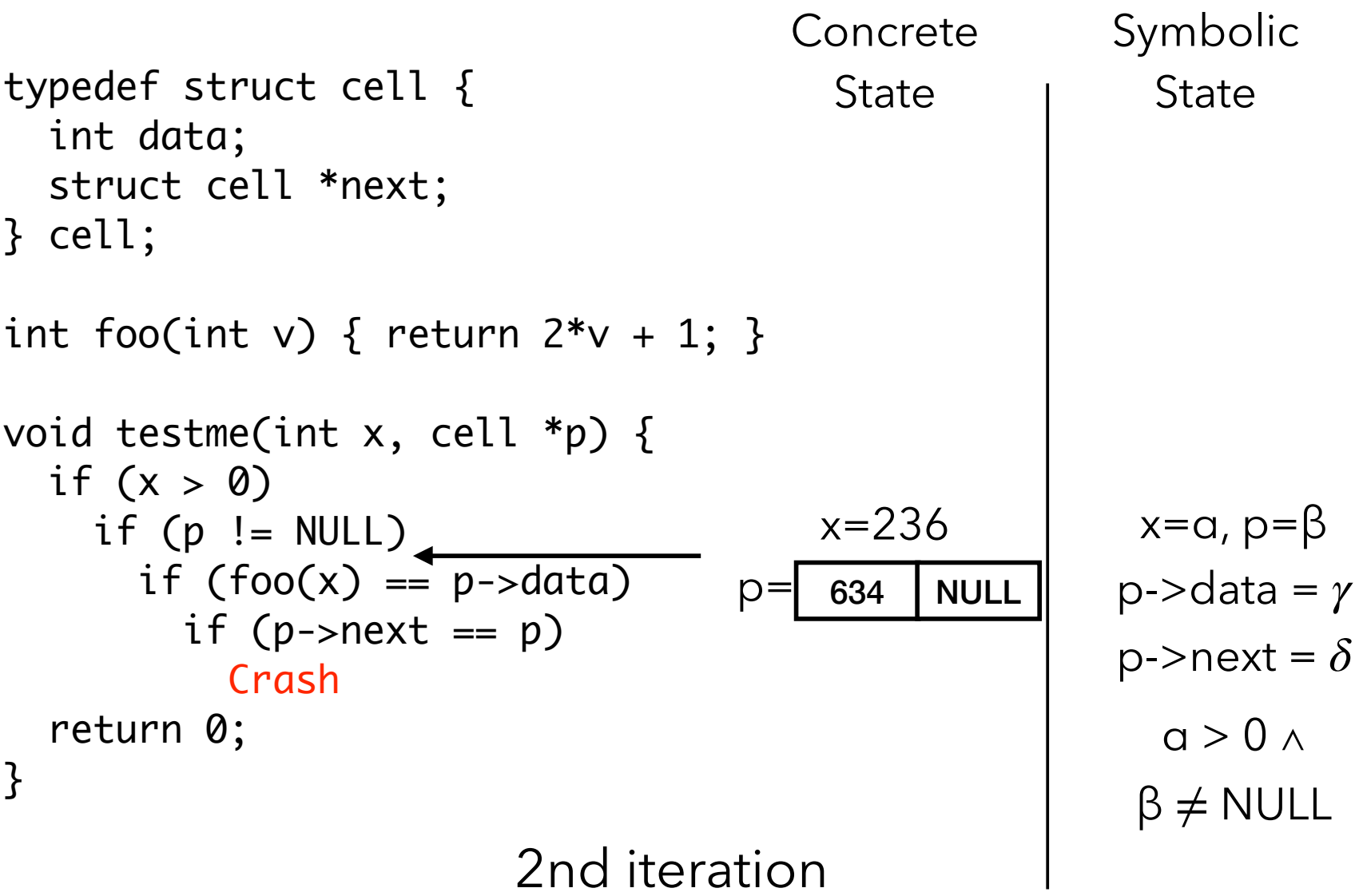


Example: Concolic Testing of Data Structures (Cont'd)

```
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;
}
```

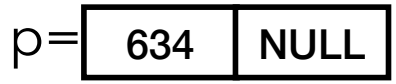
	Concrete State	Symbolic State
1st iteration	x=236 p=NULL	x=α, p=β α > 0

Example: Concolic Testing of Data Structures (Cont'd)



Example: Concolic Testing of Data Structures (Cont'd)

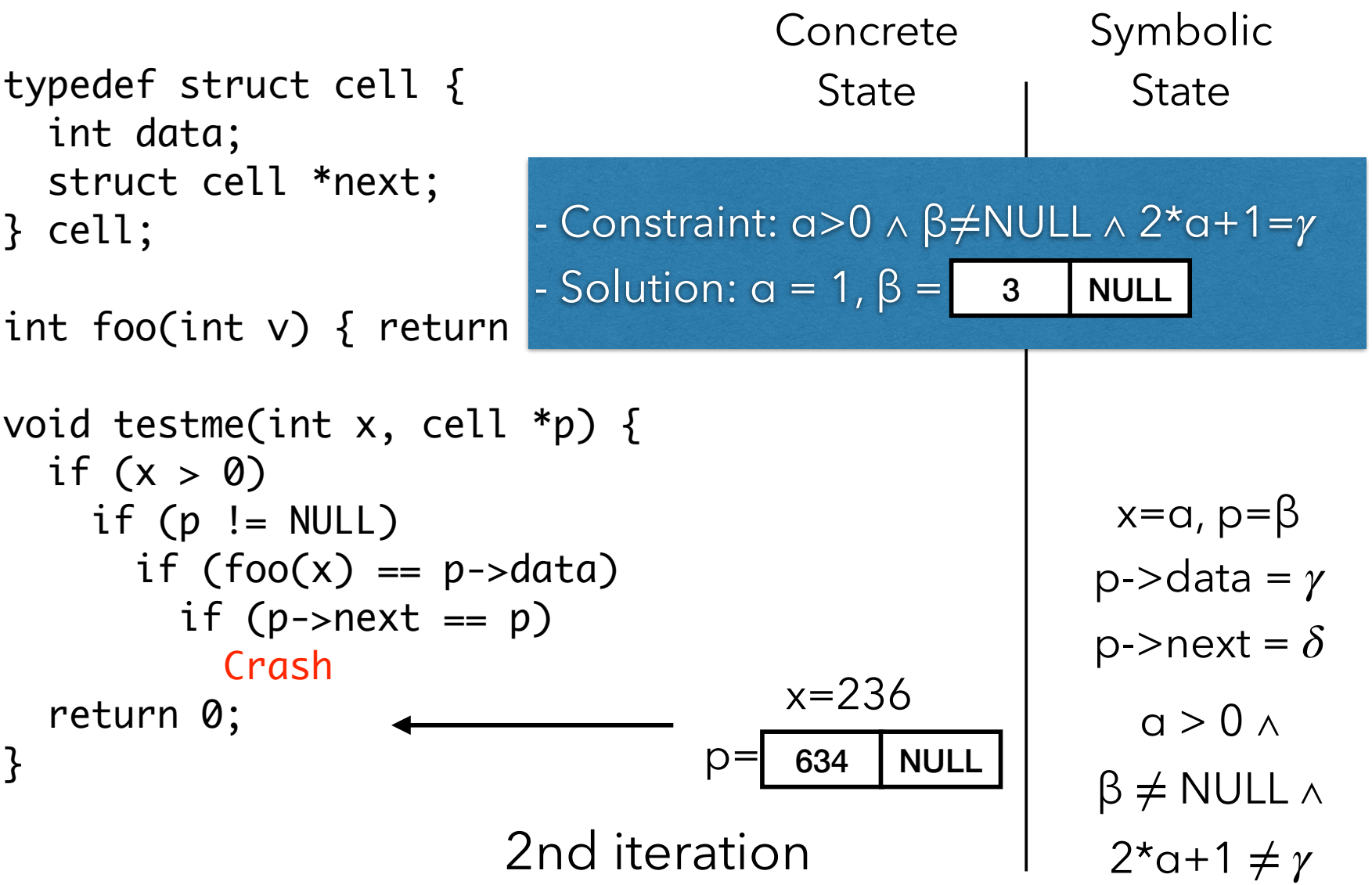
```
typedef struct cell {
  int data;
  struct cell *next;
} cell;

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;
}
```

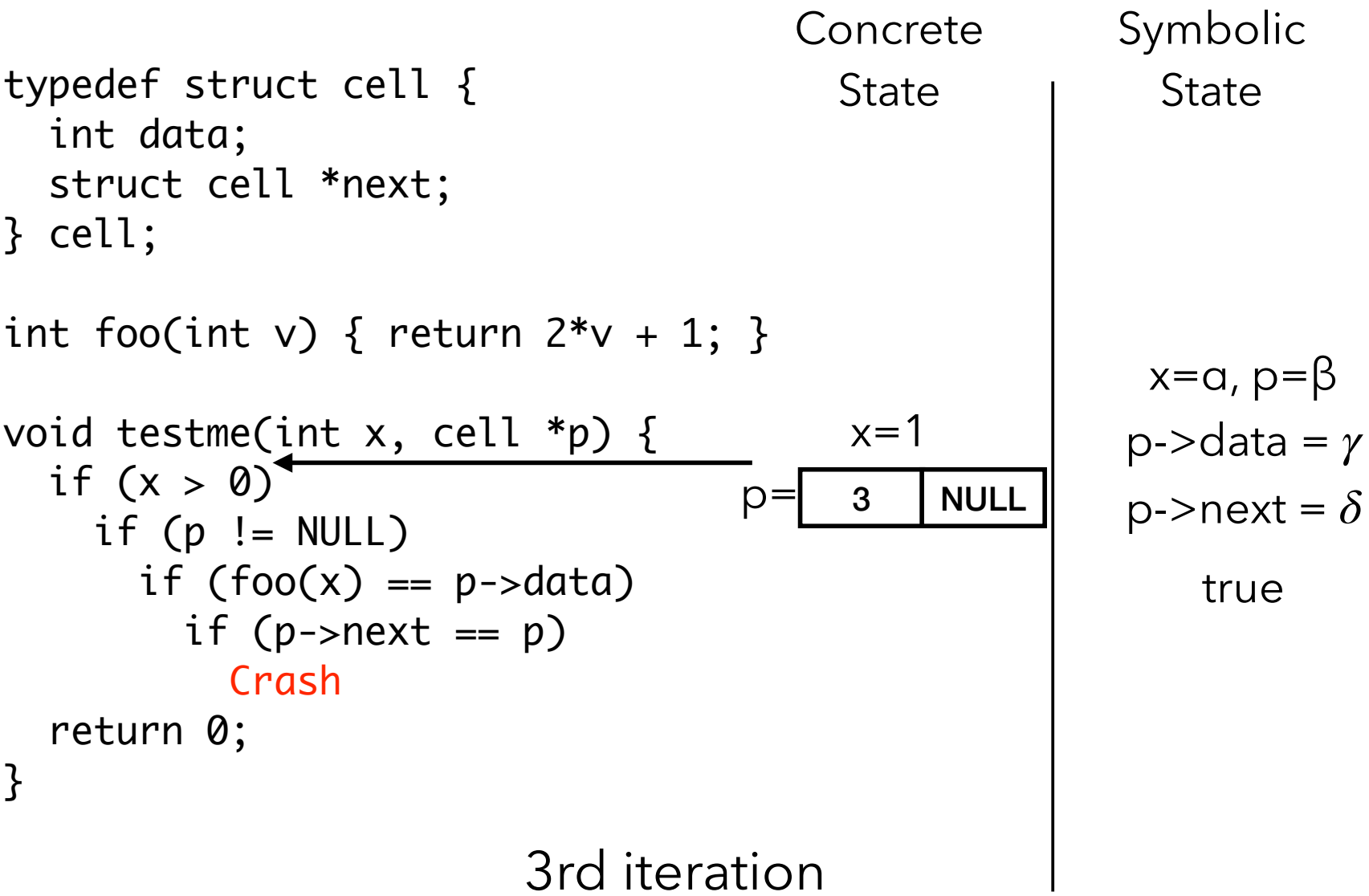
2nd iteration

Concrete State	Symbolic State
	x=α, p=β
	p->data = γ
	p->next = δ
	α > 0 ∧
	β ≠ NULL ∧
	2*α+1 ≠ γ

Example: Concolic Testing of Data Structures (Cont'd)



Example: Concolic Testing of Data Structures (Cont'd)

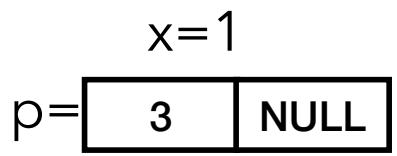


Example: Concolic Testing of Data Structures (Cont'd)

```
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 = γ

p->next = δ

α > 0 ∧

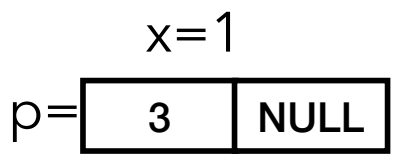
β ≠ NULL ∧

2*α+1 = γ

3rd iteration

Example: Concolic Testing of Data Structures (Cont'd)

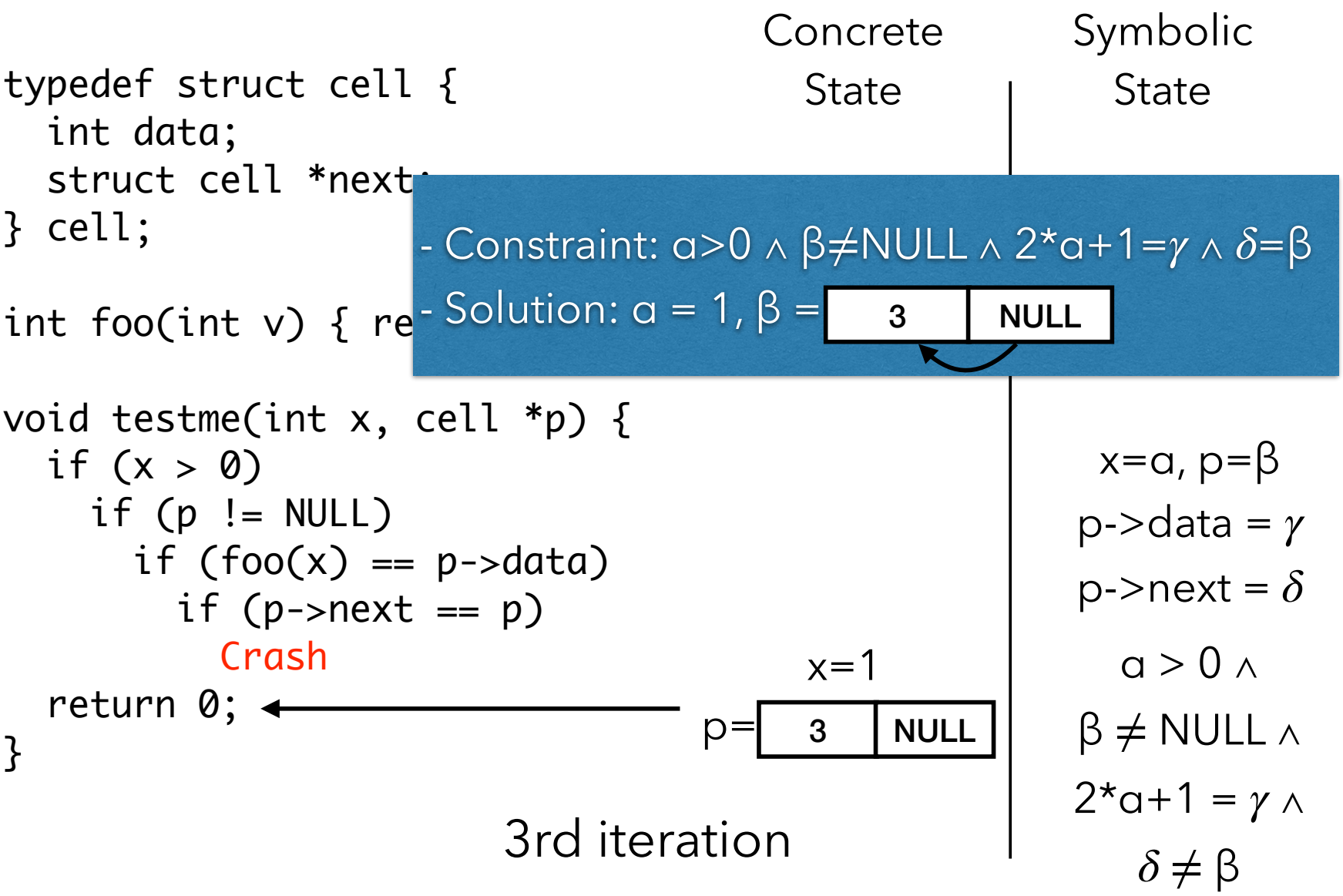
```
typedef struct cell {
  int data;
  struct cell *next;
} cell;

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

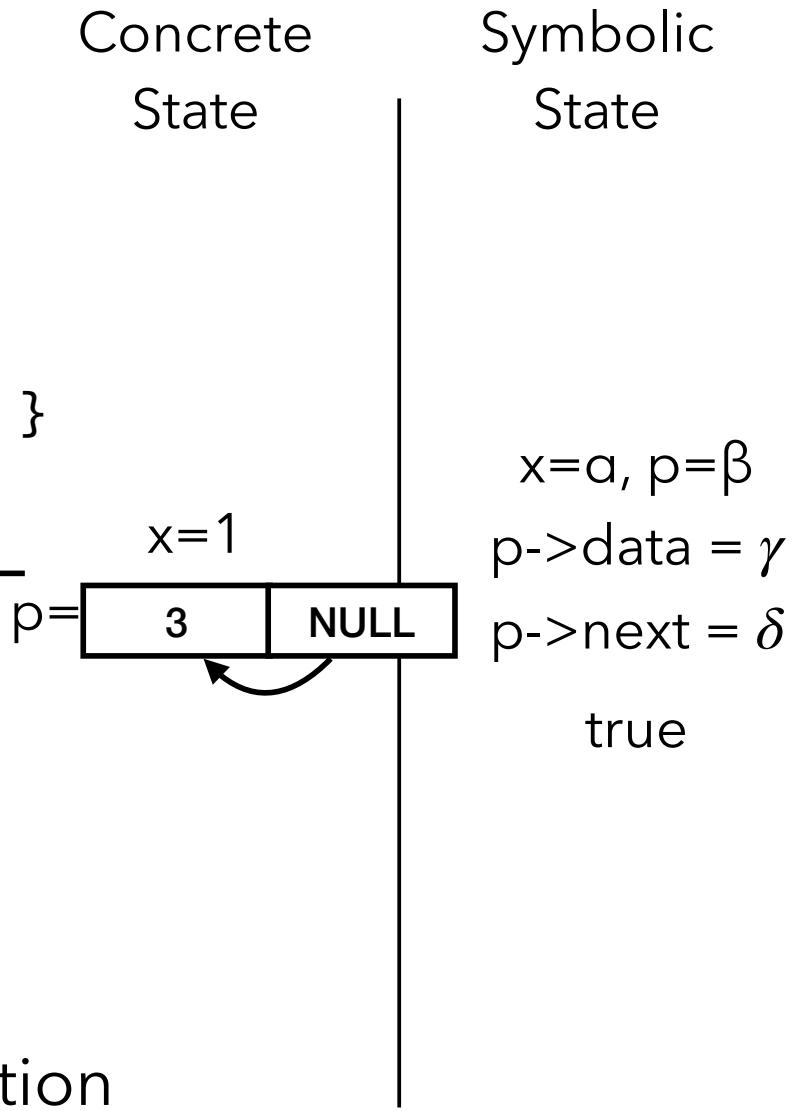
Concrete State	Symbolic State
	x=α, p=β
	p->data = γ
	p->next = δ
x=1	α > 0 ∧
p= 3 NULL	β ≠ NULL ∧
	2*α+1 = γ ∧
	$\delta \neq \beta$

Example: Concolic Testing of Data Structures (Cont'd)



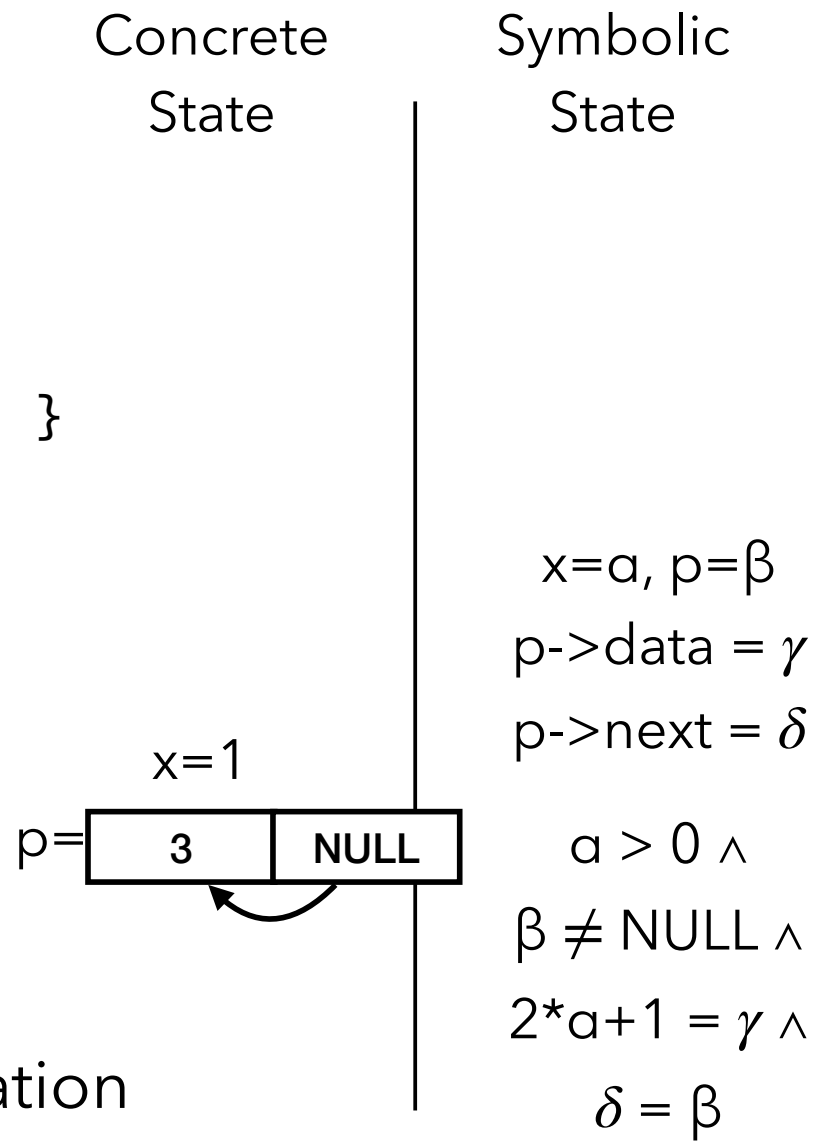
Example: Concolic Testing of Data Structures (Cont'd)

```
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;
}
```



Example: Concolic Testing of Data Structures (Cont'd)

```
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;
}
```



Summary

We explored how concolic testing works in more complex scenarios.

- Concolic testing of loops – there were two notable points.
  - It is enough to record path constraints that are data-dependant on inputs, because solutions of those constraints decide the next branches to explore.
  - Negating path conditions does not always increase code coverage.
- Concolic testing of data structures