

Computer-Aided Reasoning for Software

CSSE507

Symbolic Execution

Emina Torlak

emina@cs.washington.edu

Today

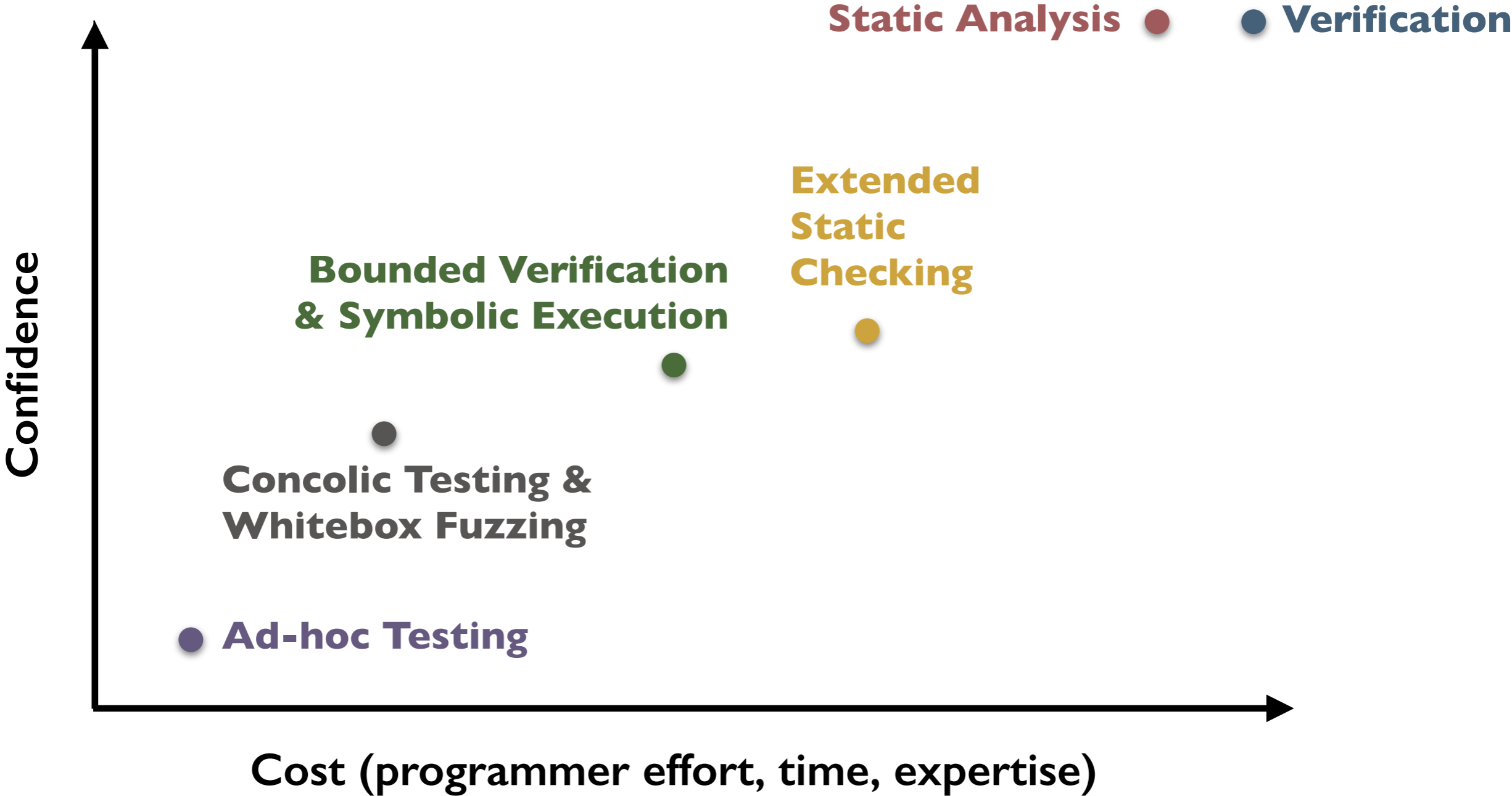
Last lecture

- VC generation with weakest liberal preconditions

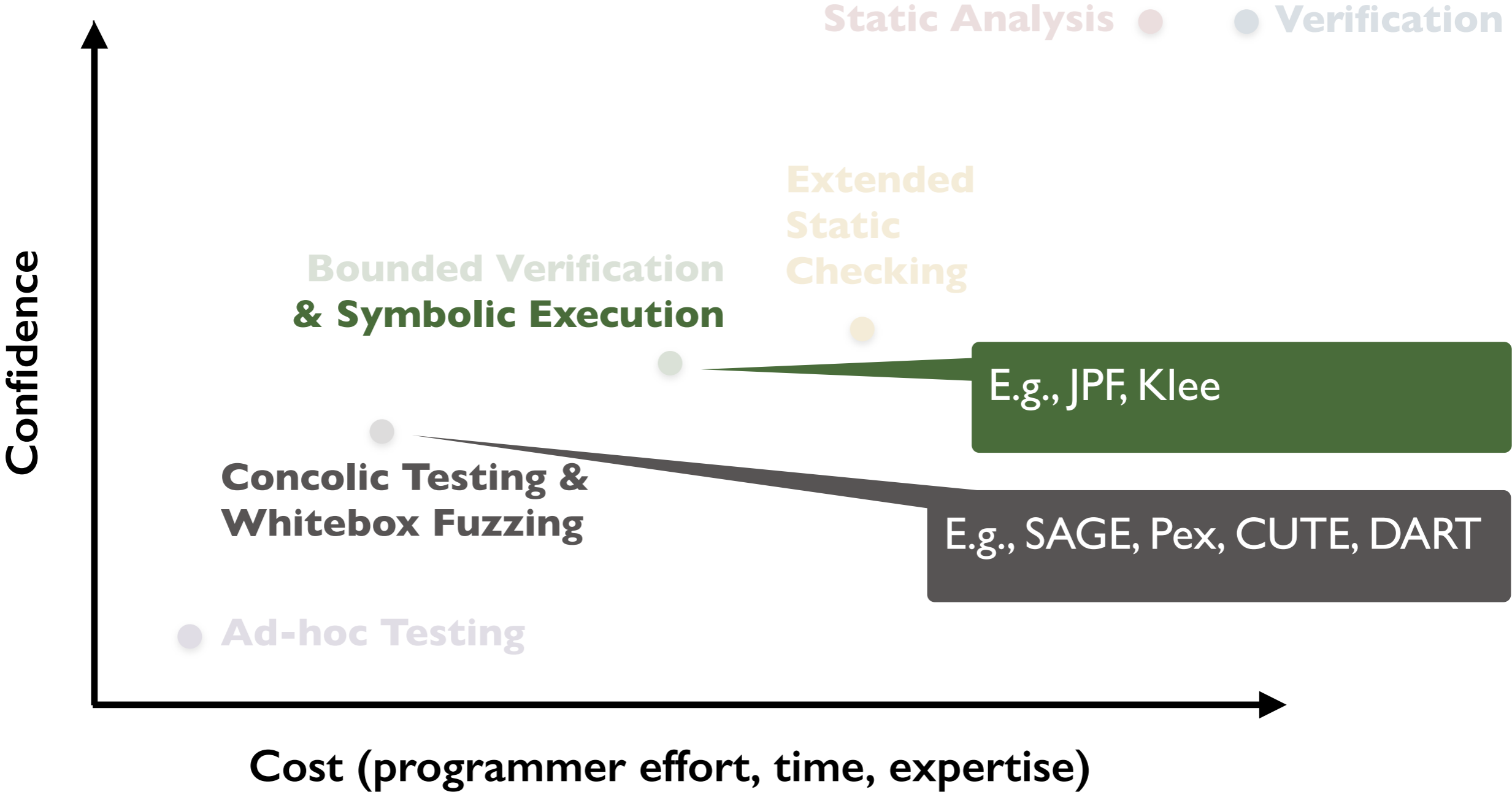
Today

- Symbolic execution: strongest postconditions for finite programs
- Concolic testing

The spectrum of program validation tools



The spectrum of program validation tools



A brief history of symbolic execution

1976: *A system to generate test data and symbolically execute programs* (Lori Clarke)

1976: *Symbolic execution and program testing* (James King)

2005-present: practical symbolic execution

- Using SMT solvers
- Heuristics to control exponential explosion
- Heap modeling and reasoning about pointers
- Environment modeling
- Dealing with solver limitations

Symbolic execution: basic idea

```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Symbolic execution: basic idea

```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Execute the program on *symbolic values*.

Symbolic execution: basic idea

```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

$x \mapsto A$
 $y \mapsto B$

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

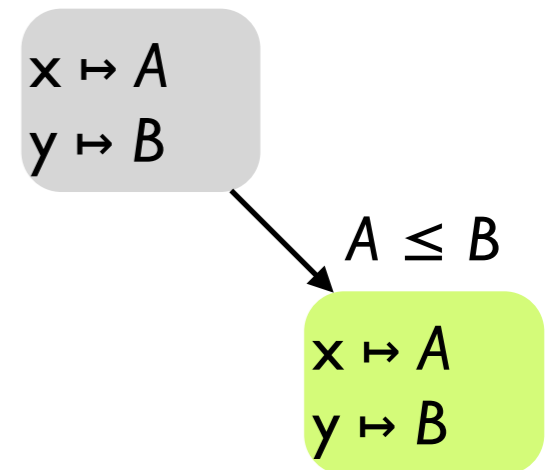
Symbolic execution: basic idea

```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.



Symbolic execution: basic idea

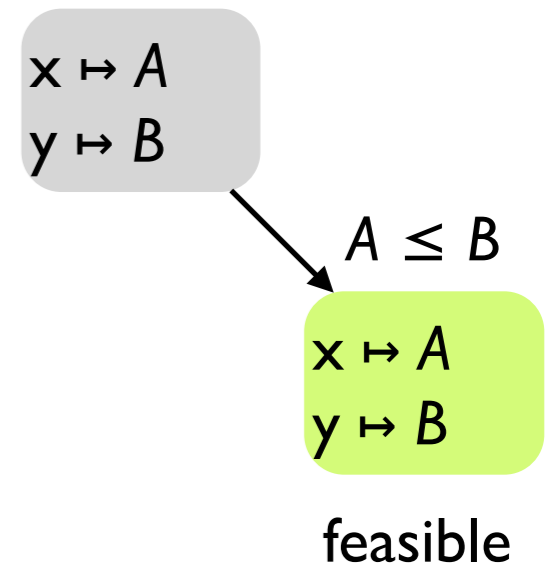
```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Symbolic execution: basic idea

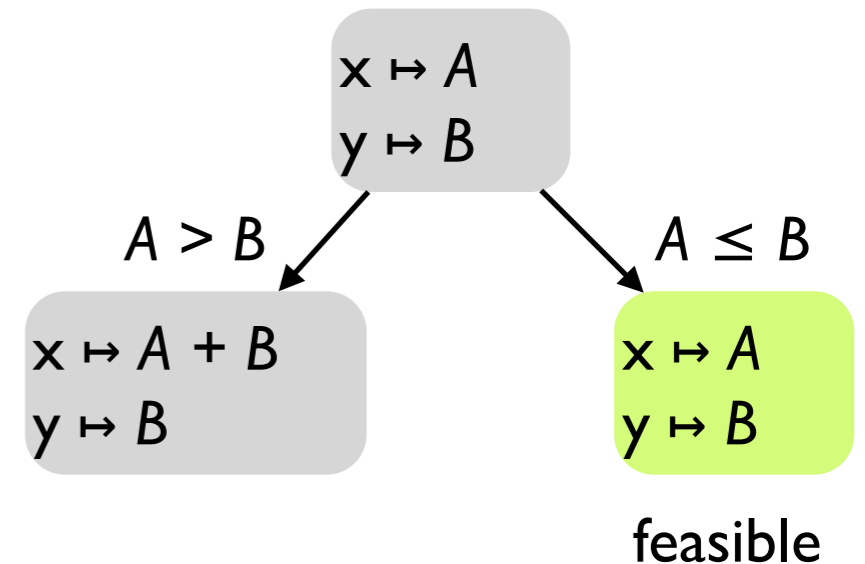
```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Symbolic execution: basic idea

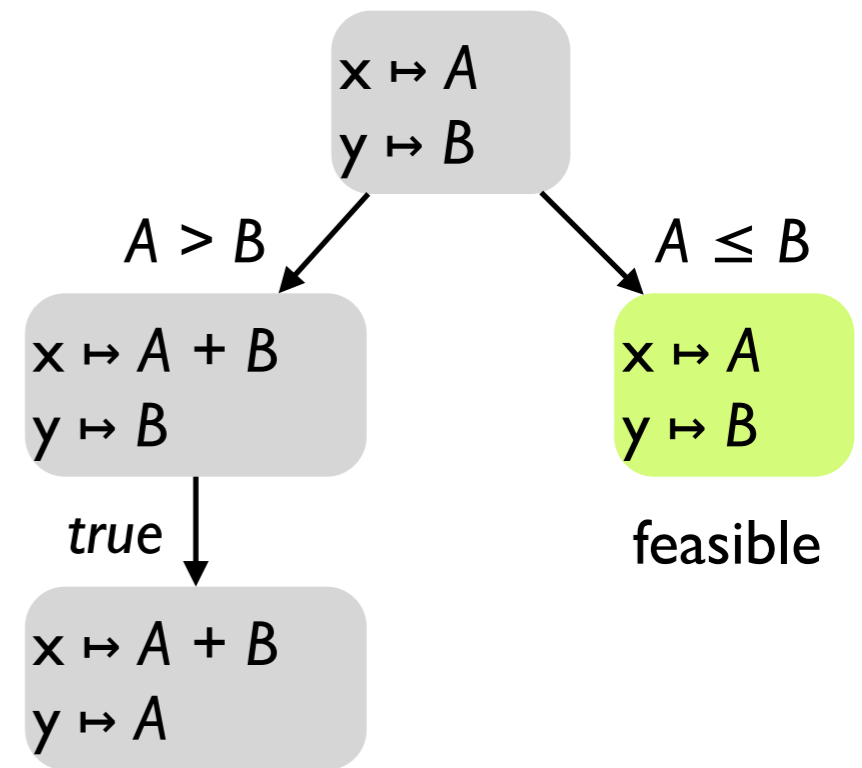
```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Symbolic execution: basic idea

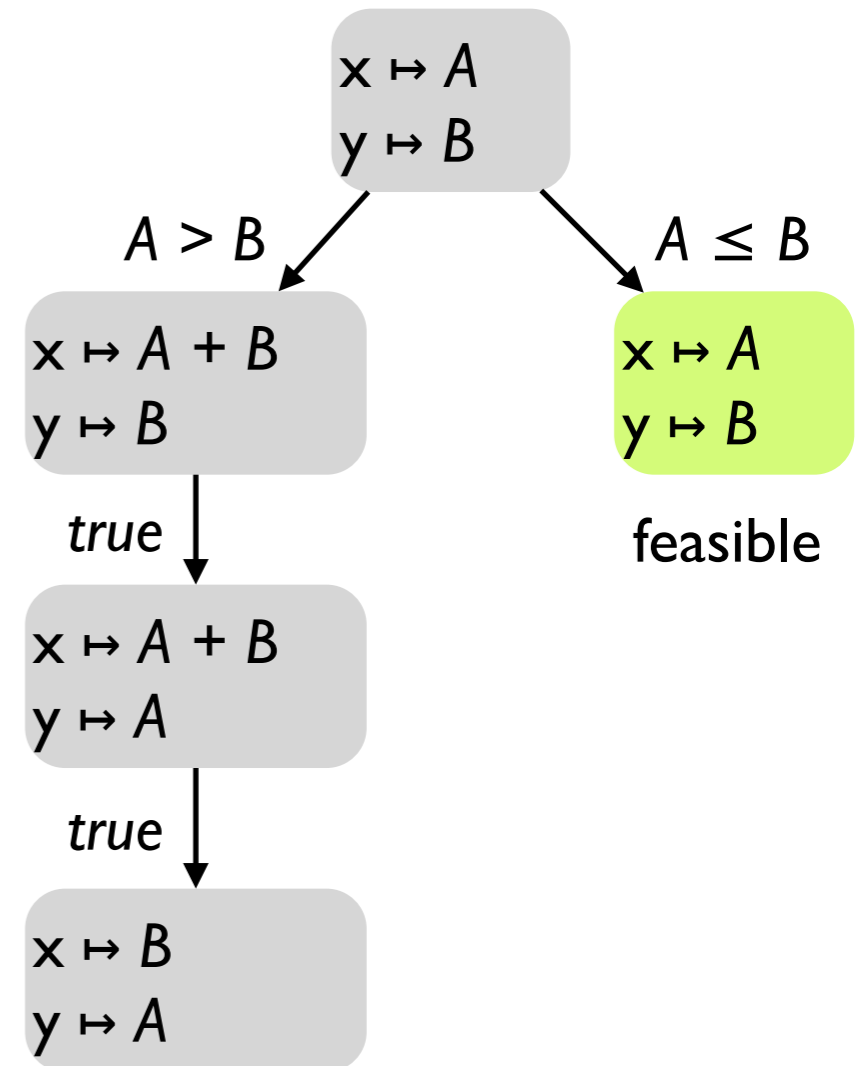
```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Symbolic execution: basic idea

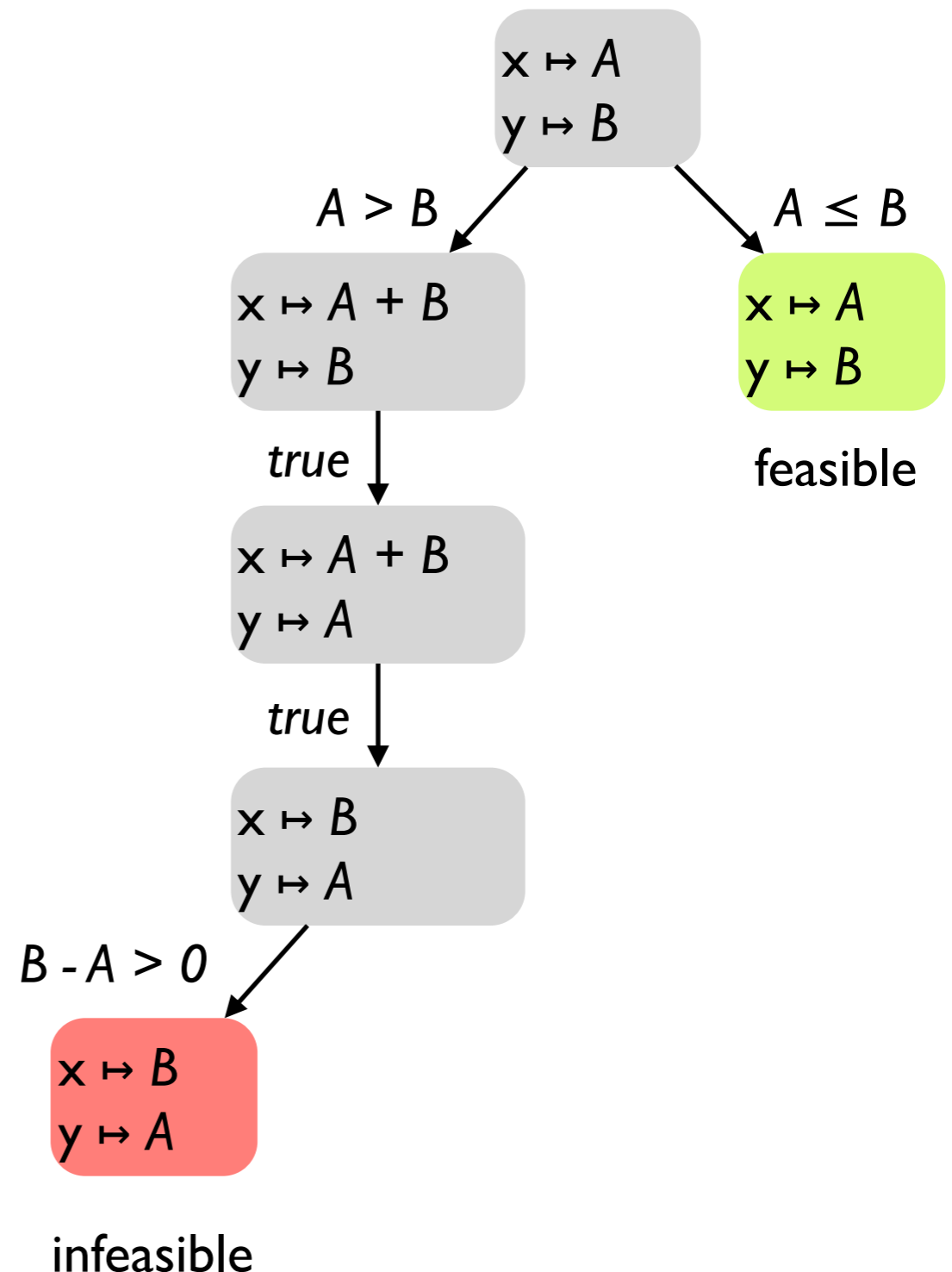
```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Symbolic execution: basic idea

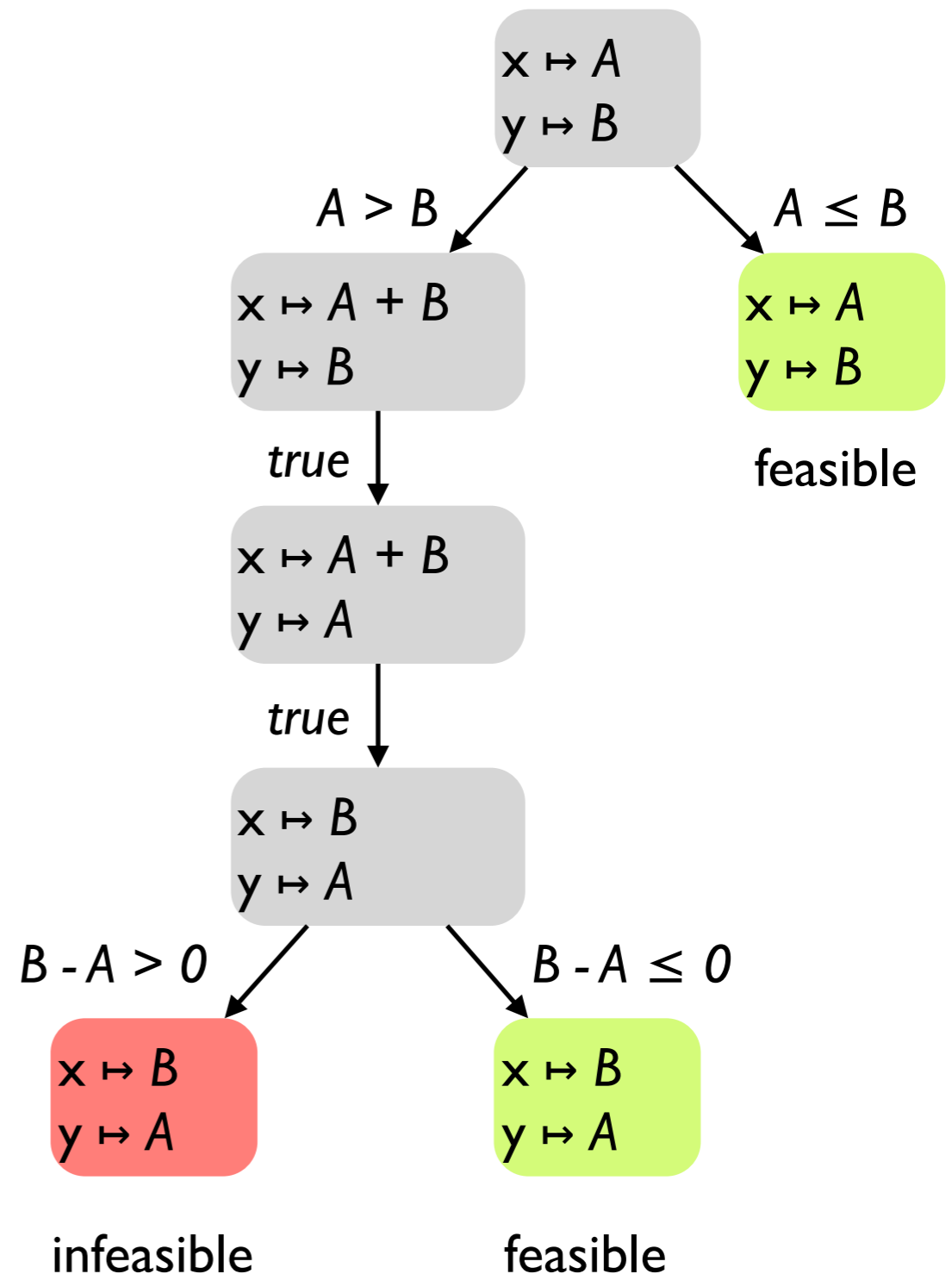
```
def f (x, y):  
    if (x > y):  
        x = x + y  
        y = x - y  
        x = x - y  
        if (x - y > 0):  
            assert false  
    return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths

Heap modeling: symbolic data structures and pointers

Solver limitations: dealing with complex PCs

Environment modeling: dealing with native / system / library calls

Loops and recursion

Dealing with infinite execution trees:

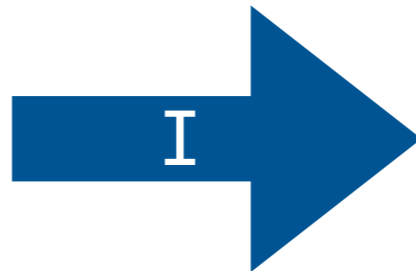
- Finitize paths by unrolling loops and recursion (bounded verification)
- Finitize paths by limiting the size of PCs (bounded verification)
- Use loop invariants (verification)

Loops and recursion

Dealing with infinite execution trees:

- Finitize paths by unrolling loops and recursion (bounded verification)
- Finitize paths by limiting the size of PCs (bounded verification)
- Use loop invariants (verification)

```
init;  
while (C) {  
    B;  
}  
assert P;
```

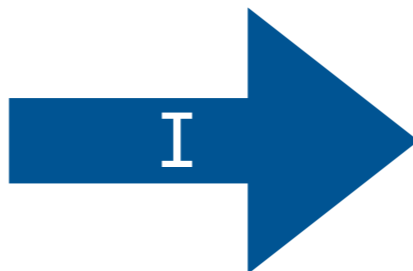


Loops and recursion

Dealing with infinite execution trees:

- Finitize paths by unrolling loops and recursion (bounded verification)
- Finitize paths by limiting the size of PCs (bounded verification)
- Use loop invariants (verification)

```
init;  
while (C) {  
    B;  
}  
assert P;
```



```
init;  
assert I;  
havoc targets(B);  
assume I;  
if (C) {  
    B;  
    assert I;  
    assume false;  
}  
assert P;
```

Path explosion

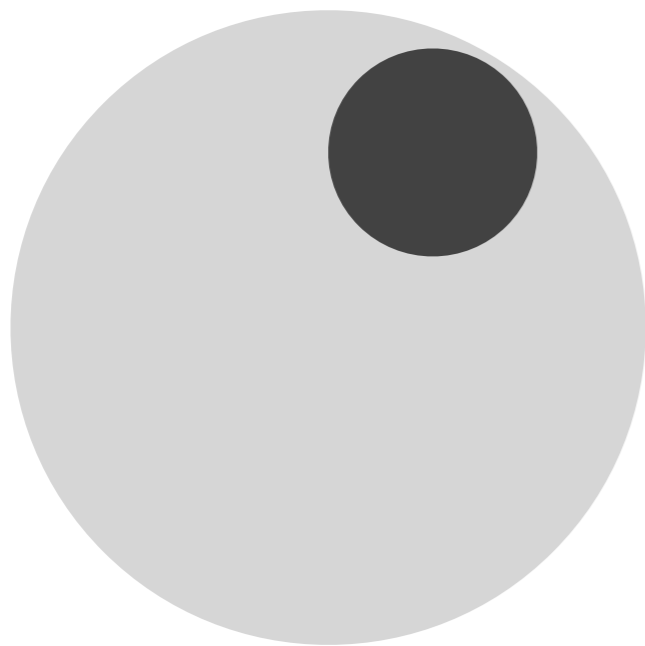
Achieving good coverage in the presence of exponentially many paths:

- Select next branch at random
- Select next branch based on coverage
- Interleave symbolic execution with random testing

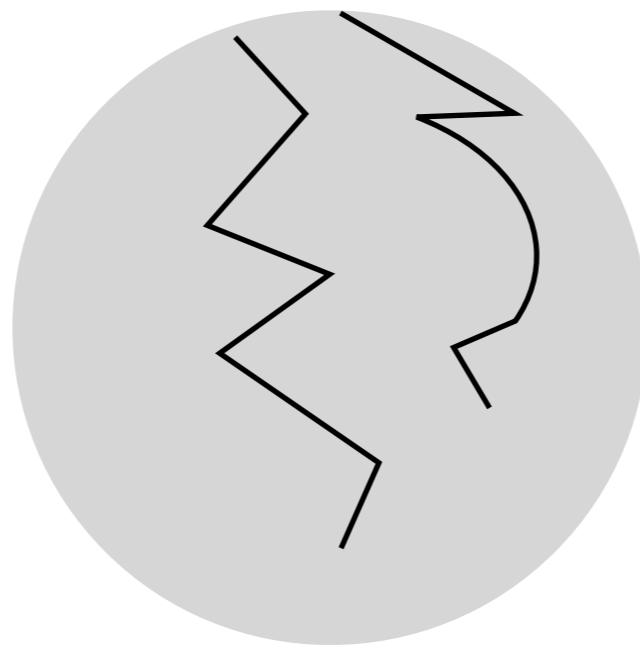
Path explosion

Achieving good coverage in the presence of exponentially many paths:

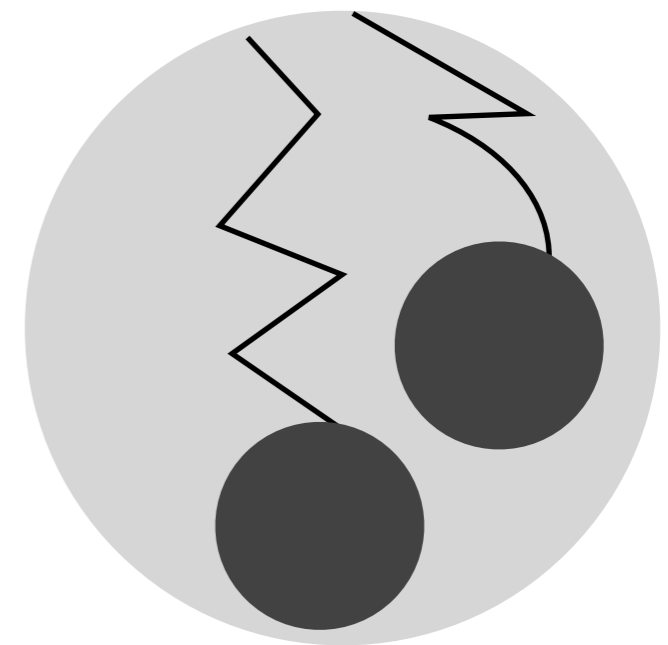
- Select next branch at random
- Select next branch based on coverage
- Interleave symbolic execution with random testing



symbolic execution



random testing



interleaved execution

Heap modeling

Modeling symbolic heap values and pointers

- Bit-precise memory modeling with the theory of arrays (EXE, Klee, SAGE)
- Lazy concretization (JPF)
- Concolic lazy concretization (CUTE)

Heap modeling: lazy concretization

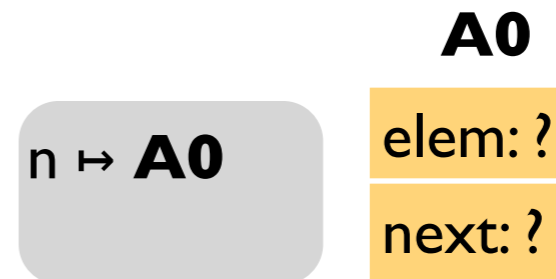
```
class Node {  
    int elem;  
    Node next;  
}
```

```
n = symbolic(Node);  
x = n.next;
```

Heap modeling: lazy concretization

```
class Node {  
    int elem;  
    Node next;  
}
```

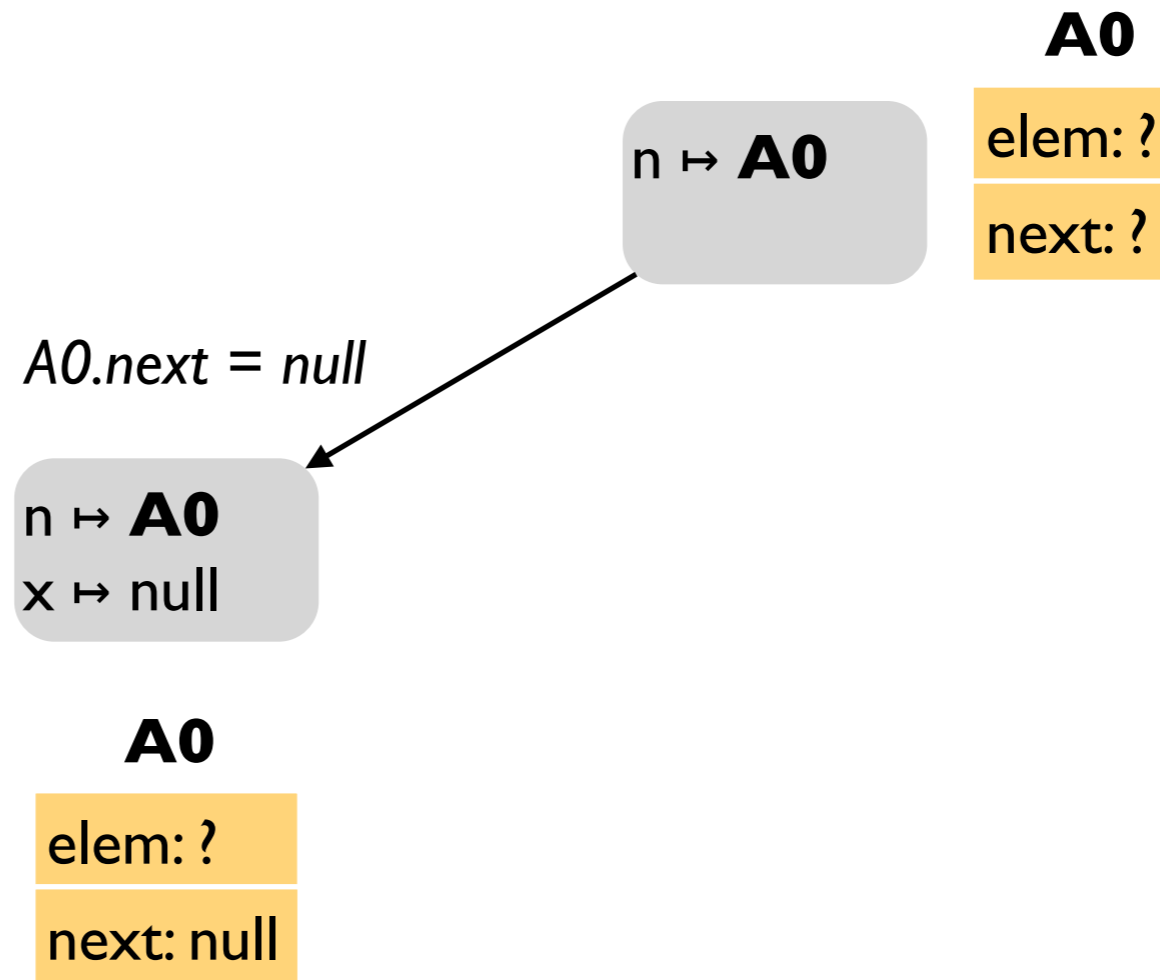
```
n = symbolic(Node);  
x = n.next;
```



Heap modeling: lazy concretization

```
class Node {  
  int elem;  
  Node next;  
}
```

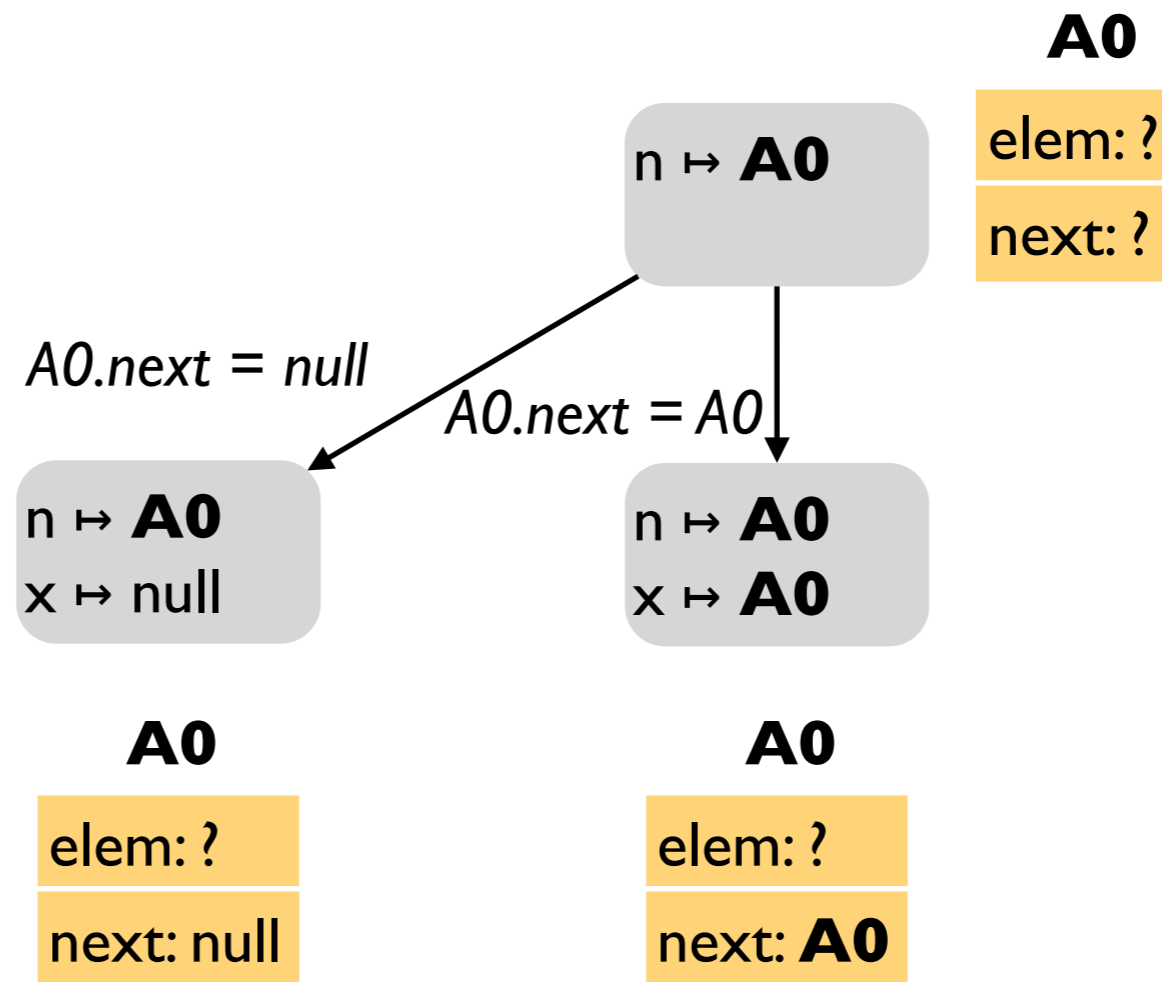
```
n = symbolic(Node);  
x = n.next;
```



Heap modeling: lazy concretization

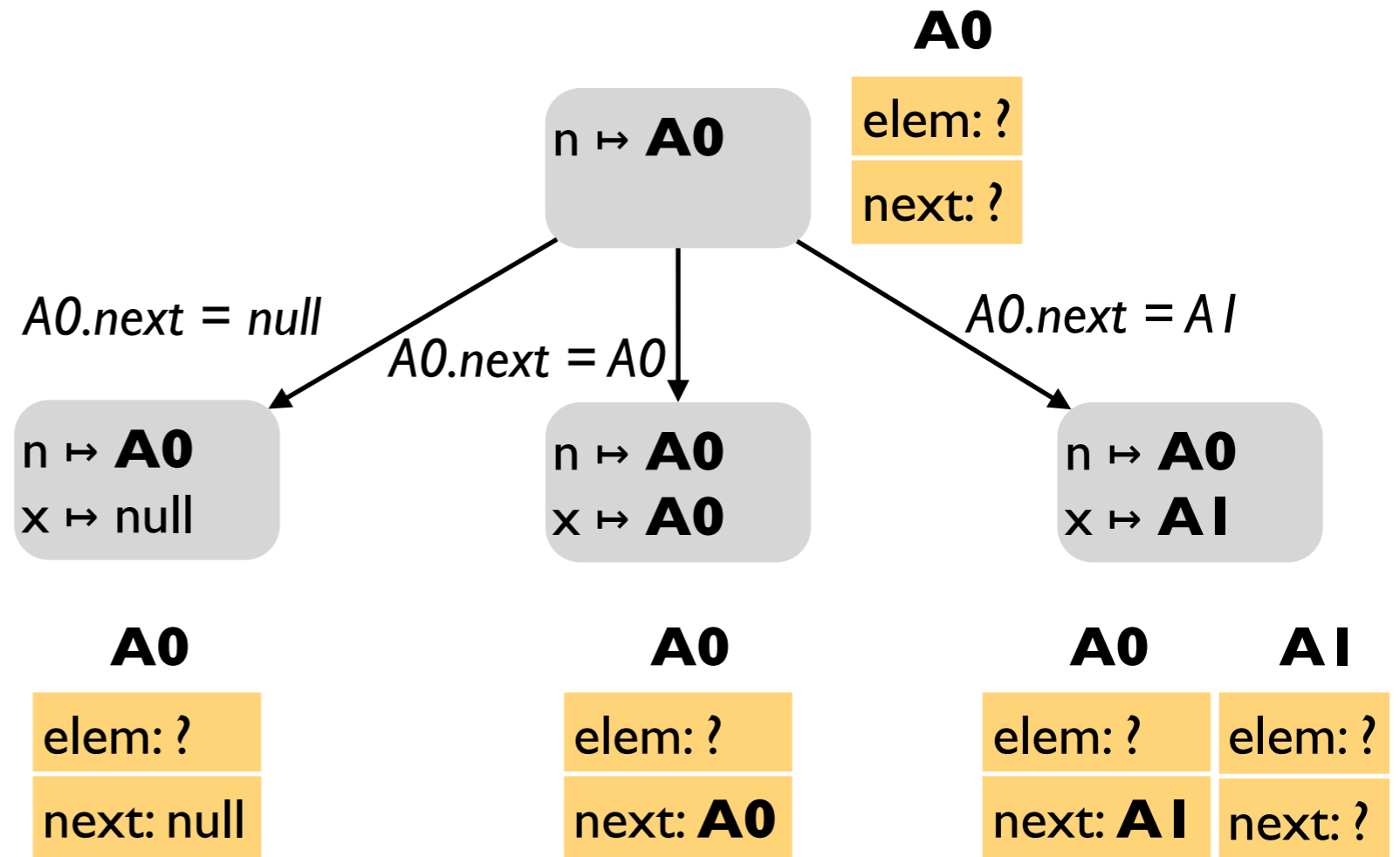
```
class Node {  
  int elem;  
  Node next;  
}
```

```
n = symbolic(Node);  
x = n.next;
```



Heap modeling: lazy concretization

```
class Node {  
  int elem;  
  Node next;  
}  
  
n = symbolic(Node);  
x = n.next;
```



Heap modeling: concolic testing

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

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

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    assert false;
    return 0;
}
```

Heap modeling: concolic testing

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

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

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    assert false;
    return 0;
}
```

Concrete

$p \mapsto \text{null}$
 $x \mapsto 236$

PC

$x > 0 \wedge p = \text{null}$

Execute concretely and symbolically. Negate last decision and solve for new inputs.

Heap modeling: concolic testing

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

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

```
int testme(cell *p, int x) {  
    if (x > 0)  
        if (p != NULL)  
            if (f(x) == p->v)  
                if (p->next == p)  
                    assert false;  
    return 0;  
}
```

Concrete

$p \mapsto \text{null}$
 $x \mapsto 236$

PC

$x > 0 \wedge p = \text{null}$

A0

next: null

v: 634

$p \mapsto \mathbf{A0}$
 $x \mapsto 236$

$x > 0 \wedge p \neq \text{null} \wedge$
 $p.v \neq 2x + 1$

Execute concretely and symbolically. Negate last decision and solve for new inputs.

Heap modeling: concolic testing

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

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

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    assert false;
    return 0;
}
```

Concrete

PC

$p \mapsto \text{null}$
 $x \mapsto 236$

$x > 0 \wedge p = \text{null}$

A0

next: null

v: 634

$p \mapsto \mathbf{A0}$
 $x \mapsto 236$

$x > 0 \wedge p \neq \text{null} \wedge$
 $p.v \neq 2x + 1$

A0

next: null

v: 3

$p \mapsto \mathbf{A0}$
 $x \mapsto 1$

$x > 0 \wedge p \neq \text{null} \wedge$
 $p.v = 2x + 1 \wedge$
 $p.next \neq p$

Execute concretely and symbolically. Negate last decision and solve for new inputs.

Heap modeling: concolic testing

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

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

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    assert false;
    return 0;
}
```

Concrete

PC

$p \mapsto \text{null}$
 $x \mapsto 236$

$x > 0 \wedge p = \text{null}$

A0

next: null

v: 634

$p \mapsto \mathbf{A0}$
 $x \mapsto 236$

$x > 0 \wedge p \neq \text{null} \wedge$
 $p.v \neq 2x + 1$

A0

next: null

v: 3

$p \mapsto \mathbf{A0}$
 $x \mapsto 1$

$x > 0 \wedge p \neq \text{null} \wedge$
 $p.v = 2x + 1 \wedge$
 $p.next \neq p$

A0

next: **A0**

v: 3

$p \mapsto \mathbf{A0}$
 $x \mapsto 1$

$x > 0 \wedge p \neq \text{null} \wedge$
 $p.v = 2x + 1 \wedge$
 $p.next = p$

Execute concretely and symbolically. Negate last decision and solve for new inputs.

Solver limitations

Reducing the demands on the solver:

- On-the-fly expression simplification
- Incremental solving
- Solution caching
- Substituting concrete values for symbolic in complex PCs (CUTE)

Environment modeling

Dealing with system / native / library calls:

- Partial state concretization
- Manual *models* of the environment (Klee)

Summary

Today

- Practical symbolic execution and concolic testing

Next lecture

- Angelic execution