CS 292C Computer-Aided Reasoning for Software

Lecture 13: Symbolic Execution

Yu Feng
Fall 2019

Summary of previous lecture

- 3rd homework is out
- Reasoning about (partial) correctness with Hoare Logic
- Strongly encouraged: complete the main tutorial at http://rise4fun.com/Dafny/tutorial

Simple Imperative Programming Language

Expression E

- $Z \mid V \mid E_1 + E_2 \mid E_1 * E_2$

Conditional C

True | False | $E_1 = E_2 \mid E_1 \leq E_2$

A minimalist programming language for demonstrating key features of Hoare logic.

Statement S

- skip (Skip)
- abort. (Abort)
- V := E (Assignment)
- $S_1; S_2$. (Composition)
- **if** C **then** S_1 **else** S_2 (If)
- **while** C **do** S (While)

Hoare logic rules

$$\frac{}{\vdash \{P\}\ \mathrm{Skip}\ \{P\}}$$

$$\frac{\vdash \{P\}\ S_1\ \{R\} \quad \vdash \{R\}\ S_2\ \{Q\}}{\vdash \{P\}\ S_1; S_2\ \{Q\}}$$

$$\frac{}{\vdash \{\mathrm{true}\}\ \mathrm{abort}\ \{\mathrm{false}\}}$$

$$\frac{\vdash \{P \wedge C\}\ S_1\ \{Q\} \qquad \vdash \{P \wedge \neg C\}\ S_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ C\ \textbf{then}\ \mathbf{S_1}\ \textbf{else}\ \mathbf{S_2}\ \{Q\}}$$

$$\frac{}{\vdash \{Q[E/x]\}\ x := E\ \{Q\}}$$

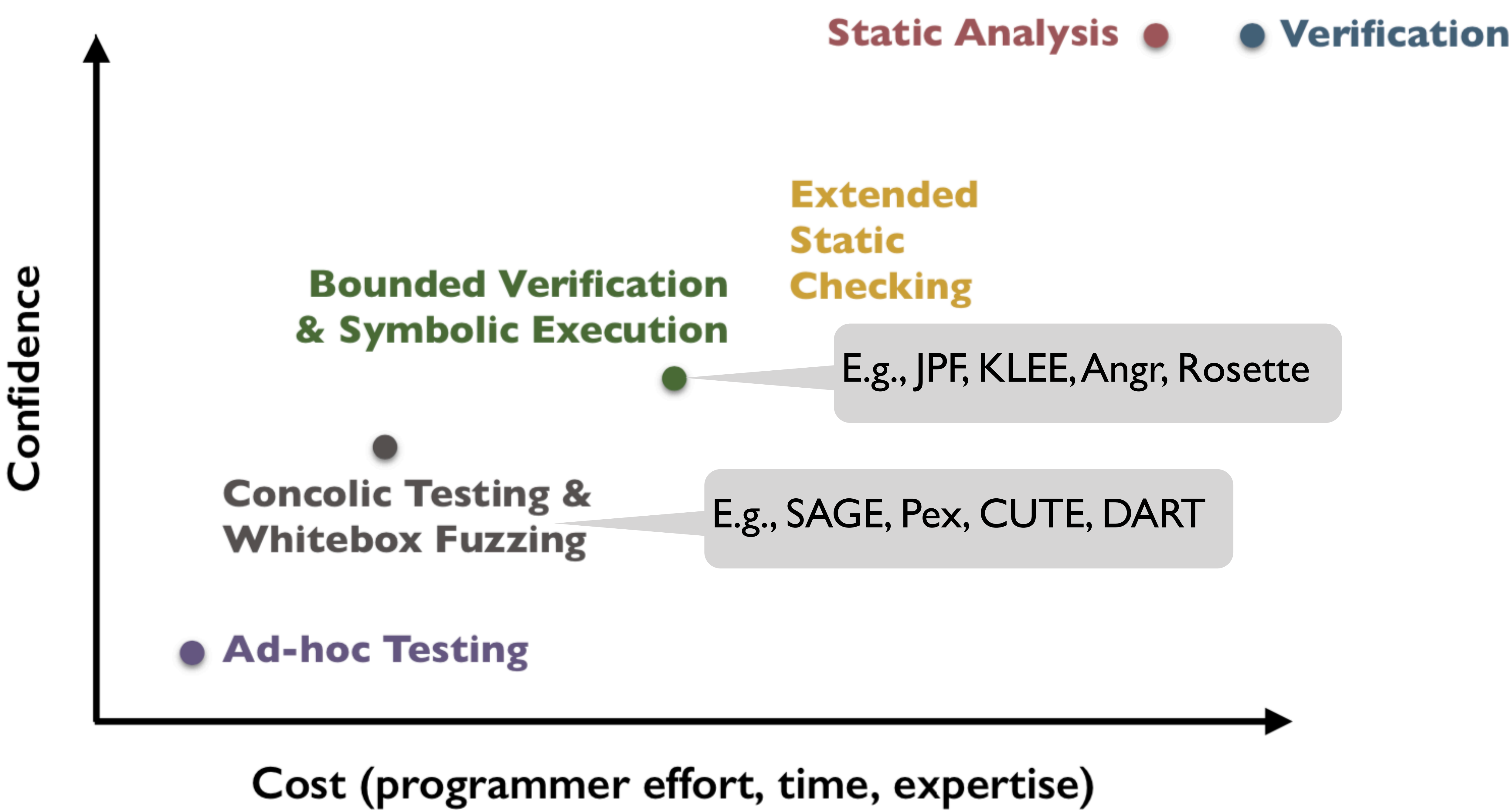
$$\frac{\vdash \{I \wedge C\}\ S\ \{I\}}{\vdash \{I\}\ \textbf{while}\ C\ \textbf{do}\ S\ \{I \wedge \neg C\}}$$

$$\frac{\vdash \{P_1\}\ S\ \{Q_1\} \quad P \Rightarrow P_1 \quad Q_1 \Rightarrow Q}{\vdash \{P\}\ S\ \{Q\}}$$

Outline of this lecture

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

The spectrum of program validation tools



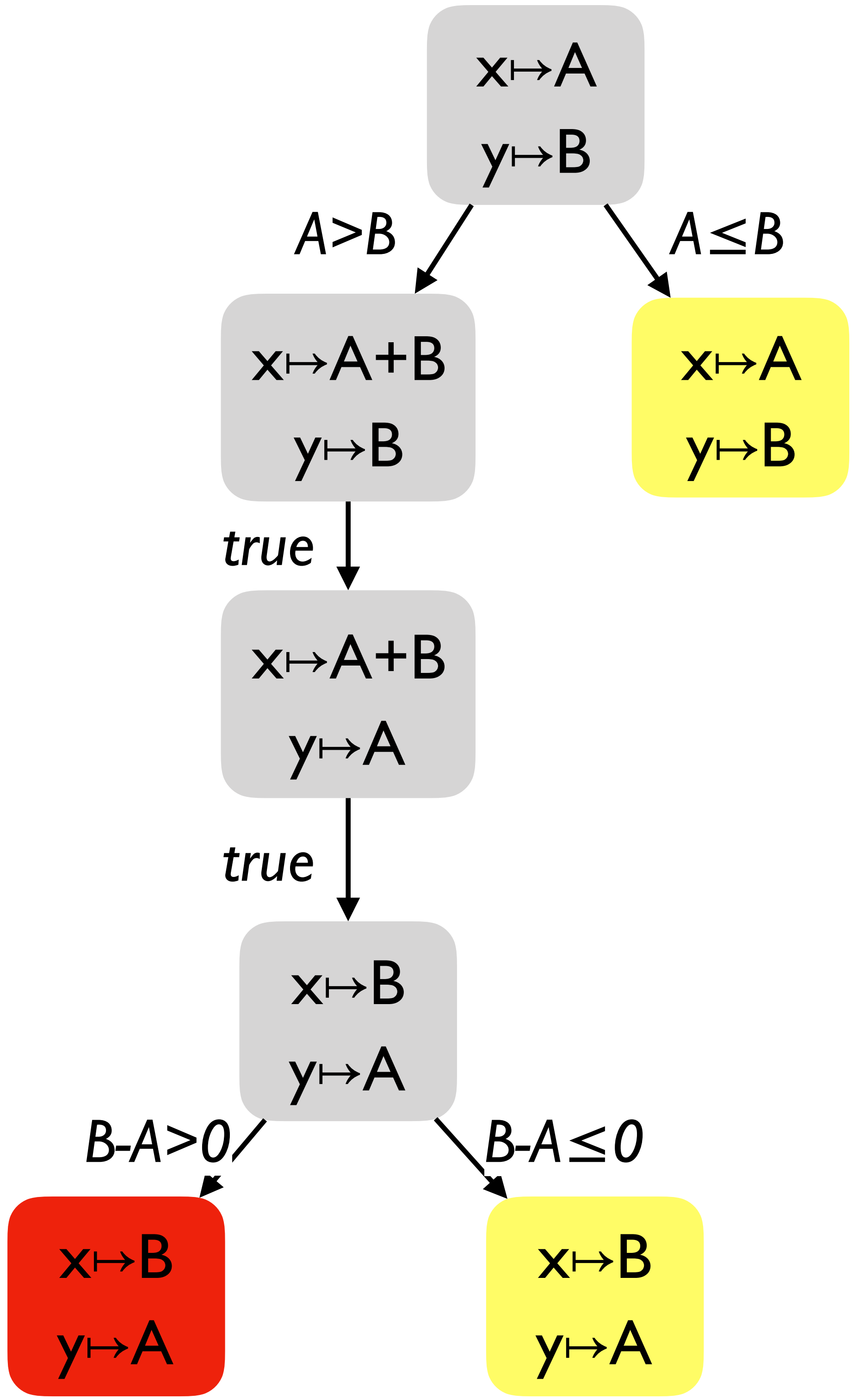
A brief history of symbolic execution

- **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)
```

- 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)

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

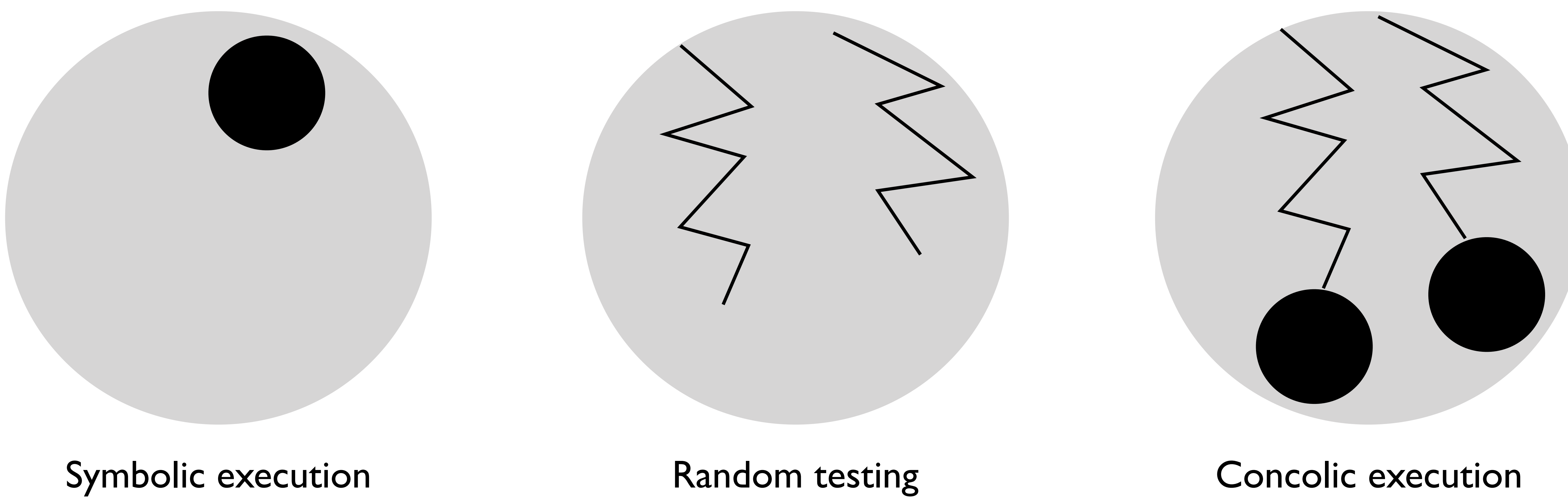
Loop invariant I

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

Path explosion

Achieving good coverage in the presence of exponentially many paths:

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



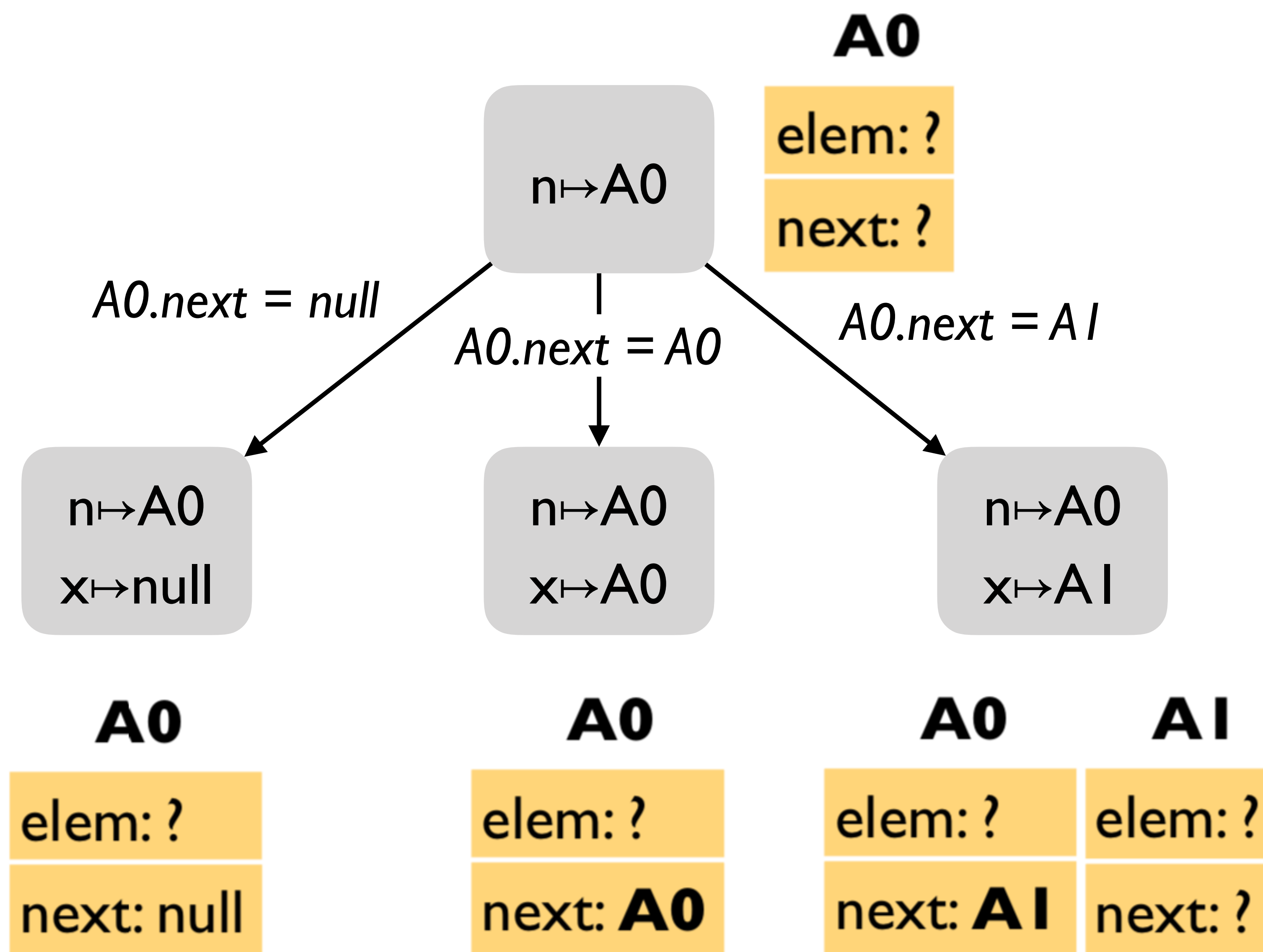
Heap modeling

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: 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 ↦ null x ↦ 236	$x > 0 \wedge p{=}null$
A0 next: null v: 634	p ↦ **A0** x ↦ 236	$x > 0 \wedge p{\neq}null \wedge$ $p.v \neq 2x + 1$
A0 next: null v: 3	p ↦ **A0** x ↦ 1	$x > 0 \wedge p{\neq}null \wedge$ $p.v = 2x + 1 \wedge$ $p.next \neq p$
A0 next: **A0** v: 3	p ↦ **A0** x ↦ 1	$x > 0 \wedge p{\neq}null \wedge$ $p.v = 2x + 1 \wedge$ $p.next = p$

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)

TODOs by next lecture

- Work on your final project! (50%)