Symbolic Execution

Testing, Quality Assurance, and Maintenance
Fall 2023

Prof. Arie Gurfinkel

based on slides by Prof. Johannes Kinder and others

UNIVERSITY OF
WATERLOO

How would you test this program?

Remember this little program Foo()?

Is it possible to **automatically** generate a test-suite that

- achieves full branch coverage
- discovers weather division by 0 is possible
- identifies all infeasible test requirements (i.e., dead code)

Remarkably, the answer is YES

```
def Foo(x, y):
""" requires: x and y are int
    ensures: returns floor(max(x,y)/min(x, y))"""
   if x > y:
      return x / y
   else
      return y / x
```

`floor(z)` is the largest integer not greater than z

Symbolic execution in a nutshell

- to identify division by 0, add explicit tests (i.e., oracles)
- **traverse** the program to compute each program path
  - path1: `x > y, y == 0` ; path2: `x > y, y != 0, return x / y` ; …
- **solve** constraints for each path using a constraint (or logic) solver
  - path1: `x=10, y=0`; path2: `x=10, y=1`; …
- **run** the program on tests generated by previous step
- all testing is now automatic

```
def Foo(x, y):
""" requires: x and y are int
    ensures: returns floor(max(x,y)/min(x, y))"""
   if x > y:
      assert y != 0
      return x / y
   else
      assert x != 0
      return y / x
```

`floor(z)` is the largest integer not greater than z

Symbolic Execution is a Combination of

Automatically explore program paths

- Execute program on “symbolic” input values
- “Fork” execution at each branch
- Record branching conditions

Constraint solver

- Decides path feasibility
- Generates test cases for paths and bugs

History

Int. Conference on Reliable Software 1975

James C. King:
A new approach to program testing

Robert S. Boyer, Bernard Elspas, Karl N. Levitt:
SELECT—a formal system for testing and debugging programs by symbolic execution

Recent work on proving the correctness of programs by formal analysis [5] shows great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future.

History (2)

SAT / SMT solvers lead to boom in 2000s

- Constraint solving becomes a commodity
- Makes classic algorithms viable in practice

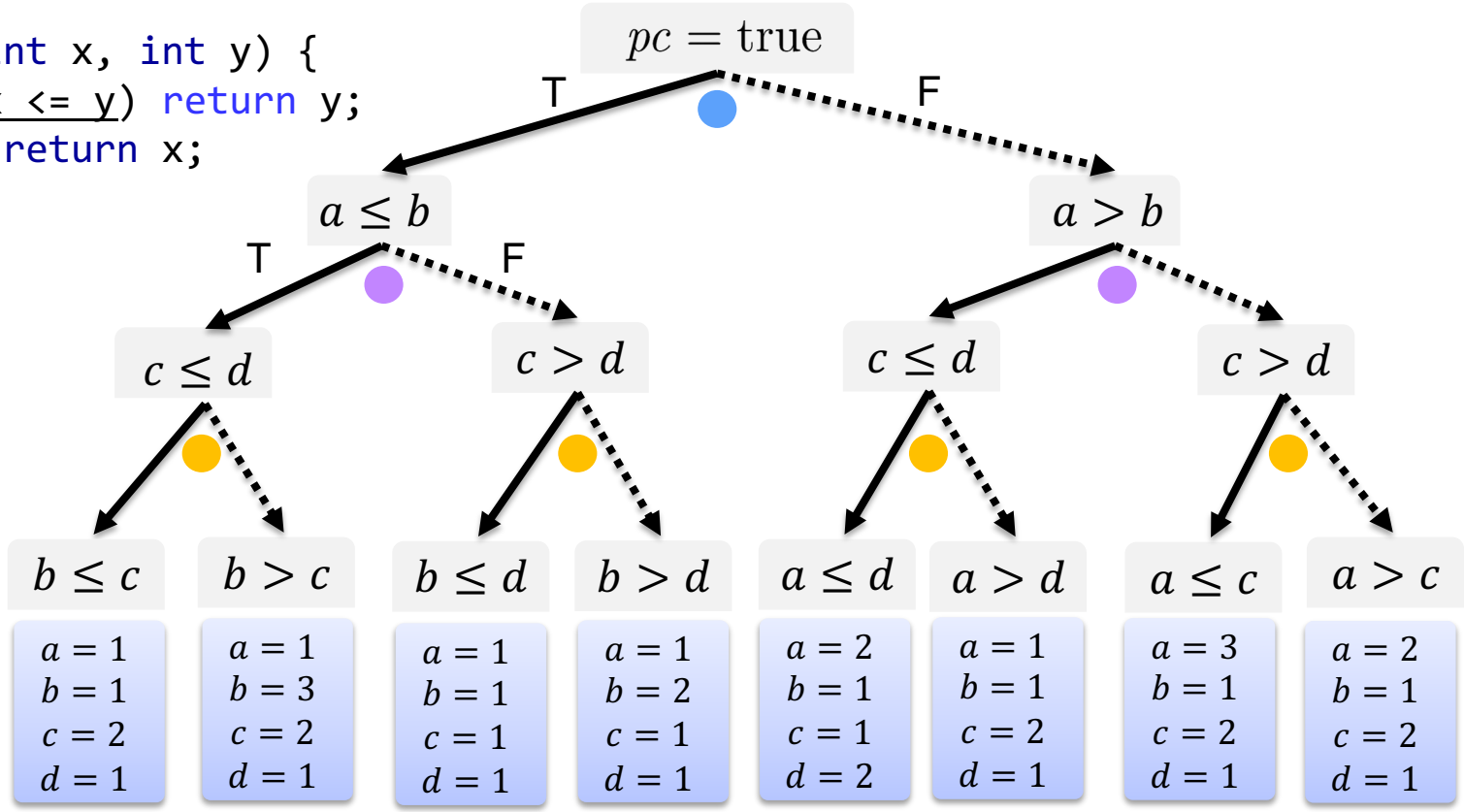
Conceptual breakthroughs (Dynamic Symbolic Execution)

- Patrice Godefroid, Nils Klarlund, Koushik Sen: *DART: directed automated random testing*. PLDI 2005
- Cristian Cadar, **Vijay Ganesh**, Peter M. Pawlowski, David L. Dill, Dawson R. Engler: *EXE: automatically generating inputs of death*. CCS 2006

Symbolic Execution Illustrated

```
int Max(int a, int b, int c, int d) {
    return Max(Max(a, b ●), Max(c, d ●) ●);
}
```

```
int Max(int x, int y) {
    if (x <= y) return y;
    else return x;
}
```



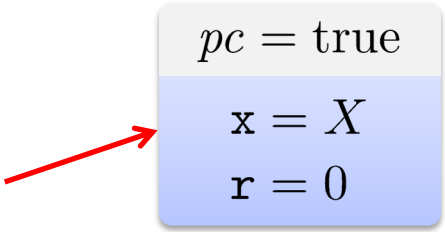
$pc = \text{true}$

$\mathtt{x} = X$

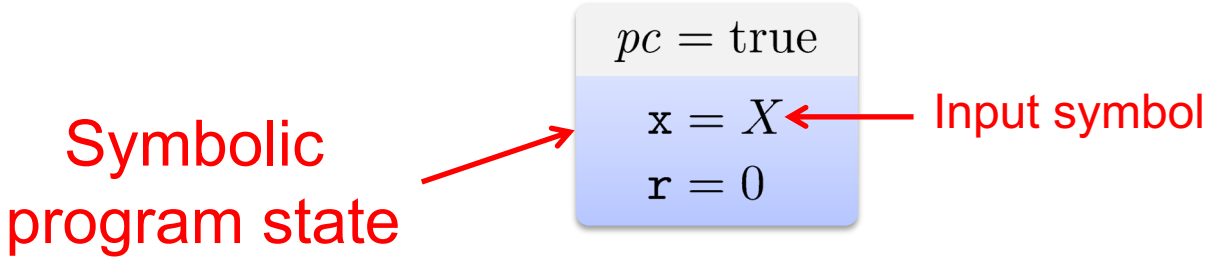
$\mathtt{r} = 0$

```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8) {
6      r = x - 7
7    }
8
9    if (x < 5) {
10     r = x - 2
11   }
12
13   return r
   }
```

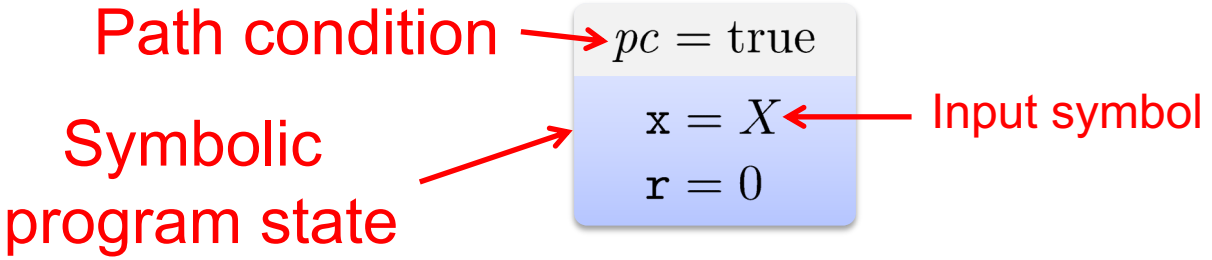
Symbolic program state




```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8) {
6      r = x - 7
7    }
8
9    if (x < 5) {
10     r = x - 2
11   }
12
13   return r
   }
```

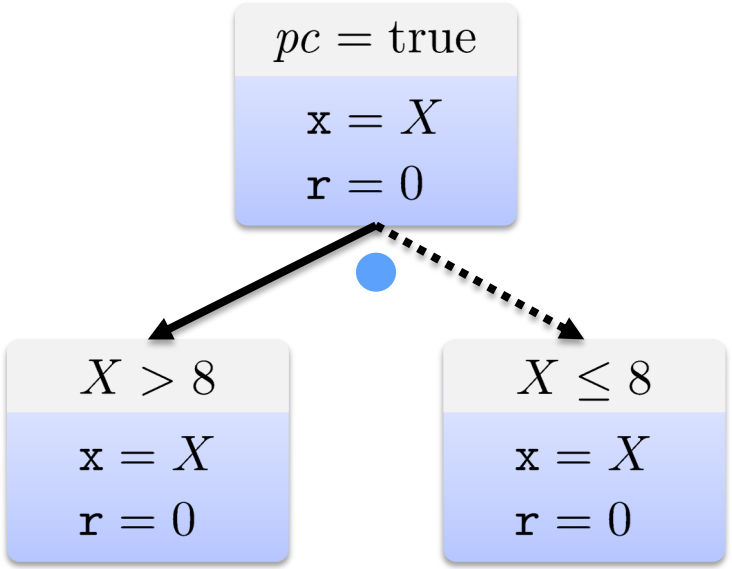



```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8) {
6      r = x - 7
7    }
8
9    if (x < 5) {
10     r = x - 2
11   }
12
13   return r
   }
```

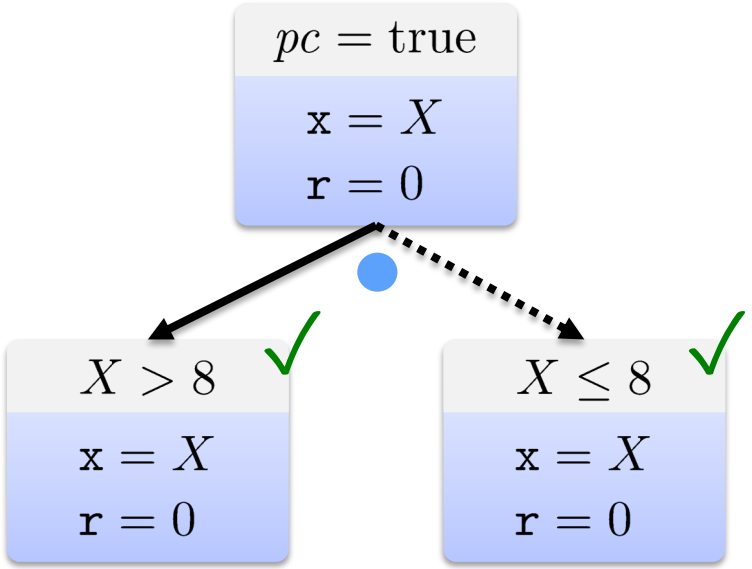



```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8) {
6      r = x - 7
7    }
8
9    if (x < 5) {
10     r = x - 2
11   }
12
13   return r
   }
```

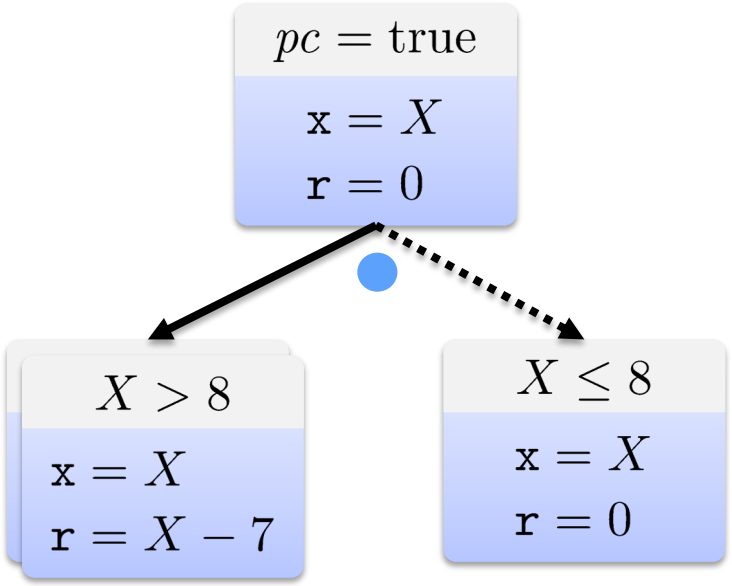
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8) {
6      r = x - 7
7    }
8
9    if (x < 5) {
10     r = x - 2
11   }
12
13   return r
   }
```



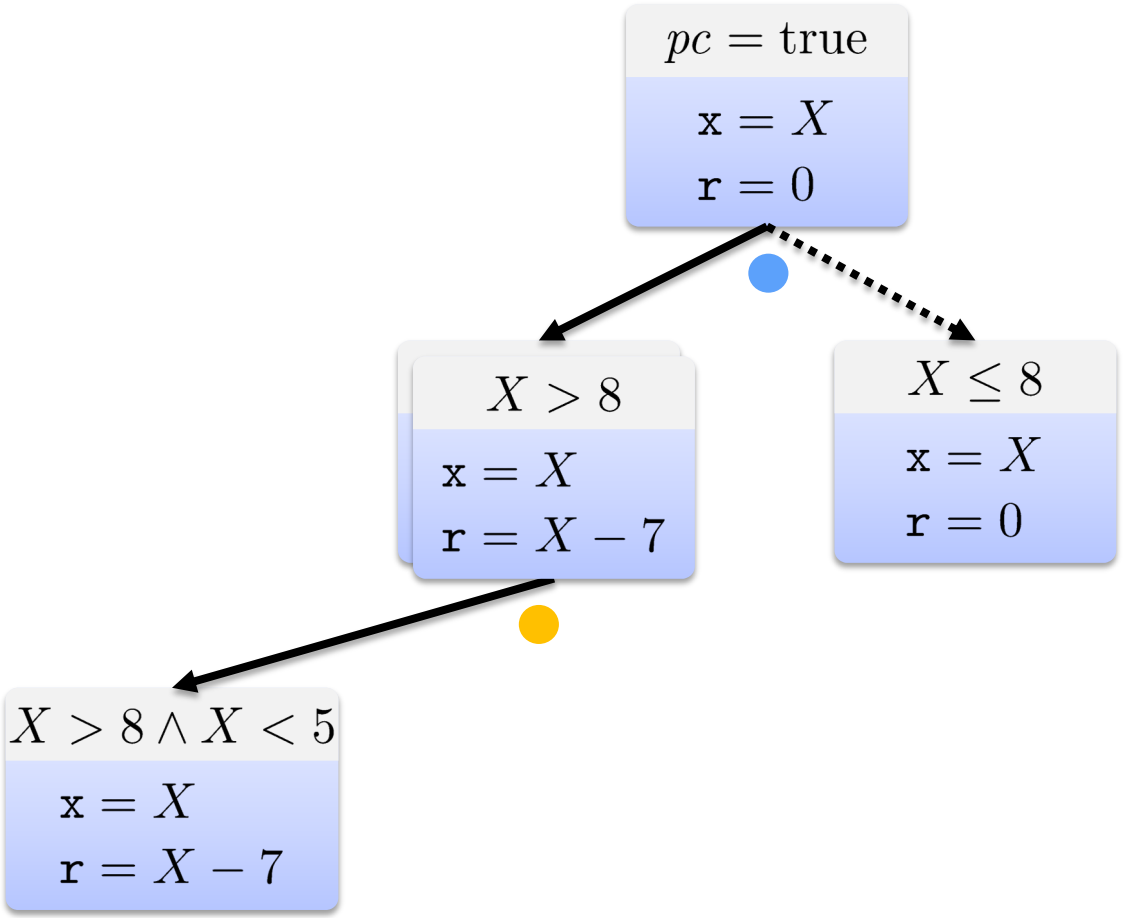
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8) {
6      r = x - 7
7    }
8
9    if (x < 5) {
10     r = x - 2
11   }
12
13   return r
   }
```



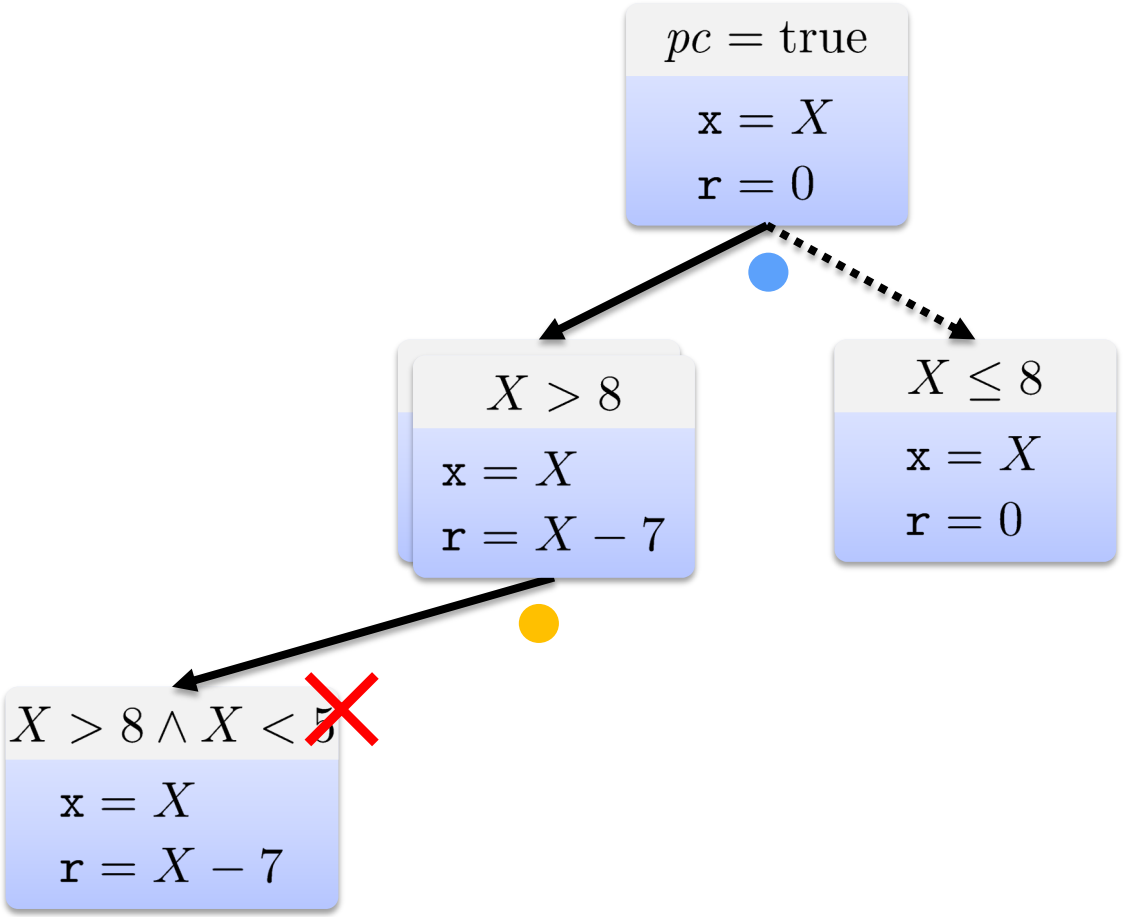
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8) {
6      r = x - 7
7    }
8
9    if (x < 5) {
10     r = x - 2
11   }
12
13   return r
   }
```



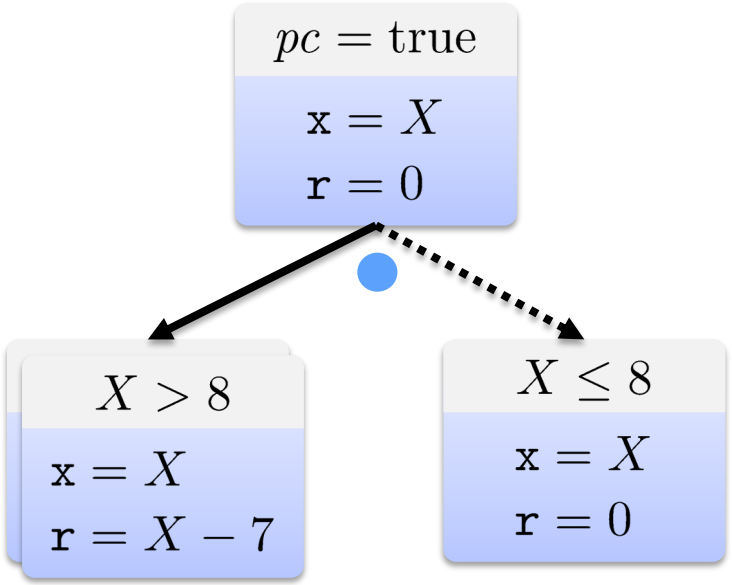
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8 ●) {
6      r = x - 7
7    }
8
9    if (x < 5 ●) {
10     r = x - 2
11   }
12
13   return r
   }
```



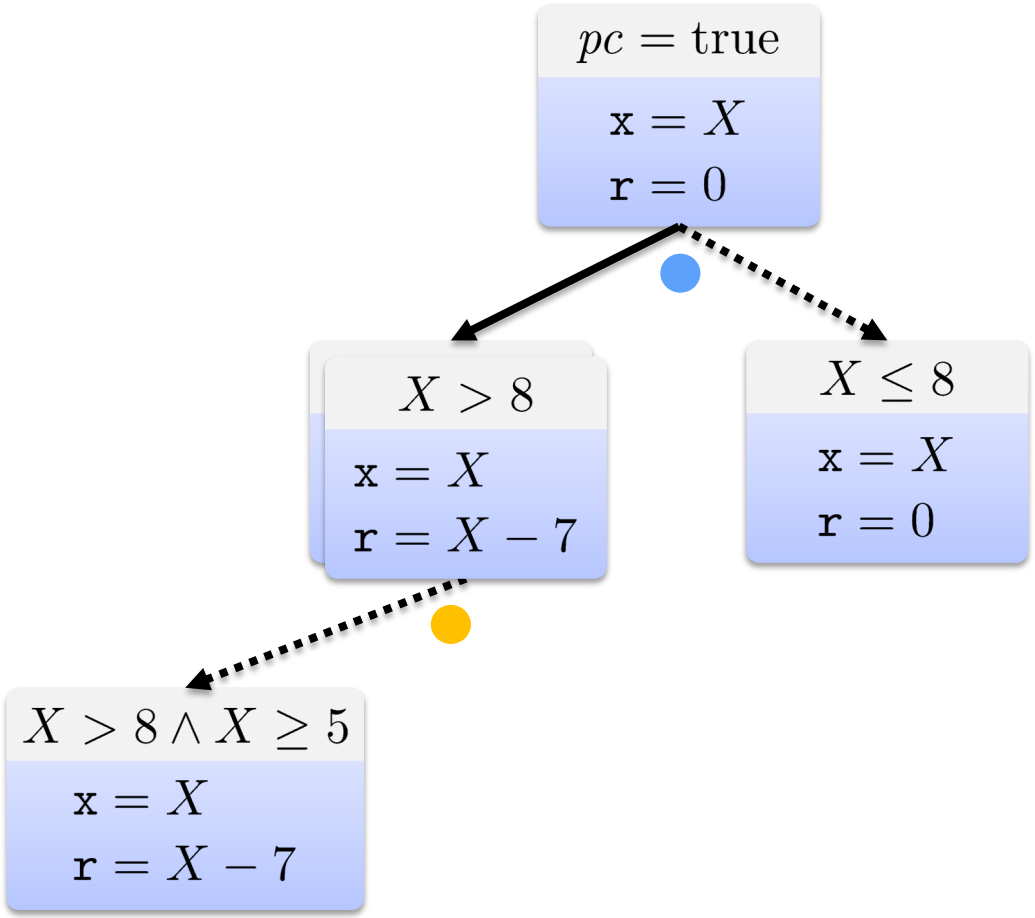
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8 ●) {
6      r = x - 7
7    }
8
9    if (x < 5 ●) {
10     r = x - 2
11   }
12
13   return r
   }
```



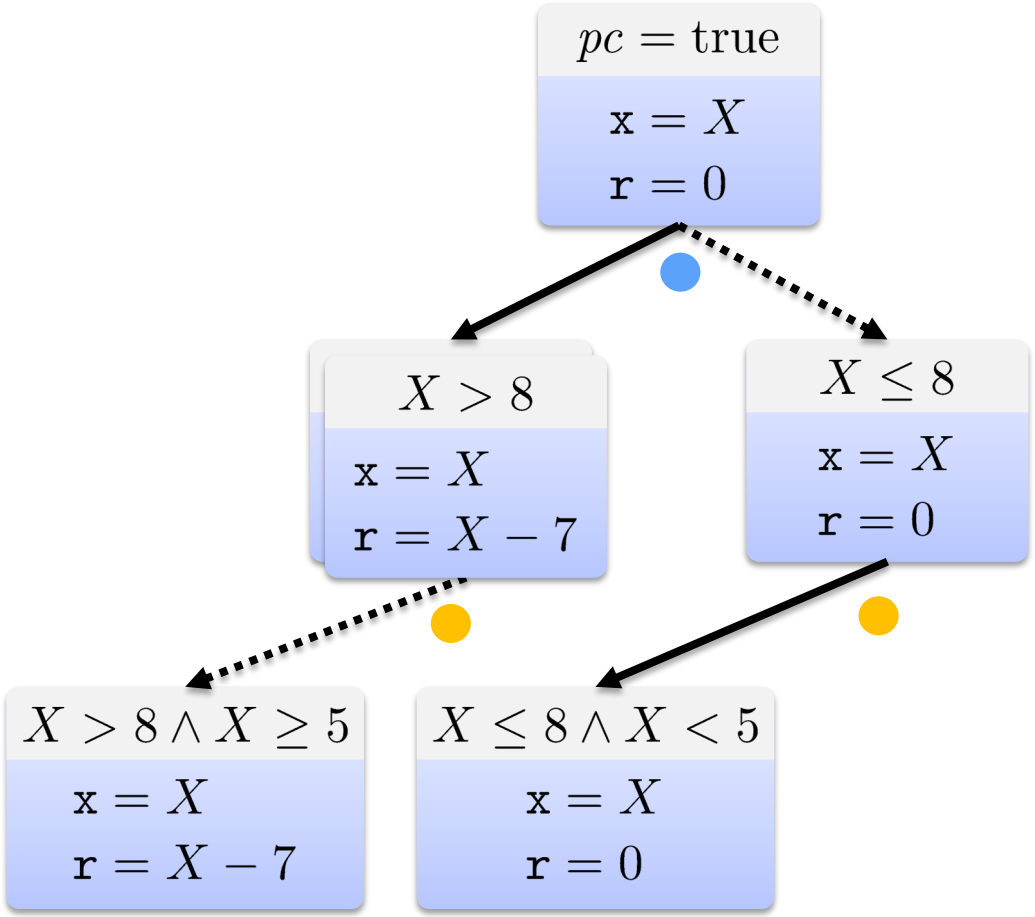
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8 ●) {
6      r = x - 7
7    }
8
9    if (x < 5 ●) {
10     r = x - 2
11   }
12
13   return r
   }
```



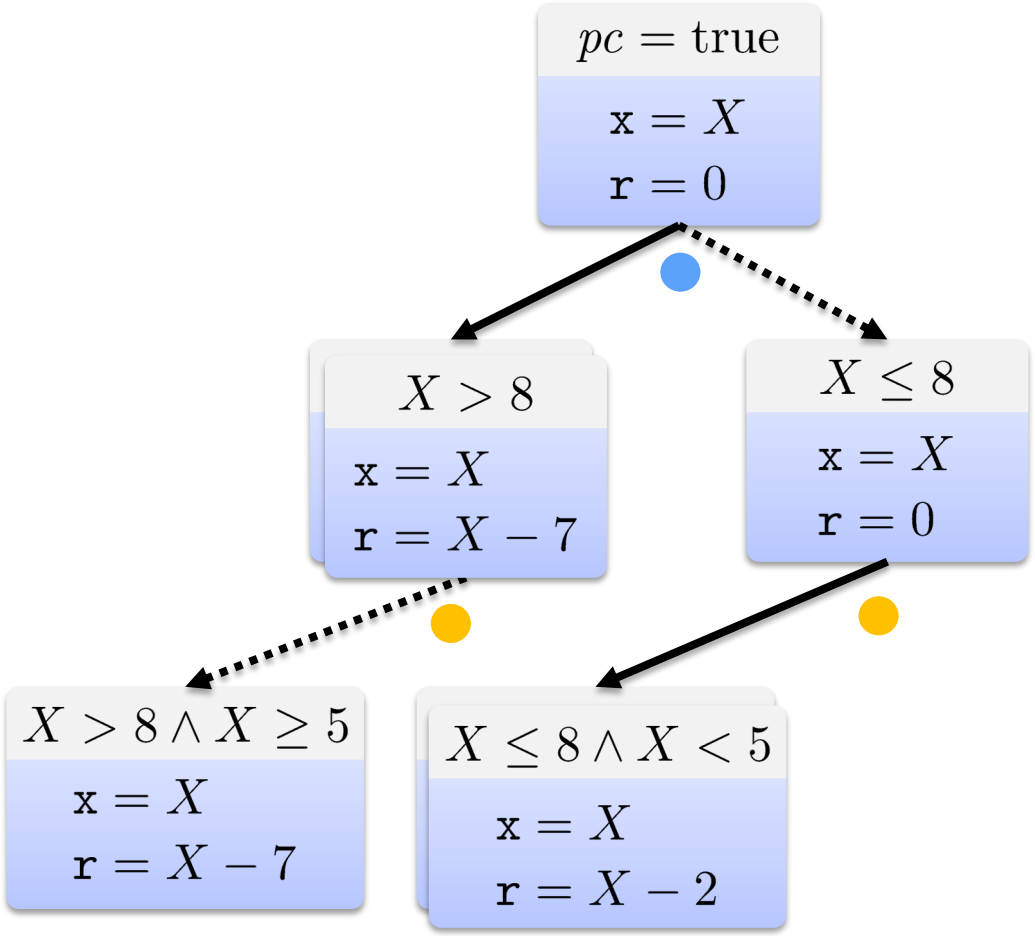
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8 ●) {
6      r = x - 7
7    }
8
9    if (x < 5 ●) {
10     r = x - 2
11   }
12
13   return r
   }
```



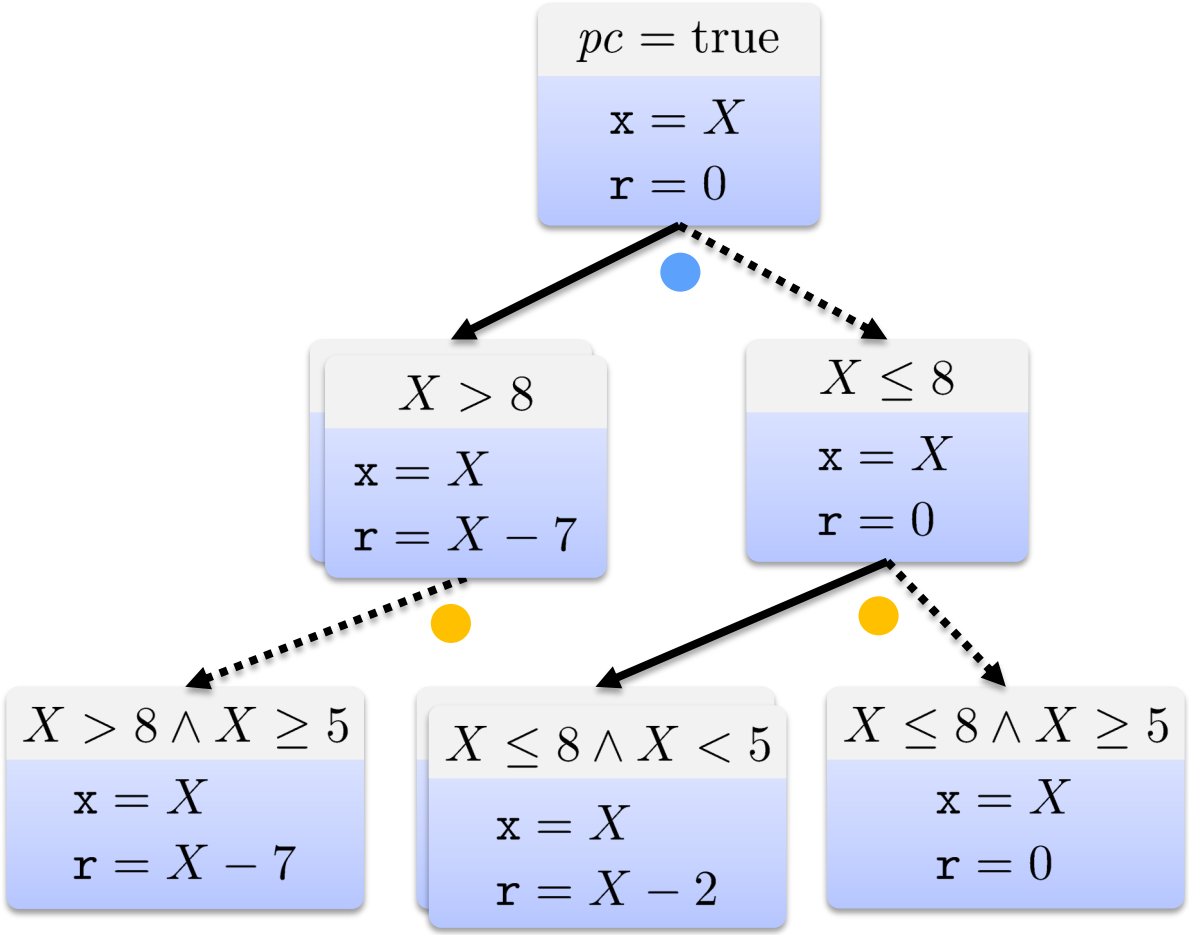
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8 ●) {
6      r = x - 7
7    }
8
9    if (x < 5 ●) {
10     r = x - 2
11   }
12
13   return r
   }
```



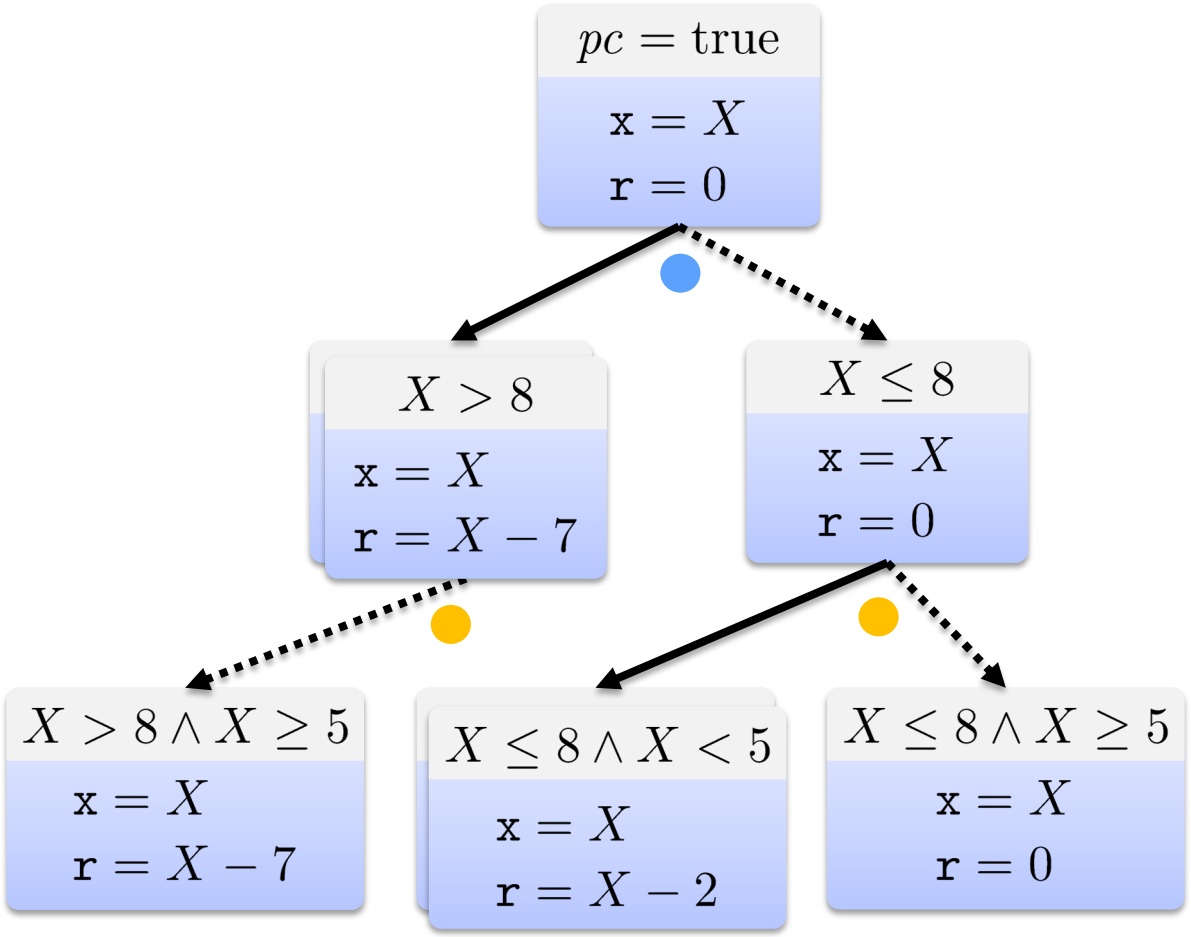
```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8 ●) {
6      r = x - 7
7    }
8
9    if (x < 5 ●) {
10     r = x - 2
11   }
12
13   return r
   }
```



```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8 ●) {
6      r = x - 7
7    }
8
9    if (x < 5 ●) {
10     r = x - 2
11   }
12
13   return r
   }
```



```
1  int proc(int x) {
2
3    int r = 0
4
5    if (x > 8 ●) {
6      r = x - 7
7    }
8
9    if (x < 5 ●) {
10     r = x - 2
11   }
12
13   return r
   }
```



Satisfying assignments:

X = 9 X = 4 X = 7

Test cases:

proc(9) proc(4) proc(7)

Symbolic Execution

Analysis of programs by tracking symbolic rather than actual values

- a form of Static Analysis

Symbolic reasoning is used to reason about *all* the inputs that take the same path through a program

Builds constraints that characterize

- conditions for executing paths
- effects of the execution on program state

Symbolic Execution

Uses symbolic values for input variables.

Builds constraints that characterize the conditions under which execution paths can be taken.

Collects **symbolic path conditions**

- a path condition for a path P is a formula PC such that PC is satisfiable if and only if P is executable

Uses theorem prover (**constraint solver**) to check if a path condition is satisfiable and the path can be taken.

Symbolic Execution: Alternative Explanation

Symbolic execution creates a functional representation of each path in a Control Flow Graph of a program

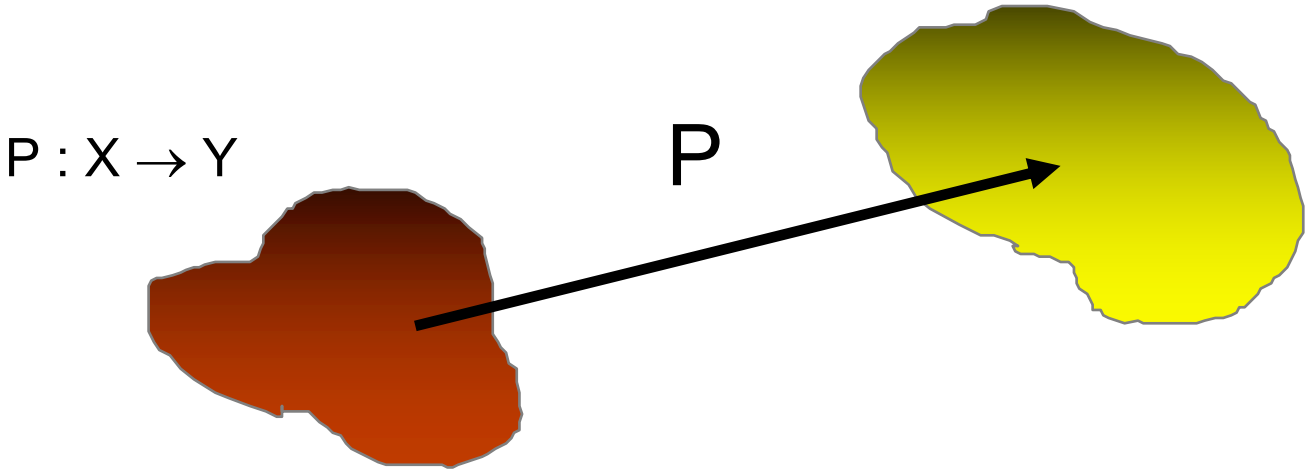
- i.e., each program path becomes a function from inputs to outputs

Notation:

For a program path P (i.e, a sequence of instructions)

- D[P] is the domain for path P
  - the inputs that force the program to take path P_i
- C[P] is the computation for path P
  - the result of executing the path

Program as a Function



Program P is composed of partial functions corresponding to the executable paths

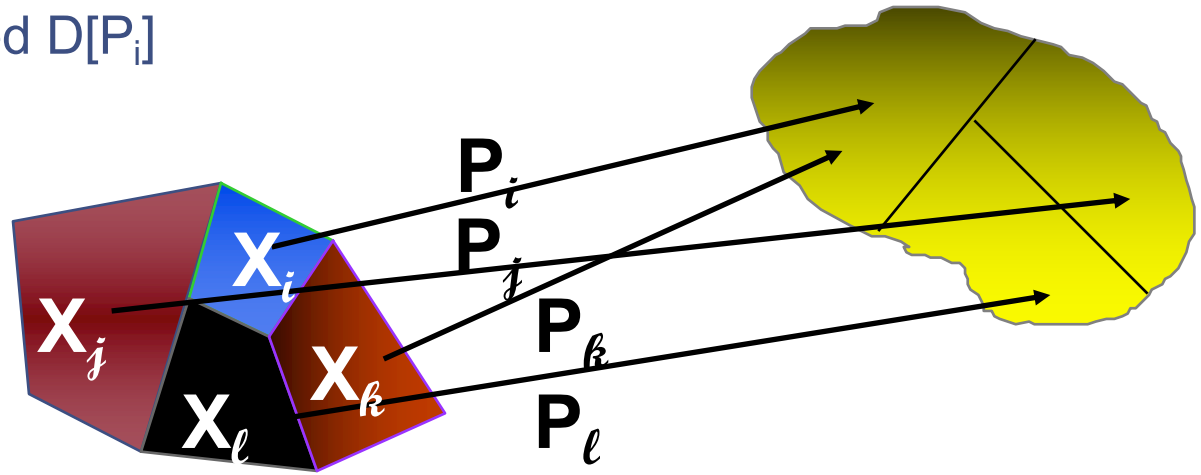
$P = \{P_1,...,P_r\}$

Each partial function $P_i : X_i \rightarrow Y$ maps some part of the input of the program to the output

Effect of Symbolic Execution

X_i is the domain of path P_i

Denoted $D[P_i]$



$X = D[P_1] \cup ... \cup D[P_r] = D[P]$

$D[P_i] \cap D[P_j] = \emptyset,\ i \neq j$

Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```

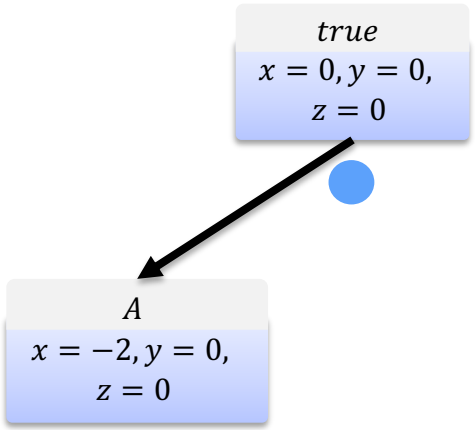
$$true$$
$$x = 0, y = 0, z = 0$$

Common to all states:

$$a = A, b = B, c = C$$

Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```

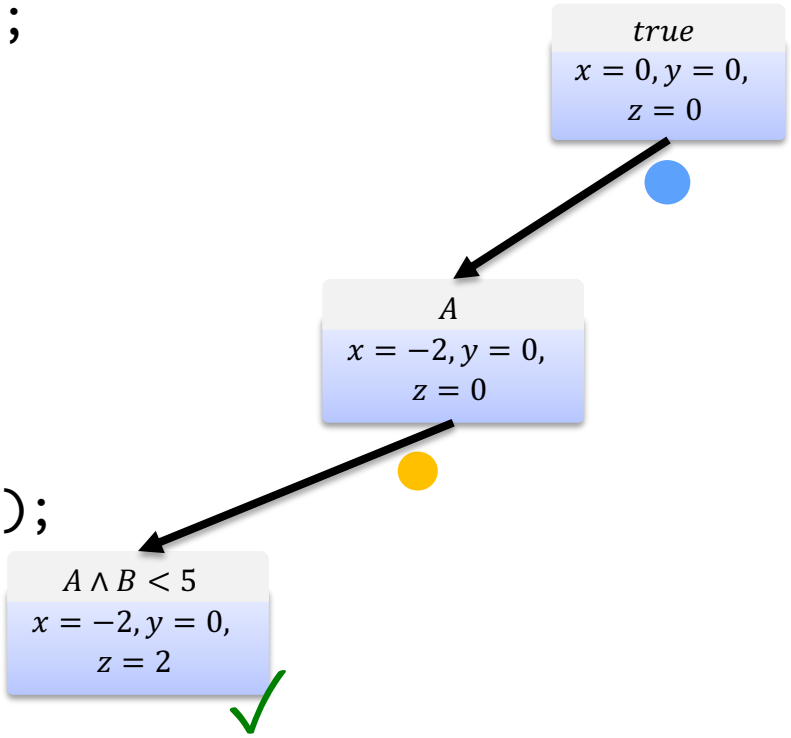


Common to all states:

$a = A, b = B, c = C$

Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```

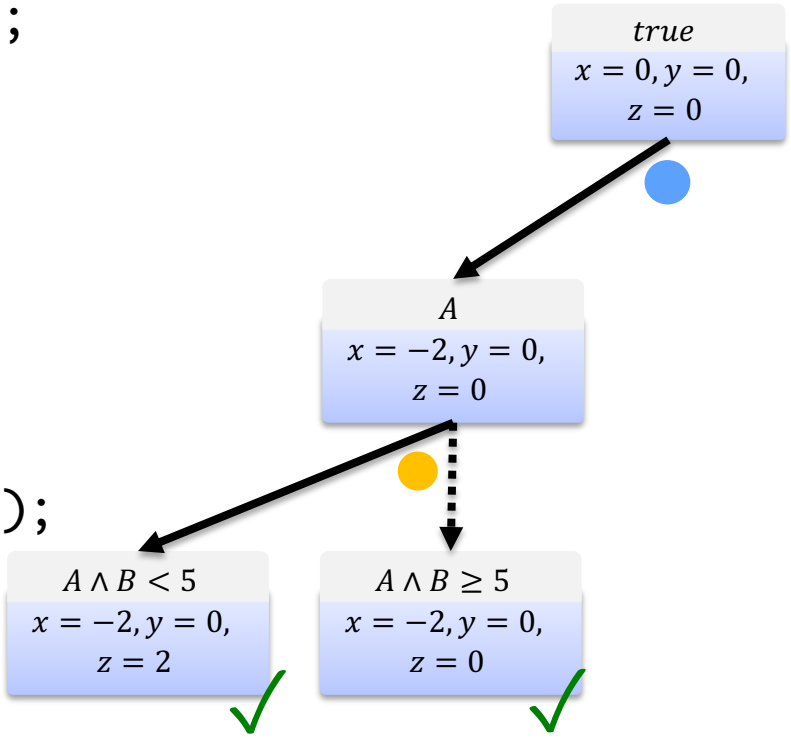


Common to all states:

$a = A, b = B, c = C$

Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```

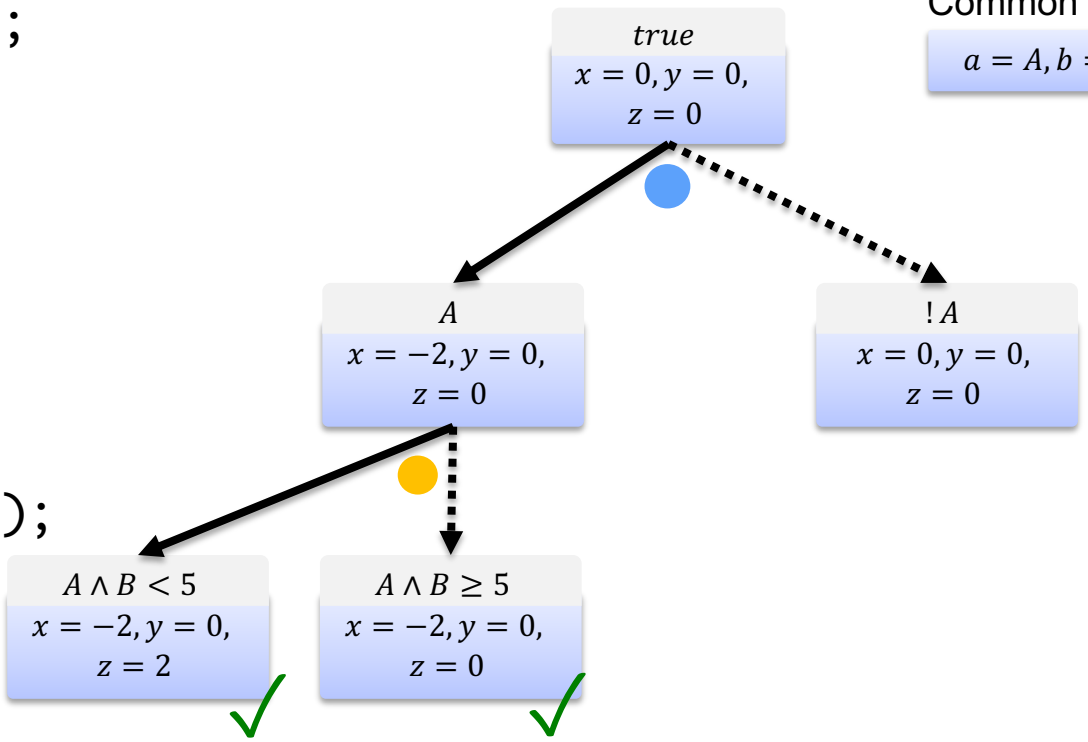


Common to all states:

$a = A, b = B, c = C$

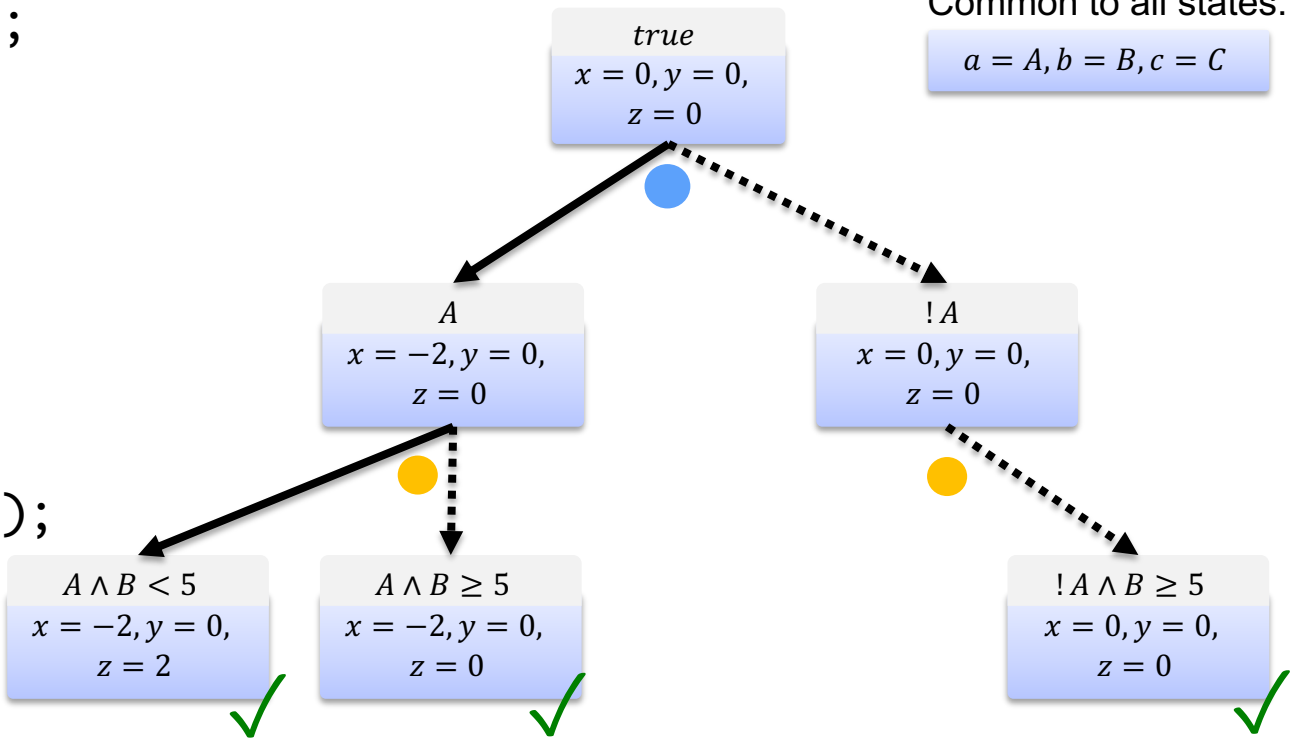
Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



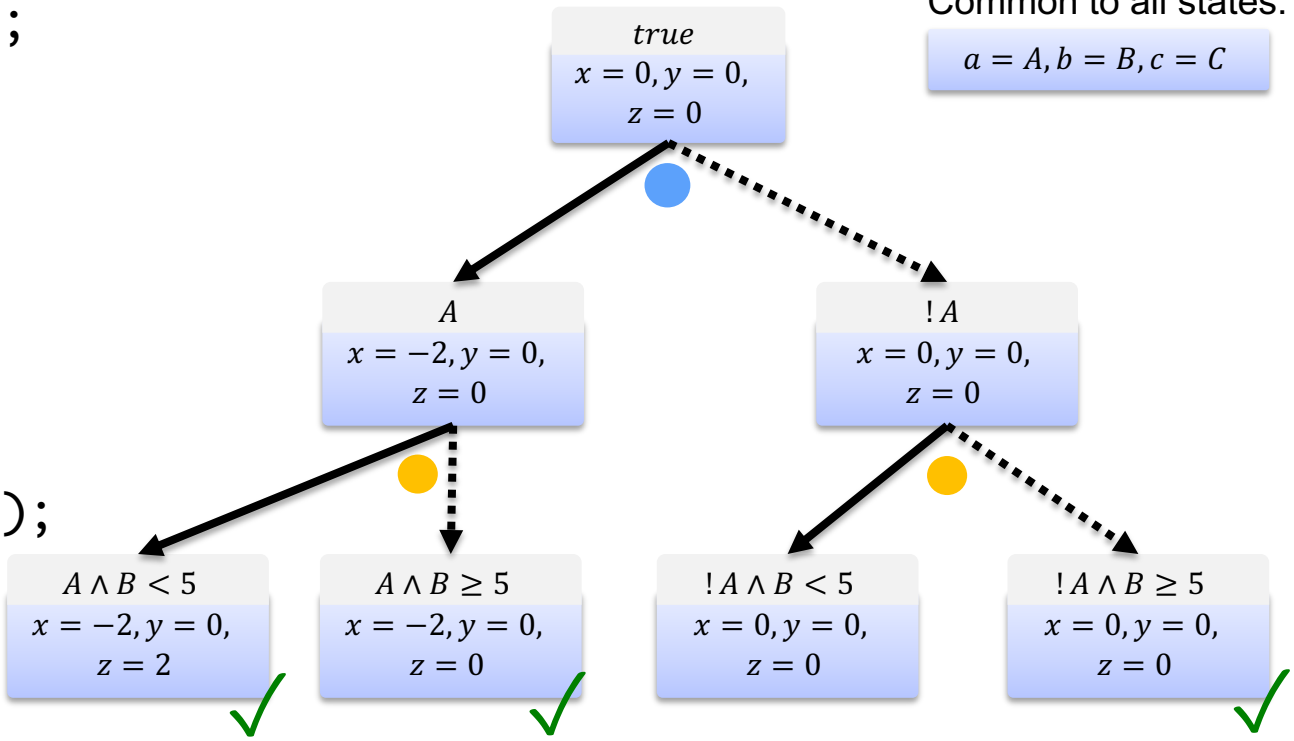
Exercise: Use SymExec to Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



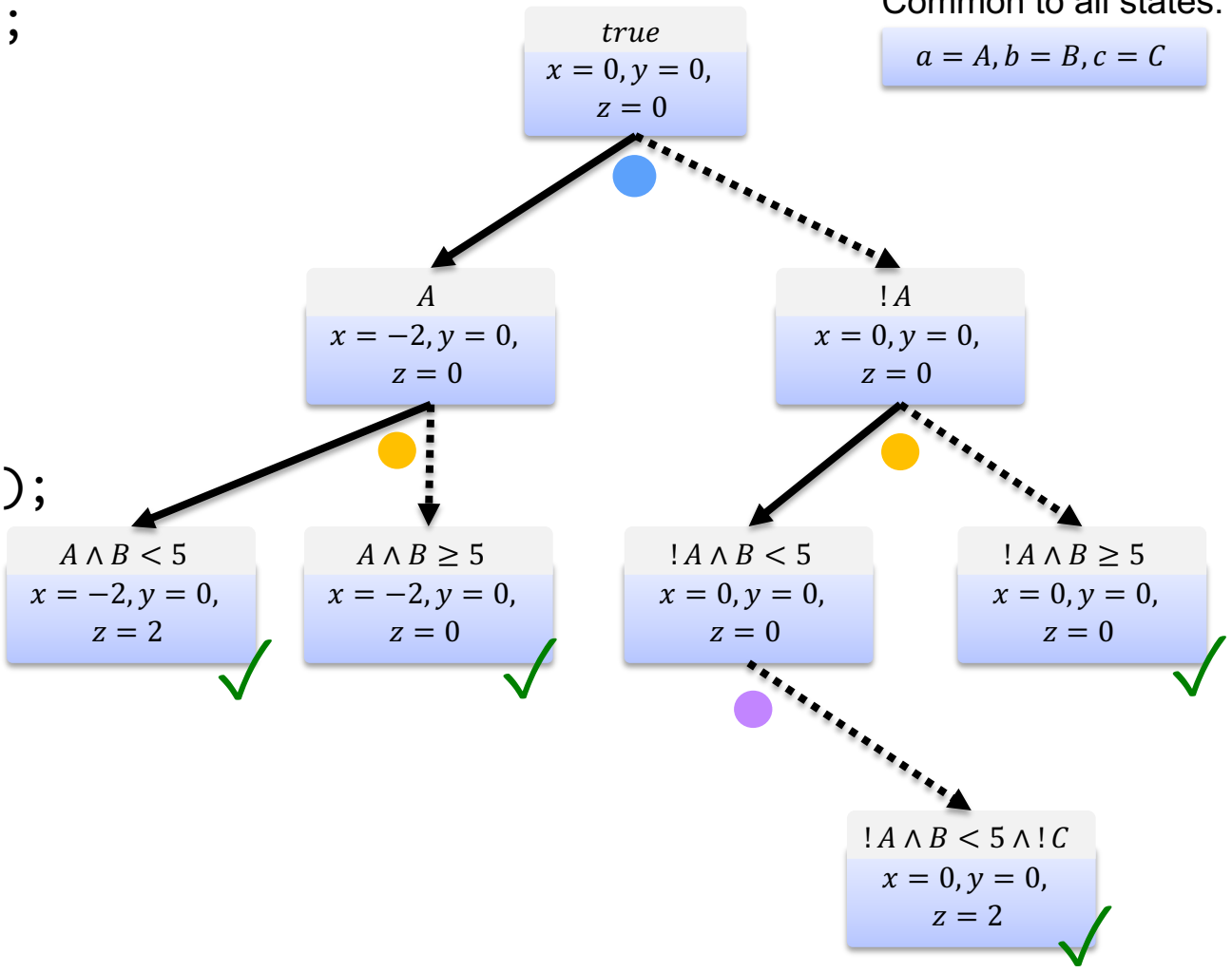
Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



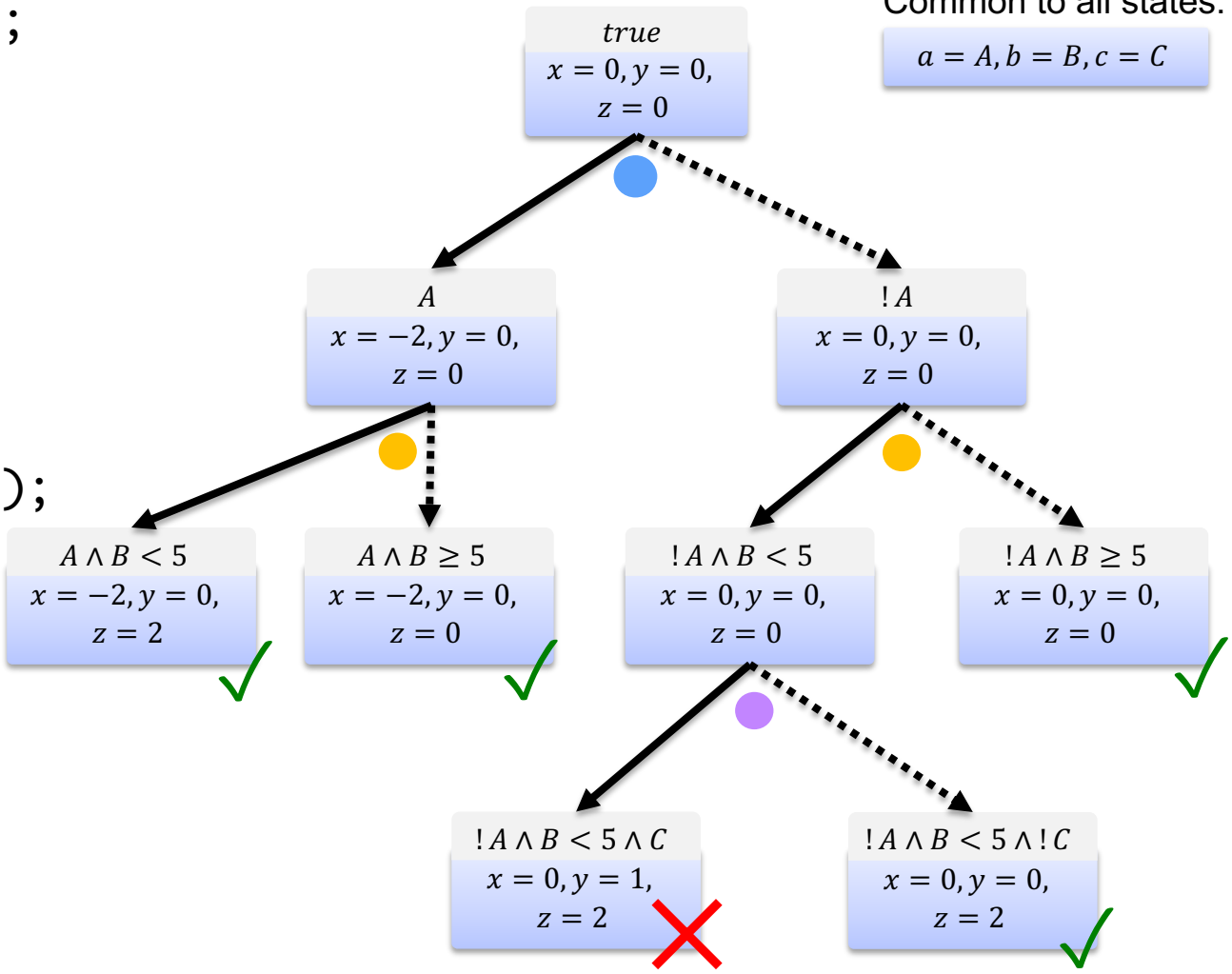
Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



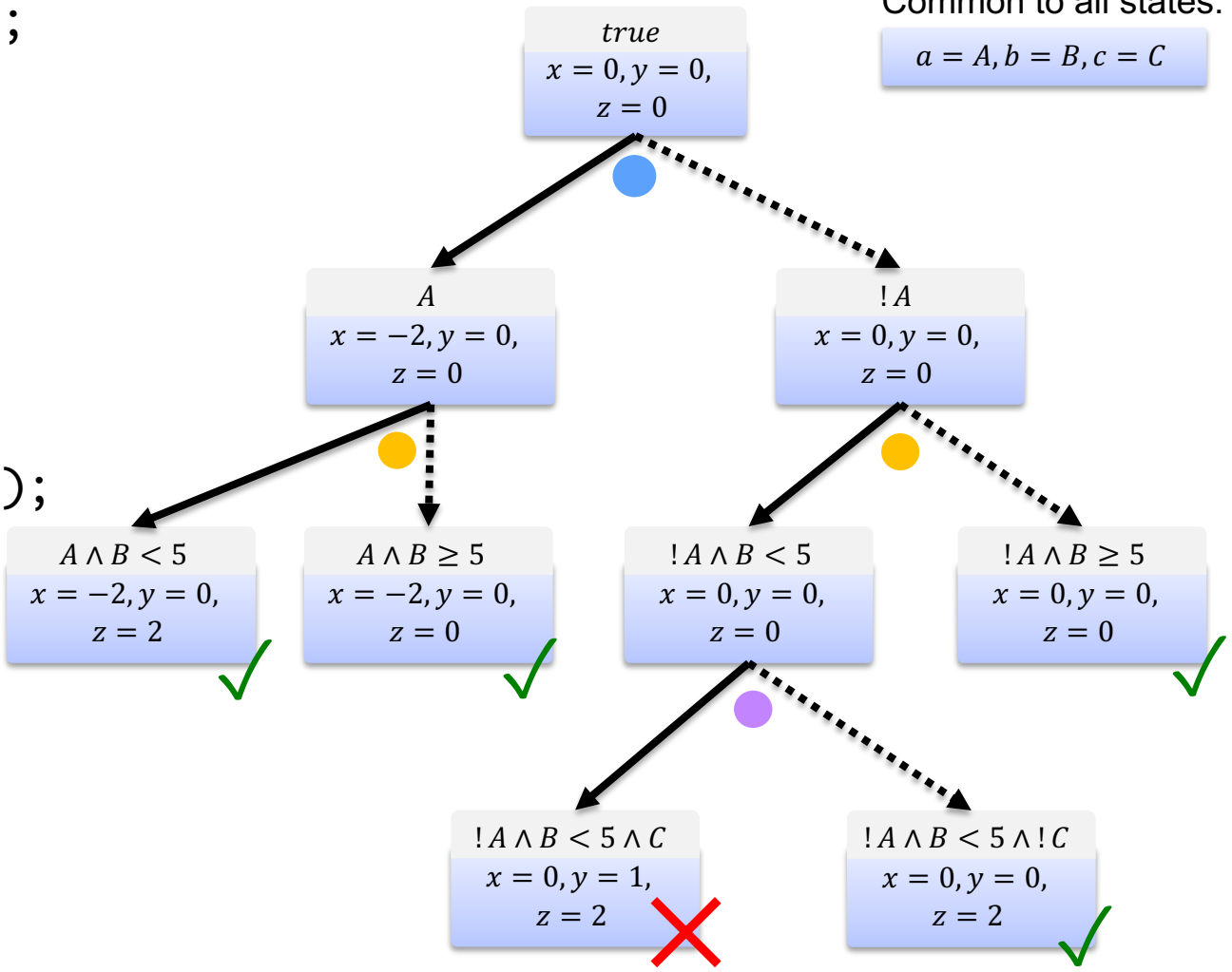
Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



Exercise: Find a Violation

```
int x=0, y=0, z=0;
if (a●) {
  x = -2;
}
if (b < 5●) {
  if (!a && c●)
      { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



a = false, b = 2, c = true -> x = 0, y = 1, z = 2 : assert(0+1+2 != 3)

Finding Bugs using Symbolic Execution

Symbolic execution enumerates paths

- Runs into bugs that trigger whenever path executes
- Assertions, buffer overflows, division by zero, etc., require specific conditions

Assertions (and other oracles) are compiled into conditionals

- Treat assertions as conditions
- Creates explicit error paths
- Bug exists if error() call is reachable

```
assert x != NULL
```

```
if (x == NULL)
    error();
```

Finding Bugs with Symbolic Execution

Instrument program with properties

- Translate any safety property to reachability
- Same as fuzzing

Division by zero

```
y = 100 / x
```

```
assert x != 0
y = 100 / x
```

Buffer overflows

```
a[x] = 10
```

```
assert x >= 0 && x < len(a)
```

Implementation can be explicit or implicit

- explicit: like sanitizers, instrument the code with checks
- Implicit: symbolic execution engine injects extra checks at runtime

Problems of (Classical) Symbolic Execution

Some code is hard to analyze

- it is surprising how hard it can be to solve even very simple-looking constraints
- some code (e.g., crypto hash) is provably hard to invert

Path explosion

- Number of paths is exponential in the size of the program
- control flow, loops, procedures, concurrency, …

Inputs: real code has more than just integers!

- pointers, data structures, …
- files, databases, …
- threads, thread schedules, …
- sockets, …