CPSC 593L: Topics in Programming Languages

Concolic Execution

September 28th, 2022

Instructor: Caroline Lemieux

Term: 2022W1

Class website: carolemieux.com/teaching/CPSC539L_2022w1.html

Recall: Test-Input Generation

- Assume a program *P* which takes in input *i*
- Goal of automated Test-Input Generation:
  - Given *P*, generate inputs *i* which expose bugs… or other interesting behaviors

Recall: Approaches to Test Input Generation

- Test-input generation
  - *Generate test inputs that expose bugs in a program*

Fuzzing, Concolic + Symbolic Execution

- Test case / Test Suite Generation
  - *Generate test suites that expose bugs in a program*

How can we find the error?

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

How can we find the error?

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Random fuzzing over ints?

How can we find the error?

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Coverage-guided fuzzing?

How can we find the error?

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Coverage-guided fuzzing?

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) {
              ERROR;
        }
  }
}
```

Concolic Testing

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

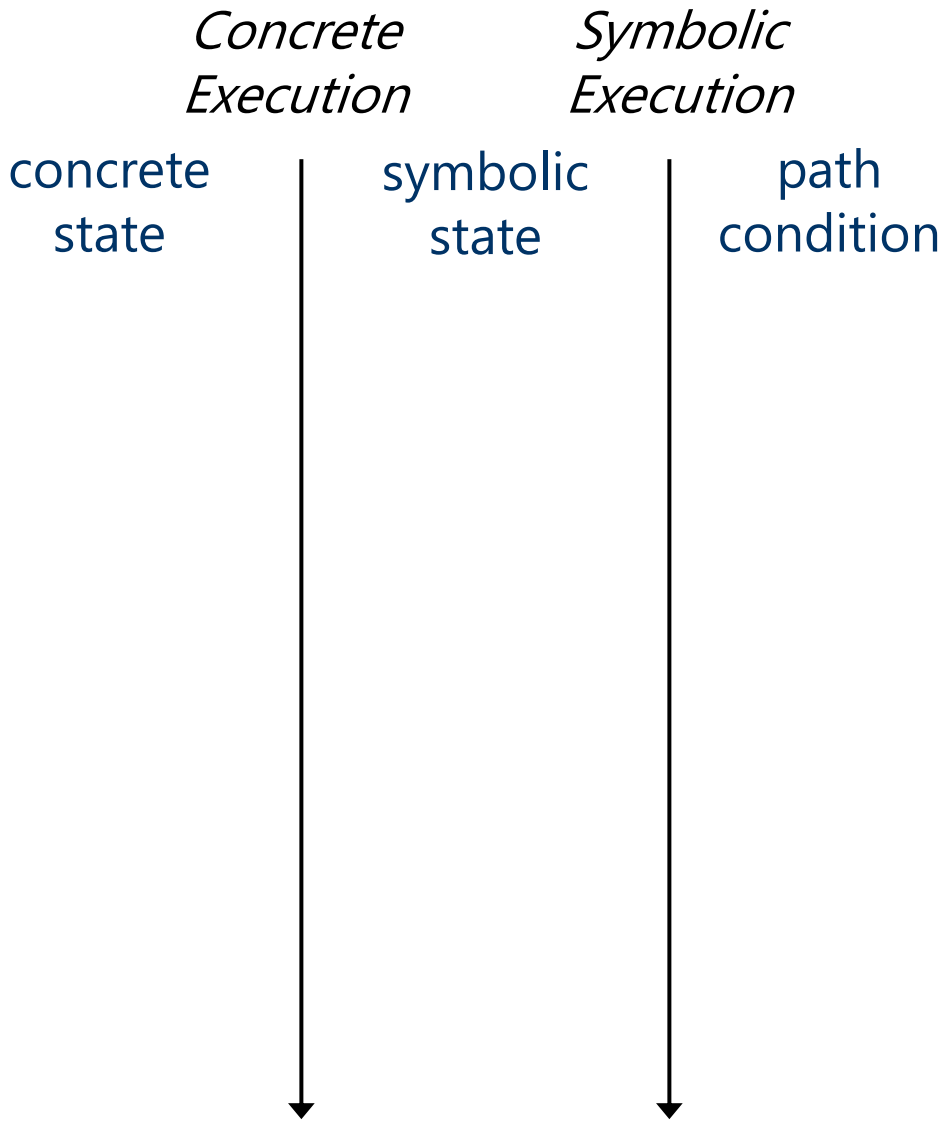
"seed" with concrete input:
$x = 22, y = 7$

```
void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

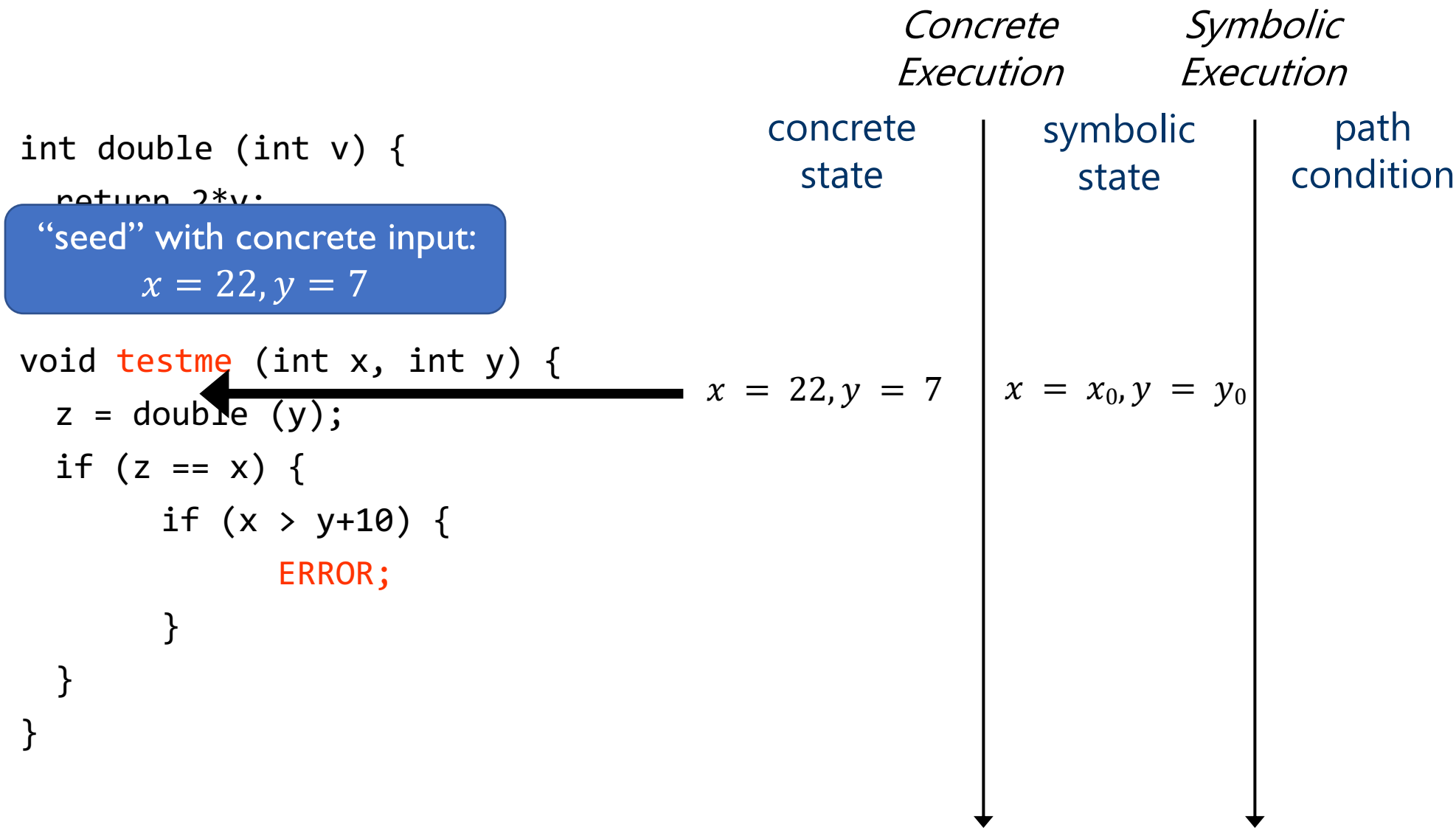
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

"seed" with concrete input:
$x = 22, y = 7$

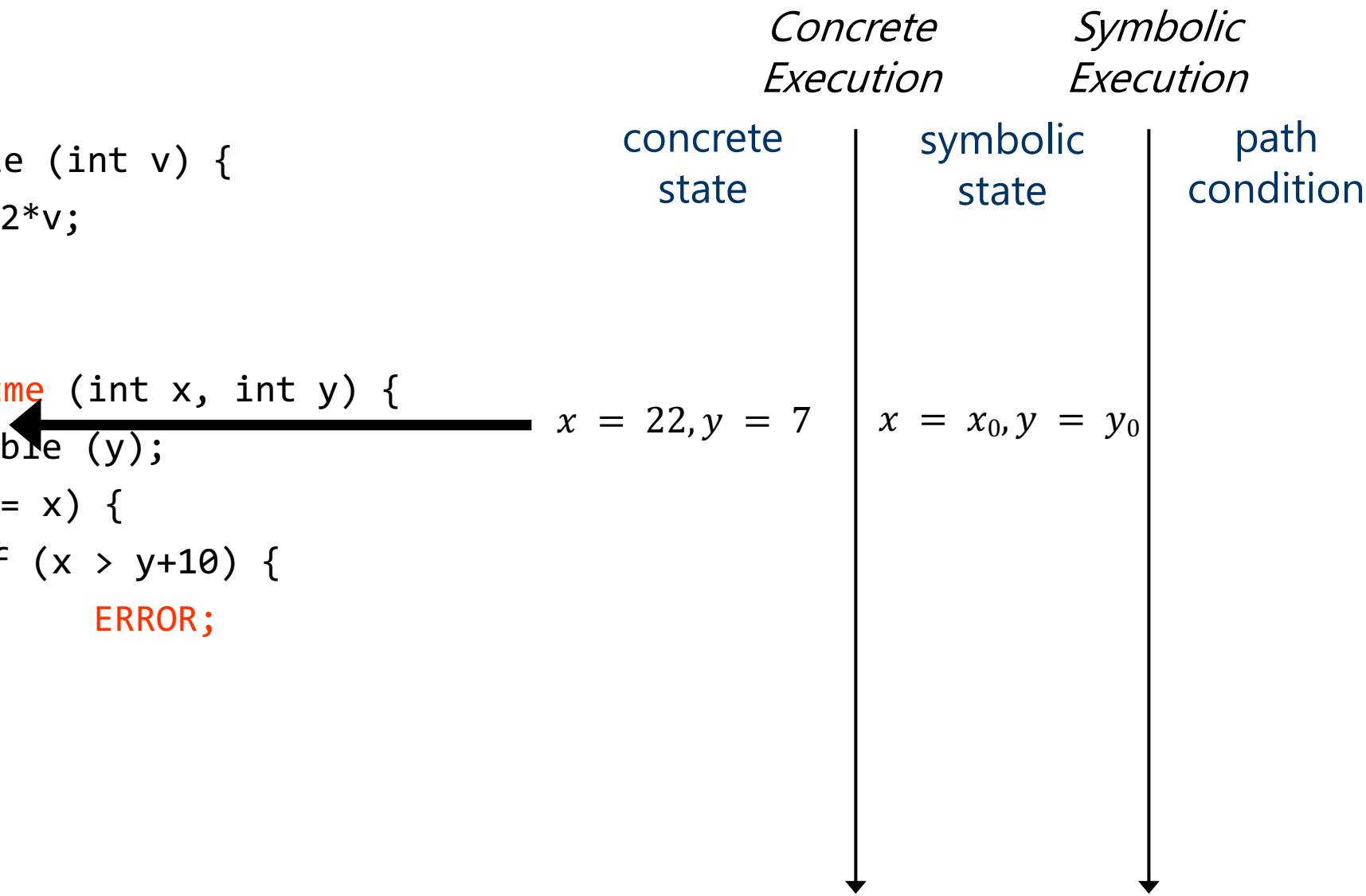


Concolic Testing



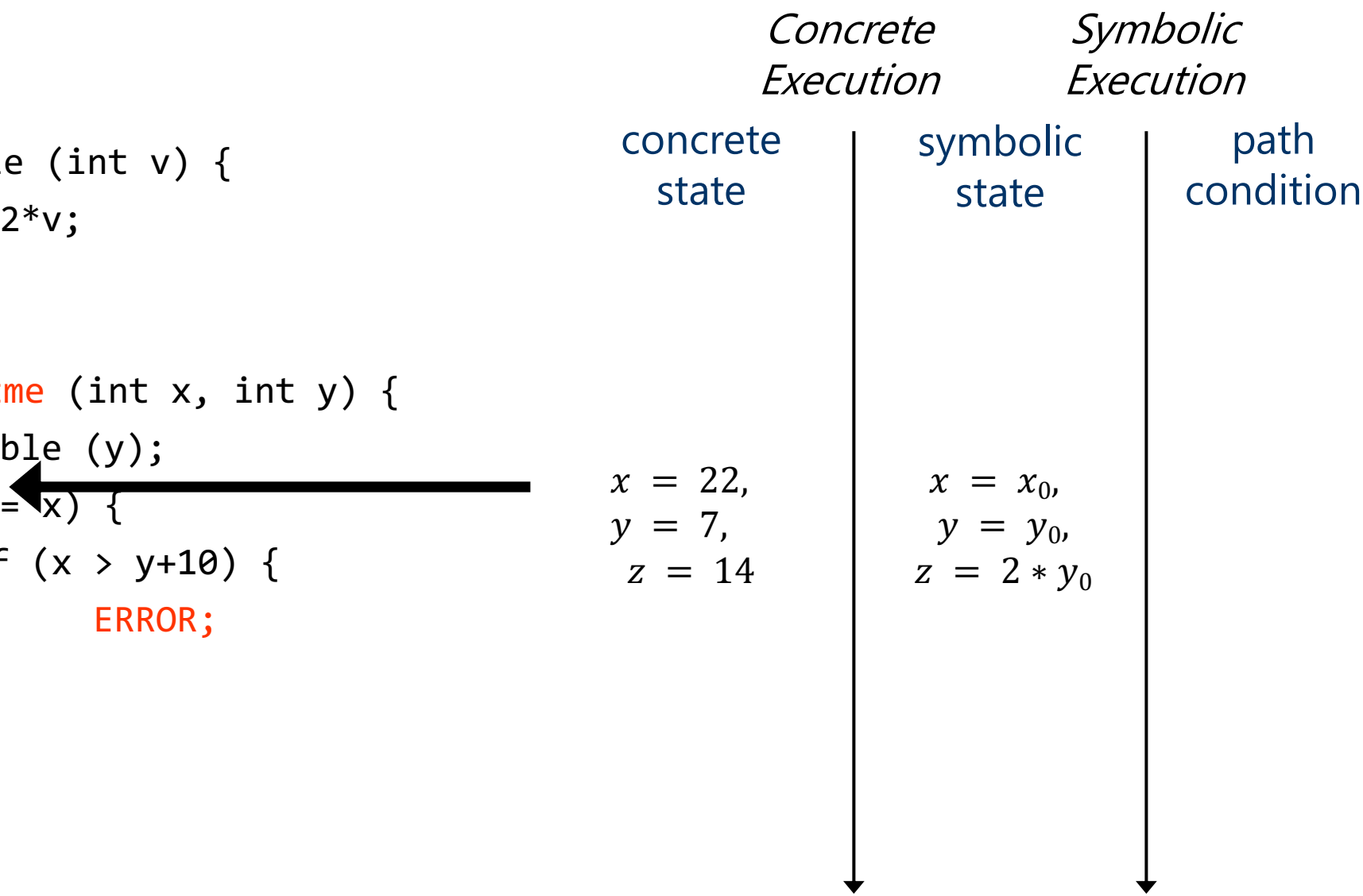
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```



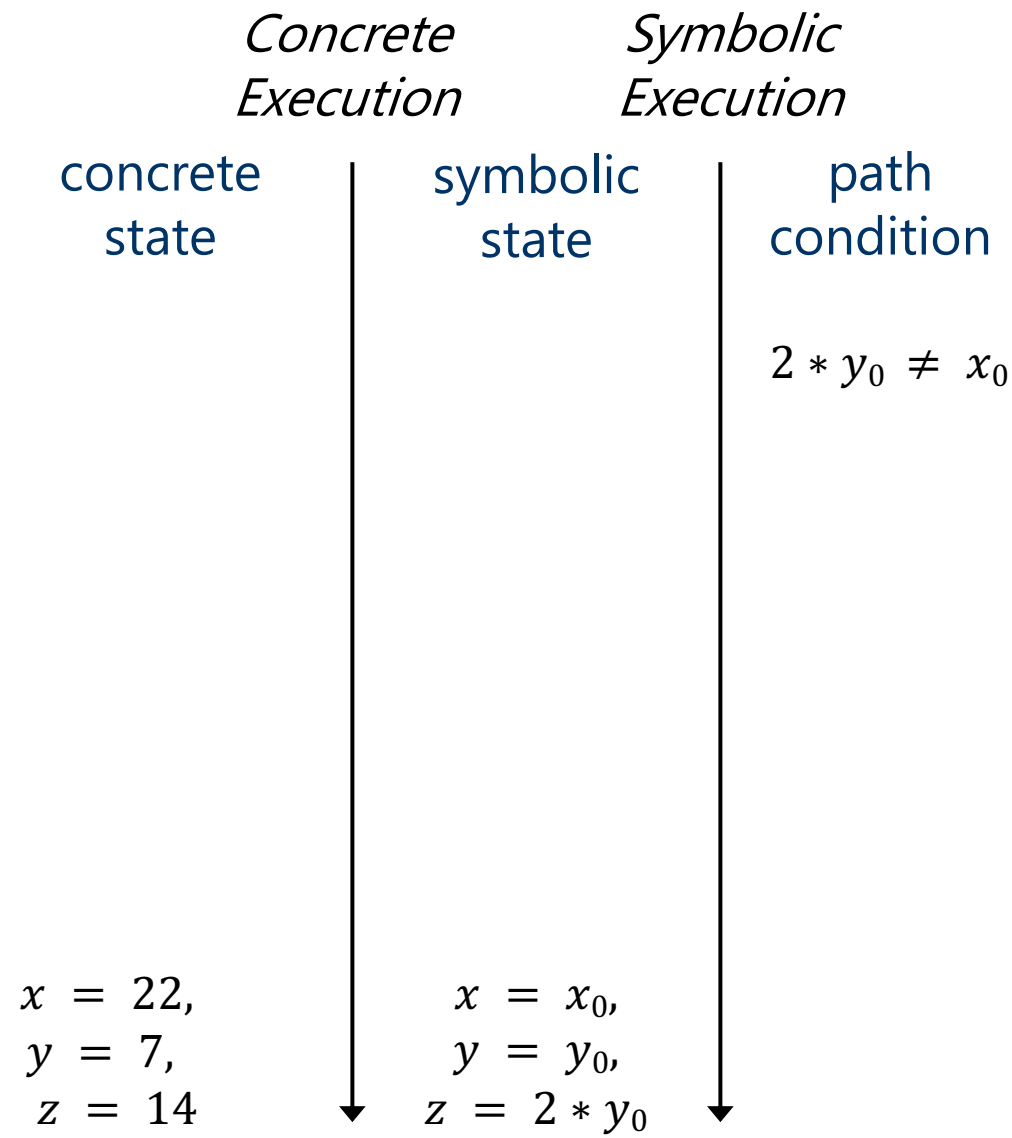
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

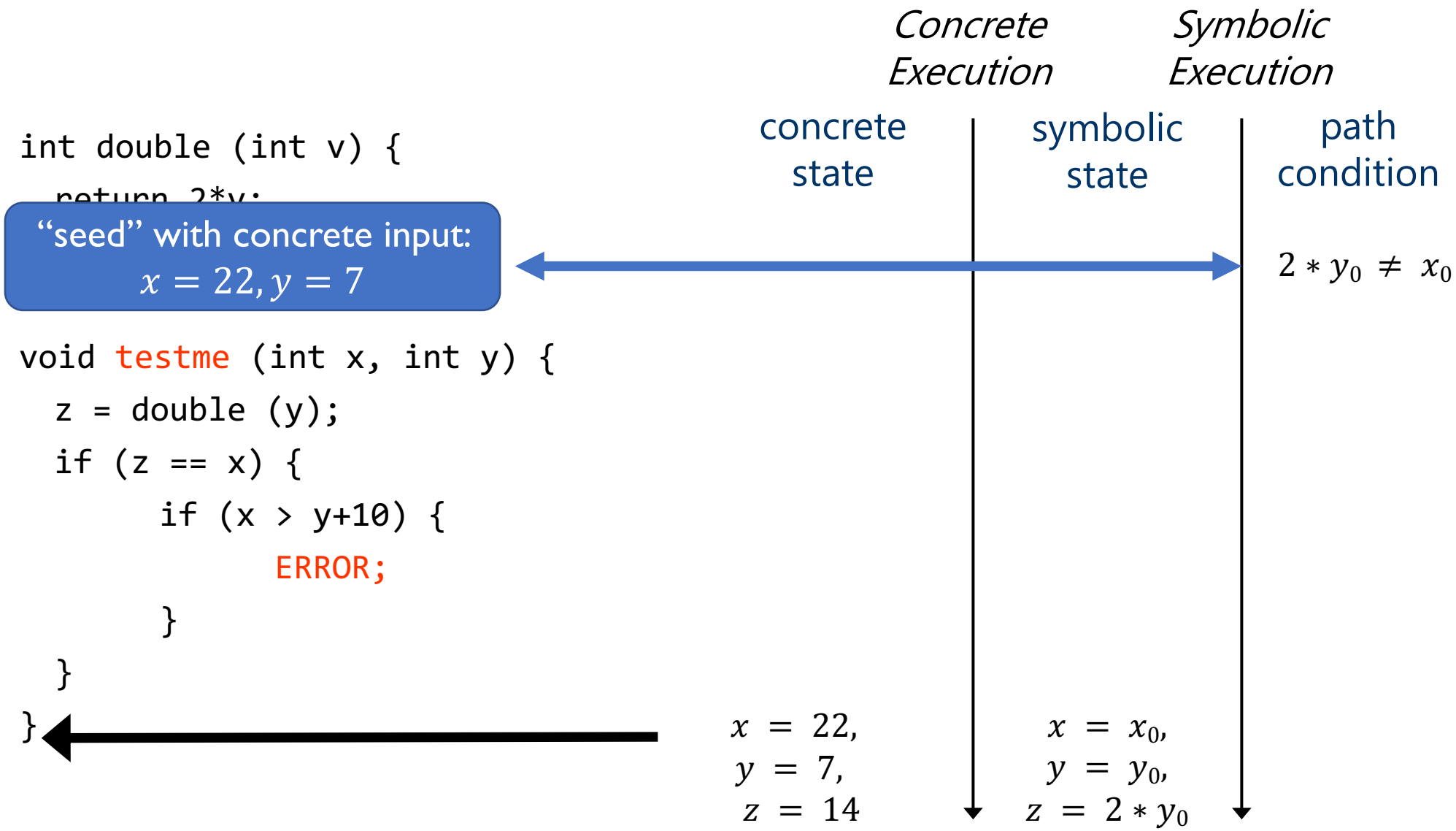


Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```



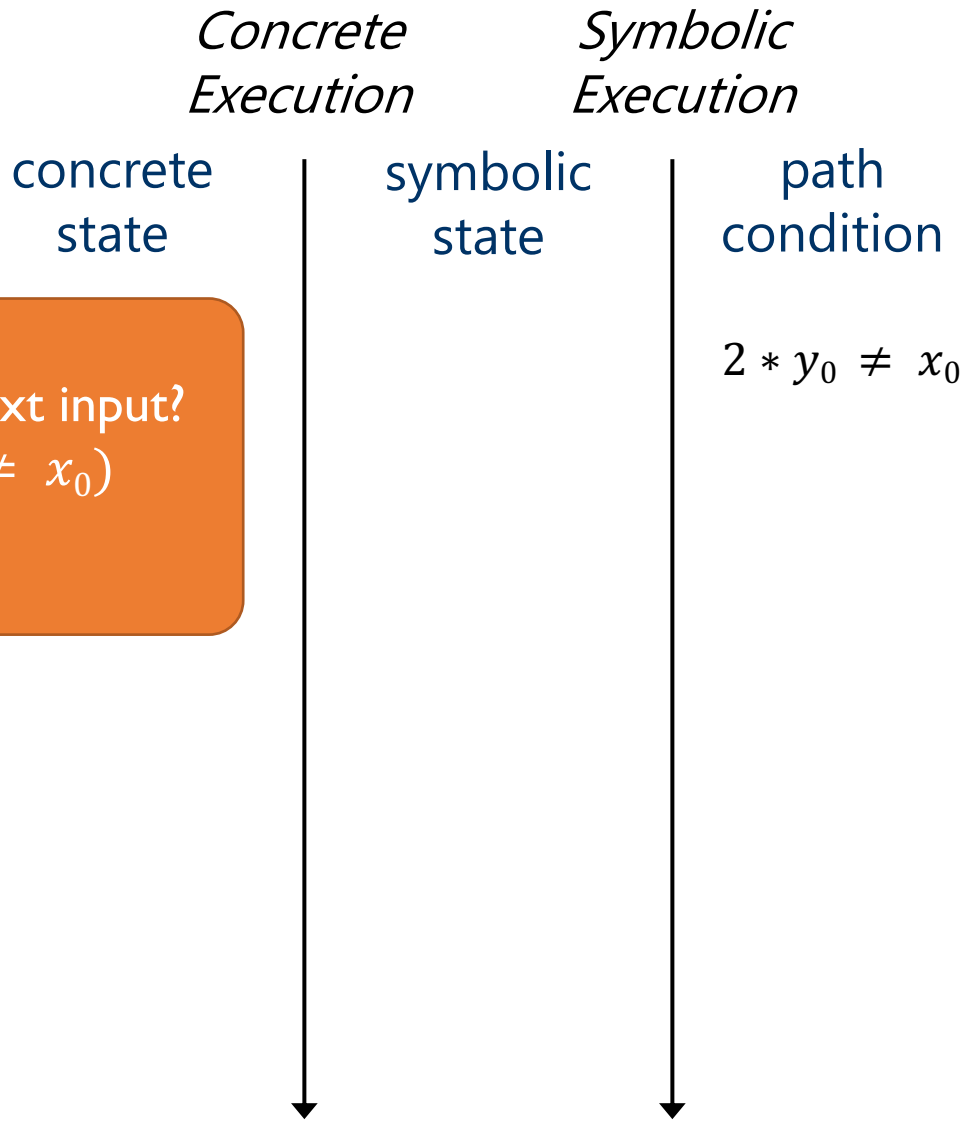
Concolic Testing



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) {
              ERROR;
        }
  }
}
```

How to generate next input?
Solve: $\sim(2 * y_0 \neq x_0)$



Concolic Testing

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

void testme (int x,
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
		$2 * y_0 \neq x_0$

How to generate next input?
Solve: $\sim(2 * y_0 \neq x_0)$
A.k.a: $2 * y_0 = x_0$

Recall: SAT

Find an assignment to Boolean variables a, b, c s.t.

$$a \wedge (\sim \mathrm{b} \vee c)$$

is true.

What about

$$a \wedge (\sim \mathrm{b} \vee c) \wedge (\mathrm{b} \vee \sim \mathrm{a})$$

?

SMT

- Satisfiability Modulo Theories = SAT + extra logics
- E.g. SAT + linear inequalities:

$$(x \geq 8) \wedge (\sim(y \geq 2) \vee (x + y \geq -3))$$

Complexity

- SAT + SMT are both NP-Complete
- In practice, modern SAT and SMT solvers can often work well on "non-pathological" SAT/SMT formulae

SMT Competition

Table 12: Best Main Track solvers (by division)

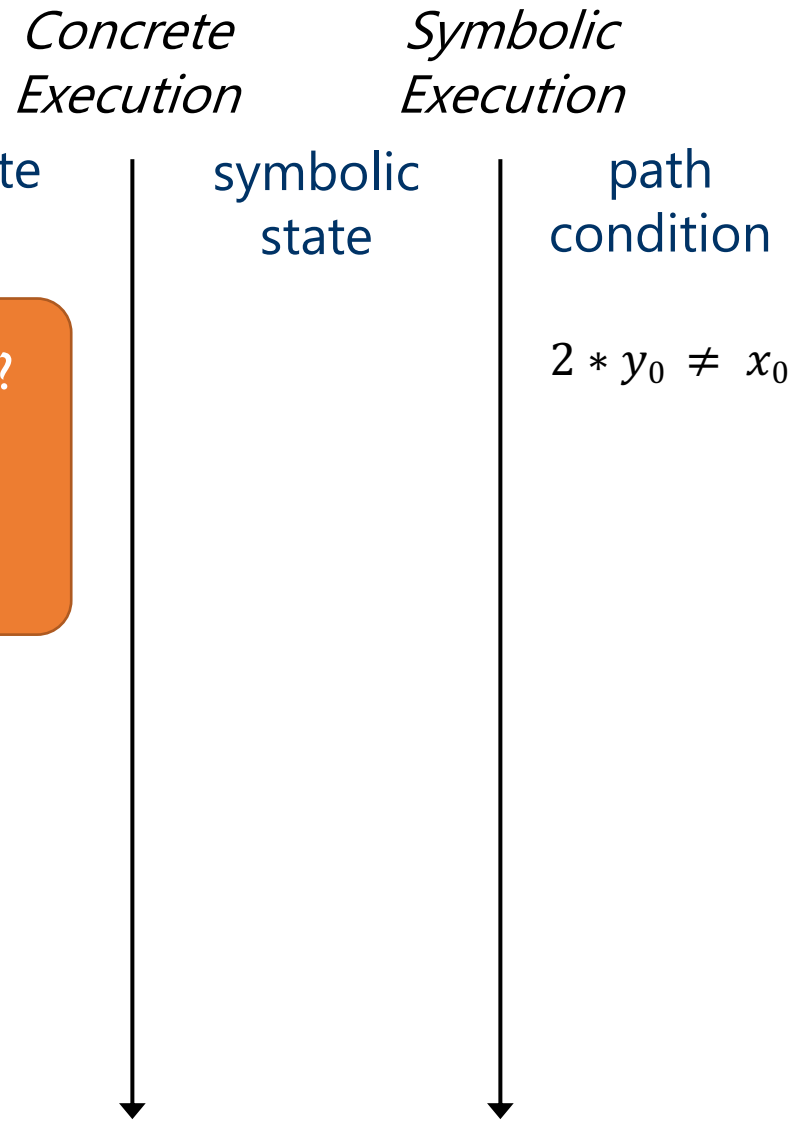
Division	2015	2016	2017	2018
ABVFP				CVC4
ALIA	CVC4 [Z3]	CVC4 [Z3]	CVC4 [Z3]	CVC4 [Z3]
AUFBVDTLIA			CVC4	CVC4
AUFDTLIA			CVC4	CVC4
AUFLIA	CVC4	CVC4	CVC4	CVC4
AUFLIRA	CVC4 [Z3]	Vampire [Z3]	Vampire [Z3]	CVC4 [Z3]
AUFNIRA	CVC4	Vampire	Vampire	CVC4
BV	CVC4 [Z3]	Q3B	Q3B [Z3]	CVC4
BVFP				CVC4
FP				CVC4
LIA	CVC4	CVC4	CVC4 [Z3]	CVC4 [Z3]
LRA	CVC4	CVC4	CVC4 [Z3]	CVC4 [Z3]
NIA	CVC4 [Z3]	ProB [Z3]	CVC4 [Z3]	CVC4 [Z3]
NRA	CVC4	Vampire	Redlog	Vampire [Z3] | Vampire
QF_ABV	Boolector	Boolector	Boolector	Boolector
QF_ABVFP			—	CVC4
QF_ALIA	Yices	Yices	Yices	Yices
QF_ANIA	CVC4 [Z3]	CVC4	CVC4	CVC4 [Z3]
QF_AUFBV	CVC4 [MathSAT]	CVC4 [MathSAT]	Yices [MathSAT]	CVC4
QF_AUFLIA	Yices	Yices	Yices	Yices
QF_AUFNIA	CVC4	CVC4	CVC4 [Z3]	CVC4 [Z3]
QF_AX	Yices	Yices	Yices	Yices
QF_BV	Boolector	Boolector	Boolector | MinkeyRink	Boolector | MinkeyRink
QF_BVFP	Z3	[Z3]	COLIBRI [Z3]	CVC4
QF_DT			CVC4	CVC4
QF_FP	Z3	[MathSAT]	COLIBRI [Z3]	COLIBRI
QF_IDL	Yices [Z3]	Yices [Z3]	Yices	Yices
QF_LIA	CVC4 [MathSAT]	CVC4 [MathSAT]	CVC4 [MathSAT]	SPASS-SATT
QF_LIRA	Yices	Yices	Yices [Z3]	Yices [Z3]
QF_LRA	CVC4	CVC4	CVC4	CVC4
QF_NIA	AProVE [Z3]	Yices [Z3]	CVC4	CVC4
QF_NIRA	CVC4	CVC4	SMT-RAT	SMT-RAT
QF_NRA	Yices [Z3]	Yices [Z3]	Yices	Yices [Z3]
QF_RDL	Yices	Yices	Yices	Yices
QF_SLIA				CVC4
QF_UF	Yices	Yices	Yices	Yices
QF_UFBV	Boolector	Boolector	Boolector	Boolector
QF_UFIDL	Yices	Yices	Yices	Yices
QF_UFLIA	Yices [Z3]	Yices [Z3]	Yices	Yices
QF_UFLRA	Yices	Yices	Yices	Yices
QF_UFNIA	CVC4	Yices | CVC4	Yices	Yices
QF_UFNRA	CVC3 [Z3]	Yices	Yices [Z3]	Yices
UF	CVC4	CVC4	Vampire	CVC4 | Vampire
UFBV	CVC4 [Z3]	CVC4 [Z3]	CVC4 [Z3]	CVC4 [Z3]
UFDT			CVC4	CVC4
UFDTLIA			Vampire	CVC4
UFIDL	CVC4 [Z3]	CVC4	CVC4	CVC4 [Z3]
UFLIA	CVC4	CVC4	CVC4	CVC4
UFLRA	CVC3	Vampire [Z3]	CVC4 [Z3]	CVC4 [Z3]
UFNIA	CVC4	Vampire	Vampire	Vampire [Z3] | Vampire

Weber, Tjark et al. 'The SMT Competition 2015–2018'. 1 Jan. 2019 : 221 – 259.

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) {
              ERROR;
        }
  }
}
```

How to generate next input?
Solve: $\sim(2 * y_0 \neq x_0)$
A.k.a: $2 * y_0 = x_0$



Concolic Testing

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

void testme (int
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Concrete Execution

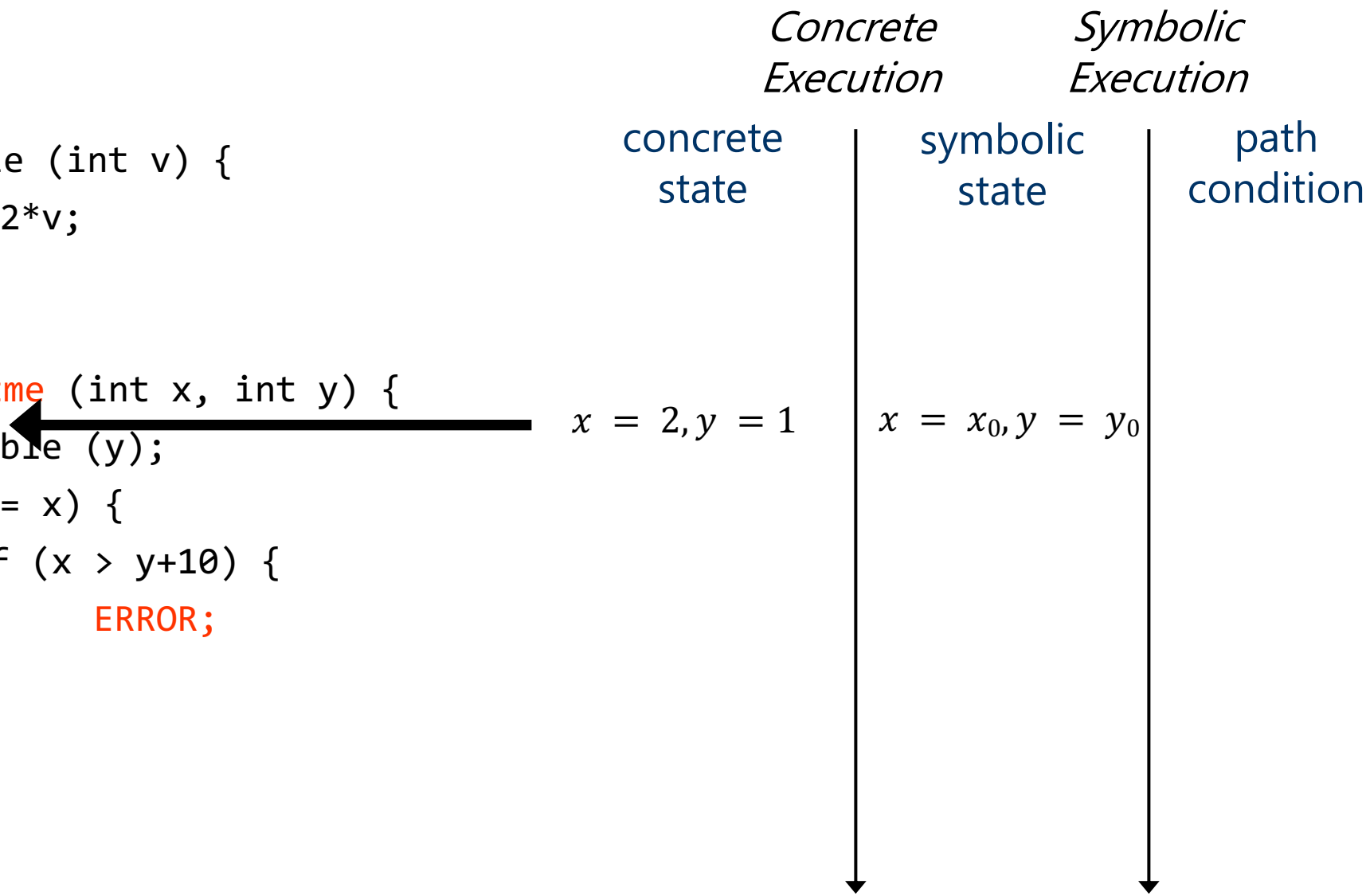
Symbolic Execution

concrete state	symbolic state	path condition
		$2 * y_0 \neq x_0$

How to generate next input?
Solve: $\sim(2 * y_0 \neq x_0)$
A.k.a: $2 * y_0 = x_0$
(possible) solution: $x = 2, y = 1$

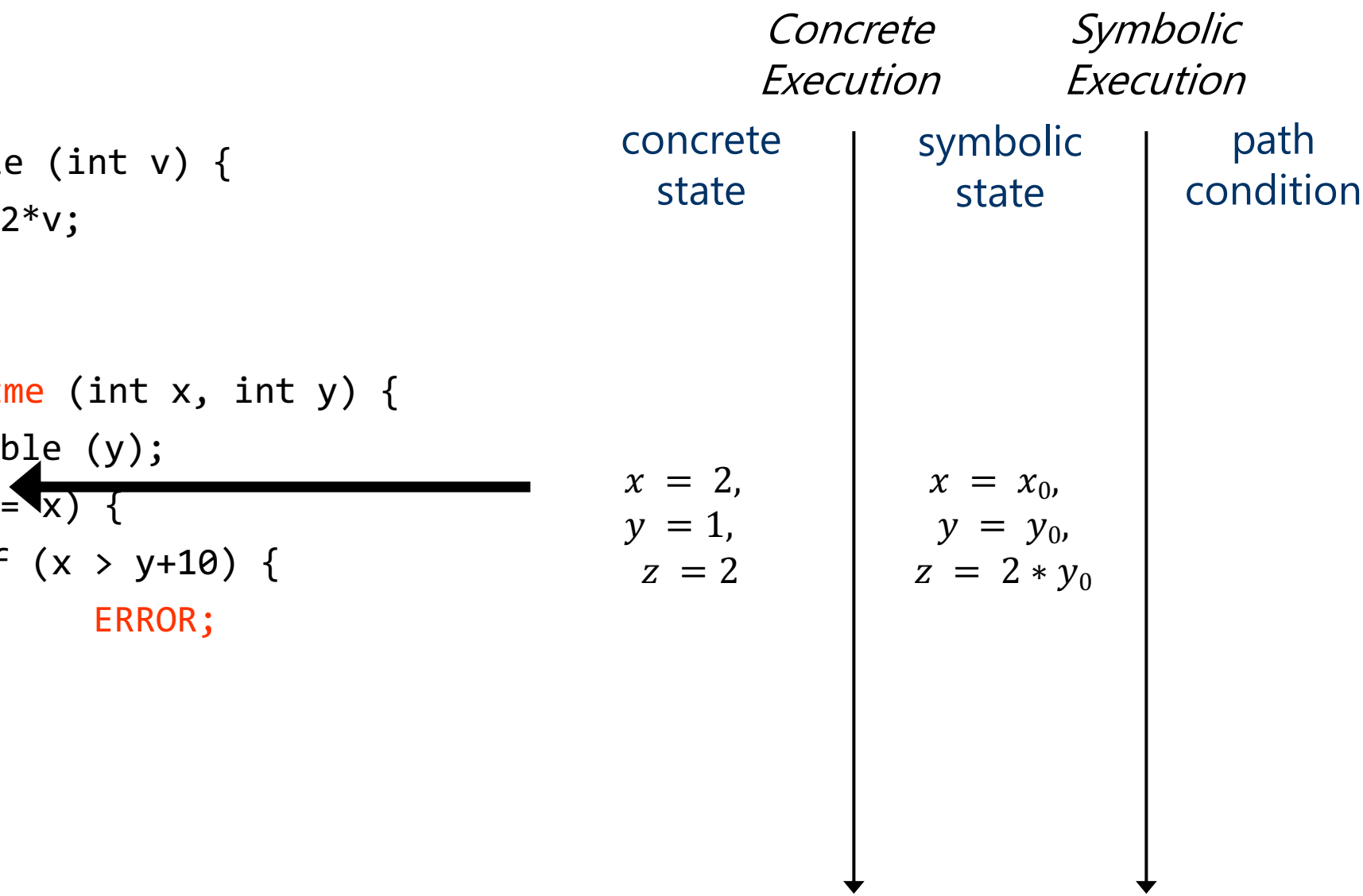
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```



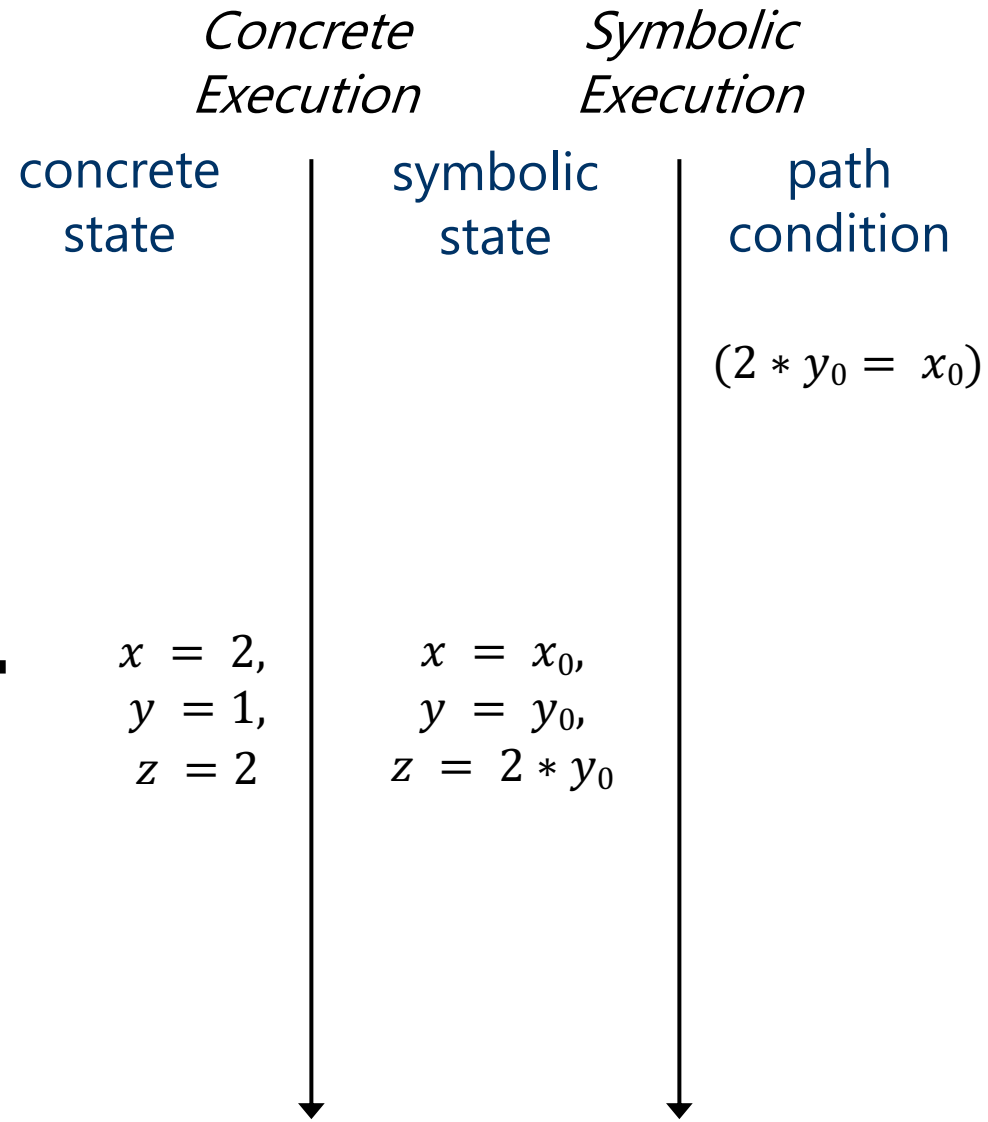
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```



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) {
            ERROR;
       }
  }
}
```



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) {
              ERROR;
        }
  }
}
```

Concrete Execution

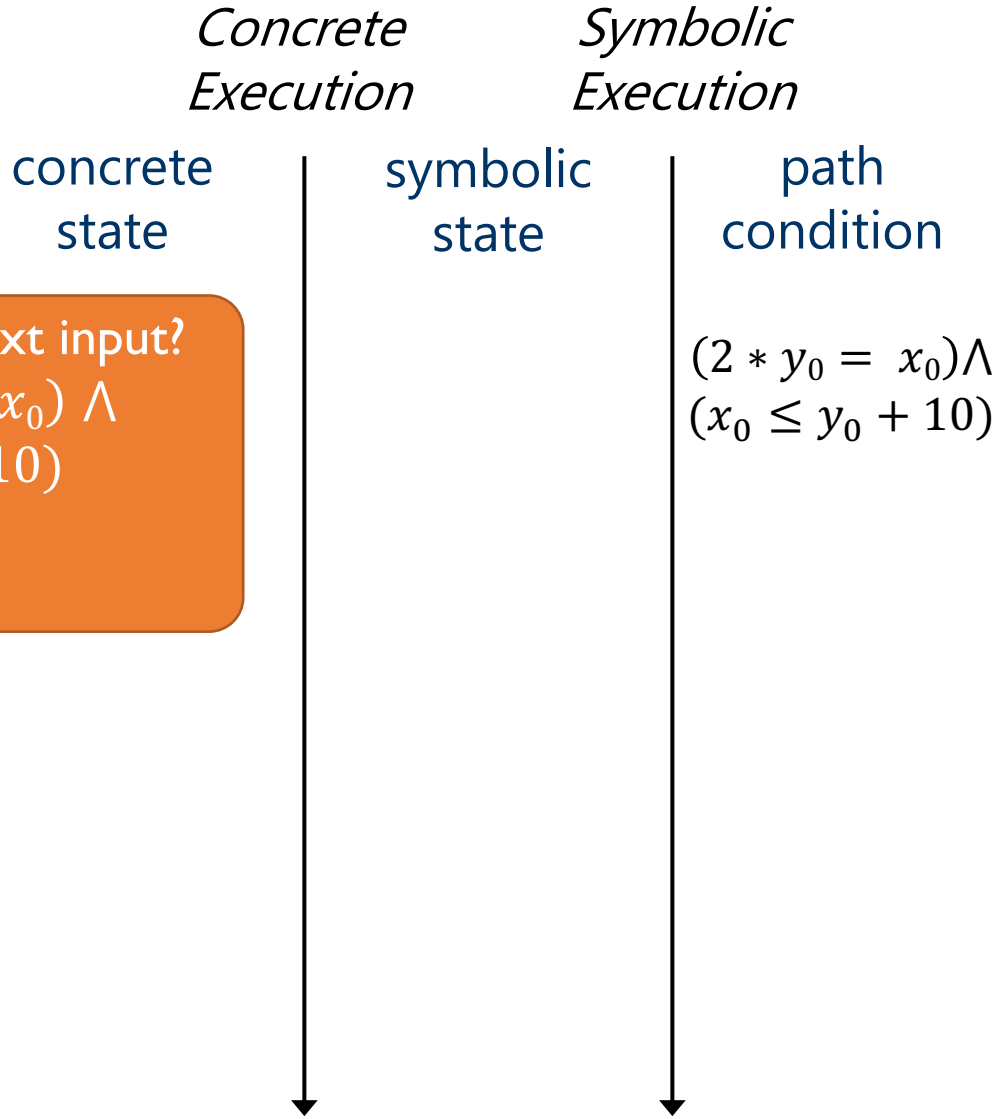
Symbolic Execution

concrete state	symbolic state	path condition
		$(2 * y_0 = x_0) \wedge (x_0 \leq y_0 + 10)$
$x = 2,$ $y = 1,$ $z = 2$	$x = x_0,$ $y = y_0,$ $z = 2 * y_0$	

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) {
              ERROR;
        }
  }
}
```

How to generate next input?
Solve: $(2 * y_0 = x_0) \wedge \sim(x_0 \leq y_0 + 10)$



Concolic Testing

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

void testme (int x,
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
		$(2 * y_0 = x_0) \wedge (x_0 \leq y_0 + 10)$

How to generate next input?
Solve: $(2 * y_0 = x_0) \wedge (x_0 > y_0 + 10)$

Concolic Testing

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

void testme (int
  z = double (y)
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Concrete Execution

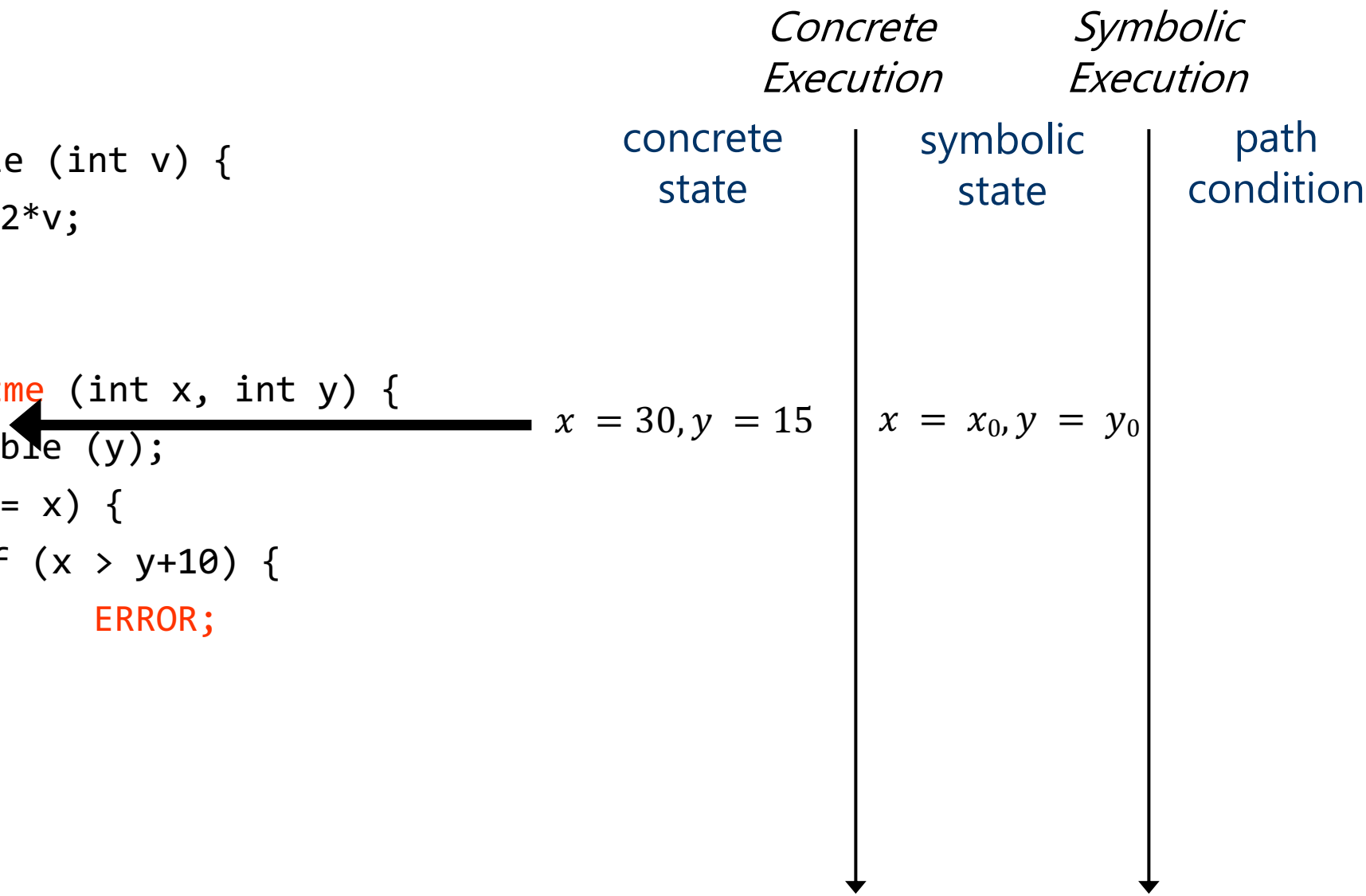
Symbolic Execution

concrete state	symbolic state	path condition
		$(2 * y_0 = x_0) \Lambda$ $(x_0 \leq y_0 + 10)$

How to generate next input?
Solve: $(2 * y_0 = x_0) \Lambda$ $(x_0 > y_0 + 10)$
(possible) solution: $x = 30, y = 15$

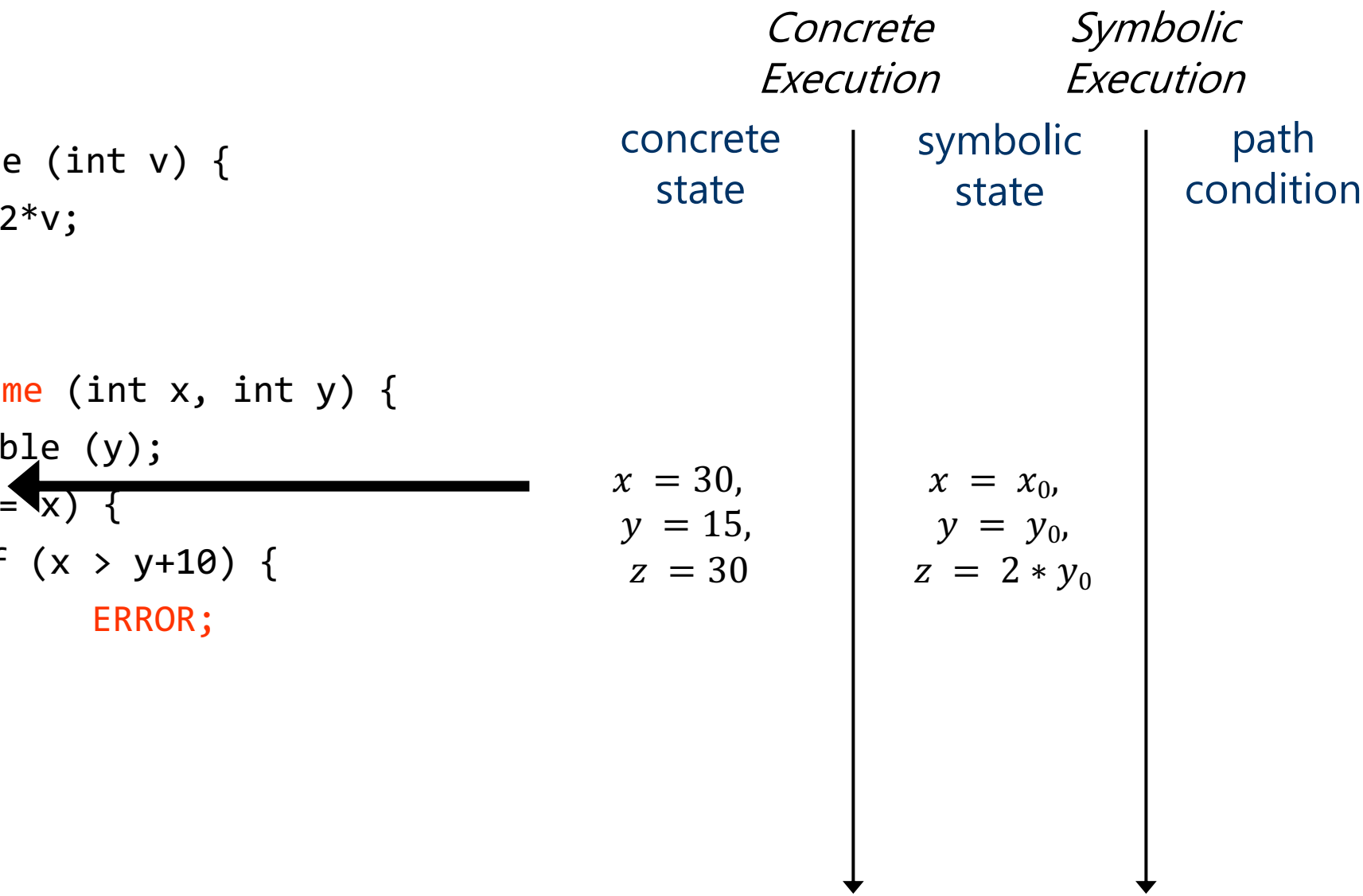
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```



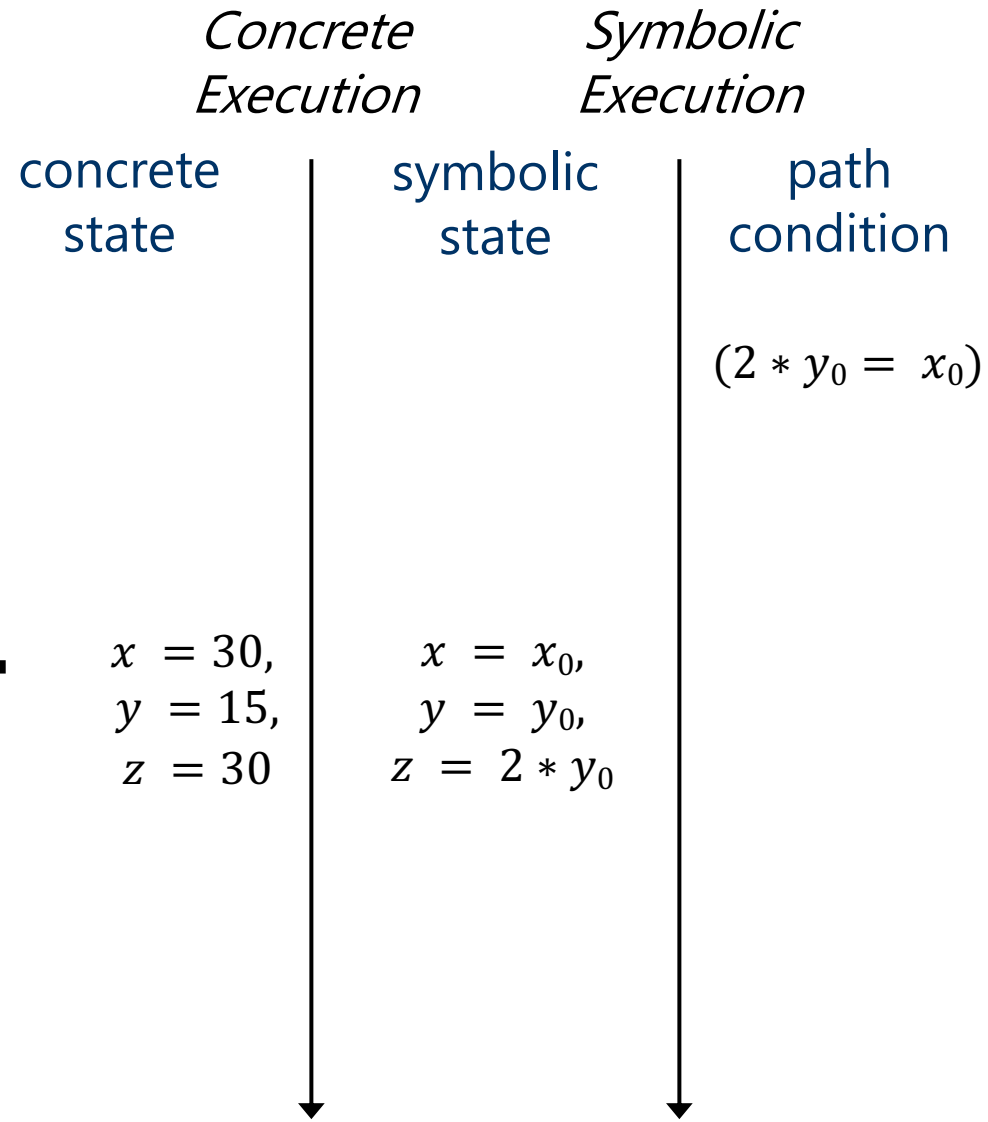
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```



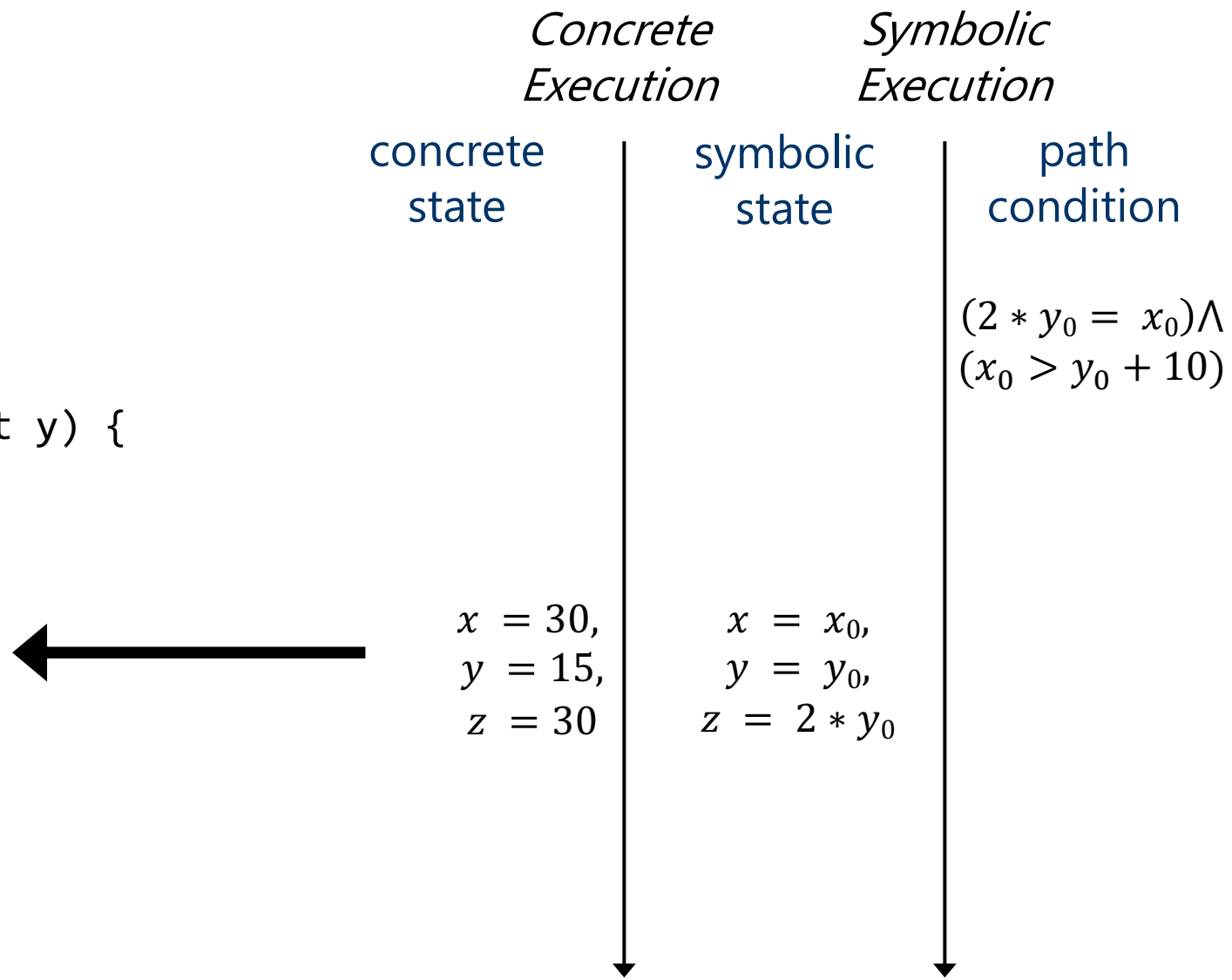
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
       if (x > y+10) {
            ERROR;
       }
  }
}
```



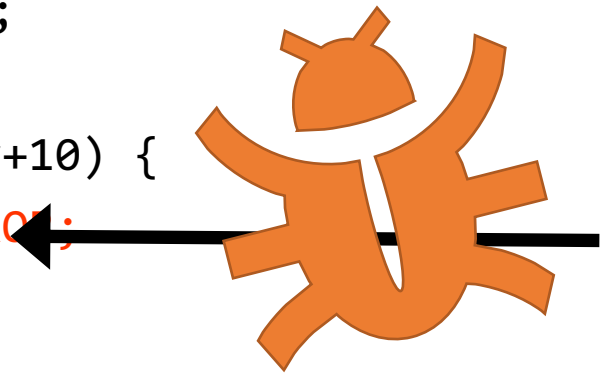
Concolic Testing

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```



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) {
              ERROR;
        }
  }
}
```



Concrete Execution | *Symbolic Execution*

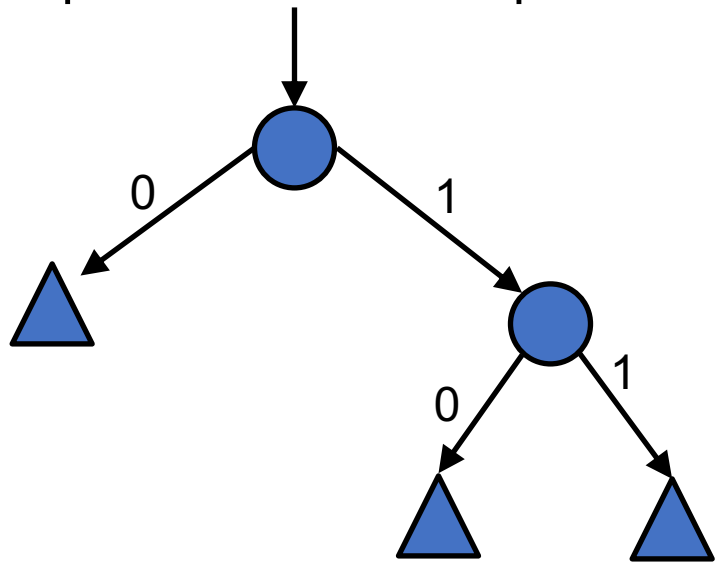
concrete state	symbolic state	path condition
		$(2 * y_0 = x_0) \wedge (x_0 > y_0 + 10)$
$x = 30,$ $y = 15,$ $z = 30$	$x = x_0,$ $y = y_0,$ $z = 2 * y_0$	

Abstract View

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Computation Tree

- Each node is the execution of a branch (triangle == exit)
- Each edge is the execution of a basic block
- Each path in the tree is a "path"

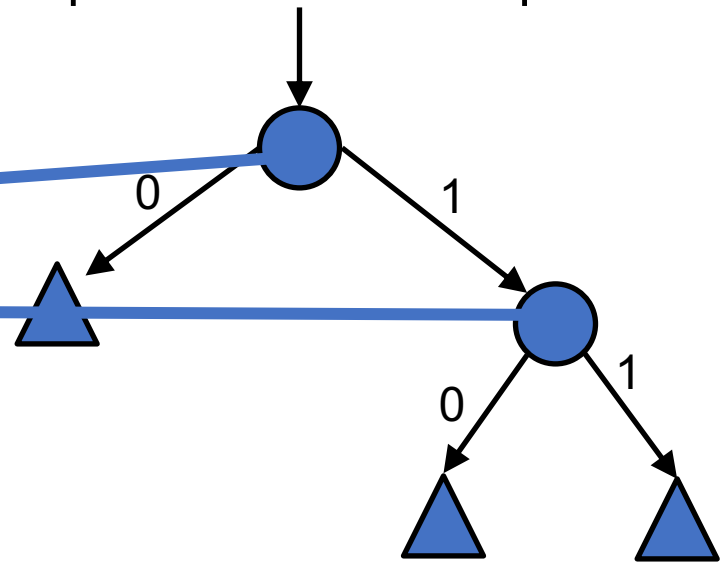


Abstract View

Computation Tree

- Each node is the execution of a branch (triangle == exit)
- Each edge is the execution of a basic block
- Each path in the tree is a "path"

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

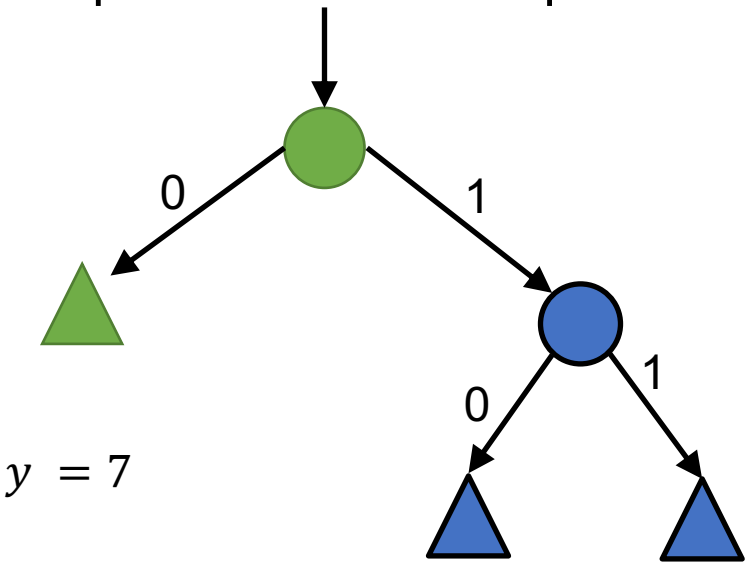


Abstract View

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Computation Tree

- Each node is the execution of a branch (triangle == exit)
- Each edge is the execution of a basic block
- Each path in the tree is a "path"



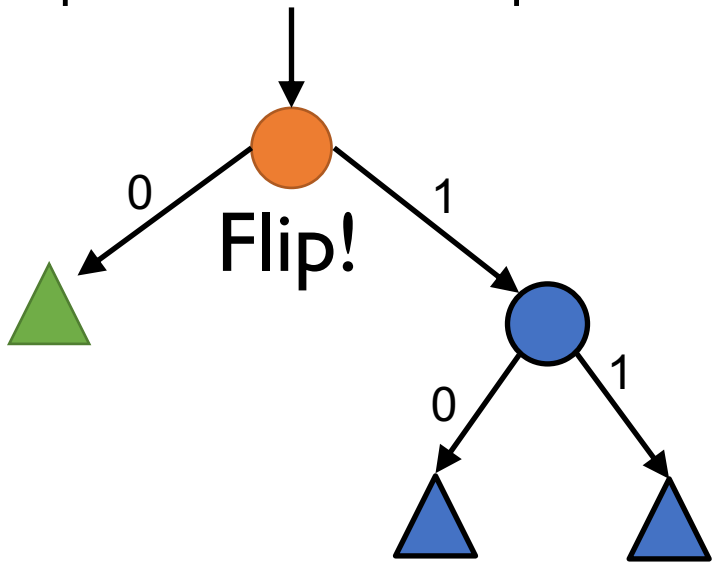
$x = 22, y = 7$

Abstract View

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Computation Tree

- Each node is the execution of a branch (triangle == exit)
- Each edge is the execution of a basic block
- Each path in the tree is a "path"

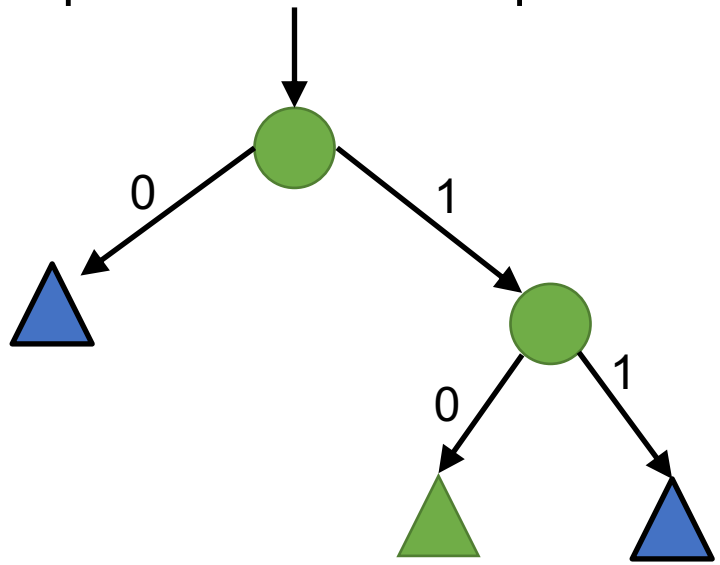


Abstract View

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Computation Tree

- Each node is the execution of a branch (triangle == exit)
- Each edge is the execution of a basic block
- Each path in the tree is a "path"



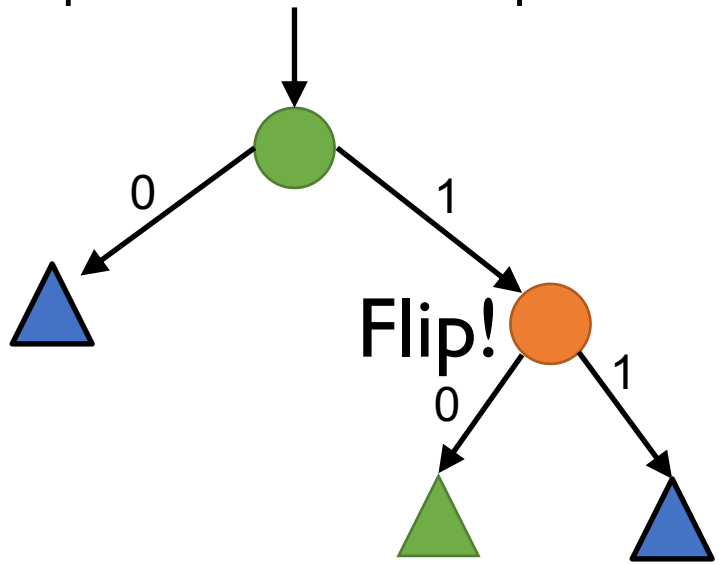
$x = 2, y = 1$

Abstract View

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Computation Tree

- Each node is the execution of a branch (triangle == exit)
- Each edge is the execution of a basic block
- Each path in the tree is a "path"

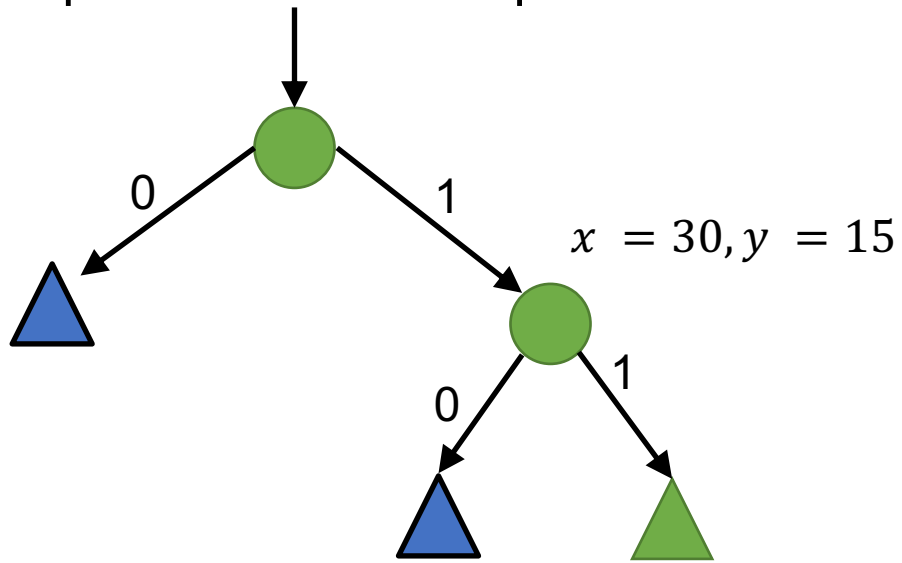


Abstract View

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Computation Tree

- Each node is the execution of a branch (triangle == exit)
- Each edge is the execution of a basic block
- Each path in the tree is a "path"

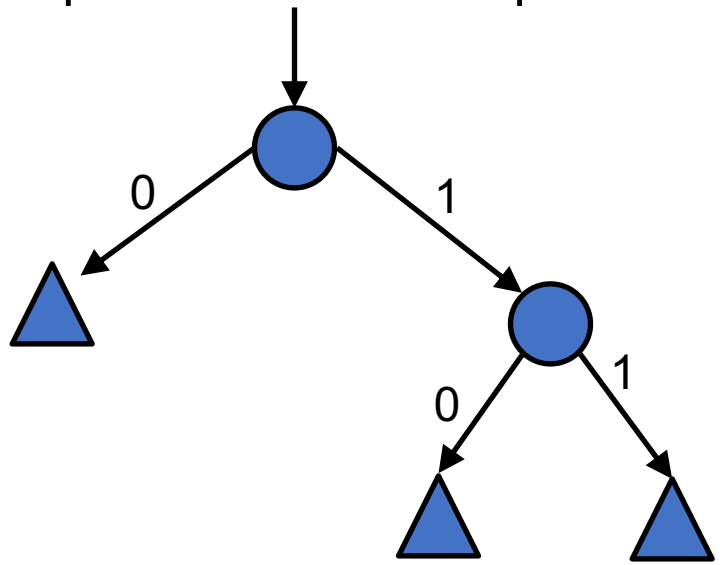


Abstract View

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

Computation Tree

- Each node is the execution of a branch (triangle == exit)
- Each edge is the execution of a basic block
- Each path in the tree is a "path"



Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              ERROR;
        }
  }
}
```

- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there

Symbolic Execution

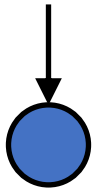
```
int double (int v) {
  return 2*v;
}

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              y = x/0;
        }
  }
}
```

- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there

Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              y = x/0;
        }
  }
}
```

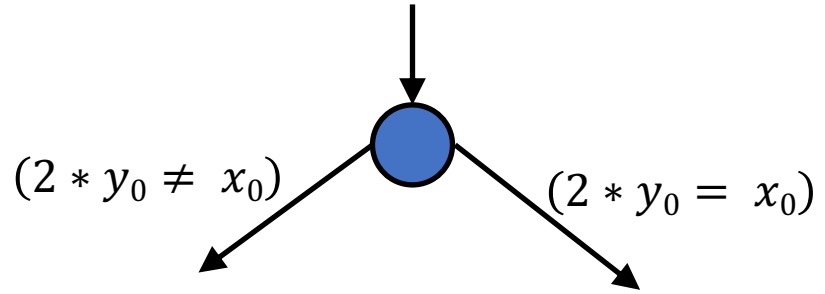
- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there



Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              y = x/0;
        }
  }
}
```

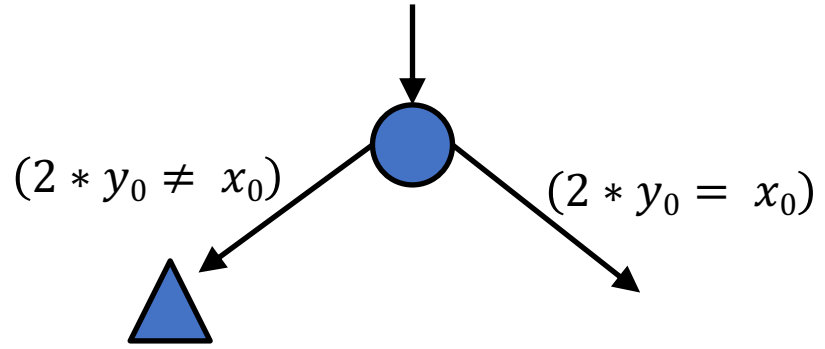
- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there



Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              y = x/0;
        }
  }
}
```

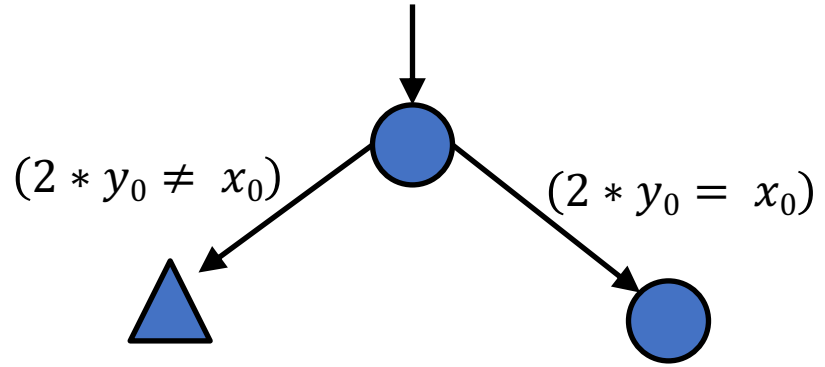
- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there



Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              y = x/0;
        }
  }
}
```

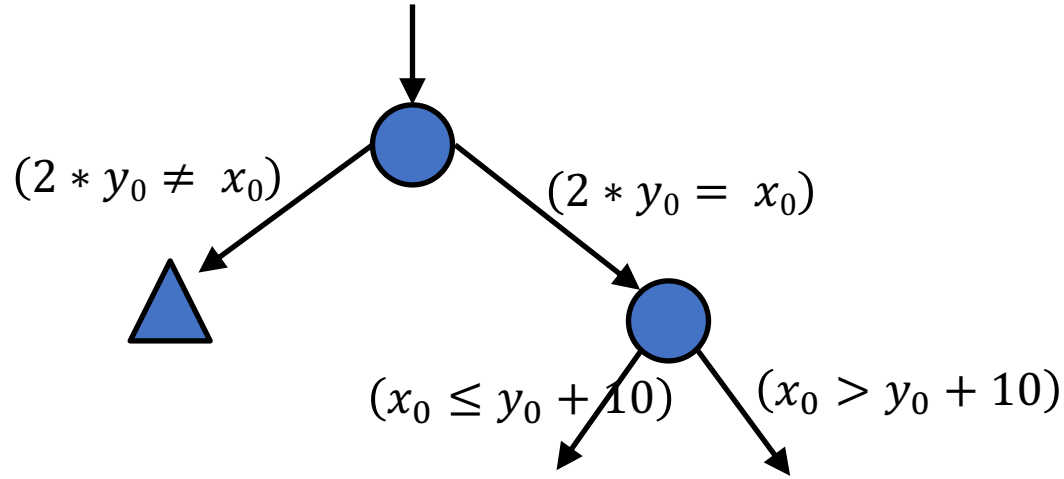
- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there



Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              y = x/0;
        }
  }
}
```

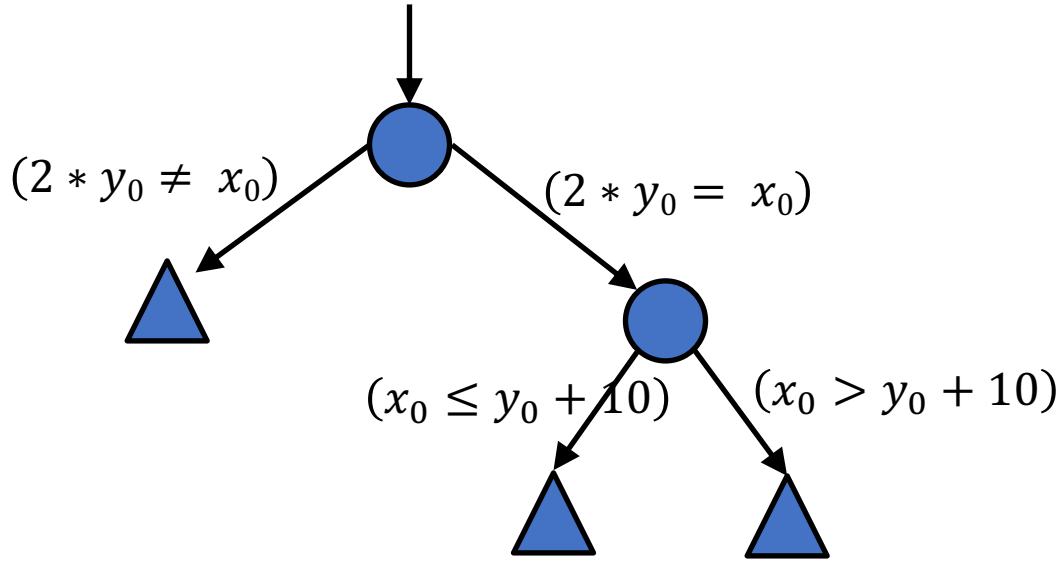
- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there



Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              y = x/0;
        }
  }
}
```

- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there

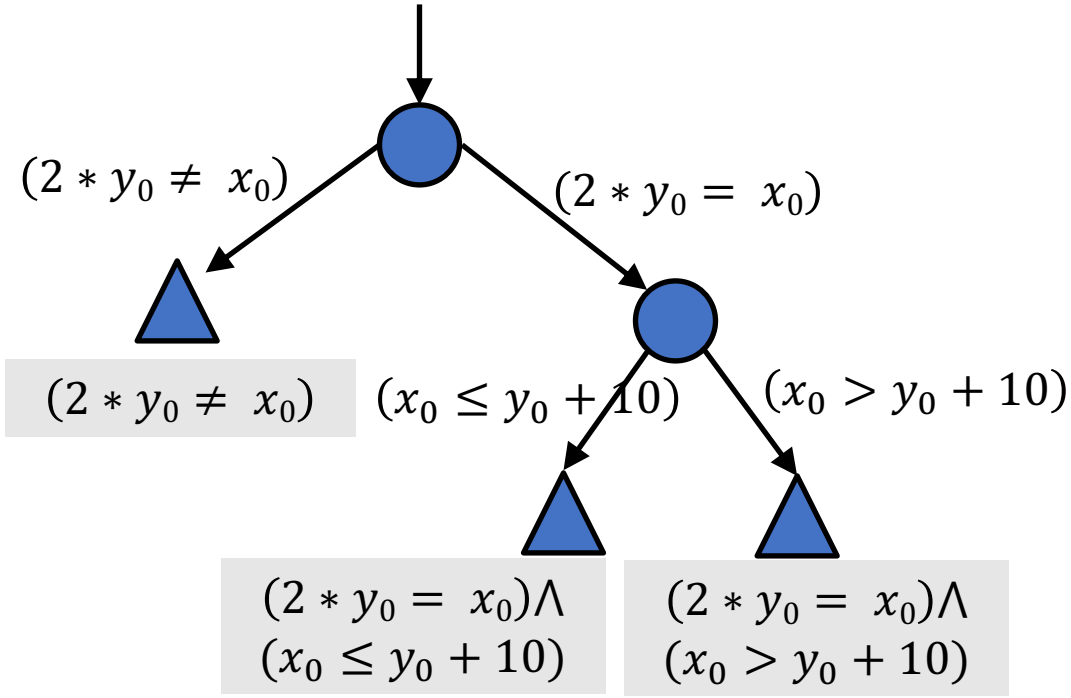


Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
              y = x/0;
        }
  }
}
```

path conditions

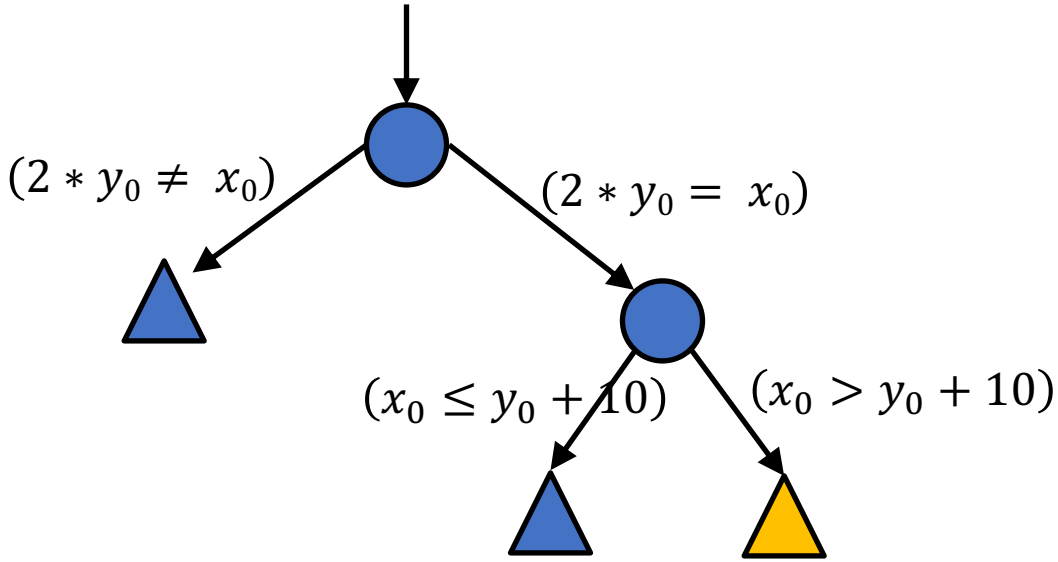
- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there



Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
            y = x/0;
        }
  }
}
```

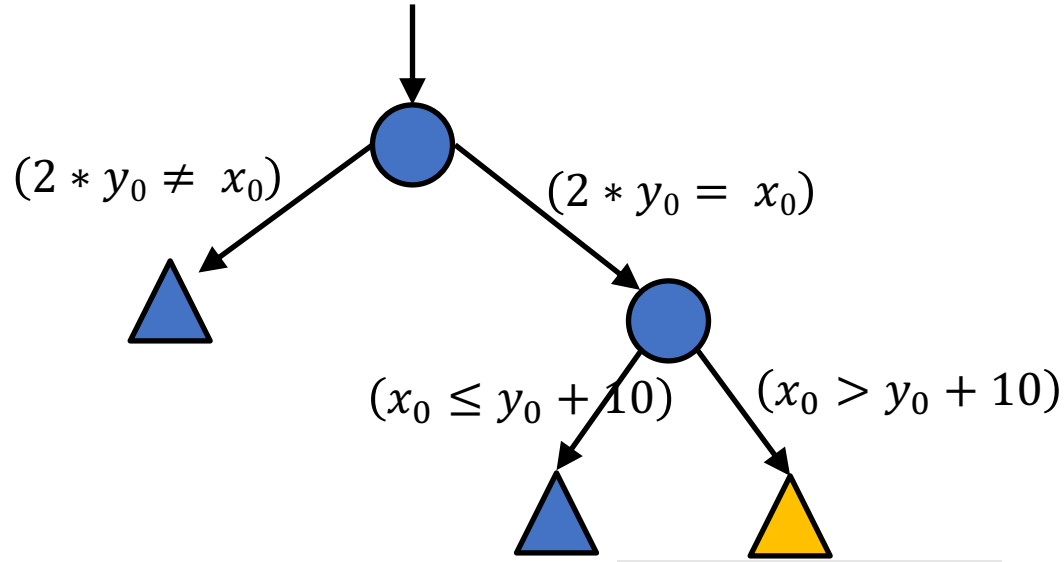
- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there



Symbolic Execution

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

void testme (int x, int y) {
  z = double (y);
  if (z == x) {
        if (x > y+10) {
            y = x/0;
        }
  }
}
```

- Build out computation tree without any concrete inputs
- Instantiate paths if bug is found there



Solve path condition to witness bug

$(2 * y_0 = x_0) \wedge (x_0 > y_0 + 10)$