# Symbolic vs. Concrete Testing

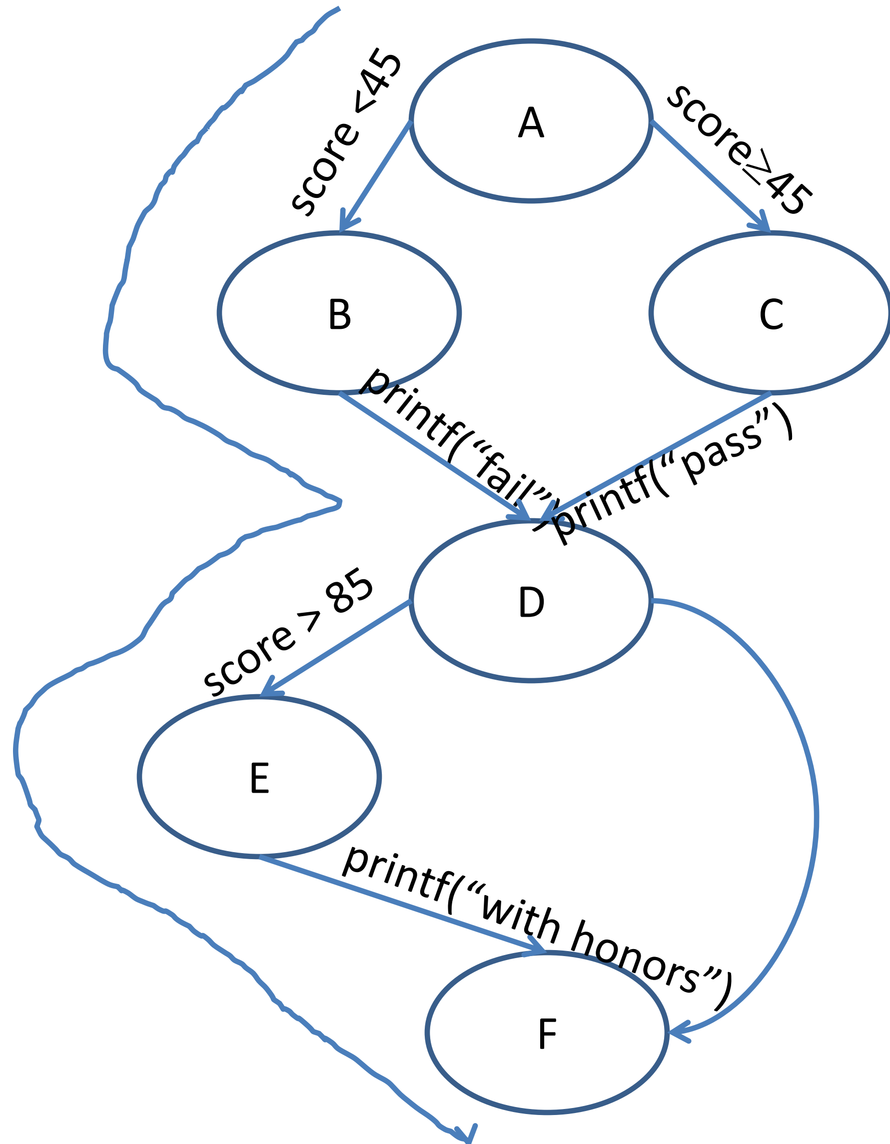
Mooly Sagiv

# Program Path

- Program Path
  - A path in the control flow of the program
    - Can start and end at any point
    - Appropriate for imperative programs
- Feasible program path
  - There exists an input that leads to the execution of this path
- Infeasible program path
  - No input that leads to the execution

# Infeasible Paths

```
void grade(int score) {
A:  if (score <45) {
B:    printf("fail");
    }
    else
 C:    printf("pass");
    }
 D:  if (score > 85) {
 E:     printf("with honors");
        }
F:
}
```



# Concrete vs. Symbolic Executions

- Real programs have many infeasible paths
  - Ineffective concrete testing
- Symbolic execution aims to find rare errors

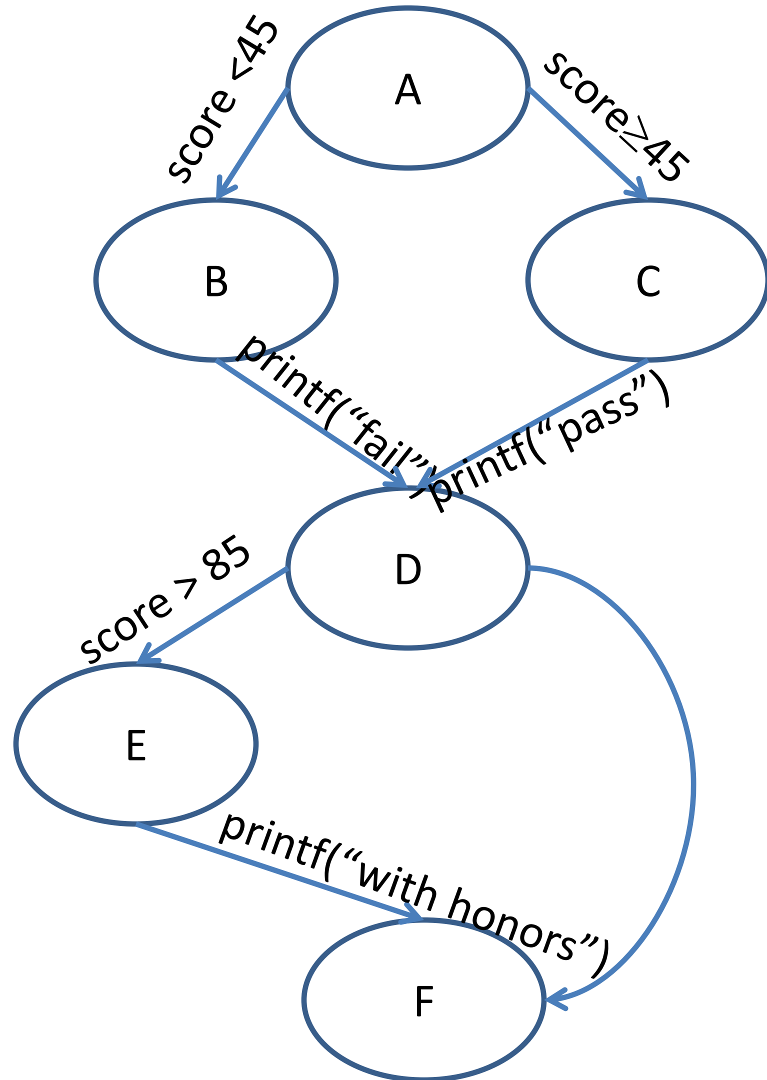
# Symbolic Testing Tools

- EFFIGY [King, IBM 76]
- PEX [MSR]
- SAGE [MSR]
- SATURN[Stanford]
- KLEE[Stanford]
- Java pathfinder[NASA]
- Bitscope [Berkeley]
- Cute [UIUC, Berkeley]
- Calysto [UBC]

# Finding Infeasible Paths Via SMT

```
void grade(int score) {
A:  if (score <45) {
B:    printf("fail");
    }
   else
 C:    printf("pass");
    }
 D:  if (score > 85) {
 E:     printf("with honors");
        }
F:
}
```

$score < 45 \wedge score > 85$ UNSAT



# Plan

- Random Testing
- Symbolic Testing
- Concolic Testing

# Fuzzing [Miller 1990]

- Test programs on random unexpected data
- Can be realized using black/white testing
- Can be quite effective
  - Operating Systems
  - Networks
- …
- Usually implemented via instrumentation
- Tricky to scale for programs with many paths

```
If (x == 10001) {

    ….
    if (f(*y) == *z) {
   ….
```

```
int f(int *p) {

  if (p !=NULL) {
    return q ;

}
```

# Symbolic Exploration

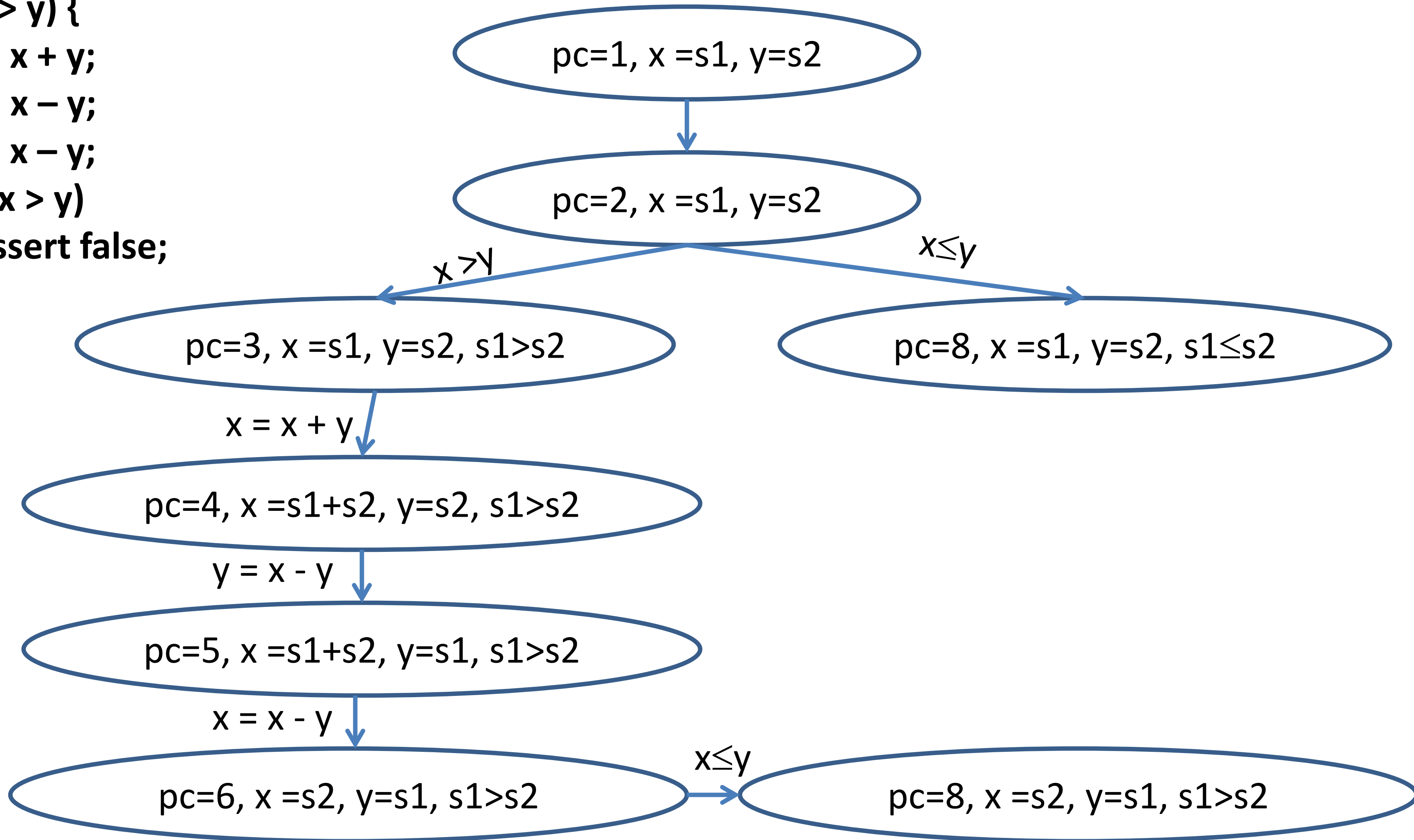
- Execute a program on symbolic inputs
- Track set of values symbolically
- Update symbolic states when instructions are executed
- Whenever a branch is encountered check if the path is feasible using a theorem prover call

# Symbolic Execution Tree

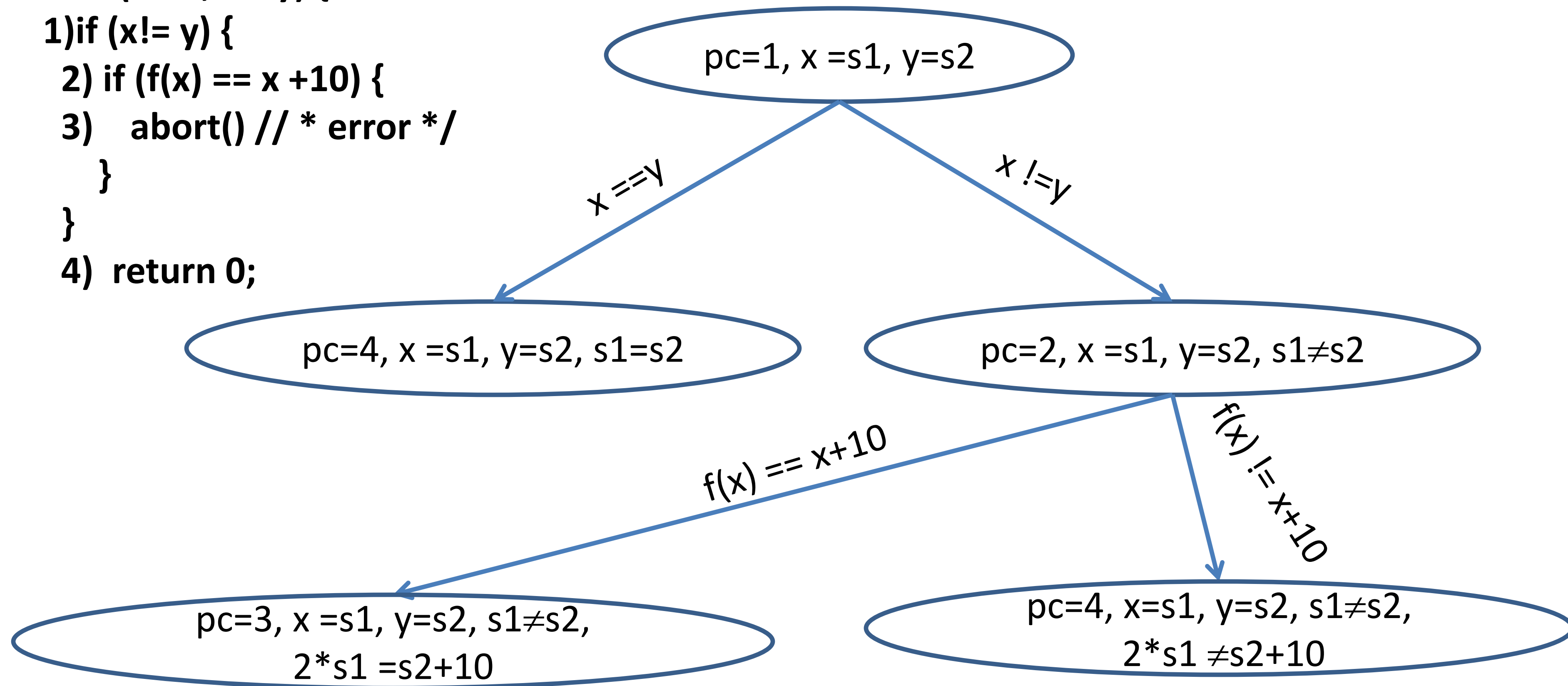
- The constructed symbolic execution paths
- Nodes
  - Symbolic Program States
- Edges
  - Potential Transitions
- Constructed during symbolic evaluation
- Each edge requires a theorem prover call

# Simple Example

```
1)int x, y;
2)if (x > y) {
  3) x = x + y;
  4) y = x – y;
  5) x = x – y;
  6) if (x > y)
    7) assert false;
8)}
```



# Another Example



# Pointers

```
struct foo {int i; char c;}
bar(struct foo *a) {
    1) if (a->c == 0)  {
     2) *((char *)a + sizeof(int)) = 1 ;
     3)   if  (a->c !=0) {
           4) abort();
         }
      }
    5)
}
```

# Non-Deterministic Behavior

```
int x; y;
1) if (nondet()) {
      2) x = 7;
      }
 else  {
    3) x = 19 ;
   }
4)
```

# Loops

```
1) int i;
2) while  i < n {
      i = i + 1;
     }
3) if (n ==10^6) {
4)     abort();
5) }
```

# Scaling Issues for Symbolic Exploration

# Challenge 1: Limitations of Theorem Provers

```
foobar(int  x, int y) {
1)    if (x * x * x > 0) {
2)        if  (x>0 && y ==10) {
3)            abort() ; }
4)    }
5)        else {
6)             if (x > 0 && y == 20)  {
7)                 abort  ;}
8)             }
9)    }
```

# Challenge 2: External Calls

```
1)  FILE *fp;
2)  fp = fopen(“test.txt", “w");
3)  if (fp) {
4)      struct stat buffer;
5)      if (stat (“text.txt”, &buffer) != 0) {
6)          abort();
7)      }
8)  }
```

# Challenge 3: #Theorem prover calls

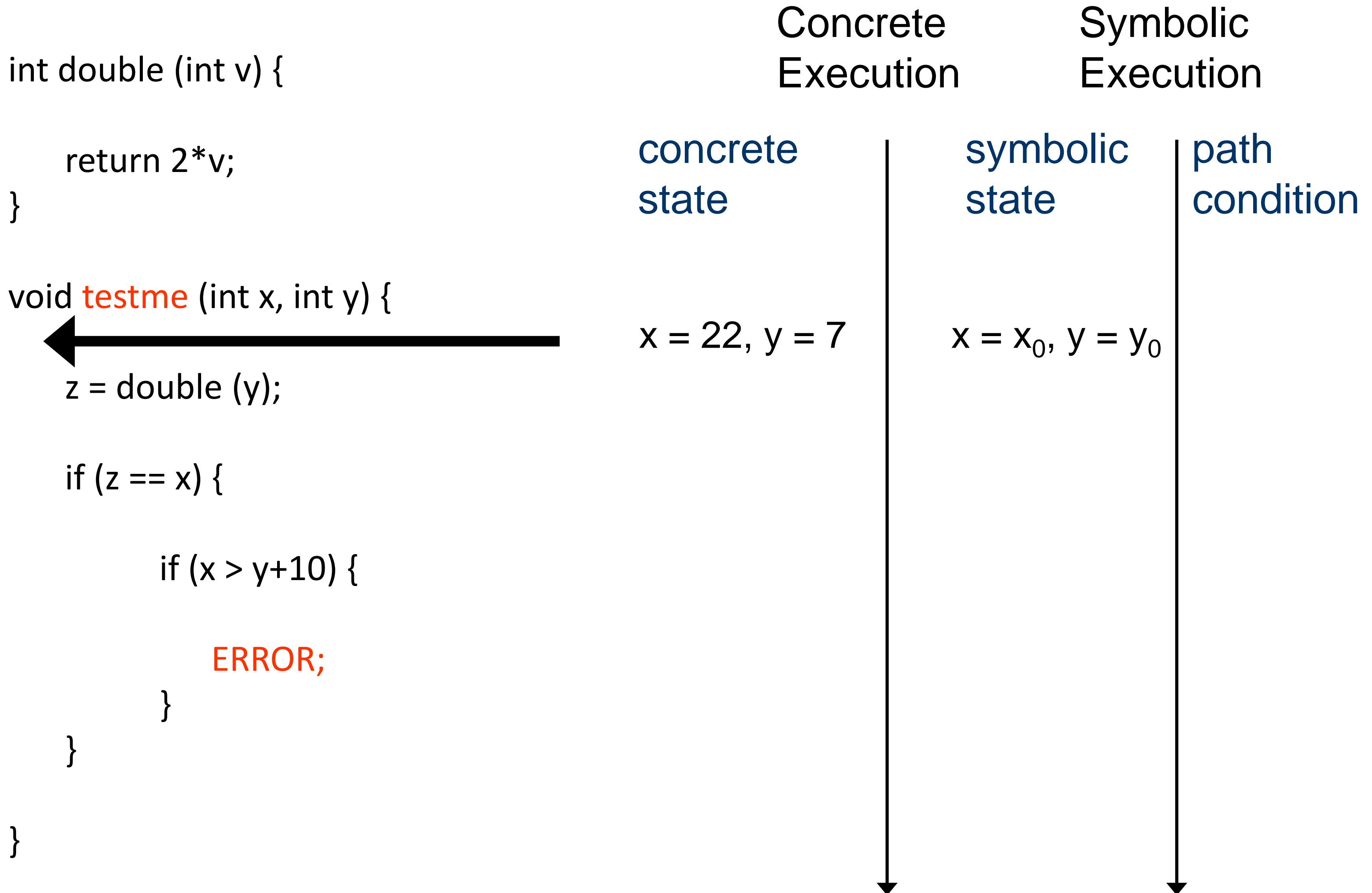
```
1) int i = 0;
2) while  i < n {
     i = i + 1;
    }
3) if (n==10^6) {
4)     abort();
5) }
```

# Concolic Testing

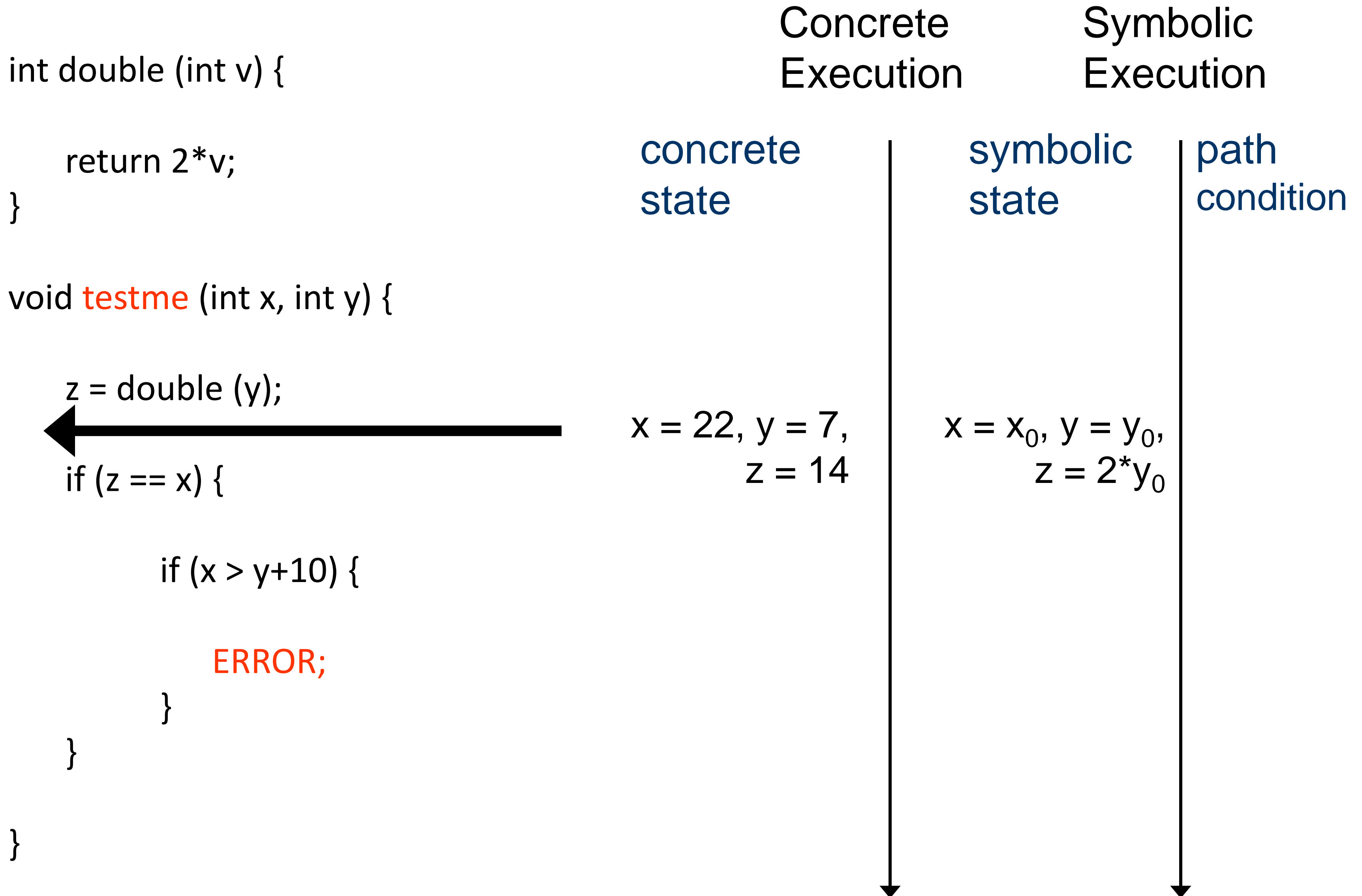
## **Conc**rete + Symb**olic** = Concolic

- Combine concrete testing (concrete execution) and symbolic testing (symbolic execution)
- Trade coverage (miss bugs) for scalability
- Reduce the number of theorem prover calls
- Reduce the complexity of path formulas
- Can cope with external calls

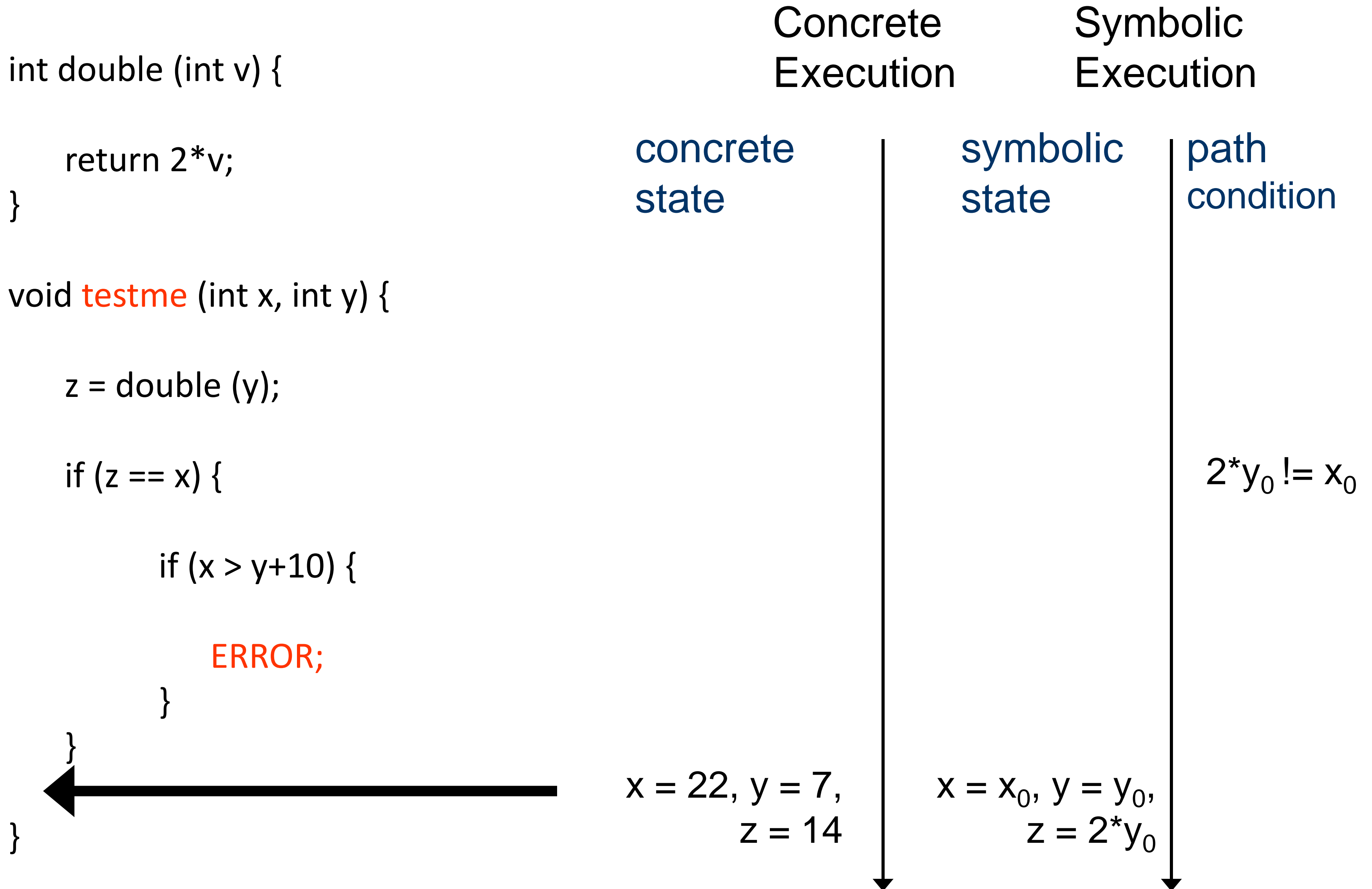
# Concolic Testing Approach



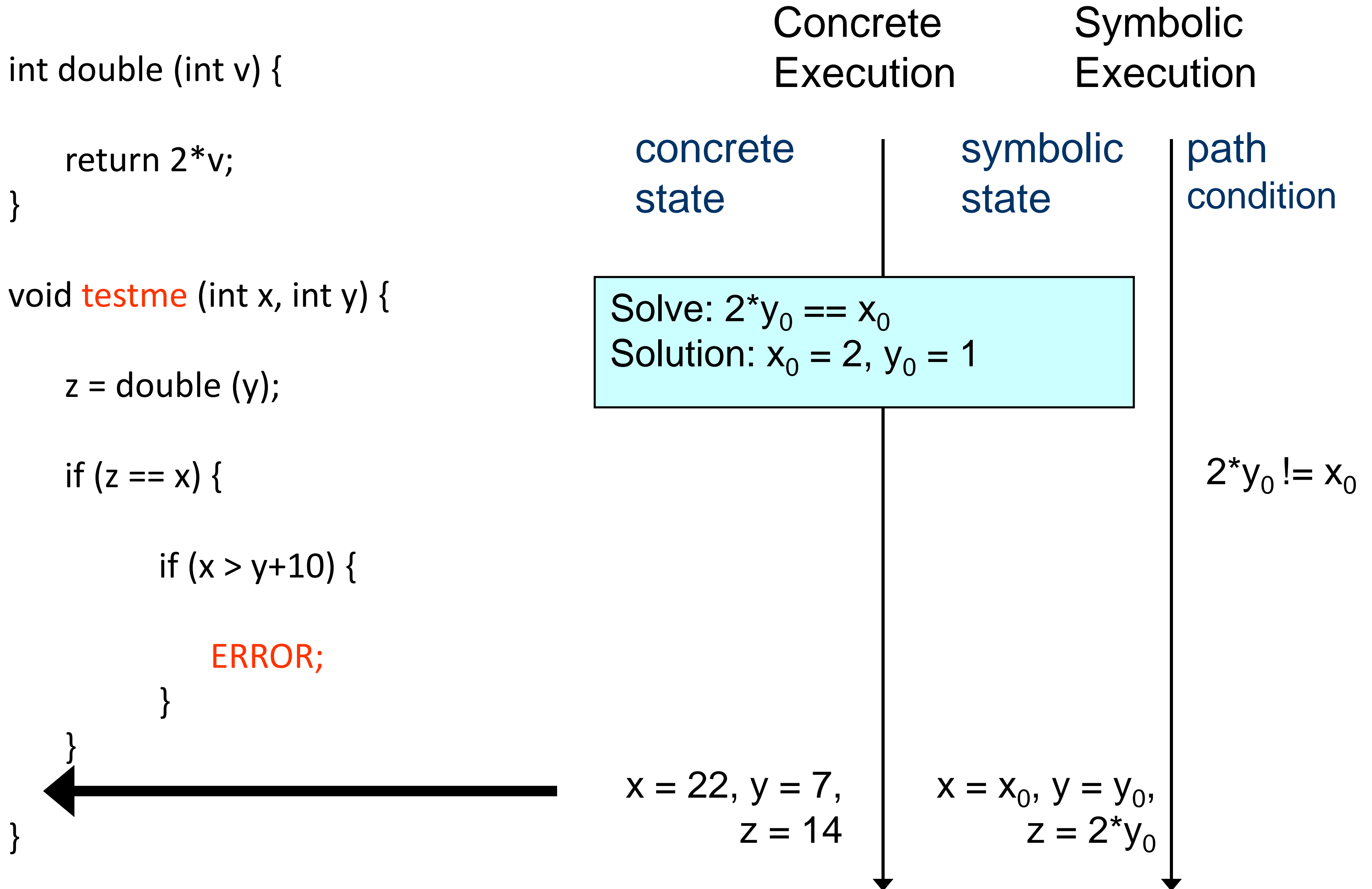
# Concolic Testing Approach



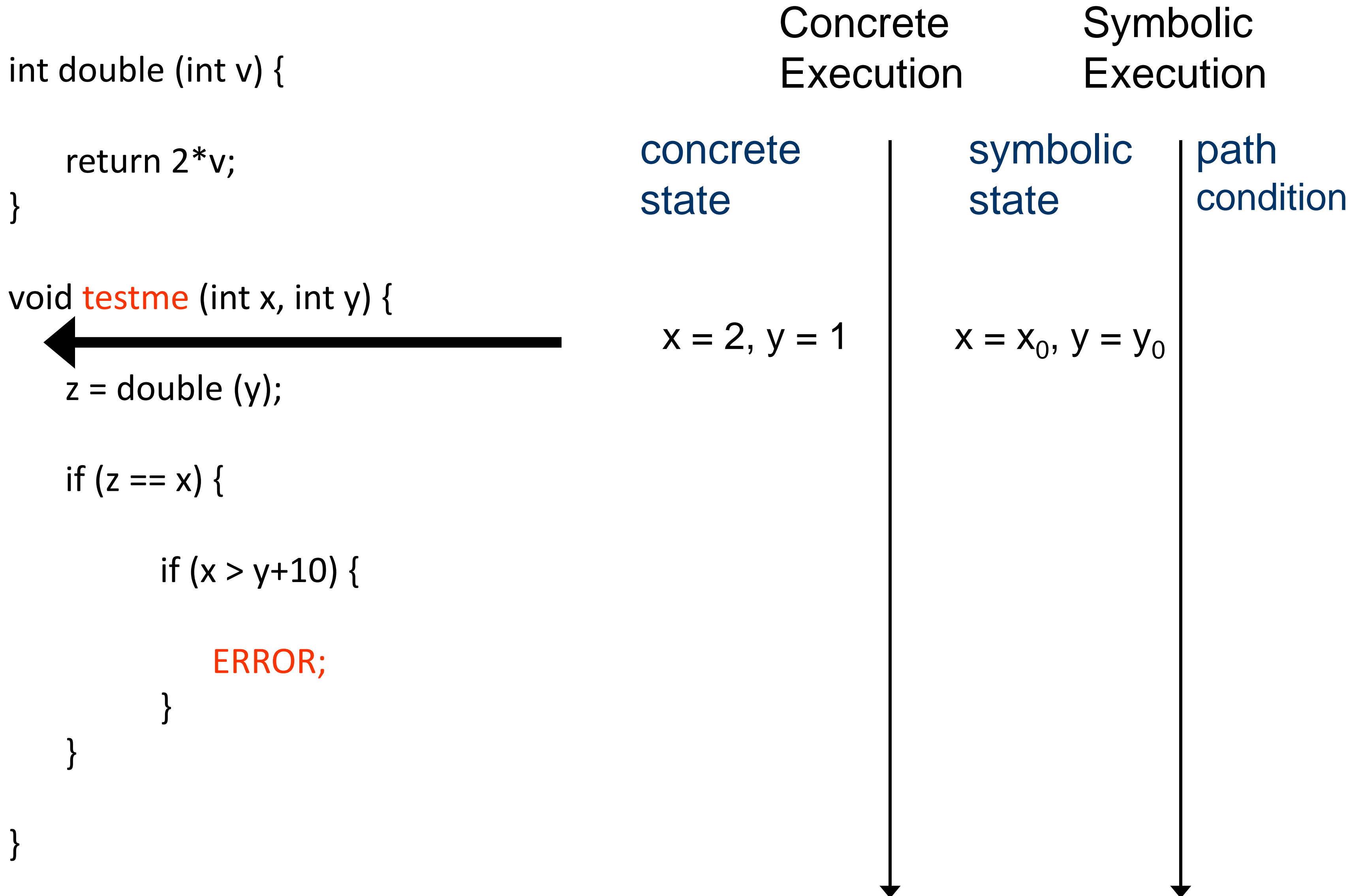
# Concolic Testing Approach



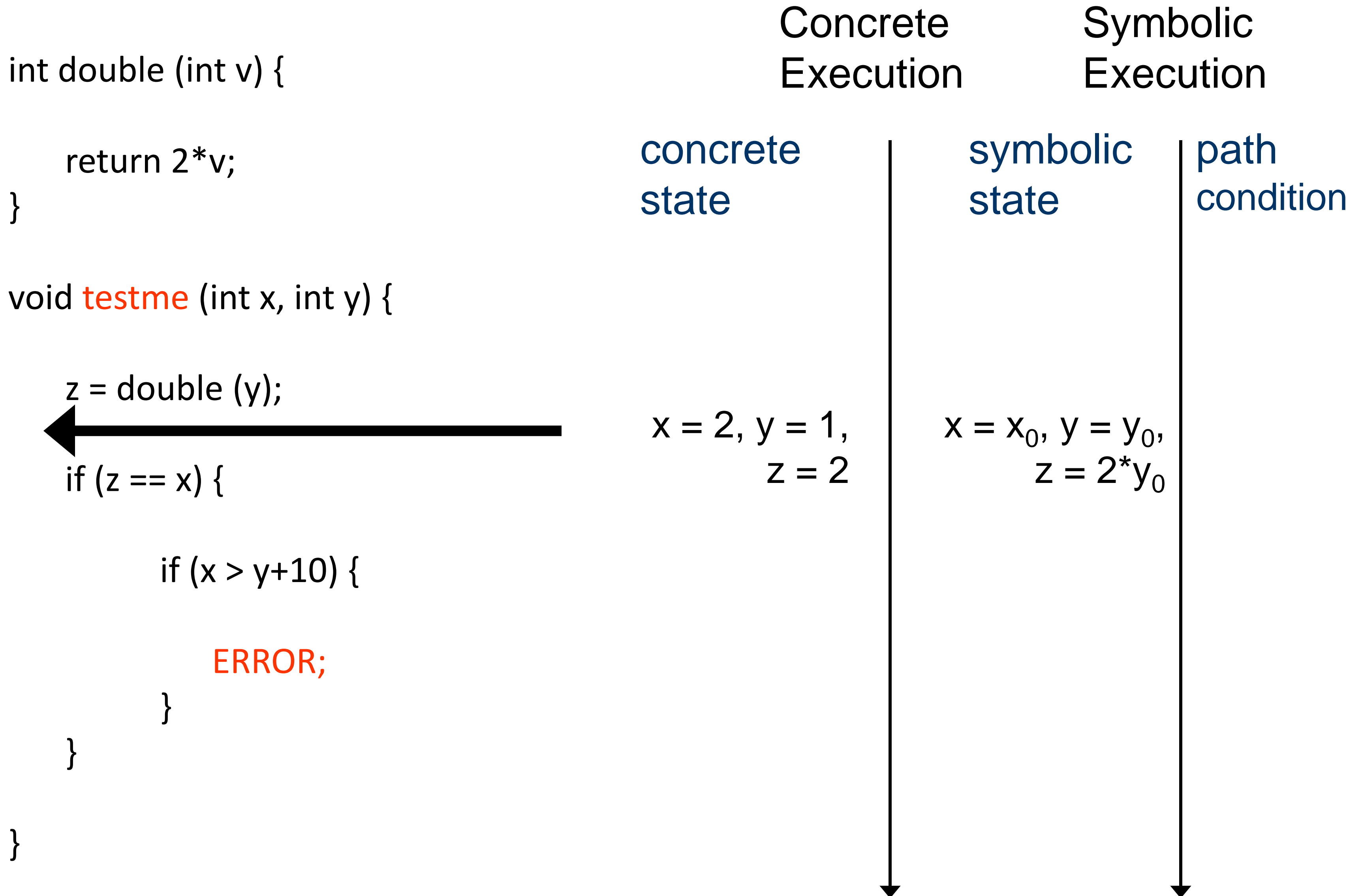
# Concolic Testing Approach



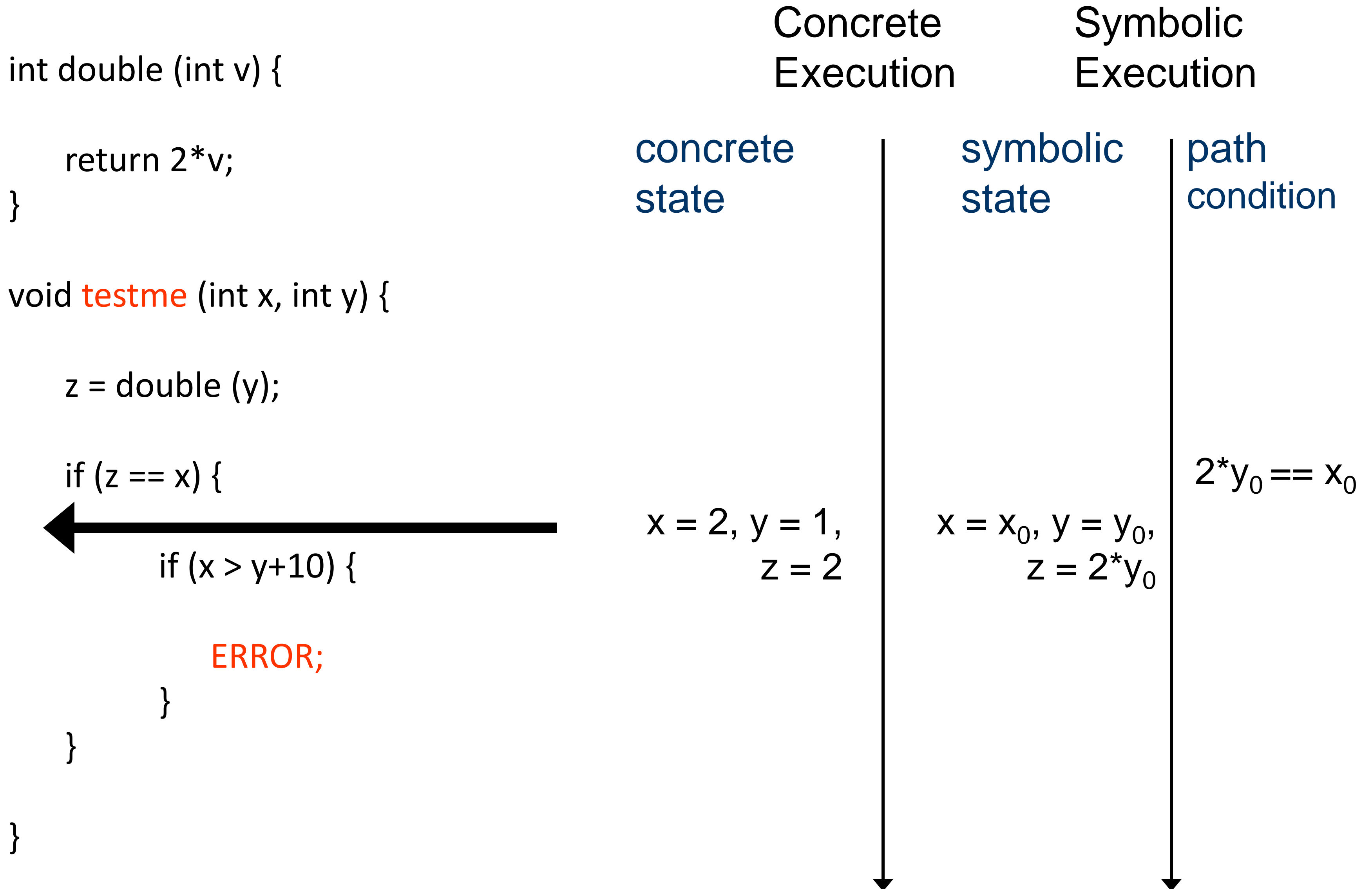
# Concolic Testing Approach



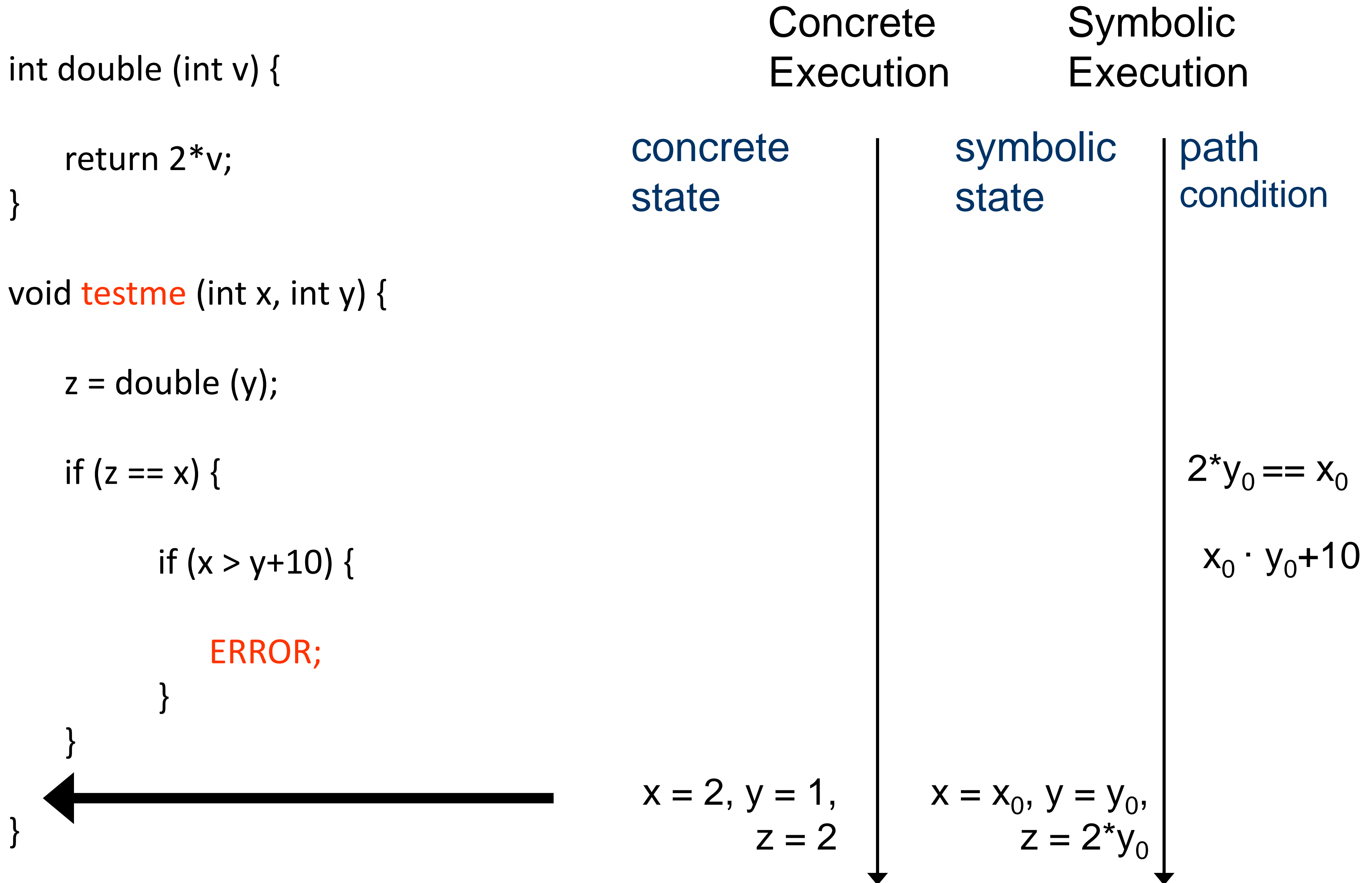
# Concolic Testing Approach



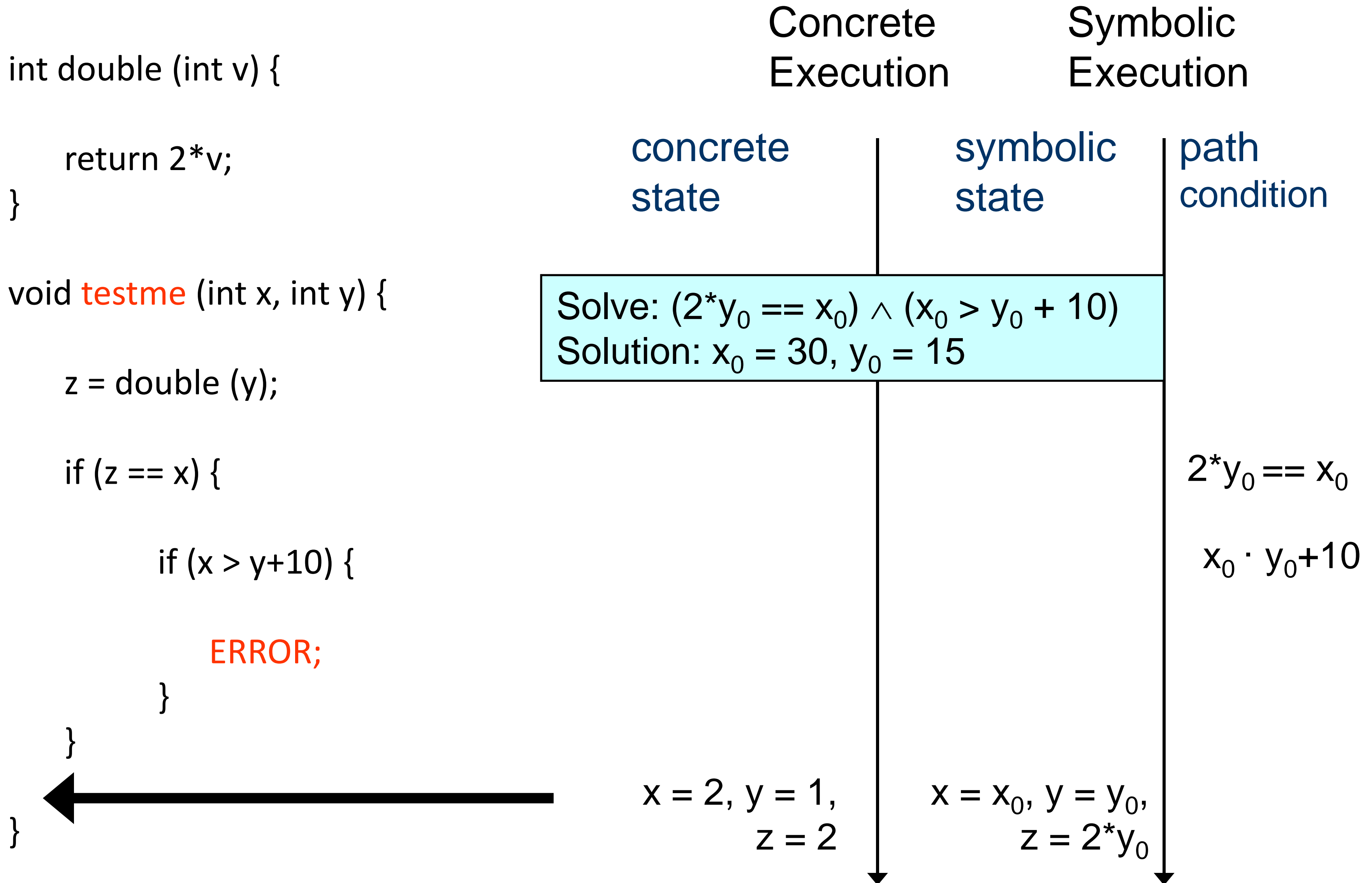
# Concolic Testing Approach



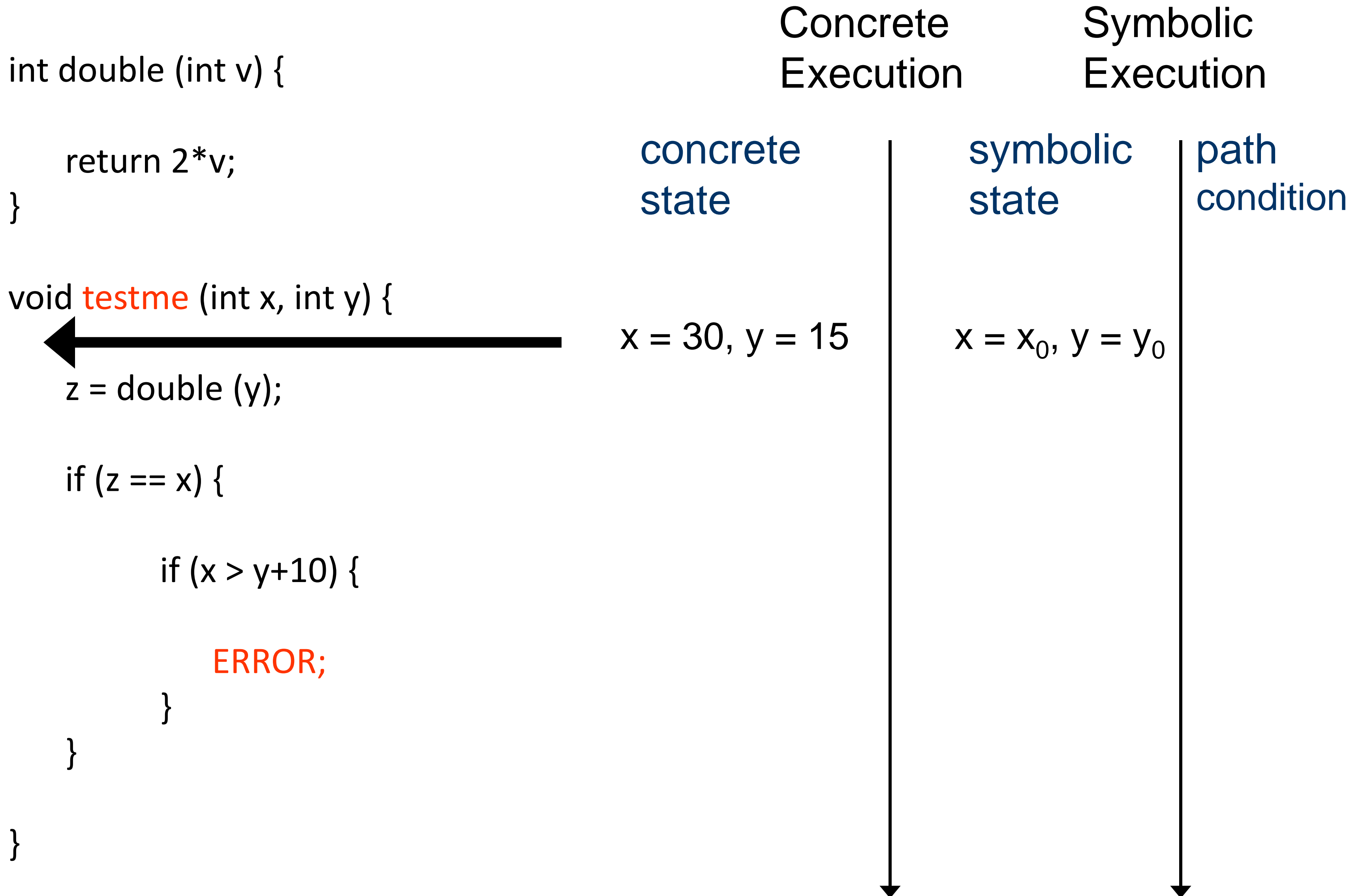
# Concolic Testing Approach



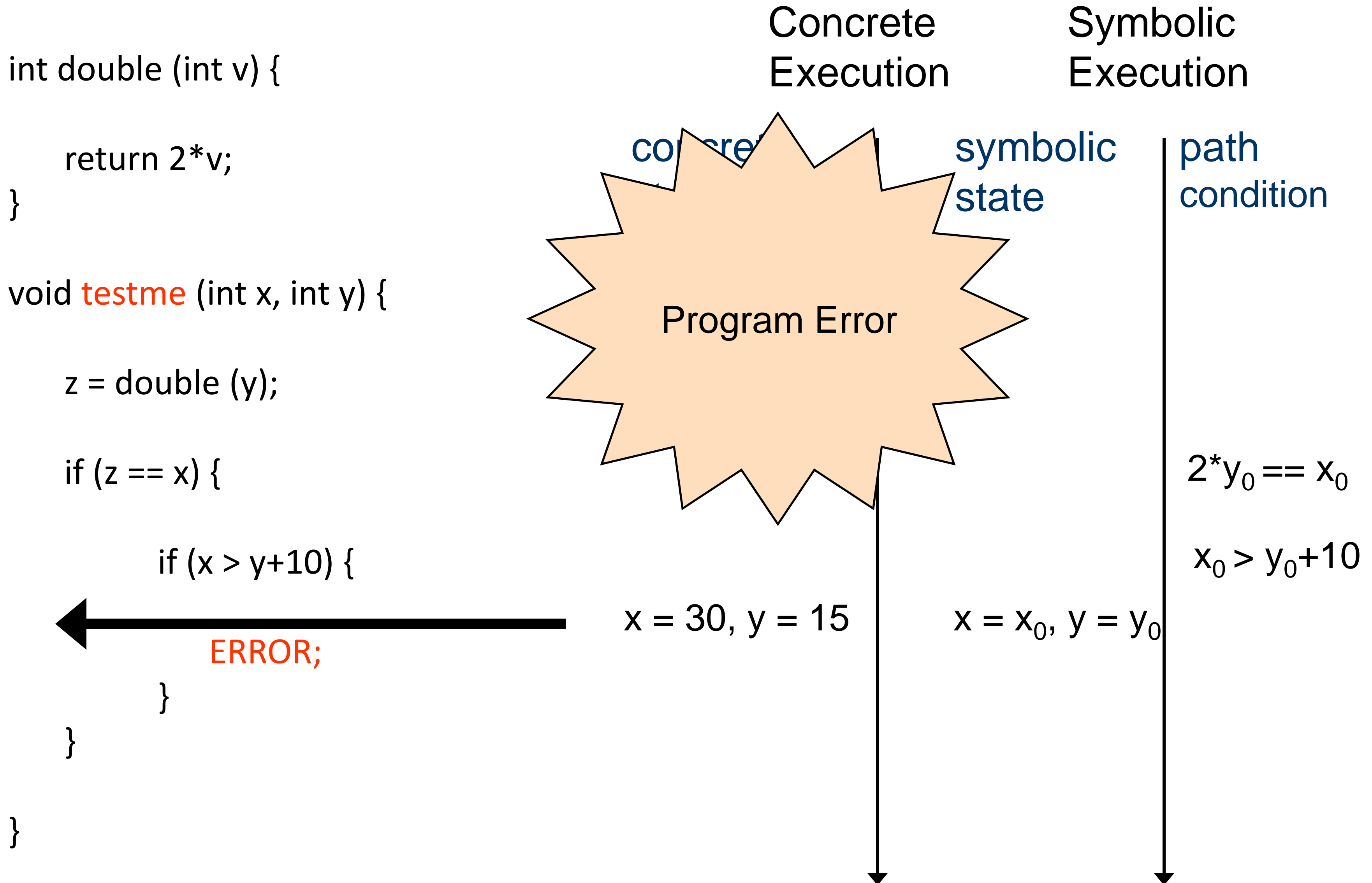
# Concolic Testing Approach



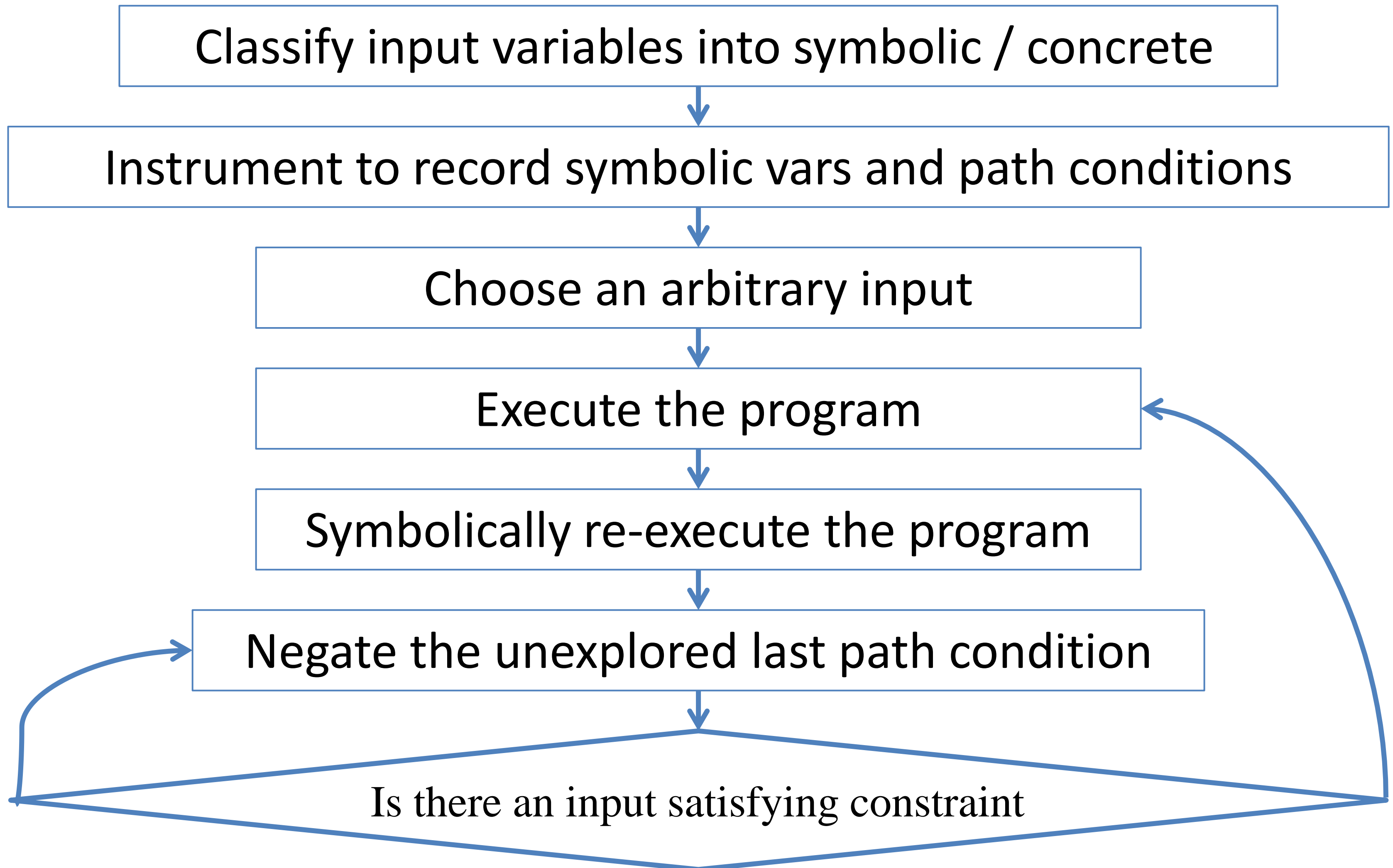
# Concolic Testing Approach



# Concolic Testing Approach



# The Concolic Testing Algorithm



# Interesting Example Concolic Testing

```
foobar(int  x, int y) {
1)    if (x * x * x > 0) {
2)        if  (x>0 && y ==10) {
3)            abort() ; }
4)    }
5)        else {
6)             if (x > 0 && y == 20)  {
7)                 abort  ;}
8)             }
9)    }
```

# Pointers

```
struct foo {int i; char c;}
bar(struct foo *a) {
    1) if (a->c == 0)  {
      2) *((char *)a + sizeof(int)) = 1 ;
      3)   if  (a->c !=0) {
              4) abort();
          }
       }
     5)
}
```

# Loops

```
1) int i= 0;
2) while  i < n {
      i = i + 1;
     }
3) if (n ==10^6) {
4)     abort();
5) }
```

# Handling External Calls

```
1)  FILE *fp;
2)  fp = fopen(“test.txt", “w");
3)  if (fp) {
4)      struct stat buffer;
5)      if (stat (“text.txt”, &buffer) != 0) {
6)          abort();
7)      }
8)  }
```

# Concolic Testing Techniques

- Linear underapproximation
- Modeling System call models
- Simple heuristics
  - Two possible values for pointers
  - …
- Efficient instrumentation
- Interface extraction
- Generating random inputs of arbitrary types
- …

# Original DART Approach [PLDI'05]

- Tailored for C
- Three types of functions
  - Program functions
  - External functions handled non-deterministically
  - Library functions handled as black-box (concrete only)

# SAGE: Whitebox Fuzzing for Security Testing

- Check correctness of Win’7, Win’8
- 200+ machine years
- 1 Billion+ SMT constraints
- 100s of apps, 100s of bugs
- 1/3 of all Win7 WEX security bugs found
- Millions of dollars saved

# Summary

- Concolic testing is powerful
- Scaling is an issue
- Future progress in symbolic reasoning can help
  - Example: handling dynamically allocated memory