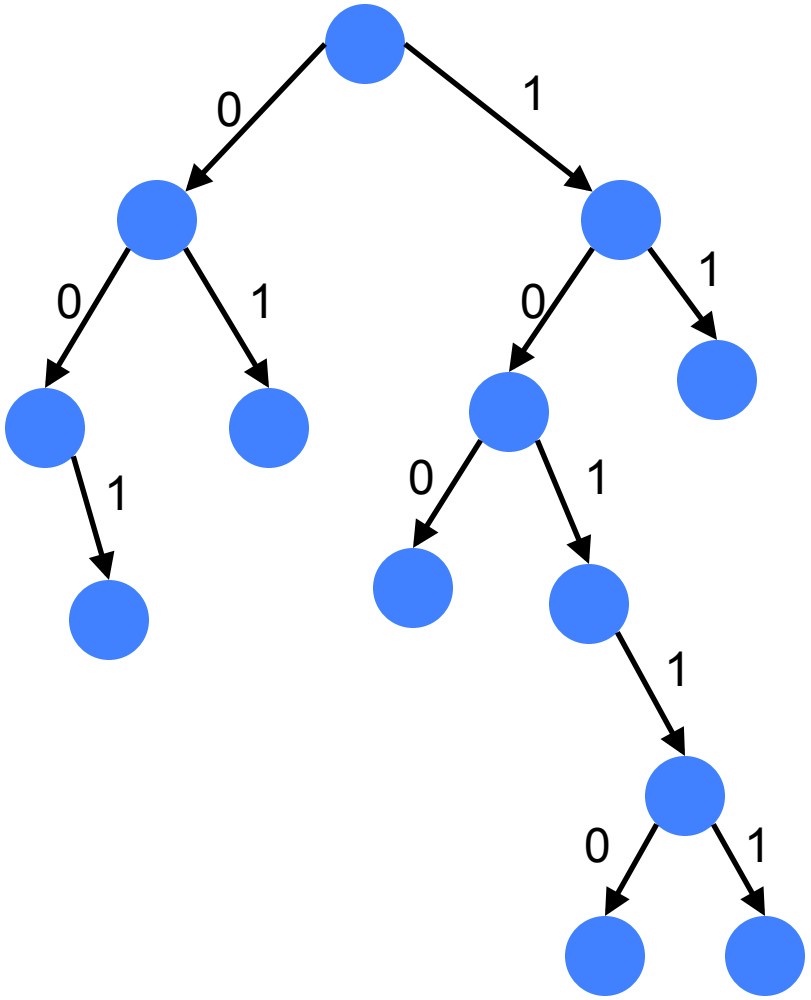
CPEN 422
Software Testing and Analysis

Lecture 13

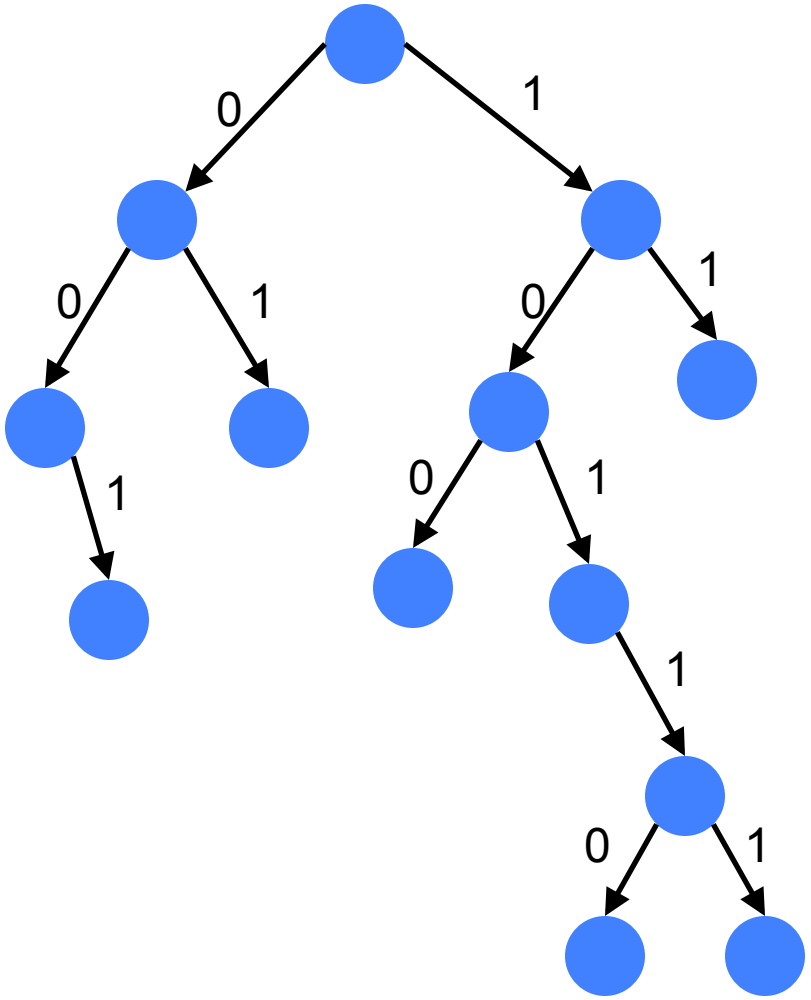
Dynamic Symbolic Execution (concolic testing)

What is Symbolic Execution? What is it good for?



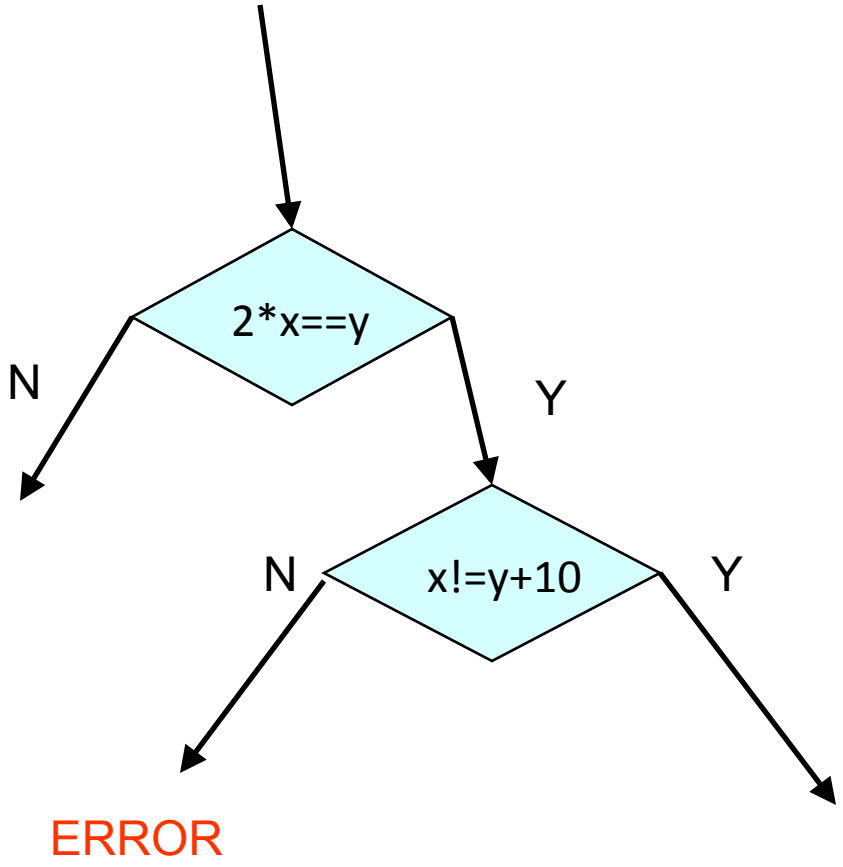
Symbolic Execution: Motivation

- Execution paths of a program can be seen as a binary tree with possibly infinite depth
  - Computation tree
- Each node represents the execution of a "if then else" statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs



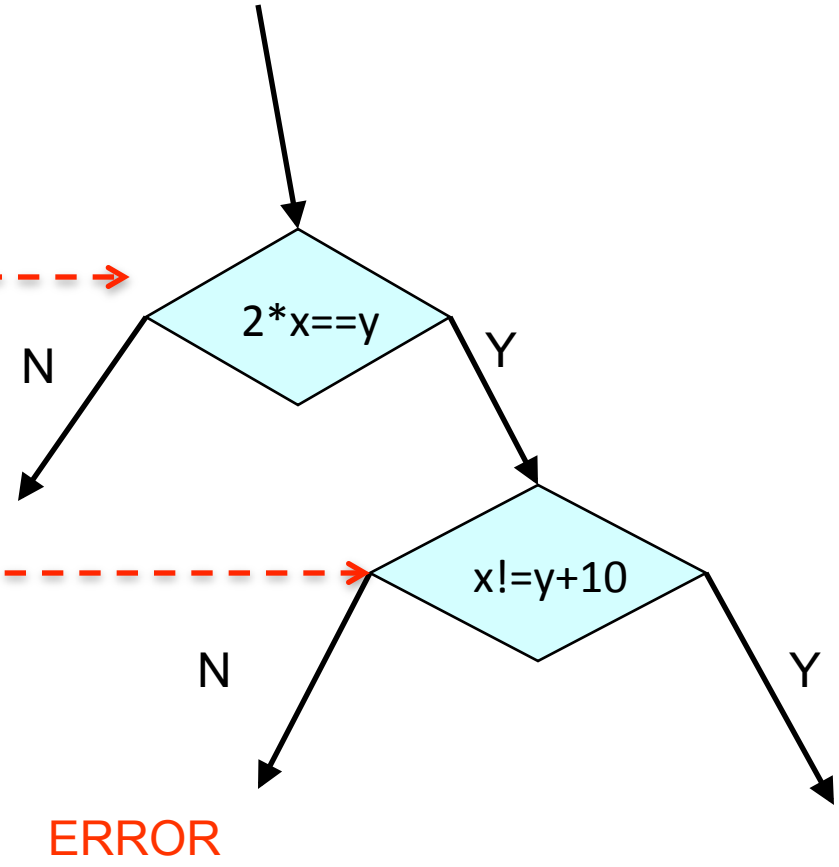
Example of Computation Tree

```
void test_me(int x, int y) {
  if(2*x==y) {
      if(x != y+10) {
          printf("I am fine here");
      } else {
          printf("I should not reach here");
          ERROR;
      }
  }
}
```

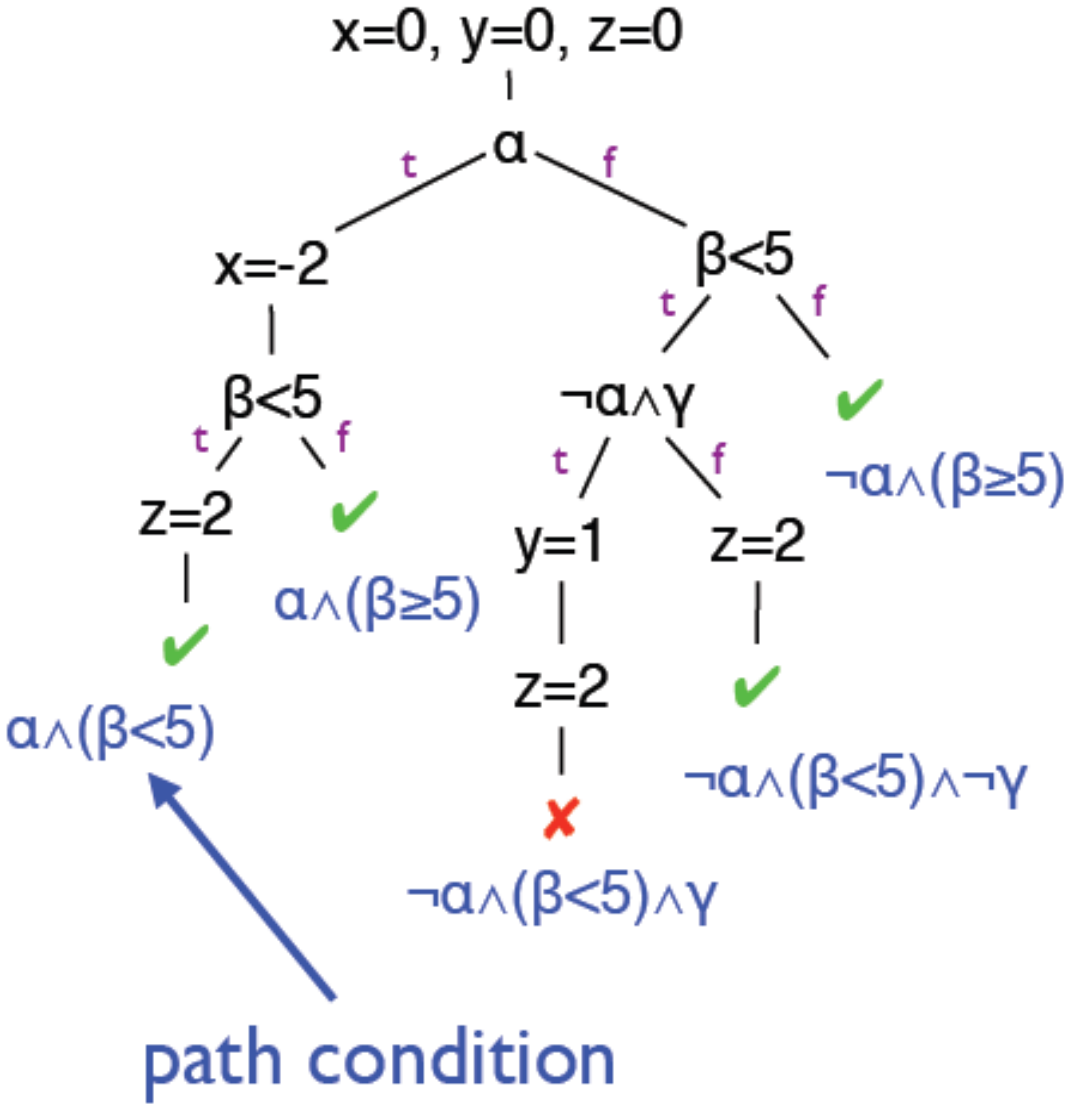


Example of Computation Tree

```
void test_me(int x, int y) {
  if(2*x==y){
     if(x != y+10){
          printf("I am fine here");
     } else {
          printf("I should not reach here");
          ERROR;
     }
  }
}
```



```
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);
```



Symbolic Execution: Limitations?

- Sym-ex is computationally infeasible for many programs
  - Tries to find every execution path, which is exponential in the number of branches.
  - Does not **scale** for large programs.

Symbolic Execution: Limitations

```
void again_test_me(int x,int y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y
- solve x*x*x+ 3*x*x+9 = y to take else branch
- **Don't know how to solve !!**
  - **Stuck ?**

Concolic Testing

CONCRETE EXECUTION + SYMB**OLIC** EXECUTION

= **CONCOLIC** EXECUTION

Koushik Sen, Darko Marinov, and Gul Agha. 2005. **CUTE: a concolic unit testing engine for C.** In *Proc. International Symposium on Foundations of Software Engineering* (ESEC/FSE-13). ACM, 263-272.

Goal

Automated Unit Testing of real-world Programs

- Generate test inputs
- Execute unit under test on generated test inputs
  - so that all reachable statements are executed
- Any generic assertion violation gets caught
  - Examples: exceptions thrown, crashes, out of memory issues, etc

Concolic Execution

- Generates **concrete inputs** one by one.
- The program is executed on that input simultaneously (both **concretely** & **symbolically**).
- The symb. ex. follows the path taken by the concrete ex. maintains a symb. state, generates path constraints.
- The info. collected by the concrete ex. guides symb. ex. and is used to generate inputs for the next ex.
- This process continues until all different feasible paths are executed at least once.

Concolic vs. Symbolic Execution

- Sym-ex is computationally infeasible for many programs
  - Tries to find every execution path, which is exponential in the number of branches.
- Concolic tries to reduce the number of paths and also addresses problems with constraint solver
  - If constraint can't be solved, it defaults to **random values**.
  - Symbolic execution helps to generate concrete inputs for next execution.

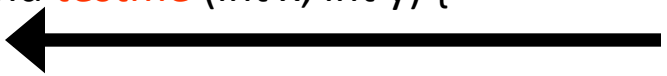
Concolic Testing Example

```
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 Test Driver:
  - random values for x and y
- Probability of reaching ERROR is extremely low

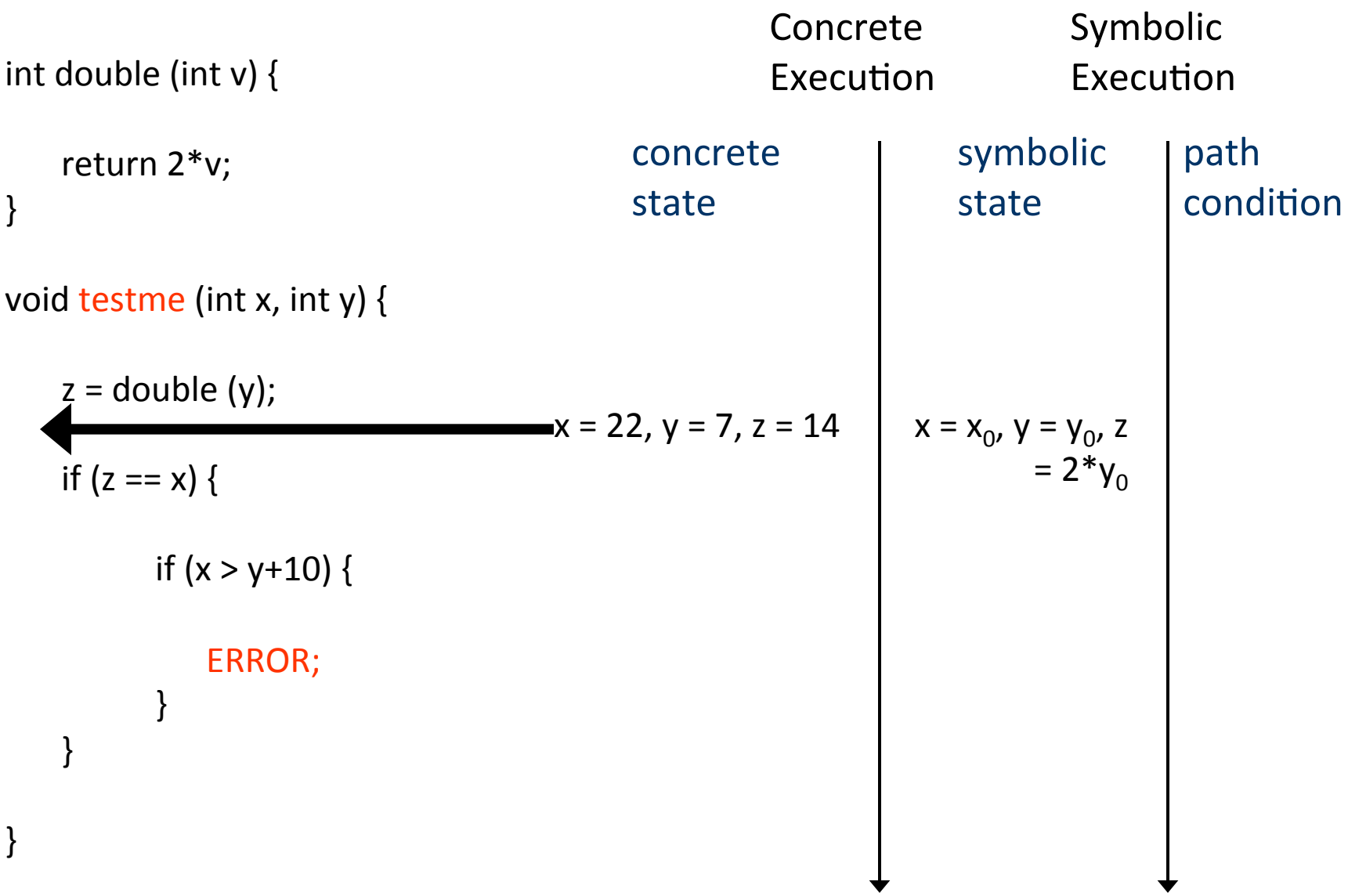
Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

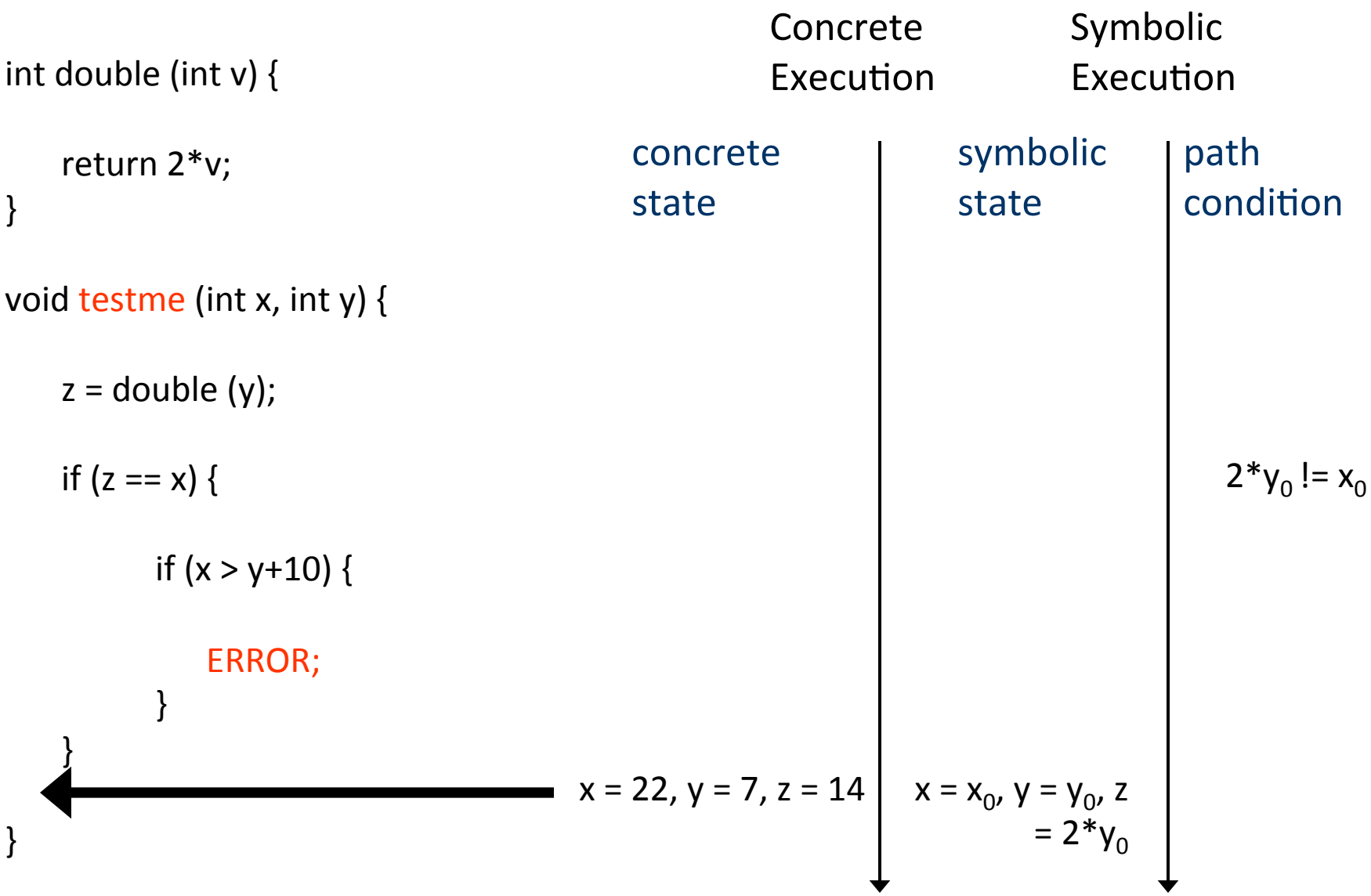


Concrete Execution: concrete state	Symbolic Execution: symbolic state	path condition
x = 22, y = 7	$x = x_0, y = y_0$	

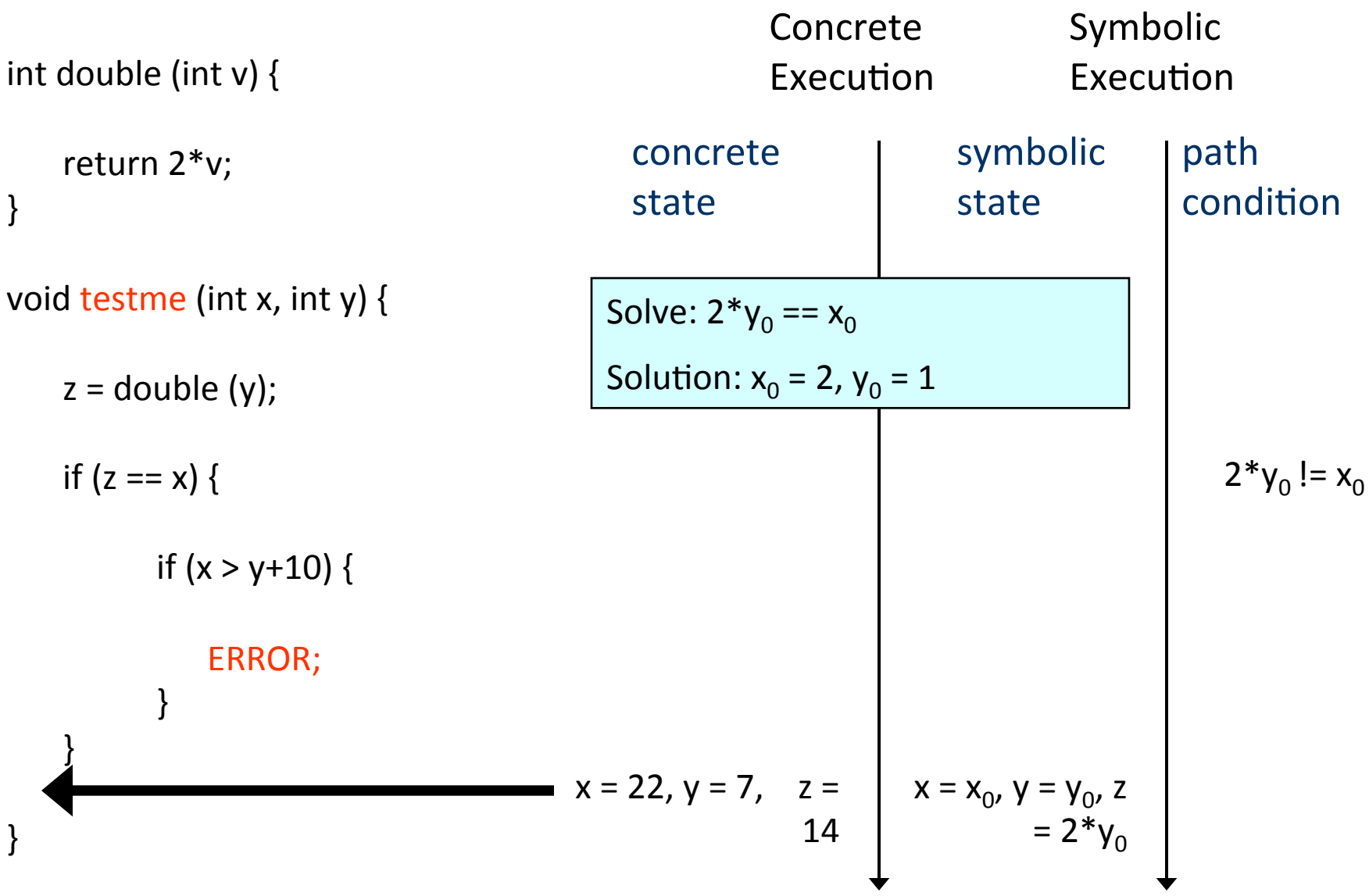
Concolic Testing Approach



Concolic Testing Approach

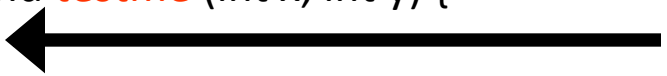


Concolic Testing Approach



Concolic Testing Approach

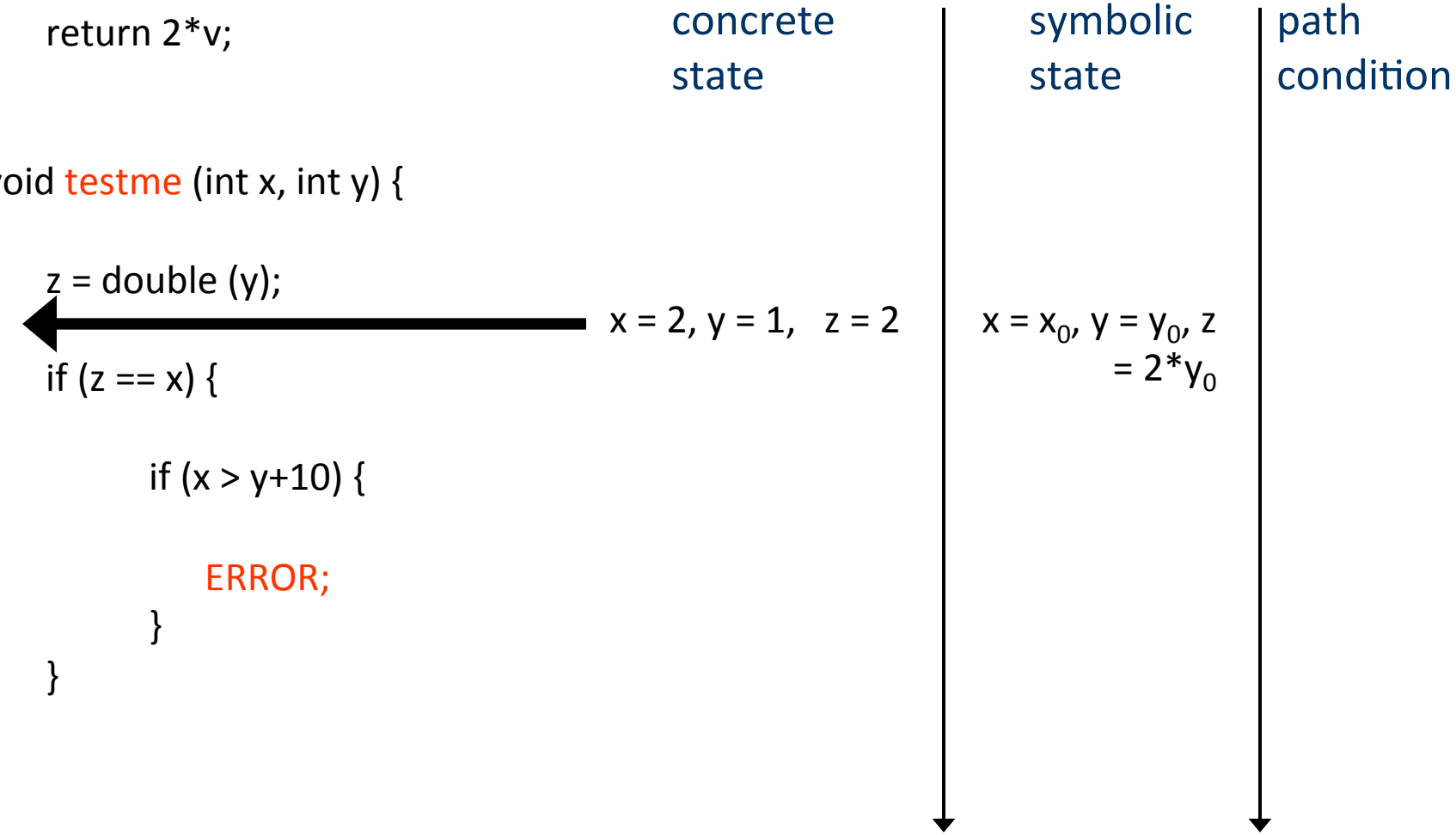
```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```



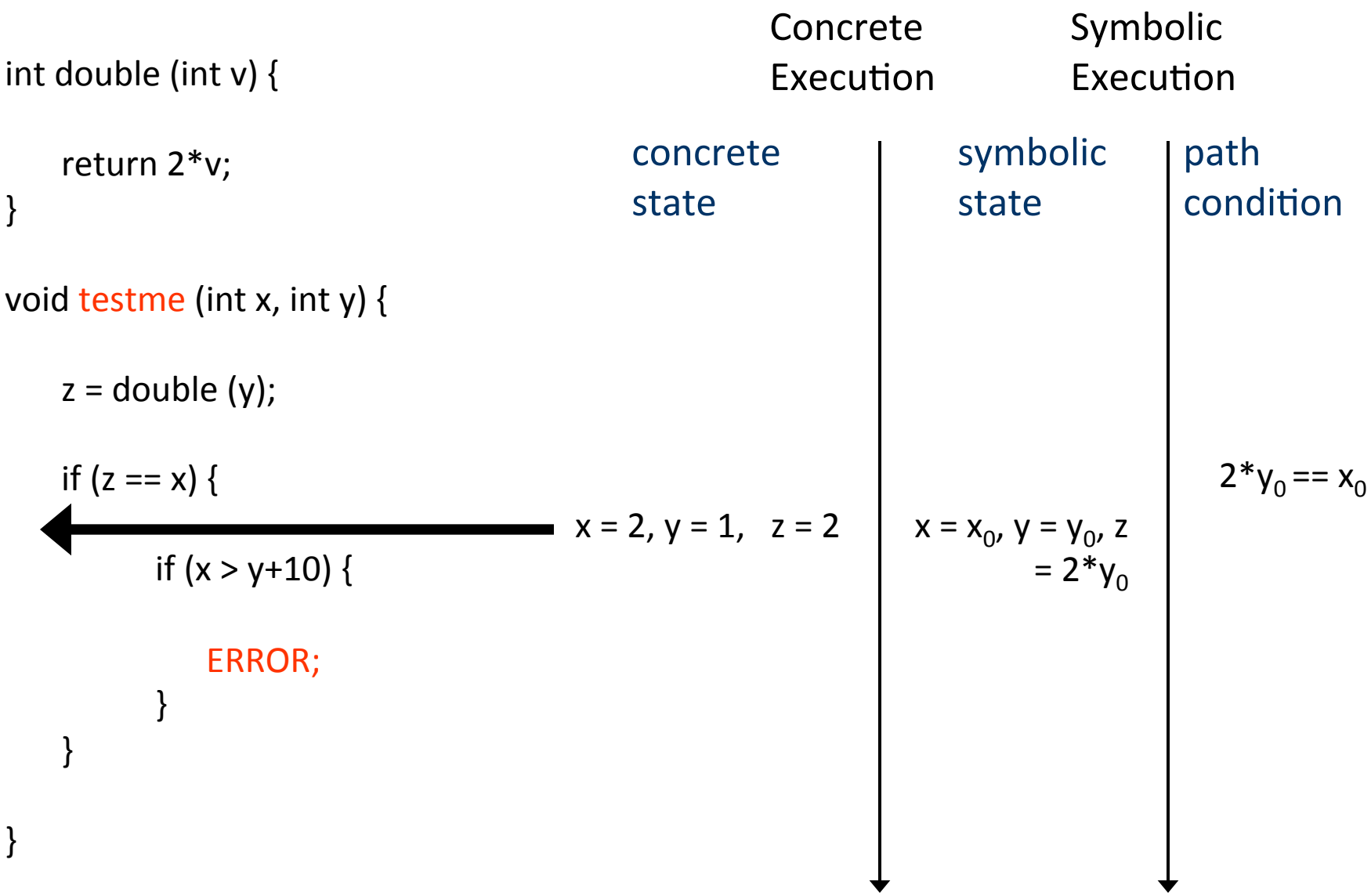
Concrete Execution: concrete state	Symbolic Execution: symbolic state	path condition
x = 2, y = 1	$x = x_0, y = y_0$	

Concolic Testing Approach

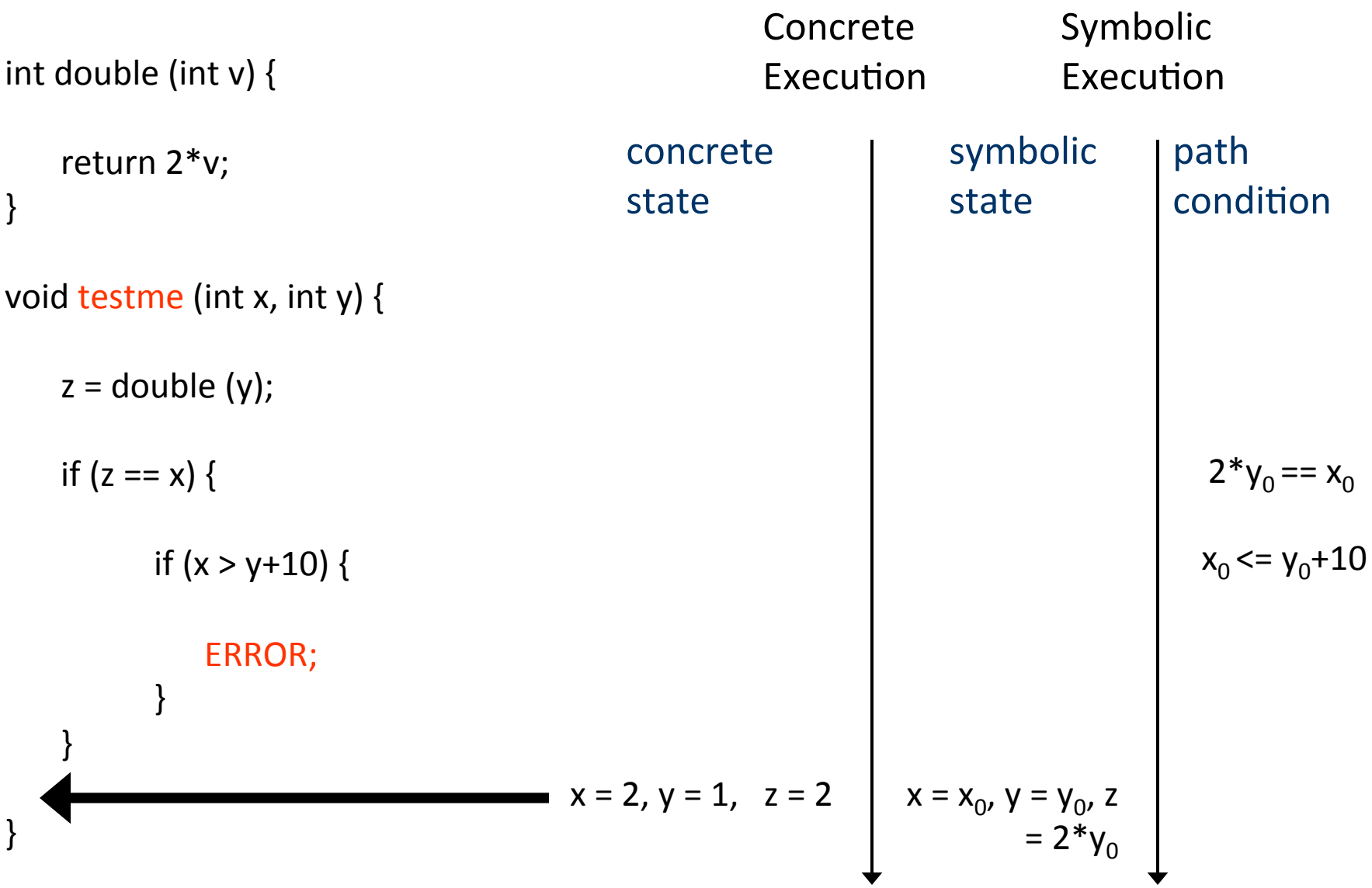
```
int double (int v) {
    return 2*v;
}

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



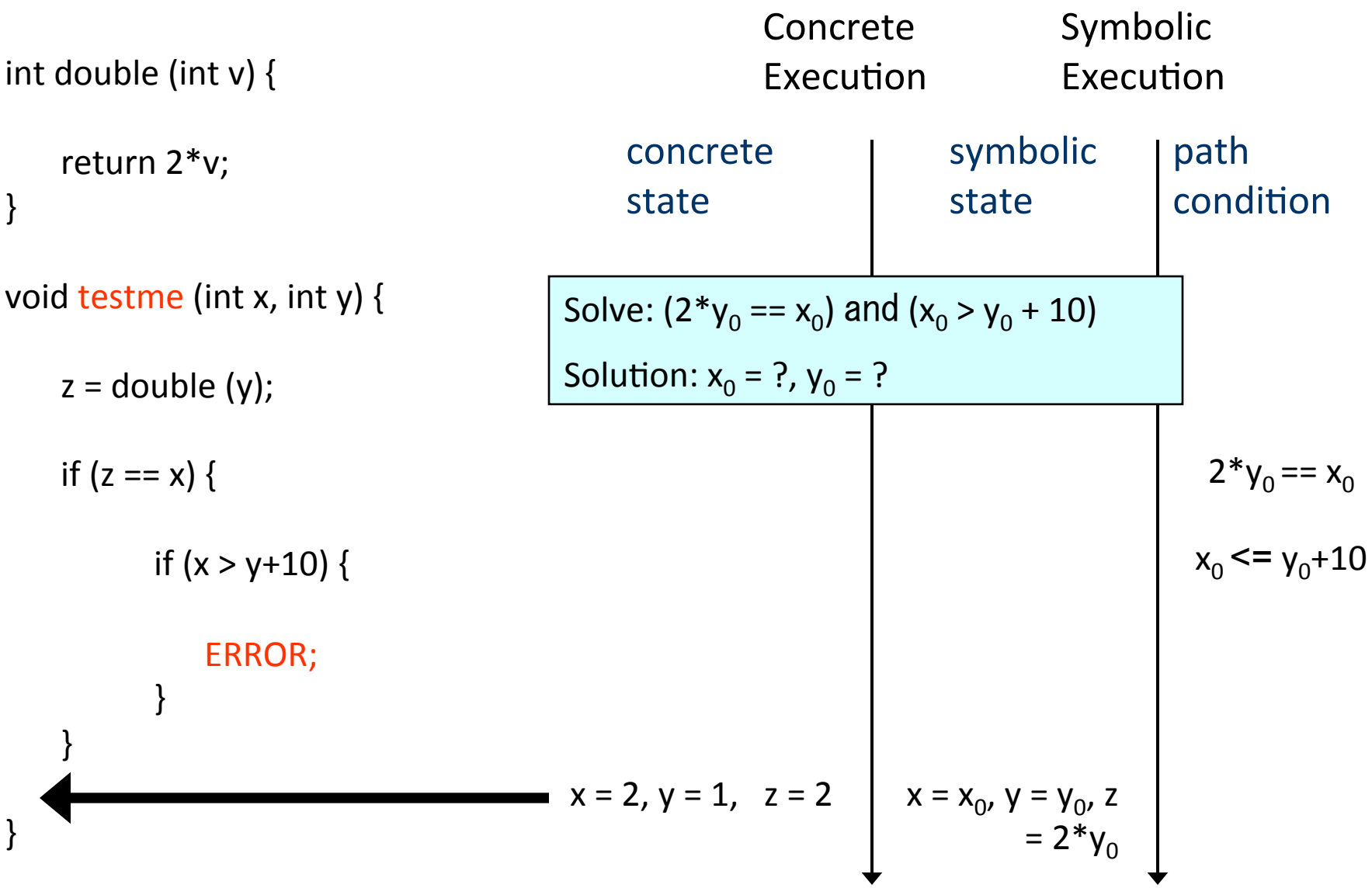
Concolic Testing Approach



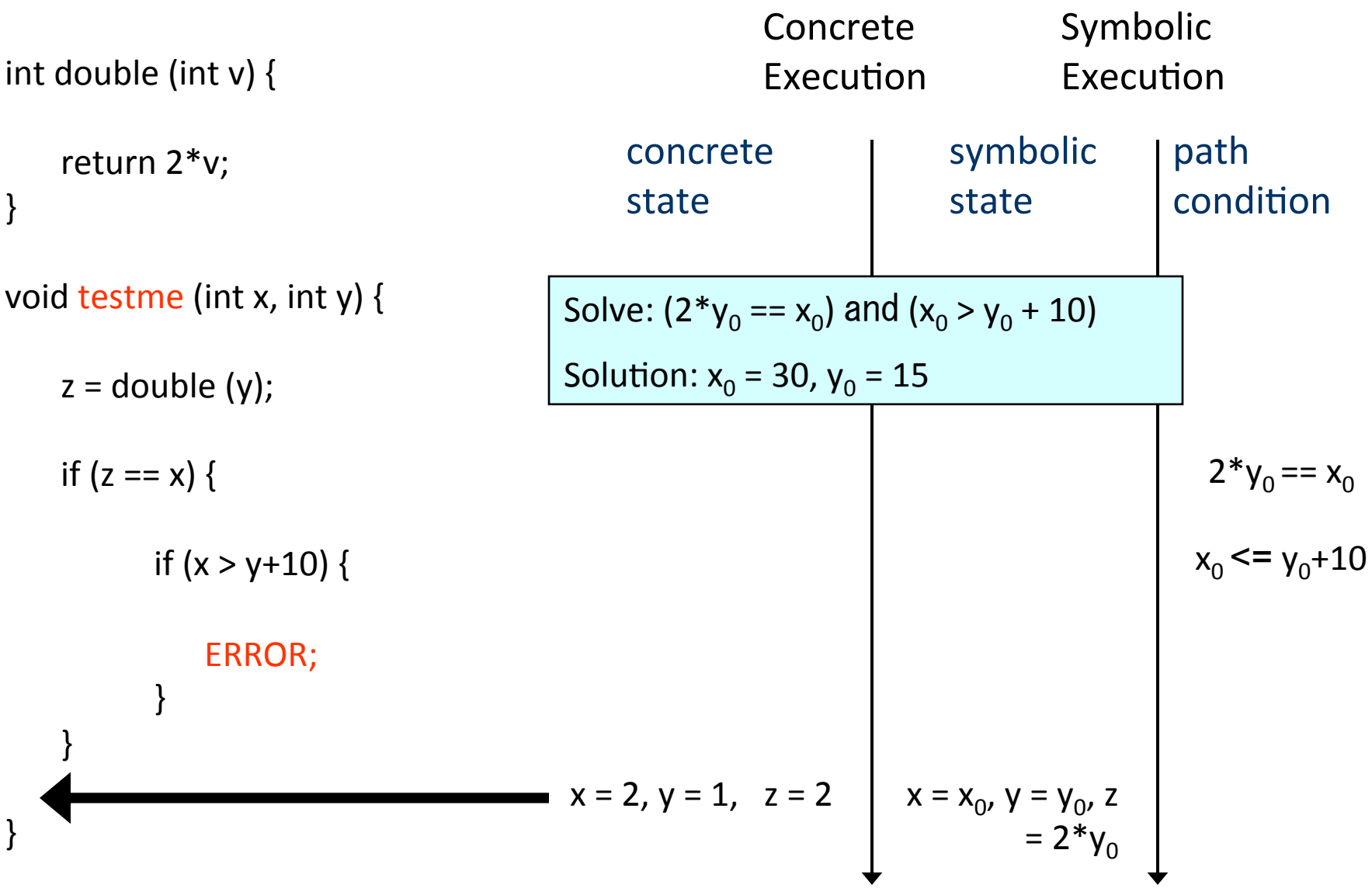
Concolic Testing Approach



Concolic Testing Approach

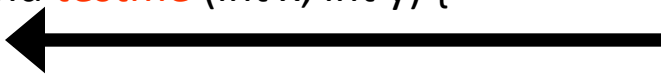


Concolic Testing Approach



Concolic Testing Approach

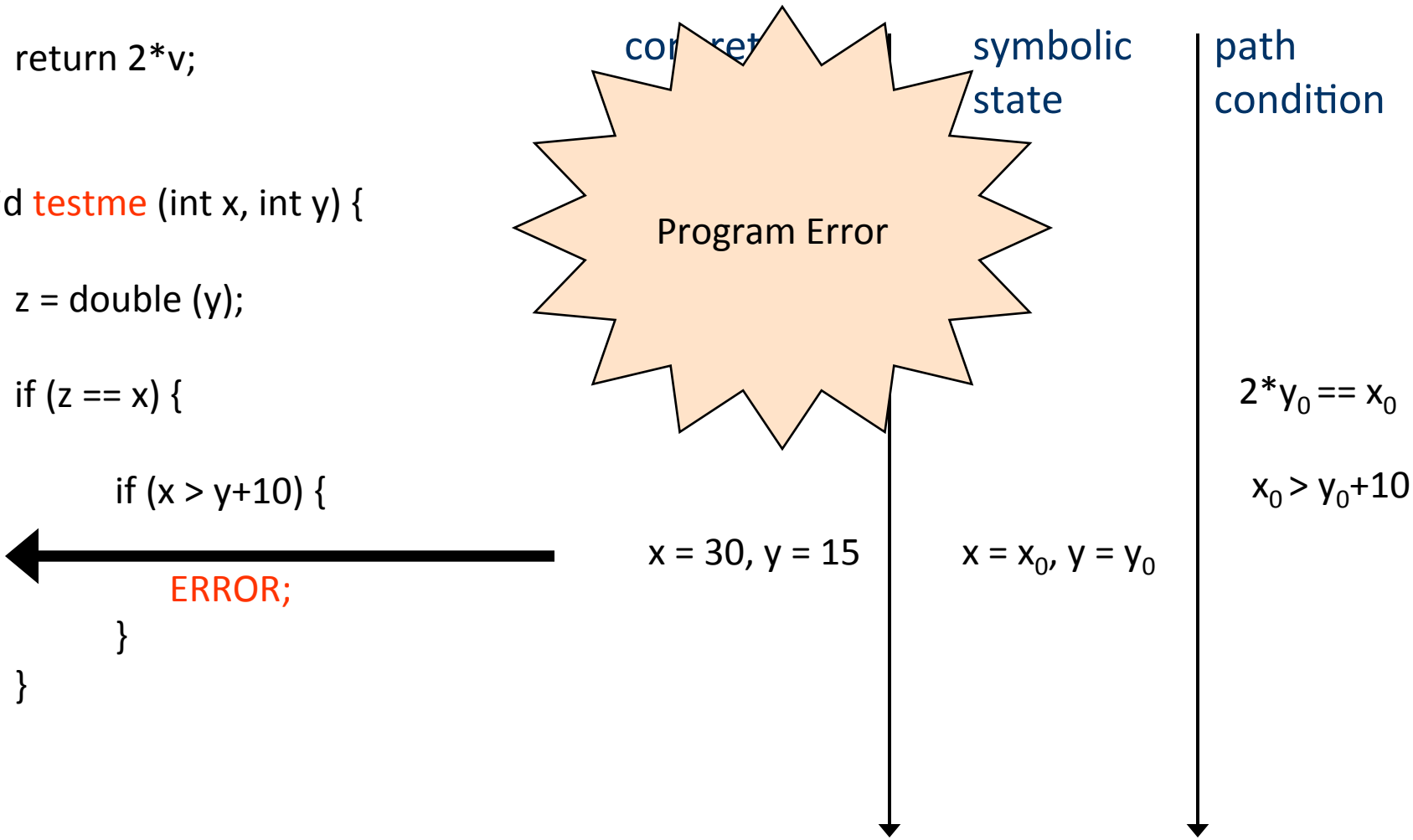
```
int double (int v) {
    return 2*v;
}

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
x = 30, y = 15	$x = x_0, y = y_0$	

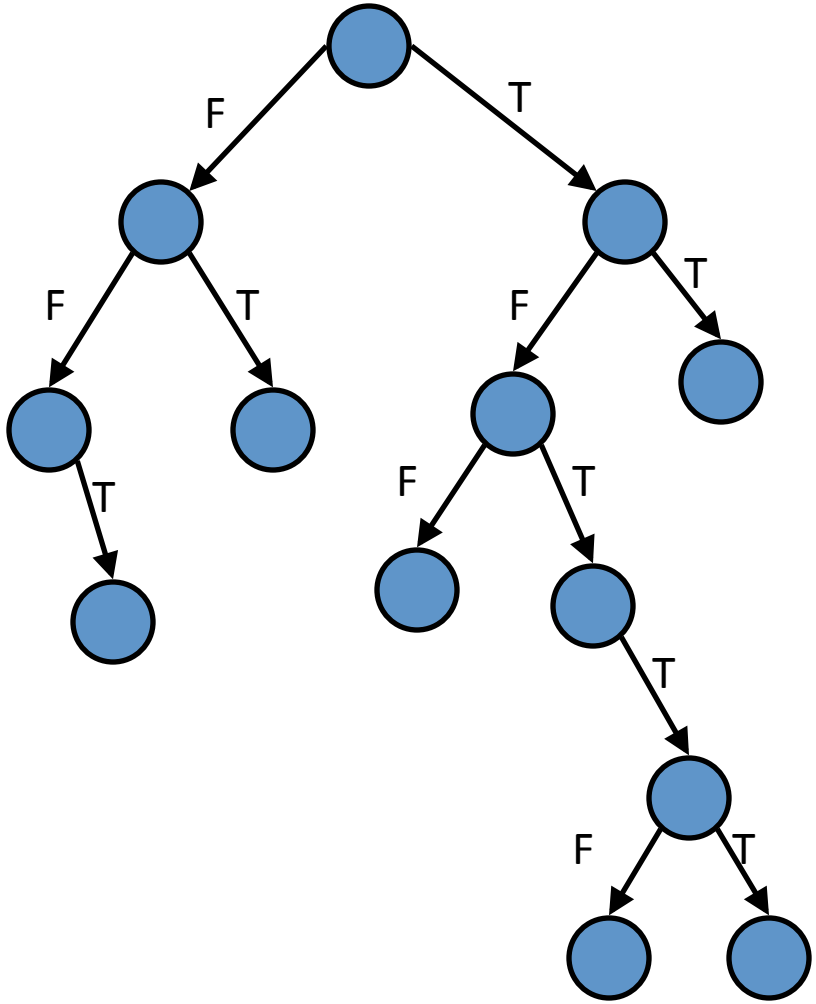
Concolic Testing Approach

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

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



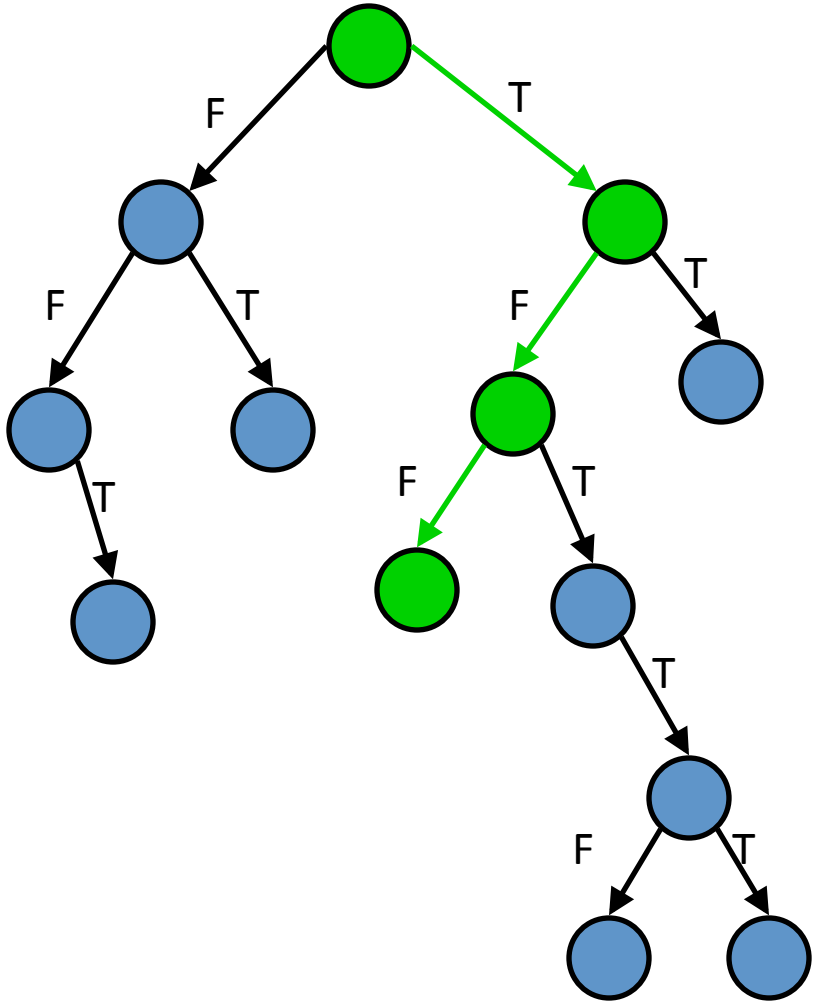
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions



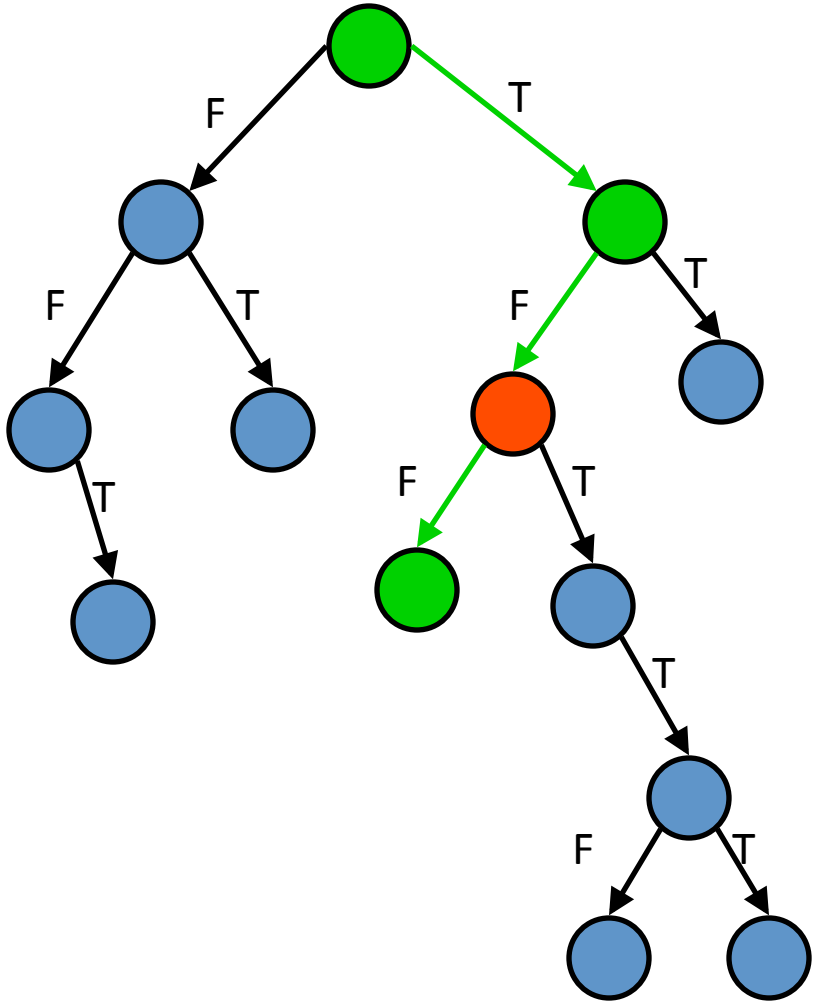
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions



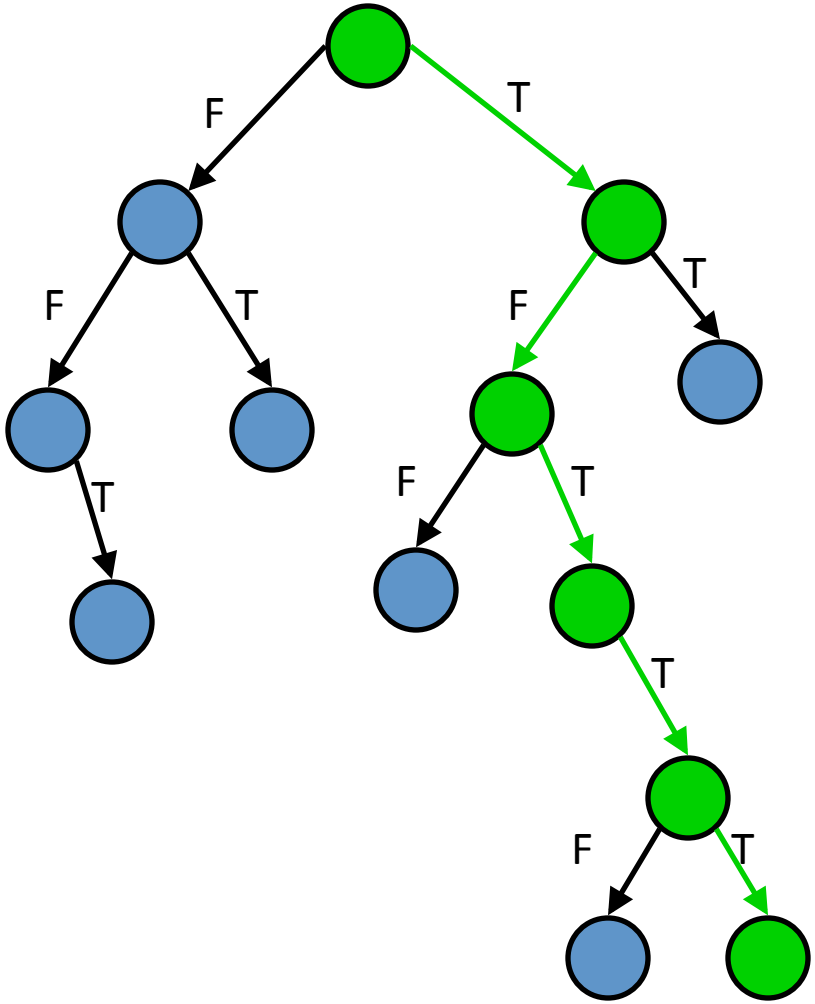
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions



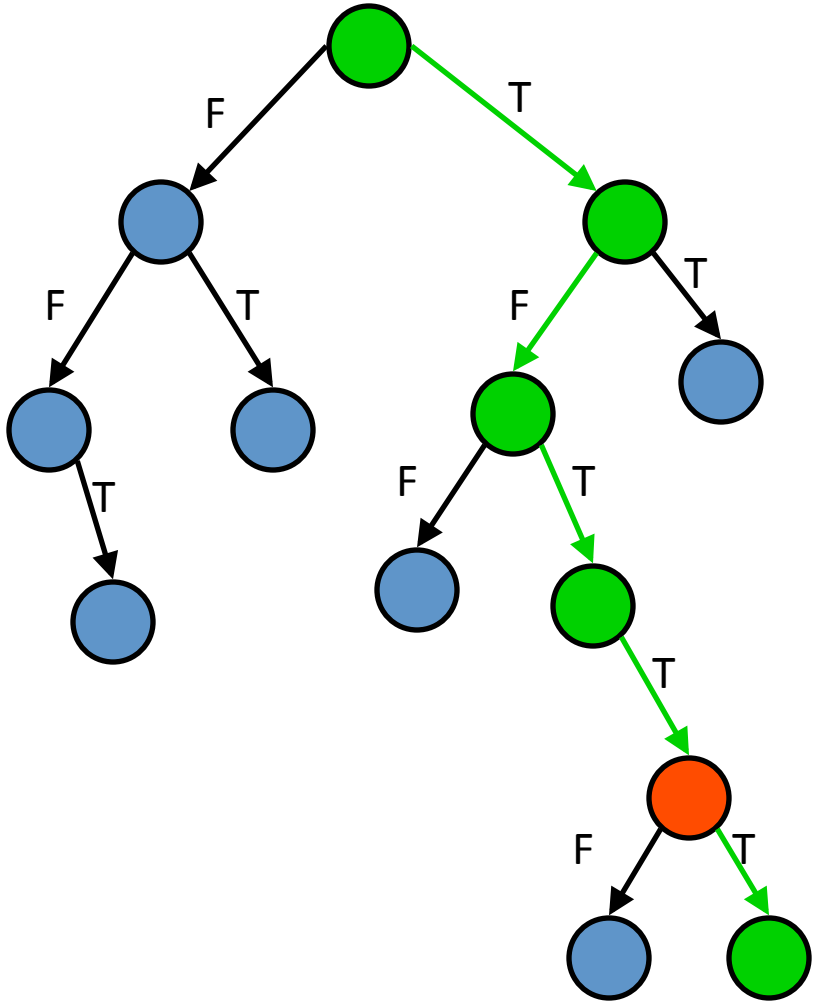
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions



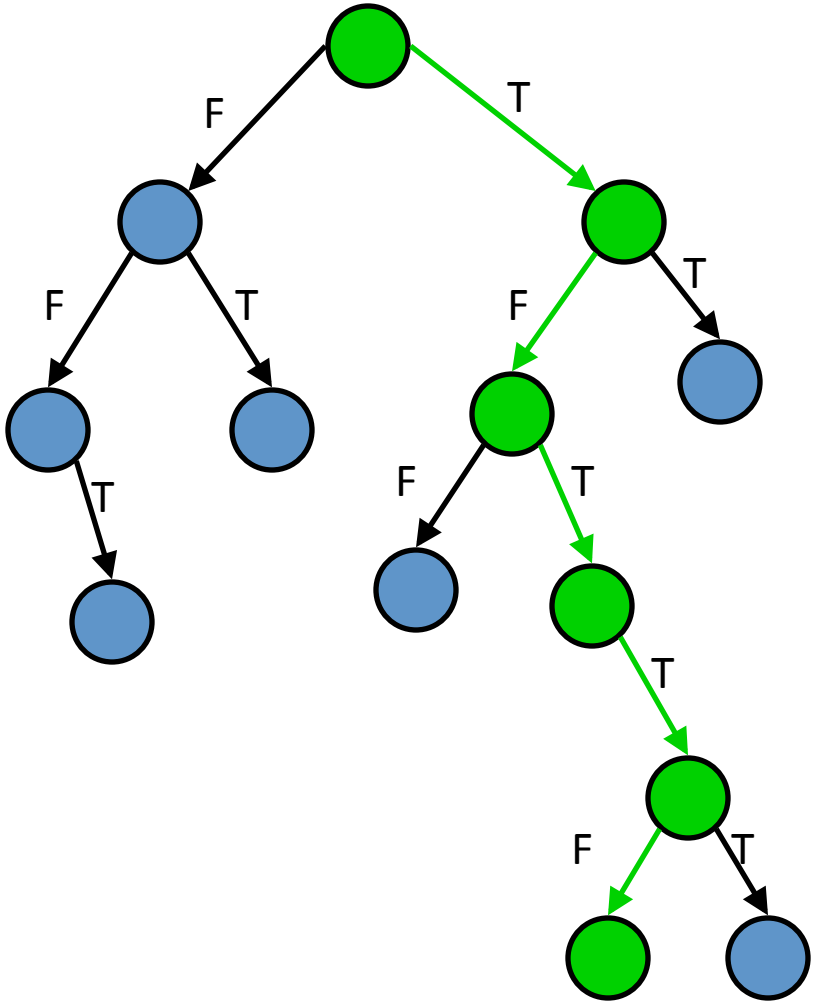
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions



Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions



Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y)
    {
    z = x*x*x + 3*x*x + 9;
    if(z != y){
            printf("Good branch");
    } else {
            printf("Bad branch");
            abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y
- solve x*x*x+ 3*x*x+9 = y to take else branch
- Don't know how to solve !!
  - **Stuck ?**

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y)
    {
    z = x*x*x + 3*x*x + 9;
    if(z != y){
            printf("Good branch");
    } else {
            printf("Bad branch");
            abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y
- solve x*x*x+ 3*x*x+9 = y to take else branch
- Don't know how to solve !!
  - Stuck ?
  - NO : concolic testing handles this.

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y)
    {
    z = x*x*x + 3*x*x + 9;
    if(z != y){
            printf("Good branch");
    } else {
            printf("Bad branch");
            abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y)
    {
    z = x*x*x + 3*x*x + 9;
    if(z != y){
            printf("Good branch");
    } else {
            printf("Bad branch");
            abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y)
    {
    z = x*x*x + 3*x*x + 9;
    if(z != y){
            printf("Good branch");
    } else {
            printf("Bad branch");
            abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y)
    {
    z = x*x*x + 3*x*x + 9;
    if(z != y){
            printf("Good branch");
    } else {
            printf("Bad branch");
            abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int y)
    {
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y
- solve 9 = y to take else branch
- execute next run with x = -3 and y= 9
  - got error (reaches abort)

Simultaneous Symbolic & Concrete Execution

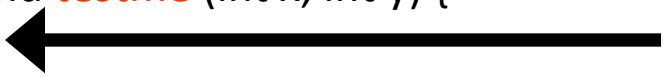
```
void again_test_me(int x,int y)
    {
    z = x*x*x + 3*x*x + 9;
    if(z != y){
}
```

Replace symbolic expression by concrete value when symbolic expression becomes unmanageable (i.e. non-linear)

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y
- solve 9 = y to take else branch
- execute next run with x = -3 and y= 9
  - got error (reaches abort)

Novelty: Simultaneous Concrete and Symbolic Execution

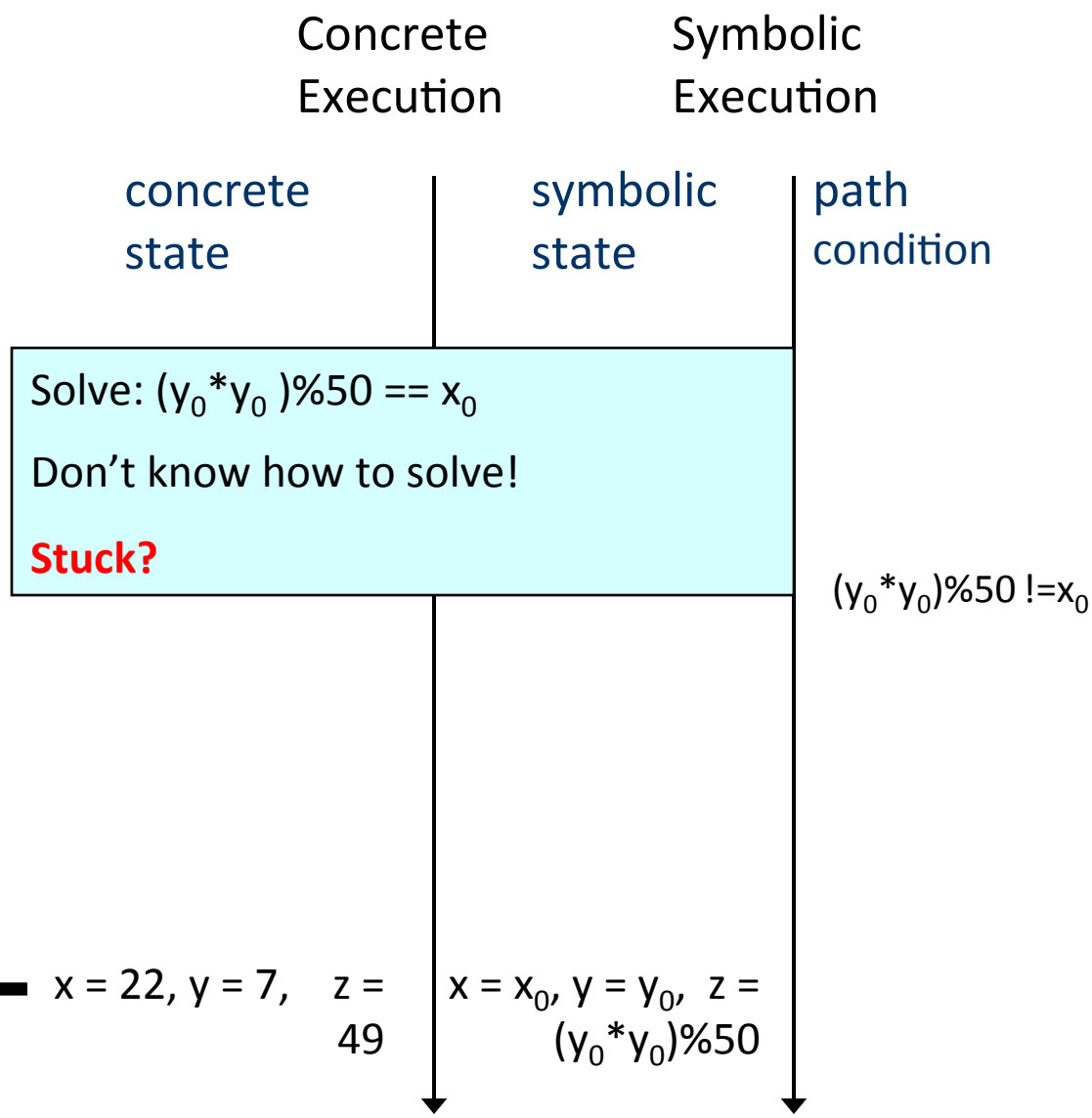
```
int foo (int v) {
    return (v*v) % 50;
}

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



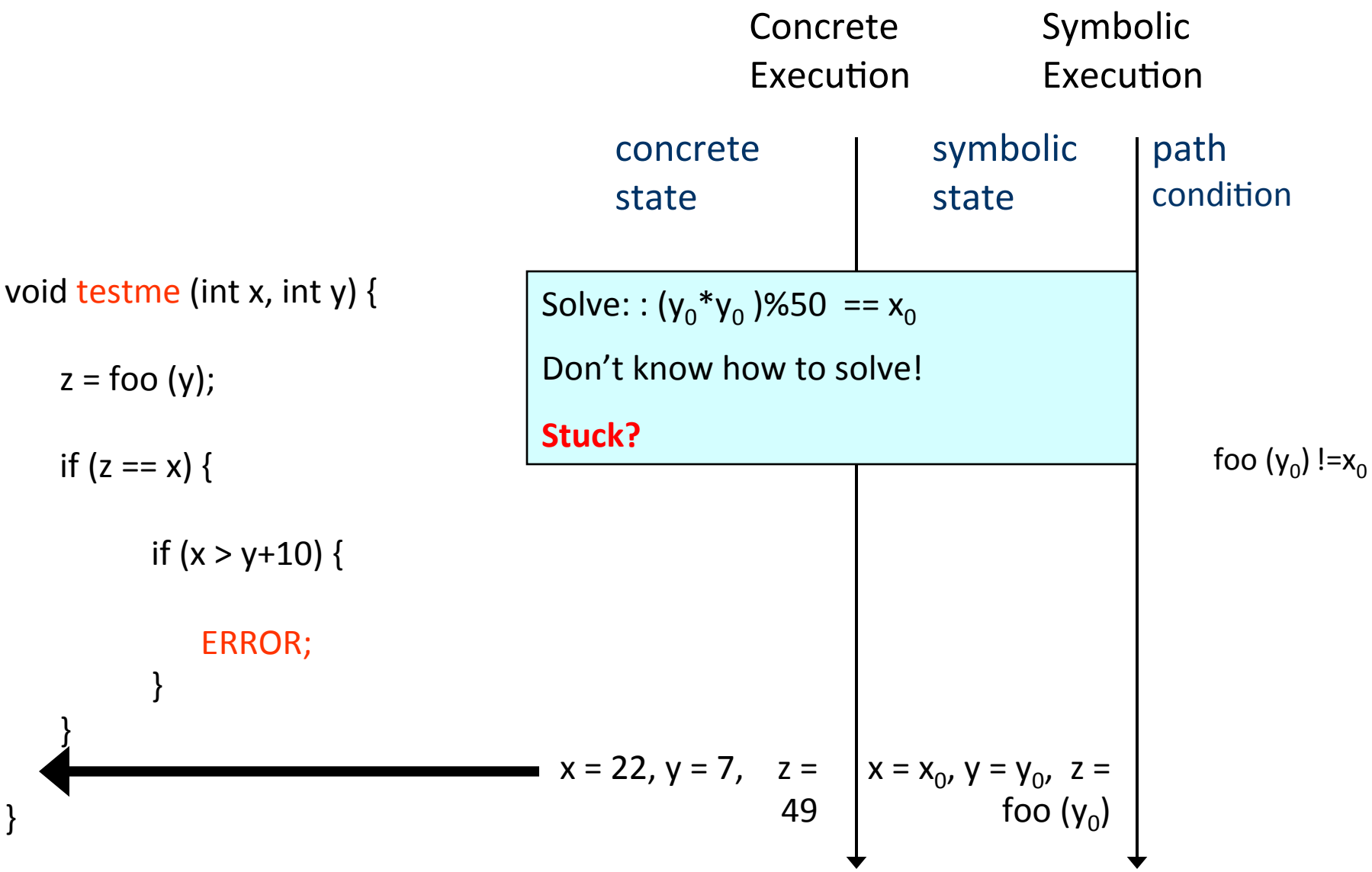
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22, y = 7	$x = x_0, y = y_0$	

Novelty: Simultaneous Concrete and Symbolic Execution

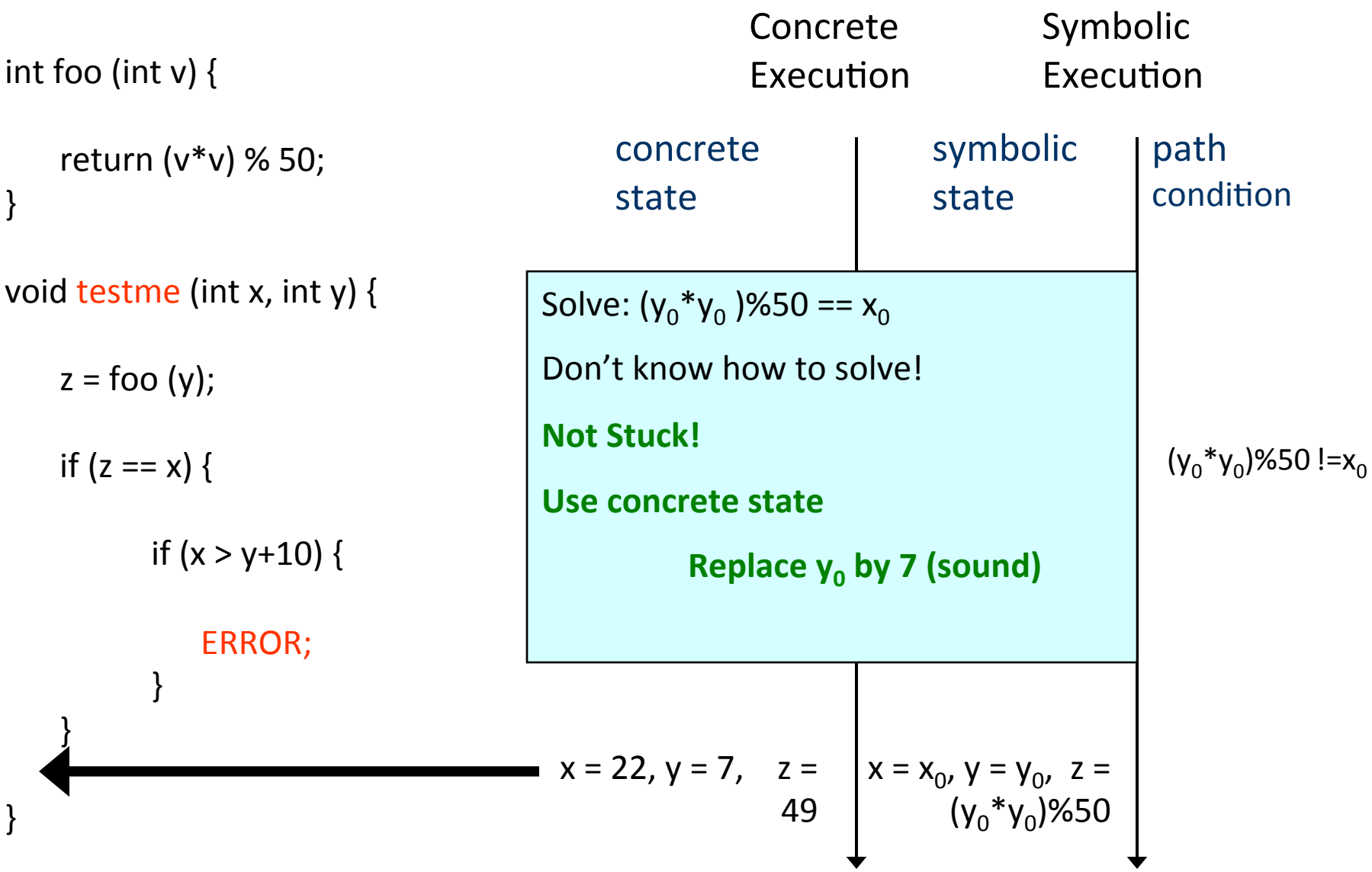
```
int foo (int v) {
    return (v*v) % 50;
}

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



Novelty: Simultaneous Concrete and Symbolic Execution

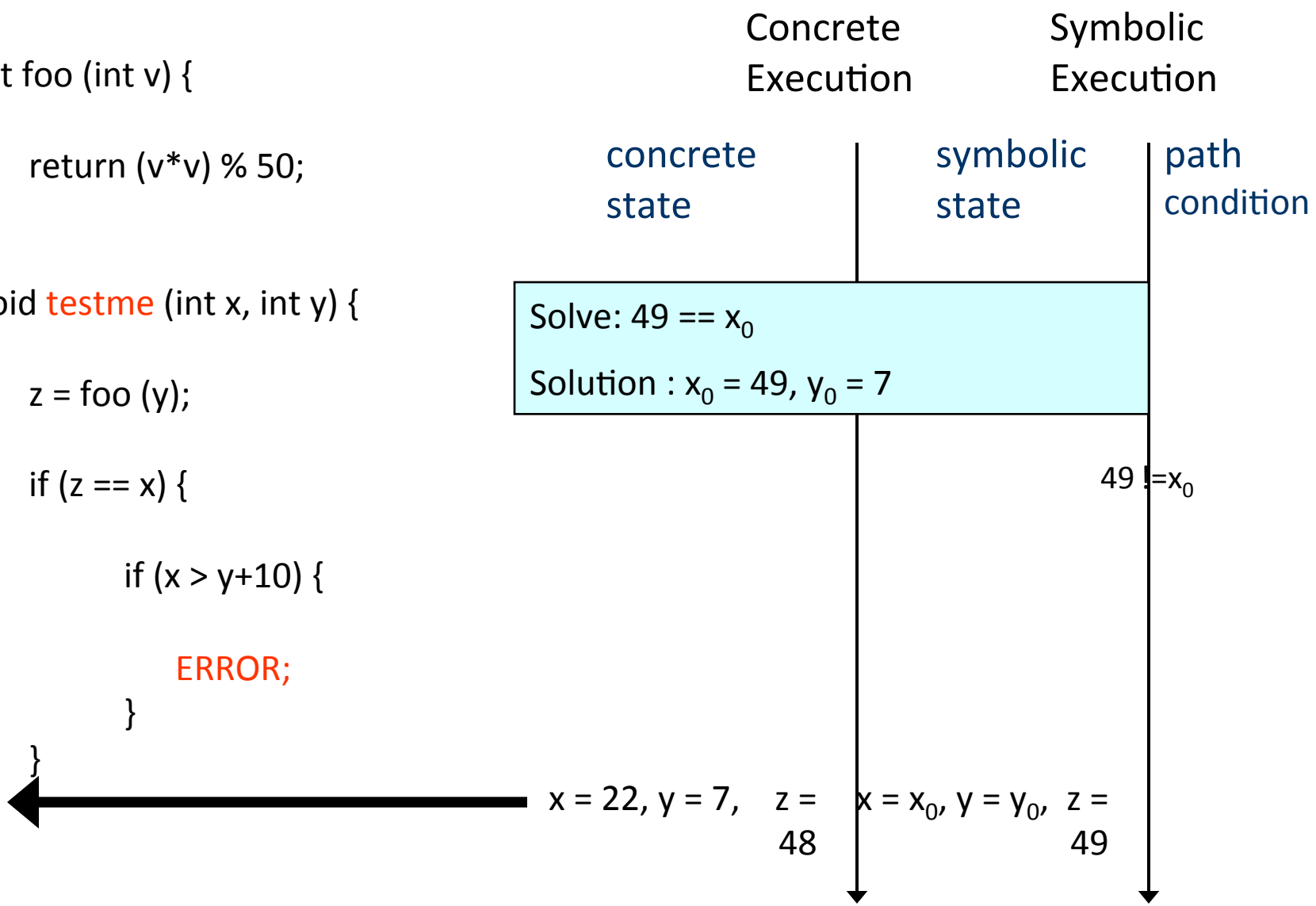


Novelty: Simultaneous Concrete and Symbolic Execution



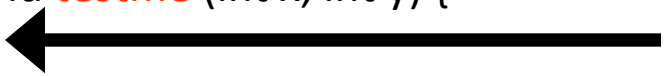
Novelty: Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

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



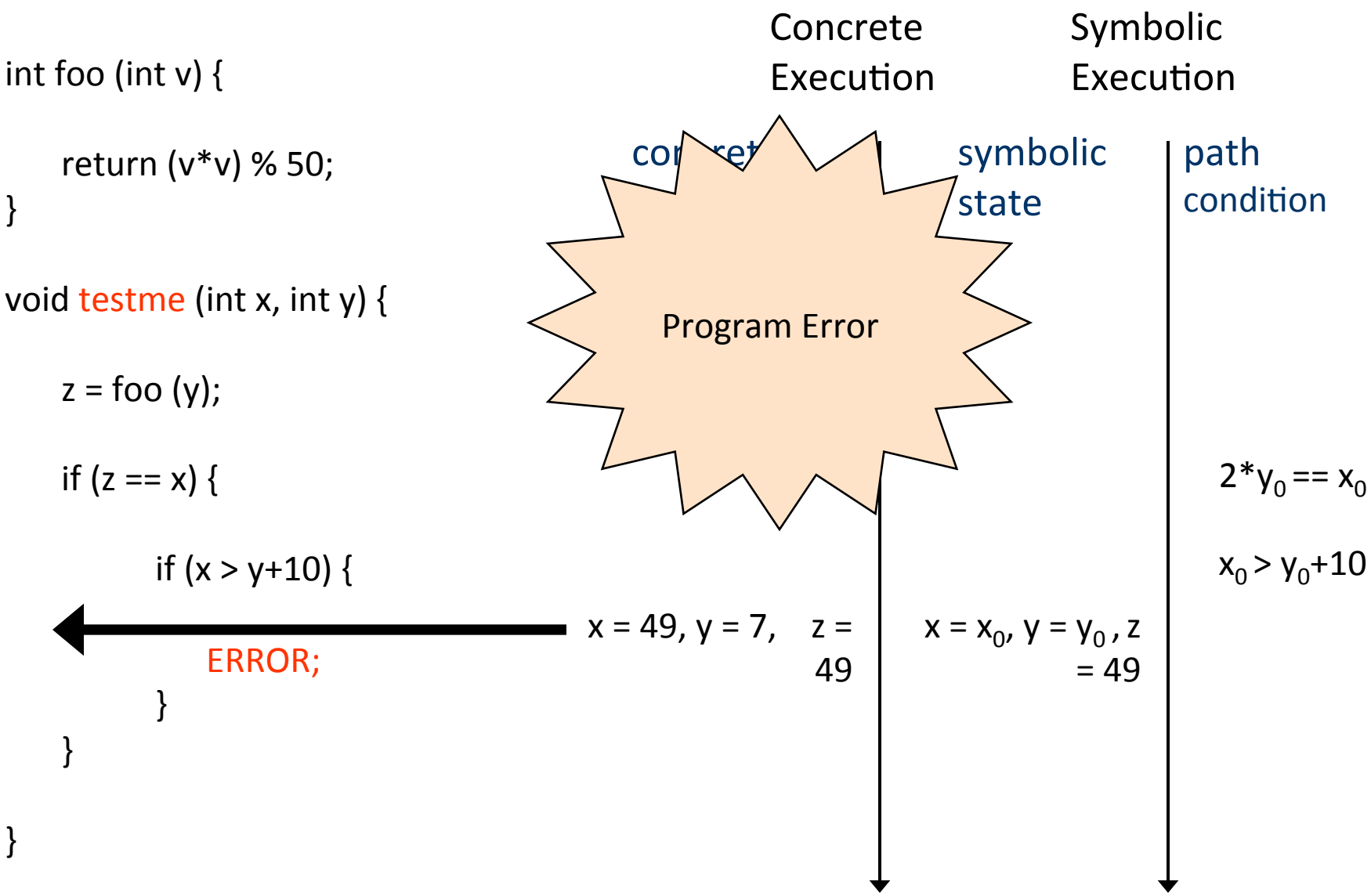
Novelty: Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

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



Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 49, y = 7	$x = x_0, y = y_0$	

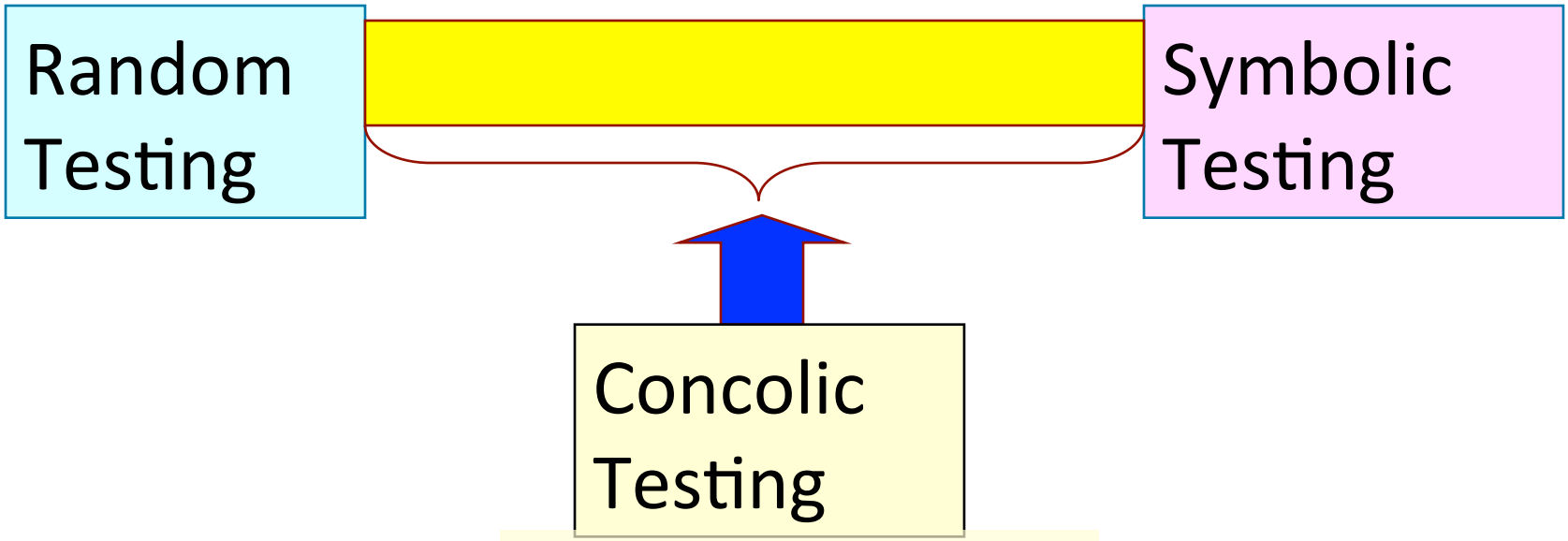
Novelty: Simultaneous Concrete and Symbolic Execution



Concolic Execution: Handling unmanageable symbolic expressions

- Concrete execution enables symbolic execution to overcome incompleteness of constraint solvers
  - replace symbolic expressions by concrete values if symbolic expressions become too complex.

Concolic Testing: A Middle Approach



+ Complex programs
+ Efficient
- Less coverage
+ No false positive

+ Complex programs
+/- Somewhat efficient
+ High coverage
+ No false positive

- Simple programs
- Not efficient
+ High coverage
- False positive

Implementations

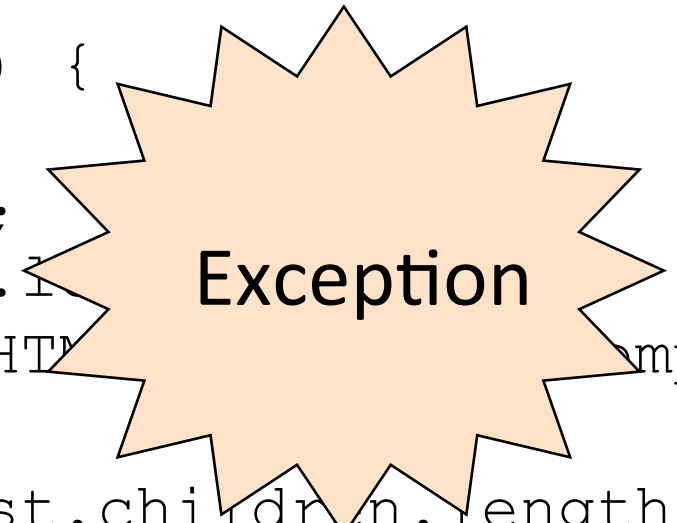
- DART and CUTE for C programs
- jCUTE for Java programs
- Microsoft has four implementations
  - SAGE, PEX, YOGI, Vigilante
- EXE (Stanford)
- JavaScript
  - Jalangi: function arguments (Berkeley)
  - ConFix: DOM interactions (SALT Lab @ UBC)

```
function sumTotalPrice() {
  sum = 0;
  itemList = $("items");
  if (itemList.children.length == 0)
    dg("message").innerHTML = "List is empty!";
  else {
    for(i = 0; i<itemList.children.length; i++){
      p = parseInt(itemList.children[i].value);
      if (p > 0) sum += p;
      else $("message").innerHTML += "Wrong
                value " + i;
    }
   $("total").innerHTML = sum;
  }
  return sum;
}
//$ = document.getElementById
```

A test case

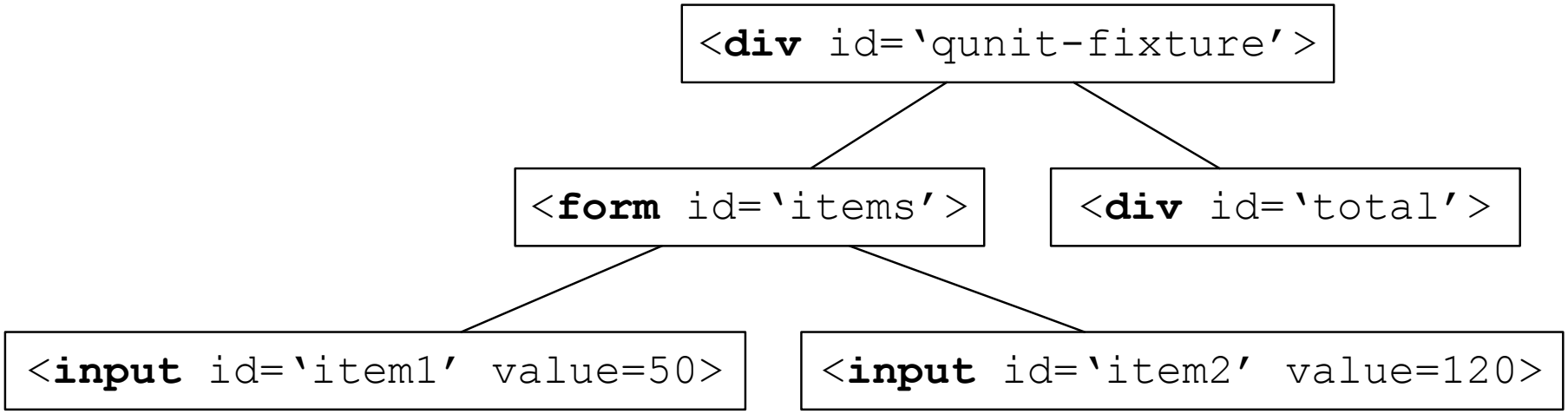
```
test("A test for sumTotalPrice", function() {
  sum = sumTotalPrice();
  //equal(sum, 170, "Function sums correctly.");
});
```

```
function sumTotalPrice() {
  sum = 0;
  itemList = $("items");
  if (itemList.children.l
    dg("message").innerHT                    mpty!";
  else {
    for(i = 0; i<itemList.children.length; i++){
      p = parseInt(itemList.children[i].value);
      if (p > 0) sum += p;
      else $("message").innerHTML += "Wrong
                value " + i;
    }
   $("total").innerHTML = sum;
  }
  return sum;
}
//$ = document.getElementById
```



```
function sumTotalPrice() {
  sum = 0;
  itemList = $("items");
  if (itemList.children.length == 0)
    $("message").innerHTML = "List is empty!";
  else {
    for(i = 0; i<itemList.children.length; i++){
      p = parseInt(itemList.children[i].value);
      if (p > 0) sum += p;
      else $("message").innerHTML += "Wrong value " + i;
    }
   $("total").innerHTML = sum;
  }
  return sum;
} //$ = document.getElementById
```

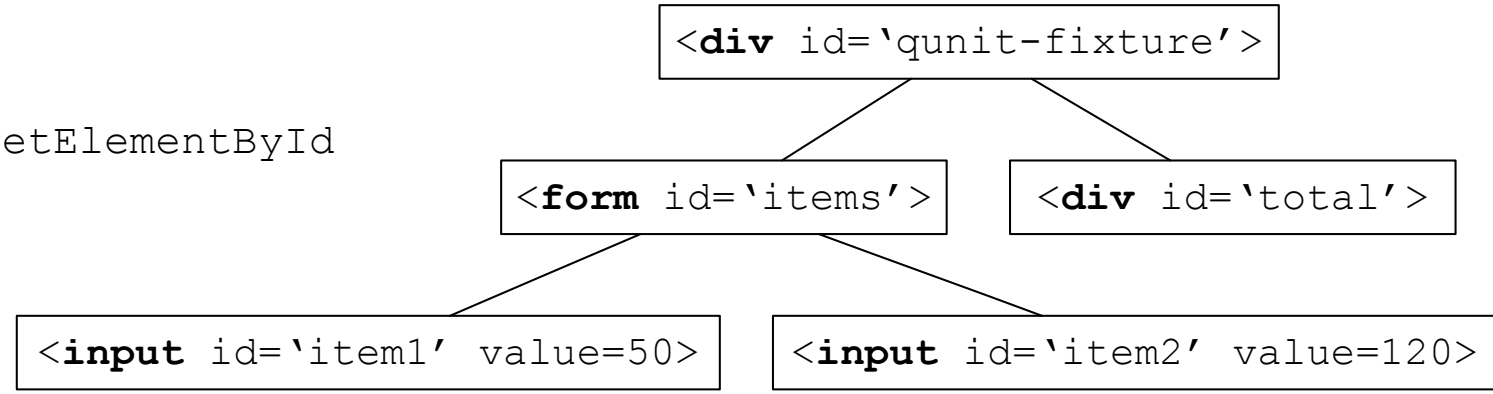
```
test("A test for sumTotalPrice", function() {
  sum = sumTotalPrice();
  equal(sum, 170, "Function sums correctly.");
});
```



<form id='items'>

<input id='item1' value=50>
<input id='item2' value=120>

```
function sumTotalPrice() {
  sum = 0;
  itemList = $("items");
  if (itemList.children.length == 0)
    $("message").innerHTML = "List is empty!";
  else {
    for(i = 0; i<itemList.children.length; i++){
      p = parseInt(itemList.children[i].value);
      if (p > 0) sum += p;
      else $("message").innerHTML += "Wrong value " + i;
    }
   $("total").innerHTML = sum;
  }
  return sum;
}
//$ = document.getElementById
```



Test fixture generated by ConFix

```
test("A test for sumTotalPrice", function() {

  $("#qunit-fixture").append('<form id="items"><
    input type="text" id="item1" value=50><input
    type="text" id="item2" value=120></form><div
    id="total"/>');

   sum = sumTotalPrice();
   equal(sum, 170, "Function sums correctly.");
});
```

Concolic Testing in a Nutshell

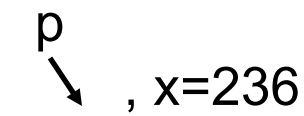
- Generate concrete inputs one by one
  - each input leads program along a different path
- On each input execute program both concretely and symbolically
  - Both cooperate with each other
    - concrete execution guides the symbolic execution
    - concrete execution enables symbolic execution to overcome incompleteness of theorem prover
      - replace symbolic expressions by concrete values if symbolic expressions become complex
    - symbolic execution helps to generate concrete input for next execution
      - increases coverage

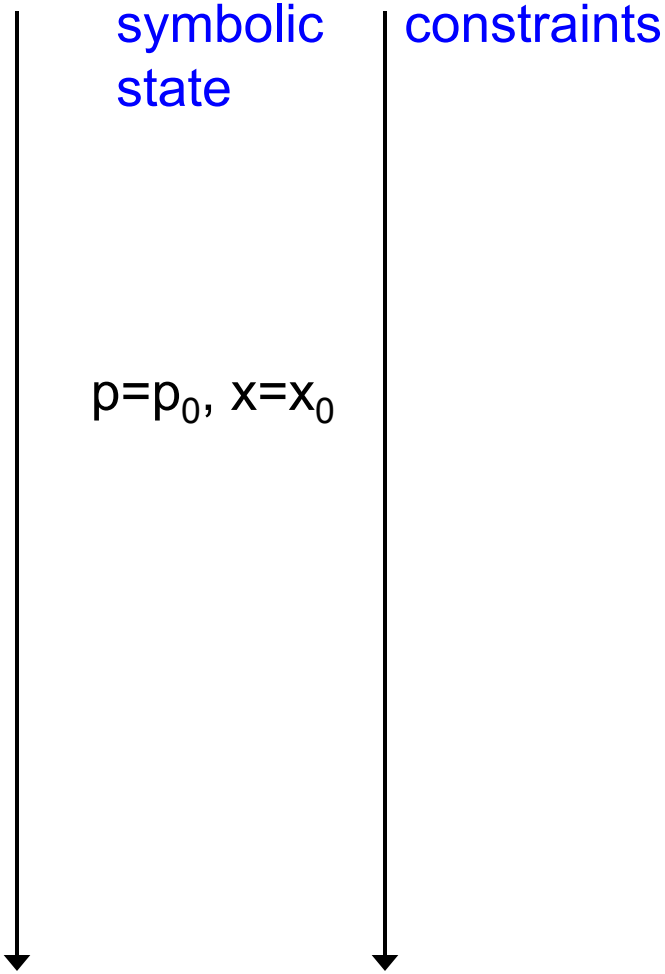
Another example: pointes

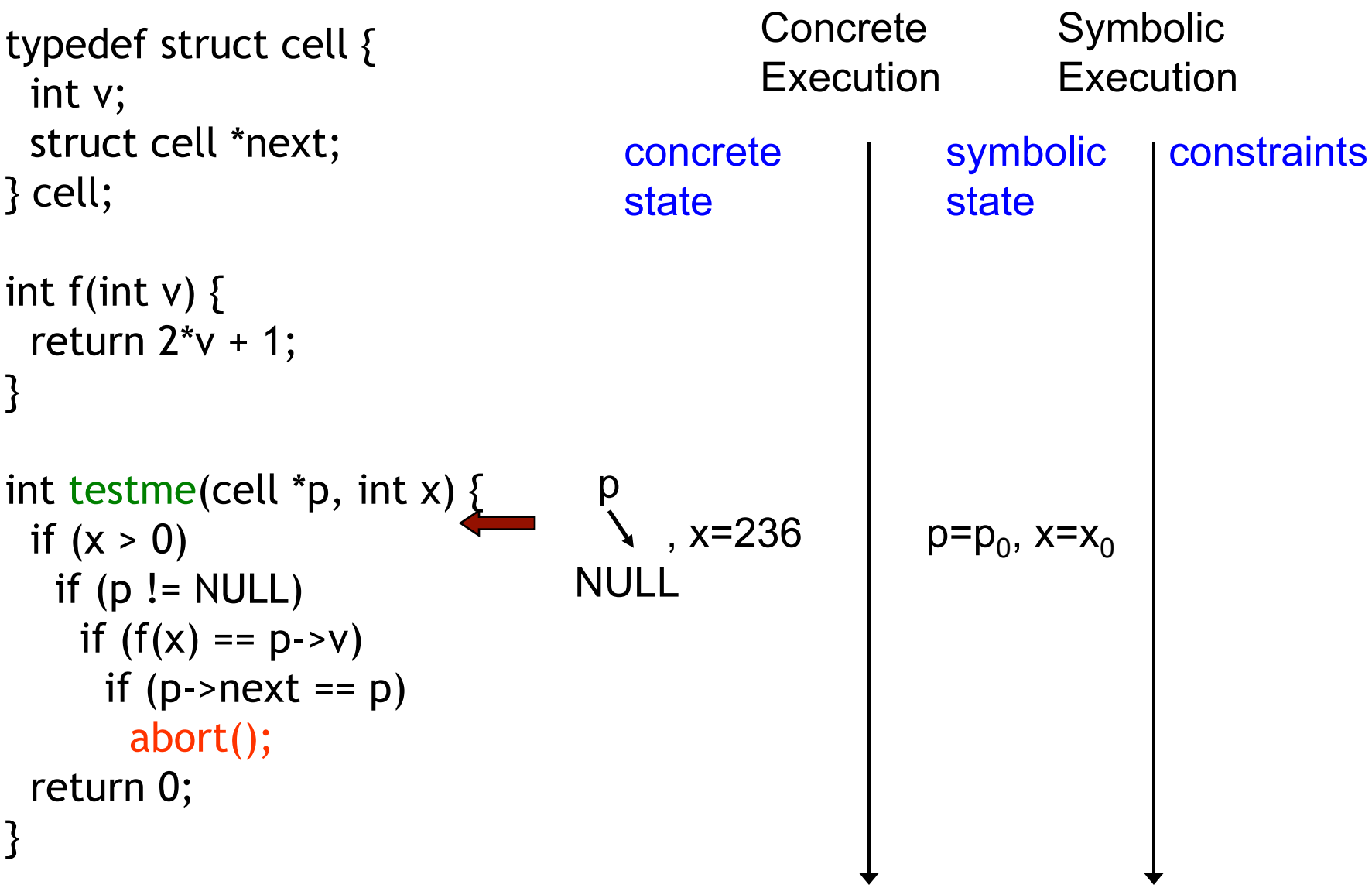
(POINTERS ARE NOT NEEDED FOR FINAL EXAM)

CUTE Approach

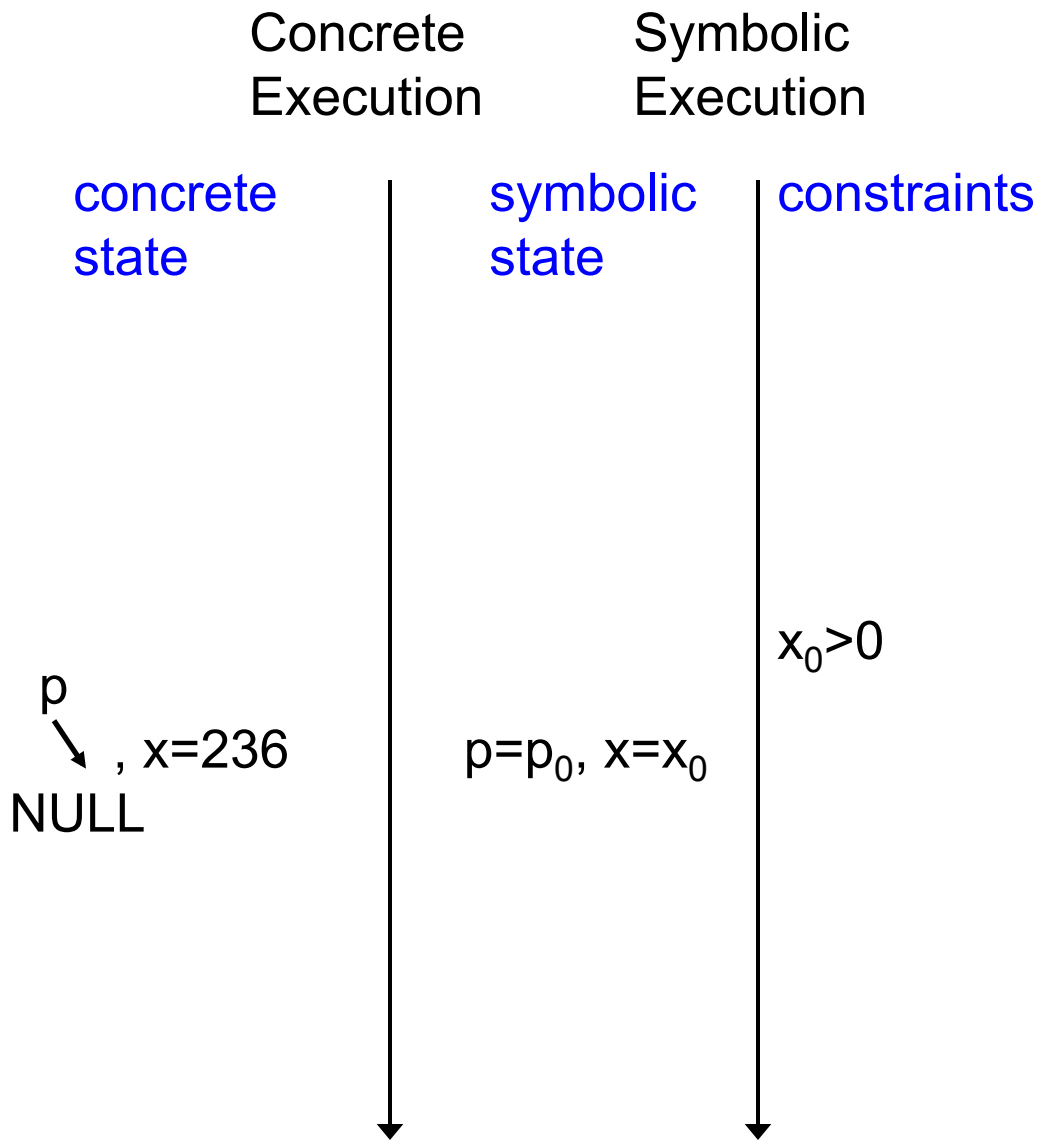
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          abort();
  return 0;
}
```



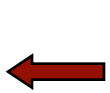


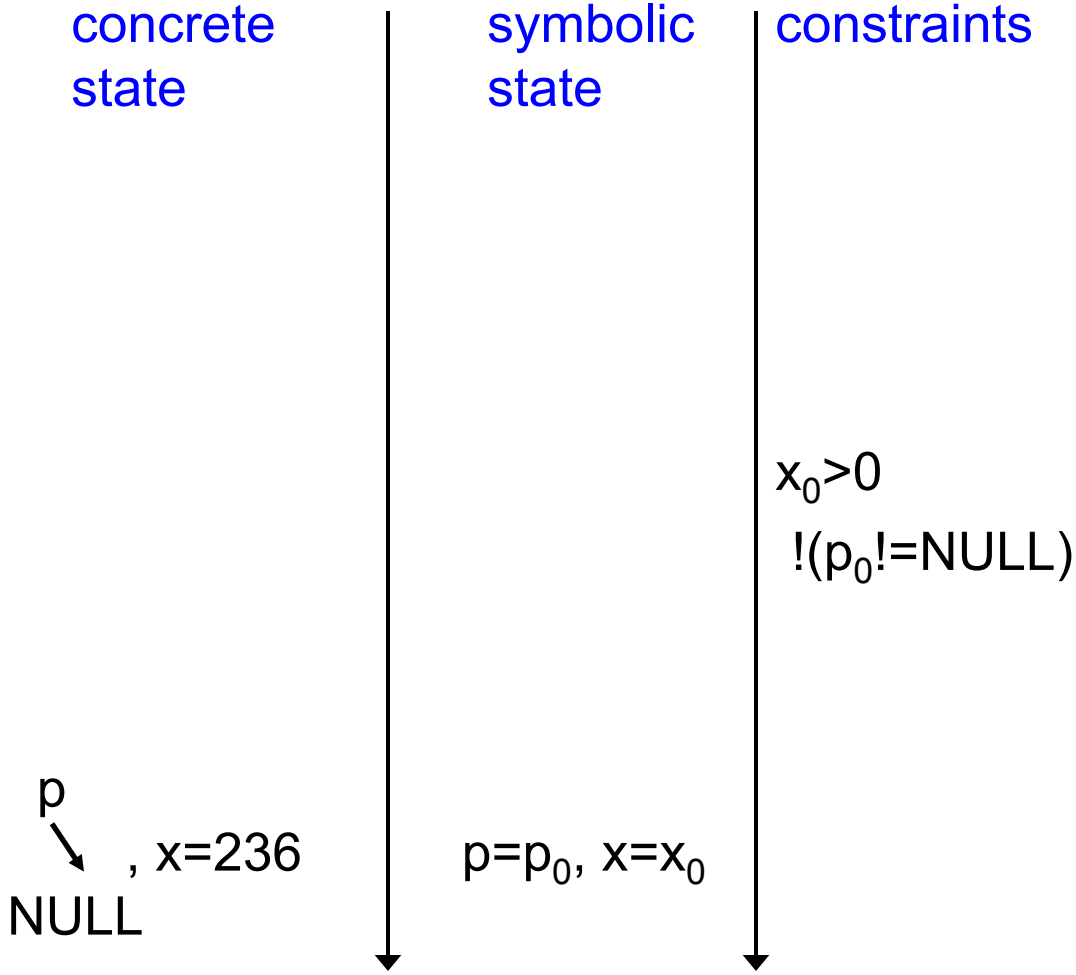

typedef struct cell {
int v;
struct cell *next;
} cell;
int f(int v) {
return 2*v + 1;
}
int testme(cell *p, int x) {
if (x > 0)
if (p != NULL)
if (f(x) == p->v)
if (p->next == p)
abort();
return 0;
}
Concrete Execution
Symbolic Execution
concrete state
symbolic state
constraints
p
, x=236
NULL
p=p0, x=x0

```
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)
          abort();
  return 0;
}
```

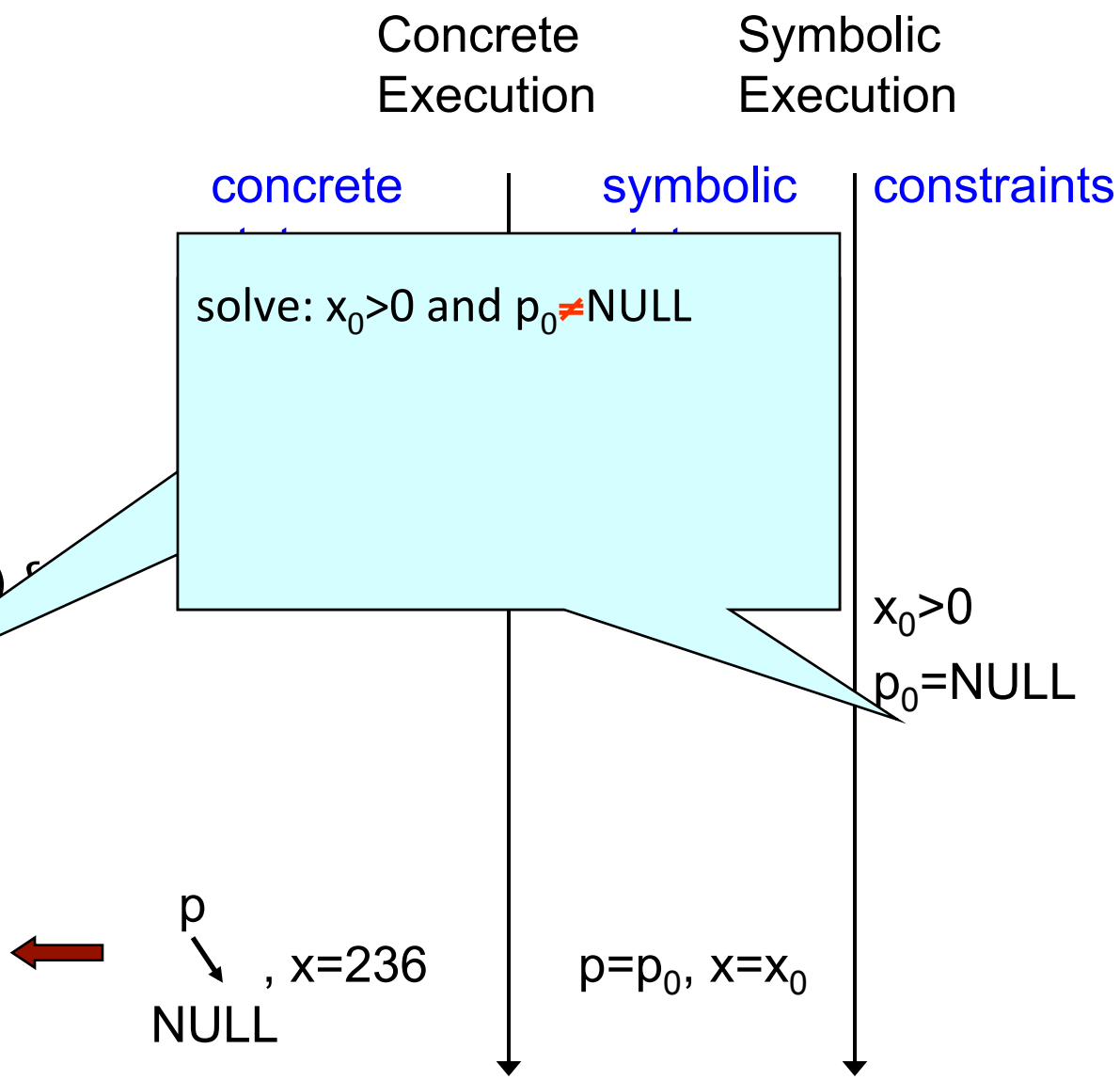


```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          abort();
  return 0;
}
```

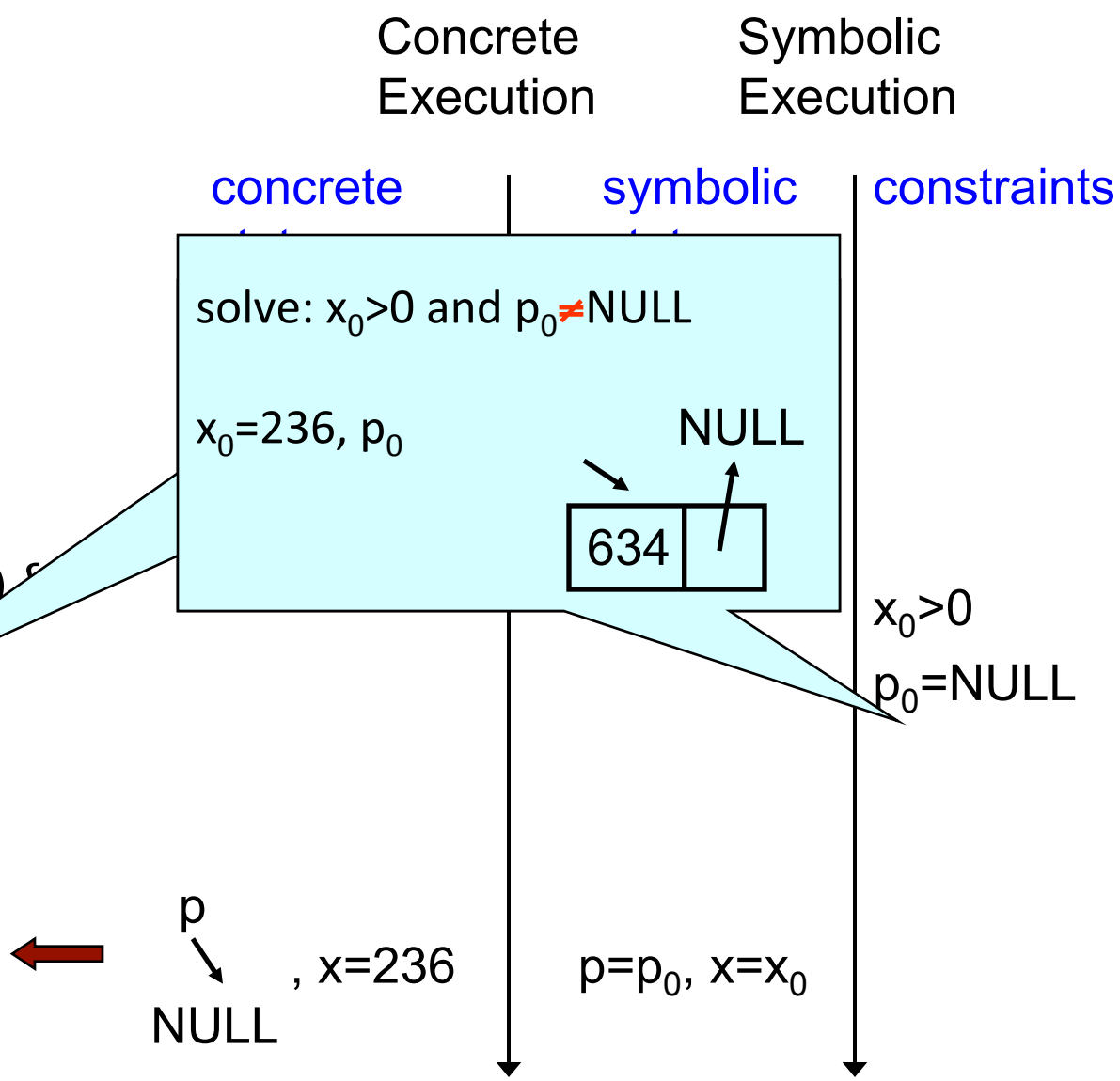




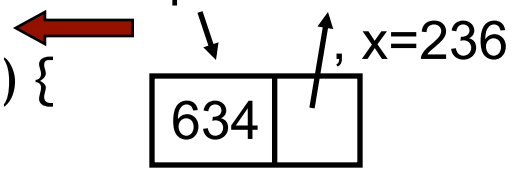
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          abort();
  return 0;
}
```



```
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)
          abort();
  return 0;
}
```



```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

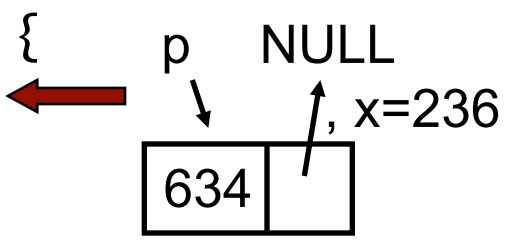
concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

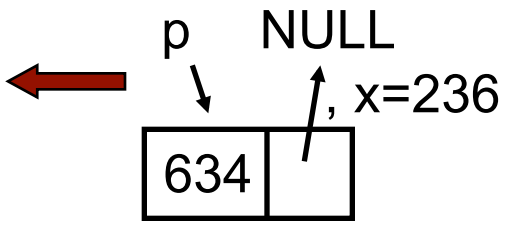
symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

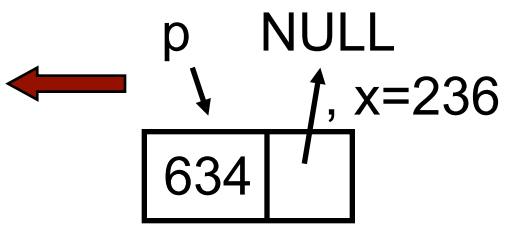
constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

$p_0 \neq NULL$

```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

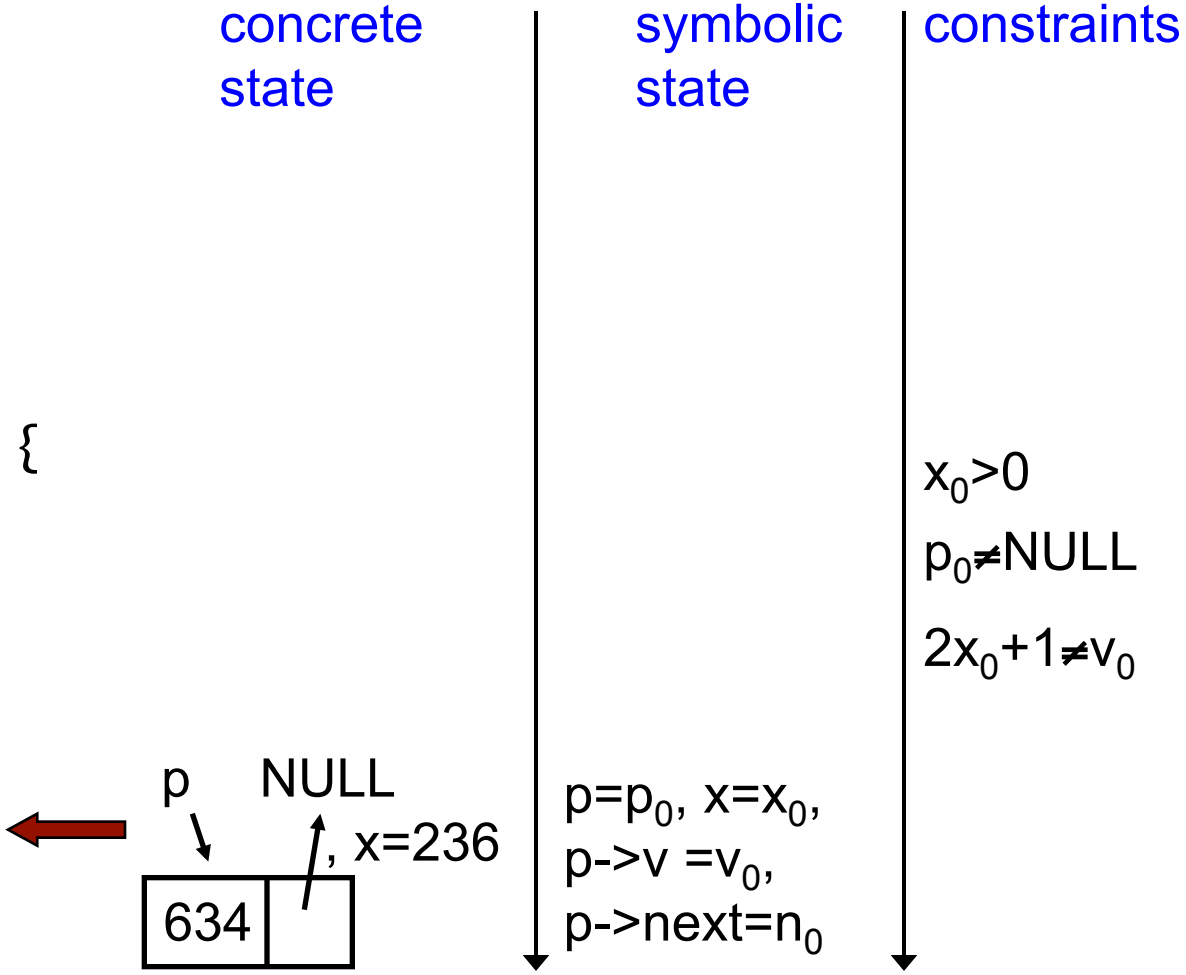
$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

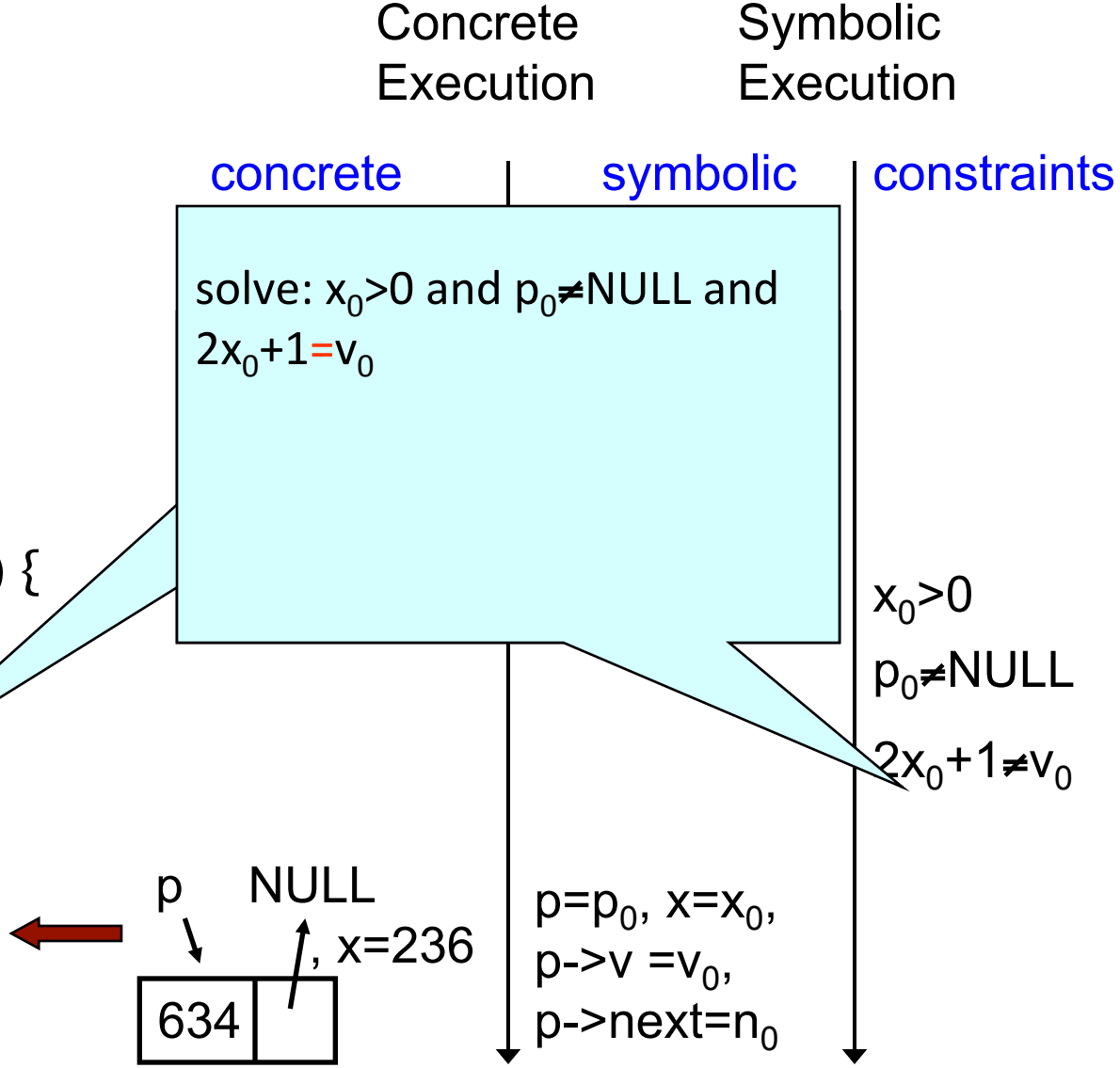
$p_0 \neq NULL$

$2x_0+1 \neq v_0$

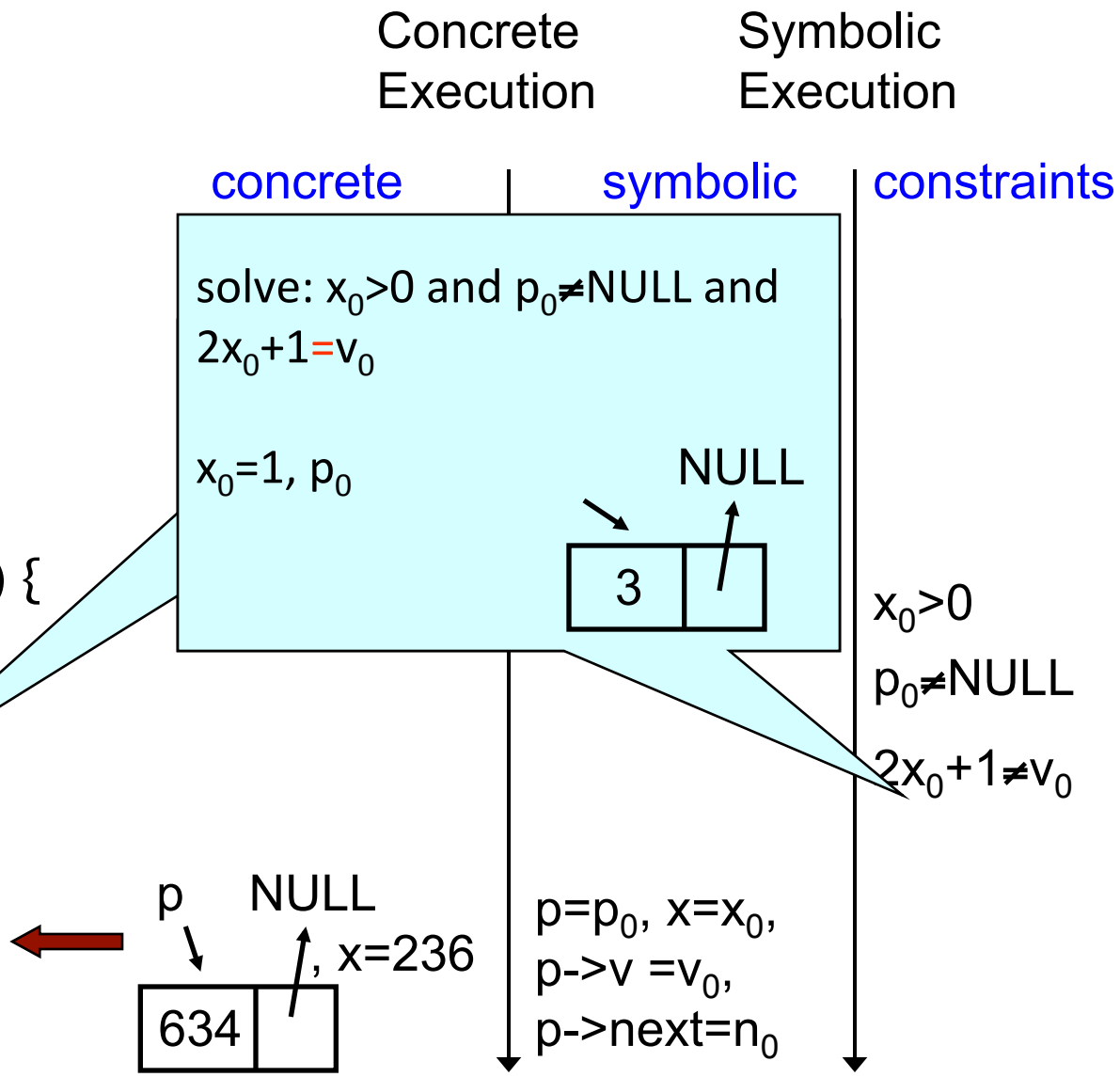
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          abort();
  return 0;
}
```



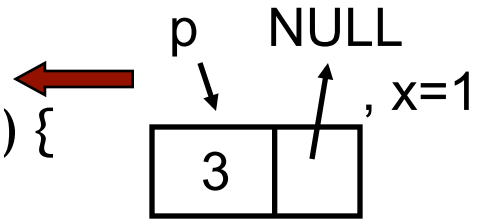
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          abort();
  return 0;
}
```



```
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)
          abort();
  return 0;
}
```



```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

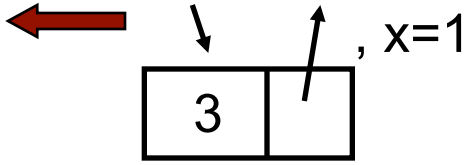
concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

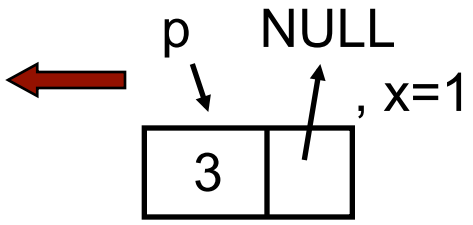
symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

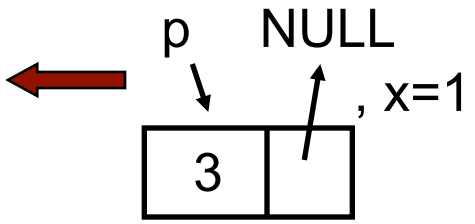
constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

$p_0 \neq NULL$

```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

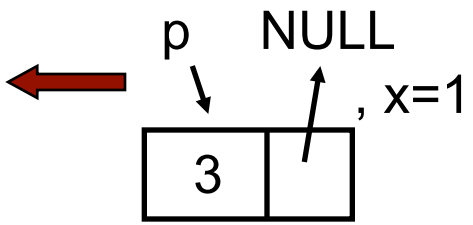
$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

```
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)
          abort();
  return 0;
}
```

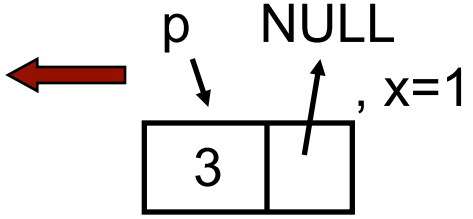
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

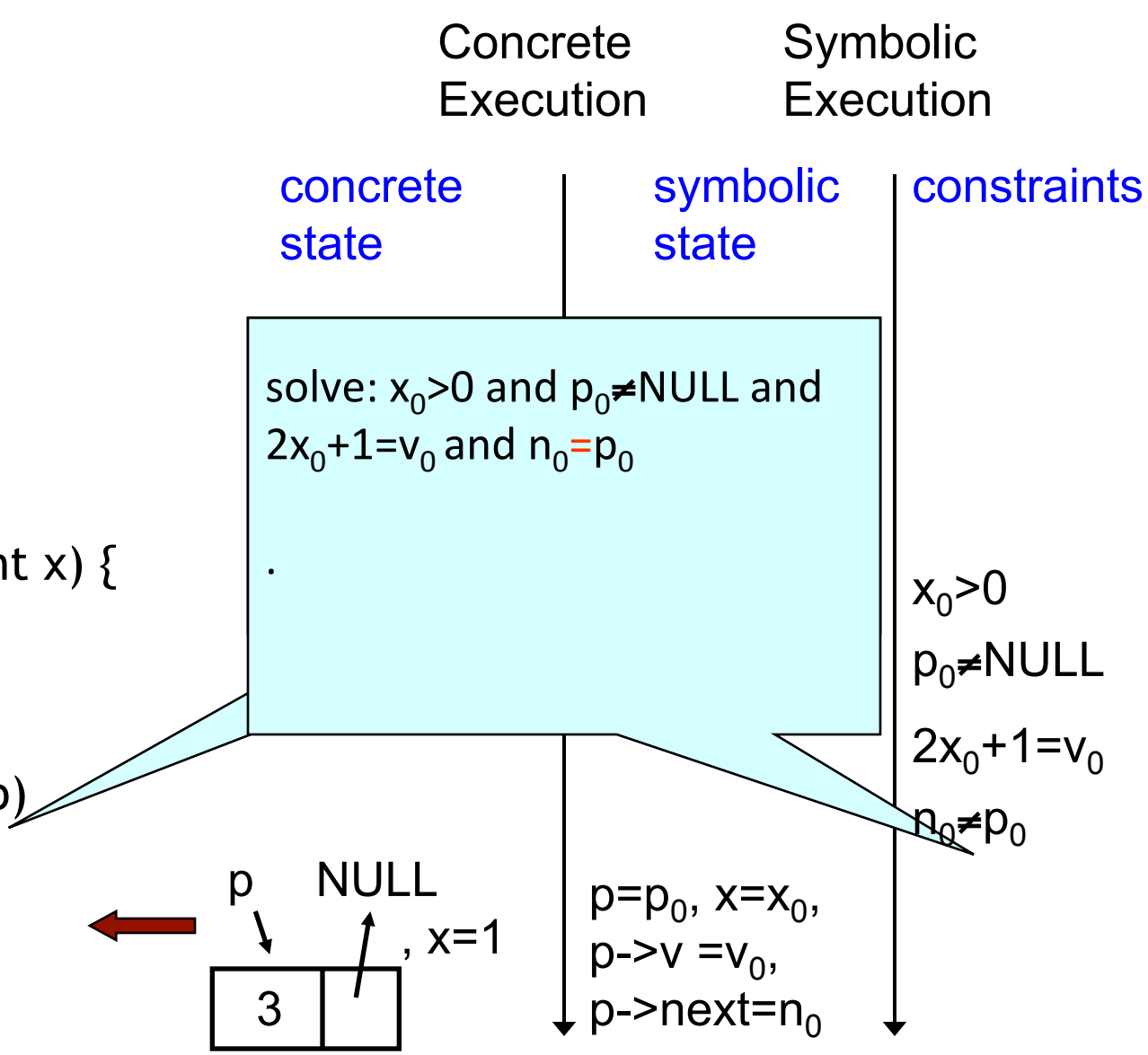
$x_0>0$

$p_0 \neq NULL$

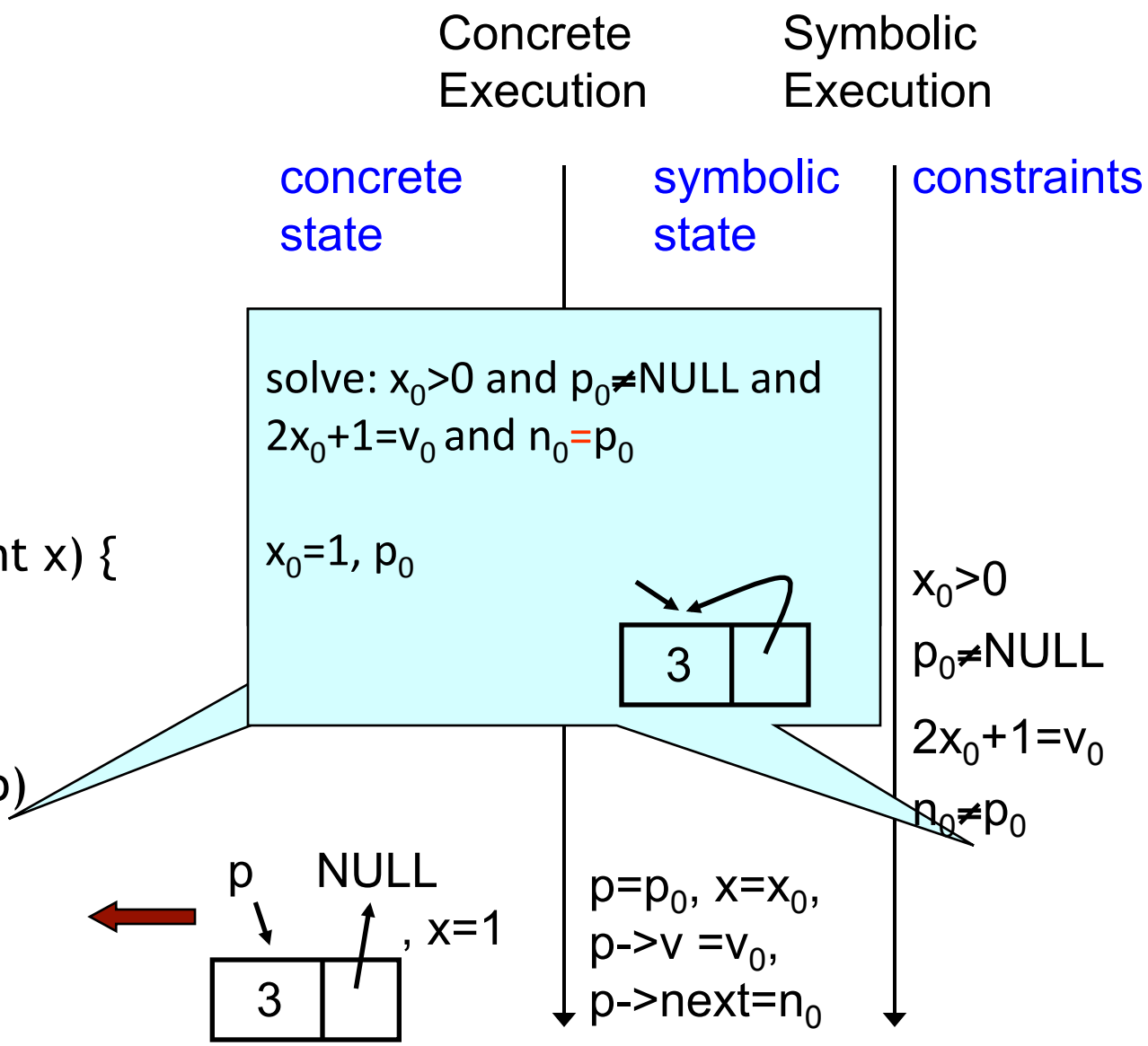
$2x_0+1=v_0$

$n_0 \neq p_0$

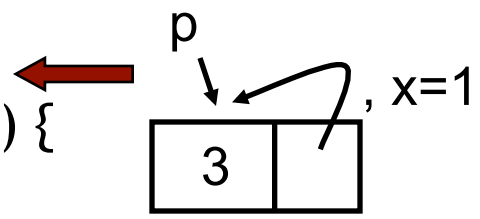
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          abort();
  return 0;
}
```



```
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)
          abort();
  return 0;
}
```



```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

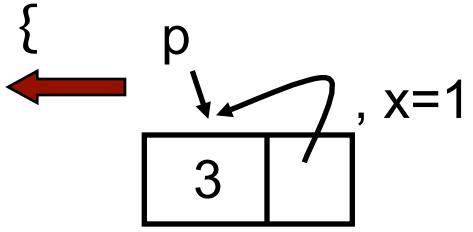
concrete state

symbolic state

constraints

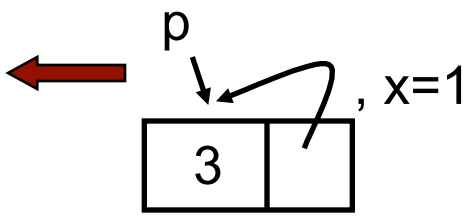
$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

```
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)
          abort();
  return 0;
}
```



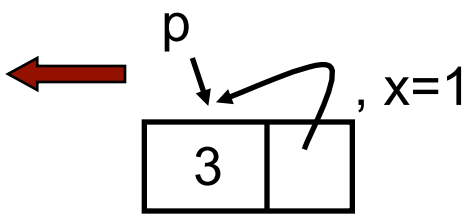
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$

```
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)
          abort();
  return 0;
}
```



Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
	$p=p_0$, $x=x_0$, $p->v=v_0$, $p->next=n_0$	$x_0>0$ $p_0 \neq NULL$

```
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)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

```
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)
          abort();
  return 0;
}
```