软件分析

动态符号执行和抽象解释

熊英飞
北京大学
2014

提醒

- 下周上课前为课程项目组队截止日期

复习：符号执行

- 请列举符号执行不能得到精确结果的场景

- 有循环的情况
- 分支太多，在有限时间内探索不完的情况
- 约束求解失败的情况

约束求解失败的情况

- 形成了复杂条件
  - $x^5 + 3x^3 == y$
  - p->next->value == x
- 调用了系统调用
  - If (file.read()==x)

- 动态符号执行
  - 混合程序的真实执行和符号执行
  - 在约束求解无法进行的时候，用真实值代替符号值
    - 如果真实值x=10，则$x^5 + 3x^3 == y$变为103000==y，可满足

动态符号执行

- 动态符号执行主要用于生成测试输入
- 代表性工作：
  - Concolic Testing, Koushik Sen
    - 主要工具：CUTE
  - Execution-Generated Testing, Cristian Cadar
    - 主要工具：KLEE

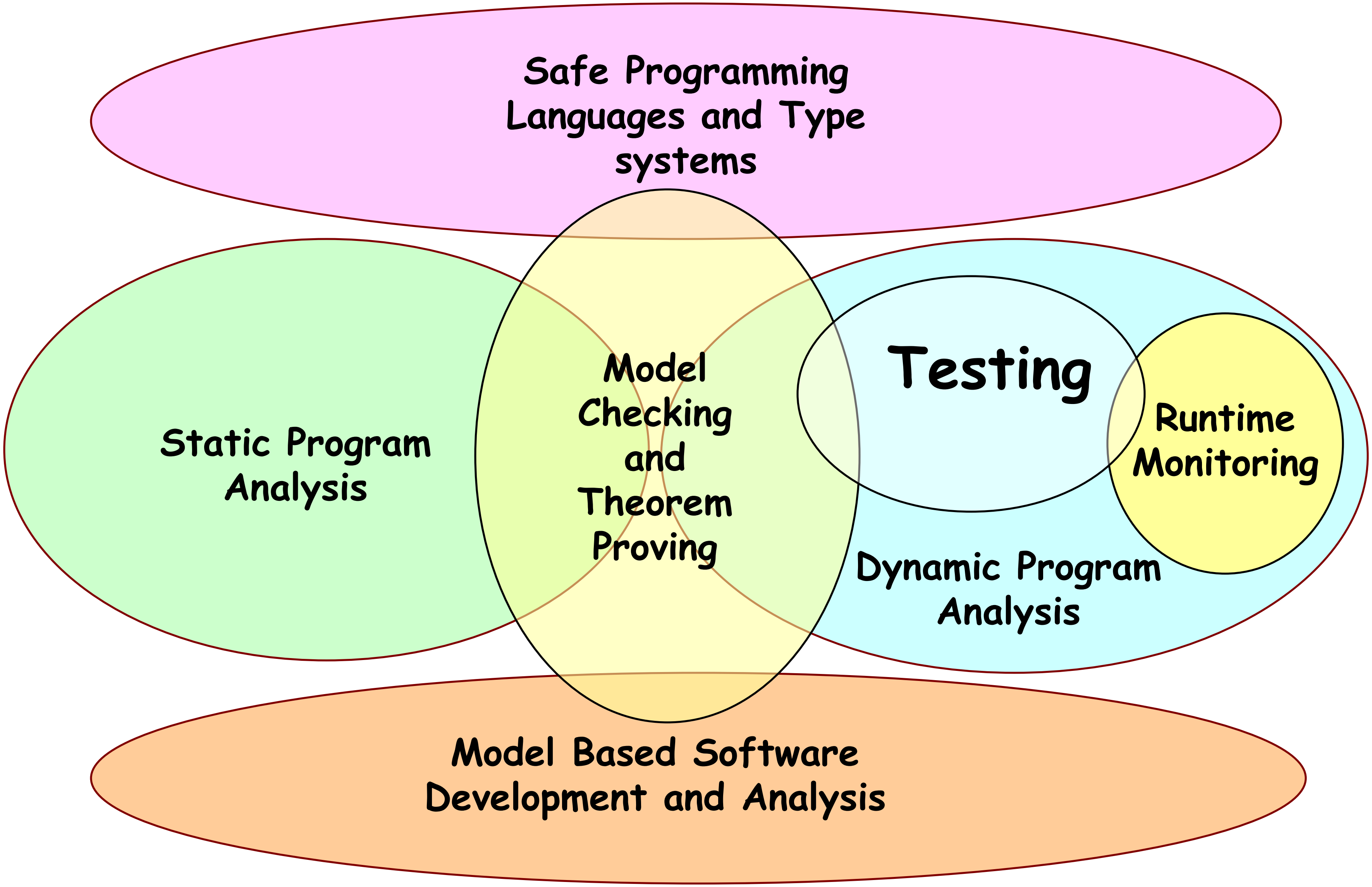
DART and CUTE: Concolic Testing

Koushik Sen
University of California, Berkeley

Joint work with Gul Agha, Patrice Godefroid, Nils Klarlund,

Rupak Majumdar, Darko Marinov

Big Picture



Safe Programming Languages and Type systems

Static Program Analysis

Model Checking and Theorem Proving

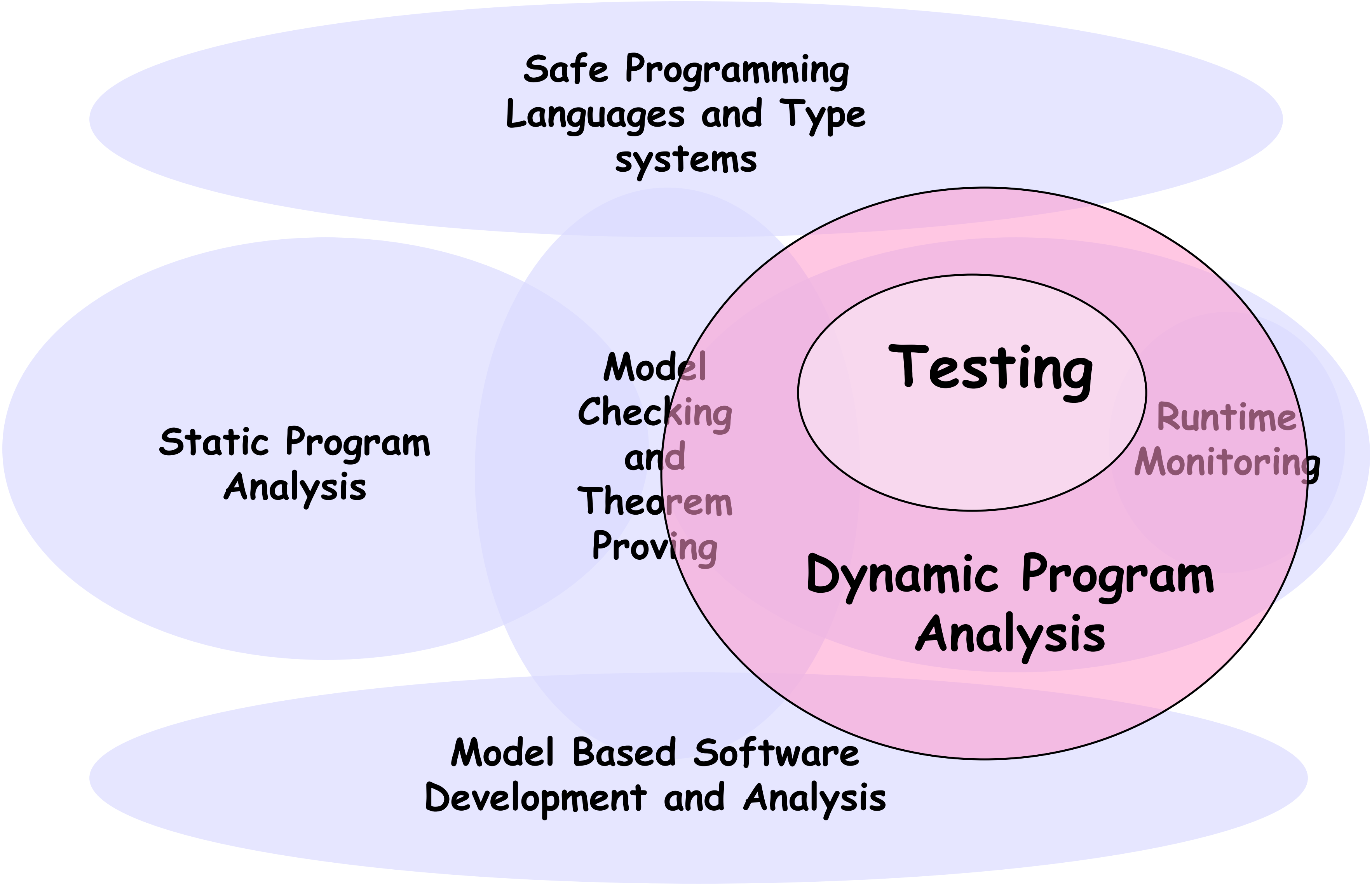
Testing

Runtime Monitoring

Dynamic Program Analysis

Model Based Software Development and Analysis

Big Picture



Safe Programming Languages and Type systems

Static Program Analysis

Model Checking and Theorem Proving

Testing

Runtime Monitoring

Dynamic Program Analysis

Model Based Software Development and Analysis

A Familiar Program: QuickSort

```
void quicksort (int[] a, int lo, int hi) {
    int i=lo, j=hi, h;
    int x=a[(lo+hi)/2];

    // partition
    do {
          while (a[i]<x) i++;
          while (a[j]>x) j--; if (i<=j) {
                     h=a[i];
                     a[i]=a[j];
                     a[j]=h;
                     i++;
                     j--;
          }
    } while (i<=j);

    // recursion
    if (lo<j) quicksort(a, lo, j);
    if (i<hi) quicksort(a, i, hi);
}
```

A Familiar Program: QuickSort

```
void quicksort (int[] a, int lo, int hi) {
    int i=lo, j=hi, h;
    int x=a[(lo+hi)/2];

    // partition
    do {
            while (a[i]<x) i++;
            while (a[j]>x) j--; if (i<=j) {
                        h=a[i];
                        a[i]=a[j];
                        a[j]=h;
                        i++;
                        j--;
            }
    } while (i<=j);

    // recursion
    if (lo<j) quicksort(a, lo, j);
    if (i<hi) quicksort(a, i, hi);
}
```

- Test QuickSort
    - Create an array
    - Initialize the elements of the array
    - Execute the program on this array
- How much confidence do I have in this testing method?
- Is my test suite *Complete*?
- Can someone generate a small and *Complete* test suite for me?

Automated Test Generation

- Studied since 70’s
  - King 76, Myers 79
- 30 years have passed, and yet no effective solution
- What Happened???

Automated Test Generation

- Studied since 70's
  - King 76, Myers 79
- 30 years have passed, and yet no effective solution
- What Happened???
  - Program-analysis techniques were expensive
  - Automated theorem proving and constraint solving techniques were not efficient

Automated Test Generation

- Studied since 70's
  - King 76, Myers 79
- 30 years have passed, and yet no effective solution
- What Happened???
  - Program-analysis techniques were expensive
  - Automated theorem proving and constraint solving techniques were not efficient
- In the recent years we have seen remarkable progress in static program-analysis and constraint solving
  - SLAM, BLAST, ESP, Bandera, Saturn, MAGIC

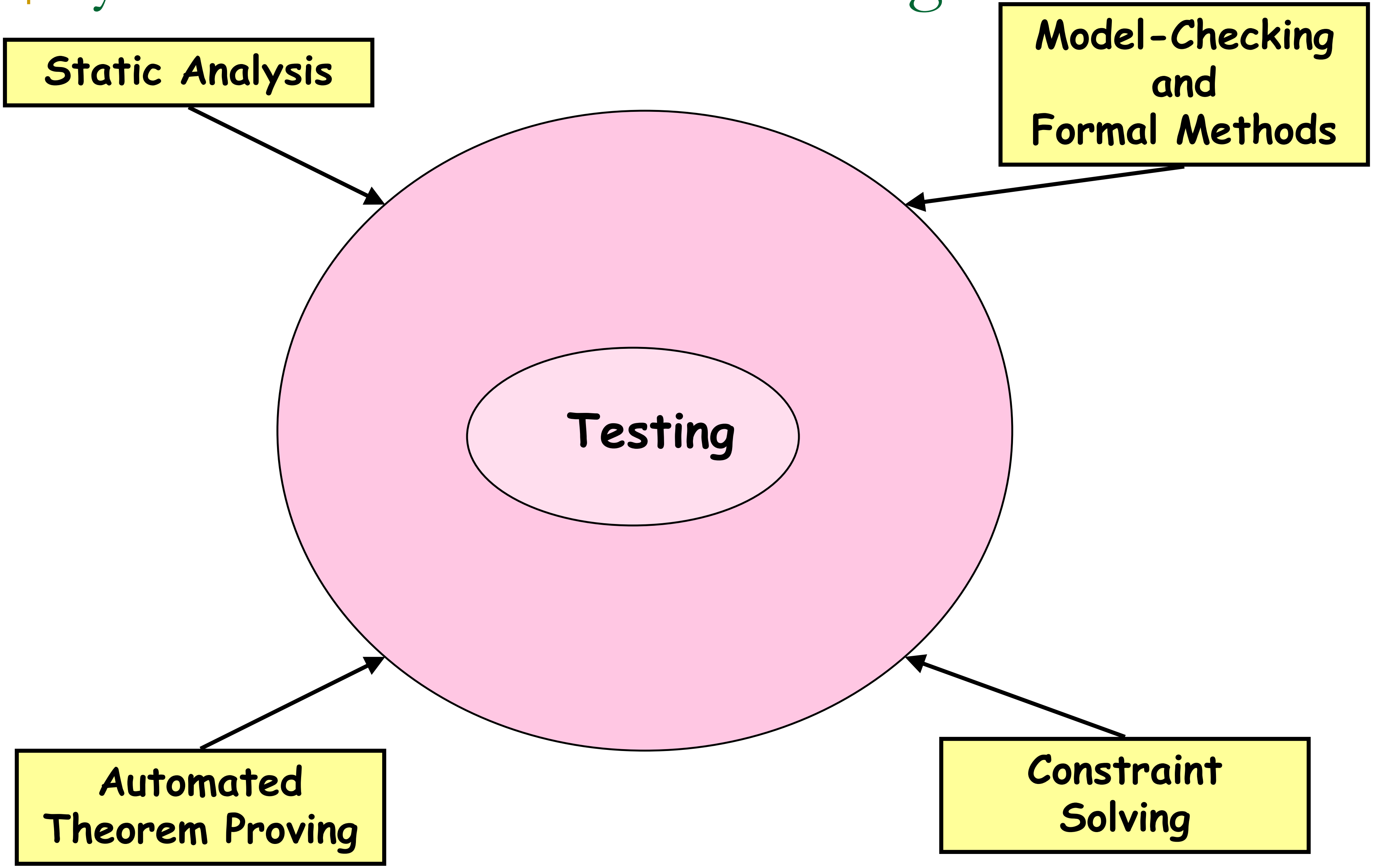
Automated Test Generation

- Studied since 70's
  - King 76, Myers 79

Question: Can we use similar techniques in Automated Testing?

- What Happened???
  - Program-analysis techniques were expensive
  - Automated theorem proving and constraint solving techniques were not efficient
- In the recent years we have seen remarkable progress in static program-analysis and constraint solving
  - SLAM, BLAST, ESP, Bandera, Saturn, MAGIC

Systematic Automated Testing



Static Analysis

Model-Checking
and
Formal Methods

Testing

Automated
Theorem Proving

Constraint
Solving

CUTE and DART

- Combine random testing (concrete execution) and symbolic testing (symbolic execution)

  [PLDI’05, FSE’05, FASE’06, CAV’06,ISSTA’07, ICSE’07]

Concrete + Symb**olic** = Concolic

Goal

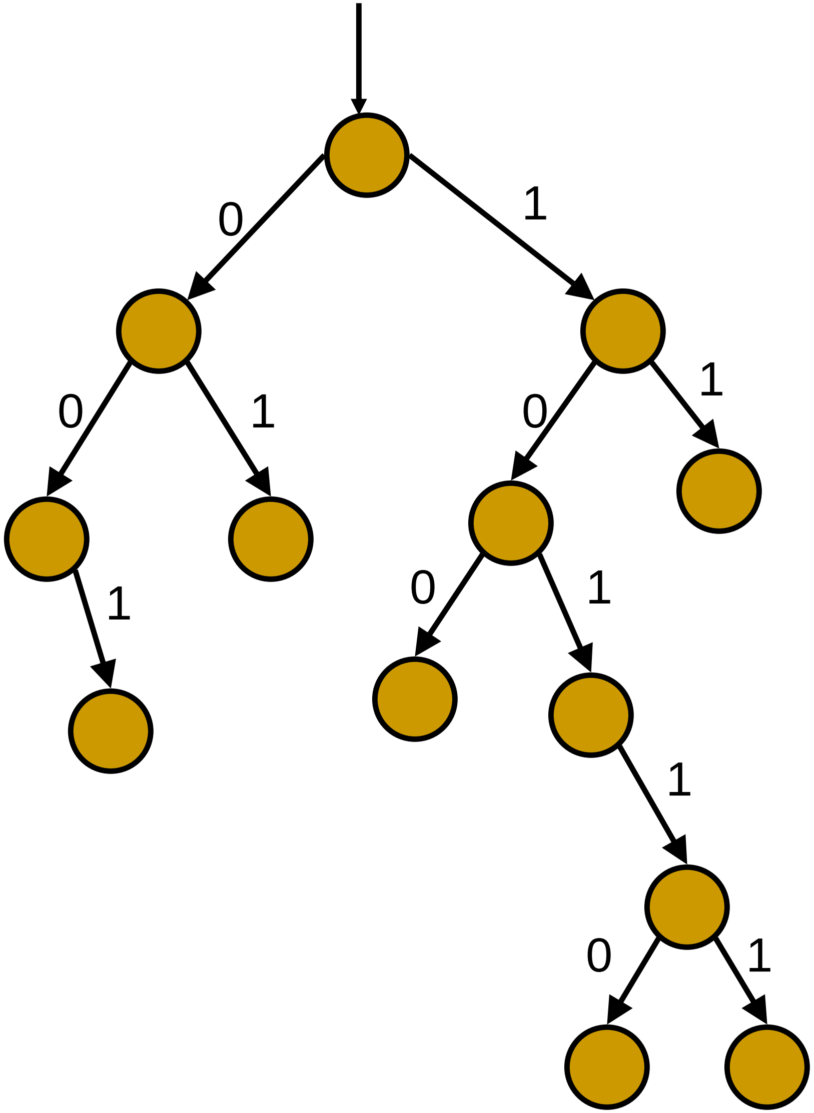
- Automated Unit Testing of real-world C and Java Programs
  - Generate test inputs
  - Execute unit under test on generated test inputs
    - so that all reachable statements are executed
  - Any assertion violation gets caught

Goal

- Automated Unit Testing of real-world C and Java Programs
  - Generate test inputs
  - Execute unit under test on generated test inputs
    - so that all reachable statements are executed
  - Any assertion violation gets caught
- Our Approach:
  - Explore all execution paths of an Unit for all possible inputs
    - Exploring all execution paths ensure that all reachable statements are executed

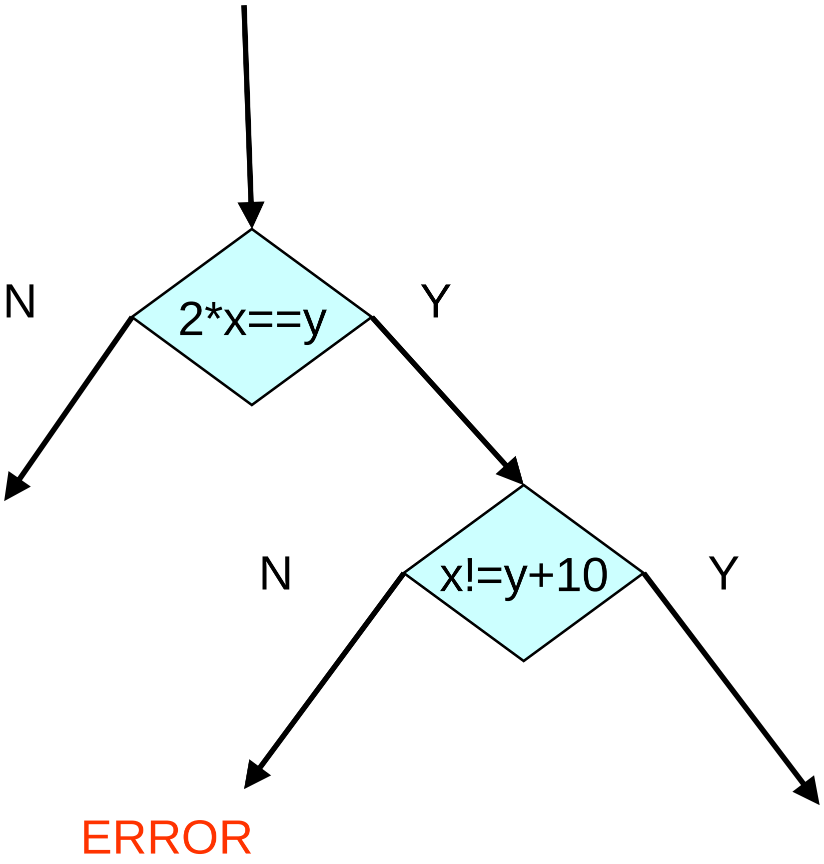
Execution Paths of a Program

- 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;
     }
  }
}
```



Concolic Testing: Finding Security and Safety Bugs

Divide by 0 Error

Buffer Overflow

`x = 3 / i;`

`a[i] = 4;`

Concolic Testing: Finding Security and Safety Bugs

Key: Add Checks Automatically and Perform Concolic Testing

Divide by 0 Error

```
if (i !=0)
    x = 3 / i;
else
    ERROR;
```

Buffer Overflow

```
if (0<=i  &&  i < a.length)
    a[i] = 4;
else
    ERROR;
```

Existing Approach I

- Random testing
  - generate random inputs
  - execute the program on generated inputs
- Probability of reaching an error can be astronomically less

```
test_me(int x){
    if(x==94389){
        ERROR;
    }
}
```

Probability of hitting ERROR = $1/2^{32}$

Existing Approach II

- Symbolic Execution
  - use symbolic values for input variables
  - execute the program symbolically on symbolic input values
  - collect symbolic path constraints
  - use theorem prover to check if a branch can be taken
- Does not scale for large programs

```
test_me(int x){
    if((x%10)*4!=17){
        ERROR;
    } else {
        ERROR;
    }
}
```

Symbolic execution will say both branches are reachable:

False positive

Existing Approach II

- Symbolic Execution
  - use symbolic values for input variables
  - execute the program symbolically on symbolic input values
  - collect symbolic path constraints
  - use theorem prover to check if a branch can be taken
- Does not scale for large programs

```
test_me(int x){
    if(bbox(x)!=17){
        ERROR;
    } else {
        ERROR;
    }
}
```

Symbolic execution will say both branches are reachable:

False positive

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;
          }
     }

}
```

- Random Test Driver:
  - random value 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	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22, y = 7	$x = x_0$, $y = y_0$	

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;
        }
    }

}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22, y = 7, z = 14	$x = x_0$, $y = y_0$, $z = 2*y_0$	

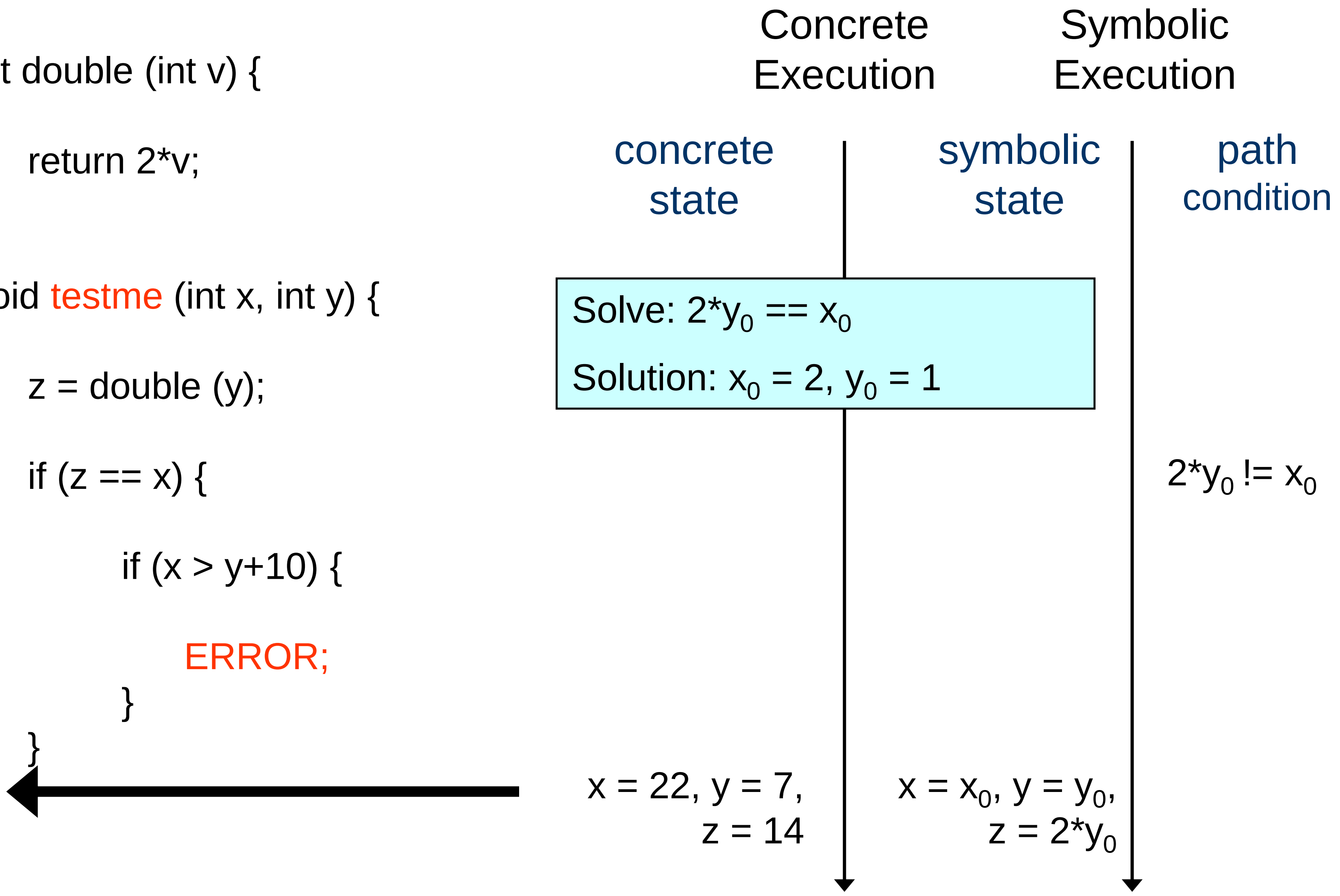
Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

Concrete Execution		Symbolic Execution
concrete state	symbolic state	path condition
		$2*y_0 \mathrel{!}= x_0$
x = 22, y = 7, z = 14	$x = x_0$, $y = y_0$, $z = 2*y_0$	

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;
        }
    }

}
```



Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
		$2*y_0$!= x_0
x = 22, y = 7, z = 14	x = x_0, y = y_0, z = $2*y_0$	

Solve: $2*y_0 == x_0$

Solution: $x_0 = 2$, $y_0 = 1$

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 = 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;
      }
   }

}
```

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

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;
        }
    }
}
```

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

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;
        }
    }
}
```

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

$2*y_0 == x_0$

x_0 ▪ y_0+10

x = 2, y = 1,
z = 2

$x = x_0$, $y = y_0$,
$z = 2*y_0$

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;
        }
    }
}
```

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

Solve: $(2*y_0 == x_0)$ Æ $(x_0 > y_0 + 10)$

Solution: $x_0 = 30$, $y_0 = 15$

$2*y_0 == x_0$

x_0 ▪ y_0+10

$x = 2, y = 1,$
$z = 2$

$x = x_0, y = y_0,$
$z = 2*y_0$

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;
        }
    }

}
```

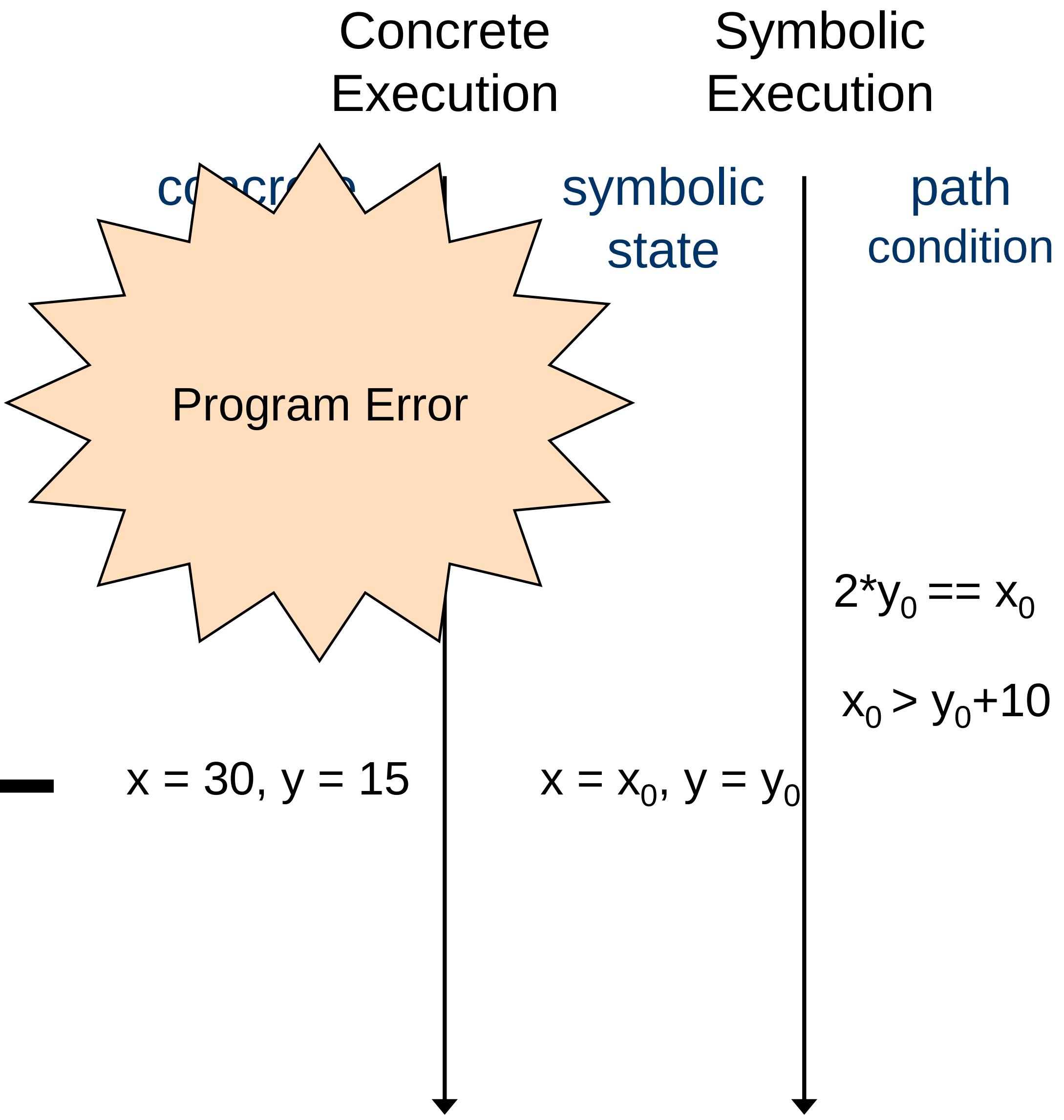
Concrete Execution

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;
        }
    }

}
```



Concrete Execution

Symbolic Execution

symbolic state

path condition

Program Error

$2*y_0 == x_0$

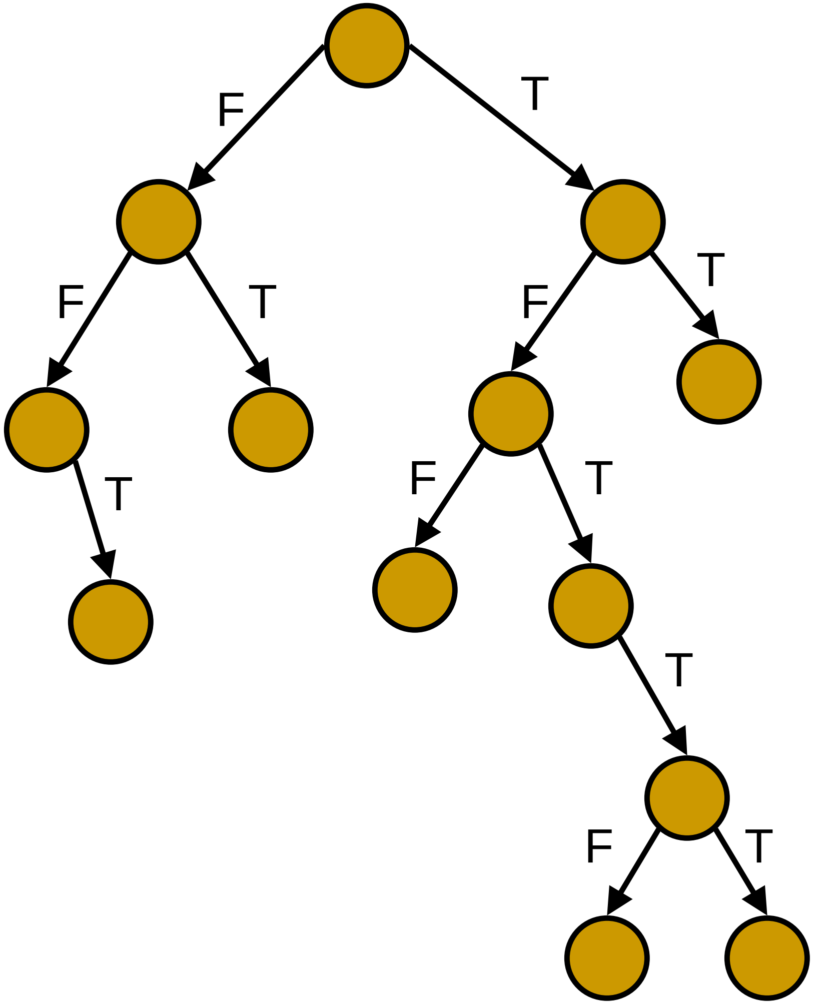
$x_0 > y_0+10$

x = 30, y = 15

$x = x_0, y = y_0$

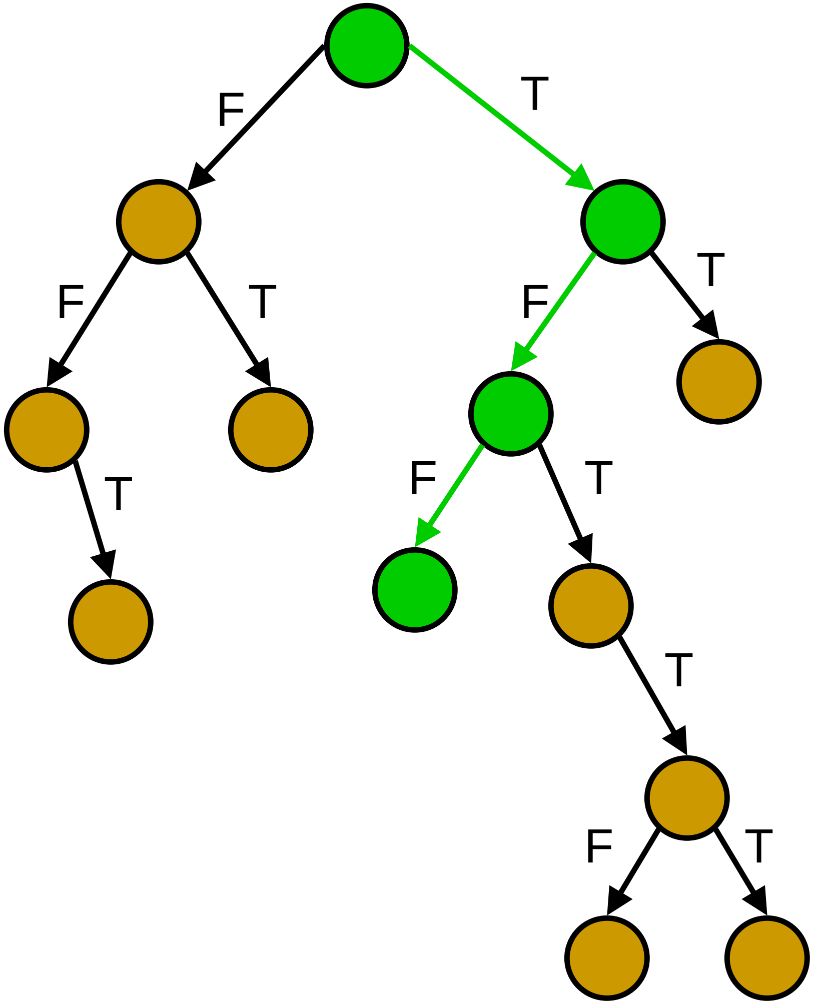
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors



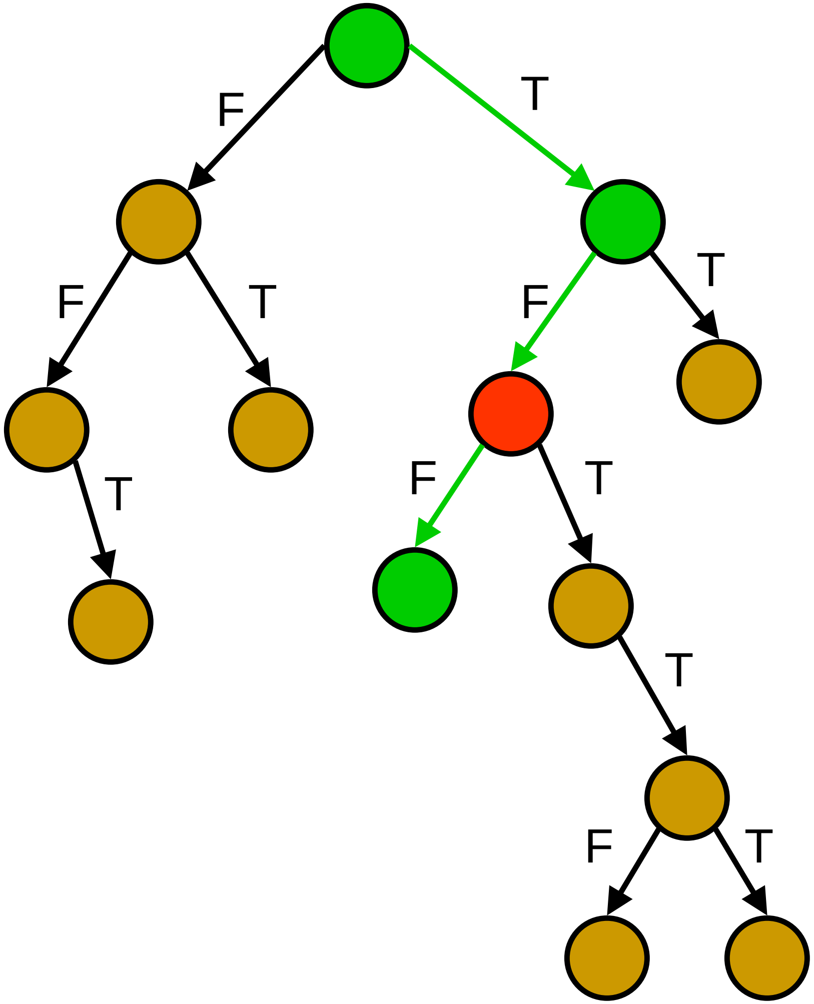
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors



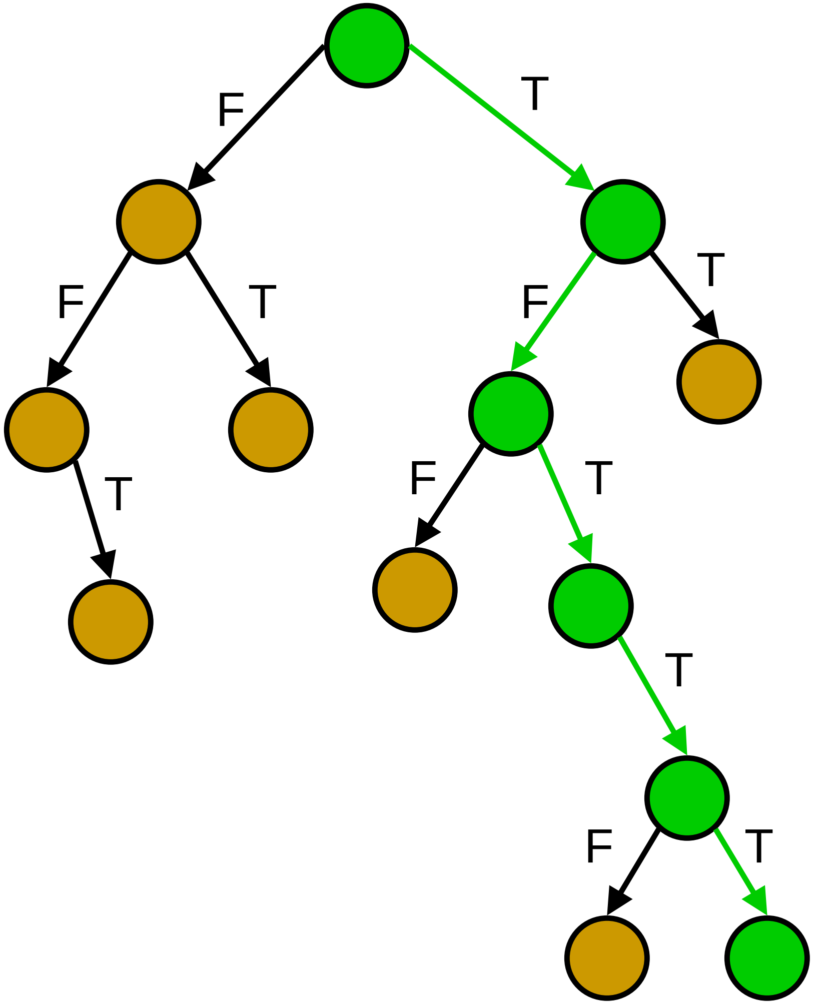
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors



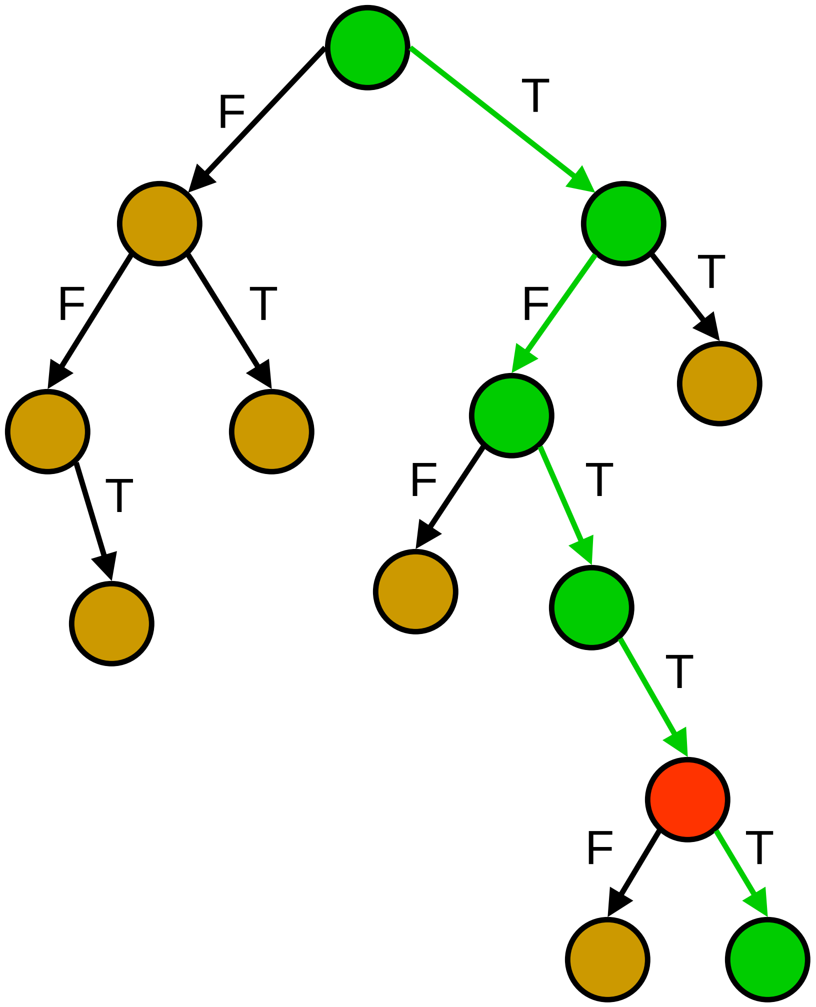
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors



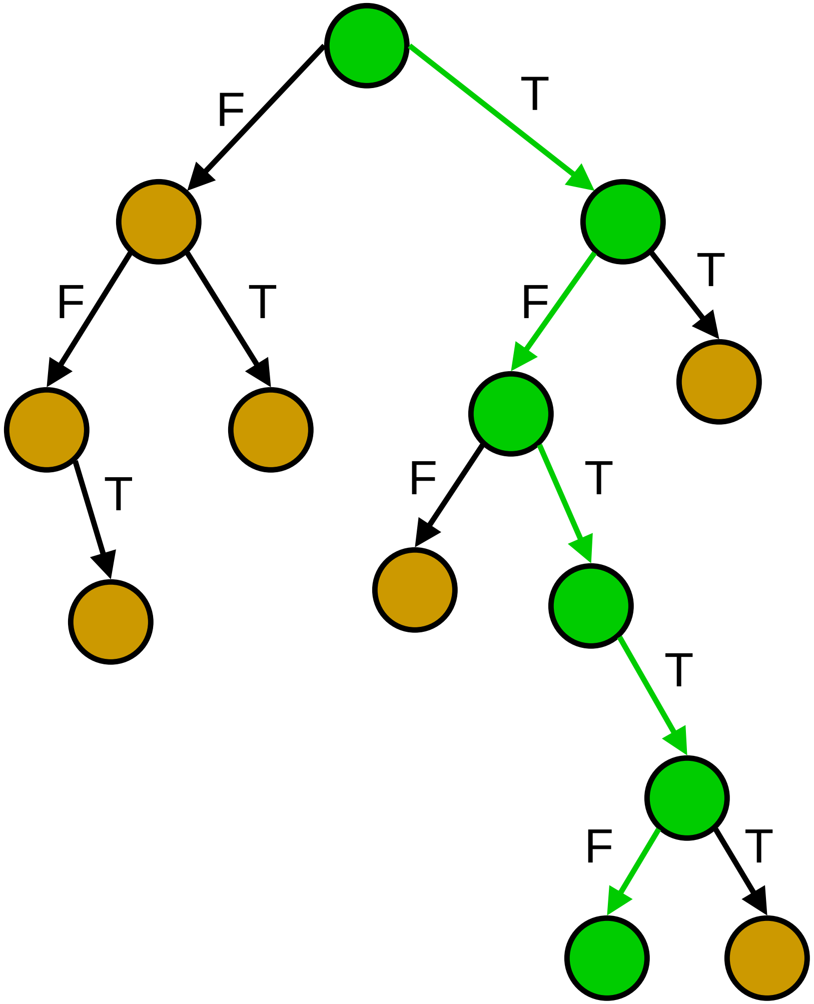
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors



Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors



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;
        }
    }
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
x = 22, y = 7, z = 49	$x = x_0$, $y = y_0$, $z = (y_0 * y_0)\%50$	$(y_0*y_0)\%50 \mathrel{!}= x_0$

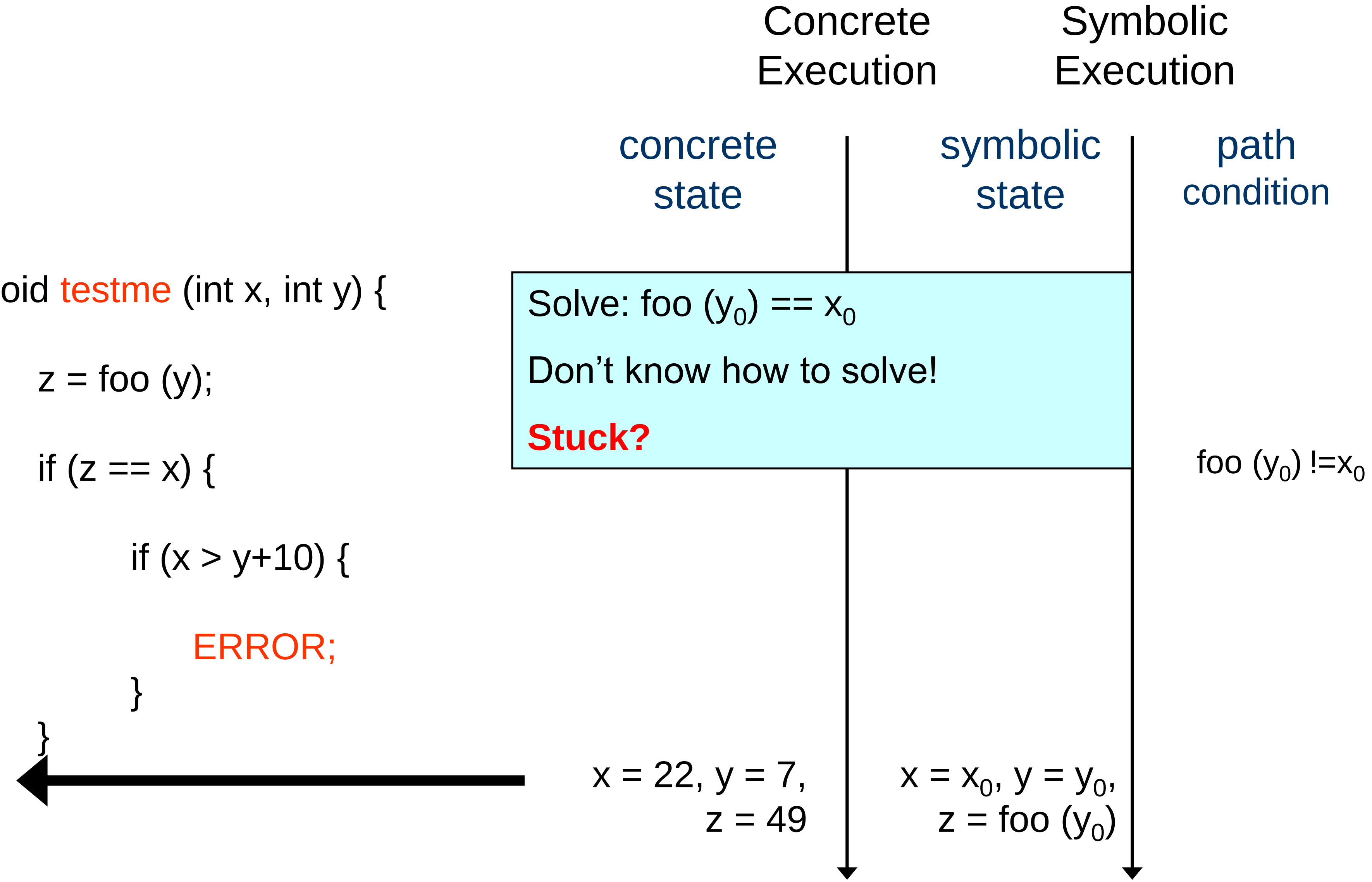
Solve: $(y_0*y_0)\%50 == x_0$

Don't know how to solve!

Stuck?

Novelty : Simultaneous Concrete and Symbolic Execution

```
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

Solve: foo (y_0) == x_0

Don't know how to solve!

Stuck?

foo (y_0) !=x_0

x = 22, y = 7,
z = 49

x = x_0, y = y_0,
z = foo (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;
        }
    }
}
```

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

Solve: $(y_0*y_0)\%50 == x_0$

Don’t know how to solve!

Not Stuck!

Use concrete state

Replace y_0 by 7 (sound)

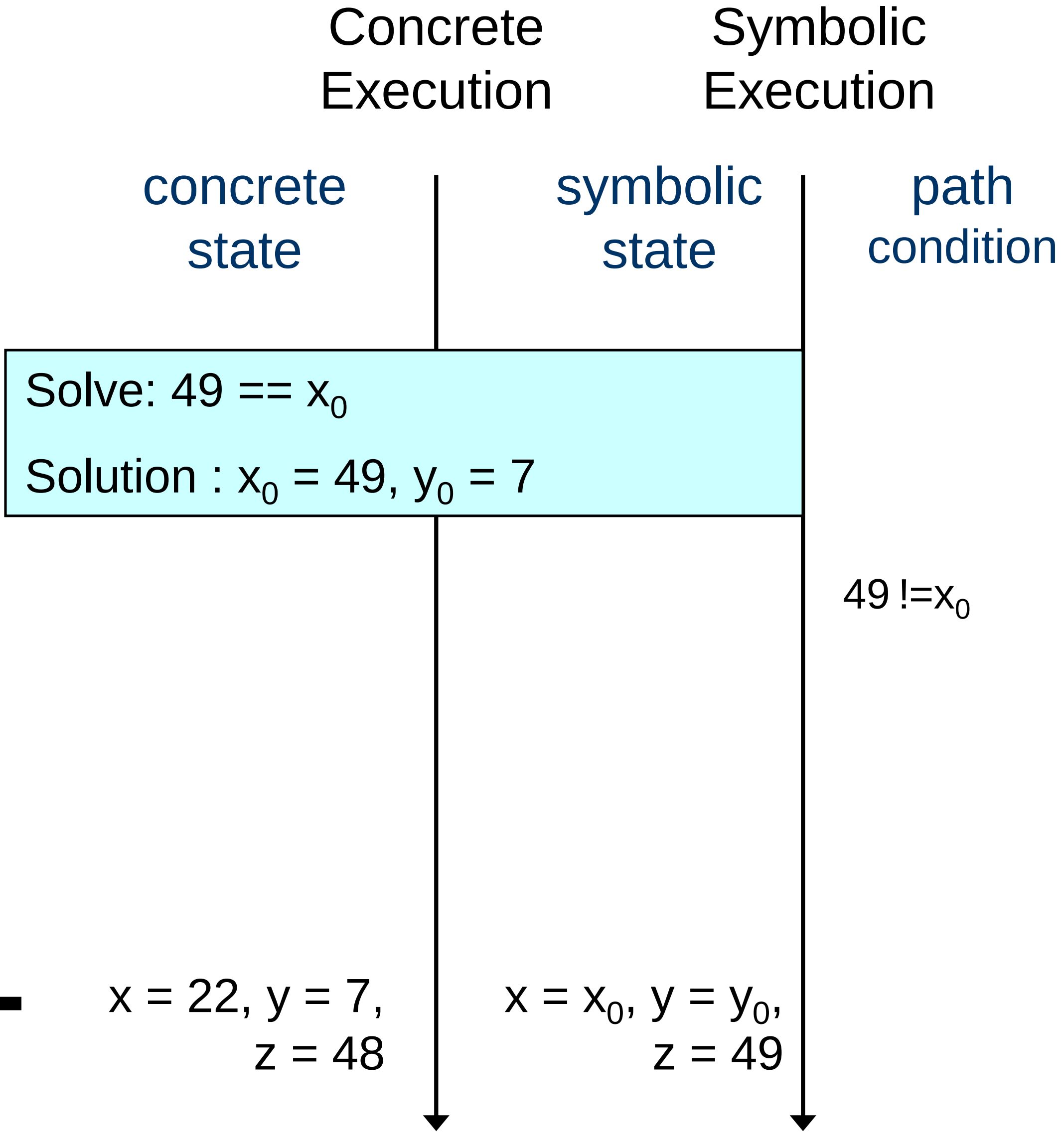
$(y_0*y_0)\%50 \mathrel{!}= x_0$

$x = 22, y = 7,$
$z = 49$

$x = x_0, y = y_0,$
$z = (y_0 * y_0)\%50$

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

Solve: $49 == x_0$

Solution : $x_0 = 49$, $y_0 = 7$

$49 \,!{=} x_0$

$x = 22$, $y = 7$, $z = 48$

$x = x_0$, $y = y_0$, $z = 49$

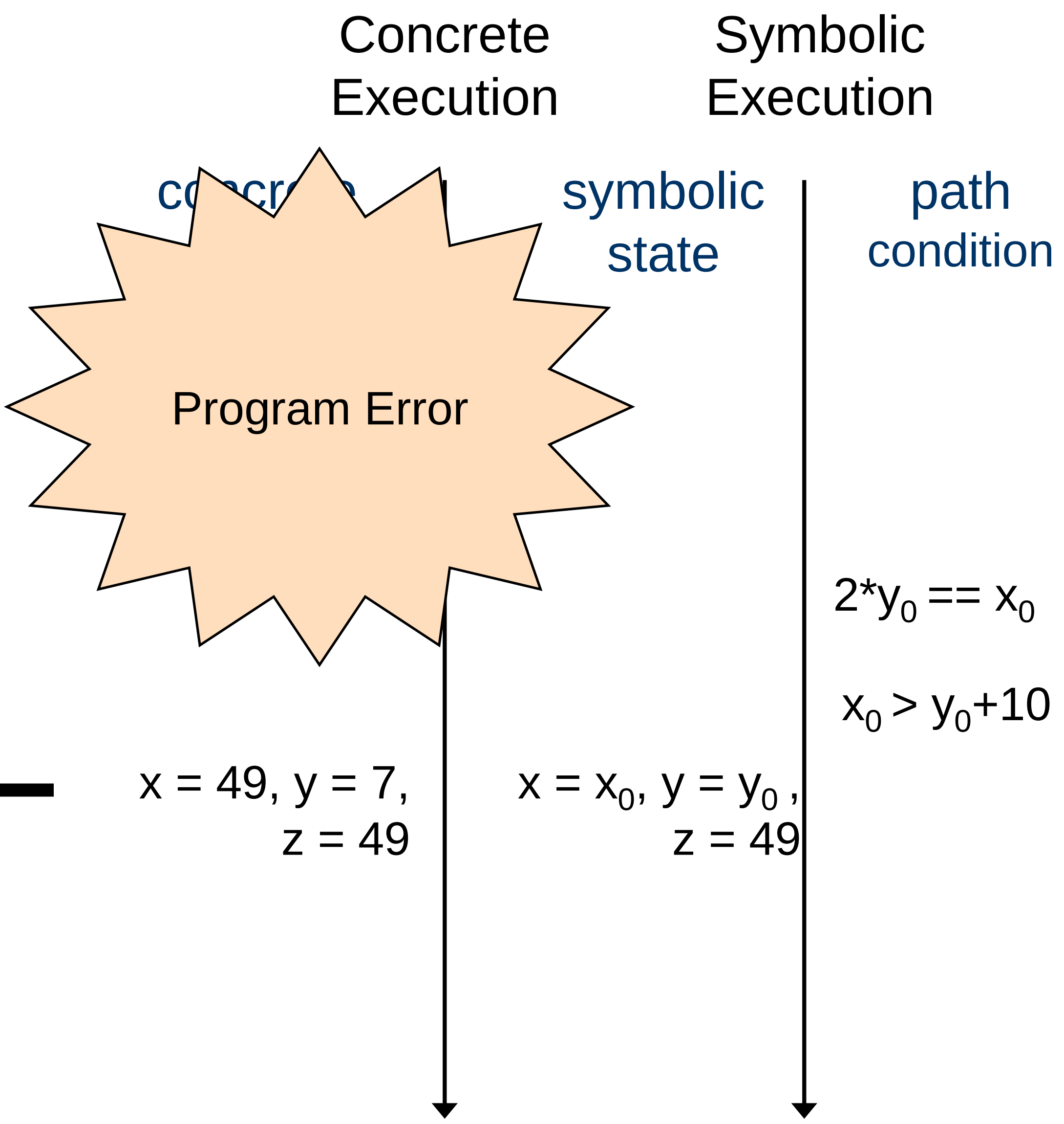
Novelty : Simultaneous Concrete and Symbolic Execution

```
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

```
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

symbolic state

path condition

Program Error

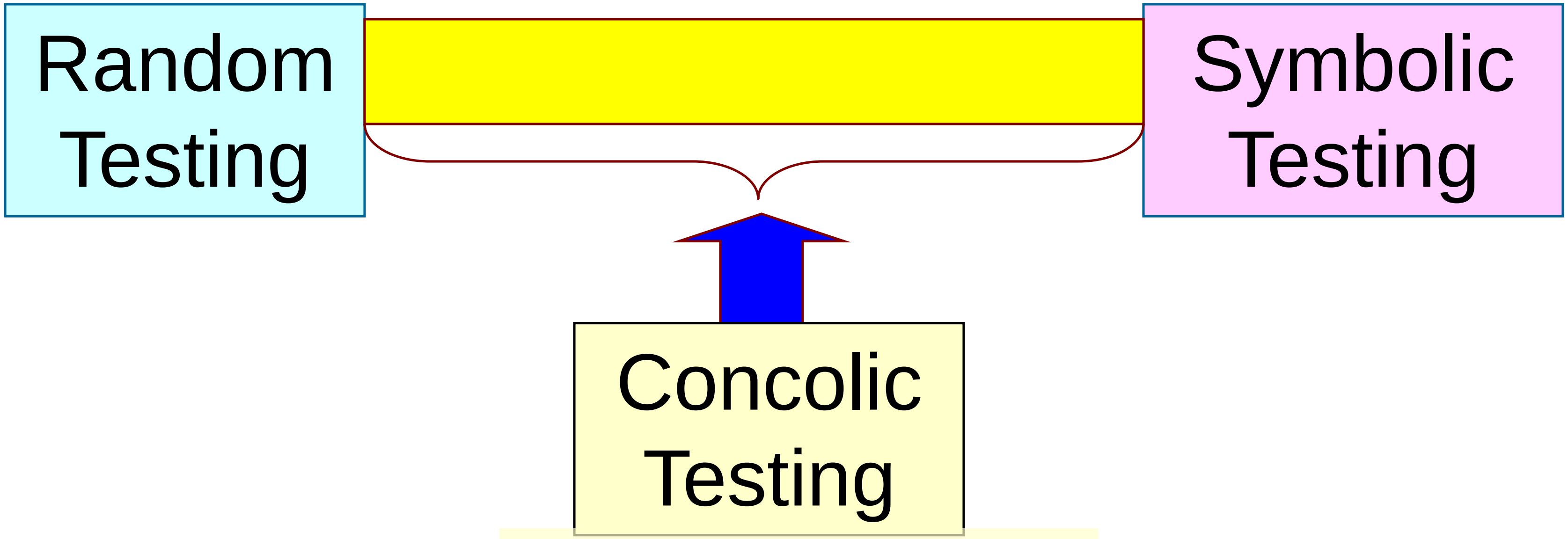
$2*y_0 == x_0$

$x_0 > y_0+10$

x = 49, y = 7, z = 49

$x = x_0, y = y_0, z = 49$

Concolic Testing: A Middle Approach



Random Testing

Symbolic Testing

Concolic Testing

- \+ 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
  - Goto http://srl.cs.berkeley.edu/~ksen/ for CUTE and jCUTE binaries
- MSR has four implementations
  - SAGE, PEX, YOGI, Vigilante
- Similar tool: EXE at Stanford
- Easiest way to use and to develop on top of CUTE
  - Implement concolic testing yourself

Testing Data Structures (joint work with Darko Marinov and Gul Agha)

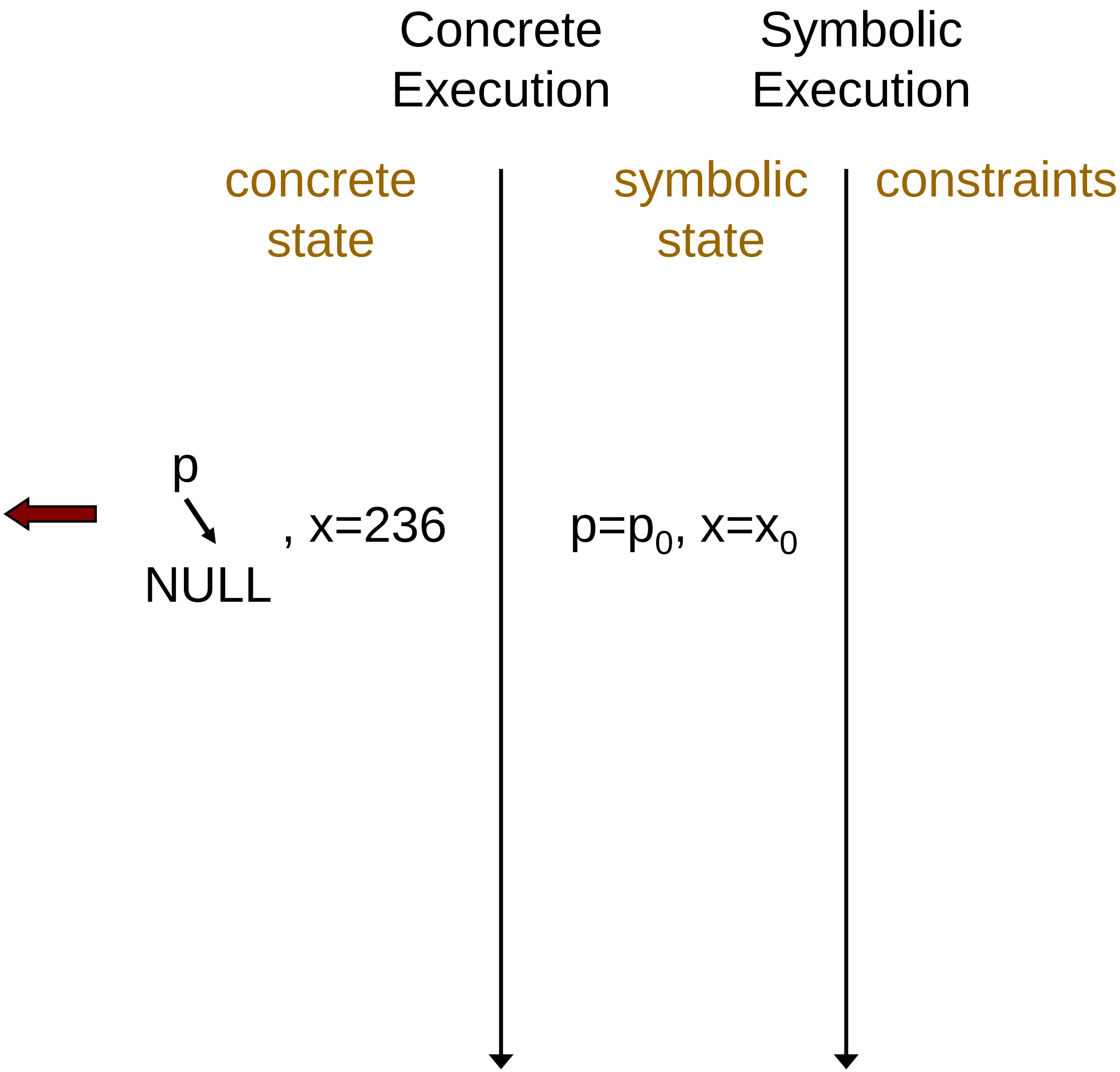
Example

```
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;
}
```

- Random Test Driver:
  - random memory graph reachable from p
  - random value for x
- Probability of reaching abort( ) is extremely low

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;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

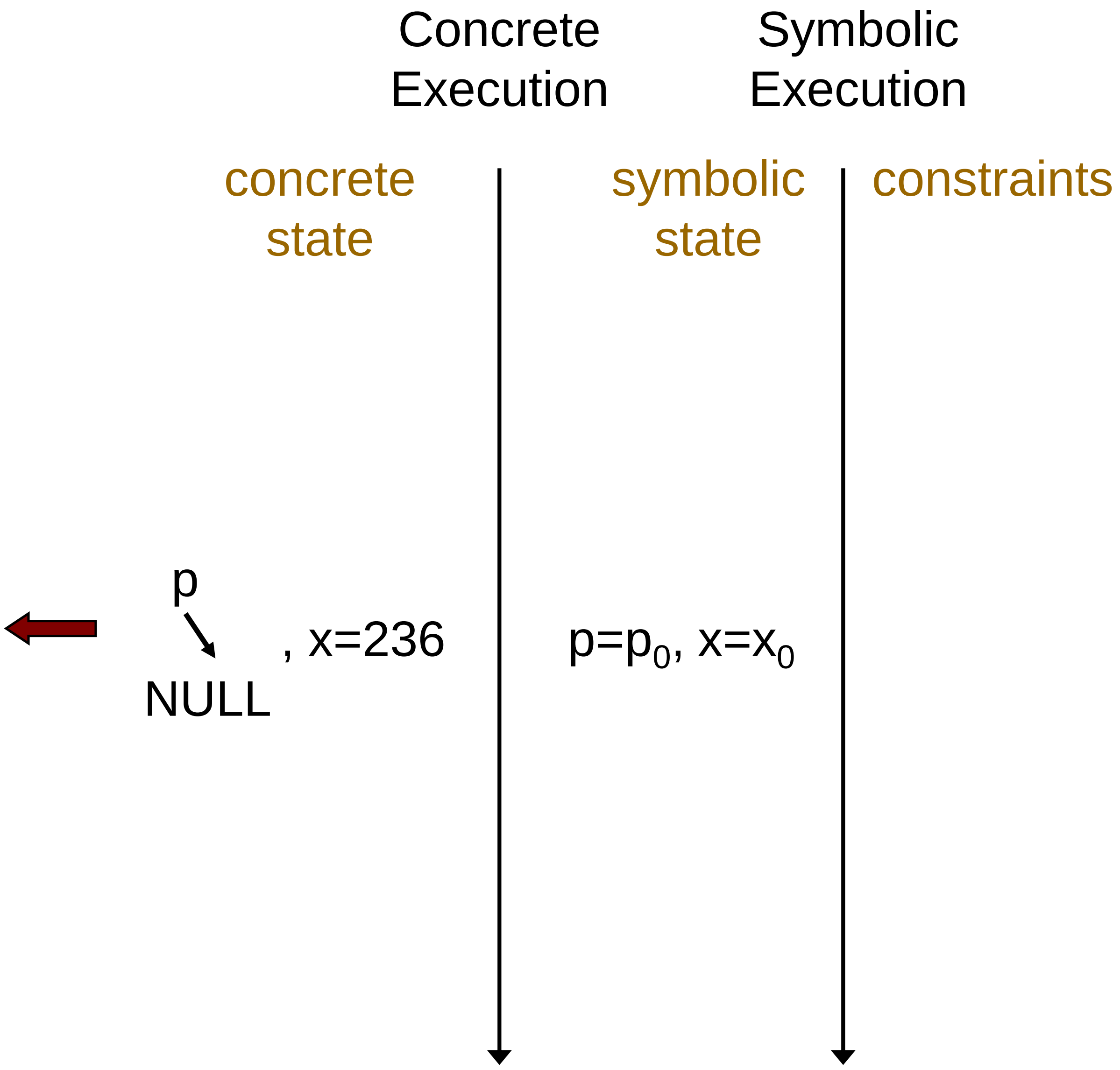
constraints

p → NULL, x=236

$p=p_0, x=x_0$

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;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

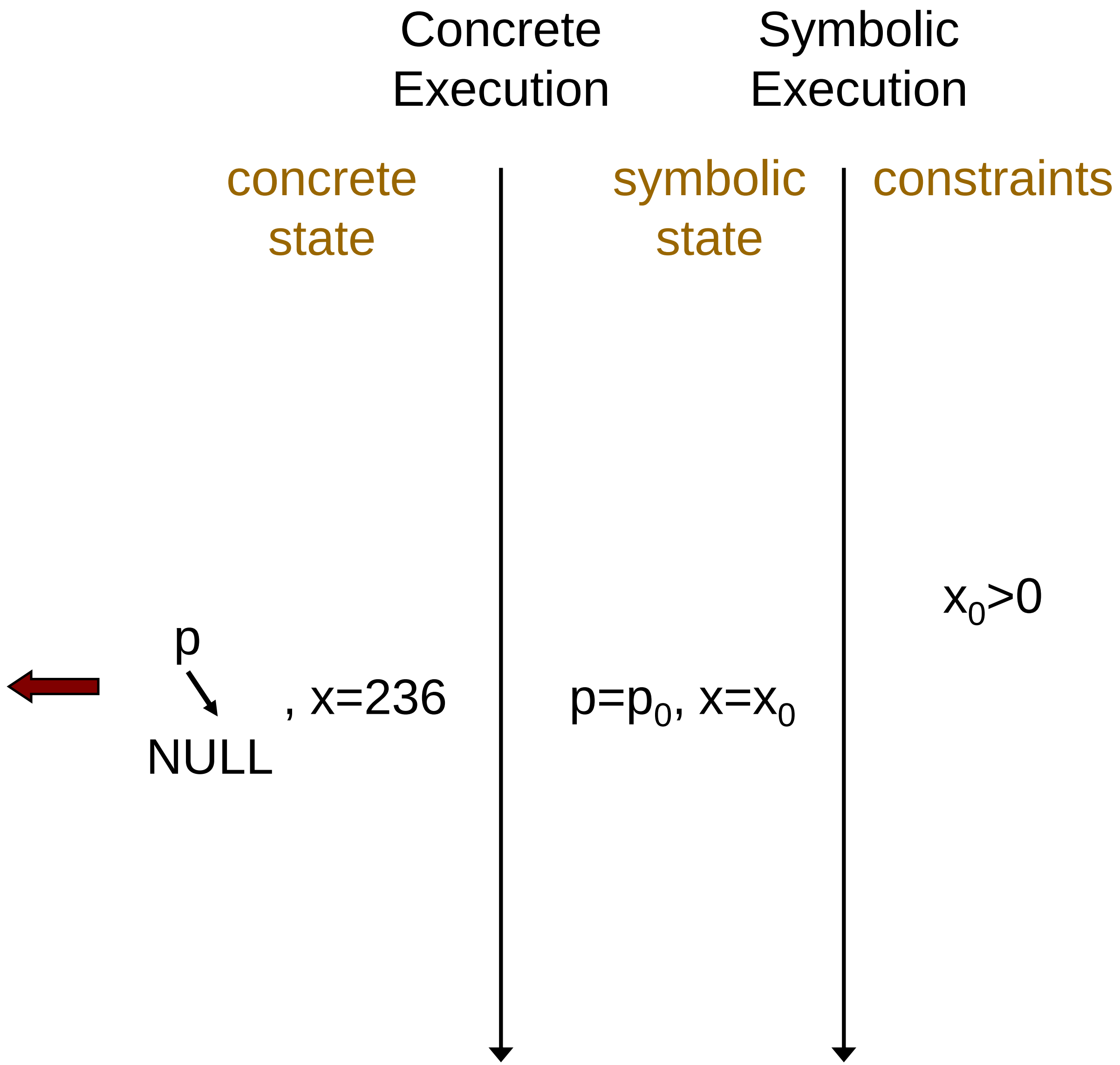
constraints

p → NULL, x=236

$p=p_0$, $x=x_0$

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;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

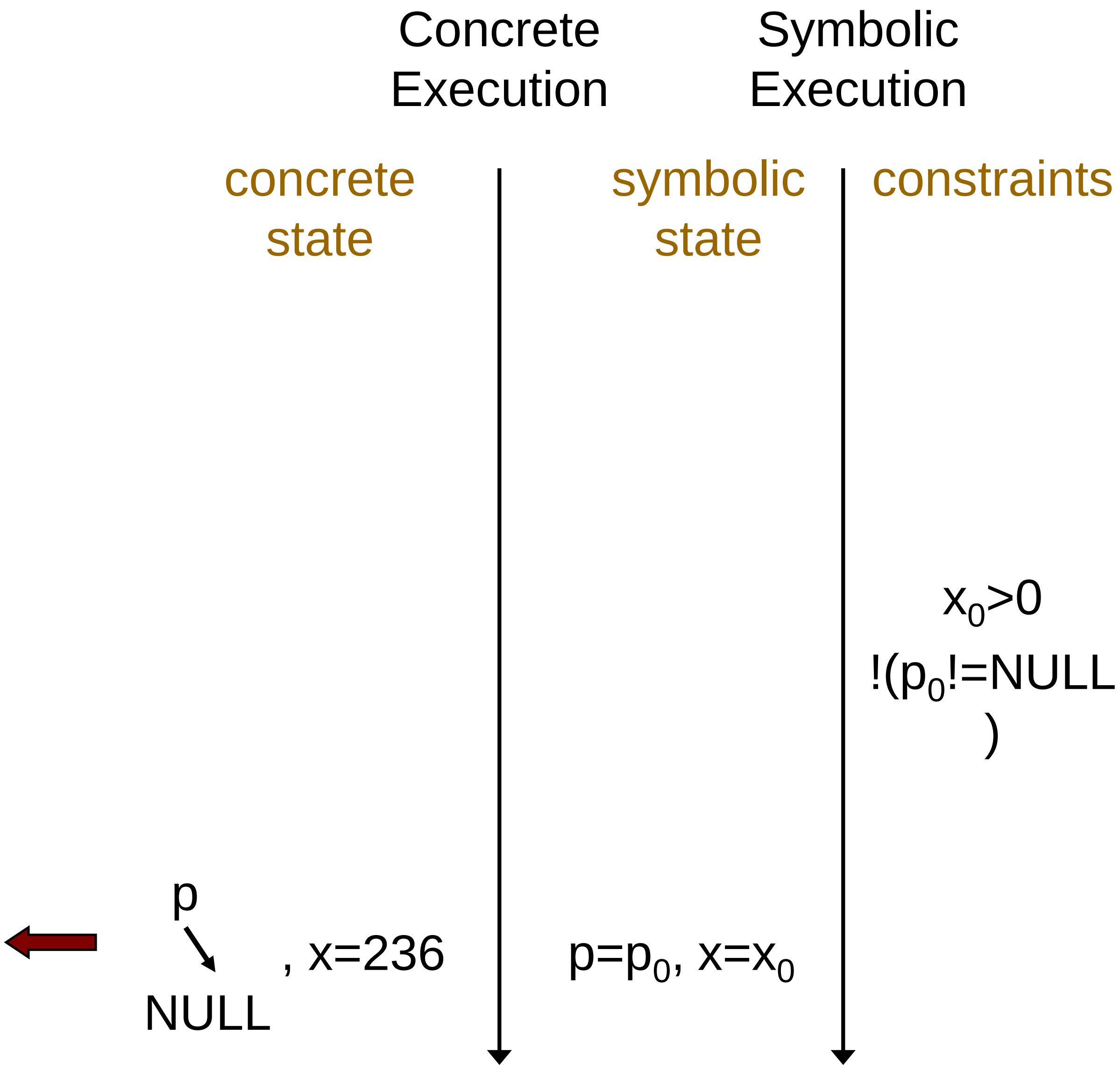
p → NULL, x=236

$p=p_0$, $x=x_0$

$x_0>0$

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;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

p → NULL, x=236

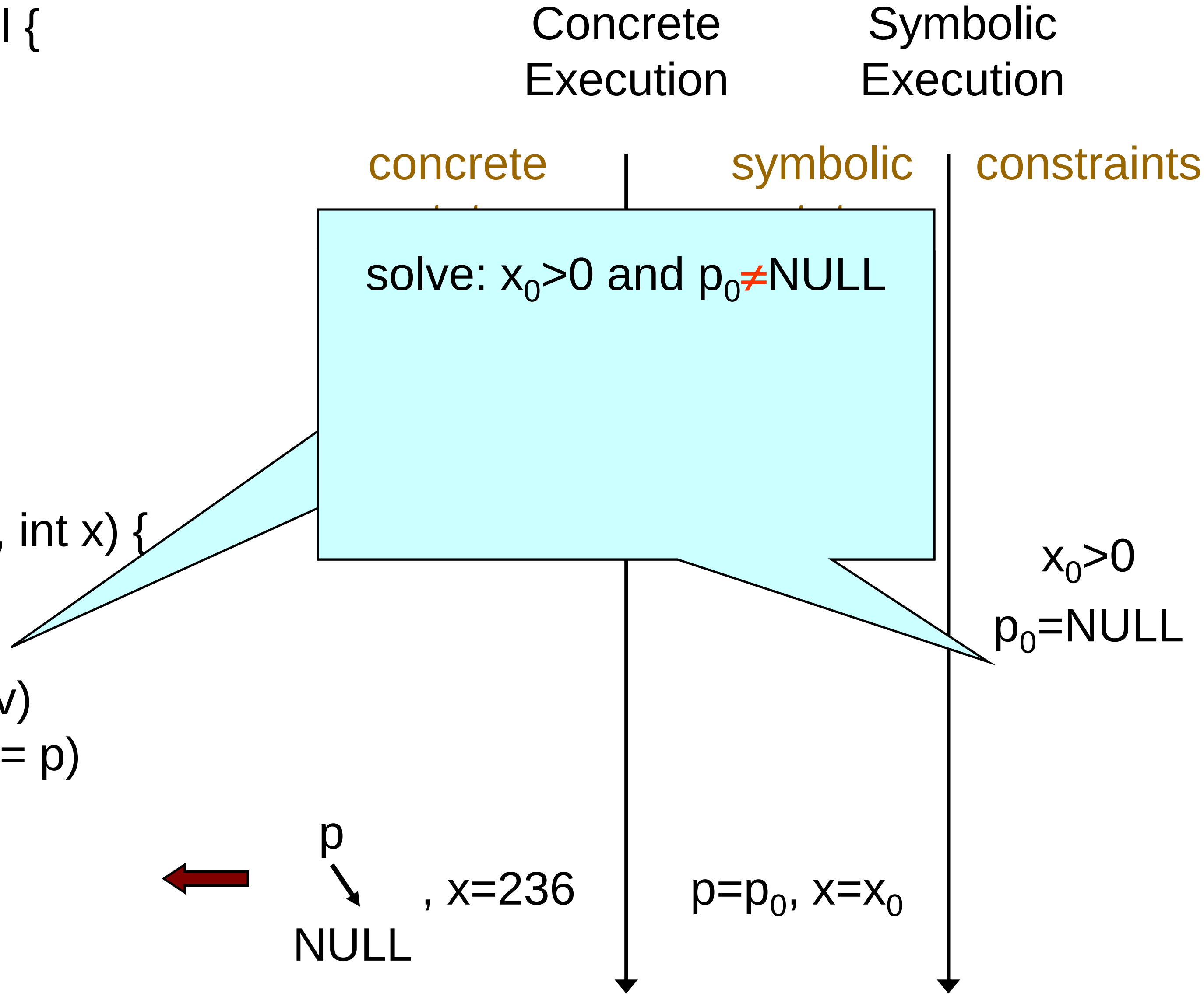
$p=p_0$, $x=x_0$

$x_0>0$

$!(p_0!=NULL)$

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;
}
```



solve: $x_0>0$ and $p_0 \neq NULL$

, x=236

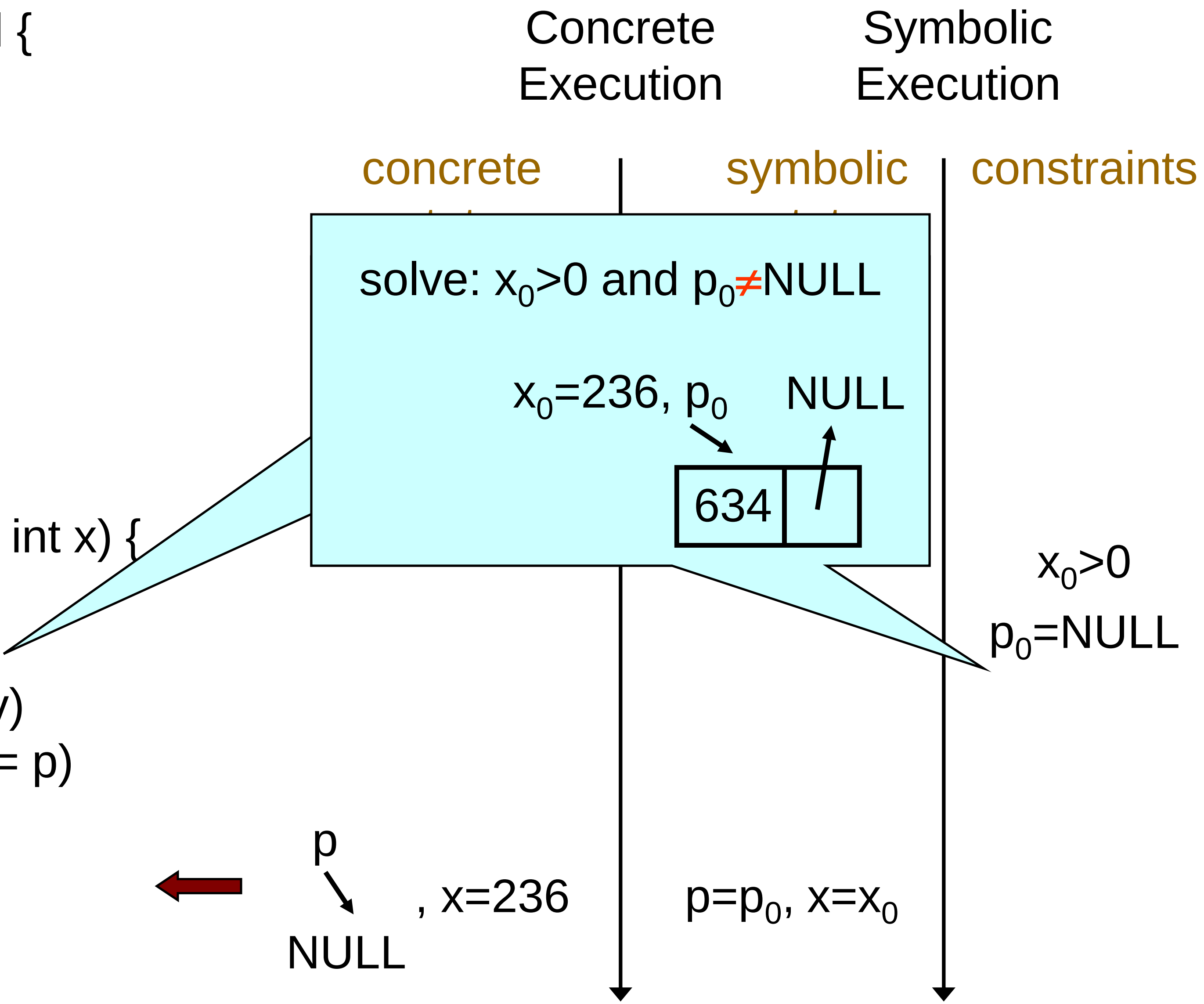
$p=p_0$, $x=x_0$

$x_0>0$

$p_0=NULL$

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;
}
```



Concrete Execution

Symbolic Execution

concrete

symbolic

constraints

solve: $x_0>0$ and $p_0 \neq NULL$

$x_0=236$, p_0 → 634 → NULL

$x_0>0$

$p_0=NULL$

p → NULL, x=236

$p=p_0$, $x=x_0$

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;
}
```

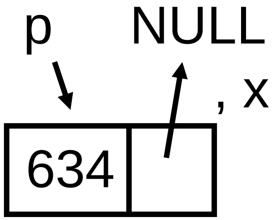
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



, x=236

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

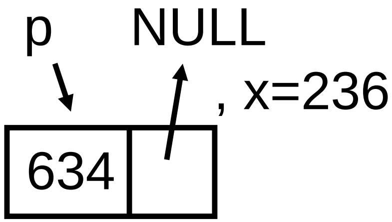
CUTE Approach

```
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;
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
p → 634, next → NULL, x=236	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$



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;
}
```

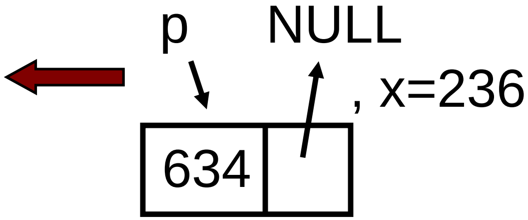
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

$x_0>0$

$p_0 \neq \text{NULL}$

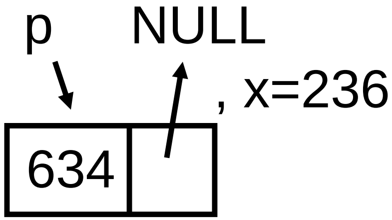
CUTE Approach

```
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;
}
```

Concrete Execution

Symbolic Execution

concrete state



symbolic state

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

constraints

$x_0>0$

$p_0 \neq NULL$

$2x_0+1 \neq v_0$

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;
}
```

Concrete Execution

Symbolic Execution

concrete state

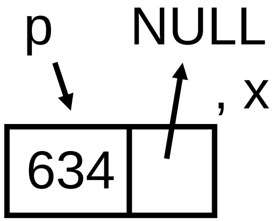
symbolic state

constraints

$x_0>0$

$p_0 \neq NULL$

$2x_0+1 \neq v_0$

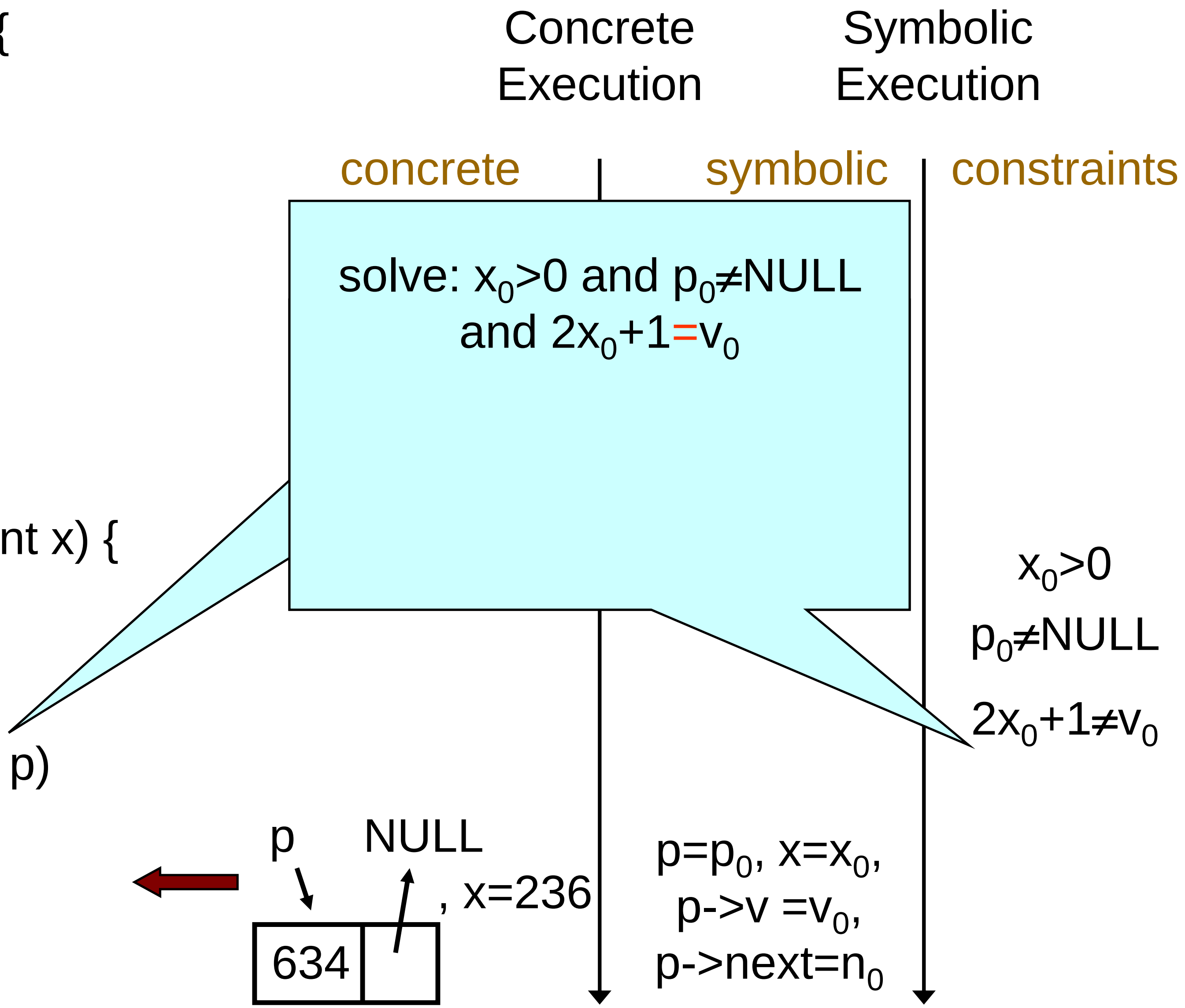


, x=236

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

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;
}
```



Concrete Execution

Symbolic Execution

concrete | symbolic | constraints

solve: $x_0>0$ and $p_0 \neq NULL$ and $2x_0+1=v_0$

, x=236

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

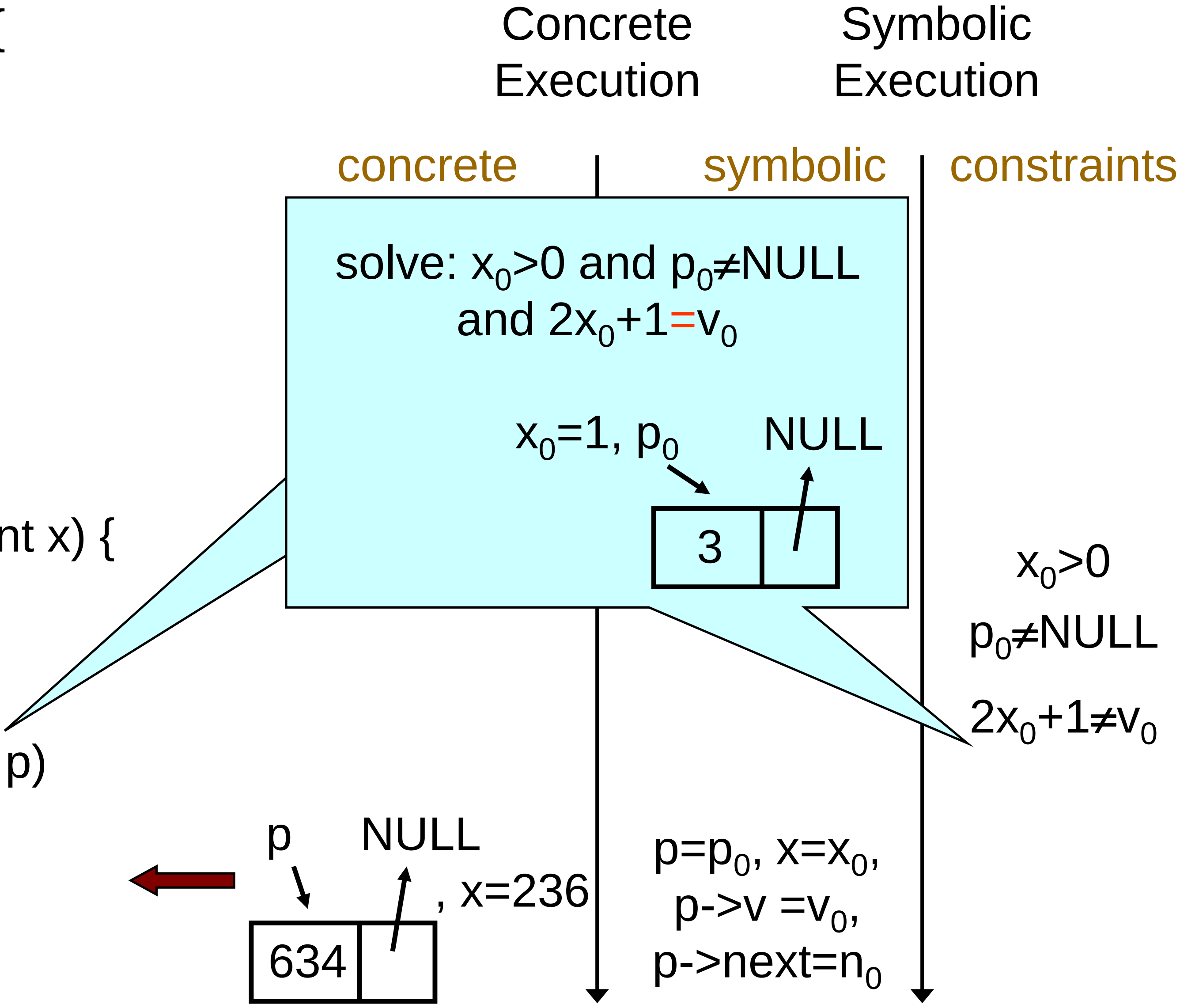
$x_0>0$

$p_0 \neq NULL$

$2x_0+1 \neq v_0$

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;
}
```



Concrete Execution

Symbolic Execution

concrete | symbolic | constraints

solve: $x_0>0$ and $p_0 \neq$ NULL and $2x_0+1=v_0$

$x_0=1$, p_0

$x_0>0$

$p_0 \neq$ NULL

$2x_0+1 \neq v_0$

, x=236

$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$

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;
}
```

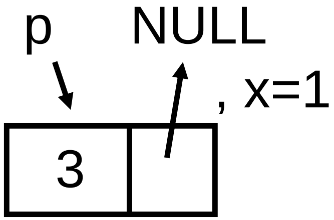
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

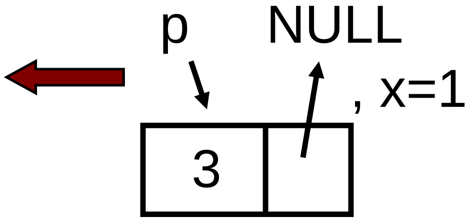
CUTE Approach

```c
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

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

constraints

$x_0>0$

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;
}
```

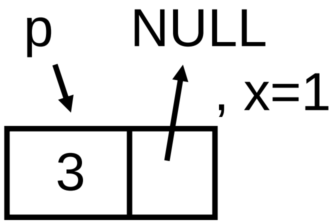
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



, x=1

$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next = n_0$

$x_0>0$

$p_0 \neq NULL$

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;
}
```

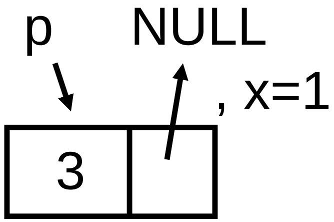
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

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;
}
```

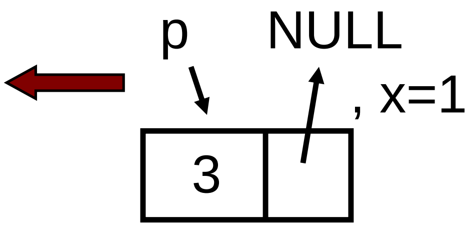
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

$x_0>0$
$p_0 \neq NULL$
$2x_0+1=v_0$
$n_0 \neq p_0$

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;
}
```

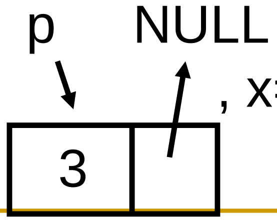
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



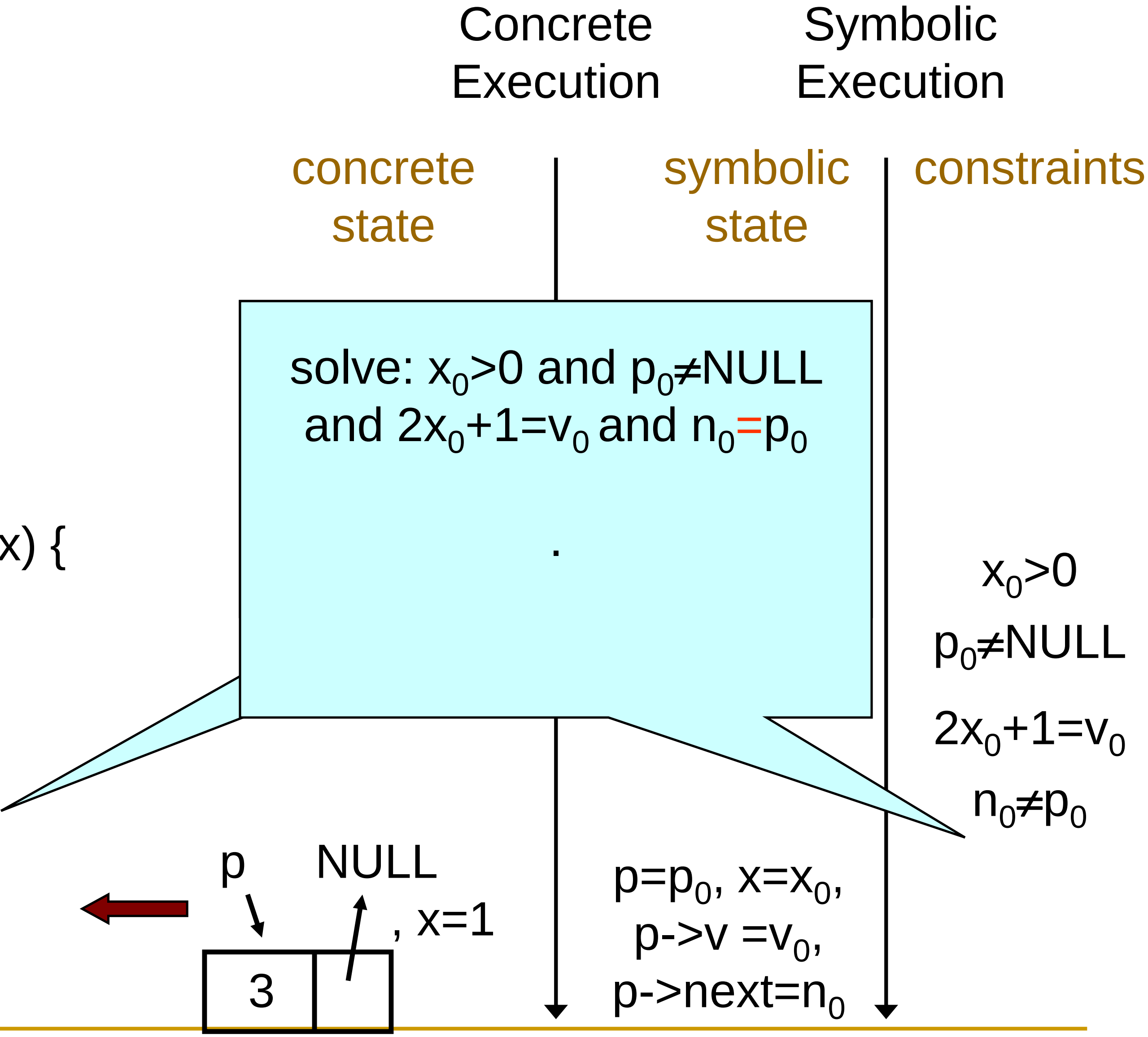
, x=1

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

$x_0>0$
$p_0 \neq NULL$
$2x_0+1=v_0$
$n_0 \neq p_0$

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;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

solve: $x_0>0$ and $p_0 \neq NULL$ and $2x_0+1=v_0$ and $n_0=p_0$

.

$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$

$x_0>0$

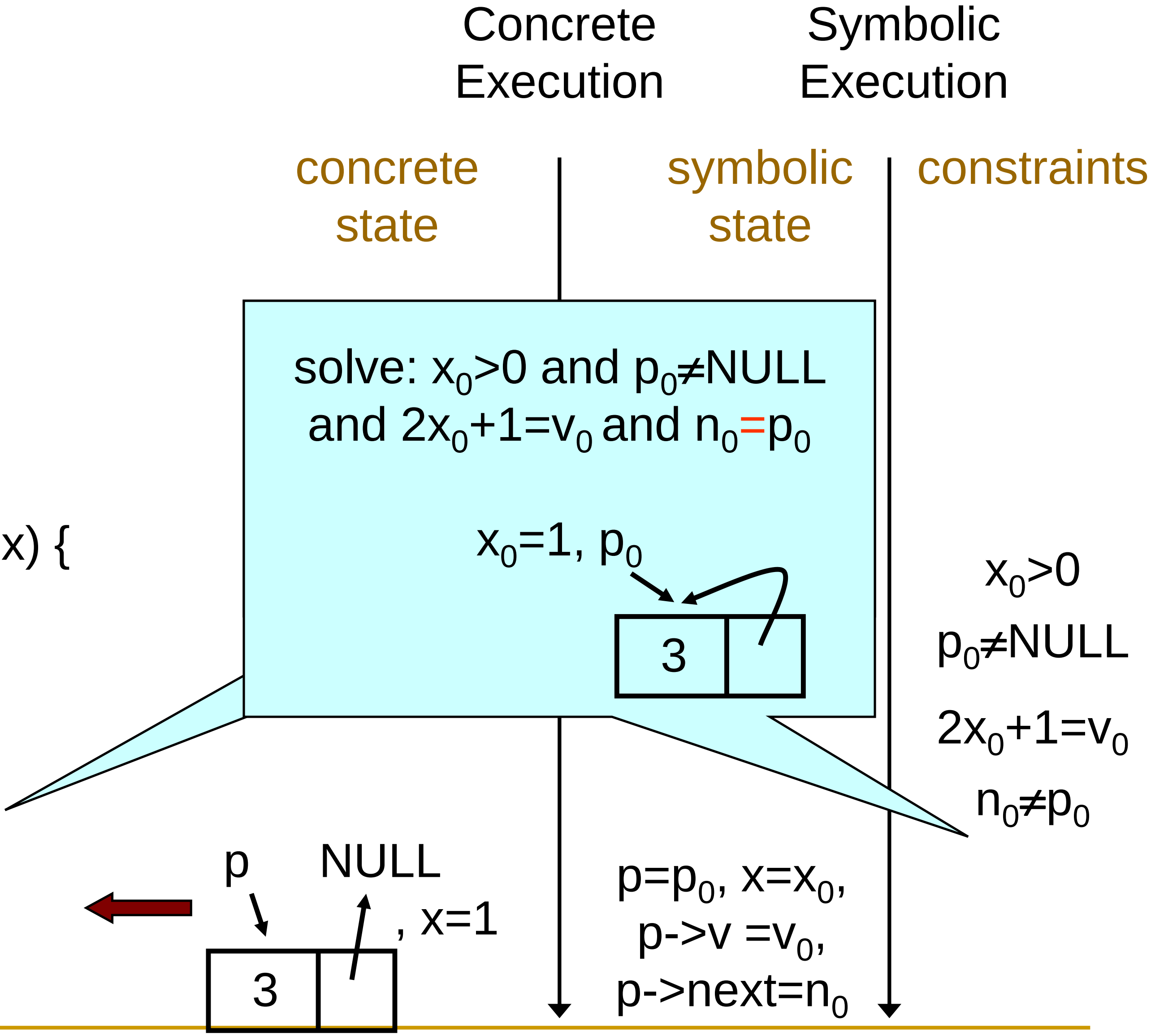
$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

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;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

solve: $x_0>0$ and $p_0 \neq NULL$ and $2x_0+1=v_0$ and $n_0=p_0$

$x_0=1$, p_0

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

p NULL

, x=1

$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$

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;
}
```

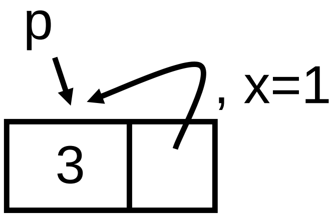
Concrete Execution

Symbolic Execution

concrete state

symbolic state

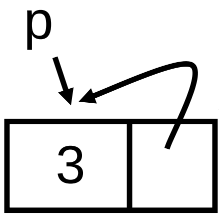
constraints



$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

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;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$

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;
}
```

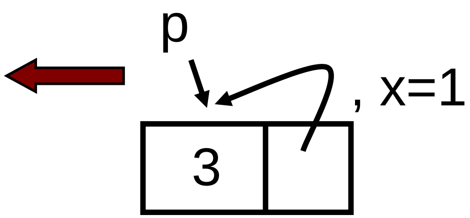
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$

$x_0>0$

$p_0\neq NULL$

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;
}
```

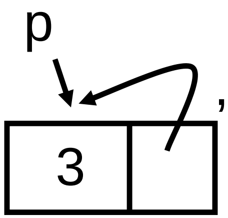
Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints



, x=1

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

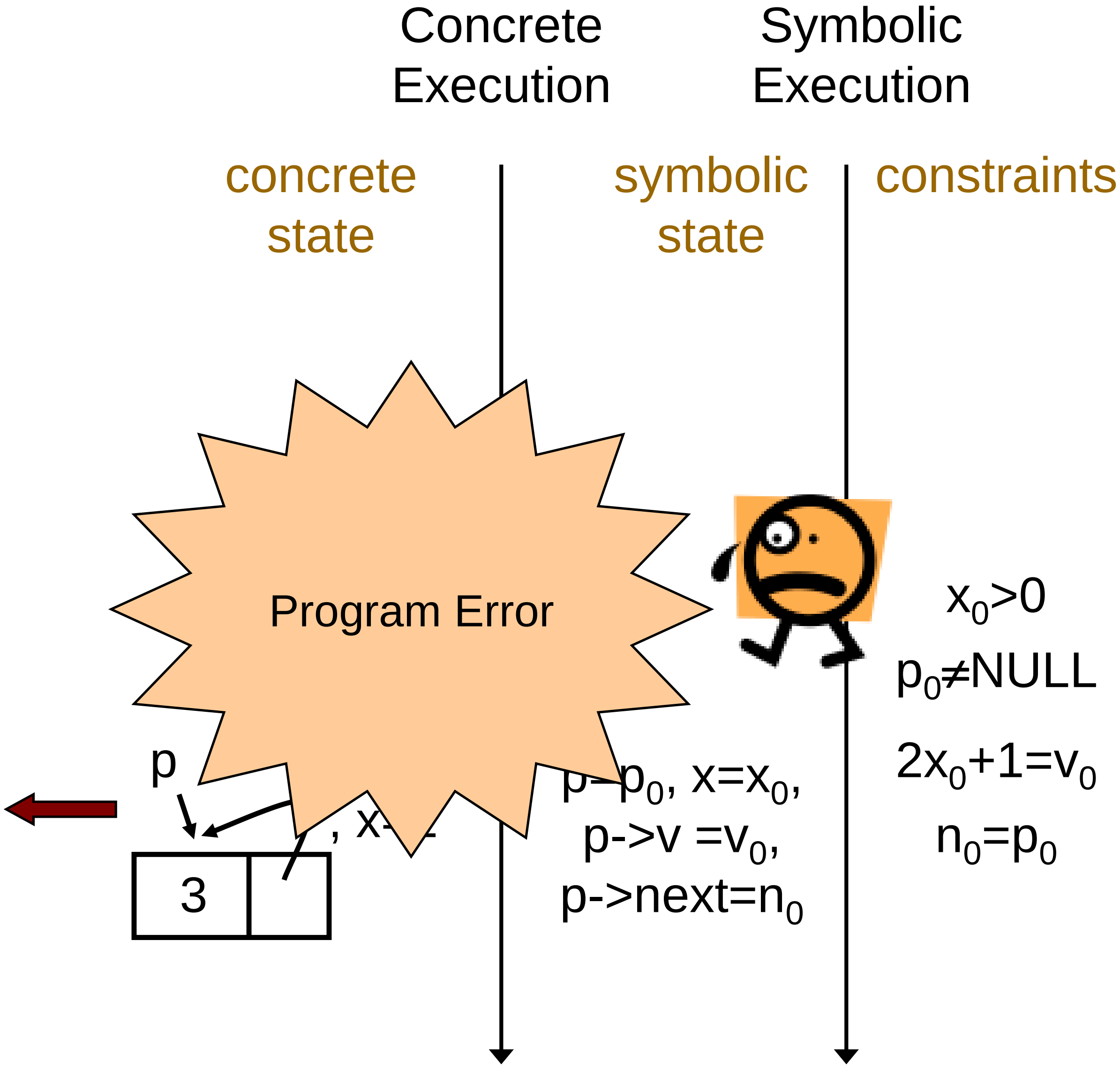
$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

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;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next = n_0$

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0=p_0$

CUTE in a Nutshell

- Generate concrete inputs one by one
  - each input leads program along a different path

CUTE 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

CUTE 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

CUTE 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
      - resolve aliases for pointer using concrete values
      - handle arrays naturally

CUTE 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
      - resolve aliases for pointer using concrete values
      - handle arrays naturally
    - symbolic execution helps to generate concrete input for next execution
      - increases coverage

Data-structure Testing

Solving Data-structure Invariants

```
int isSortedSlist(slist * head) {
  slist * cur, *tmp;
  int i,j;
  if (head == 0) return 1;
  i=j=0;
  for (cur = head; cur!=0; cur = cur->next){
    i++;
    j=1;
    for (tmp = head; j<i; tmp = tmp->next){
      j++;
      if(cur==tmp) return 0;
    }
  }
  for (cur = head; cur->next!=0; cur = cur->next){
    if(cur->i > cur->_next->i) return 0;
  }
  return 1;
}
```

```
testme(slist *L,slist *e){
      CUTE_assume(isSortedSlist(L));
      sglib_slist_add(&L,e);
      CUTE_assert(isSortedSlist(L));
}
```

Data-structure Testing

Generating Call Sequence

```
 for (i=1; i<10; i++) {
   CU_input(toss);
   CU_input(e);
    switch(toss){
   case 2:         sglib_hashed_ilist_add_if_not_member(htab,e,&m);
    break;
   case 3:         sglib_hashed_ilist_delete_if_member(htab,e,&m);
    break;
   case 4:         sglib_hashed_ilist_delete(htab,e); break;
   case 5:         sglib_hashed_ilist_is_member(htab,e); break;
   case 6:         sglib_hashed_ilist_find_member(htab,e); break;
   }
}
```

抽象解释

抽象解释

- 用于论证抽象正确性的理论
- 相当数量的文献采用抽象解释来论证正确性

- 转向使用Alex Aiken的课程胶片

课后作业

- 简答：如果用抽象解释理论论证数据流分析的安全性，抽象域、具体域和σ、μ、α、γ分别是什么？符号执行呢？
  - 简述概念即可，不需要写出形式定义