CUTE: A Concolic Unit Testing Engine for C

Koushik Sen Darko Marinov Gul Agha

University of Illinois Urbana-Champaign

Goal

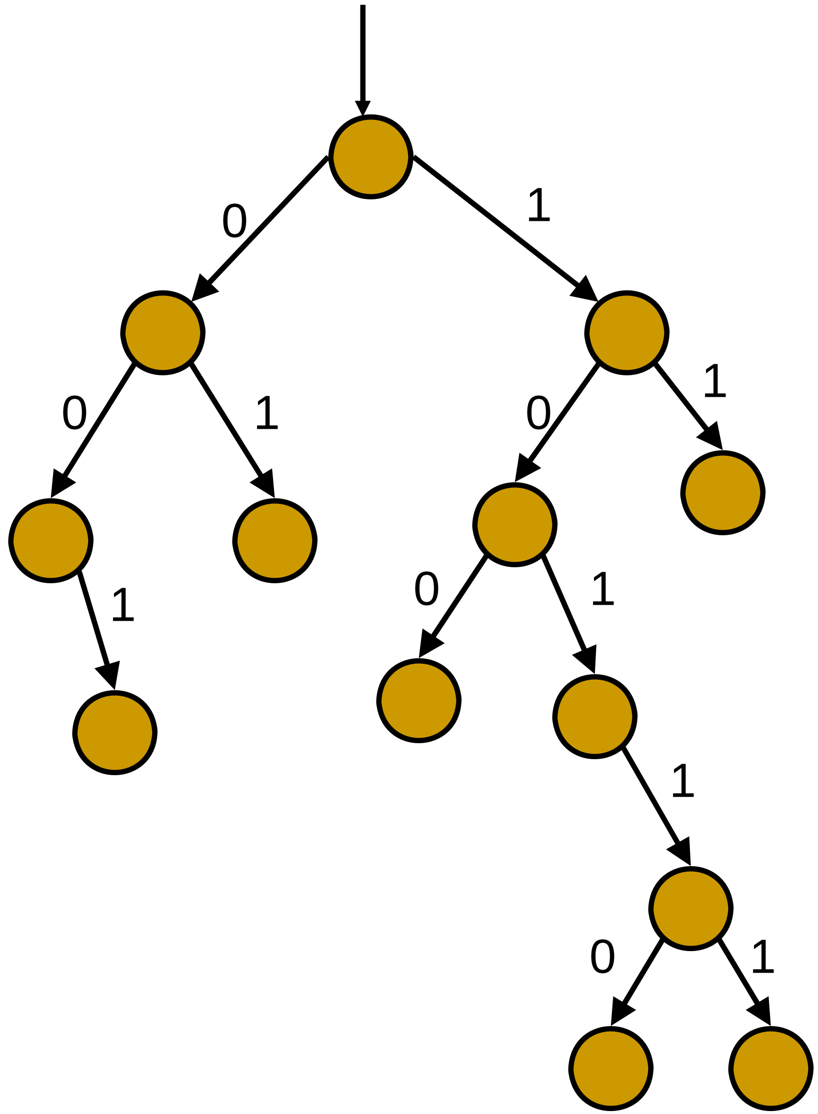
- Automated Scalable Unit Testing of real-world C Programs
  - Generate test inputs
  - Execute unit under test on generated test inputs
    - so that all reachable statements are executed
  - Any assertion violation gets caught

Goal

- Automated Scalable Unit Testing of real-world C 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

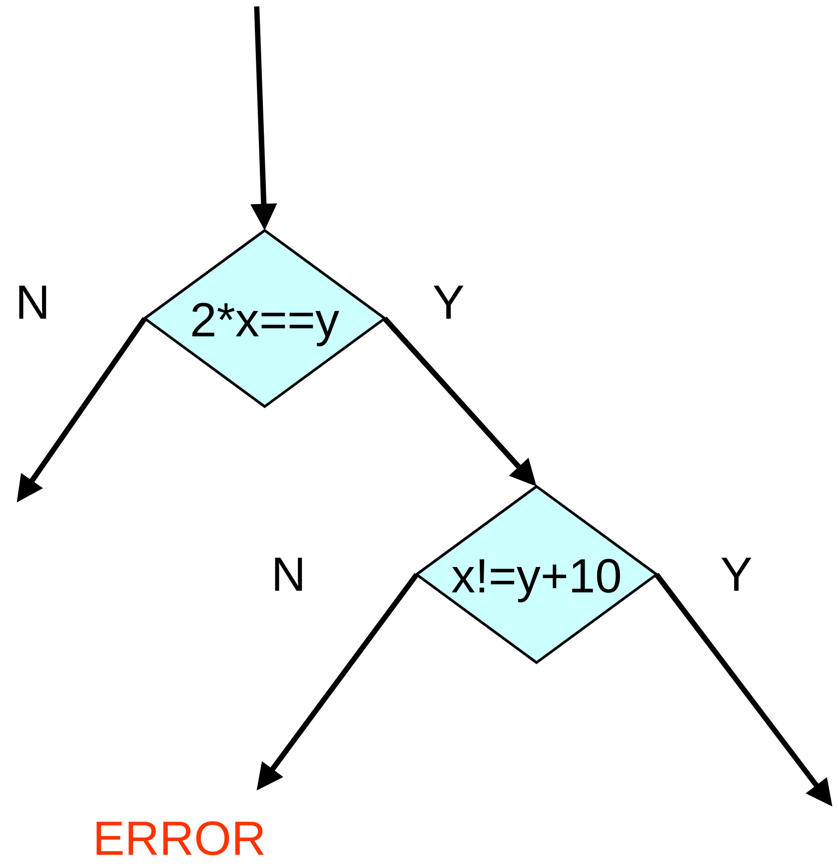
Execution Paths of a Program

- 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;
    }
  }
}
```



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

Approach

- Combine concrete and symbolic execution for unit testing
  - **Conc**rete + Symb**olic** = Concolic
- In a nutshell
  - Use concrete execution over a concrete input to guide symbolic execution
  - Concrete execution helps Symbolic execution to simplify complex and unmanageable symbolic expressions
    - by replacing symbolic values by concrete values
- Achieves Scalability
  - Higher branch coverage than random testing
  - No false positives or scalability issue like in symbolic execution based testing

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
			$x_0>0$
			$!(p_0!=NULL)$
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 | symbolic | constraints

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

$x_0>0$

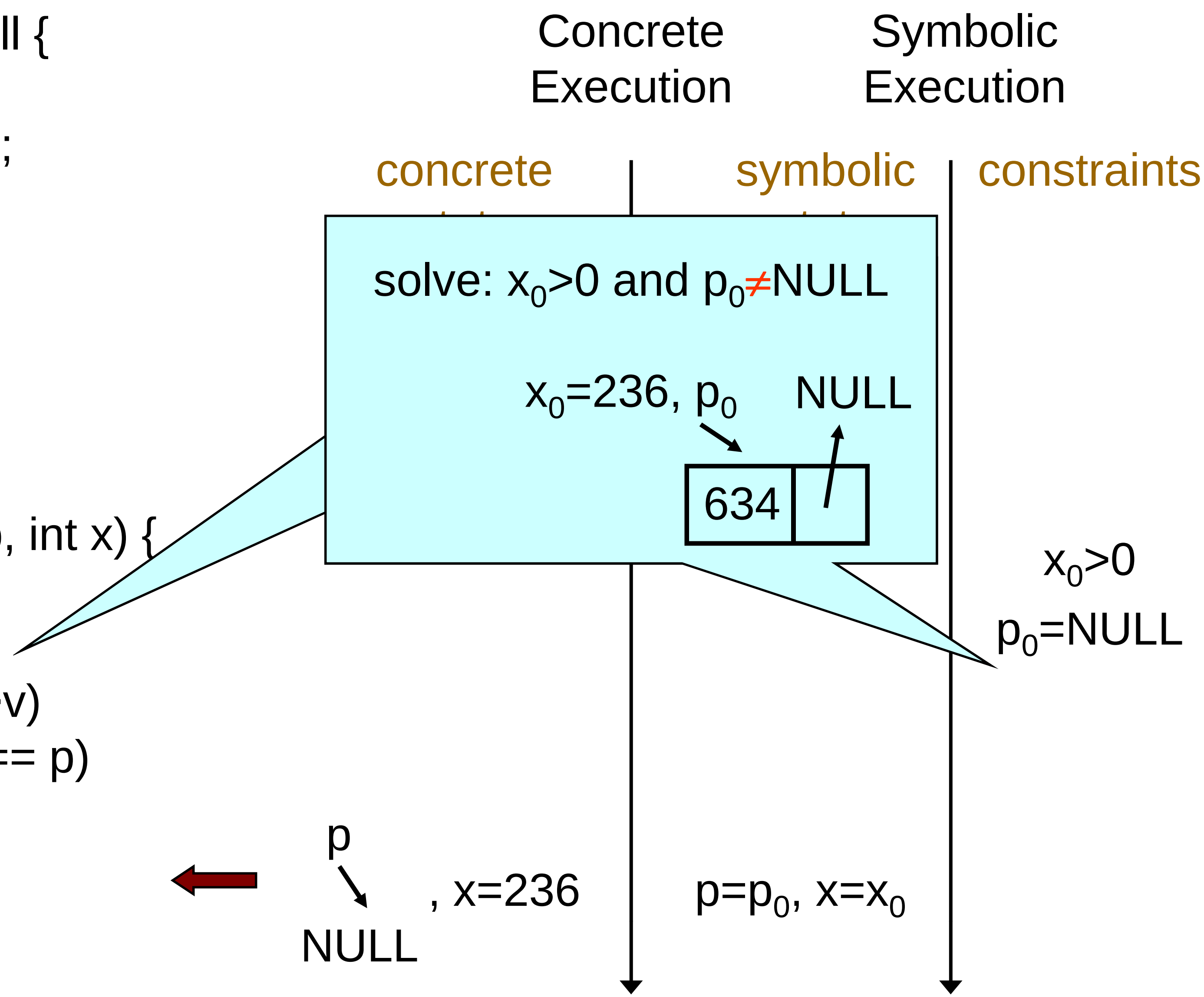
p_0=NULL

p → NULL, x=236

$p=p_0$, $x=x_0$

CUTE Approach

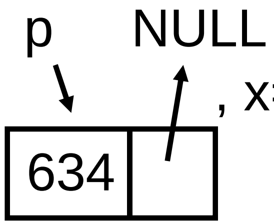
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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;
}
```



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



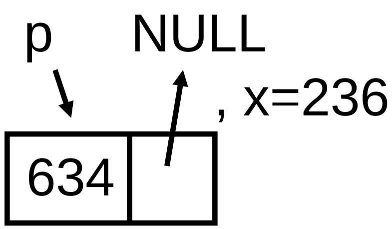
, x=236

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

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



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

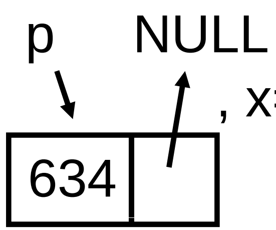
$x_0>0$

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



, x=236

symbolic state

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

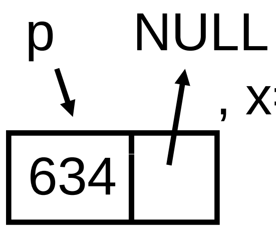
constraints

$x_0>0$

$p_0 \neq NULL$

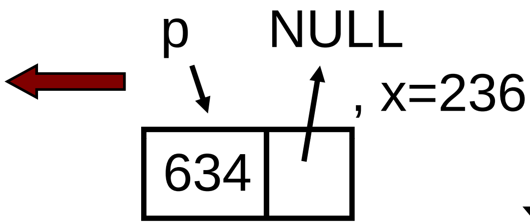
CUTE Approach

```
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	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1 \neq v_0$

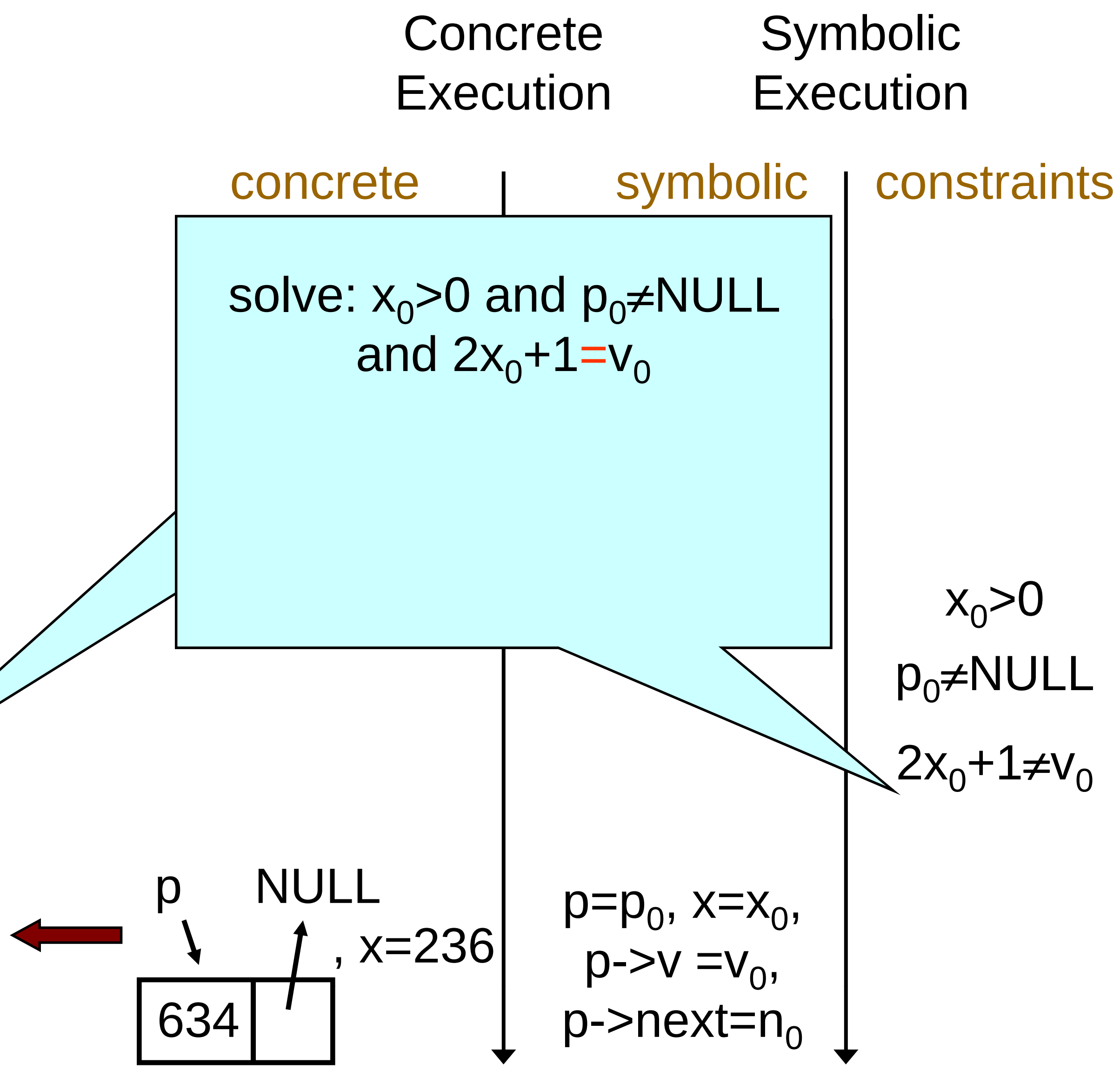
CUTE Approach

```
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
	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1 \neq v_0$

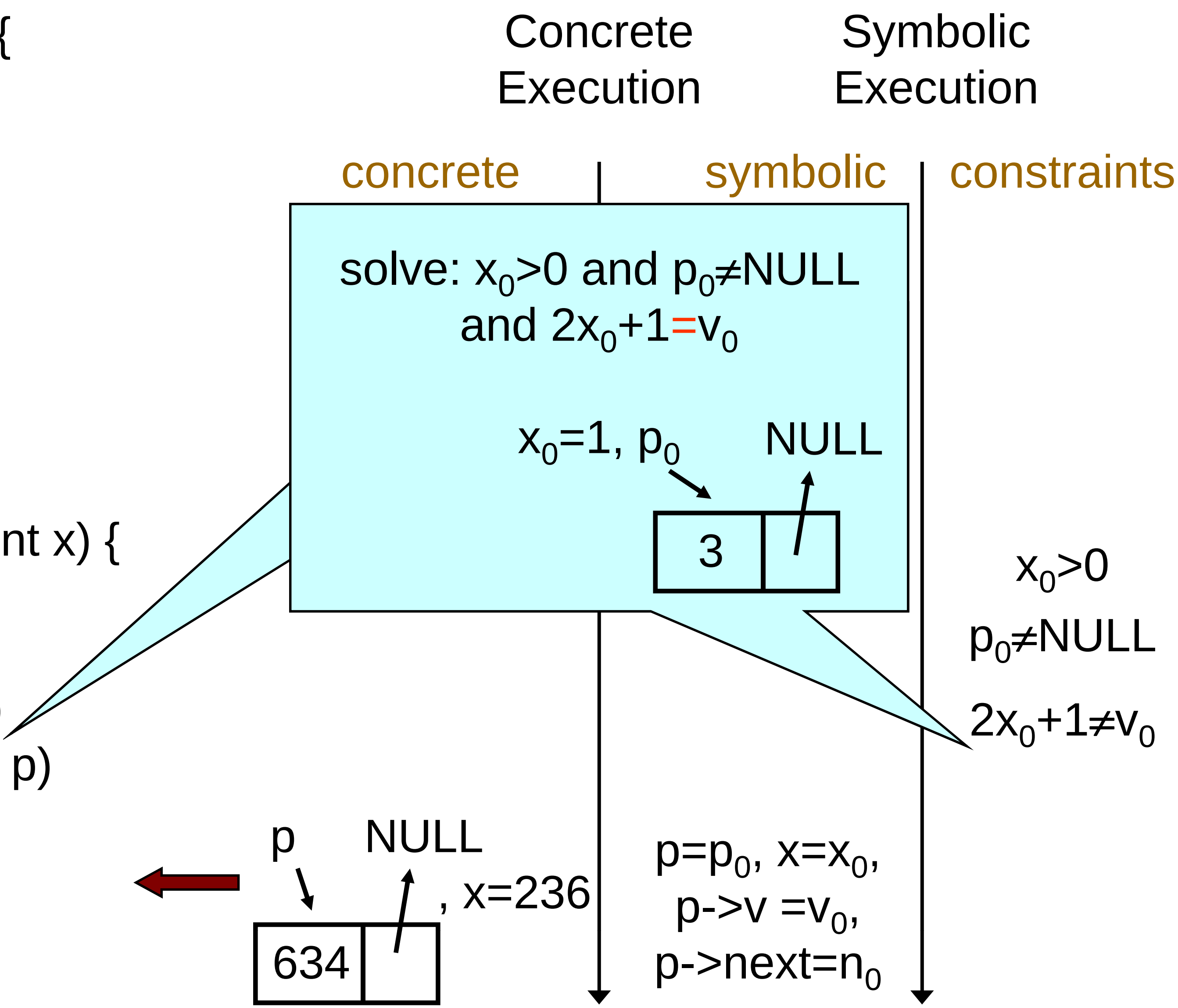
CUTE Approach

```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;
}
```



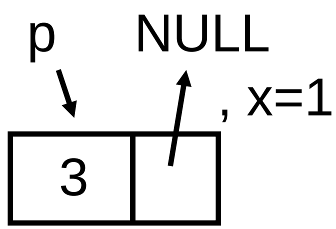
CUTE Approach

```
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;
}
```



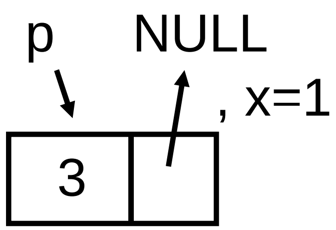
CUTE Approach

```
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->v =v_0, p->next=n_0	

CUTE Approach

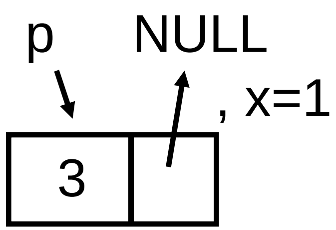
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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
	$p=p_0$, $x=x_0$, $p\text{->}v=v_0$, $p\text{->}next=n_0$	$x_0>0$

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

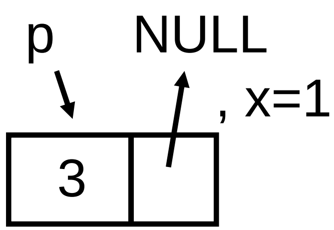


symbolic state: $p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$

constraints: $x_0>0$, $p_0 \neq NULL$

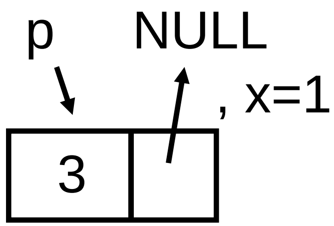
CUTE Approach

```
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

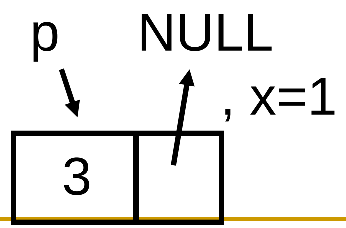
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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

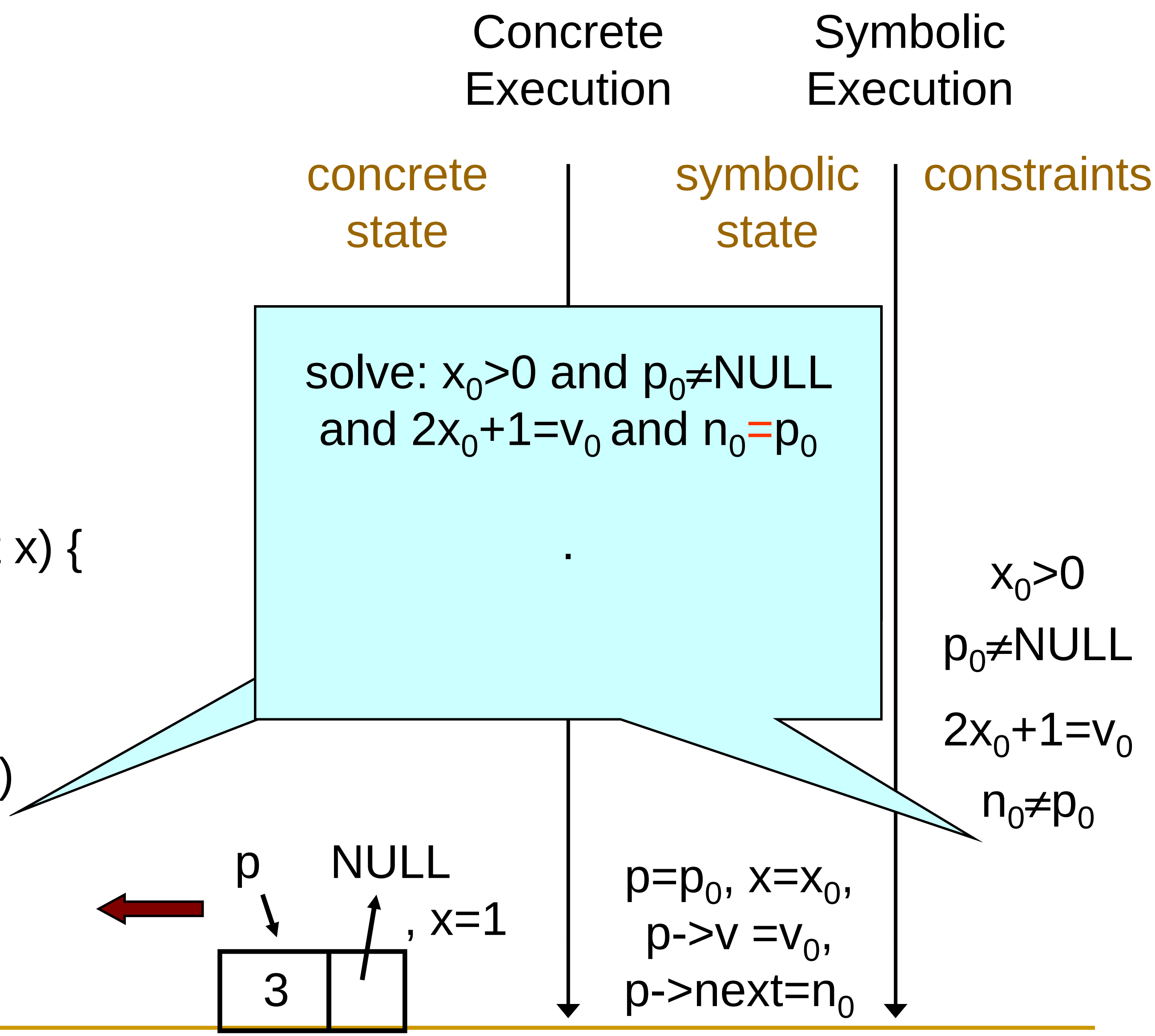
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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
		$x_0>0$
		$p_0 \neq NULL$
		$2x_0+1=v_0$
		$n_0 \neq p_0$
	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	



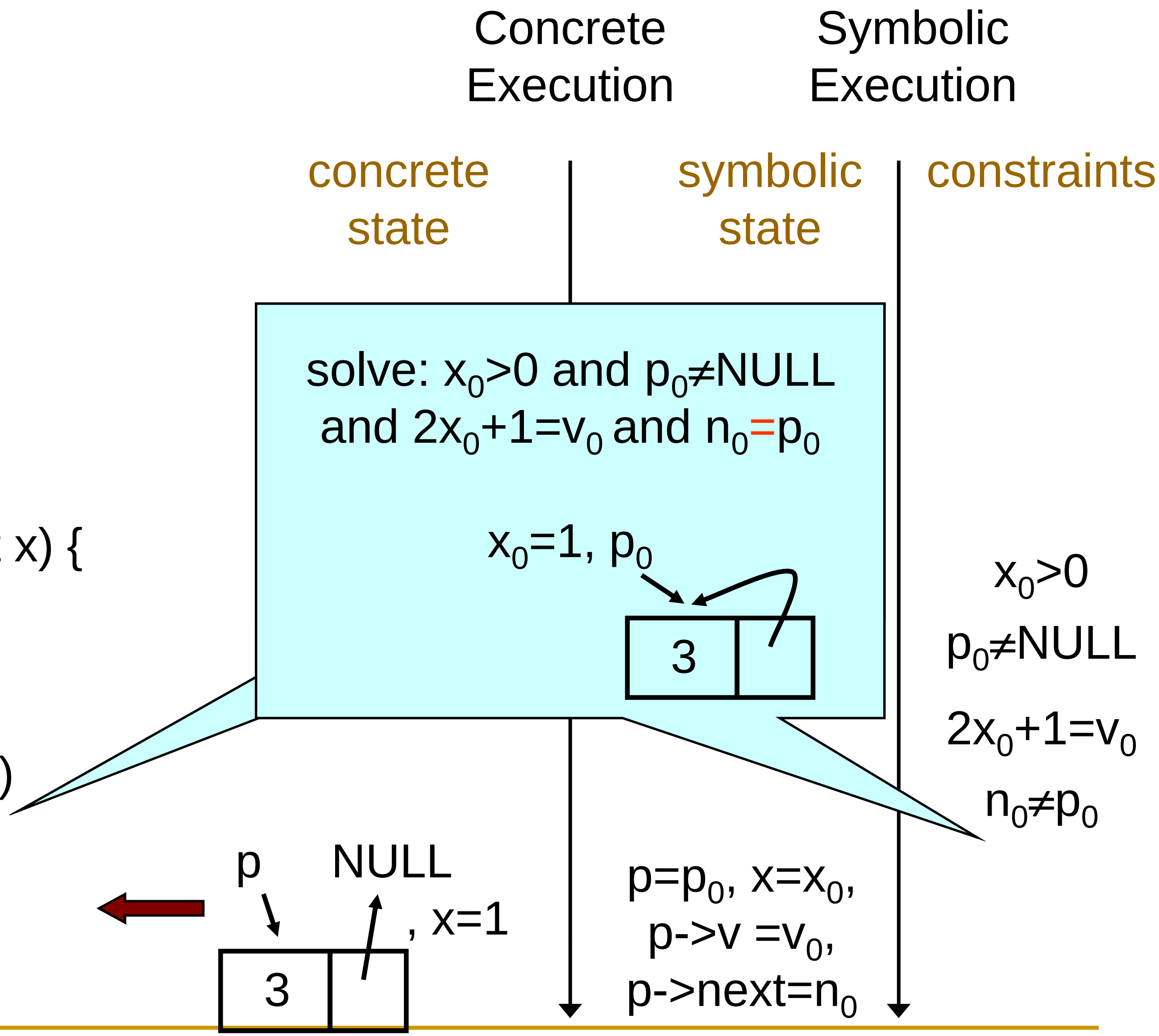
CUTE Approach

```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;
}
```



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;
}
```

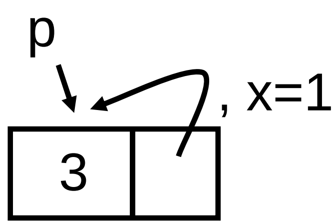


CUTE Approach

```
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

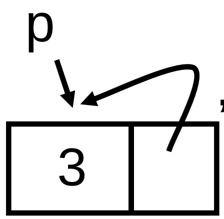
concrete state | symbolic state | constraints



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

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;
}
```

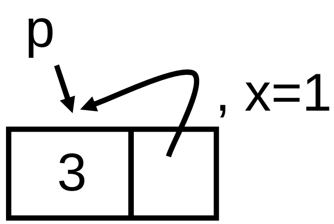
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next = n_0$	$x_0>0$

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

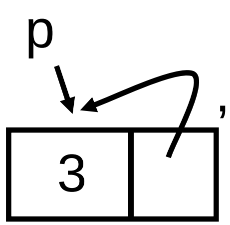


symbolic state: $p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$

constraints: $x_0>0$, $p_0 \neq NULL$

CUTE Approach

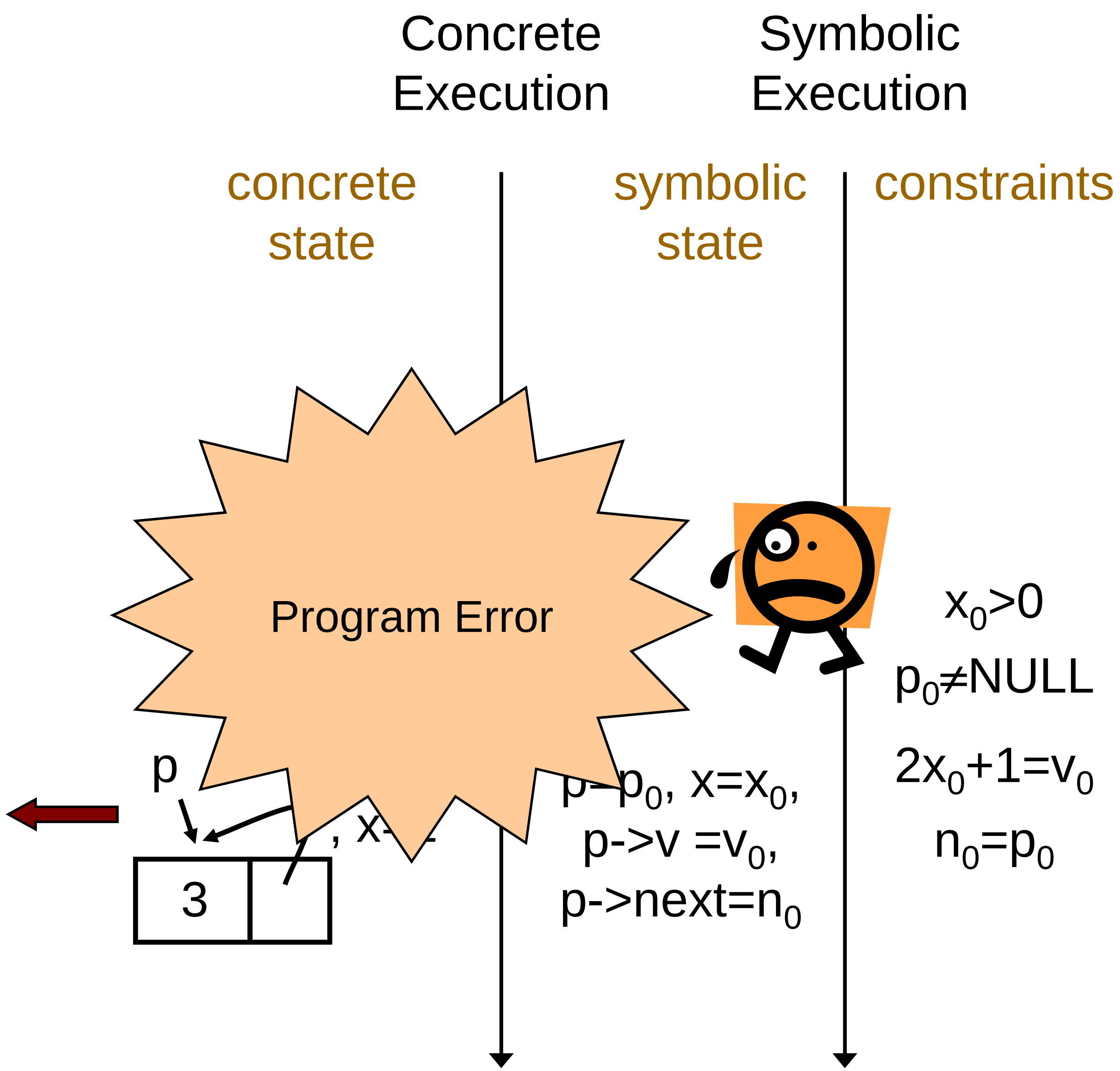
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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
	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1=v_0$

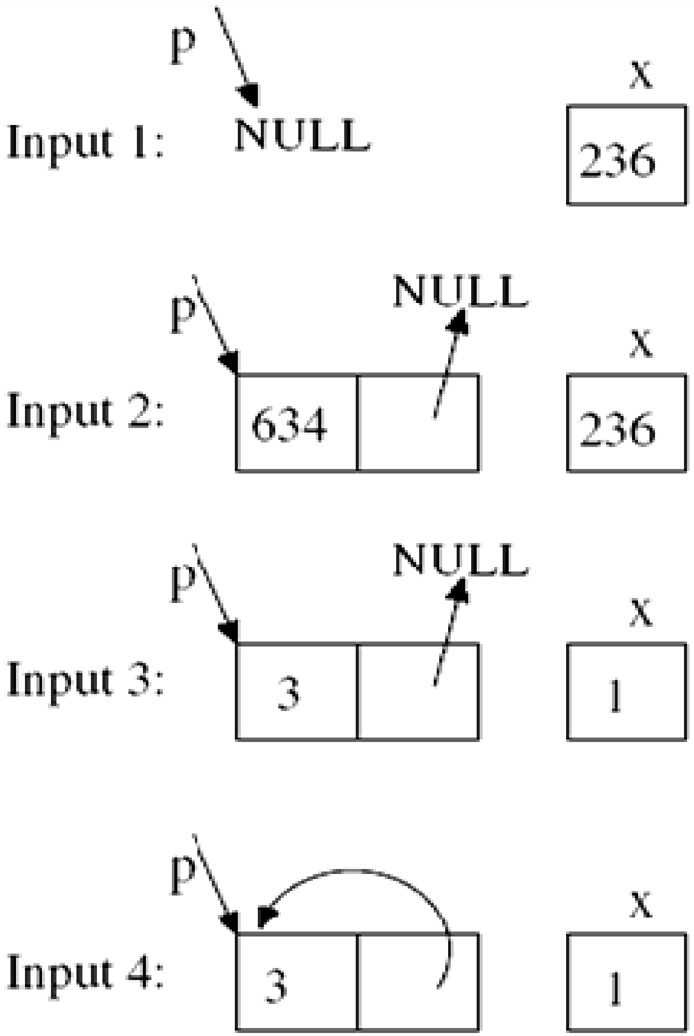
CUTE Approach

```
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;
}
```



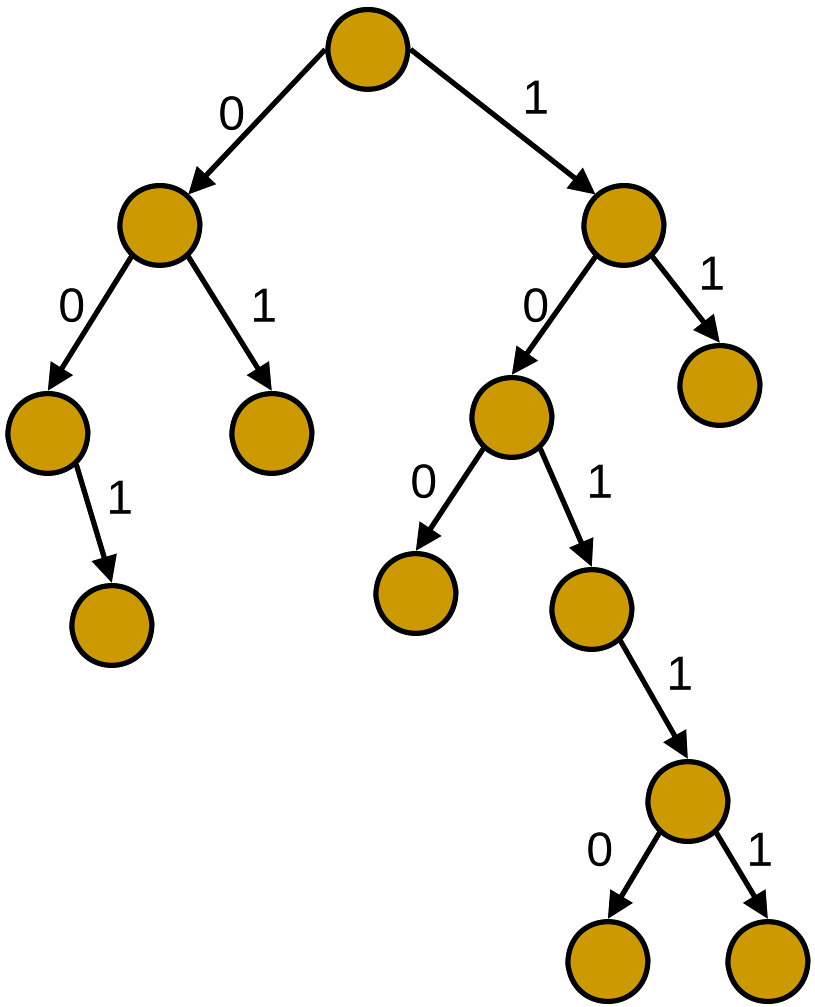
Pointer Inputs: Input Graph

```
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;
}
```



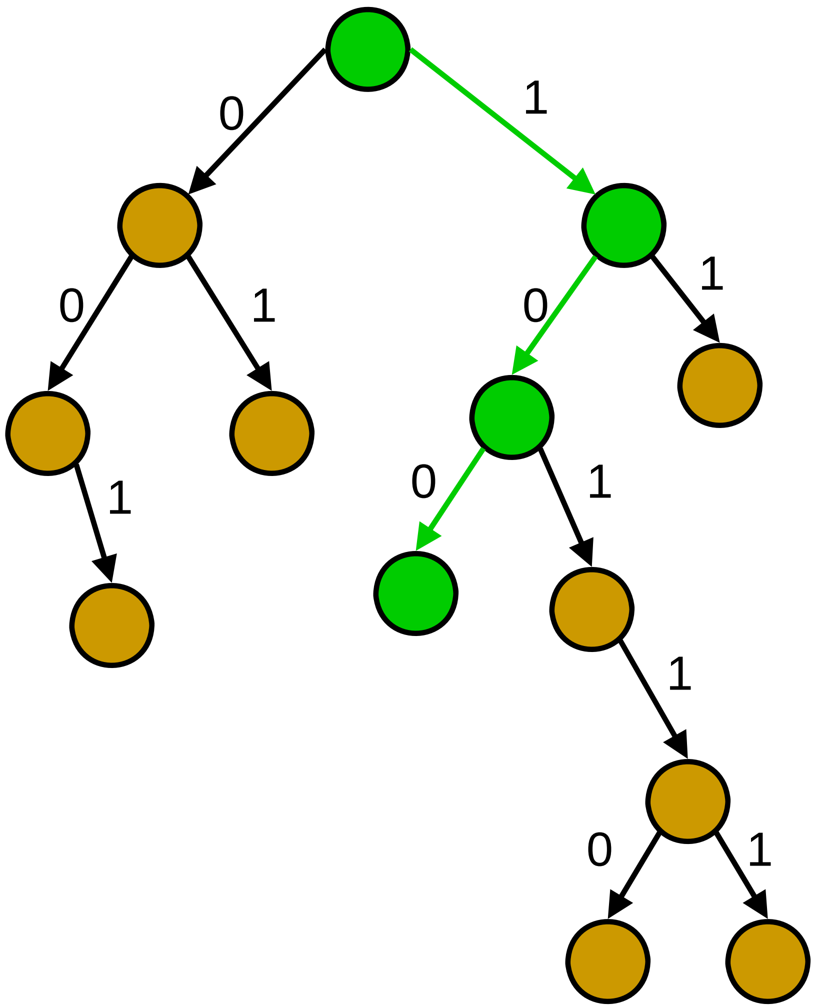
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



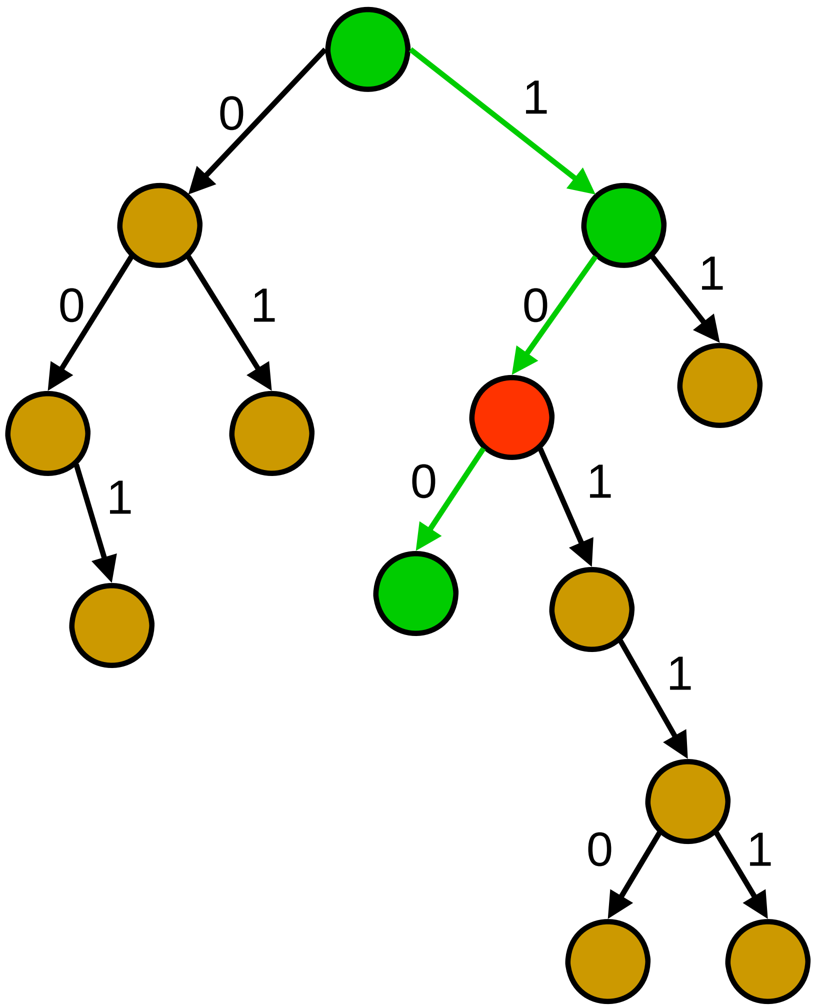
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



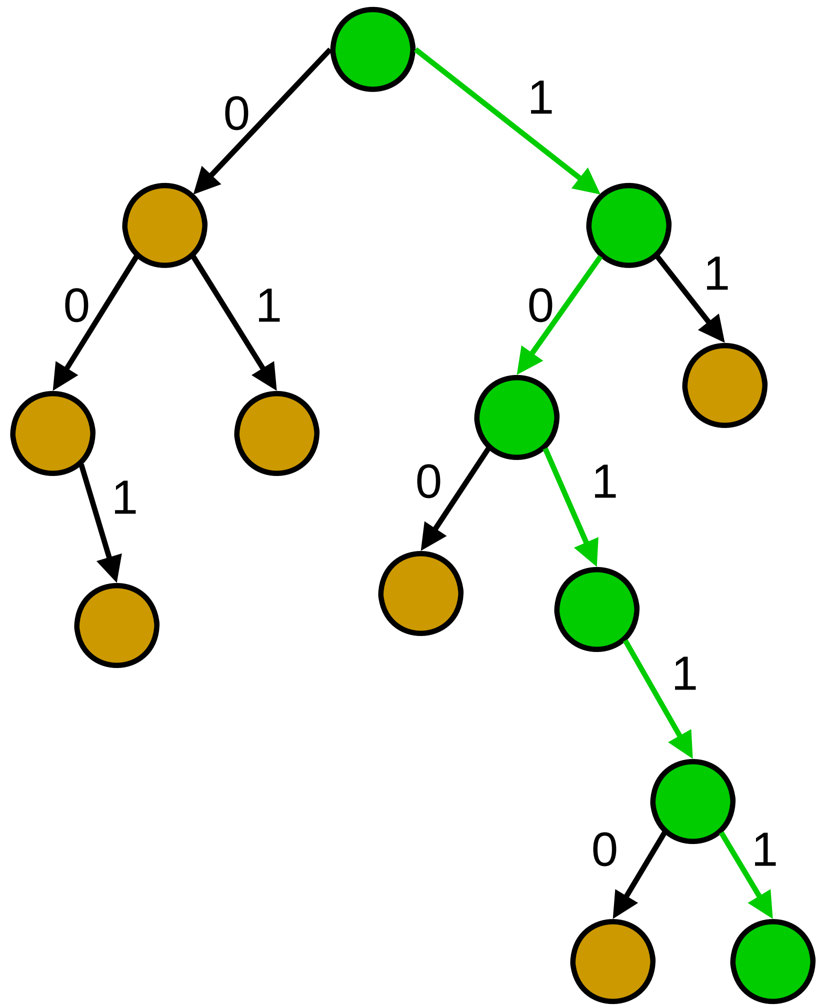
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



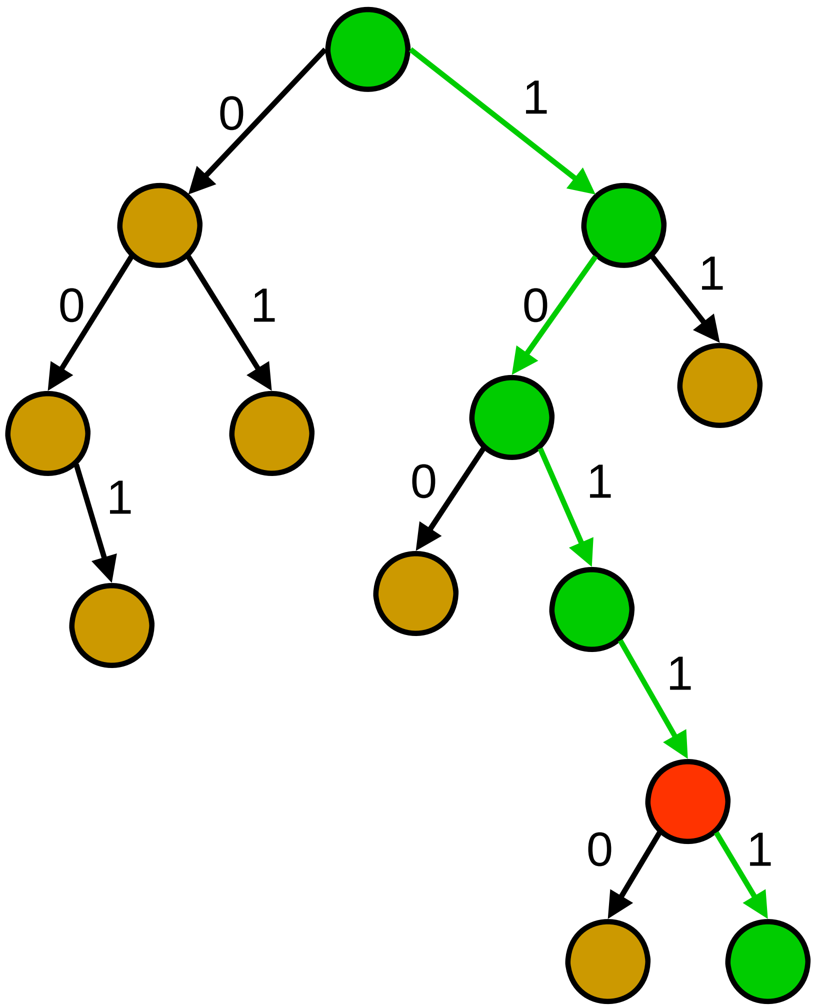
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



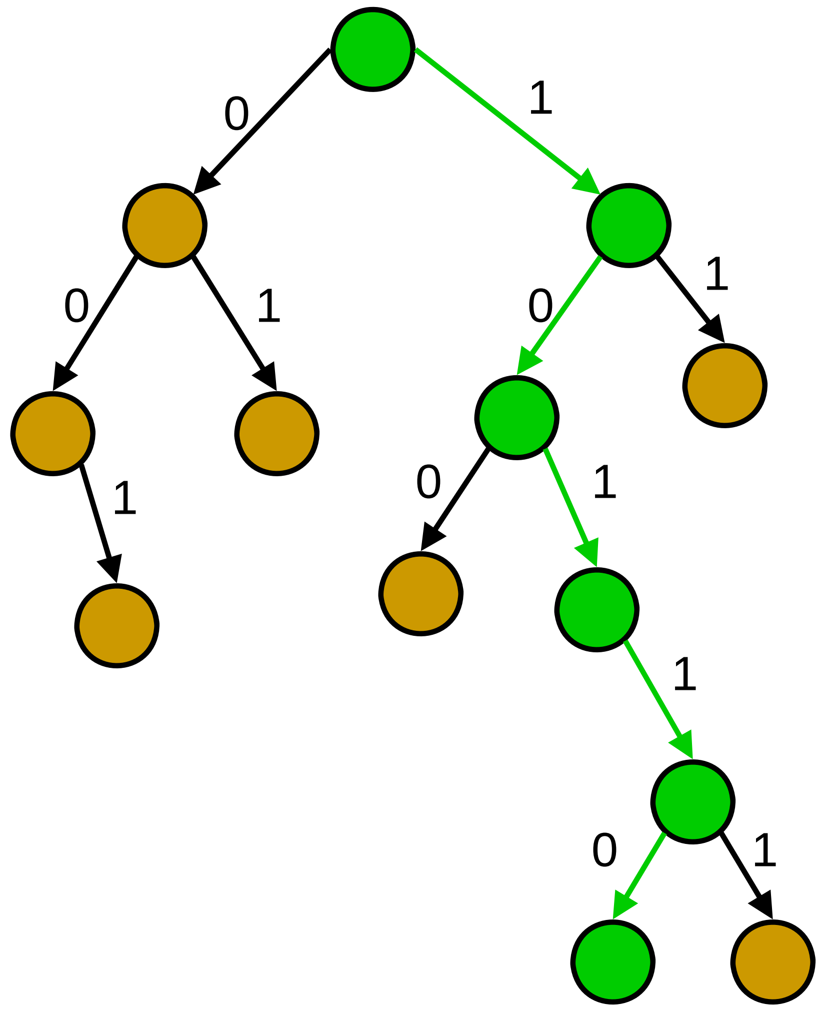
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



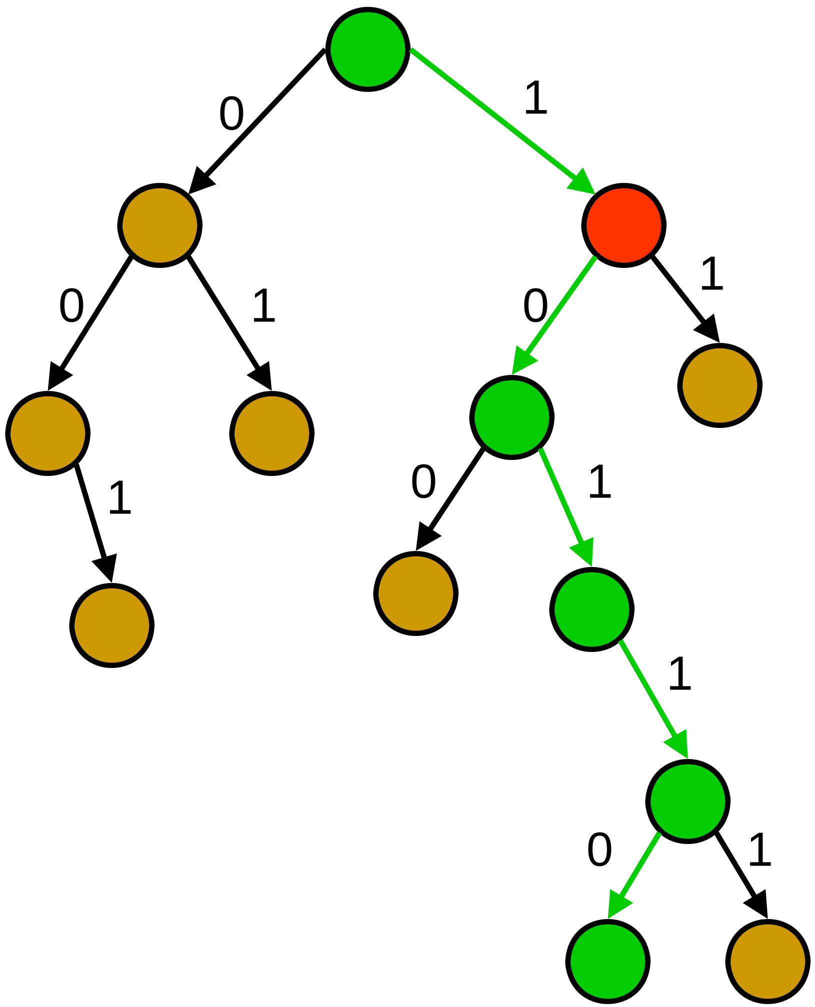
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



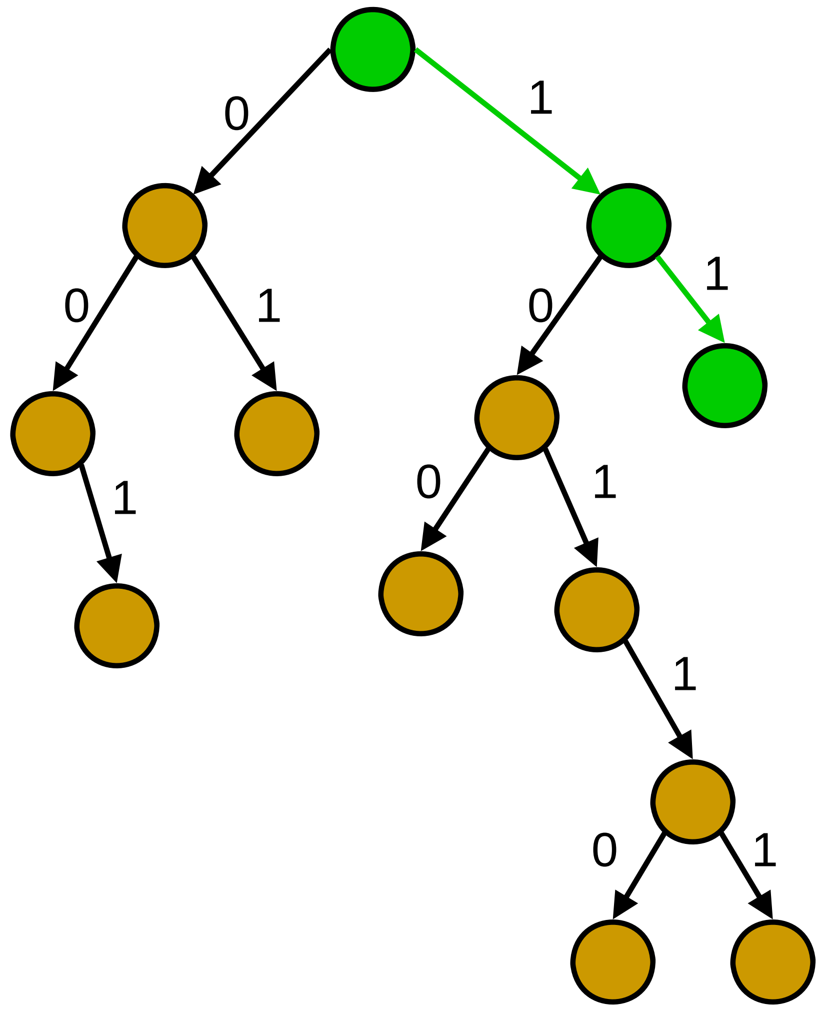
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - check for assertion violations
  - check for program crash
  - combine with valgrind to discover memory leaks
  - detect invariants



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

Testing Data-structures of CUTE itself

- Unit tested several non-standard data-structures implemented for the CUTE tool
  - cu_depend (used to determine dependency during constraint solving using graph algorithm)
  - cu_linear (linear symbolic expressions)
  - cu_pointer (pointer symbolic expressions)
- Discovered a few memory leaks and a couple of segmentation faults
  - these errors did not show up in other uses of CUTE
  - for memory leaks we used CUTE in conjunction with Valgrind

SGLIB: popular library for C data-structures

- Used in Xrefactory a commercial tool for refactoring C/C++ programs
- Found two bugs in sglib 1.0.1
  - reported them to authors
  - fixed in sglib 1.0.2
- Bug 1:
  - doubly-linked list library
    - segmentation fault occurs when a non-zero length list is concatenated with a zero-length list
    - discovered in 140 iterations (< 1second)
- Bug 2:
  - hash-table
    - an infinite loop in hash table is member function
  - 193 iterations (1 second)

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
- take then branch with constraint x*x*x+ 3*x*x+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
- 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 : CUTE handles this smartly

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){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

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)

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();
    }
}
```

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

Related Work

- “DART: Directed Automated Random Testing” by Patrice Godefroid, Nils Klarlund, and Koushik Sen (PLDI’05)
  - handles only arithmetic constraints
- CUTE
  - Supports C with
    - pointers, data-structures
  - Highly efficient constraint solver
    - 100 -1000 times faster
  - arithmetic, pointers
  - Provides Bounded Depth-First Search and Random Search strategies
  - Publicly available tool that works on ALL C programs

Discussion

- CUTE is
  - light-weight
  - dynamic analysis (compare with static analysis)
    - ensures no false alarms
  - concrete execution and symbolic execution run simultaneously
    - symbolic execution consults concrete execution whenever dynamic analysis becomes intractable
  - real tool that works on **all** C programs
    - completely automatic
- Requires actual code that can be fully compiled
- Can sometime reduce to Random Testing
- Complementary to Static Analysis Tools

Current Work

Concurrency Support

- dynamic pruning to avoid exploring equivalent interleaving

- Application to find Dolev-Yao attacks in security protocols