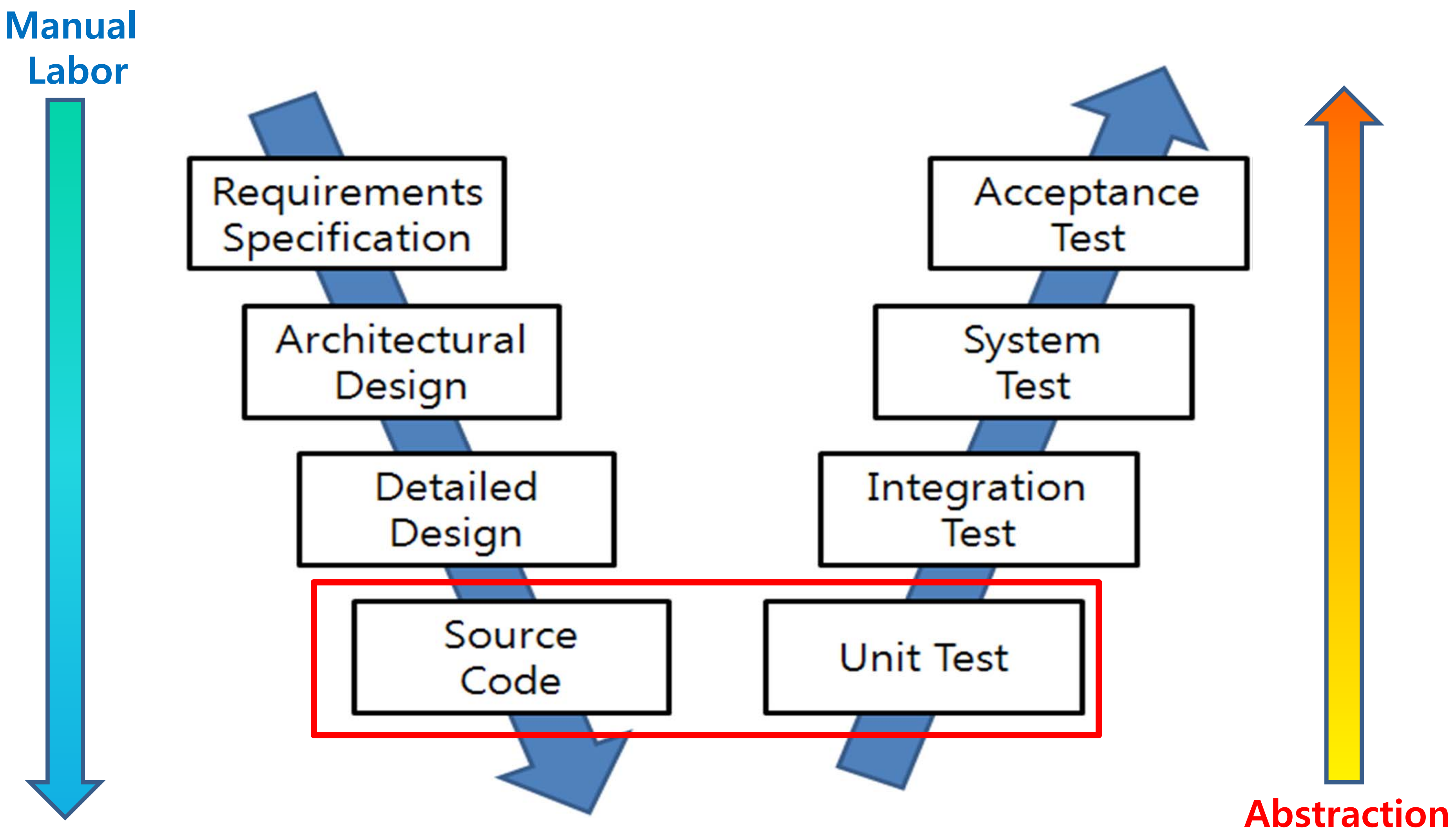
Automated Software Analysis Techniques For High Reliability: A Concolic Testing Approach

Moonzoo Kim
Provable Software Lab, CS Dept, KAIST

Contents

- Automated Software Analysis Techniques
  - Background
  - Concolic testing process
  - Example of concolic testing
- Case Study: Busybox utility
- Future Direction and Conclusion

Main Target of Automated SW Analysis



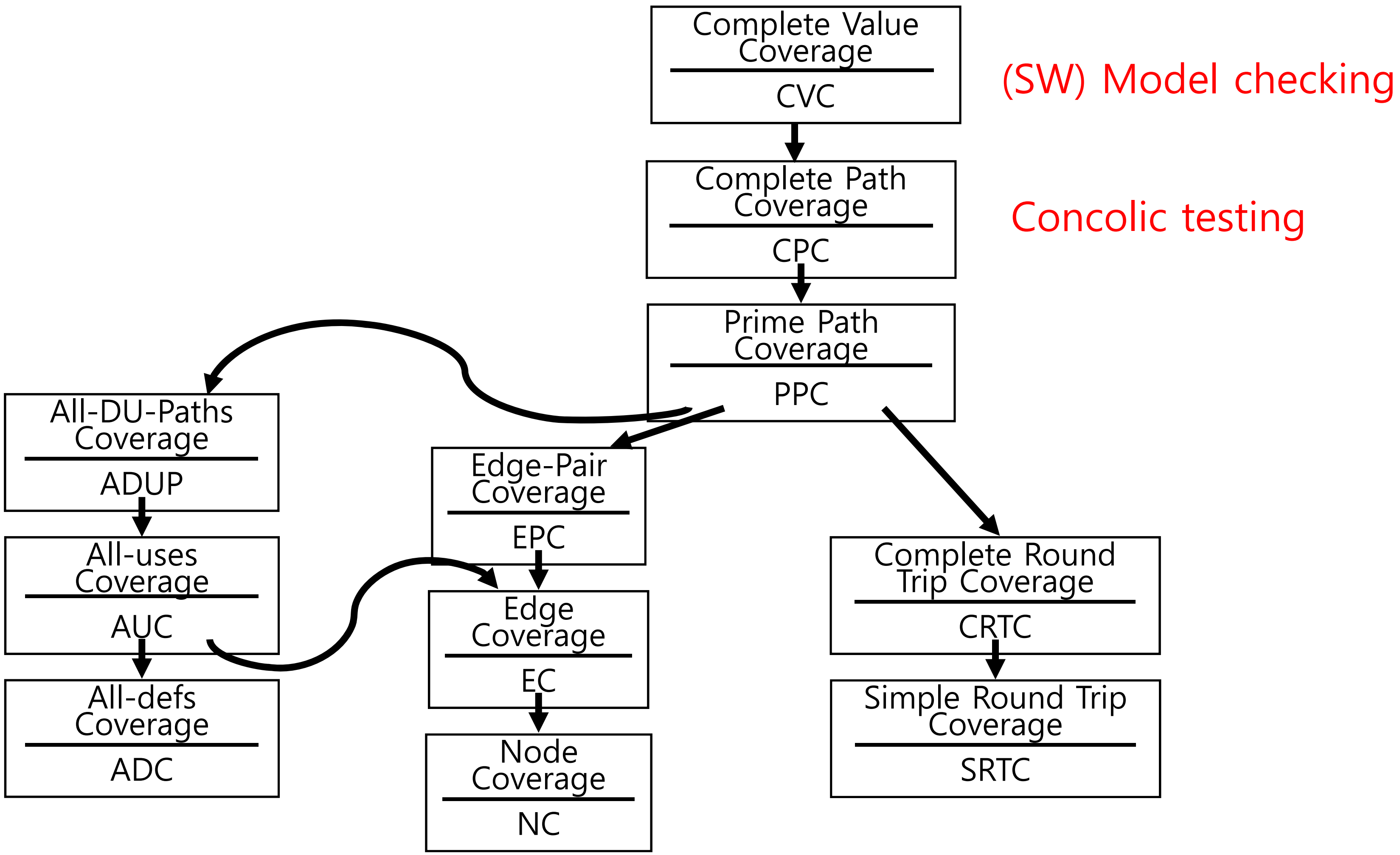
Automated Software Analysis Techniques

- Aims to explore possible behaviors of target systems in an exhaustive manner
- Key methods:
  - Represents a target program/or executions as a "logical formula"
  - Then, analyze the logical formula (a target program) by using logic analysis techniques

Weakness of conventional testing →

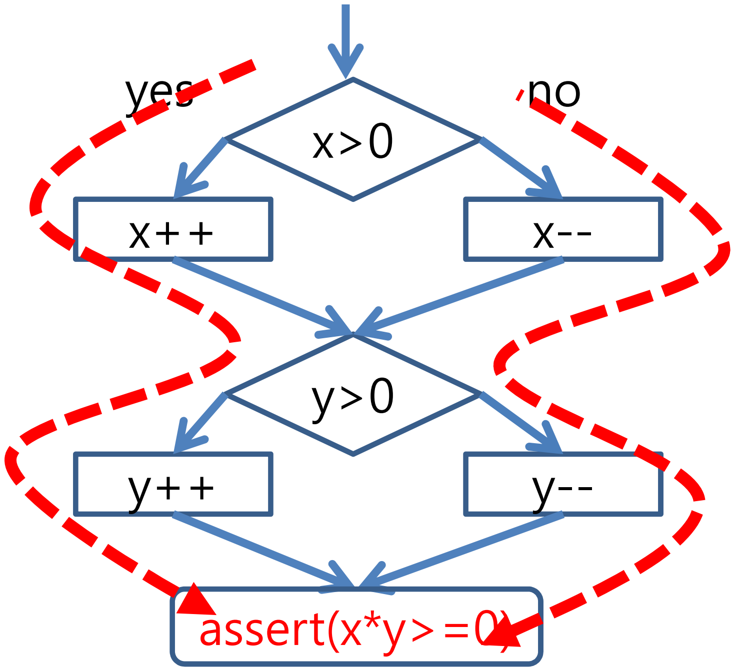
- Symbolic execution (1970)
- Model checking (1980)
- SW model checking (2000)
- Concolic testing (2005 ~)

Hierarchy of SW Coverages



Weaknesses of the Branch Coverage

```
/* TC1: x= 1, y= 1;
   TC2: x=-1, y=-1;*/
  void foo(int x, int y) {
    if ( x > 0)
        x++;
    else
        x--;
    if(y >0)
        y++;
    else
        y--;
    assert (x * y >= 0);
}
```



Systematic testing techniques are necessary for quality software!
-> Integration testing is not enough
-> Unit testing with automated test case generation is desirable for both productivity and quality

Dynamic v.s. Static Analysis

	Dynamic Analysis (i.e., testing)	Static Analysis (i.e. model checking)
Pros	•Real result •No environmental limitation •Binary library is ok	•Complete analysis result •Fully automatic •Concrete counter example
Cons	•Incomplete analysis result •Test case selection	•Consumed huge memory space •Takes huge time for verification •False alarms

-> Concolic testing

Concolic Approach

- Combine concrete and symbolic execution
  - **Conc**rete + Symb**olic** = Concolic
- In a nutshell, concrete execution over a concrete input guides symbolic execution
  - No false positives
- Automated testing of real-world C Programs
  - Execute target program on automatically generated test inputs
  - All possible execution paths are to be explored
  - Higher branch coverage than random testing

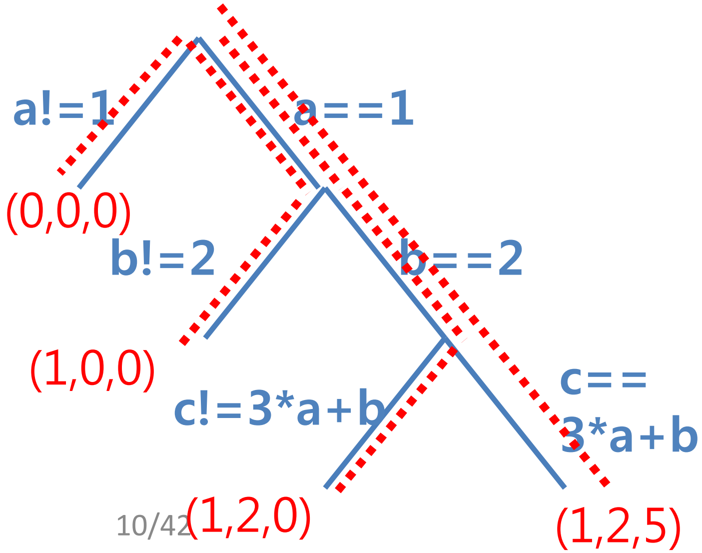
Overview of Concolic Testing Process

Itera-
tions

1. Select input variables to be handled symbolically
2. A target C program is statically instrumented with probes, which record symbolic path conditions
3. The instrumented C program is executed with given input values
   - Initial input values are assigned randomly
4. Obtain a symbolic path formula φ_i from a concrete execution over a concrete input
5. One branch condition of φ_i is negated to generate the next symbolic path formula ψ_i
6. A constraint solver solves ψ_i to get next concrete input values
   - Ex. φ_i : (x < 2) && (x + y >= 2) and ψ_i : (x < 2) && (x + y < 2). One solution is x=1 and y=0
7. Repeat step 3 until all feasible execution paths are explored

Concolic Testing Example

```
// Test input a, b, c
void f(int a, int b, int c) {
   if (a == 1) {
     if (b == 2) {
        if (c == 3*a + b) {
           Error();
} } } }
```



- Random testing
  - Probability of reaching Error( ) is extremely low
- Concolic testing generates the following 4 test cases
  - (0,0,0): initial random input
    - Obtained symbolic path formula (SPF) ϕ: a!=1
    - Next SPF ψ generated from ϕ: !(a!=1)
  - (1,0,0): a solution of ψ (i.e. !(a!=1))
    - SPF ϕ: a==1 && b!=2
    - Next SPF ψ: a==1 && !(b!=2)
  - (1,2,0)
    - SPF ϕ: a==1 && (b==2) && (c!=3*a +b)
    - Next SPF ψ: a==1 && (b==2) && !(c!=3*a +b)
  - (1,2,5)
    - Covered all paths and Error() reached

Example

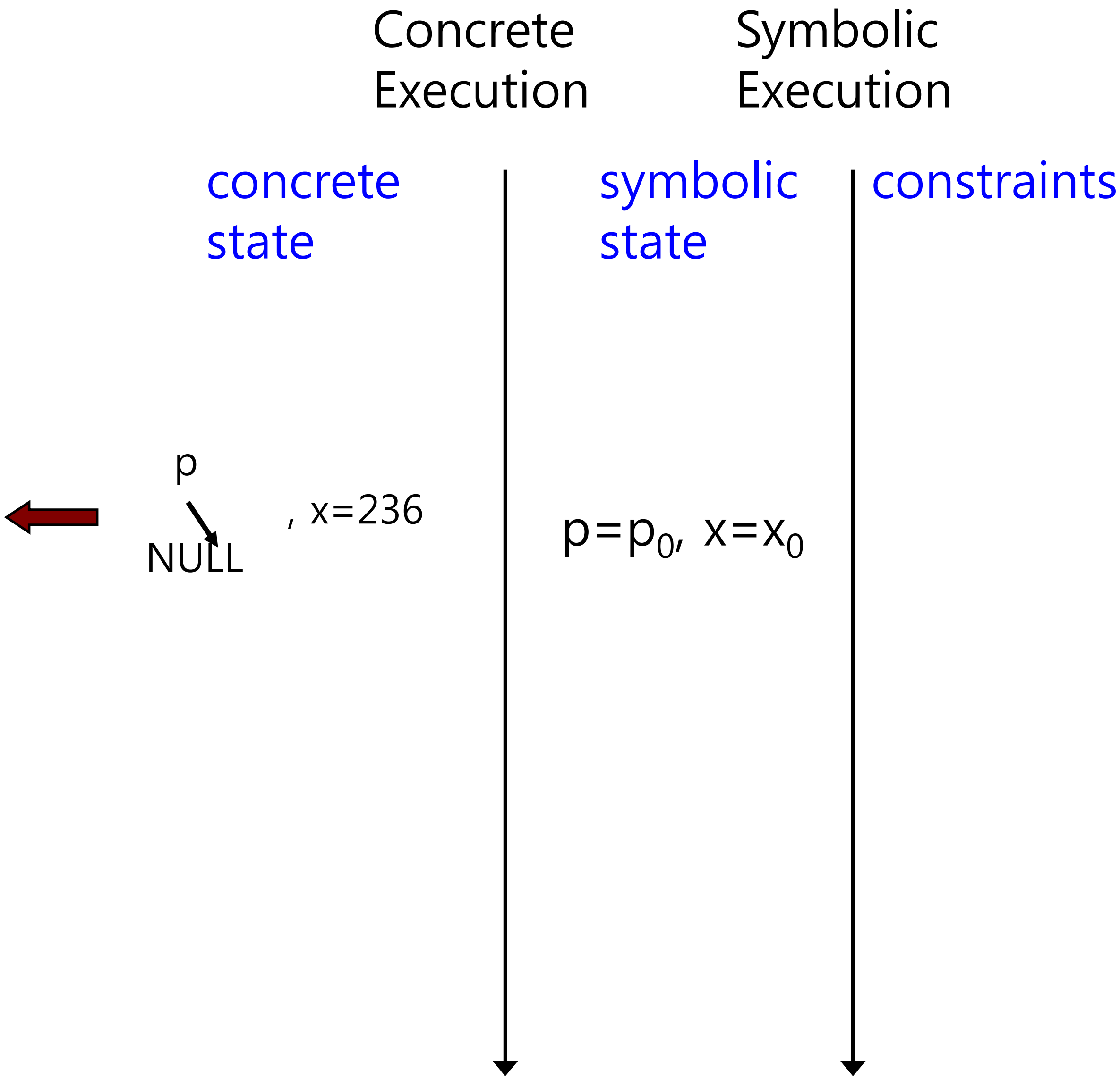
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          Error();
  return 0;
}
```

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

Example from the slides "CUTE: A Concolic Unit Testing Engine for C" by K.Sen 2005

Concolic Testing

```
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)
          Error();
  return 0;
}
```



Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
p → NULL , x=236		$p=p_0, x=x_0$	

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	constraints
p → NULL, x=236	$p=p_0, x=x_0$	$x_0>0$

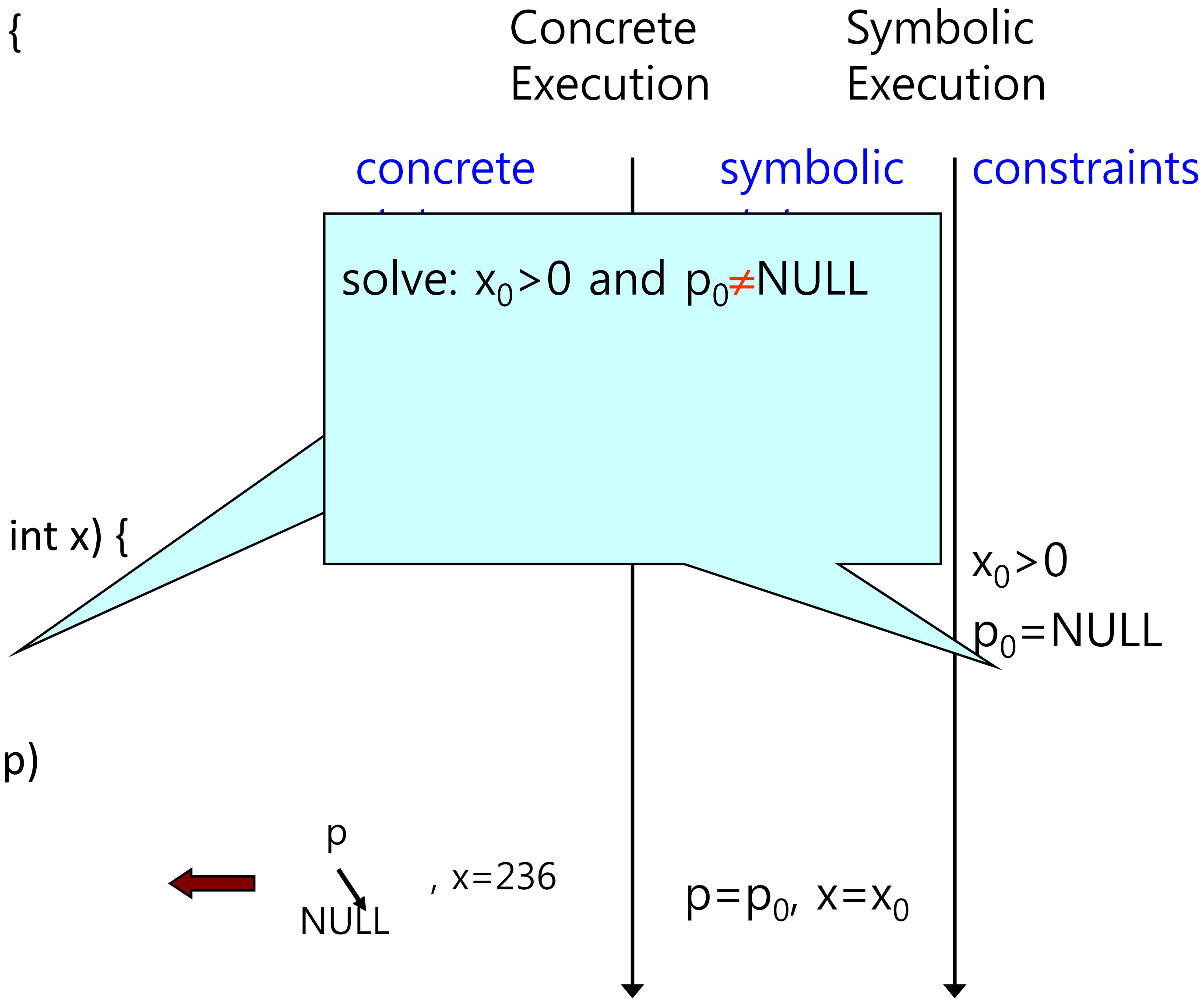
Concolic Testing

```
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)
          Error();
  return 0;
}
```

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	constraints
	p → NULL , x=236		$p=p_0, x=x_0$	$x_0>0$
				$!(p_0!=NULL)$

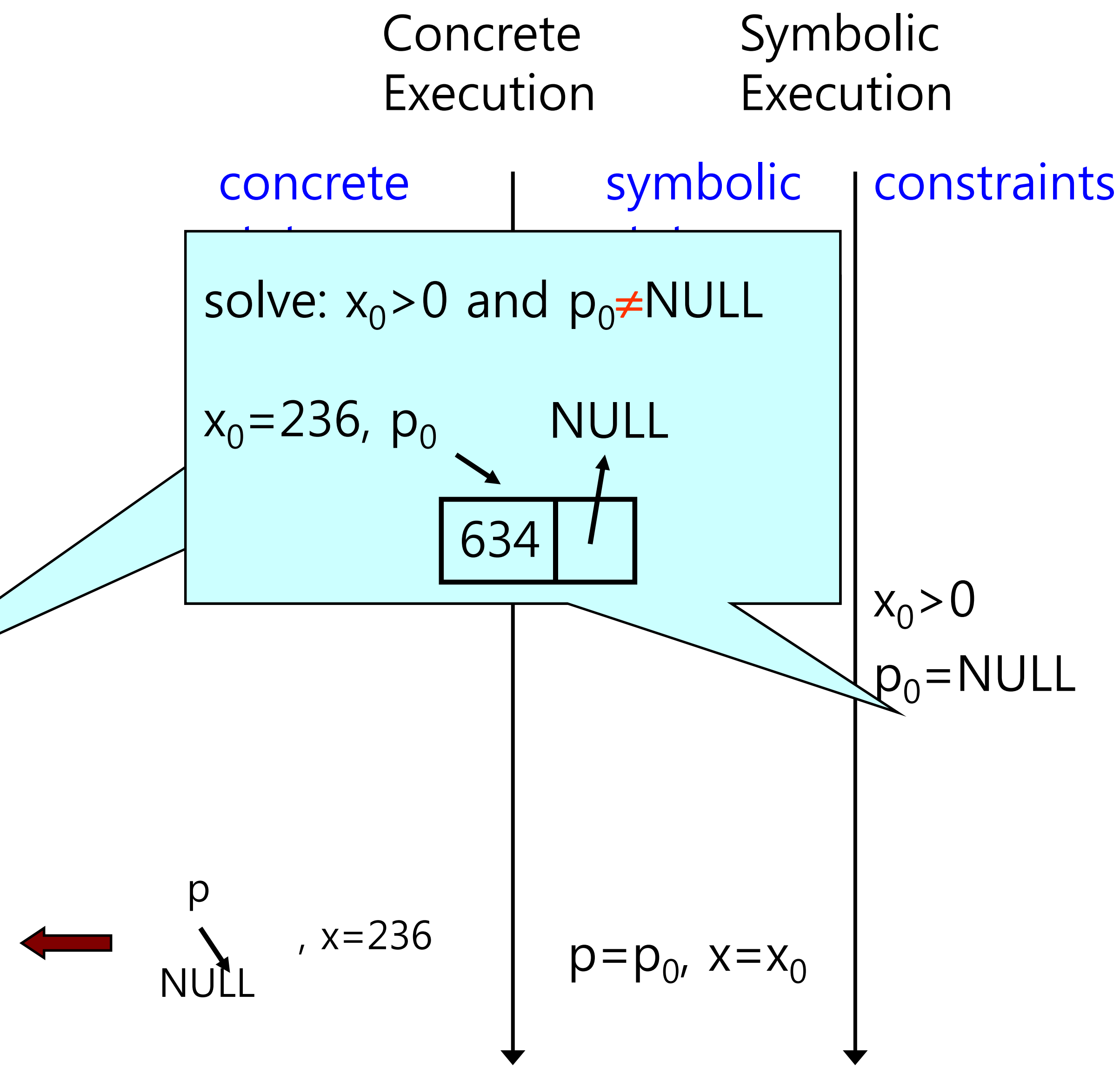
Concolic Testing

```
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)
          Error();
  return 0;
}
```



Concolic Testing

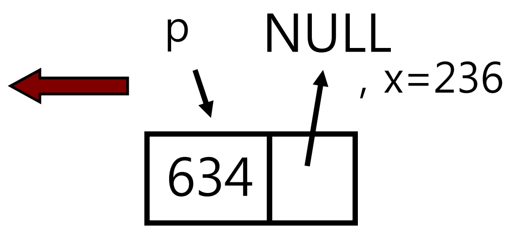
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          Error();
  return 0;
}
```



Concolic Testing

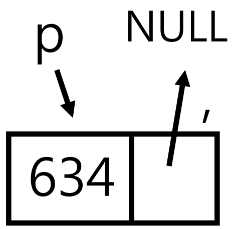
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          Error();
  return 0;
}
```

Concrete Execution | Symbolic Execution

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=236	$p=p_0$, $x=x_0$, $p\text{->}v=v_0$, $p\text{->}next=n_0$	$x_0>0$

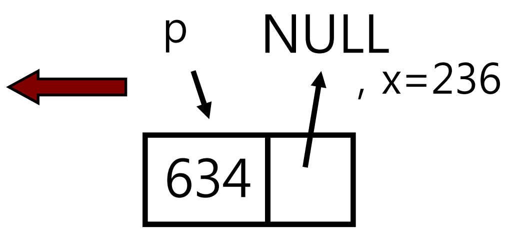
Concolic Testing

```
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)
          Error();
  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

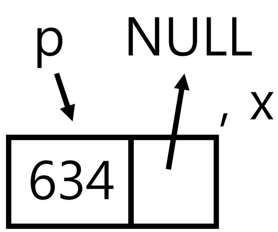
$x_0>0$

$p_0 \neq NULL$

Concolic Testing

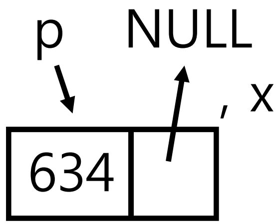
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          Error();
  return 0;
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	constraints
 , x=236	p=p_0, x=x_0, p->v =v_0, p->next=n_0	$x_0>0$ $p_0 \neq NULL$ $2x_0+1 \neq v_0$

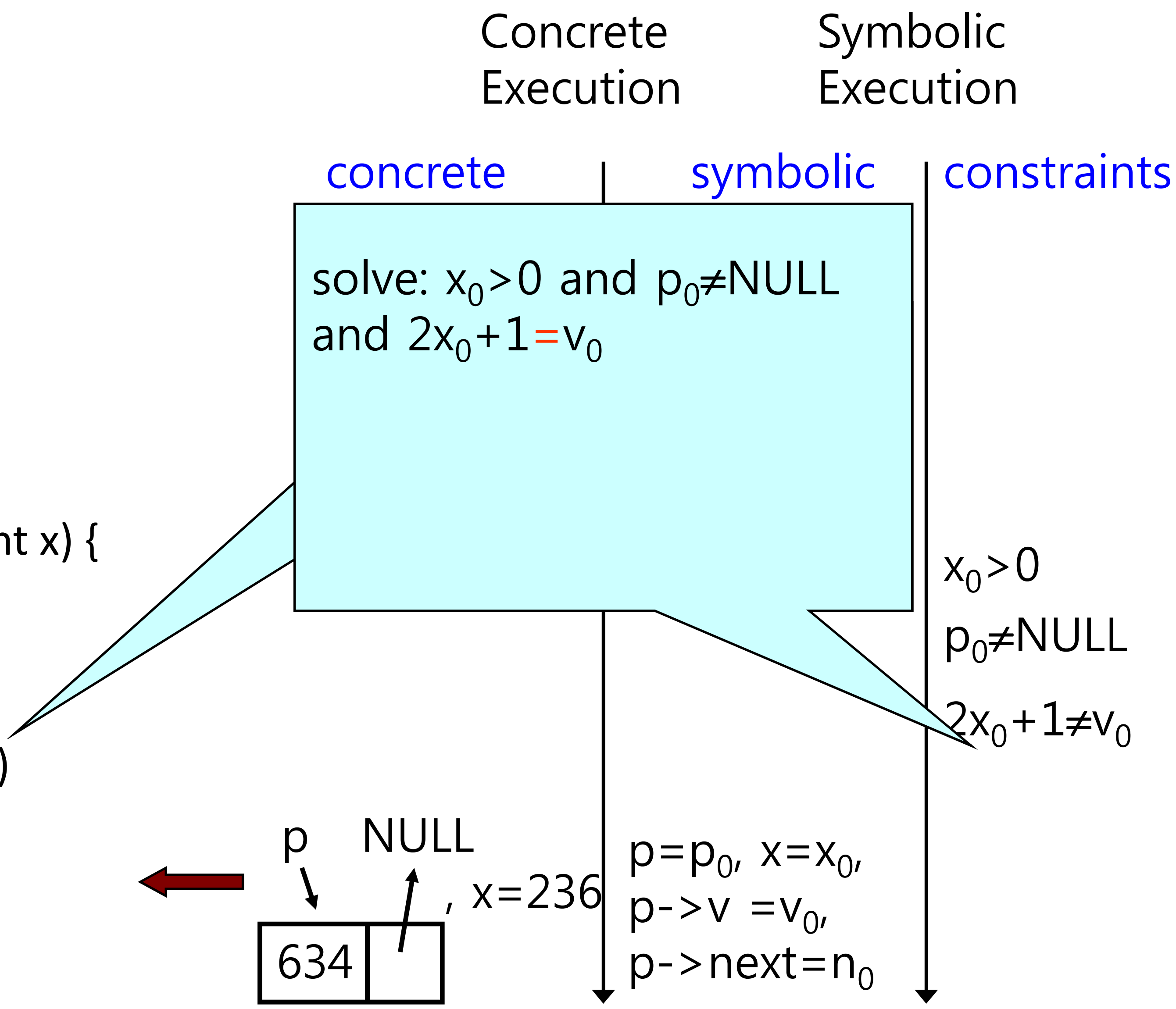
Concolic Testing

```
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)
          Error();
  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	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	

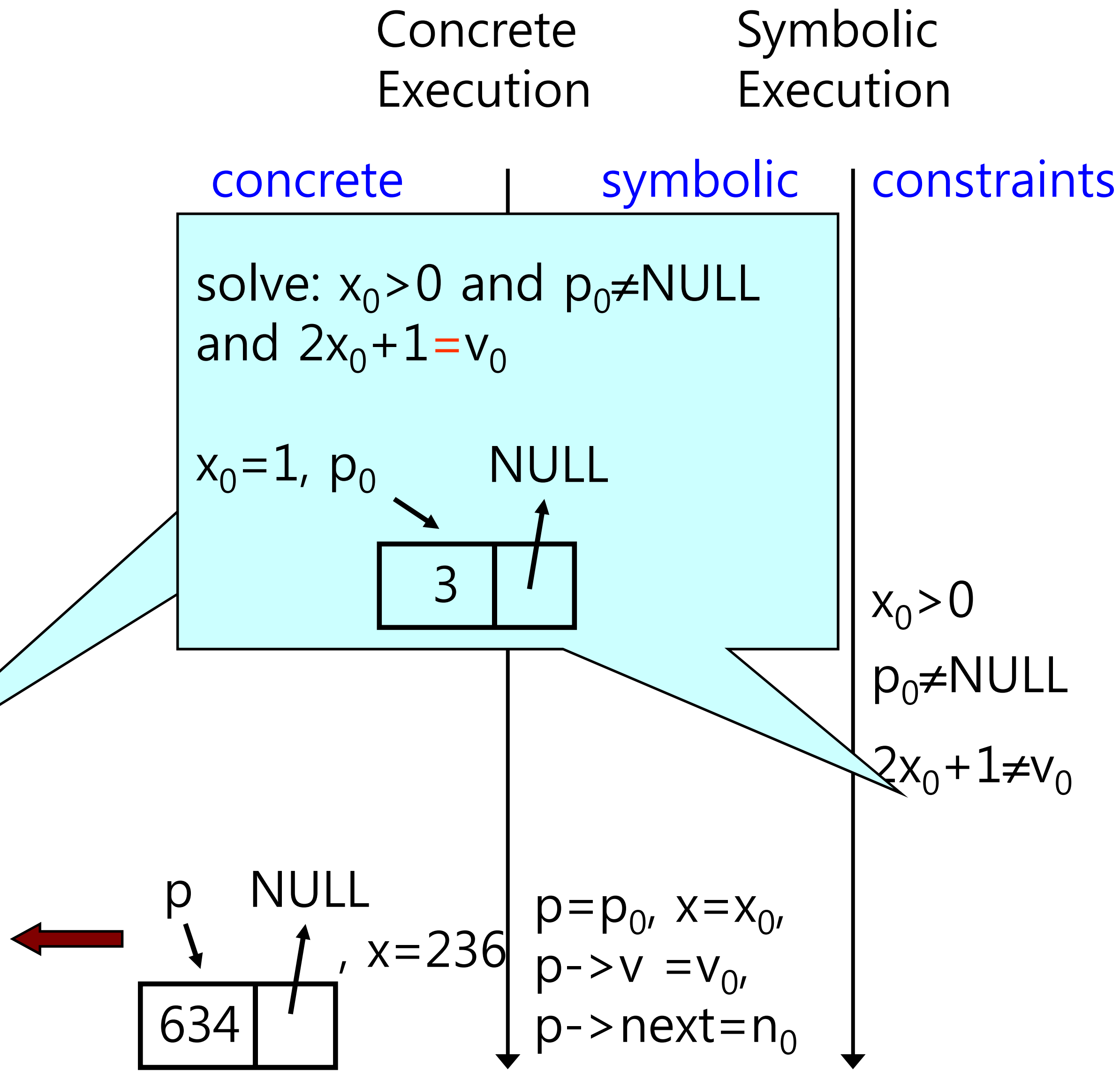
Concolic Testing

```
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)
          Error();
  return 0;
}
```



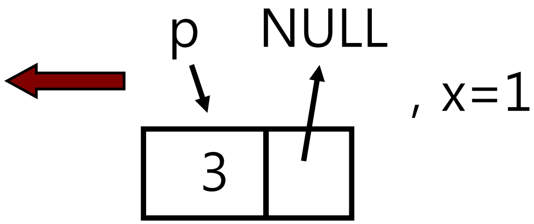
Concolic Testing

```
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)
          Error();
  return 0;
}
```



Concolic Testing

```
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)
          Error();
  return 0;
}
```

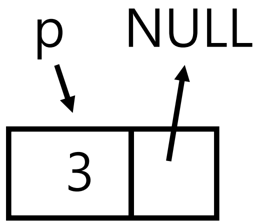
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next = n_0$	

Concolic Testing

```
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)
          Error();
  return 0;
}
```

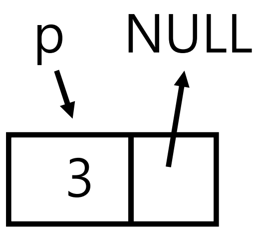
Concrete Execution

Symbolic Execution

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

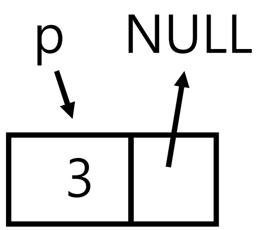
Concolic Testing

```
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)
          Error();
  return 0;
}
```

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

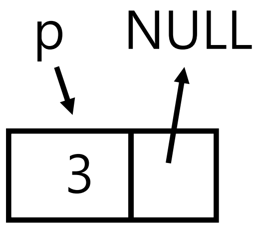
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , $x=1$	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1=v_0$

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution | Symbolic Execution

concrete state



, x=1

symbolic state

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

constraints

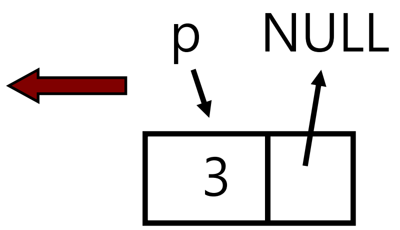
$x_0>0$
$p_0 \neq NULL$
$2x_0+1=v_0$
$n_0 \neq p_0$

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints

Concrete state:



, x=1

Symbolic state:

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

Constraints:

$x_0>0$

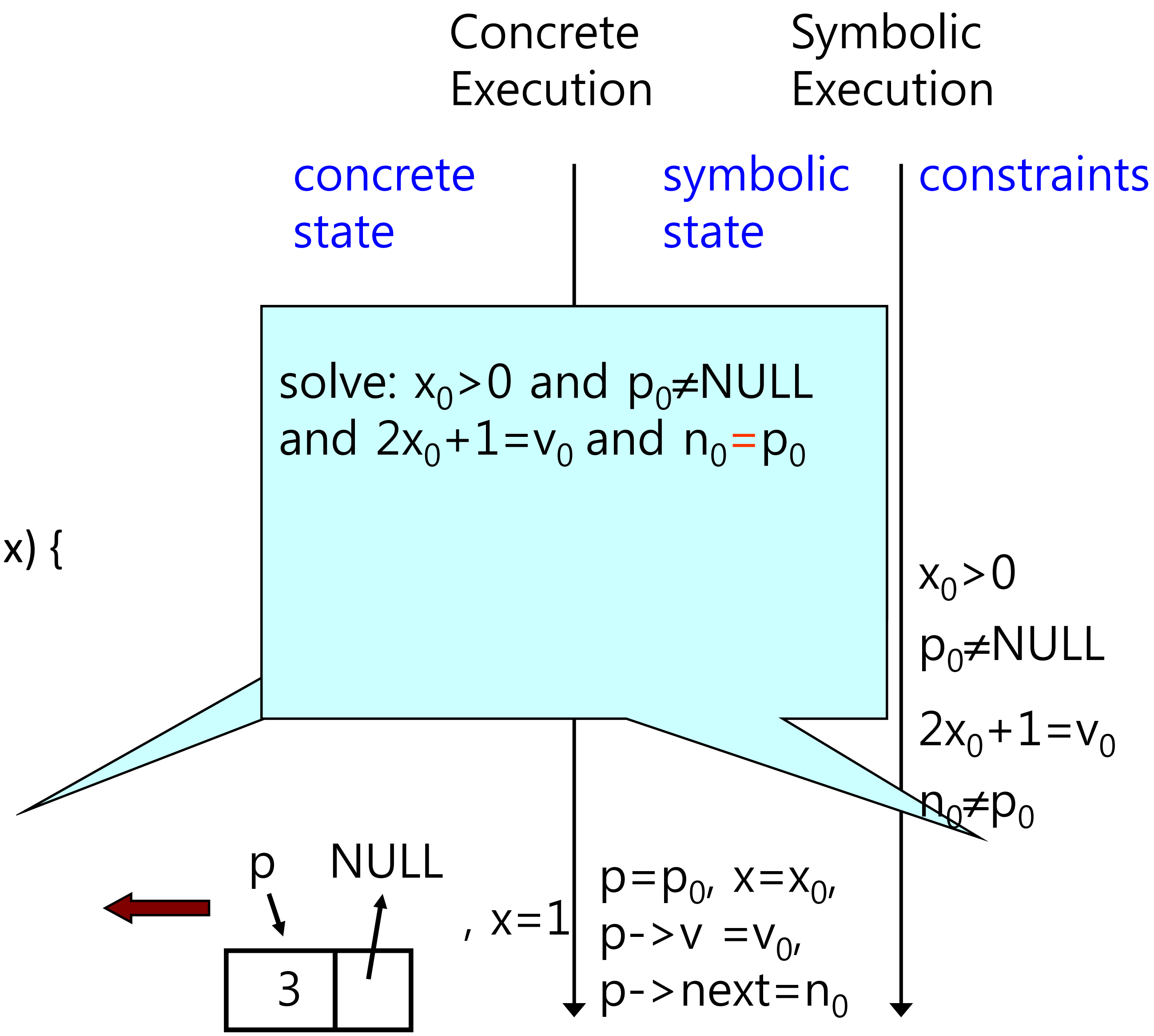
$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

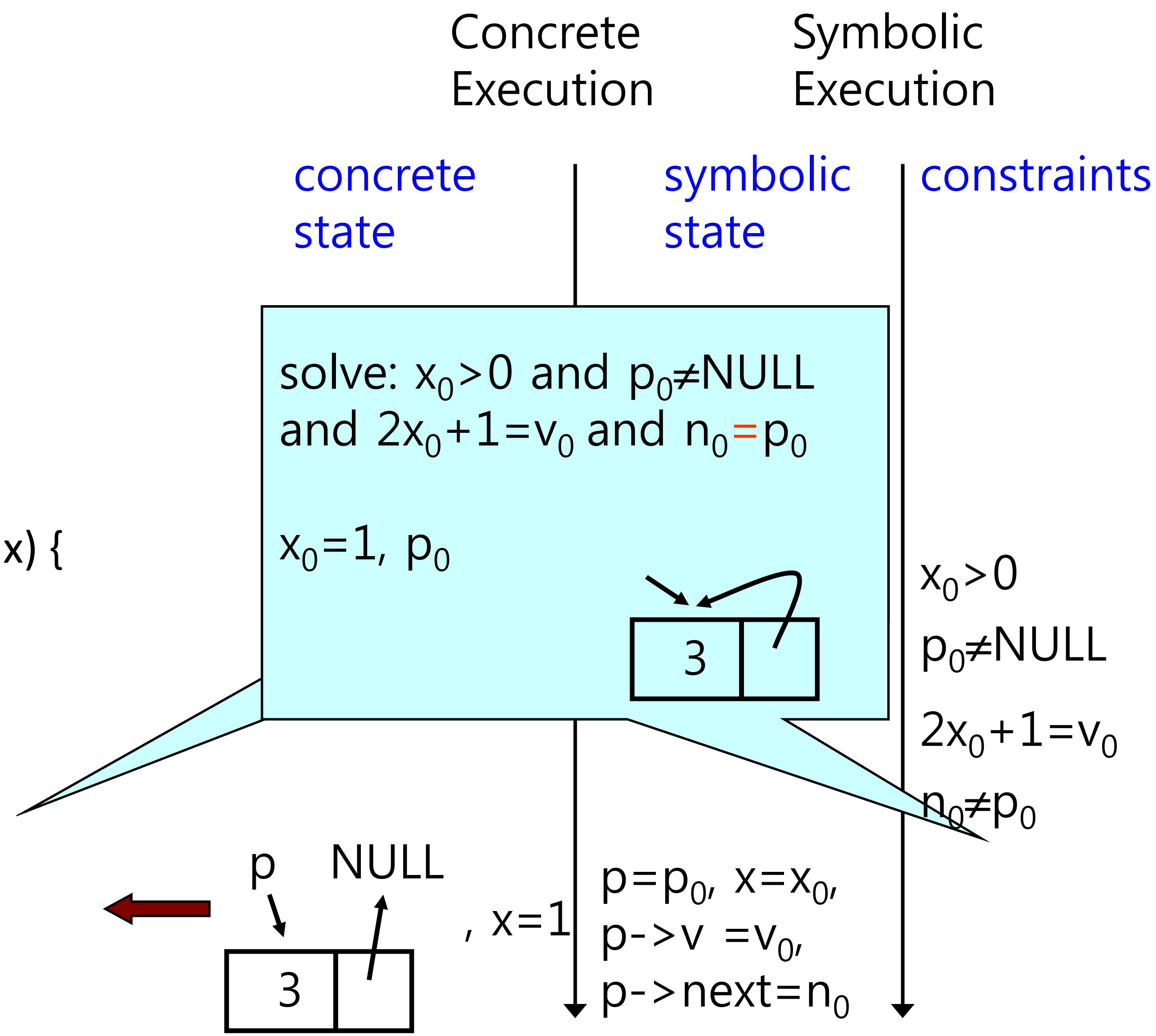
Concolic Testing

```
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)
          Error();
  return 0;
}
```



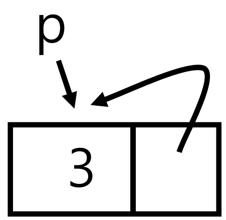
Concolic Testing

```
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)
          Error();
  return 0;
}
```



Concolic Testing

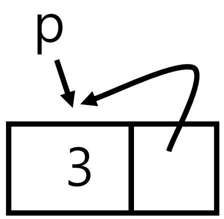
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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)
          Error();
  return 0;
}
```

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

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution | Symbolic Execution

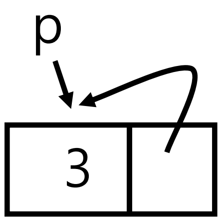
concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next = n_0$	$x_0>0$

Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution

concrete state



, x=1

Symbolic Execution

symbolic state

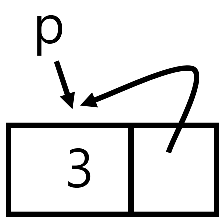
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

constraints

$x_0>0$
$p_0 \neq NULL$

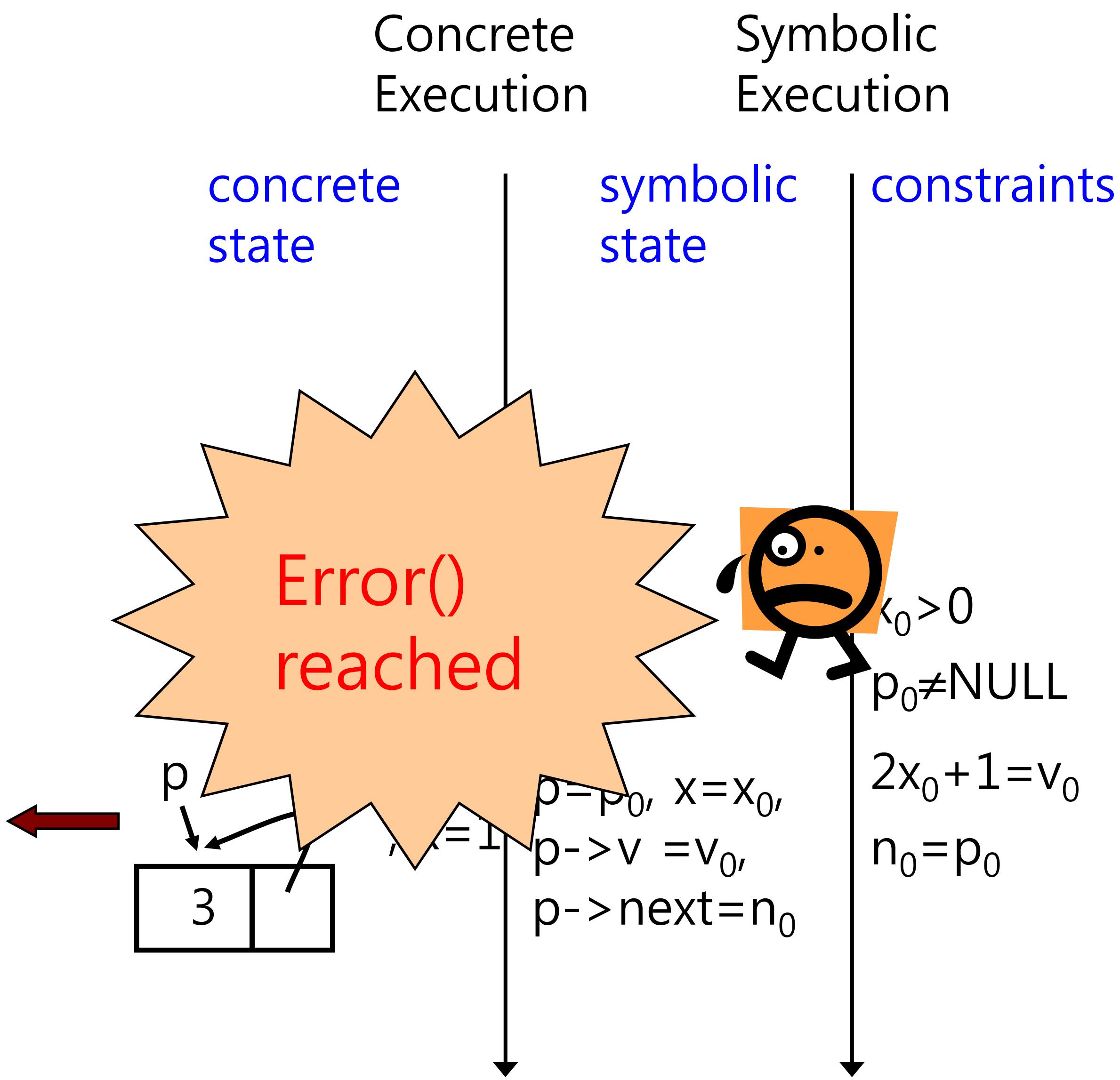
Concolic Testing

```
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)
          Error();
  return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
 	, x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next = n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1=v_0$

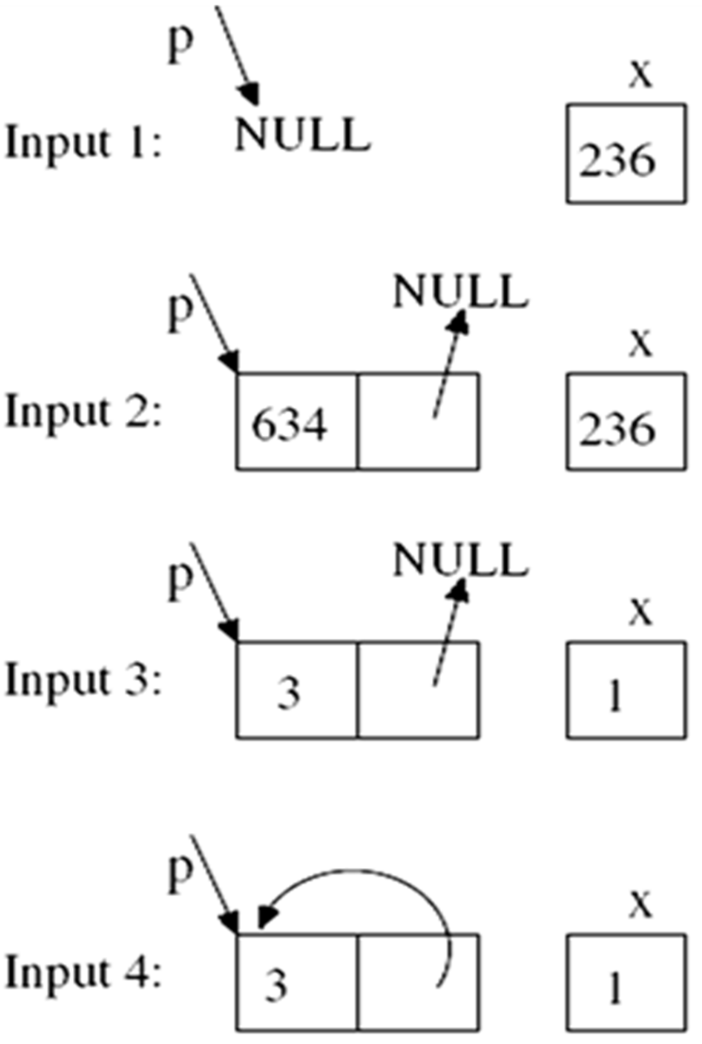
Concolic Testing

```
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)
          Error();
  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)
          Error();
  return 0;
}
```



Summary: Concolic Testing

- Pros
  - Automated test case generation
  - High coverage
  - High applicability (no restriction on target programs)
- Cons
  - If a target program has a complex statement, coverage might not be complete
    - Ex. if(sin(x) + cos(x) == 0.3) { error(); }
  - Current limitation on pointer and array
  - Slow analysis speed due to a large # of TCs

Case Study: Busybox

- We test a busybox by using CREST.
  - BusyBox is a one-in-all command-line utilities providing a fairly complete programming/debugging environment
  - It combines tiny versions of ~300 UNIX utilities into a sin gle small executable program suite.
  - Among those 300 utilities, we focused to test the followi ng 10 utilities
    - **`grep, vi, cut, expr, od , printf, tr, cp, ls, mv.`**
    - We selected these 10 utilities, because their behavior is easy to understand so that it is clear what variables should be declared as symbolic
    - Each utility generated 40,000 test cases for 4 different search strategies

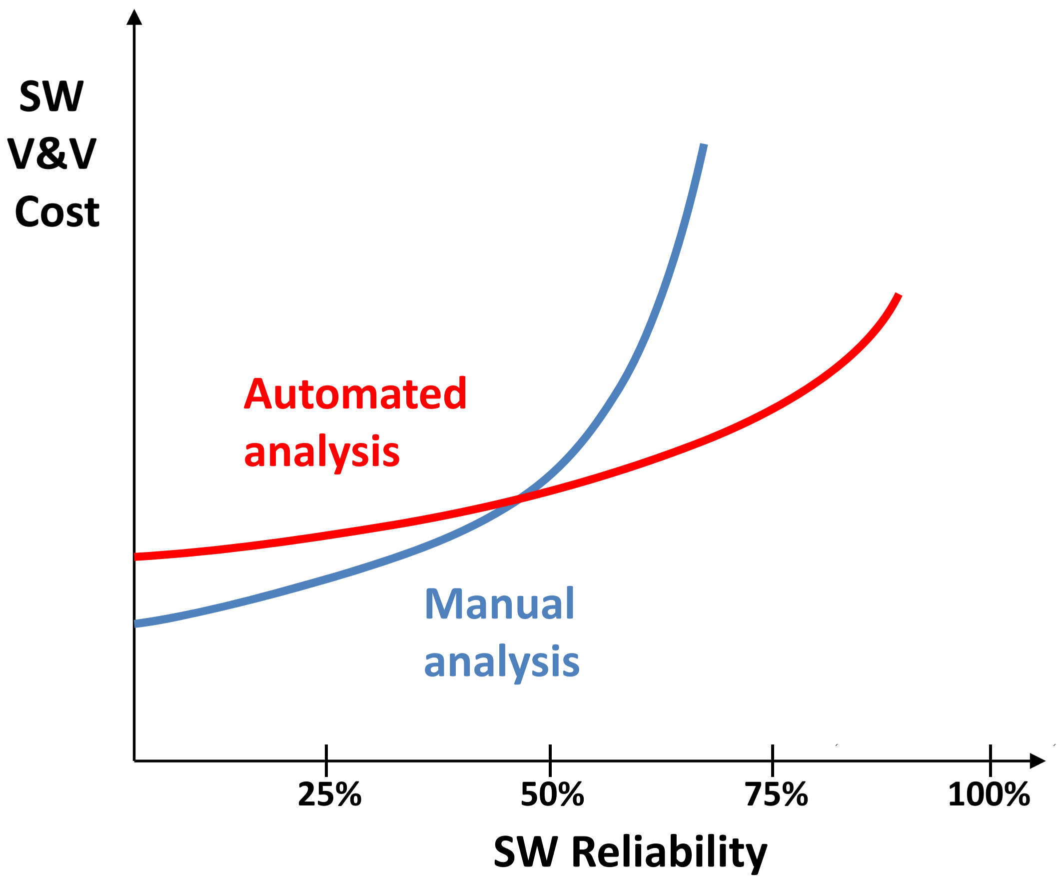
Busybox Testing Result

Utility	LOC	# of branches	DFS #of covered branch/time	CFG #of covered branch/time	Random #of covered branch/time	Random input #of covered branch/time	Merge of all 4 strategies #of covered branch/time
grep	914	178	105(59.0%)/2785s	85(47.8%)/56s	136(76.4%)/85s	50(28.1%)/45s	136(76.4%)
vi	4000	1498	855(57.1%)/1495s	965(64.4%)/1036s	1142(76.2)/723s	1019(68.0%)/463s	1238(82.6%)
cut	209	112	67(59.8%)/42s	60(53.6%)/45s	84(75.0%)/53s	48(42.9%)/45s	90(80.4%)
expr	501	154	104(67.5%)/58s	101(65.6%)/44s	105(68.1%)/50s	48(31.2%)/31s	108(70.1%)
od	222	74	59(79.7%)/35s	72(97.3%)/41s	66(89.2%)/42s	44(59.5%)/30s	72(97.3%)
printf	406	144	93(64.6%)/84s	109(75.7%)/41s	102(70.8%)/40s	77(53.5%)/30s	115(79.9%)
tr	328	140	67(47.9%)/58s	72(51.4%)/50s	72(51.4%)/50s	63(45%)/42s	73(52.1%)
cp	191	32	20(62.5%)/38s	20(62.5%)/38s	20(62.5%)/38s	17(53.1%)/30s	20(62.5%)
ls	1123	270	179(71.6%)/87s	162(64.8%)/111s	191(76.4%)/86s	131(52.4%)/105s	191(76.4%)
mv	135	56	24(42.9%)/0s	24(42.9%)/0s	24(42.9%)/0s	17(30.3%)/0s	24(47.9%)
AVG	**803**	**264**	**157.3(59.6%)/809s**	**167(63.3%)/146s**	**194.2(73.5%)/117s**	**151.4(57.4%)/83s**	**206.7(78.4%)/ 1155s**

Future Direction

- Tool support will be strengthened for automated SW analysis
  - Ex. CBMC, BLAST, CREST, KLEE, and Microsoft PEX
  - Automated SW analysis will be performed routinely like GCC
  - Labor-intensive SW analysis => automated SW analysis by few experts
- Supports for concurrency analysis
  - Deadlock/livelock detection
  - Data-race detection
- Less user input, more analysis result and less false alarm
  - Fully automatic C++ syntax & type check (1980s)
  - (semi) automatic null-pointer dereference check (2000s)
  - (semi) automatic user-given assertion check (2020s)
  - (semi) automatic debugging (2030s)

Conclusion: Manual Analysis v.s. Automated Analysis



- Traditional manual analysis is easy to apply for programs w/ low quality
- However, automated analysis can achieve high quality cost effectively
- Automated software analysis techniques are (almost) ready to be applied in industry