Research Issues in Concolic Testing

Moonzoo Kim
Provable Software Lab, CS Dept, KAIST

Contents

- Motivation for Automated Testing
  - Ex. Triangle example
- Concolic (CONCrete + symbOLIC) Testing Approach
  - Overview of the concolic testing framework
- Research Issues in Concolic Testing
  - To improve
    - Efficiency: by parallelized concolic testing
    - Effectiveness
      - By using stronger test requirements
      - By using existing test cases

Ex. Testing a Triangle Decision Program

Input : Read three integer values from the command line. The three values represent the length of the sides of a triangle.

Output : Tell whether the triangle is

- 부등변삼각형 (Scalene) : no two sides are equal
- 이등변삼각형(Isosceles) : exactly two sides are equal
- 정삼각형 (Equilateral) : all sides are equal

Create a Set of Test Cases for this program

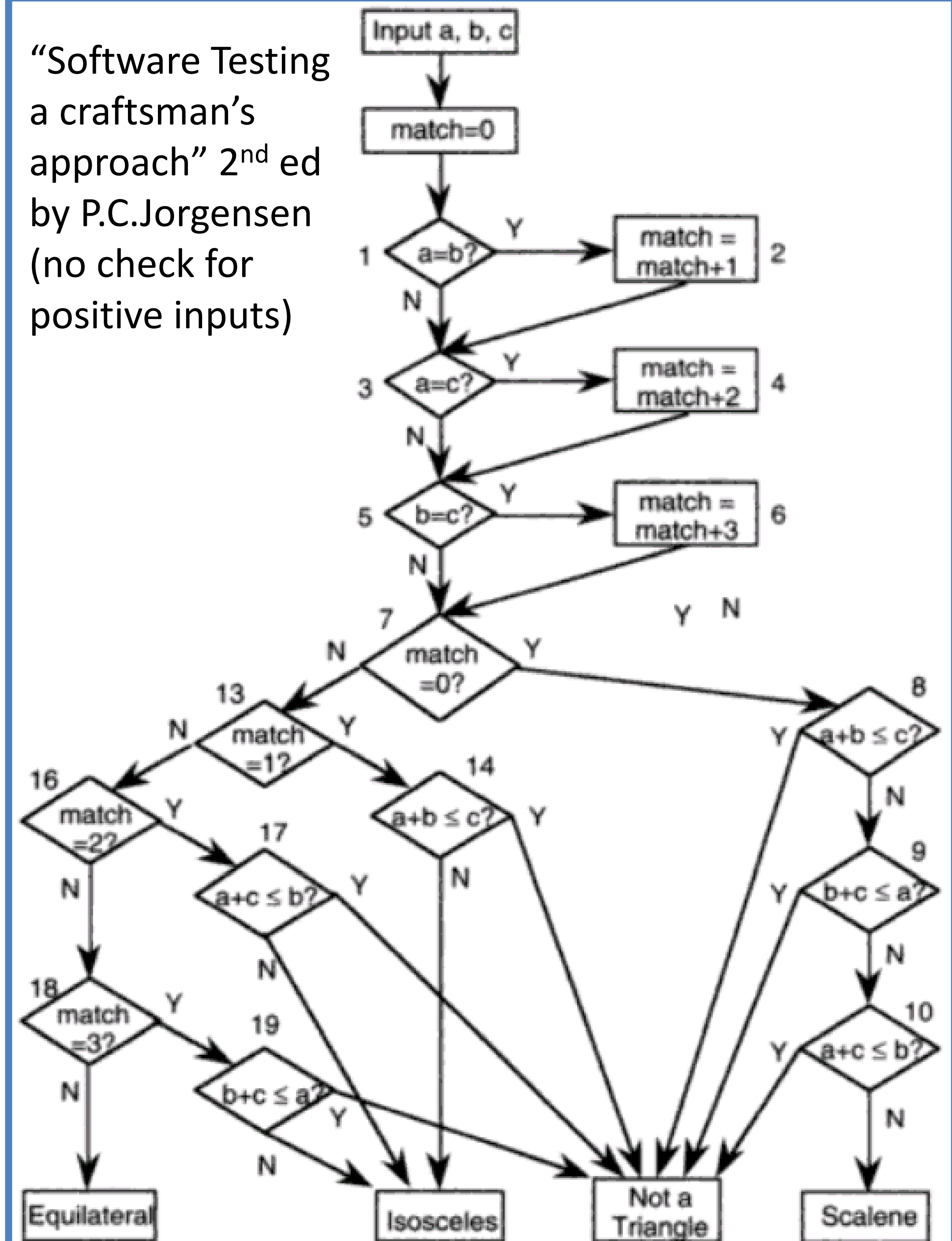
(3,4,5), (2,2,1), (1,1,1) ?

Moonzoo Kim
Provable SW Lab
KAIST

Ex. Precondition (Input Validity) Check

- Condition 1: $a > 0$, $b > 0$, $c > 0$
- Condition 2: $a < b + c$
  - Ex. (4, 2, 1) is an invalid triangle
  - Permutation of the above condition
    - $a < b + c$
    - $b < a + c$
    - $c < a + b$
- What if b + c exceeds 2^{32} (i.e. overflow)?
  - long v.s. int v.s. short. v.s. char
- Refinements:
  - What if “integer value” is relaxed to “floating value” ?
    - Round-off errors should be handled explicitly
- Developers often fail to consider implicit conditions
  - Cause of many hard-to-find bugs

- # of test cases required?
  - ① 4
  - ② 10
  - ③ 50
  - ④ 100
- # of feasible unique execution paths?
  - 11 paths
  - guess what test cases needed



Concolic Testing Framework

- A combined approach of
  - Dynamic concrete analysis
  - Static symbolic analysis
- Automated Unit Testing of real-world C Programs
  - Execute a unit under test on automatically generated test inputs so that all possible execution paths are explored
- In a nutshell
  1. Use a concrete execution over a concrete input to obtain a symbolic execution path formula α_i
  2. One branch condition of α_i is negated to generate the next symbolic execution path formula α_{i+1}
  3. A constraint solver gets concrete input values to satisfy α_{i+1}
     - Ex. α_{i+1} : (x < 2) && (2x + 3y < 7). One solution is x=1 and y=1
  4. Repeat step 1 until all feasible execution paths are explored

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;
}
```

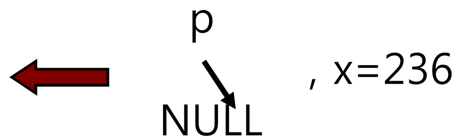
- Random Test Driver:
  - random memory graph reachable from p
  - random value for x
- Probability of reaching Error( ) is extremely low

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;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints



, x=236

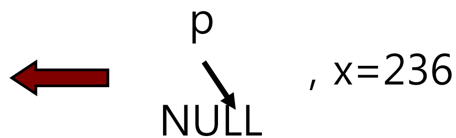
$p=p_0, x=x_0$

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



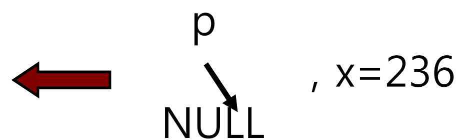
, x=236

$p=p_0, x=x_0$

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;
}
```

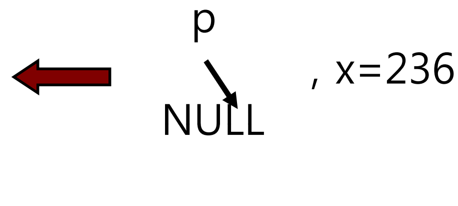
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
		$x_0>0$
p → NULL, x=236	$p=p_0$, $x=x_0$	



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
		$x_0>0$
		$!(p_0!=NULL)$
, x=236	$p=p_0, x=x_0$	



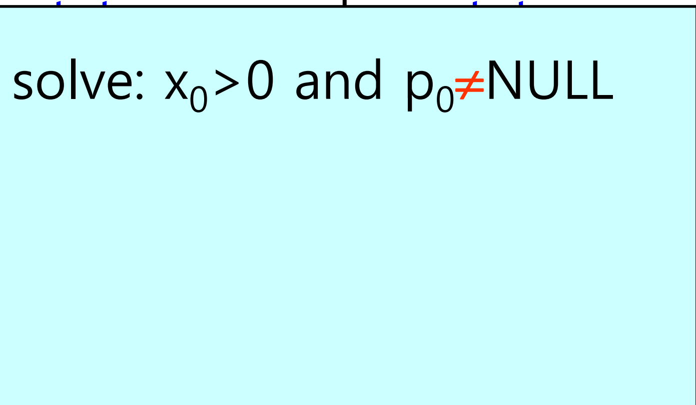
Concolic Testing

```
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

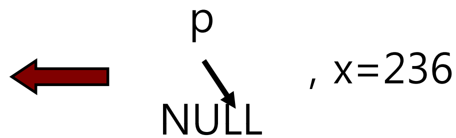
Symbolic Execution

concrete | symbolic | constraints



$x_0>0$

$p_0=NULL$



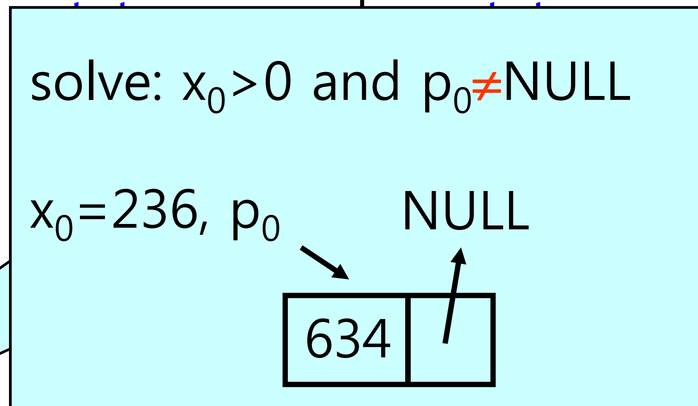
$p=p_0, x=x_0$

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;
}
```

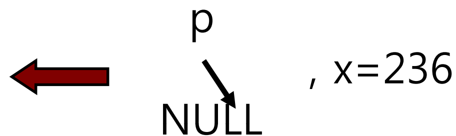
Concrete Execution | Symbolic Execution

concrete | symbolic | constraints



$x_0>0$

p_0=NULL

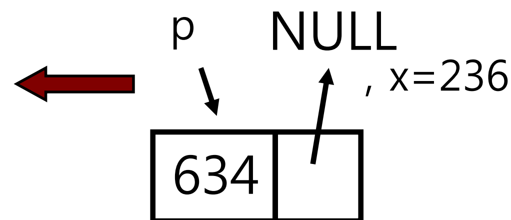


$p=p_0$, $x=x_0$

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;
}
```

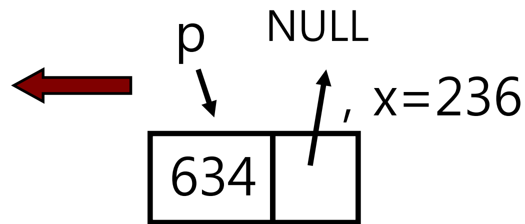
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints



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

Concolic Testing

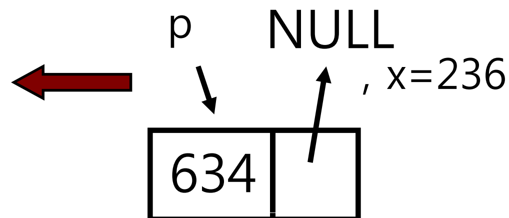
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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	constraints



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

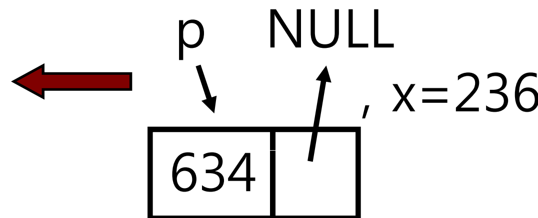
constraints: $x_0>0$, $p_0 \neq NULL$

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



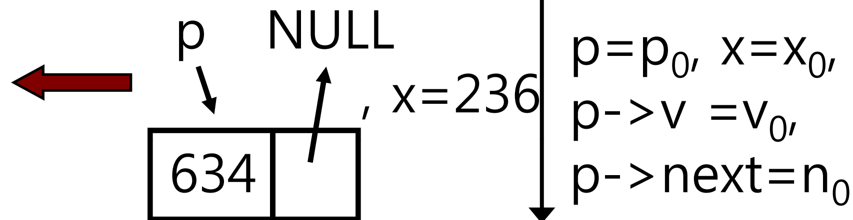
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

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

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;
}
```

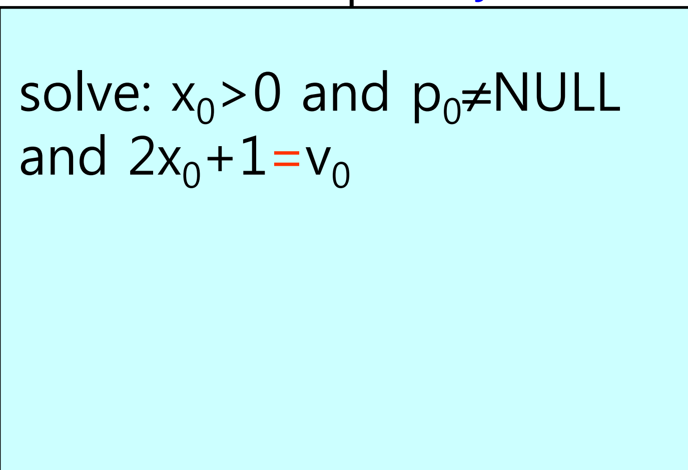
Concrete Execution | Symbolic Execution

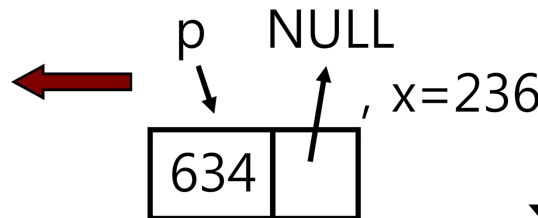
concrete state	symbolic state	constraints
		$x_0>0$
		$p_0 \neq NULL$
		$2x_0+1 \neq v_0$
	$p=p_0$, $x=x_0$, $p\text{->}v=v_0$, $p\text{->}next=n_0$	

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





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

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

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;
}
```

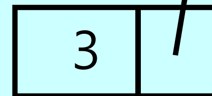
Concrete Execution

Symbolic Execution

concrete | symbolic | constraints

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

$x_0=1$, p_0 NULL

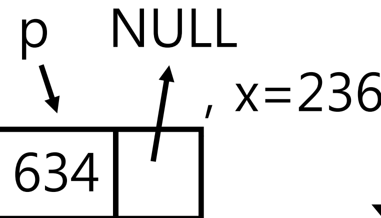


$x_0>0$

$p_0\neq$NULL

$2x_0+1\neq v_0$

p NULL



, x=236

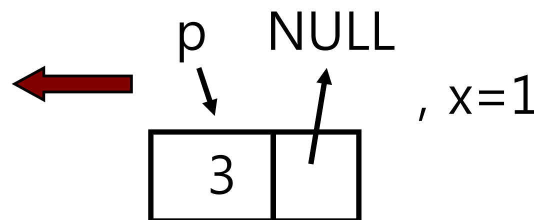
$p=p_0$, $x=x_0$,
$p\text{->}v = v_0$,
$p\text{->}next=n_0$

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;
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

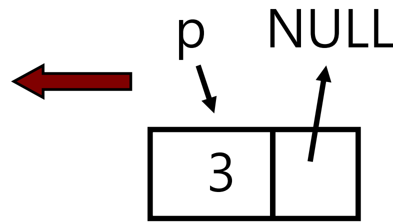


, x=1

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

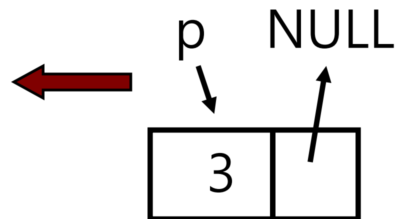
Concolic Testing

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

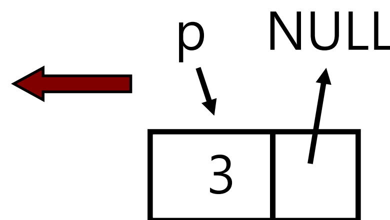
Concolic Testing

```
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		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;
}
```

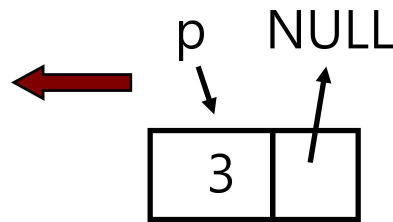
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=1	$p=p_0$, $x=x_0$, $p\text{->}v = v_0$, $p\text{->}next=n_0$	$x_0>0$ $p_0 \neq NULL$ $2x_0+1=v_0$

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;
}
```

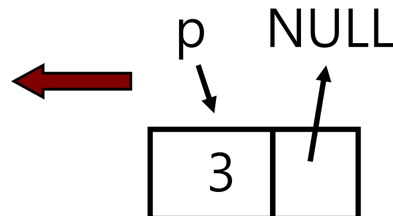
Concrete Execution

Symbolic Execution

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

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

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

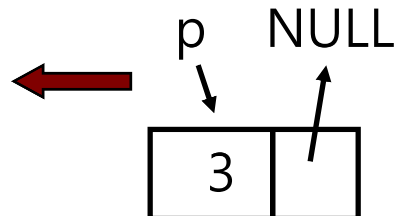
solve: $x_0>0$ and $p_0 \neq NULL$ and $2x_0+1=v_0$ and $n_0=p_0$

$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

$n_0 \neq p_0$

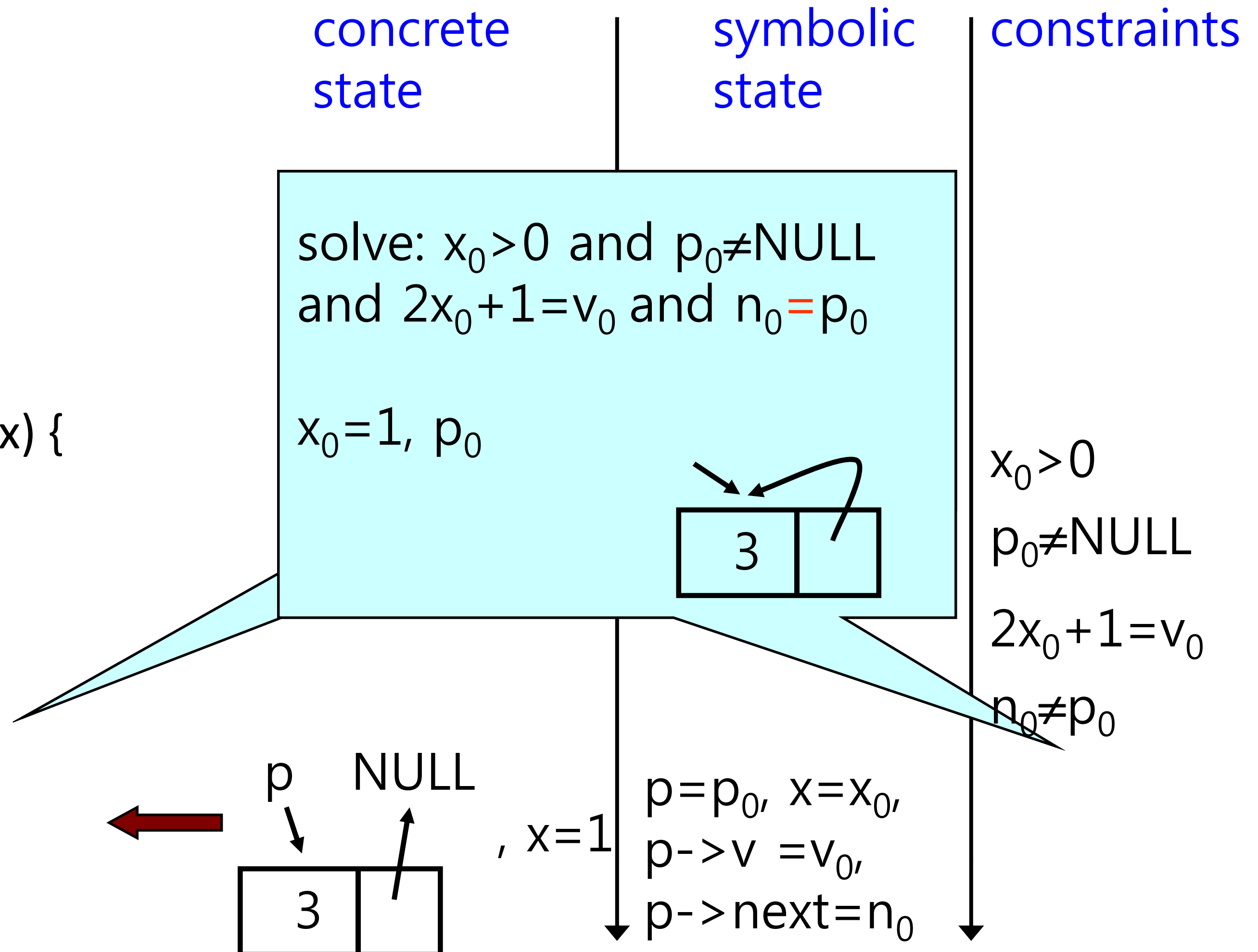


, x=1

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

Concolic Testing

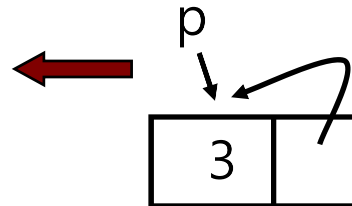
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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;
}
```



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

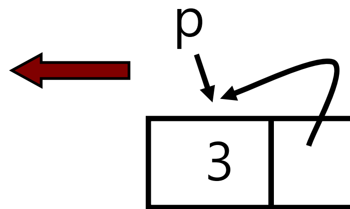


, x=1

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

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;
}
```

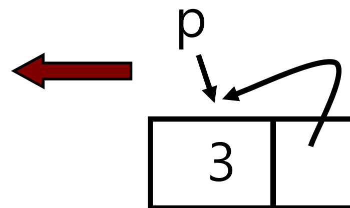
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	constraints
 , x=1	p=p_0, x=x_0, p->v =v_0, p->next=n_0	x_0>0

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

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

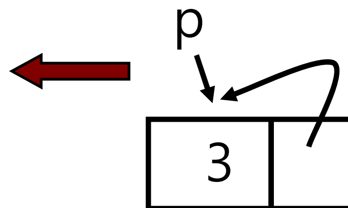
$x_0>0$
$p_0 \neq NULL$

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

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

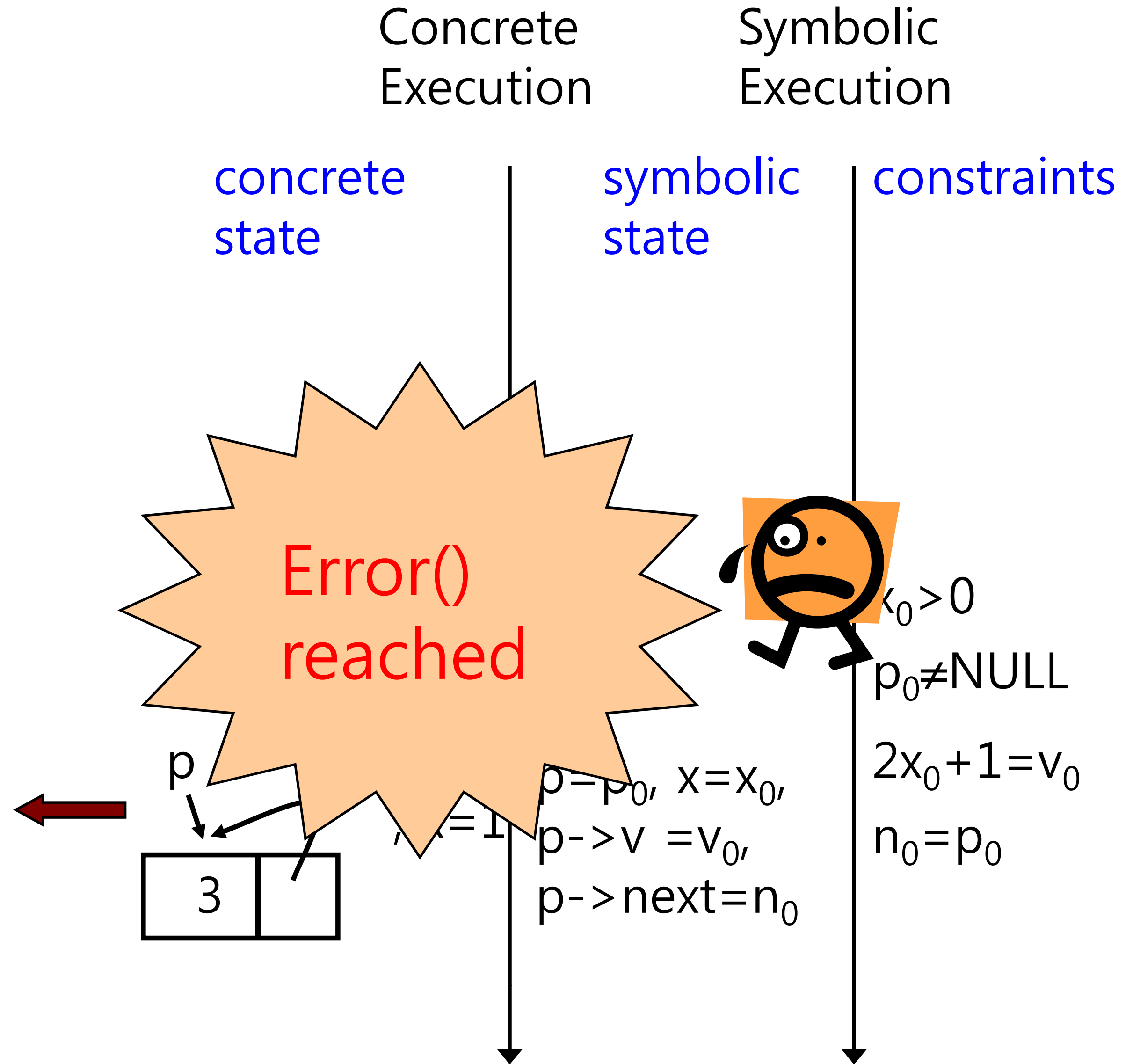
$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

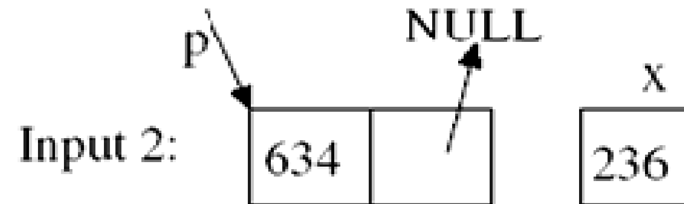
Concolic Testing

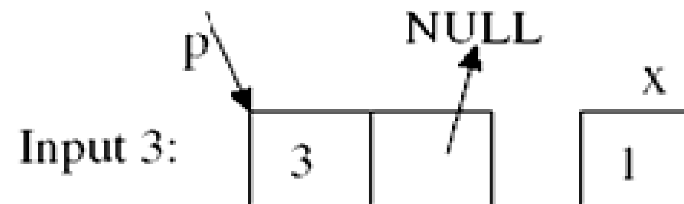
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

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;
}
```

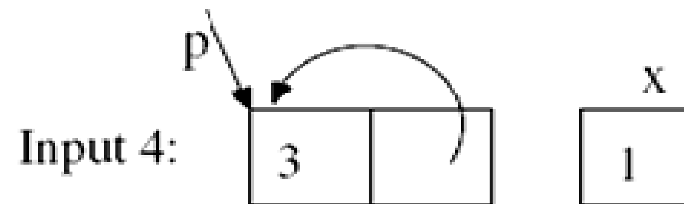


Pointer Inputs: Input Graph

```
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;
}
```





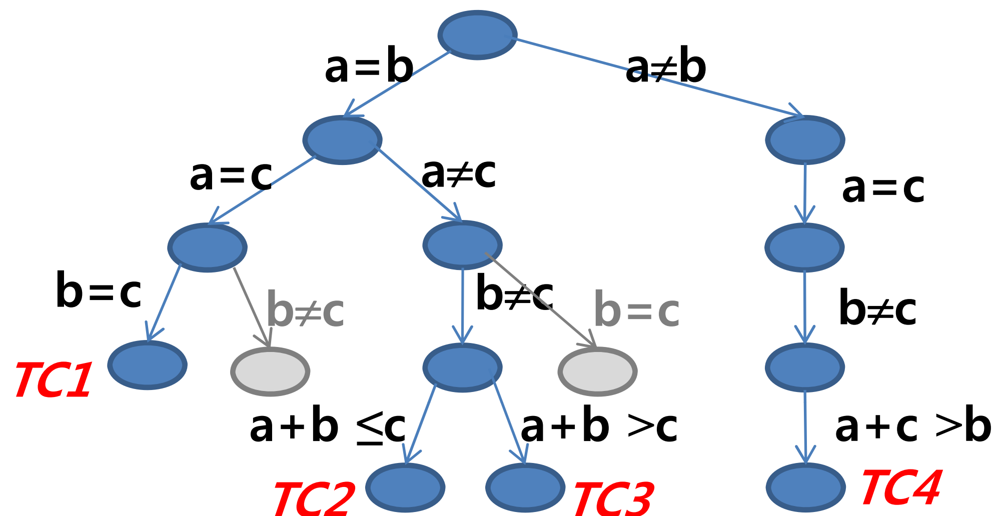


Observation 1

- Concolic testing is slow (efficiency problem)
  - It generates billions of test cases, since it is explicit path model checking
  - Each path formula can be very long
  - Solving path formula for concrete next inputs is slow
- Suggested solution
  - To parallelize multiple concolic testing executions
    - Starting with multiple random inputs
    - Multiple concolic executions in parallel by negating multiple conditions in a path formula

Example. Triangle Program

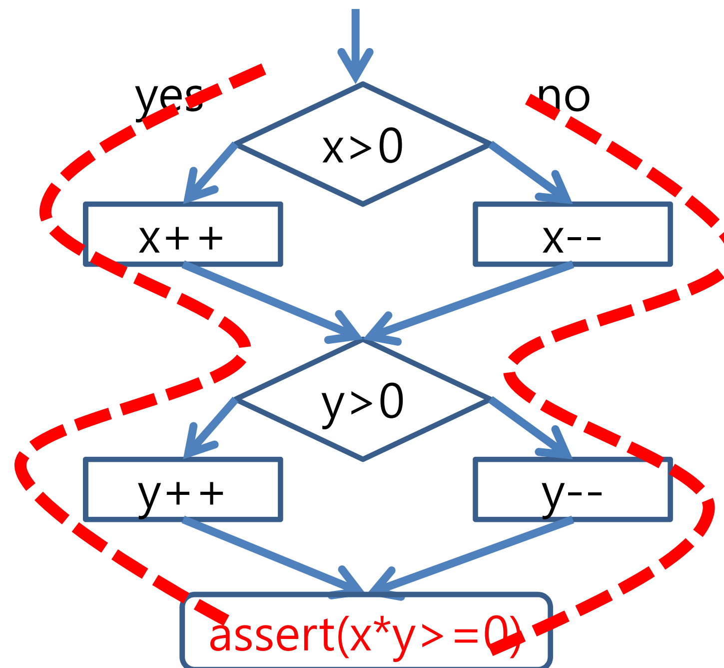
Test case	Input (a,b,c)	Executed path conditions (PC)	Next PC	Solution for the next PC
1	1,1,1	$a=b \wedge a=c \wedge b=c$	$a=b \wedge a=c \wedge b\neq c$	Unsat
			$a=b \wedge a\neq c$	1,1,2
2	1,1,2	$a=b \wedge a\neq c \wedge b\neq c \wedge a+b \leq c$	$a=b \wedge a\neq c \wedge b\neq c \wedge a+b > c$	2,2,3
3	2,2,3	$a=b \wedge a\neq c \wedge b\neq c \wedge a+b > c$	$a=b \wedge a\neq c \wedge b=c$	Unsat
			$a\neq b$	2,1,2
4	2,1,2	$a\neq b \wedge a=c \wedge b\neq c \wedge a+c>b$	$a\neq b\wedge a=c \wedge b\neq c \wedge a+c \leq b$	2,5,2



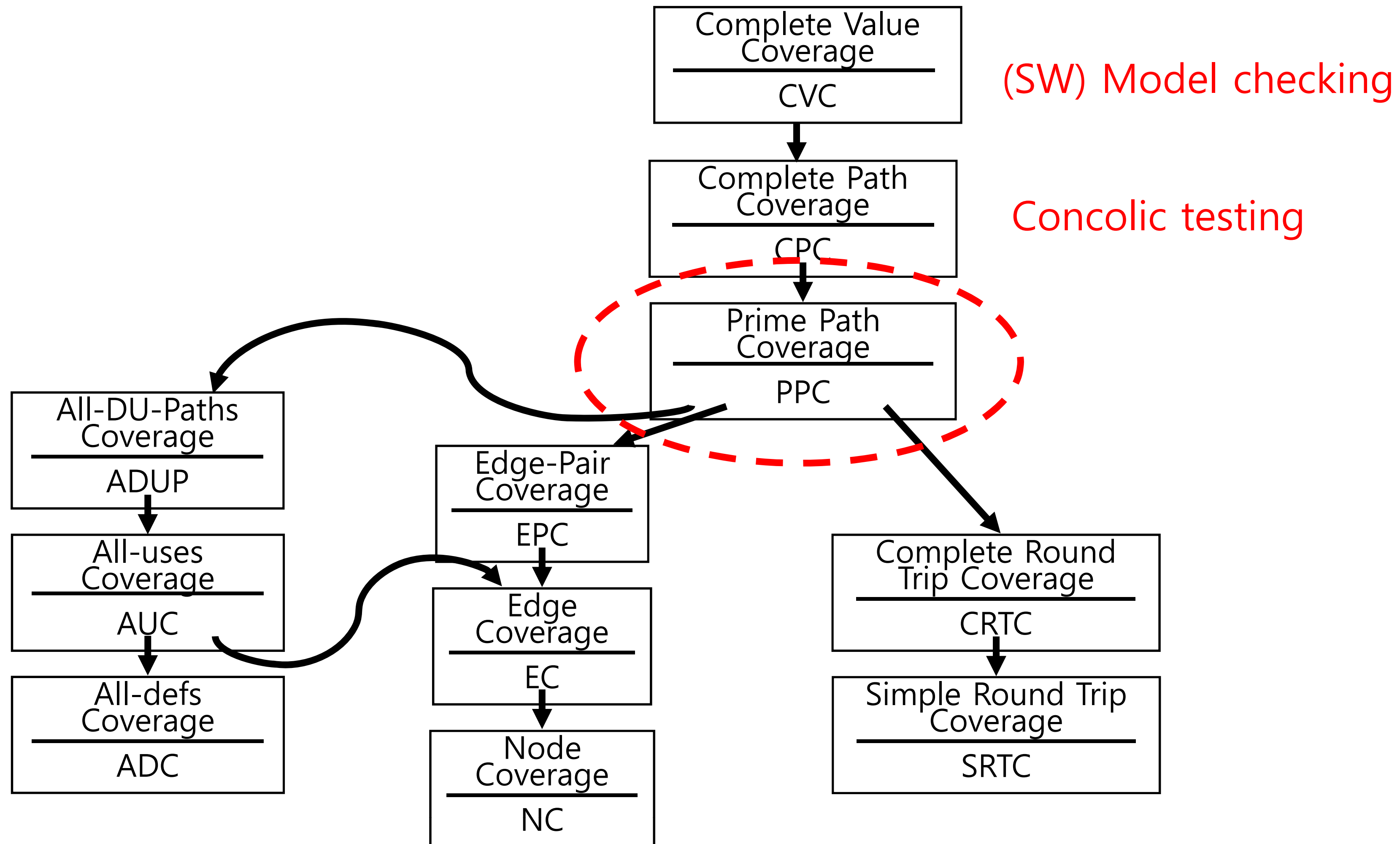
Observation 2

- Concolic testing can achieve higher effectiveness than manual testing, since test cases are automatically generated
  - i.e. aiming more than 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);
}
```



Hierarchy of SW Coverages



KAIST

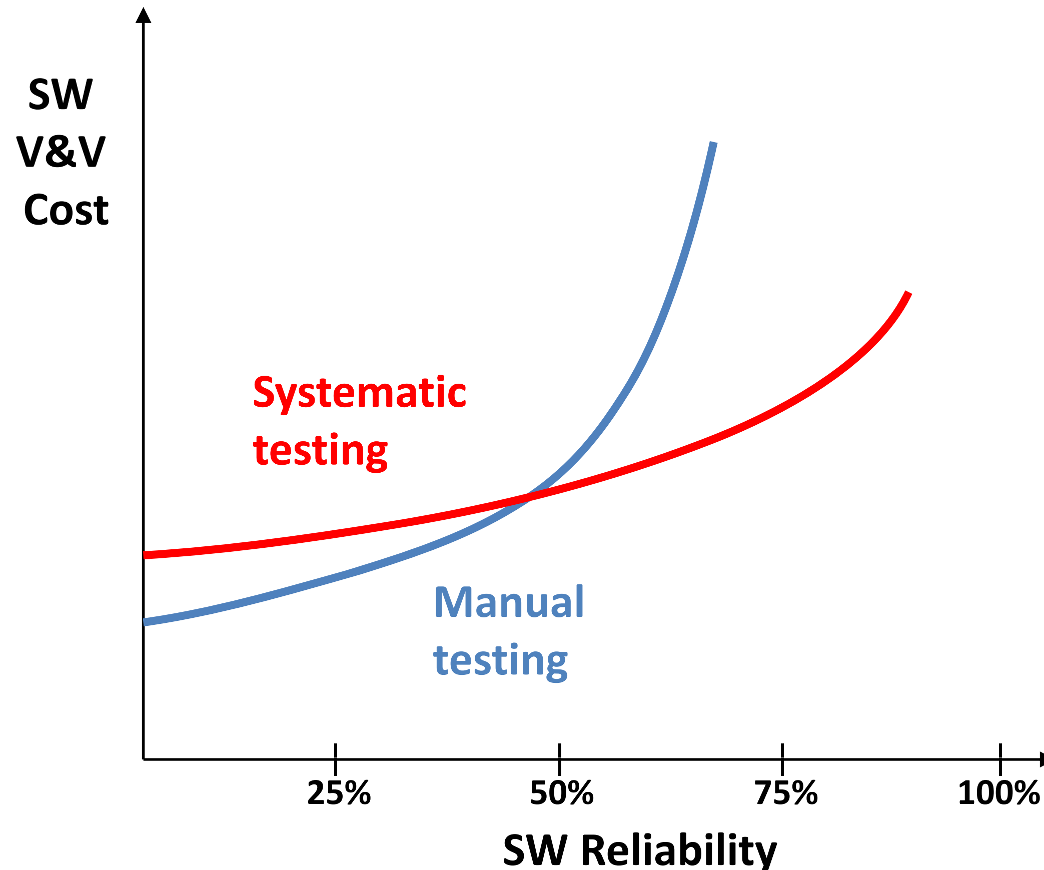
Observation 3

- Concolic testing can achieve higher coverage by usi ng existing manual test cases
  - i.e. in a regression testing context
- Many chances that concolic testing misses possible execution paths due to
  - External binary library call
  - Complex path condition
    - Today's SMT solver has many limitations
    - Ex. Cannot handle non-linear arithmetic
      - $\sin(x) + \cos(x) > x$
- To exploit manually generated test cases (which mi ght be generated manually) to by-pass

Observation 3 (cont.)

- Heuristics: Exploit manually generated test cases to by-pass previously mentioned difficulties
  - Note that concolic testing starts from an initial rando m input
    - We can extend one initial input to a set of manually create d inputs which can (hopely) pass difficult pass condition
- This step can be incorporated into a regression t esting framework well
  - i.e. using concolic testing to cover affected elements of a revised target program

Conclusion: Manual Testing v.s. Automated Testing



- Traditional manual testing is easy to apply for programs with low reliability
- However, systematic testing can achieve much higher reliability
- Concolic testing can improve the reliability of target programs cost-effectively
- Consolic testing has much rooms to improve