CSE503: SOFTWARE ENGINEERING

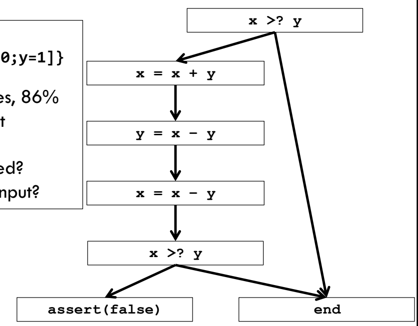
SYMBOLIC TESTING, AUTOMATED TEST GENERATION … AND MORE!

David Notkin
Spring 2011

CFG for (edge) coverage

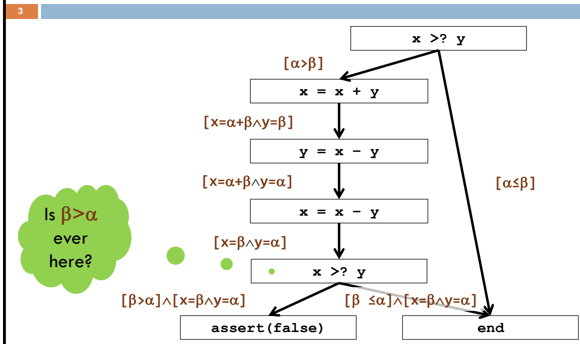
2

- Test inputs {[x=1;y=0],[x=0;y=1]}
- Cover 6/7 edges, 86%
- Which edge isn't covered?
- Can it be covered?
- With what test input?



Symbolic execution [x=α;y=β]

3

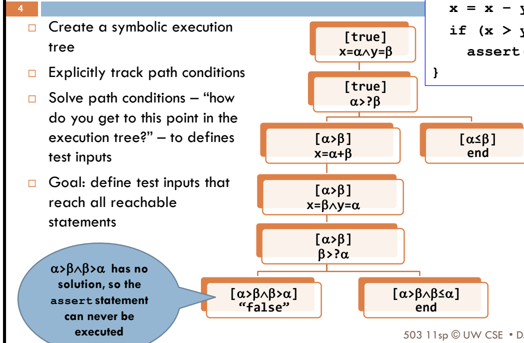


What's really going on?

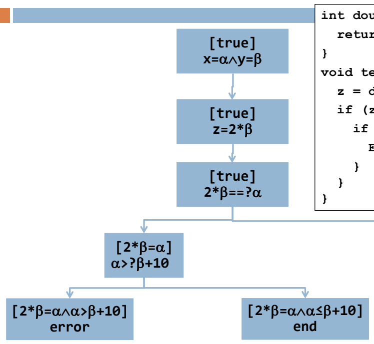
```
if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert(false)
}
```

4

- Create a symbolic execution tree
- Explicitly track path conditions
- Solve path conditions – "how do you get to this point in the execution tree?" – to defines test inputs
- Goal: define test inputs that reach all reachable statements



Another example (Sen and Agha)




```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }
  }
}
```

Error: possible by solving equations

$2*\beta=\alpha \wedge \alpha>\beta+10$

$\equiv 2*\beta=\alpha \wedge 2*\beta>\beta+10$

$\equiv 2*\beta=\alpha \wedge \beta>10$

$\equiv 2*\beta=\alpha \wedge \beta>10$

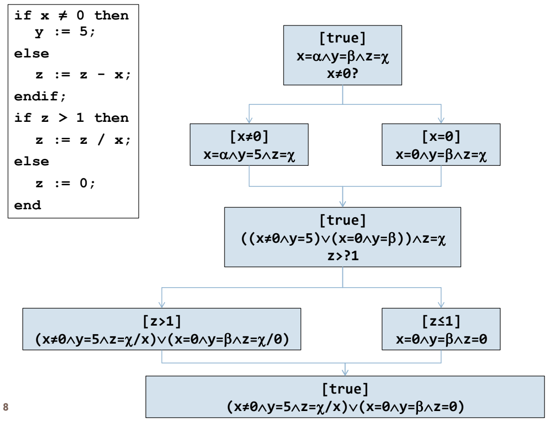
- Any solution to this will cause the error state to be reached
  - {x=22, y=11}, {x=200, y=100}, …

OK, do this in small groups for…


```
if x ≠ 0 then
  y := 5;
else
  z := z - x;
endif;
if z > 1 then
  z := z / x;
else
  z := 0;
end
```



```
if x ≠ 0 then
  y := 5;
else
  z := z - x;
endif;
if z > 1 then
  z := z / x;
else
  z := 0;
end
```



Way cool – we're done!

9

- First example can't reach **`assert(false)`**, and it's easy to reach **`end`** via both possible paths
- Second example: can reach **`error`** and **`end`** via both possible paths
- Third example: can avoid edge coverage weakness
- Well, what if we can't solve the path conditions?
  - Some arithmetic, some recursion, some loops, some pointer expressions, etc.
  - We'll see an example
- What if we want specific test cases?

503 11sp © UW CSE • D. Notkin

Concolic testing: Sen et al.

10

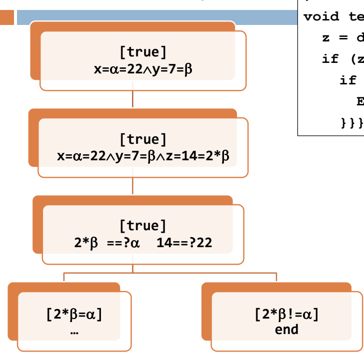
- Basically, combine concrete and symbolic execution
- More precisely…
  - Generate a random concrete input
  - Execute the program on that input both concretely and symbolically simultaneously
  - Follow the concrete execution and maintain the path conditions along with the corresponding symbolic execution
  - If and when the symbolic constraints cannot be solved by a solver, use the path conditions collected by this guided process to constrain the generation of inputs for the next iteration
  - Repeat until test inputs are produced to exercise all feasible paths

503 11sp © UW CSE • D. Notkin

2nd example redux
1st iteration x=22, y=7

11

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
}}}
```



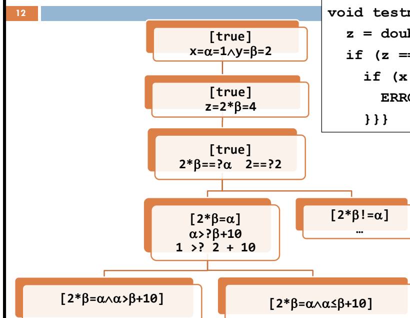
- **Now solve $2*\beta=\alpha$ to force the other branch**
- **`x=1; y=2` is one solution**

503 11sp © UW CSE • D. Notkin

2nd example
2nd iteration x=1, y=2

12

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
}}}
```

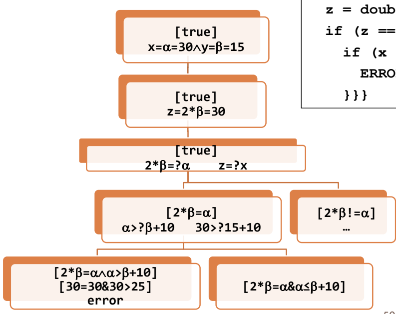


- Now solve **$2*\beta=\alpha\wedge\alpha\leq\beta+10$** to force the other branch
- **`x=30;y=15`** is one solution

503 11sp © UW CSE • D. Notkin

2nd example
3rd iteration x=30, y=15


```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }}}
```



Three concrete test cases


```
int double (int v){ return 2*v;}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }
  }
}
```

x	y	
22	7	Takes first else
2	1	Takes first then and second else
30	15	Takes first and second then

Concolic testing example: P. Sağlam

- Random seed
  - **x=-3; y=7**
- Concrete
  - **z=9**
- Symbolic
  - $z=x^3+3x^2+9$
  - Take **then** branch with constraint $x^3+3x^2+9 \neq y$

```
void 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();
  }
}
```

- Take **else** branch with constraint $x^3+3x^2+9=y$

Concolic testing example: P. Sağlam

- Solving is hard for $x^3+3x^2+9=y$
- So use **z**'s concrete value, which is currently **9**, and continue concretely
- **9!=7** so **then** is good
- Symbolically solve **9=y** for **else** clause
- Execute next run with **x=-3; y=9** so **else** is bad

```
void 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();
  }
}
```

- When symbolic expression becomes unmanageable (e.g., non-linear) replace it by concrete value

Example

Following N slides from Sağlam

```
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

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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;
}
```

At `int testme(cell *p, int x) {`: concrete state: p → NULL, x=236; symbolic state: p=p_0, x=x_0

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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;
}
```

At `if (x > 0)`: concrete state: p → NULL, x=236; symbolic state: p=p_0, x=x_0

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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;
}
```

At `if (p != NULL)`: concrete state: p → NULL, x=236; symbolic state: p=p_0, x=x_0; constraints: $x_0>0$

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → NULL, x=236

Symbolic state: $p=p_0$, $x=x_0$

Constraints: $x_0>0$, !(p_0!=NULL)

CUTE Approach

Concrete Execution | Symbolic Execution

concrete | symbolic | constraints

```
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

Concrete state: p → NULL, x=236

Symbolic state: $p=p_0$, $x=x_0$

Constraints: $x_0>0$, p_0=NULL

CUTE Approach

Concrete Execution | Symbolic Execution

concrete | symbolic | constraints

```
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_0=236, p_0 → [634 |] → NULL

Concrete state: p → NULL, x=236

Symbolic state: $p=p_0$, $x=x_0$

Constraints: $x_0>0$, p_0=NULL

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → [634 |] → NULL, x=236

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

CUTE Approach

Concrete Execution | Symbolic Execution

```
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 state: p → [634 |] → NULL, x=236

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

constraints: $x_0>0$

CUTE Approach

Concrete Execution | Symbolic Execution

```
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 state: p → [634 |] → NULL, 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

Concrete Execution | Symbolic Execution

```
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 state: p → [634 |] → NULL, x=236

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

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

CUTE Approach

Concrete Execution | Symbolic Execution

```
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 state: p → [634 |] → NULL, x=236

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

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete | symbolic | constraints

```
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 $p_0 \neq NULL$ and $2x_0+1=v_0$

p NULL 634, 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

Concrete Execution | Symbolic Execution

concrete | symbolic | constraints

```
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 $p_0 \neq NULL$ and $2x_0+1=v_0$

$x_0=1$, p_0 NULL 3

p NULL 634, 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

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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;
}
```

p NULL 3, x=1

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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;
}
```

p NULL 3, x=1

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

$x_0>0$

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → [3 |] → NULL, x=1

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

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → [3 |] → NULL, x=1

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

Constraints: $x_0>0$, $p_0 \neq NULL$, $2x_0+1=v_0$

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → [3 |] → NULL, x=1

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

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → [3 |] → NULL, x=1

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

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 $p_0 \neq NULL$ and $2x_0+1=v_0$ and $n_0=p_0$

.

Concrete state: p, NULL, 3, x=1

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

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 $p_0 \neq NULL$ and $2x_0+1=v_0$ and $n_0=p_0$

$x_0=1$, p_0 → 3

Concrete state: p, NULL, 3, x=1

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

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p, 3, x=1

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p, 3, x=1

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

Constraints: $x_0>0$

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → [3 | ·], x=1

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

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

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → [3 | ·], x=1

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

Constraints: $x_0>0$, $p_0 \neq NULL$, $2x_0+1=v_0$

CUTE Approach

Concrete Execution | Symbolic Execution

concrete state | symbolic state | constraints

```
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 state: p → [3 | ·], x=1

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

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

Concolic testing example: P. Sağlam

45

- Random
  - Random memory graph reachable from **p**
  - Random value for **x**
  - Probability of reaching **abort( )** is extremely low
- (Why is this a somewhat misleading motivation?)

```
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;
}
```

503 11sp © UW CSE • D. Notkin

Concolic: status

- The jury is still out on concolic testing – but it surely has potential
- There are many papers on the general topic
- Here's one that is somewhat high-level Microsoft-oriented
  - Godefroid et al. Automating Software Testing Using Program Analysis *IEEE Software* (Sep/Oct 2008)
  - They tend to call the approach DART – Dynamic Automated Random Testing