KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler

Presented by Alex Polozov

Concolic testing

Run the program under the *symbolic virtual machine:*

- Instruction calls on *concrete values* are evaluated as usual
- Instruction calls on *symbolic values* create a new symbolic value

```
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

```
// Assuming klee_make_symbolic(digits, 5 * sizeof(char), “digit
```

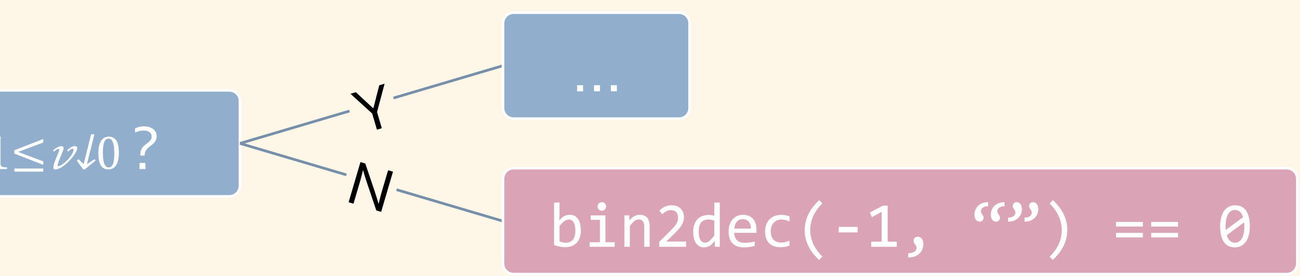
n
digits[0]
digits[1]
digits[2]
digits[3]
digits[4]

```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

n	
digits[0]	
digits[1]	
digits[2]	
digits[3]	
digits[4]	
result	
power2	

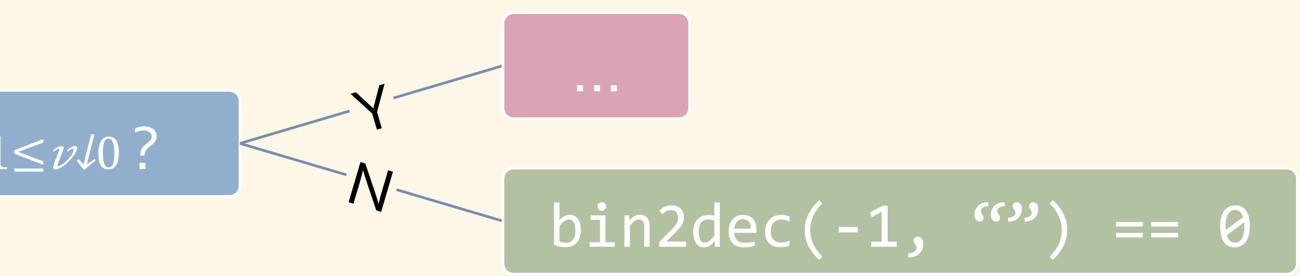
```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
          i <= n;
          ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

n	
digits[0]	
digits[1]	
digits[2]	
digits[3]	
digits[4]	
result	
power2	



Path condition: $\Phi = (v_0 < 1)$

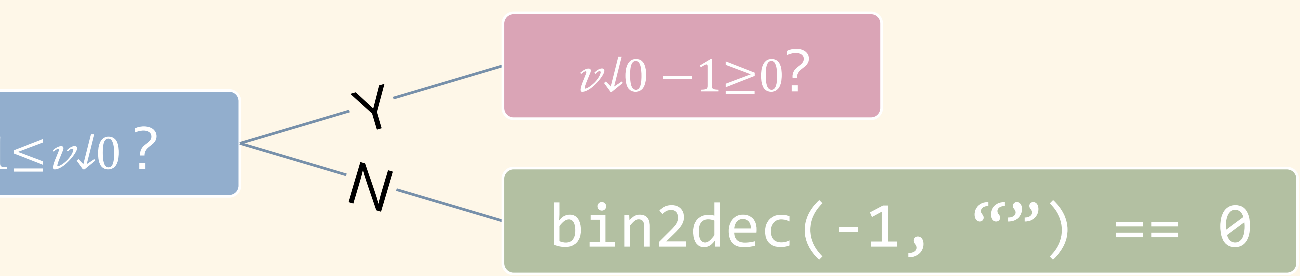
```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```



Path condition: $\Phi = (v_0 \geq 1)$

n	
digits[0]	
digits[1]	
digits[2]	
digits[3]	
digits[4]	
result	
power2	

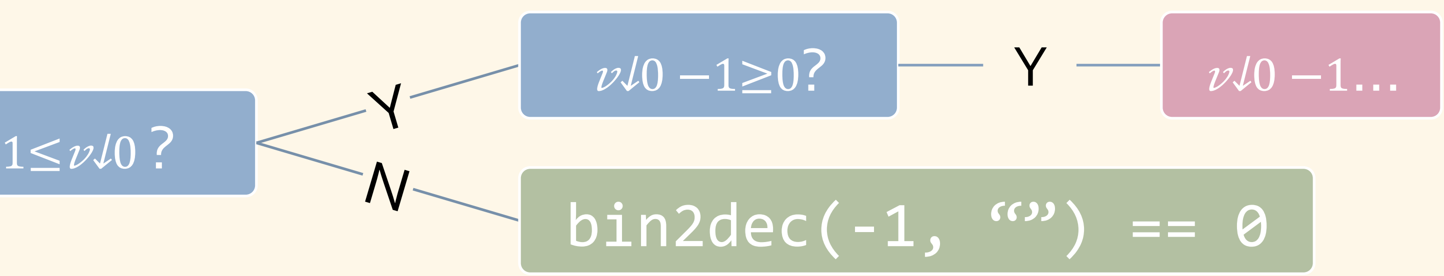
```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```



Path condition: $\Phi = (v_0 \geq 1)$

n	
digits[0]	
digits[1]	
digits[2]	
digits[3]	
digits[4]	
result	
power2	

```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

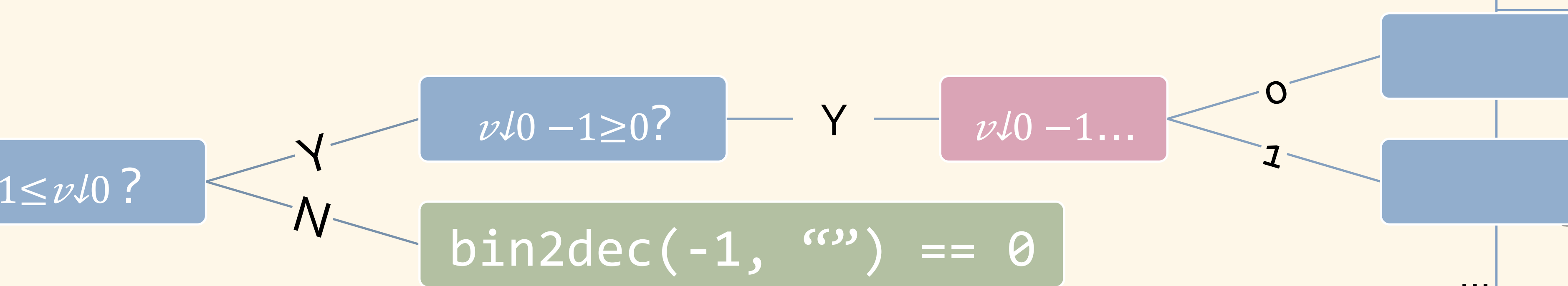


Path condition: $\Phi = (v_0 \ge 1)$

n	
digits[0]	
digits[1]	
digits[2]	
digits[3]	
digits[4]	
result	
power2	

```
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
       result += (digits[n - i] - '0') * power2;
    return result;
}
```

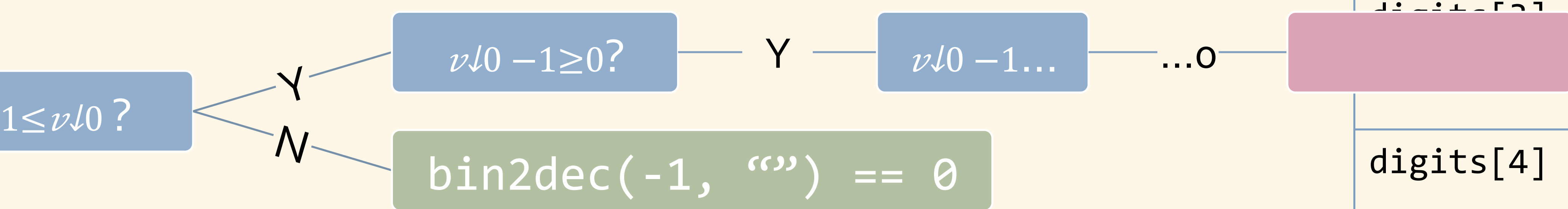
n
digits[0]
digits[1]
digits[2]
its[3]
its[4]
…
result
power2



Path condition: $\Phi = (v\downarrow 0 \geq 1)$

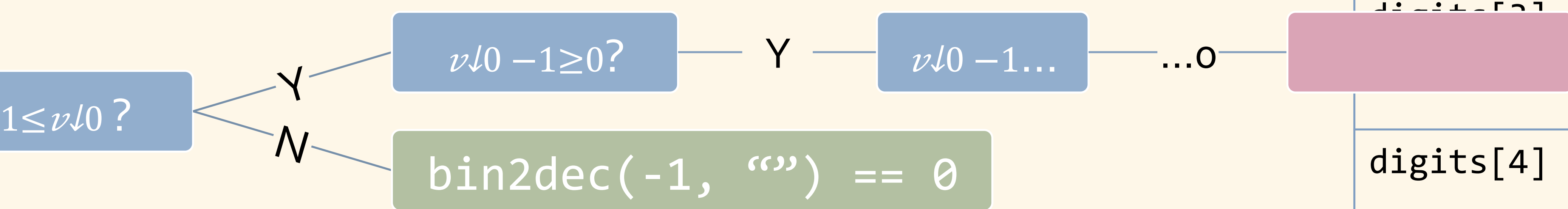
```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

n	
digits[0]	
digits[1]	
digits[2]	
[illegible]	
digits[4]	
result	
power2	



Path condition: $\Phi = (v_0 \ge 1) \wedge (v_0 - 1 = 0)$

```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

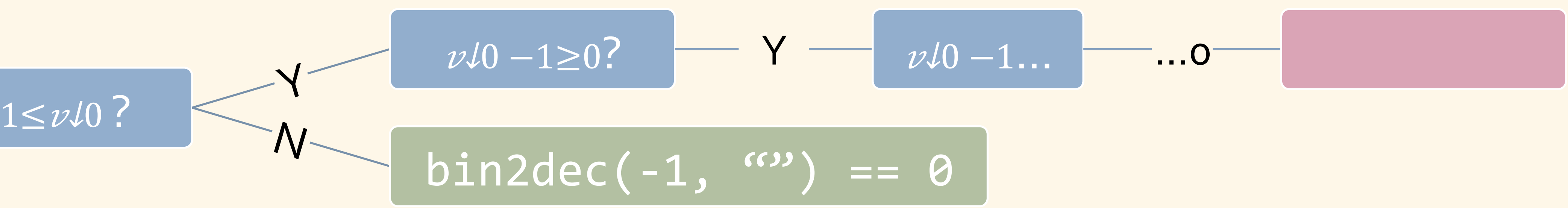


Path condition: $\Phi = (v\downarrow 0 = 1)$

n
digits[0]
digits[1]
digits[2]
digits[3]
digits[4]
result
power2

```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

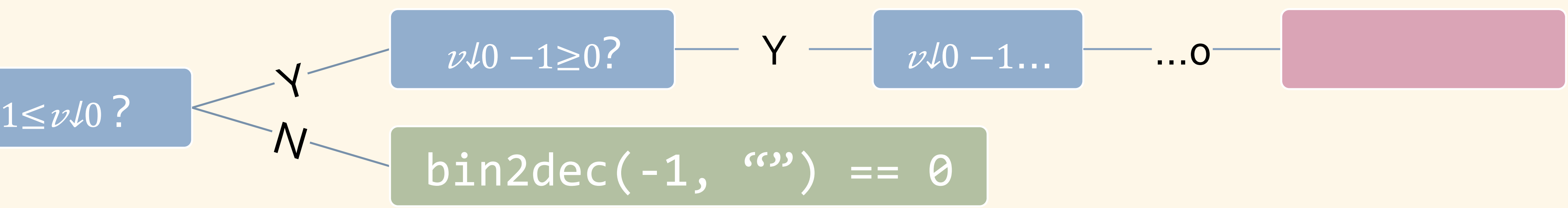
n	$v{\downarrow}0$
digits[0]	$v{\downarrow}1$
digits[1]	$v{\downarrow}2$
digits[2]	$v{\downarrow}3$
digits[3]	$v{\downarrow}4$
digits[4]	$v{\downarrow}5$
result	$v{\downarrow}1$ -4
power2	1
i	1



Path condition: $\Phi=(v{\downarrow}0 =1)$

```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

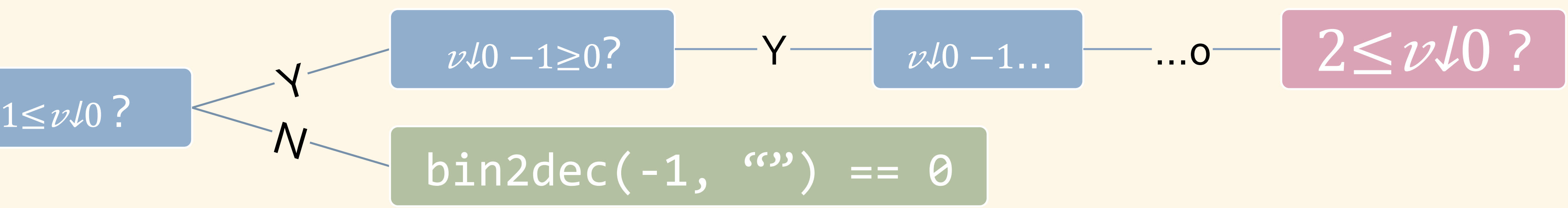
n	v_0
digits[0]	v_1
digits[1]	v_2
digits[2]	v_3
digits[3]	v_4
digits[4]	v_5
result	v_1 … -4
power2	2
i	2



Path condition: $\Phi = (v_0 = 1)$

```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

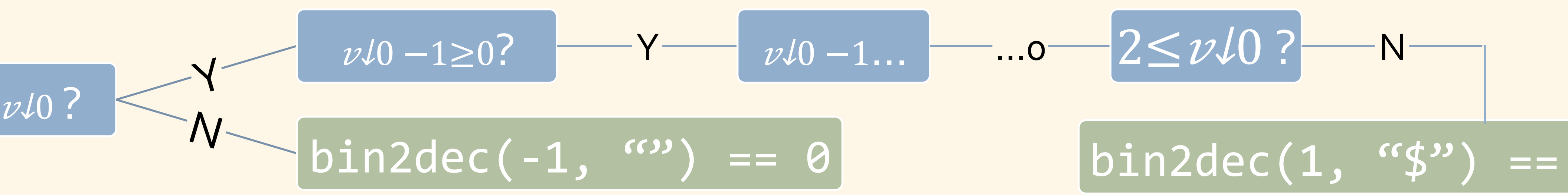
n	v_0
digits[0]	v_1
digits[1]	v_2
digits[2]	v_3
digits[3]	v_4
digits[4]	v_5
result	v_1 -4
power2	2
i	2



Path condition: $\Phi = (v_0 = 1)$

```c
int bin2dec(int n, char *digits) {
    int result = 0;
    for (int power2 = 1, i = 1;
         i <= n;
         ++i, power2 *= 2)
        result += (digits[n - i] - '0') * power2;
    return result;
}
```

n	v_0
digits[0]	v_1
digits[1]	v_2
digits[2]	v_3
digits[3]	v_4
digits[4]	v_5
result	v_1 -4
power2	2
i	2



Path condition: $\Phi = (v_0 = 1)$

Concolic testing

Run the program under the *symbolic virtual machine:*

- Instruction calls on *concrete values* are evaluated as usual
- Instruction calls on *symbolic values* create a new symbolic value

Fork new processes on conditional instructions:

- 1 process if Φ implies the condition or its negation
- 2 processes otherwise
- N processes for a load/store that may alias N locations

Test case generation

Validating assertions

$[\Phi = (n \geq 1) \wedge (i > n) \wedge (n \leq SIZE)]$

```
int res = data[i % (n - 1)];
```

$\exists n, i : \Phi \wedge (i \,\%\, (n-1) \geq SIZE)$? No.

$\exists n, i : \Phi \wedge (i \,\%\, (n-1) < 0)$? No.

$\exists n, i : \Phi \wedge n - 1 = 0$? Yes:

$n = 1 \wedge i = 2$.

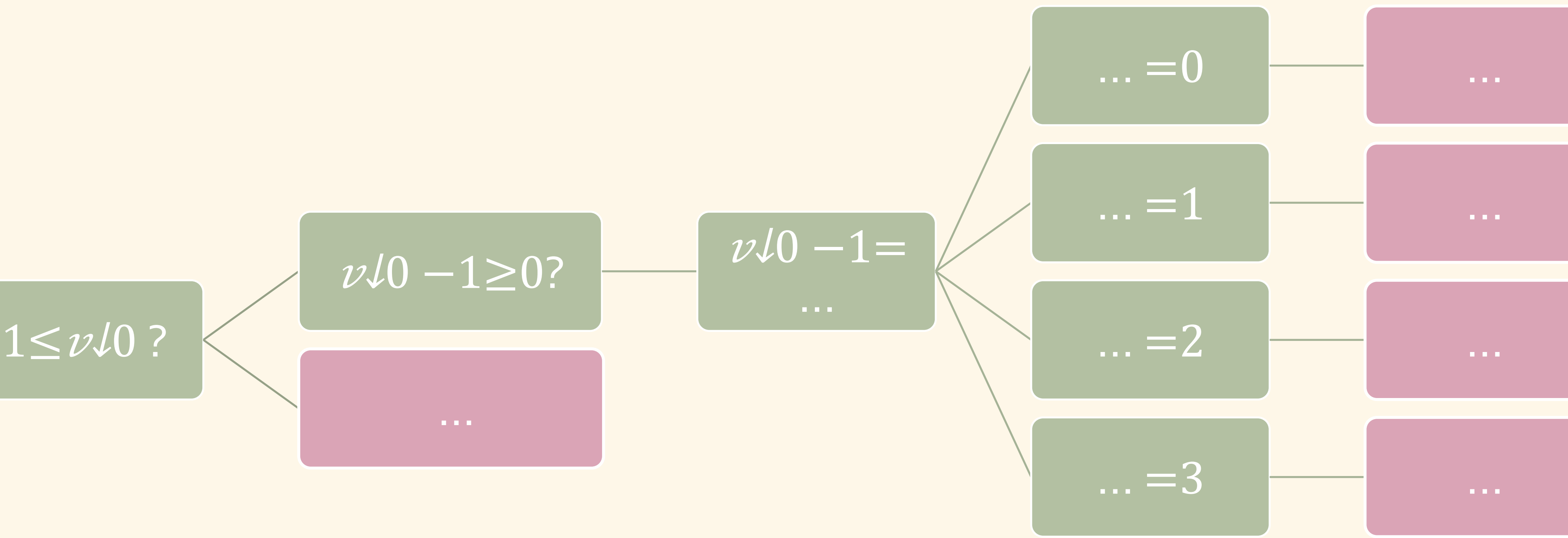
Concretization of final states

$[\Phi = (n > 1) \wedge (i > n)]$

```
return i % (n - 1);
```

$\exists i, n, r : \Phi \wedge r = (i \,\%\, (n-1))$? Yes.

Scheduling



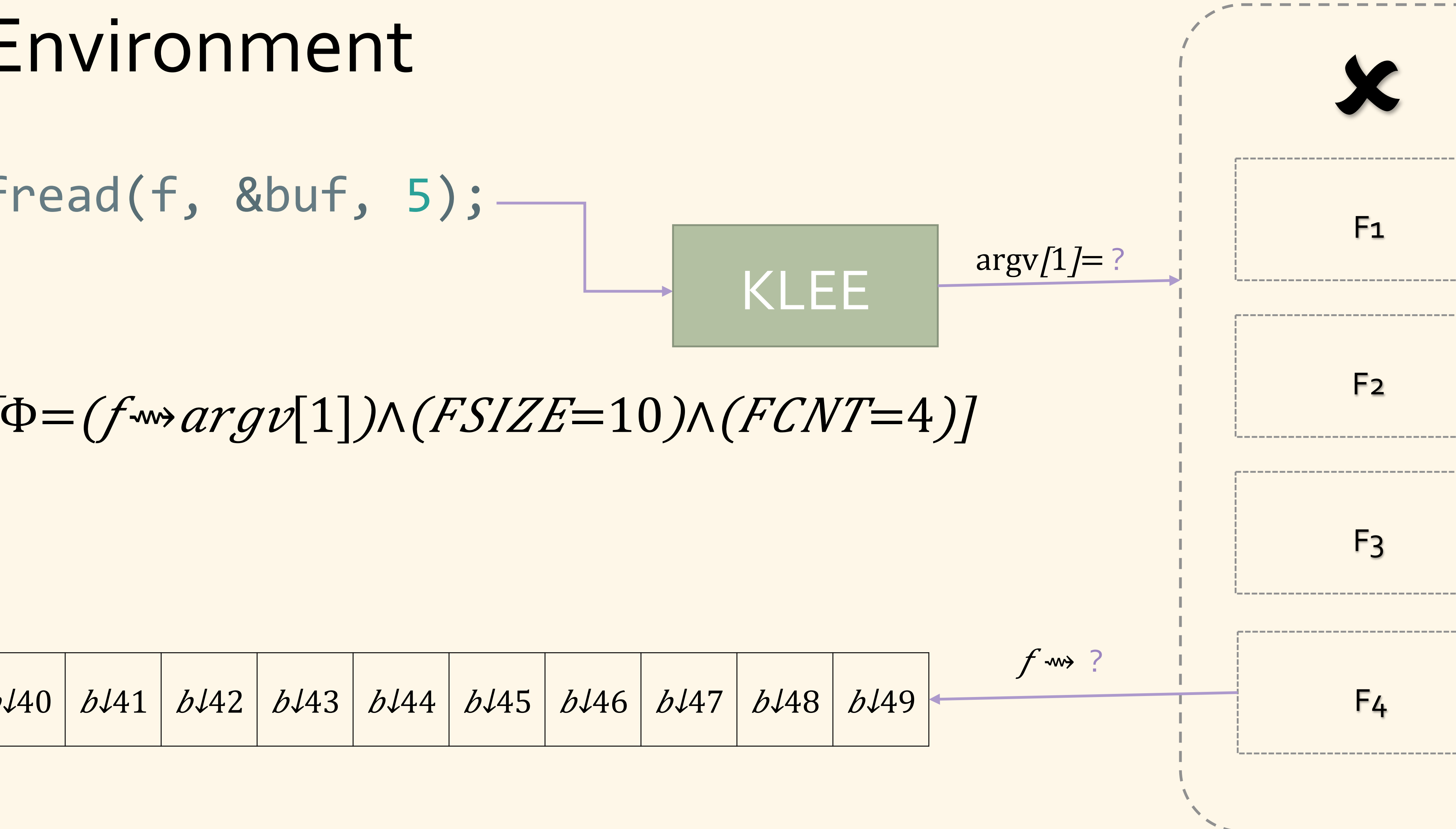
Scheduling

- At any moment, KLEE maintains exponentially many forks
- Requirements:
  - Each fork should get some computing time “in the limit”
  - State explosion in one part of the program should not prevail over the others
  - KLEE should prefer forks that cover new code
  - A fork’s computation should not block execution
- Solution:
  - Employ multiple *strategies for fork selection*
  - Switch between all strategies uniformly
  - Constrain each fork with a time limit

Environment

```
fread(f, &buf, 5);
```

$$\Phi=(f \rightsquigarrow argv[1]) \wedge (FSIZE=10) \wedge (FCNT=4)$$



Environment

Modeling at the level of OS calls, not stdlib functions!

- ✓ Short implementation
- ✓ Allows modeling uncommon system failures
- ✓ Trivial test case reproducibility

- ✗ Limited variability of the file system structure

Thank you!

Discussion time!