symbolic execution

have an emulator/virtual machine

but represent input values as *symbolic variables*

- like in algebra

choose a path through the program, track *constraints*

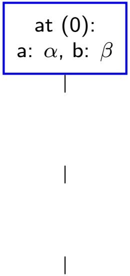
- what values did input need to have to get here?

then solve constraints based on variables to create real test case

- no solution? impossible path
- find solution? test case

example 0

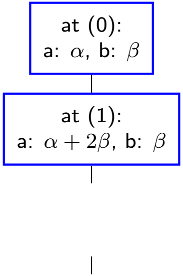
```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```



at (0):
a: α, b: β

example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```

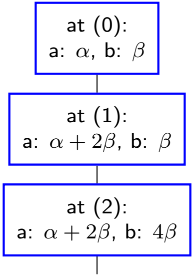


at (0):
a: α, b: β

at (1):
a: $\alpha + 2\beta$, b: β

example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```



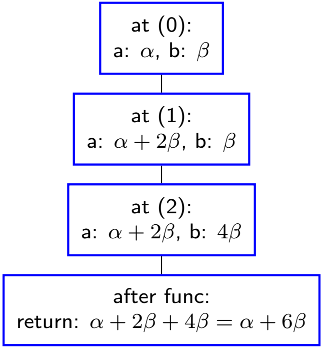
at (0):
a: α, b: β

at (1):
a: $\alpha + 2\beta$, b: β

at (2):
a: $\alpha + 2\beta$, b: 4β

example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```



at (0):
a: α, b: β

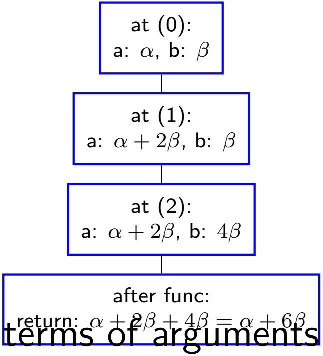
at (1):
a: $\alpha + 2\beta$, b: β

at (2):
a: $\alpha + 2\beta$, b: 4β

after func:
return: $\alpha + 2\beta + 4\beta = \alpha + 6\beta$

example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```



at (0):
a: α, b: β

at (1):
a: $\alpha + 2\beta$, b: β

at (2):
a: $\alpha + 2\beta$, b: 4β

after func:
return: $\alpha + 2\beta + 4\beta = \alpha + 6\beta$

can express return value of function in terms of arguments

then can solve for possible value of arguments

example: if return $==$ 10, then can enumerate:

$(\alpha,\beta) = (10{,}0)$
$(\alpha,\beta) = (4{,}1)$

actually doing this

angr is a binary analysis toolkit written in Python

- has Ghidra-like GUI, but not very stable/maintained as far as I can tell

among other things, converts assembly into intermediate form

supports symbolic execution

angr setup

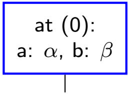
```
import angr
import claripy

p = angr.Project("./example0",
                 load_options='auto_load_libs': False)

foo_addr = p.loader.main_object.get_symbol('foo').rebased_addr
input_a = claripy.BVS('initial_a', 32) # 32-bit bit vector
input_b = claripy.BVS('initial_b', 32) # 32-bit bit vector
init_state = p.factory.call_state(foo_addr, input_a, input_b)
simgr = p.factory.simulation_manager(init_state)
# <SimulationManager with 1 active>
```

angr running

```
print(f"RIP=simgr.active[0].regs.rip versus foo_addr:#x")
    # RIP=<BV64 0x4011f9> versus 0x4011f9
print(f"EAX=simgr.active[0].regs.eax")
    # RAX=<BV reg_eax_3_32> (unknown value)
simgr.step()
    # simgr = <SimulationManager with 1 active>
simgr.step()
    # simgr = <SimulationManager with 1 deadended>
state = simgr.deadended[0]
print(f"EAX=state.regs.eax")
    # EAX=initial_a_0_32 +
    #      (initial_b_1_32[30:0] .. 0) +
    #      (initial_b_1_32[29:0] .. 0)
state.solver.add(state.regs.eax == 10)
print(state.solver.eval(input_a), state.solver.eval(input_b))
    # 10 0
state.solver.add(input_b != 0)
print(state.solver.eval(input_a), state.solver.eval(input_b))
    # 4294901754 715838808
```

example 1

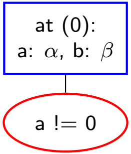
```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



Adapted from Hicks, “Symbolic Execution for Finding Bugs”

example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```

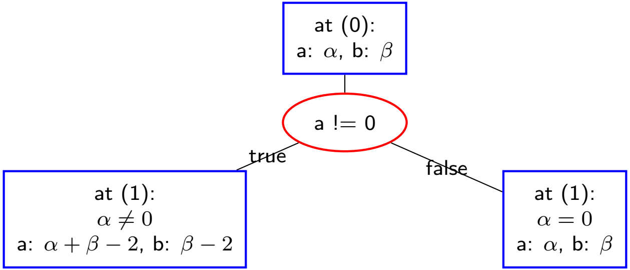


every variable represented as an *equation*

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 1

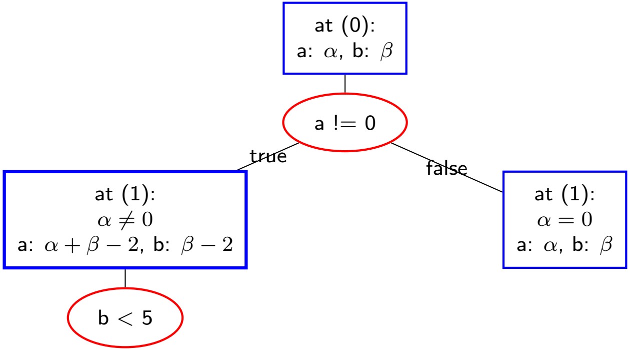
```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 1

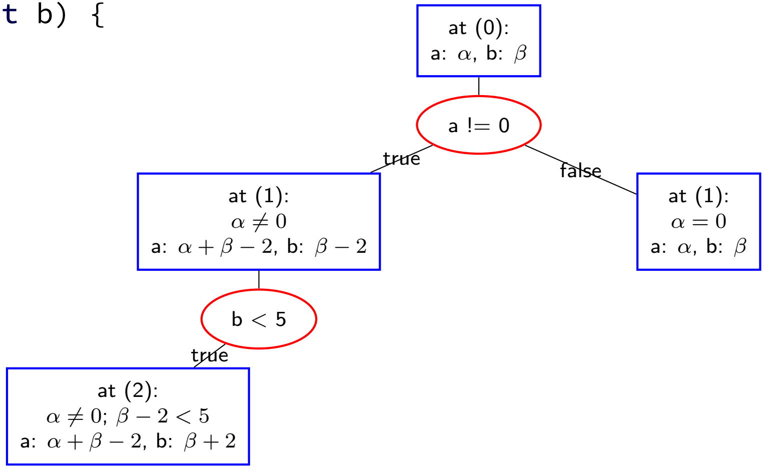
```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



at (0):
a: α, b: β

a $!=$ 0

true

false

at (1):
$\alpha \neq 0$
a: $\alpha + \beta - 2$, b: $\beta - 2$

at (1):
$\alpha = 0$
a: α, b: β

b $<$ 5

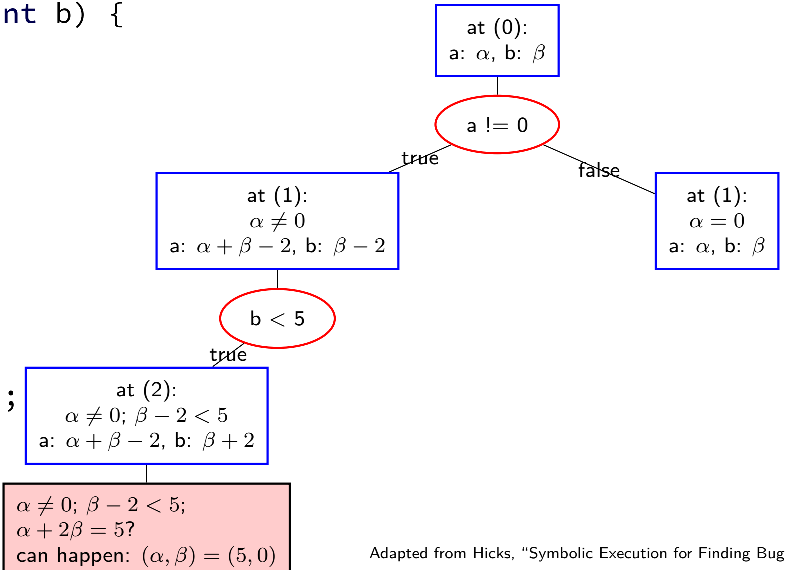
true

at (2):
$\alpha \neq 0$; $\beta - 2 < 5$
a: $\alpha + \beta - 2$, b: $\beta + 2$

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



at (0):
a: α, b: β

a != 0

true

false

at (1):
$\alpha \neq 0$
a: $\alpha + \beta - 2$, b: $\beta - 2$

at (1):
$\alpha = 0$
a: α, b: β

b < 5

true

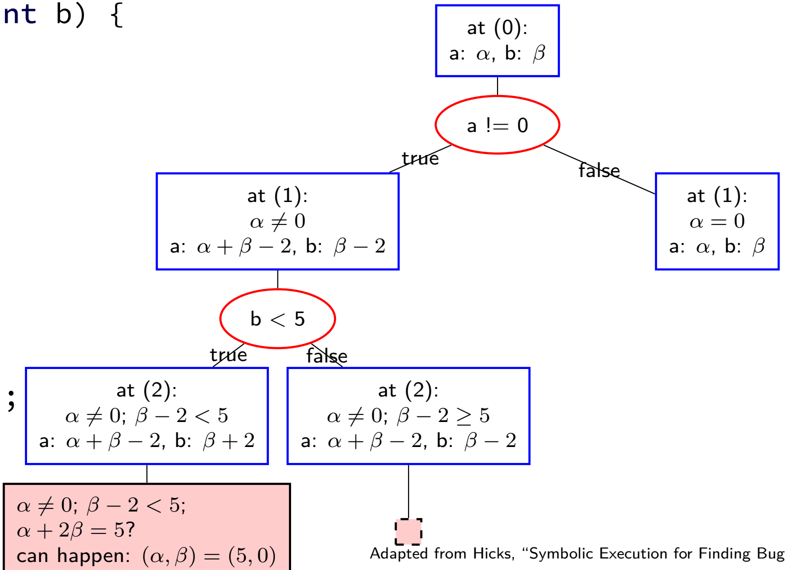
at (2):
$\alpha \neq 0$; $\beta - 2 < 5$
a: $\alpha + \beta - 2$, b: $\beta + 2$

$\alpha \neq 0$; $\beta - 2 < 5$;
$\alpha + 2\beta = 5$?
can happen: $(\alpha, \beta) = (5, 0)$

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



at (0):
a: α, b: β

a != 0

true

at (1):
$\alpha \neq 0$
a: $\alpha + \beta - 2$, b: $\beta - 2$

false

at (1):
$\alpha = 0$
a: α, b: β

b < 5

true

at (2):
$\alpha \neq 0$; $\beta - 2 < 5$
a: $\alpha + \beta - 2$, b: $\beta + 2$

false

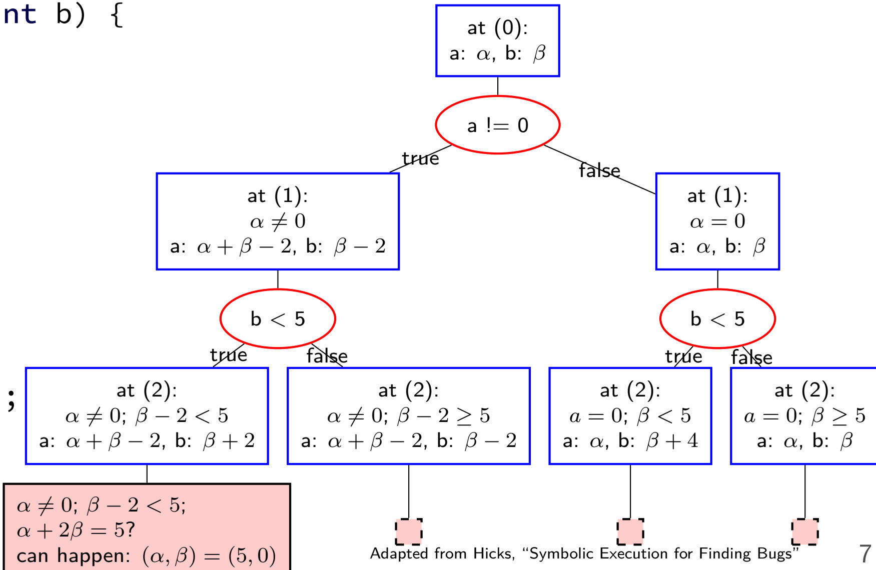
at (2):
$\alpha \neq 0$; $\beta - 2 \geq 5$
a: $\alpha + \beta - 2$, b: $\beta - 2$

$\alpha \neq 0$; $\beta - 2 < 5$;
$\alpha + 2\beta = 5$?
can happen: $(\alpha, \beta) = (5, 0)$

Adapted from Hicks, “Symbolic Execution for Finding Bugs”

example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



at (0):
a: α, b: β

a != 0

true

at (1):
$\alpha \neq 0$
a: $\alpha + \beta - 2$, b: $\beta - 2$

false

at (1):
$\alpha = 0$
a: α, b: β

b < 5

true

at (2):
$\alpha \neq 0$; $\beta - 2 < 5$
a: $\alpha + \beta - 2$, b: $\beta + 2$

false

at (2):
$\alpha \neq 0$; $\beta - 2 \geq 5$
a: $\alpha + \beta - 2$, b: $\beta - 2$

b < 5

true

at (2):
$a = 0$; $\beta < 5$
a: α, b: $\beta + 4$

false

at (2):
$a = 0$; $\beta \geq 5$
a: α, b: β

$\alpha \neq 0$; $\beta - 2 < 5$;
$\alpha + 2\beta = 5$?
can happen: $(\alpha, \beta) = (5, 0)$

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

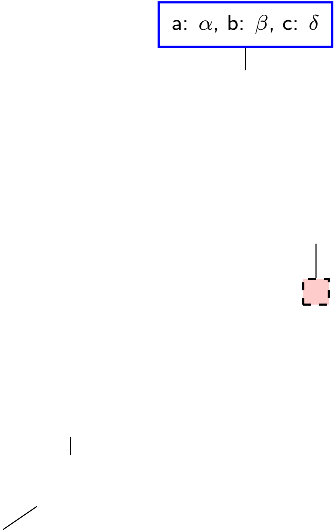
example 1 in angr

```
p = angr.Project("./example1", load_options='auto_load_libs': False)

foo_addr = p.loader.main_object.get_symbol('foo').rebased_addr
INTERESTING_addr = p.loader.main_object.get_symbol('INTERESTING').rebased_
input_a = claripy.BVS('initial_a', 32)
input_b = claripy.BVS('initial_b', 32)
init_state = p.factory.call_state(foo_addr, input_a, input_b)

simgr = p.factory.simulation_manager(init_state)
print("at beginning:", simgr)
simgr.explore(find=INTERESTING_addr)
print("after explore:", simgr)
for state in simgr.found:
    found_a = state.solver.eval(input_a)
    found_b = state.solver.eval(input_b)
    print(f'(a, b) = (found_a, found_b)')
```

```
after explore: <SimulationManager with 4 deadended, 4 found>
(a, b) = (0, 1)
(a, b) = (0, 5)
(a, b) = (1, 2)
```

example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```

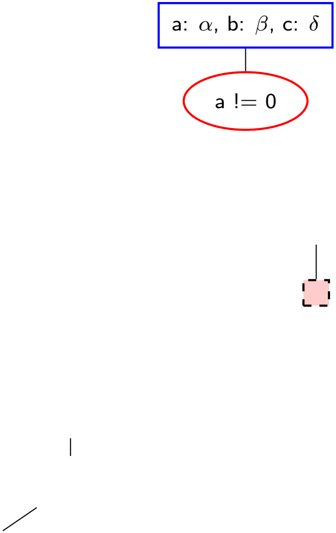


a: α, b: β, c: δ

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 2

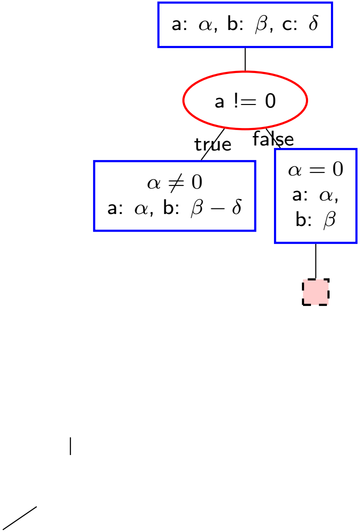
```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



a: α, b: β, c: δ

a != 0

true

false

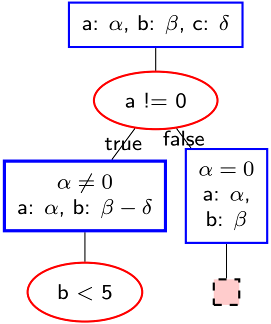
$\alpha \neq 0$
a: α, b: $\beta - \delta$

$\alpha = 0$
a: α,
b: β

Adapted from Hicks, “Symbolic Execution for Finding Bugs”

example 2

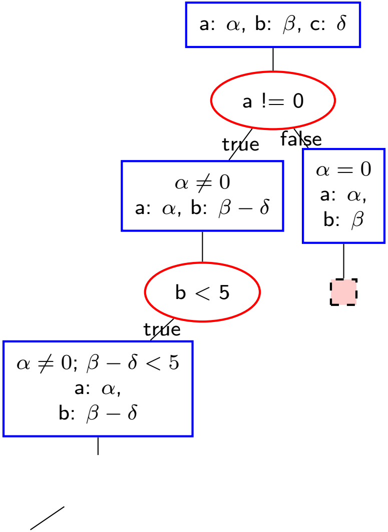
```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 2

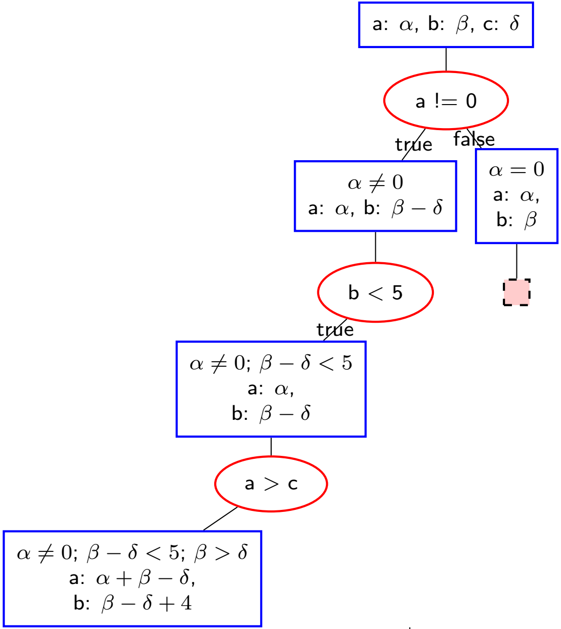
```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```


a: α, b: β, c: δ
a != 0
true
false
$\alpha \neq 0$
a: α, b: $\beta - \delta$
$\alpha = 0$
a: α,
b: β
b < 5
true
$\alpha \neq 0$; $\beta - \delta < 5$
a: α,
b: $\beta - \delta$

Adapted from Hicks, “Symbolic Execution for Finding Bugs”

example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



a: α, b: β, c: δ

a != 0

true

false

$\alpha \neq 0$
a: α, b: $\beta - \delta$

$\alpha = 0$
a: α,
b: β

b < 5

true

$\alpha \neq 0$; $\beta - \delta < 5$
a: α,
b: $\beta - \delta$

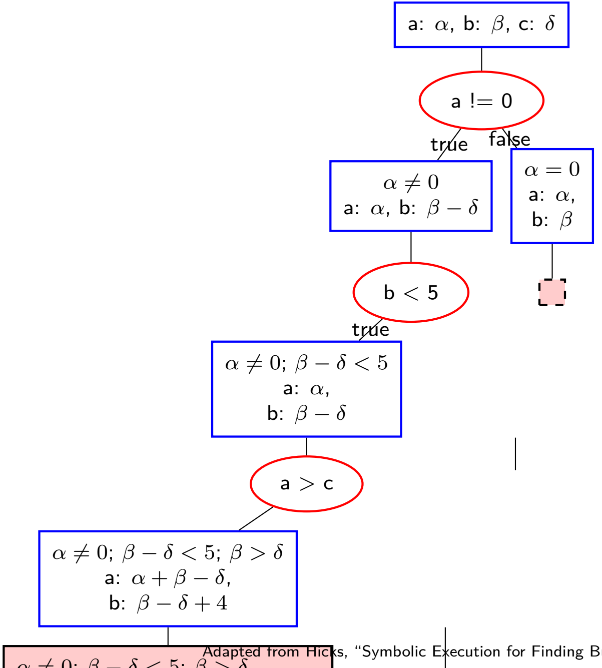
a > c

$\alpha \neq 0$; $\beta - \delta < 5$; $\beta > \delta$
a: $\alpha + \beta - \delta$,
b: $\beta - \delta + 4$

Adapted from Hicks, “Symbolic Execution for Finding Bugs”

example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



a: α, b: β, c: δ

a != 0

$\alpha \neq 0$
a: α, b: $\beta - \delta$

$\alpha = 0$
a: α,
b: β

b $<$ 5

$\alpha \neq 0$; $\beta - \delta < 5$
a: α,
b: $\beta - \delta$

a $>$ c

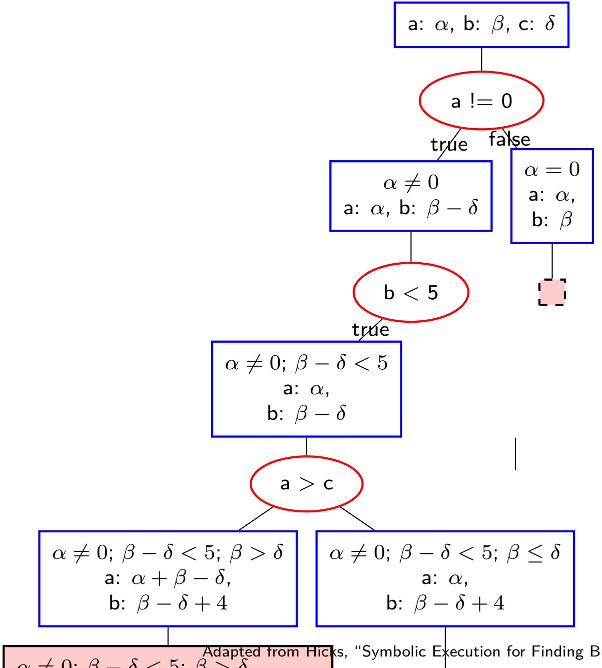
$\alpha \neq 0$; $\beta - \delta < 5$; $\beta > \delta$
a: $\alpha + \beta - \delta$,
b: $\beta - \delta + 4$

$\alpha \neq 0$; $\beta - \delta < 5$; $\beta > \delta$

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



a: α, b: β, c: δ

a != 0

true

false

$\alpha \neq 0$
a: α, b: $\beta - \delta$

$\alpha = 0$
a: α,
b: β

b < 5

true

$\alpha \neq 0$; $\beta - \delta < 5$
a: α,
b: $\beta - \delta$

a > c

$\alpha \neq 0$; $\beta - \delta < 5$; $\beta > \delta$
a: $\alpha + \beta - \delta$,
b: $\beta - \delta + 4$

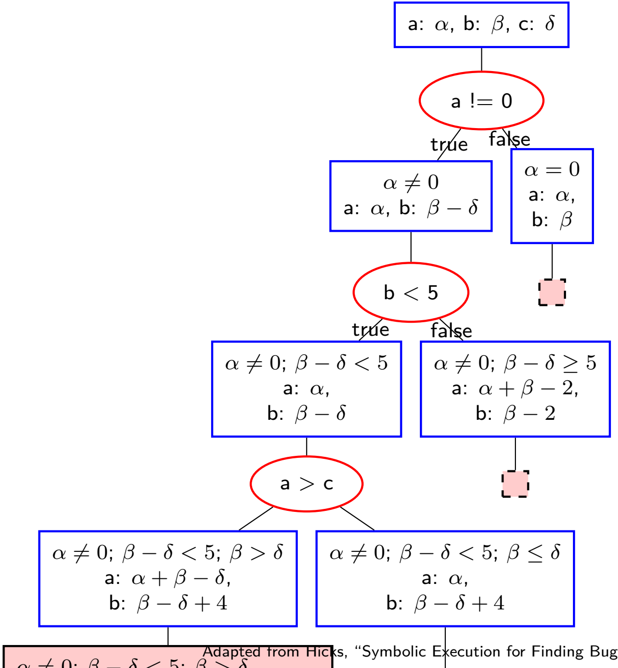
$\alpha \neq 0$; $\beta - \delta < 5$; $\beta \leq \delta$
a: α,
b: $\beta - \delta + 4$

$\alpha \neq 0$; $\beta - \delta < 5$; $\beta > \delta$

Adapted from Hicks, “Symbolic Execution for Finding Bugs”

example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



a: α, b: β, c: δ
a != 0
true
false
$\alpha \neq 0$
a: α, b: $\beta - \delta$
$\alpha = 0$
a: α,
b: β
b < 5
true
false
$\alpha \neq 0$; $\beta - \delta < 5$
a: α,
b: $\beta - \delta$
$\alpha \neq 0$; $\beta - \delta \geq 5$
a: $\alpha + \beta - 2$,
b: $\beta - 2$
a > c
$\alpha \neq 0$; $\beta - \delta < 5$; $\beta > \delta$
a: $\alpha + \beta - \delta$,
b: $\beta - \delta + 4$
$\alpha \neq 0$; $\beta - \delta < 5$; $\beta \leq \delta$
a: α,
b: $\beta - \delta + 4$
$\alpha \neq 0$; $\beta - \delta < 5$; $\beta > \delta$

Adapted from Hicks, “Symbolic Execution for Finding Bugs”

using for bounds checking

```
void foo() {
    char array[100];
    ...
    /* check inserted automatically: */
        assert(i >= 0 && i < 100);
    array[i] = ...;
    ...
}
```

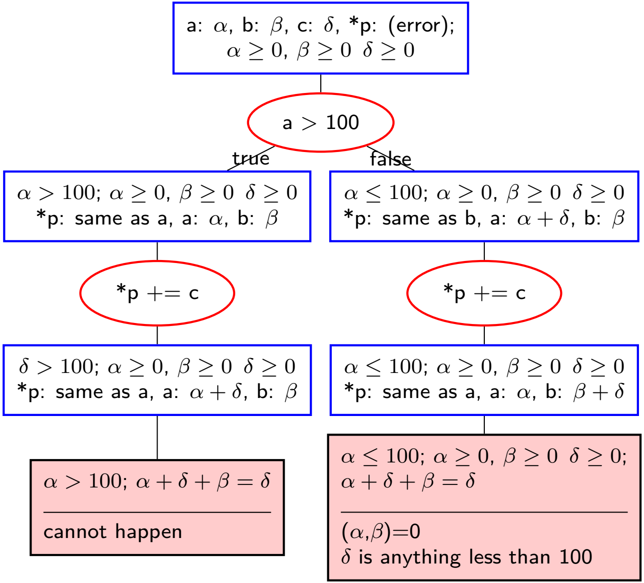
using symbolic execution to find memory bugs?

add assertions for bounds checks

need to track array sizes to do symbolic execution anyways

example 3

```
unsigned a, b;
void foo(unsigned c) {
    int *p;
    if (a > 100) {
        p = &a;
    } else {
        p = &b;
    }
    *p += c;
    assert(a + b == c);
}
```



a: α, b: β, c: δ, *p: (error);
$\alpha \geq 0$, $\beta \geq 0$ $\delta \geq 0$

a $>$ 100

true

false

$\alpha > 100$; $\alpha \geq 0$, $\beta \geq 0$ $\delta \geq 0$
*p: same as a, a: α, b: β

$\alpha \leq 100$; $\alpha \geq 0$, $\beta \geq 0$ $\delta \geq 0$
*p: same as b, a: $\alpha + \delta$, b: β

*p += c

*p += c

$\delta > 100$; $\alpha \geq 0$, $\beta \geq 0$ $\delta \geq 0$
*p: same as a, a: $\alpha + \delta$, b: β

$\alpha \leq 100$; $\alpha \geq 0$, $\beta \geq 0$ $\delta \geq 0$
*p: same as a, a: α, b: $\beta + \delta$

$\alpha > 100$; $\alpha + \delta + \beta = \delta$

cannot happen

$\alpha \leq 100$; $\alpha \geq 0$, $\beta \geq 0$ $\delta \geq 0$;
$\alpha + \delta + \beta = \delta$

(α,β)=0
δ is anything less than 100

exercise

```
void example(unsigned x, unsigned y) {
    if (x > y) return;
    x = x + y;
    assert(x + y + 1 > y);
}
```

1: to see if the assertion is meant, the equation we should solve (if initial values of x, y, are X, Y)?

2: what is an input that fails the assertion? (hint: integer overflow)

equation solving

can generate formula with bounded inputs

can always be solved by trying all possibilities

but actually solving is *NP-hard (i.e. not generally possible)*

luck: there exists solvers that are *often* good enough

…for small programs

…with lots of additional heuristics to make it work

tricky parts in symbolic execution

dealing with pointers?

- one method: one path for each valid value of pointer

solving equations?

- NP-hard (boolean satisfiablity) — not practical in general
- “good enough” for small enough programs/inputs
- ...after lots of tricks

how many paths?

- $< 100\%$ coverage in practice
- small input sizes (limited number of variables)

real symbolic execution

not yet used much outside of research

old technique (1970s), but recent resurgence

equation solving (‘SAT solvers’/‘SMT solvers’) is now much better

example usable tools: KLEE, symcc (test case generating)

KLEE optimizations

lots of optimizations to make search time pratical

prioritize paths that produce good tests

- try to execute *new code*
- try to find new paths new root of tree

reuse equation solving results:

- remove irrelevant variables from equation solving queries
  - e.g. if (x == 10) doesn't need variables unrelated to x's value
- cache of prior queries with "no solution"

results from 1 hour of compute time (from 2008 paper):

- avg. 91% coverage on Linux coreutils (basic command line tools)
- versus developer tests: 68% covergae

backup slides