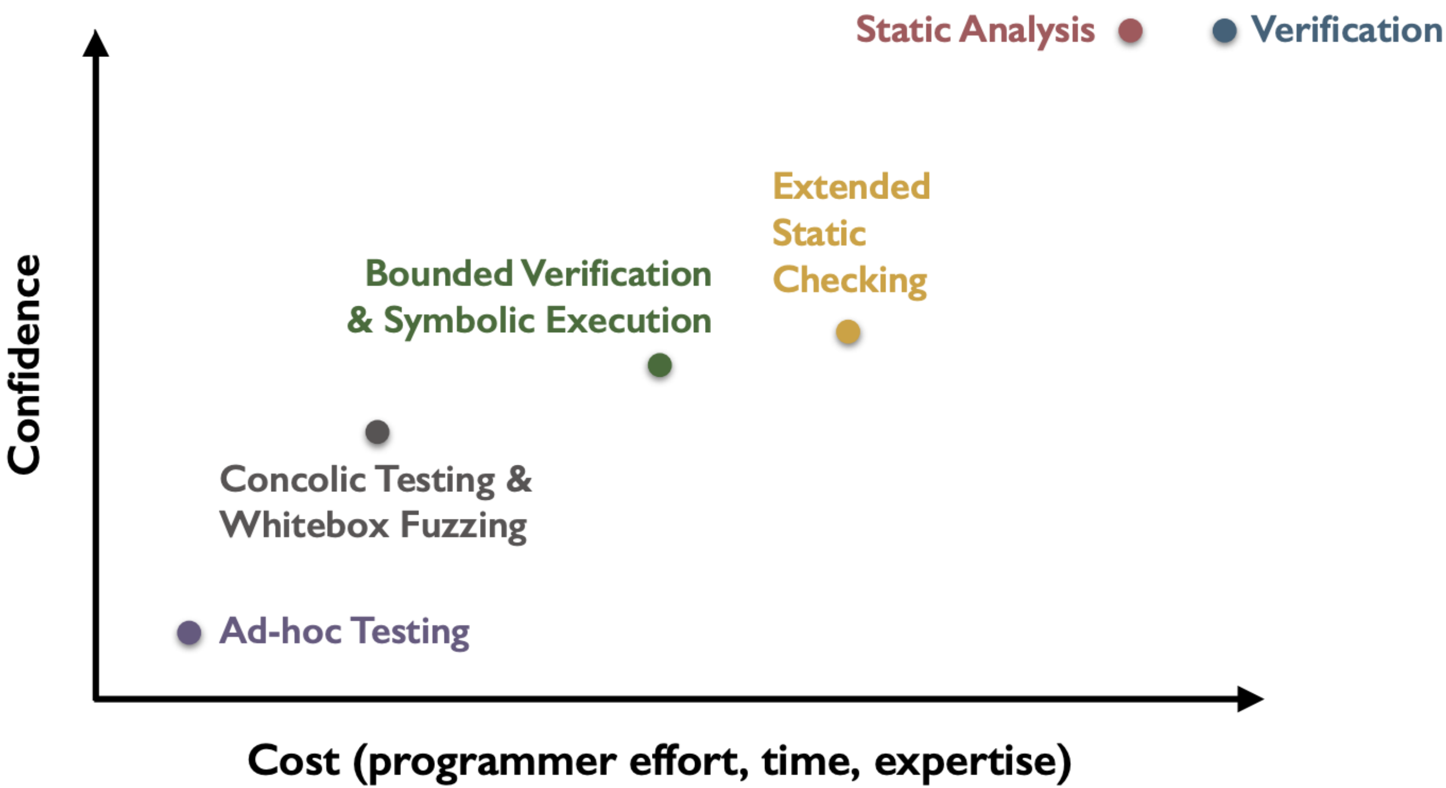
CMSC 433
Programming Language Technologies and Paradigms

Symbolic Execution

Based on the slides from Jeff Foster, Mike Hicks, and Emina Torlak

The Spectrum of Program Validation Tools



Introduction

- Verification and Static Analysis are great
  - Lots of interesting ideas and tools
- But can developers use it?
  - Formal verification of computer programs are hard.
  - Commercial static analysis tools have a huge code mass to deal with developer confusion, false positives, warning management, etc.

Testing is not Enough

Testing works, but each test only explores one possible execution

assert(f(3) == 5)

Can testing detect whether the following program throws an exception?

```
int f(int64_t a, int64_t b){
  if(a == 324572)
    if(b == 65535)
      assert fail;
  else
    …
}
```

Symbolic Execution

- Symbolic execution is a way to generalize testing.
  - A bug finding technique that is easy to use
  - No false positives
  - Produces a concrete input (a test case) on which the program will fail to meet the specification
  - But it cannot, in general, prove the absence of errors
- Key idea
  - Evaluate the program on symbolic input values
  - Use an automated theorem prover to check whether there are corresponding concrete input values that make the program fail.

A Brief history of Symbolic Execution

- 1976: A system to generate test data and symbolically execute programs (Lori Clarke)
- 1976: Symbolic execution and program testing (James King)
- 2005-present: practical symbolic execution
  - Using SMT solvers
  - Heuristics to control exponential explosion
  - Heap modeling and reasoning about pointers
  - Environment modeling
  - Dealing with solver limitations

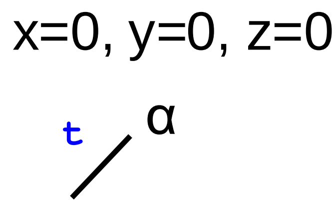
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```

x=0, y=0, z=0

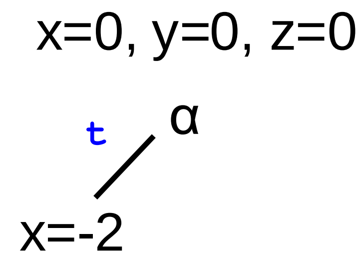
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



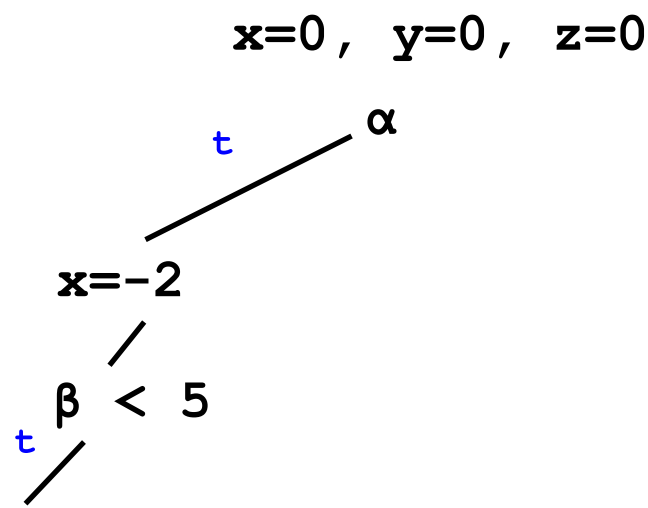
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



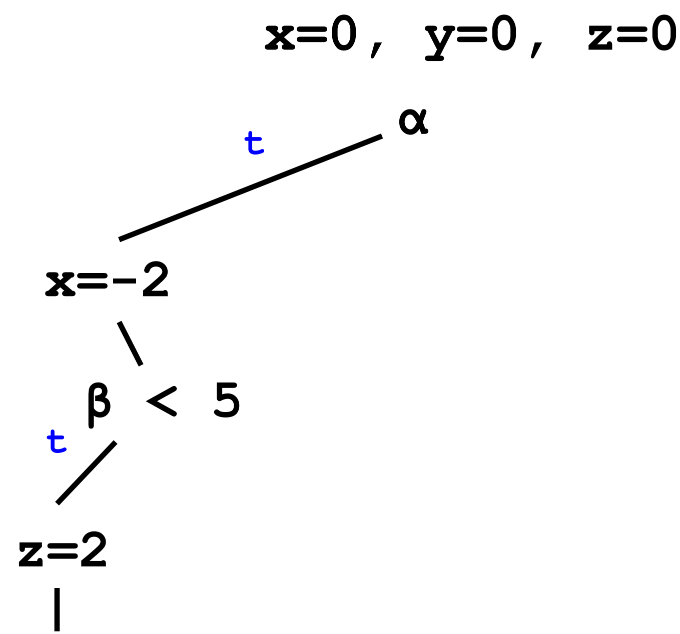
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



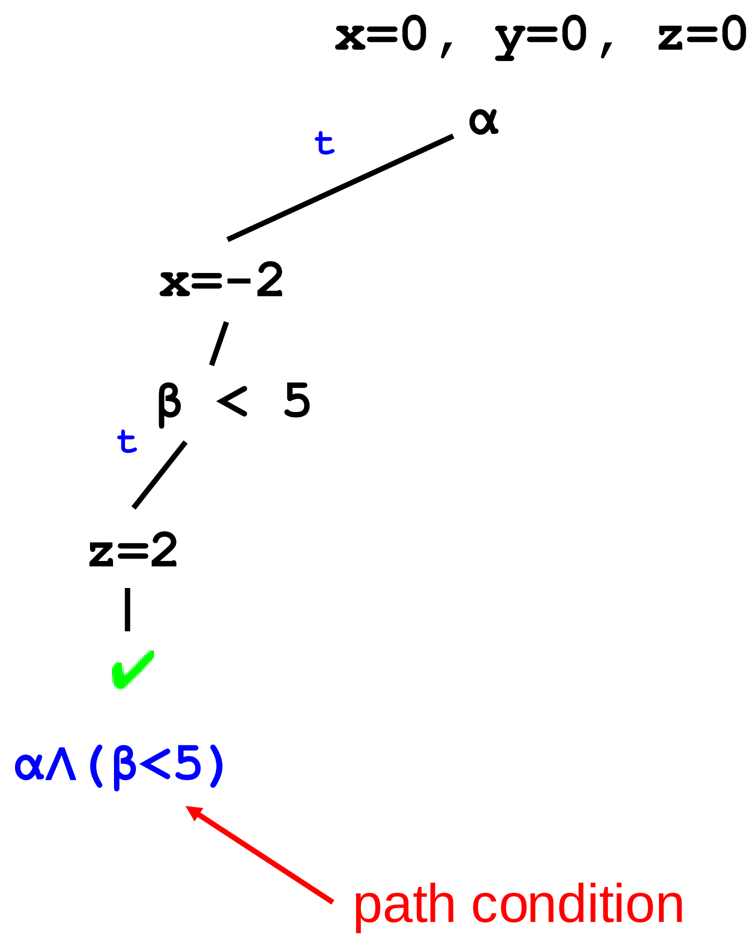
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



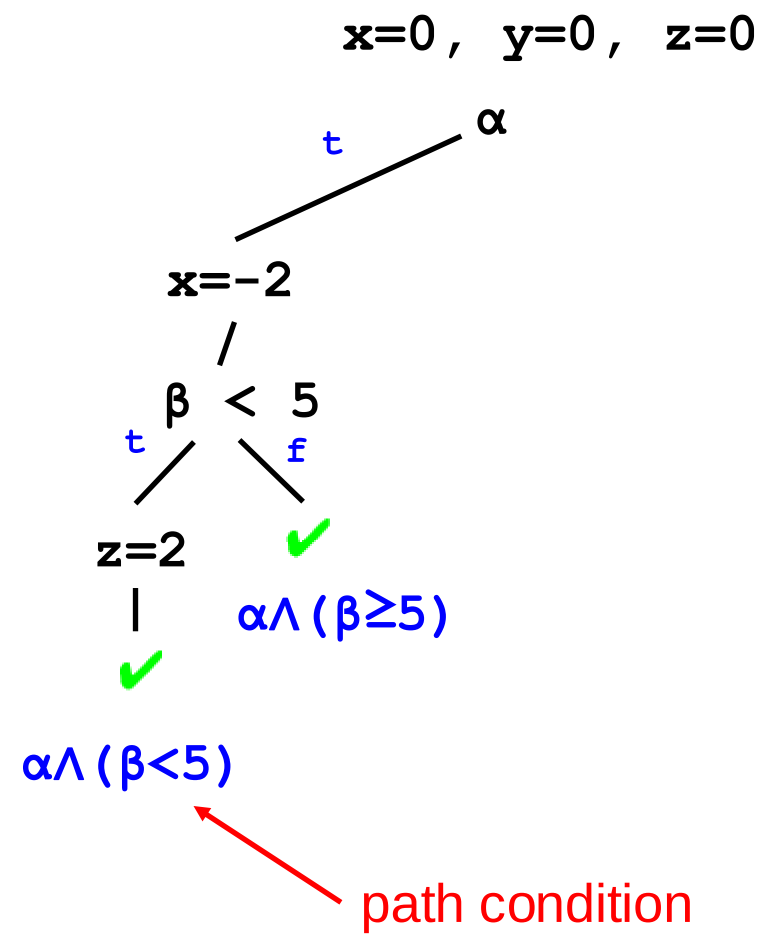
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                  // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



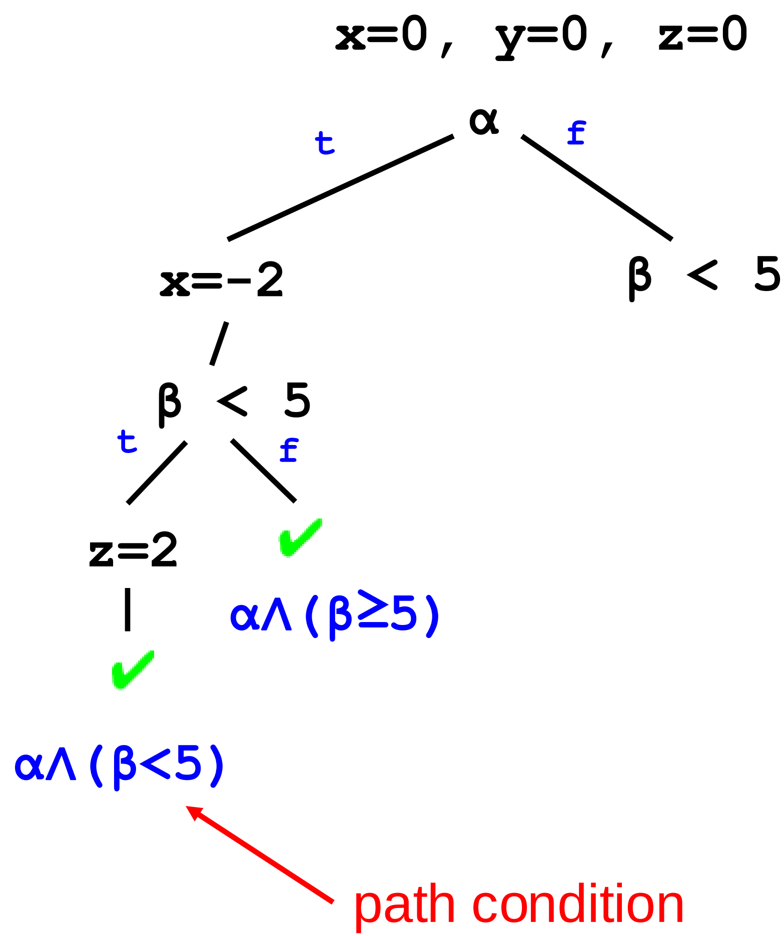
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



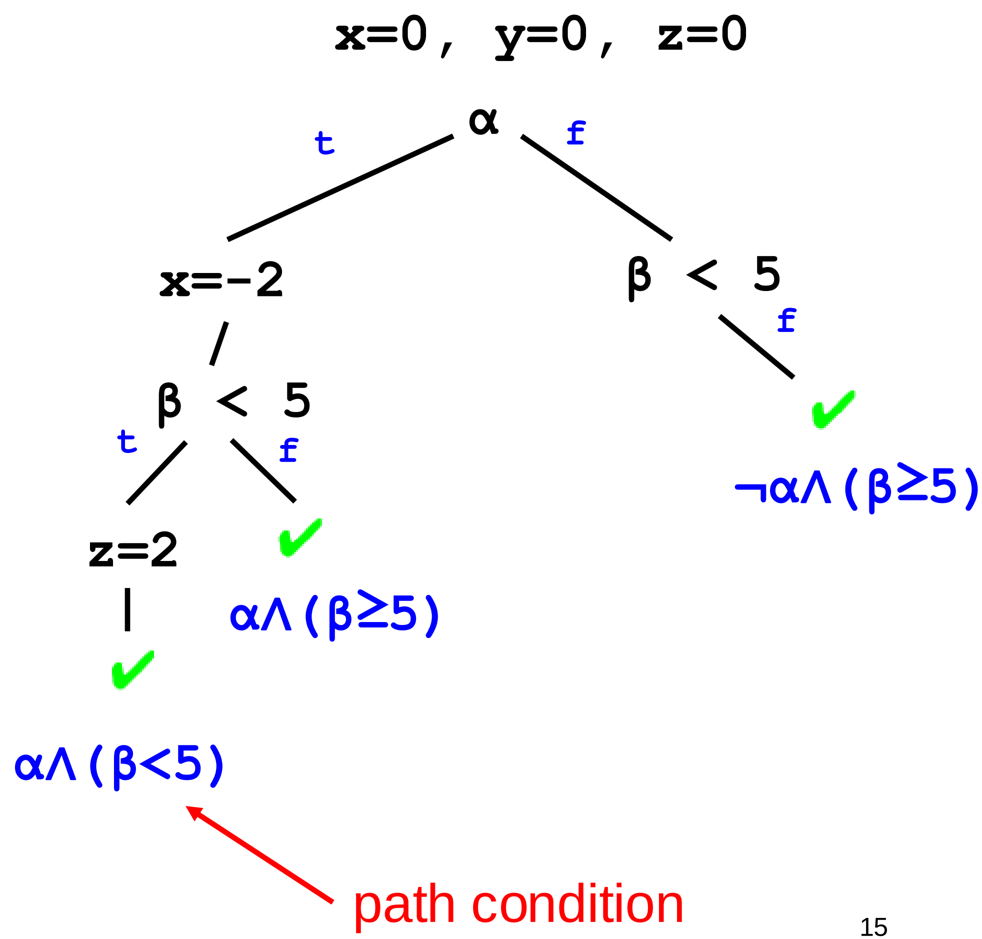
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                  // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



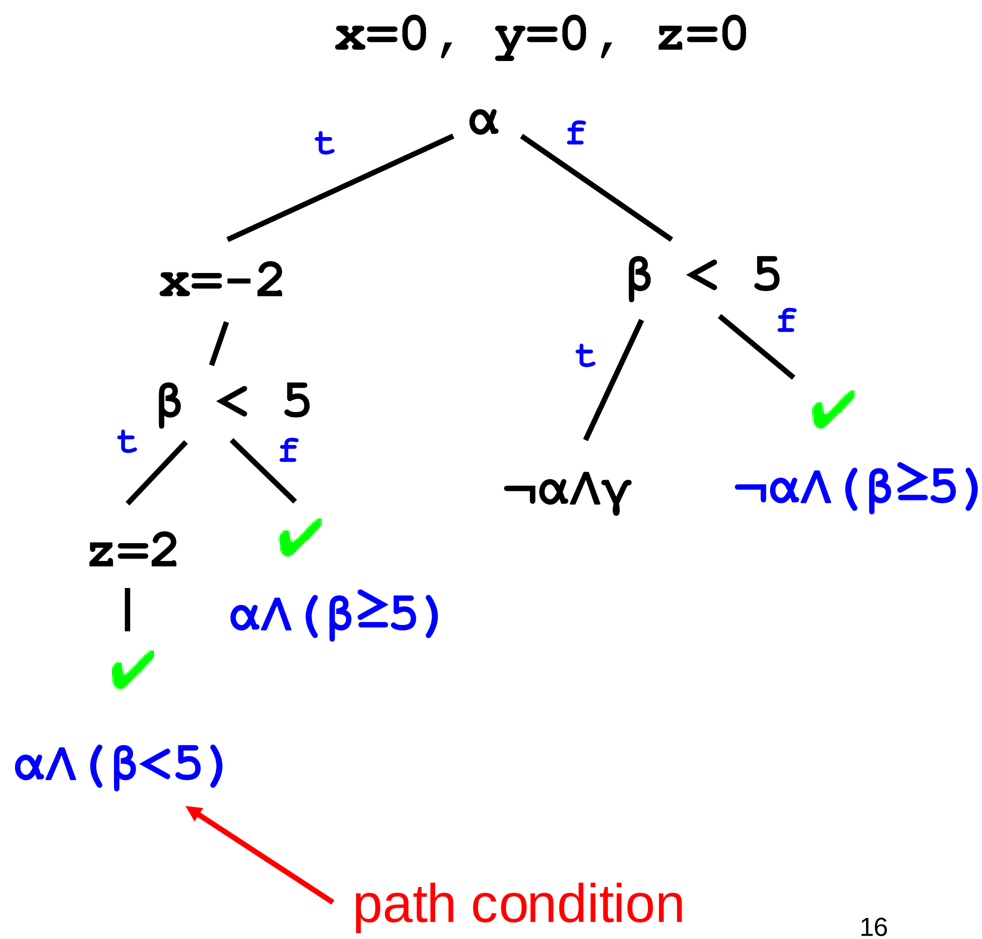
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



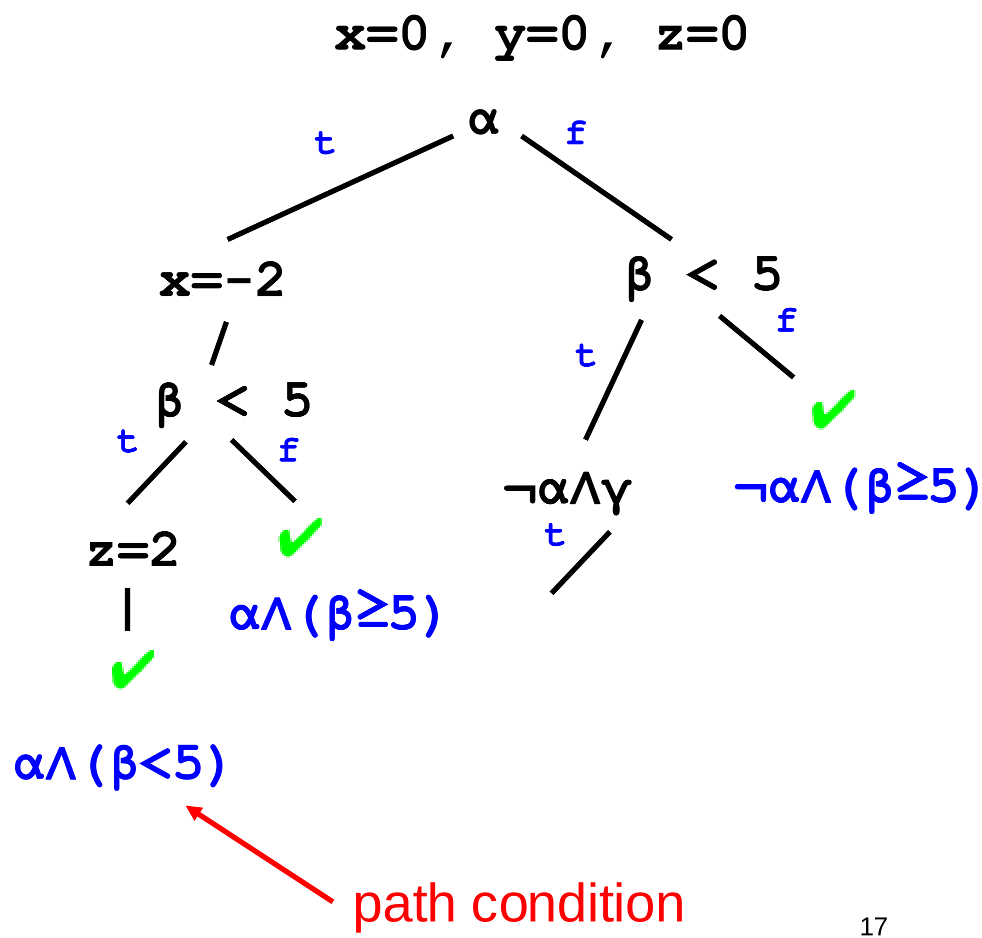
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                  // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



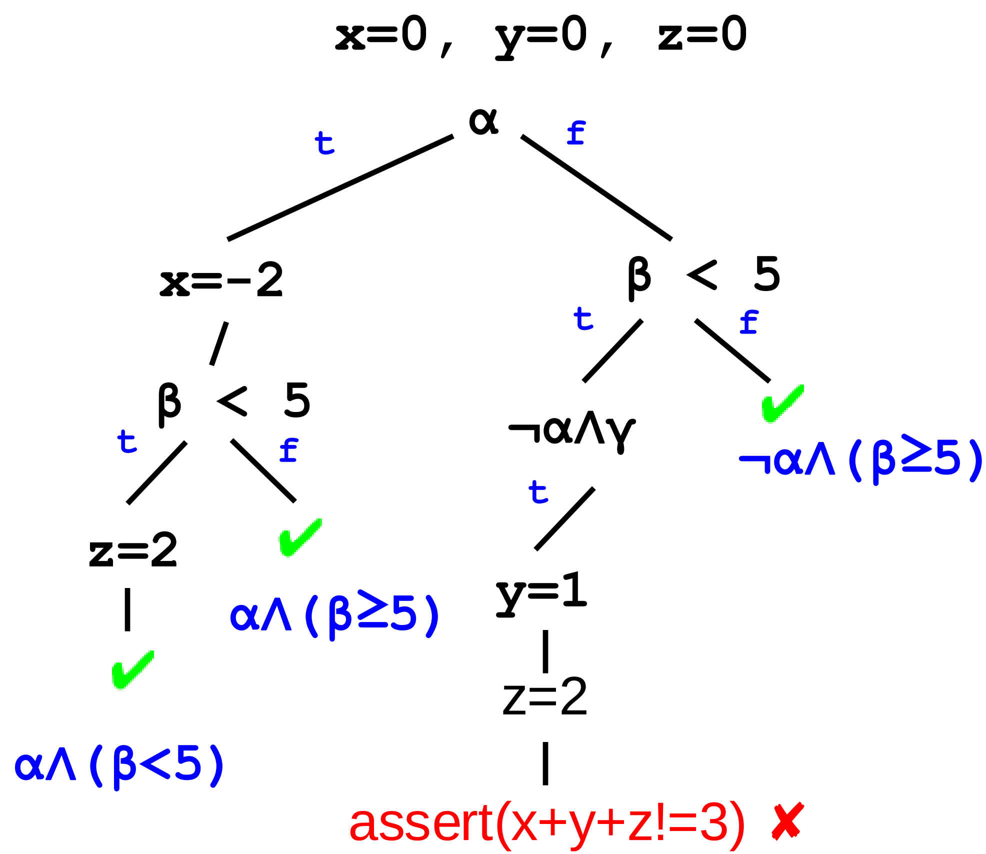
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                  // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



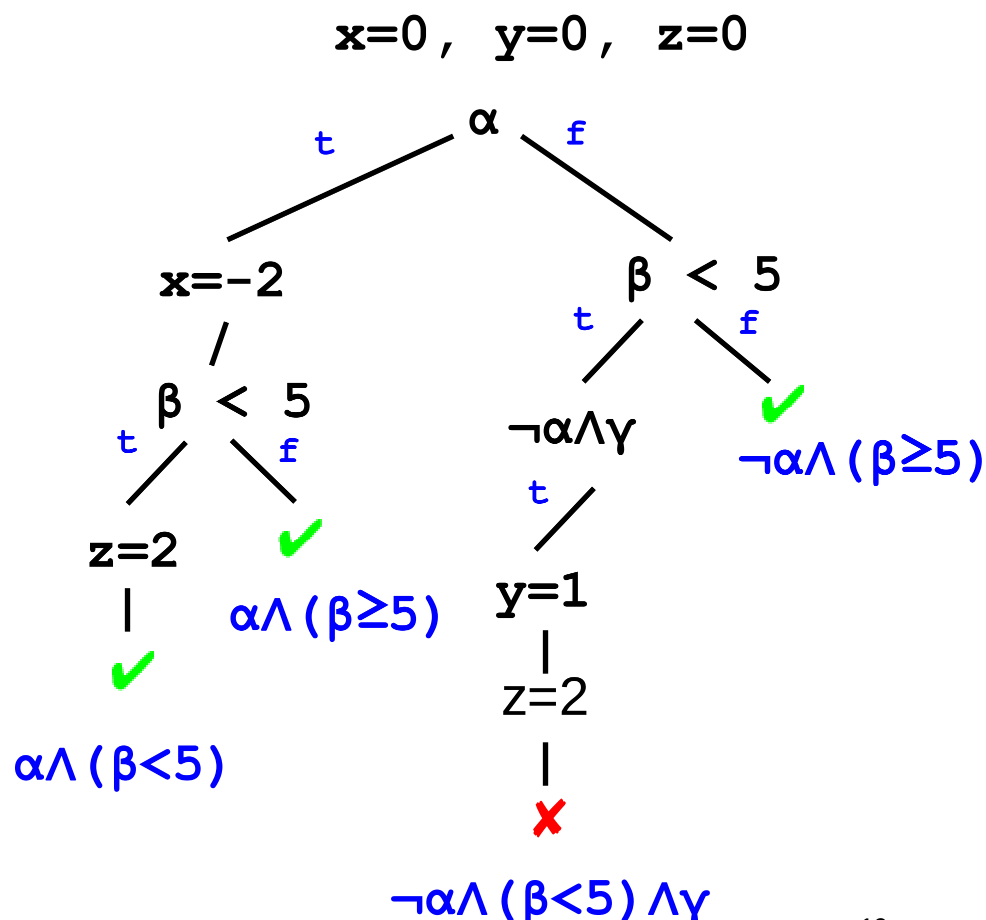
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



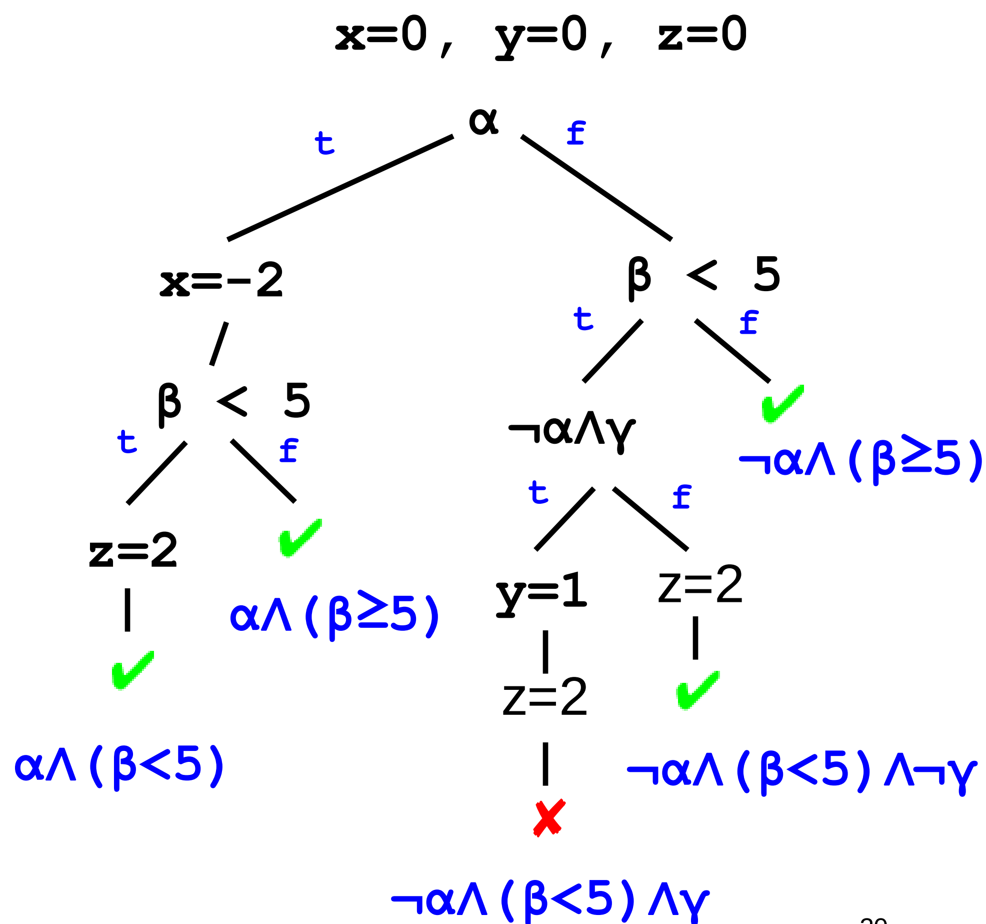
Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Insights

- Execute the program on symbolic values.
- Symbolic state maps variables to symbolic values.
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.
- All paths in the program form its execution tree, in which some paths are feasible, and some are infeasible.

Insights

- Each symbolic execution path stands for *many* program runs
  - In fact, exactly the set of runs whose concrete values satisfy the path condition

- Thus, we can cover a lot more of the program's execution space than testing can

Practical Issues

- Loops and recursion: infinite execution trees
- Path explosion: exponentially many paths
- Heap modeling: symbolic data structures and pointers
- Solver limitations: dealing with complex path conditions
- Environment modeling: dealing with native / system / library calls

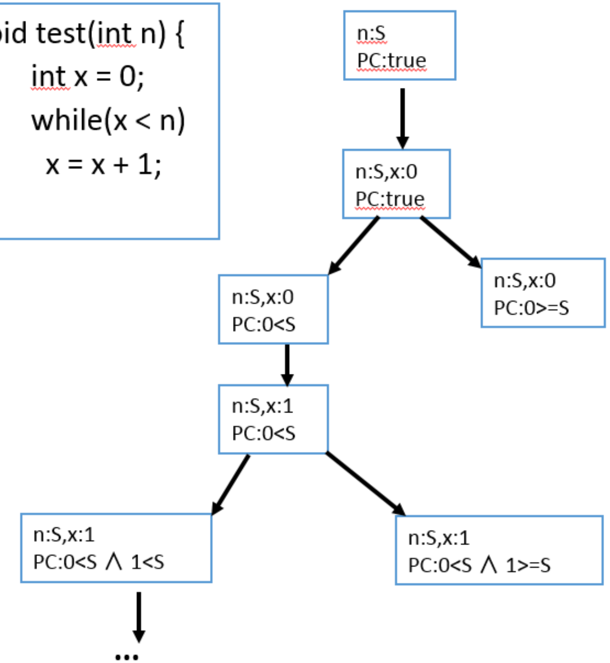
Loops and Recursion

- Dealing with infinite execution trees:
  - Finitize paths by unrolling loops and recursion (bounded verification)
  - Finitize paths by limiting the size of path conditions (bounded verification)
  - Use loop invariants (verification)

Example Code

```
void test(int n) {
    int x = 0;
    while(x < n)
      x = x + 1;
}
```

Infinite symbolic execution tree



Path Explosion

- Achieving good coverage in the presence of exponentially many paths:
  - Select next branch at random
    - sacrifice completeness, but still better than ad-hoc testing/fuzzing
  - Select next branch based on coverage
  - Interleave symbolic execution with random testing

Heap modeling

- Modeling symbolic heap values and pointers
  - Bit-precise memory modeling with the theory of arrays (EXE, Klee, SAGE)
  - Lazy concretization (JPF)
  - Concolic lazy concretization (CUTE)

```
void f(int i, int j){
  int a[1] = 0;
  if(i > 1 || j > 1)
    return;
  a[i] = 5;
  assert(a[j] != 5);
}
```

What values of i and j to make the assert() fail?

Solver limitations

- Reducing the demands on the solver:
  - On-the-fly expression simplification
  - Incremental solving
  - Solution caching and reuse
  - Substituting concrete values for symbolic in complex path conditions (CUTE)

Environment modeling

- The software components must interact with the external environments
  - Dealing with system / native / library calls:

- The symbolic executor must model the environment
  - Partial state concretization
  - Manual *models* of the environment (Klee)
    - file systems
    - network stack

Recent Success

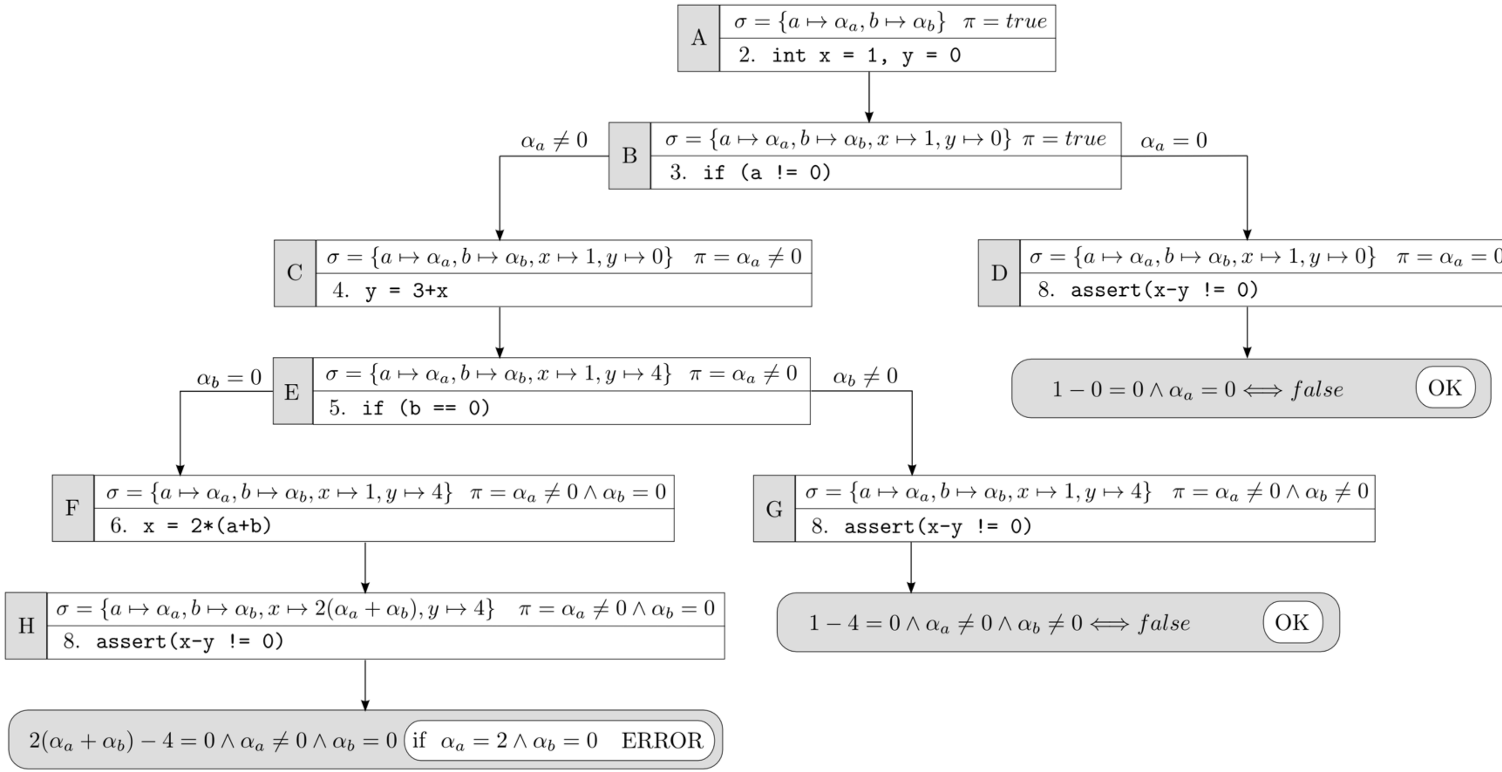
- SAGE
  - Microsoft internal tool
  - Symbolic execution to find bugs in file parsers
    - - E.g.,JPEG, DOCX, PPT,etc
  - Cluster of *n* machines continually running SAGE
- KLEE
  - Open source symbolic executor
  - Runs on top of LLVM
  - Has found lots of problems in open-source software
- Angr, BAP/Mayhem,Pex, jCute, Java PathFinder

Summary

- Symbolic execution is a bug finding technique based on automated theorem proving:
- Evaluates the program on symbolic inputs, and a solver finds concrete values for those inputs that lead to errors.
- Many success stories in the open-source community and industry.

Demo

```
void foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0)
            x = 2*(a+b);
    }
    assert(x-y != 0);
}
```


A
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$ $\pi = true$
2. int x = 1, y = 0
B
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = true$
3. if (a != 0)
$\alpha_a \neq 0$
$\alpha_a = 0$
C
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a \neq 0$
4. y = 3+x
D
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a = 0$
8. assert(x-y != 0)
$1 - 0 = 0 \wedge \alpha_a = 0 \iff false$
OK
E
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0$
5. if (b == 0)
$\alpha_b = 0$
$\alpha_b \neq 0$
F
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
6. x = 2*(a+b)
G
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b \neq 0$
8. assert(x-y != 0)
$1 - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b \neq 0 \iff false$
OK
H
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
8. assert(x-y != 0)
$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$
if $\alpha_a = 2 \wedge \alpha_b = 0$ ERROR

Angr Example

```
https://github.com/cmsc433/Fall2024_public/tree/ma
in/code/symbolic
```

Generate Tests to Cover All Branches

```
f2(a,b,c){
  x = 1
  y = 0
  if a != 0 :
    y = x + 3
    if b == y :
      if c == a :
        x = 4 * b
      else:
        x = 8 * (a + b)
    else:
      if c == (4 + a) :
        x = 4 * b
      else:
        x = 2 * (a + b)
  else:
    y = x + 10
    if b == y :
       x = 3 * (a + b)
    else:
       if c == x :
          x = 4 * (a + b)
       else:
          x = 4 * a
  return x;
}
```

Generate Tests to Cover All Branches

```
f2(a,b,c){
  x = 1
  y = 0
  if a != 0 :
    y = x + 3
    if b == y :
      if c == a :
        x = 4 * b
      else:
        x = 8 * (a + b)
    else:
      if c == (4 + a) :
        x = 4 * b
      else:
        x = 2 * (a + b)
  else:
    y = x + 10
    if b == y :
       x = 3 * (a + b)
    else:
       if c == x :
          x = 4 * (a + b)
       else:
          x = 4 * a
  return x;
}
```

Symbolic execution path conditions:

```
a == 0,b == (1 + 10)
a != 0,b == (1 + 3)
a == 0,b != (1 + 10),c != 1
a == 0,b != (1 + 10),c == 1
a != 0,b != (1 + 3),c != a
a != 0,b != (1 + 3),c == a
```