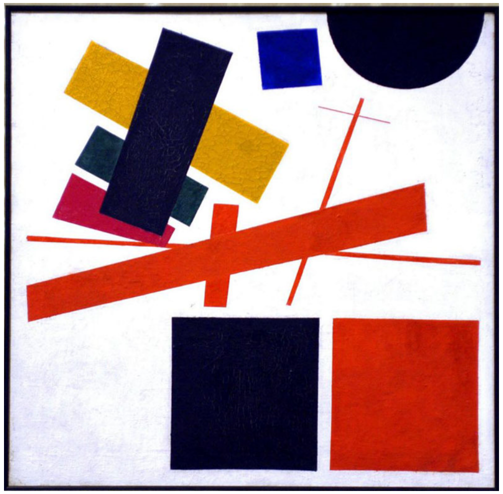
CPEN 422
Software Testing and Analysis



Symbolic Execution

Learning Objectives

- Understand the goal and implication of symbolically executing programs
- Learn how we can use symbolic execution to support branch coverage
- Understand the limitations of symbolic execution
- Learn concolic testing and its applications

Problem

– How can we automatically test foo and find the error?

```
void foo(int x) {
  if (x != 9999) {
      ...
  } else {
    ERROR;
  }
```

Random testing?

- How can we automatically test foo and find the error?
- Answer: Invoke foo with randomly generated concrete values.
- There are 2^{32} possible inputs.
- Probability of reaching ERROR is only $1 / 2^{32}$.
- Redundant inputs.

```
void foo(int x) {
  if (x != 9999) {
      …
  } else {
    ERROR;
  }
```

Reasoning?

- How can we automatically test foo and find the error?
- x is a primitive type variable and only accepts integers.
- to reach the ERROR, x must be !(!= 9999).

- Answer x = 9999

```
void foo(int x) {
  if (x != 9999) {
      …
  } else {
    ERROR;
  }
}
```

Symbolic Execution

- Analysis of programs by tracking **symbolic** rather than **actual** values -> Static Analysis
- Is used to reason about all the **inputs** that take the same path through a program
- Builds **predicates** that characterize
  - Conditions for executing paths
  - Effects of the execution on program state

Predicate

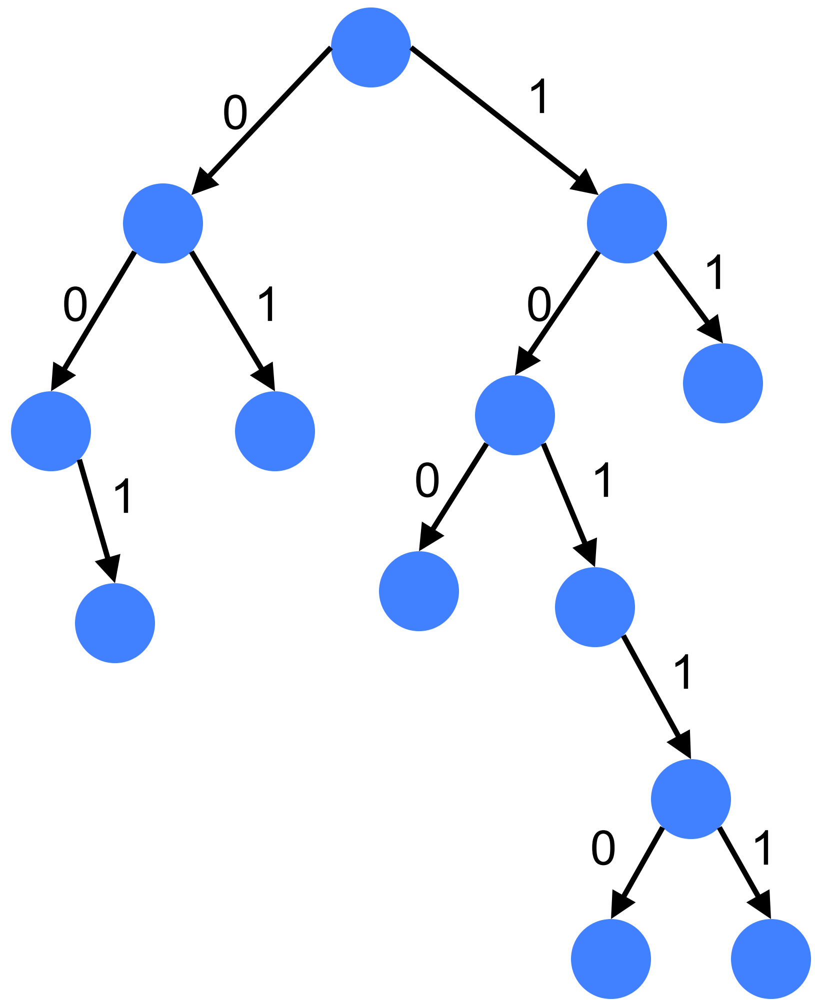
"Boolean-valued statement that may be true or false depending on the values of its variables"

```
{x|p(x)} = {x|x is a positive
               integer less than 5}
```

```
p(2)    T
p(6)    F
```

Symbolic Execution: Motivation

- Execution paths of a program can be seen as a binary tree with possibly infinite depth
  - Computation tree
- Each node represents the execution of a "if then else" statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs



Symbolic Execution

- Uses symbolic values for input variables.
- Builds predicates that characterize the conditions under which execution paths can be taken.
- Collects **symbolic path constraints**.
- Uses theorem prover (**constraint solver**) to check if an answer exists and the branch can be taken.
- **Negates** path constraints to take the other side of the condition (branch)

Constraint Solving

- Mathematically determine if a set of constraints can be solved.

- Famous constraint solvers:
  - Z3, CVC, lp solver

Constraint Solving

- Primarily linear equations:
  - if (3 + x < 6) { y = x * 2; } else { … }
  - Constraint to solve: **(3 + x < 6)**
  - what is x?
  - **Constraint Solver: (x < 3) -> x = 2**
  - **(IF branch covered)**

  - Negate constraint: (3 + x >= 6)
  - **Constraint Solver: x >= 3 -> x = 4**
  - **(Else branch covered)**

String Constraint Solving

```
void main(char[] in){
 int count=0;
 if (in[0] == 'b')
   count++;
 if (in[1] == 'a')
   count++;
 if (in[2] == 'd')
   count++;
 if (count == 3)
   ERROR;
}
```

(in[0]!= ’b’)^ (in[1]=’a’)
Solver: **`xaz’**

Hampi (string solver)

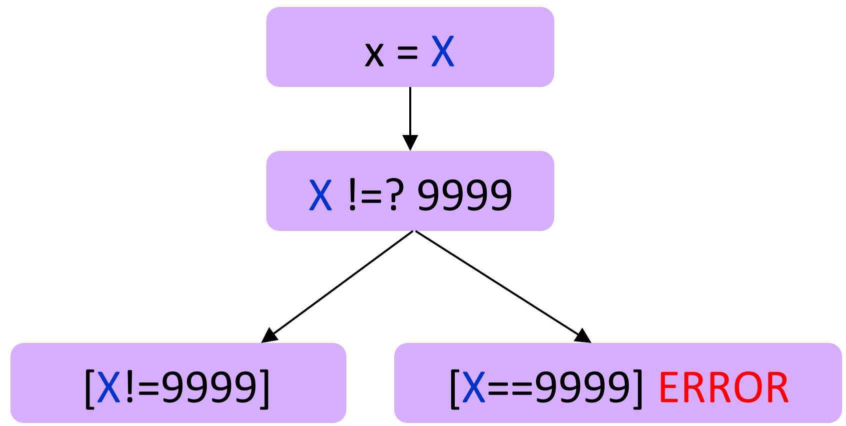
Applications of Symbolic Execution

- Guiding the test input generation to cover all branches
- Useful for identifying infeasible program paths
- Formal verification of mission-critical software
- Security testing

Symbolic Execution: foo

```
void test_function (int x) {
    if (x != 9999) {
        ...
    } else {
        ERROR;
    }
}
```

Symbolic state



Example: binary search

- How does symbolic execution identify an alternative execution path?
  - Example: Binary Search.

```c
char * binarySearch( char *key, char *dictKeys[ ], char *dictValues[ ],
                              int dictSize) {

   int low=0;
   int high = dictSize - 1;
   int mid;
   int comparison;

   while (high >=low) {
     mid = (high + low) / 2;
     comparison = strcmp( dictKeys[mid], key );
     if (comparison < 0) {
       /* dictKeys[mid] too small; look higher */
          low=mid+1;
     } else if ( comparison > 0) {
        /* dictKeys[mid] too large; look lower */
           high=mid-1;
     } else {
        /* found */
          return dictValues[mid];
     }
   }
   return 0; /* null means not found */
  }
```

sorted input array

returns corresponding value from dictValues or null if key does not appear in dictKeys.

Symbolic Execution: Example

Concrete values

before

low	12
high	15
mid	–

mid = (high + low) / 2;

after

low	12
high	15
mid	13

Symbolic values

before

low	L
high	H
mid	–

mid = (high + low) / 2;

after

low	L
high	H
mid	$\frac{L+H}{2}$

Note: No symbol is required for constants.

Symbolic Execution: Example

The symbolic state after entering one loop iteration.

```
while (high >=low) {
  mid = (high + low) / 2;
  comparison = strcmp( dictKeys[mid], key );
  if (comparison < 0) {
    /* dictKeys[mid] too small; look higher
       low=mid+1;
  } else if ( comparison > 0) {
     /* dictKeys[mid] too large; look lower
       high=mid-1;
  } else {
     /* found */
       return dictValues[mid];
  }
}
```

Symbolic state

$$\mathrm{low} = 0$$

$$\wedge \quad \mathrm{high} = \frac{H-1}{2} - 1$$

$$\wedge \quad \mathrm{mid} = \frac{H-1}{2}$$

Symbolic Execution: Example

Assuming a *True* outcome (leading to a second iteration of the loop), the loop condition becomes a constraint in the symbolic state immediately after the while test.

```
while (high >=low) {
  mid = (high + low) / 2;
  comparison = strcmp( dictKeys[mid], key );
  if (comparison < 0) {
    /* dictKeys[mid] too small; look higher
       low=mid+1;
  } else if ( comparison > 0) {
     /* dictKeys[mid] too large; look lower
        high=mid-1;
  } else {
     /* found */
       return dictValues[mid];
  }
}
```

Symbolic state

$$
\begin{aligned}
& \mathrm{low} = 0 \\
\wedge \quad & \mathrm{high} = \frac{H-1}{2} - 1 \\
\wedge \quad & \mathrm{mid} = \frac{H-1}{2} \\
\wedge \quad & \boxed{\frac{H-1}{2} - 1 \geq 0}
\end{aligned}
$$

Symbolic Execution: Example

Later, when we consider the branch assuming a *False* outcome of the test, the new constraint is negated.

```
while (high >=low) {
  mid = (high + low) / 2;
  comparison = strcmp( dictKeys[mid], key );
  if (comparison < 0) {
    /* dictKeys[mid] too small; look higher
       low=mid+1;
  } else if ( comparison > 0) {
     /* dictKeys[mid] too large; look lower
        high=mid-1;
  } else {
     /* found */
       return dictValues[mid];
  }
}
```

Symbolic state

$$\text{low} = 0$$

$$\wedge \quad \text{high} = \frac{H-1}{2} - 1$$

$$\wedge \quad \text{mid} = \frac{H-1}{2}$$

$$\wedge \quad \frac{H-1}{2} - 1 \geq 0$$

$$\wedge \quad \neg\left(\frac{H-1}{2} - 1 \geq 0\right)$$

or, equivalently, $\frac{H-1}{2} - 1 < 0$

Executing while (high >= low) {

Add an expression that records the condition for the execution of the branch (PATH CONDITION)

```
            before
            low = 0
and         high = (H-1)/2 -1
and         mid = (H-1)/2

while (high >= low){

            after
            low = 0
and         high = (H-1)/2 -1
and         mid = (H-1)/2
and         (H-1)/2 - 1 >= 0
... and not((H-1)/2 - 1 >= 0)
             (H-1)/2-1<0
             H < 3
```

if the TRUE branch was taken

if the FALSE branch was taken

Is the assert reachable?

```
int x, y;
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
   if (x – y > 0)
      assert(false);
}
```

Concrete state

x = 1, y = 0

1 >? 0

x = 1 + 0 = 1

y = 1 – 0 = 1

x = 1 – 1 = 0

0 – 1 >? 0

Symbolic state

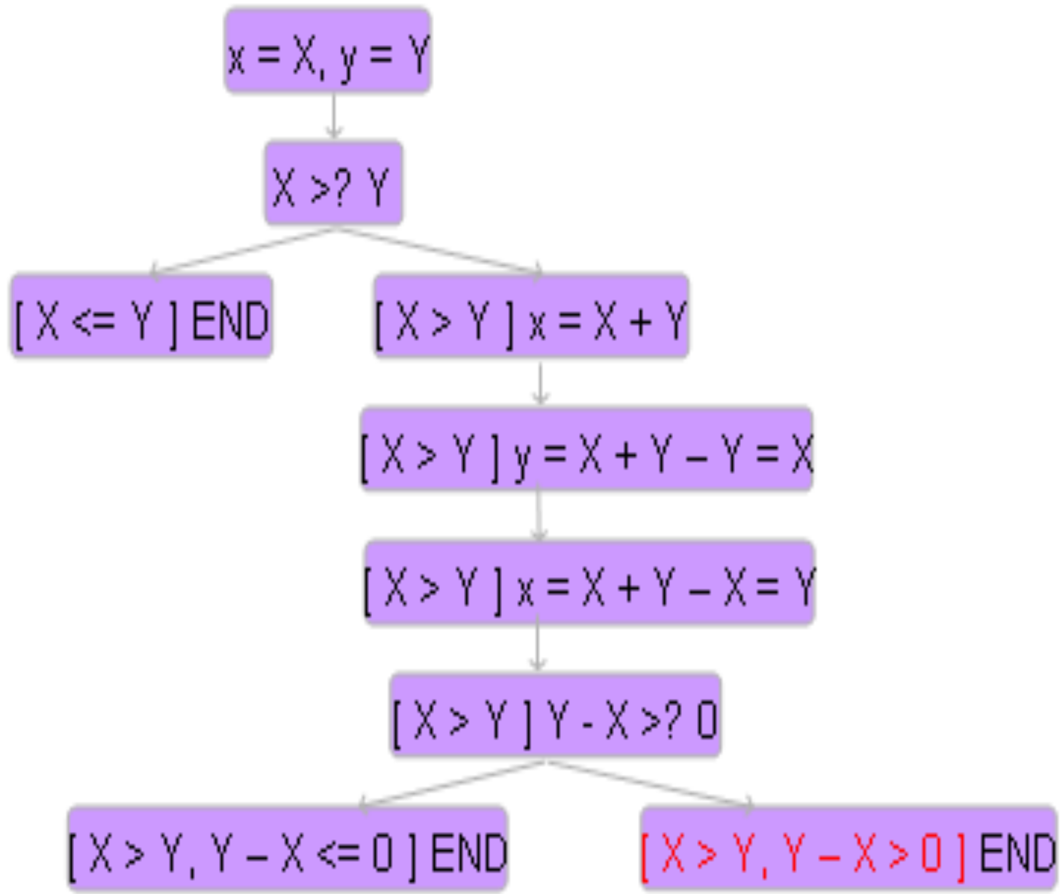
x = X, y = Y

X >? Y

?

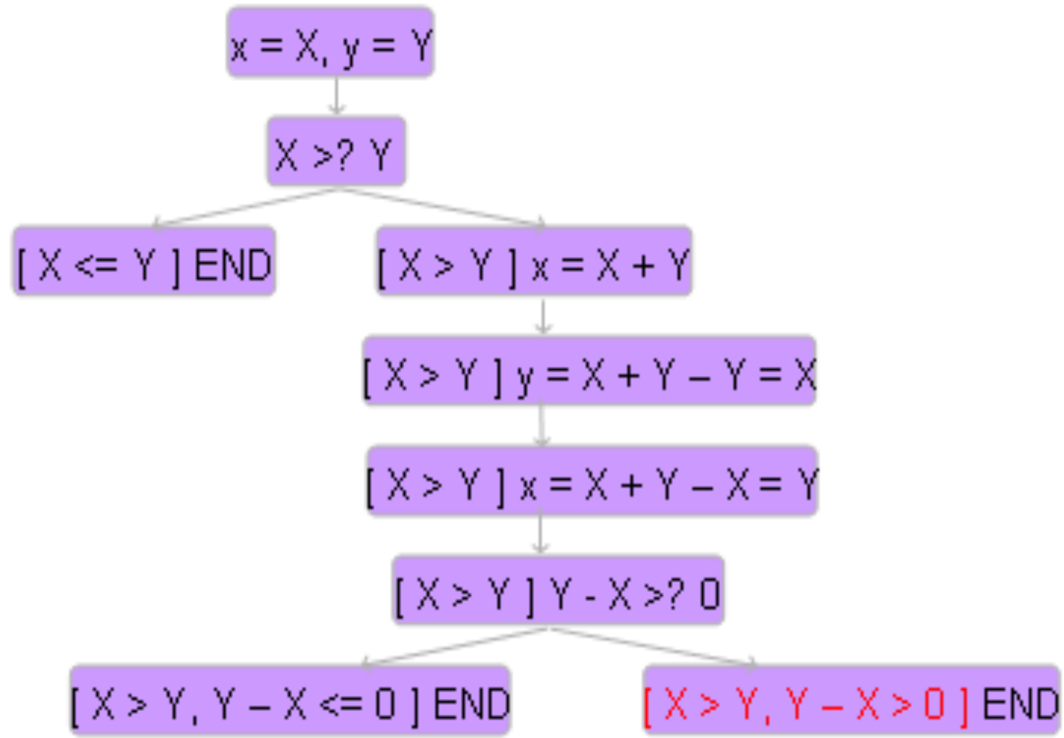
```
int x, y;
if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
    if (x – y > 0)
        assert(false);
}
```

Symbolic state



Symbolic Execution: Exercise

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
   }
```



– At 1), X<Y or X >=Y can hold so this represents a fork in the path, i.e., path condition is either X<Y or X>=Y
– 6) cannot be executed because X>Y & (Y–X>0) is FALSE

In-class Exercise: symbolically find the error

a = α, b = β, c = γ

x=0, y=0, z=0

```
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);
```

```
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);
```

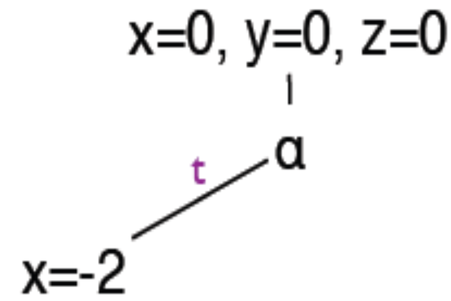
a = α, b = β, c = γ

x=0, y=0, z=0

α

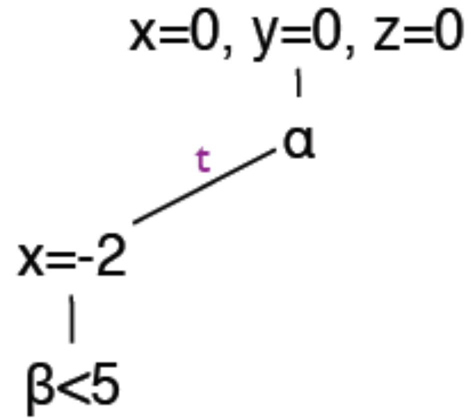
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```

a = α, b = β, c = γ

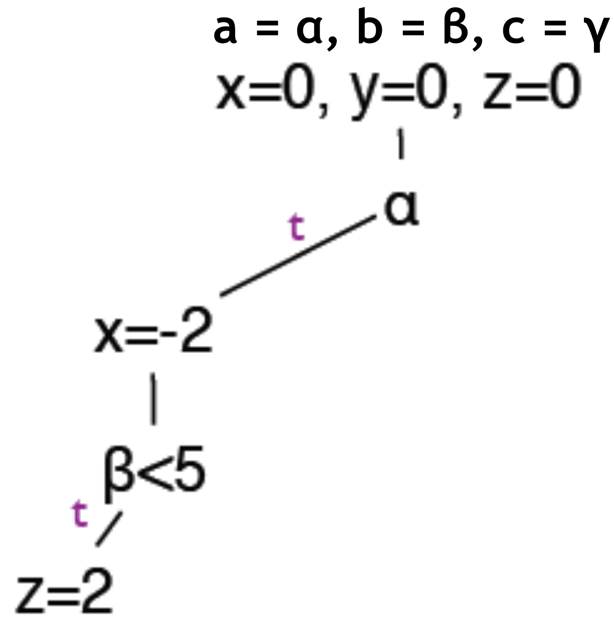


```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```

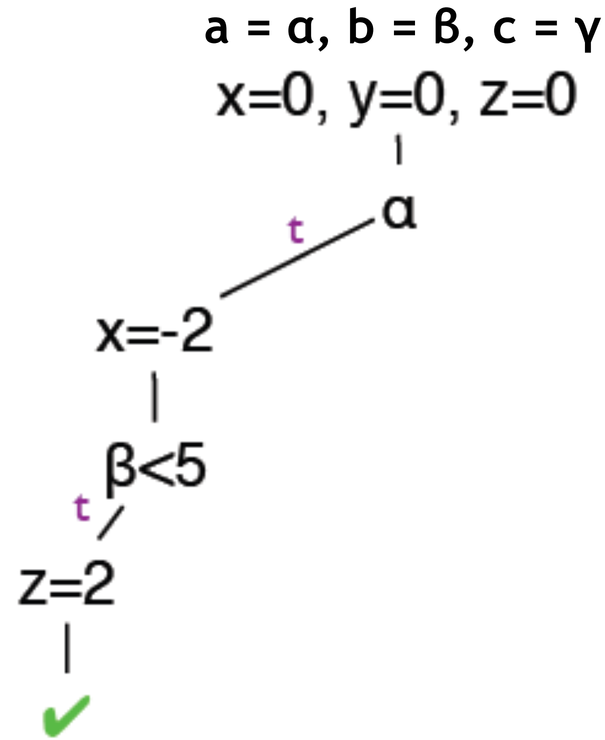
a = α, b = β, c = γ



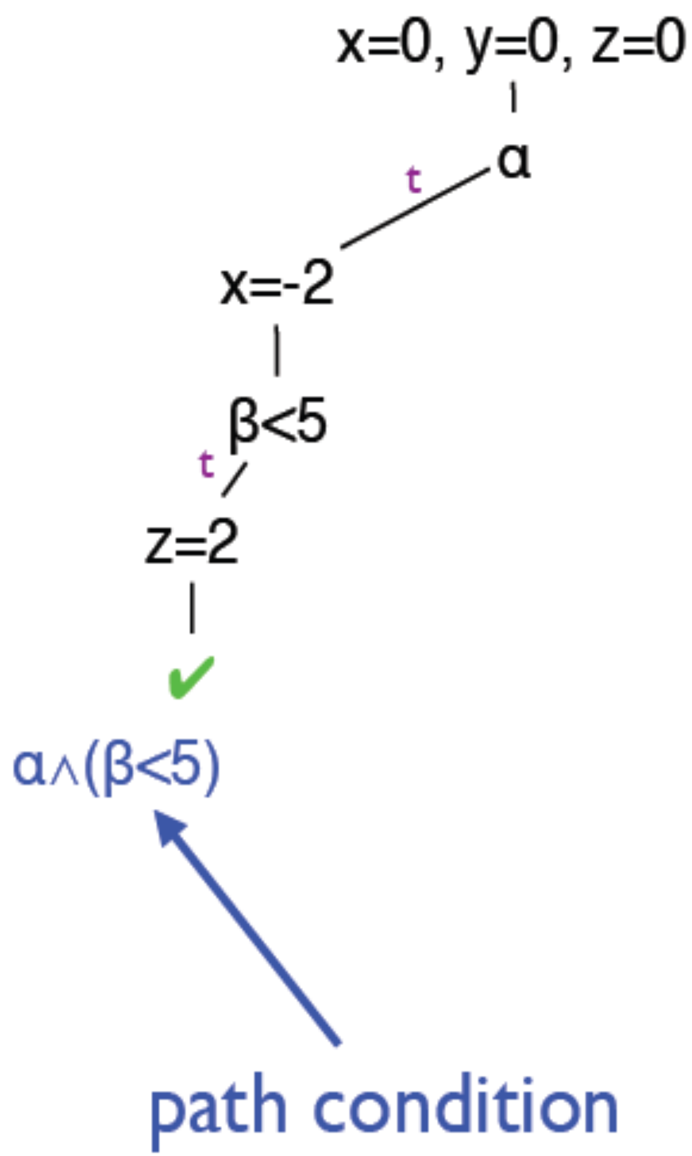
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



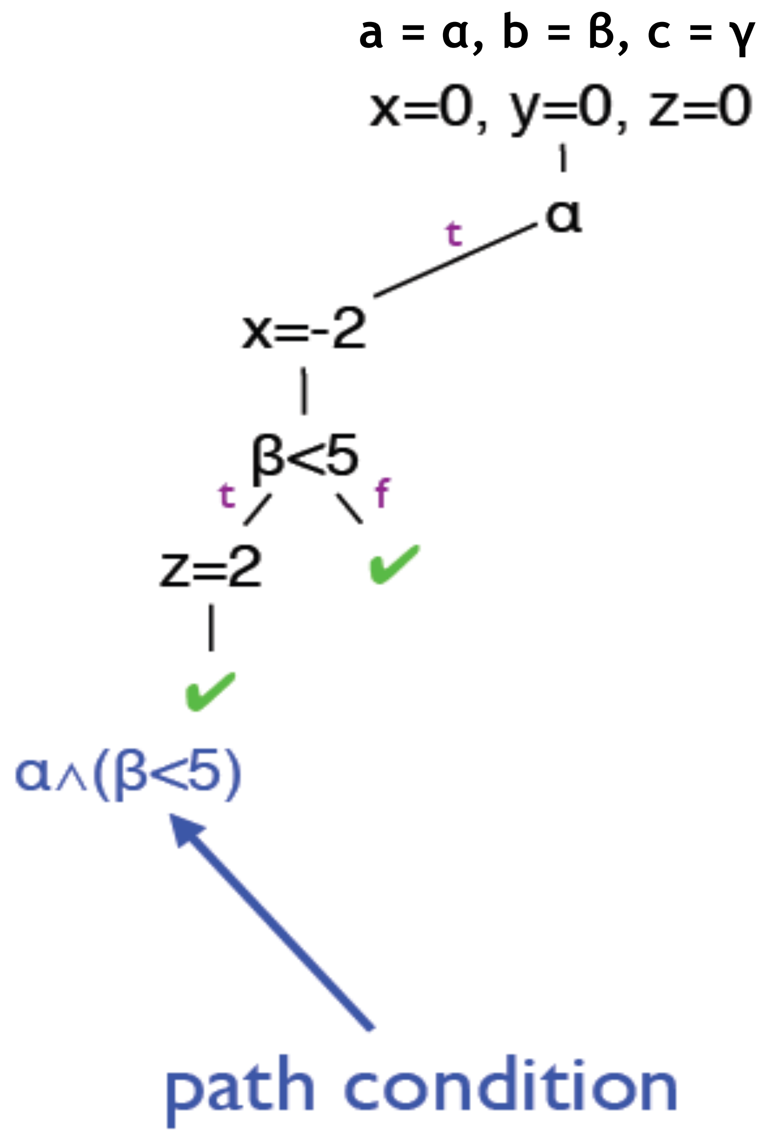
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



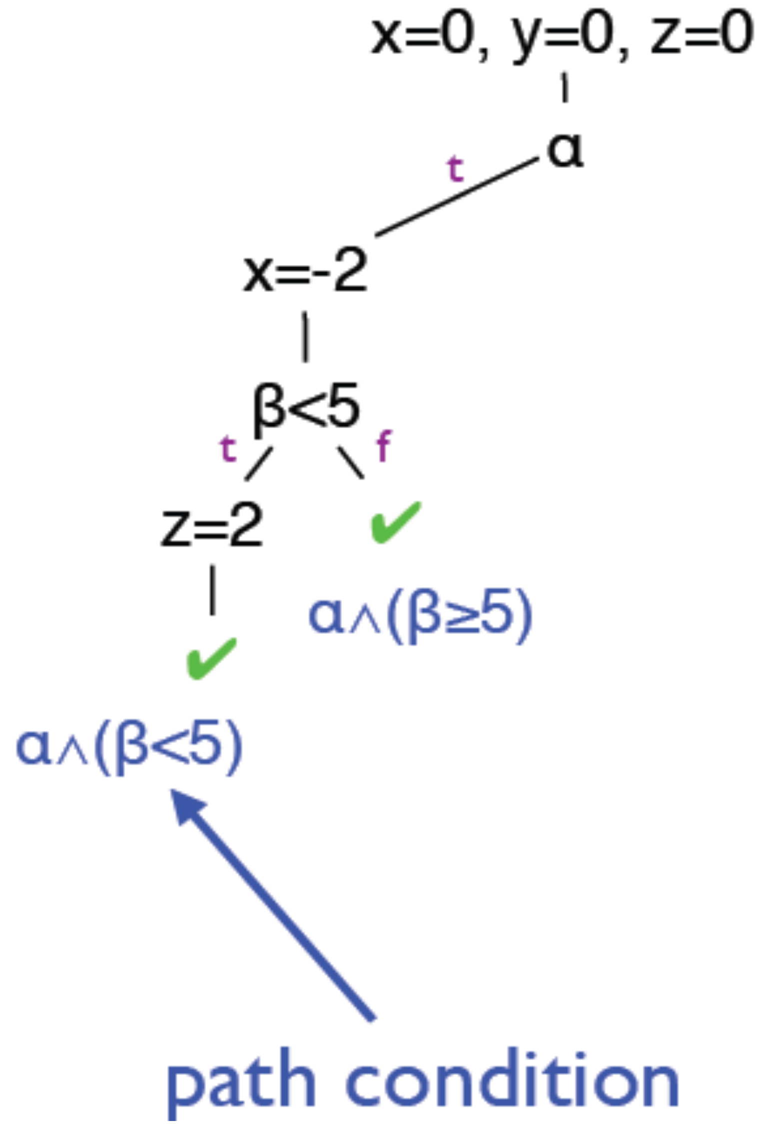
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



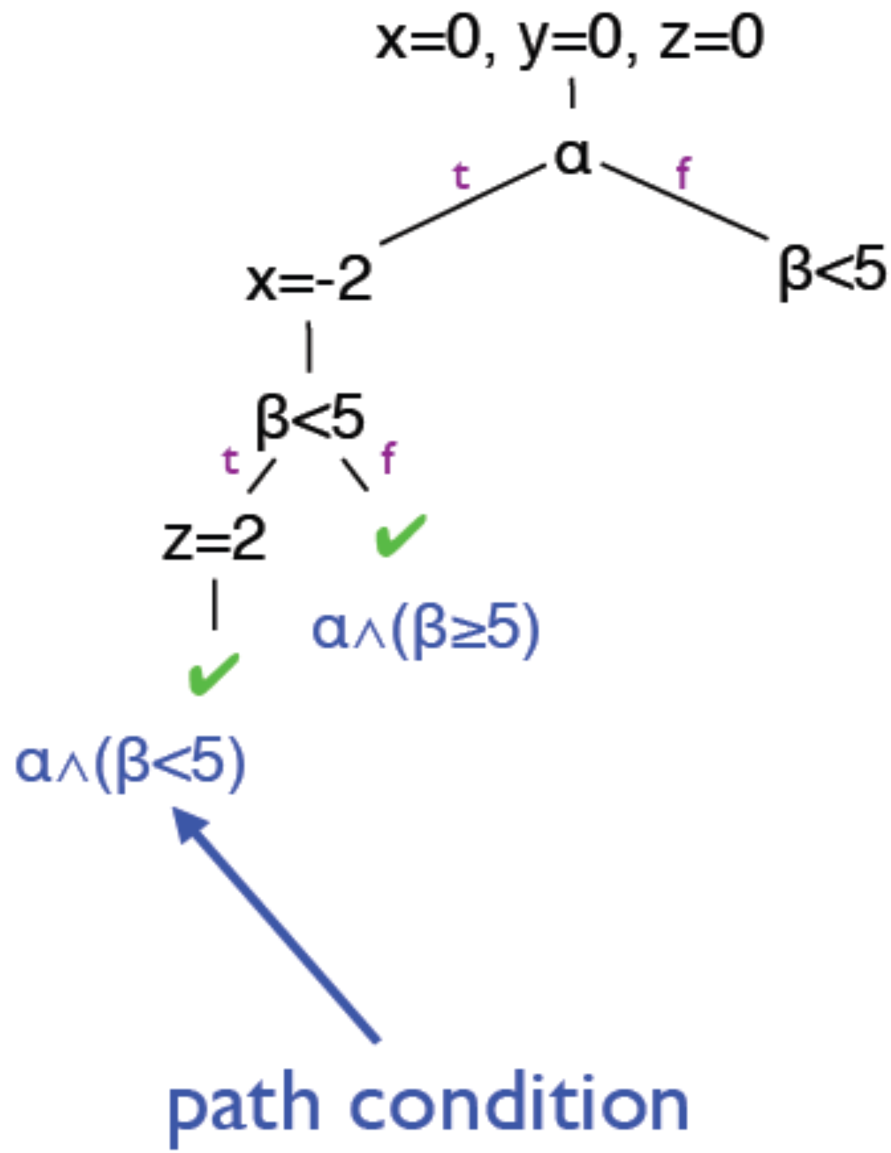
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



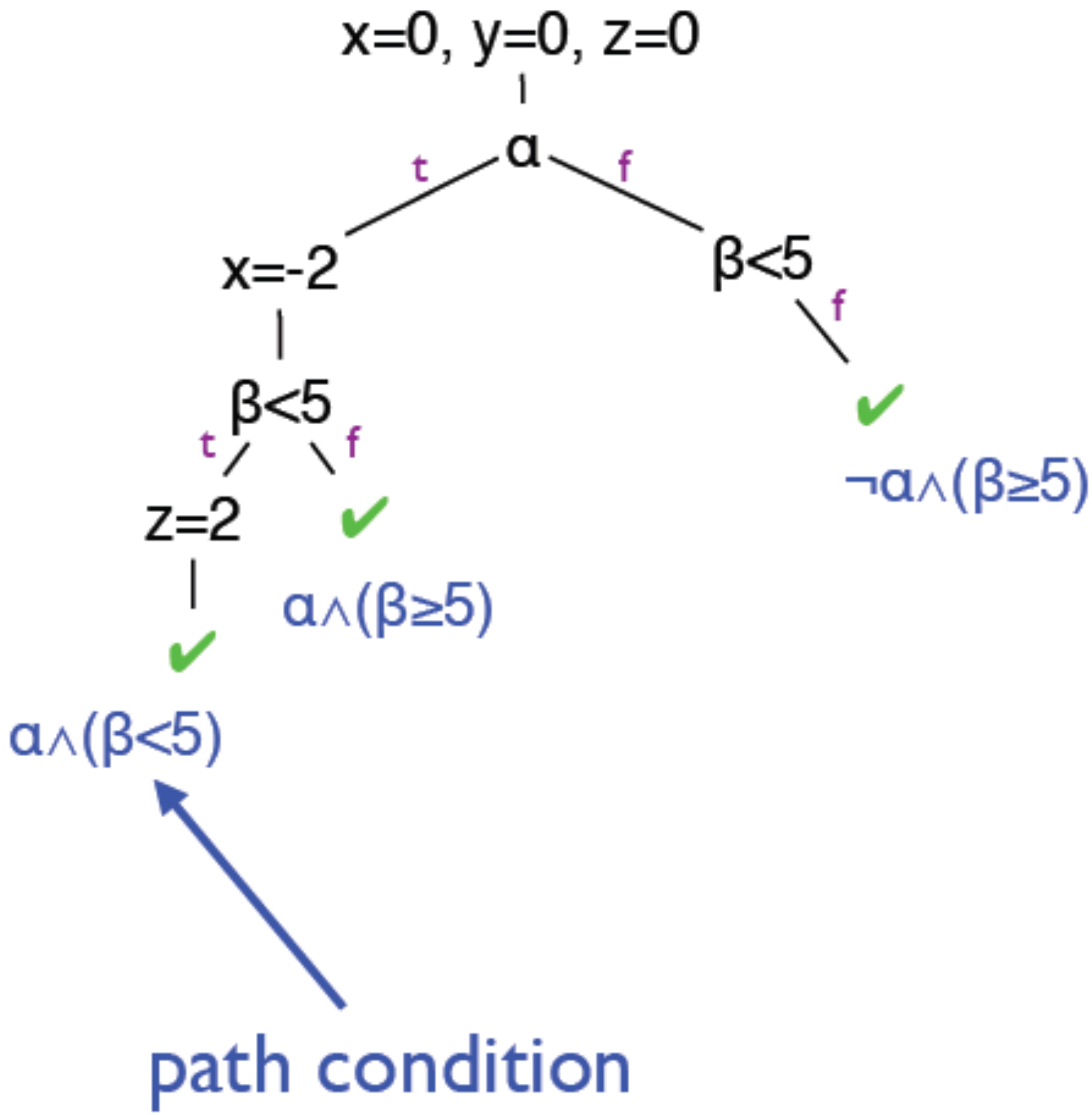
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



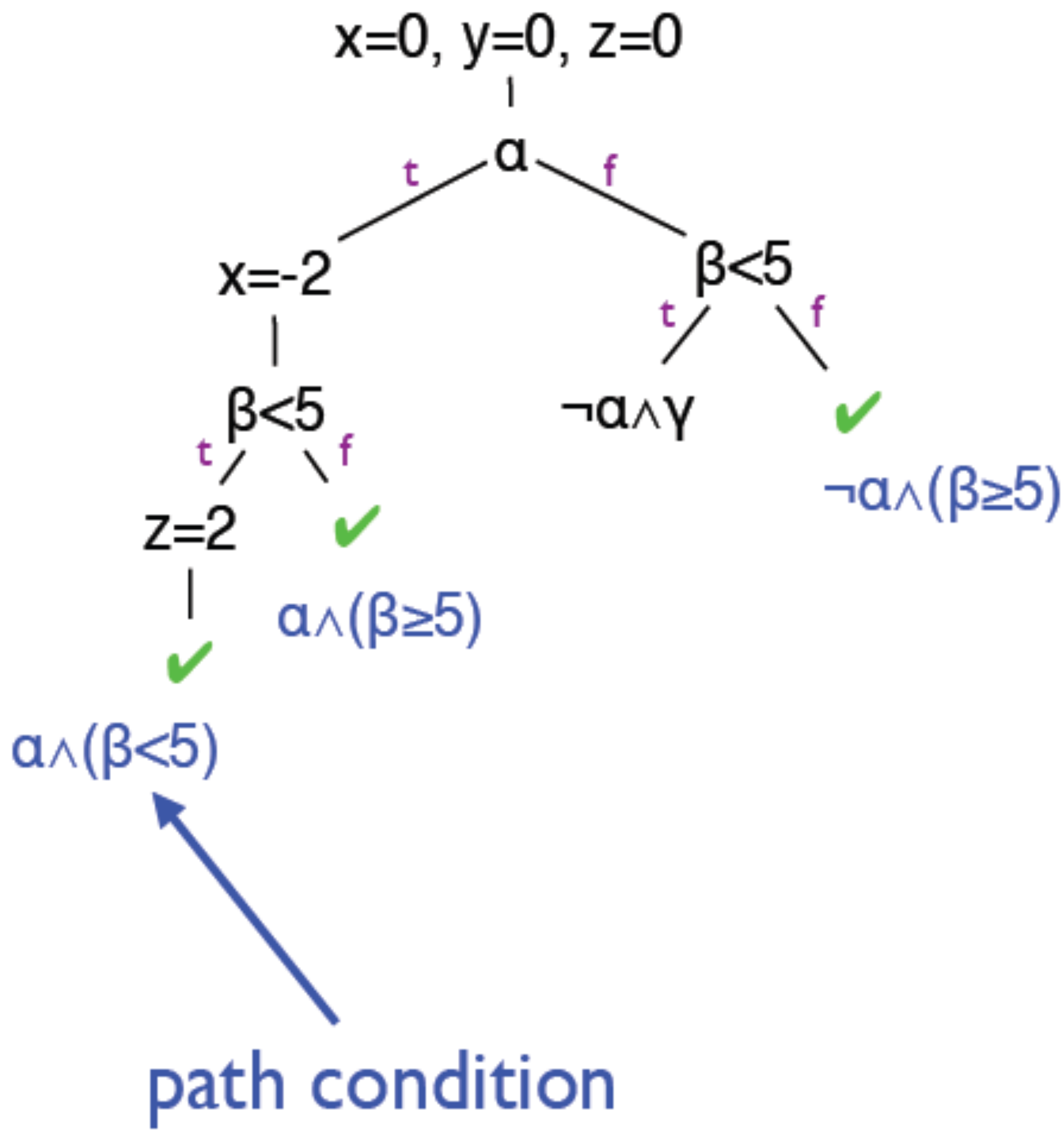
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



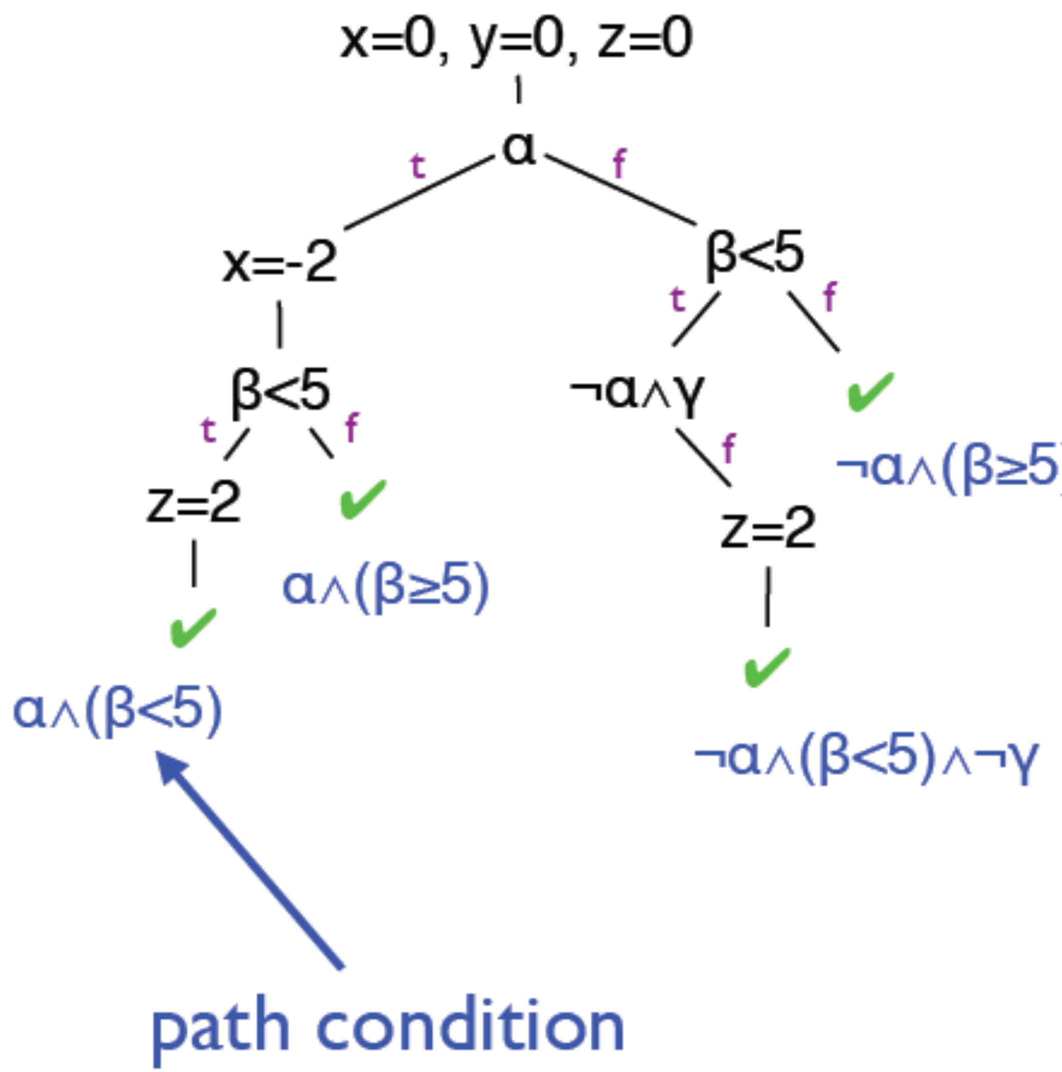
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



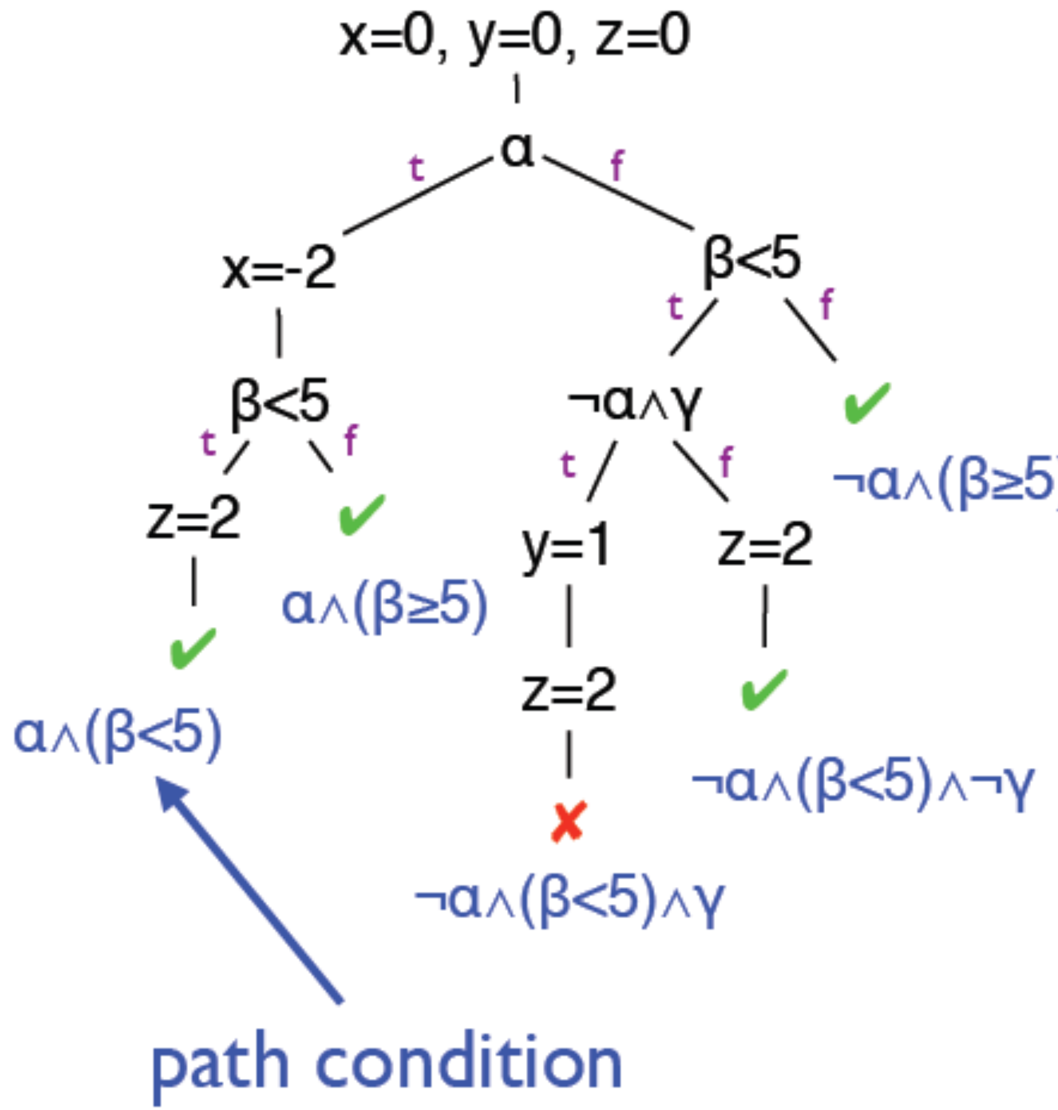
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



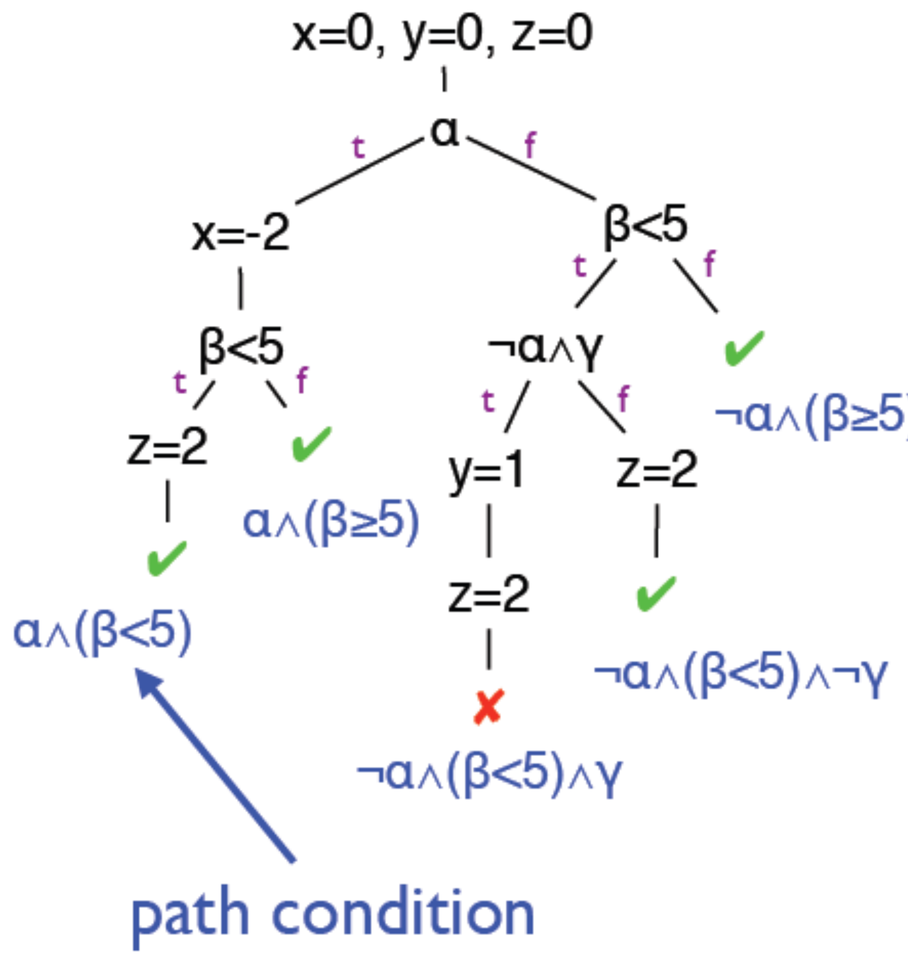
```
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c) { y = 1; }
  z = 2;
}
assert(x+y+z != 3);
```



```
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);
```



```
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);
```



a = false, b = 2, c = true -> x = 0, y = 1, z = 2 : Assert(0+1+2 != 3)

Symbolic Execution: Limitations

- Sym-ex is computationally infeasible for many programs
  – Tries to find every execution path, which is exponential in the number of branches.
  – Does not scale for large programs.

Symbolic Execution: Limitations

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

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y
- solve x*x*x+ 3*x*x+9 = y to take else branch
- Don't know how to solve !!
  - **Stuck ?**