Symbolic Execution

Zhaoguo Wang

Adapted From:

<<A Survey of Symbolic Execution Techniques>>

Predicate Logic（谓词逻辑）

1.1 Propositional Logic

Step 0. Proof.

Step 1. Convert program into mathematical formula.

Step 2. Ask the computer to solve the formula.

1.2 First Order Logic

Step 1. Convert it into first order logic formula.

Step 2. Ask the computer to solve the formula.

1.3 Auto-active Proof

Step 1. Axiom system

Step 2. Ask the computer to check the invariants

Outline – This Lecture

- SMT
- Symbolic execution engine

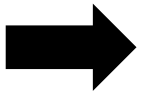
A Quick Recap

```
void swap(bool& a, bool& b)
{
        a = a ^ b;
        b = b ^ a;
        a = a ^ b;
}
```

We want to prove that the program really swaps the value of a and b.

A Quick Recap

```
void swap(bool& a, bool& b)
{
        a = a ^ b;
        b = b ^ a;
        a = a ^ b;
}
```

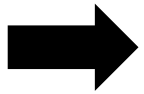


$$(A1 \leftrightarrow (A0 \wedge \neg B0) \vee (\neg A0 \wedge B0)) \wedge$$
$$(B1 \leftrightarrow B0) \wedge$$
$$(A2 \leftrightarrow A1) \wedge$$
$$(B2 \leftrightarrow (A1 \wedge \neg B1) \vee (\neg A1 \wedge B1)) \wedge$$
$$(A3 \leftrightarrow (A2 \wedge \neg B2) \vee (\neg A2 \wedge B2)) \wedge$$
$$(B3 \leftrightarrow B2) \rightarrow$$
$$(A3 \leftrightarrow B0) \wedge (A0 \leftrightarrow B3)$$

Prove it to be a tautology

A Quick Recap

```
void swap(bool& a, bool& b)
{
        a = a ^ b;
        b = b ^ a;
        a = a ^ b;
}
```

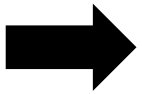


$$(A1 \leftrightarrow (A0 \wedge \neg B0) \vee (\neg A0 \wedge B0)) \wedge$$
$$(B1 \leftrightarrow B0) \wedge$$
$$(A2 \leftrightarrow A1) \wedge$$
$$(B2 \leftrightarrow (A1 \wedge \neg B1) \vee (\neg A1 \wedge B1)) \wedge$$
$$(A3 \leftrightarrow (A2 \wedge \neg B2) \vee (\neg A2 \wedge B2)) \wedge$$
$$(B3 \leftrightarrow B2) \wedge$$
$$\neg((A3 \leftrightarrow B0) \wedge (A0 \leftrightarrow B3))$$

Prove it to be unsatisfiable with SAT solver

Predicate Logic

```
void swap(int& a, int& b)
{
        a = a ^ b;
        b = b ^ a;
        a = a ^ b;
}
```

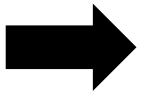


$$
\begin{aligned}
&(\forall A0)(\forall A1)(\forall A2)(\forall A3)(\forall B0)(\forall B1)(\forall B2)(\forall B3)(\\
&\quad (A1 = xor(A0, B0) \wedge \\
&\quad B1 = B0 \wedge \\
&\quad A2 = A1 \wedge \\
&\quad B2 = xor(A1, B1) \\
&\quad B3 = B2 \\
&\quad A3 = xor(A2, B2) \\
&\quad) \rightarrow \\
&\quad ((A3 = B0) \wedge (B3 = A0)) \\
&)
\end{aligned}
$$

We can generate predicate logic WFF similarly

Predicate Logic

```
void swap(int& a, int& b)
{
        a = a ^ b;
        b = b ^ a;
        a = a ^ b;
}
```

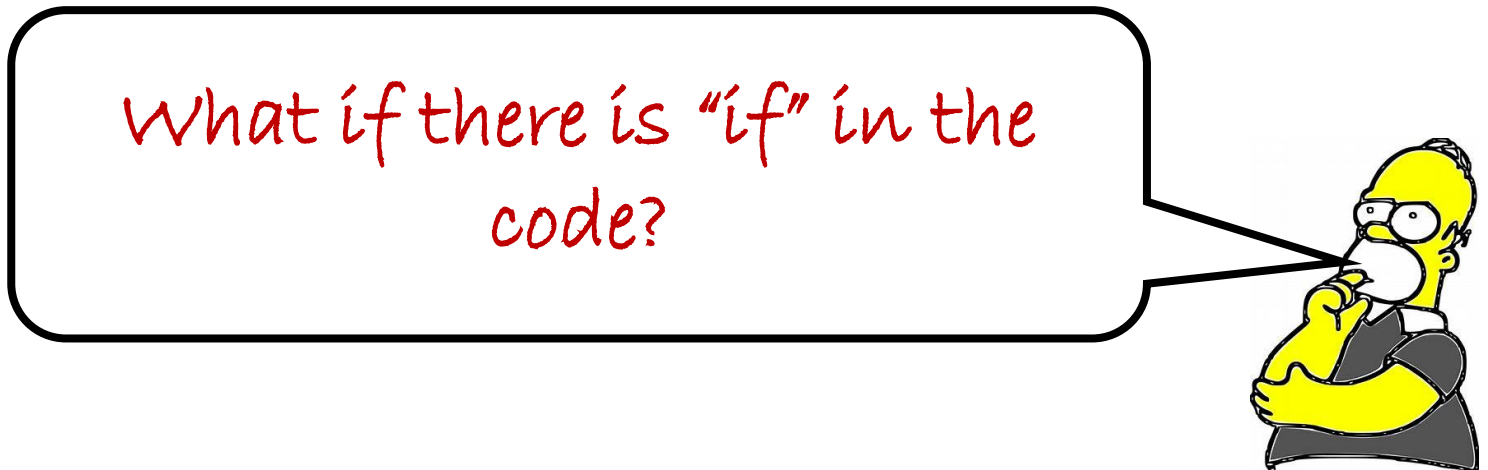


$$
\begin{aligned}
&(\exists A0)(\exists A1)(\exists A2)(\exists A3)(\exists B0)(\exists B1)(\exists B2)(\exists B3)(\\
&\quad (A1 = xor(A0, B0) \wedge \\
&\quad B1 = B0 \wedge \\
&\quad A2 = A1 \wedge \\
&\quad B2 = xor(A1, B1) \\
&\quad B3 = B2 \\
&\quad A3 = xor(A2, B2) \\
&\quad) \wedge \\
&\quad \neg((A3 = B0) \wedge (B3 = A0)) \\
&)
\end{aligned}
$$

Prove it to be unsat with SMT solver

Symbolic Execution (符号执行)

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



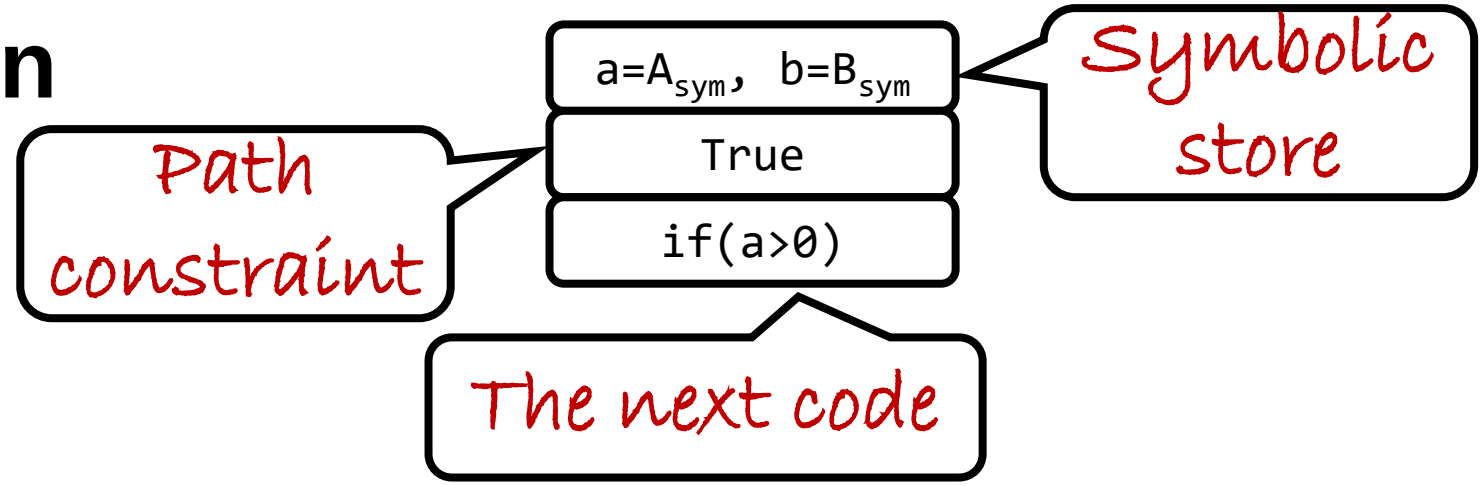
Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```

We can generate multiple WFFs for different execution paths.

Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



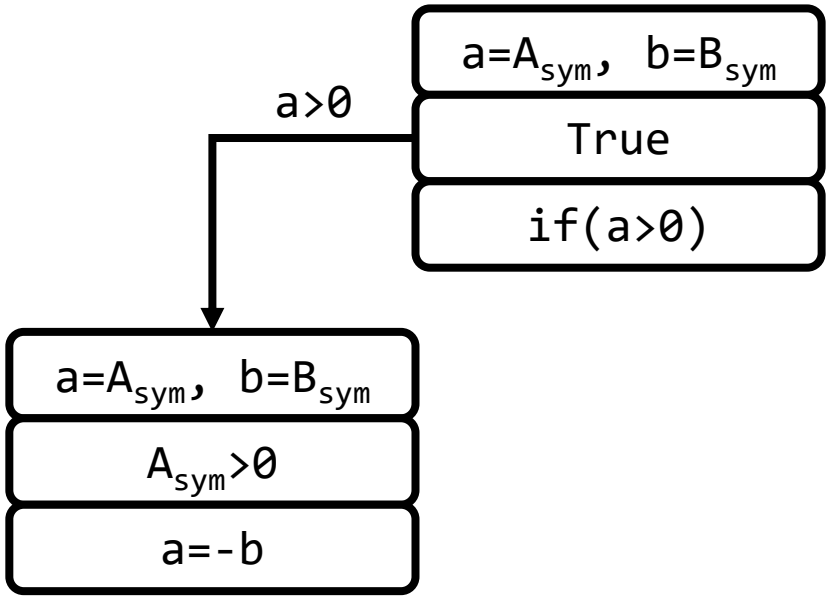
Symbolic execution uses the symbols to represent the value of variables.

It maintains 3 things to generate WFF.

- Symbolic store: the value (symbolic expression) for each variable
- Path constraints: the path condition
- Next code: the next instruction to be executed

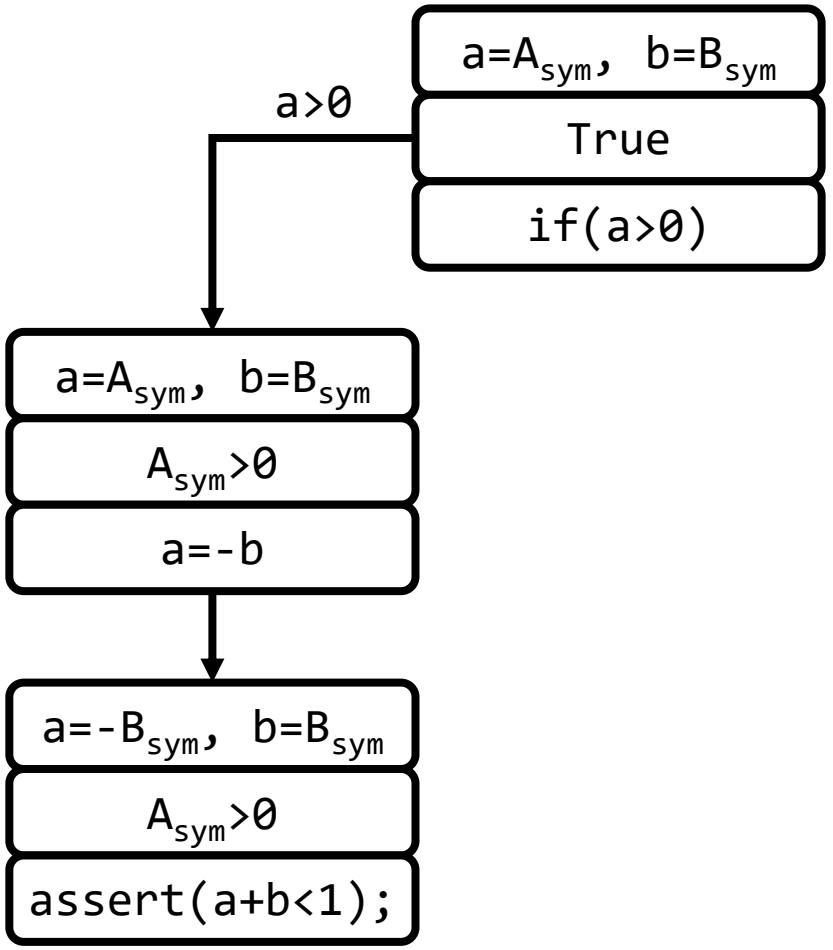
Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



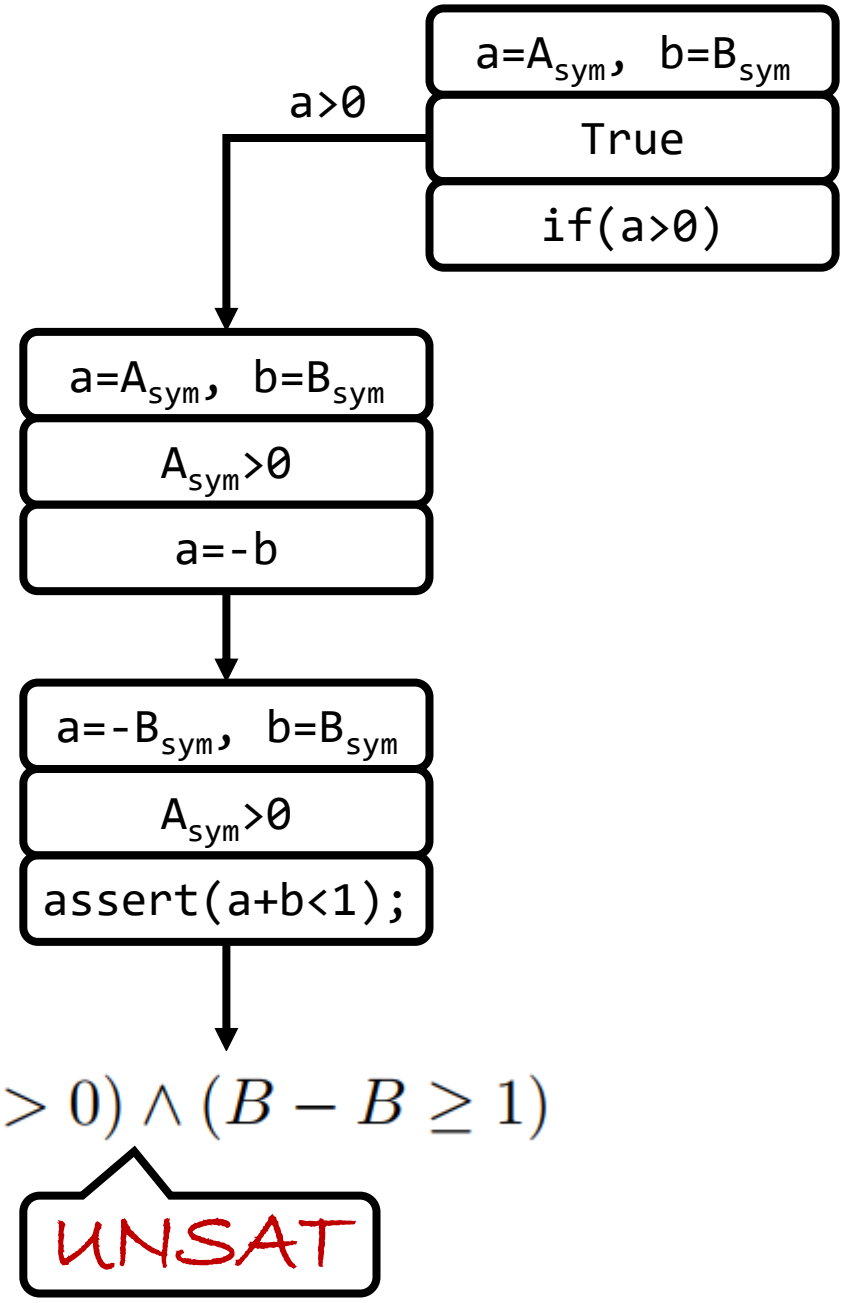
Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



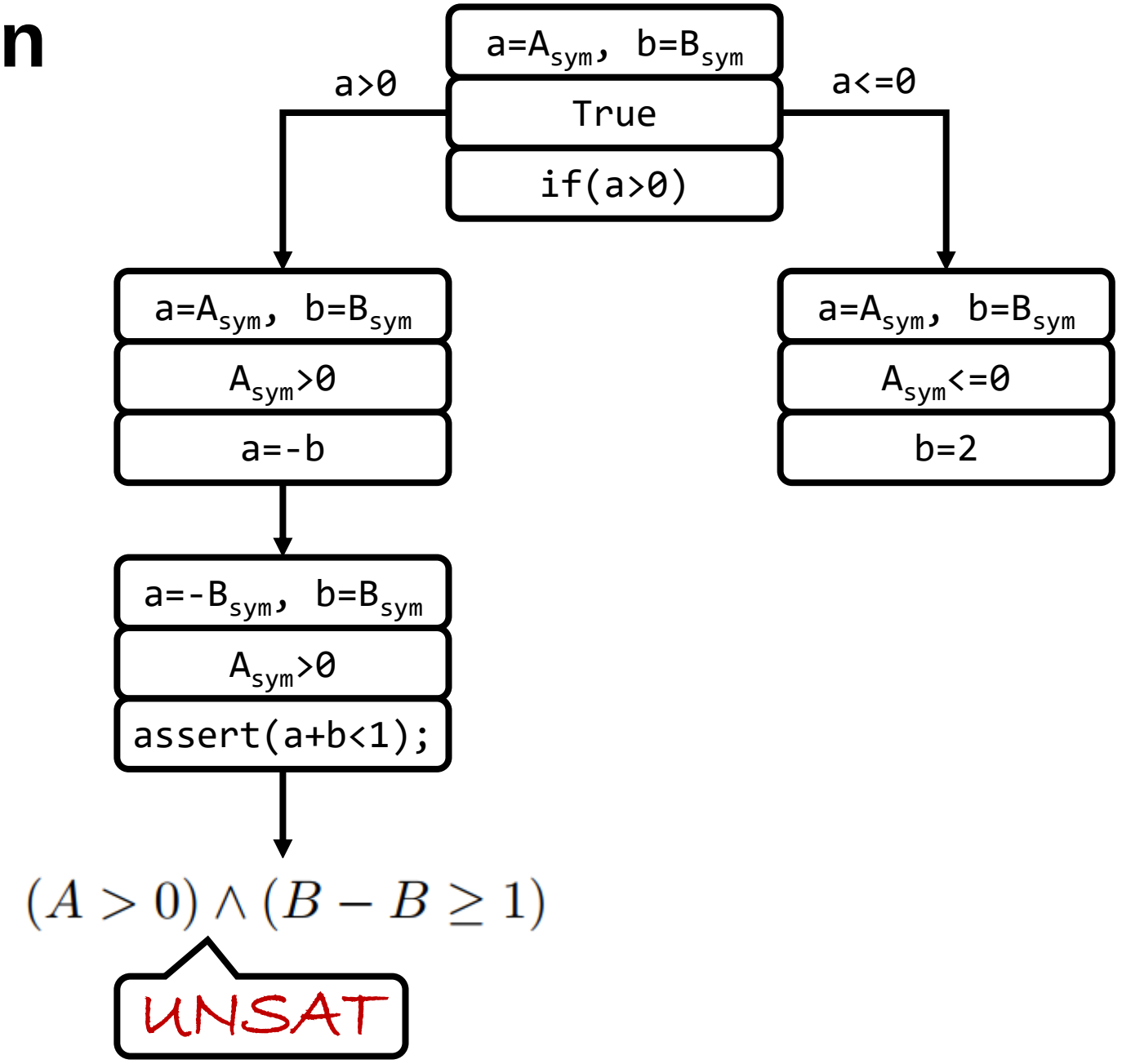
Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



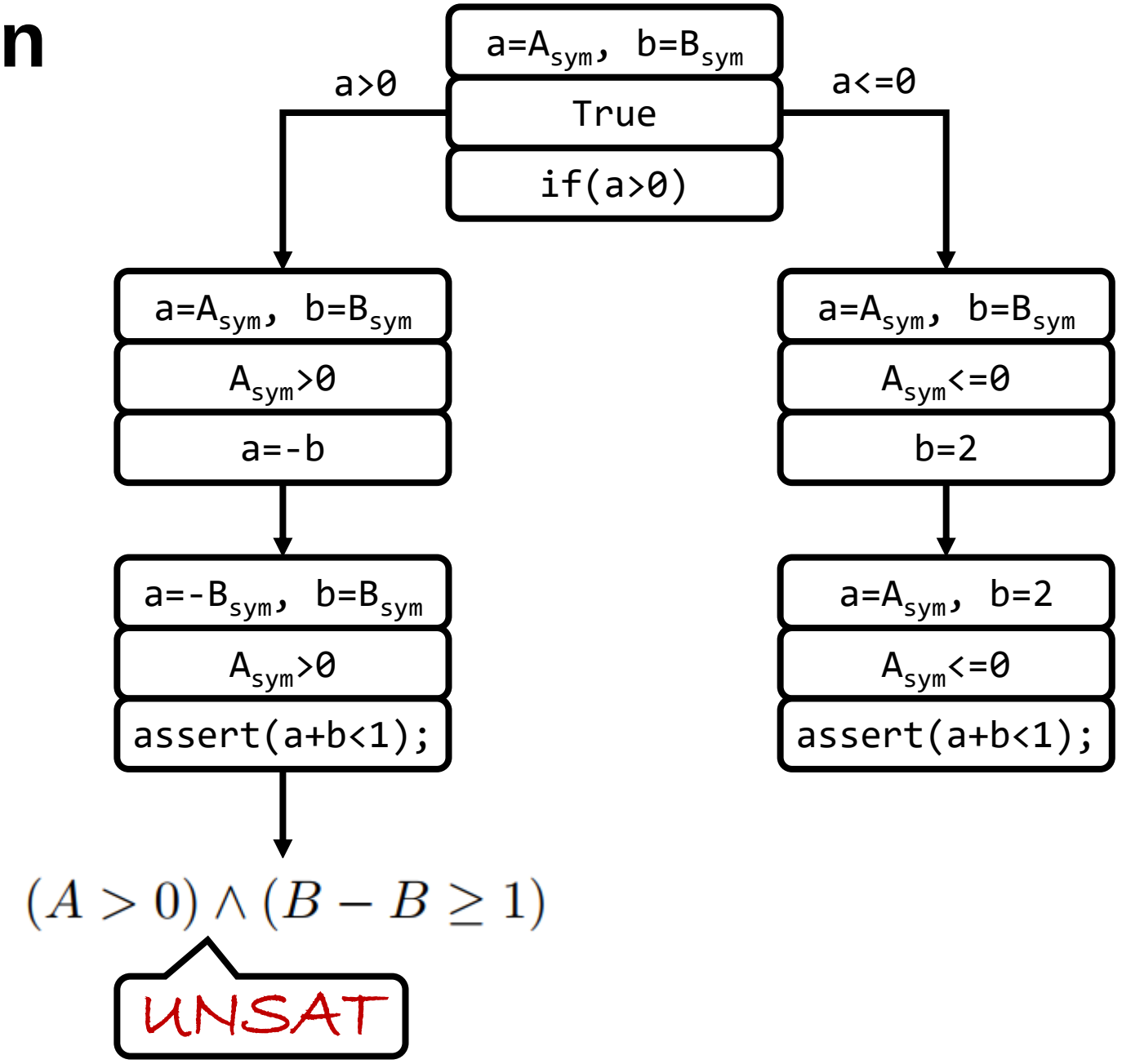
Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



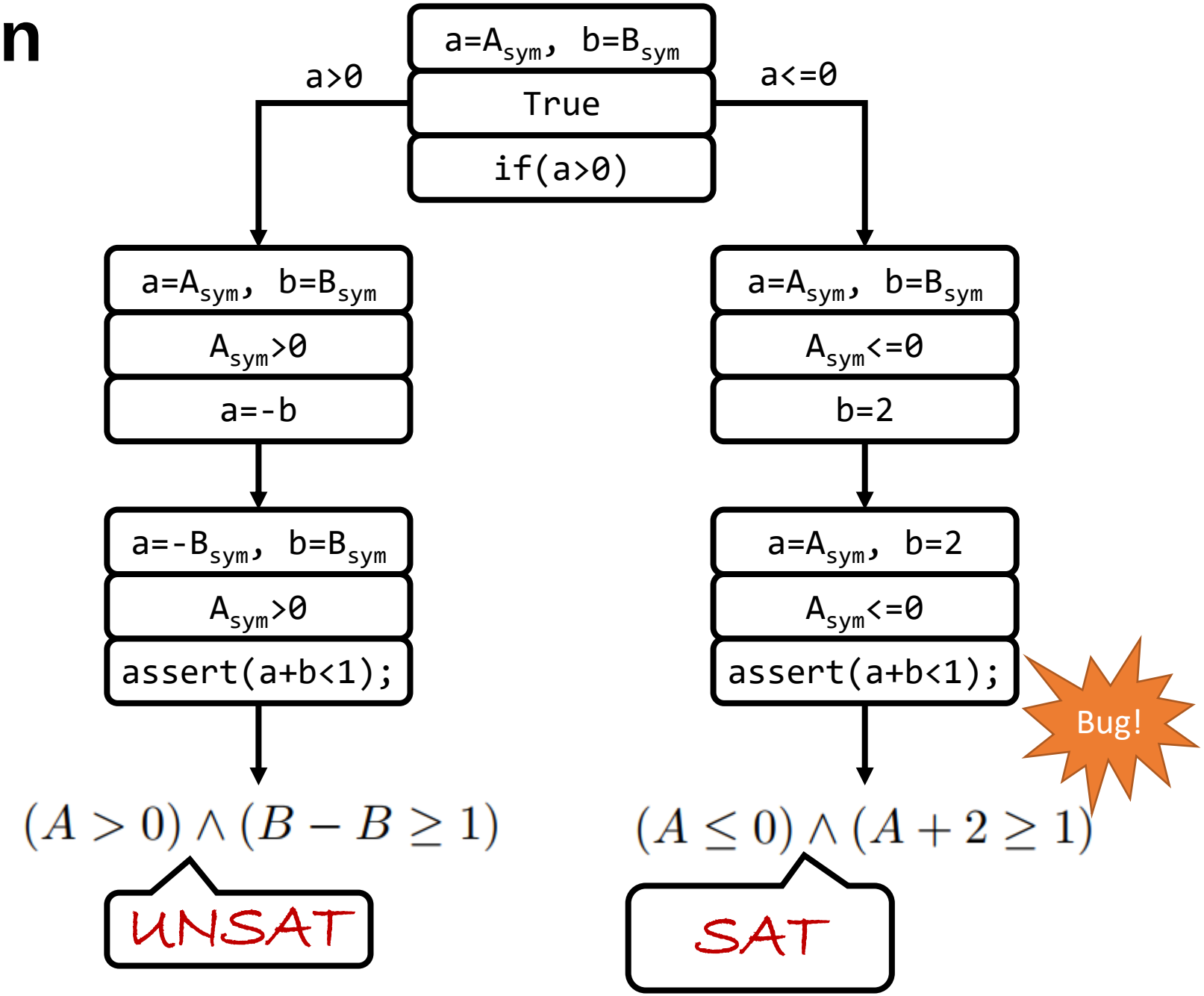
Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



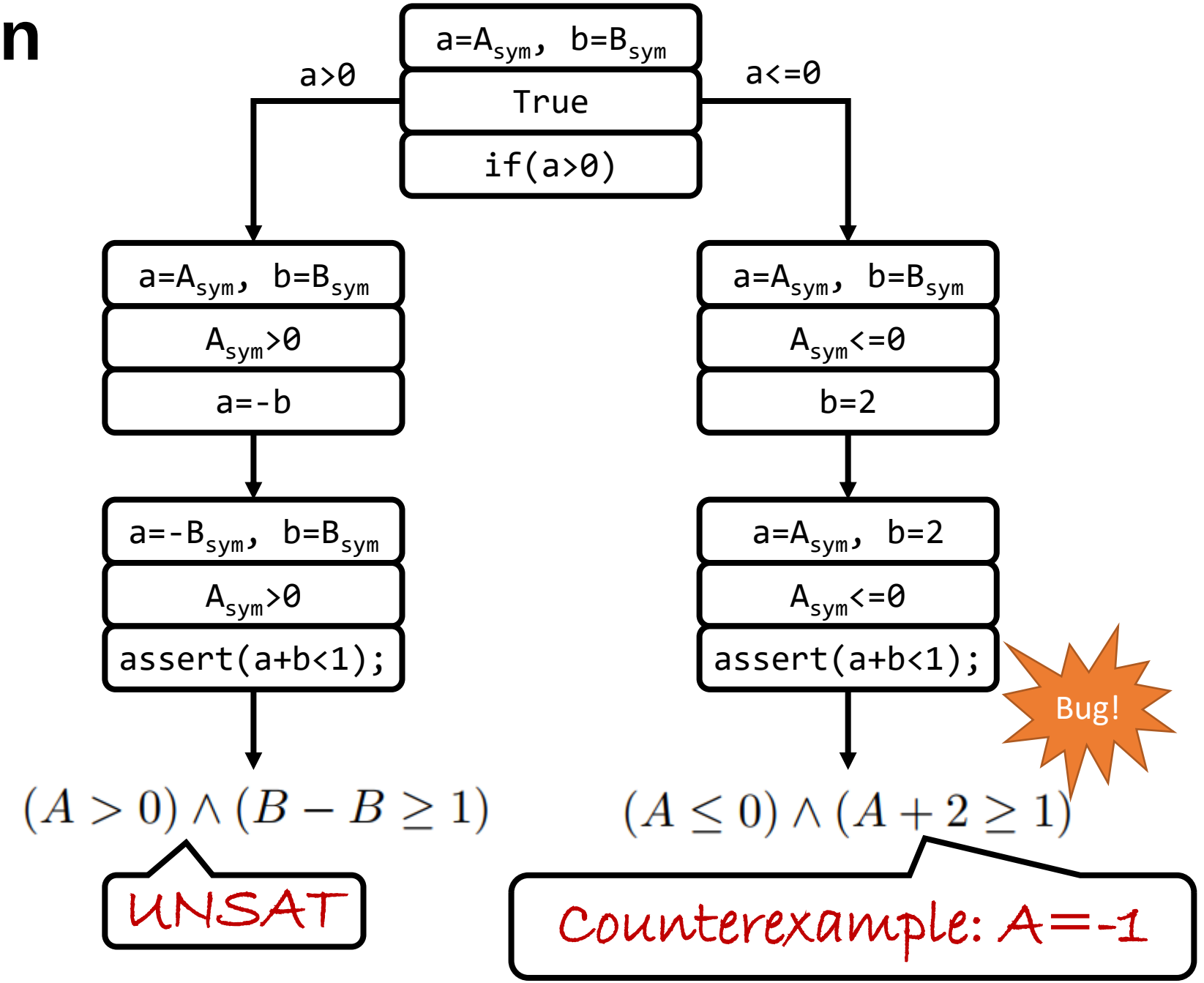
Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



Symbolic Execution

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```

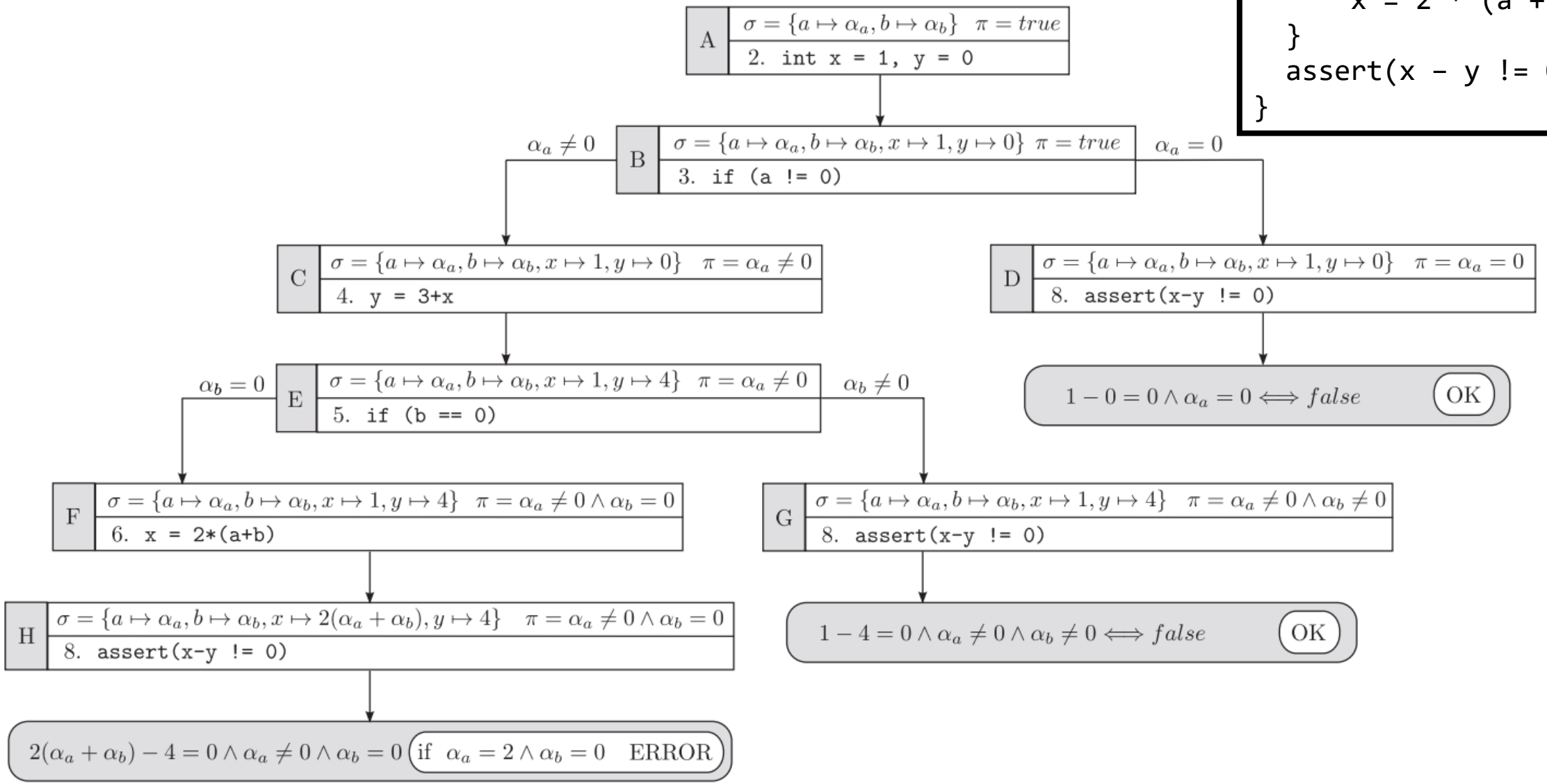


Example

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

Example

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

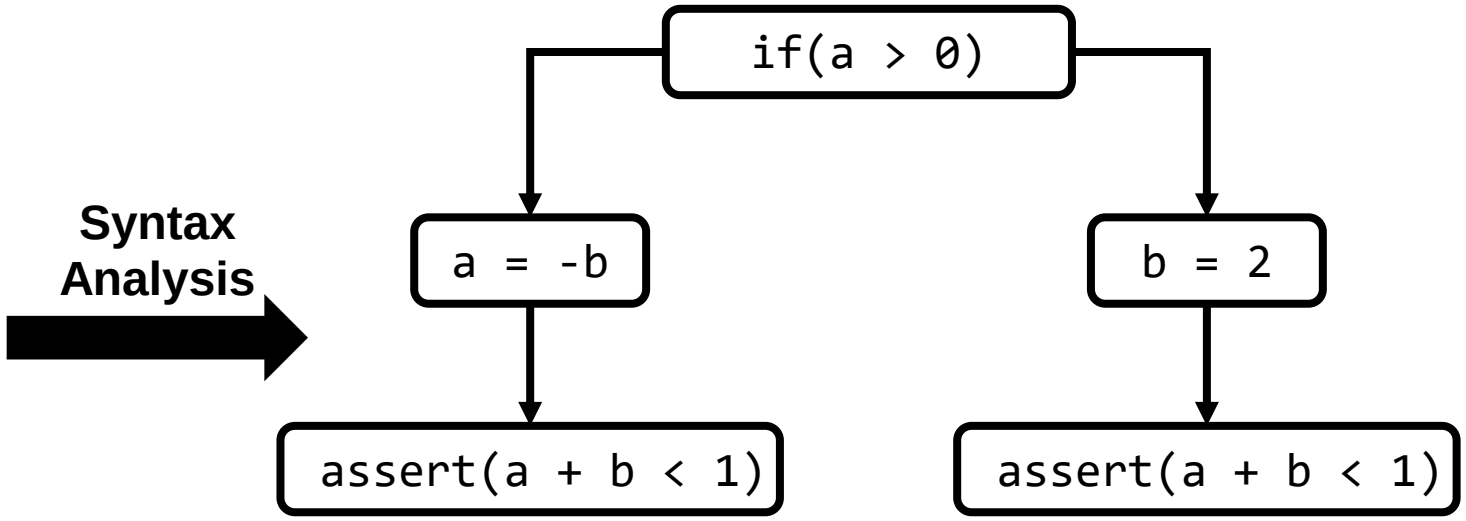


Lab2

MiniSEE: A symbolic execution engine for simple C program

func1.c

```
int func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



MiniSEE first parses the function into a syntax tree via syntax analysis

- Each node denotes a statement
- Each arrow denotes the execution order between two statements

Then MiniSEE performs symbolic execution based on the tree

Lab2

MiniSEE: A symbolic execution engine for simple C program

func1.c

```
int func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```


```
> ./minisee func1.c

-1 0
```

Lab2

MiniSEE: A symbolic execution engine for simple C program

func2.c

```
int func(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);
}
```


```
> ./minisee func2.c

2 0
```

Lab2

You can even use MiniSEE to verify your ICS lab1 solutions

ics.c

```
// compute !x without using !
int bang(int x){
  int r1, neg_x, r2, res;
  r1 = x >> 31;
  neg_x = ~x + 1;
  r2 = neg_x >> 31;
  res = (r1 | r2);
  assert((x == 0 && res == 1)
      || (x != 0 && res == 0));
  return res;
}
```


```
> ./minisee ics.c

0
```

Lab2

You can even use MiniSEE to verify your ICS lab1 solutions

ics.c

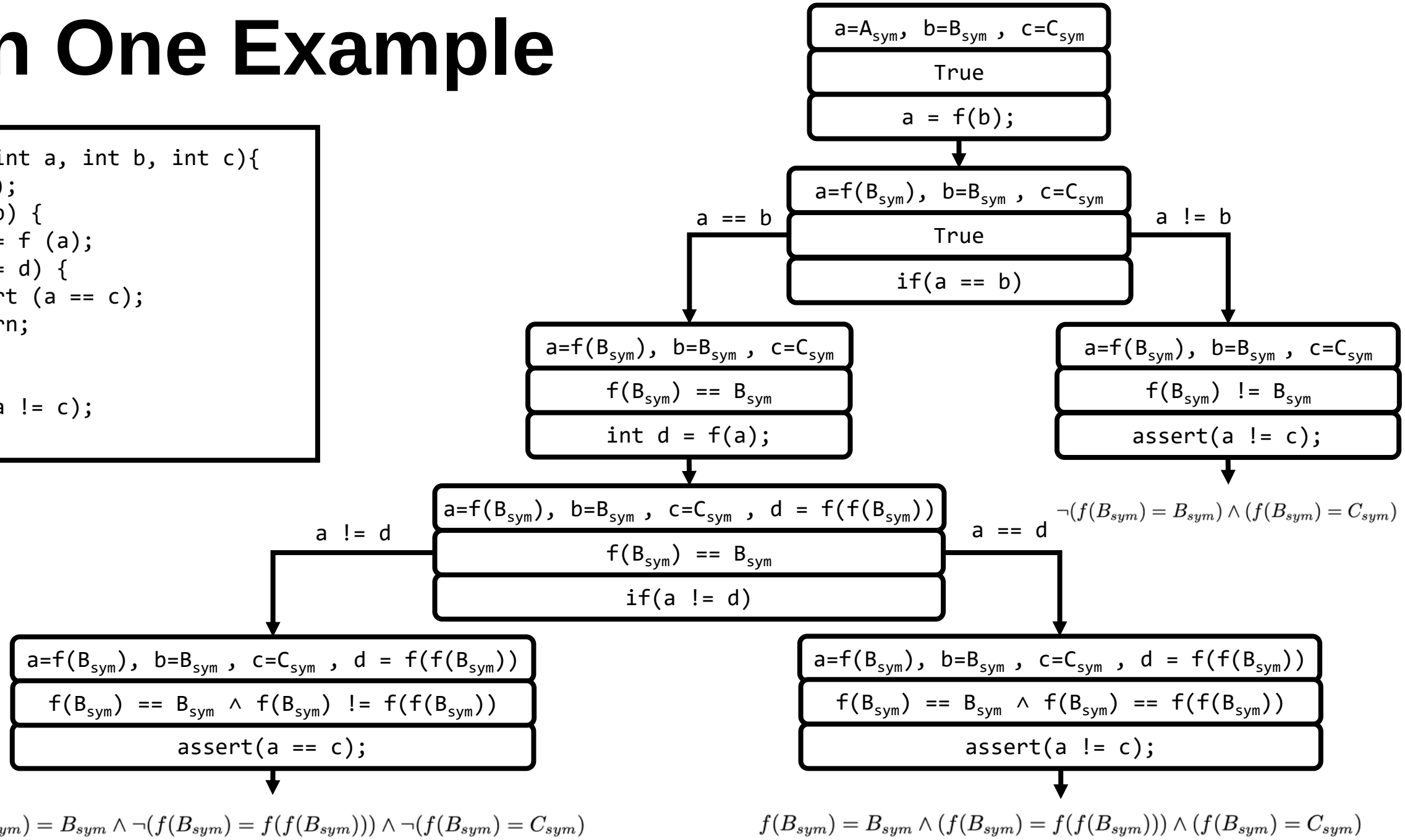
```
// compute !x without using !
int bang(int x){
  int r1, neg_x, r2, res;
  r1 = x >> 31;
  neg_x = ~x + 1;
  r2 = neg_x >> 31;
  res = (r1 | r2) + 1;
  assert((x == 0 && res == 1)
      || (x != 0 && res == 0));
  return res;
}
```


```
> ./minisee ics.c

verified
```

All in One Example

```
void func(int a, int b, int c){
  a = f (b);
  if(a == b) {
    int d = f (a);
    if(a != d) {
      assert (a == c);
      return;
    }
  }
  assert (a != c);
}
```



Lazy SMT

```
void func(int a, int b, int c){
  a = f (b);
  if(a == b) {
    int d = f (a);
    if(a != d) {
      assert (a == c);
      return;
    }
  }
  assert (a != c);
}
```

$$f(B_{sym}) = B_{sym} \wedge$$
$$\neg(f(B_{sym}) = f(f(B_{sym}))) \wedge$$
$$\neg(f(B_{sym}) = C_{sym})$$

$$p1 \wedge \neg p2 \wedge \neg p3$$

DPLL

DPLL

$$p_1 \wedge \neg p_2 \wedge \neg p_3$$

$$\emptyset \parallel p_1,\ \overline{p_2},\ \overline{p_3}, \Rightarrow (PureLiteral)$$

$$p_1 \parallel p_1,\ \overline{p_2},\ \overline{p_3}, \Rightarrow (PureLiteral)$$

$$p_1\ \overline{p_2} \parallel p_1,\ \overline{p_2},\ \overline{p_3}, \Rightarrow (PureLiteral)$$

$$p_1\ \overline{p_2}\ \overline{p_3} \parallel p_1,\ \overline{p_2},\ \overline{p_3}, \Rightarrow (PureLiteral)$$

Lazy SMT

```
void func(int a, int b, int c){
  a = f (b);
  if(a == b) {
    int d = f (a);
    if(a != d) {
      assert (a == c);
      return;
    }
  }
  assert (a != c);
}
```

$$f(B_{sym}) = B_{sym} \wedge$$
$$\neg(f(B_{sym}) = f(f(B_{sym}))) \wedge$$
$$\neg(f(B_{sym}) = C_{sym})$$

$$p1 \wedge \neg p2 \wedge \neg p3$$

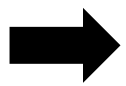
DPLL

$$p1 = T, p2 = F, p3 = F$$

$$f(B_{sym}) = B_{sym},$$
$$f(B_{sym}) \neq f(f(B_{sym})),$$
$$f(B_{sym}) \neq C_{sym}$$

EUF

$$f(B_{sym}) = B_{sym},$$
$$f(B_{sym}) \neq f(f(B_{sym})),$$
$$f(B_{sym}) \neq C_{sym}$$



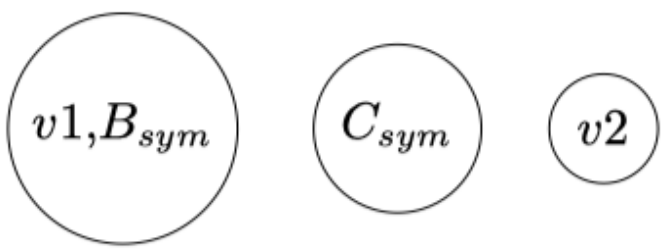
$$v1 = B_{sym}, v1 \neq v2, v1 \neq C_{sym},$$
$$v1 := f(B_{sym}), v2 := f(v1)$$

EUF

$$v1 = B_{sym}, v1 \neq v2, v1 \neq C_{sym},$$
$$v1 := f(B_{sym}), v2 := f(v1)$$

EUF

$$v1 = B_{sym}, v1 \neq v2, v1 \neq C_{sym},$$
$$v1 := f(B_{sym}), v2 := f(v1)$$

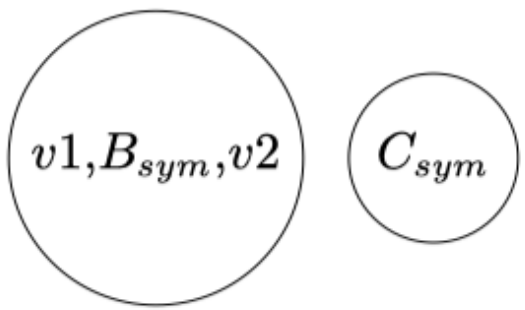


EUF

$v1 = B_{sym} \Rightarrow f(v1) = f(B_{sym})$

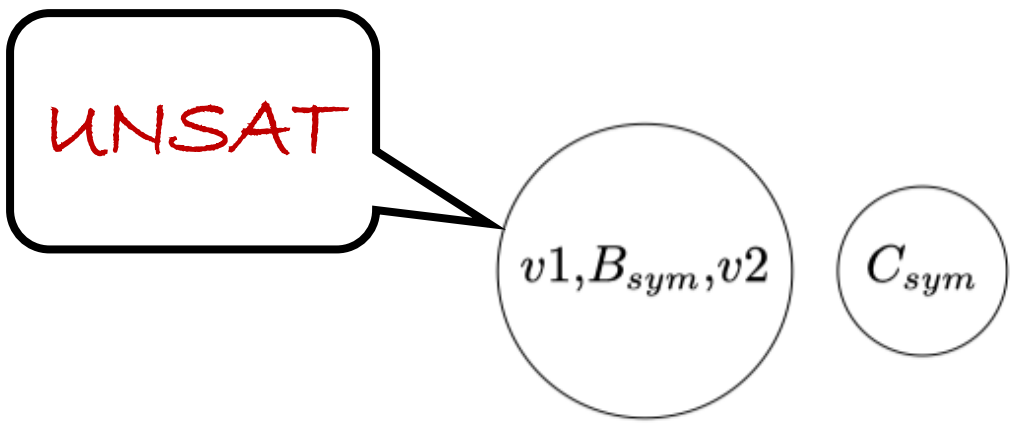
$v1 = B_{sym}, v1 \neq v2, v1 \neq C_{sym},$

$v1 := f(B_{sym}), v2 := f(v1)$



EUF

$$v1 = B_{sym}, v1 \neq v2, v1 \neq C_{sym},$$
$$v1 := f(B_{sym}), v2 := f(v1)$$



Lazy SMT

```
void func(int a, int b, int c){
  a = f (b);
  if(a == b) {
    int d = f (a);
    if(a != d) {
      assert (a == c);
      return;
    }
  }
  assert (a != c);
}
```

$$f(B_{sym}) = B_{sym} \wedge$$
$$\neg(f(B_{sym}) = f(f(B_{sym}))) \wedge$$
$$\neg(f(B_{sym}) = C_{sym})$$

$$p1 \wedge \neg p2 \wedge \neg p3$$

DPLL

$$p1 = T, p2 = F, p3 = F$$

$$f(B_{sym}) = B_{sym},$$
$$f(B_{sym}) \neq f(f(B_{sym})),$$
$$f(B_{sym}) \neq C_{sym}$$

Lazy SMT

```
void func(int a, int b, int c){
  a = f (b);
  if(a == b) {
    int d = f (a);
    if(a != d) {
      assert (a == c);
      return;
    }
  }
  assert (a != c);
}
```

$$f(B_{sym}) = B_{sym} \wedge$$
$$\neg(f(B_{sym}) = f(f(B_{sym}))) \wedge$$
$$\neg(f(B_{sym}) = C_{sym})$$

$$p1 \wedge \neg p2 \wedge \neg p3$$

DPLL

$$\neg(p1 \wedge \neg p2 \wedge \neg p3)$$

$$f(B_{sym}) = B_{sym},$$
$$f(B_{sym}) \neq f(f(B_{sym})),$$
$$f(B_{sym}) \neq C_{sym}$$

DPLL

$$p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge (\neg p_1 \vee p_2 \wedge p_3)$$

$$\emptyset \parallel p_1,\ \overline{p_2},\ \overline{p_3},\ \overline{p_1} \vee p_2 \vee p_3 \Rightarrow (Decide)$$

$$p_1^d \parallel p_1,\ \overline{p_2},\ \overline{p_3},\ \overline{p_1} \vee p_2 \vee p_3 \Rightarrow (Decide)$$

$$p_1^d\ \overline{p_2^d} \parallel p_1,\ \overline{p_2},\ \overline{p_3},\ \overline{p_1} \vee p_2 \vee p_3 \Rightarrow (UnitProp)$$

$$p_1^d\ \overline{p_2^d}\ p_3 \parallel p_1,\ \overline{p_2},\ \overline{p_3},\ \overline{p_1} \vee p_2 \vee p_3 \Rightarrow (Backtrack)$$

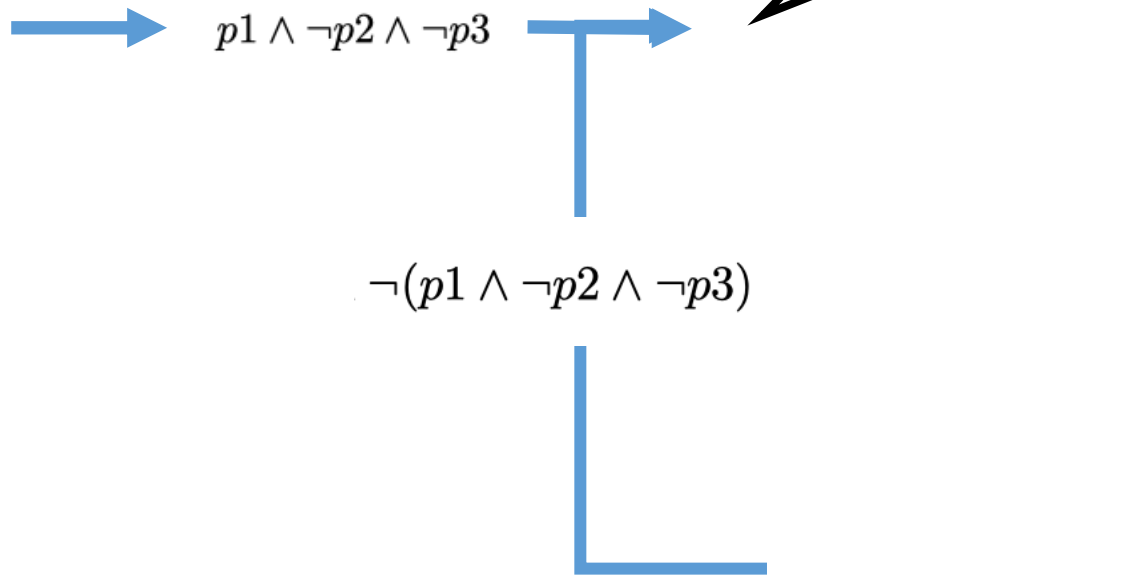
$$p_1^d\ p_2 \parallel p_1,\ \overline{p_2},\ \overline{p_3},\ \overline{p_1} \vee p_2 \vee p_3 \Rightarrow (Backtrack)$$

$$\overline{p_1} \parallel p_1,\ \overline{p_2},\ \overline{p_3},\ \overline{p_1} \vee p_2 \vee p_3 \Rightarrow (Fail)$$

Lazy SMT

```
void func(int a, int b, int c){
  a = f (b);
  if(a == b) {
    int d = f (a);
    if(a != d) {
      assert (a == c);
      return;
    }
  }
  assert (a != c);
}
```

$$f(B_{sym}) = B_{sym} \wedge$$
$$\neg(f(B_{sym}) = f(f(B_{sym}))) \wedge$$
$$\neg(f(B_{sym}) = C_{sym})$$



Thanks & Questions