proposed solution created on 2025-11-13 17:29

a ☒ An imprecise and sound analyzer can be used for bug finding in a program. +1
b ☒ A precise and sound analyzer can be used for bug finding in a program. +1
c ☐ An imprecise and unsound analyzer can be used to verify a program.
d ☐ A precise and unsound analyzer can be used to verify a program.
e ☒ An imprecise and sound analyzer can be used to verify a program. +1

[a] True

It might not find all bugs due to the imprecision, but it still can find some/most bugs depending on the level of imprecision.

[b] True

precise & sound analyser can correctly verify all asserts -> it can definitely find bugs

[c] False

unsound analyser cannot verify a program

[d] False

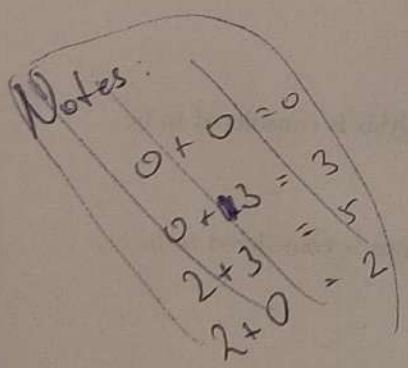
unsound analyser cannot verify a program

[e] True

The analyser must be precise enough though

2.) Program:

```
void baz(unsigned int n) {
  unsigned int m = 2;
  if(n >= 1 && n < 3) {
    m = 0;
  }
  assert m == 2 || m == 0;
}
```



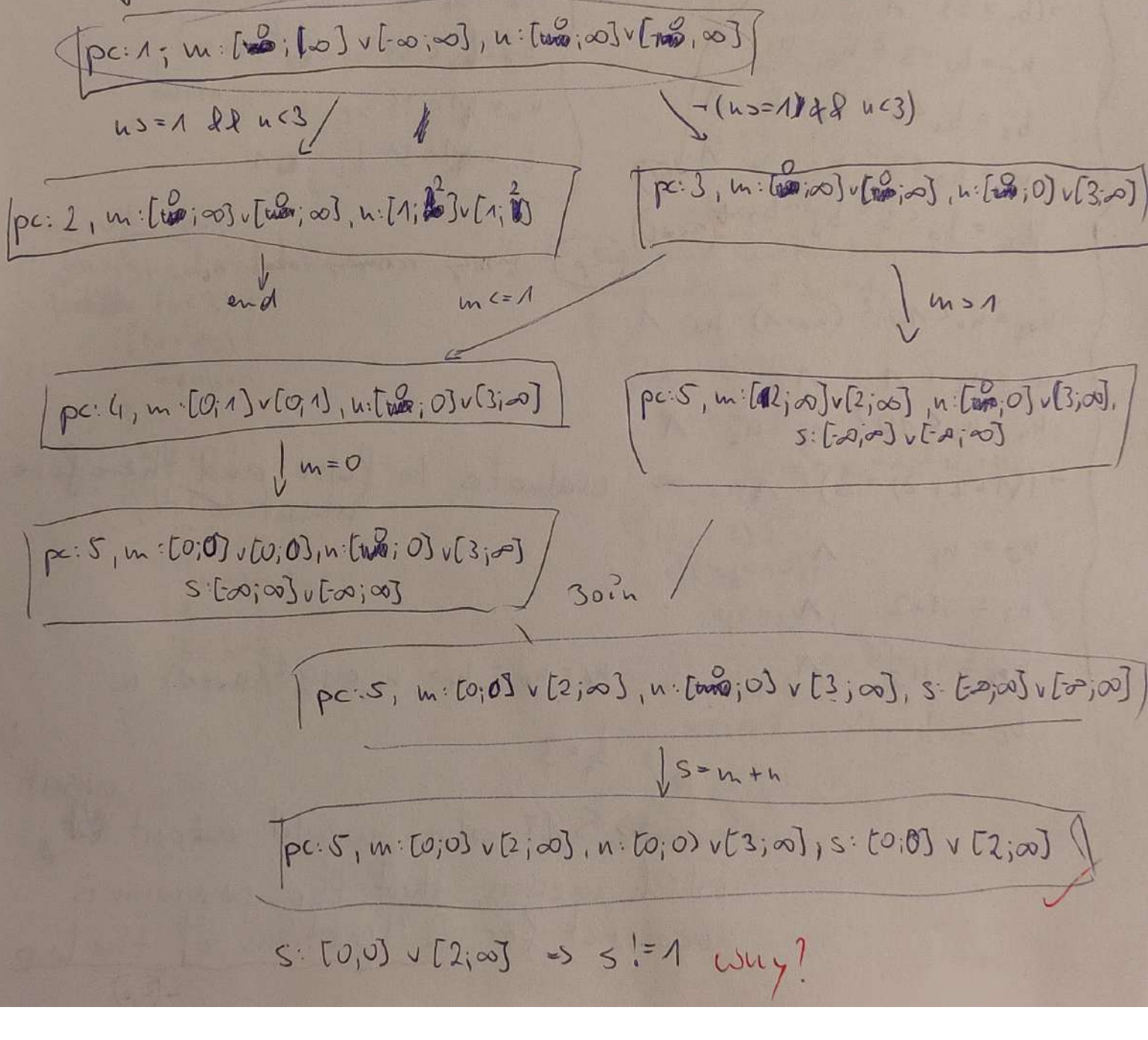
**2.) Abstract Interpretation** (33 points) 28

Let us consider the following extended interval domain:
The set of abstract values is given by $\mathcal{D} = \{[a,b] \vee [c,d] \mid a,c \in \mathbb{N} \cup \{-\infty\}$ and $b,d \in \mathbb{N} \cup \{\infty\}\} \cup \{\bot\}$.
Let *var* be a declared variable, *val*(*var*) the concrete value of *var* and $[u,v] \vee [w,x]$ its abstract state; the abstract transformers maintain the invariant $val(var) \in [u,v] \cup [w,x]$.

- Perform Abstract Interpretation on function bar below, giving the (abstract) control flow graph. For ease of brevity, you can omit variable *s* from all program states until line 5. (18 points) 18
- Finally establish that the asserted condition holds. (5 points) 0

```
0 void bar(unsigned int m, unsigned int n){
1   if(n >= 1 && n < 3)
2     return;
3   if(m <= 1)
4     m = 0;
5   unsigned int s = m + n;
6   assert s != 1;
7 }
```

- Show that the standard interval domain is less precise than $\mathcal{D}$ by providing a program where the standard interval domain does not prove an assertion whereas $\mathcal{D}$ does. (10 points) 10



2

$D = \{$

Intervall 1 possible values for $[a,b]$:

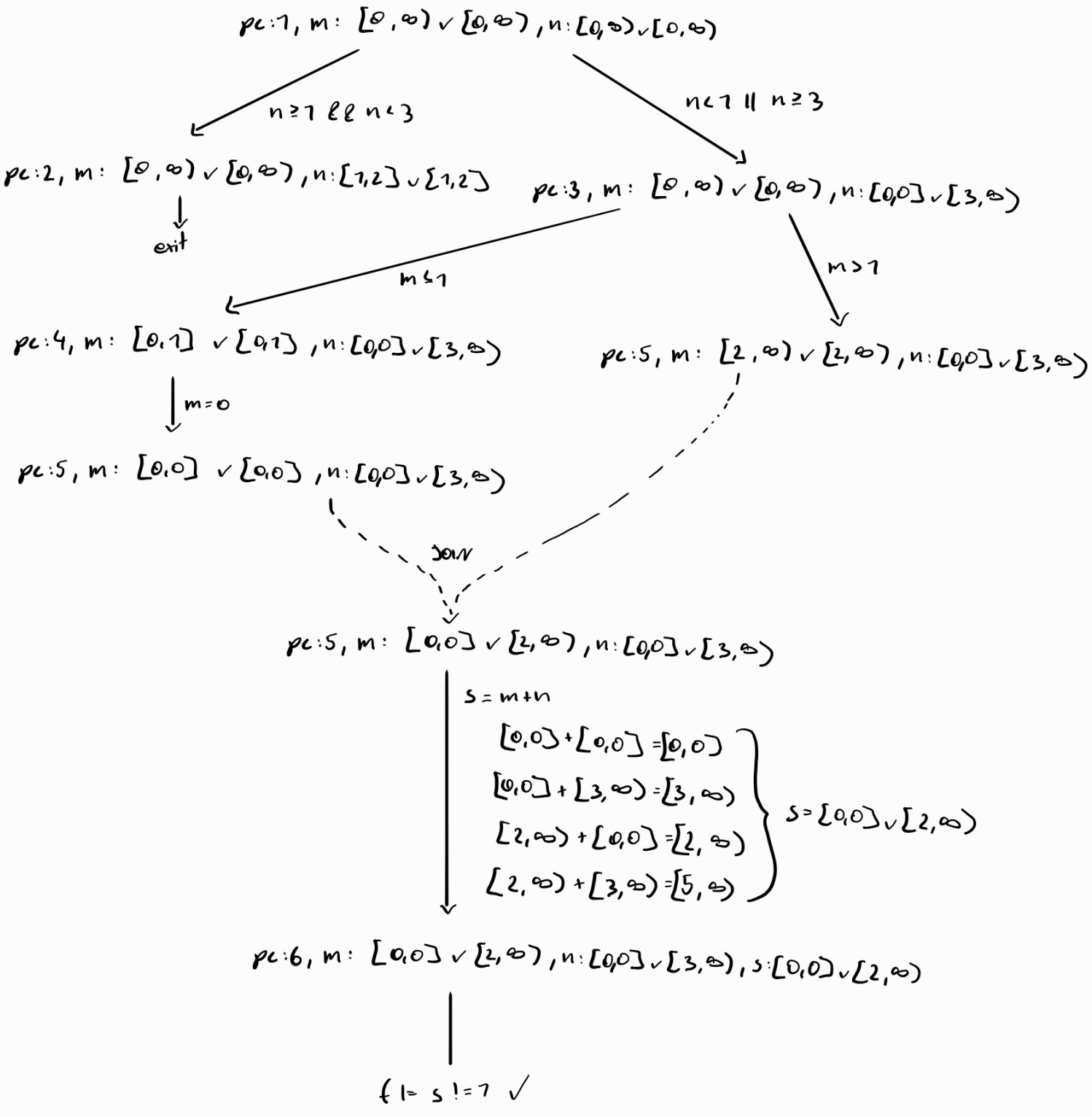
Integer values from $-\infty$ to $\infty$

Intervall 2 possible values for $[c,d]$:

Integer values from $-\infty$ to $\infty$

$\}$

a abstract interpretation



[b] establish that the assertion condition holds

$f \models s \neq 1$ ✓ must hold because s can either be in the interval $[0,0]$ or $[2,\infty)$
i.e. $s \in \mathbb{N}$, except for $s=1$

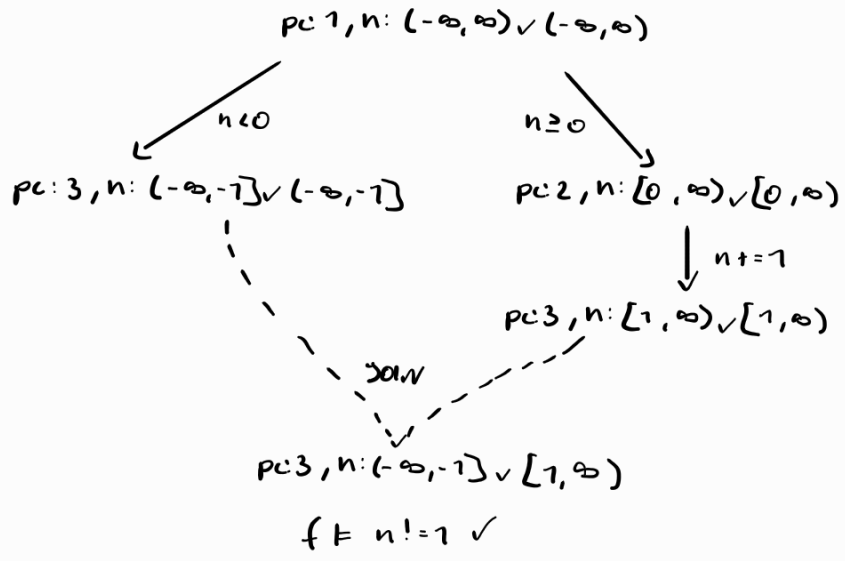
[c] show that the standard interval domain is less precise than D by providing a program where the standard interval domain does not prove an assertion whereas D does.

- for this task we can just use the program that has been provided. If we try to solve it using the standard interval domain we end up with

- Alternatively you can create a simple program like this, where the asserted value can be inside of two non-overlapping intervals:

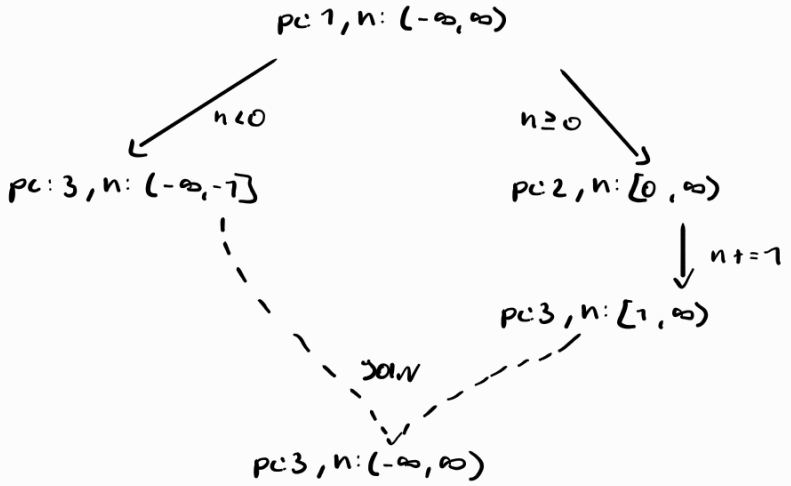
```
0: void foo(int n) {
1:     if (n≥0) {
2:         n+=1
       }
3:     assert n!=0
```

[c1] With domain D




$f \models n \neq 1$ ✓

[c2] With standard interval domain




$f \not\models n \neq 1$

standard interval domain fails

21

## 3.) Bounded Model Checking (33 points)

Consider the C function below and verify that it is free from division-by-zero errors by performing Bounded Model Checking with a loop-unrolling bound of 2.

- Show the program after the step of converting it into SSA form. (12 points) 12
- Show the constraints obtained before the bit-blasting step. (8 points) 6
- Would a SAT solver finally output *SAT* or *UNSAT*? In case it would output *SAT*, give a satisfying assignment of the integer variables in the constructed constraints. In case it would output *UNSAT*, show that such an assignment cannot exist. (5 points) 2
- What can we learn about function `f` from the SAT solver's output? (3 points) 1
- Assuming that the bit-width of **short** variables on the system executing the code is 16, how many boolean variables would a SAT solver require in order to obtain a valid encoding of your constraints *after* bit-blasting? Justify your answer briefly. (5 points) 0

↳ $2^{16}$ per variable in the program

```
unsigned short f(unsigned short n){
  unsigned short b = 1;
  while(b < 3){
    if(n < 10){
      n++;
    }
    b = b + 2;
  }
  return 5/n;
}
```

simplify:

```
unsigned short b = 1;
while(b < 3){
  if(n < 10){
    n = n+1;
  }
  b = b + 2;
}
return 5/n;
```

unroll loop:

```
unsigned short b = 1;
if(b<3){
  if(n<10){
    n = n+1;
  }
  b = b+2;
  if(b<3){
    if(n<10){
      n = n+1;
    }
    b = b+2;
    assert b>3;
  }
}
return 5/n;
```

SSA: `(unsigned short n0){`

```
unsigned short b0 = 1;
if(b0<3){
  if(n0<10){
    n1 = n0+1;
  }                    < n2 = n0<10 ? n1 : n0
  b1 = b0+2;
  if(b1<3){
    if(n2<10){
      n3 = n2+1;
    }                  < n4 = n2<10 ? n3 : n2
    b2 = b1+2;
    assert b2>3;
  }
  n5 = b1<3 ? n4 : n2
  b3 = b1<3 ? b2 : b1
}
n6 = b0<3 ? n5 : n0
b4 = b0<3 ? b3 : b0
return 5/n6;
```

1 Add assertion to check that division by zero does not occur
AND simplify control flow

```
unsigned short f(unsigned short n) {
    unsigned short b=1;
    while(b<3) {
        if(n<10) {
            n=n+1;
        }
        b=b+2;
    }
    assert n!=0;
    return 5/n;
}
```

2 Unroll loops with bound=2

```
unsigned short f(unsigned short n) {
    unsigned short b=1;
    if (b<3) {
        if(n<10) {
            n=n+1;
        }
        b=b+2;
        if (b<3) {
            if(n<10) {
                n=n+1;
            }
            b=b+2;
            assert b>=3;
        }

    }
    assert n!=0;
    return 5/n;
}
```

3 Single Static Assignment form (SSA)

```
unsigned short f(unsigned short n) {
    unsigned short n0,n1,n2,n3,n4,n5,b0,b1,b2;
    b0=1;
    if (b0<3) {
        if(n<10) {
            n0=n+1;
        }
        n1=n<10?n0:n;
        b1=b0+2;
        if (b1<3) {
            if(n1<10) {
                n2=n1+1;
            }
            n3=n1<10?n2:n1;
            b2=b1+2;
            assert b2>=3;
        }
        n4=b1<3?n3:n1;
    }
    n5=b0<3?n4:n;

    assert n5!=0;
    return 5/n5;
}
```

## 4 Convert into constraints

$b_0 = 1 \wedge$

$n_0 = n + 1; \wedge$

$n_1 = n < 10 \; ? \; n_0 : n \wedge$

$b_1 = b_0 + 2 \wedge$

$n_2 = n_1 + 1 \wedge$

$n_3 = n_1 < 10 \; ? \; n_2 : n_1 \wedge$

$b_2 = b_1 + 2 \wedge$

$n_4 = b_1 < 3 \; ? \; n_3 : n_1 \wedge$

$n_5 = b_0 < 3 \; ? \; n_4 : n \wedge$

$\neg(n_5 != 0 \wedge b_2 \geq 3)$

## 5 Simplify constraints

$n_0 = n + 1 \wedge$

$n_1 = n < 10 \; ? \; n_0 : n \wedge$

$n_2 = n_1 + 1 \wedge$

$n_3 = n_1 < 10 \; ? \; n_2 : n_1 \wedge$

$n_4 = 3 < 3 \; ? \; n_3 : n_1 \wedge$

$n_5 = 1 < 3 \; ? \; n_4 : n \wedge$

$(n_5 == 0 \vee \underbrace{5 < 3}_{\text{false}})$

$\longrightarrow$

$n_0 = n + 1 \wedge$

$n_1 = n < 10 \; ? \; n_0 : n$

$n_2 = n_1 + 1 \wedge$

$n_3 = n_1 < 10 \; ? \; n_2 : n_1$

$n_4 = 3 < 3 \; ? \; n_3 : n_1$ } Always false $n_4 = n_1$

$n_5 = 1 < 3 \; ? \; n_4 : n \wedge$ } Always true $n_5 = n_4$

$n_1 = n_4 = n_5$

$n_5 == 0$

$\swarrow$

$n_0 = n + 1 \wedge$

$n_1 = n < 10 \; ? \; n_0 : n \wedge$

$n_2 = n_1 + 1 \wedge$

$n_3 = n_1 < 10 \; ? \; n_2 : n_1 \wedge$

$n_1 == 0$

$\longrightarrow (n < 10 \; ? \; n+1 : n) == 0$

## 6 Check if SAT or UNSAT

<> Returns UNSAT

- n is an unsigned short: $n \in \mathbb{N} \wedge n \in [0, 2^{16} - 1]$
- n can only increase or stay the same
- input n=0, triggers if(n<10) causing an increase

So if we look at the final constraints:

$(n < 10 \; ? \; n+1 : n) == 0$

We have 2 cases:

Case 1: $n < 10$

$n = n + 1 \Rightarrow n \in \mathbb{N} \wedge n \in [1, 2^{16} - 1]$, n cannot reach 0, n==0 is true

Case 2: $n \geq 10$

n stays the same, i.e. n==0 is true

## 7 What can we learn from the SAT solvers output?

The program does not contain any division-by-zero errors.

We can claim this because we are using unrolling assertions, making the analysis sound.

## 8 How many boolean variables would a SAT solver require in order to obtain a valid encoding of the constrains after bit blasting?

10 variables: $n, n_0, n_1, n_2, n_3, n_4, n_5, b_0, b_1, b_2$

16 bit per variable

10 variables x 16bit = 160bit

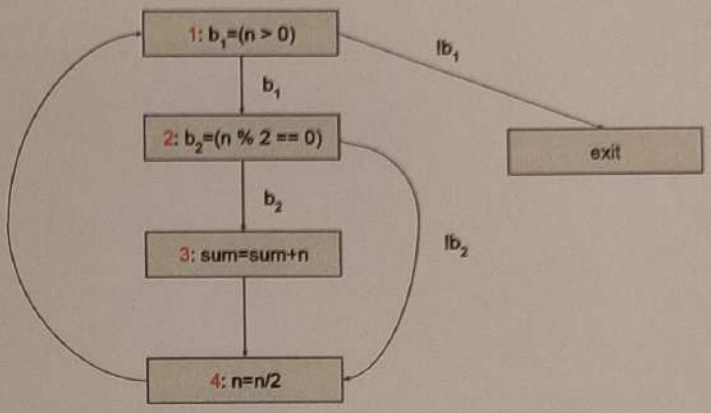
## 4.) Symbolic Execution (20 points) 20

Consider function smod below and its control flow graph (CFG). Perform Symbolic Execution on the path that is given in the table below: the loop body is executed exactly two times, and in the first loop iteration, line 4 is executed, whereas in the second loop iteration, line 4 is not executed. The numbers in column "Path" of the table refer to the labels of the nodes in the CFG. 3

- First, fill out the table, which tracks the symbolic map and the path condition as in the lecture. (15 points) 15
- Now, give a concrete value for input $n$ of the function such that the described path would be executed. (5 points) 5

↳ n = 2 ✓

```
int sum; // global var
void smod(int n){
  while(n > 0){
    if(n % 2 == 0)
      sum=sum+n;
    n=n/2;
  }
}
```




| Path | Symbolic map | Path condition |
|---|---|---|
| 1 | $n \mapsto N$ | true |
| 2 | $b_1 \rightarrow N > 0$ | $N > 0$ |
| 3 | $b_2 \rightarrow N \% 2 == 0$ | $N \% 2 == 0$ |
| 4 | $sum \rightarrow sum + N$ | true |
| 1 | $n \rightarrow N/2$ | true |
| 2 | $b_1 \rightarrow N/2 > 0$ | $N/2 > 0$ |
| 4 | $b_2 \rightarrow N/2 \% 2 == 0$ | $N/2 \% 2 \neq 0$ |
| 1 | $n \rightarrow (N/2)/2$ | true |
| exit | $b_1 \rightarrow (N/2)/2 > 0$ | $\neg((N/2)/2 > 0)$ |

✓

6/2 = 3 2 = 1

4

a Fill out the table

mandated Path: 1 -> 2 -> 3 -> 4 -> 1 -> 2 -> 4 -> 1 -> exit

| Path | Symbolic Map | Path condition |
|---|---|---|
| 1 | $n \to N$ | true |
| 2 | $b_1 \to N > 0$ | $N > 0$ |
| 3 | $b_2 \to N \bmod 2 = 0$ | $N \bmod 2 = 0$ |
| 4 | $sum \to sum + N$ | true |
| 1 | $n \to \frac{N}{2}$ | true |
| 2 | $b_1 \to \frac{N}{2} > 0$ | $\frac{N}{2} > 0$ |
| 4 | $b_2 \to \frac{N}{2} \bmod 2 = 0$ | $\frac{N}{2} \bmod 2 \neq 0$ |
| 1 | $n \to \frac{N}{4}$ | true |
| exit | $b_1 \to \frac{N}{4} > 0$ | $\frac{N}{4} \leq 0$ |

b Find concrete value so that the mandated path will be executed

Original Path condition:

$N > 0 \wedge N \bmod 2 = 0 \wedge \frac{N}{2} > 0 \wedge \frac{N}{2} \bmod 2 \neq 0 \wedge \frac{N}{4} \leq 0$

implies $N < 4$
because $N$ is an integer
meaning only values $< 4$ can
become 0, due to truncating.

Only possible solution
$n = 2$