☒ An imprecise and sound analyzer can be used for bug finding in a program. +1

☒ A precise and sound analyzer can be used for bug finding in a program. +1

☐ An imprecise and unsound analyzer can be used to verify a program.

☐ A precise and unsound analyzer can be used to verify a program.

☒ An imprecise and sound analyzer can be used to verify a program. +1

2.) Program:

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

✓

Notes:

0 + 0 = 0

0 + 3 = 3

2 + 3 = 5

2 + 0 = 2

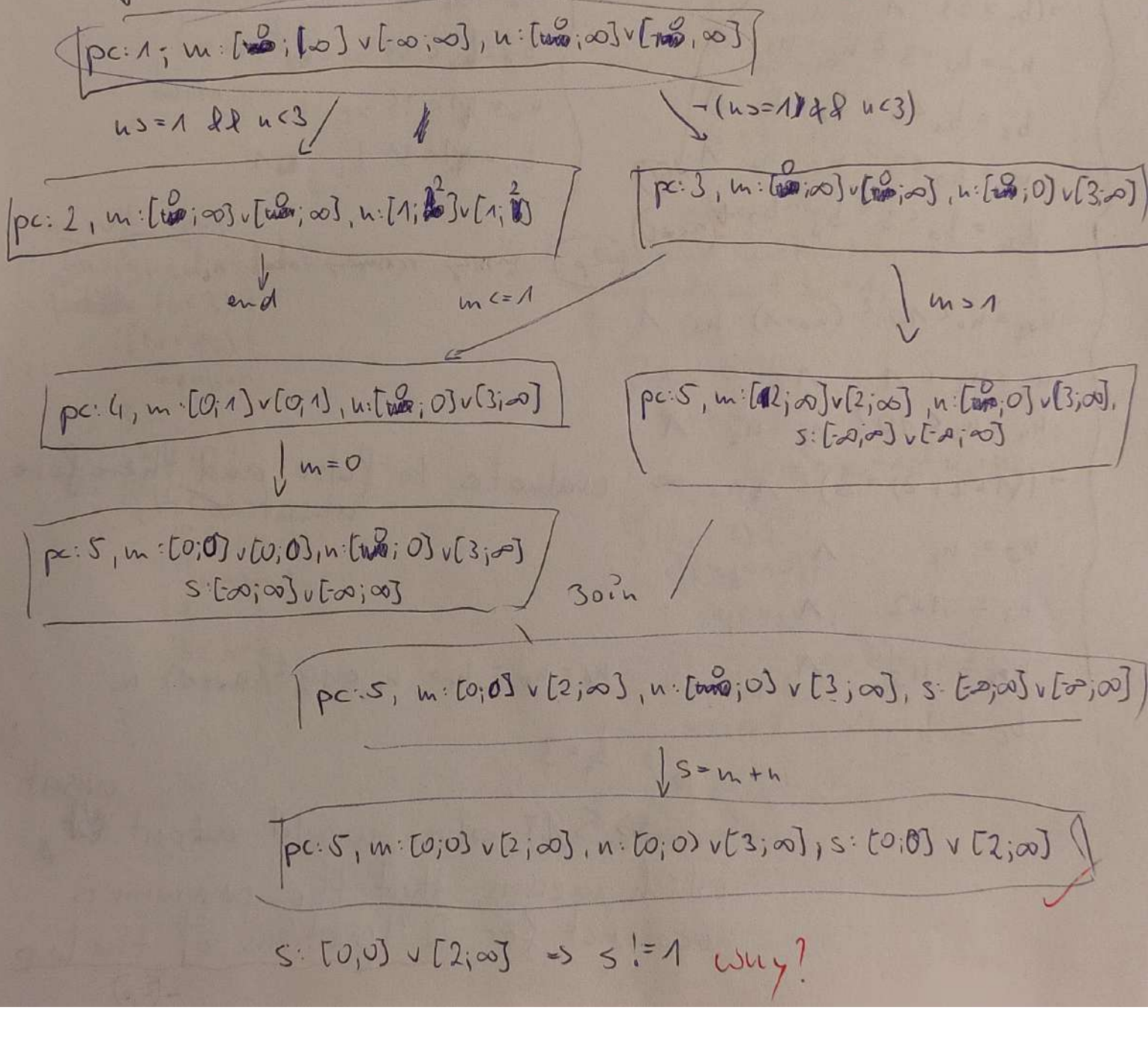
2.) Abstract Interpretation (33 points) 28

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



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;
```

3.) $b_0 = 1 \wedge$

$n_1 = n_0 + 1 \wedge$

$n_2 = n_0 < 10 \; ? \; n_1 : n_0 \wedge$

$b_1 = b_0 + 2 \wedge$

$n_3 = n_2 + 1 \wedge$

$n_4 = n_2 < 10 \; ? \; n_3 : n_2 \wedge$

$b_2 = b_1 + 2 \wedge$

$\neg(b_2 > 3) \wedge$

$n_5 = b_1 < 3 \; ? \; n_4 : n_2 \wedge$

$b_3 = b_1 < 3 \; ? \; b_2 : b_1 \wedge$

$n_6 = b_0 < 3 \; ? \; n_5 : n_0 \wedge$

$b_4 = b_0 < 3 \; ? \; b_3 : b_0 \wedge \neg(n_6 \neq 0)$

(V)

$n_2 = n_0 < 10 \; ? \; (n_0 + 1) : n_0 \wedge$

$n_3 = n_2 + 1 \wedge$

$n_4 = n_2 < 10 \; ? \; n_3 : n_2 \wedge$

$b_2 = 1 + 2 + 2 \wedge$

$\neg(b_2 \geq 3) \wedge$

$n_5 = (1+2) < 3 \; ? \; n_4 : n_2 \wedge$

$b_3 = (1+2) < 3 \; ? \; b_2 : (1+2) \wedge$

$n_6 = 1 < 3 \; ? \; n_5 : n_0 \wedge$

$b_4 = 1 < 3 \; ? \; b_3 : 1 \wedge$

-2p missing reasoning about n_6

$n_2 = n_0 < 10 \; ? \; (n_0 + 1) : n_0 \wedge$

$n_3 = n_2 + 1 \wedge$

$n_4 = n_2 < 10 \; ? \; n_3 : n_2 \wedge$

$\neg((1+2+2) > 3) \wedge$ $\Rightarrow$ evaluates to false and therefore unsat

$n_5 = n_2 \wedge$

$b_3 = 1 + 2 \wedge$

$n_6 = n_5 \wedge$

$b_4 = b_3$

$n_6 = n_5 = n_2 = n_0 < 10 \; ? \; (n_0 + 1) : n_0$

$b_4 = 3$

$\Rightarrow$ SAT solver would output unsat which means that the programm is correct for 2 iterations of the loop

-2p

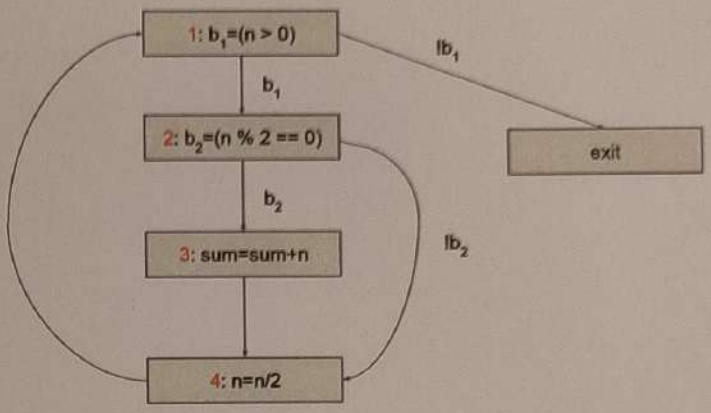
4.) Symbolic Execution (20 points) 20

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