VU Programm- und Systemverifikation
Assignment 1: Assertions, Testing, and Coverage

Name: _____ Matr. number: _____

Due: April 30, 1pm (no extension!)

1 Coverage Metrics

Consider the following program fragment and test suite:

```
bool range_check (unsigned m, unsigned n) {
  if (m > n) {
    unsigned t = m;
    m = n;
    n = t;
  }
  bool result = false;
  bool tmp = true;
  unsigned i = m;
  while ((i > 0) && (i < n)) {
    i = i + 1;
    if (i % m == 0)
        result = result || tmp;
  }
  return result;
}
```

Inputs		Outputs
`m`	`n`	result
3	7	true
1	0	false
2	5	true

1.1 Control-Flow-Based Coverage Criteria (3.5 points)

Indicate (✓) which of the following coverage criteria are satisfied by the test-suite above (important: assume that the term "decision" refers to all Boolean expressions that contain variables, including `result || tmp`).

	satisfied	
Criterion	yes	no
statement coverage	✓	
branch coverage	✓	
decision coverage		✓
modified condition/decision coverage		✓

For each coverage criterion that is *not* satisfied, explain why this is the case:

decision coverage `(result || tmp)` is never false

MC/DC `(result || tmp)` is never false

1.2 Data-Flow-Based Coverage Criteria (3.5 points)

Indicate (✓) which of the following coverage criteria are satisfied by the test-suite above (here, the parameters of the function do not constitute definitions, the `return` statement is a c-use, and `result = (result || tmp)` is a c-use but not a p-use):

	satisfied	
Criterion	yes	no
all-defs	✓	(✓)
all-c-uses		✓
all-p-uses		✓
all-c-uses/some-p-uses		✓
all-p-uses/some-c-uses		✓

For each coverage criterion that is not satisfied, explain why this is the case:

all-defs Technically, also not satisfied because there's no du-chain from `tmp=true` to its use that does not traverse the loop-head twice. However, this is subtle, and we've counted both answers (yes and no) as correct.

all-c-uses `result = result || tmp` never used in `result || tmp`; also no loop-free def-clear path from `result = false` to `result = result || tmp`

all-p-uses `m = n` never used in `i % m == 0`

all-p-uses/some-c-uses follows from **all-p-uses**

all-c-uses/some-p-uses follows from **all-c-uses**

1.3 Achieving Full Coverage (1 point)

Consider the two coverage criteria below.

- If the test-suite from above does not satisfy the coverage criterion, augment it with the *minimal* number of test-cases such that this criterion is satisfied. If full coverage cannot be achieved, explain why.
- If the coverage criterion is already achieved, explain why.

all-p-uses

Inputs		Outputs
`m`	`n`	result
7	3	

all-c-uses

Inputs		Outputs
`m`	`n`	result
1	3	

1.4 Modified Condition/Decision Coverage (1 point)

Consider the expression $((a \wedge b) \vee c)$, where a, b, and c are Boolean variables. Provide a *minimal* number of test cases such that modified condition/decision coverage is achieved for the expression. Clarify for each test case *which* condition(s) independently affect(s) the outcome.

MC/DC

Inputs			Outcome	Covered
`a`	`b`	`c`	`(a && b) || c`	
0	1	0	0	$\neg a$, $\neg c$
1	0	0	0	$\neg b$, $\neg c$
0	0	1	1	c
1	1	0	1	a, b

2 Equivalence Partitioning and Boundary Testing

The function

```
bool mcas(float aoa_sensor0, float aoa_sensor1, float trim);
```

is a (extremely simplified) implementation of Boeing's updated MCAS (Maneuvering Characteristics Augmentation System) software (e.g., the state of the autopilot and the flaps is ignored). The system activates when the sensed *angle of attack* (AOA) exceeds a threshold based on airspeed and altitude. For the sake of simplicity, we assume that this threshold is a constant represented by `t` (a valid angle larger than 0 and smaller than 90 degrees).

If the threshold `t` is exceeded by at least one sensor, MCAS tilts the horizontal stabilizer upward at a rate of 0.27 degrees per second for a total travel of 2.5 degrees in just under 10 seconds. The output value of `mcas` indicates whether the stabilizer trim still needs to be increased at the specified rate. If the two sensors disagree by at least 5 degrees, the MCAS system is deactivated for safety reasons.

- The angle of attack sensors `aoa_sensor0` and `aoa_sensor1` provide the output of two sensors measuring the angle of attack (between 0 and 90 degrees).
- The parameter `trim` represents the movement of the horizontal stabilizer trim that has already been achieved since the activation of MCAS (between 0 and 2.5 degrees).

2.1 Equivalence Partitioning (3.5 points)

From the specification above, derive equivalence classes for the function `mcas`. Use the table below to partition them into *valid equivalence classes* (valid inputs) and *invalid equivalence classes* (invalid inputs). Label each of the equivalence classes clearly with a number (in the according column). For each correct *equivalence class* you can score $\frac{1}{2}$ a point (up to 3.5 points).
(Do not provide test-cases here – that's task 2.2)

Condition	**Valid**	**ID**	**Invalid**	**ID**
$(\texttt{aoa\_sensor0} > t) \vee (\texttt{aoa\_sensor1} > t)$	$(\texttt{aoa\_sensor0} > t) \wedge (\texttt{aoa\_sensor1} > t)$	1		
$(\texttt{aoa\_sensor0} > t) \vee (\texttt{aoa\_sensor1} > t)$	$(\texttt{aoa\_sensor0} > t) \wedge (\texttt{aoa\_sensor1} \leq t)$	2		
$(\texttt{aoa\_sensor0} > t) \vee (\texttt{aoa\_sensor1} > t)$	$(\texttt{aoa\_sensor0} \leq t) \wedge (\texttt{aoa\_sensor1} > t)$	3		
$(\texttt{aoa\_sensor0} > t) \vee (\texttt{aoa\_sensor1} > t)$	$(\texttt{aoa\_sensor0} \leq t) \wedge (\texttt{aoa\_sensor1} \leq t)$	4		
$\lvert\texttt{aoa\_sensor0} - \texttt{aoa\_sensor1}\rvert \geq 5$	no	5		
$\lvert\texttt{aoa\_sensor0} - \texttt{aoa\_sensor1}\rvert \geq 5$	yes	6		
$0 \leq \texttt{aoa\_sensor0} \leq 90$	yes	7	no	8
$0 \leq \texttt{aoa\_sensor1} \leq 90$	yes	9	no	10
$0 \leq \texttt{trim} \leq 2.5$	yes	11	no	12

2.2 Boundary Value Testing (3.5 points)

Use *Boundary Value Testing* to derive a test-suite for the function `mcas`. Specify the inputs points for `mcas`. Indicate clearly which equivalence classes each test-case covers by referring to the numbers from task (a). You can receive up to 3.5 points ($\frac{1}{2}$ a point per test-case), where redundant test-cases and test-cases that do not represent boundary values do not count.

We use ϵ to represent `FLT_EPSILON` and assume that t is a constant such that $5 \leq t \leq (85 - \epsilon)$. Note that this is not a "complete" set of test cases – there are many more acceptable tests that can be obtained by *combining* the equivalence classes in different ways. For simplicity, we also ignore the issue that a comparison of floating point numbers using equality is problematic. For invalid equivalence classes, we only list the single invalid equalence class that is covered.

Input	**Output**	**Classes Covered**
`aoa_sensor0` $= t + \epsilon$, `aoa_sensor1` $= t + \epsilon$, `trim`=0	true	1, 5, 7, 9, 11
`aoa_sensor0` $= t + \epsilon$, `aoa_sensor1` $= t + \epsilon$, `trim`=2.5	false	1, 5, 7, 9, 11
`aoa_sensor0` $= t + \epsilon$, `aoa_sensor1` $= t$, `trim`=0	true	2, 5, 7, 9, 11
`aoa_sensor0` $= t$, `aoa_sensor1` $= t + \epsilon$, `trim`=0	true	3, 5, 7, 9, 11
`aoa_sensor0` $= t$, `aoa_sensor1` $= t$, `trim`=0	false	4, 5, 7, 9, 11
`aoa_sensor0` $= t + \epsilon$, `aoa_sensor1` $= t + 5 + \epsilon$, `trim`=0	false	1, 6, 7, 9, 11
`aoa_sensor0` $= t + \epsilon$, `aoa_sensor0` $= t - 5$, `trim`=0	false	2, 6, 7, 9, 11
`aoa_sensor0` $= 90 + \epsilon$, `aoa_sensor1` $= 90$, `trim`=0	error	8
`aoa_sensor0` $= -\epsilon$, `aoa_sensor1` $= 0$, `trim`=0	error	8
`aoa_sensor0` $= 0$, `aoa_sensor1` $= -\epsilon$, `trim`=0	error	10
`aoa_sensor0` $= 90$, `aoa_sensor1` $= 90 + \epsilon$, `trim`=0	error	10
`aoa_sensor0` $= 0$, `aoa_sensor1` $= 0$, `trim` $= -\epsilon$,	error	12
`aoa_sensor0` $= 0$, `aoa_sensor1` $= 0$, `trim` $= 2.5 + \epsilon$,	error	12
...		

3 Invariants (4 points)

Consider the following program, where `x` and `y` are non-negative natural numbers (possibly 0):

```
x = 0; y = 50;
while (x != y) {
  x = x + 1;
  if (x % 2 == 0) {
    y = y + 1;
  }
}
```

Consider the formulas below; tick the correct box (☑) to indicate whether they are loop invariants for the program above.

- If the formula is an inductive invariant for the loop, provide an informal argument that the invariant is inductive.
- If the formula P is an invariant that is *not* inductive, give values of `x` and `y` before and after the loop body demonstrating that the Hoare triple

$$\{P \wedge B\} \quad \texttt{x} = \texttt{x} + 1;\ \texttt{if}\ (\texttt{x}\%2 == 0)\ \texttt{y} = \texttt{y} + 1;\ \texttt{else skip};\quad \{P\}$$

  (where B is `(x != y)`) does not hold.
- Otherwise, provide values of `x` and `y` that correspond to a reachable state showing that the formula is *not* an invariant.

$(x \leq 100) \wedge (y \leq 100)$	☐ Inductive Invariant	× Non-inductive Inv.	☐ Neither
Justification: Invariant, since program terminates with `x`=`y`=100. Non-inductive, since successor of `x`=50 and `y`=100 violates assertion.			
$(y - x) \leq 50$	× Inductive Invariant	☐ Non-inductive Inv.	☐ Neither
Justification: Base case holds (`x`=0, `y`=50). Induction step: `x` is always increased by 1, `y` only sometimes, therefore difference (`y`-`x`) can only become smaller.			
$(x \neq y)$	☐ Inductive Invariant	☐ Non-inductive Inv.	× Neither
Justification: Program terminates with `x`=`y`=100, violating the assertion.			
$(x \leq y)$	× Inductive Invariant	☐ Non-inductive Inv.	☐ Neither
Justification: Base case holds: `x`=0, `y`=50. Induction step: Since $(\texttt{x} \neq \texttt{y}) \wedge (\texttt{x} \leq \texttt{y})$ upon loop entrance, and `x` gets incremented by one at most, $(\texttt{x} \leq \texttt{y})$ still holds afterwards.			