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

Name: _____ Matr. number: _____

Due: April 7, 2pm

## 1 Coverage Metrics

Consider the following program fragment and test suite:

```
boolean coprime (unsigned x, unsigned y) {
  unsigned k = x;
  unsigned m = y;
  if ((k == 0) || (m == 0)) {
    return ((k == 1) || (m == 1));
  }
  while (k != m) {
    if (k > m) {
        k = k - m;
    } else {
        m = m - k;
    }
  }
  return (k == 1);
}
```

| Inputs | | Outputs |
|---|---|---|
| x | y | result |
| 0 | 1 | true |
| 0 | 5 | false |
| 14 | 15 | true |
| 14 | 21 | false |

### 1.1 Control-Flow-Based Coverage Criteria (3 points)

Indicate (✓) which of the following coverage criteria are satisfied by the test-suite above (assume that the term "decision" refers to all Boolean expressions in the program).

| | satisfied | |
|---|---|---|
| **Criterion** | yes | no |
| path coverage | | |
| statement coverage | | |
| branch coverage | | |
| decision coverage | | |
| condition/decision coverage | | |

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

### 1.2 Data-Flow-Based Coverage Criteria (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, and the `return` statements are p-uses but not c-uses):

| | 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:

### 1.3 Modified Condition/Decision Coverage (1 point)

*If* the test-suite from above does not satisfy the MC/DC 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.

**MC/DC**

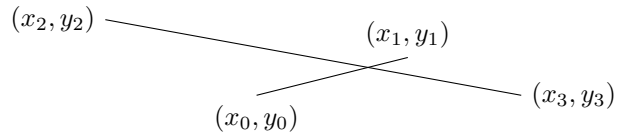
| Inputs | | Outputs |
|---|---|---|
| `x` | `y` | result |
| | | |

## 2 Equivalence Partitioning and Boundary Testing

The function

```
int intersect(float x0, float y0, float x1, float y1,
              float x2, float y2, float x3, float y3)
```

takes as parameters 4 points $(x_i, y_i)$ (for $0 \leq i \leq 3$) and determines whether the two *line segments* $(x_0, y_0) - (x_1, y_1)$ and $(x_2, y_2) - (x_3, y_3)$ intersect:




- All points are of type $\mathbb{R} \times \mathbb{R}$.
- The points $(x_0, y_0)$ and $(x_2, y_2)$ must be the "left" points of the line segments, i.e., $x_0 \leq x_1$ and $x_2 \leq x_3$ (as in the figure above).
- The line segments must have a non-zero length.
- `intersect` returns
  - 0 if the two lines are *parallel*
  - 1 if the two lines are *not* parallel and do not intersect
  - 2 if the two lines intersect

### 2.1 Equivalence Partitioning (4 points)

From the specification above, derive equivalence classes for the method `intersect`. 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 half a point (up to 4 points).
(Do not provide test-cases here – that's task (b))

| **Condition** | **Valid** | **ID** | **Invalid** | **ID** |
|---|---|---|---|---|
| | | | | |

### 2.2 Boundary Value Testing (4 points)

Use *Boundary Value Testing* to derive a test-suite for the method `intersect`. Specify the inputs points $p_0 = (x_0, y_0), p_1 = (x_1, y_1), p_2 = (x_2, y_2)$, and $p_3 = (x_3, y_3)$ (e.g., $p_0 = (3, 3), p_1 = (1, 2), p_2 = (0, 0), p_3 = (4, 4)$). Indicate clearly which equivalence classes each test-case covers by referring to the numbers from task (a). You can receive up to 4 points ($\frac{1}{2}$ a point per test case), where test-cases that do not represent boundary values do not count.

| **Input** | **Output** | **Classes Covered** |
|---|---|---|
| | | |

## 3 Invariants (3 points)

```
int x, y;
x = 0;
y = 0;
while (x < 2016 && y < 63) {
  y = y + 1;
  x = x + y;
}
```

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, provide an (informal) proof 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

$$\{P \wedge B\} \quad \texttt{y} = \texttt{y} + \texttt{1}; \ \texttt{x} = \texttt{x} + \texttt{y} \quad \{P\}$$

  (where $B = (x < 2016 \wedge y < 63)$) does not hold.
- Otherwise, provide values of `x` and `y` that correspond to a reachable state showing that the formula is *not* an invariant.

| | | | |
|---|---|---|---|
| $x \geq y$ | ☐ Inductive Invariant | ☐ Non-inductive Invariant | ☐ Neither |
| Justification: | | | |
| $y \geq 0 \wedge x \geq y$ | ☐ Inductive Invariant | ☐ Non-inductive Invariant | ☐ Neither |
| Justification: | | | |
| $x = 0 \vee x \neq 3 \cdot y$ | ☐ Inductive Invariant | ☐ Non-inductive Invariant | ☐ Neither |
| Justification: | | | |

Please hand in your assignment via TUWEL (as a single PDF file) by April 7, 2016, 2pm.