6.0 ECTS/4.5h VU Programm- und Systemverifikation (184.741)

June 17, 2015

Kennzahl (study id)	Matrikelnummer (student id)	Familienname (family name)	Vorname (first name)	Gruppe (version)
				A

1.) Coverage

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

(a) **Control-Flow-Based Coverage Criteria**

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:

(7 points)

(b) **Data-Flow-Based Coverage Criteria**

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:

(11 points)

(c) *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

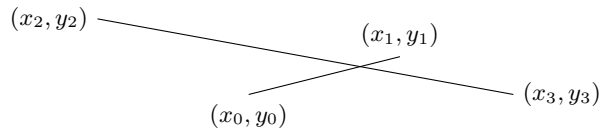
(2 points)

2.) Black/Gray-Box 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

(a) **Equivalence Partitioning**

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 one point (up to 10 points).
(Do not provide test-cases here – that's task (b))

Condition	Valid	ID	Invalid	ID

(10 points)

(b) **Boundary Value Testing**

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 10 points, where test-cases that do not represent boundary values do not count.

Input	**Output**	**Classes Covered**

(10 points)

3.) (a) **Invariants**

```
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

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

  (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:			

(10 points)

(b) **Hoare Logic**

Prove the Hoare Triple below (assume that the domain of all variables in the program are the natural numbers including 0, i.e., $\texttt{x}, \texttt{y} \in \mathbb{N}_0$ or, equivalently, both `x` and `y` are of type `unsigned`). You need to find a sufficiently strong loop invariant.

Annotate the following code directly with the required assertions. Justify each assertion by stating which Hoare rule you used to derive it, and the premise(s) of that rule.

$\{\mathsf{true}\}$

```
if (y > x) {

  t := x;

  x := y;

  y := t;

} else {

  skip;

}

while ((y > 0) && (x % y != 0)) {

  y := y - 1;

}
```

$\{(x \neq 0) \vee (y = 0)\}$

(10 points)

4.) Decision procedures

(a) Consider the following formulas in propositional logic; are they satisfiable? If yes, provide a satisfying assignment over booleans, if not, give the reasoning that leads to this conclusion.

$$(\neg a \vee d) \wedge \neg b \wedge c \wedge \neg e \wedge (a \vee c \vee \neg a) \wedge (e \vee \neg a \vee \neg d) \wedge \neg e \tag{1}$$

$$\neg b \wedge (b \vee \neg a) \wedge e \wedge (\neg d \vee a \vee \neg c) \wedge (c \vee c) \wedge d \wedge (\neg e \vee e \vee \neg e) \tag{2}$$

$$(\neg e \vee d) \wedge \neg e \wedge b \wedge (\neg a \vee e \vee \neg c) \wedge a \wedge (b \vee \neg e \vee a) \wedge c \tag{3}$$

$$(\neg b \vee d) \wedge (\neg e \vee \neg b) \wedge b \wedge a \wedge (\neg d \vee a \vee d) \wedge (b \vee b) \wedge (\neg c \vee c) \tag{4}$$

$$(\neg a \vee \neg e) \wedge d \wedge (\neg d \vee \neg d) \wedge (\neg d \vee a) \wedge \neg a \wedge e \wedge b \tag{5}$$

(5 points)

(b) Consider the following formulas in Equality Logic; are they satisfiable? If yes, provide a satisfying assignment over integers, if not, give the reasoning based on equivalence classes that leads to this conclusion.

$$a = b \ \wedge \ b = c \ \wedge \ c = d \ \wedge \ b = e \ \wedge \ c \neq d \tag{6}$$

$$a = c \ \wedge \ a \neq b \ \wedge \ b = d \ \wedge \ c \neq d \ \wedge \ d = e \ \wedge \ f = e \tag{7}$$

$$b = c \wedge c = d \wedge e = f \ \wedge \ f = g \wedge g = h \wedge d \neq h \wedge a = f \wedge b = e \wedge b = g \wedge c = f \tag{8}$$

(6 points)

(c) Check the satisfiability of the following SMT formulas. Assume that $x, y, z, a, b, c \in \mathbb{Z}$ are integer constants, and $f : \mathbb{Z} \to \mathbb{Z}$ and $g : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ are unary and binary uninterpreted functions over integers respectively. Whenever a formula is satisfiable, give a satisfying assignment for it, i.e., integer values for all variables and function intepretations over integers that make the formula true under the assignment. Whenever a formula is not satisfiable, give a reason why it is unsatisfiable.

$$f(f(f(1))) = 1 \wedge f(f(1)) \neq g(x,1) \wedge x = f(f(1)) \\ \wedge g(1,x) = 2 \wedge g(1,x) = g(x,1) \wedge f(x) = g(1,x) \quad (9)$$

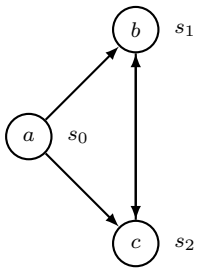
$$g(x,y) = g(y,x) \wedge g(0,1) = a \wedge g(1,0) = b \wedge a \neq b \wedge g(x,x) = x \wedge g(y,y) = y \\ \wedge (g(x,x) = 0 \vee g(x,x) = 1) \wedge (g(y,y) = 0 \vee g(y,y) = 1) \quad (10)$$

$$g(2,y) = 6 \wedge g(y,x) = g(x,y) \wedge g(y,2) = 8 \quad (11)$$

(9 points)

5.) Temporal Logic

Consider the following Kripke Structure:



(a) For each formula, give the states of the Kripke structure, where the formula holds. In other words, consider the computation trees starting with one of the states from the set $\{s_0, s_1, s_2\}$, and for each tree check, whether the given formula holds on it, or not.

i. $a \wedge \mathbf{EX}\, b \wedge \mathbf{EX}\, c$
ii. $\mathbf{AF}\,\mathbf{AG}\,(\mathbf{AX}\, b \vee \mathbf{AX}\, c)$
iii. $\mathbf{EF}\,\mathbf{EG}\,\neg b$
iv. $\mathbf{AG}\,(b\ \mathbf{AU}\ c)$
v. $a\ \mathbf{AU}\,(\mathbf{EG}\,\neg c)$

(10 points)

6.) Concurrency

Consider the following program running the threads `thread1` and `thread2` in parallel:

```
1  #include <pthread.h>
2  #include <stdlib.h>
3  #include <assert.h>

5  // data structure to store value
       and absolute value
6  typedef struct {
7    float abs;
8    float val;
9  } value_t;

11 value_t n, m;
12 pthread_mutex_t *m_lock;
13 pthread_mutex_t *n_lock;

15 // first thread
16 void* thread1 (void *p) {
17   pthread_mutex_lock (n_lock);
18   // copy value of m to n
19   n.abs = m.abs;
20   n.val = m.val;
21   pthread_mutex_unlock (n_lock);
22   pthread_exit(NULL);
23   return NULL;
24 }

26 // second thread
27 void* thread2 (void *p) {
28   pthread_mutex_lock (m_lock);
29   // copy value of n to m
30   m.abs = n.abs;
31   m.val = n.val;
32   pthread_mutex_unlock (m_lock);
33   pthread_exit(NULL);
34   return NULL;
35 }

37 // main function
38 int main()
39 {
40   // initialize n
41   n_lock = malloc
42     (sizeof(pthread_mutex_t));
43   pthread_mutex_init
44     (n_lock, NULL);
45   n.abs = 5;
46   n.val = -5;

48   // initialize m
49   m_lock = malloc
50     (sizeof(pthread_mutex_t));
51   pthread_mutex_init
52     (m_lock, NULL);
53   m.abs = 7;
54   m.val = 7;

56   pthread_t t1, t2;
57   pthread_create
58     (&t1, NULL, thread1, NULL);
59   pthread_create
60     (&t2, NULL, thread2, NULL);

62   pthread_join (t1, NULL);
63   pthread_join (t2, NULL);

65   assert (n.abs == n.val ||
66           n.abs == -n.val);

68   pthread_exit (NULL);
69   return 0;
70 }
```

The program contains two locks `n_lock` and `m_lock`, which are acquired and released by the `pthread`-library functions `pthread_mutex_lock` and `pthread_mutex_unlock`, respectively. The function `pthread_create` spawns a new thread executing the function provided as third argument with the parameter provided as fourth argument (which is always `NULL` in our case), and stores its identifier in the first parameter. The function `pthread_join` waits until the thread identified by the first parameter finishes.

(a) Identify an execution in which the assertion in lines 65-66 fails.

- Describe the order in which the statements of `thread1` and `thread2` are executed in the failing execution (by referring to the line numbers).
- Identify the problematic data dependencies (hazards) between the two threads.

(5 points)

(b) For the execution described in task (a), describe the fault, the error, and the failure.

Fault

Error

Failure

(5 points)