Program Testing and Analysis

—Solutions of Mid-term Exam—

Department of Computer Science
TU Darmstadt

Winter semester 2016/17, December 12, 2016

Note: The solutions provided here may not be the only valid solutions.

Part 1 [4 points]

1. Which of the following statements is true? (Only one statement is true.)
   - ☐ Functional testing checks whether the program computes a mathematical function.
   - ☐ Functional testing is a testing technique for individual functions.
   - ☒ Functional testing checks the functional requirements of a program, which describe what the program is supposed to accomplish.
   - ☐ Functional testing checks whether the user interface of an application is usable for non-experts.
   - ☐ Functional testing is a testing methodology for programs written in a functional programming language.

2. Which of the following statements is true? (Only one statement is true.)
   - ☐ Feedback-directed test generation uses feedback from the developer to create effective tests.
   - ☒ Feedback-directed test generation uses feedback from test executions to steer the generation of additional tests.
   - ☐ Feedback-directed test generation considers the program as a blackbox.
   - ☐ Feedback-directed test generation gathers path constraints and solves them using a SMT solver.
   - ☐ Feedback-directed test generation provides feedback about the effectiveness of a test suite to the developer.

3. Which of the following statements is true? (Only one statement is true.)
   - ☒ The control flow graph of a function with a finite number of statements always has a finite number of nodes.
   - ☐ The abstract syntax tree of a function with a finite number of statements may have an infinite number of nodes.
   - ☐ The execution tree of a function with a finite number of statements always has a finite number of edges.
   - ☐ The control flow graph and the abstract syntax tree of a function generally have the same set of nodes.
   - ☐ The execution tree of a function with a finite number of statements always has a finite number of nodes.

4. Which of the following statements is true? (Only one statement is true.)
   - ☐ A test suite with full branch coverage detects all bugs.
   - ☐ A test suite with full path coverage detects all bugs.
   - ☐ A test suite will full statement coverage also has full path coverage.
   - ☐ A test suite with full DU-pair coverage also has full statement coverage.
   - ☒ A test suite with full path coverage also has full branch coverage.

Part 2 [8 points]

Consider the transition rules that define the semantics of expressions for the abstract machine for SIMP programs.

$$
\begin{array}{rcll}
\langle n \cdot c, r, m\rangle & \rightarrow & \langle c, n \cdot r, m\rangle & \\
\langle b \cdot c, r, m\rangle & \rightarrow & \langle c, b \cdot r, m\rangle & \\
 & & & \\
\langle \neg B \cdot c, r, m\rangle & \rightarrow & \langle B \cdot \neg \cdot c, r, m\rangle & \\
\langle (B_1 \ \wedge \ B_2) \cdot c, r, m\rangle & \rightarrow & \langle B_1 \cdot B_2 \cdot \wedge \cdot c, r, m\rangle & \\
\langle \neg \cdot c, b \cdot r, m\rangle & \rightarrow & \langle c, b' \cdot r, m\rangle & \text{if } b' = not\ b \\
\langle \wedge \cdot c, b_2 \cdot b_1 \cdot r, m\rangle & \rightarrow & \langle c, b \cdot r, m\rangle & \text{if } b_1\ and\ b_2 = b \\
 & & & \\
\langle (E_1\ op\ E_2) \cdot c, r, m\rangle & \rightarrow & \langle E_1 \cdot E_2 \cdot op \cdot c, r, m\rangle & \\
\langle (E_1\ bop\ E_2) \cdot c, r, m\rangle & \rightarrow & \langle E_1 \cdot E_2 \cdot bop \cdot c, r, m\rangle & \\
\langle op \cdot c, n_2 \cdot n_1 \cdot r, m\rangle & \rightarrow & \langle c, n \cdot r, m\rangle & \text{if } n_1\ op\ n_2 = n \\
\langle bop \cdot c, n_2 \cdot n_1 \cdot r, m\rangle & \rightarrow & \langle c, b \cdot r, m\rangle & \text{if } n_1\ bop\ n_2 = b \\
 & & & \\
\langle !l \cdot c, r, m\rangle & \rightarrow & \langle c, n \cdot r, m\rangle & \text{if } m(l) = n
\end{array}
$$

The above rules are exactly what has been presented in the lecture. Suppose to change the rule

$$\langle op \cdot c, n_2 \cdot n_1 \cdot r, m\rangle \ \rightarrow \ \langle c, n \cdot r, m\rangle \qquad \text{if } n_1\ op\ n_2 = n$$

into this rule:

$$\langle op \cdot c, n_1 \cdot n_2 \cdot r, m\rangle \ \rightarrow \ \langle c, n \cdot r, m\rangle \qquad \text{if } n_1\ op\ n_2 = n$$

1. Describe how this change affects the semantics of expressions.

   Solution:

   The changed rule inverts the order in which operands are taken from the results stack. As a result, the operand that has been evaluated and pushed first (second) will be considered as the second (first) operand when performing the operation. This change affects the behavior for operations that are not symmetric, such as arithmetic subtraction.

2. Provide a SIMP expression that has a different semantics under the changed rules than under the original rules. Try to use an expression that is as simple as possible.

   Solution:

   ```
   3 - 1
   ```

3. Show that the semantics of your expression is different under the two sets of rules. For this purpose, provide for both sets of rules the sequence of transitions that computes the value of the expression.

   - Sequence of transitions for the original rules:

     Solution:

     Usual abstract machine:

     $\langle (3-1)\circ c, r, m\rangle$

     $\rightarrow \langle 3\circ 1\circ -\circ c, r, m\rangle$

     $\rightarrow \langle 1\circ -\circ c, 3\circ r, m\rangle$

     $\rightarrow \langle -\circ c, 1\circ 3\circ r, m\rangle$

     $\rightarrow \langle c, 2\circ r, m\rangle$

   - Sequence of transitions for the changed rules:

     Solution:

     Modified abstract machine:

     $\langle (3-1)\circ c, r, m\rangle$

     $\rightarrow$ ... (2 steps as above)

     $\rightarrow \langle -\circ c, 1\circ 3\circ r, m\rangle$

     $\rightarrow \langle c, -2\circ r, m\rangle$

     The difference is that the value of the expression is -2 instead of 2.

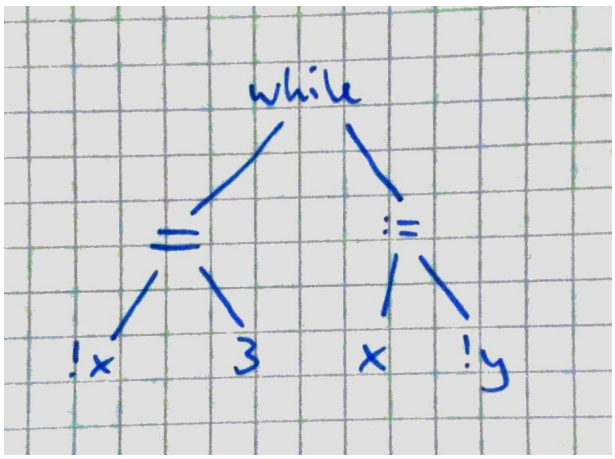
Part 3 [15 points]

Consider the following SIMP program:

```
while !x = 3 do x := !y
```

1. Draw the abstract syntax tree for the program.

   Solution:

2. Suppose that the program is executed with an initial store $s = \{x \mapsto 3, y \mapsto 1, z \mapsto 4\}$. Give the evaluation sequence using small-step operational semantics. For your reference, the axioms and rules are provided in the appendix. Use the following template to write your solution. You do not have to provide any proof trees for rules that have preconditions.

   Solution:

   $\langle$`while !x = 3 do x := !y`, $s\rangle$

   $\rightarrow \langle$`if !x = 3 then (x := !y; while !x = 3 do x := !y) else skip`, $s\rangle$

   $\rightarrow \langle$`if 3 = 3 then (x := !y; while !x = 3 do x := !y) else skip`, $s\rangle$

   $\rightarrow \langle$`if True then (x := !y; while !x = 3 do x := !y) else skip`, $s\rangle$

   $\rightarrow \langle$`x := !y; while !x = 3 do x := !y`, $s\rangle$

   $\rightarrow \langle$`x := 1; while !x = 3 do x := !y`, $s\rangle$

   $\rightarrow \langle$`skip; while !x = 3 do x := !y`, $s[x \mapsto 1]\rangle$

   $\rightarrow \langle$`while !x = 3 do x := !y`, $s[x \mapsto 1]\rangle$

   $\rightarrow \langle$`if !x = 3 then (x := !y; while !x = 3 do x := !y) else skip`, $s[x \mapsto 1]\rangle$

   $\rightarrow \langle$`if 1 = 3 then (x := !y; while !x = 3 do x := !y) else skip`, $s[x \mapsto 1]\rangle$

   $\rightarrow \langle$`if False then (x := !y; while !x = 3 do x := !y) else skip`, $s[x \mapsto 1]\rangle$

   $\rightarrow \langle$`skip`, $s[x \mapsto 1]\rangle$

3. Is the program divergent?

   Solution: No

4. Is the program blocked?

   Solution: No

5. Does the program terminate?

   Solution: Yes

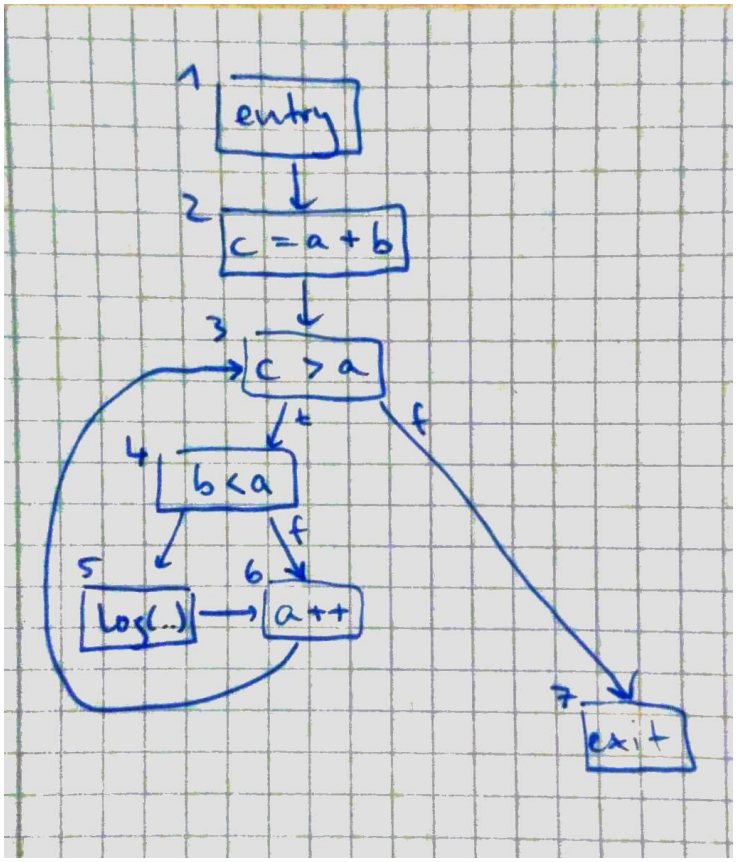
Part 4 [13 points]

Consider the following JavaScript function:

```
1 function f(a, b) {
2     var c = a + b;
3     while (c > a) {
4         if (b < a) {
5           console.log("hi there");
6         }
7         a++;
8     }
9 }
```

1. Draw the control flow graph of the function.

   Solution:



2. Give the DU-pairs for the three variables a, b, and c. Do not count function parameters as definitions. Use the line numbers to identify code locations.

   Solution:

   - a: (7,3), (7,4), (7,7)
   - b: none
   - c: (2,3)

3. Suppose the function is tested with the following test suite:

   - `f(1, 1)`
   - `f(1, 0)`

   (a) What is the statement coverage achieved by the test suite?

   Solution: 4/5

   (b) What is the branch coverage achieved by the test suite?

   Solution: 3/4

   (c) What is the loop coverage achieved by the test suite?

   Solution: 2/3

   (d) What is the DU-pairs coverage achieved by the test suite?

   Solution: 2/4

4. Extend the test suite with a minimal number of additional tests that provide full statement coverage, full branch coverage, full loop coverage, and full DU-pair coverage.

   Solution: `f(4, 2)`

Part 5 [12 points]

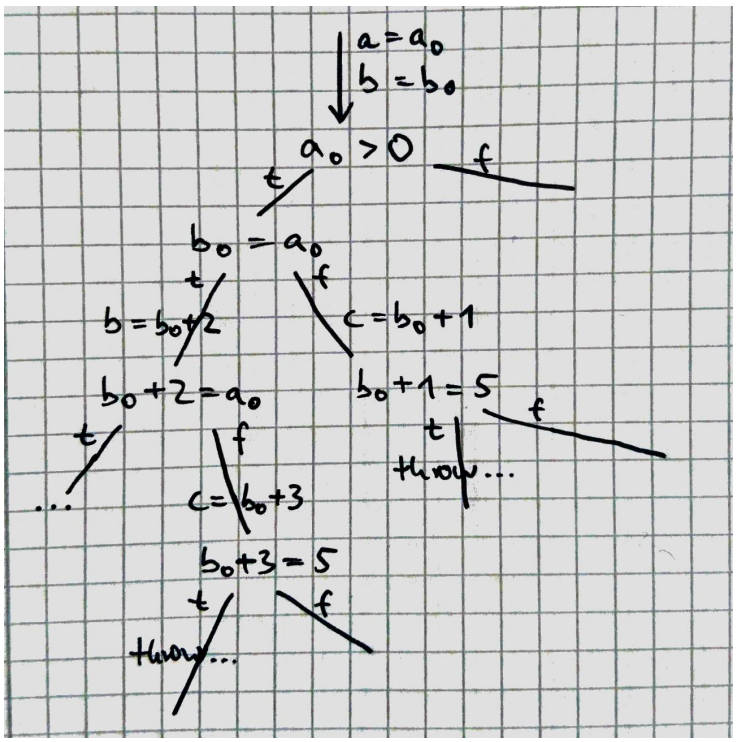
Consider the following JavaScript program:

```
1  function f(a, b) {
2      if (a > 0) {
3          while (b === a) {
4              b = b + 2;
5          }
6          var c = b + 1;
7          if (c === 5) {
8              throw "Error";
9          }
10     }
11 }
```

Suppose to use concolic testing to analyze the program, where `a` and `b` are considered as symbolic variables.

1. Draw the execution tree of the program. If the tree is infinitely large, use "..." to represent repeating parts of the tree.

   Solution:



2. Suppose that concolic testing starts with the following concrete inputs: `a = 1` and `b = 2`. Illustrate the execution using the following table.

   Solution:

Line	After executing the line		
	State of concrete execution	State of symbolic execution	Path condition
2	$a = 1, b = 2$	$a = a_0, b = b_0$	$a_0 > 0$
3	$a = 1, b = 2$	$a = a_0, b = b_0$	$a_0 > 0 \wedge b_0 \neq a_0$
6	$a = 1, b = 2, c = 3$	$a = a_0, b = b_0, c = b_0 + 1$	$a_0 > 0 \wedge b_0 \neq a_0$
7	$a = 1, b = 2, c = 3$	$a = a_0, b = b_0, c = b_0 + 1$	$a_0 > 0 \wedge b_0 \neq a_0 \wedge b_0 + 1 \neq 5$

3. What is the formula that concolic testing gives to the SMT solver after the first execution?

   Solution:

   $a_0 > 0 \wedge b_0 \neq a_0 \wedge b_0 + 1 = 5$

4. Suppose that the formula is satisfiable and that it yields the solution $a_0 = 1$ and $b_0 = 4$. Illustrate the second execution using the following table.

   Solution:

Line	After executing the line		
	State of concrete execution	State of symbolic execution	Path condition
2	$a = 1, b = 4$	$a = a_0, b = b_0$	$a_0 > 0$
3	$a = 1, b = 4$	$a = a_0, b = b_0$	$a_0 > 0 \wedge b_0 \neq a_0$
6	$a = 1, b = 4, c = 5$	$a = a_0, b = b_0, c = b_0 + 1$	$a_0 > 0 \wedge b_0 \neq a_0$
7	$a = 1, b = 4, c = 5$	$a = a_0, b = b_0, c = b_0 + 1$	$a_0 > 0 \wedge b_0 \neq a_0 \wedge b_0 + 1 = 5$
8	$a = 1, b = 4, c = 5$	$a = a_0, b = b_0, c = b_0 + 1$	$a_0 > 0 \wedge b_0 \neq a_0 \wedge b_0 + 1 = 5$

5. Does the second execution reach the `throw` statement?

   Solution: Yes

Part 6 [8 points]

Consider the following concurrent program (using a Java-like syntax). Suppose that Thread 1 executes before the other threads, and that Threads 2 and 3 execute concurrently.

```
// Thread 1:
boolean b = true;
int foo = 20;
```

```
// Thread 2:
synchronized(L) { // acquire lock L
    if (b)
        foo = 30;
}                // release lock L
```

```
// Thread 3:
foo = 10;
synchronized(L) { // acquire lock L
    b = false;
}                // release lock L
```

1. List all data races that may occur in any execution of the program.

   Solution:

   There is exactly one race: Between the write in `foo = 30` and the write in `foo = 10`.

2. Consider an execution of the program that leads to a data race on `foo`. Show how the dynamic data race detector Eraser finds the data race. Consider the refined lockset algorithm of Eraser, which tracks the state of each shared memory location. Use the following table to provide your solution. You can assume that the execution stops when Eraser has detected the data race.

   Solution:

Instruction	After executing the instruction		
	locksHeld	Lockset of `foo`	State of `foo`
`boolean b = true;`	$\{\}$	$\{L\}$	Virgin
`int foo = 20;`	$\{\}$	$\{\}$	Exclusive
acquire L	$\{L\}$	$\{L\}$	Exclusive
`if (b)`	$\{L\}$	$\{L\}$	Exclusive
`foo = 30;`	$\{L\}$	$\{L\}$	Shared-modified
release L	$\{\}$	$\{L\}$	Shared-modified
`foo = 10;`	$\{\}$	$\{\}$	Shared-modified

   The last statement causes Eraser to report a data race because the lockset of `foo` is empty while the state of `foo` is "shared-modified".