Program Testing and Analysis

—Solutions of Mid-term Exam—

Department of Computer Science
TU Darmstadt

Winter semester 2015/16, November 30, 2015

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.)

   - ☐ An analysis that overapproximates a program's behavior to detect bugs cannot report false positives.
   - ☒ Executing a complex program with five different inputs underapproximates its behavior.
   - ☐ Testing is a way to overapproximate a program's behavior.
   - ☐ An analysis that underapproximates a program's behavior may consider infeasible paths.
   - ☐ Manual testing may overapproximate the behavior of a program.

2. Which of the following statements is true? (Only one statement is true.)

   - ☐ Functional testing is a form of white-box testing.
   - ☐ For programs with heap structures, exhaustive testing is feasible because objects can be abstracted into `null` and non-`null`.
   - ☐ Random testing cannot detect bugs.
   - ☐ The purpose of functional testing is to detect errors in the specification.
   - ☒ Testing can only show the presence of bugs, never their absence.

3. Which of the following statements is true? (Only one statement is true.)

   - ☐ The purpose of testing is to maximize coverage.
   - ☐ A test suite that achieves 20% branch coverage covers all paths.
   - ☐ Achieving >80% path coverage is a realistic goal for testing complex software.
   - ☒ Compared to path coverage, definition-use-pair coverage reduces the number of paths to test.
   - ☐ When computing definition-use pairs, a conditional check can never be consider a variable use.

4. Which of the following statements is true? (Only one statement is true.)

   - ☒ Test cases generated by the feedback-directed random test generator Randoop may use the class under test in a way not anticipated by the developer.
   - ☐ Randoop avoids creating redundant test cases by checking for exceptions thrown during test execution.
   - ☐ A test oracle decides how much coverage a test achieves.
   - ☐ The main idea of adaptive random testing is to avoid generating tests that throw exceptions.
   - ☐ The name "fuzz testing" is due to the fact that fuzz testing generates test cases where the decision whether the test exposes a bug is fuzzy.

Part 2 [3 points]

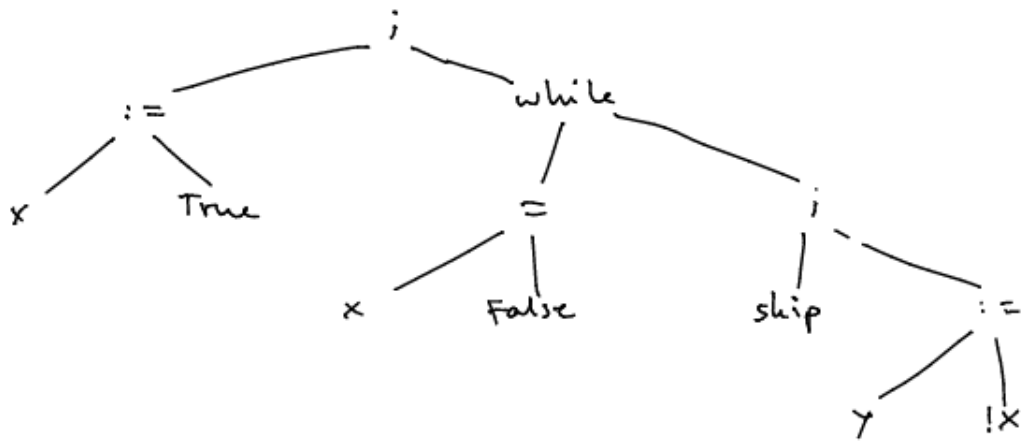
The following questions are about the syntax of SIMP. For your reference, here is the abstract grammar of SIMP, as discussed in the lecture.

$$
\begin{aligned}
P &::= C \mid E \mid B \\
C &::= l := E \mid C; C \mid \mathit{if}\ B\ \mathit{then}\ C\ \mathit{else}\ C \mid \mathit{while}\ B\ \mathit{do}\ C \mid \mathit{skip} \\
E &::= !l \mid n \mid E\ \mathit{op}\ E \\
\mathit{op} &::= + \mid - \mid * \mid / \\
B &::= \mathit{True} \mid \mathit{False} \mid E\ \mathit{bop}\ E \mid \neg B \mid B \wedge B \\
\mathit{bop} &::= < \mid > \mid =
\end{aligned}
$$

Consider the following SIMP program: `x:=True; while (x=False) do (skip; y:=!x)`

1. Draw the abstract syntax tree of the program.

Solution:



Part 3 [4 points]

Give a JavaScript function and a set of tests for the function that shows the following:

"At least n% branch coverage" does not imply "at least n% statement coverage".

Use the following template for your solution.

Function:

Solution:

```
function (a) {
  var x,y,z;
  if (a) {
    x=1;
    y=2;
    z=3;
    x=4;
  } else {
    x=5;
  }
}
```

Test inputs for the function:

Solution: a=false

The test inputs achieve 50% branch coverage and $3/7 = 43\%$ statement coverage.

Part 4 [10 points]

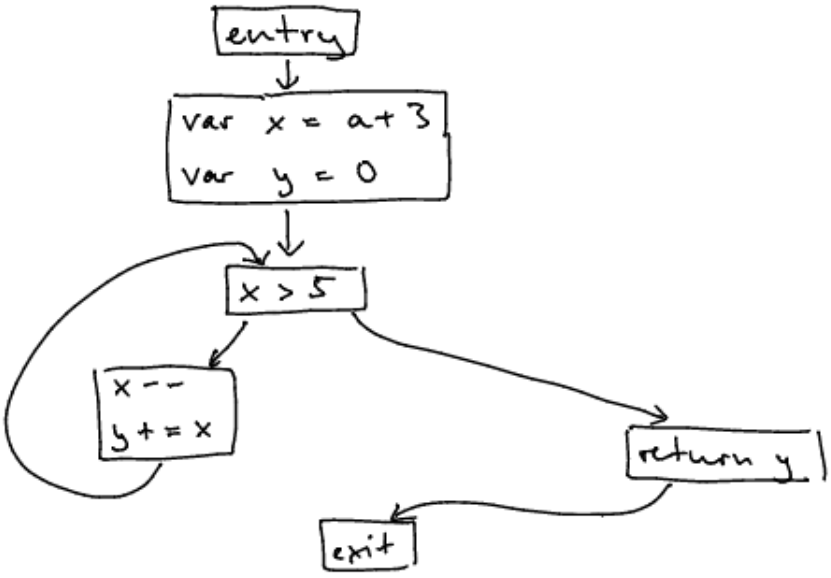
Consider the following JavaScript function.

```
1 function f(a) {
2   var x = a + 3;
3   var y = 0;
4   while (x > 5) {
5     x--;
6     y += x;
7   }
8   return y;
9 }
```

1. The return value of `f(3)` is 5.

2. Draw the control flow graph of the function.

   Solution:



3. Suppose you are asked to write tests for the function and that you should maximize path coverage.

   The total number of paths is infinite.

4. Consider the following test suite.

   - `f(0)`
   - `f(1)`
   - `f(2)`
   - `f(3)`

   The total number of paths covered by this test suite is 2.

5. Provide one additional test input that covers a path not covered by the above test inputs.

   - `f(4)`

Part 5 [17 points]

Consider the following SIMP program:

```
if (!x<3) then skip else (while !x>5 do x:=!x-5; y:=!x)
```

1. Give the semantics of the program as a sequence of transitions of the abstract machine for SIMP that was introduced in the lecture. For your reference, the following page gives the transition rules (copied from Fernandez' book).

   You only have to give the first eight transitions, as well as the final configuration of the abstract machine. Use the following template to present your solution. (We provide two lines for each configuration. The template starts with the initial configuration.)

   $\langle$`if (!x<3) then skip else (while !x>5 do x:=!x-5; y:=!x)` $\circ nil, nil, \{x \mapsto 6, y \mapsto 42\}\rangle$

   $\rightarrow \langle$`!x<3` $\circ$ `if` $\circ nil, skip \circ$ `while !x>5 do x:=!x-5; y:=!x` $\circ nil, \{x \mapsto 6, y \mapsto 42\}\rangle$

   $\rightarrow \langle$`!x` $\circ$ `3` $\circ$ `<` $\circ$ `if` $\circ nil, skip \circ$ `while !x>5 do x:=!x-5; y:=!x` $\circ nil, \{x \mapsto 6, y \mapsto 42\}\rangle$

   $\rightarrow \langle$`3` $\circ$ `<` $\circ$ `if` $\circ nil, 6 \circ skip \circ$ `while !x>5 do x:=!x-5; y:=!x` $\circ nil, \{x \mapsto 6, y \mapsto 42\}\rangle$

   $\rightarrow \langle$`<` $\circ$ `if` $\circ nil, 3 \circ 6 \circ skip \circ$ `while !x>5 do x:=!x-5; y:=!x` $\circ nil, \{x \mapsto 6, y \mapsto 42\}\rangle$

   $\rightarrow \langle$`if` $\circ nil, False \circ skip \circ$ `while !x>5 do x:=!x-5; y:=!x` $\circ nil, \{x \mapsto 6, y \mapsto 42\}\rangle$

   $\rightarrow \langle$`while !x>5 do x:=!x-5; y:=!x` $\circ nil, nil, \{x \mapsto 6, y \mapsto 42\}\rangle$

   $\rightarrow \langle$`!x>5` $\circ while \circ nil,$ `!x>5` $\circ$ `x:=!x-5; y:=!x` $\circ nil, \{x \mapsto 6, y \mapsto 42\}\rangle$

   $\rightarrow^* \langle nil, nil, \{x \mapsto 1, y \mapsto 1\}\rangle$

2. Does the program terminate successfully?
   - [x] Yes.
   - [ ] No.

Part 6 [3 points]

Suppose we are extending the SIMP language that was introduced in the lecture with a `do-while` command. The resulting language is called SIMP'. The syntax for SIMP' commands will be the following, where $< C >$ and $< B >$ are non-terminals:

$$C ::= l \coloneqq E \mid C; C \mid if\ B\ then\ C\ else\ C \mid while\ B\ do\ C \mid do\ C\ while\ B \mid skip$$

Informally, the semantics of `do C while B` is to execute the command `C` at least once, and to then repeatedly execute `C` as long as the boolean expression `B` evaluates to true. For example, the following SIMP' program executes the loop body two times:

```
x:=2; do x:=!x-1 while x>0
```

Extend the small-step operational semantics for SIMP that was given in the lecture by providing an additional rule or axiom that describes the semantics of the new command. Use the following template and fill in $GAP1$ and $GAP2$:

$$(\text{DO-WHILE})\frac{GAP1}{\langle do\ C\ while\ B, s\rangle \rightarrow \langle GAP2\rangle}$$

- $GAP1$ should be "nothing" (or "true").

- $GAP2$ should be $\langle C; if\ B\ then\ do\ C\ while\ B\ else\ skip, s\rangle$.

Part 7 [6 points]

The following questions are about the big-step operational semantics of SIMP. For your reference, we provide the semantics S as discussed in the lecture (copied from Fernandez' book):

$$\frac{}{\langle c, s\rangle \Downarrow \langle c, s\rangle \ \text{ if } c \in Z \cup \{True, False\}} \ (\text{const})$$

$$\frac{}{\langle !l, s\rangle \Downarrow \langle n, s\rangle \ \text{ if } s(l) = n} \ (\text{var})$$

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle \quad \langle B_2, s'\rangle \Downarrow \langle b_2, s''\rangle}{\langle B_1 \ \wedge \ B_2, s\rangle \Downarrow \langle b, s''\rangle \ \text{ if } b = b_1 \, and \, b_2} \ (\text{and})$$

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle}{\langle \neg B_1, s\rangle \Downarrow \langle b, s'\rangle \ \text{ if } b = not \ b_1} \ (\text{not})$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1 \, op \, E_2, s\rangle \Downarrow \langle n, s''\rangle \ \text{ if } n = n_1 \, op \, n_2} \ (\text{op})$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1 \, bop \, E_2, s\rangle \Downarrow \langle b, s''\rangle \ \text{ if } b = n_1 \, bop \, n_2} \ (\text{bop})$$

$$\frac{}{\langle skip, s\rangle \Downarrow \langle skip, s\rangle} \ (\text{skip}) \qquad \frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle l := E, s\rangle \Downarrow \langle skip, s'[l \mapsto n]\rangle} \ (:=)$$

$$\frac{\langle C_1, s\rangle \Downarrow \langle skip, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle skip, s''\rangle} \ (\text{seq})$$

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C_1, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle if \, B \, then \, C_1 \, else \, C_2, s\rangle \Downarrow \langle skip, s''\rangle} \ (\text{if}_\text{T})$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle if \, B \, then \, C_1 \, else \, C_2, s\rangle \Downarrow \langle skip, s''\rangle} \ (\text{if}_\text{F})$$

$$\frac{\langle B, s\rangle \Downarrow \langle True, s_1\rangle \ \langle C, s_1\rangle \Downarrow \langle skip, s_2\rangle \ \langle while \ B \ do \ C, s_2\rangle \Downarrow \langle skip, s_3\rangle}{\langle while \ B \ do \ C, s\rangle \Downarrow \langle skip, s_3\rangle} \ (\text{while}_\text{T})$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle}{\langle while \ B \ do \ C, s\rangle \Downarrow \langle skip, s'\rangle} \ (\text{while}_\text{F})$$

Suppose that we replace the rule (seq) defining sequential composition with the following rule (seq_{new}):

$$\frac{\langle C_1, s\rangle \Downarrow \langle skip, s'\rangle \qquad \langle C_2, s\rangle \Downarrow \langle skip, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle skip; s''\rangle}$$

We call the revised semantics, which uses the rule (seq_{new}), S_{new}.

1. Consider the SIMP program $x :=!y; z :=!x$ and the initial store $s = \{x \mapsto 0, y \mapsto 1\}$.

   - What is the store of the last configuration in the program's evaluation sequence under S_{new}? (Note: You do not have to write the sequence.)

     $s_{final} = \{x \mapsto 0, y \mapsto 1, z \mapsto 0\}$

   - What is the store of the last configuration in the program's evaluation sequence under the original semantics S? (Again, you do not have to write the sequence.)

     $s_{final} = \{x \mapsto 1, y \mapsto 1, z \mapsto 1\}$

2. Which of the following statements is true? (Only one statement is true.)

- ☐ Programs that terminate under S will also terminate under S_{new}.
- ☐ The rules (seq) and (seq_{new}) are equivalent.
- ☐ A semantics that contains both rules (seq) and (seq_{new}) is deterministic.
- ☒ S is closer to the JavaScript semantics than S_{new}.
- ☐ A semantics that contains both rules (seq) and (seq_{new}) is closer to the JavaScript semantics than S.

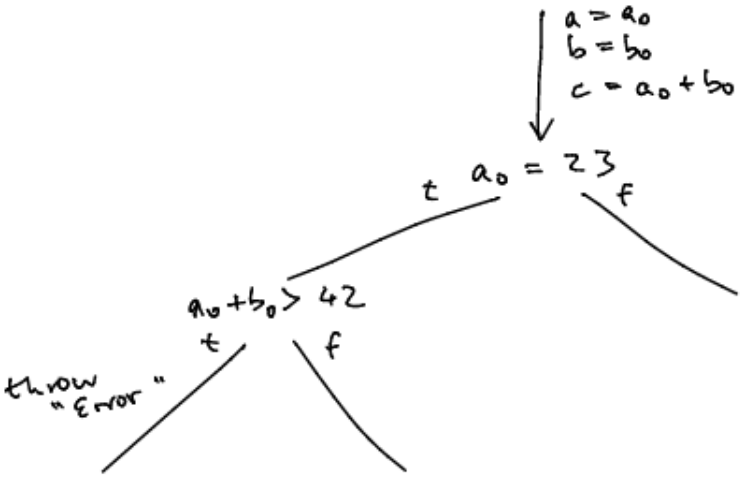
Part 8 [5 points]

Consider the following JavaScript function. Suppose we symbolically execute the function and consider `a` and `b` as symbolic inputs.

```
1 function f(a, b) {
2   var c = a + b;
3   if (a === 23) {
4     if (c > 42) {
5       throw "Error";
6     }
7   }
8 }
```

1. Draw the execution tree of the program.

   Solution:



2. What is the path condition for the path that reaches the error? (Write a quantifier-free formula.)

   Solution: $(a_0 = 23) \wedge (a_0 + b_0 > 42)$

3. Suppose that the path condition from the previous question is given to an SMT solver. Provide a concrete solution that the solver may yield.

   - $a_0 = 23$
   - $b_0 = 23$

Part 9 [8 points]

The following questions are about feedback-directed GUI testing, as described by Artzi et al. (ICSE'11). Consider a web application that consists of two pages:

- Page 1 contains two buttons, and the following event handlers are attached to them:

```
function button1Handler() {
  x = 5;
}

function button2Handler() {
  if (x > 3} {
    window.location = "page2.html"; // go to page 2
  }
}
```

  The initial value of variable `x=0`.

- Page 2 contains a single button, and the following event handler is attached to it:

```
function button3Handler() {
  aaaa
}
```

Suppose that the Artemis test generator has already executed the following sequence of input events:

- Load page 1, click button 2, click button 1

1. Suppose that Artemis extends the existing sequence with an additional event. What are the sequences that Artemis adds into the worklist? (Write one or more sequences of events.)

   Solution:

   - Load page 1, click button 2, click button 1, click button 1 [S1]
   - Load page 1, click button 2, click button 1, click button 2 [S2]

2. Which of these sequences could the default strategy of Artemis trigger next?

   Solution: Both, because the default strategy selects sequences randomly.

3. Instead of the default strategy, now consider the coverage-guided prioritization strategy of Artemis. Assume that the "load page 1" event has coverage 1.
   Which priorities does the strategy assign to the sequences from Question 1?

   Solution:

   - Coverage for handler1: $\frac{1}{1}$. Coverage for handler2: $\frac{2}{3}$. (Handler3 is not yet known.)
   - $cov(S1) = 1 - \frac{2}{3} \cdot \frac{1}{1} \cdot \frac{1}{1} = \frac{1}{3}$
   - $cov(S2) = 1 - \frac{2}{3} \cdot \frac{1}{1} \cdot \frac{2}{3} = \frac{5}{9}$

4. Suppose that Artemis decides to extend the initial sequence with "click button 2", and that it executes this extended event sequence, which will lead to page 2. Suppose that Artemis further extends the extended sequence. What are the possible next sequence(s) that Artemis adds into the worklist? (Write one or more sequences of events.)

   Solution:

   - Load page 1, click button 2, click button 1, click button 2, click button 3