Program Testing and Analysis

—Final Exam—

Department of Computer Science
TU Darmstadt

Winter semester 2016/17, March 21, 2017

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.)
   - ☐ For any given program, specification mining infers specifications that describe the behavior intended by the developer.
   - ☐ Bug detectors based on mined specifications cannot have false positives because the specifications describe the intended behavior of the program.
   - ☐ In a correct program, the pre-condition and the post-condition of a function are logically equivalent.
   - ☐ An invariant is a property that holds on some but not on all runs of a program.
   - ☒ A post-condition of a function is a program invariant.

2. Which of the following statements is true? (Only one statement is true.)
   - ☒ Information flow analysis tracks whether data from a source influences data at a sink.
   - ☐ Information flow analysis is equivalent to control flow analysis.
   - ☐ Information flow analysis may use declassification to increase the secrecy of a value.
   - ☐ Information flow analysis is equivalent to data flow analysis.
   - ☐ Information flow analysis tracks whether data from a sink influences data at a source.

3. Which of the following statements is true? (Only one statement is true.)
   - ☒ Symbolic execution systematically analyzes not yet analyzed paths of the program.
   - ☐ For any program with less than ten statements, symbolic execution covers all paths in a finite amount of time.
   - ☐ There is no program for which symbolic execution covers all paths.
   - ☐ Symbolic execution systematically searches for undefined behavior.
   - ☐ For any given program, symbolic execution covers all paths of the program.

4. Which of the following statements is true? (Only one statement is true.)
   - ☒ A speedup of 1.25x is the same as a performance improvement of 20%.
   - ☐ The speedup of one execution over another ranges between -100% and 100%.
   - ☐ The performance improvement of one execution over another is always a positive number.
   - ☐ A speedup of 1.25x is the same as a performance improvement of 23%.
   - ☐ A speedup of 1.25x is the same as a performance improvement of 25%.

Part 2 [14 points]

Consider the following SIMP program:

```
while !x-1 > 5 do (if !a = 5 then x := !x-3 else skip)
```

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 appendix provides the transition rules (copied from Fernandez' book).

   You only have to give the first six 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.)

   Solution:

   $\langle$`while !x-1 > 5 do (if !a = 5 then x := !x-3 else skip)` $\circ\ nil, nil, \{x \mapsto 7, a \mapsto 5\}\rangle$

   $\rightarrow \langle$`!x-1 > 5` $\circ$ `while` $\circ\ nil,$ `!x-1 > 5` $\circ$ `if !a = 5 then x := !x-3 else skip` $\circ\ nil, \{x \mapsto 7, a \mapsto 5\}\rangle$

   $\rightarrow \langle$`!x-1` $\circ$ `5` $\circ$ `>` $\circ$ `while` $\circ\ nil,$ `!x-1 > 5` $\circ$ `if !a = 5 then x := !x-3 else skip` $\circ\ nil, \{x \mapsto 7, a \mapsto 5\}\rangle$

   $\rightarrow \langle$`!x` $\circ$ `1` $\circ$ `-` $\circ$ `5` $\circ$ `>` $\circ$ `while` $\circ\ nil,$ `!x-1 > 5` $\circ$ `if !a = 5 then x := !x-3 else skip` $\circ\ nil, \{x \mapsto 7, a \mapsto 5\}\rangle$

   $\rightarrow \langle$`1` $\circ$ `-` $\circ$ `5` $\circ$ `>` $\circ$ `while` $\circ\ nil,$ `7` $\circ$ `!x-1 > 5` $\circ$ `if !a = 5 then x := !x-3 else skip` $\circ\ nil, \{x \mapsto 7, a \mapsto 5\}\rangle$

   $\rightarrow \langle$`-` $\circ$ `5` $\circ$ `>` $\circ$ `while` $\circ\ nil,$ `1` $\circ$ `7` $\circ$ `!x-1 > 5` $\circ$ `if !a = 5 then x := !x-3 else skip` $\circ\ nil, \{x \mapsto 7, a \mapsto 5\}\rangle$

   $\rightarrow \langle$`5` $\circ$ `>` $\circ$ `while` $\circ\ nil,$ `6` $\circ$ `!x-1 > 5` $\circ$ `if !a = 5 then x := !x-3 else skip` $\circ\ nil, \{x \mapsto 7, a \mapsto 5\}\rangle$

   $\rightarrow^* \langle nil, nil, \{x \mapsto 4, a \mapsto 5\}\rangle$

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

Part 3 [6 points]

Consider the axiom and rules that define the big step operational semantics of SIMP, as they have been presented in the lecture:

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

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

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

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

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

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

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

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

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

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

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

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

Suppose that the SIMP language is extended with an integer expression inspired by the conditional (ternary) operator of JavaScript, Java, C, etc. The abstract syntax of the new integer expression is $B \ ? \ E \ : \ E$, where B is a boolean expression and the Es each represent an integer expression. The semantics of the new expression is that it evaluates to the first integer expression if the boolean expression evaluates to true and to the second integer expression otherwise.

For example, based on the extended SIMP language, the following programs are valid SIMP:

```
// an expression that yields 42 because 1 is not equal to 2
1 = 2 ? 23 : 42

// an assignment that writes 5 into variable x
x := True ? 5 : 7
```

Extend the big step operational semantics to support the conditional operator. You should add axioms and rules to what is given above.

Solution:

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle E_1, s'\rangle \Downarrow \langle n_1, s''\rangle}{\langle B \ ? \ E_1 : E_2, s\rangle \Downarrow \langle n_1, s''\rangle} \ (?:)_\text{T}$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle B \ ? \ E_1 : E_2, s\rangle \Downarrow \langle n_2, s''\rangle} \ (?:)_\text{F}$$

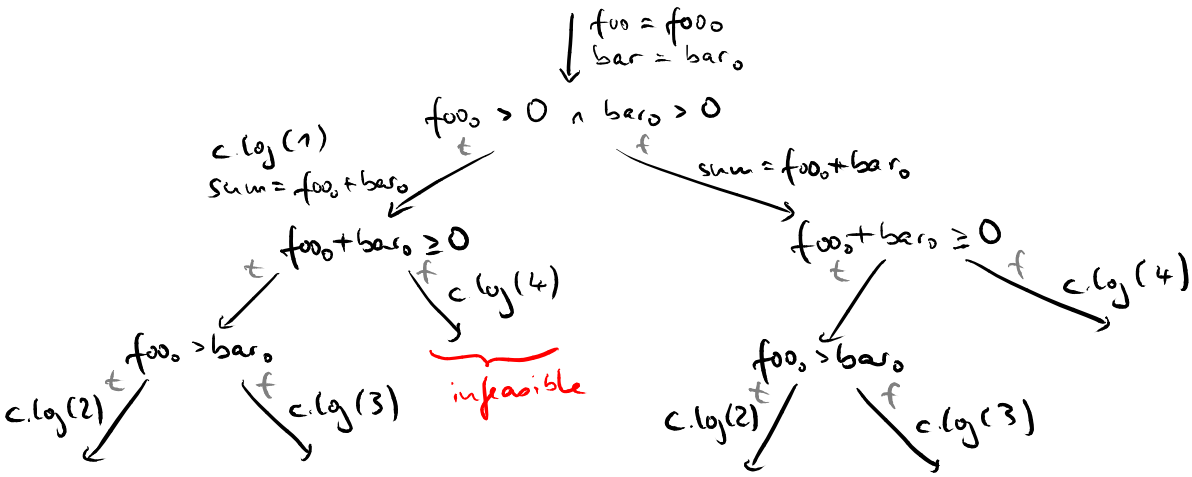
Part 4 [12 points]

Consider the following JavaScript function. Suppose that we symbolically execute the function while treating `foo` and `bar` as symbolic variables.

```
function symb(foo, bar) {
  if (foo > 0 && bar > 0) {
    console.log(1);
  }
  var sum = foo + bar;
  if (sum >= 0) {
    if (foo > bar) {
      console.log(2);
    } else {
      console.log(3);
    }
  } else {
    console.log(4);
  }
}
```

1. Draw the execution tree of the function.

   Solution:



2. The function has **6** paths, of which **5** are feasible.

3. What is the path condition of an execution that prints "13" to the console?

   Solution:
   $foo_0 > 0 \wedge bar_0 > 0 \wedge foo_0 + bar_0 \geqslant 0 \wedge foo_0 \leqslant bar_0$

4. Suppose this path condition is given to an SMT solver. Provide a concrete solution that the solver may yield.

   Solution:
   $foo_0 = 3 \wedge bar_0 = 3$

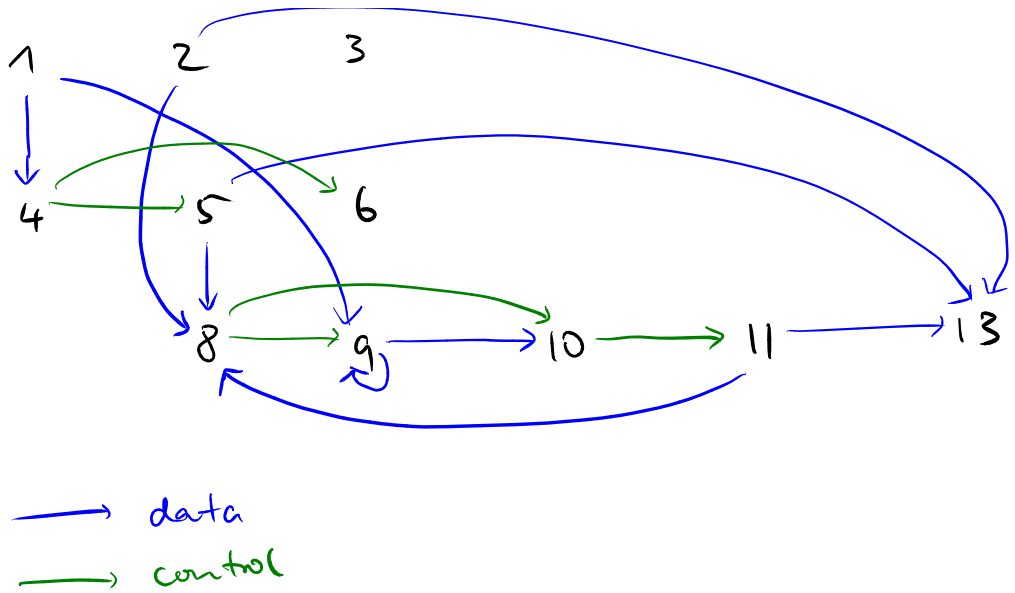
Part 5 [12 points]

Consider the following JavaScript code:

```
1  var x = ..
2  var y = ..
3  var z = ..
4  if (x == 3) {
5    y = true;
6    z = false;
7  }
8  while (y) {
9    x = x - 2;
10   if (x < 0)
11     y = false;
12 }
13 var res = y;
```

1. Provide the program dependence graph. You can use the line numbers to refer to statements. Use solid lines for data flow edges and dashed lines for control flow edges.

   Solution:



2. Suppose that x, y, and z are initialized as follows:

```
1 var x = 2;
2 var y = true;
3 var z = true;
```

   Give the execution history.

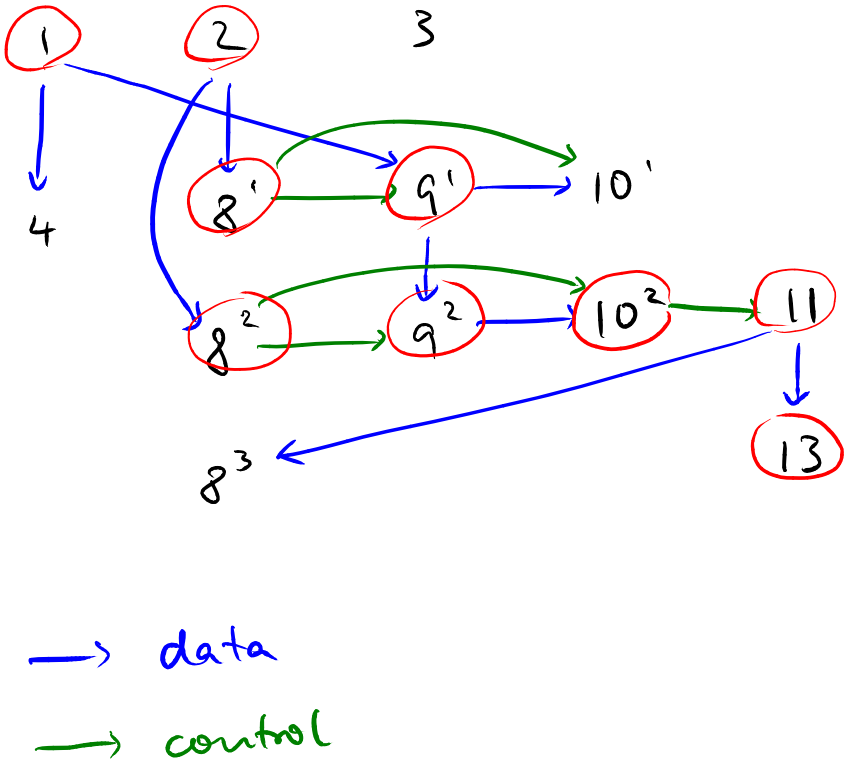
   Solution:

   1, 2, 3, 4, 8, 9, 10, 8, 9, 10, 11, 8, 13

3. Suppose we want to compute the dynamic backward slice with the last statement (`var res = y`) as the slicing criterion.

   (a) Provide the dynamic dependence graph, using the "revised approach" presented in the lecture.

   Solution:



   (b) Write the sliced program.

   Solution:

```
var x = ..
var y = ..
while (y) {
  x = x - 2;
  if (x < 0)
    y = false;
}
var res = y;
```

Part 6 [12 points]

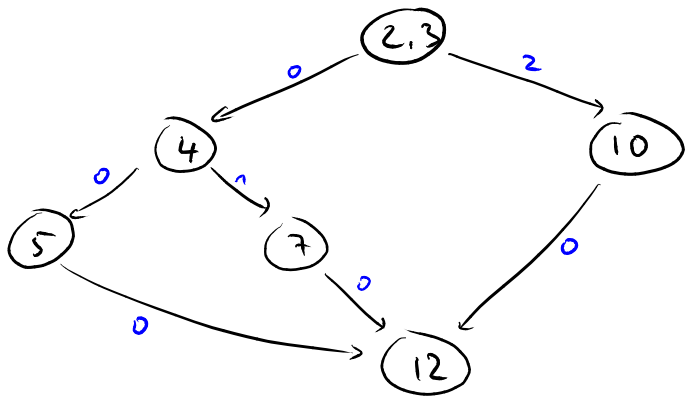
Consider the following JavaScript code:

```
1  function f(a) {
2    var b = a;
3    if (b > 23) {
4      if (b == 42) {
5        console.log("or that");
6      } else {
7        console.log("or that");
8      }
9    } else {
10     console.log("or something else");
11   }
12   return b;
13 }
```

The following applies Ball-Larus path profiling.

1. Give the control flow graph of the function. You can abbreviate statements with their line number.

   Solution:



2. Compute for each node in the graph the $NumPaths$, and assign to each edge in the graph an integer, so that the sum of integers along a path yields a unique number for each path. Follow the Ball-Larus algorithm presented in the lecture. Use the following table for $NumPaths$. Provide the integers of edges by adding them to the above control flow graph.

   Solution:

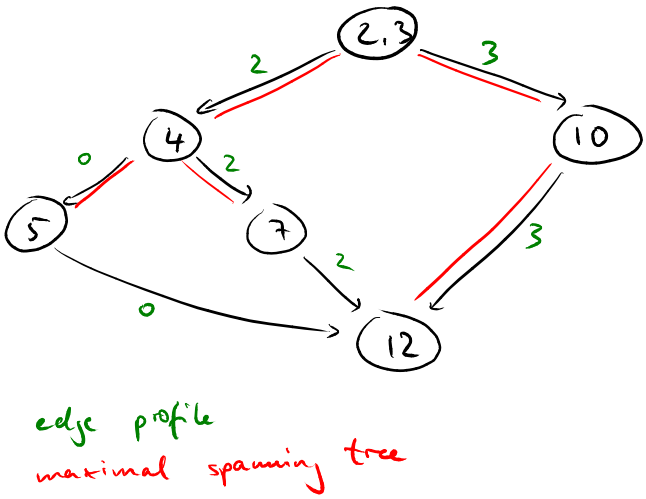
Node n	$NumPaths(n)$
2,3	3
4	2
5	1
7	1
10	1
12	1

3. The second step of the Ball-Larus algorithm assigns addition operations to edges of the control flow graph. To determine where to add addition operations, suppose the function is called as follows:

```
f(5);
f(5);
f(5);
f(25);
f(26);
```

Based on these calls, provide the edge profile by reproducing the control flow graph while indicating how often each edge has been executed:

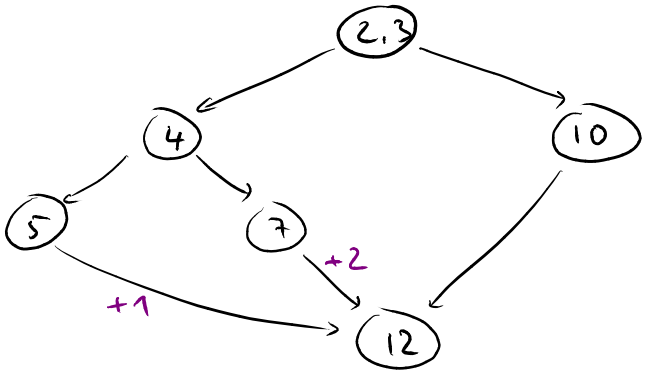
Solution:



4. In this graph, show a spanning tree that maximizes the overall cost of edges that are part of the spanning tree. Based on the spanning tree, indicate the edges at which to perform addition operations. The overall goal is to minimize the number of addition operations for the given edge profile.

   (a) Reproduce the graph and indicate where to perform the addition operations:

   Solution:



   (b) Based on the above solution, provide the path encodings in the following table:

Path	Encoding
2,3 → 10 → 12	0
2,3 → 4 → 5 → 12	1
2,3 → 4 → 7 → 12	2