Program Testing and Analysis

—Solutions of Final Exam—

Department of Computer Science
TU Darmstadt

Winter semester 2015/16, March 15, 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.)
   (a) ☒ Artemis uses feedback from executed event handlers to decide which events to trigger next.
   (b) ☐ Automated GUI testing has fully explored the behavior of an application when it has triggered each event (e.g., click on a particular button) once.
   (c) ☐ Automated GUI testing does not require any test oracle because the application is used through its user interface.
   (d) ☐ EventBreak guarantees to find any responsive problem that is caused by two related events.
   (e) ☐ SwiftHand explores an application based on a user-provided finite state model of the application.

2. Which of the following statements is true? (Only one statement is true.)
   (a) ☒ The path conditions summarize the branch decisions taken when executing a particular path through a program.
   (b) ☐ The execution tree of a program with a finite control flow graph is finite.
   (c) ☐ Symbolic execution scales to more complex programs than concolic execution because it does not need to execute the program.
   (d) ☐ Symbolic execution ignores all interactions of a program with its environment because they are irrelevant for the program's behavior.
   (e) ☐ Concolic execution can perfectly predict the path that is going to be executed with a particular input because it knows all runtime values.

3. Which of the following statements is true? (Only one statement is true.)
   (a) ☐ Full branch coverage implies full path coverage.
   (b) ☐ Full loop coverage implies full path coverage.
   (c) ☒ Full path coverage implies full branch coverage.
   (d) ☐ Full branch coverage implies full loop coverage.
   (e) ☐ Full loop coverage implies full branch coverage.

4. Which of the following statements is true? (Only one statement is true.)
   (a) ☐ A data race occurs when two concurrent threads attempt to read the same value.
   (b) ☐ A program that uses thread-safe classes is free of concurrency bugs.
   (c) ☐ Eraser's lockset algorithm analyzes the order in which memory accesses occur and detects unordered accesses.
   (d) ☐ Concurrency bugs are easy to debug because they occur deterministically.
   (e) ☒ CHESS systematically explores the interleavings of a concurrent program.

Part 2 [8 points]

Consider a multi-threaded program with an initial state where `flag1` and `flag2` are 0, and where the following two pieces of code execute in concurrent threads:

```
1  // Thread 1
2  flag1 = 1;
3  if (flag2 === 0) {
4    console.log("T1 is here");
5  }
```

```
1  // Thread 2
2  flag2 = 1;
3  if (flag1 === 0) {
4    console.log("T2 is here");
5  }
```

(Please ignore the fact that JavaScript does not support this kind of concurrency. The example uses JavaScript for your convenience.)

- Assume that the program executes under sequential consistency and that calls to `console.log` are atomic. What are the possible outputs of the program?

  Solution:

  There are three possible outputs:

  - "T1 is here"
  - "T2 is here"
  - No output at all

  I.e., it is impossible that both threads write.

- Provide a schedule where the program prints `T2 is here`. Use the line numbers and the thread numbers to describe in which order the statements execute.

  Solution:

  - thread 2, line 2
  - thread 2, line 3
  - thread 2, line 4
  - thread 1, line 2
  - thread 1, line 3

Part 3 [10 points]

Consider the following program and suppose it gets analyzed with the Daikon invariant detector:

```
1  function foo(x, y) {
2    var k = x, z = 0;
3    if (x !== y) {
4      k = x;
5      while (k > 0) {
6        // LOOP_INVAR
7        z += y;
8        k--;
9      }
10   }
11   return z;
12 }
13
14 var a = 3, b = 3, c = 5;
15 foo(a, b);
16 foo(a, c);
17 foo(b, c);
18 foo(c, b);
```

- Provide two invariants that Daikon would infer at the entry of function `foo`.

  Solution:
  $3 \leqslant x \leqslant 5$ and $3 \leqslant y \leqslant 5$

- Provide two invariants that Daikon would infer at location `LOOP_INVAR`.

  Solution:
  $k > 0$ and $3 \leqslant x \leqslant 5$

Suppose that Daikon has inferred the following invariants at the exit of function `foo`:

(1) `k >= 0`
(2) `z >= 0`

- Which of these invariants may not hold in executions where function `foo` is called with arguments different from the above calls?

  Solution:
  Both invariants may not hold.

- Provide arguments for `foo` that violate at least one of these invariants, and explain why the invariant is violated.

  Solution:

  - `foo(-2,-2)` violates the $k \leqslant 0$ invariant because `k` is set to -2 at line 2 and not modified afterwards.
  - `foo(1,-3)` violates the $z \leqslant 0$ invariant because line 7 adds -3 to `z`, so `z` is -3 at the function exit.

Part 4 [12 points]

Consider the following JavaScript program.

```
1  var a = readInput();
2  var b = readInput();
3  var x = 5;
4  var y = -5;
5  while (a) {
6    if (b) {
7      y++;
8    }
9    x = a;
10   a--;
11 }
12 var sliceHere = x;
```

Compute the static backward slice for variable `sliceHere` at line 12. Use the slicing approach of Weiser (IEEE TSE, 1984) and its formulation as a graph reachability problem, as it has been introduced in the lecture. To describe your solution, follow the steps outlined below.

1. Provide the data flow dependences between statements in the program. Use the following table to summarize the dependences. Each table cell represents a pair of statements. Mark all pairs of statements that have a definition-use relationship.

   Solution:

Def	Use									
	1	2	3	4	5	6	7	9	10	12
1					x			x	x	
2						x				
3										x
4							x			
5										
6										
7							x			
9										x
10					x			x	x	
12										

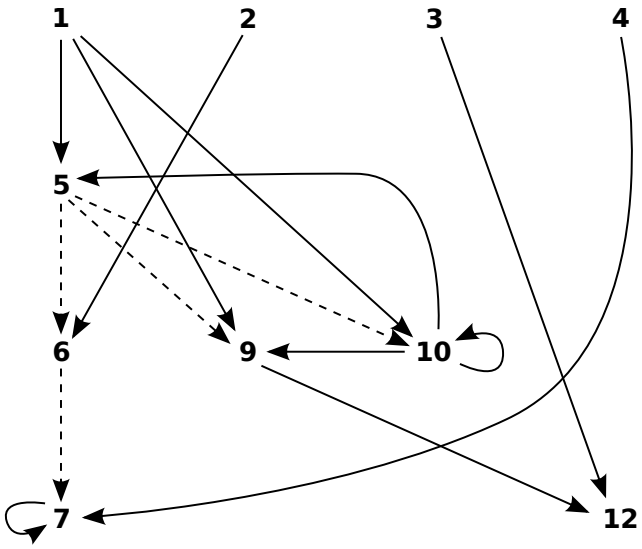
2. Provide the control flow dependences between statements in the program. Describe your solutions as a sequence of "Statement .. is control-flow dependent on statement .." sentences.

   Solution:

   - Statement 6 is control flow-dependent on statement 5.
   - Statement 7 is control flow-dependent on statement 6.
   - Statement 9 is control flow-dependent on statement 5.
   - Statement 10 is control flow-dependent on statement 5.

3. Summarize the data flow dependences and the control flow dependences into a program dependence graph. Use the following template to draw your solution.

Solution:



⟶ .. destination is data-dependent on source

- - - -► .. destination is control-dependent on source

4. What is the slice for the slicing criterion (variable `sliceHere` at line 12)? Write down the source code of the sliced program as a syntactically correct program.

Solution:

```
1  var a = readInput();
2  var x = 5;
3  while (a) {
4    x = a;
5    a--;
6  }
7  var sliceHere = x;
```

Part 5 [12 points]

Consider the following JavaScript program, which is annotated with an information flow policy. The policy marks `pwd` as "top secret" and `user` as "confidential". All other values are considered to be "public" by default. Furthermore, the policy states that `console.log` is an untrusted sink, i.e., only public information is allowed to be passed into `console.log`.

```
1  var pwd = "a";      // top secret
2  var user = "Joe";   // confidential
3  var pwdLength = pwd.length;
4  var count = 0;
5  while (pwdLength > 0) {
6    pwdLength--;
7    count++;
8  }
9  var s = user+ "'s password";
10 s += " has "+count+" character(s)";
11 console.log(s);     // untrusted sink
```

The security labels in this example are elements of a three-element lattice where

- the set of security classes is $\{"top\ secret", "confidential", "public"\}$,
- the partial order is defined by $\{"top\ secret" \rightarrow "confidential", "confidential" \rightarrow "public"\}$,
- the lower bound is $"public"$, and
- the upper bound is $"top\ secret"$.

Questions:

- Define the greatest lower bound operator for the above lattice by giving the results of the operator for each pair of security classes

  Solution:
  $glb(top\ secret, top\ secret) = top\ secret$
  $glb(top\ secret, confidential) = glb(confidential, top\ secret) = confidential$
  $glb(top\ secret, public) = glb(public, top\ secret) = public$
  $glb(confidential, confidential) = confidential$
  $glb(confidential, public) = glb(public, confidential) = public$
  $glb(public, public) = public$

- Is the lattice a universally bounded lattice?

  Solution:
  Yes, it is a universally bounded lattice.

- Suppose a dynamic information flow analysis that tracks both explicit and implicit flows. What are the security labels of variables and expressions, and what is the security stack at different points during the execution? Use the following template to provide your answer (you need to fill in the last two columns).

  Solution:

Line	Variable or expression	Security label of variable or expression (after executing the line)	Security stack (after executing the line)
1	`pwd`	top secret	—
2	`user`	confidential	—
3	`pwdLength`	top secret	—
4	`count`	public	—
5	`pwdLength > 0`	top secret	top secret
6	`pwdLength`	top secret	top secret
7	`count`	top secret	top secret
5	`pwdLength > 0`	top secret	—
9	`s`	confidential	—
10	`s`	top secret	—
11	`s`	top secret	—

- Does the execution violate the information flow policy? Why (not)?

  Solution:
  Yes, because the password and the user name, which are marked as top secret and confidential, flow into `console.log`, which is an untrusted sink.

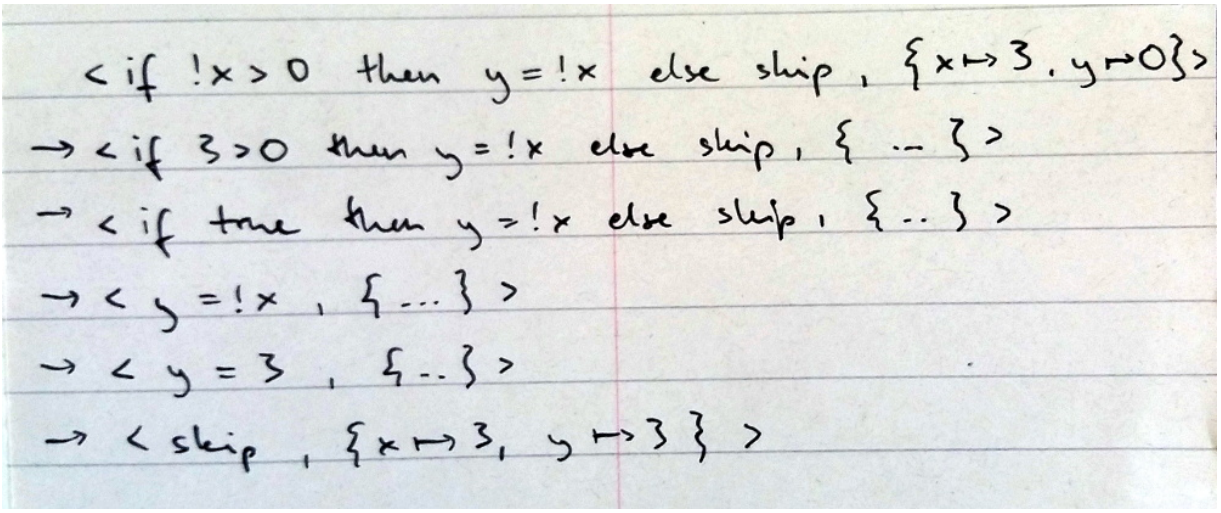
Part 6 [12 points]

Suppose the following SIMP program and an initial store $\{x \mapsto 3, y \mapsto 0\}$:

```
if !x > 0 then y = !x else skip
```

- Provide the evaluation sequence of the program using the small-step operational semantics of SIMP. For your reference, the following page provides the axioms and rules that have been introduced in the lecture (copied from Fernandez' book).

  Solution:

- Is the program divergent?

  Solution: No

- Is the program blocked?

  Solution: No

- Is the program terminating?

  Solution: Yes

Reduction Semantics of Expressions:

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

$$\frac{}{\langle n_1\ op\ n_2, s\rangle \to \langle n, s\rangle \ \text{ if } n = (n_1\ op\ n_2)}\ (\text{op})$$

$$\frac{}{\langle n_1\ bop\ n_2, s\rangle \to \langle b, s\rangle \ \text{ if } b = (n_1\ bop\ n_2)}\ (\text{bop})$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 op E_2, s\rangle \to \langle E_1' op E_2, s'\rangle}\ (\text{op}_\text{L}) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle n_1 op E_2, s\rangle \to \langle n_1 op E_2', s'\rangle}\ (\text{op}_\text{R})$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 bop E_2, s\rangle \to \langle E_1' bop E_2, s'\rangle}\ (\text{bop}_\text{L}) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle n_1 bop E_2, s\rangle \to \langle n_1 bop E_2', s'\rangle}\ (\text{bop}_\text{R})$$

$$\frac{}{\langle b_1 \wedge b_2, s\rangle \to \langle b, s\rangle \ \text{ if } b = (b_1\ and\ b_2)}\ (\text{and})$$

$$\frac{}{\langle \neg b, s\rangle \to \langle b', s\rangle \ \text{ if } b' = not\ b}\ (\text{not}) \qquad \frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle \neg B_1, s\rangle \to \langle \neg B_1', s'\rangle}\ (\text{notArg})$$

$$\frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \to \langle B_1' \wedge B_2, s'\rangle}\ (\text{and}_\text{L}) \qquad \frac{\langle B_2, s\rangle \to \langle B_2', s'\rangle}{\langle b_1 \wedge B_2, s\rangle \to \langle b_1 \wedge B_2', s'\rangle}\ (\text{and}_\text{R})$$

Reduction Semantics of Commands:

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

$$\frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to \langle C_1'; C_2, s'\rangle}\ (\text{seq}) \qquad \frac{}{\langle skip; C, s\rangle \to \langle C, s\rangle}\ (\text{skip})$$

$$\frac{\langle B, s\rangle \to \langle B', s'\rangle}{\langle if\, B\ then\ C_1\ else\ C_2, s\rangle \to \langle if\, B'\ then\ C_1\ else\ C_2, s'\rangle}\ (\text{if})$$

$$\frac{}{\langle if True\ then\ C_1\ else\ C_2, s\rangle \to \langle C_1, s\rangle}\ (\text{if}_\text{T})$$

$$\frac{}{\langle if False\ then\ C_1\ else\ C_2, s\rangle \to \langle C_2, s\rangle}\ (\text{if}_\text{F})$$

$$\frac{}{\langle while\ B\ do\ C, s\rangle \to \langle if\, B\ then\ (C; while\ B\ do\ C)\ else\ skip, s\rangle}\ (\text{while})$$