Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Pooja Rani

Assignment 10 — 18.11.2020 – v1.2
Dynamic Program Analysis

Please submit this exercise by email to pascal.gadient@inf.unibe.ch before 25 November 2020, 10:15am.

You must submit your code as editable text, i.e., use plain text file(s).

Exercise 1: Theory (1 pt)

a) Suppose you have an application and you want to test all code paths with dynamic analysis. Is this approach reasonable? Justify!

b) Is monkey testing a dynamic analysis technique? Justify!

Exercise 2: Contracts (3 pts)

In this exercise, we use the syntax and features from *jContracts* available here. Consider the Java code below.

```
/**
 * @pre number >= 100.0
 * @post return == true
 * @inv number < 0 || number >= 0
 */
public boolean add(int number) {
  if (number < 10) {
    this.number = this.number + number;
    return true;
  } else {
    return false;
  }
}
```

a) Are the contracts for `add(int number)` valid, *i.e.,* exists a configuration that passes all three checks? If yes, which configuration succeeds? If no, what is the problem?

b) Can the invariant contract be removed without any side effects? If yes, why do no side-effects occur? If no, which side-effects would occur?

c) Suppose you have a Java method that calculates from a given number the number multiplied by itself as shown in the code below. Design a post contract that validates the result, *i.e.,* the contract must be violated whenever a result is wrong.

```
public float square(int number) {
  return number * number;
}
```

Exercise 3: Profiling (3 pts)

In this exercise, we use the Python programming language and a memory profiler package. If you do not already have Python installed please install it from here. Next, you have to install the memory profiler package with the command `pip install -U memory_profiler`. Please ensure that you execute that command with superuser access rights (use sudo on Linux and macOS, or right-click the shell executable and select "Run as administrator" on Windows). Next, please create a text file with the name `allocateMemory.py` that contains the Python code below.

```
01    @profile
02    def getAllocatedMemory():
03      a = [1] * (10 ** 6)
04      b = [2] * (2 * 10 ** 7)
05      del b
06      return a
07
08    if __name__ == '__main__':
09      getAllocatedMemory()
```

Finally, please execute `python -m memory_profiler allocateMemory.py` and inspect the console output.

a) What is the return value of the method?

b) How much memory is consumed at most during the execution?

c) In which line is the most memory consumed during the execution of this method?

d) What does the command `del b` do?

e) Is it mandatory to use the `del` command to free memory in Python?

f) How could you improve the memory consumption without changing the return value of the method?

Exercise 4: Code coverage (3 pts)

In this exercise, we will have a look at the output of *gcov*, a dynamic analysis code coverage reporting tool that comes with *GCC*. To obtain code coverage data from an application with *gcov*, you need to (re)compile its source code with a compiler from *GCC*, and you have to set additional parameters so that the compiler injects on the fly code for collecting statistics during run time. In the next step, the newly generated binary must be executed. During the execution, the application itself logs statistics and stores them in a file. Finally, *gcov* opens these generated files and presents them in a nice textual representation. Because the set up of *GCC* can be cumbersome (especially on Windows machines) we already performed the outlined steps for you.

The generated output of *gcov* shown below was generated for a simple Fibonacci application. When executed, the Fibonacci application asked the user to enter the number of iterations to perform.

You can find more details regarding the output format of *gcov* in the corresponding man pages which you can find here.

```
    -:    0:   Source:fibonacci.c
    -:    0:   Graph:fibonacci.gcno
    -:    0:   Data:fibonacci.gcda
    -:    0:   Runs:1
    -:    1:   #include <stdio.h>
    1:    2:   int main() {
    1:    3:     int i, n, t1 = 0, t2 = 1, nextTerm;
    1:    4:     printf("Enter the number of iterations to perform:  ");
    1:    5:     scanf("%d", &n);
    -:    6:
    1:    7:     if (n > 50) {
#####:    8:       printf("This will take some time, please be patient.\n");
    -:    9:     }
    -:   10:
    1:   11:     printf("Fibonacci series:  ");
   13:   12:     for (i = 1; i <= n; ++i) {
   12:   13:       printf("%d, ", t1);
   12:   14:       nextTerm = t1 + t2;
   12:   15:       t1 = t2;
   12:   16:       t2 = nextTerm;
    -:   17:     }
    -:   18:
    1:   19:     return 0;
    -:   20:   }
```

a) How many times has the application been executed in order to collect code coverage data?

b) How many lines have been executed?

c) How many lines have not been executed?

d) Which statement was executed more than any other statement?

e) Which fibonacci number has been provided by the user?

f) What is a fundamental limitation of dynamic analyses? In other words, what is the problem with user input?

Exercise 5: Invariant detection (3 pts BONUS)

In this exercise, we will work with an analysis output of Daikon. Daikon is open-source software and its documentation can be found here. Inspect the code below and answer the corresponding questions.

```
01   /*@ modifies this.theArray[*], this.topOfStack, this.theArray[this.topOfStack], this.theArray[this.topOfStack-1]; */
02   /*@ ensures (\result != null)  ==  (\old(this.topOfStack) >= 0); */
03   /*@ ensures (\result != null)  ==>  (\old(this.theArray[this.topOfStack]) != null); */
04   /*@ ensures (\result != null)  ==>  (!(\forall int i; (0 <= i && i <= \old(this.theArray.length-1)) ==> (\old(\typeof(this.theArray[i])) != \typeof(\result)))); */
05   /*@ ensures (\result != null)  ==>  (\typeof(this.theArray) != \typeof(\result)); */
06   /*@ ensures (\result != null)  ==>  (this.theArray[\old(this.topOfStack)] == null); */
07   /*@ ensures (\result != null)  ==>  (this.topOfStack - \old(this.topOfStack) + 1 == 0); */
08   /*@ ensures (\result != null)  ==>  (this.topOfStack < this.theArray.length-1); */
09   /*@ ensures (\result == null)  ==  (\old(this.topOfStack) == -1); */
10   /*@ ensures (\result == null)  ==  (this.topOfStack == \old(this.topOfStack)); */
11   /*@ ensures (\result == null)  ==>  ((\forall int i; (0 <= i && i <= \old(this.theArray.length-1)) ==> (\old(this.theArray[i]) == null))); */
12   /*@ ensures (\result == null)  ==>  ((\forall int i; (0 <= i && i <= \old(this.theArray.length-1)) ==> (\old(\typeof(this.theArray[i])) == \typeof(null)))); */
13   /*@ ensures (\result == null)  ==>  ((\forall int i; (0 <= i && i <= this.theArray.length-1) ==> (this.theArray[i] == null))); */
14   /*@ ensures (\result == null)  ==>  ((\forall int i; (0 <= i && i <= this.theArray.length-1) ==> (\typeof(this.theArray[i]) == \typeof(null)))); */
15   /*@ ensures (\result == null)  ==>  (this.topOfStack == -1); */
16   /*@ ensures \typeof(this.theArray) != \typeof(\result); */
17   /*@ ensures this.topOfStack <= \old(this.topOfStack); */
18   /*@ ensures !(\forall int i; (0 <= i && i <= \old(this.theArray.length-1)) ==> (\old(\typeof(this.theArray[i])) != \typeof(\result))); */
19   /*@ ensures \old(this.topOfStack) <= this.theArray.length-1; */
20   /**
21    * Return and remove most recently inserted item from the stack.
22    * @return most recently inserted item, or null, if stack is empty.
23    */
24   public Object topAndPop( )
25   {
26       if( isEmpty( ) )
27           return null;
28       Object topItem = top( );
29       theArray[ topOfStack-- ] = null;
30       return topItem;
31   }
```

a) To which line in the Java code corresponds the postcondition in line 06?

b) Why does the postcondition in line 02 use greater than or equal (>=), but not strict inequality (>)?

c) What are the possible values for the variable `this.topOfStack` according to the relevant post-conditions of `topAndPop()`? Please report for each relevant postcondition its line number and the value-related constraints.