Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Prof. Elisabetta Di Nitto and Matteo Rossi

20133 Milano (Italia)
Piazza Leonardo da Vinci, 32
Tel. (39) 02-2399.3400
Fax (39) 02-2399.3411

Software Engineering II

June 17th, 2020

Last Name
First Name
Id number (Matricola)

Notes

This exam is handled online. Rules

1. Only use a computer, NOT a tablet, NOR a smartphone
2. Activate the feed of your webcam
3. Share the screen of your computer
4. Keep the microphone on
5. No dual screens
6. No virtual machines
7. When you upload a file through the form, make sure to include your "person id" (the 8-digit number that starts with "10") in the name of the file, AT THE BEGINNING OF THE NAME (so the name of the file should be, say, "10143828_etc.", if your person id is "10143828").
8. The exam is open book, so you can check the course materials (notes, slides, books, past exams, etc.), which can be in paper form, or in electronic form. If the materials are in electronic form, you MUST use the same computer on which you are taking the exam to display them.
9. You cannot interact with other people during the exam.
10. The exam is composed of three exercises. Read carefully all points in the text!
11. **Total available time: 1h and 30 mins**

Scores of each question:

Question 1 (MAX 7) _____

Question 2 (MAX 6) _____

Question 3 (MAX 3) _____

Question 1 Alloy (7 points)
We want to develop an integrated system that supports the execution of online exams. Each online exam is handled by one or more instructors, it can be taken by some students, it is associated with a virtual room and with the exam text, plus the answers by the students. The following rules hold:

1) Only students who registered for the exam can enter the virtual room.
2) While the exam is running, students in the room must have their camera, mic and screen sharing on, otherwise the system assumes they have exited the room.
3) If the exam is running or it has ended, students that result to have exited the room cannot enter again.

Consider the following Alloy signatures that provide a partial model for the problem at hand:

```
sig Person {}
sig Instructor extends Person {}
sig Exam {
   instructors: some Instructor,
   students: set Student,
   vr: VirtualRoom,
   test: ExamTest,
   examStatus: EStatus
}

sig Description {}

sig ExamTest {
   description: Description,
   answers: set Student
}

sig EStatus {}
one sig Planned extends EStatus{}
one sig Running extends EStatus{}
one sig Ended extends EStatus{}
```

Point A (2 points). Define the concept of *Student*, who must own some devices (camera, mic and screen sharing). Add all signatures/constraints/facts needed to make the *Student* concept self-contained and coherent with the description above.

Point B (1 point). Define the concept of *VirtualRoom* that separately captures the students who entered the room at some point, those that are currently in the room, and those that exited the room, in addition to the instructors supervising the room. Add all signatures/constraints/facts you need.

Point C (1 point). Define a fact modelling constraint 1) "Only students who registered for the exam can enter the virtual room".

Point D (1 point). Define a fact modelling constraint 2) "While the exam is running, students in the room must have their camera, mic and screen sharing on, otherwise the system assumes they have exited the room".

Point E (2 points). Define a predicate modelling operation *letIn* that is invoked to let students into the room, and which enforces constraint 3) "If the exam is running or it has ended, students that result to have exited the room cannot enter again. If the exam is planned, they can get in even if they have been out".

Solution

Point A

```
sig Student extends Person {
   devices: some Device
}{#(devices & Camera) = 1 and #(devices & Mic) = 1 and #(devices & Screen) = 1 }

sig Status {}
one sig On extends Status {}
one sig Off extends Status {}

sig Device {
   status: Status
}

sig Camera extends Device {}
sig Mic extends Device {}
sig Screen extends Device {}
```

Point B

```
sig VirtualRoom {
   studentsEnteredRoom: set Student,
   studentsInRoom: set Student,
   instructorsInRoom: some Instructor,
   studentsExitedRoom: set Student
}{
studentsInRoom & studentsExitedRoom = none
studentsEnteredRoom = studentsInRoom + studentsExitedRoom
}
```

Point C

```
fact onlyRegisteredStudentsInVR {
   all s: Student | all e: Exam | s in e.vr.studentsEnteredRoom implies
                                  s in e.students
}
```

Point D

```
fact studentsMustHaveDevicesOn {
 all e: Exam, s: Student |
        e.examStatus in Running and s in e.vr.studentsInRoom
        implies
        ( all d: Device | d in s.devices implies d.status in On )
}
```

Point E

```
pred letIn(e, e': Exam, s: Student) {
   //precondition
   s in e.students and
   ( not (e.examStatus = Planned) implies (not s in e.vr.studentsEnteredRoom) )


   //postcondition
```

```
    e'.instructors = e.instructors
    e'.students = e.students
    e'.vr.studentsInRoom =  e.vr.studentsInRoom + s
    e'.vr.instructorsInRoom =  e.vr.instructorsInRoom
    e'.vr.studentsExitRoom = e.vr.studentsExitRoom - s
    e'.test = e.test
    e'.examStatus = e.examStatus
}
```

Question 2 Testing (red) (6 points)

Consider the following fragment of C code.

```
0   mycode(int S[], int Slength)
1     int h, x, p, res = 0;
2     if (S == null)
3         return 23;
4     x = Slength;
5     if (x == 0)
6         return -5;
7     h = 0;
8     while (h < Slength) {
9         p = S[h] - S[x-(h+1)];
10        if (p != 0)
11            return 14;
12        h  = h+1;
13    }
14    return -5
```

a. Identify the def-use pairs for program `mycode` and say whether they highlight anything unusual.

b. Consider the following paths for program `mycode`.

1. 0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 13, 8, 14

2. 0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 13, 8, 9, 10, 12, 13, 8, 14

For each path: symbolically execute the program covering it, highlight the path condition associated with it, and define a test case that realises the path.

Solution

a.
S: (0, 2), (0, 9)
Slength: (0, 4), (0, 8)
h: (7, 8), (7, 9), (7, 12), (12, 8), (12, 9), (12, 12)
x: (4, 5), (4, 9)
p: (9, 10)

res: (1, ?)

Def-use analysis shows that variable res is defined, but never used.

b.1
0. Slength = SL
2. S ≠ null
4. x = SL
5. x ≠ 0
7. h = 0
8. h < SL
9. p = S[0] – S[x-1]
10. p = 0
12. h = 1
8. h ≥ SL

The path condition is SL = 1 (notice that x = 1, and S[0] = S[1-1]).
Test case: S = [4], Slength = 1

b.2
0. Slength = SL
2. S ≠ null
4. x = SL
5. x ≠ 0
7. h = 0
8. h < SL
9. p = S[0] – S[x-1]
10. p = 0
12. h = 1
8. h < SL
9. p = S[1] – S[x-2]
10. p = 0
12. h = 2
8. h ≥ SL

The path condition is SL = 2, S[0] = S[1] (notice that x = SL = 2, so 0 = S[0] – S[x-1] = S[1] – S[x-2])
Test case: S[-2, -2], Slength = 2

Question 2 Testing (green) (6 points)

Consider the following fragment of C code.

```
0   thiscode(int T[], int Tlength)
1     int k, v, r, ret = 0;
2     if (T == null)
3         return -13;
4     v = Tlength;
5     if (v == 0)
6         return -2;
7     k = 0;
8     while (k < Tlength) {
```

```
9         r = T[k] - T[v-(k+1)];
10        if (r != 0)
11            return 8;
12        k  = k+1;
13    }
14    return -2
```

a. Identify the def-use pairs for program `thiscode` and say whether they highlight anything unusual.

b. Consider the following paths for program `thiscode`.

1. 0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 13, 8, 14

2. 0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 13, 8, 9, 10, 12, 13, 8, 14

For each path: symbolically execute the program covering it, highlight the path condition associated with it, and define a test case that realises the path.

Solution

a.
T: (0, 2), (0, 9)
Tlength: (0, 4), (0, 8)
k: (7, 8), (7, 9), (7, 12), (12, 8), (12, 9), (12, 12)
v: (4, 5), (4, 9)
r: (9, 10)
ret: (1, ?)

Def-use analysis shows that variable ret is defined, but never used.

b.1
0. Tlength = TL
2. T ≠ null
4. v = TL
5. v ≠ 0
7. k = 0
8. k < TL
9. r = T[0] – T[v-1]
10. r = 0
12. k = 1
8. k ≥ TL

The path condition is TL = 1 (notice that v = 1, and T[0] = T[1-1]).
Test case: T = [4], Tlength = 1

b.2
0. Tlength = TL
2. T ≠ null
4. v = SL
5. v ≠ 0
7. k = 0
8. k < SL

9. r = T[0] – T[v-1]
10. r = 0
12. k = 1
8. k < SL
9. r = T[1] – T[v-2]
10. r = 0
12. k = 2
8. k ≥ SL

The path condition is TL = 2, T[0] = T[1] (notice that v = TL = 2, so 0 = T[0] – T[v-1] = T[1] – T[v-2])
Test case: T[-2, -2], Tlength = 2

Question 3: Architecture (red) (3 points)

We want to build a pipeline of the type

DataCollection → DataProcessing → DataStorage

using components that are at our disposal.
More precisely, *DataCollection* components are cheap (200 euros each), but not very available (availability of 80%). *DataProcessing* components, instead, are quite expensive (5000 euros each), but have good availability (availability of 99.9%). Finally, *DataStorage* components cost 2000 euros each, and have availability of 95%.
We have a budget of 20000 euros.
Define an architecture that stays within the budget while achieving an availability of 99.99%.

Motivate adequately your choices.

Solution:

None of the components at our disposal has the required availability of 99.99%, which means that each of them will have to be replicated and put in parallel with others.

For what concerns the *DataCollection* components, putting 6 of them in parallel achieves an availability of $1-0.2^6 = 0.999936$, or 99.9936% availability.
Putting 2 *DataProcessing* components in parallel achieves an availability of $1-0.001^2 = 0.999999$, or 99.9999%.
Putting 4 *DataStorage* components in parallel achieves an availability of $1-0.05^4 = 0.99999375$, or 99.999375%.
In total, we obtain an availability of 0.999936*0.999999*0.99999375, which is 0.99992875 and greater than the desired one.
The total cost is 200*6+5000*2+2000*4 = 19200, which is within the budget.

Question 3: Architecture (green) (3 points)

We want to build a pipeline of the type

DataCollection → DataProcessing → DataStorage

using components that are at our disposal.
More precisely, *DataCollection* components are cheap (300 euros each), but not very available (availability of 90%). *DataProcessing* components, instead, are quite expensive (4000 euros each), but have good availability (availability of 99%). Finally, *DataStorage* components cost 1500 euros each, and have availability of 92%.
We have a budget of 20000 euros.
Define an architecture that stays within the budget while achieving an availability of 99.99%.

Motivate adequately your choices.

Solution:

None of the components at our disposal has the required availability of 99.99%, which means that each of them will have to be replicated and put in parallel with others.

For what concerns the *DataCollection* components, putting 5 of them in parallel achieves an availability of $1-0.9^5 = 0.99999$, or 99.999% availability.
Putting 3 *DataProcessing* components in parallel achieves an availability of $1-0.01^3 = 0.999999$, or 99.9999%.
Putting 4 *DataStorage* components in parallel achieves an availability of $1-0.08^4 = 0.99995904$, or 99.995904%.
In total, we obtain an availability of 0.99999*0.999999*0.99995904, which is 0.99994804 and greater than the desired one.
The total cost is 300*5+4000*3+1500*4 = 19500, which is within the budget.