Dipartimento di Elettronica, Informazione e Bioingegneria

Politecnico di Milano

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

Prof. Matteo Camilli, Elisabetta Di Nitto, and Matteo Rossi

Software Engineering 2 – Written Exam 1 (WE1)

January 13th, 2023

Last name, first name and Id number (Matricola)
Number of paper sheets you are submitting as part of the exam

Notes

A. Remember to write your name and Id number (matricola) on each piece of paper that you hand in.
B. You may use a pencil.
C. Incomprehensible handwriting is equivalent to not providing an answer.
D. The use of any electronic apparatus (computer, cell phone, camera, etc.) is strictly forbidden, except for an ebook reader or a plain calculator.
E. The exam is composed of three exercises. Read carefully all points in the text!
F. Total available time for WE1: 1h and 30 mins

Question 1 – Alloy (7 points)

An aircraft is equipped with a fleet of drones for surveillance purposes. The surveillance area controlled by the aircraft can be divided in different sectors. Each sector can be assigned to a drone. There cannot be sectors assigned to multiple drones of the same aircraft.

Alloy_1 (2 points) Define an Alloy model that formalizes the description above.

Alloy_2 (1 points) Define a predicate to check if, given a certain surveillance area and an aircraft, the assignment is complete, that is, if all sectors of the area have been assigned to a drone belonging to the specified aircraft.

Alloy_3 (2 points) Extend the model to incorporate the following: Each sector may have one right, left, top and bottom other sectors. These are the sector neighbors.

Alloy_4 (2 points) Define a predicate to represent the change of sector assignment to a drone, considering that a drone can be reassigned only to a sector that is a neighbor of the current one and that the assignment should not break the general rules defined for the drone fleet.

Solution

Alloy_1

```
sig SurveillanceArea {}

sig Sector {
  belongsTo: SurveillanceArea
}

sig Drone {

  assignedTo: lone Sector
}

sig Aircraft {
  s: SurveillanceArea,
  fleet: set Drone
}{ no disj d1, d2 : fleet | d1.code = d2.code }

fact NoMultipleDroneInSector {
  all a: Aircraft | no disj d1, d2: a.fleet | #(d1.assignedTo) > 0 and
                                              #(d2.assignedTo) > 0 and
                                              d1.assignedTo = d2.assignedTo
}
```

Notice that one could also add some constraints to make sure, for example, that all sectors assigned to the drones of an aircraft *a* belong to the surveillance area of *a*.

Alloy_2

```
pred AssignmentComplete [a: SurveillanceArea, airc: Aircraft] {
   airc.fleet.assignedTo = belongsTo.a
}
```

We could also add the constraint that area *a* is the one surveilled by *airc*.

Alloy_3
The `Sector` signature can be modified as follows

```
sig Sector {
  belongsTo: SurveillanceArea,
  right: lone Sector,
  left: lone Sector,
  top: lone Sector,
  bottom: lone Sector,
  neighbors: set Sector
}{ neighbors = right + left + top + bottom and
   not this in neighbors and
   #right > 0 implies not ( right in left + top + bottom ) and
   #left > 0 implies not ( left in top + bottom ) and
   #top > 0 implies not ( top in bottom )
}
```

The following fact allows us to make sure the concept of neighbors is reciprocal

```
fact neighborsIsReciprocal {
     right = ~left and top = ~bottom
}
```

Alloy_4

```
sig Code {}

sig Drone {
  code: Code,
  assignedTo: lone Sector
}
```

Notice that we have introduced a field, `code`, in the `Drone` signature, to identify drones for the purposes of indicating, in the next predicate, that the drone that receives the new assignment is indeed the original one.

```
pred DroneReassignment [d: Drone, d': Drone, a: Aircraft, a': Aircraft, s: Sector]
{
  // precondition
  d in a.fleet
  no otherDrone: Drone | otherDrone in a.fleet and
       otherDrone.assignedTo = s
  d.assignedTo in s.neighbors
  s.belongsTo = a.surv

  //postcondition
  a'.s = a.s
  a'.fleet = a.fleet - d + d'
  d'.assignedTo = s
  d'.code = d.code
}
```

Question 2 – JEE (5 points)

Assume you are working for a software company that has recently decided to develop an internal system to let employees print documents directly from their smartphones. The software system has the following high-level requirements:

1. **R1**: the system shall allow the employees to authenticate through *username* and *password* (registration is not required since it is already managed by the system administrator);
2. **R2**: the system shall allow authenticated employees to print a given document using a printer among the available ones by specifying a motivation message;
3. **R3**: the system shall log the tuple <*timestamp, username, motivation*>, for each printed document, into a single text file named *printers.log* and stored in the local file system.

The first version of the software system adopts a multitier architecture developed using JEE. Specifically, the solution adopts two HttpServlet components that expose RESTful APIs to the clients, which are mobile applications running on the employees' devices. These APIs include the operations listed in the table below.

Main APIs

URI	method	description	parameters
/api/v1/userservice/login	POST	Login check and dispatch token to user.	username, password

/api/v1/printservice/printers	GET	Query the available printers.	token
/api/v1/printservice/print/{printerID}	POST	Create a new print job for a given printer	token, document, motivation message

A stateless session bean, called *UserManager*, implements the functions needed to satisfy R1. Another stateless session bean, called *PrinterManager*, implements the functions to meet R2 and R3. Concerning requirement R2, *PrinterManager* acts as a printer server, sending the print jobs to the appropriate printers using Internet Printing Protocol (IPP). Concerning requirement R3, the bean:

- opens the log file (after creation, if it does not exist) in the local directory */var/log/printers* of the file system;
- appends a new line containing the proper tuple;
- closes the file.

After a few hours since the deployment in production, the operators observe a lot of runtime exceptions of type *java.io.IOException* due to the corruption of the log file.

JEE_1 (1 point)
Explain possible root causes for this issue that relate to the wrong usage of the JEE framework in this context.

JEE_2 (2 points)
Once the root causes have been identified, the company plans a new release of the software system to fix the issue. How would you use the available JEE APIs to change the current implementation and avoid runtime exceptions? Motivate your answer.

JEE_3 (2 points)
After a while, the company grows (e.g., from 100 to 1000 employees) and the number of available printers grows as well (e.g., from 10 to 100). At a certain point, the employees start experiencing performance issues while using the application, especially in certain periods when many employees must print a lot of documents. So, the company plans a new release of the software that should fix such performance issues. How would you change the current requirements R1-R3 and the current JEE solution to deal with this issue? Motivate your answer.

Solution
JEE_1
The problem is due to the adoption of a stateless bean to implement the *PrinterManager*. Without any assumption on the operating system and low-level file locking mechanisms, multiple instances of the bean can concurrently access the same log file. This may cause file corruption and runtime IO exceptions.

JEE_2
Without any assumption on the operating system and low-level file locking mechanisms, the *PrinterManager* shall avoid race conditions while writing into a single text file. Using the available JEE APIs, we could wrap its functions in a unique Singleton Bean.

JEE_3
New R2: "the system shall allow authenticated employees to print a given document using the printer having the shortest queue among the available ones by specifying a motivation message".
New R3: "the system shall log the tuple <timestamp, username, motivation>, for each printed document, into a file associated with the used printer, named *«printerID».log*, and stored in the local file system".
The new release of the software meets the new requirements and uses multiple Singleton Beans (one for each available printer) for logging the print jobs into separate files. This way we avoid a unique Singleton component that represents a bottleneck.

Question 3 – Testing (4 points)

Consider the following function *decode* written in C language.

You can assume that function *hex_vals* takes as input a *char* that represents a hexadecimal value (between 0 and F) and returns an *int* value, that is, its decimal representation (-1 if the input is not a hexadecimal value). Function *unicode*, instead, takes as input an *int* value and returns a *char*, that is, the corresponding unicode representation.

```
0:  int decode(char s[], char t[]) {
1:     char c, dh, dl;
2:     int v, i = 0, j = 0;
3:     while (i < strlen(s)) {
4:         c = s[i];
5:         if (c == '+') {
6:             t[j++] = ' ';
7:         } else if (c == '%' && i + 2 < strlen(s)) {
8:             if (hex_vals(s[i + 1]) > 0 && hex_vals(s[i + 2]) > 0) {
9:                 v = hex_vals(s[i + 1]) * 16 + hex_vals(s[i + 2]);
10:                t[j++] = unicode(v);
11:            } else {
12:                return -1;
13:            }
14:            dh = s[i + 1];
15:            dl = s[i + 2];
16:            i += 2;
17:        } else {
18:            t[j++] = c;
19:        }
20:        i++;
21:    }
22:    t[j] = '\0';
23:    return 0;
24: }
```

Test_1 (2 points)
Build the control flow graph of function *decode* with def-use labels and then identify all def-use pairs for variables *i*, *j*, *dh*, *dl.* Then explain the potential issues they highlight, if any.

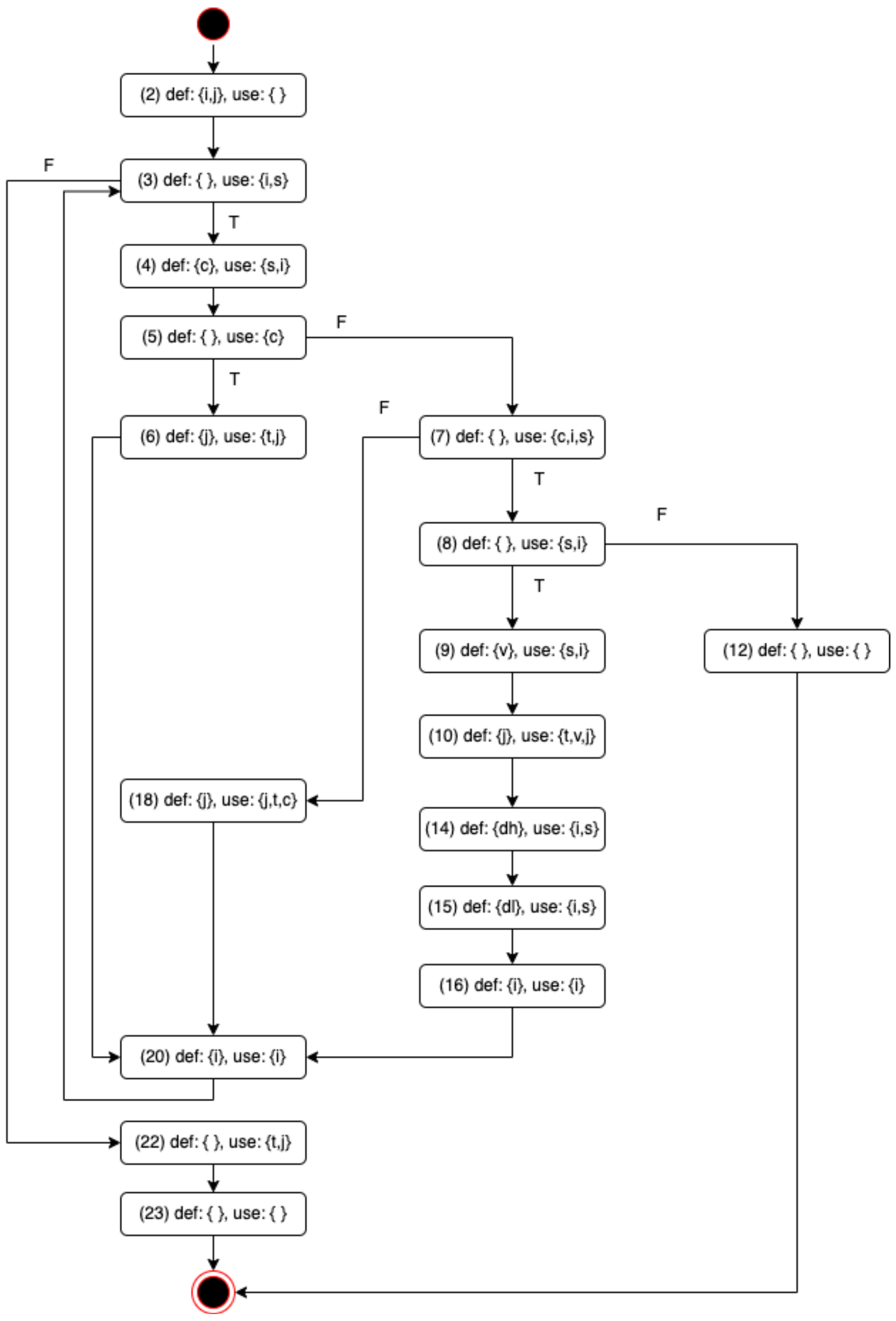
Test_2 (2 points)
Consider the following execution path:

- 0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 14, 15, 16, 20, 3, 22, 23

Symbolically execute the function covering it and find both the path condition and the final symbolic assignment.

Solution

Test_1



Notice that at lines 6, 10, 18, 22 parameter *t* is *used*, rather than defined, because it a pointer (i.e., an address), that is used to modify the cell of the array of index *j*, but the address is not modified itself.

i	(2,3) (2,4) (2,7) (2,8) (2,9) (2,14) (2,15) (2,16) (2,20) (16,20) (20,3) (20,4) (20,7) (20,8) (20,9) (20,14) (20,15) (20,16) (20,20)
j	(2,6) (2, 10) (2,18) (2, 22) (6,6) (6, 10) (6,18) (6, 22) (10,6) (10, 10) (10,18) (10, 22) (18,6) (18, 10) (18,18) (18, 22)
dh	NA
dl	NA

No issues for both *i* and *j*.

Variable *dh* defined in (14) and never used.
Variable *dl* defined in (15) and never used.

Test_2

```
0: s=S, t=T
1: -
2: i=0, j=0
3: {|S|>0}
4: c=S[0]
5: {S[0]!='+'}
7: {S[0]=='%' and |S|>2}
8: {S[1] in ['0', …, 'F'] and S[2] in ['0', …, 'F']}
9: v=hex_vals(S[1])*16 + hex_vals(S[2])
10: j=1, t[0]=unicode(hex_vals(S[1])*16 + hex_vals(S[2]))
14: dh=S[1]
15: dl=S[2]
16: i=2
20: i=3
3: {|S|<=3}
22: t[1]='\0'
23: -
```

Path condition: {|S|=3, S[0]='%', S[1] in ['0',...,'F'], S[2] in ['0',...,'F']}
Symbolic assignment: s=S, t=T, t[0]=unicode(hex_vals(S[1])*16 + hex_vals(S[2])) , t[1]='\0', i=3, j=1, v=hex_vals(S[1])*16 + hex_vals(S[2]), c=S[0]