Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Prof. Matteo Camilli, 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 2 – Written Exam 1 (WE1)

January 10, 2024

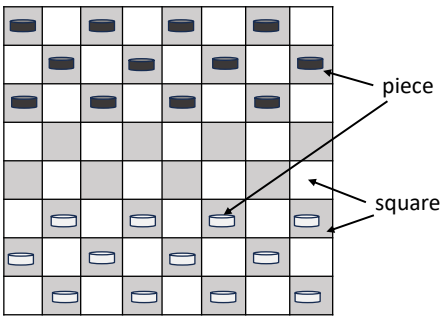
Last name, first name and Id number (matricola):

Number of paper sheets you are submitting as part of the exam:

Notes

A. Write your name and Id number (matricola) on each piece of paper that you hand in.
B. You may use a pencil.
C. Unreadable handwriting is equivalent to not providing an answer.
D. The use of any electronic devices (computer, smartphone, 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 (8 points)

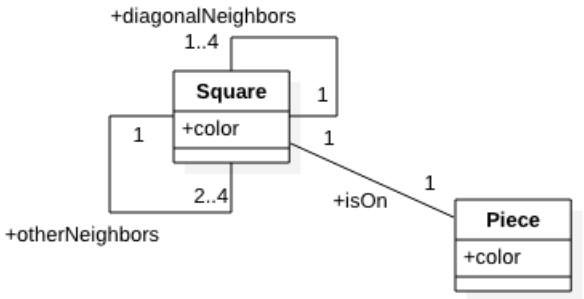


Consider the Italian Draughts (Dama Italiana) board game. The game is played on a board like the one shown in the figure. Two colors, black and white, alternate on the board so that all squares have other squares of their same color on the diagonals and squares of the opposite color on the other neighbor locations. Initially, pieces are positioned on the board as shown in the figure. They are either white or black and are all positioned on black squares. The game proceeds by moving pieces on the diagonals. In each move, a piece can move on a neighbor square on one of the diagonals if that square is not occupied by another piece.

Alloy_1 (3 points)
Given the following signatures:

```
abstract sig Color {}
one sig White extends Color {}
one sig Black extends Color {}
```

Use Alloy6 to model the elements of the game reported in the following UML class diagram (which is a simplification of the real board).



As shown in the diagram, given a square, we differentiate between the squares belonging to its diagonals and those belonging to the other neighbor positions. We want to ensure, also, that the squares on the diagonals have the same color as the reference one and the squares on the other positions have the opposite color. At each time instant, a piece is on a specific square and can change square during the evolution of the game.

Alloy_2 (1 point) Define a function `getSquares` that, given a piece, returns the set of squares that are free, that is, are not occupied by other pieces, in its diagonals.

Alloy_3 (2 points) Define the predicate `canCapture` that, given a piece, is *true* if that piece can capture another piece in its neighborhood. Capture of a piece A can take place if: i) A is on one of the diagonal neighbor positions of the current piece and ii) there is at least a free position on the diagonal neighbor positions of A.

Alloy_4 (2 points) Define the predicate `move` that, given a piece, models its movement from its current square to a different one. Moves can take place only on a free square on the diagonal neighbors of the current square (if any).

Solution

Alloy_1

```
abstract sig Color {}
one sig White extends Color {}
one sig Black extends Color {}

sig Square {
  color: Color,
  diagonalNeigh: some Square,
  otherNeigh: some Square,
}{
#diagonalNeigh >= 1 and #diagonalNeigh <= 4
#otherNeigh >= 2 and #otherNeigh <= 4
this not in diagonalNeigh + otherNeigh
}

//All squares have neighbors of the same color on the diagonals and of the
//opposite color on the other positions
fact squareColors {
   all s: Square | (all n: s.otherNeigh | s.color not in n.color) and
                   (all d: s.diagonalNeigh | s.color in d.color)
}
```

```
//The neighborhood relations are symmetric
fact squareOrganization {
  all s: Square | s in (s.otherNeigh).otherNeigh and
                      s in (s.diagonalNeigh).diagonalNeigh
}


sig Piece {
  player: Color,
  var onSquare: Square
}{onSquare.color in Black}

fact noMultiplePiecesOnASquare {
   all disj p1, p2: Piece | always (p1.onSquare & p2.onSquare = none)
}
```

Alloy_2

```
fun getSquares(p: Piece): set Square {
   (p.onSquare).diagonalNeigh - (Piece.onSquare)
}
```

Alloy_3

```
pred canCapture (p: Piece) {
  some p1: Piece | p1.onSquare in (p.onSquare).diagonalNeigh and
           (some s1: Square | s1 in getSquares[p1])
}
```

Alloy_4

```
pred move (p: Piece) {
  some s: Square | s in (p.onSquare).diagonalNeigh and
                   not s in Piece.onSquare and
                   p.onSquare' = s
}
```

Question 2 – Testing (5 points)

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

```
0:  void bar(int x) {
1:    int z, y;
2:    for (int i=0; i<2; i++) {
3:        y = pp(x);  // black-box function
4:        if (y > 3)
5:            i -= y;
6:        else
7:            i += y;
8:    }
9:    if (2*y > 6)
```

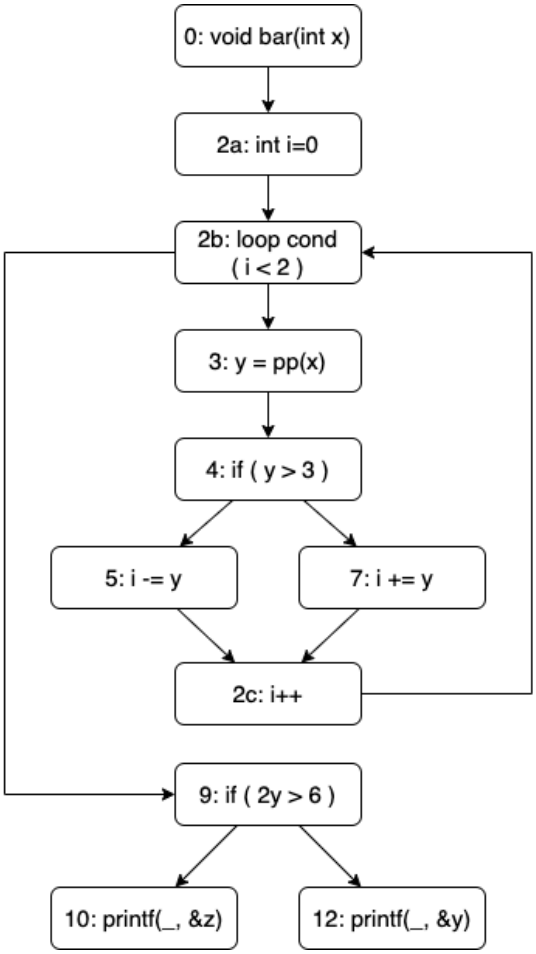
```
10:       printf("Log: %d", &z);
11:   else
12:       printf("Log: %d", &y);
13: }
```

Testing_1 (1 point) Draw the Control-Flow Graph (CFG) of the function *bar*.

Testing_2 (2 points) Derive all reaching definitions for variables *x*, *z*, *y*, *i*.
Testing_3 (2 points) Derive all UD chains and def-use pairs for variables *x*, *z*, *y*, *i*. What kind of potential issues emerge from the analysis?

Solution

Testing_1
Note that in the code fragment at lines 10 and 12 there are two mistakes: we want to print the values of z and y, not their addresses.



Testing_2
$RD_{IN}(0) = \{(x,?)\ (z,?)\ (y,?)\ (i,?)\}$
$RD_{OUT}(0) = RD_{IN}(2a) = \{(x,0)\ (z,?)\ (y,?)\ (i,?)\}$
$RD_{OUT}(2a) = \{(x,0)\ (z,?)\ (y,?)\ (i,2a)\}$
$RD_{IN}(2b) = RD_{OUT}(2a) \cup RD_{OUT}(2c) = \{(x,0)\ (z,?)\ (y,?)\ (y,3)\ (i,2a)\ (i,2c)\}$
$RD_{OUT}(2b) = RD_{IN}(2b) = RD_{IN}(3)$
$RD_{OUT}(3) = RD_{IN}(4) = \{(x,0)\ (z,?)\ (y,3)\ (i,2a)\ (i,2c)\}$
$RD_{OUT}(4) = RD_{IN}(5) = RD_{IN}(7) = \{(x,0)\ (z,?)\ (y,3)\ (i,2a)\ (i,2c)\}$

$RD_{OUT}(5) = \{(x,0)\ (z,?)\ (y,3)\ (i,5)\}$
$RD_{OUT}(7) = \{(x,0)\ (z,?)\ (y,3)\ (i,7)\}$
$RD_{IN}(2c) = RD_{OUT}(5) \cup RD_{OUT}(7) = \{(x,0)\ (z,?)\ (y,3)\ (i,5)\ (i,7)\}$
$RD_{OUT}(2c) = \{(x,0)\ (z,?)\ (y,3)\ (i,2c)\}$
$RD_{IN}(9) = RD_{OUT}(2b) - \{(i,2a)\ (i,2c)\} = \{(x,0)\ (z,?)\ (y,?)\ (y,3)\}$
$RD_{OUT}(9) = RD_{IN}(9) = RD_{IN}(10) = RD_{IN}(12)$
$RD_{OUT}(10) = RD_{IN}(10)$
$RD_{OUT}(12) = RD_{IN}(12)$

Note that the scope of variable i is limited to the for loop, given its declaration as part of the for. This implies that it is not part of $RD_{IN}(9)$

Testing_3

UD chains for variables: *x*, *z*, *y*, *i.*

UD(i, 2b) = {2a,2c}
UD(x, 3) = {0}
UD(y, 4) = {3}
UD(i, 5) = {2a, 2c}
UD(y, 5) = {3}
UD(i, 7) = {2a, 2c}
UD(y, 7) = {3}
UD(i, 2c) = {5, 7}
UD(y, 9) = {?, 3}
UD(z, 10) = {?}
UD(y, 12) = {?,3}

Def-use pairs for variables: *x*, *z*, *y*, *i.*

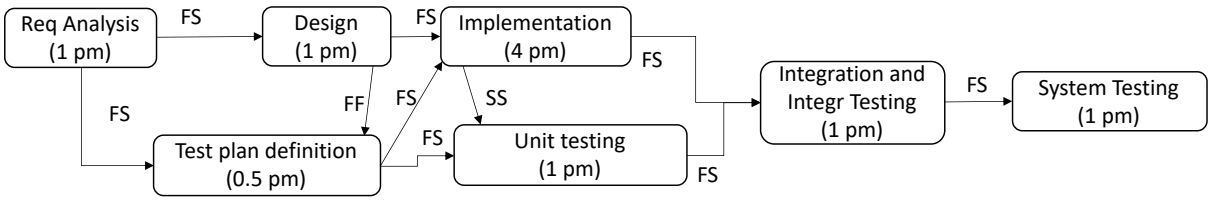
x: (0,3)
z: (?,10)
y: (3,4) (3,7) (3,5) (3,9) (?,9) (?,12) (3,12)
i: (2c,2b) (2a,2b) (2a,7) (2c,7) (2a,5) (2c,5) (7,2c) (5,2c)

Possible issues:

- Variable *z* used without definition at block 10
- Variable *y* used without definition at block 9 and 12. Note that through symbolic execution we can find out that this situation can never happen since the path that excludes the execution of the for loop will never be followed. However, the analysis technique adopted in this case is pessimistic and does reveal this as a potential problem.

Question 3 – Project management (3 points)

Consider the following Project Schedule Network Diagram, which includes also the effort associated with each task expressed in person-months (pm).



Assume that your project team is composed of the following experts: 2 people focusing on requirement analysis and design, 4 people focusing on implementation, unit testing, integration, and integration testing, 4 people focusing on quality assurance.
Define a Gantt chart of the project and identify the critical path. The Gantt chart should minimize the total project duration.
Assume, for simplicity, that the team is working 4 weeks per month, does not take any holiday, and that the time spent for coordination between team members is negligible (that is, team members can work perfectly in parallel).
Provide the motivations for your choices.

Solution
Considering 2 people focusing on requirement analysis and design, both activities can be performed in two weeks each.
Considering the finish-to-finish (FF) dependency of the test plan definition activity from the design activity, even if the first one requires only two weeks effort and the QA group is composed of 4 people, it will finish together with the design activity. This implies that the QA group will be involved only part time in this whole period. The group will start defining the test plan based on the outcomes of the requirement analysis phase and will amend and complete it based on the results produced by the design activity.
Both the implementation and unit testing activities will be carried out by the development subteam (4 people). They, in fact, will conduct unit testing while developing the software. This implies that these two activities, requiring an overall effort of 5 pm, will be completed in 5 weeks. The two activities, of course, will proceed in parallel given the role of unit testing in software development.
Integration and integration testing will be performed by the whole implementation team in one week. Finally, system testing will be performed by the QA team and will last one week.
The resulting Gantt chart is shown below. For simplicity, the dependencies that are already reported in the Project Schedule Network Diagram are not included here. The overall duration of the project will be 11 weeks. The critical path is the one shown in red.

	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11
Req Analysis											
Design											
Test plan def											
Implementation											
Unit testing											
Integration and integr testing											
System testing											