Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Prof. Elisabetta Di Nitto, Luca Mottola

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

Software Engineering II

February 13 2017

Last Name
First Name
Id number (Matricola)

Note

1. The exam is not valid if you don't fill in the above data.
2. Write your answers on these pages. Extra sheets will be ignored. You may use a pencil.
3. The use of any electronic apparatus (computer, cell phone, camera, etc.) is strictly forbidden.
4. You cannot keep a copy of the exam when you leave the room.

Question 1 Alloy (7 points)
Consider construction cubes of three different sizes, small, medium, and large. You can build towers by piling up these cubes one on top of the other respecting the following rules:

- A large cube can be piled only on top of another large cube
- A medium cube can be piled on top of a large or a medium cube
- A small cube can be piled on top of any other cube
- It is not possible to have two cubes, A and B, simultaneously positioned right on top of the same other cube C

Question 1: Model in Alloy the concept of cube and the piling constraints defined above.

Question 2: Model also the predicate canPileUp that, given two cubes, is true if the first can be piled on top of the second and false otherwise.

Question 3: Consider now the possibility of finishing towers with a top component having a shape that prevents further piling, for instance, a pyramidal or semispherical shape. This top component can only be the last one of a tower, in other words, it cannot have any other component piled on it.
Rework your model to include also this component. You do not need to consider a specific shape for it, but only its property of not allowing further piling on its top. Modify also the canPileUp predicate so that it can work both with cubes and top components.

Solution

Model that answers to Questions 1 and 2

```
abstract sig Size{}
sig Large extends Size{}
sig Medium extends Size{}
sig Small extends Size{}

sig Cube {
  size: Size,
  piledOn: lone Cube
}{piledOn != this}

fact noCircularPiling {
  no c: Cube | c in c.^piledOn
}

fact pilingUpRules {
  all c1, c2: Cube | c1.piledOn = c2 implies (
    c2.size = Large or
    c2.size = Medium and (c1.size = Medium or c1.size = Small) or
    c2.size = Small and c1.size = Small)
}

fact noMultipleCubesOnTheSameCube {
 no disj c1, c2: Cube | c1.piledOn = c2.piledOn
}

pred canPileUp[cUp: Cube, cDown: Cube] {
    cUp.piledOn = cDown and
    (cDown.size = Large or
    cDown.size = Medium and (cUp.size = Medium or cUp.size = Small) or
```

```
    cDown.size = Small and cUp.size = Small)
}

pred show {}
run show
run canPileUp
```

Model that answers to Question 3

```
abstract sig Size{}
sig Large extends Size{}
sig Medium extends Size{}
sig Small extends Size{}

abstract sig Block{
  piledOn: lone Cube
}

sig Cube extends Block {
  size: Size
}{piledOn != this}

sig Top extends Block {
}

fact noCircularPiling {
  no c: Cube | c in c.^piledOn
}

fact noMultipleBlocksOnTheSameCube {
 no disj b1, b2: Block | b1.piledOn = b2.piledOn
}

fact pilingUpRules {
  all c1, c2: Cube | c1.piledOn = c2 implies (
    c2.size = Large or
    c2.size = Medium and (c1.size = Medium or c1.size = Small) or
    c2.size = Small and c1.size = Small)
}

pred canPileUp[bUp: Block, cDown: Cube] {
    bUp.piledOn = cDown and (bUp in Top or
    (cDown.size = Large or
    cDown.size = Medium and (bUp.size = Medium or bUp.size = Small) or
    cDown.size = Small and bUp.size = Small))
}

pred show {}
run show
run canPileUp
```

Questions 2 JEE (6 points)

Consider the following scenario. We need to develop an online product management application for an enterprise. The application should allow product managers to:

- Create products, inserting the corresponding name and available quantity in the database.
- Update the available quantity of an existing product.
- Perform simple analytics about the products, such as finding products whose available quantity is lower than a given threshold.

You have been contracted to develop this system using JEE. In particular, you are requested to start developing the following elements:

- **A simple entity representing products**. It includes the product name, an ID, an integer that represents the available quantity.
- **A bean** that queries the database and offers the following operations:
    - public Product *insertProduct*(String name, Integer, quantity)
    - public Product *updateProductQuantity*(String productName, Integer newQuantity)
    - public List <Product> *selectExpiringProducts*(Integer quantityLowerBound)

Notes:

- You don't need to implement getter and setter methods for the entities, as it is assumed that they can be automatically generated by any IDE.
- You don't need to consider and handle possible exceptions that might be thrown.

Solution

See the code in the associated zip file

Question 3 Architectures and Testing (7 points)

Consider the following fragment of Java code:

```
public class Foo {
0.      public static int foo(int x, int y) {
1.              int l;
2.              if (x <= 0 || y == 0)
3.                      return y;
4.              l = y % 2;
5.              while (x >= 2) {
6.                      if (x > 0 || l >= 0)
7.                              x = Math.abs(x) - 1;
8.                      else return -Math.abs(x);
                }
9.              return x;
        }
}
```

You are to:

1) Execute the code symbolically, making sure that every execution path reaching the **`while`** executes **at most one** iteration of the loop. You are to present the results of symbolic executions in the form:

   <symbolic values of variables, sequence of instructions executed, symbolic result, corresponding path condition>

2) Determine potential software defects, if any, that symbolic execution may identify in the fragment of code above. For every such issue, explain why symbolic execution can pinpoint the problem and mention at least one alternative approach at verifying code that would **not** be able to identify the same issue.
3) Assume you include the code above in a component called LOO with 95% availability. Two additional components offering the same functionality of LOO are available: component BOO with 98% availability, and component MOO with 70% availability. What component configuration (combination of the three components) would you use to obtain an overall availability equal to or above 99%? Note that you can use only one copy of each component, and you could use fewer than all available components.

Solution
Point 1

Path <0, 1, 2, 3>

0. x = X, y = Y
1.
2. X <=0 or Y == 0
3. return Y

Thus we have that:
<x = X, y = Y; 0, 1, 2, 3; Y; X <=0 or Y == 0>

Path <0, 1, 2, 4, 5, 6, 7, 5, 9>

0. x = X, y = Y
1.

2. X > 0 and Y != 0
4. l = Y % 2
5. X >= 2
6. X >= 2 (since the first conjunct of the condition is true, because of short-circuit evaluation, the truth value of the second part is immaterial)
7. x = abs(X)-1 = X – 1 (X is necessarily positive because of the path condition)
5. X -1 < 2 (because the loop exits at this point; this entails that X is equal to 2 and Y is irrelevant for the execution of this path)
9. return 1

Thus we have that:
<x = X, y = Y, l = Y % 2; 0, 1, 2, 4, 5, 6, 7, 5, 9; 1; X == 2>

Path <0, 1, 2, 4, 5, 9>

0. x = X, y = Y
1.
2. X > 0 and Y != 0
4. l = Y % 2
5. X < 2
9. return 1 (this is necessarily the case, because X>0 and X < 2 for this path to be executed)

Thus we have that:
<x = X, y = Y, l = Y % 2; 0, 1, 2, 4, 5, 9; 1; X == 1>

Path <0, 1, 2, 4, 5, 6, 8>: since to enter in the loop we need to have X >= 2, this implies that the **if** condition in the loop will be always true; thus, this path cannot be executed and any instruction in the else part is unreachable.

Point 2

By symbolically executing the code fragment above we can discover the unreachability of path <0, 1, 2, 4, 5, 6, 8, 9> Also, we can understand that the value of y is relevant only in the case of path <0, 1, 2, 3>. For all paths entering the **while** loop, the value of y is irrelevant. Moreover, the value of l is never used and does not contribute to any path condition.

We could discover the same issues by inspecting the code. Using testing, however, we would not identify the unused variables and unreachable fragment of code. Traditional testing is not necessarily exhaustive. We may apply different coverage criteria, for example, by exercising all conditions in the code, to eventually discover that this function never returns negative values.

Point 3

Because the overall availability of individual components is lower than 99%, the most natural way to achieve >=99% availability is to place components in parallel. It turns out it is sufficient to place BOO in parallel to LOO to achieve 1-(0.02*0.05)=0.999 availability.