Dipartimento di Elettronica e Informazione
Politecnico di Milano
20133 Milano (Italia)
Piazza Leonardo da Vinci, 32
Tel. (39) 02-2399.3400
Fax (39) 02-2399.3411

prof. Carlo Ghezzi

Software Engineering II

14 January 2009

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 (5 POINTS)

In a social networking application a *person* can have any number of *friends*. The friendship relation is symmetric but not reflexive.

For a person *x*, the set of *candidate friends* for *x* is defined. It includes all and only (1) the friends of the friends of *x* and (2) recursively, the friends of the candidate friends of *x*. It does not contain *x* nor *x*'s friends.

(1) Provide an Alloy formal specification of the previous informal statements. Specifically, introduce a sig *Person*, with possible constraints, and a function *candFriends* which yields the candidate friends of a person. (**3 POINTS**)

(2) Specify an operation *promote* which promotes a person *q* to the role of a friend for a person *p*. The operation can only be applied if *q* is a *candidate friend* of *p*. (**2 POINTS**)

ANSWER

```
sig Person {
        friend: set Person
}


fact {
        no p: Person | p in p.friend
        all p1, p2: Person | p1 in p2.friend <=> p2 in p1.friend
}

fun candidateFriend (p: Person) : set Person {
 p.friend.^friend - (p + p.friend)
}


pred promote (p, p', q: Person) {
        q in candidateFriend[p]
        p'.friend=p.friend+q
}

pred showPromote (p, p', q: Person) {
        promote [p, p', q]
        #Person.friend>2
}

run showPromote for 4
```

Question 2 – Testing (6 POINTS)

Consider the following code fragment:

```
1       get(x);
2       get(y);
3       while x ≠ y {
4               if x>y
5                       x=x-y;
                else
6                       y=y-x;
        }
7       return (x)
```

1. Draw the control graph of the program and, for each node of the graph, show the the used and the defined variables. (**2 POINTS**)

2. Suppose you wish to generate test cases that cover all definition-use (def-use) pairs for each variable. A def-use pair for a variable (e.g., X), is defined by two instructions, the first of which is a definition of X, the second is a use of X, and between the two instructions there exists **at least** one execution path along which no other subsequent definition is found.

    a. List all the def-use pairs for the previous fragment (to identify the statements corresponding to the def-use pairs, please use the number used above to label the statements). (**2 POINTS**)
    b. Write a set of test cases that covers all def-use pairs and, for each test case, explicitly describe which def-use pairs are covered. (**1 POINT**)

OPTIONAL (1 EXTRA POINT)

How can you compare the coverage criterion discussed here with statement coverage? Does a test set that satisfies coverage of all de-use pairs also cover all statements? Or vice-versa? Or no relation exists between the two. Briefly elaborate on this.

ANSWER

(Part 1 omitted for simplicity)

Def-use pairs:
(1, 3) (1, 4) (1, 5) (1, 6), (1, 7)
(2, 3) (2, 4) (2, 5) (2, 6) (2, 7)
(5, 3) (5, 4) (5, 5) (5, 6) (5, 7)
(6, 3) (6, 4) (6, 5) (6, 6) (6, 7)

Test cases 1 x=8, y=4 and x=4, y=8 cover paths 1, 2, 3, 4, 5, 3, 7 and 1, 2, 3, 4, 6, 3, 7. Therefore they cover the first two sets of def-use pairs above.
Test cases x=12, y=4 and x=4, y=12 cover paths 1, 2, 3, 4, 5, 3, 4, 5, 3, 7 and 1, 2, 3, 4, 6, 3, 4, 6, 3, 7. Therefore, they cover the two other sets of def-use pairs.

Every statement corresponds to a definition and/or a use of some variable. Thus coverage of all def-use pairs covers, by definition, all statements that correspond to a definition (provided that a path exists from that definition to a corresponding use). Considering statements that correspond to a use, all of those to which there is a path from a corresponding definition are also covered. The only statements that may not be covered are those corresponding to a definition that does not lead to a corresponding use or a use to which there is no definition that leads. Both cases correspond to incorrect or pathological programs (which can be detected by dataflow static analysis). So if we consider consider correct programs, coverage of all def-use pairs implies coverage of all statements.

Question 3 – Development process (3 POINTS)

Suppose you are asked to implement a protocol which supports data transmission between a pc and a sensor/actuator that controls a certain plant. The protocol is defined by an ISO standard, for which a precise and exhaustive description is available.

Which development process (i.e., lifecycle model) would you adopt? Waterfall? Prototype-based? Or would you adopt an agile methodology, like extreme programming?

Provide a concise justification of your answer. If more than one alternative could be considered, briefly illustrate the advantages and possible disadvantages of each.

ANSWER

In our case, requirements are well-known and stable. A waterfall lifecycle looks ideal, because there is no need for re-cycling. One might consider adopting a lifecycle based on prototyping if there are various implementation alternatives that might affect response time or memory usage. Each prototype might explore different implementations for certain parts. An “agile” methodology does not make sense, because there is no need for involving the user. Perhaps, however, certain specific techniques of agile (estreme) programming, like pair programming or test-driven development, might be adepted.

Question 4 – Requirements (6 POINTS)

A library supports users in borrowing and consulting books. Two modes are supported:

- **Borrowing** books (**B**): the user must be registered and in this case he/she can take the book out of the library building.
- **Consulting** books (**C**): the user can only consult the book inside the library building.

We wish to implement an automated system that supports the library management. According to the Jackson/Zave approach, assume (among others that are ignored for simplicity) the following ***goal*** to be achieved:

> *At any given point in time, if a book is not in the library building, it must be borrowed by a registered user.*

In fact, if this goal is satisfied the library staff can contact the borrower in case they need to get back the book for some urgent reason.

Suppose that, among others, the following ***requirement*** is defined in order to constrain the interface between the system to be and the external world:

> *The system allows only registered users to borrow a book..*

(1) Using predicates *out(b)* (to indicate that a book *b* is outside the library building), *registered(u)* (to indicate if user *u* is registered), *borrowed(b, u)* (to indicate that book *b* is borrowed by user *u*), formalize the goal and the requirement stated above. (**3 POINTS**)

(2) Define a predicate that describes the assumption on the domain under which the goal can be satisfied, assuming that the implementation satisfies the requirement. You may provide an intuitive justification instead of a formal one. (**2 POINTS**)

OPTIONAL (1 EXTRA POINT)

Can you provide a concise intuitive explanation of how the domain might exhibit the property specified by such predicate?

ANSWER

GOAL
forall b:book (out(b) =>
exists u: user borrowed(b, u) and registered(u)

REQ
forall b: book, u: user borrowed(b, u) => registered(u)

DOM
forall b:book | out(b) =>
exists u: user borrowed(b, u)

Intuitively, this predicate specifies that a book cannot be allowed to go out of the building unless it is borrowed. This means that either we trust library customers to behave correctyly (i.e., we trust people who just consult books not to leave the library carrying the book) or, even better, we assume that some control system is in place so that a suitable RFID is attached to books and an acoustic signal is generated if a book which is not borrowed exists the building.

Question 5 – Software architecture (3 POINTS)

Briefly compare the main characteristics of two software architectures, one based on the "observer pattern", which supports a coordination among components based on the concept of "listener", and another where coordination among components is based on a "publish/subscribe" middleware, where a broker (dispatcher) supports subscriptions and event (message) delivery to components.

What is common to the two architectures? What is different? Briefly explain.

ANSWER

In both cases certain software components can subscribe to events generated by other components, and be notified whan that happens.
The main differences between case A (pattern observer) and case B (middleware publish subscribe) are:

- in A the subscriber must concat directly the component that generates the events of interest, while in B one subscribes by specifying a pattern that descrive the events of interest, ignoring which components actually generate them. Similarly, the components that generate events ignore who is listening and how many listeners exist.
- Adding or removing event generators or event listeners does not affect the other components, in case B.