This file collects some exercises adapted from previous exams and focusing on the following subjects
Symbolic execution
Automated analysis - def-use
Testing - Statechart coverage
Planning, design, identification of acceptance testing
Architecture and availability
Alloy

Question Testing (4 points) (18/01/2013)

Consider the following function, written in a C-like programming language:

```
1 int foo(int a, int b) {

2  a++;

3  while (a < b) {
4    if (a != b) {
5      a++;
6    }
7   }
8  return a;
9 }
```

You are to:

- Execute `foo` symbolically limiting the execution of the loop statement to *exactly two iterations*. Show, for each non-conditional statement being executed, the *symbolic value* of all variables and the *path condition* that applies at that point in the execution.
- Define the pre-condition to the execution of `foo` such that the `while` loop is executed exactly twice.

Solution

The symbolic execution of `foo` unfolds as follows, considering `A`, `B` the starting symbols for variable `a` and `b`, respectively:

1) `a = A, b = B` when foo starts executing;
2) `a = A+1, b = B` after executing line 2;
3) if `[A+1 < B]` the while loop executes;
4) as the execution entered in the loop, `a != b` necessarily; then the execution continues on line 5 and `a = A+2, b = B`;
5) the **`while`** condition is checked again and the loop executes if `[A+2 < B]`;
6) again, as the execution entered the loop, `a != b` necessarily; the execution continues on line 5 and `a = A + 3, b = B`;
7) the while condition is checked again; as the loop should not be executed again, the path-condition should be `[A+3 < B]`; in this case the execution jumps to line 7

8) the function ends and the value `A + 3` is returned.

The pre-condition on the execution of foo that guarantees that the loop is executed exactly twice is then `b = a+3`

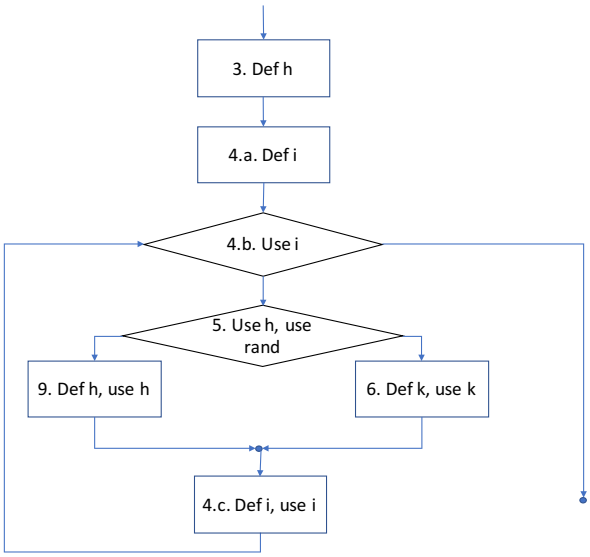
Question 2 Testing (01/03/2013)
Consider the following function, written in a C-like language where function `rand()` returns a pseudo-random (integer) number:

```
1 int foo() {

2     int h,k;
3     h = 0;
4     for (int i = 0; i < 10; i++) {
5          if (h > rand()) {
6               k++;
7          }
8          else {
9                 h++;
10        }
11    }
12 }
```

You are required to:

- Identify the data flow graph for function `foo()` along with definition and uses of all variables involved;
- Define the def-use pairs for foo()
- Provide an example of what kind of potential problem a typical verification effort using data flow analysis may discover in `foo()`.

Solution



variable	<def, use> pairs

h	<3, 5>, <3, 9>, <9, 5>, <9, 9>
i	<4.a, 4.b>, <4.a, 4.c>, <4.c, 4.b>, <4.c, 4.c>
k	<6, 6>

The problem we note is concerned with k and with its def-use pair <6, 6>: here k is used before being defined. So, depending on the value of rand and on the outcome of the condition at point 5, we can end up in an execution problem with this code.

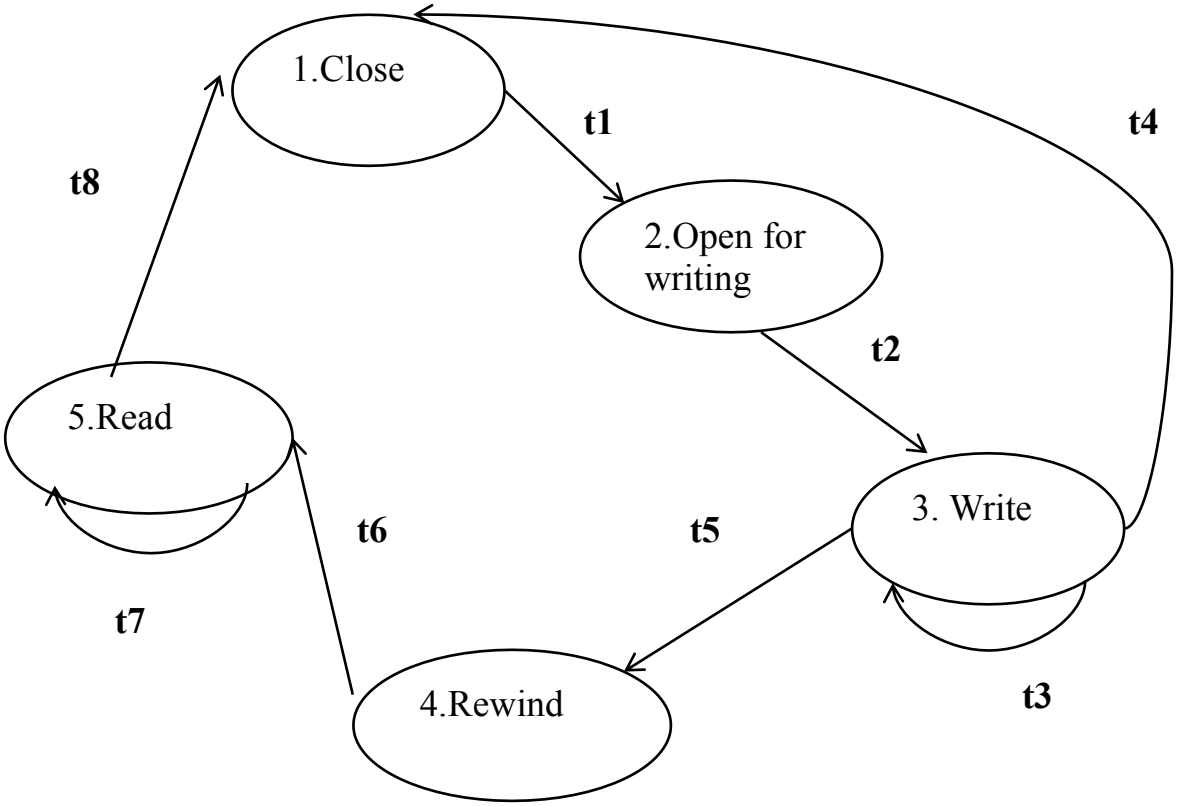
Question 3: Testing (6 points) (15/01/2015)
Consider the following specification of possible operations on sequential files.

To perform write (WRITE) operations on a given file, it is first necessary to open it (OPEN). After writing, to perform read (READ) operations, it is first necessary to carry out a rewind (REWIND) operation that moves the cursor to the beginning of the file.

Describe a possible approach to use this specification to derive test cases.
Add explicitly any hypothesis you think is important to complete the specification.

Solution

Given the description above, the approach to be used is black box testing.
Having the specification, it is possible deriving a state diagram describing the system behavior and then identifying the test cases.
A possible state diagram is illustrated below:



Test cases depend on the selected coverage criterion: coverage of states, or coverage of transitions.

In the first case, the test cases are (numbers refer to states):
TC1: 1 2 3 1
TC2: 1 2 3 4 5 1

In the second case:
TCa: t1 t2 t3 t3 t4
TCb: t1 t2 t4
TCc: t1 t2 t5 t6 t7 t8
TCd: t1 t2 t5 t6 t8

Note that, as the statechart describes only the allowed state transitions, it is not possible to use a coverage criterion on it to identify test cases that indentify the attempts to perform forbidden or non-specified operations, for instance, the case when someone tries to open, write and read (states 1, 2, 3, 5) or open, rewind and read (1, 2, 4, 5). The first of these test cases could have been identified is we had modelled in the statechart also the exception handling cases concerning in particular the fact that the write – read sequence should lead into an exceptional or incorrect state.

Question 4: Testing (5 points) (09/07/2015)

Consider the following fragment of a program using natural numbers (nat)

```
1. nat y, a, b;
2. read (a, b, y);
3. while (a < y) {
4.        if (a < b)
5.                  a = b;
```

```
6.        else if (a > b)
7.                  b = a;
8.           else a++; }
9. ...
```

You are to:

- Define the flow graph for the fragment above augmented with the annotations concerning the definition and usage of the variables
- Execute the program symbolically showing different possible execution paths involving the execution of statement 7.
- For each of these paths show the path condition and possible test cases.

Solution

A) Flow graph is quite basic, definition and usage of variables is as follows:
1, 2: Def a, b, c
3: Use a, y
4: Use a, b
5: Use b, Def a
6: Use a, b
7: Use a, Def b
8: Use a, Def a

B) Symbolic execution of possible paths involving statement 7

We try to identify those paths that pass through 7 and stay in the while for the smallest possible number of iterations:
Path 1
<1, 2, 3, 4, 6, 7, 3, 9> **Not possible, in fact:**
1, 2: a = A, b = B, y = Y, path condition: true
3: a = A, b = B, y = Y, path condition: A<Y
4: a = A, b = B, y = Y, path condition: A<Y and not(A<B)
6: a = A, b = B, y = Y, path condition: A<Y and A>B
7: a = A, b = A, y = Y, path condition: A<Y and not(A<B)
3: a = A, b = A, y = Y, path condition: A<Y and not(A<B) and not(A>Y), which is not possible

Path II
<1, 2, 3, 4, 6, 7, 3, 4, 6, 7, 3, …> **Not possible, in fact:**
1, 2: a = A, b = B, y = Y, path condition: true
3: a = A, b = B, y = Y, path condition: A<Y
4: a = A, b = B, y = Y, path condition: A<Y and not(A<B)
6: a = A, b = B, y = Y, path condition: A<Y and A>B
7: a = A, b = A, y = Y, path condition: A<Y and not(A<B)
3: a = A, b = A, y = Y, path condition: A<Y and not(A<B)
4: a = A, b = A, y = Y, path condition: A<Y and not(A<B) and not(A<A)
6: a = A, b = A, y = Y, path condition: A<Y and not(A<B) and not(A<A) and A>A, which is not possible

Path III
<1, 2, 3, 4, 6, 7, 3, 4, 5, 3, …> **Not possible, in fact:**

1, 2: a = A, b = B, y = Y, path condition: true
3: a = A, b = B, y = Y, path condition: A<Y
4: a = A, b = B, y = Y, path condition: A<Y and not(A<B)
6: a = A, b = B, y = Y, path condition: A<Y and A>B
7: a = A, b = A, y = Y, path condition: A<Y and not(A<B)
3: a = A, b = A, y = Y, path condition: A<Y and not(A<B)
4: a = A, b = A, y = Y, path condition: A<Y and not(A<B) and A<A, which is not possible

Path IV
<1, 2, 3, 4, 6, 7, 3, 4, 6, 8, 3, 9> **, path condition A<Y and A>B and A+1==Y, in fact:**
1, 2: a = A, b = B, y = Y, path condition: true
3: a = A, b = B, y = Y, path condition: A<Y
4: a = A, b = B, y = Y, path condition: A<Y and not(A<B)
6: a = A, b = B, y = Y, path condition: A<Y and A>B
7: a = A, b = A, y = Y, path condition: A<Y and not(A<B)
3: a = A, b = A, y = Y, path condition: A<Y and not(A<B)
4: a = A, b = A, y = Y, path condition: A<Y and not(A<B) and not(A<A)
6: a = A, b = A, y = Y, path condition: A<Y and not(A<B) and A==A
8. a = A+1, b = A, Y = Y, path condition: A<Y and not(A<B)
3. a = A+1, b = A, Y = Y, path condition: A<Y and not(A<B) and A+1==Y
9. …

Path V
<1, 2, 3, 4, 5, 3, 4, 6, 7, 3, …> **Not possible, in fact:**
1, 2: a = A, b = B, y = Y, path condition: true
3: a = A, b = B, y = Y, path condition: A<Y
4: a = A, b = B, y = Y, path condition: A<Y and A<B
5: a = B, b = B, y = Y, path condition: A<Y and A<B
3: a = B, b = B, y = Y, path condition: A<Y and A<B and B < Y
4: a = B, b = B, y = Y, path condition: A<Y and A<B and B < Y and not(B<B)
6: a = B, b = B, y = Y, path condition: A<Y and A<B and B < Y and not(B<B) and B>B, which is not possible

Path VI
<1, 2, 3, 4, 5, 3, 4, 6, 8, 3, 4, 6, 7, 3, 4, 6, 8, 3, 9>, **path condition A<Y and A<B and B+2==Y, in fact:**
1, 2: a = A, b = B, y = Y, path condition: true
3: a = A, b = B, y = Y, path condition: A<Y
4: a = A, b = B, y = Y, path condition: A<Y and A<B
5: a = B, b = B, y = Y, path condition: A<Y and A<B
3: a = B, b = B, y = Y, path condition: A<Y and A<B and B < Y
4: a = B, b = B, y = Y, path condition: A<Y and A<B and B < Y and not(B<B)
6: a = B, b = B, y = Y, path condition: A<Y and A<B and B < Y and B==B
8: a = B+1, b = B, y = Y, path condition: A<Y and A<B and B < Y and B==B
3: a = B+1, b = B, y = Y, path condition: A<Y and A<B and B < Y and B+1<Y
4: a = B+1, b = B, y = Y, path condition: A<Y and A<B and B < Y and B+1<Y and not(B+1<B)
6: a = B+1, b = B, y = Y, path condition: A<Y and A<B and B < Y and B+1<Y and B+1>B
7: a = B+1, b = B+1, y = Y, path condition: A<Y and A<B and B < Y and B+1<Y
3: a = B+1, b = B+1, y = Y, path condition: A<Y and A<B and B < Y and B+1<Y
4: a = B+1, b = B+1, y = Y, path condition: A<Y and A<B and B < Y and B+1<Y and not(B+1<B+1)

6: a = B+1, b = B+1, y = Y, path condition: A<Y and A<B and B < Y and B+1<Y and B+1==B+1
8: a = B+2, b = B+1, y = Y, path condition: A<Y and A<B and B < Y and B+1<Y
3: a = B+2, b = B+1, y = Y, path condition: A<Y and A<B and B < Y and B+2==Y

Path VII
<1, 2, 3, 4, 6, 8, 3, 4, 6, 7, 3, 4, 6, 8, 3, 9>, **path condition A<Y and A==B and A+2==Y, in fact:**
1, 2: a = A, b = B, y = Y, path condition: true
3: a = A, b = B, y = Y, path condition: A<Y
4: a = A, b = B, y = Y, path condition: A<Y and not(A<B)
6: a = A, b = B, y = Y, path condition: A<Y and A==B
8: a = A+1, b = B, y = Y, path condition: A<Y and A==B
3: a = A+1, b = B, y = Y, path condition: A<Y and A==B and A+1<Y
4: a = A+1, b = B, y = Y, path condition: A<Y and A==B and A+1<Y and not(A+1<B)
6: a = A+1, b = B, y = Y, path condition: A<Y and A==B and A+1<Y and A+1>B
7: a = A+1, b = A+1, y = Y, path condition: A<Y and A==B and A+1<Y and A+1>B
3: a = A+1, b = A+1, y = Y, path condition: A<Y and A==B and A+1<Y
4: a = A+1, b = A+1, y = Y, path condition: A<Y and A==B and A+1<Y and not(A+1<A+1)
6: a = A+1, b = A+1, y = Y, path condition: A<Y and A==B and A+1<Y and A+1==A+1
8: a = A+2, b = A+1, y = Y, path condition: A<Y and A==B and A+1<Y
3: a = A+2, b = A+1, y = Y, path condition: A<Y and A==B and A+1<Y and A+2==Y

Other paths could be analysed but we have identified the three ones that pass through 7 and stay in the while for the smallest possible number of iterations.
Note: the text is not asking you to be exhaustive. Even a smaller but meaningful number of path analyses would be acceptable.

C) Test cases descend from the three path conditions, for instance:
a = 2, b = 1, y = 3
a = 1, b = 2, y = 4
a = 2, b = 2, y = 4

Questions 3, 4 and 5, Planning (5 points), Design (5 points) and Testing (5 points)
Design a simple project management tool for a company. The application should:

3. Handle information about projects (name, purpose, budget, tasks) and people (name, skills, employee number, role, department, allocated projects, allocated tasks).
4. Enable people to log-in/out.
5. Allow people with role “project manager” to allocate other people to projects and tasks.
6. Allow people to visualize the tasks they are working on and input the level of accomplishment of these tasks.
7. Compute the set of people allocated to a specific project/task.
8. Support people in requesting help across the company based on skills (the user selects a set of skills and the system suggests people with these skills).

A) Estimate the size of the project defined above, using Function Points and assuming that the application is going to be developed in JAVA (1 FP = 53 SLOC). To apply Function Points you can use the table below:

Function Types	Weights		
	Simple	Medium	Complex
N. Inputs	3	4	6
N.Outputs	4	5	7
N. Inquiry	3	4	6
N. ILF	7	10	15
N. ELF	5	7	10

B) Define:

- The Actors involved in the application
- The Use Cases describing the main functionality of the application: it is not necessary to detail all the use cases. Rather, draw them in one or more use case diagrams and provide the details of the most important one.
- A diagram highlighting the key components of the system architecture and the way they are connected. Explain the role of each component.

C) Identify possible acceptance test cases for the system focusing on functional aspects. While defining these cases specify the preconditions (if any) that should be true before executing the tests, the inputs provided during the test, and the expected outputs. Finally, explain why you think the selected tests are important for the specific system being considered.

Solution (Part A)

Our problem is the following:

Design a simple project management tool for a company. The application should:

1. *Handle information about projects (name, purpose, budget, tasks) and people (name, skills, employee number, role, department, allocated projects, allocated tasks).*
2. *Enable people to log-in/out.*
3. *Allow people with role "project manager" to allocate other people to projects and tasks.*
4. *Allow people to visualize the tasks they are working on and input the level of accomplishment of these tasks.*
5. *Compute the set of people allocated to a specific project/task.*
6. *Support people in requesting help across the company based on skills (the user selects a set of skills and the system suggests people with these skills).*

ILFs: (people, projects, tasks and skills. We can assume that all have simple structure as they are basic data structure containing a relatively small number of instances) 4*7 = **28**
ELFs: (we assume that this software is a standalone one. If we had assumed that this was connected, for instance, with the human resources software for managing exployee, we would have include here at least 1 ELF) **0**
EIs:

- Login – simple
- Logout – simple
- Project creation – simple
- Task creation and allocation to project – medium as it requires the usage of 2 entities
- Input level of accomplishment of a task – simple
- People assignment to task – medium as it requires the usage of at least 2 entities

3*4 + 4*2 = **20**
EIQs:

- Help request – medium as it needs to analyse a potentially large number of people from the whole company
- Get the set of people allocated to a specific project/task - simple
- Visualize user's tasks - simple

1*4+2*3 = **10**
EOs: 0 (application does not allow output)
TOTAL: **28+20+10 = 58**

Assuming 53 LOC per each FP we have 58*53= 3074 SLOC

Solution (Part B)

B.1)
Actors Involved are:

Project Manager (PM), responsible for project and task artifacts with all connected relations
Person/Developer, responsible for his/her own tasks and their progress status
Helper (anyone), responsible for providing help based on owned skills/expertise

B.2)
Use-Cases: we have one per each of the EI or EQ identified in the FP analysis

- Login

- Logout
- Create project
- Create task and allocate it to the project
- Input level of accomplishment of a task
- Assign people to task
- Request help
- Get the set of people allocated to a specific project/task
- Visualize user's tasks

The most important use case is probably "Assign people to task" as it requires specific attention not to overload people. So, during the assignment, the system will have to double check that the selected person can actually be allocated to the selected task given the availability of the person and the foreseen effort associated to the task. The description does not say anything about what the system should do in case the assignment is not possible. The option we choose to adopt in this solution is that if a task t requires x hours effort and person p can be available to the project for y hours with y<x in the timeframe of the task, then the person is allocated for y hours and the system informs the project manager that he/she needs to find another person to complete the allocation.

Use case name: Assign people to task
Participating actors: project manager, developer
Entry condition: the project manager has created a project and at least a task in the project
Flow of events

1. The project manager selects the project and the task to be allocated
2. The project manager selects the software engineer to be allocated to the task
3. The system performs the assignment and evaluates its compatibility. If the software engineer is available for less time than needed, the system informs the manager that the allocation to the task should be completed and the use case continues from point 2. Otherwise the system informs the manager than the allocation has been finalized.

Exit condition
The software engineer is allocated to the task.
Exceptions: if no software engineer is available for allocation, the use case terminates with a warning for the manager.

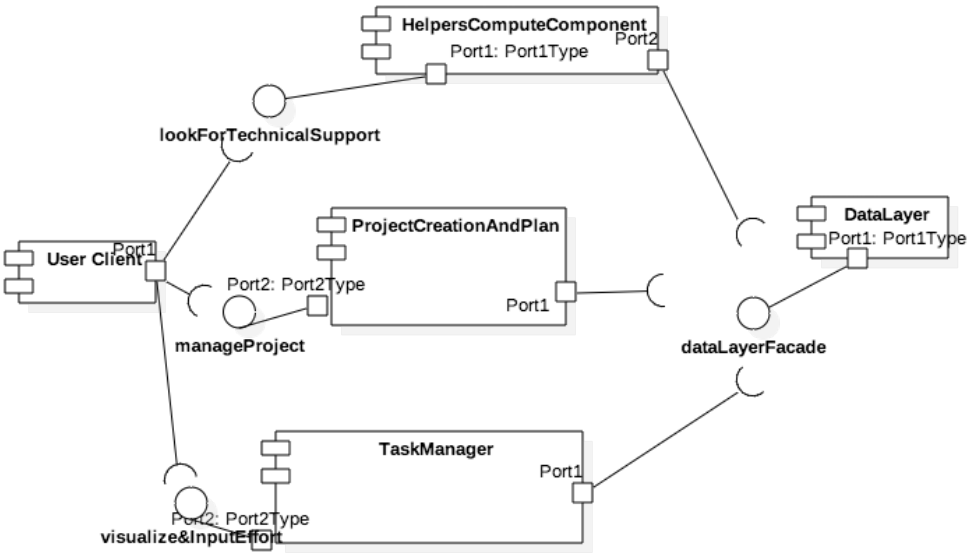
Arguably, the most important use-case is Look for technical support / help:

Use case name: Look for technical support
Participating actors: project manager or developer, helper
Entry condition: the project manager or a developer realizes that there is a need for help
Flow of events

- The developer polls the system to get suggestions on possible competent helping-hands
- The system evaluates tasks and completion of remaining company members to identify people who have the skills and some spare time to help.
- The system offers to the developer a list of people he/she could contact to get help.
- The system informs the helper that his/her name and skills have been communicated to a colleague.

Exit condition
The use case terminates when the system has computed the list of people. This list can be empty is none of the company members fulfills the requirements in terms of skills and free time.
Exceptions: none

B.3)
Architecture:



UserClient is used by all system users and provides interfaces on a role basis. Project managers will be able, through the features offered by the component ProjectCreationAndPlan, to create and manage projects, dividing work into tasks and allocating said tasks to other developers. Developers will be able to visualize their tasks and update the effort they spent on tasks. To enable these functions, the User Client will interact with TaskManager.
Finally, both project managers and developers will be able to exploit the feature offered by the HelpersComputeComponent to identify those people in the company that can provide some help to them. Since the text does not specify this further, for the sake of simplicity, we assume that the interaction with such people occurs outside the system.

Solution Part C

The exercise is about identifying possible acceptance test cases
- focusing on functional aspects.
- specify the preconditions (if any) that should be true before executing the tests,
- the inputs provided during the test,
- and the expected outputs.

Finally, to explain why you think the selected tests are important for the specific system being considered.

Given the most critical use-cases specified as part of exercise 3.B, it is reasonable to assume that those same use-cases should become scenarios to derive acceptance tests by means of black-box testing. This means that every of the following use-cases should be formulated into an acceptance test specification:

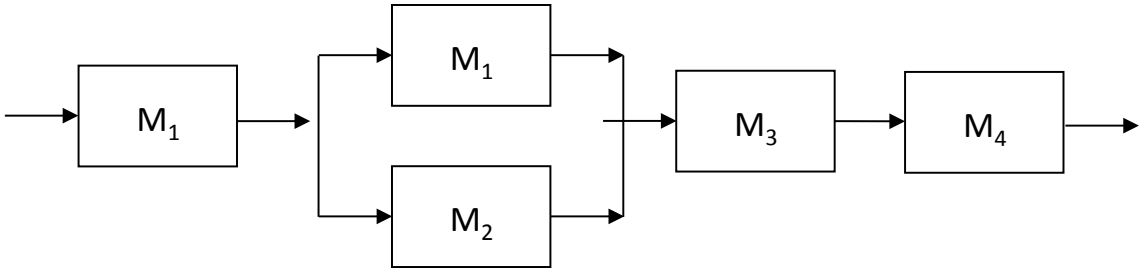
1. **Project staff allocation:**
    - PRECOND – at least one project is inserted and at least one task is ready to be allocated to at least one developer; also, at least one project manager is allocated to the project; finally, the project should be directly related to the project manager following an

"instantiation" relation, i.e. the project manager created the project himself and has "ownership" rights;
    - INPUTS – project manager inputs the name of the person or skill needed for the task and should be able to specify the allocation;
    - OUTPUTS – system returns new development network structure that takes the allocation into account;
2. **Task status visualization and update**
    - PRECOND: at least one developer is working on a task X to which he was allocated; system is currently showing old status or <no-status> for task X;
    - INPUTS: task X.ID is used to retrieve the status related to the task and the person responsible for it, if the person coincides with the requestor then modification should be permitted;
    - OUTPUTS: a new task status should be presented via the system;
3. **Login-logout**
    - PRECOND: system is live; security check component is live;
    - INPUTS: employee credentials;
    - OUTPUTS: visualization of employee tasks;
4. **Get the set of people allocated to a specific project/task**
    - PRECOND: there is at least one project with at least one task and at least one task allocation per tasks present;
    - INPUTS: a user recognized as "project manager" requires to visualize the set of people allocated to a specific project or a specific task;
    - OUTPUTS: graph representation of project/task allocation;
5. **Technical support / help**
    - PRECOND: task-allocations have been done, project is started (i.e., task status updates are present and progress can be computed)
    - INPUT: "need-help-on-task" request;
    - OUTPUT: list of possible candidates for lending a helping hand, matched based on skills and current availability;

These tests are critical since they ensure that key functionalities of the core business logic to be exhibited by the system are actually taken care of. User-acceptance tests performed around said functionalities will make sure that the system performs its intended function in the way the user expects it to.

Question 2 Architecture and availability (4 points) (13/02/2014)

Consider the architecture shown in the figure below.



Knowing that the availabilities of components are the following:

Component	Availability
M1	0.99
M2	0.98
M3	0.98
M4	0.97

1. Compute the availability of the proposed architecture. Given that you are not allowed to use computers and other devices, we do not expect to see precise calculations, but we would like to see the steps you follow in the calculation.
2. Given the following non-functional requirement: "the system must offer an availability equal or greater to 0.98, explain if the architecture as is allows us to fulfill the requirement and, if not, propose proper modifications. Motivate your answers.

Solution
Point 1
Let's call A_5 the availability of the parallel block composed of M_1 and M_2.
$A_5=1-(1-A_1)(1-A_2) = 1-(1-0.99)(1-0.98) = 1-0.01*0.02=1-1/100*2/100=1-2/10000=1-0.0002=0.9998$
$Afin=A_1*A_5*A_3*A_4=0.99*0.9998*0.98*0.97=0.9409$

Point 2
Even without performing the calculation it emerges that the overall availability is lower than 0.98 because of the presence of a weaker component.
In order to fulfill the requirements we can parallelize M_4. In this case, the availability of the last block will be $1-(1-A_4)^2=0.9991$, which leads to a total availability of $0.99*0.9998*0.98*0.9991=0.9691$.
To further increase the availability, we should focus on the next weak element in the chain, that is, M_3. We can parallelize it thus achieving $1-(1-A_3)^2= 1-(1-0.98)^2= 0.9996$. This allows us to achieve the availability: $0.99*0.9998*0.9996*0.9991=0.9885$.

Question 1 Alloy (8 points) 31/08/2017

A computer's memory can be modeled in Alloy based on the following signatures:

```
sig Address {}
sig Value {}
sig Computer {
    memory: Address -> lone Value
}
```

Using Alloy, you are to:

1) define a fact that states that the memory of the computer is never empty, that is, at least one memory address exists that is not uninitialized
2) model the behavior of predicate
   `delete [c: Computer, addr: Address, c': Computer]`
   such that memory at address `addr` becomes uninitialized
3) model the behavior of predicate

move `[c: Computer, oldAddr: Address, newAddr: Address, c': Computer]`
such that, if a value exists in address `oldAddr` and `newAddr` is uninitialized, the value is moved to `newAddr` and `oldAddr` becomes uninitialized; nothing changes if `oldAddr` is uninitialized

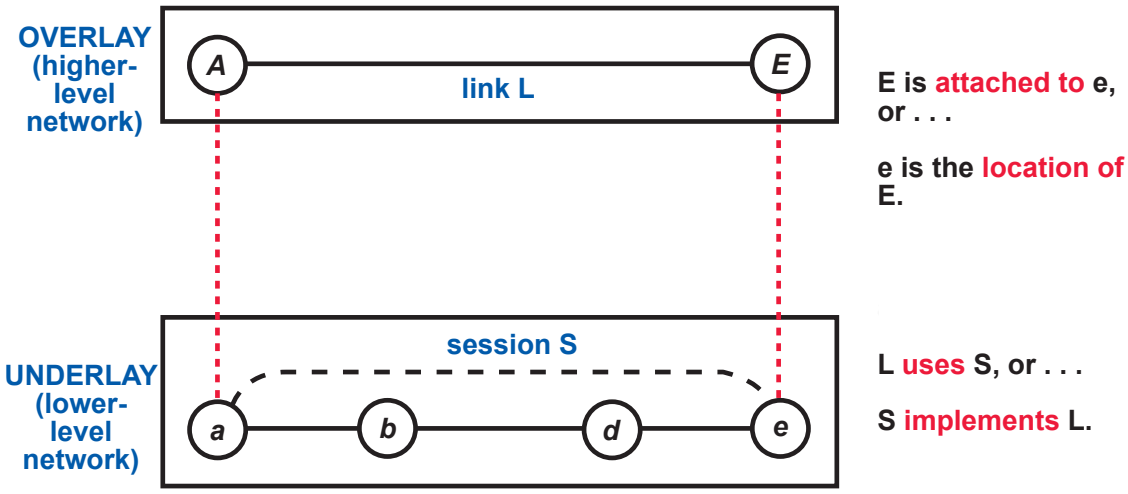
4) change the signatures above to extend the model to the case of a computer with multiple independent memory segments

Solution

```
sig Address {}
sig Value {}
sig Computer {
 memory: Address -> lone Value
}

fact notEmpty {
  all c: Computer | some addr: Address, val: Value | {addr->val} in c.memory
}

pred delete [c: Computer, addr: Address, c': Computer] {
  (some val: Value | val = addr.(c.memory)) implies
   c'.memory = c.memory - {addr -> Value}
}

pred move [c: Computer, oldAddr: Address, newAddr: Address, c': Computer] {
  (some val: Value | val = oldAddr.(c.memory)) and
  (no val: Value | val = newAddr.(c.memory))
  implies
  (some val: Value | val = oldAddr.(c.memory) and
                              c'.memory = c.memory - {oldAddr -> val} + {newAddr -> val})
}

pred show {}

//run show

run delete
```

Question 1 Alloy (7 points) 28/06/2017

Observe the following figure. It describes a communication system composed of an overlay network linking *nodes* A and E. This overlay is a virtual network built on top of an underlay network. In the figure, the underlay network is composed of four nodes (a, b, d, and e) and link L exploits the links between a, b, d and e in the underlay network to ensure that A and E can communicate.



Consider the following Alloy signatures:

```
sig Network {uses: lone Network} {this not in uses}

sig Node {
        belongsTo: Network,
        isLinkedTo: some Node,
        isAttachedTo: lone Node
} {this not in isAttachedTo and this not in isLinkedTo}
```

A) Explain the meaning of these signatures with respect to the figure above and indicate which elements in the figure are not explicitly modeled by the two signatures.

B) Write facts to model the following constraints:

- Linked nodes have to be in the same network
- A node belonging to a certain network can only be attached to nodes of the corresponding underlay network
- If a network is an overlay one, then there should not be nodes in this network that are not attached to other nodes
- A network should always contain some nodes

C) Write the predicate isReachable that, given a pair of Nodes, n1 and n2, is true if there exists a path that from n2 reaches n1, possibly passing through any intermediate node.

Solution to part B and C

```
fact linkedNodesInTheSameNetwork {
        all disj n1, n2: Node | n1 in n2.isLinkedTo implies #(n1.belongsTo & n2.belongsTo) > 0
}

fact isAttachedToInConnectedNetworks {
        all disj n1, n2: Node | n1 in n2.isAttachedTo implies #(n1.belongsTo & n2.belongsTo.uses)
> 0
}
```

```
fact overlayNodeShouldBeAttached {
        all ntw: Network | (some ntw2: Network | ntw2 in ntw.uses) implies all n: Node |
n.belongsTo = ntw implies n.isAttachedTo != none
}

fact notEmptyNetwork {
        all ntw: Network | some n: Node | n.belongsTo = ntw
}

//n1 is reachable from n2
pred isReachable[n1: Node, n2: Node] {
                n1 in n2.^isLinkedTo
}
```

Question 3 Function Points and testing (6 points) 28/06/2017

A company managing gasoline stations aims at developing an information system.

This system visualizes the status of all gasoline stations in terms of quantity of gasoline available and sales. Such data are acquired from sensors located within the gasoline tanks and from the cash registers.

The system, also manages the process through which a customer refuels his/her car: the customer selects a pump and an amount to pay, inserts his/her credit card to enable the payment, the system after getting the confirmation of the transaction from the bank, enables the selected pump, the customer refuels the car and, optionally, inserts a fidelity card. In this case, the points corresponding to the gasoline purchase have to be added to this card.

Finally, the system alerts the supplying department about the need to buy new gasoline when it goes below a certain threshold.

A) Identify: **Internal Logic Files**, **External Inputs** and, if they exist in this case, **External Interface Files**. Define the complexity of each of them providing a motivation.

B) You are planning the system testing activities for this piece of software. Define the main test cases that you think are needed in this case. For each test case define the inputs, the expected system state before the execution of the test and the expected result.

Solution

A)
ILF
GasolineStation: Simple complexity since the structure of this data is simple and we can assume to have a limited number of gasoline stations to manage.
Tank: Simple complexity as the structure of this data is very simple
User (includes the fidelity card): Average complexity as the data structure is simple but we can have a relatively large number of users.

EIF

Streams generated by tank sensors: Simple complexity as it can be seen as a single data transmitted by the sensor when the gasoline goes below a certain threshold.
Data acquired by cash register: Average complexity as they
Confirmation of credit card transaction

External inputs
Refuel request by user

B)
Refuel request
Input: the customer selects the pump to use and the amount to pay
Expected system state: the pump is free and available
Expected result: the pump is reserved, the customer is asked to perform the payment

Finalize payment and refuel
Input: the customer inserts the credit card
Expected system state: the pump is reserved and the system knows the amount selected by the customer
Expected result: if the credit card is valid and the payment finalized, the system enables the pump to release the required amount of gas

Visualize the status of all gasoline stations
Input: the user selects the menu to visualize the status of the stations
Expected system state: before the execution: any
Expected result: a list of station with the corresponding gas level is shown

Alert the supplying department about the need to buy new gasoline
Input: a measurement of the gas level goes below a certain threshold
Expected system state: before the execution: any
Expected result: the supplying department is alerted

Question 3 Architectures and Testing (7 points) 13/02/2017

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.

Question 1 Alloy (7 points) 13/02/2017
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
```