Examination
Model Based Testing
DIT848 / DAT260

Software Engineering and Management
Chalmers | University of Gothenburg

Saturday June 1st, 2013

Time	08:30-12:30
Location	Lindholmen
Responsible teacher	Gerardo Schneider
Phone	0700 44 18 33
Tasks	**5** (20 pts each)
Total number of pages	**10** (including this page)
Max score	100 pts
Grade limits	3 (G): at least 50 pts (see more details below) 4: at least 65 pts (see more details below) 5 (VG): at least 80 pts (see more details below)

ALLOWED AID:

- Books on testing
- All lecture notes (including printouts of lectures' slides)
- Students own notes
- English dictionary
- **NOT ALLOWED:** Any form of electronic device (dictionaries, agendas, computers, mobile phones, etc)

PLEASE OBSERVE THE FOLLOWING:

- Motivate your answers (a simple statement of facts not answering the question is considered to be invalid);
- Start each task on a new paper;
- Sort the tasks in order before handing them in;
- Write your student code on each page and put the number of the task on **every** paper;
- Read carefully the section below "ABOUT THE FORMAT OF THE EXAM".

ABOUT THE FORMAT OF THE EXAM:

The exam consists of 5 tasks, each one concerned with a specific part of the course content.
Each task is worth 20 points. In order to reach the level to pass with **3** (**G**) you need **at least 50 points** out of the total, and **at least 6 points per task.** To pass with **4** you need **at least 65 points** out of the total, and **at least 8 points per task**.

In order to pass with distinction (**5**/**VG**) you need to reach **at least 80 points** out of the total, and you must score **at least 14 points per task**.

IMPORTANT: Note that you should have a minimum number of points per task in order to pass, so avoid letting unanswered tasks.

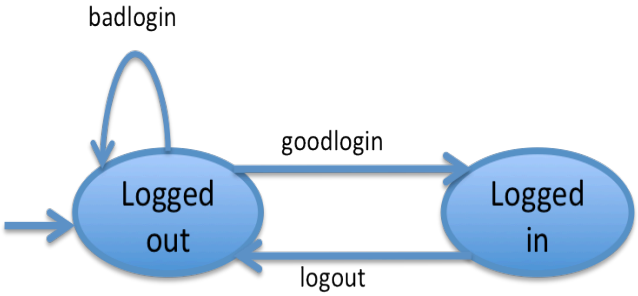
Task 1 - Test in general

1) Below you will find 10 statements about different issues related to testing. **Determine whether the statements are true or false**. **En each case justify your answer giving clear arguments to defend your judgment (if the answer is false provide the correct fact; if it is true briefly explain why).** Note that some of the statements are concerned with hypothetical situations where a failure is said to be found by a specific testing technique (e.g., unit or integration testing); in those cases a "true" answer is the one corresponding to the case when the error can be detected with the given technique in a first instance (and "false" otherwise). Your answer will not be considered complete if you do not justify it. **(20 pts - 2 pts each)**

1) The criteria to stop testing is that all the test cases pass.
2) Test-Driven Development is a conceptual framework used to test drivers.
3) Code inspection is as a testing technique useful for testing usability.
4) We can use unit test for detecting whether a variable is used many times for different purposes in the same program.
5) The best way to test whether an application behaves as expected when executed in a particular platform (e.g., Linux) is by using some kind of system test.
6) The module that computes a specific functionality of an application has been tested using unit test and white box testing and seems correct. However, some unexpected result is found when the subsystem containing the module and a mock module interacting with the former is tested. This error was found during integration test.
7) For some unknown reason an Internet application becomes very slow after a user logs in and logs out successively more than 10 times on a row. This problem is found during system test when performing stress testing.
8) Test-Driven Development (TDD) is a new trend on testing focusing on the test activity and not on the programming activity. The rationale behind this approach is that code can always be automatically generated from the test cases and thus you get already tested code for free.
9) An application developed for smart phones has been tested in different phone models carefully chosen to test whether the application runs in different platforms. It was realized that for certain models of phones the application crashed almost all the time for unknown reasons. This was discovered when performing integration test.
10) Accidentally, an assignment inside a loop was deleted by the programmer. This assignment updated the control variable of the loop condition, so the variable remained with the same value as when the loop was entered the first time, making impossible to quit the loop. This was not detected when performing unit test and checking for statement coverage.

Task 2 - State machines and black box testing

A programmer needs to write a software for a login system. As expected the specification starts by saying that a user may log in into the system only when the user enters the right username and password, and otherwise the user is not allowed to log in. The specification continues with more complex constraints like that it should be possible to allow more than one user at a time (but with a bound on the number of users who can be logged in simultaneously), and that there are restrictions on the number of failed attempts per user to login. Before making a full implementation of the system the programmer decides to write a first abstract model of his program, in the form of a Finite State Machine (FSM). The model is depicted in the picture in the right where the user can login if the information is correct (modeled with the action 'goodlogin') and it is not allowed to login provided the information is incorrect (action 'badlogin'). At any moment the user can log out (action 'logout').



1) You are in charge now of writing an Extended Finite State Machine (EFSM) for the full login system. As a first step towards that you start by writing simple machines that will be further refined towards the final EFSM. In this exercise you are requested to **write an EFSM for a login system only modeling that a user is allowed to try maximum 5 bad logins (by entering username and password). After the 5th attempt the machine should be blocked and no other login will be possible. (6 pts)**

2) In what follows there is a list of informal descriptions of test cases the programmer would like to extract from the EFSM you have developed to solve exercise 1 above, having in mind the full implementation of a login system. **State whether the test cases could be extracted or not from such an EFSM. The programmer only wants to write "good" test cases, that means that if the test case doesn't pass then it is because there is a failure. If a test case is not "good" then state that. Write YES, NO, or BAD for each one of the 3 possible cases above. In each case briefly justify your answer.** (Note that your answer will not be considered completely correct if you do not justify it).
(4 pts – 1 pt each)

   a. After trying 5 non-successful attempts to log in with a specific username, the same user can try to log in again with a different username.
   b. After having logout a user can log in again.
   c. If a user is logged in no other user can log in.
   d. A logged in user is automatically logged out after 30 min.

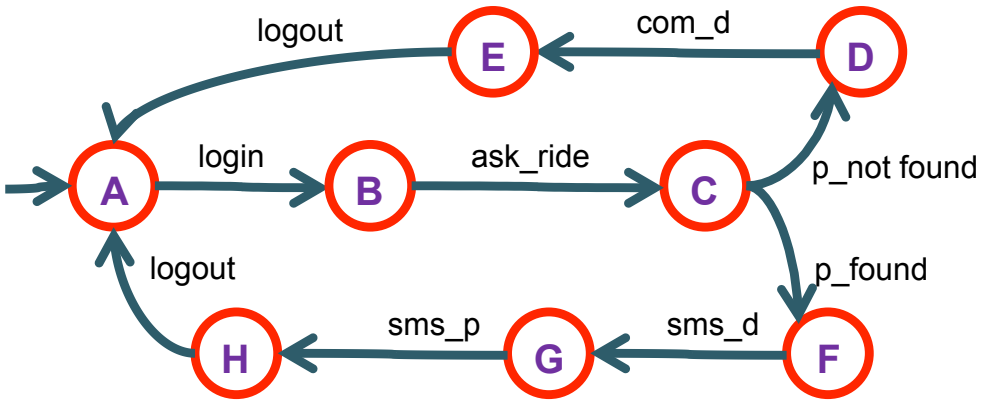
3) **Give an EFSM that models a more complex login system with the following specification.** Each user is allowed to have maximum 5 bad logins, after which the user is blocked for 10 hours. The user is allowed to try to log in again after that period of time. Besides, the login system should allow maximum 10 users to be logged in simultaneously. If a new user tries to log in when there already are 10 users logged in, then the system gives a

message saying that the user will be allowed to log in as soon as some of the logged in users logs out. That is, there will be a queue of 'pending users' waiting that some user log out. Each 'pending user' will be allowed to stay in the queue for a maximum period of 30 min, after that the user is removed from the queue and is informed that he should try to log in again later. Finally, any logged in user without activity for more than 20 min is automatically logged out. **(10 pts)**

Note: Draw new machines for each exercise separately. Be sure you provide meaningful names for of each action, variable, state, etc., and provide a short explanation of each in case of ambiguity.

Task 3 - White box testing, coverage analysis

Let the following FSM represent the model of a SUT:



NOTE: **A** is the initial and the final state.

`1)` In what follows there are 7 statements on different *coverage criteria* issues or situations related to the above FSM. **Determine whether the statements are true or false**. **In each case justify your answer** giving clear arguments to defend your judgment. **In case of a false answer give the sequence of actions to be performed to satisfy the corresponding criteria**. **In case of a true answer, briefly explain why it is the case.** Note that your answer will not be considered complete if you don't justify it. **(14 pts – 2 pts each)**

`a)` The following test case achieves a full coverage according to the ***All-states*** *transition-based coverage criteria*:

- login, ask_ride, p_not_found, com_d, logout, p_found, sms_d, sms_p. logout

`b)` The following test cases achieve full coverage according to the ***All-one-loop-paths*** *transition-based coverage criteria*:

- login, ask_ride, p_not_found, com_d, logout;
- login, ask_ride, p_found, sms_d, sms_p, logout

`c)` The test cases performed for the ***All-one-loop-paths*** above (cf. exercise b) also achieve 100% coverage according to the ***All-transitions*** *transition-based coverage criteria.*

`d)` If the FSM is modified in such a way that two loops are added, one in state D (labeled *wait_d*) and the other in F (labeled *wait_f*), the following test case would achieve full coverage according to the ***All-one-loop-paths*** *transition-based coverage criteria*:

- login, ask_ride, p_not_found, wait_d com_d, logout, login, ask_ride, p_found, wait_f, sms_d, sms_p, logout

`e)` The *control-flow oriented* ***Decision/Condition Coverage*** *(D/CC) criteria* is not applicable to the FSM above.

`f)` It doesn't make sense to apply data-coverage criteria in the above FSM as FSMs are control-oriented and there is no data.

g) If the arrows of the transitions labeled with login and ask_ride are reversed, then all the states are unreachable and thus the application of any *transition-based coverage criteria* would not give any meaningful test case (for the test case where no action is executed).

2) For the FSM of the figure provide a solution for the problem of *testing combination of actions (transitions) of length 2* (Hint: use de Bruijn sequences) **(6 pts)**

1) Task 4 – MBT / ModelJUnit

You will find below 10 statements and situations about different issues related to model-based testing (MBT) and ModelJUnit. **Determine whether the statements are true or false**. **In each case justify your answer** giving clear arguments to defend your judgment **(if the answer is false provide the correct fact, if it is true write a short reason showing you understand why it is the case).** Note that your answer will not be considered complete if you don't justify it.
(20 pts – 2 pts each)

1) Though in theory it is established that an adapter is written as a wrapper around the implementation serving as interface to the model, in ModelJUnit this is not necessarily true and the adapter may be written together with the model.
2) It is not possible to have an automatic generation (and execution) of test cases from a model to perform online testing unless the test cases are transformed into concrete ones.
3) When a tester using MBT was asked why his models made references to variables in the code, he answered that it was common practice in MBT as it is good to test the values of variables of a program. The answer of the tester is correct (answer according to your understanding of MBT based on the definition, underlying principles of the technique, etc.).
4) Assume you are writing an EFSM as a model to be used to extract test cases for a calculator implementation. You don't have access to the implementation and only know the interface. Your model should definitively generate test cases to check that the operands and operators are stored in the right order in the calculator's stack.
5) Assume you are writing an EFSM as a model to be used to extract test cases for the implementation of a stack to be used as an external module to be called by a calculator. You don't have access to the implementation and only know the interface. Your model should definitively generate test cases to check that the operands and operators are stored in the right order in the calculator's stack.
6) If the test cases automatically extracted from your model when using MBT achieve 100% statement coverage at the code level, then you can claim that you have also achieved 100% state coverage in your model.
7) FSM is a transition-based notation and it is not possible to be used to test data-oriented systems.
8) EFSM is a transition-based notation and it is not possible to be used to test data-oriented systems.
9) ModelJUnit allows to represent EFSMs by explicitly describing all the states and all the transitions between the states. It is thus not possible to generate EFSMs with unbounded or infinite behavior (that is, all test cases generated are finite).
10) In ModelJUnit it is good to have transitions without labels (actions) as this allows the generator of test cases to extract many different tests for those transitions without labels.

Task 5 – Property-based testing and QuickCheck

1) Assume that you have implemented a Module `Stackl` that includes an implementation of stacks in Haskell with some additional operations besides the standard ones. The module introduces a parameterized data type, `Stackl`, such that for every type `a`, we have the type `Stackl a` of stacks of `a`s, e.g. a `Stackl Int` is a stack holding integers. This stack implementation is pure, i.e. there are no side effects. As a consequence, the functions that manipulate stacks always return new stacks as their result, instead of modifying stacks in place. The interface for this module includes the following (standard) functions:

```
push :: a -> Stack a -> Stack a
pop :: Stack a -> Stack a
top :: Stack a -> a
isEmpty :: Stack a -> Bool
empty :: Stack a
```

The function `push` takes an element `x` and a stack `s`, and produces a new stack with `x` as its topmost element, followed by the elements of `s`. The constant `empty` represents the empty stack, while the function `isEmpty` simply checks whether its argument is the empty stack or not. The functions `pop` and `top` are used to take a stack apart: `pop` pops off an element from its argument, returning the resulting stack, and `top` returns the topmost element in its argument stack (without changing the stack). Their behaviour is undefined when applied to the empty stack.

Besides these standard operations, the module contains the following additional functions:

```
pushl :: Stack a -> [a] -> Stack a
stack2list :: Stack a -> [a]
```

The function `pushl` takes a stack `s` a list of elements `l` of type `a` and returns the stack where all the elements in the list `l` has been pushed on top of the stack `s` (the head of the list is pushed first and then the operation proceed recursively to push the rest of the list). The operation `stack2list` takes a stack `s` and returns the list of all the elements in the stack, inserting each element taken from the stack in the head of the list (the top of the stack will be the last element of the list).

You can assume that besides the standard operations on lists you also have available the following functions on lists:

```
sorted :: [a] -> Bool
maximum :: [a] -> a
size :: [a] -> Int
++ :: [a] -> [a] -> [a]
```

`sorted` returns true if a list is sorted, `maximum` returns the maximum element of a list, and `size` returns the size of the list. Finally, `++` concatenates two lists where the new list is composed of the first followed by the second one. (Though not explicitly written in the types, it is assumed that `sorted` and `maximum` operates on types where an order is defined.)

Provide a solution to the following questions/situations regarding QuickCheck properties for the above module. **(12 pts – 3 pts each)**

a. A programmer wants to check that the pushl operation works well when pushing the concatenation of two lists on a stack, and writes the following property:

```
prop_pushConcat s l1 l2 = pushl s (l1 ++ l2) == l1 ++ (pushl s l2)
```

Is the property correct? If not say why and provide a correct property.

b. Write a property about the top element of a stack where the latter has been created by pushing a sorted list into an existing stack.

c. A programmer would like to write a property to check that the result of taking the list representation of a stack after having pushing a list, is what is expected. He is not sure how to write the complete property and get stuck after writing the following:

```
prop_listRep s l = not (isEmpty s) ==> stack2list (pushl s l) == …
```

Complete the above property so the equality holds.

d. A programmer wants to write a property about the size of a list that has been obtained from a stack after pushing a list into it (that is, about the following: size (stack2list (pushl s l)). He writes the following property:

```
prop_size s l = size (stack2list (pushl s l)) == size (l + s)
```

Is the property correct? If not say what is wrong and give a correct property.

2) Write a generator that generates non-empty lists of integers of arbitrary size satisfying the following constraints:

1. The list is sorted.
2. The first element of the list is a random number between 1 and 100.
3. Each element of the list is randomly generated in such a way that the element is bigger than the previous one and it differs at most in 100 from the previous one.

That is, if $[a_1, a_2, \ldots, a_n]$ is a list generated according to the above specification, then it should satisfy that:

$0 < a_1 <= 100$, and
$0 < a_{i+1} - a_i <= 100$ (for $0<i<n-1$)

The following is a valid example of a generated list: [87, 122, 123, 222]. On the other hand, the lists [2, 104, 105, 200] , [105, 106, 110, 201] and [77, 56, 139, 150] are not valid. **(8 pts)**