Examination (SOLUTION!!)
Model Based Testing
DIT848 / DAT260

Software Engineering and Management
Chalmers | University of Gothenburg

Monday August 26, 2013

Time	14:00-18:00
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) Acceptance test is always done in the real environment, by the user alone without interaction with the testers.
2) Regression test should be done immediately after finishing integration test, before any big modification of the code.
3) There are static and dynamic verification techniques. Static techniques are by definition not applied to the source code but to a model of the system.
4) When testing a software simulating a calculator (like the one developed during the assignments), a tester finds that the result of a partial computation of the addition operation was stored sometimes on a variable called *sum* and sometimes on another called *add*. This was found when applying unit test.
5) It is not possible to apply white box testing to systems developed in functional programming languages (e.g., written in Haskell) due to the use of recursion.
6) You can only test programs written in functional programming using QuickCheck (or similar property-based testing tools).
7) For some unknown reasons an Internet application becomes very slow after a user logs in and logs out successively more than 15 times without any further activity in between. This is found when performing system test.
8) 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 detected when performing unit test and checking for statement coverage.
9) In order to test software usability you need to have a full implementation of the system or at least an implementation of the user interfaces (eventually having mock modules to implement the missing functionalities).
10) The V model is a graphical representation of the relationship between development and testing, and it does not explicitly suggests that testing should be performed in a waterfall manner.

SOLUTION)

1. F – It might be the case but not necessary; the testers might be part of it
2. F – it's done after making changes in the code
3. F- they can be applied to the source code too
4. F – this can be found using code inspection not unit test
5. F – it is possible; recursion is difficult to test but not impossible
6. F – you can test Haskell programs using any technique
7. T – stress testing finds this kind of errors
8. T – you can detect this when using both techniques
9. T – you need the interface to be ready
10. T – the model is generic and not specific to a particular development model

Task 2 - State machines

A *turnstile*, a device used to control access to places like subways, is a gate with three rotating arms at waist height, one across the entryway (see picture in the right). Initially the arms are locked, barring the entry, preventing customers from passing through. Depositing a token in a slot on the turnstile unlocks the arms, allowing a single customer to push through. After the customer passes through, the arms are locked again until another token is inserted.
The turnstile could be modeled as a simple Finite-State Machine (FSM) accepting inputs that affect its states: putting a token in the slot (*token*) and pushing the arm (*push*). In the locked state, pushing on the arm has no effect; no matter how many times the input *push* is given it stays in the locked state. Putting a token in, that is giving the machine a *token* input, shifts the state from *Locked* to *Unlocked*. In the unlocked state, putting additional tokens in has no effect; that is, giving additional *token* inputs does not change the state. However, a customer pushing through the arms, giving a *push* input, shifts the state back to *Locked*.

1) **Write a FSM representing the behavior of a turnstile** as described above. **(6 pts)**

2) **Give an EFSM that models a turnstile with the following (more complex) specification.** Assume now that the turnstile, besides accepting a *token*, also accepts normal coins (with values 1, 5 and 10 SEK) and credit cards. The price is 18 SEK. When paying with tokens the turnstile behaves as before.

   When paying with coins the customer should add enough coins to cover the price to unlock the arm. The turnstile can give back money up to 2 SEK (that is, if more than 20 SEK are inserted into the machine then only 2 SEK are given back, the rest remains inside the machine). This complex turnstile has a timeout: if not enough money has been inserted within 20 seconds then the machine gives the money back. Besides, there is an additional button *return* that when pushed allows the user to get all the inserted money back, provided this is done before the required amount is inserted (at least 18 SEK) and before the timeout. In case of timeout or when pressing the *return* button the arm remains locked.

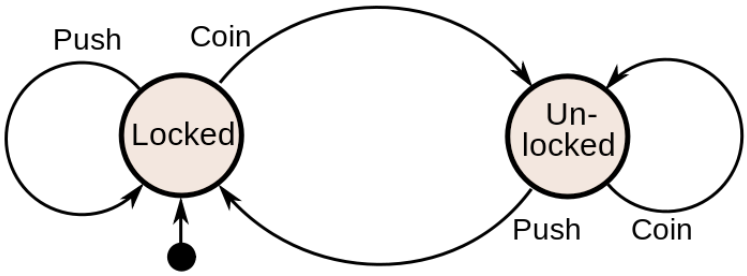
   When paying with credit card, the machine behaves as expected, that is: 1) The customer must enter the credit card; 2) The machine ask for the pin; 3) If the pin is correct the machine beeps and ask the user to get the card, and after getting the card the arm is unlocked and the user can proceed; 4) If the pin is incorrect the machine gives the message “wrong pin” allowing the user to enter the pin 2 more times, and act as in 3) above in case the pin is correct or gives the card back otherwise. In case of payment with credit card there is a timeout of 30 seconds after which the card is automatically returned to the user (that is a correct pin should be given within 30 seconds after inserting the card).

   After successful payment and when the user push the arm to pass, the machine returns to the locked state; similarly in case of timeout and after giving money back when pressing the *return* button. The return button has no effect if pressed when paying with tokens or credit card. **(14 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.

SOLUTION)

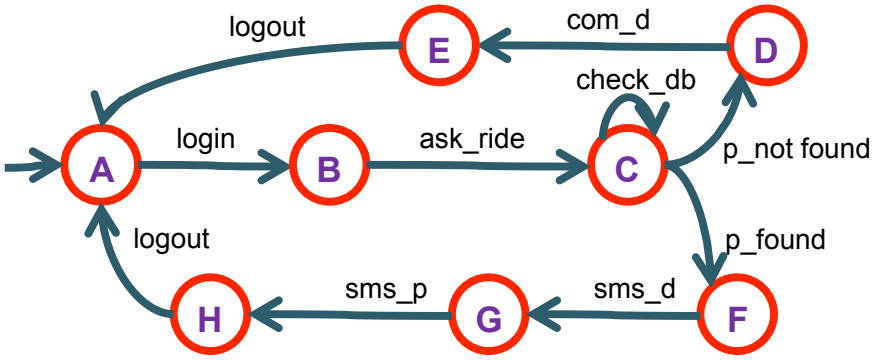
1)



2) CONTACT THE TEACHER TO DISCUSS YOUR SOLUTION

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.

In what follows there are 10 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. **(20 pts – 2 pts each)**

1) The following test cases achieve 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

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

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

3) 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.*

4) 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, check_db, p_not_found, wait_d, com_d, logout, login, ask_ride, p_found, wait_f, sms_d, sms_p, logout

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

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

7) 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).

8) When seeing the FSM as a digraph we can claim that it is Eulerized.

9) Assuming each action in the FSM takes 1 time unit, the minimal time to test it in parallel would be 6 time units, using 2 machines.

10) In order to test combination of actions of length 2 in the FSM we can apply the de Bruijn algorithm. In order to do so we would first need to create the dual graph which will contain 10 states labeled as follows: *login, ask_ride, p_not_found, com_d, logout, p_found, sms_d*, *logout, check_ db,* and *sms_p.*

SOLUTION)

1) F – the test case is ill-formed as the second test case cannot start with p_found.
2) F – it should also include the test cases
   - login, ask_ride, p_not_found, com_d, logout;
   - login, ask_ride, check_db, p_found, sms_d, sms_p, logout
3) T – it does
4) F - it does repeat the initial state 3 times (the test should be split into 3 parts, adding two more tests to go through the 2 loops):
   login -> ask_ride -> p_not found -> com_d -> logout
   login -> ask_ride -> check_db -> p_not found -> com_d -> logout
   login -> ask_ride -> p_not found -> wait_d -> com_d -> logout
   login -> ask_ride -> p_found -> sms_d -> sms_p -> logout
   login -> ask_ride -> check_db -> p_found -> sms_d -> sms_p -> logout
   login -> ask_ride -> p_found -> wait_f -> sms_d -> sms_p -> logout
5) T – it doesn't make sense as there are no conditions
6) T
7) T – trivial
8) F – it would need to add repetition of the edges login and ask_ride.
9) T
10) T

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) In MBT, unless test cases are transformed into concrete ones it is not possible to have an automatic generation (and execution) of test cases from a model to perform online testing.
2) When using transition-based models in MBT, it is advisable that all transitions have labels (actions).
3) It is not possible to get online (automatic) test extraction from a model of a telephone vocal service (like the Qui-Donc example seen in the lectures) using ModelJUnit unless you write an adapter.
4) In ModelJUnit it is possible to implement an adapter either by modifying the EFSM model or by writing a separate module that interacts with the model and the SUT.
5) ModelJUnit allows to represent EFSMs by explicitly describing all the states and all the transitions between the states. It is possible to generate EFSMs with unbounded or infinite behavior (that is, to generate test cases that might not terminate).
6) Assume you are writing an EFSM as a model to be used to extract test cases for the implementation of a queue to be used as an external module to be called by a software implementing a ticket system. You don't have access to the implementation and only know the interface. Your model should definitively generate test cases to check that the queue behaves as expected (e.g., that the first in the queue is indeed served first).
7) 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.).
8) EFSM is a transition-based notation and it is not possible to be used to test data-oriented systems.
9) In MBT you can claim that you have achieved 100% state coverage in your model if the test cases automatically extracted from your model achieve 100% statement coverage at the code level.
10) When using ModelJUnit it is not possible to write different models (at different abstraction levels) as this will be confusing for the tool.

SOLUTION:

1. T – you need to have concrete tests to perform online testing
2. T – without labels the system is non-deterministic so not good for automatic test extraction
3. T – you need an adaptor to generate online testing
4. T – both are possible
5. T – it is possible to write infinite test cases
6. T – You need to test that the queue behaves correctly
7. F - You don't test for values of variables, and it is not common practice as you are writing a model and not putting program variables in your model
8. F – you can have data in EFSM
9. F – you cannot make such kind of claim in general as it is not the case
10. F – you can always define different abstraction levels; it is in fact advisable to do so

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` and 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
null :: [a] -> Bool
maximum :: [a] -> a
length :: [a] -> Int
++ :: [a] -> [a] -> [a]
sort :: [a] -> [a]
```

`sorted` returns true if a list is sorted (in increasing order), `null` returns true if the list has no elements, `maximum` returns the maximum element of a list, and `length` returns the size of the list. `sort` returns the same list given as parameters but sorted in increasing order, and 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`, `sort` 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 checking the top element of the concatenation of two non-empty lists after being sorted, and writes the following property:

```
prop_top_sort s l1 l2 =
                    not (null l2) ==>  top (pushl s (sort (l1 ++ l2))) == maximum(l2)
```

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

b. Complete the property below, such that it expresses what happens when pushing two non-empty lists into a stack (hint: use the concatenation operation):

```
prop_push_two_lists s l1 l2 = (pushl (pushl s l1) l2 ) == ...
```

c. Write a property prop_top_sorted s specifying what the top element of the stack s is when s has been created by inserting a non-empty sorted list into it, i.e. what should top s be equal to under the precondition not (isEmpty s) && sorted (stack2list s).

```
prop_top_sorted s = …
```

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: length (stack2list (pushl s l)). He writes the following property:

```
prop_length s l = length (stack2list (pushl s l)) == length (l + s)
```

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

2) A tester has to write an implementation of 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]. On the other hand, the lists [2, 104], [105, 106, 110] and [77, 56, 139, 150] are not valid.

The code below is supposed to be an implementation of the generator described above:

```
import Test.QuickCheck
import Data.List
genListSorted :: Gen Int
genListSorted = do
        intlist <- listOf1 ( elements [1..100] )
        return ( tail ( map product ( inits intlist ) ) )

main = sample genListSorted
```

The above generator is not completely correct. Explain what are the errors and modify the code above so it is correct. (8 pts)

NOTE: In case you are not familiar with some of the Haskell functions used in the code above, this is a description:

`listOf1 :: Gen a -> Gen [a]` - Generates a non-empty list of random length.

`elements :: [a] -> Gen a` - Generates one of the given values. The input list must be non-empty.

`inits :: [a] -> [[a]]` - The inits function returns all initial segments of the argument, shortest first. For example, `inits "abc" == ["","a","ab","abc"]`

`product :: Num a => [a] -> a` - The product function computes the product of a finite list of numbers.

`tail :: [a] -> [a]` - Extracts the elements after the head of a list, which must be non-empty.

`map :: (a -> b) -> [a] -> [b]` - `map f xs` is the list obtained by applying f to each element of xs, i.e.,

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

SOLUTION)

1. Solution
   a. Problem: the top stack will be the maximum of two lists and not just the second list; also a precondition is missing concerning l1.

      ```
      prop_top_sort s l1 l2  = (not (null l1) && not (null l2)) ==>   top (pushl s (sort(l1 ++ l2))) == maximum(l1++l2)
      ```

   b.
      ```
      prop_push_two_lists s l1 l2  =  (pushl (pushl s l1) l2 ) ==  pushl s (l1++l2)
      ```

   c.
      ```
      prop_top_sorted s  =
         not (isEmpty s) && sorted (stack2list s) ==> top s == maximum (stack2list s )
      ```

   d. No, this is a correct property:

      ```
      length (stack2list (pushl s l)) == length (stack2list s) + length l
      ```

2.

```
import Test.QuickCheck
import Data.List

genListSorted :: Gen [Int]
genListSorted = do
   intlist <- listOf1 ( elements [1..100] )
   return ( tail ( map sum ( inits intlist ) ) )

main = sample genListSorted
```