Examination
Model Based Testing
DIT848 / DAT260

Software Engineering
Chalmers | University of Gothenburg

Thursday August 30, 2012

Time	08:30-12:30
Location	Lindholmen
Responsible teacher	Gerardo Schneider
Phone (contact)	070-712 83 52 (Wolfgang Ahrendt)
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 (G): 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 leaving tasks unanswered.

Task 1 - Test in general

A software company uses Java for development, and performs tests as specified in what follows (for this question the order in which the tests are presented is not relevant, only the activity performed):

1. Individual developer writes and runs unit tests (in JUnit)
2. Individual developer does coverage analysis
3. A team does code review
4. A team does integration testing
5. A team does regression testing
6. Testers do system tests
7. Customer does acceptance tests

Below follows a list of problems that may occur at different stages in the project. For each of the problems, identify the test in the list above where the problem is most likely to be found, or if it cannot be detected by any of the above tests specify whether another verification technique could find it. In each case justify your answer. **(2 pts each)**

a) A variable is used many times for different purposes in the same program, making it difficult to understand the code, though the program works according to the specification.
b) The condition in a while-statement always evaluates to false, so the body of the while loop is never executed.
c) An assignment inside a loop was accidentally deleted by the programmer. This assignment updated the control variable of the loop condition, so the variable remains with the same value as when the loop is entered the first time, making it impossible to quit the loop.
d) The application becomes very slow when the number of simultaneous transactions is bigger than 55, eventually producing the whole system to stop.
e) One programmer forgot to update the result variable of a method so it always return the same value.
f) A security expert of the company (not a developer) found a security problem on the application related to a leak of information.
g) A method happens never to be called.
h) The customer clicked on a certain button when running the application expecting to get back to a page containing certain information, but it did not get what she expected. This was not mentioned in the specification.
i) Some changes have been made in a class, after it has been thoroughly tested, and the programmer thinks they may affect other parts of the program.
j) The application does not behave as expected when executed in Linux.

Task 2 – Models (State machines)

1) This task is concerned with a wireless communicating system between a car onboard computer and a smart phone. The smart phone is used to open and close the car, and also to upload statistical information from the car computer, automatically transferred to the mobile phone when this is detected on a given radius of proximity (after having opened the car with the phone). Due to security concerns this transmission is not done to any mobile phone in the range of the wireless communication, but only to the phones of the owners (which have been previously registered in a database in the car computer). It is assumed that the car computer already has those phones registered in the database.

Your task in this exercise is **to define a *Finite-State Machine* (*FSM*) for the car computer according to the following specification**: 1) The car computer keeps waiting on a standby state till a mobile phone is detected; 2) If the mobile phone is not in the database then it is ignored; 3) If the mobile phone is already registered (it is in the database) then a connection is established and a window asking for a code is shown in the screen of the phone; 4) If the correct code is entered then the car is opened; if not the car computer keeps waiting for the correct code; 5) After the car is opened (due to the input of the correct code) then certain predefined data is automatically uploaded into the mobile phone; 6) The transmission is then finished and the car computer goes back to standby. **(6 pts)**

2) Give 2 test cases that can be extracted from the above FSM, and two that cannot be extracted from it (in the latter case provide test cases that you think might be useful to further test your system, though they cannot be extracted from the FSM as it is too abstract). **(4 pts)**

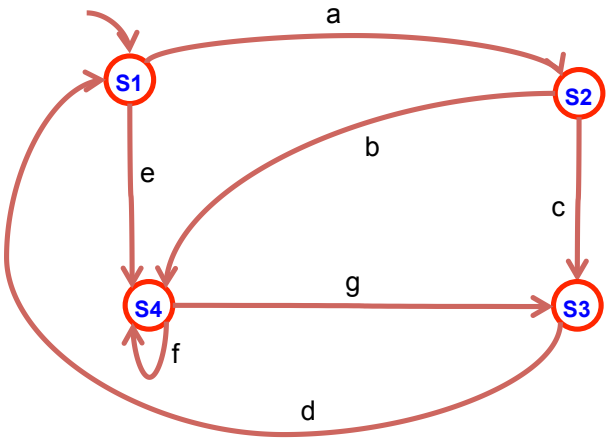
3) Draw an *Extended Finite-State Machine* (*EFSM*) for a modified extension of the system described in exercise 1) above.

The new description of the system is as follows: 1) The car computer keeps waiting on a stand-by state till a mobile phone is detected; 2) If the mobile phone is not in the database then it is ignored; 3) If the mobile phone is already registered (it is in the database) then a connection is established and a window asking for a code is shown in the screen of the phone; 4) The user of the phone can only have 3 failed attempts to enter a correct code; if the correct code is entered then the car is opened; after failing 3 times the phone is blocked; 5) After the car is opened (due to the input of the correct code on the phone, not when opened with a normal key) then certain predefined data is automatically uploaded into the mobile phone; 6) After the data is transmitted there is a timeout for the engine: it must start within 15 min, otherwise the car is automatically closed and the car computer goes to standby; 7) If the engine starts within 15 min then the car computer goes to standby. **(10 pts)**

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

Task 3 - White box testing, coverage analysis

Let the following FSM represent the model of a SUT:



Remark: **S1** is the initial state, and **S4** is the final state.

`1)` Below you can see different solutions to *transition-based structural model coverage* criteria for the FSM given above. These solutions are given as a sequence of actions to be performed to satisfy a given criteria:

`i.` s1: d-a, d-e
s2: a-b, a-c
s3: c-d, g-d
s4: e-g, e-f, b-g, b-f, f-f, f-g

`ii.` a-c-d-e

`iii.` a-b-g-d-e-f,
a-c-d-e

`iv.` e,
a-b,
a-c-d-e,
e-g-d-e,
a-b-g-d-e

`v.` As in `iv` above, plus adding all the paths also containing "f"

`vi.` e-g-d-e

Determine which one of the following criteria is being solved by each one of the above solutions (that is, you should associate to each one of the solutions described in `i-vi` above exactly one of the criteria `a)-h)`): **(12 pts – 2 pts each)**

`a)` All-states
`b)` All-transition-pairs (from each state)
`c)` All-loop-free-paths
`d)` All-one-loop-paths

e) All-round-trips
f) All-paths
g) All-transitions
h) None of the above

2) For the FSM given in the figure, **Determine whether the statements are true or false**. **Justify your answer** giving clear arguments to defend your judgment in case a statement is **false** in your opinion (state why the answer is false and provide the correct fact). **Note that your answer will not be considered complete if you don't provide the correct fact in case the statement is false. (8 pts – 2 pts each)**

a) The FSM is strongly connected, but not complete as not all actions are possible from every state.
b) A possible solution for the *Street Sweeper* problem could be obtained by adding one copy of transition "a", one copy of transition "g", and two copies of transition "d" and performing a traversal of the graph.
c) A solution for the problem of *testing combination of actions of length 2* using de Bruijn sequences would give exactly the same graph but adding copies of some transitions in order to get an Eulerized graph.
d) Assuming that each transition represents a task taking 1 hour and that the machine goes to a reset state by taking transition "d", the best time that can be obtained for transition coverage by using parallelization (on the *Street Sweeper* test sequence) is 5 hours and using 2 machines.

Task 4 – MBT / ModelJUnit

You will find below 10 statements and situations about different issues related to model-based testing and ModelJUnit. **Determine whether the statements are true or false**. **Justify your answer** giving clear arguments to defend your judgment in case a statement is **false** in your opinion (state why the answer is false and provide the correct fact). **Note that your answer will not be considered complete if you don't provide the correct fact in case the statement is false.**
(20 pts – 2 pts each)

1) The ModelJUnit library is a set of Java classes designed as an extension of JUnit for facilitating performing MBT.
2) Adaptors in ModelJUnit are special kind of EFSMs.
3) When using ModelJUnit for off-line testing you do not need to change the code of the SUT as the tests are done manually (interactively), and not automatically executed in the SUT.
4) For off-line testing you do not need to transform abstract test cases into concrete ones.
5) There are many approaches known as Model Base Testing and all have in common that they are concerned with the generation of test cases.
6) When doing MBT using ModelJUnit it is not possible to perform coverage analysis.
7) FSMs are more abstract than EFSMs, thus test cases obtained from FSMs are in general more abstract than those obtained from EFSMs.
8) One of the main advantages of MBT is that you can completely reuse the development models.
9) Getting a 100% transition coverage in the EFSM means getting a 100% statement coverage in the SUT.
10) It is easier to obtain test cases from EFSMs than from pre/post notations.

Task 5 – Property-Based Testing and QuickCheck

1) You will find below 3 statements and situations about different issues related to property-based testing and QuickCheck. **Determine whether the statements are true or false**. **Justify your answer** giving clear arguments to defend your judgment in case a statement is **false** in your opinion (state why the answer is false and provide the correct fact). **Note that your answer will not be considered complete if you don't provide the correct fact in case the statement is false. (6 pts – 2 pts each)**

    a. When you execute `quickCheck prop_xx`, QuickCheck will automatically generate a given number of test cases and give the largest possible counter-example in case the property `prop_xx` is not satisfied.

    b. *Generators* in QuickCheck are special functions used to write complex Haskell properties.

    c. The QuickCheck `Arbitrary` class provides a function `arbitrary` to generate data of each possible type. It facilitates the definition of data generators.

2) Let us assume that a module Trees includes an implementation of *binary trees* in Haskell. Let type `Tree` be such that `Tree a` is a tree of type `a` (where `a` is supposed to be an ordered type, with its standard definition) along with the following operations:

```
insert :: Ord a => a -> Tree a -> Tree a
delete :: Ord a => a -> Tree a -> Tree a
member :: Ord a => a -> Tree a -> Bool
isEmpty :: Tree a -> Bool
empty :: Tree a
merge :: Ord a => Tree a -> Tree a -> Tree a
```

Let us assume that there is an implementation for the above respecting the following informal specification. `insert a t` will insert the element `a` as a leaf in `t` (note that it is underspecified which traversal algorithm is used to find the place where to insert the element). `delete a t` will delete *all* the occurrences of `a` in case it is an element of the tree). `member a t` is true if `a` is an element of the tree `t`, and false otherwise. `isEmpty t` is true is the tree is empty. `empty` generates the empty tree. `merge t1 t2` combines both trees on a new tree (the specification does not say how this operation is done).

Binary search trees (BST) are binary trees used to store data for which there is a valid notion of order, with a view to searching for specific data efficiently (as opposed to using a flat list structure). Given a tree `t`, we say that it has the *BST property* if either of these conditions hold:

i. `t` is empty;
ii. if `t` is of the form `Node lt x rt`, then both `lt` and `rt` have the BST property, and for all `y` in `lt` we have `y < x`, and for all `z` in `rt` we have `x < z`, i.e. all elements of `lt` are lower than `x`, and all elements of `rt` are greater than `x`.

Let us assume that there is a property `prop_BST t` that is true when the tree `t` satisfies the BST property. Whenever a tree `t` is a BST (ie., it satisfies the BST property) then the functions `insert`, `delete` and `merge` also preserve the BST property as an invariant. So, `insert` takes an element of type `a` and a BST, and returns a new BST with the extra element in the correct

position. The function delete takes an element of type a and a BST, and returns a new BST without the element. IsEmpty checks whether the BST is empty, and empty creates an empty BST. The function merge merges two BSTs and returns a BST containing all the elements of the two given BSTs. Repeated elements are stored only once. Moreover, member checks that a given element is present in the tree. The function member utilizes the invariant to make the search more efficient. Nothing is said about merge.

Besides the above, let us assume you are provided with the following implementation of a function flatten :: Tree a -> [a] that gives the subjacent list of the elements of the tree (according to a certain tree traversal):

```
flatten Empty = []
flatten (Node lt x rt) = flatten lt ++ x : flatten rt
```

Moreover, assume that you have the following functions over lists: sort l which sorts in increasing order the list l; ordered l which returns true is the list l is sorted; an infix function "==" which compares two lists giving true if both lists are of the same size and are equal element by element, and false otherwise.

Assuming you are given an implementation of a module on trees as specified above, provide solutions to the following: **(14 pts)**

a. Write a property in QuickCheck about the function flatten when used on BSTs concerning the order of the elements in the resulting list (Note 1: You are asked to write a property about the flatten function not about BSTs. Note 2: You need to be sure you are operating on BSTs and not on mere binary trees.) **(2 pts)**

b. You are told that the implementation of insert on binary trees does it respecting the BST property. Write a property prop_insertl x t in QuickCheck to check whether this is indeed the case. **(2 pts)**

c. You would like to know whether two given binary trees contain the same elements. Would it be necessary to write a QuickCheck property in order to test this? **(2 pts)**

d. Does the following property hold?

```
prop_delete x t = prop_bst t => flatten (delete x (insert x t)) == flatten t
```

If you answer NO, provide the right property (as above, in QuickCheck). **(2 pts)**

e. Does the following property hold for general binary trees?

```
prop_delete x t = flatten (delete x (insert x t)) == flatten t
```

If you answer NO, provide the right property in QuickCheck. (Note: Remember that the functions insert and delete preserve the BST property as an invariant) **(2 pts)**

f. Let the following be a property on the combination of merging two trees when inserting 2 different elements on the trees:

```
prop_merge x y t1 t2 = merge (insert x t1) (insert y t2) == insert x (insert y
(merge t1 t2)),
```

where `x` and `y` are elements of type `a`, and `t1`, `t2` are trees, and "==" here is *structural equality* between trees, that is the trees have the same structure.
Does the above property hold in general for binary trees? Does it hold for BSTs?
If you answer NO to any of the above 2 questions, please provide the right property (in QuickCheck). **(4 pts)**