Examination
Model Based Testing
DIT848 / DAT260

Software Engineering and Management
Chalmers | University of Gothenburg

Tuesday May 21, 2012

Time	14:00-18:00
Location	Lindholmen
Responsible teacher	Gerardo Schneider
Phone	031 772 6073
Tasks	**5** (20 pts each)
Total number of pages	**XXX** (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 5 statements about different issues related to testing. **Determine whether the statements are true or false**. If a statement is **false** in your opinion, then **justify your answer giving clear arguments to defend your judgment** (state why the answer is false and provide the correct fact). **Your answer will not be considered complete if you do not justify it when false**. **(10 pts - 2 pts each)**

1) There are two kinds of testing: dynamic and static.
2) Validation testing is about showing that the software meets its requirements.
3) *Testing* is just another name for *debugging*.
4) The best approach to perform integration test is the Big Bang approach.
5) In order to perform integration test you need to have a full implementation of your system.

2) Your company has been commanded the development of a mobile application for interacting with a boat onboard computer. This application will upload information from the onboard computer and display the information on different graphs showing some statistics about the latest boat trips. The communication between the mobile phone and the onboard computer is done via a wireless local network, which as usual has a given radio of range. The onboard computer automatically detects the device if in the given range, and asks the user to log in (providing a username and password), unless the phone has been already registered on the computer's database in which case the connection is automatic. The ID of a phone is in the database after being connected the first time. To increase the confidence of the quality of the above software product, your company uses different kind of testing, during development, as listed below:

a) A team does code review
b) Developer writes and runs unit tests (in JUnit)
c) Developer does coverage analysis
d) A team does integration testing
e) Testers do system tests
f) Testers perform regression tests when applicable
g) 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 step in the list above where the problem is most likely to be found. Justify your answer. **Your answer will not be considered complete if you do not justify it when false. (10 pts - 2 pts each)**

1) The application has passed all the unit tests but when tried in the real environment the onboard computer asks the user to manually log in each time the phone comes into the range of the wireless network.
2) For some unknown reason the application becomes very slow after connecting a few times.
3) A given functionality of the application doesn't work when tried in the real environment. By performing code inspection the programmer sees that the underlying function has been implemented, but obviously it is not being executed.
4) In many occasions, when it is the first time that the phone is being detected by the onboard computer, a timeout is produced when the login information is sent.
5) In certain models of phones the application crashes almost all the time for unknown reasons.

SOL)

1)
1. F – testing is always dynamic
2. T
3. F – debugging is testing + correcting the errors
4. F – this is the less advisable way to do it
5. F – No, you don’t need a full implementation

2)
1. Acceptance test (g) (also during system (e))
2. stress/system test (e) and also acceptance (g)
3. Combination of coverage analysis (c) and unit tests (b)
4. timing response test (system test - e)
5. configuration test (system test - e)

Task 2 - State machines

1) This task is concerned with part of a car sharing ride system where *demanders* (asking for a ride) log in to a web system asking for a *provider* (having a car and offering places in the car) to share a particular route. Your task is to define a *Finite-State Machine* (FSM) for the following specification: 1) The demander logs in to the system; 2) the demander provides information on the particular route he/she wants and other information useful for the ride; 3) the system checks whether there is a provider satisfying the demander's request; 4) if a provider is found then an SMS is sent to both the provider and the demander confirming the ride, and the demander is logged out from the system: 5) if no provider is found, this is communicated to the demander, who is automatically logged out. **(8 pts)**

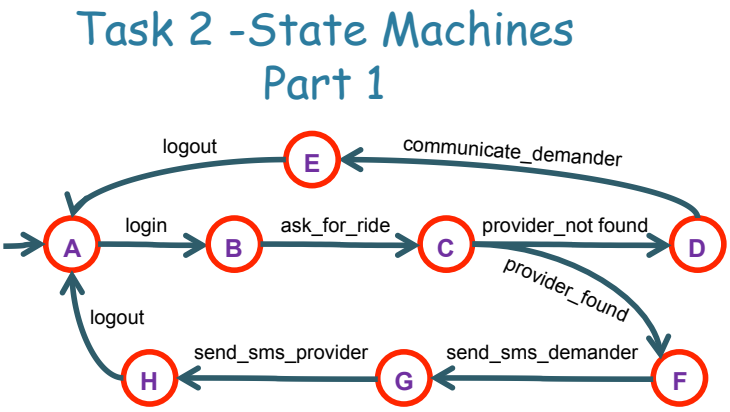
2) Give 2 test cases that can be extracted from your FSM, and two that cannot be extracted from it. **(4 pts)**

3) Draw an *Extended Finite-State Machine* (EFSM) for a variation of the system described above in 1. The new description of the system is as follows. 1) The demander logs in into the system and asks for a ride as before. 2) If a potential provider is found then it is first checked that the provider can offer the ride, which only happens if there are less than 4 confirmed demanders for that particular provider. 3) If the ride request can be accepted, then an SMS is sent to both provider and demander confirming the ride, a counter counting the number of demanders for that particular provider is increased, and the demander is logged out. 4) If a provider is not found, then the demander is put on a queue for 30 minutes after which the system checks again whether a provider for the requested ride is found; this is repeated at most 5 times, and if finally a provider is not found then the demander is automatically logged out. **(8 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.

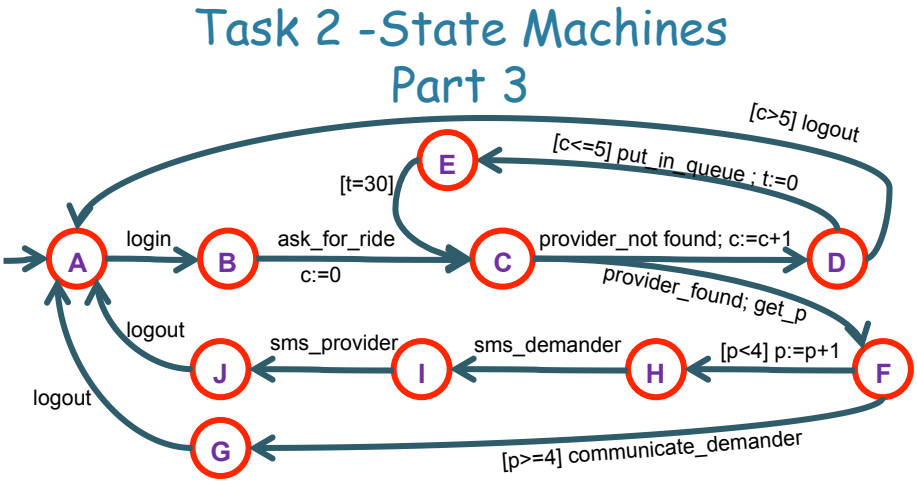
SOL)

1) .



2) Test cases you can extract:
1. After login if there is provider then the demander gets an sms indicating that.
2. If no provider exists for that ride then the user is logged out after getting a notification.

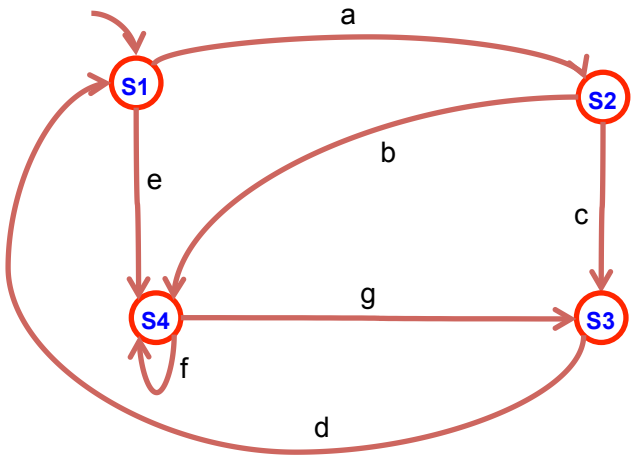
Test cases you cannot extract:
1. If a provider does exist for the ride, the user may still not get the guarantee of a ride due to overbooking.
2. Any timing constraints in what concerns how much time to wait for getting a confirmation of a ride.



3) .

Task 3 - White box testing, coverage analysis

Let the following FSM represent the model of a SUT:



(**S1** is the initial state, and **S4** is the final state)

1) Give a solution to the following *transition-based structural model coverage* criteria (give the sequence of actions to be performed to satisfy the corresponding criteria): **(10 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-transitions

2) For the FSM of the figure:

a) Say whether the FSM is: i) deterministic, ii) initially connected, iii) minimal, iv) strongly connected, v) complete. **(3 pts)**
b) Provide a solution for the *Street Sweeper* problem **(3 pts)**
c) Provide a solution for the problem of *testing combination of actions of length 2* (Hint: use de Bruijn sequences) **(4 pts)**

SOL)

1)

a. a-b-g
b. From s1: a-b, a-c, e-f, e-g;
   from s2: b-f, b-g, c-d;
   from s3: d-a, d-e;
   from s4: g-d, f-g, f-f
   (If the students consider the state as being between the transitions, this is the answer:
   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)
c. e
   a-b
   (Since any other sequence would necessarily have to pass through s1, thus repeating a config/state)
d. Add to the above visiting "f" too
e. a-b-g-d-e-f,
   a-c-d-e

2)

a. Deterministic (i), initially connected (ii), minimal (iii), strongly connected (iv)
b. Add copies of transitions a, g, d (e.g: a-c-d-e-f-g-d'-a'-b-g'-d'')
c. To Be Done!

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) Having 100% state coverage is the minimum to aim when generating test cases automatically in MBT.
2) In order to generate test cases automatically in MBT we must define an adaptor.
3) When using ModelJUnit for online testing you do not need to change the code of the SUT as this is taken care by the compiler.
4) In the adaptation technique it is needed to write a script in order to transform the test cases in the abstract level to the concrete level (SUT).
5) You can add additionally code in ModelJUnit to measure the coverage of the test cases generated automatically.
6) MBT may also be used to generate test cases automatically for doing offline testing.
7) EFSMs are more expressive than FSMs, though it has in general less visible states.
8) If you want to use MBT for online testing you need to relate abstract test cases (at the level of the EFSM) with concrete test cases (at the level of the SUT).
9) 100% transition coverage in the EFSM guarantees 100% branch coverage in the SUT.
10) In MBT both transition-based and pre/post notations are used to model the same kind of systems and the test cases generated from both approaches is the same.

SOL)

1) F – you should aim at least at a 100% transition coverage
2) F – You might use transformation and adaptation.
3) F – you might need to change the code
4) F – this is the case for the transformation, not the adaptation
5) T
6) T
7) T
8) T
9) F – It doesn't as there might be many branches in the SUT abstracted away in the EFSM
10) F – Transition-based is control oriented, while pre/post is data-oriented.

Task 5 – Property-Based Testing and QuickCheck

1) *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). Assume that a module BST includes an implementation of binary search trees in Haskell. Let type `Tree` be such that `Tree a` is a tree of type `a` (with its standard definition, as defined in assignment 6), 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
```

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`.

In what follows we assume that our trees satisfy the BST property (i.e., the functions `insert`, `delete` and `merge` 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.

Assuming you are given an implementation of the above module (preserving the BST property) as specified above, provide solutions to the following (if needed you might consider the existence of a function `flatten :: Ord a => Tree a -> [a]` that gives an ordered list of the elements of the tree): **(8 pts – 2 pts each)**

a. Write a property concerning what happens if you try to delete the same element more than once.

b. Write a property concerning what happens when you insert an element and then delete it (do not give the trivial property that the number of elements is the same).

c. Write a property concerning what happens when you first delete an element and then you insert the same element (do not give the trivial property that the number of elements is the same).

d. Write a property that checks whether 2 BSTs are not equal in the sense of not containing the same elements.

2) You will find below 4 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. (12 pts)**

a. Property-based testing (e.g., QuickCheck) is a model-based testing technique where we need to write a full model of the system represented as properties. **(2 pts)**

b. Generators in QuickCheck are used to randomly generate data for test cases that are then automatically applied to the SUT. **(2 pts)**

c. Given a Haskell module BST as described in question 1 above, the following is a property of BSTs:

```
prop_merge1 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 BSTs, and "==" here is *structural equality* between BSTs, that is the trees have the same structure. **(4 pts)**

d. The following property written in Haskell formalizes the fact that when we insert an element on a BST, the BST property (as stated in question 1 above) is preserved for trees of type `Int`: **(4 pts)**

```
prop_bst x t = bstProp t &&& bstProp (insert x t)
 where _ = t :: Tree Int
     bstProp Empty = True
     bstProp (Node lt y rt) = bstProp lt &&& bstProp rt
                    &&& all (>y) (flatten lt)
                    &&& all (<y) (flatten rt)
```

SOL) (see file BST.hs in the same directory as this file!)

1) a. prop_delete1 :: Int -> Tree Int -> Bool
prop_delete1 x t = delete x (delete x t) == delete x t

b. (Note that the it is not necessarily true that you get the same tree!)
prop_delete2 :: Int -> Tree Int -> Property
prop_delete2 x t = not (member x t) ==> flatten (delete x (insert x t)) == flatten t

c. (Note that the it is not necessarily true that you get the same tree!)
prop_delete3 :: Int -> Tree Int -> Property
prop_delete3 x t = (member x t) ==> (flatten (insert x (delete x t)) == flatten t)

d. The statement should be read as "Write a property that checks that 2 BSTs are not equal if they don't contain the same elements."

prop_equal :: Tree Int -> Tree Int -> Property
prop_equal t1 t2 = not (flatten t1 == flatten t2) ==> t1 /= t2

2) a. F – you write properties, not necessarily a full model.

b. T

c. F – There is no guarantee of getting the same tree. You should write:
prop_merge1 x y t1 t2 = flatten (merge (insert x t1) (insert y t2)) == flatten (insert x (insert y (merge t1 t2)))

d. F - The problem is that the symbols < and > are interchanged (so the solution is to make the following change: "&& all (<y) (flatten lt) && all (>y) (flatten rt)")