DIT636/DAT560 - Assignment 3: Fault-Based Testing and Finite-State Verification

Due Date: Sunday, March 12th, 23:59 (Via Canvas)

There are two questions worth a total of 100 points. You may discuss these problems in your teams and turn in a single submission for the team (zipped archive) on Canvas. Answers must be original and not copied from online sources.

Cover Page: On the cover page of your assignment, include the name of the course, the date, your group name, and a list of your group members.

Peer Evaluation: All students must also submit a peer evaluation form. Peer evaluation is a separate, individual submission on Canvas.

Problem 1 - Mutation Testing (45 Points)

In this question, you will apply Mutation Testing to the code from the CoffeeMaker example from Assignment 2. The CoffeeMaker code can be found at:
https://chalmers.instructure.com/courses/22401/files/folder/Assignments?preview=2615925

1. Create six mutants for classes from the CoffeeMaker project. Your report should include the mutated code, explaining how it differs from the original code and why it belongs in one of the categories indicated below. In this explanation, be detailed. Feel free to give examples of test input **(20 Points).**
    a. One mutant must be **invalid** (does not compile).
    b. One must be **equivalent** to the original code (you inserted a fault, but no test case can possibly yield a different solution to the original code).
    c. Two mutants must be **valid-but-not-useful** (all tests, or almost all tests, will expose this mutation).
    d. Two mutants must be **useful** (only a small number of specifically-designed tests will expose this mutation).

    You must apply at least four different mutation operators, and you must use at least one mutation operator from each of the three categories in the attached handout (for more information, see Chapter 16 of Software Testing and Analysis).

2. Assess the test suite that you created for **Assignment 2** using the set of mutants that you derived.
    a. Identify which test cases expose which mutants (test cases that expose a mutant pass on the original code and fail on the mutated code). **(10 Points)**.
    b. Explain why these tests expose each mutant. For any non-equivalent mutants not exposed, explain why they were not exposed. **(10 Points)**
    c. Then, if needed, design additional test cases that detect the remaining mutants and describe why they succeed in detecting them, highlighting what differentiates the new tests from past test cases. **(5 Points)**

Hint: You do not have to use the same classes or methods for all mutant categories. Try mutating different parts of the code. You may use any class except Main or the exceptions.

Problem 2 - Finite-State Verification (55 Points)

For this exercise, you are required to create a finite-state model of a traffic-light controller and verify its properties using the NuSMV symbolic model-checker (download from http://nusmv.fbk.eu/)

A tutorial for NuSMV can be found at: https://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf

NuSMV is a command-line tool. It is available for all major operating systems. To check your properties, you can simply run the following command from the command line:

(Windows) `NuSMV <model filename>`
(Mac/Linux) `./NuSMV <model filename>`

Consider a simple model, with a property at the end:

```
MODULE main
VAR
request : boolean;
state : {ready, busy};

ASSIGN
init(state) := ready;
next(state) := case
    state = ready & request = TRUE : busy;
    TRUE : {ready, busy};
esac;

SPEC AG (state = ready);
```

This model has two variables, **`request`** and **`state`**. **`request`** is an input from the outside environment, outside of our control. Therefore, its value is set randomly at each timing step (with possible values “true” and “false”). **`state`** is an internal variable of our model, with values “ready” and “busy”. We set its value based on the value of **`request`** and the current value of **`state`**. If the current value is “ready” and we get a request, we transition to the value “busy”. Otherwise, we set the next value of **`state`** randomly.

The property states that the value of **`state`** is always “ready”, and will always remain “ready”. This is absolutely not going to be the case. Therefore, when we run NuSMV, we get a counterexample illustrating a situation where the property is violated (the value of **state** becomes “busy”):

```
C19ZRMR:bin ggay$ ./NuSMV main.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:32:58 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification AG state = ready  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    request = FALSE
    state = ready
  -> State: 1.2 <-
    state = busy
```

Because we set the value randomly in the absence of a request, it will eventually become “busy” no matter what we do, as is the case in this example. The counterexample consists of two steps (two state transitions). In the first, **`request`** is “false”, and **`state`** is “ready”. Because **`request`** is “false”, we set the next value of **`state`** randomly. As a result, in the second step, **`state`** becomes “busy” (**`request`** is not printed, as its value is not relevant).

Now, it is your turn to design a slightly more complicated model - a traffic-light controller.

- Assume that the controller manages traffic and pedestrian lights at the intersection of two roads, both with two-way traffic.
- Pedestrians can request access to cross the road by pressing a “walk button”.
- Assume that the system has traffic sensors for each direction to detect if vehicles are present and waiting to pass through, which allows the system to manage traffic flow efficiently by varying the amount of time the lights are green for each road/direction based on demand. Your model should capture and represent this notion of varying time in some manner (i.e., do not completely abstract away time).
- There are emergency vehicle sensors for each direction which lets the system provide priority access for emergency vehicles by switching lights appropriately.

You may state and make any other reasonable simplifying assumptions that you need. A simplified traffic light model appears in the slides for Lecture 14. Understanding that model is a good first step in solving this problem. There is also a model example in Exercise Session 6.

In your submission, you must address the following:

1. Define the scope, assumptions, and requirements for the system that you intend to model – a brief description of what you have modeled, any assumptions that you have made and the key requirements you expect the system to satisfy. There is not a specific format this must be in. We are interested in understanding your thought process and assumptions. **(10 Points)**
2. Build a finite state model of the system in the NuSMV language. Be sure to write sufficient comments. (Though not required, you may find drawing state diagrams helpful). **(20 Points)**
3. Write at least three **safety** properties ("something bad must never happen") in temporal logic (CTL or LTL) that must be satisfied by the system. Explain your properties and state which system requirements those properties are derived from. **(10 Points)**
4. Write at least three **liveness** properties ("something good must eventually happen") in temporal logic (CTL or LTL) that must be satisfied by the system. Explain your properties and state which system requirements those properties are derived from. **(10 Points)**
5. Verify your properties on your system using the NuSMV symbolic model checker and provide a transcript of your NuSMV session. **(5 Points)**

Note: Yes, we know, NuSMV is a research tool and its website looks like something from the 90's. However, (1) similar tools are used in MANY companies writing safety-critical code like Volvo, Boeing, and Amazon (e.g., https://www.amazon.science/working-at-amazon-from-nasa-ames-research-center-to-automated-reasoning-group-aws-neha-rungta), (2) this tool is FREE (most industrial tools are either not available or cost $$$$$$), and (3), NuSMV is a relatively beginner-friendly tool. Knowing how to use any of these tools could prove very useful in getting a job in a verification role.