DIT636 / DAT560 - Assignment 4: Mutation Testing and Finite-State Verification

Due Date: Friday, March 14th, 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.

Report Template: A template for your report can be found here.

Cover Page: On the cover page of your report, 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. This is a separate, individual submission on Canvas.

Problem 1 - Mutation Testing (45 Points)

In this question, you will apply Mutation Testing to the CoffeeMaker example from Assignment 2.

The CoffeeMaker code can be found at:
https://github.com/Greg4cr/dit636_examples/tree/main/src/as2-coffeemaker

1. Create six mutants for classes from the CoffeeMaker project.
    - Your report should include the mutated code and an explanation of 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).**
        i. One mutant must be **invalid** (does not compile).
        ii. 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).
        iii. Two mutants must be **valid-but-not-useful** (all tests, or almost all tests, will expose this mutation).
        iv. 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 across this set, and you must use at least one mutation operator from each of the three categories in the attached handout.
2. Assess the test suite that you created for **Assignment 2** using these mutants:.
    - 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)**.
    - Explain why these tests expose each mutant. For any non-equivalent mutants not exposed, explain why they were not exposed. **(10 Points)**
3. 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)**
    - If all mutants were detected by existing tests, just make it clear in the report.

Include both the report as well as the code for the test cases in your submission.

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.

For more information on the mutations, see Chapter 16 of Software Testing and Analysis.

Problem 2 - Finite-State Verification (55 Points)

For this exercise, you are required to create a finite-state model of a traffic controller for a “lift bridge” (see below) and verify its properties using the NuSMV symbolic model checker.

NuSMV is a free, open-source model checker. Similar tools are used at companies writing safety-critical code like Volvo, Boeing, and Amazon. Knowing how to use any of these tools could prove very useful in getting a specialized well-paying job.

NuSMV can be downloaded from http://nusmv.fbk.eu/
A tutorial for NuSMV can be found at: https://nusmv.fbk.eu/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:

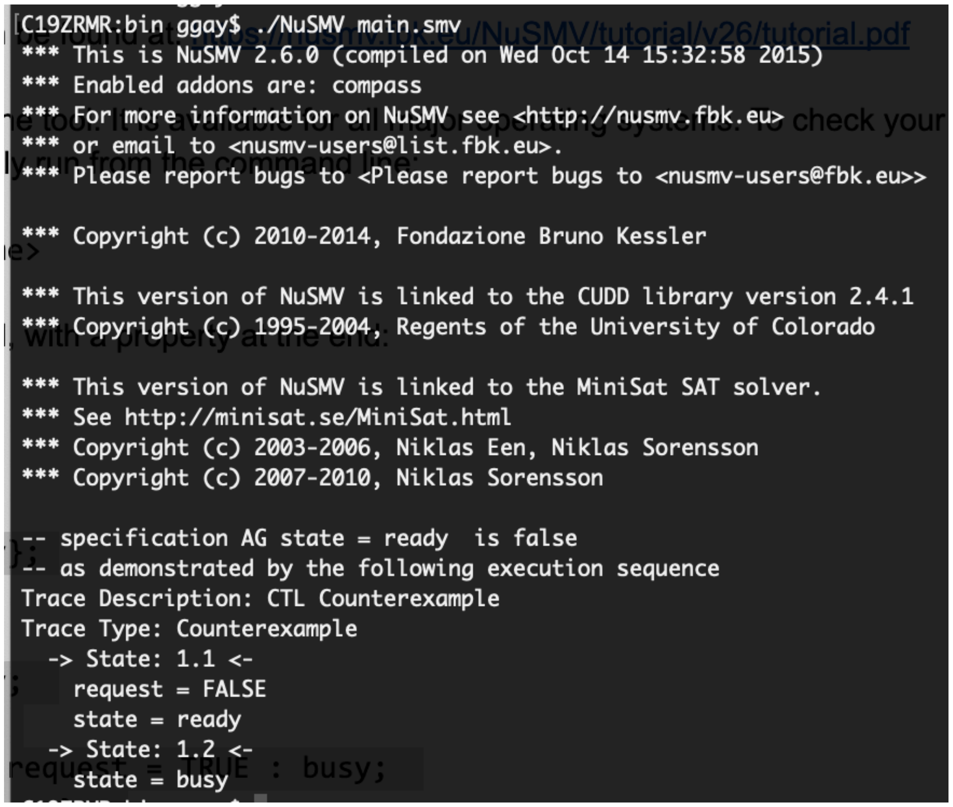
```
MODULE main
VAR
request : boolean;
state : {ready, busy};

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 - the control software for a “lift bridge” (e.g., Hisingsbron in Gothenburg).

- The bridge allows car traffic in both directions. At the start of the bridge (on both sides), there is a traffic light that should turn red when cars are not allowed onto the bridge, and should be green when cars are allowed. There should also be a transition (yellow) period when the light is about to turn red.
- The bridge can lift a segment into the air to allow boat traffic to pass. When the segment is lifted, cars should not be allowed to drive across the bridge. There is a similar traffic light for boards, that is red when the bridge is lowered (i.e., boats cannot pass), green when the bridge is lifted, and yellow during a transition.
- Assume that the system has sensors for both car and boat traffic to detect if either is present and waiting to pass through.

- This controller must manage traffic flow efficiently by varying the amount of time that the light is green for vehicles to pass or red (i.e., the bridge is raised) to allow ship traffic to pass, based on demand. Your model should capture and represent this notion of varying time and varying demand in some manner (i.e., do not completely abstract away time).
- There is an emergency vehicle sensor for road traffic, which lets the system provide priority access for emergency vehicles. If an emergency vehicle is waiting to pass, then ship traffic should be halted as soon as it is safe to do so.
- Note that it takes a short period of time for the bridge to raise or lower, so you cannot switch between road and ship traffic immediately. The light for vehicles should be red while the bridge is in motion, as new cars cannot enter at this time. This means that the transition should affect the time that the system calculates for a green light for boats.

You may state and make any other reasonable simplifying assumptions that you need.

For inspiration, consider the traffic light model that appears in the slides for Lecture 14. Understanding that model is a good first step in solving this problem, as it models a similar situation. There is also a useful model example in Exercise Session 6.

In your submission, you must address the following:

1. In the report, define the scope, assumptions, and requirements for the system that are modeling. **(10 Points)**
   - Give a brief description of the system and environment you have modeled.
   - State any assumptions that you have made.
   - State the requirements you expect the system to satisfy.
   - There is not a specific format this must be in. We are most interested in understanding your thought process and assumptions. Feel free to change the template.
2. Build a finite state model of the system in the NuSMV language. Be sure to write sufficient comments in the model. **(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. **(10 Points)**
   - In the report, explain each property and state which system requirements those properties are derived from.
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. **(10 Points)**
   - In the report, explain each property and state which system requirements those properties are derived from.
5. Verify your properties on your system using the NuSMV symbolic model checker and provide a transcript of your NuSMV session. **(5 Points)**
   - This transcript should be provided as a separate file, not placed in the report.