DIT635 - Assignment 3: Mutation Testing and Finite-State Verification

Due Date: Friday, March 13, 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. This is a seperate, individual submission on Canvas. Not submitting a peer evaluation will result in a penalty of five points on this assignment.

Problem 1 (45 Points)

In this question, you will apply Mutation Testing to the code from the CoffeeMaker example from Assignment 1. The CoffeeMaker code can be found at:
http://Greg4cr.github.io/courses/spring20dit635/Assignments/CoffeeMaker_JUnit.zip

1. Generate at least four mutants for classes of your choice in the CoffeeMaker code (before you apply any of your fixes from Problem 1). Your report should include the mutated code, noting how it differs from the original code. **(20 Points)**
    a. You must create at least one invalid, one valid-but-not-useful (non-equivalent), one useful, and one equivalent mutant.
    b. Each mutant must be created by applying a different mutation operator, and you must use at least one mutation operator from each of the three categories in the attached handout.
    c. You do not have to use the same classes or methods for all mutant categories. Try mutating different parts of the code.
2. Assess your test suite that you created for Assignment 1, with respect to the set of mutants that you derived - Are you able to kill all of the non-equivalent mutants with your test suite? If not, describe which non-equivalent mutants cannot be differentiated from the original code using your original test suite, and why they cannot be differentiated. Write additional tests that can kill those non-equivalent mutants. **(15 Points)**
3. Identify a minimal subset of tests from your test suite that is sufficient to kill all of the non-equivalent mutants. **(10 Points)**

Problem 2 (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/ - **we will not provide technical support for this tool**)

- 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 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.

In your submission, you must address the following:

1. Define the scope and the 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. **(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)**

ID	Operator	Description	Constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by $\mathsf{abs}(e)$	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Figure 16.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.