

DAT560 / DIT636 Software quality and testing

12th March 2023 Group 19, Members: Avinash Shukla, Ossian Ålund

Problem 1 - Mutation Testing

Case1:

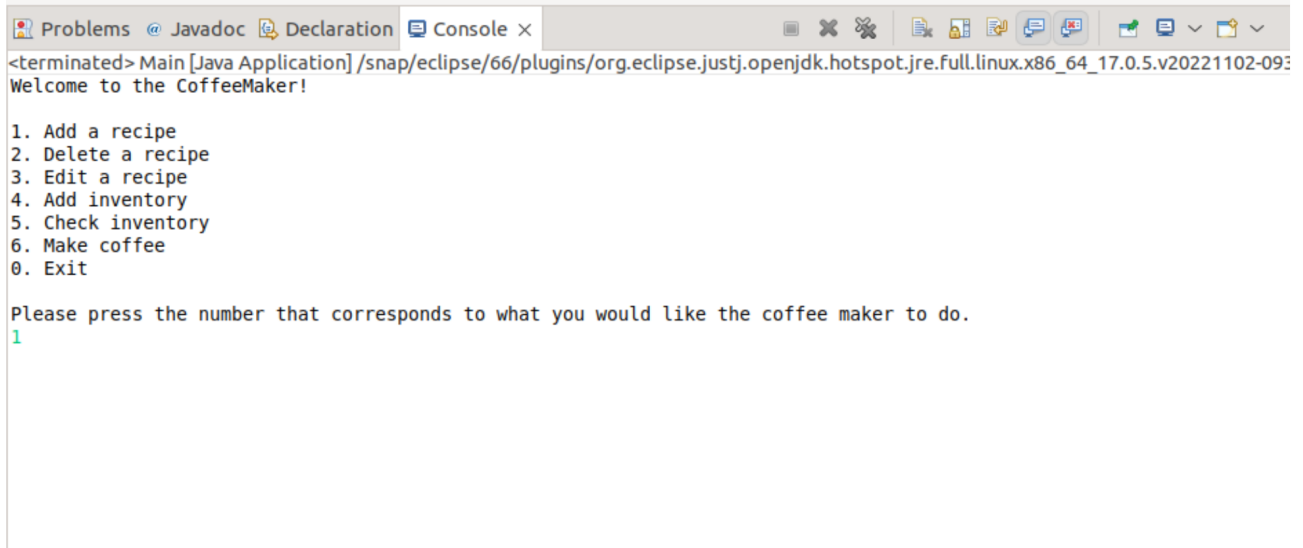
Original Code:

```
if (userInput >= 0 && userInput <=6) {  
    if (userInput == 1) addRecipe();  
    if (userInput == 2) deleteRecipe();  
    if (userInput == 3) editRecipe();  
    if (userInput == 4) addInventory();  
    if (userInput == 5) checkInventory();  
    if (userInput == 6) makeCoffee();  
    if (userInput == 0) System.exit(0);  
}
```

Mutated Code:

```
if (userInput >= 0 && userInput <=6) {  
    if (userInput != 1) addRecipe();  
    if (userInput == 2) deleteRecipe();  
    if (userInput == 3) editRecipe();  
    if (userInput == 4) addInventory();  
    if (userInput == 5) checkInventory();  
    if (userInput == 6) makeCoffee();  
    if (userInput == 0) System.exit(0);  
}
```

In this scenario for the addRecipe() function I made a change in terms of userInput i.e. != 1 when user run the code and enters the user input value 1 code will not run. Attached herewith is the screenshot. It falls under Decision Mutation.



```
<terminated> Main [Java Application] /snap/eclipse/66/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.5.v20221102-093
Welcome to the CoffeeMaker!

1. Add a recipe
2. Delete a recipe
3. Edit a recipe
4. Add inventory
5. Check inventory
6. Make coffee
0. Exit

Please press the number that corresponds to what you would like the coffee maker to do.
1
```

Case 2:

Original code:

```
/**
 * Delete recipe user interface that processes input.
 */

public static void deleteRecipe() {
    Recipe [] recipes = coffeeMaker.getRecipes();
    for(int i = 0; i < recipes.length; i++) {
        if (recipes[i] != null) {
            System.out.println((i+1) + ". " + recipes[i].getName());
        }
    }
    int recipeToDelete = recipeListSelection("Please select the number of the recipe to
delete.");

    if(recipeToDelete < 0) {
        mainMenu();
    }

    String recipeDeleted = coffeeMaker.deleteRecipe(recipeToDelete);
}
```

```

    if (recipeDeleted != null) {
        System.out.println(recipeDeleted + " successfully deleted.\n");
    } else {
        System.out.println("Selected recipe doesn't exist and could not be
deleted.\n");
    }
    mainMenu();
}

```

Mutated Code:

```

/**
 * Delete recipe user interface that processes input.
 */
public static void deleteRecipe() {
    Recipe [] recipes = coffeeMaker.getRecipes();
    for(int i = 0; i < recipes.length; i++) {
        if (recipes[i] != null) {
            System.out.println((i+1) + ". " + recipes[i].getName());
        }
    }
    int recipeToDelete = recipeListSelection("Please select the number of the recipe to
delete.");

    if(recipeToDelete < 0) {
        mainMenu();
    }

    String recipeDeleted = coffeeMaker.deleteRecipe(recipeToDelete);

    if (recipeDeleted != null) {
        System.out.println(recipeDeleted + " successfully deleted.\n");
    }
    mainMenu();
}

```

In this case I deleted the else block and it does not effect the delete functionality. It comes under statement mutations.

Case 3.1:

Original code:

```
public static void deleteRecipe() {
    Recipe [] recipes = coffeeMaker.getRecipes();
    for(int i = 0; i < recipes.length; i++) {
        if (recipes[i] != null) {
            System.out.println((i+1) + ". " + recipes[i].getName());
        }
    }
    int recipeToDelete = recipeListSelection("Please select the number of the recipe to delete.");

    if(recipeToDelete < 0) {
        mainMenu();
    }

    String recipeDeleted = coffeeMaker.deleteRecipe(recipeToDelete);

    if (recipeDeleted != null) {
        System.out.println(recipeDeleted + " successfully deleted.\n");
    } else {
        System.out.println("Selected recipe doesn't exist and could not be deleted.\n");
    }
    mainMenu();
}
```

Mutated code:

```
public static void deleteRecipe() {
    Recipe [] recipes = coffeeMaker.getRecipes();
    for(int i = 0; i > recipes.length; i++) {
        if (recipes[i] != null) {
            System.out.println((i+1) + ". " + recipes[i].getName());
        }
    }
    int recipeToDelete = recipeListSelection("Please select the number of the recipe to delete.");

    if(recipeToDelete < 0) {
        mainMenu();
    }
}
```

```

    }

    String recipeDeleted = coffeeMaker.deleteRecipe(recipeToDelete);

    if (recipeDeleted != null) {
        System.out.println(recipeDeleted + " successfully deleted.\n");
    } else {
        System.out.println("Selected recipe doesn't exist and could not be
deleted.\n");
    }
    mainMenu();
}

```

In the above scenario we can delete the recipe.

Case 3.2:

Original code:

```

int recipeToDelete = recipeListSelection("Please select the number of the recipe to
delete.");

```

```

    if(recipeToDelete < 0) {
        mainMenu();
    }

```

Mutated code:

```

int recipeToDelete = recipeListSelection("Please select the number of the recipe to
delete.");

```

```

    if(recipeToDelete > 0) {
        mainMenu();
    }

```

Case 4.1:

Original code:

```

int userInput = Integer.parseInt(inputOutput("Please press the number that corresponds
to what you would like the coffee maker to do."));

```

```

    if (userInput >= 0 && userInput <=6) {

```

```

        if (userInput == 1) addRecipe();
        if (userInput == 2) deleteRecipe();
        if (userInput == 3) editRecipe();
        if (userInput == 4) addInventory();
        if (userInput == 5) checkInventory();
        if (userInput == 6) makeCoffee();
        if (userInput == 0) System.exit(0);
    }

```

Mutated code:

`int userInput = Integer.parseInt(inputOutput("Please press the number that corresponds to what you would like the coffee maker to do."));`

```

    if (userInput >= 0 && userInput <=6) {
        if (userInput == 1) addRecipe();
        if (userInput == 2) deleteRecipe();
        if (userInput == 3) editRecipe();
        if (userInput == 4) addInventory();
        if (userInput == 5) checkInventory();
        if (userInput != 6) makeCoffee();
        if (userInput == 0) System.exit(0);
    }

```

Case 4.2:

Original code:

```

String amountPaid = inputOutput("Please enter the amount you wish to pay");
    int amtPaid = 0;
    try {
        amtPaid = Integer.parseInt(amountPaid);
    } catch (NumberFormatException e) {
        System.out.println("Please enter a positive integer");
        mainMenu();
    }

```

Mutated code:

```

String amountPaid = inputOutput("Please enter the amount you wish to pay");
    int amtPaid = 0;
    try {
        amtPaid = Integer.parseInt(amountPaid);
    }

```

deleted catch block.

Case 5:

Original code:

```
int userInput = Integer.parseInt(inputOutput("Please press the number that  
corresponds to what you would like the coffee maker to do."));
```

```
    if (userInput >= 0 && userInput <=6) {  
        if (userInput == 1) addRecipe();  
        if (userInput == 2) deleteRecipe();  
        if (userInput == 3) editRecipe();  
        if (userInput == 4) addInventory();  
        if (userInput == 5) checkInventory();  
        if (userInput == 6) makeCoffee();  
        if (userInput == 0) System.exit(0);  
    }
```

Mutated code:

```
int userInput = Integer.parseInt(inputOutput("Please press the number that  
corresponds to what you would like the coffee maker to do."));
```

```
    if (userInput >= 6 && userInput <=0) {  
        if (userInput == 1) addRecipe();  
        if (userInput == 2) deleteRecipe();  
        if (userInput == 3) editRecipe();  
        if (userInput == 4) addInventory();  
        if (userInput == 5) checkInventory();  
        if (userInput == 6) makeCoffee();  
        if (userInput == 0) System.exit(0);  
    }
```

Statement Modification

ID	Operator	Description	
----	----------	-------------	--

sdl	Statement deletion	Deleted an else statement	Case2
sdl	Statement deletion	Deleted a catch statement	Case4.2

Expression Modification

ID	Operator	Description	
ror	Relational operator replacement	Replaced == with !=	Case 1
ror	Relational operator replacement	Replaced == with !=	Case 4.1
ror	Relational operator replacement	Replaced < and > operator	Case 3.1
ror	Relational operator replacement	Replaced < and > operator	Case 3.2

Operand Modification

ID	Operator	Description	
crp	Constant for constant replacement	Replaced constant 6 and 0	Case5

Problem 2 - Finite-State Verification (55 Points)

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)**

Description and key requirements and scope: The physical scope of the system includes the traffic lights and pedestrian lights, the sensors for detecting vehicles, the walk buttons for pedestrians, and the emergency vehicle sensors. The system itself is a traffic-light controller that manages traffic and pedestrian lights at the intersection of two roads, both with two-way traffic. The system's primary function is to efficiently manage traffic flow and ensure the safety of pedestrians by controlling the traffic and pedestrian lights in response to various inputs from the environment, such as vehicles, pedestrians, and emergency vehicles.

Assumptions: we assume all traffic laws will be followed by vehicles and pedestrians. We're assuming the system will be correctly installed and that all necessary infrastructure is working. We're assuming pedestrians will make use of the crossover button (some stops have buttons that do nothing just to pass the time).

Requirements:

1. **The system must efficiently manage traffic flow and reduce congestion for vehicles.**
2. **The system must prioritize pedestrian safety and ensure timely access for pedestrians.**
3. **The system must provide priority access for emergency vehicles and ensure their safe and timely passage through the intersection.**
4. **The system must use sensors to detect the presence of vehicles and pedestrians and adjust the traffic and pedestrian lights accordingly.**
5. **The system must be able to handle a wide range of traffic conditions, including rush hour traffic and periods of low traffic volume.**
6. **The system must comply with relevant traffic laws and regulations.**

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)**

```
MODULE main
VAR
  // Inputs
  walk_request : boolean;
  traffic_left : boolean;
  traffic_right : boolean;
  emergency_left : boolean;
  emergency_right : boolean;

  // Outputs
  green_light_left : boolean;
  green_light_right : boolean;
  pedestrian_light : boolean;

  // Internal state
  state : {normal, emergency_left, emergency_right};
  timer_left : 0..10;
  timer_right : 0..10;

ASSIGN
  // Initialize state
  init(state) := normal;
  init(timer_left) := 0;
  init(timer_right) := 0;

  // Next state
  next(state) := case
    // Emergency vehicle approaching from left
    state = normal & emergency_left = TRUE : emergency_left;
    // Emergency vehicle approaching from right
    state = normal & emergency_right = TRUE : emergency_right;
    // Emergency vehicle already granted priority access from left
    state = emergency_left & timer_left = 10 : normal;
    // Emergency vehicle already granted priority access from right
    state = emergency_right & timer_right = 10 : normal;
    // Emergency vehicle approaching from both directions
```

```

    state = normal & emergency_left = TRUE & emergency_right = TRUE : emergency_left;
    // Normal state
    TRUE : normal;
esac;

// Traffic light outputs
green_light_left := case
    state = normal & timer_left < 5 & (timer_right >= 5 | traffic_right = FALSE) : TRUE;
    state = emergency_right : FALSE;
    state = emergency_left : TRUE;
    TRUE : FALSE;
esac;

green_light_right := case
    state = normal & timer_right < 5 & (timer_left >= 5 | traffic_left = FALSE) : TRUE;
    state = emergency_left : FALSE;
    state = emergency_right : TRUE;
    TRUE : FALSE;
esac;

// Pedestrian light output
pedestrian_light := case
    state = normal & walk_request = TRUE & (traffic_left = FALSE | traffic_right = FALSE) :
TRUE;
    TRUE : FALSE;
esac;

// Timers
timer_left := case
    state = normal & timer_left < 10 & (green_light_left = TRUE | (timer_left > 0 & traffic_right =
TRUE)) : timer_left + 1;
    state = emergency_left : 0;
    TRUE : timer_left;
esac;

timer_right := case
    state = normal & timer_right < 10 & (green_light_right = TRUE | (timer_right > 0 & traffic_left =
TRUE)) : timer_right + 1;
    state = emergency_right : 0;
    TRUE : timer_right;
esac;

SPEC
// Safety property

```

```

AG(!(state = emergency_left & state = emergency_right));

// Liveness property
AG((traffic_left = TRUE | traffic_right = TRUE) -> (AF(green_light_left = TRUE) |
AF(green_light_right = TRUE)));

// Fairness property
FAIRNESS
((traffic_left = TRUE | traffic_right = TRUE) -> (green_light_left = TRUE | green_light_right =
TRUE | pedestrian_light = TRUE));

```

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)**

Property 1: The pedestrian light cannot be green at the same time as the traffic light for the same direction.

This property is derived from the requirement that The system must comply with relevant traffic laws and regulations, seeing as this is in all likelihood not within the law.

AG $\neg(\text{pedestrian_light} = \text{green} \wedge \text{traffic_light} = \text{green})$

Property 2: Emergency vehicle sensors must always be given priority over pedestrian requests.

This property is derived from the requirement that emergency vehicles must always be given priority over pedestrian requests to ensure a timely response to emergencies.

G(emergency_sensor = true \rightarrow (F traffic_light = green \wedge pedestrian_light = red))

Property 3: The system should never allow pedestrians to cross when vehicles are passing through the intersection.

This property is derived from the requirement that the system must ensure the safety of pedestrians by preventing them from crossing the road when vehicles are passing through the intersection.

AG $\neg(\text{pedestrian_light} = \text{green} \wedge \text{vehicle_present} = \text{true})$

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)**

Property 1: If a pedestrian presses the walk button, the pedestrian light must eventually turn green.

This property is derived from the requirement that the system must respond to pedestrian requests by eventually allowing them to cross the road safely.

G(pedestrian_request = true \rightarrow F pedestrian_light = green)

Property 2: If there are no vehicles waiting to pass through, the traffic light must eventually turn green for the other direction.

This property is derived from the requirement that the system must efficiently manage traffic flow by varying the amount of time the lights are green for each road/direction based on demand.

AG(vehicle_present = false \wedge traffic_light = red \rightarrow F traffic_light' = green)

Property 3: If there is an emergency vehicle waiting to pass through, the traffic light must eventually turn green for that direction.

This property is derived from the requirement that emergency vehicles must be given priority access by switching lights appropriately to ensure a timely response to

AG(emergency_sensor = true \rightarrow F traffic_light = green)