Lab 4 - Week 6. Model Based Testing

Aim

Apply the theory from the Model Based Testing lecture regarding I/O label transition systems

Prerequisites

- Read or reread the paper ***Model based testing with labelled transition systems*** **by** ***Jan Tretmans***.
- Look at the models and datatypes defined in the ***LTS.hs*** **file**.

Submission Guidelines

- Submit one Haskell file per exercise (only Haskell files allowed).
- Name each file as `ExerciseX.hs` for exercises and `EulerX.hs` for Euler problems.
- Ensure that the respective module of each file has the same naming format (`module ExerciseX where`).
- Follow the exercise naming conventions closely. Some exercises go through automated testing, so it is important not to change the indicated declarations.
- Do not include any personally identifiable information in the submissions.
- If using additional dependencies, indicate so in a comment at the top of the file.
- Indicate the time spent on each exercise like so `Time Spent: X min`. Make sure it is **in minutes.**
- Codegrade: For each assignment, you need to create a new group with all teammates. Please note that once you submit, you cannot change the team structure, so be cautious.

Imports

```
import Data.List
import LTS
```

```
import Test.QuickCheck
```

Exercise 1

The IOLTS datatype allows, by definition, for the creation of IOLTS's that are not valid.

1. Make a list of factors that result in invalid IOLTS's.
2. Write a function that returns true iff a given LTS is valid according to the definition given in the Tretmans paper with the following specification:

```
validateLTS :: IOLTS -> Bool
```

3. Implement multiple concrete properties for this function (you will use QuickCheck to test them in a following Exercise)

Deliverables: list of factors, implementation, concise test report, indication of time spent.

Exercise 2

This exercise related to implementing generation for IOLTS

1. Implement at least one random generator `ltsGen :: Gen IOLTS` for Labelled Transition Systems.
2. You **may (👀)** also implement additional generators to generate LTS/IOLTS's with certain properties.
3. Use your generator(s) to test the properties implemented in the previous exercise.

Deliverables: Random IOLTS generator(s), QuickCheck tests for validateLts, an indication of time spent.

Exercise 3

This exercise relates to suspension traces (straces):

1. Implement a function that returns all suspension traces of a given IOLTS

```
straces :: IOLTS -> [Trace]
```

2. Use your `IOLTS generator` and your `straces` function to create a random traces generator for QuickCheck
3. Test your `straces` function using QuickCheck

NOTE: To help your implementation, we have provided a `traces` function and other helper functions in the `LTS.hs` . Before attempting this exercise, try to understand what consitutes an strace, read Tretmans' paper closely and comprehend the theory. This is the hardest exercise in this lab.

Deliverables: Haskell program, QuickCheck Tests, random traces generator, an indication of time spent.

Exercise 4

1. Implement the `after` function (infix) for IOLTS corresponding with the definition in the Tretmans paper. Use the following specification:

```
after :: IOLTS -> Trace -> [State]
```

2. Create tests to test the validity of your implementation with both `traces` and `straces.`

Deliverables: Haskell program, tests, short test report, indication of time spent.

Exercise 5

It's finally time to use our IOLTS system to test an implementation. Look at the door implementation provided in the `LTS.hs` file. These door implementations are our SUT `doorImpl1` is correct, but the other doors have flaws. Create an IOLTS that specifies the correct behavior for this door. The states returned by the door implementation are internal states, and do not necessarily correspond with states in your model. Each time the door is used, the previously returned state must be passed as the State argument. The initial state for each of the door implementations is 0.

Write a function:

```
testLTSAgainstSUT :: IOLTS -> (State -> Label -> (State, La
bel)) -> Bool
```

that returns True if the SUT correctly implements the IOLTS and either returns false or throws an error if it doesn't. Think about a way to show descriptive errors, such that it is apparent what the flaw is and how to fix it.

Deliverables: Haskell program, description of each bug, indication of time spent.

Exercise 6 - Bonus

Create a method visualizeLTS that creates a visual representation of a given (IO)LTS. Create another function that applies the method visualizer to a random LTS created by using your random LTS generator.

Deliverables: Implementations, indication of time spent.
