Lab 2 - Week 4. Model Based Testing

Aim

Apply the theory from the Model Based Testing lecture regarding I/O label tranistion 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**.
- Make a list of questions on specific points that cause difficulty of understanding.

Submission Guidelines

- Please submit one Haskell file per exercise (only Haskell files will be admitted) per group (one set of answers per group).
- Name each file in the following format: `ExerciseX.hs` for exercises, where X is the exercise number eg. `Exercise1.hs` (note the capital), and the euler problems: `EulerX.hs` where X is the problem number. All your Haskell files need to have a capital first letter for the tests to pass.
- Make sure the respective module of each file is in the same naming format (`module ExerciseX where`)
- Each submitted file should contain one module with the same naming convention as the file name.
- Follow closely the naming conventions indicated by each exercise (some exercises go through automatic testing and it creates an overhead for the TAs if your files don’t compile)
- For anonymity, do not add any personally identifiable information in the submissions, such as your name, email, etc
- Codegrade: You will have to create a new group for every submission so make sure all your teammates are aware and have joined before the submission. We recommend you

make the codegrade group in the beginning of the assignment

- If you are using additional dependencies please indicate so in a comment on top of the file.
- Please indicate the time spent on every exercise (following the format: **Time Spent: X min**).

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 specfication:

```
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 implemeting 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, indication of time spent.

Exercise 3

This exercise relates to suspention 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`

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

Exercise 4

1. Implement the function `after` (infix) for IOLTS corresponding with the definition in the Tretmans paper.
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, Label)) -> 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 an 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.
