Model Checking SS23
Assignment 10
Due: June 22th, 2023, 16:00

For this assignment sheet we will further extend our skills using `storm` by incorporating `stormpy`. We are going to use `storm` 's python bindings to compute and evaluate schedulers and to simulate execution paths.

You can download a ready-to-use docker image with:

```
docker pull movesrwth/stormpy
```

For this assignment sheet we will use two different simulation scripts. Their usage is covered in their respective exercises.
We will update the git repository with further examples here:

https://git.pranger.xyz/sp/MC-ProbMC-PrismFiles

Have fun!

1. [**5 Points**] Car - Pedestrian.

   For this first exercise you are going to extend the model that we have created in class such that both the car and the pedestrian move nondeterministically.

   Please use the skeleton file from the git repository as a starting point.

   The car should be able to execute the following actions:

   - `changeLeft`: Change to the lane left of the car and increment its `car_street_pos` as long as it is not on the leftmost lane.
   - `changeRight`: Change to the lane right of the car and increment its `car_street_pos` as long as it is not on the rightmost lane.
   - `accelerate`: Increment the cars velocity by one as long as it has not yet reached the maximum velocity. Increment the cars position by its current velocity.
   - `decelerate`: Decrement the cars velocity by one as long as it has not yet reached the minimum velocity. Increment the cars position by its current velocity.
   - `driveStraight`: Increment the cars position by its current velocity.

   The pedestrian should be able to move freely to adjacent positions in the north, south, east or west, as long as she stays within the boundaries of the model. Additionally, whenever the pedestrian moved, she has a probability of $\frac{2}{9}$ of tripping. If the pedestrian fell over, she needs two of her time steps to get up again.

   Furthermore you need to change the formula for `crash` such that it correctly computes whether the car has driven over the pedestrian, i.e. you need to account for the cells that the car has traversed when incrementing `car_street_pos`.

   ```
   sudo docker run --rm -v $(pwd):/media -it movesrwth/stormpy /bin/sh
   -c "python3 /media/car_pedestrian_simulator.py"
   ```

   After you have successfully modelled the above you need to evaluate your abstract model. You are going to use the provided simulator script to make `storm` compute a scheduler and evaluate the induced Markov Chain.

   Evaluate the following scenarios:

   (a) The car has a minimum velocity of 0. Report the first 5 frames of an execution trace of the induced Markov Chain. Also report the behaviour of the car that you've observed in a few sentences.

   (b) The car has a minimum velocity of 1. Report a complete execution trace of the induced Markov Chain.

2. [**5 Points**] Process Scheduling.

   For this exercise you will implement an abstract model of two processors and a scheduler.

   The scheduler behaves nondeterministically and decides which processor should execute the next task. Each task may depend on some of the previously computed results. This is already covered with the implementation in the skeleton file from the git repository, you will not need to make changes to the scheduler.

   Your task is to implement the behaviour of the two processors: As soon task n is being scheduled for execution on processor i, signaled by setting `task<n> = <i>`, it will change its status to adding or multiplying accordingly.

   The processor will increment its `add_tick_n` variable with the probability of `Pn_SUCCESS` as long as it is strictly smaller than `ADD_DUR_n`. As soon as `add_tick_n = ADD_DUR_n` the process will set its status to `done` and execute the `pn_done` action. This action is synchronized with a scheduler action, which in turn will set the process back into an idle state. This process has to be implement analogously for multiplying and both processors.

   The necessary constants are defined at the top of the file:

```
const int MULT_DUR_1 = 8;
const int ADD_DUR_1  = 4;
const double P1_SUCCESS = 1.0;
const int MULT_DUR_2 = 3;
const int ADD_DUR_2  = 3;
const double P2_SUCCESS = 0.8;
```

   Furthermore we want our processors to work in parallel. In order to tell `storm` to synchronize the computation between both processors you have to label all your newly added commands with the same action label, e.g. `sync`.

   Once you have finished the modelling part of this exercise, you can use the provided simulator script to compute a scheduler and evaluate the induced Markov Chain:

```
sudo docker run --rm -v $(pwd):/media -it movesrwth/stormpy /bin/sh
-c "python3 /media/task_scheduling_simulator.py"
```

The provided `prism` -file contains two reward structures:

```
// reward structure: elapsed time
rewards "time"
  task6!=3 : 1;
endrewards

// reward structures: energy consumption
rewards "energy"
  p1=0 : 10/1000;  // processor 1 idle
  p1>0 : 100/1000;  // processor 1 working
  p2=0 : 10/1000;  // processor 2 idle
  p2>0 : 100/1000;  // processor 2 working
endrewards
```

We want to evaluate different computed schedulers

(a) Report an execution trace for `"R{"time"}min=? [ F "tasks_complete" ];"` using the provided constants.

(b) Report an execution trace for `"R{"time"}min=? [ F "tasks_complete" ];"` using the following constants:

```
const int MULT_DUR_1 = 12;
const int ADD_DUR_1  = 12;
const double P1_SUCCESS = 1.0;
const int MULT_DUR_2 = 3;
const int ADD_DUR_2  = 3;
const double P2_SUCCESS = 1.0;
```

(c) Report an execution trace and constants for `"R{"time"}min=? [ F "tasks_complete" ];"` such that only processor 2 will be used.

(d) Report an execution trace for `"R{"energy"}min=? [ F "tasks_complete" ];"` using the provided constants, but change the success rate of processor 2, such that both processors execute 3 tasks each.