Project 2: Rush Hour

4 guys

May 2024

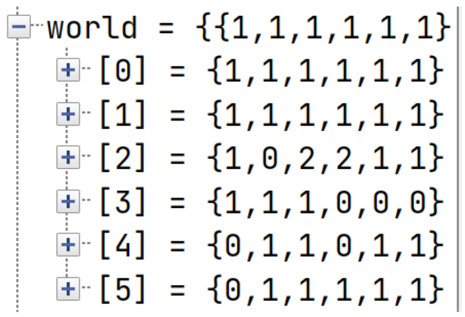
1 The Premise

Rush Hour is a solitaire game where the player moves cars and trucks forward and backward on a 6x6 grid to try and free a red car that is trying to leave. All cars have dimensions 2x1 and all trucks have dimensions 3x1. The difficulty lies in the fact that none of the cars can turn and that the grid is relatively small and filled with obstructive vehicles.

2 Our Model

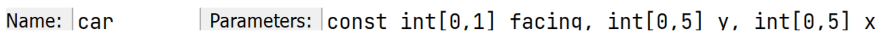
2.1 The Grid

The grid is modelled using a 6x6 matrix, called the world matrix. 0s on this matrix indicate no vehicle is in that location, 1s indicate a car or truck, and 2s indicate the red car. By having the processes that model our cars avoid entering tiles that are not 0s we avoid collisions and guarantee that the cars behave as they should.



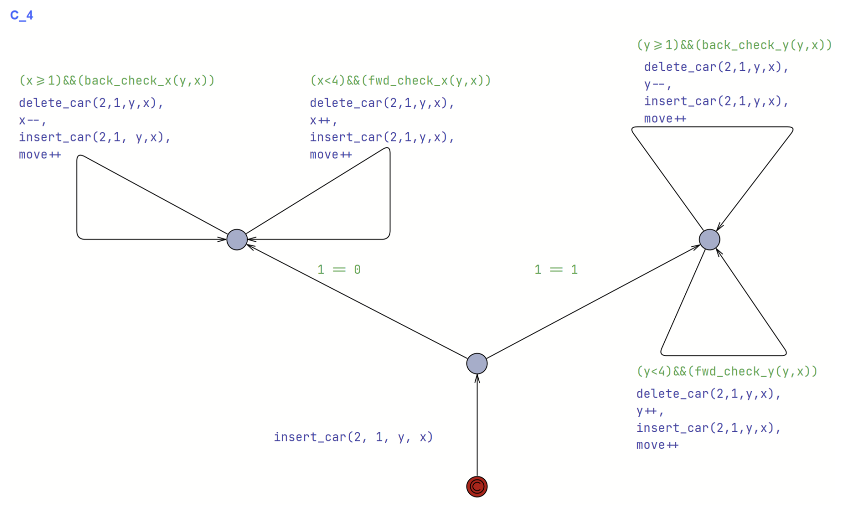
```
world = {{1,1,1,1,1,1}
  [0] = {1,1,1,1,1,1}
  [1] = {1,1,1,1,1,1}
  [2] = {1,0,2,2,1,1}
  [3] = {1,1,1,0,0,0}
  [4] = {0,1,1,0,1,1}
  [5] = {0,1,1,1,1,1}
```

Figure 1: The world matrix, initialized on the hardest configuration



```
Name: car        Parameters: const int[0,1] facing, int[0,5] y, int[0,5] x
```

Figure 2: The car parameters



```
C_4

(x ≥ 1)&&(back_check_x(y,x))
delete_car(2,1,y,x),
x--,
insert_car(2,1, y,x),
move++

(x<4)&&(fwd_check_x(y,x))
delete_car(2,1,y,x),
x++,
insert_car(2,1,y,x),
move++

(y ≥ 1)&&(back_check_y(y,x))
delete_car(2,1,y,x),
y--,
insert_car(2,1,y,x),
move++

1 == 0

1 == 1

(y<4)&&(fwd_check_y(y,x))
delete_car(2,1,y,x),
y++,
insert_car(2,1,y,x),
move++

insert_car(2, 1, y, x)
```

Figure 3: The car process

2.2 The Vehicles

Every vehicle in our model is modeled as its own process, all the cars share a template, and all the trucks share a template. We can conveniently determine if a vehicle can move vertically or horizontally, and determine its starting position on the grid by varying its parameters. The first parameter determines which direction the car is facing, with 0 indicating horizontal movement, and 1 indicating vertical movement. The next two parameters indicate the x and y positions of the front of the car. The red car has its own template because it only ever faces horizontally[1], so it doesn't need the edges for vertical movement. Additionally, it needs an extra edge and location that guarantees that it goes to the exit as soon as the opportunity presents itself. This is especially important because that is our winning state.

2.2.1 Initializing The Matrix

We use the *insert_car* function to insert our cars into the matrix. It checks if the car is facing vertically or horizontally and then fills in the appropriate entries of the matrix with 1s. The red car uses the *insert_rcar* function that fills in the matrix with 2s rather than 1s, and trucks use the *insert_truck* function, which has an added line of code that adds the third portion of the truck. To guarantee

[1]Updating the red car process to accommodate vertical movement would be relatively simple

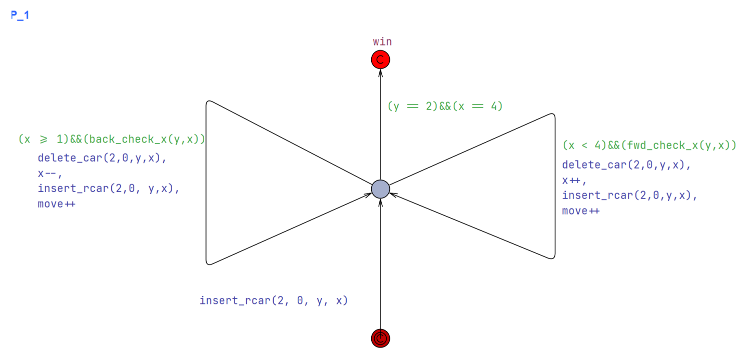


Figure 4: The red car process

```
void insert_car(int length,int facing, int y, int x){
    if (facing == 0){
        world[y][x] = 1;
        world[y][x+length-1] = 1;
    }
    else{
        world[y][x] = 1;
        world[y+length-1][x] = 1;
    }
}

void delete_car(int length,int facing, int y, int x){
    if (facing == 0){
        world[y][x] = 0;
        world[y][x+length-1] = 0;
    }
    else{
        world[y][x] = 0;
        world[y+length-1][x] = 0;
    }
}
```

Figure 5: The insert_car, and delete_car functions

that all the vehicles run through these functions before UPPAAL tries to solve the game all the vehicle processes begin on a committed location to force them to move through the edge that runs these functions.

2.2.2 Guards and Edges

Once the vehicles have all moved through the edges that initialize the matrix they can move to the most used portion of their processes. In non-red vehicles the process branches off into two sections, the guards on these branches guarantee that the vehicles go to the section that corresponds to the movement

```
bool back_check_x(int y, int x){
    if (world[y][x-1] == 0){
        return true;
    }
    return false;
}

bool back_check_y(int y, int x){
    if (world[y-1][x] == 0){
        return true;
    }
    return false;
}

bool fwd_check_x(int y, int x){
    if (world[y][x+2] == 0){
        return true;
    }
    return false;
}

bool fwd_check_y(int y, int x){
    if (world[y+2][x] == 0){
        return true;
    }
    return false;
}
```

Figure 6: The check functions

indicated by their faces. Once the vehicles reach the appropriate section each traversal of an edge corresponds to that vehicle moving 1 tile. These edges have guards that prevent them from leaving the grid, and guard functions that check to make sure the tiles they move into only contain 0s, as moving into any tile not containing a 0 would cause a collision. The update code on these edges moves the vehicles.

2.2.3 Moving

The vehicles in our model don't exactly move, they are repeatedly deleted from where they are, and then inserted where they're supposed to end up after moving one tile. This solution is elegant because it reuses the functions for initializing the matrix. The delete functions work relatively similarly to the insert functions, in that they check which direction the vehicle is facing and then turn the appropriate entries of the matrix into 0s. There is no *delete_rcar* function because the absence of a red car is the same as the absence of any other car in our model, they're all represented by 0s, so the *delete_car* function is sufficient.

3 The Query

$\exists \diamond P\_1.win$ This is our query for all versions of the problem. It simply asks UPPAAL to find a set of actions that allow the red car process (denoted as P_1) to enter its winning location, which is only accessible if the red car has already moved to the exit tile.

4 Problems

Solutions to configurations that we solved with the constraints of all of these problems can be found in our UPPAAL files, they are set up to initialize themselves, and running the query through the verifier will give the correct solution.

4.1 Problem 3

Problem 3 is the simplest version of the rush hour problem, our model has to traverse some relatively easy configurations of the grid.

- Puzzle card 13 is 32 moves.
- Puzzle card 21 is 49 moves.

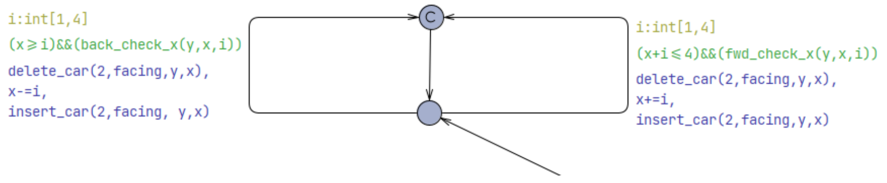

i:int[1,4]
(x ≥ i)&&(back_check_x(y,x,i))
delete_car(2,facing,y,x),
x-=i,
insert_car(2,facing, y,x)
C
i:int[1,4]
(x+i ≤ 4)&&(fwd_check_x(y,x,i))
delete_car(2,facing,y,x),
x+=i,
insert_car(2,facing,y,x)

Figure 7: The car process for bonus problem 1

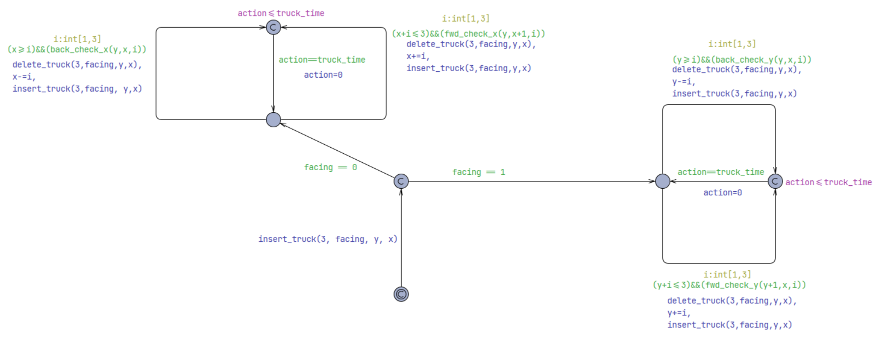

action ≤ truck_time
i:int[1,3]
(x ≥ i)&&(back_check_x(y,x,i))
delete_truck(3,facing,y,x),
x-=i,
insert_truck(3,facing, y,x)
action==truck_time
action=0
i:int[1,3]
(x+i ≤ 3)&&(fwd_check_x(y,x+1,i))
delete_truck(3,facing,y,x),
x+=i,
insert_truck(3,facing,y,x)
i:int[1,3]
(y ≥ i)&&(back_check_y(y,x,i))
delete_truck(3,facing,y,x),
y-=i,
insert_truck(3,facing,y,x)
facing == 0
facing == 1
action==truck_time
action=0
action ≤ truck_time
insert_truck(3, facing, y, x)
i:int[1,3]
(y+i ≤ 3)&&(fwd_check_y(y+1,x,i))
delete_truck(3,facing,y,x),
y+=i,
insert_truck(3,facing,y,x)

Figure 8: Truck process

4.1.1 Bonus Problem 1

The premise of this problem is that one step in our model should allow any vehicle to move any number of tiles, as long as moving to that location would be a legal move.
The overall architecture (e.g. the number of edges) remained the same, however, our functions did need to be modified. We implemented the *back_check* and *front_check* functions with for loops (we discovered some limitations of the Uppaal compiler which made this part a bit trickier) so that it was able to check for all of the tiles for the desired move. We added *select* onto our move edges so Uppaal could decide on the number of moves, and edited our guards accordingly so that the vehicle couldn't go out of bounds.

4.2 Problem 4

Problem 4 is seemingly identical to problem 3, however as the configurations are more complex our model struggled a little bit and we had to remove the move counter and replace it with a clock to reduce the state space.

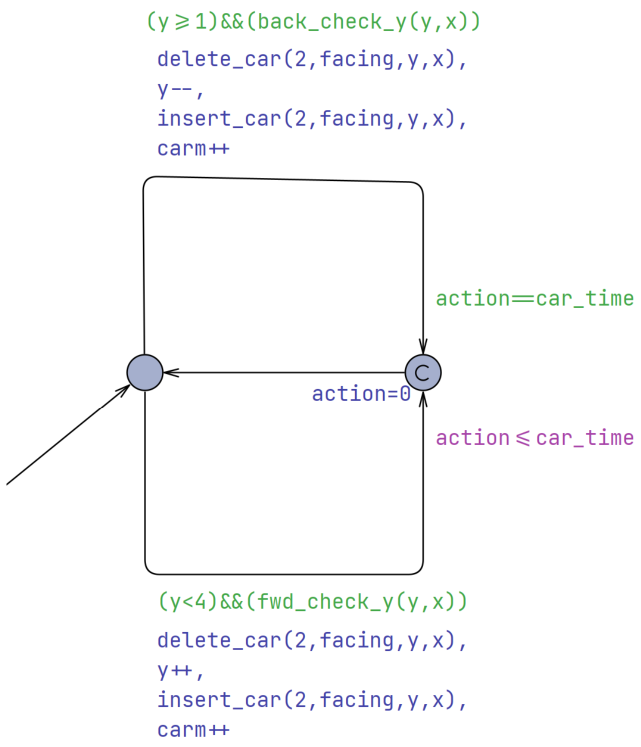

(y ⩾ 1)&&(back_check_y(y,x))
delete_car(2,facing,y,x),
y--,
insert_car(2,facing,y,x),
carm++
action==car_time
C
action=0
action ⩽ car_time
(y<4)&&(fwd_check_y(y,x))
delete_car(2,facing,y,x),
y++,
insert_car(2,facing,y,x),
carm++

Figure 9: The car process for problem 5 with the extra committed location

4.3 Problem 5

The premise of this problem is that it takes 2 seconds to move a car and 5 seconds to move a truck. Modelling this requires adding another committed location to all the movement edges, so that after a vehicle moves, it waits at that committed location the appropriate amount of time, and while it waits the entire model stops, and then the edge back resets the clock. This guarantees that the appropriate amount of time is left between all moves.

4.3.1 Super Bonus Problem

The premise of this problem is that you can move vehicles with both hands, but you're left-handed so your left hand takes 3 seconds to move a vehicle, while your right hand takes 5 seconds to move a vehicle. We modelled this by having each hand be its own process with synchronisation for R! and L! respectively. Then each edge that moves a vehicle needs an R or an L to be

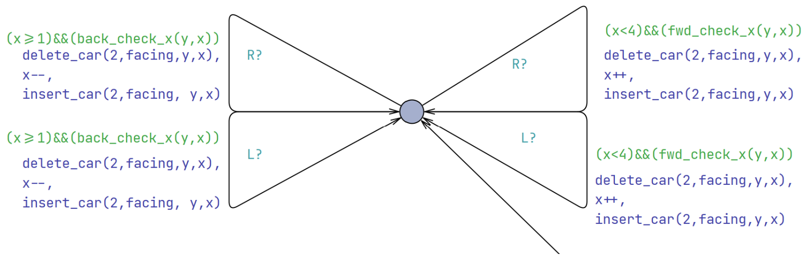

(x⩾1)&&(back_check_x(y,x))
delete_car(2,facing,y,x),
x--,
insert_car(2,facing, y,x)
R?
(x<4)&&(fwd_check_x(y,x))
delete_car(2,facing,y,x),
x++,
insert_car(2,facing,y,x)
R?
(x⩾1)&&(back_check_x(y,x))
delete_car(2,facing,y,x),
x--,
insert_car(2,facing, y,x)
L?
L?
(x<4)&&(fwd_check_x(y,x))
delete_car(2,facing,y,x),
x++,
insert_car(2,facing,y,x)

Figure 10: The car process with duplicated movement edges

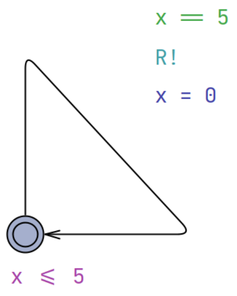
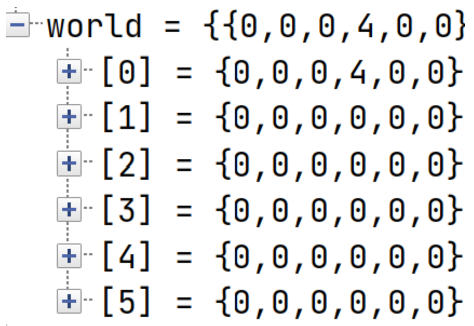


Figure 11: The right-hand process, with synchronisation R!

traversed. As UPPAAL does not allow two synchronisations to be on the same edge this required duplicating every movement edge in every process in the model. Modelling it this way means that one vehicle could vacate a tile at the same time as another vehicle enters it, which would not be physically possible while sliding objects around on a board. This means solutions generated by this model may not be usable in the real world, or may be a few seconds slower in the real world.[2]

Using this model we solved Rush Hour's hardest configuration 1.1, it took a minimum of 175 seconds, and 93 moves in total. Using this version of the model to solve other configurations would require minimal setup.



```
world = {{0,0,0,4,0,0}
  [0] = {0,0,0,4,0,0}
  [1] = {0,0,0,0,0,0}
  [2] = {0,0,0,0,0,0}
  [3] = {0,0,0,0,0,0}
  [4] = {0,0,0,0,0,0}
  [5] = {0,0,0,0,0,0}
```

Figure 12: The grid with a wall in the position for the first wall problem

4.4 Problem 6

The premise of this problem is that there are walls on the grid, these walls take up space exactly like a car except they're 1x1 in dimension and cannot move. Modelling these walls is easy in our model, we simply add a 4 into the grid matrix wherever we want a wall to be, and then let UPPAAL take care of the rest.

[2]Avoiding this degree of freedom would require each vehicle to occupy the tile it's leaving, as well as the tile it's moving into for the entire duration of its movement. This would require altering the vehicle processes, and the functions that move vehicles considerably.

4.5 Problem 7 - Quality

Checking the quality of the model in regards to the criteria discussed by Frits Vaandrager in the article What is a Good Model.

- The models have a clearly specified **object of modelling** and are truthful. The rush hour game can be represented one-to-one by a two-dimensional matrix. In our model the empty spaces are represented by 0s, the red car is represented by 2s and all other cars are represented by 1s. When tackling the more difficult problems time is represented by UPPAALs time module and different hands were represented with their own processes.
- The models have a clearly specified **purpose**, which is to find the best ways to win different rush hour configurations within the boundaries set forth in the problem description.
- Our models are somewhat traceable, we traded simplicity for **traceability** when we made all non-red cars be represented by 1. It could be possible to represent different cars with characters or different numbers but we judged that the added layer of complexity was not worth it. They were however traceable enough that we noticed our cars jumping over other cars on our iteration of the multiple-movement problem.
- The models are **simple** which helps them be **extensible and reusable**. Re-usability was tested and proven when we could, quite quickly, switch between different setups and solving the hardest problems was only a matter of plugging in the correct starting coordinates. The extensibility of the base models was also tested and proven when problems such as time and two hands were tackled.
- The simplicity of the model facilitates **interoperability and sharing** but we did not specifically design around it and we did not collaborate with other groups in order to determine how easy interoperability would be.

4.6 Final Thoughts

We quite enjoyed using UPPAAL to model for the assignments and by the end of the course had gained some proficiency in using it to model problems. The rush hour assignment was somewhat more challenging to draw out on a whiteboard compared to Wooldridges vacuum robot which led to us placing greater emphasis on checking the quality of the model instead of drawing out solutions and working out improvements to the model that way. We did run into some problems with syntax constantly and there were a few times where we sorely needed a debugging tool to more easily see what was going wrong.