T-219-REMO
Project 2 – Rush Hour
Group: Coffee Machine

Guðmundur Egill Árnason Hreiðar Ólafur Arnarsson
Michaela Nerrether Sandra Ösp Stefánsdóttir
Maciej Sierzputowski

11th of May 2017

Table of Contents

1 Introduction

We were asked to model the board game Rush Hour and use UPPAAL to verify a winning strategy for a set of puzzles.

We ended up making four different multi-process models, two non-timed and two timed. In this report we will show the different models, explain them and show their differences. Then we will show how we used them to solve four Rush Hour puzzles of increasing difficulty and complexity along with results of running those puzzles through the different models. Finally we will show what conclusions we take from the results and the project in whole.

2 Specifications

2.1 Rush Hour the Game

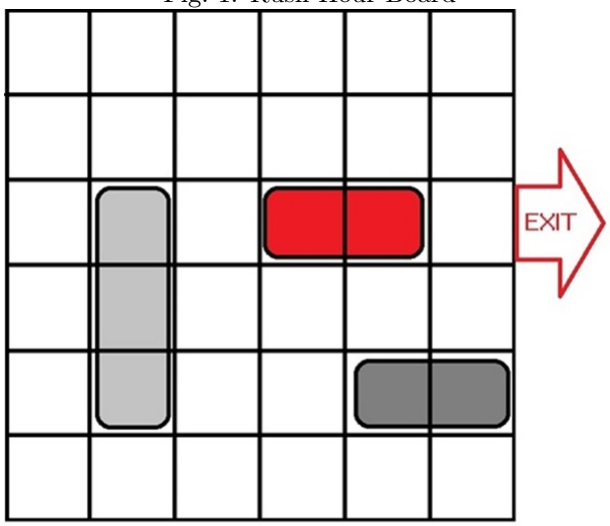
2.1.1 About the Game

Rush Hour is a sliding block puzzle game that is played on a 6 x 6 board. On the board there are a number of vehicles of different sizes (cars are of size 2 x 1 and trucks are of size 3 x 1). The vehicles are not able to turn, they can only move backwards and forwards in their respective direction.

2.1.2 Goal of the Game

The purpose of the game is to move the vehicles by driving forwards and backwards in such a way that the red car can leave the board at the exit on the right side of the board (see Fig. 1).

Fig. 1: Rush Hour Board



2.2 Tool

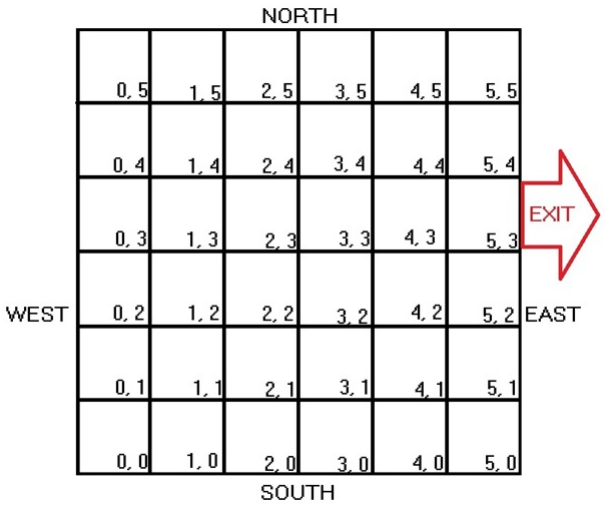
For this project we used Uppaal. Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables[1].

[1] http://uppaal.org (accessed May 2017)

2.3 Notes

- Sizes are for 2 squares for a car and 3 squares for a truck.
- X-Coordinate = 0, 1, 2, 3, 4, 5.
- Y-Coordinate = 0, 1, 2, 3, 4, 5.
- Directions = 0, 1, 2, 3 for North, East, South and West. North = 0 etc.

Fig. 2: Board Structure



2.4 Vehicle Capabilities

1. Each vehicle knows its location on the board.
2. Each vehicle can move forward and backward, if nothing occupies the square in question (and it is not out of bounds).
3. Turning the vehicle is impossible.
4. The vehicles can only move one position at a time.

3 Non-timed Models

3.1 The Back-and-Forth Model

This automaton models a vehicle on the board. At the very beginning of the game, each car is initialized (placed on the board) one-by-one. A vehicle will only start moving once every other vehicle has been placed on the board.

A vehicle may have the opportunity to go back or forth at each turn, depending on whether there is an open square on the board behind or in front of it. This continues until the red car has reached its destination, with its trunk set to the exit coordinates.

In order to ensure the integrity of the board this model allows only one car movement at a time.

We declared a global boolean matrix for the board and also an integer value to log how many steps it took too complete the puzzles.

```
bool board[6][6] := { {0,0,0,0,0,0},
                      {0,0,0,0,0,0},
                      {0,0,0,0,0,0},
                      {0,0,0,0,0,0},
                      {0,0,0,0,0,0},
                      {0,0,0,0,0,0}
                    };
int steps;
```

In the local declarations section we created four integer variables, two for the location on the x and y asis, one for each asis, and then two for the forth and back to set the directions the car is facing.

```
int  forth, back, x_location, y_location;
```

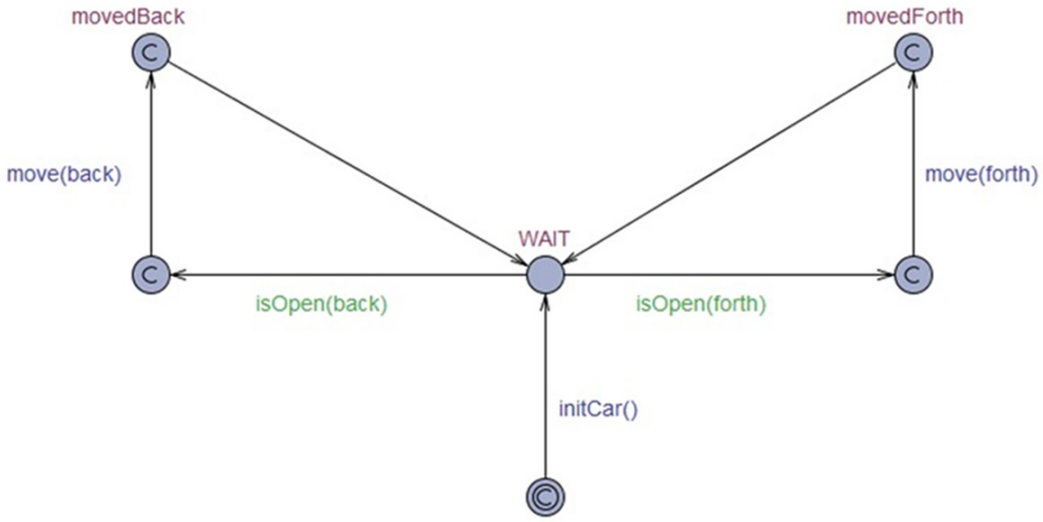


Fig. 3: The Back-and-Forth Model Automaton (Non-Timed)

3.2 Trinity Model

We declared a few global variables in Uppaal's Declarations section.

```
int grid[6][6];
bool gameCreated = false;
bool playersCreated = false;
int vehNum = 0;
int vehicles = 0;
int steps = 0;
```

The `grid` two dimensional array sets the grid we will be using. With it we will be able keep track of where each vehicle is located on the board. The boolean variables `gameCreated` and `playersCreated` are used to make the automata know when they are allowed to enter the moving of vehicles phase. The purpose of the integer variables `vehNum` and `vehicles` is to keep track of the total amount of vehicles added to the game board before being able to play the game. Finally, the integer variable `steps` keeps track of how often vehicles have been moved.

In Uppaal's System Declaration we declare the vehicles to be populated on the game board.

The model is split into three automata; *Game*, *Vehicle* and *Player*. *Game* (Fig. 4) sets up the game board by running the function `initialize()` which fills the `grid` array with zeroes. *Game* also accepts the variable `vehs`, which represents the number of vehicles on the board. During initialization the variable is used to set the global variable `vehNum`. Additionally the global variable `gameCreated` is set to `true`.

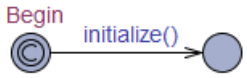


Fig. 4: Game Automaton

Vehicle (Fig. 5) populates the game board with all the vehicles except for the red car. It accepts a few variables namely the x-axis position, y-axis position, length and the direction the car is facing. It then uses these variables to add the car to the given (x,y) position on the board by calling the function `create()` for each vehicle created in the System declarations section. For each vehicle the previously mentioned variable `vehicles` gets increased until it reaches the `vehNum` variable which is the total number of vehicles on the board. Once that is fulfilled the global variable `playersCreated` is set to true which allows the vehicles to be able to perform their moves on the board. The move functions (`moveUp()`, `moveDown()`, `moveLeft()` and `moveRight()`) are responsible for the movement of the cars, while the check functions (`checkUp()`, `checkDown()`, `checkLeft()` and `checkRight()`) are responsible for checking if the movement is possible; such as making sure the car doesn't move off the grid or move onto a grid space already occupied by another car.

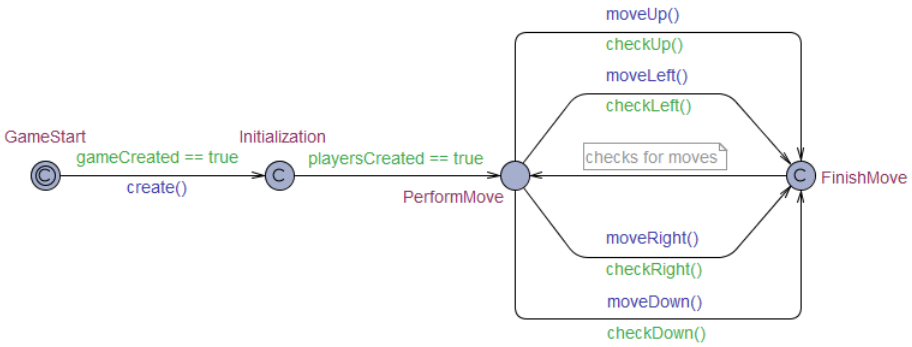


Fig. 5: Vehicle Automaton

Player (Fig. 6) populates the game board with the red car. Code and functions look very similar to the code and functions for *Vehicle* except that *Player* has static values for y-axis position, direction and size. The red car gets added to the board just like the vehicles but it uses a local version of the `create()` function instead which is the same as the *Vehicle* one but it has statically predefined values except for the x-axis one. It also increases the value of the `vehicles` variable as the red car is also a vehicle. The move and check functions are reduced only to left and right movements and checks as the the red car doesn't move vertically. The `playerWin()` function checks if the red car

has reached the winning position which is the 4th position on the x-axis. We used that function as a query to check if the player wins the game.

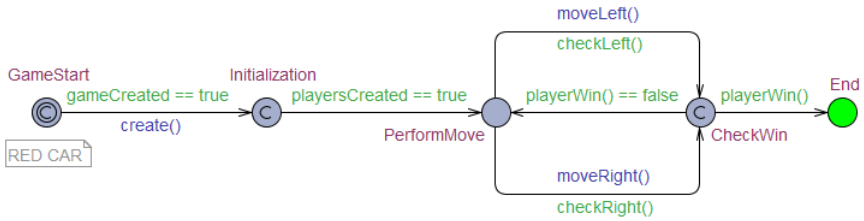


Fig. 6: Player Automaton

3.3 Differences

The main differences are the number of modelled automata in each model. The Back and Forth model only uses one modelled automaton while the Trinity model uses three.

The Back and Forth automaton only considers movement of a vehicle to be back and forth (hence the name of the model) no matter which direction the vehicle is facing. It uses only one check function and one movement function that are used for all directions. In the Trinity model, however, the *Vehicle* automaton has one check function and one movement function for each direction a vehicle is facing. The *Player* automaton for the red car only has move functions and check functions for left and right.

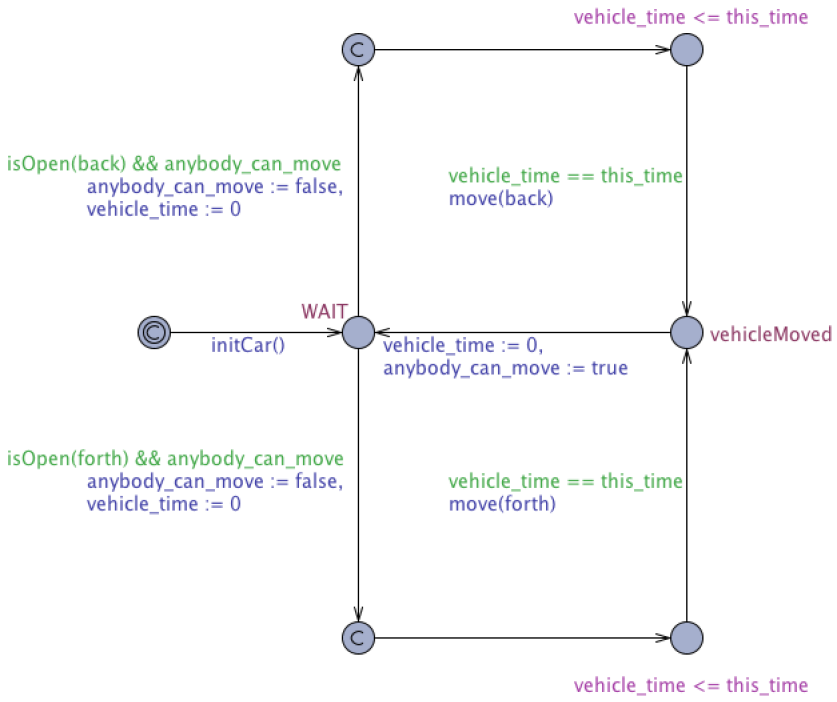
While both models use an array to represent the game board they use the arrays differently. The Back and Forth model uses a boolean array to keep track of unoccupied spaces while the Trinity model does that as well as keeping track of the red car in an integer array. The Back and Forth model does not explicitly keep track of where the red car is on the board. It assumes that the user of the verifier will specify which car to check for which condition (e.g. red car at certain coordinates).

4 Timed Models

4.1 The Timed Back & Forth Model

In the timed version we have that a car takes 2 time units and a truck takes 5 time units. In order to account for this—difference dependent on vehicle type—we use the size property. Since a vehicle's size is sufficient to uniquely identify it as either a car or a truck, we check it at the initialization step and then set a local variable named "this_time"to either 2 or 5.

Fig. 7: The Timed Back & Forth Model



4.2 Trinity Timed Model

In the timed model we added two clocks and a boolean variable in the global declaration section.

```
clock total;
clock time;
bool waiting;
```

The clock variable `total` is responsible for counting the total amount of time units it takes to solve a given puzzle, while the `time` variable is responsible for the time a vehicle waits after performing a move and the boolean variable `waiting` stops the other vehicles from performing actions while a vehicle is in the waiting state.

In the *Vehicle* (Fig. 8) automaton we added additional states, the `CarWait` state and the `TruckWait` state. After a car performs a move he ends in one of the corresponding states depending on whether the vehicle is a truck or a car. He then waits there for either two or five time units before moving out of the waiting state and setting the `waiting` variable to false, then any car is able perform his next move.

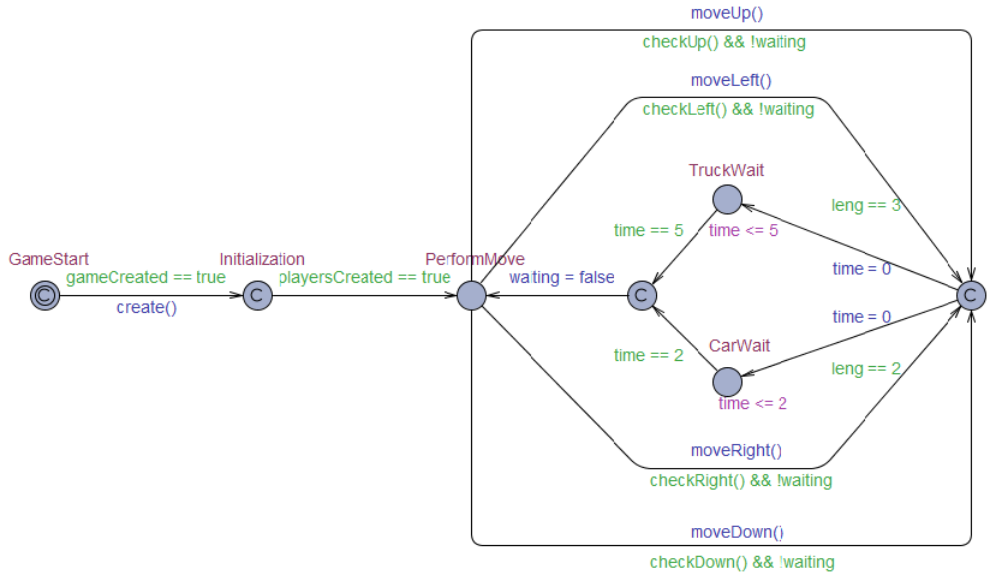


Fig. 8: Timed Vehicle Automaton

The *Player* (Fig. 9) automaton has a single waiting state because the cars size is fixed so when he performs a move he always waits for 2 time units. The same global variables `time` and `waiting` are used in the automaton so it works concurrently with the *Vehicle* automaton.

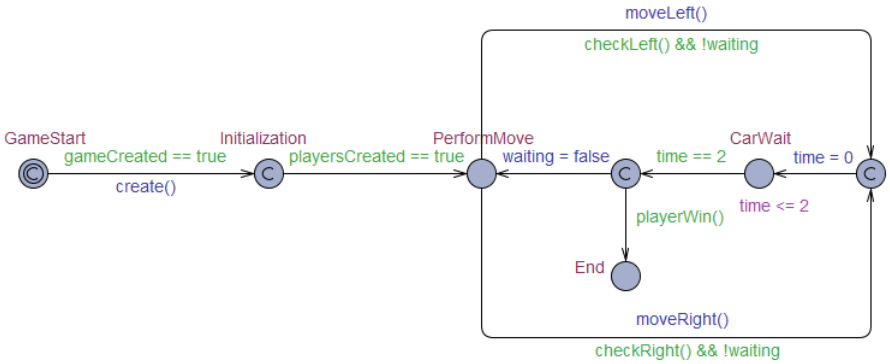


Fig. 9: Timed Player Automaton

4.3 Differences

The main difference is still the amount of automata in each model, the Trinity model still has three automata while Back and Forth still has only one.

The automaton in Back and Forth sets all its clock or time unit related settings in a function while Trinity does it in the automaton itself, both for *Vehicle* and *Player*.

Back and Forth has a time check for each movement and only executes the move in question when the set time has be achieved while Trinity does a time check after a vehicle has moved.

5 Puzzles to solve

We were given three puzzles to solve, each of an increasing difficulty and complexity. The images in this chapter depict the starting positions of all the cars and trucks for each puzzle. We will also show the results when checking the puzzles against the non-timed (shortest) and timed (fastest) models. We also include the running time for each puzzle on each model. We made sure to run all the models on the same computer to minimize variations related differences in hardware, drivers and other software.

5.1 Intermediate Puzzle

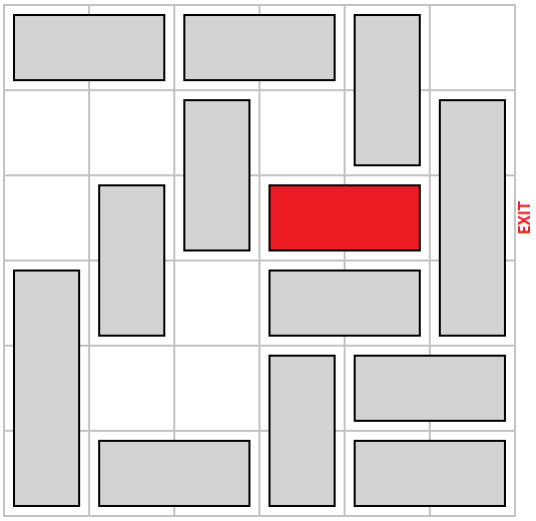


Fig. 10: Intermediate puzzle

Back and Forth Model

- Shortest: 32 steps running for 11.724 s
- Fastest: 76 time units in 32 steps running for 17.065 s

Trinity Model

- Shortest: 32 steps running for 11.54 s
- Fastest: 76 time units in 32 steps running for 14.077 s

5.2 Advanced Puzzle

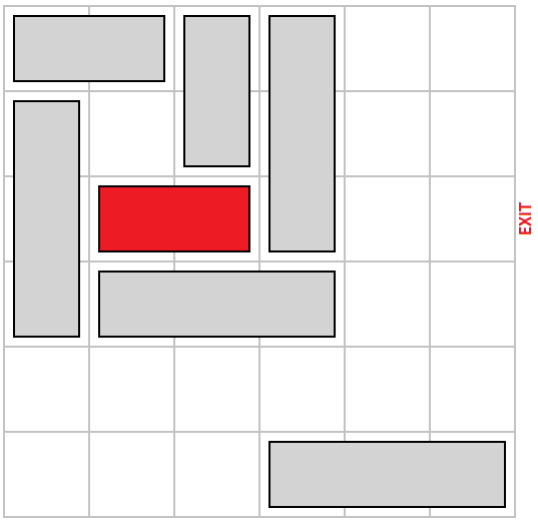


Fig. 11: Advanced puzzle

Back and Forth Model

- Shortest: 49 steps running for 583 ms
- Fastest: 188 time units in 49 steps running for 1.0376 s

Trinity Model

- Shortest: 49 steps running for 547 ms
- Fastest: 188 time units in 49 steps running for 1.035 s

5.3 Hardest Puzzle

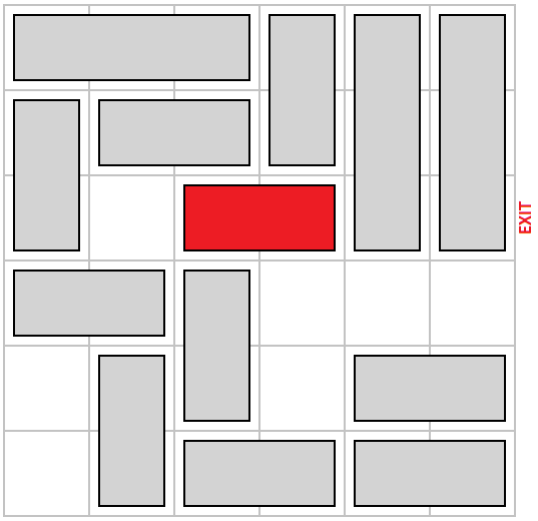


Fig. 12: Hardest puzzle

Back and Forth Model

- Shortest: 93 steps in 56.298 s
- Fastest: 240 time units in 93 steps running for 70.329 s

Trinity Model

- Shortest: 93 steps in 52.505 s
- Fastest: 240 time units in 93 steps running for 68.881 s

5.4 Loading Puzzles into Models

To switch between puzzles we had to set up the system declarations for each puzzle. We ended up making documents to ease the input vehicle declaration. For the Back and Forth model the file is `BackAndForth-puzzles.txt` and for the Trinity model the file is `Trinity-puzzles.txt`.

6 Queries

6.1 Back and Forth Model

The same query was used for the non-timed and timed versions.

```
E<> Car0.x_location == 4
```

The query checks if there exists a computation for some state, in which Car0 (the red car) arrives with its trunk at x coordinate 4. This is the victory condition.

6.2 Trinity Model

We used different queries for the non-timed and timed versions for the Trinity model. The main reason for not using the same query was due to an Uppaal error not recognizing a boolean function with the fastest diagnostic trace active.

Non-timed Trinity Model

```
E<>(player.playerWin() == true)
```

If there exists a strategy where *playerWin()* in the *Player* automaton returns a true value then this query should give us a result. The *playerWin()* function returns true if the "backendǫf the red car reaches the second to last column, which means that the frontend of the red car is at the exit.

Timed Trinity Model

```
E<>(player.End)
```

We gave the final state in the *Player* automaton the name *End*. The query above should give us a result if there exists a strategy where that *End* state can be reached.

7 Results

Here we have the results from running the three puzzles through our models.

Model		Steps	Time units	Run time (s)
Back and Forth	Non-timed	32	-	11.724
	Timed	32	76	17.065
Trinity	Non-timed	32	-	11.540
	Timed	32	76	14.077

Table 1: Intermediate Puzzle

Model		Steps	Time units	Run time (s)
Back and Forth	Non-timed	49	-	0.583
	Timed	49	188	1.0376
Trinity	Non-timed	49	-	0.547
	Timed	49	188	1.035

Table 2: Advanced Puzzle

Model		Steps	Time units	Run time (s)
Back and Forth	Non-timed	93	-	56.298
	Timed	93	240	70.329
Trinity	Non-timed	93	-	52.505
	Timed	93	240	68.881

Table 3: Hardest Puzzle

8 Conclusion

We found that both models find the same amount of steps for the shortest strategy and the same amount of time units taken. However, we also found that the models vary in their runtime by some (tiny) amount.

For each puzzle we get the same amount steps for each model; 32 steps for the Intermediate puzzle, 49 steps for the Advanced puzzle and 93 steps for the Hardest puzzle. We already know that the solution for the Hardest puzzle is supposed to be 93 steps[2].

For each timed variant of the model we get the same time units for each puzzle; 76 time units for the Intermediate puzzle, 188 time units for the Advanced puzzle and 240 time units for the Hardest puzzle.

This makes us come to the conclusion that the results we are getting are the correct results.

[2]https://quomodocumque.wordpress.com/2012/02/18/the-hardest-rush-hour-position/

When it comes to the difference in running time we cannot dismiss the possibility of outside programs having an effect. Since we did not have access to a clean testing machine it is difficult for us to determine exactly whether the cause lies in the model designs themselves or not. We do know that there is a difference in the amount of `if` statements and `for`-loops between the main models; Back and Forth, and Trinity. This gives us an inkling of where we could dig deeper in analyzing the diagnostic traces.

It is worth mentioning that it is possible to drastically reduce the number of if-conditional calls in the Back and Forth model as for every isOpen() call, which has an if-conditional, there is a move call(), which uses the very same if-conditional. Thus for every movement now made, the model might have one less if-conditional.

9 Final thoughts on working with Uppaal

My grandfather once said: "Follow your dreams". Little did he know I would find them in the form of a model verification tool named Uppaal. At first glance it looks like a simple tool which is meant to solve some easy puzzles but once one gets to know it, it can be quite addictive, with its rapid and powerful feedback for varyingly complicated problems. Not only does one see that the sky is the limit, but one actually begins to see where the sky is (in computing terms). One problem after another and one falls deeper into the rabbit hole. From our experiences of the Vacuum Cleaning Problem, The Solitaire Game Problem to the Rush Hour Game Problem, we have willingly dove deep into the cave of Uppaal. The rush of joy (perhaps even adrenaline), that one gets upon successfully verifying a query, is significant. To work with Uppaal is to work with a tool which brings you increased success, and that's what it feels like to work with Uppaal, success.