POLYCREATE - Finite State Machine Project SI4

Household robot controller

Realised by : *Hiba Khayati & Nadine Carel AZEBAZE ZEWO*

Supervised by : M. Julien Deantoni

Année : 2021/2022

Table of contents

I - Minimal Viable Product

For this project, we have fully implemented the MVP. We have chosen to implement the extension: “Storage of objects in the scene”; however, we did not complete its implementation: The robot can grip objects and its position is saved so that he can come back to his last position.

What was asked for the MVP:

For the minimal version of our project, the robot must move in the environment avoiding obstacles as much as possible and without getting stuck, without falling in stairs or holes (if there are any) and avoiding hitting objects. Also, it should not cross the virtual walls that delimit spaces where the robot should not go. Finally, it must cover the room as much as possible so that it is clean.

II- Work Organization

At the beginning of the project, we first chose our extensions and ranked them according to our choice of priorities; then we divided the work into two. We have divided among ourselves the different features of the MVP; two of which are each. As the weeks went by, each of us moved forward on his features; switching on a few occasions when one of us was stuck too long.

III- Justification for choices of implementation

1. Avoid obstacles

The main purpose of this robot is to clean the space it’s in. To do that, it must have the capacity of moving around the space freely and thus avoiding obstacles. To achieve this aim, the robot checks his surroundings as he comes across them. When he comes across an obstacle, he then tries to avoid it by going back or turning until no obstacle is found in his direction.

a. Architecture:

To avoid obstacles, the robot detects them by using the distance sensors instead of waiting to bump into them to detect them. This facilitates the ‘non hitting objects’ part of the MVP.

In the special cases where the sensors cannot detect the obstacles (the objects for instance such as the bottles that are considered as a sous-group of the group of obstacles).

The robot's system bases his analysis on whether there is an object seen on the front camera to go grip and thus move freely or worst-case scenario there is a collision and thus an obstacle to escape from.

b. Problems encountered / modifications to the environment given and further comments:

One big problem encountered in this part wasn't a state chart issue. Whenever the robot starts its life cycle, it detects a collision and starts turning even if there is none in the environment.

After some analysis of the code written, the state chart and the environment. The problem seems to rise because the robot spawns with a collision with the ground. Which is detected by one of the robot's bumpers. This collision is then analyzed as a random normal one and thus the robot makes this unexpected turn before starting anything else.

The solution found to prevent this issue is rather simple, but we judged the necessity of noting it here in case any wonder about the modification of the environment arose. So, it is very probable, that in the experimentation, the robot will start with a sudden turn instead of going straight.

c. State chart:

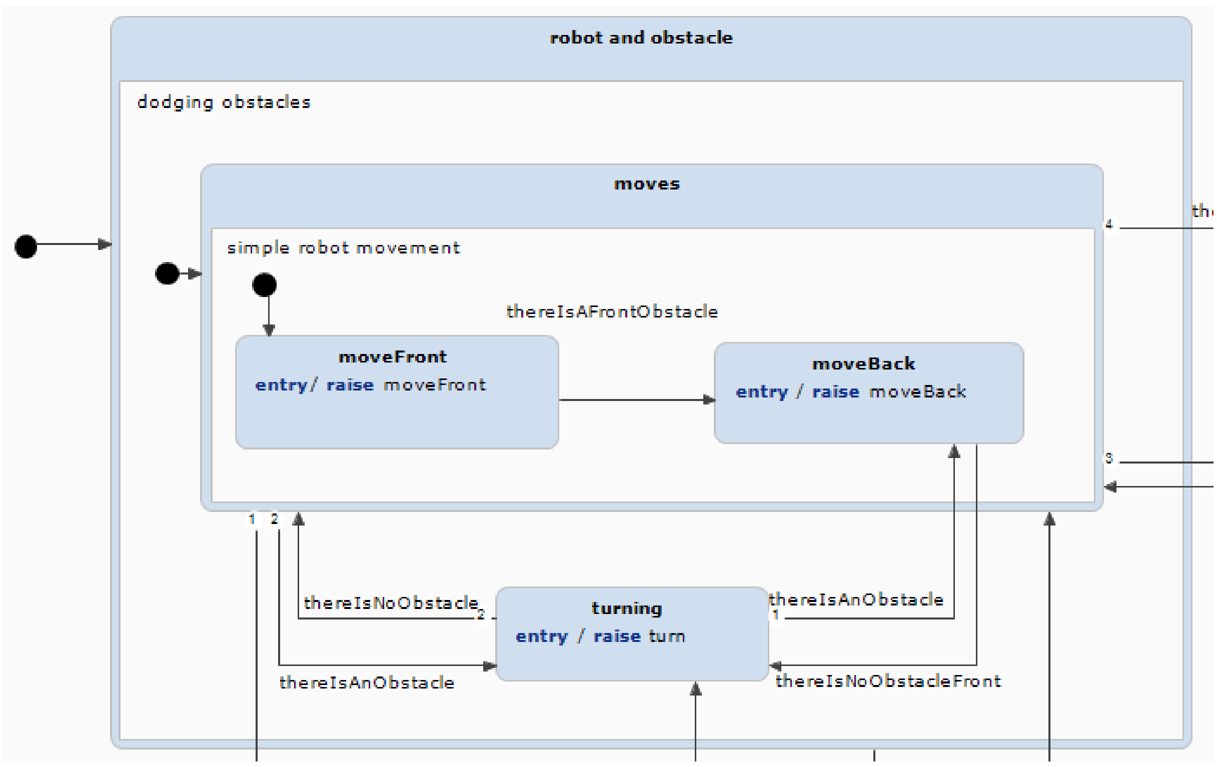


Figure 1: State chart for obstacle dodging

2. Falling in stairs or holes

It is imperative for the robot to detect when it is approaching a hole, a stair, or any other type of cliff so that the system can turn and change directions to prevent a sudden fall.

The robot is mounted 1 cm above the floor; therefore, the robot will halt and change directions if a distance less than 1 cm is detected.

a. Architecture:

The solution that we implemented for the avoidance of holes and voids (stair steps) actually involves the sensors of the robot notably: "cliff sensor". These sensors located under the robot make it possible to control the support on which the robot moves. This solution is quite similar to the one proposed for the problem "collision with objects". Here too, the bulk of the work (control) is done in the code and thanks to the statechart the robot knows in what state it enters if ever it encounters a hole or a vacuum. This solution is all the easier to implement because it does not require us to create new states; those already present are enough; we just need to structure the order of passage in each state according to the reaction we wanted our robot to have.

To avoid falling into a hole or a gap. The robot checks the value returned by its Cliff's sensors. Those 4 sensors are Distance sensors located under the robot. If any of these sensors is situated near a cliff, the value of the sensors switches to 0.

b. Problems encountered / modifications to the environment given and further comments:

One problem we have encountered is to avoid stairs. Indeed, if it is a hole on the ground, the robot avoids it easily. On the other hand, if it is a step and the robot is heading towards the front vacuum, it falls or freezes with a ration of 2/4 because its wheels which have to make it reverse then turn no longer touch the flat surface. In the others especially if it meets the vacuum being inclined in relation to the orientation of the step; it avoids it easily. Because of this malfunction, we believe that our solution here could be reviewed and improved

c. State chart:

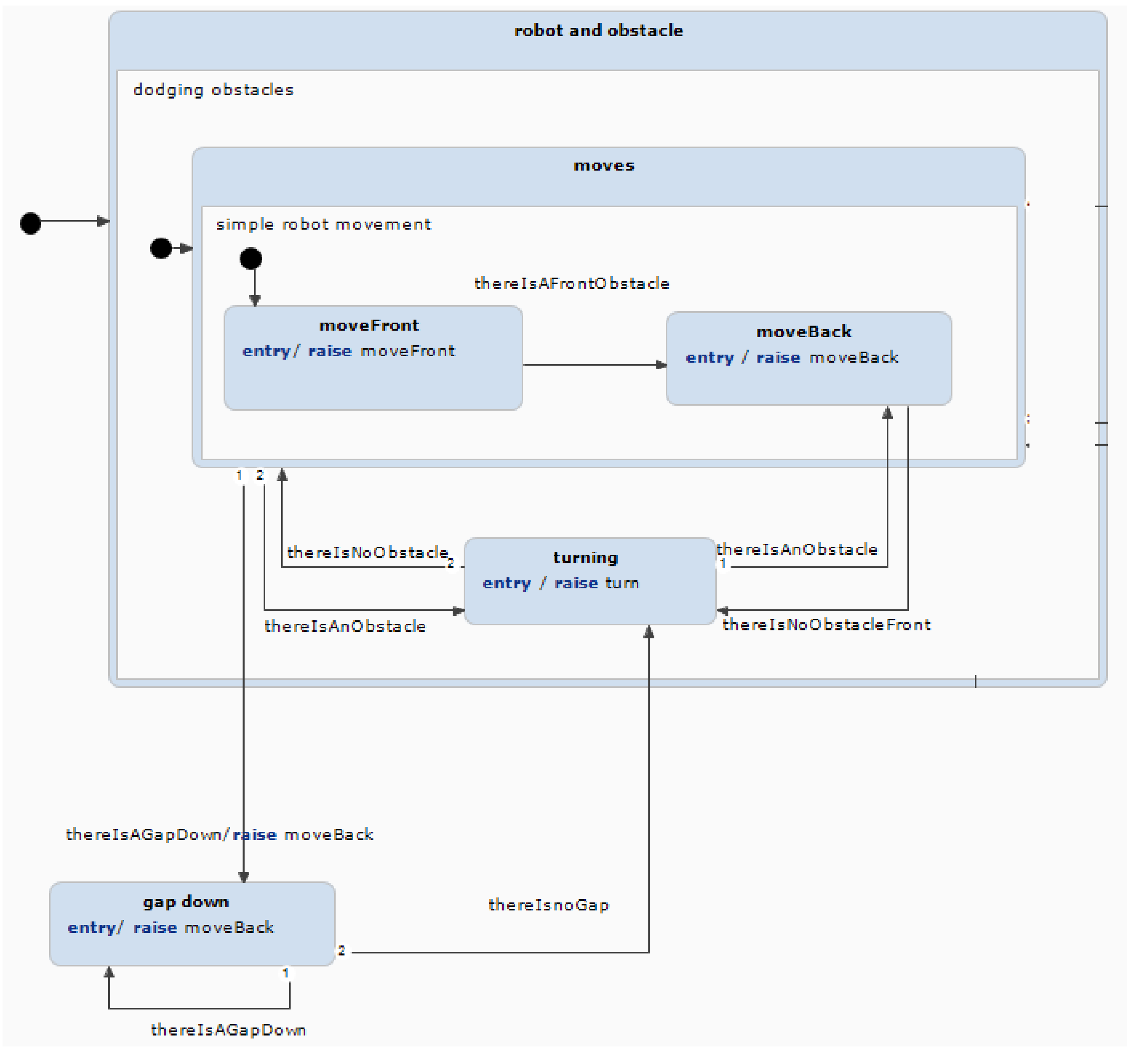


Figure 2 : State chart for gaps and cliff recognition

3. Not crossing virtual walls

A virtual wall is a barrier that the robot is not allowed to cross. Once coming across a virtual wall, the robot considers it as an obstacle that he cannot go past.

a. Architecture:

Most of the implementation in this part is a control rather than code. Therefore, the state chart does most of the work if not all.

A consideration was given to the case in which a robot is between three virtual walls.

The robot makes a 180° turn whenever it encounters a virtual wall. In the case where there's another virtual wall right across him (almost impossible to happen in a real-life scenario) the robot cannot make a 180° turn again as he will stay in that state of turning back and forth forever. He therefore turns at around a 90° angle before checking again if there's a virtual wall. If so, he then turns one last time to continue the cleaning process.

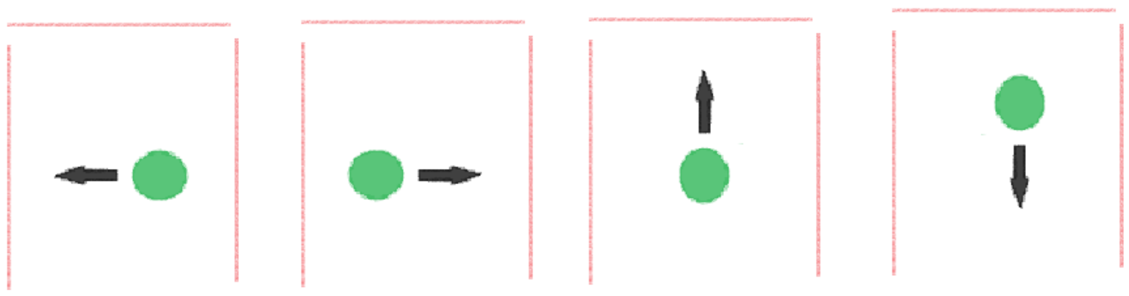


Figure 3 : Three virtual walls scenario

b. Problems encountered / modifications to the environment given and further comments:

As noted, the limits of our project analyses this highly unprobeable case. We do not consider the case of 4 virtual walls englobing the robot as we judge it is not only unlikely (unless done for "fun" by a crazy and lonely user).

c. State chart:

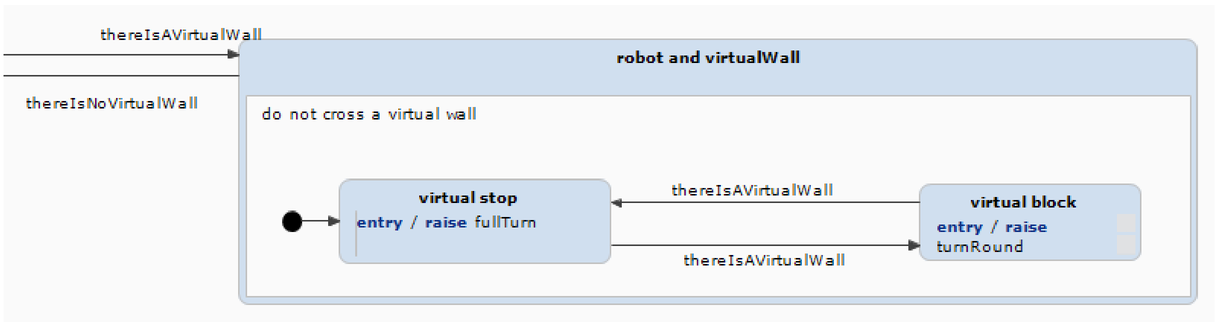


Figure 4 : State chart for virtual walls

4. Not hitting objects

In consideration of the environment the robot is in, not to hit objects is a very important part of the cleaning process. In real-life, a robot that cleans your space by pushing your vases until they all shatter into pieces like your sanity, is... lame. And for sure, no one is purchasing a robot like that. This part is linked to the storing objects in the scene extension.

So the robot doesn't hit objects. The control of the environment by means of the sensors has been set up so that the robot can detect a possible collision with an object before entering it; whether from the front or from the sides.

a. Architecture:

This solution is implemented directly in the code; indeed it does not need to implement additional states in the state chart since it is about control. Moreover, the initial code already contained methods such as isVirtualWall(), isCollisionLeft() ... which allowed us to manage this problem more easily This solution is implemented directly in the code the control resides more on the position of the object in the camera.

A “checkHelper” class plays the role of a “listener” for on the environment. It takes the environmental data and checks the presence of anything remotely interesting to react to. The control here was dependent on the code; meaning that using the relative position of the object to the camera and a bunch of trial and error, we have found a minimal distance which seems to be optimal in order to dodge or not an object.

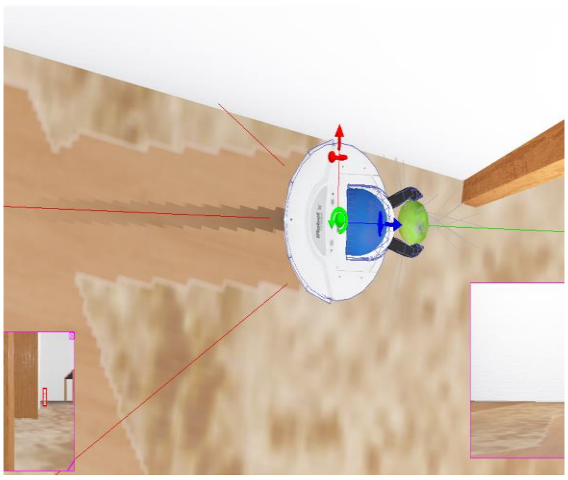
b. Problems encountered/ modifications to the environment given and further comments:

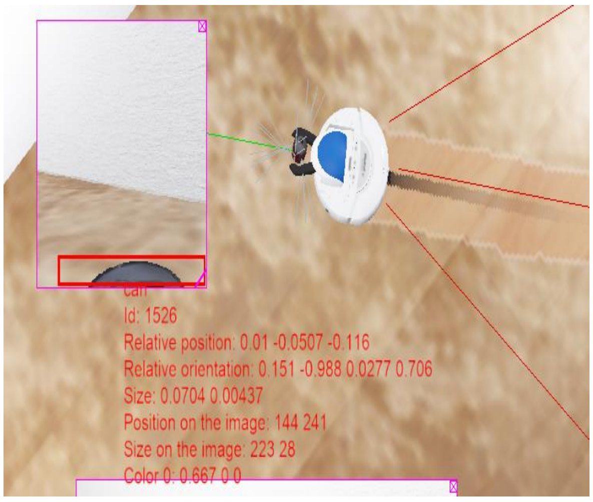
One problem encountered was mainly the use of sensors. Indeed, with some parts of the documentation being a little bit hard to wrap our head around, the understanding of the role of each sensor and its use was rather ambiguous at the beginning of the project

II – Extensions

1. Storing objects in the scene

This extension controls the robot so that it moves the movable objects in the scene and brings them to a predefined place of choice. To do this, the gripper on the back is used.





```
can
Id: 1526
Relative position: 0.01 -0.0507 -0.116
Relative orientation: 0.151 -0.988 0.0277 0.706
Size: 0.0704 0.00437
Position on the image: 144 241
Size on the image: 223 28
Color 0: 0.667 0 0
```

a. Architecture:

We chose the extension "storage of objects in the scene". At this stage of the project, we could not complete this extension. The robot detects objects and knows how to grasp them: indeed, once at a certain distance from the object, the robot makes a complete half-turn and then detects (using the "checkHelper") that the robot is well positioned in relation to the object; Then he backs up and grabs the object. This is materialized in the state chart with states. We chose this solution because its final implementation is rather simple and involves both the statechart and the code so that we do not add a complex method.

When the robot grabs the object, its position is saved

b. Problems encountered/ modifications to the environment given and further comments:

The problems we encountered were, in particular, the correct arrangement of the robot in relation to the object in front of the reverse; this was a more technical aspect, particularly for the calculation of angle ...

The extension could not be completed due to lack of time and also due to the fact that a lot of time was invested in the problem raised above. In the following, it would be a matter of defining a point in the robot environment where it would deposit the objects. The robot would then return to the point where it was when it grabbed the object and continue its cleaning from there.

c. State chart:

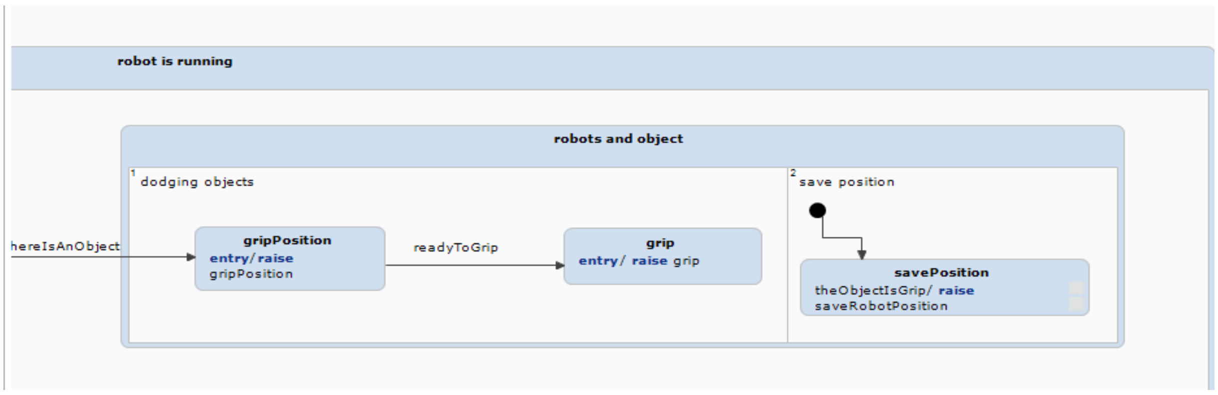


Figure 5: state chart grip objects

5. Covering the room as much as possible

The robot's main purpose is to clean as much of the space available to it as possible. To accomplish this goal, a cleaning algorithm must be implemented. An algorithm that cleans the space based on logic is always more precise than a random one. As it avoids wasting time by recleaning an already clean area.

a. Problems encountered/ modifications to the environment given and further comments:

Numerous mistakes and issues rose in this subsection. Considering having an optimal cleaning algorithm as necessary to the use of the robot, this part was the one we spend the most time on, only to end up removing everything within the last weeks of development and keeping a much simpler version. We've encountered issues with the turn function which was the key to our algorithm.

The algorithm we had worked on was born from a rather simple idea; counting the number of times we hit a corner (considering a corner to be hitting two walls) and then brushing through each wall. Many problems were left unsolved or not well thought out here, as the room may be of peculiar shape which would mean that detecting if the cleaning process should be off lies on using the camera to detect the look of the floor and having an internal map to recognize where the robot has already been and thus, cleaned. Both of those ideas could very much be interesting to implement on a larger project with more time and more background knowledge on state charts than what we first through ourselves in with.

An optimal, simpler idea yet still sophisticated would be making a square with the robot that eventually gets bigger and bigger before adding other parts of the state chart.

IV-Verification and Validation

The questions we asked the system fall into the two categories: safety and liveness.

Safety	Liveness
when leaving gapDown, is there a way to enter turning without entering moveBack first?	after entering moveFront do we always pass in it again?
when entering blocage, Do I ever pass through turning after?	when leaving gapDown, is there a way to enter turning without entering moveBack first ?
	after entering moveBack Do we always enter turn?

In appendix are some validation tests for our project

V-Further remarks

At the end of this project, the fact that it remains unfinished leads us to question the things we should have done differently. First and foremost, our complete handling of the project was slow, resulting in a delay in the weeks to follow. Then, what should be improved would be our working method; Having worked on a good part of the project each on our side, the late observation of the lack of coherence and stagnation of our work was a huge problem for us. And trying to try and merge bits of what everyone worked on just leads to a code and state chart mass murder. It would probably have been better to mount a first minimal version of a common agreement and then implement from this version the different extensions and features. Our approach to the project, including the use of current concepts, was not the best. We would have needed at the very beginning for each extension and feature identified the concept put forward. This way, at implementation; we would have already known what to do.

The safest way to work on this project would have been to lay down a solid foundation for the control of our state chart at first and build upon it in synchronicity with each other rather than in parallel. There are also numerous things we know and agree that we definitely could have done better. Such as a cleaner and nicer code with less dependencies than the current format we have. When it comes down to the final version of the state chart, we still believe it's better than other implementations we've seen. It isn't God tier for sure, and some parts such as the virtual wall for instance could have been handled on a larger scale with a different approach that would have made the state chart nicer overall.

Nevertheless, we chose to have this version in order to differentiate between what is control and what isn't. Instead of having big function that raise more that one event, we judge from what we've come to understand about state chart that this shines a light at a wrong power dynamic between code and state chart.

Overall, this was a very interesting project, that some of us want to pursue and redo on their own terms, due to how interesting, enlightening and fun it was. And even through the hurdles we came across, we're proud to have been able to have the version we have submitted today.

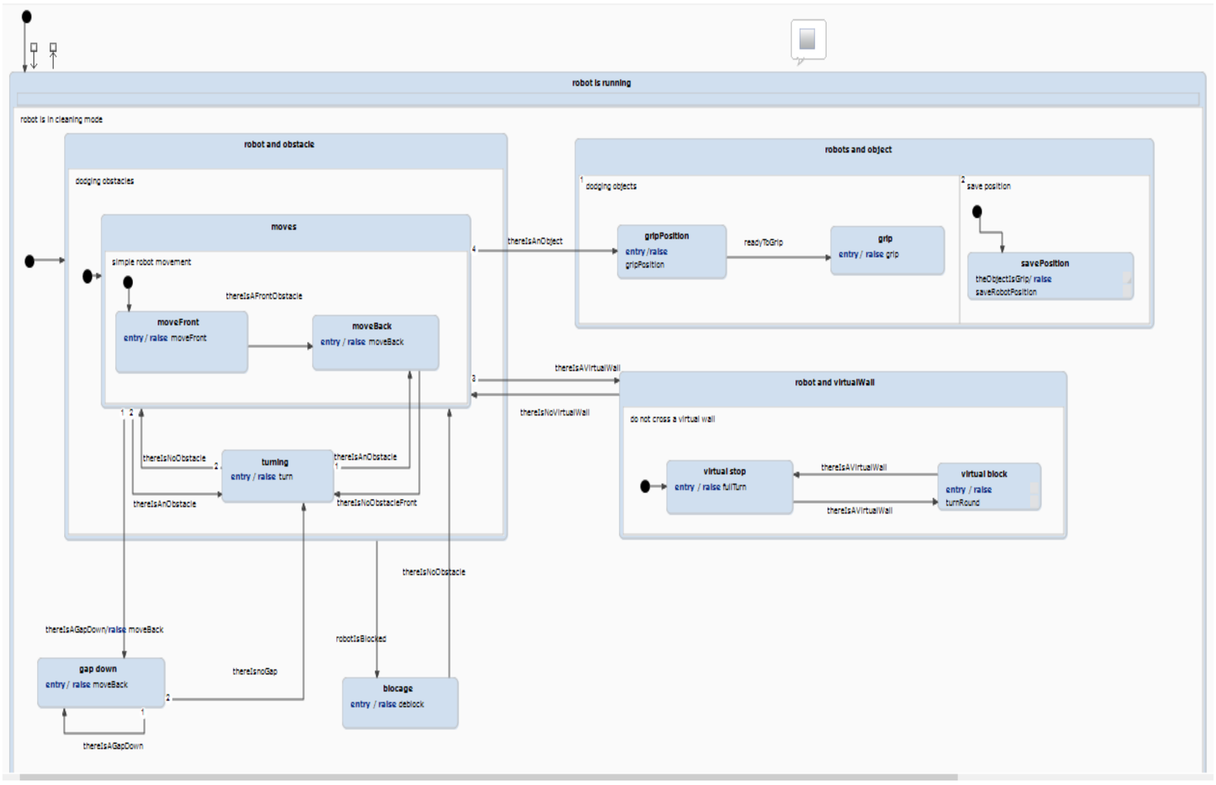
Appendix

a. Validation tests

```
library
macros.mcl
end_library

(*testMVP.ysc*)

(* after entering moveFront we always pass in it again *)
(*[true* . isPresent("moveFront","entering")] INEV("moveFront","entering")*)

(* after entering moveBack we always enter turn *)
(*[true* . isPresent("moveBack","entering")] INEV("turning","entering") true*)

(* after entering B it is fairly always possible to enter A *)
(*<true* . isPresent("stateB","entering") . true* .
isPresent("stateC","entering")> @*)

(* when leaving gapDown, there is no way to enter turning without entering
moveBack first *)
(*[true* . isPresent("gapDown","leaving") . (not
isPresent("moveBack","entering"))* . isPresent("turning","entering")] false*)

(* when entering blocage, I never pass through turning after *)
(*<true* . isPresent("blocage","entering") . true* . not
isPresent("turning","entering")> @*)
```

(*[true* . isPresent("moveBack","entering")] mu X . (< true > true and [isPresent("turning","entering") and not isPresent("moveFront","occurring")] X)*)

b. complete statechart



robot is running
robot is in cleaning mode
robot and obstacle
dodging obstacles
moves
simple robot movement
thereIsAFrontObstacle
moveFront
entry / raise moveFront
moveBack
entry / raise moveBack
thereIsNoObstacle
turning
entry / raise turn
thereIsAnObstacle
thereIsAnObstacle
thereIsNoObstacleFront
thereIsAnObject
robots and object
dodging objects
gripPosition
entry /raise gripPosition
readyToGrip
grip
entry / raise grip
save position
savePosition
theObjectIsGrip/ raise saveRobotPosition
thereIsAVirtualWall
thereIsNoVirtualWall
robot and virtualWall
do not cross a virtual wall
virtual stop
entry / raise fullTurn
thereIsAVirtualWall
thereIsAVirtualWall
virtual block
entry / raise turnRound
thereIsNoObstacle
thereIsAGapDown/raise moveBack
robotIsBlocked
thereIsnoGap
gap down
entry / raise moveBack
blocage
entry / raise deblock
thereIsAGapDown