eYSIP2017

Model Based Design using Statecharts and Code generation for robotic platform

Intern: Manav Guglani
Mentor: Naveen C
Duration of Internship: 22/05/2017 – 07/07/2017

2017, e-Yantra Publication

Model Based Design using Statecharts and Code Generation for Robotic Platform

Abstract

Statechart based modeling is used when there is strict time constraint for developing the project. This kind of model also makes programmers job easy as system generated code is not prone to syntactic errors. Development time, human resources required also reduces a lot using this modeling technique. We can have independent components for each project, this helps in parallel progress of work. This can also be used to educate students in schools as this model provides high level abstraction which hides all the technical internal details. Here in this project, we have tried to implement a eYRC 2015 theme *puzzle solver* using statechart based modeling.

Completion status

1. The syntax and semantics of statecharts were learnt as described by David Harel.
2. Existing statechart models of some systems were understood.
3. Yakindu SCT software was learnt.
4. Puzzle solver theme was modeled using statecharts in Yakindu statecharts tool.
5. The theme was successfully implemented using the automatically generated code by Yakindu and integrating it with firebird V libraries.

1.1 Hardware parts

- Firebird V is used for the implementation of the theme.
- Detail of hardware: Hardware manual for the firebird V, Vendor link (nex robotics),

1.2 Software used

1. Yakindu Statechart Tool was used to model the systems using statecharts. It was also used for simulation, debugging and generate the c code corresponding to the statechart model.
   version- 1.0.2
   download page link
   For installation steps go the the link Installation steps for Yakindu Statechart Tool

2. Atmel studio 7 was used for compiling and editing the c code. The stk500v2 programmer was integrated with the atmel studio 7 to facilitate the burning of compiled code directly.
   version- 7.0.1188
   download page link
   For installation steps go the the link Installation steps for atmel studio

1.3 Software and Code

Github link for the repository of code is here
Brief explanation of various parts of code
The statechart consists of the states *puzzle solver* and *stop*. Until the puzzle is solved, it remains in *puzzle solver* state. Here is brief explanation of different modules of *puzzle solver* state.

- **LineFollower state machine**

  It contains two states, *LineFollowerOff* and *LineFollowerOn* as shown in the figure 1.1. Raising event *StartLineFollower* when it is in *LineFollowerOff* state makes transition to *LineFollowerOn* state. The bot then starts following the line. Similarly, raising an event *StopLineFollower* when it is in *LineFollowerOn* state makes a transition to *LineFollowerOff* and robot stops following the line. The *LineFollowerOn* state is further divided into some substates as shown in figure 1.2. This state

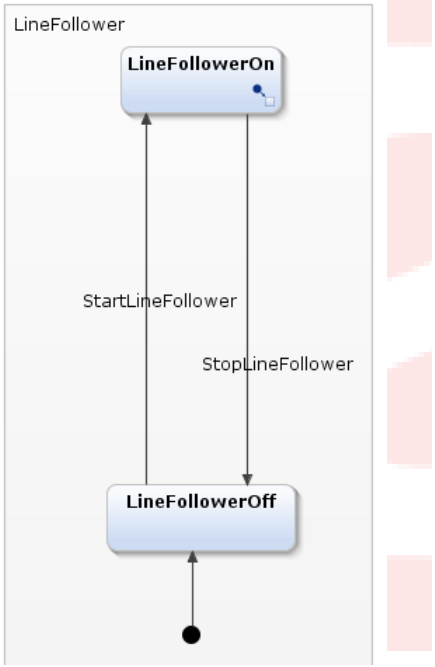

Figure 1.1: line follower machine

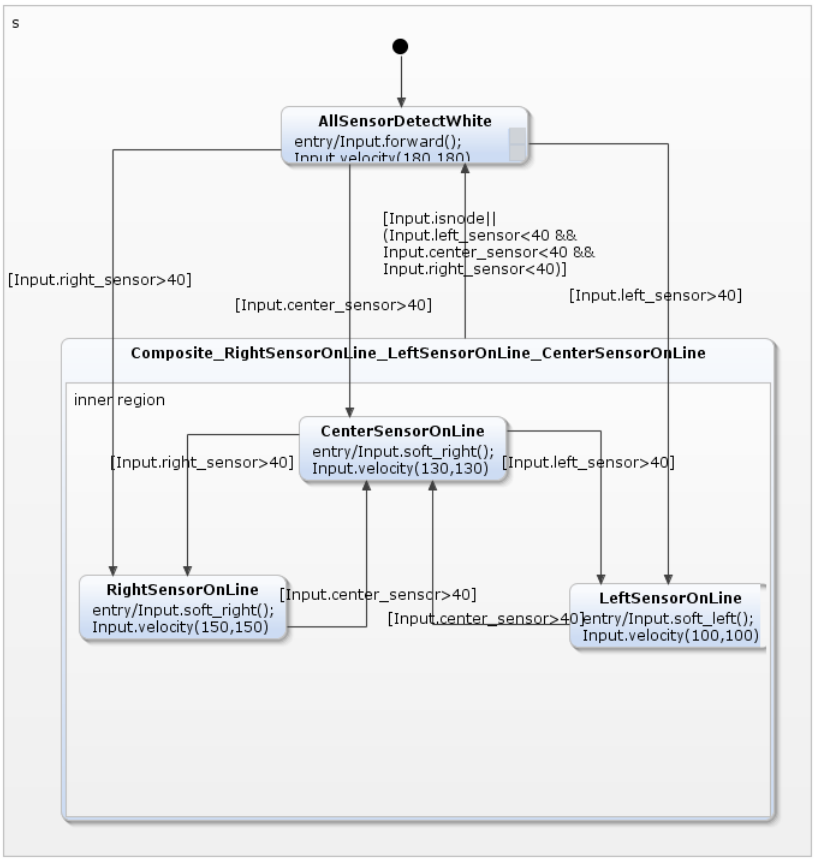

Figure 1.2: LineFollowerOn state

machine tries to follow the black line with the line between the left and center sensor. It contains states *AllSensorsDetectWhite*, *LeftSensorDetectLine*, *CenterSensorDetectLine* and *RightSensorDetectLine*. The actions are taken accordingly as the sensor values change after some threshold values.

- **State machine for making turn**
  It contains two states, *TurningOn* and *TurningOff* as shown in figure 1.3. When robot needs to take a turn in the grid, we need to give value to a variable turn as *left, right, back* and *forward* according to the needed turn. After that, we need to raise an event *StartTurning*. Then it will enter the *TurningOn* state. The robot will start turning. As the turn is completed, it will automatically go to the *TurningOff* state. *TurningOn* state is further divided into further states as *GoingForward, TurningLeft, TurningRight* and *TurningBack* as shown in figure 1.4. The state which is entered is selected on the basis of corresponding value of the variable turn. If system is in *GoingForward* state, it will remain here and move forward. When the node is not detected i.e., the node has passed, it will exit this state and go to *TurningOff* state.

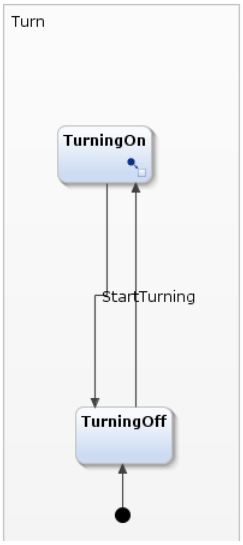

Figure 1.3: Turn machine

If system is in *TurningRight* state, it will move bot forward 55 mm, break, move 55 degrees right and then keeps on turning right. When center sensor detects black (line), it will exit this state and go to

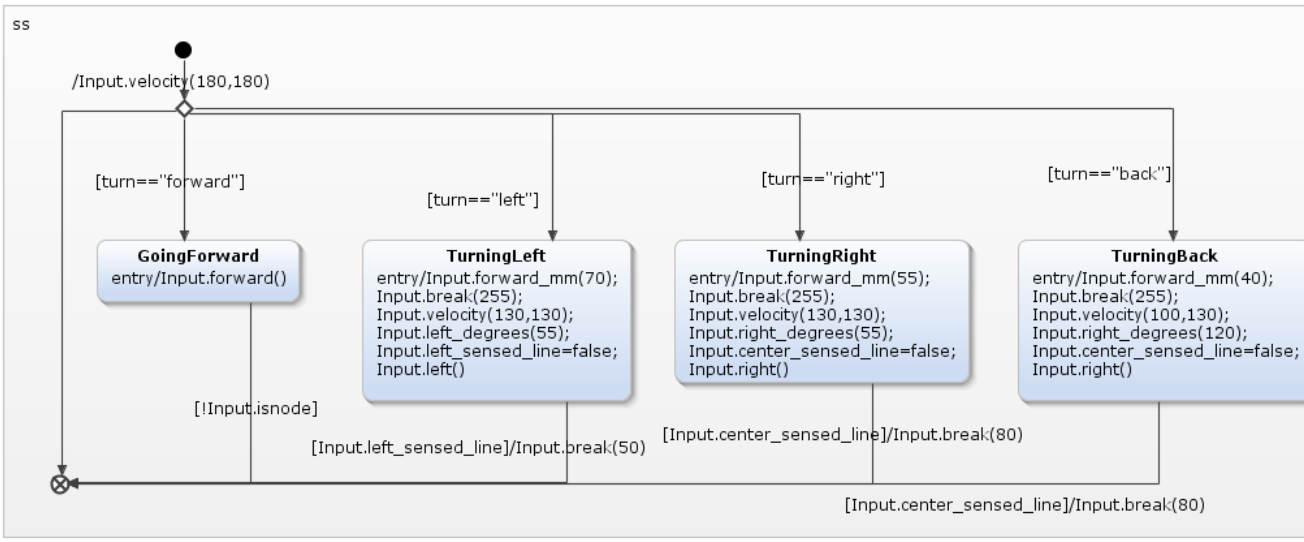

Figure 1.4: TurningOn state

TurningOff state. If system is in *TurningLeft* state, it will move forward 70 mm, break, move 55 degrees left and then keeps on turning left. When left sensor detects black (line), it will exit this state and go to *TurningOff* state. If system is in *TurningBack* state, it will move forward 40 mm, break, move 55 degrees right and then keeps on turning right with different velocities (to make sure of proper alignment with line as line should be between left and center sensor). When center sensor detects black (line), it will exit this state and go to *TurningOff* state.

- **Orientation state machine**
  It keeps track of the orientation of the bot (figure 1.5). It contains four states *East, West, North* and *South*. It uses the turning state to orient the robot in a particular direction. For example, if *OrientInEast* event is raised, in whichever of the four state the system is, it will enter in *East* state by using the turning state machine with proper turn. Another way of using this state machine is raising an event *left, right, front* or *back*. It will turn accordingly and update the orientation by going to the proper state.

- **Coordinates state machine**
  It contains only one state *Coordinates* as shown in figure 1.6. When an event *update coordinates* is raised, it will update the x and y coordinates by incrementing or decrementing the appropriate value according to

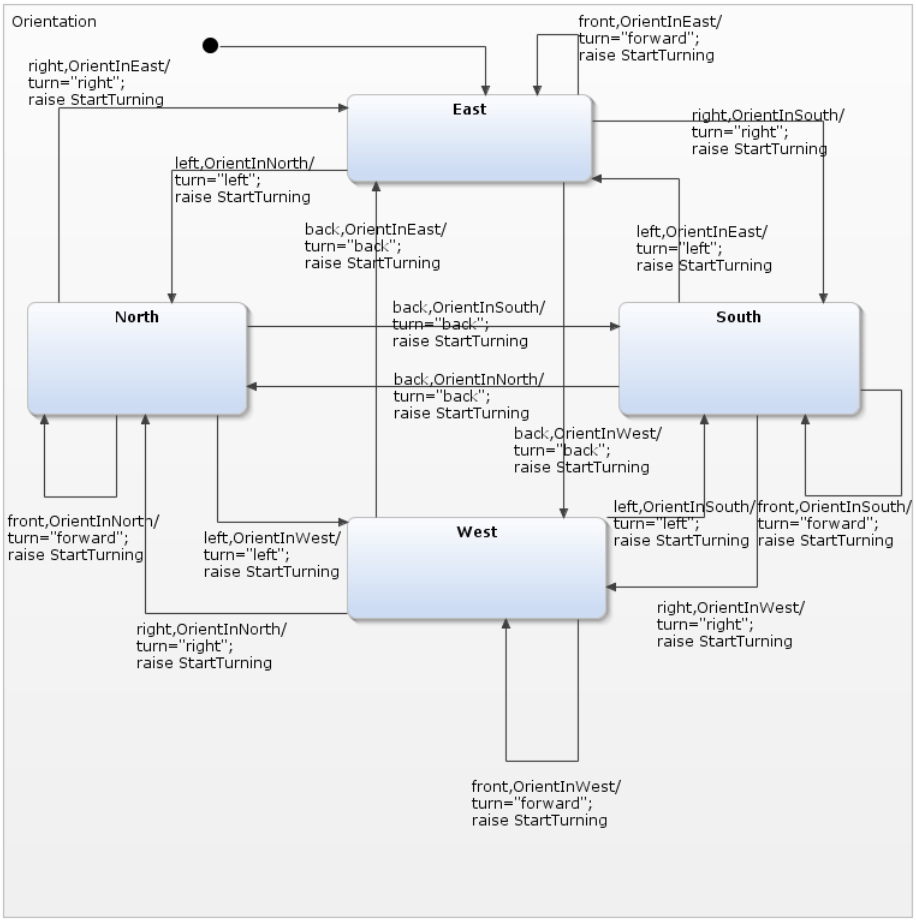

Figure 1.5: Orientation machine

which of the state in Orientation state machine is active.

- **GotoTargetNode**
  It takes the robot to the target node *tx* , *ty* (figure 1.7). To use it, just give the values to *tx* and *ty*. After that, raise an event *gotoxy*. It will take the robot to the target coordinates in the grid. It contains substates *Idle, NodeAction, GoingToNextNode* as shown in figure 1.8.

  1. Idle state
     When *GotoTargetNode* is not being used, it remains in the *Idle* state. When event *gotoxy* is raised, it moves to *NodeAction* state
  2. NodeAction State
     It takes the appropriate action at particular node for going to next node. If target node is reached, system goes to *idle* state.

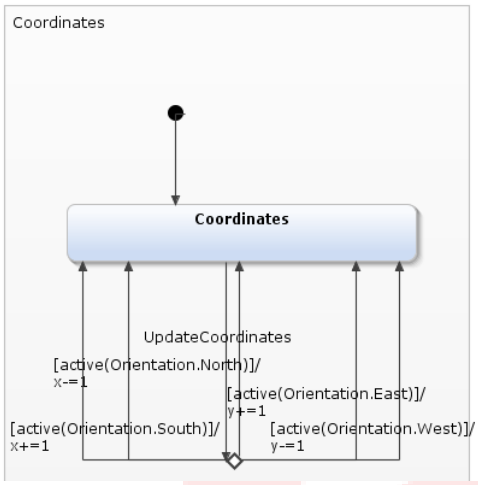

Figure 1.6: Coordinates machine

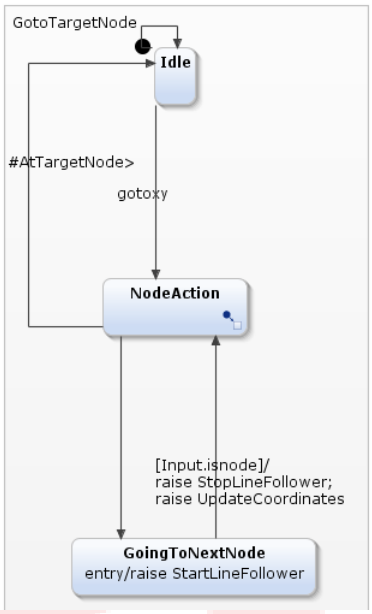

Figure 1.7: GoToTargetNode machine

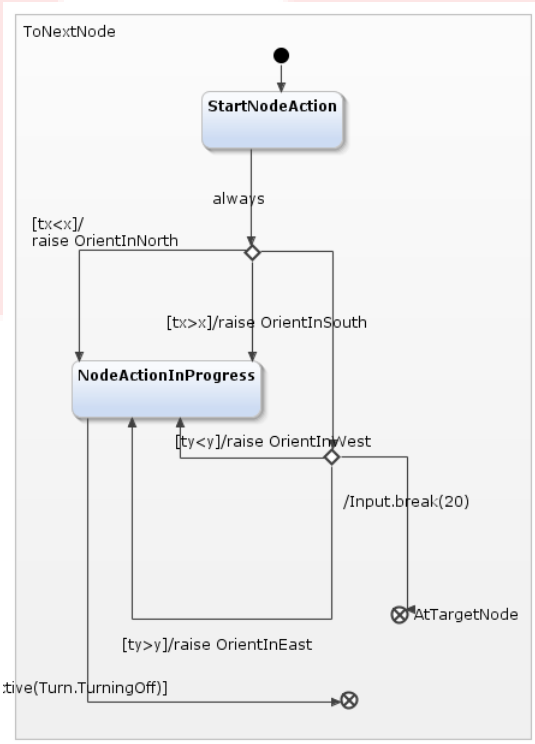

Figure 1.8: node action state

Otherwise, first it checks whether the target x is equal to present x or not. If not, it moves the robot in x direction. Otherwise, it checks whether the target y is equal to the present y or not. If not, it moves the bot in y direction. When node action is being taken (turning in proper direction or moving out of the node), it remains in *NodeActionInProgress* state. When the action is completed, it exits the *NodeAction* state and goes to *GoingToNextNode* state. When target node is reached, it will exit at *AtTargetNode* exit point.

3. *GoingToNextNode* state
   When system enters this state, it turns on the line follower. Robot starts following the line until the next node is reached. When next node is reached, it stops the line follower, updates the coordinates and goes to *NodeAction* State.

- **Main**
  It is *main* state machine which uses all other state machines to accomplish the task (figure 1.9). Initially it enters start state and it starts the line follower. As a node is detected, line follower is stopped. The target coordinates are fetched from c function and event *gotoxy* is raised. Now it is in *go to pick node* state. When the target node is reached, it enters *picking* state (figure 1.10). In *picking* state, robot starts following the line and when some distance is reached, it picks the number. After a node is detected, it updates the coordinates, stops the line follower and exits the *picking* state. It enters the *going in middle of d1* state where it gives target coordinate as *ty*=2 and raises *gotoxy* to reach to the middle of the grid d1. When it is reached, it gets next target coordinates and enters *going to drop node* state. Here, it raises *gotoxy*. The robot is now going towards drop node. When it reaches there, it goes to *dropping* state (figure 1.11). This state is similar to *picking* state except that it will drop the number at a certain distance. After that, it gets next coordinates. If complete puzzle is solved (num==-1), it will exit the state machine. Otherwise, it will go to the *going in middle of the d2*. After reaching in the middle of d2, it will go in *go to pick node* state and the cycle repeats.

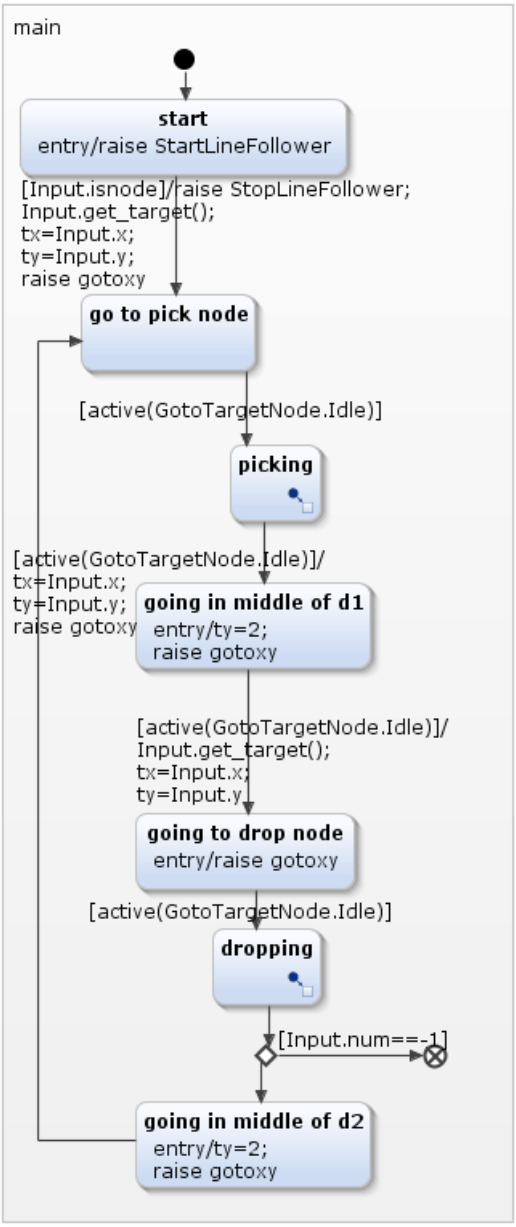

Figure 1.9: main machine

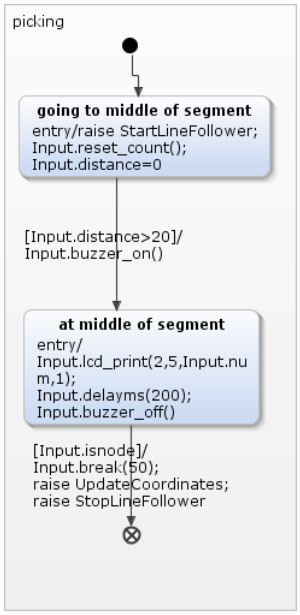

Figure 1.10: picking state

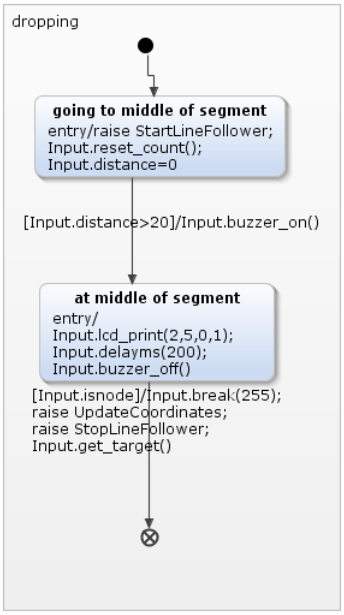

Figure 1.11: dropping state

Stop state

After complete puzzle is solved, system comes out of puzzle solver state and enters in this state where it stops and beeps a buzzer for 2 sec.

1.4 Use and Demo

Instructions for demonstration

1. Open Atmel Studio 7.
2. In main.c file, enter the sequence of numbers in d1 grid in "d1" array. Bottom left number should be entered first, then second number in the row and so on. after completing first row, move to the upper rows one by one.
3. Enter the numbers present in grid d2 in 2d array "d2" where you need to add cell number and the number present in the cell for each number. Cell number starts from 0 at bottom left and increases as we move right and then on next row. Maximum of 4 numbers can be added in it. If you want to add less then 4 numbers, then fill {-1,-1} for the others.
4. Compile the code and burn in the firebird V.
5. Put the bot just ahead of start bar and start the bot. It will take some time to find the optimum solution according to how many numbers are entered. After that, it will start solving the puzzle and beep a long buzzer when the puzzle is solved.

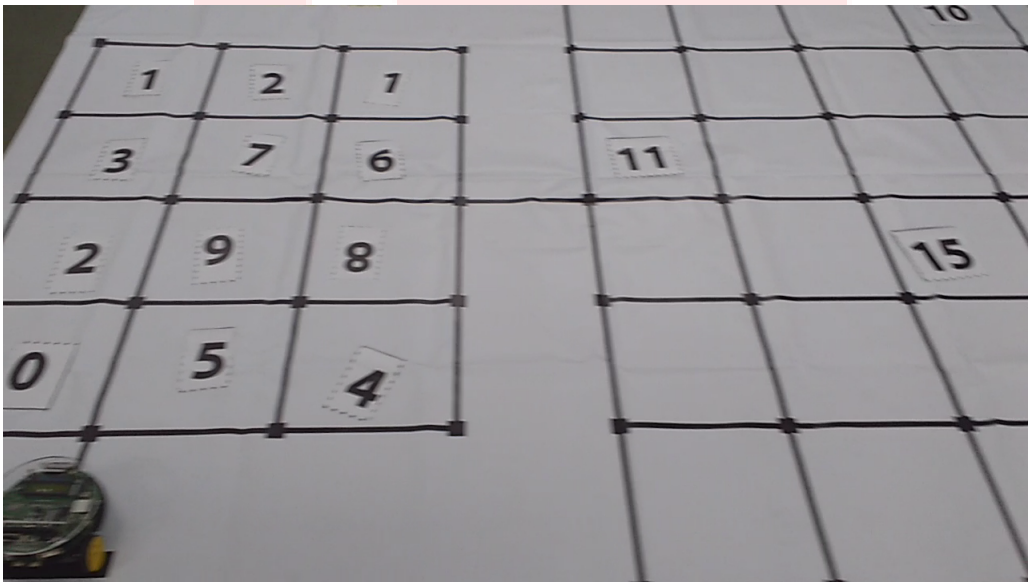

Figure 1.12: final setup image

Youtube Link of demonstration video

1.5 Future Work

Inline simulation can be used for easy debugging in statecharts. By this feature, a developer may be able to see live which states are active as the code is running in real time. This feature may be of great help to developers to debug where things are going wrong.

1.6 Bug report and Challenges

There is a bug in Yakindu SCT that if we copy any orthogonal region from one place and paste it in parallel with other regions then there is a problem that the sequence in which the control reaches the different states become ambiguous. This bug wasted too much time. Finally, step by step simulation was used to debug this unusual behavior. So, it is advisable that if you want to copy and paste any region, then instead of pasting directly, make a new orthogonal region and paste there all the sub states in it. This does not gives the problem of sequence.

One of the most important challenge faced was changing the perspective. Initially, I was thinking in terms of sequential programming rather than thinking in terms of concurrent executing state machines machines. This changing of perspective helped a lot in modeling the system as independent modules rather than a sequential flow of control from one state to another.

Bibliography

[1] David Harel, "Statecharts: A visual formalism for complex systems", *Department of applied mathematics, The Weizmann Institute of Science, Rehovot, Israel.*

[2] Kavi Arya, Blossom Coelho, Shradda Pandya, "A Model Based Approach to System Building Using the E-Yantra Educatonal Robot", *Department of Computer Science, IIT Bombay.*