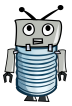


We wrote our own library which contains custom blocks that allow us to build simulations. Because of the complexity of the whole system we decided to use a unit-test-like approach to test specific components.

You will find an overview over all customs blocks in the library file itself.

1 Environment

Because there is no pre-built block in Simulink which would suffice for the environment, we needed to come up with our own approach. The maze and all entities in it are described in a text file which is loaded into Matlab and transformed into a matrix. This matrix represents the entire environment and is used in raycast computations to determine sensor readings of the proximity sensors and IR sensors at a given point in time and space. This happens within our custom sensor blocks.

E-Puck's motor is realized by a custom differential drive block implementing the real world physics by using adequate differential equations. The block assures that the Tin Bot cannot drive through other objects by looking up the occupation status in the matrix and restricting the motion appropriately.

The complete physical Tin Bot is represented by the block "Tin Bot Physical". This block feeds the differential drive's data (position and orientation) in the appropriate sensor blocks — taking the orientation offset of the sensors into account — such that their values can be computed appropriately using raycasting.

According to the Tin Bot's current position given by the differential drive the matrix is updated such that Tin Bots cannot drive through each other. Therefore, a Tin Bot Introducer block has to be connected to the block representing the physical Tin Bot. This block writes an appropriate value in the matrix at the Tin Bot's current position. To use multiple Tin Bots within the simulation, the Introducer blocks are chained such that at the end there is a matrix containing all the Tin Bots.

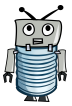
The map used by the Tin Bot's sensors, however, must not contain its own Tin Bot. Therefore, the Tin Bot Eliminator block deletes the respective entry from the environment. This prevents the sensors from detecting themselves.

The LPS block takes the environment's data and feeds it into the Tin Bots every two seconds. This data is then used within the software blocks. The LPS block allows for two states, enabled and disabled as described in our specification document.

The output of the control software is fed back to the environment, which allows us to model real world physics.

2 Components

Besides the model of the environment, there are various other components mainly modeling the software running on the Tin Bot as Stateflow charts.



As default model of operation, there's the Right Hand Follower, which does not keep track of previous locations, and just applies the right-hand rule until another component has a better idea what to do.

Next, there's the detection of good points in time to measure the exact angle to the victim. We assume that the intersection of all measurements should give a reasonably good result. This functionality is spread across Traffic Cop Eyes, Victim Finder, Victim Detector; each of which are simple state machines.

As soon as the Tin Bot knows the location of the victim, it's only maze solving with a (partailly) known maze. Path Finder takes care to compute the waypoints and (if necessary) trigger recomputation or fall-back to right-hand-rule. Path Executor (which drives towards the next waypoint) is a real state machine, but both are implemented as a Stateflow chart.

Finally, there's the (Blind) Traffic Cop, which orchestrates the previous modules, and decides "who is allowed to drive", by means of a state machine.

All these modules (together called "Controller") need the concept of "current location and direction", which is given by the Approximator. This module takes the LPS data (if available) and motor output, and tries to interpolate the current position and direction from it. This module applies the same differential equations as the Differential Drive.

3 Test Cases

As already mentioned we mostly used a unit-test-like approach to test all aspects of the system, only the last two tests cover the combination of the system components.

drive_test tests the physical model of differential drive isolated from any logic.

proximity_test tests the physical model of the proximity sensors separately.

ir_test tests the physical model of the isolated IR sensor.

traffic_blind_test tests the "blind" part of the traffic cop, i.e. without any IR-sensor input.

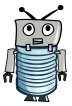
victim_direction_test tests victim direction calculation, i.e. the logic used to approximate the angle from the Tin Bot to the victim based on IR-sensor input.

path_finder_test tests internal map generation and escape path calculation, i.e. in essence also a test for **UC-D**, since one can see that the logic is able to deal with formerly unknown obstacles and behave reasonably.

approximator_test tests approximation of current position and orientation based on the motors data as opposed to real data provided by the LPS. Also checks update mechanism in case the LPS is turned on.

follow_right_hand_test tests the correct application of the right hand rule taking proximity sensor data into account. Covers large parts of the project since most components are involved.

controller_test tests the complete control software, and use cases **UC-E** and **UC-F**



Use cases **UC-A** and **UC-B** are modeled implicitly, because startup corresponds to starting the simulation, and shutdown corresponds to stopping the simulation. Use case **UC-C** corresponds to a nice to have feature and is not model. We did, however, not model more than one Tin Bot at a time, so the failure of one has no impact anyways.