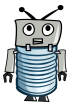




We wrote our own library which contains custom blocks that allow us to build simulations. Because of the complexity of the whole system we decided to use a unit-test-like approach to test specific components.

You will find an overview over all customs blocks in the library file itself which is also attached at the end of this document for convenience.

# 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.

The 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.

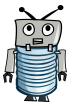
According to the Tin Bot's current position given by the differential drive, the matrix is updated such that Tin Bots cannot drive through each other. Therefore, a *Tin Bot Introducer* block has to be connected to the block representing the physical Tin Bot. This block writes an appropriate value in 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 *Victim Introducer* block uses a similar mechanism as the Tin Bot Introducer to place the victim inside the matrix representing the map. The *Victim* block participates in the algebraic loop which models the process of being picked up by a Tin Bot.

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





# 2. Components

In the following description of our components we define the term "*introduce*" as recording the position of an object in the matrix representing the environment by setting a certain value in the particular cell(s) the object is located in.

## A. Physical Environment

### A.1. Tin Bot Introducer

Records the position of the Tin Bot inside the matrix representing the environment. Because the matrix is used as input for the drives, the Block delays the position by one sample to prevent algebraic loops using *x0* respectively *y0* as starting position.

**Inputs:**

**row_in** Current row the Tin Bot should be placed in.

**col_in** Current column the Tin Bot should be placed in.

**reset** (Not used in our simulation, resets the internal state.)

**x0** Starting position of the Tin Bot.

**y0** Starting position of the Tin Bot.

**map_in** Matrix representing the environment without the corresponding Tin Bot.

**Outputs:**

**map_out** Matrix representing the environment with the corresponding Tin Bot.

**row_out** Row where the Tin Bot has been introduced (delayed).

**col_out** Column where the Tin Bot has been introduced (delayed).

### A.2. Tin Bot Eliminator

**Inputs:**

**map_in** Matrix representing the environment (all objects have to be introduced).

**row** Row where the corresponding Tin Bot has been introduced.

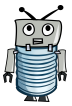
**col** Column where the corresponding Tin Bot has been introduced.

**Outputs:**

**map_out** Matrix representing the environment without the corresponding Tin Bot.

### A.3. Victim Introducer

**Inputs:**





**map_in** Matrix representing the environment without the victim.

**victim** Current position of the Victim as two dimensional vector ($[x\ y]$).

**Outputs:**

**map_out** Matrix representing the environment with the victim.

### A.4. IR Sensor

Implements an IR sensor using raycasting.

**Inputs:**

**map** Matrix representing the environment (all objects have been introduced).

**row** Row where the IR-sensor is located.

**col** Column where the IR-sensor is located.

**direction** Direction the IR-sensor is heading toward.

**victim** Current position of the Victim as 2 dimensional vector ($[x\ y]$).

**Outputs:**

**triggered** Whether the IR sensor has been triggered or not.

### A.5. Proximity Sensor

Implements a proximity sensor using raycasting.

**Inputs:**

**map** Matrix representing the environment (all objects have to be introduced).

**row** Row where the proximity sensor is located.

**col** Column where the proximity sensor is located.

**direction** Direction the proximity sensor is heading towards.

**Outputs:**

**distance** Measured distance to object.

**object** Object type detected (only for debugging purposes).

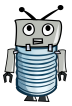
### A.6. Differential Drive

Models a differential drive using appropriate differential equations.

**Inputs:**

**motor_left** Velocity of left wheel.

**motor_right** Velocity of right wheel.



**phi** Direction the E-Puck is heading towards.

**Outputs:**

**dx/dt** Derivative of the $x$ coordinate.

**dy/dt** Derivative of the $y$ coordinate.

**dphi/dt** Derivative of the direction.

### A.7. Differential Drive Restricted

Restricts the movement of the E-Puck to non-occupied areas on the map.

**Inputs:**

**reset** Internal reset of the integrator blocks to the starting positions (not used).

**x0** $x$ coordinate of the starting position.

**y0** $y$ coordinate of the starting position.

**phi0** Direction the E-Puck is heading towards at the beginning.

**motor_left** Velocity of left wheel.

**motor_right** Velocity of right wheel.

**map** Matrix representing the environment (all objects have to be introduced).

**Outputs:**

**x** Current $x$ coordinate.

**y** Current $y$ coordinate.

**phi** Current direction.

**row** Current row.

**col** Current column.

## B. Software

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

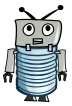
### B.1. Tin Bot Approximator

This component takes the driving information (velocity of the wheels) as well as occasional updates via LPS, and approximates their current position and orientation.

**Inputs:**

**motor_left** Speed of left motor.

**motor_right** Speed of right motor.



**lps** Incoming LPS data (no data: $[-1\ -1\ -1]$, data point: $[x\ y\ \varphi]$).

**Outputs:**

**current_x_y_phi** Approximated current position and orientation.

### B.2. Traffic Cop Eyes

Detects good points in time to measure the exact angle to the victim, and interrupts "Traffic Cop" (see below) to start the measurement.

**Inputs:**

**current_x_y_phi** Approximate position and orientation of the Tin Bot.

**ir_raw** Current IR sensor readings.

**found_victim_phi** Flag to indicate that new angular information is available.

**found_victim_xy** Flag to indicate that new Cartesian information is available.

**Outputs:**

**need_angle** Whether a new angular measurement should be taken.

### B.3. Victim Direction

This component is responsible for measuring and computing the precise angle to the victim.

**Inputs:**

**ir** Current IR sensor readings.

**current_x_y_phi** Approximate position and orientation of the Tin Bot.

**run_finder** Flag whether this subsystem shall be active.

**Outputs:**

**mot_l, mot_r** Motor instructions.

**found_victim** Flag to indicate that *v_phi* can be read.

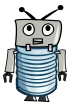
**v_phi** If flag is set, this is the absolute angle to the victim, as per the most recent measurement. Otherwise: garbage.

### B.4. Victim Detector

This component is responsible for taking angular measurements, and triangulating them to the victim's concrete coordinates.

**Inputs:**

**found_victim_phi** Flag to indicate that *victim_angle* can be read.





**victim_angle** If flag is set, this is the absolute angle to the victim, as per the most recent measurement. Otherwise: garbage.

**current_x_y_phi** Approximate position and orientation of the Tin Bot.

**Outputs:**

**found_victim_xy** Flag to indicate that *victim_x, victim_y* can be read.

**victim_x, victim_y** If flag is set, this are the coordinates of the victim. Otherwise, garbage.

### B.5. Right Hand Follower

This is the default model of operation, which does not keep track of previous locations, but just applies the right hand rule until another component has a better idea what to do.

**Inputs:**

**run_rhr** Flag whether this subsystem shall be active.

**proximity** Current proximity sensor readings (eight dimensional vector).

**Outputs:**

**motor_left, motor_right** Motor instructions.

### B.6. Sensor Filter

This component "filters" the sensor inputs for Path Finder (see below), so that the victim is not recognized as an obstacle. (Since it is "docked" to the back of the Tin Bot, this cannot cause the Tin Bot to drive into a wall.)

**Inputs:**

**ir** Current IR sensor readings (eight dimensional vector).

**proximity_in** Current proximity sensor readings (eight dimensional vector).

**Outputs:**

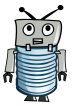
**proximity_out** Filtered proximity sensor readings (eight dimensional vector).

### B.7. Path Finder

As soon as the Tin Bot knows the location of the victim, the reminder of the problem is basically a maze solving task under a partially known maze. Path Finder takes care of computing the waypoints leading to the victim at first, and out of the maze when the victim has been picked up.

Note that Path Finder constantly collects information about the environment, even if *compute* is cleared.

**Inputs:**





**proximity** Filtered proximity sensor readings (eight dimensional vector).

**current_x_y_phi** Approximate position and orientation of the Tin Bot.

**dst_x, dst_y** Destination location (either victim or nearest exit).

**compute** Flag whether this subsystem shall be active.

**Outputs:**

**drive** Flag whether the Path Executor subsystem shall be active.

**next_x, next_y** Next waypoint.

**no_path** Flag whether $A^*$ search has failed, and some fall-back system needs to run.

**path_completed** Flag whether the Path Finder and Path Executor subsystems are done.

### B.8. Path Executor

The Path Executor tries to reach each waypoint directly, i.e. drives straight to the given point.

**Inputs:**

**drive** Flag whether this subsystem shall be active.

**current_x_y_phi** Approximate position and orientation of the Tin Bot.

**dest_x, dest_y** Destination location (next waypoint).

**Outputs:**

**wheel_left, wheel_right** Motor instructions.

### B.9. Blind Traffic Cop

Decides "who is allowed to drive", and keeps track of the overall Tin Bot state.

**Inputs:**

**need_angle** Flag whether this subsystem shall be active.

**found_victim_phi** Flag to indicate that new angular information is available.

**found_victim_xy** Flag to indicate that new Cartesian information is available.

**no_path** Flag whether $A^*$ search has failed, and some fall-back system needs to run.

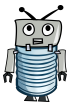
**path_completed** Flag whether the Path Finder and Path Executor subsystems is done.

**victim_x, victim_y** If corresponding flag is set, this are the coordinates of the victim. Otherwise: garbage.

**initial_x, initial_y** Start coordinates, functioning as the exit's coordinates as well.

**Outputs:**

**dst_x, dst_y** Destination location (either victim or nearest exit).



**run_path_finder** Flag whether the Path Finder/Executor subsystems shall be active.

**run_rhr** Flag whether the right hand rule subsystem (Right Hand Follower) shall be active.

**run_victim_finder** Flag whether the Victim Direction subsystem shall be active.

### B.10. Traffic Cop

Wrapper for Blind Traffic Cop and Traffic Cop Eyes.

## C. Communications

### C.1. LPS

**Inputs:**

**x** Current $x$ coordinate of the corresponding Tin Bot.

**y** Current $y$ coordinate of the corresponding Tin Bot.

**phi** Current direction the corresponding Tin Bot is heading towards.

**en** Whether the LPS is enabled or not.

**Outputs:**

**lps** Outgoing LPS data (no data: $[-1\ -1\ -1]$, data point: $[x\ y\ \varphi]$).

## D. Abstractions and Compositions

### D.1. Tin Bot Controller

Wrapper for the following components:
Victim Direction, Victim Detector, Follow Right Hand, Sensor Filter, Path Finder, Path Executor, Traffic Cop

### D.2. Tin Bot Software

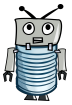
Wrapper for the following components:
Tin Bot Controller, Tin Bot Approximator

### D.3. Tin Bot Abstraction

Model of the physical part of the Tin Bot.

### D.4. Tin Bot

Complete model of the Tin Bot.





### D.5. Victim

Complete model of the Victim.

## 3. Test Cases

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

### A. Unit-Tests

**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 IR sensors of the Tin Bot.

**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 motor's 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.

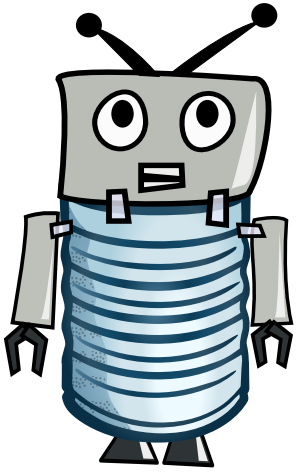
### B. System-Test

The test **controller_test** implements the concrete virtual prototype. It tests the complete control software within the simulated environment and use cases **UC-D**, **UC-E** and **UC-F** as well as requirements **MR2**, **MR4**, **MR5**, **MR6**, **MR8**, **MR10**, **NR1**, **NR4**, **MR14**, **MR15**, **MR16**, **MR17**, **NR7**.

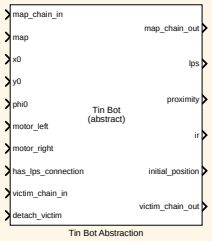
Use cases **UC-A** and **UC-B** are modeled implicitly, because the startup corresponds to starting the simulation, and the shutdown corresponds to stopping the simulation. Use case **UC-C** corresponds to a nice-to-have feature and is not modeled. We did, however, not yet model tests with more than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

# Tin Bot
## Group 6

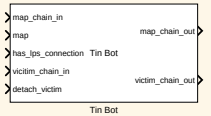
**Library**
**Version 2**
**(02.06.2016)**



## Abstractions and Compositions



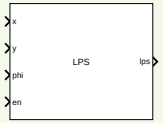
Tin Bot Abstraction



Tin Bot

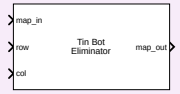
Victim

## Communications

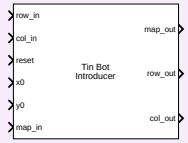


LPS

## Physical Environment

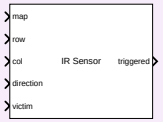


Tin Bot Eliminator

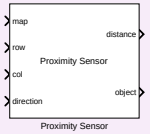


Tin Bot Introducer

Victim Introducer

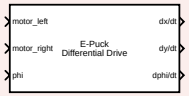


IR Sensor

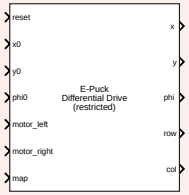


Proximity Sensor

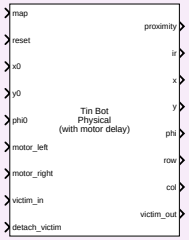
### Drive



E-Puck Differential Drive

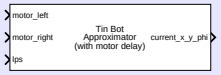


E-Puck Differential Drive Restricted



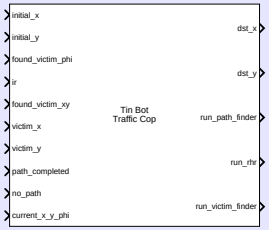
Tin Bot Physical

## Software

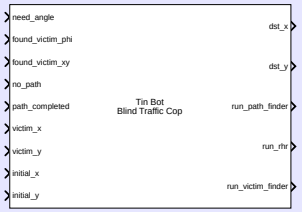


Tin Bot Approximator

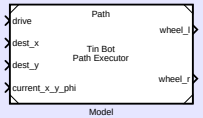
Tin Bot Software



Traffic Cop

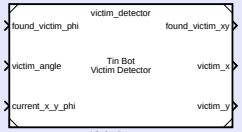


Blind Traffic Cop

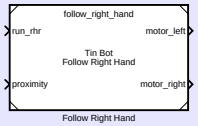


Model

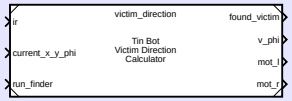
Sensor Filter



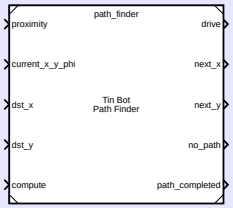
Victim Detector



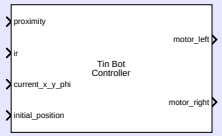
Follow Right Hand



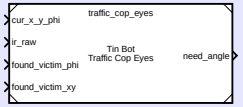
Victim Direction



Path Finder



Tin Bot Controller



Traffic Cop Eyes

## Tests

- ▶ Drive Test
- ▶ Controller Test
- ▶ Proximity Sensor Test
- ▶ Right Hand Rule Test
- ▶ IR Sensor Test
- ▶ Blind Traffic Cop Test
- ▶ Approximator Test
- ▶ Probalistic Path Finder Test
- ▶ Victim Direction Test