Mecha2000

Autonomous robot for LARC 2013's IEEE Open

Matías Estrada
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay

José Lombardi
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay

Nicolás Furquez
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay

Lázaro Pereira
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay

***Abstract*—Following the assignment of embedded robotics class our team developed a robot capable of resolving the challenge of the IEEE in IEEE Open category. This paper describes the technical characteristics of Mecha2000, which is the name of the robot and the team. This robot was designed using Usb4butia board and uses techniques of behavior-based robotics, which guarantee good response times in a controlled environment. Mecha2000 ranked first in the category Sumo.uy IEEE Open.**

Keywords—robotics, behavior-based robotics, Mecha2000, sumo.uy, usb4butia, IEEE Open, Larc 2013.

I. INTRODUCTION

This document contains the hardware and software design to solve the challenge IEEE Open presented at the LARC2013. The first part of the document records the paradigm used and behaviors found. Then will go provide a description of the physical design and the hardware used.

II. PROBLEM DESCRIPTION

The challenge of IEEE Open 2013 is about collecting cans (garbage) on a beach (scenario) without colliding with obstacles disposed therein and then deposit the items collected in a tank.

A. Paradigms

We used a behavior-based reactive paradigm [9], where each specific behavior has execution priority over the rest. This determines that all behaviors are sensing in parallel, but only the one with the highest priority will control the robot actuators and it is interrupted if and only if a higher priority behavior, needs to be run. That paradigm is reactive robot will act on the basis of sensory stimuli, without planning and without considering their movements for a given movement previous stimuli nor environmental status information than it did it move..

B. Environment

The environment is partially observable because the agent can not sense the overall state of it, you only have access to what is in front of the sensors located on the agent. One can also say that the environment is stochastic, because you can not determine the resulting state after the agent acts. As the task that the robot is part of a play, with a beginning and end, and you can re-start, the environment can be described as episodic.

Other features are the continuity of space and that the environment is semi-dynamic because the cans, sand and the chair does not change place without the intervention of the agent, but as time passes, you lose the possibility of lifting cans if nothing is done.

III. SOLUTION DESCRIPTION

A. Robotic Platform

- BeagleBoard: used as a computer center of the robot, we must take into account its limitations so that the development will be limited by its processing capacity.
- Motors AX-12 and AX-18 [3]: actuators that allows us to mobilize the robot on stage and perform the actions to collect cans and then throw the container.
- Bioloid kit [3]: parts kits that we needed for assembly robot.
- Acrylic Base, reusing old parts kits Butiá [2].
- Input Output USB4Butiá board [2]

B. Sensors

Sensors are an important part of building a situated robot, since these depend on the actions carried out. Sensors that are considered are:

- Camera
- Infrared Sensor

Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay

Each physical sensor has a class that represents and is responsible for processing the information provided by the sensor. All these classes have the same structure, inherited from the Sensor class, thus we can add sensors without causing too many changes at the code level.

The classes that are implemented are:

- SensorCamaraWhite: responsible for obtaining an image of the camera and process it.
- SensorDistancia: is responsible for taking the values returned by the infrared sensor.

The sensors can be executed in the form of threads independent of one another or sequentially. For simplicity we decided to perform the sensing in sequence, each sensor collects information stage independently and simultaneously, the sensors do not decide what behavior to execute, just collect information.

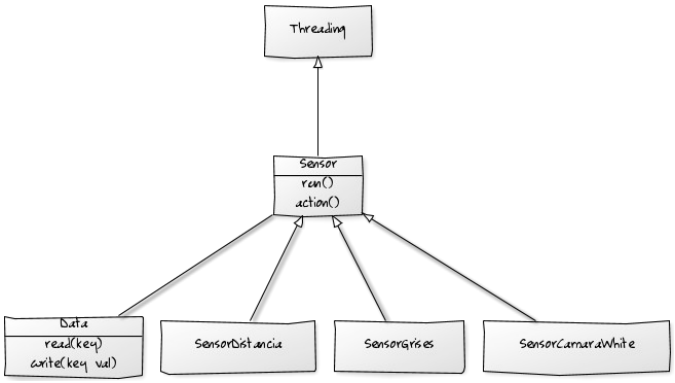

Fig. 1. Sensor Architecture

The Sensor class is the superclass of all implementations of the particular sensor drivers. Has an associated data structure which serves for data transfer between the individual sensors and behaviors. Each particular implementation can be written in the dictionary like structure.

The purpose of this is to decouple the sensors of behavior, and that the latter can read data structure assuming that this is always current.

The run () function is called by the start of the thread, therefore is only called if the sensors are used concurrently. What this function does is, in an infinite loop calling action () and before sending the thread to sleep. In Sensor, the function action () is abstract, and and it is the only one that subclasses must implement. In this way each sensor controller using a separate thread.

1) SensorDistancia:

It runs on a separate thread from the main thread, / * Data Just type in reading every time action () is called * /. He writes under the key 'SensorDistancia :: <puerto_u4b>'.

2) SensorCamaraWhite:

This driver is more complex, as it includes image processing and writes to the data structure processed information about the position of the can and the red recipient.

For the imaging process we use the library libopencv [4] version for python. This library was chosen because of its high power and popularity, which makes it possible to find large quantities of articles, and examples of their use.

The class constructor initializes all parameters such as the size of capture, so reduce the processing time and memory usage.

The function action () takes a picture from the camera, transforms into the HSV color space [5] and then continue with the different stages of processing.

The first thing you do is detect the color white (sand) and keep the outline bigger and closer to the robot (greater axis) and using ConvexHull function [7] define a new contour that will serve as a mask for the rest the process. In Figure 2 you can see the sand ConvexHull marked in blue.

The mask is an image of the same size of the original image, only black and white, where white represents a one in a binary mask. To generate a mask is printed a polygon or contour with all the stuffing into a black and white image.

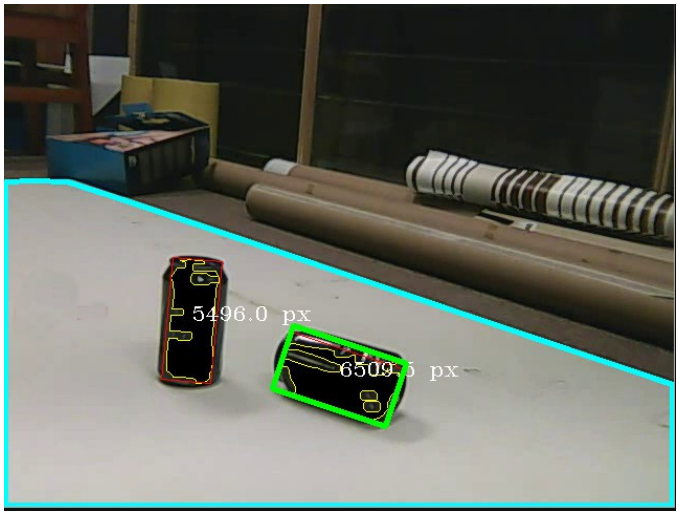

Fig. 2. Processed Images

Then he makes a threshold [8] to detect black, filtered with sand mask, and detected contours (contours marked in yellow in Fig 2). The sand mask ensures that black which is detected elements are cans not from outside the court, nor strong shadows.

After filtering by size of rectangle area that includes each of the detected contours (marked in red in Figure 2), we choose the rectangle closest to the robot (marked green in Figure 2) or whatever it has the largest area.

Having chosen a rectangle, it is assumed to a can and is written in to the data structure with the keys 'Camara :: lata_x', 'Camara :: lata_y' and 'Camera :: find'.

In case you can not detect a rectangle is set to 'Camara :: find' value 'FALSE'.

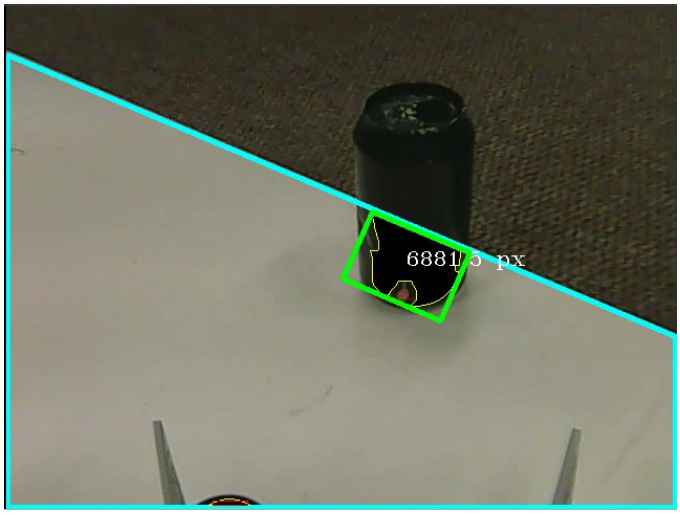

Fig. 3. Solution problems

The way that was chosen to discard external noise makes the sensor to have problems detecting some edge cases . An example is shown in Figure 3, which can be seen that sand mask causes it to lose part of the can, becoming more noticeable when the robot approaches the can, because at a time only is going to can and carpet, no white background (in this case simulating the sand), and can not be detected.

Detection of red trash was made the same way, only instead of doing the threshold by black it was done by levels of red. The driver chooses the larger rectangle and writes under the keys 'Camara :: tacho', 'Camara :: tacho_x' and 'Camera :: tacho_y'.

C. *Behavior Architecture*

To solve the problem we used a behavioral architecture implemented by the group, inspired by the subsumption of LeJOS implementation [6]. It is characterized by being based on the paradigm reactive behavior-based control.

Like LeJOS implementing behaviors are chosen for their priority and this is given by its position in an array in which behaviors are found.

There is an Arbitrator in charge of asking each behavior in its array of behaviors if anyone is ready to be executed by calling TakeControl function (): boolean. From the way it runs, the first to return true is going to be executed. The releaser depends on the behavior and is encoded in TakeControl function, which in all cases corresponds to check to the data structure mentioned in section sensors in a particular key (unless wander behavior that always returns true).

The fixed action pattern is represented by states within each behavioral which in general depend on the time spent. An example is the behavior to avoid water that once the release triggers the behavior (gray sensor type a value less than a limit on to the data structure) takecontrol is returning true but no longer is detecting water with sensors for a configurable time.

The following lists and describes the behavior employed:

1) CompWander

The behavior is based on wandering the scenario with the objective of finding objects or obstacles. within the behavior We keep a state, telling us is being done at all times, changing over time to other states. What you have is actually a non-deterministic state machine to achieve a random motion. The movements performed are: move forward, turn left and turn right.

2) CompLata

The behavior is activated when the key value 'Camara :: encontro' within the data structure is set to "True". When active, it gets the value of another key in the data structure, 'Camara :: lata_x', which tells me the position along the x axis of the object within the visual field of the robot. With this value focus the robot path to approach the target. To center the target, making turns to the right and left and when centered proceeds to be aimed in the correct position in order to pick it up. To see if the target is within an acceptable distance, we need to know the position in the Y axis value stored in 'Camera :: lata_y' within the data structure. With this value we can regulate the speed at which we aproach us to to target.

3) CompCargarLata

Behavior is activated when the camera sensor writes in the data structure that a can was found. It activates the clamps and grasps what it has in front the clamps. The behavior continues to operate while the clamps are moving.

4) CompEvitar

This behavior is designed to prevent collisions with obstacles, in this case chairs, but also has the ability to avoid colliding with obstacles similar to a high chair. The behavior is triggered when the value stored in any of the keys 'SensorDistancia :: <puerto_u4b>' exceeds a predefined threshold value. In that case the robot executes a sequence of actions to avoid colliding with the obstacle.

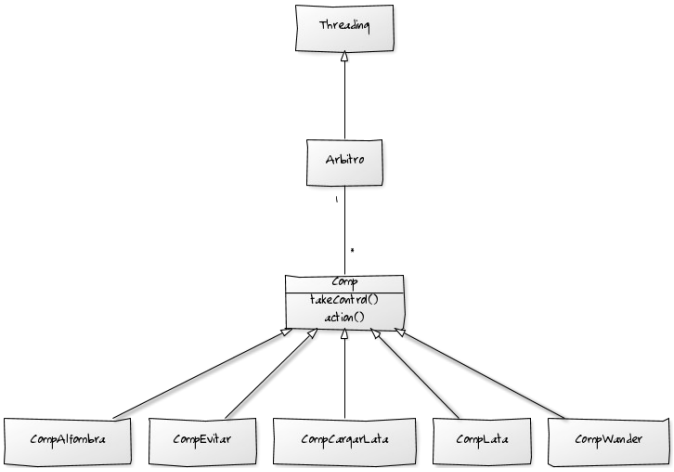

Fig. 4. Behaviors Architecture

5) CompAgua

The behavior aims to avoid falling out of the scenario, in fact if we look at the context of the challenge, it would not fall into the water. Can also be regarded as a reflection in the presence of water under the robot. As in the previous case, the behavior is activated when the value stored in one of the keys 'SensorGrises :: <puerto_u4b>' exceeds a preset limit. In that case the robot executes a sequence of actions that would avoid falling into the water.

IV. Physical Structure

A. Chasis

The chassis of the robot is developed with acrylic and aluminum which gives both stability and lightness not to demand AX-18 motors that are responsible for moving the tracks.

B. Clamps

The clamps are made of aluminum arm and wire which allows to collect and filter the sand cans which could grab when loading it, both clamps are driven by motors AX-12

C. Cans deposit

Cans deposit is designed to be as light as possible. To them was designed using cartonplast, which is very rigid and lightweight. To deposit the cans in the dumpster, the container has at its core a AX-12 engine which allows it to swing and so download cans.

V. Conclusion

Building a robot requires an extensive commitment both to see mechanical, electronic and computational aspects of the robot. Each of these tasks is very different in the area to be investigated the possibilities to improve the performance of the robot to the required task.

Since the robot still has some details to improve, this solution is not final, and these improvements will be reflected in subsequent solutions in order to progress towards the LARC.

References

[1] Reglamento oficial para la categoría IEEE Open, http://ewh.ieee.org/reg/9/robotica/Reglas/LARC2012_open-rules_v1.1.pdf, visitada Setiembre 2013.

[2] Proyecto Butia, http://www.fing.edu.uy/inco/proyectos/butia, visitada Setiembre 2013

[3] Robotis - Dynamixel, http://www.robotis.com/xe/dynamixel_en, visitada Setiembre 2013

[4] libopencv Open Source Computer Vision, http://opencv.org/, visitada Setiembre 2013

[5] Modelo de color HSV, http://es.wikipedia.org/wiki/Modelo_de_color_HSV, visitada Septiembre 2013

[6] Java for Lego Mindstorms, http://lejos.sourceforge.net visitada Setiembre 2013

[7] Convex Hull. http://docs.opencv.org/doc/tutorials/imgproc/shapedescriptors/hull/hull.html visitada Setiembre 2013

[8] Basic Thresholding Operations, http://docs.opencv.org/doc/tutorials/imgproc/threshold/threshold.html visitada Setiembre 2013

[9] Robots Autónomos y Aprendizaje por refuerzo, http://www.fleifel.net/ia/robotsyaprendizaje.php visitada Setiembre 2013