Introduction to UniboDISI DDR-Robots (2018)

Antonio Natali

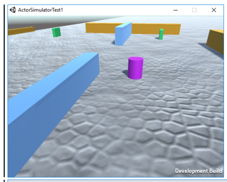
Alma Mater Studiorum – University of Bologna
via dell'Università 50,47521 Cesena, Italy,
Viale Risorgimento 2,40136 Bologna, Italy
antonio.natali@studio.unibo.it

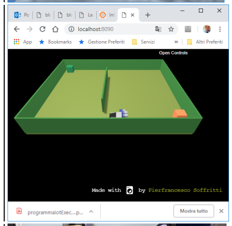
Table of Contents

1 Virtual and Real robots

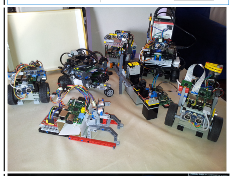
Our company (`Unibo`) has developed:

A virtual environment (named `U-Env`) based on the `Unity` system (see Subsection 2.1), that includes a virtual robot that accepts commands sent on a `TCP` connection on port `8090`.

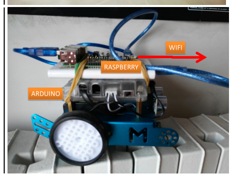




A virtual environment (named `W-Env`) built in `JavaScript` (see Subsection 2.2.1), that includes a virtual robot that accepts commands sent on a `TCP` connection on port `8999`.



A physical `ddr` robot built in-house (see Section 3), modelled as an observable `POJO`.



A physical robot based upon a `mBot robot` (see Section 4).

2 The virtual environments

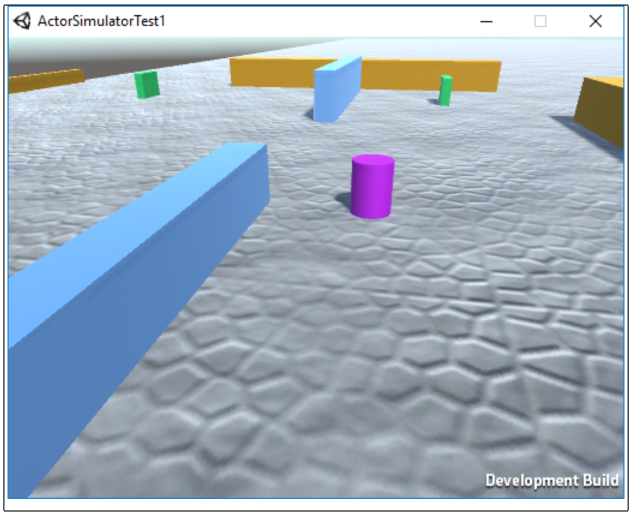
During software analysis and testing, it could be preferable to use virtual devices rather real, physical devices.

2.1 A virtual environment based on Unity (uenv)

Unity can be used as an advanced simulation environment, able to provide a virtual working environment and sensor data.

Our virtual environment built upon `Unity` (called `uenv`) is included in the file:[1] `it.unibo.issMaterial/issdocs/Lab/virtualRobot.zip`.

Let us *unzip* this file, and run `VirtualRobotE80.exe`. We obtain a scene showing an environment made of a set of walls and fixed obstacles, a mobile obstacle (the cylinder) and a sonar (the small boxes in green, named `sonar1` and `sonar2`):



The original Unity environment has been modified to interact with *QActor* systems.

The operation `createSimulatedActor` can be used in a *QActor* model to 'inject' into the virtual environment a qactor (to be called `rover`, at the moment) that can be moved in a Unity scene by using proper commands (`onward, backwards, left, right, stop)`.

When the `rover` is intercepted by one of the sonar, the (modified) Unity system emits the event `sonar : sonar(SONAR, TARGET, DISTANCE)`.

Moreover, the virtual game object that represents the `rover` is equipped (in its front) with a sonar, that emits the event `sonarDetect : sonarDetect(X)` when detects an obstacle.

[1] You could clone the project `it.unibo.issMaterial` by executing: `git clone https://github.com/anatali/iss2018.git`

Both these events that cam be handled by our applications. For example:

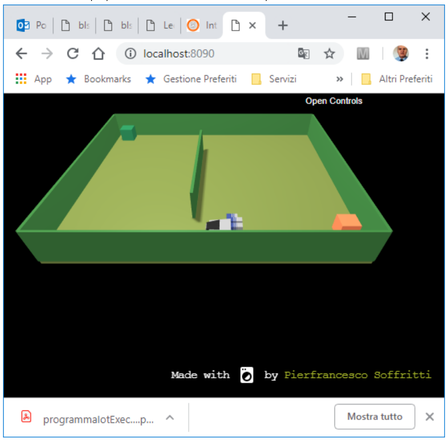
```
System testUnityRover
Event sonarDetect  : sonarDetect(X) //From (virtual robot) sonar
Event sonar        : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar

Context ctxRover ip [ host="localhost" port=8070 ]
EventHandler evh for sonarDetect , sonar -print ;

QActor rover context ctxRover {
   Plan init normal [
         println("rover START");
         javaOp "workWithUnity(\"localhost\")" ;
         javaOp "createSimulatedActor(\"rover\",\"Prefabs/CustomActor\")" ;
         right 50 time ( 1000 ) //position
      ]
      switchTo moveVitualRobot

   Plan moveVitualRobot [
      println("moveVitualRobot")
   ]
   reactive onward 40 time ( 5000 )
         whenEnd -> endOfMove
         whenTout 30000 -> handleTout
         whenEvent sonarDetect -> handleObstacle
         or whenEvent sonar  -> handleSonar
   finally repeatPlan

   Plan handleSonar resumeLastPlan [
      onward 50 time ( 300 ) ; //out of sonar range
      stop 50 time ( 1000 )   //stop for a while ...
   ]
   Plan handleObstacle resumeLastPlan [ backwards 50 time ( 3500 ) ]
   Plan endOfMove resumeLastPlan [ println("endOfMove") ]
   Plan handleTout  [ println("handleTout") ]

}
```

Listing 1.1. `testUnityRover.qa`

The point to highlight here is that, when the virtual robot (`rover`) is intercepted by a sonar, the modified Unity system emits the *QActor* event `sonar : sonar(SONARNAME, TARGET, DISTANCE)` where `SONARNAME` is `sonar1` or `sonar2`. Moreover, the sonar on the front of the `rover` emits the event `sonarDetect : sonarDetect(TARGET)` when it detects an obstacle.

2.2 A virtual environment based on JavaScript (wenv)

This virtual environment (called wenv) has been built by Pierfrancesco Soffritti[2] using the https://threejs.org/ JavaScript library.



The scene is described in a configuration file. For example:

```
const config = {
    floor: {
        size: { x: 34, y: 30 }
    },
    player: {
         position: { x: 0.065, y: 0.075 },
         speed: 0.2
    },
    sonars: [
         {
            name: "sonar2",
            position: { x: 0.94, y: 0.88},
            senseAxis: { x: true, y: false }
        }
     ],
    movingObstacles: [
    ],
    staticObstacles: [
        {
        name: "wallUp",
        centerPosition: { x: 0.5, y: 1},
        size: { x: 1, y: 0.01}
        },
         {
            name: "wallDown",
            centerPosition: { x: 0.5, y: 0},
            size: { x: 1, y: 0.01}
        },
       {
            name: "wallLeft",
            centerPosition: { x: 0, y: 0.5},
            size: { x: 0.01, y: 1}
```

[2] See https://github.com/PierfrancescoSoffritti/ConfigurableThreejsApp

```
        },
        {
            name: "wallRight",
            centerPosition: { x: 1, y: 0.5},
            size: { x: 0.01, y: 1}
        },
        {
            name: "obstacle",
            centerPosition: {x: 0.45, y: 0.49},
            size: { x: 0.01, y: 0.57}
        }
    ]
}

export default config;
```

Listing 1.2. `sceneConfigSimple.js`

The virtual robot accepts commands sent on a `TCP` connection on port `8999`.

```
moveForward :  `{ "type": "moveForward", "arg": 300 }`
moveBackward :  `{ "type": "moveBackward", "arg": 300 }`

turnRight :   `{ "type": "turnRight", "arg": 300 }`
turnLeft : `{ "type": "turnLeft", "arg": 300 }`

stop :    `{ "type": "alarm" }`
```

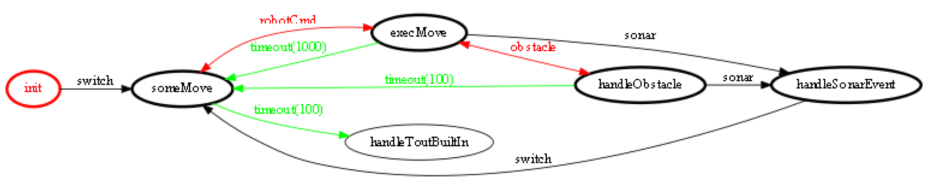
Moreover, the virtual environment sends to the connected client (via `TCP` on the port `8999`) the following data:

```
webpage-ready : `{ "type": "webpage-ready, "arg": {} }`
sonar-activated : `{
     "type:" "sonar-activated",
     "arg": { "sonarName": "sonarName", "distance": 1, "axis": "x" }
}`
collision : `{
    "type": "collision",
    "arg": { "objectName": "obstacle-1" }
}`
```

An example of usage at this basic level can be found in: `it.unibo.mbot.virtual.clientTcp.java`.

2.2.1 A virtual environment based on JavaScript (`wenv`) .

An example of usage of the `wenv` from a *QActor* model represented by the following (generated) FSM diagram



can be written as follows:

```
System demoRobotInWenv
Dispatch obstacle : obstacle(TARGET)
Dispatch robotCmd : cmd(X)              //X=w|a|s|d|h

Event sonarDetect : obstacle(X)            //From wenv-robot sonar
Event sonar     : sonar(SONAR, TARGET, DISTANCE) //From wenv sonar

Context ctxRobotInWenv  ip [ host="localhost" port=8098 ] -httpserver
EventHandler evobstacle for sonarDetect {
    forwardEvent robotdemoinwenv -m obstacle
};

QActor robotdemoinwenv context ctxRobotInWenv {
Rules{
    move(w).   move(d).   move(w).   move(s).   move(a).
    move(d).   move(w).   move(a).   move(s).   move(h).
 }
    State init normal [
        println("robotdemoinwenv STARTS");
        javaRun it.unibo.robotVirtual.basicRobotExecutor.setUp("localhost")
    ]
    switchTo someMove

    State someMove[
        [ !? move(X)] println( doingMove(X) );
        [ ?? move(X) ] selfMsg robotCmd : cmd(X)
    ]
    transition stopAfter 100
        whenMsg robotCmd -> execMove

    State execMove resumeLastPlan[
        onMsg robotCmd : cmd(MOVE) ->
            javaRun it.unibo.robotVirtual.basicRobotExecutor.doMove( MOVE )
    ]
    transition whenTime 1000 -> someMove
        whenMsg obstacle -> handleObstacle,
        whenEvent sonar -> handleSonarEvent

    State handleObstacle resumeLastPlan [ printCurrentMessage ]
    transition whenTime 100 -> someMove
        whenEvent sonar -> handleSonarEvent

    State handleSonarEvent [ printCurrentEvent ]
    switchTo someMove
 }
```

Listing 1.3. `demoRobotInWenv.qa`

Note that:

- The moves to do are specified as a sequence of facts in tuProlog-like syntax.
- The Applcation designer has written the class `it.unibo.robotVirtual.basicRobotExecutor`that provides static methods to `setUp` the connection with the `wenv` and to send commands (`doMove`) to the virtual robot.
- The state `execMove` is 'reactive' to the event `sonar` and to the message `obstacle`.

– The message obstacle is generated by an event handler related to the event sonarDetect.

In fact, the In fact, the wenv generates a stream of sonar events as long as the virtual robot is under the action area of a sonar, while the obstacle event is generated only once (when the virtual robot hits an obstacle).

2.2.2 The Application class basicRobotExecutor .

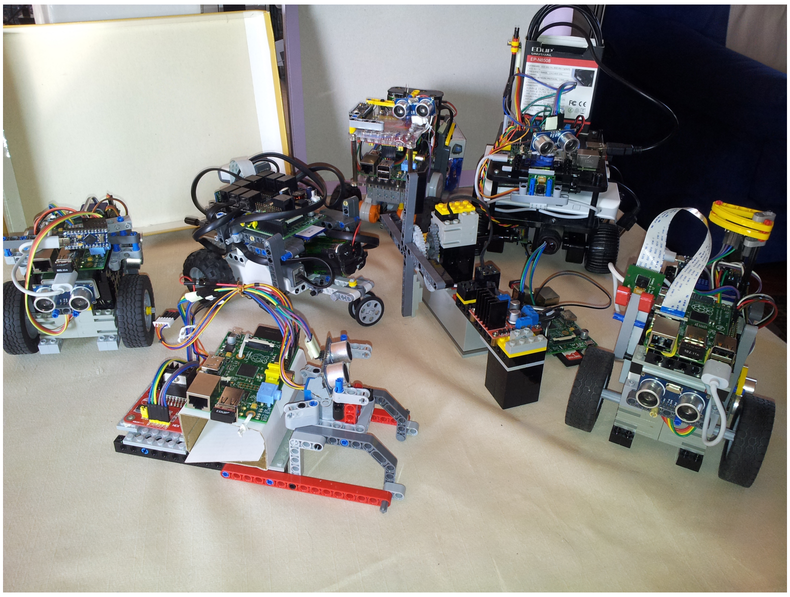
The Application class basicRobotExecutor exploits the basic clientTcp to implement the logical operations required at model level. The name of the class starts with a lower-case letter for a constraint imposed by the current implementation of the *QActor* software factory. Moreover, each operation provided by the Java class must have as its first argument a variable of type QActor.

```
package it.unibo.robotVirtual;
import it.unibo.qactors.akka.QActor;
import it.unibo.utils.clientTcpForVirtualRobot;

public class basicRobotExecutor {
	public static void setUp(QActor qa, String hostNameStr) {
	    try {
	        qa.println("robotVirtual setUp " + hostNameStr );
	        clientTcpForVirtualRobot.initClientConn(qa, hostNameStr, "8999");
	    } catch (Exception e) {
	        e.printStackTrace();
	    }
	}
	public static void doMove( QActor qa, String cmd ) { //Args MUST be String
//	    qa.println("robotVirtual doMove " + cmd );
	  switch( cmd ) {
	    case "h" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'alarm', 'arg': 0 }");break;
	    case "w" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'moveForward', 'arg': -1 }");break;
	    case "a" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'turnLeft', 'arg': 400 }");break;
	    case "d" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'turnRight', 'arg': 400 }");break;
	    case "s" : clientTcpForVirtualRobot.sendMsg(qa,"{'type': 'moveBackward', 'arg': -1
	        }");break;
	  }
	}
}
```

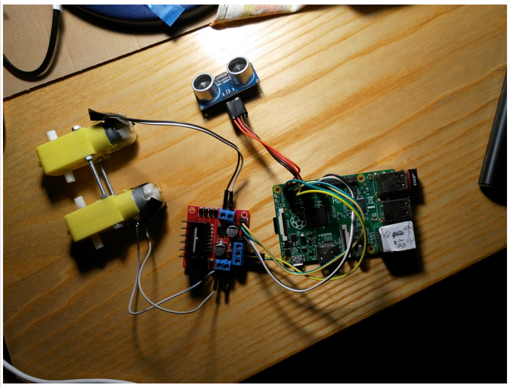
Listing 1.4. basicRobotExecutor.java

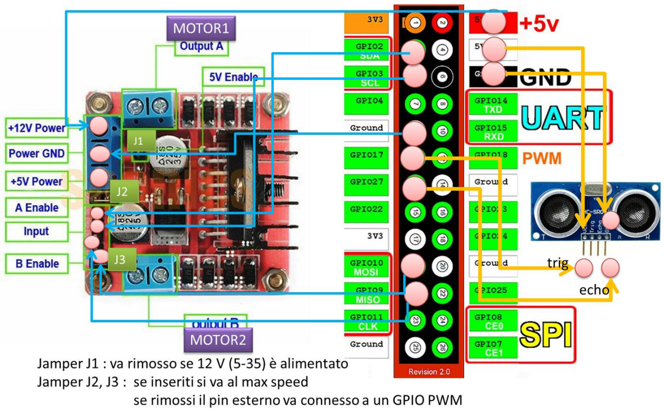
3 The in-house (nano) robot

From the physical point of view, a `BaseRobot` is a custom device built with low-costs components including sensors, actuators and processing units like *RaspberryPi* and *Arduino*. Examples are given in the following picture:



The hardware components of a `BaseRobot` and their configuration can be quite different from robot to robot. For example, a DDR-robot built around a RaspberryPi could have the basic hardware structure shown hereunder:





In this case, the correct configuration of the hardware connections can be tested by some simple bash code :

```bash
#!/bin/bash
# -------------------------------------------------------------------
# nanoMotorDriveA.sh
# test for nano0
# Key-point: we can manage a GPIO pin by using the GPIO library.
# On a PC, edit this file as UNIX
# -------------------------------------------------------------------

in1=2 #WPI 8 BCM 2 PHYSICAL 3
in2=3 #WPI 9 BCM 3 PHYSICAL 5
inwp1=8
inwp2=9

if [ -d /sys/class/gpio/gpio2 ]
then
 echo "in1 gpio${in1} exist"
 gpio export ${in1} out
else
 echo "creating in1 gpio${in1}"
 gpio export ${in1} out
fi

if [ -d /sys/class/gpio/gpio3 ]
then
 echo "in2 gpio${in2} exist"
 gpio export ${in2} out
else
 echo "creating in2 gpio${in2}"
 gpio export ${in2} out
fi

gpio readall

echo "run 1"
 gpio write ${inwp1} 0
 gpio write ${inwp2} 1
 sleep 1.5

echo "run 2"
 gpio write ${inwp1} 1
 gpio write ${inwp2} 0
 sleep 1.5

echo "stop"
 gpio write ${inwp1} 0
 gpio write ${inwp2} 0

gpio readall
```

Listing 1.5. `nanoMotorDriveA.sh`

3.1 Beyond the hardware level

However, some software layer is required to hide configuration differences as much as possible and to build a 'technology independent' layer, to be used by application designers.

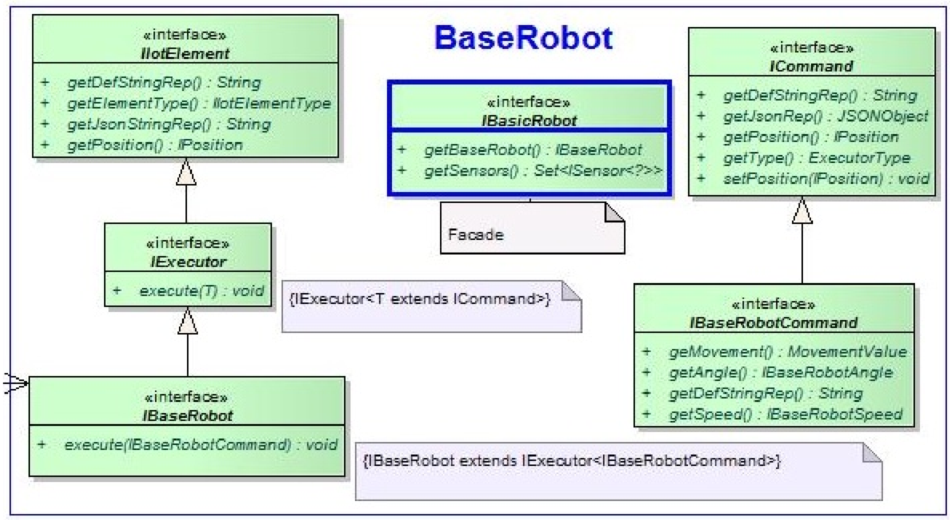
A software layer of this kind is provided by the library `labbaseRobotSam.jar`. More specifically:

it.unibo.lab.baseRobot	The basic software for differential drive robots that are able to move and to acquire sensor data. Library: *labbaseRobotSam.jar*.
it.unibo.robotRaspOnly.BasicRobotUsageNaive in project *it.unibo.mbot2018*	Example of the usage of the `API` of a BaseRobot.

3.2 A model for the BaseRobot

The main goal of the *labbaseRobotSam.jar* library is to simplify the work of an application designer by exposing at application level a very simple model of a `BaseRobot`:

- As regards the `structure`, a `BaseRobot` can be viewed as a entity composed of two main parts:
  - An executor (with interface `IBaseRobot`), able to move the robot according to a a prefixed set of movement commands (`IBaseRobotCommand`)
  - A set of `GOF` -observable sensors (each with interface `ISensor`), each working as an active source of data.
- As regards the `interaction`, a `BaseRobot` can be viewed as a `POJO` that implements the interface `IBaseRobot`, while providing a (possibly empty) set of observable sensors;
- As regards the `behavior`, a `BaseRobot` is an object able to execute `IBaseRobotCommand` and able to update sensor observers defined by the application designer.



The interface `IBasicRobot` is introduced as a (`GOF`) *Facade* for the model.

```
package it.unibo.iot.baseRobot.hlmodel;
import java.util.Set;
import it.unibo.iot.executors.baseRobot.IBaseRobot;
import it.unibo.iot.sensors.ISensor;

public interface IBasicRobot {
    public IBaseRobot getBaseRobot();   //selector
    public Set<ISensor<?>> getSensors(); //selector
}
```

Listing 1.6. `IBasicRobot.java`

3.3 The BasicRobot class

The class `BasicRobot` provides a factory method to create a `BaseRobot` and to select its main components.

```
package it.unibo.iot.baseRobot.hlmodel;
import java.util.Set;
import it.unibo.iot.configurator.Configurator;
import it.unibo.iot.executors.baseRobot.IBaseRobot;
import it.unibo.iot.sensors.ISensor;

public class BasicRobot implements IBasicRobot{
private static IBasicRobot myself = null;
public static IBasicRobot getRobot(){
    if( myself == null ) myself = new BasicRobot();
    return myself;
}
public static IBaseRobot getTheBaseRobot(){
    if( myself == null ) myself = new BasicRobot();
    return myself.getBaseRobot();
}
//--------------------------------------------
private Configurator configurator;
private IBaseRobot robot ;
    //Hidden constructor
    protected BasicRobot() {
        init();
    }
    protected void init(){
        configurator = Configurator.getInstance();
        robot       = configurator.getBaseRobot();
    }
    public IBaseRobot getBaseRobot(){
        return robot;
    }
    public Set<ISensor<?>> getSensors(){
        Set<ISensor<?>> sensors = configurator.getSensors();
        return sensors;
    }
}
```

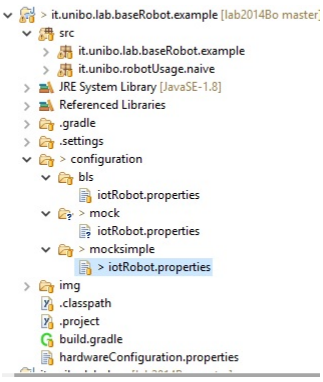
Listing 1.7. `BasicRobot.java`

This class shows that the work to set-up and access to the internal structure of a `BaseRobot` is delegated to a `Configurator`. This Configurator reads the specification of the robot structure written in a file named *iotRobot.properties* (see Subsection 3.5).

3.4 Using a BaseRobot

Let us introduce a robot with configuration named `mocksimple` (see Subsection 3.5), equipped with two motors and a distance sensor, all simulated.

3.4.1 The project workspace The application designer must organize its project workspace as shown in the following snapshot:

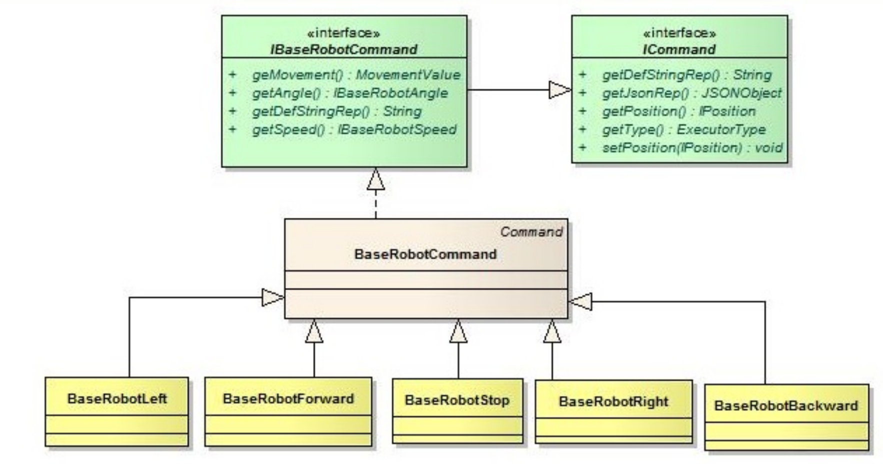


3.4.2 The code The application: *i)* first creates a sensor observer and adds it to all the sensors; *ii)* then it tells the robot to execute the commands (sent from an user console) it is able to understand.

```
private final IBaseRobotSpeed SPEED_LOW = new BaseRobotSpeed(BaseRobotSpeedValue.ROBOT_SPEED_LOW);
private final IBaseRobotSpeed SPEED_MEDIUM = new
     BaseRobotSpeed(BaseRobotSpeedValue.ROBOT_SPEED_MEDIUM);
private final IBaseRobotSpeed SPEED_HIGH = new BaseRobotSpeed(BaseRobotSpeedValue.ROBOT_SPEED_HIGH);
private IBasicRobot basicRobot;
private IBaseRobot robot ;

    public BasicRobotUsageNaive() {
        basicRobot = BasicRobot.getRobot();
        robot      = basicRobot.getBaseRobot();
//      addObserverToSensors(basicRobot);
    }
    public void handleUserCommands() {
        try {
            while(true) {
                int v    =  System.in.read();
                if( v == 13 || v == 10 ) continue;
                char cmd = (char)v;
                System.out.println( "INPUT= " + cmd + "(" + v +")");
                executeTheCommand( cmd );
```

```java
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void executeTheCommand( char cmd ) {
        IBaseRobotCommand command = null;
        switch( cmd ) {
            case 'h' : command = new BaseRobotStop(SPEED_LOW );break;
            case 'w' : command = new BaseRobotForward(SPEED_HIGH );break;
            case 's' : command = new BaseRobotBackward(SPEED_HIGH );break;
            case 'a' : command = new BaseRobotLeft(SPEED_MEDIUM );break;
            case 'd' : command = new BaseRobotRight(SPEED_MEDIUM );break;
            default: System.out.println( "Sorry, command not found");
        }
        if( command != null ) robot.execute(command);
    }
    protected void addObserverToSensors( ){
        ISensorObserver observer = new SensorObserver();
//      for (ISensor<?> sensor : basicRobot.getSensors()) {
//          System.out.println( "doJob sensor= " + sensor.getDefStringRep() + " class= " + sensor.getClass().getName() );
//          sensor.addObserver(observer);
//      }
        addObserverToSensors(observer);
    }

    public void addObserverToSensors( ISensorObserver observer ){
        for (ISensor<?> sensor : basicRobot.getSensors()) {
            System.out.println( "adding observer to sensor: " + sensor.getDefStringRep() );
            sensor.addObserver(observer);
        }
    }

 public static void main(String[] args) throws Exception{
        new BasicRobotUsageNaive().handleUserCommands();
 }
}
```

Listing 1.8. `BasicRobotUsageNaive.java`



3.5 The work of the Configurator

An object of class *it.unibo.iot.configurator.Configurator*:

1. first looks at the file `hardwareConfiguration.properties` to get the name of the robot (e.g. mock)
2. then, it consults the file `iotRobot.properties` into the directory `configuration/mock`

For each specification line, the `Configurator` calls (by using Java reflection) a factory method of the specific `DeviceConfigurator` class associated to the name of the robot.

For example, let us consider a Mock robot equipped with two motors and a distance sensor, all simulated: (file *configuration/mocksimple/iotRobot.properties*):

```
# ==================================
# ioRobot.properties for mocksimple robot
# Pay attention to the spaces
# ==================================
# --------- MOTORS ---------
motor.left=mock
motor.left.private=false
#
motor.right=mock
motor.right.private=false
# --------- SENSORS ---------
distance.front=mock
distance.front.private=false
# --------- COMPOSED COMPONENT ---------
actuators.bottom=ddmotorbased
actuators.bottom.name=motors
actuators.bottom.comp=motor.left,motor.right
actuators.bottom.private=true
# --------- MAIN ROBOT ---------
baserobot.bottom=differentialdrive
baserobot.bottom.name=mocksimple
```

```
baserobot.bottom.comp=actuators.bottom
baserobot.bottom.private=false
```

Listing 1.9. `configuration/mocksimple/iotRobot.properties`

The Configurator calls (using an object of class `IotComponentsFromConfiguration`):

- `getMotorDevice` of *it.unibo.iot.device.mock.DeviceConfigurator* (for **motor.left=mock**)
- `getBaseRobotDevice` of *it.unibo.iot.device.differentialdrive.DeviceConfigurator* (for **baserobot.bottom=differentialdrive**)
- `getDistanceSensorDevice` of *it.unibo.iot.device.mock.DeviceConfigurator* (for **distance.front=mock**)
- `getMotorDevice` of *it.unibo.iot.device.mock.DeviceConfigurator* (for **motor.right=mock**)
- `getActuatorsDevice` of *it.unibo.iot.device.ddmotorbased.DeviceConfigurator* (for **actuators.bottom=ddmotorbased**)

3.6 From mocks to real robots

In order to use a physical robot rather than a `Mock` robot, the software designer must simply change the specification of the robot configuration; the application code is unaffected. For example, to use our standard '`nano`' robots, we have to include into the `configuration/nano` directory the following configuration file:

```
# ==================================
# ioRobot.properties for nano robot
# Pay attention to the spaces
# ==================================
# --------- MOTORS ---------
motor.left=gpio.motor
motor.left.pin.cw=8
motor.left.pin.ccw=9
motor.left.private=false
#
motor.right=gpio.motor
motor.right.pin.cw=12
motor.right.pin.ccw=13
motor.right.private=false
# --------- SENSORS ---------
distance.front_top=hcsr04
distance.front_top.trig=0
distance.front_top.echo=2
distance.front_top.private=false
# --------- COMPOSED COMPONENT ---------
actuators.bottom=ddmotorbased
actuators.bottom.name=motors
actuators.bottom.comp=motor.left,motor.right
actuators.bottom.private=true
# --------- MAIN ROBOT ---------
baserobot.bottom=differentialdrive
```

```
baserobot.bottom.name=nano
baserobot.bottom.comp=actuators.bottom
baserobot.bottom.private=false
```

Listing 1.10. `configuration/nano/iotRobot.properties`

3.7 Sensors and Sensor Data

The current version of the `BaseRobot` system implements the following sensors:

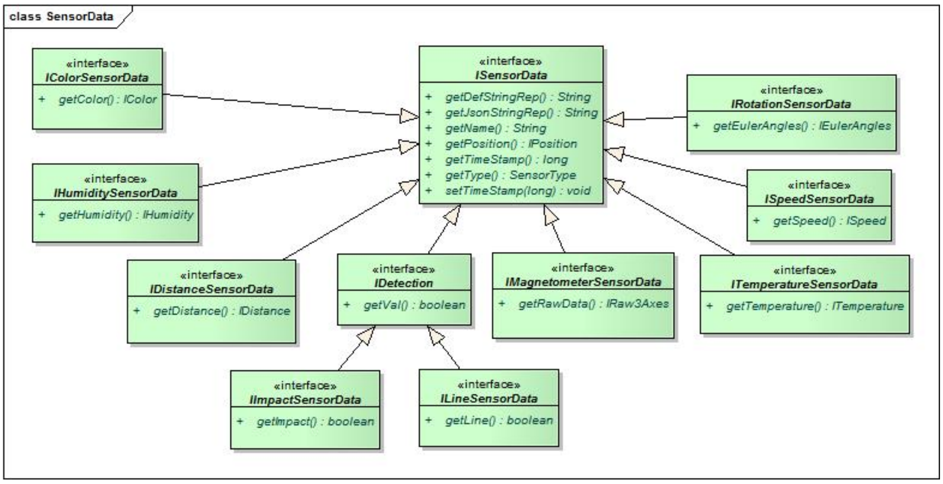
`RobotSensorType: Line | Distance | Impact | Color | Magnetometer`

Each sensor:

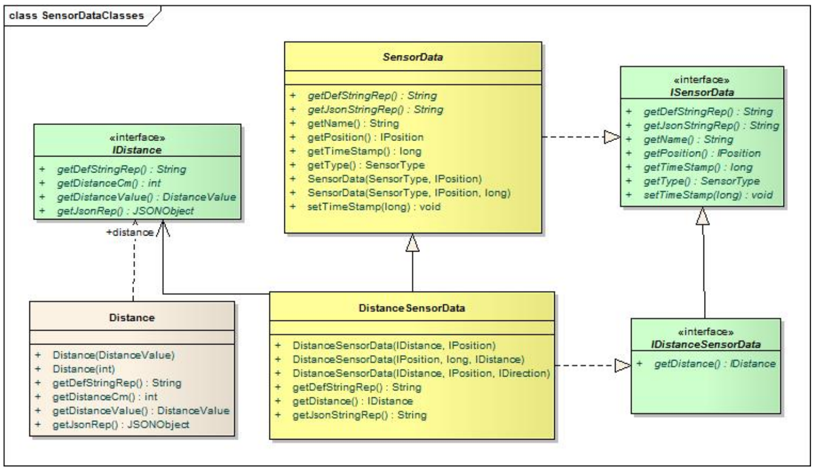
- is associated to a position that can assume one of the following values:

  ```
  DONTCARE|
  FRONT | RIGHT | LEFT | BACK | TOP | BOTTOM |
  FRONT_RIGHT | FRONT_LEFT | BACK_RIGHT | BACK_LEFT |
  TOP_RIGHT | TOP_LEFT | BOTTOM_RIGHT | BOTTOM_LEFT |
  FRONT_TOP | BACK_TOP | FRONT_TOP_LEFT | FRONT_TOP_RIGHT |
  FRONT_RIGHT_TOP | FRONT_LEFT_TOP | BACK_RIGHT_TOP | BACK_LEFT_TOP
  ```

- is a source of data, each associated to a specific class and interface:



Each class related to sensor data inherits from a base class `SensorData`. For example:



The class `SensorData` provides operations to represent data as strings in two main formats: *i)* in `Prolog` syntax and *ii)* in `Json` syntax. For example:

3.7.1 Sensor data representation in Prolog (high level)

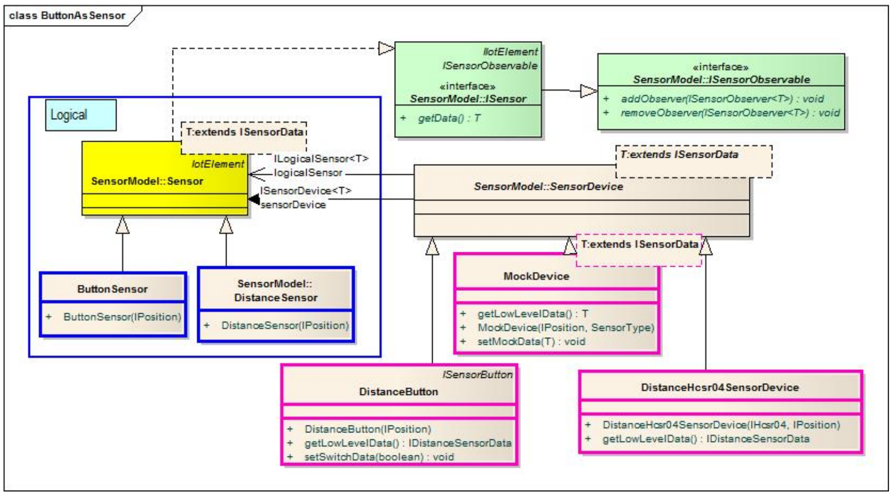
COLOR	`color(255 255 255, front)`
DISTANCE	`distance(43,forward, front)`
IMPACT	`impact(touch/loss, front)`
LINE	`line(lineLeft/lineDetected, bottom)`
MAGNETOMETER	`magnetometer(x(50),y(100),z(0), front)`

3.7.2 Sensor data representation in Json (low level)

COLOR	`"p":"f","t":"c","d":"color":"r":255,"b":255,"g":255,"tm":148...`
DISTANCE	`"p":"f","t":"d","d":"cm":43,"tm":14...`
IMPACT	`"p":"f","t":"i","d":"detection":"touch","tm":14...`
LINE	`"p":"b","t":"l","d":"detection":"lineDetected","tm":14...`
MAGNETOMETER	`"p":"f","t":"m","d":"raw3axes":"x":50,"y":100,"z":0,"tm":14...`

3.8 Sensor model

The sensor subsystem of the `BaseRobot` is based on the class `Sensor` that represents a sensor from the logical point of view. Each sensor is associated to a class that inherits form `Sensor`.

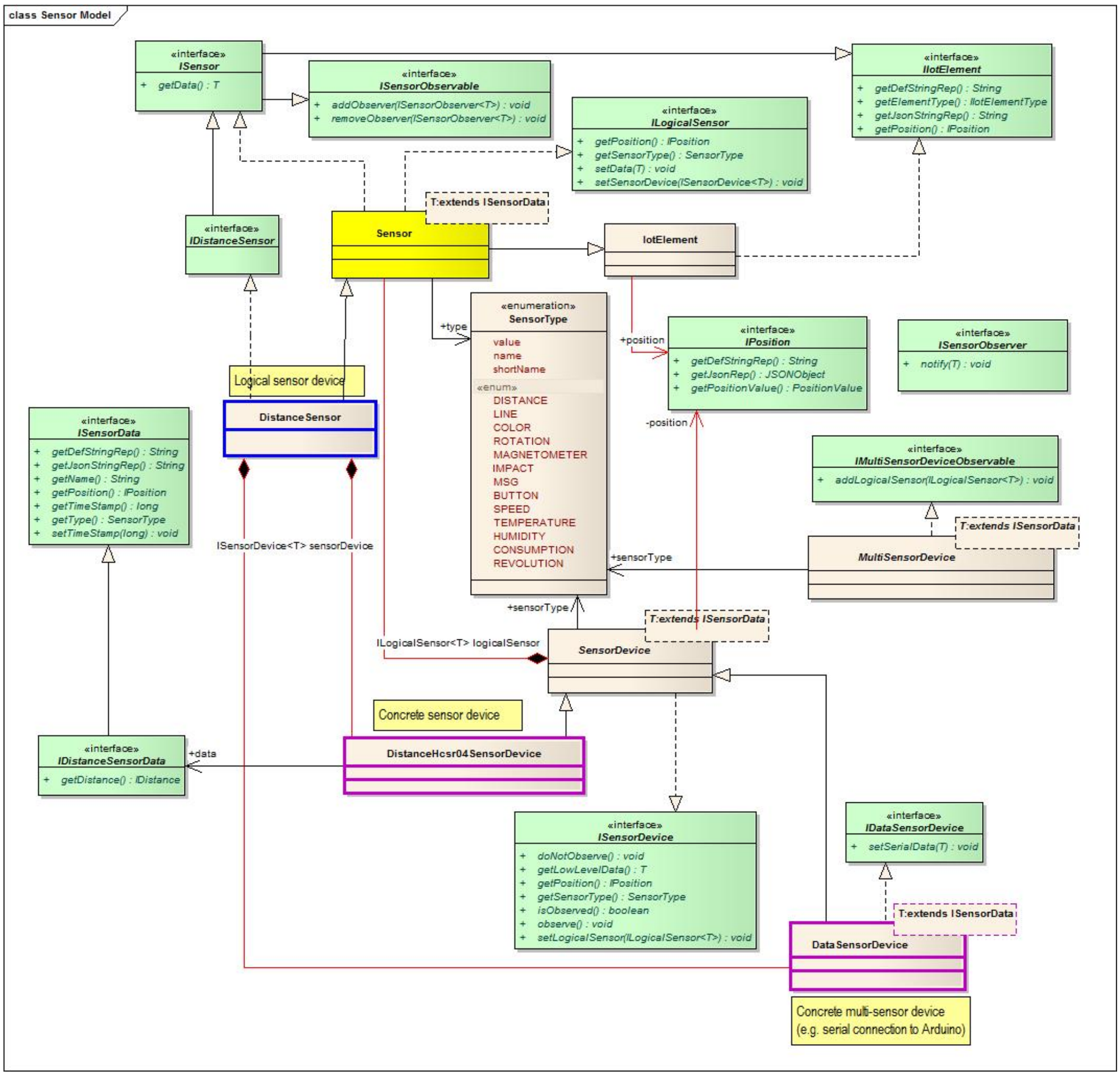


The model reported in the picture above shows that:

- A `DistanceSensor` is a logical `Sensor` associated (by the *Configurator*) to a concrete device (e.g. `DistanceHcsr04SensorDevice`). The same is true for a `ButtonSensor` (impact).
- A `DistanceHcsr04SensorDevice` is a concrete `SensorDevice` that updates its logical sensor when it produces a value. The same is true for a `DistanceButton` (impact). The diagrma shows also a `MockDevice` that can be used to simulate the behavior of the supported sensors.
- Any `Sensor` is an observable entity that, when updated from its concrete device, updates the registered application observers.

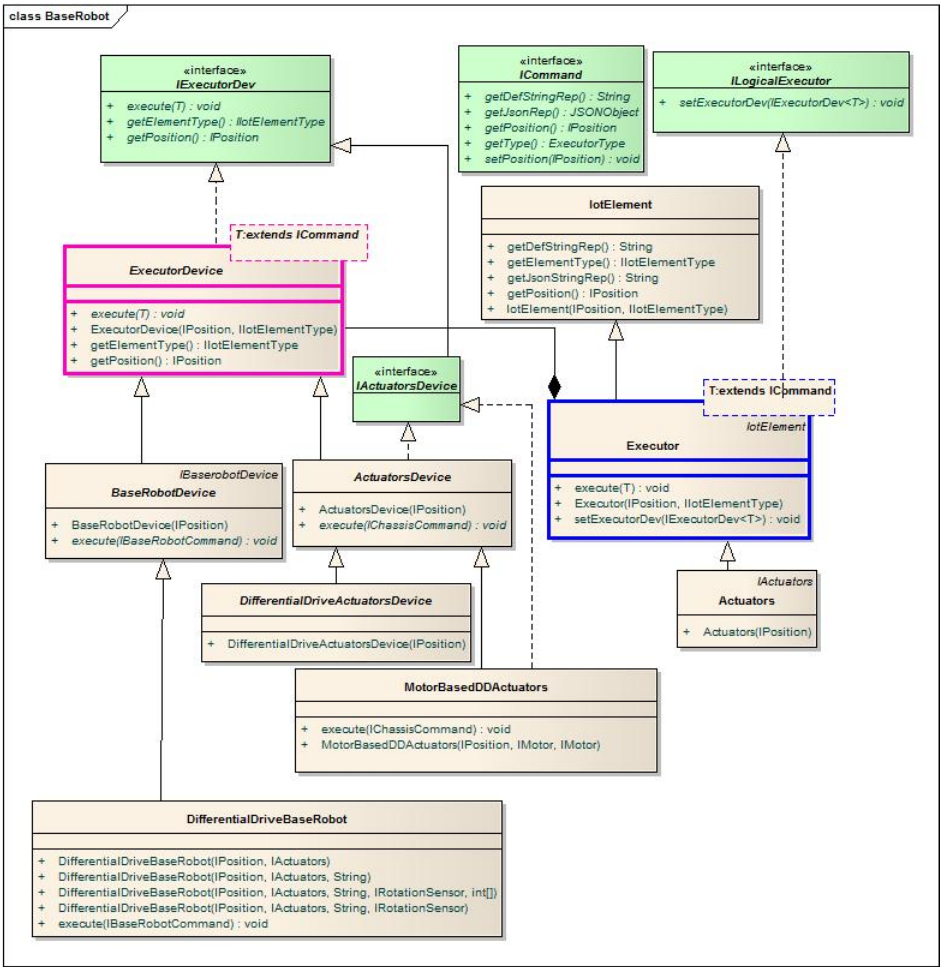
In this way, according to the `GOF` pattern *Bridge*, the `Sensor` abstraction hierarchy is decoupled from the hierarchy of `SensorDevice` implementation.

A more detailed picture is reported hereunder:



3.9 Actuators and Executors

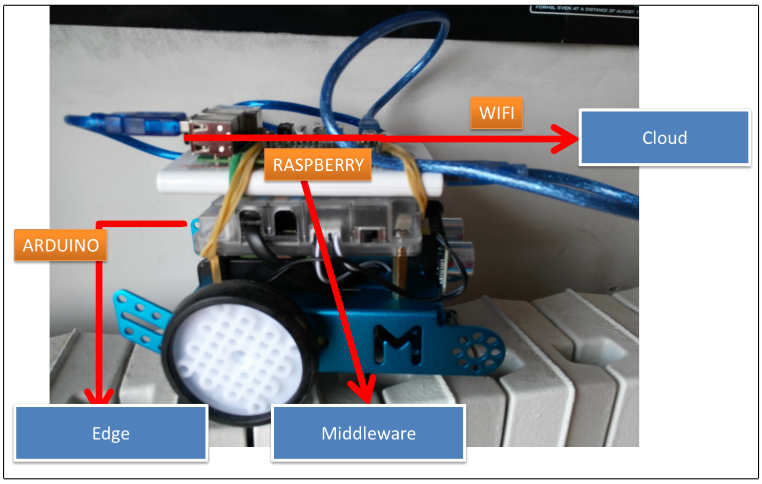
The GOF Bridge pattern has been adopted also to model the 'motors' and the more general concept of 'executor'.



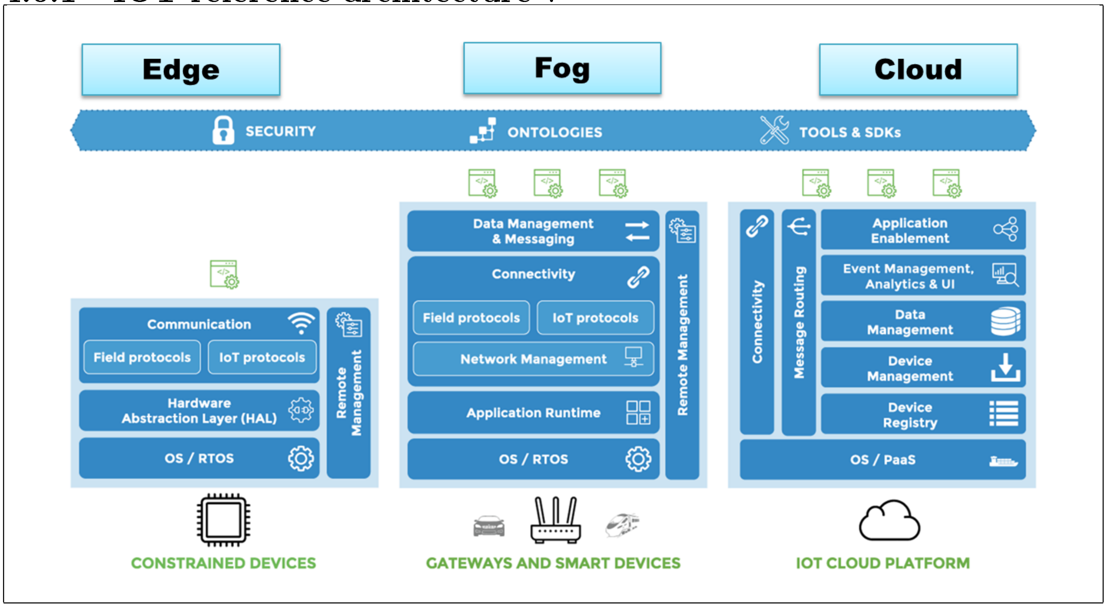
However this part of the `BaseRobot` can be ignored by the application designer and it is no more discussed here.

4 The mbot robot

The `mbot` architecture can be viewed as a simple example of a `IOT` architecture in which the `edge` part is implemented on Arduino, the `middleware` part is implemented on a RaspberryPi and the `cloud` part is implemented on a conventional PC.



4.0.1 IOT reference architecture .



Our real robot is made of a RapberryPi connected via the serial `USB` cable with the Arduino included in the Makeblock Mbot device:

Arduino handles physical devices such as motors and sensors, while RaspberryPi provides support for interaction with a remote node.

The software for the real robot is made of two main parts:

1. a low-level hardware-related part running on Arduino (see Subsection 4.1);
2. a high-level robot-control part running on the RaspberryPi (see Subsection 4.2).

4.1 The low-level code on Arduino

The code running on Arduino has been built as reported in Subsection 4.1.1. It is composed of the following parts:

1. A set of declarations:

```
#include <Arduino.h>
#include <Wire.h>
#include <SoftwareSerial.h>
#include <MeMCore.h>

MeDCMotor motor_9(9);
MeDCMotor motor_10(10);
double angle_rad = PI/180.0;
double angle_deg = 180.0/PI;
void lookAtSonar();
void move(int direction, int speed);
double sonar;
int input;
int count;
MeUltrasonicSensor ultrasonic_3(3);
MeRGBLed rgbled_7(7, 7==7?2:4);
void remoteCmdExecutor();
```

Listing 1.11. `uniboControl.ino`: the main loop

2. A function that works as an interpreter of commands to move the robot:

```
void remoteCmdExecutor()
{
    if((Serial.available()) > (0 )){
        input = Serial.read();
        //Serial.println(input);
        switch( input ){
          case 119 : move(1,150); break; //w
          case 115 : move(2,150); break; //s
          case 97 : move(3,150); break; //a
          case 100 : move(4,150); break; //d
          case 104 : move(1,0); stopFollow = true; break; //h
          case 102 : move(1,0); stopFollow = false; break; //f
          default : move(1,0); stopFollow = true;
        }
    }
}
/*
 * ----------------------------------
 * Moving
 * ----------------------------------
 */
void move(int direction, int speed)
```

```
{
    int leftSpeed = 0;
    int rightSpeed = 0;
    if(direction == 1){ //forward
        leftSpeed = speed;
        rightSpeed = speed;
    }else if(direction == 2){ //backward
        leftSpeed = -speed;
        rightSpeed = -speed;
    }else if(direction == 3){ //left
        leftSpeed = -speed;
        rightSpeed = speed;
    }else if(direction == 4){ //right
        leftSpeed = speed;
        rightSpeed = -speed;
    }
    motor_9.run((9)==M1?-(leftSpeed):(leftSpeed));
    motor_10.run((10)==M1?-(rightSpeed):(rightSpeed));
}
```

Listing 1.12. `uniboControl.ino`: the command interpreter

3. A function that works as an emitter of sonar data (a value of type `double`) that implements also a prefixed obstacle-avoidance policy (retrogress the robot when it is very near to an obstacle):

```
void lookAtSonar()
{
    sonar = ultrasonic_3.distanceCm();
    //emit sonar data but with a reduced frequency
    if( count++ > 50 ){ Serial.println(sonar); count = 0; }
    if((sonar) < (10)){ //very near
        if(((input)==(119))){
            move(1,0);
            rgbled_7.setColor(0,60,0,0);
            rgbled_7.show();
            //Serial.println("OBSTACLE FROM ARDUINO");
            _delay(0.3);
            move(2,100);
            _delay(1);
            move(2,0);
        }
    }
}
```

Listing 1.13. `uniboControl.ino`: sonar data

The `delay` function is a loop to lose time:

```
void _loop(){
}

void _delay(float seconds){
    long endTime = millis() + seconds * 1000;
    while(millis() < endTime)_loop();
}
```

Listing 1.14. `uniboControl.ino`: delay

4. A function that implements a line-follower strategy:

```
double stopFollow = true;
double sonarVal;
void lineFollow();
MeLineFollower linefollower_2(2);

void sonarDetect()
{
    sonarVal = ultrasonic_3.distanceCm();
    Serial.println(sonarVal);
    if((sonarVal) < (10)){
        move(1,0);
        stopFollow = true;
        //Serial.println("stopFollow line follow");
    }
}
void lineFollow()
{
    if( stopFollow == true ) return;
    if(((linefollower_2.readSensors())==(0))){
        move(1,200);
    }
    if(((linefollower_2.readSensors())==(1))){
        motor_9.run((9)==M1?-(0):(0));
        motor_10.run((10)==M1?-(150):(150));
    }
    if(((linefollower_2.readSensors())==(2))){
        motor_9.run((9)==M1?-(150):(150));
        motor_10.run((10)==M1?-(0):(0));
    }
    if(((linefollower_2.readSensors())==(3))){
        move(2,100);
    }
    //sonarDetect();
}
```

Listing 1.15. `uniboControl.ino`: the line-follower

5. The set-up and the main loop:

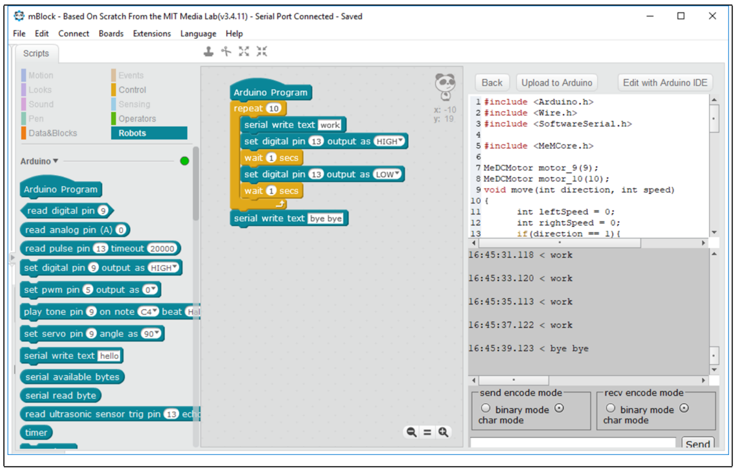
```
void setup(){
    Serial.begin(115200);
    //Serial.println("start");
}

void loop(){
    rgbled_7.setColor(0,0,60,0);
    rgbled_7.show();
    remoteCmdExecutor();
    lookAtSonar();
    lineFollow();
    _loop();
}
```

Listing 1.16. `uniboControl.ino`: the main loop

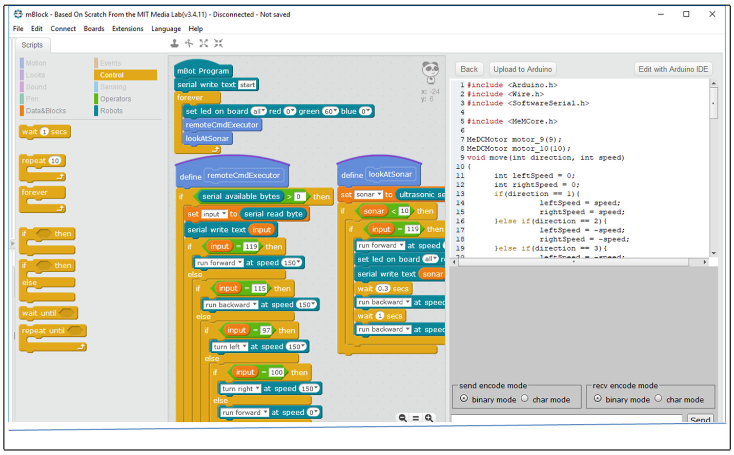
4.1.1 Mblock software .

This code on the mbot can be built with the help of the mblock IDE. mBlock is a graphical programming software which is designed based on `Scratch 2.0` and compatible with `Arduino UNO, mega 2560, leonardo`, etc.

The following picture shows a `mblock` project for the `ArduinoUno` board that blinks the built-in led:



The following picture shows a `mblock` project for the mCore, a control board specially designed for mBot. Based on Arduino Uno, mCore integrates various onboard sensors, such as buzzer, light sensor, RGB LED, etc.



Of course, this is quite limited way to build low-level code for the mbot robot. In order to work directly within a conventional Arduino IDE, we can operate as follows:

– Download the Makeblock-Official library from Github at
`https://github.com/Makeblock-official/Makeblock-Libraries`

- Unzip the file into a temporary folder.
- Change to the temporary folder and zip the "makeblock" directory content.
- In the Arduino IDE, click `Sketch->Include Library->Add` .ZIP library and select the `makeblock.zip` file you've created.
- In the Arduino IDE, click `file/Esempi/MakeBlockDrive/Firmware_For_mBlock/mbot_factory_firmware` to download the firmware
- In the Arduino IDE, open `C:/Didattica/mBot/uniboControl/uniboControl.ino`

The code of Subsection 4.1 has been built in this way.

4.2 mbotConnArduinoObj

The user-defined class `mbotConnArduinoObj` is the 'adapter' between high-level and the low-level worlds. It is written in Java and is based on another software layer (the class `SerialPortConnSupport`) that provides operations to send commands to the low-level executor on Arduino.

4.2.1 Init .

The first part of `mbotConnArduinoObj` provides operations to initialize the serial connection between a PC or a Raspeberry and Arduino:

```java
package it.unibo.mbot;
import it.unibo.mbot.serial.JSSCSerialComm;
import it.unibo.mbot.serial.SerialPortConnSupport;

public class MbotConnArduinoObj {
private SerialPortConnSupport conn = null;
private JSSCSerialComm serialConn;
private double dataSonar = 0;
private String curDataFromArduino;
private ISensorObserverFromArduino observer;

    public  void initRasp( String port ) { //"/dev/ttyUSB0"
        init( port );
    }
    public  void initPc(String port) { //"COM6"
        init( port );
    }
    private  void init(String port) {
        try {
            System.out.println("mbotConnArduinoObj starts");
            serialConn = new JSSCSerialComm(null);
            conn = serialConn.connect(port); //returns a SerialPortConnSupport
            if( conn == null ) return;
            curDataFromArduino = conn.receiveALine();
            System.out.println("mbotConnArduinoObj received:" + dataSonar);
            getDataFromArduino();
        }catch( Exception e) {
            System.out.println("mbotConnArduinoObj ERROR" + e.getMessage());
        }
    }
```

Listing 1.17. `MbotConnArduinoObj.java: init`

4.2.2 Output .

Afterwords, the class defines operations to map high-level commands into low-level commands to be sent to Arduino:

```java
    public void executeTheCommand( char cmd) {
        System.out.println("mbotConnArduinoObj executeTheCommand " + cmd + " conn=" + conn);
        switch(cmd) {
            case 'h' : mbotStop(); break;
```

```
            case 'w' : mbotForward(); break;
            case 's' : mbotBackward(); break;
            case 'a' : mbotLeft(); break;
            case 'd' : mbotRight(); break;
            case 'f' : mbotLinefollow(); break;
        }
    }
    public  void mbotForward() {
        try { if( conn != null ) conn.sendCmd("w"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotBackward() {
        try { if( conn != null ) conn.sendCmd("s"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotLeft() {
        try { if( conn != null ) conn.sendCmd("a"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotRight( ) {
        try { if( conn != null ) conn.sendCmd("d"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotStop() {
        try { if( conn != null ) conn.sendCmd("h"); } catch (Exception e) {e.printStackTrace();}
    }
    public  void mbotLinefollow( ) {
        try { if( conn != null ) conn.sendCmd("f"); } catch (Exception e) {e.printStackTrace();}
    }
```

Listing 1.18. `MbotConnArduinoObj.java: output`

4.2.3 Observer .

Then, the class defines an operation to add an observer to the data sent on the serial line by Arduino.

```
    public void addObserverToSensors( ISensorObserverFromArduino observer ){
        this.observer = observer;
    }
```

Listing 1.19. `MbotConnArduinoObj.java: addObserver`

4.2.4 Input .

Finally, there is an operation that waits for sonar values coming from Arduino (see Subsection 4.1, point 3) and maps these data into an event `realSonar:sonar(DISTANCE)`:

```
    private void getDataFromArduino() {
        new Thread() {
            public void run() {
                try {
                    System.out.println("mbotConnArduinoObj getDataFromArduino STARTED" );
                    while(true) {
                        try {
                            curDataFromArduino = conn.receiveALine();
//                          System.out.println("mbotConnArduinoObj received:" + curDataFromArduino );
                            double v = Double.parseDouble(curDataFromArduino);
                            //handle too fast change
```

```
                            double delta = Math.abs( v - dataSonar);
                            if( delta < 7 && delta > 0.5 ) {
                                dataSonar = v;
                                System.out.println("mbotConnArduinoObj sonar:" + dataSonar);
                                observer.notify(""+dataSonar);
//                              QActorUtils.raiseEvent(curActor, curActor.getName(), "realSonar",
//                                      "sonar( DISTANCE )".replace("DISTANCE", ""+dataSonar ));
                            }
                        } catch (Exception e) {
                            System.out.println("mbotConnArduinoObj ERROR:" + e.getMessage());
                        }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }.start();
    }
```

Listing 1.20. `MbotConnArduinoObj.java: input`

4.3 The adapter basicRobotExecutor

The Application class `basicRobotExecutor` exploits the basic `MbotConnArduinoObj` to implement the logical operations required at model level:

The name of the class starts with a lower-case letter for a constraint imposed by the current implementation of the *QActor* software factory. Moreover, each operation provided by the `Java` class must have as its first argument a variable of type `QActor`.

```
package it.unibo.robotMBot;
import it.unibo.mbot.MbotConnArduinoObj;
import it.unibo.qactors.akka.QActor;

public class basicRobotExecutor {
private static MbotConnArduinoObj robotSupport ; //singleton

    public static void setUp(QActor qa, String port) {
        if( robotSupport == null ) {
            robotSupport = new MbotConnArduinoObj();
            robotSupport.initPc(port);
            robotSupport.addObserverToSensors( new SensorObserverFromArduino(qa) );
        }
    }
    public static void doMove( QActor qa, String cmd ) { //Args MUST be String
        robotSupport.executeTheCommand(cmd.charAt(0));
    }
 }
```

Listing 1.21. `basicRobotExecutor.java`

5 A robot-command executor

In this section we introduce a system that implements the following set of features:

1. The system handles commands sent to a robot by an human user via a `Web-GUI` interface.
2. The system can be configured to work with:
   - the virtual robot running in the `wenv` of Subsection 2.2.1;
   - the physical robot of Section 3;
   - the physical robot of Section 4;
3. The system is able to react to events emitted by a `sonar` (both virtual and real).

   The model shows that:

- The configuration is expressed as a fact (written in the `actorKb` of the robot) of the form:

```
robotType( ROBOTYPE, setuparg(ARGS) ).
```

- The robot modelled by the actor `robotdemo` initially performs some move, to show that it is working.
- The `robotdemo` waits for an message command sent by another actor (`cmdrobotconverter`) that maps a user command sent from the built-in *QActor* `Web-GUI` into a *dispatch* `robotCmd : cmd(X)`.
- The `robotdemo` assumes that the implementation of a move command (see state `doJob` and Subsection 5.1) is performed in *asynchronous* way.

```
System demoRobot

Dispatch robotCmd : cmd(X)              //X=w|a|s|d|h
Event sonarEvent : distance(D, DIR, POS)  //from SensorObserver

Event usercmd    : usercmd(X)   //from robot GUI; X=robotgui(CMD) CMD=s(low)
Event inputcmd   : usercmd(X)   //from input GUI; X=executeInput( do(G,M) )
Event alarm      : alarm(X)     //from red button
Event obstacle   : obstacle(X)  //from red button and from SensorObserver
Event sonarDetect : sonarDetect(X)             //From (virtual robot) sonar
Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar

//Context ctxConsole ip [ host="localhost" port=8078 ] -g cyan -httpserver
Context ctxRobot   ip [ host="localhost" port=8098 ] -httpserver

QActor cmdrobotconverter context ctxRobot{
   State init normal [ ]
   transition stopAfter 600000
   whenEvent usercmd -> handleUserCmd
   finally repeatPlan

   State handleUserCmd resumeLastPlan[//MAPPING AFTER EXPERIMENTATION
```

```
        onEvent usercmd : usercmd( robotgui(h(X)) ) ->
                forward robotdemo -m robotCmd : cmd("h");
        onEvent usercmd : usercmd( robotgui(w(X)) ) ->
                forward robotdemo -m robotCmd : cmd("w");
        onEvent usercmd : usercmd( robotgui(a(X)) ) ->
                forward robotdemo -m robotCmd : cmd("a");
        onEvent usercmd : usercmd( robotgui(d(X)) ) ->
                forward robotdemo -m robotCmd : cmd("d");
        onEvent usercmd : usercmd( robotgui(s(X)) ) ->
                forward robotdemo -m robotCmd : cmd("s")
    ]
}

QActor robotdemo context ctxRobot {
Rules{
    //robotType( "robotVirtual", setuparg("localhost") ).
    robotType( "robotRealMbot", setuparg("COM6") ).
    //robotType( "robotRealRaspOnly", setuparg("") ).
}
    State init normal [
        println("robotdemo STARTS");
        [ !? robotType(T,setuparg(A))]
                javaRun it.unibo.utils.allRobots.setUp(T,A)
    ]
    switchTo someMove

    State someMove[
        delay 1000;
        javaRun it.unibo.utils.allRobots.doMove( "w");
        delay 1000;
        javaRun it.unibo.utils.allRobots.doMove( "s");
        delay 1000;
        javaRun it.unibo.utils.allRobots.doMove( "h")
    ]
    switchTo doJob

    State doJob [
         println("waiting for an user command ...")
    ]
    transition stopAfter 600000
        whenMsg robotCmd:cmd(M) do javaRun it.unibo.utils.allRobots.doMove(M),
        whenEvent sonarEvent -> handleSonarEvent,
        whenEvent sonar    -> handleSonarEvent
        finally repeatPlan

    State handleSonarEvent resumeLastPlan [
        printCurrentEvent
    ]

 }
```

Listing 1.22. `demoRobot.qa`

5.1 The adapter

In order to achieve its goals, the system-model exploits an utility class written by the Application designer:

```java
package it.unibo.utils;

import alice.tuprolog.SolveInfo;
import it.unibo.qactors.akka.QActor;

public class allRobots {

    public static void setUp( QActor qa, String robotType, String args ) {
        try {
            qa.println("allRobots " + robotType + " setUp args="+ args);
            switch( robotType ){
                case "robotRealMbot"    : it.unibo.robotMBot.basicRobotExecutor.setUp(qa,args);break;
                case "robotRealRaspOnly" : it.unibo.robotOnRaspOnly.basicRobotExecutor.setUp(qa);break;
                case "robotVirtual"     : it.unibo.robotVirtual.basicRobotExecutor.setUp(qa,args);break;
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
    }

    public static void doMove( QActor qa, String cmd ) { //Args MUST be String
        try {
            SolveInfo sol = qa.solveGoal("robotType( T, _ )");
            String robotType = sol.getVarValue("T").toString();
            qa.println("allRobots " + robotType + " doMove cmd="+ cmd);
            switch( robotType ){
                case "robotRealMbot"   : it.unibo.robotMBot.basicRobotExecutor.doMove(qa, cmd);break;
                case "robotRealRaspOnly" : it.unibo.robotOnRaspOnly.basicRobotExecutor.doMove(qa,
                     cmd);break;
                case "robotVirtual"    : it.unibo.robotVirtual.basicRobotExecutor.doMove(qa,cmd);break;
                default: qa.println("Sorry, robot type unknown");
            }
        }catch(Exception e) {
            e.printStackTrace();
        }

    }
}
```

Listing 1.23. `allRobots.java`

5.2 Activating the `wenv`

The assumption in the model is that the virtual environment `wenv` has been already activated. However, the `wenv` can be activated by the model itself. For example:

```
QActor startscene context ctxRobot {
    Plan init normal[
        println("startscene starts...");
        nodeOp "C:/Didattica2018Work/iss2018Lab/it.unibo.mbot2018/Soffritti/server/src/main.js 8999" -o
    ]
}
```