CaseStudy1
Showing sonar distances on a radar

Antonio Natali

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
antonio.natali@unibo.it

Table of Contents

1 Starting

Open an empty directory (e.g. `C:/iss2018Lab`) and

1. Clone the `iss2018Lab` repository by executing the following command:

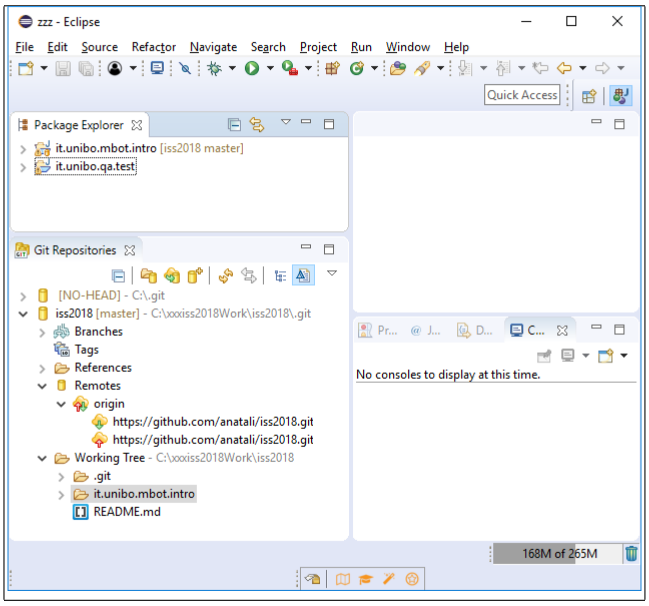
   ```
   git clone https://github.com/anatali/iss2018Lab.git
   ```

   To update the repository, the command is `git pull`.

2. Now open the Eclipse working space into your working directory (e.g. `C:/iss2018Work`) and do:

   ```
   Window -> ShowView -> Other -> Git -> Git Repositories
   Add an existing local Git repository (C:/iss2018Lab)
   ```

3. Import `(by copying) into the current working space`[1] the project `it.unibo.mbot.intro`. The result is:

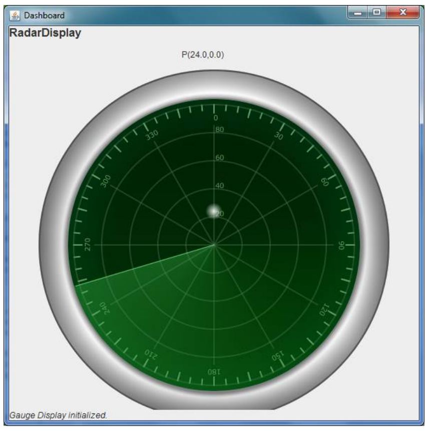

zzz - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
it.unibo.mbot.intro [iss2018 master]
it.unibo.qa.test
Git Repositories
[NO-HEAD] - C:\.git
iss2018 [master] - C:\xxxiss2018Work\iss2018\.git
Branches
Tags
References
Remotes
origin
https://github.com/anatali/iss2018.git
https://github.com/anatali/iss2018.git
Working Tree - C:\xxxiss2018Work\iss2018
.git
it.unibo.mbot.intro
README.md
Pr... @ J... D... C...
168M of 265M

No consoles to display at this time.

4. In the project *it.unibo.mbot.intro*, the file `runnable/it.unibo.ctxRadarBase.MainCtxRadarBase-1.0.zip` includes the implementation of a software system able to display distance values on an output device that simulates the screen of a radar. To execute the application, *unzip* the file (into some other directory) and execute:

   ```
   java -jar it.unibo.qactor.radar-1.0.jar
   ```

   The virtual display shown by the radar system is:



[1] To avoid any conflict in project updating.

1.1 Interacting with the radar

In order to use the radar system, we must send messages to it, by using a TCP client connection on the port 8033. The messages must be *Strings* with the following structure:

```
msg(polarMsg,dispatch,SENDER,radarguibase,POLAR,MSGNUM)
```

where

- SENDER is the name (in lowercase) of the sender ;
- POLAR is a value of the form p(D,ANGLE), with 0<=D<=80, 0<=ANGLE<=180;
- MSGNUM is a natural number

Let us implement a TCP client by using the net module of Node.js, that provides an asynchronous network wrapper. We star with the code that establishes a connection with the radar:

```
var net = require('net');
var host = "localhost";
var port = 8033;

console.log('connecting to ' + host + ":" + port);
var conn = net.connect({ port: port, host: host });
conn.setEncoding('utf8');

// when receive data back, print to console
conn.on('data',function(data) {
    console.log(data);
});
// when server closed
conn.on('close',function() {
    console.log('connection is closed');
});
conn.on('end',function() {
    console.log('connection is ended');
});
```

Listing 1.1. TcpClientToRadar.js: set up a connection

Now, let us define some utility functions to send messages:

```
function sendMsg( msg ){
    try{
        console.log("SENDING " + msg );
        conn.write(msg+"\n"); //Asynchronous!!!
    }catch(e){
        console.log("ERROR " + e );
    }
}

function sendMsgAfterTime( msg, delay ){
    setTimeout( function(){ sendMsg( msg ); }, delay);
}
```

Listing 1.2. TcpClientToRadar.js: utility

The send function writes the given data on the connection in asynchronous way; thus, it immediately returns control to the caller. The sendMsgAfterTime function allows us to delay the call after a given delay.

Finally, we send some data to the radar:

```
var msgNum=1;

sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,30)," + msgNum++ +")", 1000);
sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,90)," + msgNum++ +")", 2000);
sendMsgAfterTime("msg(polarMsg,dispatch,jsSource,radarguibase, p(50,150)," + msgNum++ +")", 3000);

setTimeout(function(){ conn.end(); }, 4000);
```

Listing 1.3. TcpClientToRadar.js: send data to radar

The radar shows the points, while the output of our client is:

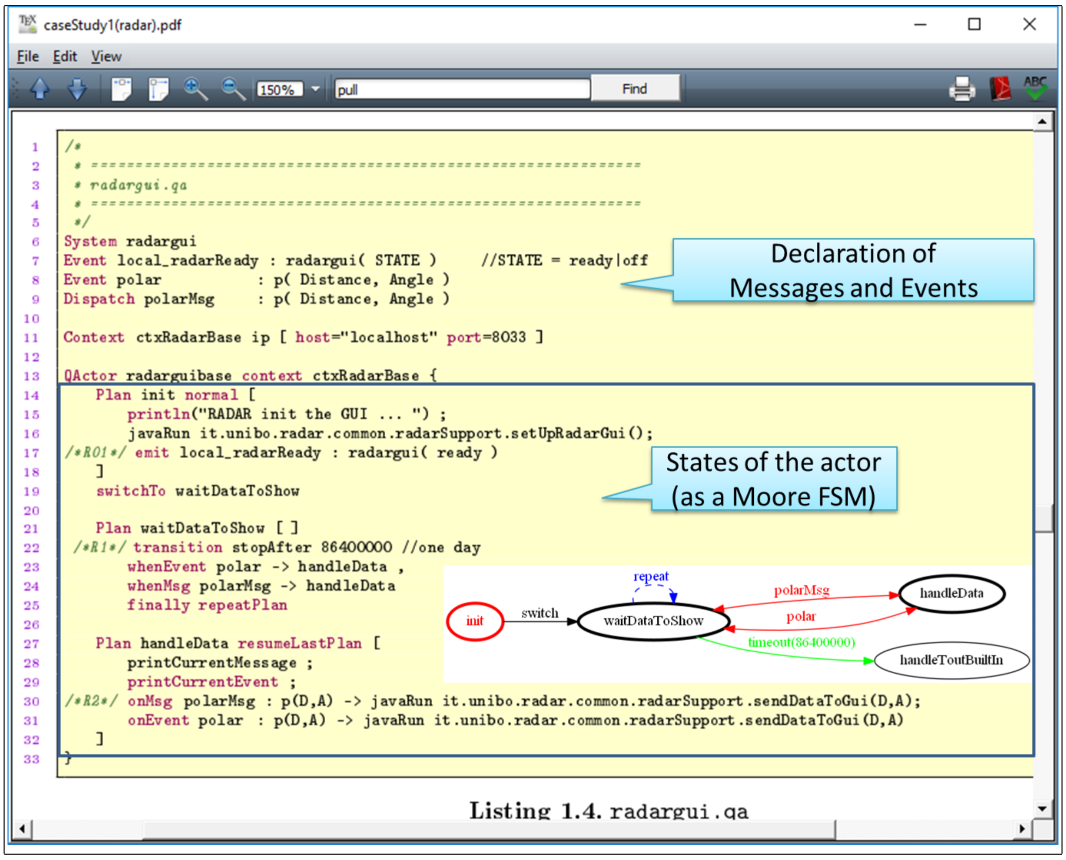
```
connecting to localhost:8033
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,30),1)
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,90),2)
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,150),3)
connection is ended
connection is closed
```

Instead of using Node, we could write a client for the radar in Java. This task is left to the reader.

1.2 Using the radar: a model-based approach

In the previous version of the radar client, we did not have any knowledge on the internal structure of the radar system. We exploited only the knowledge on the low-level structure of messages handled by the radar.

But, fortunately, there exist a high level description of the radar system, expressed in the high-level, custom modelling language *QActor*. This description is a (executable) model defined as follows:



```
/*
 * =================================================================
 * radargui.qa
 * =================================================================
 */
System radargui
Event local_radarReady : radargui( STATE )     //STATE = ready|off
Event polar             : p( Distance, Angle )
Dispatch polarMsg       : p( Distance, Angle )

Context ctxRadarBase ip [ host="localhost" port=8033 ]

QActor radarguibase context ctxRadarBase {
    Plan init normal [
        println("RADAR init the GUI ... ") ;
        javaRun it.unibo.radar.common.radarSupport.setUpRadarGui();
/*R01*/ emit local_radarReady : radargui( ready )
    ]
    switchTo waitDataToShow

    Plan waitDataToShow [ ]
 /*R1*/ transition stopAfter 86400000 //one day
        whenEvent polar -> handleData ,
        whenMsg polarMsg -> handleData
        finally repeatPlan

    Plan handleData resumeLastPlan [
        printCurrentMessage ;
        printCurrentEvent ;
/*R2*/ onMsg polarMsg : p(D,A) -> javaRun it.unibo.radar.common.radarSupport.sendDataToGui(D,A);
        onEvent polar  : p(D,A) -> javaRun it.unibo.radar.common.radarSupport.sendDataToGui(D,A)
    ]
}
```

Listing 1.4. radargui.qa

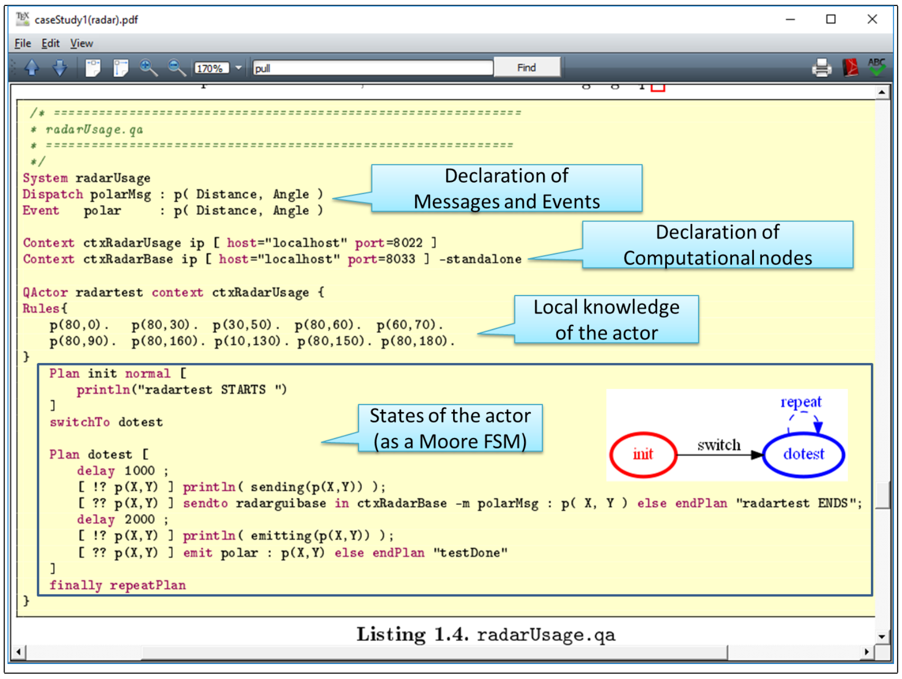
The model describes the structure, the interaction and the behaviour of the radar. More specifically, it shows that the radar is able to handle messages and events *explicitly declared* at the very beginning of the model[2]:

```
Dispatch polarMsg : p( Distance, Angle )
Event    polar    : p( Distance, Angle )
```

Since the message polarMsg is declared as a Dispatch, the interaction is of type 'fire-and-forget'.

[2] The event local_radarReady is a local event used internally.

Thus, another way to introduce a client of the radar system is to define the client by using the same modelling language used for the radar. Let us introduce an example of such a model[3]:


caseStudy1(radar).pdf
File Edit View
170%
pull
Find
Declaration of Messages and Events
Declaration of Computational nodes
Local knowledge of the actor
States of the actor (as a Moore FSM)
repeat
init
switch
dotest

```
/* ==============================================================
 * radarUsage.qa
 * ==============================================================
 */
System radarUsage
Dispatch polarMsg : p( Distance, Angle )
Event    polar    : p( Distance, Angle )

Context ctxRadarUsage ip [ host="localhost" port=8022 ]
Context ctxRadarBase ip [ host="localhost" port=8033 ] -standalone

QActor radartest context ctxRadarUsage {
Rules{
   p(80,0).   p(80,30).  p(30,50).  p(80,60).  p(60,70).
   p(80,90).  p(80,160). p(10,130). p(80,150). p(80,180).
}
   Plan init normal [
      println("radartest STARTS ")
   ]
   switchTo dotest

   Plan dotest [
      delay 1000 ;
      [ !? p(X,Y) ] println( sending(p(X,Y)) );
      [ ?? p(X,Y) ] sendto radarguibase in ctxRadarBase -m polarMsg : p( X, Y ) else endPlan "radartest ENDS";
      delay 2000 ;
      [ !? p(X,Y) ] println( emitting(p(X,Y)) );
      [ ?? p(X,Y) ] emit polar : p(X,Y) else endPlan "testDone"
   ]
   finally repeatPlan
}
```

Listing 1.4. radarUsage.qa

The model states that:

1. Our radarUsage system is a distributed system composed of two computational nodes (Contexts).
2. The node named ctxRadarBase is external to the systems (flag -standalone): it is the node that executes the given radar system. The context ctxRadarUsage represents the node in which we will run our radar client.
3. Our radar client is modelled as a *QActor* (radartest): it works as a finite state machine that (in the state dotest) sends messages and emits events. We will expand this point in in Subsection 1.2.1.
4. The messages and events involved in our system are *the same* defined in the radar model:

```
Dispatch polarMsg : p( Distance, Angle )
Event    polar    : p( Distance, Angle )
```

5. The data sent by our client are defined in the actor's knowledge base as a sequence of facts (in Prolog syntax) and are 'consumed' in the state dotest, by using guards.

From the model above, the *QActor* software factory generates an executable version written in Java. The main program is in the file:

```
it.unibo.mbot.intro/src-gen/it/unibo/ctxRadarUsage/MainCtxRadarUsage.java
```

If we run this file, the radar will show 10 points.

1.2.1 Sending messages and emitting events .

The concept of message in the *QActor* world implies that we must known the name of the message destination, that must be another *QActor*. This fact is reflected in the sentence:

```
sendto radarguibase in ctxRadarBase -m polarMsg : p( X, Y )
```

Note that the knowledge of the name of the receiving radar actor (*radarguibase*) is not required for events:

```
emit polar : p(X,Y)
```

[3] The code is in the file it.unibo.mbot.intro/src/radarUsage.qa.

2 The problem to solve

The problem now is the following:
with reference to a `mbot` physical robot working in virtual environment, build an application that sends to the radar the data sensed by the virtual and the real sonars. More specifically:

- the data of the *virtual sonar* `sonar1` must be displayed on the direction of angle=30;
- the data of the *virtual sonar* `sonar2` must be displayed on the direction of angle=120;
- the data of the *virtual sonar* on the virtual robot must be displayed on the direction of angle=90 at the fixed distance of 40;
- the data of the *real sonar* on the physical robot must be displayed on the direction of angle=0;

2.1 Requirements analysis

We ask the customer for the following basic information:

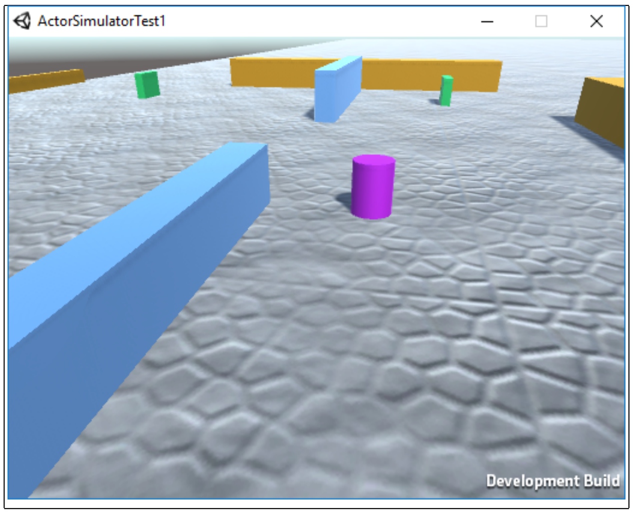
1. what is a `mbot`?
2. what is the virtual environment?
3. in which way we can obtain data from a virtual or from a real sonar?

2.1.1 The virtual environment .

As regards the virtual environment, the answer is in the project `it.unibo.issMaterial`, available by cloning the `iss2018` GIT repository:

```
git clone https://github.com/anatali/iss2018.git
```

The virtual environment is an application written in `Unity`, included in the file: `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. Details about this point are given in `IntroductionQa2017.pdf` (section 12)[4].
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

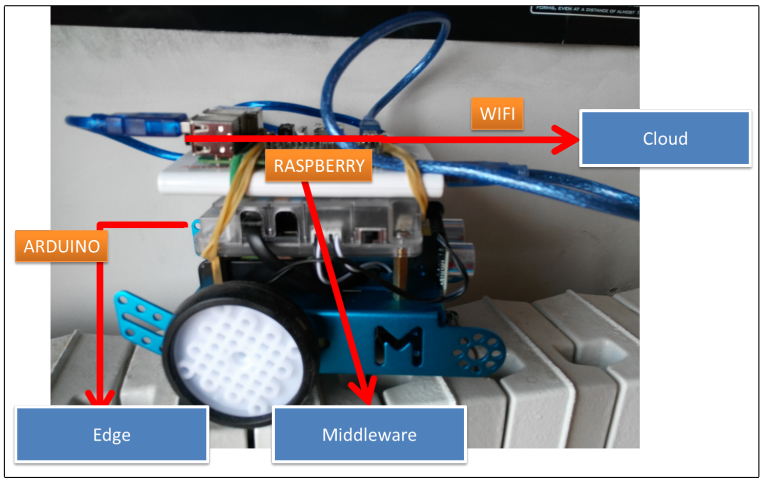
[4] The file is available in `it.unibo.issMaterial/issdocs/Material`.

sonar2. Moreover, the virtual rover is equipped (in its front) with a sonar, that emits the event sonarDetect : sonarDetect(TARGET) when detects an obstacle.

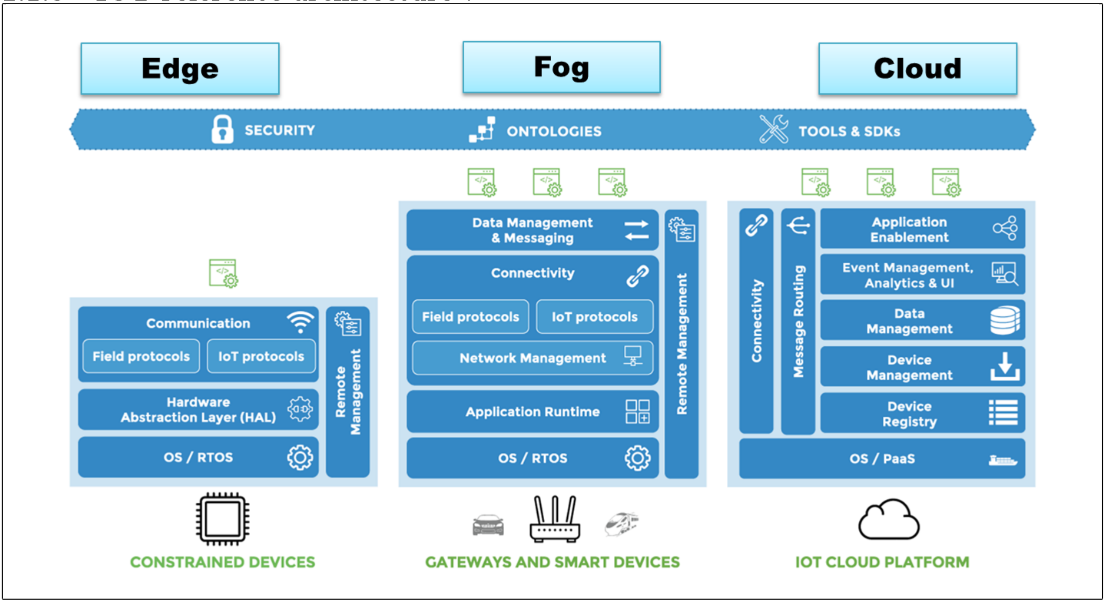
As regards the other questions, our goal is to build a model for the mbot and a model for the real sonar. The goal of these models is to clarify how we can exchange information with the corresponding entities of our application domain. The internal structure and the internal behaviour of the entities have relatively less importance at this stage.

2.1.2 The mbot robot .

The mbot architecture is an 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.



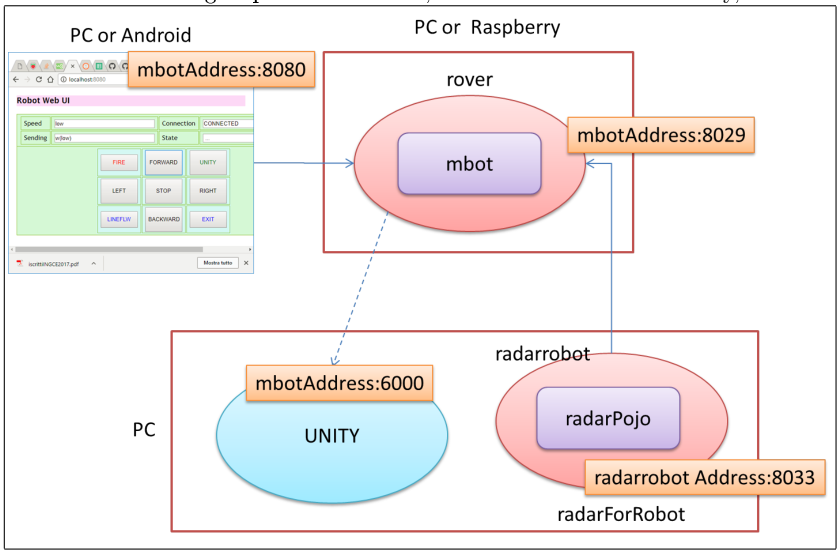
2.1.3 IOT reference architecture .



Arduino handles physical devices such as motors and sensors, while RaspberryPi provides support for interaction with a remote node. More precisely, as regards the interactions:

– the robot communicates with external world via a WIFI local network. Let us denote as mbotAddress the IP of the robot in such a network;
– the robot accepts move commands sent via a browser connected to the address mbotAddress:8080;
– through the same web interface, the robot accepts a command to connect itself to a modified virtual Unity environment working on a remote PC in the local network;
– the robot emits *QActor* events (sonar and sonarDetect). This means that we can build a new *QActor* system including the robot and a *QActor* element able to perceive and handle these events.

The following picture shows, in a informal way, the architecture of our system:



This informal diagram shows that:

1. Our software system is composed of two active entities (rover and radarrobot) and the virtual environment provided by the *QActor*-compatible Unity.
2. Each actor works on its proper computational nod,e that provides an TCP server on a specific port.
3. The actor named radarrobot runs on the PC and implements the radar by interacting with the actor named rover. Internally, the radarrobot (re)uses a POJO (radarPojo) that implements the radar GUI.
4. The actor named rover runs on the RaspberryPi[5] and gives to the mbot the capability to interact with the external world. Internally, the rover reuses a POJO (mbot) that provides a set of operations to move the physical robot and to acquire data from its sensors.
5. The rover exposes a Web interface to allow human users to send commands to it by using a browser.

2.1.4 A formal architectural model .

Our system architecture can be described in a formal way by a set of *QActor* models. Let us start with the radarrobot actor working on a PC:

```
/*
 * ================================================================
 * radarForRobot.qa
 * ================================================================
 */
System radargui
Event local_radarReady : radargui( STATE )     //STATE = ready|off
Event polar            : p( Distance, Angle )

Context ctxRadarForRobot ip [ host="localhost" port=8033 ]
Context ctxMbotControl  ip [ host="localhost" port=8029 ] -standalone
```

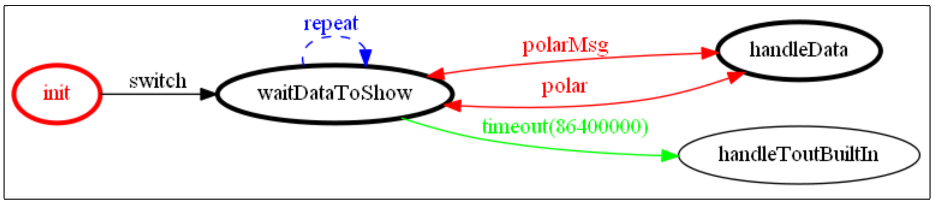
[5] The rover could run also on a conventional PC with Arduino connected to the PC.

```

QActor radarrobot context ctxRadarForRobot {
    Plan init normal [
        javaRun it.unibo.radar.common.radarSupport.setUpRadarGui()
    ]
    switchTo waitDataToShow

    Plan waitDataToShow [ ]
     transition stopAfter 86400000 //one day
        whenEvent polar -> handleData
        finally repeatPlan

    Plan handleData resumeLastPlan [
        printCurrentMessage ;
        printCurrentEvent ;
        onEvent polar  : p(D,A) -> javaRun it.unibo.radar.common.radarSupport.sendDataToGui(D,A)
    ]
}
```

Listing 1.4. `radarForRobot.qa`

This model describes without ambiguity several details not very clear when looking at the informal picture:

- Our system is composed of two subsystems: a subsystem that implements the radar (`ctxRadarForRobot`) and a subsystem that implements the robot (`ctxMbotControl`). The flag `-standalone` means that the robot subsystem is external and can run independently of the radar.
- The behaviour of the actor is a finite state machine that 'reacts' to messages (`polarMsg`) and events (`polar`):



- The `radarPojo` is implemented as a `Java` class named `it.unibo.radar.common.radarSupport`. Note that 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`. For example:

```
    public static void setUpRadarGui( QActor qa ) {
        try {
            radarControl = new RadarControl( qa.getOutputEnvView() );
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

The `radarPojo` is deployed in the `jar` named `radarPojo.jar` included in the folder `it.unibo.robot.intro/libs/unibo`.

2.1.5 The robot (on RaspberryPi .

The subsystem that implements the robot can be formally described as follows:

```
/*
 * ================================================================
 * mbotControl.qa
 * ================================================================
 */
System mbotControl
Event usercmd    : usercmd(CMD)
Event sonar      : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar
Event sonarDetect : sonarDetect(X)             //From (virtual robot) sonar
Event realSonar : sonar( DISTANCE )            //From real sonar on real robot
Event   polar   : p( Distance, Angle )
Event unityAddr : unityAddr( ADDR )            //From user interface

Context ctxMbotControl ip [ host="localhost" port=8029 ] -httpserver
```

Listing 1.5. `mbotControl.qa: starting the model`

This starting part of the model says that the robot provides a `TCP` server on port `8029` and a Web interface (flag `-httpserver`). This (sub)system does not known any other component; it can be executed in a '`standalone`' way.

2.1.6 The rover as a command interpreter .

The next part of the model defines the actors working in the `ctxMbotControl` context. The first one is the actor that extends a basic `mbot` with the possibility to receive remote commands from the human user:

```
QActor rover context ctxMbotControl {
    Plan init normal [
            println("rover START")
        ]
        switchTo waitUserCmd

    Plan waitUserCmd[  ]
    transition stopAfter 600000
        whenEvent usercmd -> execMove
    finally repeatPlan
```

Listing 1.6. `mbotControl.qa: waiting for user commands`

The behavior of the rover actor can be expressed as a set of states in which we send commands to Arduino according to the input command given by the human user (perceived as the event `usercmd`):

```
    Plan execMove resumeLastPlan[
        printCurrentEvent;
        onEvent usercmd : usercmd( robotgui(w(X)) ) -> switchTo moveForward;
        onEvent usercmd : usercmd( robotgui(s(X)) ) -> switchTo moveBackward;
        onEvent usercmd : usercmd( robotgui(a(X)) ) -> switchTo turnLeft;
        onEvent usercmd : usercmd( robotgui(d(X)) ) -> switchTo turnRight ;
        onEvent usercmd : usercmd( robotgui(h(X)) ) -> switchTo stopTheRobot ;
        onEvent usercmd : usercmd( robotgui(f(X)) ) -> javaRun it.unibo.rover.mbotConnArduino.mbotLinefollow() ;
        onEvent usercmd : usercmd( robotgui(unityAddr(X)) ) -> switchTo connectToUnity;
        onEvent usercmd : usercmd( robotgui(x(X)) ) -> switchTo terminataAppl
    ]
```

Listing 1.7. `mbotControl.qa`: command intepreter

2.1.7 The connection to the Unity virtual environment .

The built-in operation `createUnityObject` creates a virtual robot object game in the Unity virtual environment. Thus, the `connectToUnity` state can be defined as follows:

```
    Plan connectToUnity resumeLastPlan[
        onEvent usercmd : usercmd( robotgui(unityAddr(ADDR)) ) -> connectUnity ADDR ;
        addRule unityOn ;
        createUnityObject "rover" ofclass "Prefabs/CustomActor" ;
        backwards 70 time ( 800 ) ;
        right 70 time ( 1000 ) //position
    ]
```

Listing 1.8. `mbotControl.qa: connect to Unity`

2.1.8 Activating the motors for a prefixed time: reactive actions .

The basic `mbot` is implemented by the Java class `it.unibo.rover.mbotConnArduino`. Note that 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`. For example:

```
    public static void mbotForward(QActor actor) {
        try { if( conn != null ) conn.sendCmd("w"); } catch (Exception e) {e.printStackTrace();}
    }
```

Thus, the states in which we execute simple moves of the robot can be defined as follows

```
    Plan turnLeft resumeLastPlan [
        javaRun it.unibo.rover.mbotConnArduino.mbotLeft();
        [ !? unityOn ] left 40 time(750) else delay 900;
        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
    ]
    Plan turnRight resumeLastPlan [
        javaRun it.unibo.rover.mbotConnArduino.mbotRight();
        [ !? unityOn ] right 40 time(750) else delay 900;
        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
    ]
    Plan stopTheRobot resumeLastPlan[
        stop 40 time ( 10 );
        javaRun it.unibo.rover.mbotConnArduino.mbotStop()
    ]
```

Listing 1.9. `mbotControl.qa: simple moves`

2.1.9 Activating the motors for a long time .

The user command to move the robot forward or backward cannot be handled is such a simple way. In fact we have to move 'forever' (or for a long time) unless some event is raised from the external world.

In the next states we exploit the *QActor* feature of *reactive actions* (see `IntroductionQa2017.pdf` (section 14)) to perform (long lived) actions while being able to 'react' to input data such as the sonar events or other user commands:

```
    Plan moveForward resumeLastPlan[
        javaRun it.unibo.rover.mbotConnArduino.mbotForward()
    ]
    reactive onward 40 time( 15000 )
        whenEnd             -> endOfMove
        whenTout 30000      -> handleTout
        whenEvent sonarDetect -> handleRobotSonarDetect
        or whenEvent sonar  -> handleSonar
        or whenEvent usercmd -> execMove

    Plan moveBackward resumeLastPlan[
        javaRun it.unibo.rover.mbotConnArduino.mbotBackward()
    ]
    reactive backwards 40 time ( 15000 )
        whenEnd             -> endOfMove
        whenTout 30000      -> handleTout
//      whenEvent sonarDetect -> handleObstacle //no sensor on robot back
        whenEvent sonar     -> handleSonar
        or whenEvent usercmd -> execMove
```

Listing 1.10. `mbotControl.qa`

2.1.10 The emitter .

Another component that must run on the RaspberryPi is an actor able to mapssonar events into events that can be understood by the radar:

```
QActor sonardetector context ctxMbotControl{
    Plan init normal [ ]
    switchTo waitForEvents

    Plan waitForEvents[ ]
    transition stopAfter 600000
        whenEvent sonar     ->  sendToRadar,
        whenEvent sonarDetect -> sendToRadar,
        whenEvent realSonar -> sendToRadar
    finally repeatPlan

    Plan sendToRadar resumeLastPlan [
        printCurrentEvent;
        onEvent realSonar : sonar( DISTANCE )             -> emit polar : p(DISTANCE, 0) ;
        onEvent sonar     : sonar(sonar1, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,30) ;
        onEvent sonar     : sonar(sonar2, TARGET, DISTANCE ) -> emit polar : p(DISTANCE,120) ;
        onEvent sonarDetect : sonarDetect(TARGET)         -> switchTo showObstcle
    ]

    Plan showObstcle resumeLastPlan[
        println( "found obstacle" );
        emit polar : p(30,90)
//      sendto radarguibase in ctxRadarBase -m polarMsg : p( 30, 90 )
    ]
}
```

Listing 1.11. `mbotControl.qa`

Note that the choice to emit events (and not to send messages) avoid the need to known the name of any destination actor.

2.1.11 The real sonar .

The `sonardetector` actor handles events emitted both from the virtual world (`sonar`, `sonarDetect`) and from the real world (`realSonar`). The reader should imagine the code to be put on Arduino to 'create' events at the `sonardetector` level.