Radar System
(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

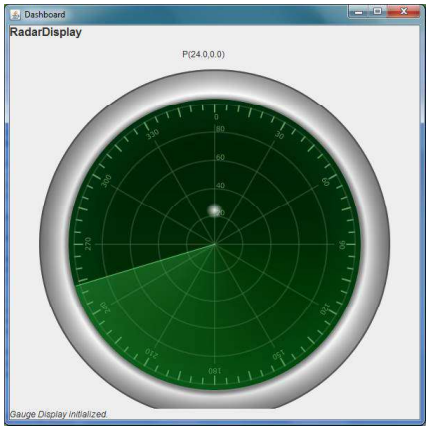
Table of Contents

1 Introduction

In the project *it.unibo.mbot2018*, 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:



1.1 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 start 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:

```
var sendMsg = function( 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:

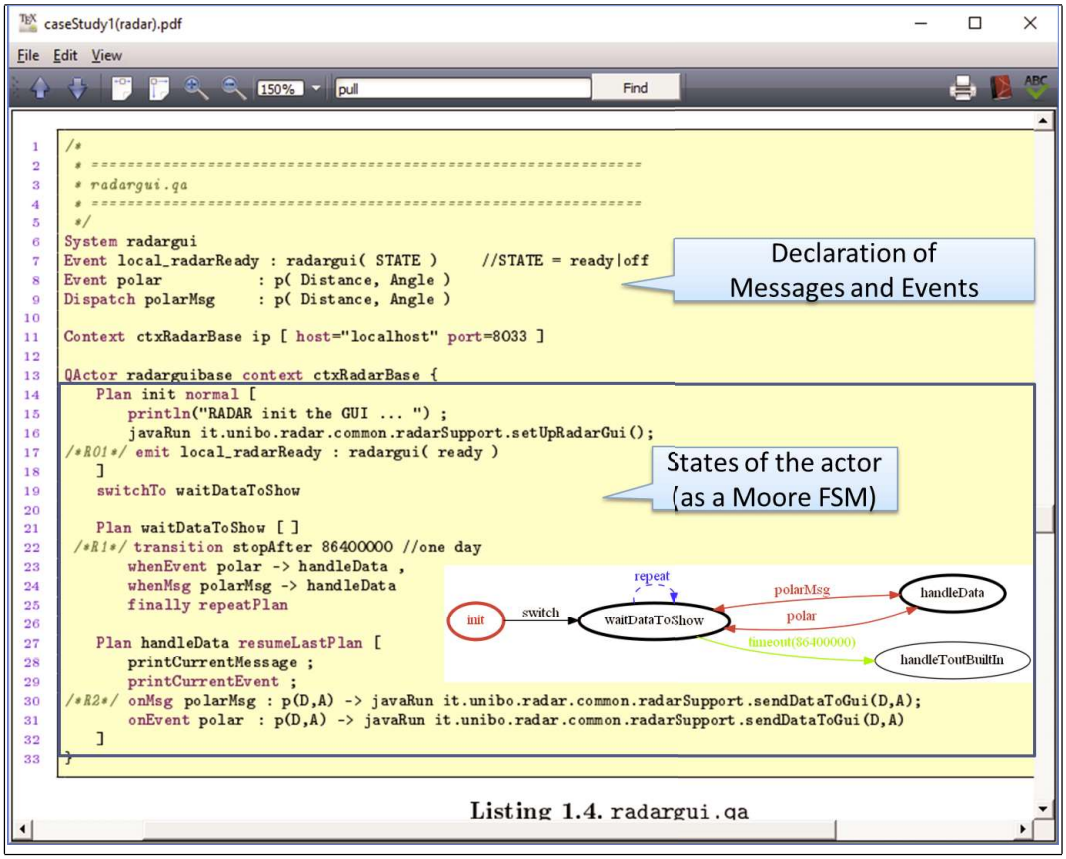
```
connecting to localhost:8033
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,30),1)
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,90),2)
SENDING msg(polarMsg,dispatch,jsSource,radarguibase, p(50,150),3)
connection is ended
connection is closed
```

Instead of using NodeJs, 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 (see the project `it.unibo.mbot.intro`):



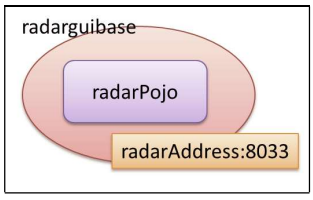

```
/*
 * ============================================================
 * radargui.qa
 * ============================================================
 */
System radargui
Event local_radarReady : radargui( STATE )     //STATE = ready|off
Event polar            : p( Distance, Angle )
Dispatch polarMsg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

1.2.1 Reusing conventional objects .

Internally, the `radargui` (re)uses a POJO (`radarPojo`) that implements the radar GUI.



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:

```
1   public static void setUpRadarGui( QActor qa ) {
2       try {
3           radarControl = new RadarControl( qa.getOutputEnvView() );
4       } catch (Exception e) {
5           e.printStackTrace();
6       }
7   }
```

1.2.2 The radar as information handler .

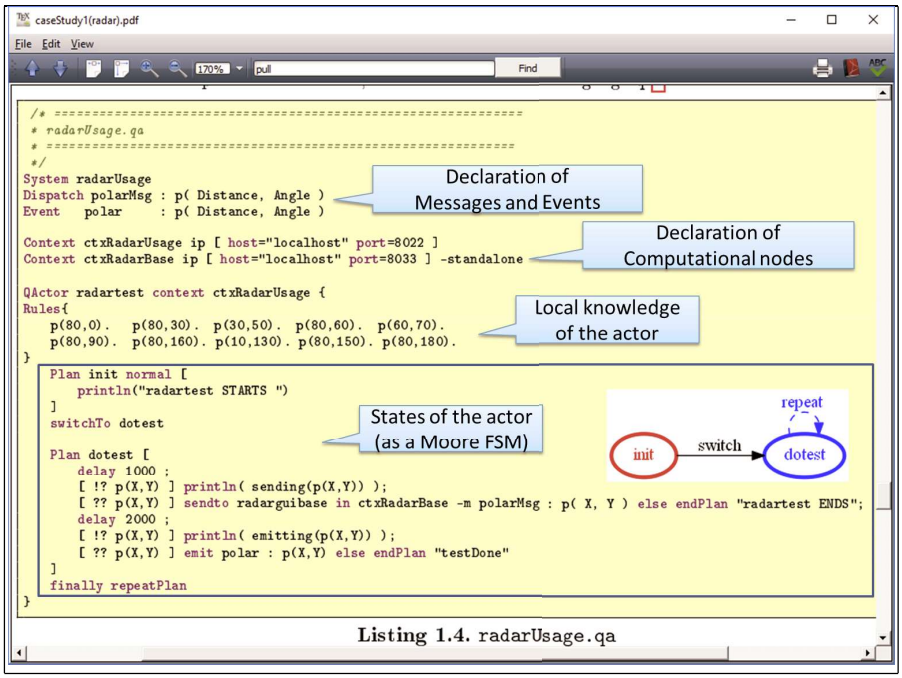
The radar is able to handle messages and events *explicitly declared* at the very beginning of the model[1]:

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

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

1.2.3 A radar client .

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[2]:




```
/* ============================================================
 * 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

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

[2] The code is in the file it.unibo.mbot.intro/src/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 1.2.4.
4. The messages and events involved in our system are *the same* defined in the radar model:

```
1 Dispatch polarMsg : p( Distance, Angle )
2 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:

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

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

1.2.4 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:

```
1 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:

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