Software Systems Engineering
Case Study 2016

Paolo Sarti and Marcello Ballanti

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
paolo.sarti2studio.unibo.it, marcello.ballantistudio.unibo.it

1 Introduction

The following report describes the software development process for an IoT application. The whole process is divided in two steps: at first, the client will communicate some requirements for the application, then new features will be requested. The report will show the impact of client requirements changes on the project on both the design and implementation phase.

2 Vision

We want to discuss the process of software development in order to overcome the limits of a technology-based approach in heterogeneous distributed system application design. We try to adopt a model-driven software development taking into account the AGILE methods for cooperation and work management. In particular, we want to:

- define a formal, executable model of the application to receive feedback from the client and ensure that requirements are clearly defined as soon as possible
- minimize the abstraction gap between the development tools and the application domain entities
- delay any technological hypotesis as much as possible in order to improve application reusability
- create flexible applications to resist requirements changes and add new features easily

3 Goals

The goal is to solve the given problem following the principles described in the vision and determine if this approach is viable and convenient. We want to build a prototype quickly and test if we can add new features to the application with minimum effort.

4 Requirements

We have to solve the following problem:

The Security Department of an Airport intends to exploit a differential drive robot equipped with a sonar (and some other device) to inspect -in a safe way- unattended bags when they are found in some sensible area of the Airport.

The software working on the inspector-roobot should support the following behavior:

- an operator drives the robot from an initial point (robot base area, RBA) towards the bag. To drive the robot the operator makes use of a remote robot control interface running on a smart device or a PC. The robot must accept commands from a single source only;
- as soon as the robot sonar perceives the bag within a prefixed distance (e.g. d=20cm):
  1. the robot automatically stops
  2. the robot starts blinking a led
  3. the robot starts a first detection phase (e.g. it moves around and performs some action according to its equiment - for example it could take some photo of the bag)
  4. the robot sends the results of its detection phase to the Airport Security Center;
- at the end of its work, the robot turns the led off and automatically returns to its RBA. During this phase the Airport Security Center could emit an 'alarm' signal; in this case the robot must restart to blink.

STEP 1
Design and build a working prototype of this inspector-robot.
Non functional requriments at step1
The goal is to build a software system able to evolve from an initial proptotype (defined as the result of a problem analysis phase) to a final, testable product, by 'mixing' in a proper (pragmatically useful) way agile (SCRUM) software development with modelling.

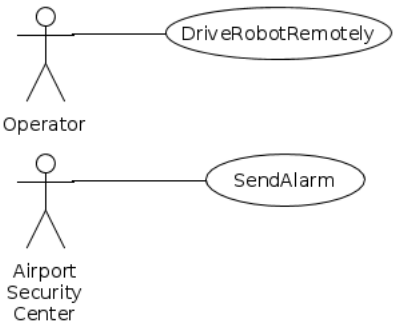
5 Requirement analysis

5.1 Use cases

The use cases describe how actors (UML actors i.e. the role played by a user or external system) interact with the system. In the requirements we can identify two external entities:

- **The operator** that drives the robot remotely from the initial point to the bag.
- **The Airport Security Center** that receives the results of the robot's detection phase and then it may emit an 'alarm' signal.

These interactions are shown by the UML below:



5.2 Scenarios

Title	DriveRobotRemotely
Description	The operator drives the robot to the suspicious bag
Relationships	
Actors	Operator
Preconditions	The robot is in the RBA, waiting for commands from the operator.
Postconditions	The robot starts the detection phase.
Main scenario	The operator uses the remote console to drive the robot. When the robot perceives the bag, it starts the detection phase.

Title	SendAlarm
Description	The Airport Security Center sends an alarm signal to the robot if needed
Relationships	
Actors	Airport Security Center
Preconditions	The Airport Security Center received the detection results
Postconditions	The robot blinks its led until it comes back to the RBA.
Main scenario	The Airport Security Center uses its interface to send the alarm to the robot. The robot blinks its led.

5.3 (Domain) model

In this phase we try to find an agreement with the client on what the entities mentioned in the requirements are and what they have to do.
The system is composed by three parts:

- **Operator's remote console**
- **Airport Security Center's remote console**
- **Differential drive robot**

A **console** is a physical or virtual device that allows communication between the system and an external entity. It can get user input data and send them to the system, show some system output data to the user or both. In this case,

the operator's console can get input from the operator and the Airport Security Center's console can receive the detection results and emit an alarm signal.
A **differential drive robot** is a composed entity that is able to use some devices to perform actions and receive data from the environment. It can also communicate with other parts of the system. All differential drive robots must have a sonar and are able to move in the environment. In this case, the differential drive robot has DC motors and wheels to move, a sonar, a led and a camera. DC motors, wheels, led, sonar and camera are the hardware components mounted on the robot.
A **DC motor** can spin the attached wheel clockwise or counter-clockwise.
A **led** can be turned on or off.
A **sonar** can send an ultrasonic signal (trigger) and generates a corresponding response waveform (echo). The waveform analysis allows to estimate the distance from an obstacle.
A **camera** is a device that can take photos when requested. It will be used by the robot in the detection phase.

The system can be formally defined with a custom language / executable metamodel developed by our software house. It allows us to describe what are the parts of the system, how they interact with each other and their behaviour.

The following is a first description of the system obtained by the requirement analysis:

```
1  RobotSystem testCase2016Analysis
2
3  Dispatch drive : drive(X)
4  Dispatch detectionResults : detectionResults(X)
5  Event alarm : alarm
6  Event obstacle : obstacle(X)
7
8  Context ctxDriveRobot ip[host="localhost" port=8010]
9  Context ctxOperator ip[host="localhost" port=8015]
10 Context ctxASC ip[host="localhost" port=8020]
11
12 QActor operator context ctxOperator
13 {
14   Plan init normal
15     println("Operator starts");
16     switchToPlan sendCommands
17
18   Plan sendCommands
19     println("Waiting for commands");
20     delay time(3000);
21     println("Sending command");
```

```
22      forward driverobot -m drive : drive("driveCmdPayLoad"
            );
23      println("Command sent");
24      delay time(2000);
25      repeatPlan 0
26  }
27
28  QActor asc context ctxASC
29  {
30    Plan init normal
31      println("ASC starts");
32      switchToPlan waitForResults
33
34    Plan waitForResults
35      receiveMsg time(600000);
36      onMsg detectionResults : detectionResults(X) ->
37      println(detectionResults(X));
38      onMsg detectionResults : detectionResults(X) ->
39      switchToPlan riskDecision
40
41    Plan riskDecision
42      println("Evaluating risks");
43      delay time(3000);
44      //It could emit the alarm signal
45      emit alarm : alarm
46  }
47
48  Robot mock QActor driverobot context ctxDriveRobot
49  {
50    Plan init normal
51      println("driverobot starts");
52      switchToPlan drive
53
54    Plan drive
55      //We'll have to make sure that the robot executes the
            commands from the first console only
56      receiveMsg time(600000) react event obstacle ->
            detect;
57      onMsg drive : drive(X) -> println(savingmove(X));
58      onMsg drive : drive(X) -> println(driving(X));
59      repeatPlan 0
60
61    Plan detect
62      println("Stopping...");
63      delay time(1000);
```

```
64      println("Start blinking the led");
65      println("Starting detection Phase...");
66      delay time(3000);
67      println("Sending results");
68      forward asc -m detectionResults : detectionResults("
            results");
69      println("Detection Results Sent");
70      println("Stop blinking the led");
71      println("Back to base");
72      switchToPlan backToBase
73
74    Plan backToBase
75      delay time(20000) react event alarm -> alarmReaction;
76      switchToPlan finish
77
78    Plan alarmReaction resumeLastPlan
79      println("Alarm!");
80      println("Start blinking the led")
81
82    Plan finish
83      println("DriveRobot ends")
84  }
```

The operator can only send commands to drive the robot. The Airport Security Center waits for the detection results and can emit the alarm only after the results have been sent.

5.4 Test plan

We can do a test plan even before starting to implement the application, as a means to specify the expected behaviour of the system in a precise way. We just need to check if the parts of the system behave and interact with each other as defined in the requirements. We can't express tests formally tough, because we already described the entities as actors, so object oriented tests (e.g. JUnit tests) are inadequate. Furthermore, some tests should check the interaction of the physiscal system with the environment and this can only be achieved by observing the actual behaviour of the system. Thus, we'll describe these tests in natural language.
In the initial phase, the operator drives the robot. We have to check the following:

- the operator can send commands to the robot
- the robot executes the commands it receives
- the robot accepts commands only from a single source
- the robot perceives the presence of an obstacle

In the detection phase, the robot inspects the bag. We'll test the following:

– the robot stops and ignores commands from the operator
– the robot starts blinking after it stopped
– the robot can take a picture of the bag
– the robot can send the results to the Airport Security Center
– the Airport Security Center can receive the results of the inspection
– the robot stops blinking at the end of this phase

In the final phase, the robot comes back to the RBA. These are the tests we'll do:

– the robot actually comes back autonomously
– the Airport Security Center can emit the alarm signal
– the robot blinks the led if it perceives the alarm

At this stage in the development process, we can't define more specific functional or integration tests, we'll add them as needed during the implementation phase. We still haven't decided what technology we will use to implement the application, so we can't write executable tests yet. However, at the end of the analysis phase, we'll have an executable logical architecture of the application and we'll be able to perform some of the tests on it.

6 Problem analysis

6.1 Logic architecture

Logic architecture can be expressed in 3 dimensions:

1. **Structure**: what parts the system is made of.
2. **Interaction**: how the parts of the system communicate with each other.
3. **Behaviour**: what the parts of the system do.

We can formally express these concepts with the DDR custom language:

```
1  RobotSystem testCase2016LogicArchitecture
2
3  Event local_inputDrive : local_inputDrive(X)   //events
       from GUI/External Input
4  Dispatch drive : drive(X)
5  Dispatch detectionResults : detectionResults(X)
6  Event alarm : alarm
7  Event local_alarm : local_alarm          //events from GUI
       /External Input
8  Event obstacle : obstacle(X)
9  Event bagFound : bagFound
10 Event endDetection : endDetection
11 Event botIsBack : botIsBack             //signals the
       return to the base of the robot
12
```

```
13 Context ctxDriveRobot ip[host="192.168.1.69" port=8010]
14 Context ctxOperator ip[host="192.168.1.2" port=8015]
15 Context ctxASC ip[host="192.168.1.2" port=8020]
16
17 QActor led context ctxDriveRobot
18 {
19   Plan init normal
20     println("Led starts");
21     switchToPlan senseStartBlink
22
23   Plan senseStartBlink
24     println("Led Off");
25     sense time(60000) bagFound -> startBlinking;
26     repeatPlan 0
27
28   Plan startBlinking
29     println("led On");
30     delay time(1000) react event endDetection ->
           senseAlarm;
31     println("Led Off");
32     delay time(1000) react event endDetection ->
           senseAlarm;
33     repeatPlan 0
34
35   Plan senseAlarm
36     println("Led Off");
37     sense time(60000) alarm-> blinkingAlarm;
38     repeatPlan 0
39
40   Plan blinkingAlarm
41     println("led On");
42     delay time(500) react event botIsBack -> finish;
43     println("Led Off");
44     delay time(500) react event botIsBack -> finish;
45     repeatPlan 0
46
47   Plan finish
48     println("Led ends")
49 }
50
51 QActor operator context ctxOperator -g cyan
52 {
53   Plan init normal
54     println("Operator starts");
55     switchToPlan senseInput
```

```
56
57    Plan senseInput
58      sense time(60000) local_inputDrive ->
            sendDriveCommands;
59      repeatPlan 0
60
61    Plan sendDriveCommands resumeLastPlan
62      onEvent local_inputDrive : local_inputDrive(X) ->
            forward driverobot -m drive : drive(X)
63  }
64
65  QActor asc context ctxASC -g green
66  {
67    Plan init normal
68      println("ASC starts");
69      switchToPlan work
70
71    Plan work
72      receiveMsg time(600000);
73      onMsg detectionResults : detectionResults(X) ->
            println(detectionResults(X));
74      switchToPlan senseAlarm
75
76    Plan senseAlarm
77      sense time(100000) local_alarm -> continue;
78      onEvent local_alarm : local_alarm -> emit alarm :
            alarm
79  }
80
81  Robot mock QActor driverobot context ctxDriveRobot
82  {
83    Plan init normal
84      println("driverobot starts");
85      solve consult("talkTheory.pl") time(0) onFailSwitchTo
            prologFailure;
86      switchToPlan drive
87
88    Plan drive
89      //We'll have to make sure that the robot executes the
            commands from the first console only
90      receiveMsg time(600000) react event obstacle ->
            detect;
91      onMsg drive : drive(X) -> println(savingmove(X));
92      onMsg drive : drive(X) -> solve X time(0);
93      repeatPlan 0
```

```
94
95    Plan detect
96      println("Stopping...");
97      robotStop speed(100) time(1000);
98      emit bagFound : bagFound;
99      println("Starting detection Phase...");
100     [?? detection(X) ] forward asc -m detectionResults :
            detectionResults(X);
101     println("Detection Results Sent");
102     emit endDetection : endDetection;
103     println("Back to base");
104     switchToPlan backToBase
105
106   Plan backToBase
107     solve backToBase time(0); //It doesn't need to react,
            as the qactor led handles that
108     switchToPlan finish
109
110   Plan finish
111     emit botIsBack : botIsBack;
112     println("DriveRobot ends")
113
114   Plan prologFailure resumeLastPlan
115     println("Failed to load talkTheory")
116 }
```

This describes the whole logic architecture of our application. It can also be executed so that the client can confirm that the analysis defined a system that behaves as required.
This architecture derives from the one obtained in the domain model and introduces new interactions and a new entity.
The **DriveRobot** receives commands from the Operator Interface in the first phase, executes its automatic operations during the detection phase, it sends results to the ASCConsole and comes back to the RBA in the end. It has to react to obstacles to begin the detection phase.
The **Operator Interface** receives commands from the operator as events and sends the corresponding commands to the robot.
The **ASCConsole** receives the detection results from the detection phase and then enables the Airport Security Center to emit the alarm.
We decided to introduce the **led** as an active entity separated from the robot because it is an active entity that has to intecract with other entities and has its own behaviour, modeling it as a passive object managed by the robot is inappropriate. The led starts to blink when the detection phase begins, stops to blink when the detection phase ends and it starts to blink if the alarm is emitted when the robot is coming back.
The **camera** will be modeled as a passive entity that can only take a picture

when the robot asks for it.
We defined the accessory event botIsBack to signal that the robot has come back to the RBA. This event can be used to stop the led if an alarm has been emitted before.
The interaction with external entities (the operator and the ASC) have been modeled as local events.

6.2 Abstraction gap

The abstraction gap is the distance between the concepts used to model the problem and those implied by the technology of choice. Thanks to the framework provided, executable code is generated from the model defined in the ddr meta-model. Thus, adopting this framework allows the application designers to use an extremely high-level description of the problem, closer to the application domain, reducing considerably the abstraction gap. The specific technology to be used can be decided later, in a configuration phase. The advantage of using a meta-model and a code generator is also that it can be extended to support more advanced concepts.

6.3 Risk analysis

Using the framework code generators, we can write most of the code independently from the specific implementation technology. Although the qa/ddr meta-model is technology independent, the code generated automatically may require some kind of environment on the computational nodes where it will be deployed (e.g. the JVM, the .NET runtime environment, a specific operating system etc).

7 Work plan

After the analysis phase, we decided to develop the application using the ddr framework, so that we don't start from scratch. We can reuse the executable logic architecture and enhance it. The framework already offers the implementation logic for some parts of the system and it offers high level abstractions that allow the developers to focus on business logic and not to worry too much about boilerplate code.
We'll use the following features offered by the framework:

- A communication system that allows the parts of the system to send and receive messages and events
- Reactive actions
- Timed actions
- The robot configuration
- DC motors driver
- sonar driver (and management of its data)

We'll implement the remaining features following the SCRUM framework for work planning. So we defined a product backlog which is a prioritized list of tasks needed to complete the project:

1. Define the robot configuration with .baseddr
2. Implement the console interfaces that allow external entities to interact with the system
3. Implement the led driver
4. Decide and implement a way to send a picture in the ddr framework
5. Develop the detection phase logic with the camera driver (as a mock entity)
6. Create an algorithm that allows the robot to come back to the RBA

8 Project

8.1 Structure

The structure is essentially the same as the logic architecture. Our robot has no camera, so we'll implement it as a mock device.

8.2 Interaction

There are no significant changes from the logic architecture.

8.3 Behavior

More details have been added to implement the missing features described in the work plan.
The consoles used by the ASC and the operator will be GUIs that allow them to interact with the system. The robot behaviour has slightly changed in the first phase: to ensure it receives messages from a single source, it memorizes the sender of the first received drive message and accepts new drive commands from that source only.

```
1  RobotSystem testCase2016Project
2
3  Event local_inputDrive : local_inputDrive(X)   //events
       from GUI/External Input
4  Dispatch drive : drive(X)
5  Dispatch detectionResults : detectionResults(X)
6  Event alarm : alarm
7  Event local_alarm : local_alarm          //events from GUI
       /External Input
8  Event obstacle : obstacle(X)
9  Event bagFound : bagFound
10 Event endDetection : endDetection
11 Event botIsBack : botIsBack             //signals the
       return to the base of the robot
```

```
12
13 Context ctxDriveRobot ip[host="192.168.1.69" port=8010]
14 Context ctxOperator ip[host="192.168.1.2" port=8015]
15 Context ctxASC ip[host="192.168.1.2" port=8020]
16
17 QActor led context ctxDriveRobot
18 {
19   Plan init normal
20     println("Led starts");
21     solve consult("ledTheory.pl") time(0) onFailSwitchTo
           prologFailure;
22     switchToPlan senseStartBlink
23
24   Plan senseStartBlink
25     println("Led Sensing");
26     solve turnTheLed(off) time(0) onFailSwitchTo
           prologFailure;
27     sense time(60000) bagFound -> startBlinking;
28     repeatPlan 0
29
30   Plan startBlinking
31     println("led On");
32     solve turnTheLed(on) time(0) onFailSwitchTo
           prologFailure;
33     delay time(500) react event endDetection ->
           senseAlarm;
34     println("Led Off");
35     solve turnTheLed(off) time(0) onFailSwitchTo
           prologFailure;
36     delay time(500) react event endDetection ->
           senseAlarm;
37     repeatPlan 0
38
39   Plan senseAlarm
40     println("Led Off, waiting alarm");
41     solve turnTheLed(off) time(0);
42     sense time(60000) alarm-> blinkingAlarm;
43     repeatPlan 0
44
45   Plan blinkingAlarm
46     println("led On");
47     solve turnTheLed(on) time(0) onFailSwitchTo
           prologFailure;
48     delay time(500) react event botIsBack -> finish;
49     println("Led Off");
```

```
50      solve turnTheLed(off) time(0) onFailSwitchTo
            prologFailure;
51      delay time(500) react event botIsBack -> finish;
52      repeatPlan 0
53
54    Plan finish
55      solve turnTheLed(offcompletely) time(0)
            onFailSwitchTo prologFailure;
56      println("Led ends")
57
58    Plan prologFailure resumeLastPlan
59      println("Prolog Failure LED")
60  }
61
62  QActor operator context ctxOperator -g cyan
63  {
64    Plan init normal
65      println("Operator starts");
66      switchToPlan senseInput
67
68    Plan senseInput
69      sense time(60000) local_inputDrive ->
            sendDriveCommands;
70      repeatPlan 0
71
72    Plan sendDriveCommands resumeLastPlan
73      onEvent local_inputDrive : local_inputDrive(X) ->
            forward driverobot -m drive : drive(X)
74  }
75
76  QActor asc context ctxASC -g green
77  {
78    Plan init normal
79      println("ASC starts");
80      switchToPlan work
81
82    Plan work
83      receiveMsg time(600000);
84      onMsg detectionResults : detectionResults(X) ->
85      solve actorOp(loadResults(X)) time(0) onFailSwitchTo
            prologFailure;
86      switchToPlan senseAlarm
87
88    Plan senseAlarm
89      sense time(100000) local_alarm -> continue;
```

```
90      onEvent local_alarm : local_alarm -> emit alarm :
            alarm
91
92    Plan prologFailure resumeLastPlan
93      println("Prolog failure ASC")
94  }
95
96  /*
97   QActor obstacleemitter context ctxDriveRobot
98  {
99    Plan init normal
100     delay time(15000);
101     emit obstacle : obstacle(12);
102     println("Emitted obstacle event")
103 }
104 */
105
106 Robot plexiBox QActor driverobot context ctxDriveRobot
107 {
108   Plan init normal
109     println("driverobot starts");
110     solve consult("talkTheory.pl") time(0) onFailSwitchTo
            prologFailure;
111     println("consulting driveRobotTheory");
112     solve consult("driveRobotTheory.pl") time(0)
            onFailSwitchTo prologFailure;
113     println("consulted driveRobotTheory");
114     switchToPlan receiveFirstCommand
115
116   Plan receiveFirstCommand
117     println("ROBOT waiting first message");
118     receiveMsg time(600000) react event obstacle ->
            detect;
119     //Save first sender
120     [?? msg(drive,dispatch, S, R, drive(X), N)] solve
            assert(firstSender(S)) time(0);
121     onMsg drive : drive(X) -> solve savemove(X) time(0)
            onFailSwitchTo savemoveFailure;
122     onMsg drive : drive(X) -> println(X);
123     onMsg drive : drive(X) -> solve X time(0)
            onFailSwitchTo prologFailure;
124     onMsg drive : drive(X) -> switchToPlan drive;
125     repeatPlan 0
126
127   Plan drive
```

```
128     receiveMsg time(600000) react event obstacle ->
            detect;
129     //To make sure that the sender is the same as the
            first one
130     [?? msg(drive,dispatch, S, R, drive(X), N)] solve
            firstSender(S) time(0) onFailSwitchTo drive;
131     onMsg drive : drive(X) -> println(X);
132     onMsg drive : drive(X) -> solve savemove(X) time(0)
            onFailSwitchTo prologFailure;
133     onMsg drive : drive(X) -> solve X time(0)
            onFailSwitchTo prologFailure;
134     repeatPlan 0
135
136   Plan detect
137     println("Stopping...");
138     robotStop speed(100) time(0);
139     delay time(1000);
140     println("Stopped");
141     solve endSavemoves time(0) onFailSwitchTo
            prologFailure;
142     emit bagFound : bagFound;
143     println("Starting detection Phase...");
144     delay time ( 3000);
145     [!? detection(X) ] forward asc -m detectionResults :
            detectionResults(X);
146     delay time ( 3000);
147     println("Detection Results Sent");
148     emit endDetection : endDetection;
149     println("Back to base");
150     switchToPlan backToBase
151
152   Plan backToBase
153     solve backToBase time(0) onFailSwitchTo prologFailure
            ;  //It doesn't need to react, as the qactor led
            handles that
154     switchToPlan finish
155
156   Plan finish
157     emit botIsBack : botIsBack;
158     println("DriveRobot ends")
159
160   Plan prologFailure
161     println("Robot Failed to load prolog theories")
162
163   Plan savemoveFailure resumeLastPlan
```

```
164     println("Failed save move")
165
166 }
```

9 Implementation

9.1 Robot configuration

The file robots.baseddr contains the configuration we used:

```
1  /*
2   * ========================================
3   * plexiBox
4   * ========================================
5   * Convenzione WiringPI!!!!!!!!!!!!!!!!
6   */
7  RobotBase plexiBox
8  //BASIC
9  motorleft  = Motor  [ gpiomotor pincw 13 pinccw 12  ]
       position: LEFT
10 motorright = Motor  [ gpiomotor pincw 4 pinccw 5  ]
       position: RIGHT
11 distanceRadar = Distance  [ sonarhcsr04 pintrig 0 pinecho
       2]  position: FRONT_TOP
12 //line = Line   [ gpioswitch pin 15 activelow ]  position
       : BOTTOM
13 //COMPOSED
14 motors = Actuators [ motorleft , motorright  ] private
       position: BOTTOM
15 Mainrobot plexiBox  [ motors ]
16 ;
17 /*
18  * ========================================
19  * mock
20  * ========================================
21  */
22 RobotBase mock
23  //BASIC
24 motorleft  = Motor  [ simulated 0  ]  position: LEFT
25 motorright = Motor  [ simulated 0  ]  position: RIGHT
26 l1Mock      = Line   [ simulated 0  ]  position: BOTTOM
27 distFrontMock= Distance [ simulated 0  ] position: FRONT
28 mgn1 = Magnetometer  [ simulated 0 ] private position:
       FRONT
29 //COMPOSED
```

```
30 rot     = Rotation [ mgn1   ] private position: FRONT
31 motors = Actuators [ motorleft , motorright   ] private
       position: BOTTOM
32 Mainrobot mock   [ motors,rot ]
33 ;
34
35 /*
36  * senseBot
37  *
38  */
39
40 RobotBase senseBot
41 //BASIC
42 motorleft  = Motor   [ gpiomotor pincw 6 pinccw 5   ]
       position: LEFT
43 motorright = Motor   [ gpiomotor pincw 4 pinccw 0   ]
       position: RIGHT
44 distanceRadar = Distance   [ sonarhcsr04 pintrig 22
       pinecho 21 ]   position: FRONT_TOP
45 //line = Line    [ gpioswitch pin 15 activelow ]   position
       : BOTTOM
46 //COMPOSED
47 motors = Actuators [ motorleft , motorright   ] private
       position: BOTTOM
48 Mainrobot senseBot   [ motors ]
49 ;
```

9.2 Operator and ASC console

The **operator console** includes a GUI that translates external input events into messages to drive the robot.

```
1  /* Generated by AN DISI Unibo */
2  /*
3  This code is generated only ONCE
4  */
5  package it.unibo.operator;
6  import java.awt.Button;
7  import java.awt.GridLayout;
8  import java.awt.Label;
9  import java.awt.Panel;
10 import java.awt.event.MouseEvent;
11 import java.awt.event.MouseListener;
12 import java.util.HashMap;
13 import java.util.Map;
```

```
14
15 import it.unibo.baseEnv.basicFrame.EnvFrame;
16 import it.unibo.is.interfaces.IOutputEnvView;
17 import it.unibo.qactors.ActorContext;
18
19 public class Operator extends AbstractOperator {
20
21   protected Map<String, String> driveCmdMap;
22
23   public final static String Forward="Forward";
24   public final static String Backward="Backward";
25   public final static String Right="Right";
26   public final static String Left="left";
27   public final static String Halt="Halt";
28
29   public Operator(String actorId, ActorContext myCtx,
       IOutputEnvView outEnvView ) throws Exception{
30     super(actorId, myCtx, outEnvView);
31   }
32
33   protected void initCmdMap(){
34     driveCmdMap=new HashMap<>();
35     driveCmdMap.put(Forward, "executeInput(move(mf,100,0)
         )");
36     driveCmdMap.put(Backward, "executeInput(move(mb
         ,100,0))");
37     driveCmdMap.put(Right, "executeInput(move(mr,100,0))"
         );
38     driveCmdMap.put(Left, "executeInput(move(ml,100,0))")
         ;
39     driveCmdMap.put(Halt, "executeInput(move(h,100,0))");
40   }
41
42   @Override
43   protected void addInputPanel(int size) {
44   }
45
46   @Override
47   protected void addCmdPanels(){
48     initCmdMap();
49     ((EnvFrame) env).setSize(800,700);
50     Panel p = new Panel();
51     GridLayout l =  new GridLayout();
52     l.setVgap(10);
53     l.setHgap(10);
```

```
54      l.setColumns(3);
55      l.setRows(3);
56      p.setLayout(l);
57
58      MouseListener ml =new MouseListener() {
59        @Override
60        public void mouseReleased(MouseEvent e) {
61  //          String cmd = ((Button)e.getSource()).getLabel()
        ;
62  //          if(!cmd.equals(Halt)){
63  //            execAction(Halt);
64  //          }
65          System.out.println("DEBUG: UNPRESSED");
66        }
67        @Override
68        public void mousePressed(MouseEvent e) {
69          Button b = (Button)e.getSource();
70          execAction(b.getLabel());
71          System.out.println("DEBUG: PRESSED" +  b.getLabel
              ());
72        }
73        @Override
74        public void mouseExited(MouseEvent e) {
75        }
76        @Override
77        public void mouseEntered(MouseEvent e) {
78        }
79        @Override
80        public void mouseClicked(MouseEvent e) {
81        }
82      };
83
84      Button forward = new Button(Forward);
85      forward.addMouseListener(ml);
86      Button backward = new Button(Backward);
87      backward.addMouseListener(ml);
88      Button right = new Button(Right);
89      right.addMouseListener(ml);
90      Button left = new Button(Left);
91      left.addMouseListener(ml);
92      Button halt = new Button(Halt);
93      halt.addMouseListener(ml);
94      p.add(new Label(""));
95      p.add(forward);
96      p.add(new Label(""));
```

```
97      p.add(left);
98      p.add(halt);
99      p.add(right);
100     p.add(new Label(""));
101     p.add(backward);
102     p.add(new Label(""));
103     ((EnvFrame) env).add(p);
104     ((EnvFrame) env).validate();
105   }
106
107   @Override
108   public void execAction(String cmd) {
109     super.execAction(cmd);
110
111     if(driveCmdMap.containsKey(cmd)){
112       String actualCmd = driveCmdMap.get(cmd);
113       platform.raiseEvent("input", "local_inputDrive", "
              local_inputDrive("+actualCmd+")");
114       return;
115     }
116   }
117
118 }
```

The **ASC console** includes a GUI that shows the image received from the robot at the end of the detection phase and then shows a button that emits the alarm if pressed.

```
1  /* Generated by AN DISI Unibo */
2  /*
3  This code is generated only ONCE
4  */
5  package it.unibo.asc;
6  import java.awt.*;
7  import java.io.ByteArrayInputStream;
8  import java.io.IOException;
9  import java.util.Base64;
10
11 import javax.imageio.ImageIO;
12
13 import java.awt.event.ActionEvent;
14 import java.awt.event.ActionListener;
15
16 import it.unibo.is.interfaces.IOutputEnvView;
17 import it.unibo.qactors.ActorContext;
18
```

```
19 public class Asc extends AbstractAsc {
20   public Asc(String actorId, ActorContext myCtx,
         IOutputEnvView outEnvView )  throws Exception{
21     super(actorId, myCtx, outEnvView);
22   }
23
24   protected Label userMsg;
25   protected Button alarm;
26   protected ImagePanel results;
27
28   @Override
29   protected void addCmdPanels(){
30     //super.addCmdPanels();
31     //photo panel
32     ((Frame) env).removeAll();
33     GridLayout l = new GridLayout();
34     l.setColumns(2);
35     l.setRows(2);
36     ((Frame) env).setLayout(l);
37     results = new ImagePanel();
38     results.setSize(300, 400);
39     ((Frame) env).add(results);
40     alarm = new Button("Alarm");
41     alarm.setBackground(Color.red);
42     alarm.addActionListener(new ActionListener() {
43       @Override
44       public void actionPerformed(ActionEvent e) {
45           execAction("Alarm");
46       }
47     });
48     alarm.setEnabled(false);
49     ((Frame) env).add(alarm);
50     userMsg = new Label("Waiting for results");
51     ((Frame) env).add(userMsg);
52     ((Frame) env).validate();
53   }
54
55   //this is called when the results are received
56   public void loadResults(String imageString){
57         byte[] imageBytes = Base64.getDecoder().decode(
              imageString);
58         try {
59       Image image = ImageIO.read(new ByteArrayInputStream
            (imageBytes));
60       results.setImage(image);
```

```
61      } catch (IOException e) {
62        System.out.println("MyPanel: Image error!");
63        e.printStackTrace();
64      }
65
66      alarm.setEnabled(true);
67      userMsg.setText("Results received");
68      ((Frame) env).validate();
69    }
70
71    @Override
72    public void execAction(String cmd) {
73      super.execAction(cmd);
74      if( cmd.equals("Alarm") ){
75        platform.raiseEvent("input", "local_alarm", "
              local_alarm");
76        userMsg.setText("Alarm sent!");
77        return;
78      }
79    }
80
81    protected class ImagePanel extends Panel{
82       /**
83       *
84       */
85      private static final long serialVersionUID = 1L;
86      private Image image;
87
88       public ImagePanel(){
89         image = null;
90       }
91
92       public void paint(Graphics g){
93              super.paint(g);
94              if(image != null){
95                int w = getWidth();
96                int h = getHeight();
97                int imageWidth = image.getWidth(this);
98                int imageHeight = image.getHeight(this);
99                int x = (w - imageWidth)/2;
100               int y = (h - imageHeight)/2;
101               g.drawImage(image, x, y, this);
102             }
103         }
104
```

```
105         public void setImage(Image image){
106             this.image=image;
107                 validate();
108         }
109     }
110 
111 }
```

9.3 Led

The **led** blinking logic is implemented directly as QActor behaviour.
The Prolog theory turnTheLed/1 allows the QActor to manage the led and actually turn it on and off calling the underlying Java code.

```
1  %createPi4jLed( PinNum) :-
2  %    actorobj( Actor ),
3  %    Actor <- getOutputEnvView returns OutView ,
4  % class("it.unibo.devices.qa.DeviceLedPi4jQa") <-
       createLed( OutView, PinNum ) returns LED.
5  
6  %turnTheLed( on ):-
7  % class("it.unibo.devices.qa.DeviceLedPi4jQa") <-
       getTheLed  returns LED,
8  % LED <- turnOn .
9  
10 %turnTheLed( off ):-
11 % class("it.unibo.devices.qa.DeviceLedPi4jQa") <-
       getTheLed  returns LED,
12 % LED <- turnOff .
13 
14 pinNum(25).
15 
16 %turnTheLed(on):-
17 % pinNum(X),
18 % class("it.unibo.devices.qa.LedDevicesFactory") <-
       getTheLedCmd(X) returns LED,
19 % LED <- turnOn.
20 
21 
22 %turnTheLed(off):-
23 % pinNum(X),
24 % class("it.unibo.devices.qa.LedDevicesFactory") <-
       getTheLedCmd(X) returns LED,
25 % LED <- turnOff.
26 
```

```
27 turnTheLed(on):-
28   pinNum(X),
29   class("it.unibo.devices.qa.LedDevicesFactory") <-
         getTheLedCmdInterpreter(X) returns LED,
30   LED <- turnOn.
31
32
33 turnTheLed(off):-
34   pinNum(X),
35   class("it.unibo.devices.qa.LedDevicesFactory") <-
         getTheLedCmdInterpreter(X) returns LED,
36   LED <- turnOff.
37
38 turnTheLed(offcompletely):-
39   pinNum(X),
40   class("it.unibo.devices.qa.LedDevicesFactory") <-
         getTheLedCmdInterpreter(X) returns LED,
41   LED <- turnOffForever.
42
43 %initialize  :-   createPi4jLed(25).
44 initialize.
45
46 :- initialization(initialize).
```

The led instance is created through a factory:

```
1  package it.unibo.devices.qa;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  /**
7   * BCM convention!!
8   */
9  public class LedDevicesFactory {
10
11   private static Map<Integer, ILed> leds;
12
13   private static String command="sudo bash/gpioPin.sh";
14   private static String commandInterpreter="sudo bash/
         gpioPinInterpreter.sh";
15
16   static {
17     leds = new HashMap<>();
18   }
19
```

```
20    public static ILed getTheLedCmd(int nPin){
21      if(leds.containsKey(nPin)&&leds.get(nPin) instanceof
            LedShellCmd){
22        return leds.get(nPin);
23      }
24      leds.put(nPin, new LedShellCmd(command, nPin));
25      return leds.get(nPin);
26    }
27
28    public static ILed getTheLedCmdInterpreter(int nPin){
29      if(leds.containsKey(nPin)&&leds.get(nPin) instanceof
            LedShellCmdInterpreter){
30        return leds.get(nPin);
31      }
32      leds.put(nPin, new LedShellCmdInterpreter(
            commandInterpreter, nPin));
33      return leds.get(nPin);
34    }
35
36    public static ILed getTheLedPi4j(int nPin){
37      if(leds.containsKey(nPin)&&leds.get(nPin) instanceof
            Pi4jLed){
38        return leds.get(nPin);
39      }
40      leds.put(nPin, new Pi4jLed(nPin));
41      return leds.get(nPin);
42    }
43
44  }
```

We implemented the led as a bash script that receives zeros and ones to turn the led on and off:

```
1   package it.unibo.devices.qa;
2
3   import java.io.PrintWriter;
4
5   import it.unibo.sartiballanti.utils.Utils;
6
7   public class LedShellCmdInterpreter extends LedShellCmd {
8
9     private PrintWriter pw;
10
11    public LedShellCmdInterpreter(String command, int nPin)
          {
12      super(command,nPin);
```

```
13      this.pw=new PrintWriter(Utils.
            executeShellCommandOutput(command +" "+ nPin));
14    }
15
16    @Override
17    public void turnOn() {
18      pw.print("1\n");
19      pw.flush();
20    }
21
22    @Override
23    public void turnOff() {
24      pw.println("0\n");
25      pw.flush();
26    }
27
28    public void turnOffForever(){
29      turnOff();
30      pw.close();
31    }
32
33  }
```

```
1  echo "$1" > /sys/class/gpio/unexport #
2  echo "$1" > /sys/class/gpio/export #
3  cd /sys/class/gpio/gpio"$1" #
4
5  echo out > direction #
6
7  while read ONOFF
8  do
9    echo $ONOFF > value #
10 done
```

9.4 Camera

The camera implements the following interface:

```
1  package it.unibo.sartiballanti.camera;
2
3  public interface ICamera {
4    public byte[] takePhoto();
5  }
```

This is the implementation of the mock camera:

```java
package it.unibo.sartiballanti.camera;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class MockFileCamera implements ICamera {

  private String imgPath;

  public MockFileCamera(String imgPath){
    this.imgPath=imgPath;
  }

  @Override
  public byte[] takePhoto() {
    try {
      return Files.readAllBytes(Paths.get(imgPath));
    } catch (IOException e) {
      e.printStackTrace();
    }
    return null;
  }
}
```

9.5 Detection phase

The **robot** uses an actor method to execute the detection phase. It takes a picture of the bag using the simulated **camera** and sends it to the ASC. In order to send send the photo as a message payload in the ddr framework, we needed to obtain a string representation of the image.

```java
/* Generated by AN DISI Unibo */
package it.unibo.driverobot;
import java.util.Base64;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.ActorContext;
import it.unibo.sartiballanti.camera.CameraFactory;

public class Driverobot extends AbstractDriverobot {
  public Driverobot(String actorId, ActorContext myCtx,
      IOutputEnvView outEnvView ,it.unibo.iot.executors.
      baseRobot.IBaseRobot baserobot) throws Exception{
    super(actorId,myCtx,outEnvView ,baserobot );
```

```
11    }
12
13    public String takeStringifiedPhoto(){
14      byte[] img= CameraFactory.getInstance().getCamera().
            takePhoto();
15      return Base64.getEncoder().encodeToString(img);
16    }
17  }
```

9.6 Back to base

The **driveRobotTheory** is used to implement a simple algorithm to come back autonomously: the robot memorizes every move it makes in the first phase, so it can come back executing the same moves backwards.

```
1  %drivecommand example
2  %executeInput(move(mf,100,1000,0))
3
4  %lastmove is the next move to save,
5  %I can get the starting time, but I can't insert it into
       the moveList until it ends.
6
7  %The savemove/1 rule uses the knowledge base to store and
       update the information,
8  %but it uses savemove/5 to get the updated lastMove and
       moveList.
9
10 %Initial facts
11 moveList([]).
12 lastMove(none,0).
13
14 savemove(executeInput(CUR)):-
15    moveList(L),
16    lastMove(LASTMOVE,MVTIME),
17    savemove(CUR,lastMove(LASTMOVE,MVTIME),L,NEWLASTMOVE,
          NEWL),
18    retract(lastMove(_,_)),
19    retract(moveList(_)),
20    assert(NEWLASTMOVE),
21    assert(moveList(NEWL)).
22
23 %Here the savemove rule is implemented without assert and
       retract.
24 %savemove(CUR,LAST,LIST,NEWLAST,NEWLIST)
```

```
25 savemove(CUR,lastMove(none,0),[],lastMove(CUR,M),[]):-
26    getCurrentMillis(M).
27
28 savemove(CUR,lastMove(move(MV,SPEED,0),FIRSTM),L,lastMove
       (CUR,M),[move(MV,SPEED,DIFF,0)|L]):-
29    getCurrentMillis(M),
30    DIFF is M - FIRSTM.
31
32 %just to put the last command in the list
33 endSavemoves:-
34    savemove(executeInput(move(h,100,1000))).
35
36 backToBase:-
37    moveList(L),
38    backToBase(L).
39
40 backToBase([]).
41
42 backToBase([H|T]):-
43    revMove(H,RH),
44    executeInput(RH),
45    backToBase(T).
46
47 getCurrentMillis(M):-
48    class("it.unibo.sartiballanti.utils.Utils") <-
          getCurrentTimeMillis returns M.
49
50 revMove(move(mf,X,Y,Z),move(mb,X,Y,Z)).
51 revMove(move(mb,X,Y,Z),move(mf,X,Y,Z)).
52 revMove(move(mr,X,Y,Z),move(ml,X,Y,Z)).
53 revMove(move(ml,X,Y,Z),move(mr,X,Y,Z)).
54 revMove(move(h,X,Y,Z),move(h,X,Y,Z)).
55
56 %For example taking a photo
57 detection(X):-
58    actorOp(takeStringifiedPhoto),
59    actorOpDone(takeStringifiedPhoto,X).
60
61 initDriveRobotTheory.
62
63 :- initialization(initDriveRobotTheory).
```

10 Testing

In the previous sections, we had an executable model that could be tested, so most of the tests that involve communication between parts of the system have been done at the end of the analysis phase. Initially, communication tests have been executed locally, then the system parts have been deployed on different computational nodes to test system behaviour as a whole, checking if the system behaved as described in the test plan.

11 Deployment

The parts of the application will be deployed on different computational nodes as JAR executable archives with some configuration files and Prolog theories. The platforms we use in this case are a Raspberry Pi board and two PCs, we just need to copy the appropriate files and execute the JAR on every node. The application will start when all the parts of the system have been started.

12 Maintenance

We developed the application using the ddr framework and delaying any technological hypothesis, so the resulting system can be easily modified to add or change features. In the next section, we'll show how new (compatible) requirements need very little changes to the previously developed system.

13 Step 2

13.1 Requirements

STEP 2 (Implementation Optional)
Extend the last requirement as follows:

- If the bag is qualifed as "harmful", the Airport Security Center emits an 'alarm' signal and activates another (properly equipped) robot that (starting from the same RBA of the robot inspector) should reach the bag in autonomous way and remove the bag from the area.

13.2 Requirements analysis

Use cases No new use cases or changes to the previous ones.

Scenarios No new scenarios or changes to the previous ones.

(Domain) model A new robot is introduced, similar to the previous one. It needs a way to remove the bomb. We just need to add the following to the previous domain model:

```
RobotSystem extensionanalysis

Event alarm : alarm

Context ctxLocal ip[ host="localhost" port=8025 ]

Robot mock QActor removerobot context ctxLocal {
  Plan init normal
    println("Removerobot starts");
    switchToPlan waitAlarm

  Plan waitAlarm
    sense time(60000) alarm -> goToBag;
    repeatPlan 0

  Plan goToBag
    println("Going to bag");
    delay time(5000);
    println("Removing Bag");
    delay time(5000);
    println("Removerobot ends")
}
```

Test plan We need to check if the second robot receives can reach the bomb and remove it. We can test this observing the whole system when the ASC emits the alarm.

13.3 Problem analysis

Logic architecture We can obtain the new logic architecture adding a new type of message, the new actor described in the domain model and slightly changing the behavior of the first robot.
The first robot will send the route to the bag to the second robot before the detection phase. The second robot will follow this route to reach the bag and remove it if an alarm is emitted.

```
RobotSystem extensionlogicarchitecture

Event alarm : alarm
Dispatch routeToBag : routeToBag(X)    //sent by the
    driverobot when the bag is found
```

```
5
6  Context ctxRemoverobot ip[ host="localhost" port=8025 ]
7
8  Robot mock QActor removerobot context ctxRemoverobot {
9    Plan init normal
10     println("Removerobot starts");
11     switchToPlan receiveRoute
12
13   Plan receiveRoute
14     receiveMsg time(60000);
15     onMsg routeToBag : routeToBag(X) -> switchToPlan
           waitAlarm;
16     repeatPlan 0
17
18   Plan waitAlarm
19     println("Waiting for alarm");
20     sense time(60000) alarm -> goToBag;
21     repeatPlan 0
22
23   Plan goToBag
24     println("Going to bag");
25     delay time(5000);
26     println("Removing Bag");
27     delay time(5000);
28     println("Removerobot ends")
29 }
```

13.4 Work plan

We are using the ddr framework, so most of the behaviour of the new robot is already defined. We just need to implement an algorithm that allows the robot to follow the route it received from the other robot and an algorithm to remove the bomb.

13.5 Project

Structure

Interaction

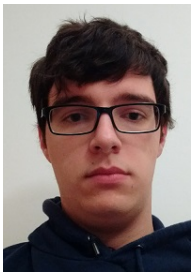
Behavior

See [1] until page 11 (`CMM`) and pages 96-105.

14 Information about the author

Paolo Sarti

Marcello Ballanti



References

1. A. Natali and A. Molesini. *Costruire sistemi software: dai modelli al codice.* Esculapio, 2009.