Final Theme System process report

Gruppo 3 Aimi Niccoló, Gallegati Mattia, Murgia Antonio, Zanotti Andrea

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
niccolo.aimi@studio.unibo.it; mattia.gallegati2@studio.unibo.it;
antonio.murgia2@studio.unibo.it; andrea.zanotti9@studio.unibo.it

1 Introduction

In this document we'll describe and discuss the process that led to the development of a robotic software system. We'll discuss how traditional software/project development techniques can and should merge with the scrum project management system.
We'll face the limits of traditional analysis approaches and purpose novel ones that better match with new programming paradigms.

2 Vision

Here we're collecting some visions that will inspire the development of this project and of software in general:

- There's no code without project, there's no project without problem analysis and there's no problem without requirements;
- There's no code without tests. That means that tests must be developed before the development of the software product;
- The team that develops the tests should be different from the one who will realize the project;
- We should use a top down approach during the development phase and bottom-up approach during the implementation phase;
- Zooming? Trying to get closer.. We should look at things initially from a far point of view like a black box and then trying to get closer to the details of the system, white box.
- Any sufficiently advanced technology is indistinguishable from magic.
- Develop the code in an IOT prospective using the ICT principles.
- Create valuable software using less time and resource possible in order to optimize software development process (factorization).
- A feature does not exist unless a test validates that it functions.
- Software entities should be open for extension but closed for modification.

– We should always check the existence of prior projects, trends or patterns regarding the technological domain we are facing. If does exist something we should study it, not only to take advantage of it during the development but also to recognize its limits and to purpose some innovative approach coming from different technological domains;
– The development of the project should be technology agnostic;
– We should find or create a formal language able to describe the results of the problem analysis. It should be similar to spoken language. This language could take advantage of different programming paradigms (functional, declarative and so on) and different programming models (actor model, message passing model and so on). This language should not be tied to any technology implementation. There should be one or more parsers/compilers for this language that will generate source code for a specific platform. This generated code should be used as the skeleton of the real product.

3 Goals

The first goal is to develop a robotic software system using the most advanced programming paradigms.
A secondary goal is to develop and to take advantage of a novel technique of requirements and problem analysis that will lead to a formal representation of the problem and to source code generation. Finally the main goal is discuss how some change (monotonic extensions) of the requirements impact on a product whose production is based on 'formal' and 'technology independent' artifacts rather than on ad-hoc code.

4 Requirements

Design and build a (prototype of a) software system that, with reference to a differential drive robot (called from now on robot):

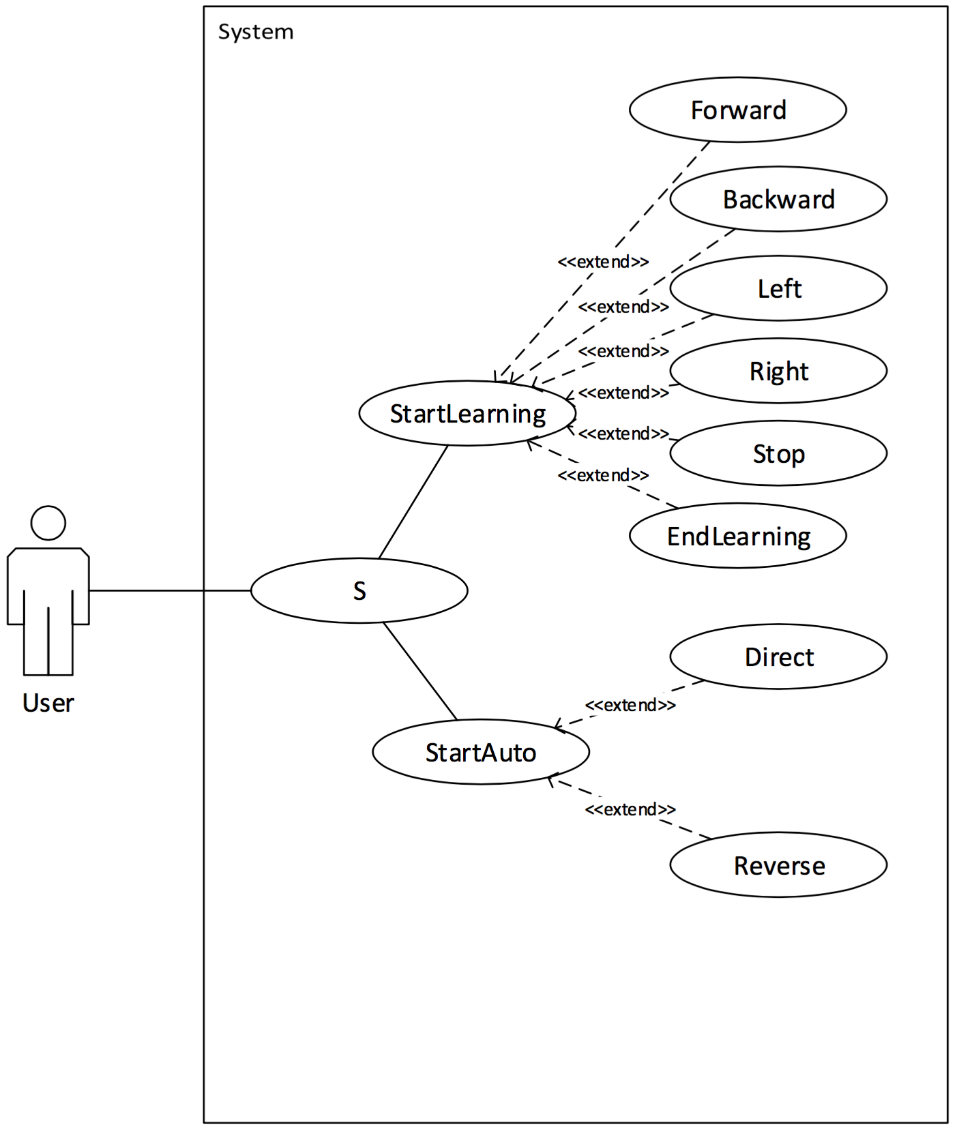
– allows a user to select between a 'learning phase' and a 'autonomous phase' during the learning phase, the user can send a sequence of move commands (e.g. forward, backward, left right, stop) to the robot. The robot must not only execute each command but it must also record the whole sequence of commands until the user decides to terminate the learning phase;
– after the termination of the learning phase, the user can tell the robot to enter the autonomous phase in a 'direct' or in a 'reverse' mode. During this phase the robot executes in autonomous way the sequence of moves it has learned, by complementing each move (e.g. forward->backward) if the selected mode is reverse;
– during the autonomous phase, the robot must be able to execute (as soon as possible) a stop command sent by the user.

After the development of this prototype, consider the possibility to enhance the functional capabilities of the robot, by allowing it:

– to perceive an obstacle during the autonomous phase and, once the obstacle is detected, to execute some alternative behaviour (in term of moves).

5 Requirement analysis

5.1 Use cases



5.2 Scenarios

Title	Execution of a sequence of actions by the robot
Description	The robot will repeat a sequence of actions previously recorded.
Relationships	“Reference to the previous use case”
Actors	User
Pre-Conditions	• User must be connected to the system via a user interface. • Robot must be “powered on” and connected to the network.
Post-Conditions	• The robot must be in an initial state where it can record new sequence of actions or repeat the recorded once.
Principle Scenario	1. The user can choose “StartLearning” to record a new sequence of actions on the robot, or “StartAuto” to let the robot execute the recorded sequence. This time we choose “StartLearning”. 2. The user commands the actions by the set of the allowed ones: Forward, Backward, Left, Right and Stop (Learning phase). When the user decides to finish this phase, he must choose “EndLearning”. 3. Now the user can choose again “StartLearning” to record a new sequence or “StartAuto”, this time we choose the last one (Autonomous phase). 4. The user can select “Direct”, so the robot will execute the actions in the same sequence of the user recorded ones. 5. At this point, the robot knows the sequence of actions saved in the learning phase, so in autonomous phase it replays those actions (in both directions) everytime the user wants, until a new learning phase is chosen.
Alternative Scenario	4. The user can choose “Reverse”: in this mode the robot will replay the sequence in a reversed way.
Open points	• Is it possible to add new actions in learning phase once it is finished?

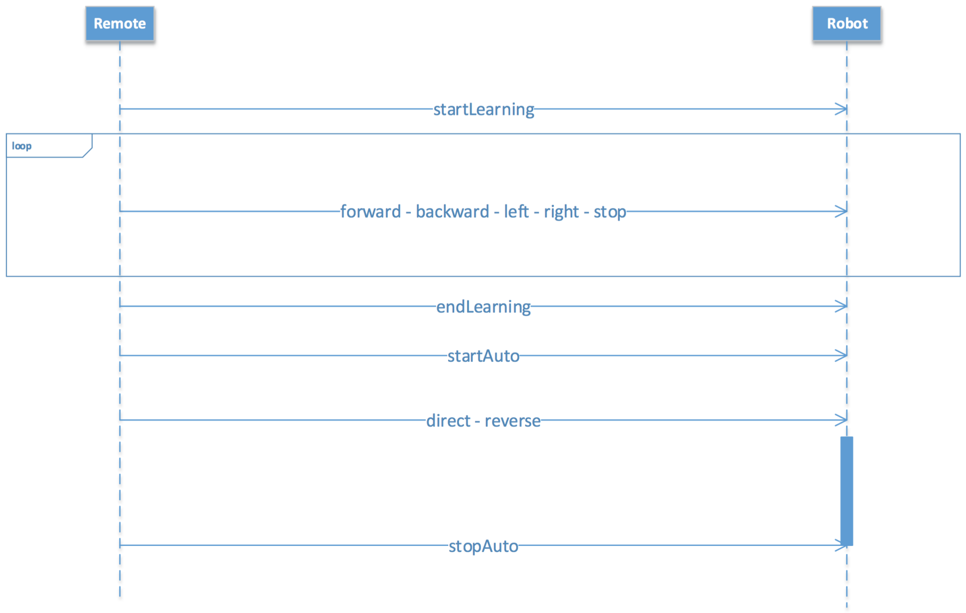
5.3 (Domain)model

Structure

ROBOT We consider the robot as a reactive and atomic entity.

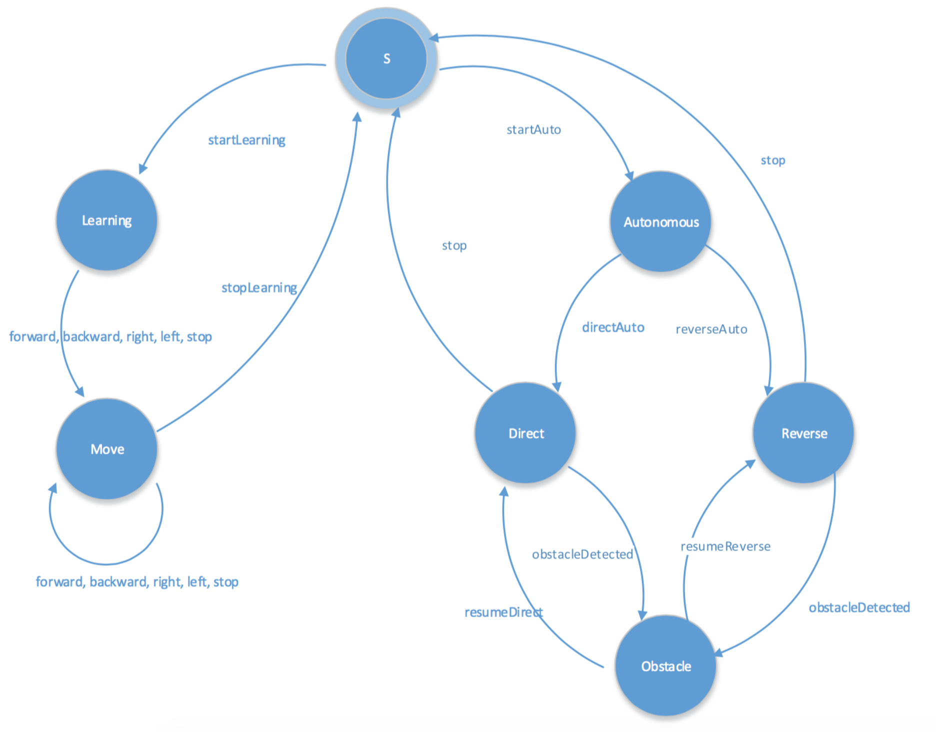
REMOTE We consider the remote as a proactive and atomic entity.

Interaction The interaction between the entities of the system is described with this sequence diagram:

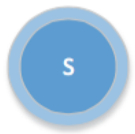


Behaviour

ROBOT The behaviour of the robot is described with this FSM:



REMOTE The behaviour of the remote is described with this FSM:



5.4 Test plan

Explaining test plan of a physical system and at this high-level of abstraction in a formal way is still an open point to us. It is not possible to write JUnit tests (that can be reused in the testing process) at this point of the project. Because of the lack of a formal way to define test plans in this chapter we will describe our test plan as natural language:

– during the learning phase we check that the robot must react to movement commands;

– after the learning phase we check that the recorded actions are the same as the previously executed ones;
– we check that the system is responsive only to the meaningful events in every possible state;

6 Problem analysis

Our technological hypothesis is JAVA and Object Oriented Programming. We will show that we need to develop a new infrastructure because the only features offered by the programming language chosen are not enough. To keep our code well developed, scalable, efficient and maintainable, we will use the most known development patterns. We must define an interaction language, that needs a communication standard. The communication standard will be essential during the Integration Test.

6.1 Abstraction gap

During the problem analysis we considered Java as our technological hypotesis. We realized that Java's unique communication tool is procedure calls, with OO paradigm it becomes really difficult to implement modular communication where entities are completely independent and loosely coupled. We need a new way to let our components interact. In agreement with our vision, we understand that the best way to overcome this problem is to develop a new software infrastructure that will map more strictly to our model representation than Java paradigms. This infrastructure will be used not only in this project, but every future developed process that will have the same needs because, according to our visions, we want to develop reusable code.
Our platform must offer some functionalities. First of all we need a formal definition of a System:
a System is one or more active entities that interact to reach some goal or provide some functionalities. A system is composed by one or more Contexts, a Contexts identifies a logical location at deployment time. The System enables the communication between the entities and synchronizes them so Entities must interact using the System only. In this way a System can be concentrated or distributed seamlessly for the application designer. Defined this general perspective we can introduce two paradigms:

– Message Based Programming
– Event Based Programming

Message Based Programming To introduce message based programming we need to define the entities that can exchange messages. We will call these entities QActors in reference to Erlang's Actors.
QActors in our platform are not message-driven but message-based. The core

difference between message driven and message based is that message driven systems can be only reactive to messages instead of being able to decide when messages and which messages are functional to evaluate. In this way we can model not only reactive entities but also proactive ones. QActors live in a context and they can communicate with other QActors using different communication primitives:

– dispatch - an asynchronous message without returning information;
– request - a message with returning information, can be executed as synchronous or asynchronous.

QActors can receive messages using the receive message primitives:

– receiveMessage - extracts the first message from the actor message queue;
– receiveTheMessage - extracts the first message that matches a pattern given by the application designer.

Event Based Programming We needed to define the concept of events as pervasive messages that are spread in the system. The message passing paradigm lacks of this feature because all messages are point to point. Events are a form of asynchronous communication that is not point to point. In our system our entities can declare to be interested in an event, when that event will be emitted the entities can react in some way.
When we introduced proactive QActors we realized that we needed the concept of Plan. A Plan is a sequence of actions (you can see actions as programming language instructions), every plan has its own logic and can switch to a different plan. When the execution of a plan reaches its end it can specify if the previous plan must continue its execution or suspend it. This abstraction was lacking of the better part of message/event driven programming: reactivity. This gap is filled by Asynchronous Actions. Asynchronous Actions are tipically time consuming actions that can be executed in blocking or non-blocking way. When executing an asynchronous action its execution can be interrupted by specified events and the QActor must react executing the associated plan. The logic of plan switching is defined as above. QActors are able to execute actions in two ways:

– executing compiled code
– interpreting meta code on the fly

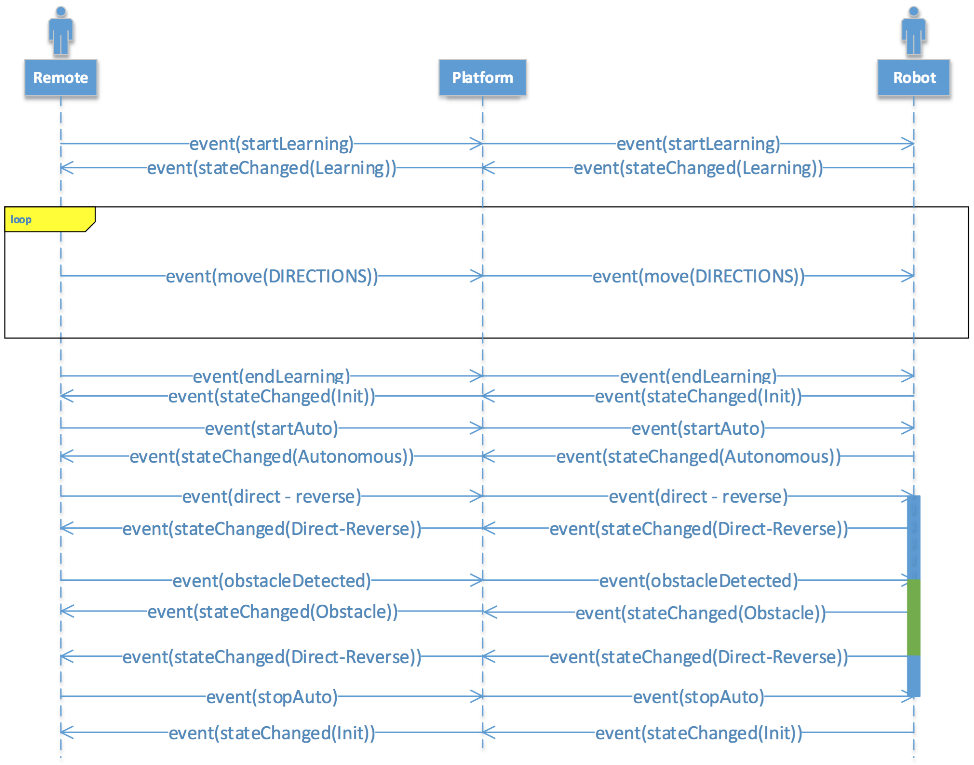
Adding the possibility to interpret code on the fly permits to the QActor to enrich its behaviour during runtime. The just described platform will be realized with Java Programming Language (our technologic assumption) but will be made to be interoperable with other technologies by the extensive use of text-based messages using the standard socket technology.
On top of the API offered by the platform we'll realize a DSL using the xtext framework. With the use of such tool we'll generate a declarative language that will provide two main benefits:

– a formal and human readable language to describe the problem analysis, entities interactions and system topology;
– code generation to be able to customize and execute the artifacts generated by the DSL interpretation.

With such tools in mind we can reimagine the traditionals problem analysis techniques and tackle the problems in a formal and more convenient way.

6.2 Logic architecture

The logic architecture is shown in the picture below:


Remote
Platform
Robot
event(startLearning)
event(stateChanged(Learning))
loop
event(move(DIRECTIONS))
event(endLearning)
event(stateChanged(Init))
event(startAuto)
event(stateChanged(Autonomous))
event(direct - reverse)
event(stateChanged(Direct-Reverse))
event(obstacleDetected)
event(stateChanged(Obstacle))
event(stateChanged(Direct-Reverse))
event(stopAuto)
event(stateChanged(Init))

```
System robot

//ChangeStateRobot
Event planChanged:planChanged(CURRENTSTATE)

//LearningPhase
Event moveOrEnd:moveOrEnd(DIRECTION)
```

```
//InitialPhase
Event start:start(PHASE)

//Auto Phase
Event mode:mode(AUTOORDIRECT)
Event obstacleDetected:obstacleDetected()
Event stop:stop()

Context  ctxRobot ip [ host="localhost" port=8037 ]

Context  ctxRemote ip [ host="localhost" port=8038 ]

EventHandler planChangedHandler for planChanged;

QActor qaRobot context ctxRobot {
        Plan init normal
                println("Robot hello");
                [!! start] getActivationEvent ev

        Plan learning
                println("learning phase");
                [!! moveOrEnd] dummyRobotMove
                //dummyrobotmove è una mossa di movimento
                // e aggiungere la regola in java


        Plan autonomous
                println("Autonomous phase")

        Plan obstacle
                println("Obstacle phase");
                delay time(3000);
                println("Obstacle destroyed");
                resumeLastPlan
}

QActor qaRemote context ctxRemote {
        Plan init normal
                println("Remote hello");
                emit moveOrEnd:moveOrEnd('X')
}
```

6.3 Risk analysis

The only remarkable risk in this development process is that the platform we will realize will be technology dependent to Java Runtime Environment. The

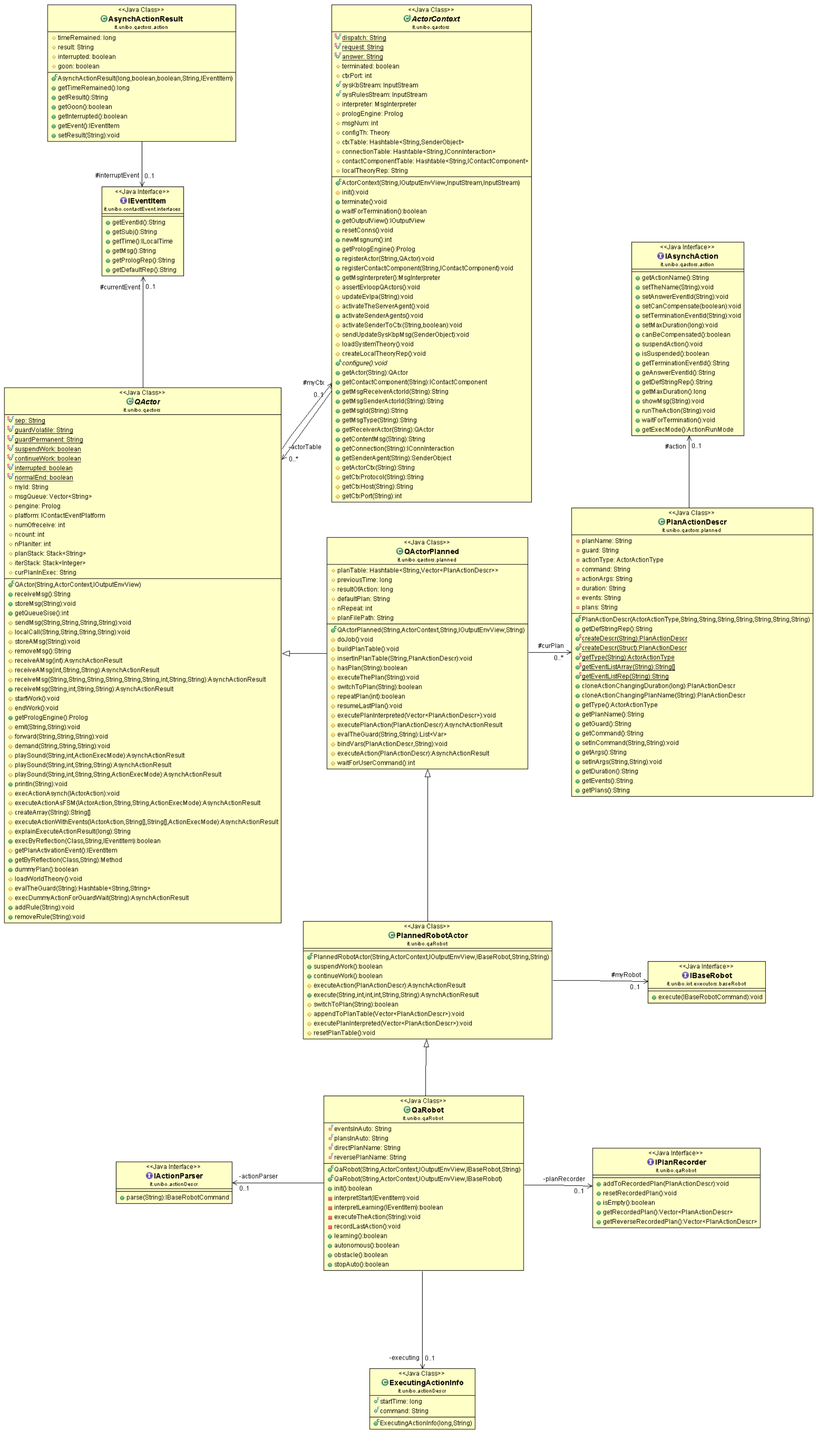
DSL is completely technology agnostic but the code generation feature will be bounded to the platform's API written in Java Language. If the deployment environment can't take advantage of the Java Platform our development process will be slower because we can't make use of the developed platform.

7 Work plan

Our team is composed by four members Aimi Niccoló , Gallegati Mattia, Murgia Antonio e Zanotti Andrea. Each of them has its own abilities. We decided to split the work between them relying on the personal experience of each member.
Aimi Niccoló : Senior model analyst, he will take care of the Requirement Analysis and Problem Analysis (with Gallegati).
Gallegati Mattia: Senior IT consultant will take care of Problem Analysis and of filling in the abstraction gap.
Murgia: Senior code developer will take care of the Implementation process and graphical interface realization.
Zanotti: Senior IT developer will take care of the Tests Plans, Tests implementation and Mantainance process. In order to keep the team cohesive every part of the development process will be led by the designed manager but every member of the team will take part to three daily meetings (duration : 30 min) where everyone will discuss shortly the progress of its work and will ask other team member opinions about some topics.
The reason why chose this kind of approach is that we believe that every member can look at the problem with its ?own eyes? and although the little experience in the other development phase can still express some interesting ideas for that phase that will lead to rapid solution of the problems.

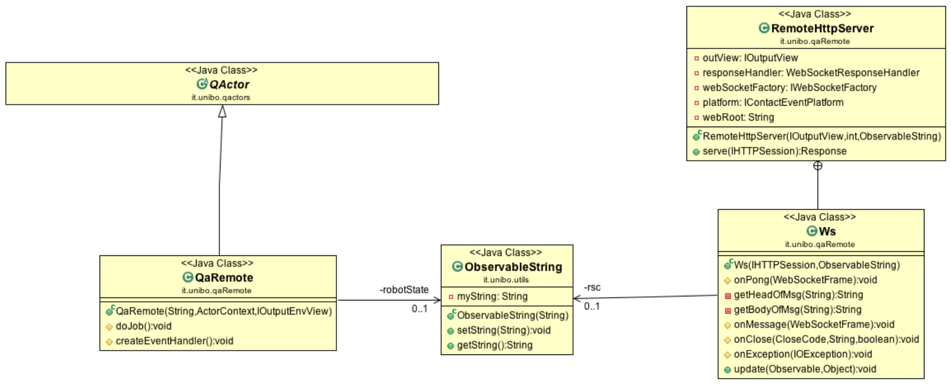
8 Project

Project architecture of the QaRobot is shown in the figure below:



<<Java Class>>
AsynchActionResult
it.unibo.qactors.action
timeRemained: long
result: String
interrupted: boolean
goon: boolean
AsynchActionResult(long,boolean,boolean,String,IEventItem)
getTimeRemained():long
getResult():String
getGoon():boolean
getInterrupted():boolean
getEvent():IEventItem
setResult(String):void
#interruptEvent 0..1
<<Java Interface>>
IEventItem
it.unibo.contactEvent.interfaces
getEventId():String
getSubj():String
getTime():ILocalTime
getMsg():String
getPrologRep():String
getDefaultRep():String
#currentEvent 0..1
<<Java Class>>
ActorContext
it.unibo.qactors
dispatch: String
request: String
answer: String
terminated: boolean
ctxPort: int
sysKbStream: InputStream
sysRulesStream: InputStream
interpreter: MsgInterpreter
prologEngine: Prolog
msgNum: int
configTh: Theory
ctxTable: Hashtable<String,SenderObject>
connectionTable: Hashtable<String,IConnInteraction>
contactComponentTable: Hashtable<String,IContactComponent>
localTheoryRep: String
ActorContext(String,IOutputEnvView,InputStream,InputStream)
init():void
terminate():void
waitForTermination():boolean
getOutputView():IOutputView
resetConns():void
newMsgnum():int
getPrologEngine():Prolog
registerActor(String,QActor):void
registerContactComponent(String,IContactComponent):void
getMsgInterpreter():MsgInterpreter
assertEvloopQActors():void
updateEvlpa(String):void
activateTheServerAgent():void
activateSenderAgents():void
activateSenderToCtx(String,boolean):void
sendUpdateSysKbpMsg(SenderObject):void
loadSystemTheory():void
createLocalTheoryRep():void
configure():void
getActor(String):QActor
getContactComponent(String):IContactComponent
getMsgReceiverActorId(String):String
getMsgSenderActorId(String):String
getMsgId(String):String
getMsgType(String):String
getReceiverActor(String):QActor
getContentMsg(String):String
getConnection(String):IConnInteraction
getSenderAgent(String):SenderObject
getActorCtx(String):String
getCtxProtocol(String):String
getCtxHost(String):String
getCtxPort(String):int
#myCtx 0..1
-actorTable 0..*
<<Java Interface>>
IAsynchAction
it.unibo.qactors.action
getActionName():String
setTheName(String):void
setAnswerEventId(String):void
setCanCompensate(boolean):void
setTerminationEventId(String):void
setMaxDuration(long):void
canBeCompensated():boolean
suspendAction():void
isSuspended():boolean
getTerminationEventId():String
geAnswerEventId():String
getDefStringRep():String
getMaxDuration():long
showMsg(String):void
runTheAction(String):void
waitForTermination():void
getExecMode():ActionRunMode
#action 0..1
<<Java Class>>
QActor
it.unibo.qactors
sep: String
guardVolatile: String
guardPermanent: String
suspendWork: boolean
continueWork: boolean
interrupted: boolean
normalEnd: boolean
myId: String
msgQueue: Vector<String>
pengine: Prolog
platform: IContactEventPlatform
numOfreceive: int
ncount: int
nPlanIter: int
planStack: Stack<String>
iterStack: Stack<Integer>
curPlanInExec: String
QActor(String,ActorContext,IOutputEnvView)
receiveMsg():String
storeMsg(String):void
getQueueSise():int
sendMsg(String,String,String,String):void
localCall(String,String,String,String):void
storeAMsg(String):void
removeMsg():String
receiveAMsg(int):AsynchActionResult
receiveAMsg(int,String,String):AsynchActionResult
receiveMsg(String,String,String,String,String,String,int,String,String):AsynchActionResult
receiveMsg(String,int,String,String):AsynchActionResult
startWork():void
endWork():void
getPrologEngine():Prolog
emit(String,String):void
forward(String,String,String):void
demand(String,String,String):void
playSound(String,int,ActionExecMode):AsynchActionResult
playSound(String,int,String,String):AsynchActionResult
playSound(String,int,String,String,ActionExecMode):AsynchActionResult
println(String):void
execActionAsynch(IActorAction):void
executeActionAsFSM(IActorAction,String,String,ActionExecMode):AsynchActionResult
createArray(String):String[]
executeActionWithEvents(IActorAction,String[],String[],ActionExecMode):AsynchActionResult
explainExecuteActionResult(long):String
execByReflection(Class,String,IEventItem):boolean
getPlanActivationEvent():IEventItem
getByReflection(Class,String):Method
dummyPlan():boolean
loadWorldTheory():void
evalTheGuard(String):Hashtable<String,String>
execDummyActionForGuardWait(String):AsynchActionResult
addRule(String):void
removeRule(String):void
<<Java Class>>
QActorPlanned
it.unibo.qactors.planned
planTable: Hashtable<String,Vector<PlanActionDescr>>
previousTime: long
resultOfAction: long
defaultPlan: String
nRepeat: int
planFilePath: String
QActorPlanned(String,ActorContext,String,IOutputEnvView,String)
doJob():void
buildPlanTable():void
insertInPlanTable(String,PlanActionDescr):void
hasPlan(String):boolean
executeThePlan(String):void
switchToPlan(String):boolean
repeatPlan(int):boolean
resumeLastPlan():void
executePlanInterpreted(Vector<PlanActionDescr>):void
executePlanAction(PlanActionDescr):AsynchActionResult
evalTheGuard(String,String):List<Var>
bindVars(PlanActionDescr,String):void
executeAction(PlanActionDescr):AsynchActionResult
waitForUserCommand():int
#curPlan 0..*
<<Java Class>>
PlanActionDescr
it.unibo.qactors.planned
planName: String
guard: String
actionType: ActorActionType
command: String
actionArgs: String
duration: String
events: String
plans: String
PlanActionDescr(ActorActionType,String,String,String,String,String,String,String)
getDefStringRep():String
createDescr(String):PlanActionDescr
createDescr(Struct):PlanActionDescr
getType(String):ActorActionType
getEventListArray(String):String[]
getEventListRep(String):String
cloneActionChangingDuration(long):PlanActionDescr
cloneActionChangingPlanName(String):PlanActionDescr
getType():ActorActionType
getPlanName():String
getGuard():String
getCommand():String
setInCommand(String,String):void
getArgs():String
setInArgs(String,String):void
getDuration():String
getEvents():String
getPlans():String
<<Java Class>>
PlannedRobotActor
it.unibo.qaRobot
PlannedRobotActor(String,ActorContext,IOutputEnvView,IBaseRobot,String,String)
suspendWork():boolean
continueWork():boolean
executeAction(PlanActionDescr):AsynchActionResult
execute(String,int,int,int,String,String):AsynchActionResult
switchToPlan(String):boolean
appendToPlanTable(Vector<PlanActionDescr>):void
executePlanInterpreted(Vector<PlanActionDescr>):void
resetPlanTable():void
#myRobot 0..1
<<Java Interface>>
IBaseRobot
it.unibo.iot.executors.baseRobot
execute(IBaseRobotCommand):void
<<Java Class>>
QaRobot
it.unibo.qaRobot
eventsInAuto: String
plansInAuto: String
directPlanName: String
reversePlanName: String
QaRobot(String,ActorContext,IOutputEnvView,IBaseRobot,String)
QaRobot(String,ActorContext,IOutputEnvView,IBaseRobot)
init():boolean
interpretStart(IEventItem):void
interpretLearning(IEventItem):boolean
executeTheAction(String):void
recordLastAction():void
learning():boolean
autonomous():boolean
obstacle():boolean
stopAuto():boolean
-actionParser 0..1
<<Java Interface>>
IActionParser
it.unibo.actionDescr
parse(String):IBaseRobotCommand
-planRecorder 0..1
<<Java Interface>>
IPlanRecorder
it.unibo.qaRobot
addToRecordedPlan(PlanActionDescr):void
resetRecordedPlan():void
isEmpty():boolean
getRecordedPlan():Vector<PlanActionDescr>
getReverseRecordedPlan():Vector<PlanActionDescr>
-executing 0..1
<<Java Class>>
ExecutingActionInfo
it.unibo.actionDescr
startTime: long
command: String
ExecutingActionInfo(long,String)

We extended the Planned class offered by the platform to embed robot functionalities realizing the PlannedRobotActor, we used the Builder pattern to instantiate the Robot basic object to issue commands to an implementation of the robot (Mock or Real robot). We also overloaded the switchToPlan, suspendWork and continueWork of QActorPlanned in order to emit an event every time a plan is changed. Project architecture of the QaRemote is shown in the figure below:

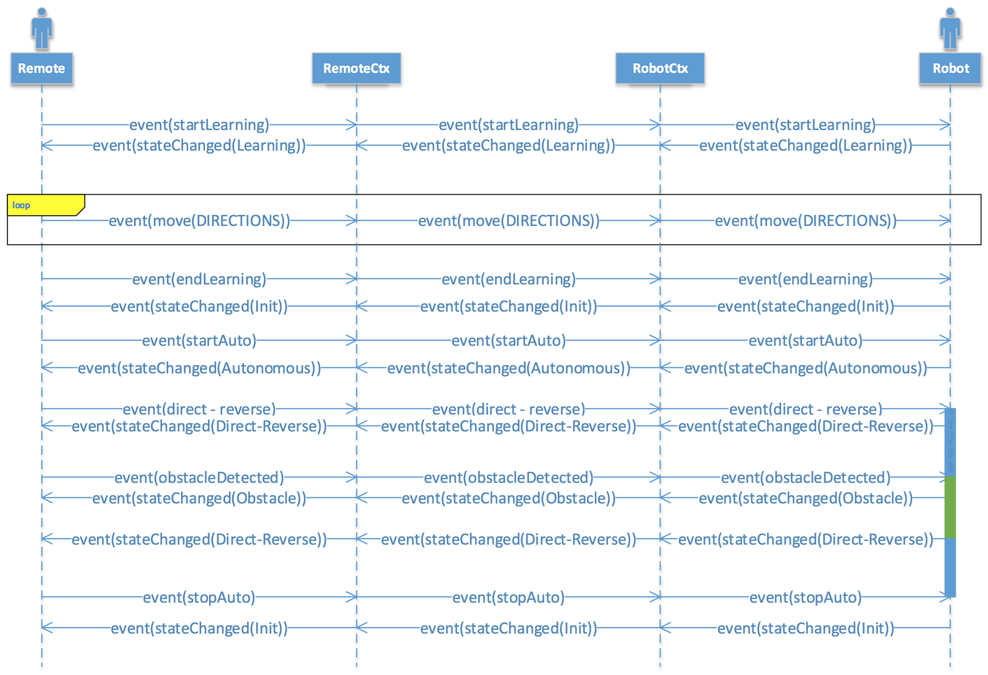

<<Java Class>>
QActor
it.unibo.qactors
<<Java Class>>
RemoteHttpServer
it.unibo.qaRemote
outView: IOutputView
responseHandler: WebSocketResponseHandler
webSocketFactory: IWebSocketFactory
platform: IContactEventPlatform
webRoot: String
RemoteHttpServer(IOutputView,int,ObservableString)
serve(IHTTPSession):Response
<<Java Class>>
QaRemote
it.unibo.qaRemote
QaRemote(String,ActorContext,IOutputEnvView)
doJob():void
createEventHandler():void
-robotState
0..1
<<Java Class>>
ObservableString
it.unibo.utils
myString: String
ObservableString(String)
setString(String):void
getString():String
-rsc
0..1
<<Java Class>>
Ws
it.unibo.qaRemote
Ws(IHTTPSession,ObservableString)
onPong(WebSocketFrame):void
getHeadOfMsg(String):String
getBodyOfMsg(String):String
onMessage(WebSocketFrame):void
onClose(CloseCode,String,boolean):void
onException(IOException):void
update(Observable,Object):void

In remote implementation we used Observer pattern to let the remote be aware of the state of the robot in order to configure the GUI to show the right buttons.

8.1 Structure

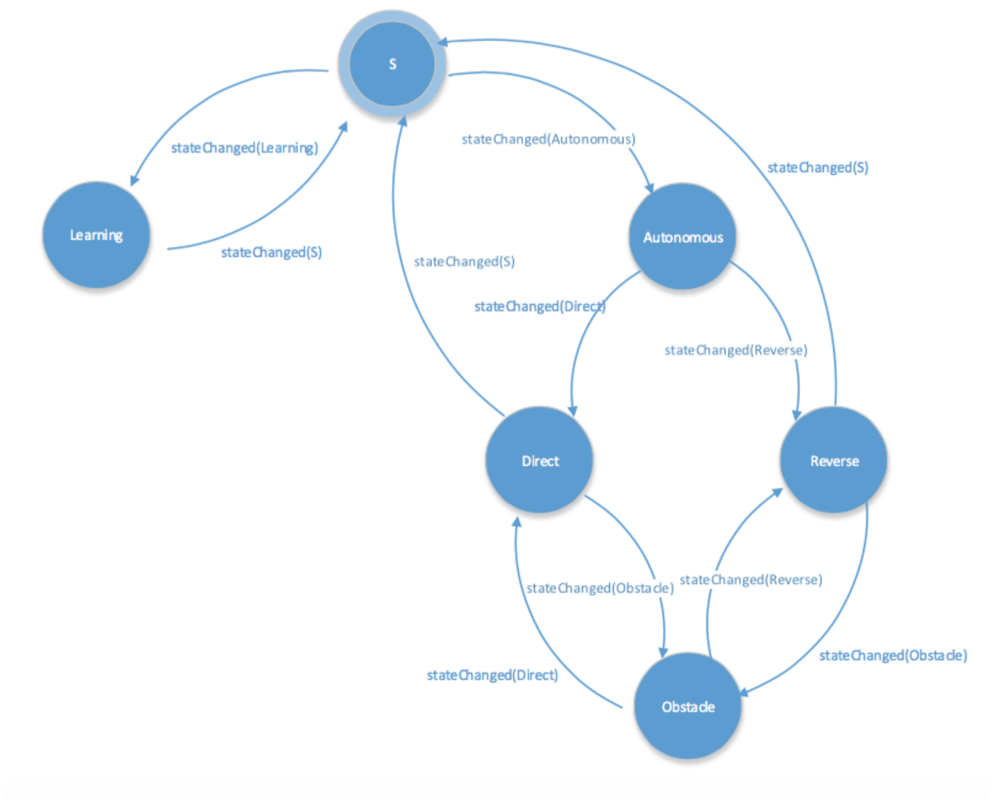
According to the defined abstraction that led to our platform the model structure remains the same as the analysis one with the only difference that the communication is mediated by the contexts where the QActors are living in.

8.2 Interaction


Remote
RemoteCtx
RobotCtx
Robot
event(startLearning)
event(stateChanged(Learning))
loop
event(move(DIRECTIONS))
event(endLearning)
event(stateChanged(Init))
event(startAuto)
event(stateChanged(Autonomous))
event(direct - reverse)
event(stateChanged(Direct-Reverse))
event(obstacleDetected)
event(stateChanged(Obstacle))
event(stateChanged(Direct-Reverse))
event(stopAuto)
event(stateChanged(Init))

8.3 Behavior

Taking advantage of the definitions of plans formalized during the abstraction gap chapter, we can realize a software artifact that is exactly the FSM formalized during the analysis step.
Regarding the Remote we realized that having a unresponsive remote (like the one of a Television) would be restrictive, so we realized a responsive GUI that enables only the commands permitted in that phase. To do that we modeled the Remote as a FSM that reacts to the state change of the robot.



9 Implementation

The GUI part of the remote is implemented as a web page and the interaction between the client browser and the remote is realized by web socket technology. In this way the lightweight and the frequent interactions doesn't pay the cost of a new connection every time a button is pressed and the communication has not to be initiated by the client (despite of AJAX that needs to create a new connection every time).

10 Testing

As previously said we cannot really test the functionalities of the robot via code, but we can send command and observe the behaviour. We can also send to the robot the events he must react to and analyze the state where he is. So for example we can emit the event StartLearning and look if the robot is in the right state.

Also the Platform must be checked, Unit testing is already developed inside the QActor Platform. Here we report an example of functional and interaction testing of the developed system:

```
public class InteractionTest {
  public static void main(String[] args){
    emit(Start.prefix ,Start.body(Start.Phases.LEARNING));
        execDummyActionForGuardWait( "PlanChanged" );
        IEventItem ev = this.getPlanActivationEvent();
        assert(ev.getMsg().equals(PlanChanged.body(Plans.LEARNINGPLAN)));

        emit(MoveOrEnd.prefix ,MoveOrEnd.body(MoveOrEnd.Movements.FORWARD));
        emit(MoveOrEnd.prefix ,MoveOrEnd.body(MoveOrEnd.Movements.LEFT));
        emit(MoveOrEnd.prefix ,MoveOrEnd.body(MoveOrEnd.Movements.BACKWARD));
        emit(MoveOrEnd.prefix ,MoveOrEnd.body(MoveOrEnd.Movements.ENDLEARNING));
        execDummyActionForGuardWait( "PlanChanged" );
        IEventItem ev = this.getPlanActivationEvent();
        assert(ev.getMsg().equals(PlanChanged.body(Plans.INITPLAN)));

        emit(Start.prefix ,Start.body(Start.Phases.AUTONOMOUS));
        execDummyActionForGuardWait( "PlanChanged" );
        IEventItem ev = this.getPlanActivationEvent();
        assert(ev.getMsg().equals(PlanChanged.body(Plans.AUTONOMOUSPLAN)));

        emit(Mode.prefix ,Mode.body(Modes.DIRECT));
        execDummyActionForGuardWait( "PlanChanged" );
        IEventItem ev = this.getPlanActivationEvent();
        assert(ev.getMsg().equals(PlanChanged.body(Plans.DIRECTPLAN)));

        execDummyActionForGuardWait( "PlanChanged" );
        IEventItem ev = this.getPlanActivationEvent();
        assert(ev.getMsg().equals(PlanChanged.body(Plans.INIT)));
  }
}
```

11 Deployment

To deploy the developed system we decided to use Raspberry Pi as main board for the robot equipped with Java and Pi4j library where we will load the developed code as Java Jar file. For the remote controller we chose to develop the user interface as a web page that will be accessible in a uniform way from mobile devices and desktop ones.
The remote web page will be accessible through the Raspberry Pi network connections (Ethernet and Wi-Fi).

12 Maintenance

Monotonic Extension We will discuss how easy is to add functionalities to our software product. We said that in our platform we can express the model through a formal language that leads to a code generation process. This way adding functionalities is simply a matter of explaining them in our formal language. Infact we can describe all the entities in the system, their interactions and the data exchanged by those entities. In order to explain this concept we will go through the process that let us add a new functionality to our robot: -The new functionality is to let the Robot be reactive to Obstacle presence during the Autonomous Phase.

In order to do that we added a new event to our DSL file (from now on .qa file), then we added a plan that handles the event. Then we explicated that actions during the Autonomous Phase are sensible to the obstacle event reacting with the defined obstacle handling plan.

A Information about the author

Photo of Aimi Niccoló

Photo of Gallegati Mattia

Photo of Murgia Antonio

Photo of Zanotti Andrea