Final Task 2018 as a case study

Antonio Natali

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

Table of Contents

Final Task 2018 as a case study 1
Antonio Natali
1 Requirements 2
2 Starting 3
2.1 Basic questions 3
2.2 Technology (in)dependency 4
2.3 A product backlog 4
2.3.1 A first (essential) system 4
2.3.2 A virtual robot 5
2.3.3 A distributed system 5
2.3.4 More business logic 5
2.3.5 Observing 6
2.3.6 A floor map 7
2.3.7 Console 7
2.3.8 Agile development 7
2.4 Software already available 7
2.4.1 The `QActor` metamodel/language 7
2.5 Software deployment 8
3 A first architecture 9
3.1 The three-dimension space 9
3.1.1 Structure 9
3.1.2 Interaction 9
3.1.3 Behaviour 9
3.2 An executable formal model 10
3.2.1 The result 11
3.3 A first review 11
4 Beyond the first toy-model 12
4.1 Working with a virtual robot 12
4.1.1 A working model 12
4.2 Custom actions (`javaRun`) 13
4.3 Moving the robot 13
4.4 The final state 13
4.5 Automatic testing 14
5 Towards a distributed system 16
5.1 The consolesimulator 16
5.2 Simulate changes of the temperature 17
5.3 Executing the distributed system 17
6 More business logic 18
6.1 The mind and the player as actors 18
6.2 The robot-actuator 18
6.3 The robot-mind 19
6.4 From events to messages 20
6.4.1 Event handling 20
6.5 Consuming pending messages 20
6.6 The robot context 21

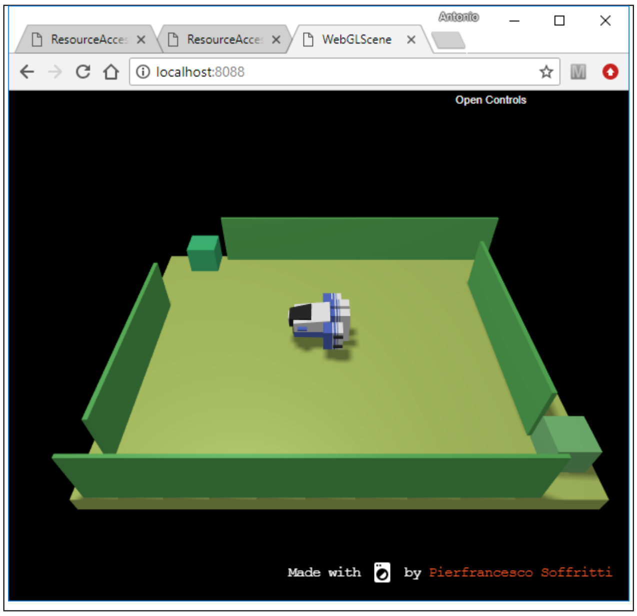
6.7 Reusing a (web) console . . . 21
6.7.1 The starter . . . 23
6.7.2 Console events . . . 23
6.7.3 Console messages . . . 23
6.7.4 Temperature provider . . . 24
7 Observing the environment . . . 25
7.1 The rule of the worldobserver . . . 25
7.2 The behavior of the worldobserver . . . 26
7.3 `LED` handling . . . 26
7.4 The new system (model1) . . . 27
7.5 Executing/testing the system . . . 28
8 The primary use-case ([R-floorClean]) . . . 30
8.1 Problem analysis . . . 30
8.2 Remarkable points . . . 33
8.3 A first logical architecture . . . 33
8.3.1 The mind and the player (as actors) . . . 33
8.4 Messages and events . . . 34
8.4.1 Message handling . . . 34
8.4.2 Event handling . . . 34
8.4.3 Messages or Events? . . . 34
8.5 Mind/player message-based interaction . . . 35
8.5.1 Types of message-based interaction . . . 35
8.5.2 The basicStep . . . 35
8.6 Planning and doing . . . 36
8.6.1 The planner . . . 36
8.6.2 The class aiutil . . . 36
8.6.3 The actuation . . . 37
8.6.4 The importance of (executable) models . . . 37
9 A first prototype (the main architecture) . . . 39
9.1 The (actor) knowledge-base . . . 39
9.2 The `robotmind` as a `FSM` . . . 40
9.3 Self-messaging . . . 40
9.4 The state doPlan . . . 40
9.5 The state doActions . . . 41
9.6 doActions as a dispatcher . . . 41
9.7 Declarative code . . . 42
9.8 waitForwardMoveAnswer . . . 42
9.9 actionDoneOk . . . 42
9.10 The current robot position . . . 42
9.11 Move history . . . 43
9.12 World models . . . 43
9.13 The 'reusable' state `nextMove` . . . 43
9.14 The actor `oncecellforward` with simulated moves . . . 44
9.14.1 Execute the simulated move . . . 45
9.14.2 Sending the answer . . . 45
9.15 About request-response . . . 46
9.16 From code to diagrams . . . 46
9.17 The final actor `oncecellforward` . . . 47
9.17.1 Doing the w-move . . . 48

9.17.2 endMoveForward . . . 48
9.17.3 checkMobileObstacle . . . 49
9.17.4 probableFixedObstacle . . . 49
9.17.5 sendAnswerAfterCollision . . . 50
9.18 Incremental design and testing . . . 50
9.18.1 Vocabulary . . . 50
9.18.2 Structure . . . 51
9.18.3 Behavior . . . 51
10 Designing, coding, testing . . . 52
10.0.4 Vocabulary . . . 52
10.0.5 Incremental design and testing . . . 52
10.0.6 First intent . . . 52
10.0.7 The path without obstacles . . . 53
10.0.8 Fixed, generic obstacles . . . 53
10.0.9 The walls . . . 53
10.0.10 Fixed, far Obstacles . . . 53
10.0.11 The global system . . . 54
55

1 Requirements

In a home of a given city (e.g. Bologna), a ddr robot - equipped with a sonar (sonar-robot) on its front - is used to clean the floor of a room (R-FloorClean).

The floor in the room is a flat floor of solid material and is equipped with two *sonars*, named sonar1 and sonar2 as shown in the picture (sonar1 is that at the top). The initial position (start-point) of the robot is detected by sonar1, while the final position (end-point) is detected by sonar2.



The robot works under the following conditions:

1. R-Start: an authorized user has sent a START command by using a human GUI interface (console) running on a conventional PC or on a smart device (Android).
2. R-TempOk: the value temperature of the city is not higher than a prefixed value (e.g. 25 degrees Celsius).
3. R-TimeOk: the current clock time is within a given interval (e.g. between 7 a.m and 10 a.m)

While the robot is working:

- it must blink a Led put on it, if the robot is a real robot (R-BlinkLed).
- it must blink a Led Hue Lamp available in the house, if the robot is a virtual robot (R-BlinkHue).
- it must avoid fixed obstacles (e.g. furniture) present in the room (R-AvoidFix) and/or mobile obstacles like balls, cats, etc. (R-AvoidMobile).

Moreover, the robot must stop its activity when one of the following conditions apply:

1. R-Stop: an authorized user has sent a STOP command by using the console.
2. R-TempKo: the value temperature of the city becomes higher than the prefixed value.
3. R-TimeKo: the current clock time is beyond the given interval.
4. R-Obstacle: the robot has found an obstacle that it is unable to avoid.
5. R-End: the robot has finished its work.

During its work, the robot can optionally:

- R-Map: build a map of the room floor with the position of the fixed obstacles. Once built, this map can be used to define a plan for an (optimal) path form the start-point to the end-point.

2 Starting

From the functional requirements we can state that:

1. Our software system is a `distributed system` composed by two main entities: a `console` running on a console-node that can be a `PC` or on a `SmartDevice` and a (real or virtual) `robot`, running on its own robot-node.

2. The `robot` is equipped with a sonar ('`sonar-robot`') put in front of it. Other sensors could be very useful for doing the work in efficient way, but at the moment it is excluded (for costs reason) the possibility to extend the sensory equipment of the robot. A real robot is also provided with a led.
3. The *software on the console-node* must allow the end-user to control the robot with very simple commands (e.g. `LOGIN`, `START/STOP`).
4. The *software on the robot-node* must execute the commands sent by the end-user via the `console` and must control the robot in doing its main task (`Use Case`, in `UML` terminology): floor-cleaning in autonomous way.
5. The robot control software must be sensible to the environment, with specific reference to the temperature (of the city), the current time, and the obstacles on the floor. In the following we will reference to all these conditions with the term `envConds`.
6. The environment includes two fixed sonar (named `sonarStart` and `sonarEnd`) that can be used to detect the staring point and the end point of the flat floor to clean.
7. The obstacles on the floors can be dynamically detected by using the `sonar-robot`, while the temperature and the time must be acquired in some way.
8. If the robot builds a map of the floor (an optional task), the fixed obstacles can be known in advance. Moreover, we can have a 'formal' mean to check the complete coverage of the floor to clean.

We can also say what follows.

1. The *business logic* of the system is mainly related to the software that implements the behaviour of the `robot`. Let us give the name `cleanerrobot` to this part, that must realize an '*autonomous*' system able to clean the floor while being able to `react` to obstacles or to events related to `envConds` (let us call, from now on, these events as '`invalidcondition-events`').
2. It can be useful/wise to partition the concept of the `cleanerrobot` in two parts: a `robot-mind` that implements the strategy required to solve the problem and a `robot-actuator` that simply executes commands to move the robot.
3. The responsibility of monitoring the environment and rising an `invalidcondition-event` can be given to the `robot-mind` or to some other, specialized entity, e.g. a `world-observer`.
4. The most relevant (and perhaps difficult) part of the business software seems related to the requirement `R-Map`. It is an optional requirement, but a first, rapid problem analysis suggests that the task of building a map of the room floor (with the position of the fixed obstacles) could be necessary to assure that the floor has been completely cleaned.

2.1 Basic questions

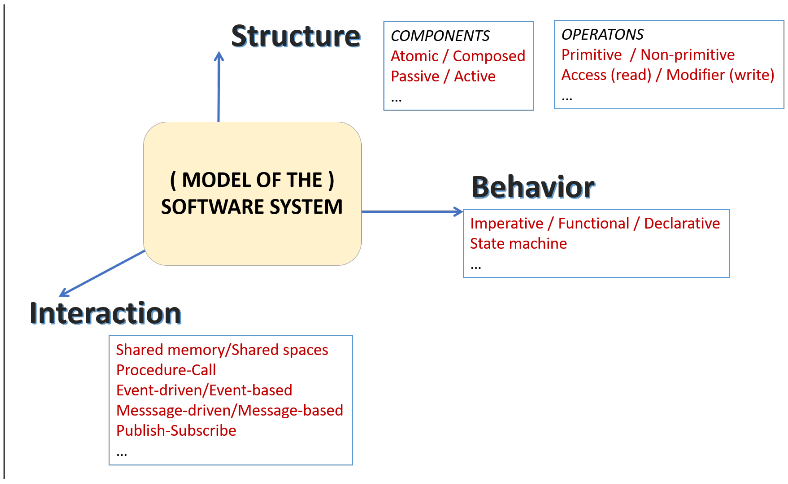
Let us recall a set of basic questions that we can pose to ourselves at the starting of each new project.

- What is our starting point? Do we start by what is already available (by following a `bottom-up` approach) or do we follow a `top-down` approach that first detects a logic architecture of the system and afterward selects the technology? See Subsection 2.2 and Subsection 2.4.
- Is it possible to organize the production in terms of a *sequence of systems*, each facing a more wide/complex set of requirements so that each system can be designed and built as the `evolution` of the previous one?
- Is it possible to order the set of requirements into a list of increasing complexity and to harmonize such a list with the product backlog defined by our product-owner? See Subsection 2.3.

2.2 Technology (in)dependency

In the first phase of our development, we want to be *technology-aware* but also as much *technology-independent* as possible. To achieve such a goal, we will start by building a `conceptual model` of the system. From https://en.wikipedia.org/wiki/Systems_modeling, we read:

> A model is a representation of a system, made of the composition of concepts which are used to help people know, understand, or simulate a subject the model represents.

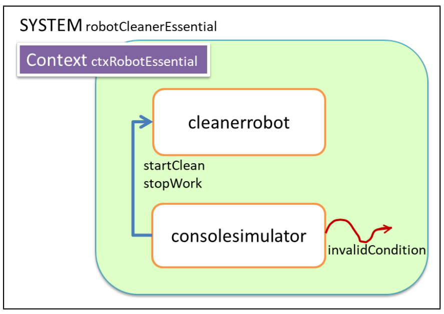


2.3 A product backlog

According to the `SCRUM` framework of software development, we will suppose that the `product backlog` defined by the `Product Owner` allows us to satisfy the set of requirements in *incremental way*, by distinguishing among a set of macro-steps. For example, we could define the sequence reported in the following sections.

2.3.1 A first (essential) system .

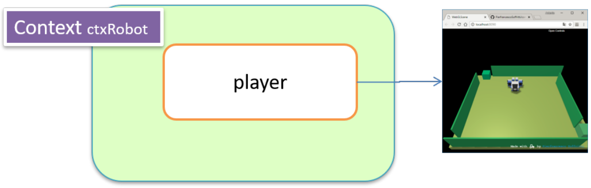
The first step defines the main components of the logic architecture of the systems, by focusing on their interaction.
Requirements: `R-Start`, `R-Stop`, `envCons(R-TempOk/R-TempKo)`
Reference: Section 3



In this phase, the behavior of each component should be simply 'simulated' in order to reproduce a set of interaction patterns, with the aim to introduce a proper set of initial (integration, functional) `Test-Plans`.

2.3.2 A virtual robot .

The second step moves towards a more realistic system. We start by introducing a virtual robot that operates into a simulated environment.
Reference: Section 4

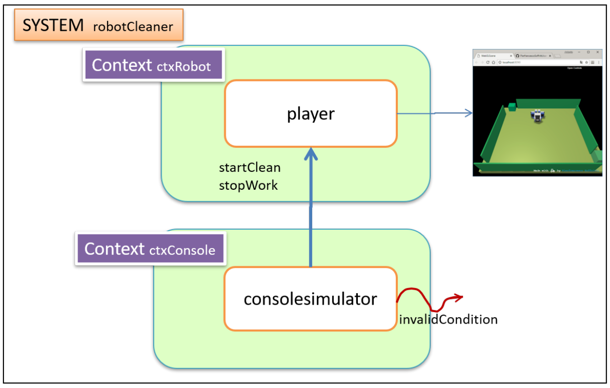


The introduction of a `player` is motivated by the pragmatic reason to encapsulate into a component all the (technological) details related to the interaction with robot, that could be real or virtual and built in very different ways.

An automated *Testing Activity* is also introduced.

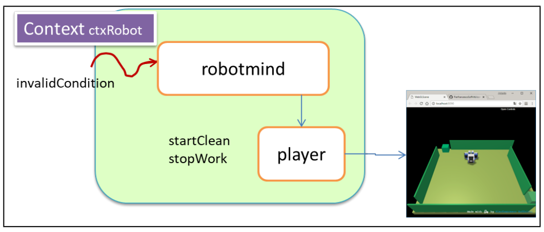
2.3.3 A distributed system .

Another contribution of our second step is to define a reference model for the logical architecture, by introducing a distributed system, in which the `player` and the `consolesimulator` run each on a different node .
Reference: Section 5

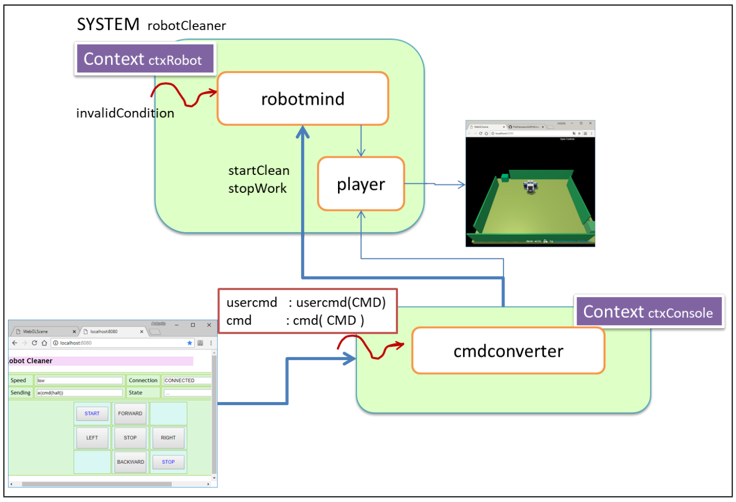


2.3.4 More business logic .

In the third step we will refine the logical architecture defined in the previous step by including the business logic into the `robot-mind`. Moreover, we intend to face the problem of `LED` handling.
Reference: Section 6
Requirements: `R-BlinkLed`, `R-BlinkLue`



In this step we will introduce also a more realistic console, by reusing software already available in our factory:

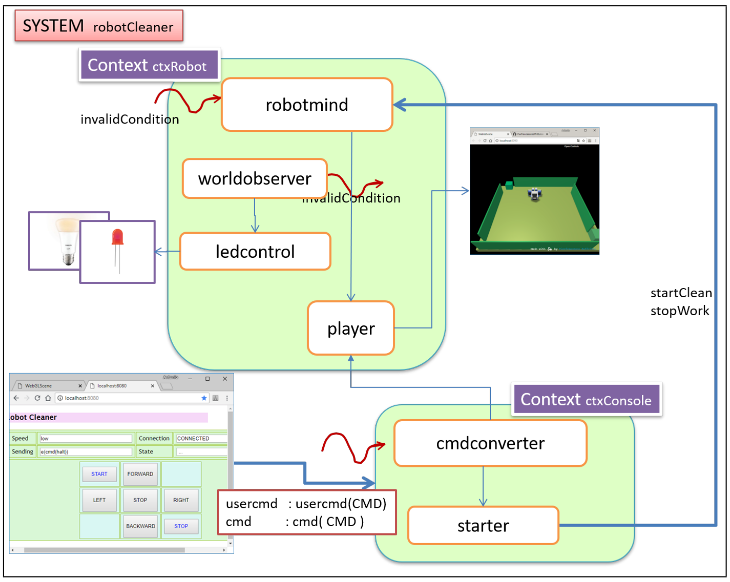


2.3.5 Observing .

In the fourth step we introduce a component that

– is able to monitor the events raised from the environment;
– builds a representation of the state of the world and of the robot;
– send messages to the actuators.

Reference: Section 7
Requirements: `Led (actuator) handling`



2.3.6 A floor map .

The fifth step is related to the creation of a floor map
Requirements: R-FloorClean
Reference: Section 8

2.3.7 Console .

The sixth step is related to the development of the console supporting user authentication
Requirements: authorized user
Reference: todo

2.3.8 Agile development .

Of course this plan is just a staring point. The sequence of the real SPRINTs, of their goals and the definition of done will be defined by the Team.

However, we note that, after the definition of a reference logic architecture (macro-step of Subsection 2.3.3), all the other steps can be designed and developed 'in parallel' rather that in sequence.

Of particular importance is the macro-step Subsection 2.3.6 that could be more demanding, in term of design, implementation and testing, with respect to the other ones.

2.4 Software already available

Our company (Unibo) has already developed:

1. A basic robot control software for physical ddr robots (BaseRobot), modelled as an observable POJO.
2. A IOT envelope (called mbot) of the BaseRobot that allows us to send commands on the network to a real robot built upon a mBot robot [1].
3. A virtual environment (named U-Env) based on the Unity system, that includes a virtual robot that accepts commands sent on a TCP connection on port 8090.
4. A virtual environment (named W-Env) built in JavaScript, that includes a virtual robot that accepts commands sent on a TCP connection on port 8999 [2].
5. A front-end system written in Node that provides a user interface and support for user-authentication.

Of course, we will - sooner or later - use as much as possible the (hopefully corrected or at least tested) software that our company has already built in the IOT domain. The intent however is to avoid the a-priori introduction of such a software. Rather, our aim is to introduce proper available software libraries, infrastructures, components etc. as the **proper consequence** of our problem analysis or of our design choices. In other words, we will not start from any available software, with the intent to '*discover its applicability*' to our needs.

2.4.1 The QActor metamodel/language. Among the Unibo software, there is also a custom language (named qa) that allows us to express in a concise way the structure, the interaction and also the behaviour of (distributed) software systems composed of a set of QActor.

A QActor is a software entity inspired to the *actor* concept, as can be found in the Akka library. The leading Q/q means 'quasi' since the qa language does introduce (with respect to Akka) its own peculiarities, including, besides the basic message-passing programming model, reactive actions and even-driven programming concepts, inspired to JavaScript and Node.

The qa language is introduced to build high-level, executable models of a software systems, to overcome limitations found in the UML modelling language, which is essentially object-based and not well suited for distributed

[1] See https://www.makeblock.com/steam-kits/mbot).
[2] See https://github.com/PierfrancescoSoffritti/ConfigurableThreejsApp and https://threejs.org/.

systems. Thus, the qa language is conceptually a meta-model, technically build as an instance of EMOF (the UML meta-model), by using the XText technology.

In the following, we will assume that the reader has basic knowledge on the qa modelling language and on the way it can be used to define models of the system to build, according to an incremental approach that starts from a high level, working specification of the logical architecture of the system.

For logical architecture we intend here a software architecture that is defined as consequence of the problem analysis phase, in the attempt to focus our attention on the problem itself, rather than on some specific technology. We say that, during problem analysis, we are technology aware but not technology dependent. The logical architecture will be progressively defined in more detailed way in order to define a project architecture and a proper implementation and deployment.

The incremental approach will lead us to define a sequence of QActor models, that can be used as the reference point for a sequence of SPRINT, in SCRUM terminology.

2.5 Software deployment

Our software products could be deployed in two different ways[3]:

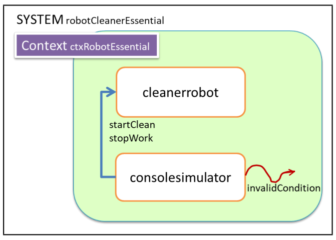
1. in a 'traditional' way, by providing a set of executable files, batch commands, etc.
2. in a Cloud environment (e.g. Amazon AWS) , by using virtual machines

[3] As regards to software development phase we could cite here DevOps, but it is not yet covered in the course.

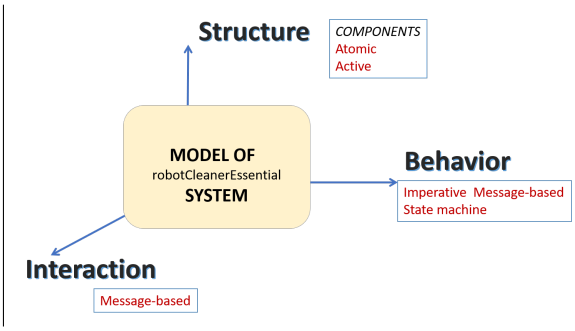
3 A first architecture

At the very first stage of our analysis, the exact nature of the `cleanerrobot` is not the most relevant aspect. What is important is to fix the overall system architecture and interaction pattern with the robot, in a technology-independent way.

At this stage, the system can be represented as in the following, informal picture:



3.1 The three-dimension space



3.1.1 Structure .

From the structural point of view, the software system to build will run on two different computational nodes: a node for the `consolesimulator` and a node for the `cleanerrobot`. The software running on each node could be written using different programming languages and run upon different operating systems; the only assumption is that the nodes can exchange information by using some communication protocol on a wireless network.

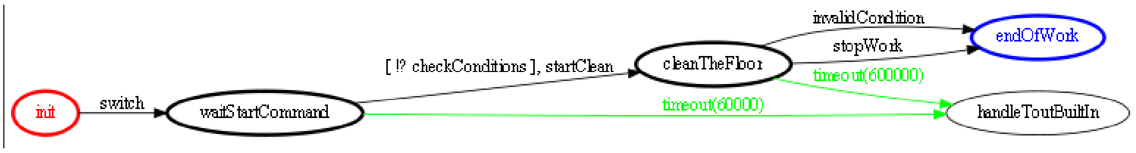
3.1.2 Interaction .

The application-specific `cleanerrobot command language` (`CRCL`) is very simple, since it can be reduced to the commands `START/STOP`. These commands are sent as messages to the robot by the `console`; both these messages can be modelled as `fire-and-forget` (*dispatch* in `QActor` terminology):

```
Dispatch startClean : startClean(V) //V= <clean details>. e.g. accuracy, current time, etc.
Dispatch stopWork   : stopWork(V) //V= <stop details> e.g. stopByuser, TimeLimit, etc.
```

3.1.3 Behaviour .

A first model of the logical behaviour of the `cleanerrobot` can be a finite state machine, working under user control, when the set of `envConds` are met:



This diagram is associated to the following set of assumptions:

- the guard `checkConditions` is `true` when all the `envConds` are met, `false` otherwise;
- the `cleanerrobot` performs its job in the state `cleanTheFloor`, by executing an `asynchronous` action that can be 'interrupted' by the event `invalidCondition` or by the message `stopWork`;
- the event `invalidCondition` is raised by some agent that checks the `envConds`. The event can be defined so to include in its payload the reason of the 'failure':

```
Event invalidCondition : invalidCondition( V ) //V <cond details> e.g. temperature, clocktime, obstacle
```

In traditional modelling languages (e.g. `UML`) we can build the state diagram above as the result of a design activity performed before writing any code. In our approach instead, we are able to obtain such a diagram as a representation of the cleanerrobot behavior produced from a formal, textual, executable model defined in the custom modelling language `QActor`.

3.2 An executable formal model

Our first `QActor` specification captures the three main aspects of any software architecture: the `structure`, i.e. the elements that compose the system, the `interaction`, i.e. how the components are connected and how they exchange information and the `behaviour` of each component[4].

```
System robotCleanerEssential
Event invalidCondition : invalidCondition( V ) //V <cond details> e.g. temperature, clocktime, obstacle
Dispatch startClean   : startClean(V) //V= <clean details>. e.g. accuracy, current time, etc.
Dispatch stopWork     : stopWork(V) //V= <stop details> e.g. stopByuser, TimeLimit, etc.

Context ctxRobotessential ip [ host="localhost" port=8030]

QActor cleanerrobot context ctxRobotessential {
Rules{
    checkConditions.
}
    Plan init normal[ ]
    switchTo waitStartCommand

// Wait for Start command from the console.
    Plan waitStartCommand[ ]
    transition stopAfter 60000
         whenMsg [ !? checkConditions ] startClean -> cleanTheFloor
// Execute the command.
    Plan cleanTheFloor [
        printCurrentMessage;
        println("cleanerrobot is cleaning the floor (asynch action)")
    ]
    transition stopAfter 600000
       whenEvent invalidCondition -> endOfWork ,
       whenMsg stopWork          -> endOfWork
// End.
    Plan endOfWork [
        onEvent invalidCondition : invalidCondition( V ) -> println( cleanerrobot(stops, invalidCondition( V ) ) );
        onMsg stopWork : stopWork(X) -> println( cleanerrobot(stops, reason(X) ) )
    ]
}
```

Listing 1.1. `robotEssential.qa`

This part of the model states that:

[4] The code of this part is in the project `it.unibo.ft2018.model0`

1. The system works - at the moment - within a single computational node (the context ctxRobotessential).
2. The system includes a singleQActor (cleanerrobot) that models the robot without any distinction between 'robot-mind, robot-actuator' and 'world-observer'.
3. The cleanerrobot actor works as the FSM introduced in Subsection 3.1.3.

In order to obtain a first working prototype, let us complete our specification by introducing a QActor that works as a consolesimulator:

```
QActor consolesimulator context ctxRobotessential{
    Plan init normal[
        forward cleanerrobot -m startClean : startClean( conds(10,25) ) ;
        delay 1000;
        emit invalidCondition : invalidCondition( temperature );
//      forward cleanerrobot -m stopWork : stopWork( user ) ;
        delay 3000
    ]
}
```

Listing 1.2. robotEssential.qa

This high-level specification is automatically translated into Java code by the QActor software factory, that produce a set of useful resources, including a main program named MainCtxRobotessential.

3.2.1 The result .

If we run the program MainCtxRobotessential, we obtain the result shown hereunder:

```
...
--------------------------------------------------------------------------------------------
cleanerrobot_ctrl currentMessage=msg(startClean,dispatch,consolesimulator_ctrl,cleanerrobot,startClean(conds(10,25)),1)
--------------------------------------------------------------------------------------------
"cleanerrobot is cleaning the floor (asynch action)"
cleanerrobot(stops,invalidCondition(temperature))
...
```

3.3 A first review

At this point we note that:

1. the business logic encapsulated in the state cleanTheFloor is reduced to a println;
2. the guard checkConditions is always true, since we extend the knowledge-base of the cleanerrobot with a (Prolog) fact that always unifies with the guard:

```
QActor cleanerrobot context ctxRobotessential {
Rules{
    checkConditions.
}
```

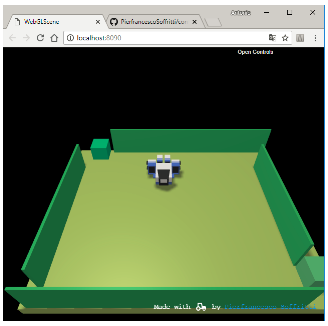
3. our system is not distributed. This can help in a first testing phase. However, we can easily evolve into a distributed system, by introducing another context devoted to the execution of the consolesimulator. The QActor software factory will automatically reconfigure the underlying run-time support to handle the exchange of information between the different contexts. The default run-time support for the QActor is based on TCP, but we will see that it can easily replaced by MQTT or CoAP.
4. if we intend to (re)use a 'real' console that already has its own command language, we will have to manage the 'translation' from the console command-language to our robot command-language.

4 Beyond the first toy-model

Let us modify our first toy-model in order to obtain a more acceptable prototype. At this stage of software development, it is wise/appropriate to make reference to a *virtual robot*, rather than to a real one.

4.1 Working with a virtual robot

Thus, the first extension to the toy-model of Subsection 3.1.3 consists in extending the business logic encapsulated in the state `cleanTheFloor` with a simple action that moves the virtual robot in a `W-Env` scene[5] `without obstacles`

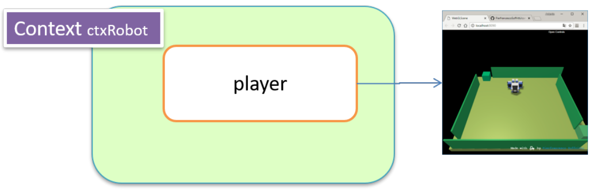


The `cleanerrobot` is now split in two parts:

- the `player`: an actor that works as an actuator of robot move-commands;
- the `robotmind`: an actor that implements the business logic.

The distinction between a 'mind' and a 'body' (the `player`) in not motivated by philosophical reasons; rather the introduction of a `player` is motivated by the pragmatic reason to encapsulate into a component all the (technological) details related to the interaction with robot, that could be real or virtual and built in very different ways.

The result of this re-factoring is shown in the following picture:



4.1.1 A working model .

To achieve the goal to move a (virtual) robot, we must initially connect the `player` with the virtual environment, on port `8999`[6].:

```
System robotCleaner

Event invalidCondition : invalidCondition( V ) //V <cond details> e.g. temperature, clocktime, obstacle
Dispatch startClean  : startClean(V) //V= <clean details>. e.g. accuracy, current time, etc.
Dispatch stopWork    : stopWork(V) //V= <stop details> e.g. stopByuser, TimeLimit, etc.

Context ctxRobot ip [ host="localhost" port=8030] //-g yellow

QActor player context ctxRobot {
Rules{
```

[5] We could use instead the `U-Env` mentioned in Subsection 2.4.

[6] The code of this part is in the project `it.unibo.ft2018.model01`

```
    checkConditions.
}
    Plan init normal[
        println("player STARTS");
        javaRun it.unibo.utils.clientTcp.initClientConn("localhost", "8999");
        delay 1000;
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': 800 }");
        delay 1000;
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveBackward', 'arg': 800 }")
    ]
    switchTo waitStartCommand
//Wait for Start command from the console;
    Plan waitStartCommand[ ]
    transition stopAfter 60000
         whenMsg [ !? checkConditions ] startClean -> cleanTheFloor
```

Listing 1.3. `robot.qa`: `player` init

4.2 Custom actions (javaRun)

As happens for any language, the expressive power of our `QActor` modeling language is limited. In particular, the concept of of moving a virtual robot in a virtual environment like `W-Env` is out of the scope of the `QActor` language. The `javaRun` key-word provides an 'escape mechanism' to run `Java` code provided by the Application designer.

In our case, we exploit the class `it.unibo.utils.clientTcp` as a support for the interaction with the `W-Env`.

The payload of the messages sent to move the robot is set according to the *interaction language* defined by the `W-Env`. A call that associates to `arg` the value `-1` means that the action must be performed as an **asynchronous** action that has no time-limit. A call that associates to `arg` a positive natural number `N` is a asynchronous actions that ends after `N` millisecs. For example the call:

```
javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': 60000 }")
```

puts in execution an action with the duration of one minute. However, the control is immediately returned, since the action is executed in asynchronous way. As a general statement, we can say that:

> The basic operations that move a robot (either virtual or real) are always executed as `asynchronous actions` that immediately return the control to the caller.

4.3 Moving the robot

In the initial state, we tell to the robot to perform some movement in order to check that the connection with the `W-Env` is working. The next step consists in introducing - in a proper states of the `player` - commands to move the virtual robot:

```
    Plan cleanTheFloor [
        printCurrentMessage;
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': -1 }") //asynch non-terminating action
    ]
    transition stopAfter 600000
       whenEvent invalidCondition -> endOfWork ,
       whenMsg stopWork          -> endOfWork
```

Listing 1.4. `robot.qa`: `player` move

4.4 The final state

When the `player` robot receives a `stopWork` message from the user or perceives a `invalidCondition` event, it enters in its final state `endOfWork` in which:

1. the robot stops;
2. the robot updates its local knowledge-base with a fact related to the motivation of its stopping.

```
   Plan endOfWork [
      javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'alarm', 'arg': 0 }"); //STOP action
      onEvent invalidCondition : invalidCondition( V ) -> addRule cleanerrobot(stops, reason(invalidCondition(V)));
      onMsg  stopWork        : stopWork(V)         -> addRule cleanerrobot(stops, reason(stopWork(V)));
      [ !? cleanerrobot(stops, REASON ) ] println( robotEndOfWork( REASON) )
   ]
```

Listing 1.5. `robot.qa`: end

The fact `cleanerrobot/2`:

```
cleanerrobot(ACTION, REASON)
```

can be useful to support automated testing (see Subsection ??).

4.5 Automatic testing

Our goal now is to define a set of testing rules and to automate the testing phase. The facts introduced in the knowledge-base of the actors (like those introduced in Subsection 4.4) can be used to write assertion about the expected state at certain points of the computation.

Our testing plan starts with a `setUp` phase that starts the `player`:

```
public class TestModel01 {
private QActor player;
   @Before
   public void setUp(){
      try {
         //ASSUMPTION: W-Env is working
         it.unibo.ctxRobot.MainCtxRobot.initTheContext();
         System.out.println(" ***TEST:*** TestPrototype waits for a while ........ " );
         Thread.sleep( 3000 ); //give the time to start and execute
         player = QActorUtils.getQActor("player_ctrl");
      } catch (Exception e) {
         fail( e.getMessage() ) ;
      }
   }
```

Listing 1.6. `TestModel01`: setUp

The class `QActorUtils` is provided by the `QActor` Software Factory; its static method `getQActor` returns a reference to the actor with the given name - in our case the actor `player`.

The class `QActorUtils` provides also operations to send messages or emit events. We use these operations:

1. `QActorUtils.sendMsg`: to send to the `player` a message `startClean`, in order to activate it;
2. `QActorUtils.emitEventAfterTime`: to emit the event `invalidCondition` related to a temperature value.

Once the robot is started, we can write a test that generates a well defined sequence of messages and events and finally checks the state of the robot:

```
   @Test
   public void testTemperatureLimit(){
      try {
         //robot.automsg("startClean", "dispatch", "startClean( conds(clock(10),temperature(25)) )");
         //SIMULATE A CONSOLE START
         QActorUtils.sendMsg(player, "player", "startClean", "startClean( conds(clock(10),temperature(25)) )");
         //SIMULATE A invalidCondition related to TEMPERATURE after 300 msec
         QActorUtils.emitEventAfterTime(player, "tester","invalidCondition","invalidCondition(temperature(30))",300);
         //WAIT for 500 msec
         Thread.sleep( 500 );
         //LOOK AT the robot Knowledge-base
         SolveInfo sol     = player.solveGoal("cleanerrobot(X, Y)");
         String reasonOfStop = sol.getVarValue("Y").toString() ;
         System.out.println(" ***TEST:*** testTemperatureLimit " + reasonOfStop );
```

```
            assertTrue( "testTemperatureLimit", reasonOfStop.contains("invalidCondition(temperature") );
        } catch (Exception e) {
            System.out.println(" ***TEST:*** testTemperatureLimit ERROR " + e.getMessage() );
            fail( e.getMessage() ) ;
        }
    }
```

Listing 1.7. `TestModel01`: test

The reference to the `player` allows us to look at its knowledge-base by means of the built-in method `solveGoal` and to make assertions. Of course, this is not the correct way to interact with an actor, since we use as a `POJO`. However, this kind of 'impure' access is acceptable during a testing phase, whose code will be kept outside the deployment.

In this version, we do not introduce the `ctxConsole` node, since its activities that can be directly simulated in the test code.

Now we can run the test with the command:

```
gradle -b build_ctxRobot.gradle build
```

and consult the reports generated by `gradle` and by `jacoco`.

5 Towards a distributed system

The second extension to the toy-model of Subsection 3.1.3 consists in introducing another context, devoted to the execution of the `consolesimulator`. This can be done in two ways:

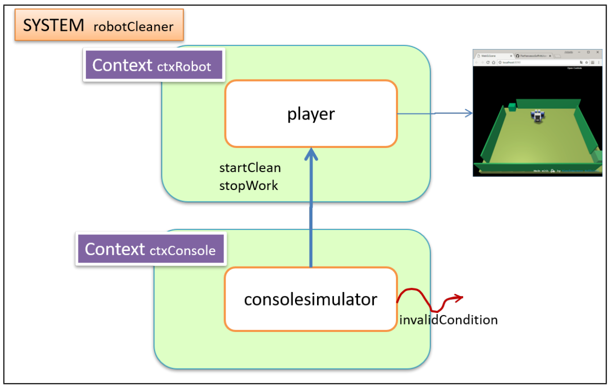
1. `Static mode`: define a model of the entire system in a single file, including all the contexts that composed the system. In this way we have a global picture of the system, while the `QActor` software factory is able to generate proper distribution files for each Context. This approach should be followed for `statically defined systems` that can prefigure the configuration of their computational nodes.
2. `Dynamic mode`: define the model of a basic component, working in a single `Context`, that will be used as a 'pivot' for `dynamically configurable systems`. New contexts can be dynamically added to the system by using the `pivot Context` as a reference point. The the `QActor` software factory is able to *update* the knowledge on the system configuration at each working node, so to provide support for high-level message/event-based interaction among the components.

In this section we will follow the second (dynamic) approach, by using the *robot model as the pivot*, and by defining the model of the `consolesimulator` in a new file:

```
System robotCleaner
Event invalidCondition : invalidCondition( V ) //V <cond details> e.g. temperature, clocktime, obstacle
Dispatch consoleCmdMsg : consoleCmd ( X )
Dispatch startClean  : startClean(V) //V= <clean details>. e.g. accuracy, current time, etc.
Dispatch stopWork    : stopWork(V) //V= <stop details> e.g. stopByuser, TimeLimit, etc.

Context ctxConsole ip [ host="localhost" port=8027 ]
Context ctxRobot ip [ host="localhost" port=8030 ] -standalone
```

Listing 1.8. `console.qa`: the contexts

At this stage, the system can be represented as in the following, informal picture:



At this stage of the work, we have just defined a first `logical architecture` of the software system that can be used as a reference model during our software development.

5.1 The consolesimulator

Now the `consolesimulator` can simply simulate, without the need of any `GUI`, the user commands `startClean/stopwork`.

```
QActor consolesimulator context ctxConsole{
    Plan init normal[
        println("consolesimulator send startClean ...");
        sendto player in ctxRobot -m startClean : startClean( conds(clock(10),temperature(25)) ) ;
        delay 1000;    //(1)
```

```
        println("consolesimulator send stopWork ...");
        sendto player in ctxRobot -m stopWork : stopWork( user ) ;
        println("consolesimulator ENDS ")
    ]
}
```

Listing 1.9. `console.qa`:

The `QActor` infrastructure will implement the details related to the message exchange between te console and the robot. As already said in Section 3, the default run-time support for the `QActor` is based on `TCP`, but we will see that it can easily replaced by `MQTT`, `CoAP` or other protocols.

5.2 Simulate changes of the temperature

In order to capture another essential aspects of the problem, let us add a new `QActor` in the `ctxConsole` to simulate the raising of a `invalidCondition` event. The delay value at point (2) should be set in relation with the delay at point (1) of the `consolesimulator` related to the `stopWork` command:

```
QActor temperaturesimulator context ctxConsole{
    Plan init normal[
        delay 700;     //(2)
        println("temperaturesimulator EMITS" );
        emit invalidCondition : invalidCondition( temperature(30) ) ;
        println("temperaturesimulator ENDS ")
    ]
}
```

Listing 1.10. `console.qa`: `temperaturesimulator`

5.3 Executing the distributed system

To execute the distributed system we have to execute a sequence of actions:

1. Activate the `W-Env` [7].
2. Activate the robot, by running the generated program `it.unibo.ctxRobot.MainCtxRobot`.
3. Activate the `consolesimulator`, by running the generated program `it.unibo.ctxConsole.MainCtxConsole`.

These actions can be done in automatic way by a *Testing Activity* similar to that we will introduce in Subsection 4.5. At the moment, if we run the generated program manually, the result will show a robot that does not clean the floor; it simply:

1. waits fro a `startClean` message (from the `consolesimulator`) ;
2. moves on a straight line in front of it, with the possibility to react to invalid working conditions;
3. stops when it perceives a `invalidCondition` event or when it receives a `stopWork` message (from the `consolesimulator`).

The execution of the prototype at this point should show the message:

```
cleanerrobot(stops,invalidCondition(temperature(30)))
```

The assumptions are:

1. The robot starts working under valid `envConds`.
2. The user sends - via the `consolesimulator` - a `stopWork` message after a time interval `T1`.
3. The temperature goes beyond the acceptable limit after a time `T2`, `T2<T1`.

[7] We call a `Node` application `main.js` stored in `C:/Didattica/Aug2018/init/WEnv/server/src`.

6 More business logic

Our next step is to start from the logical architecture of Section 5 and make the robot able to react to environment conditions `envConds` while cleaning. Moreover, we intend to face the problem of `LED` handling, i.e. the requirements `R-BlinkLed` and `R-BlinkHue`.

At this stage, we:

– simulate changes of the temperature and the reaching of a time-limit. For the sake of simplicity, we will consider here only the case of temperature-change; other environment conditions will be handled in the same way;
– emit the `invalidCondition` event when one of the `envConds` is violated;
– use a simulated `LED`.

At the very beginning of our work, we said (Section 2) that the part that gives most significant `business-value` to our work can be associated to a software component (called '`robot-mind`') that embeds the business logic of our system. Thus, let us extend the basic logical architecture of Section 5 by introducing a `QActor` to represent a `robotmind` that sends commands to the `player` in order to move the robot.

6.1 The mind and the player as actors

The important point now is to fix the 'interface' between the `robotmind` and the `player`. Logically, these components could be modeled and implemented as *functions*, *objects*, *processes*, *tasks*, *agents*, etc. The most probable choice, at the current state of the art in software engineering, is to see each component as an **`object`** (written in `Java` or in `C#/C++`) that interacts by means of procedure (method) calls. For example, the `robotmind` could be the 'master' that calls the `player` to move the robot.

However, with an object-based approach we envisage two main problems:

– we are induced to use the same programming language to write the code of each component;
– a (non-trivial) extra effort in programming is required if each component must run on a different computational node.

Since we want to design and build an heterogeneous, distributed system, by allowing wide flexibility in the implementation technology of each component, our choice here is to model each component as an `actor` and to base the interaction between the components on *messages* or on *events*. This choice means that:

> the interface of each components has to be defined by defining the messages/events that each component is able to handle, rather than in term of methods.

6.2 The robot-actuator

The robot actuator is a re-factoring of the `player` introduced in Section 4, so to male it able to execute commands sent to it as a `dispatch` defined as follows:

```
Dispatch robotCmd : robotCmd(V) //V= w | s | a | d | h
```

Thus, the `player` becomes a `QActor` able to convert a `robotCmd` into a proper implementation:

```
QActor player context ctxRobot {
    Plan init normal[
        javaRun it.unibo.utils.clientTcp.initClientConn("localhost", "8999") ;
        //Execute some initial move, just for checking
        delay 1000;
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': 800 }");
        delay 500;
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveBackward', 'arg': 400 }")
    ]
    switchTo cmdIntepreter

```

```
    Plan cmdIntepreter[
    ]
    transition stopAfter 600000
         //whenEvent sonarDetect : sonarDetect(ANY) do printCurrentEvent , //just for testing
         whenMsg  robotCmd -> cmdExecutor
    finally repeatPlan

    Plan cmdExecutor resumeLastPlan[
        //printCurrentMessage;
        onMsg robotCmd : robotCmd(w) -> javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': -1 }");
        onMsg robotCmd : robotCmd(a) -> javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnLeft', 'arg': 800 }");
        onMsg robotCmd : robotCmd(d) -> javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnRight', 'arg': 800 }");
        onMsg robotCmd : robotCmd(s) -> javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveBackward', 'arg': -1 }");
        onMsg robotCmd : robotCmd(h) -> javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'alarm', 'arg': 0 }")
    ]
}
```

Listing 1.11. `robot.qa`: the `robot-actuator`

6.3 The robot-mind

The `robot-mind` will move the robot by sending messages to the `player`, with a payload configured according to the `player` command language. In this way the `player` acts as an 'adapter' from the (technology-independent) `robot-mind` to a specific robot.

The `robot-mind` must clean the floor, while being sensible to the `invalidCondition` event. Moreover, the `robot-mind` must also be prepared to handle events emitted by the sonar devices, defined as follows:

```
sonar        : sonar( SONARID, player, DISTANCE )
sonarDetect  : sonarDetect( V )     //V =<obstacle name> e.g. ball, wall, ...
```

The event `sonar` is generated by the sonar devices `sonar1` and `sonar2`, while `sonarDetect` is generated by the `sonar-robot`. The `robotmind` is logically interested in these events in order to manage the robot activity.

Thus, the `robot-mind` can be modeled as an actor that first waits for a `startClean` message (sent from the `console`) and then (state `cleanTheFloor`) moves the robot by sending messages to the player.

```
QActor robotmind context ctxRobot {
    Plan init normal[ ]
    switchTo waitStartCommand

    Plan waitStartCommand[ ]
    transition stopAfter 600000
         whenMsg startClean -> cleanTheFloor

// Execute the command.
    Plan cleanTheFloor [ //Start to work only if there is no invalid condition
        //printCurrentMessage;
        [ ?? msg(invalidCondition,MSGTYPE,EMITTER,none,EV,N) ]
                                  selfMsg stopWork : stopWork( EV )
        else {
            println( "==== cleanTheFloor (simulation)" );
            forward player -m robotCmd : robotCmd(w);
            emit robotState : robotState( running )
        }
    ]
    transition stopAfter 600000
       whenMsg invalidCondMsg -> endOfWork ,
       whenMsg stopWork      -> endOfWork,
       whenMsg sonarDetectMsg -> handleRobotSonar ,
       whenMsg sonarMsg      -> handleFloorSonar
```

Listing 1.12. `robot.qa`: the `robotmind`

From the model speciifcation, we note that:

- the state `cleanTheFloor` does perform a simple 'forward' move. Thus, the problem of floor cleaning is postponed;

– the state cleanTheFloor emits the event

```
   robotState : robotState( running )
```

to signal to some 'oberver' (see Section 7) that the robot is working;

– the robotmind does not handle any event, but only a set of messages.

6.4 From events to messages

The robotmind does not handle events since it handles messages generated from events by a proper set of *EventHandlers*.

6.4.1 Event handling .

The occurrence of an event can immediately put in execution some code devoted to the management of that event. We qualify this kind of behaviour as event-driven behaviour, since the event 'forces' the execution of code.

An event can also trigger state transitions in components, usually working as finite state machines. We qualify this kind of behaviour as event-based behaviour, since the event is 'lost' if no actor is in a state waiting for it.

In a QActor system, the occurrence of an event activates, in event-driven way, all the ***EventHandlers*** declared in actor *Context* for that event.

```
Context ctxRobot ip [ host="localhost" port=8030]

EventHandler evhcondition for invalidCondition{ //store the event in the worldtheory of robotmind
    memoCurrentEvent -lastonly for robotmind //msg(MSGID,event,EMITTER,none,EV,N)
};
EventHandler evhrobotstate for robotState {//map a robotState event into a msg
    forwardEvent worldobserver -m robotChangeState
};
EventHandler evhtemperaturestate for temperatureState {//map a temperatureChangeValue event into msg
    forwardEvent worldobserver -m temperatureChangeValue
};
EventHandler evhsonar    for sonar{ forwardEvent robotmind -m sonarMsg };
EventHandler evhsonardetect for sonarDetect{ forwardEvent robotmind -m sonarDetectMsg };
EventHandler evhinvalidcond for invalidCondition{ forwardEvent robotmind -m invalidCondMsg };
```

Listing 1.13. robot.qa: events, messages and context

The reason to map an event into a message is that events can be lost. However,

---

the mapping of events into messages could lead to the generation of a sequence of messages that must be properly 'consumed'.

For example, a (real) sonar usually emits a stream of events that in this prototype will generate a stream of messages. The robotmind should handle the first message of the stream and discard the others.

6.5 Consuming pending messages

Let us introduce a state devoted to the handling of pending messages generated from (sonar) events:

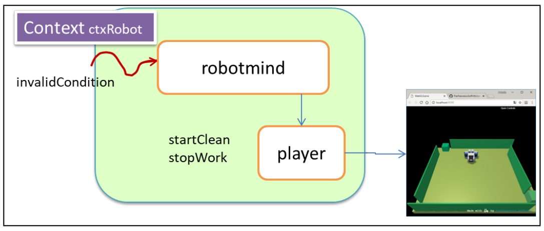
```
    Plan handlePendingMsg []
    transition whenTime 300 -> cleanTheFloor
       whenMsg sonarMsg : sonar( NAME, player, DISTANCE )
           do println( pendingMsg(sonar) ),
       whenMsg sonarDetectMsg : sonarDetect( V )
           do println( pendingMsg( sonarDetect ) )
    finally repeatPlan
```

Listing 1.14. console.qa: handlePendingMsg

The robotmind will switch to this state when needed and, when all pending messages are consumed, it will return to the state cleanTheFloor.

6.6 The robot context

The robot-context includes now two actors: the `robotmind` provides the business logic, while the `player` is an '`adapter`' towards an external entity (the real or virtual robot).



The set of events/messages occurring in the system is declared at very beginning of the system specification. In our case, the events are:

```
System robotCleaner

//Application Events
Event invalidCondition : invalidCondition( V ) //V=<cond details> e.g. temperature, clocktime, obstacle
//Events related to the W-Env
Event sonar          : sonar( NAME, player, DISTANCE )
Event sonarDetect    : sonarDetect( V )     //V =<obstacle name> e.g. ball, wall, ...
//Events related to monitoring
Event robotState     : robotState( V ) //V=<state details> e.g. running | stopped
Event temperatureState : temperatureState ( V ) //V=<temperature value in Celsius>
Event ledState       : ledState( V ) //Events related to LED control
```

Listing 1.15. `robot.qa`: events

The events devoted to monitoring will help us (see Section 7) to trace the changes of the state of the robot and of the devices.
The messages used in the system are:

```
//Application messages
Dispatch startClean   : startClean(V) //V=<clean details>. e.g. accuracy, current time, etc.
Dispatch stopWork     : stopWork(V) //V=<stop details> e.g. stopByuser, TimeLimit, etc.
//Control messages from events
Dispatch sonarMsg     : sonar( NAME, player, DISTANCE )
Dispatch sonarDetectMsg : sonarDetect( V )
Dispatch invalidCondMsg : invalidCondition( V )
//Control messages
Dispatch robotCmd     : robotCmd(V)  //V= w | s | a | d | h
//Observer control messages from events
Dispatch robotChangeState    : robotState(V) //the same as Event robotState
Dispatch temperatureChangeValue : temperatureState ( V ) //the same as Event temperatureState
//Led control messages
Dispatch ledBlink : ledBlink
Dispatch ledOff  : ledOff
```

Listing 1.16. `robot.qa`: messages

6.7 Reusing a (web) console

Until this moment, we have followed the hypothesis that the console sends commands that can be directly understood by the robot. However, the `QActor` software factory provides - for the sake of rapid prototyping - a built-in `Web-Interface` for a `Context` that can be easily tailored to specific application needs.

> The choice to introduce here a `Web-Interface` can be related to the *problem analysis* for the `R-Start` requirement, which states that the console could run on a conventional PC or on a smart device. In fact, in this way we facilitate the porting of the 'console' on any smart device, without the need to build specific apps. This consideration is a typical motivation that leads from the idea of `Internet-of-Things` to the idea of `Web-of-Things`.

A model for our new `Web-Gui` console can be introduced by simply putting a `-httpserver` flag on the declaration of the console-context:

```
Context ctxRobot ip [ host="localhost" port=8030 ] -standalone
Context ctxConsole ip [ host="localhost" port=8027 ] -httpserver
```

Listing 1.17. `console.qa`: the `ctxConsole`

When the `QActor` factory reads the flag `-httpserver`, it creates a local (quite traditional) web-server, using a web socket, that provides a built-in `Web-Interface` that emits events of the form:

```
Event usercmd : usercmd(CMD) //CMD=robotgui(M), M= w(S) | s(S) | d(S) | a(S) | h(S), S=low|high|medium
Event cmd     : cmd(CMD)    //CMD=start | end
```

Since the built-in `Web-Interface` 'speaks its own language', we have to introduce a *translator* (as said in Section 4)[8]:

```
QActor cmdconverter context ctxConsole {
   Plan init normal[ //println("cmdconverter WAITS")
   ]
   transition stopAfter 600000
       whenEvent cmd     -> commandTheRobotCleaner,
       whenEvent usercmd -> moveTheRobot
   finally repeatPlan

   Plan moveTheRobot resumeLastPlan [
//      printCurrentEvent;
        onEvent usercmd : usercmd(robotgui(w(SPEED))) -> sendto player in ctxRobot -m robotCmd : robotCmd(w) ;
        onEvent usercmd : usercmd(robotgui(s(SPEED))) -> sendto player in ctxRobot -m robotCmd : robotCmd(s) ;
        onEvent usercmd : usercmd(robotgui(a(SPEED))) -> sendto player in ctxRobot -m robotCmd : robotCmd(a) ;
        onEvent usercmd : usercmd(robotgui(d(SPEED))) -> sendto player in ctxRobot -m robotCmd : robotCmd(d) ;
        onEvent usercmd : usercmd(robotgui(h(SPEED))) -> sendto player in ctxRobot -m robotCmd : robotCmd(h)
   ]

   Plan commandTheRobotCleaner resumeLastPlan [
        printCurrentEvent;
        onEvent cmd : cmd(end)  -> sendto robotmind in ctxRobot -m stopWork : stopWork( userCmd );
        onEvent cmd : cmd(start) -> emit activateStarter : activateStarter //the same for testing
   ]
}
```

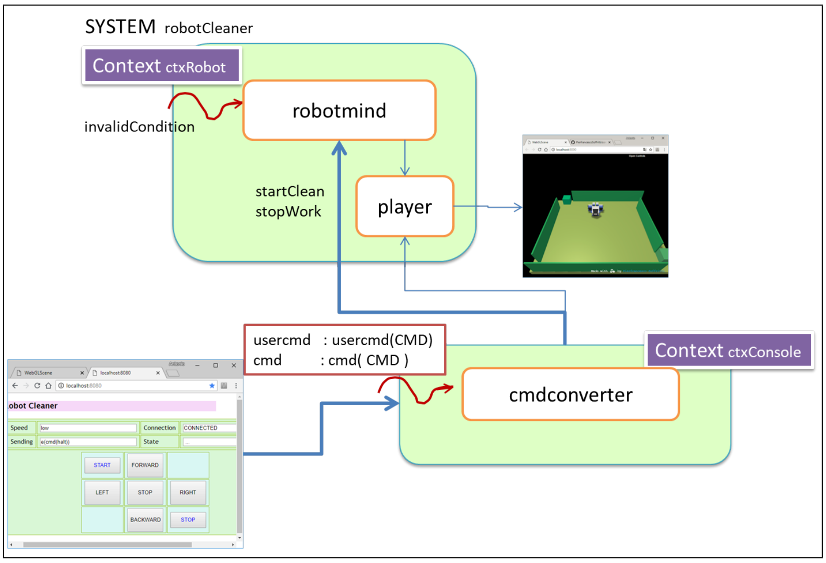
Listing 1.18. `console.qa`: the `cmdconverter`

The converter does not only generate the `startClean/stopWork` messages for the `robot-mind`, but it generates also `robotCmd` messages, according to the events emitted by the `Web-Interface`.

The `Web-Interface` does not provide only two buttons (`START/STOP`) to command the `cleanerrobot`, but also a set of buttons to move the robot. This feature is not included in the requirements, but can be very useful during the testing phase, and can be easily removed form the final `Web-Interface`.

At this stage, the system can be represented as in the following, informal picture:

[8] Of course we could change the `Web-Interface` language, but in this way, we will abandon the idea of using an available interface 'as a service'.



6.7.1 The starter .

The starter is an actor that handles the event `activateStarter` emitted when the user presses the `START` button on the `Web-interface`. Its task is to send to the `robotmind` the message `startClean:startClean(init)`.

```
QActor starter context ctxConsole{
   Plan init normal[ ]
   transition stopAfter 600000
       whenEvent activateStarter -> work
   finally repeatPlan

   Plan work resumeLastPlan [
       println("STARTER activated");
       sendto robotmind in ctxRobot -m startClean : startClean( init )
   ]
}
```

Listing 1.19. `console.qa`: starter

6.7.2 Console events .

As part of the `robotCleaner` system, the console makes reference to a set of events, to interact with the robot and with the external devices, including the `Web-interface`:

```
//Application Events
Event invalidCondition : invalidCondition( V ) //V <cond details> e.g. temperature, clocktime, obstacle
Event activateStarter : activateStarter
//Events related to the user interface
Event usercmd      : usercmd(CMD) //CMD=robotgui(M), M= w(S) | s(S) | d(S) | a(S) | h(S), S=low|high|medium
Event cmd          : cmd(CMD)    //CMD=start | end
//Events related to devices
Event temperatureState : temperatureState ( V ) //V=<temperature value in Celsius>
```

Listing 1.20. `console.qa`: events

6.7.3 Console messages .

As part of the `robotCleaner` system, the console makes reference to a set of messages, to interact with the robot and with the external devices:

```
Dispatch startClean  : startClean(V) //V= <clean details>. e.g. accuracy, current time, etc.
Dispatch stopWork    : stopWork(V) //V= <stop details> e.g. stopByuser, TimeLimit, etc.
//Robot control messages
Dispatch robotCmd    : robotCmd(V)  //V= w | s | a | d | h
```

Listing 1.21. `console.qa`: events

6.7.4 Temperature provider .

At the moment, we introduce in the `console` an actor that simulates the work of a temperature sensor:

```
QActor temperatureprovider context ctxConsole{
    Plan init normal[
    //      delay 1000;
            println("temperatureprovider EMITS 22" );
            emit  temperatureState : temperatureState( 22 ) ;
            delay 1000;
            println("temperatureprovider EMITS 24" );
            emit  temperatureState : temperatureState( 24 ) ;
//          delay 1000;
//          println("temperatureprovider EMITS 30" );
//          emit  temperatureState : temperatureState( 30 ) ;
            delay 1000;
            println("temperatureprovider EMITS 26" );
            emit  temperatureState : temperatureState( 26 )
    ]
```

Listing 1.22. `console.qa`: `temperatureprovider`

7 Observing the environment

As said in Section 2, the responsibility of monitoring the environment and rising an `invalidcondition` event can be given to the `robotmind` or to some other, specialized entity. Since the task of the `robotmind` should be strictly related to the application logic, we delegate the task of monitoring the environment to another actor (named `worldobserver`) that will run on the node of the `ctxRobot` context.

The choice to exclude the `ctxConsole` context as a container for the `worldobserver`)is motivated by the following reason:

> The responsibility of the console should be limited to provide a support for the interaction with a human user, and should exclude as much as possible activities related to the application logic. These activities should be delegated to *specialized components* that can be `used` by the console, if necessary.

The main task of a monitoring activity consists in updating a proper, explicit representation of the observed entity, when it perceives some change in it. In our case, the entity to be monitored is 'the environment' and its changes can be perceived by looking at a proper set of events (`ledState`, `temperatureState`, `robotState`). Knowledge about the robot state could be useful for adding features to the system, for example those related to requirements `R-BlinkLed` and `R-BlinkHue` (see Subsection 7.3).

An explicit representation of the current state of the environment (including the robot) can be useful:

1. to trace (`log`) the sequence of environment modifications perceived by the system;
2. to give information to the end-user about the state of the computation and of the world;
3. to add observers that can perform actions when the environment changes.

The `QActor` language allows us to associate to each `QActor` a knowledge base expressed in the `Prolog` (`tuProlog`, for the precision) logic programming language. Moreover, it allows us to define a set of rules in Prolog-style directly in the model[9]

7.1 The rule of the worldobserver

Thus, the current state of the environment can be represented by a set of facts (e.g. `tempepratureState/1, robotState/1, ledState/1`) and their modification can be handled by a proper set of rules.

```
QActor worldobserver context ctxRobot{
 Rules{
 //STATE OF THE ENVIROMENT (RESOURCE MODEL)
     temperatureState( 0 ).
     robotState( stopped(init) ).
     ledState( off ).

//[R- TimeOk and R- TempOk ]
  limitTemperatureValue (28).

  checktemperatureState(T):-
     temperatureState(T),
     limitTemperatureValue (L),
     eval (gt , T , L).

//ENVIROMENT STATE MODIFICATION RULES
  //Temperature
  changeTemperatureState( T ) :-
    replaceRule( temperatureState( _ ),temperatureState( T ) ),
    checktemperatureState( T ),!,
    output( changeTemperatureState( invalidCondition( temperature(T) ) ) ),
    emitEvent( invalidCondition, invalidCondition( temperature(T) ) ). //!!! emit
  changeTemperatureState( T ).
  //Led
  changeLedState( blink ) :-
    replaceRule( ledState( _ ),ledState( blink ) ),
    output( changeLedState(blink) ),
```

[9] We can use here only a limited subset of the `Prolog` syntax.

```
  sendMsg( ledcontrol, ledBlink, ledBlink ). //send from Prolog
changeLedState( off ) :-
  replaceRule( ledState( _ ), ledState( off ) ),
  output( changeLedState(off) ),
  sendMsg( ledcontrol, ledOff, ledOff ).
//Robot
changeRobotState( running ) :-
  replaceRule( robotState( _ ), robotState( running ) ),
  output( changeRobotState(running) ),
  changeLedState( blink ).
changeRobotState( stopped(V) ) :-
  replaceRule( robotState( _ ),robotState( stopped(V) ) ),
  output( changeRobotState(stopped) ),
  changeLedState( off ).
}
```

Listing 1.23. `robot.qa`: the `worldobserver` rules

The rules written by the application designer are added to the built-in rules loaded by the `QActor` support into the local knowledge-base of any actor. The built-in rules are written in the file `WorldTheory.pl` that is generated by the `QActor` Software Factory into the `srcMore` directory.

Among the built-in rules, there are the rules `emitEvent/2` and `sendMsg/3` that extend to the Prolog level the capabality og an actor to emit events and send messages. In our case, we exploit these rule to obtain the following effects

- when the robot changes its state, the `worldobserver` *sends* a message to a `ledControl` actor (see Subsection 7.3) in order to fulfill the requirements on Led blinking;
- when the temperature changes, the `worldobserver` *emits* the `invalidCondition` event if the value is higher than the given limit.

7.2 The behavior of the worldobserver

The behaviour of the `worldobserver` consists in looking at the events raised from the environment and in modifying the environment-representation in a proper way:

```
Plan init normal[ ]
switchTo observeTheWord

Plan observeTheWord[ println("worldobserver WAITS ...") ]
transition stopAfter 600000
   whenMsg robotChangeState     : robotState(V)      do demo changeRobotState(V),
   whenMsg temperatureChangeValue : temperatureState(T) do demo changeTemperatureState(T)
finally repeatPlan
```

Listing 1.24. `console.qa`: the world-observer

7.3 `LED` handling

A `LED` is a special kind of `actuator`, that can be modelled in two basic ways: as a conventional object (`POJO`) or as a 'service'. In our problem, we have both these model types:

- The `LED` on the robot can be modelled as a `POJO` that provides some `Interface` (say `ILed`).
- The `LED HUE Lamp` is a more complex device, that can be handled by using `REST`ful interaction based on `HTTP`.

From a logical point of view, it is convenient to model a `LED` as an *actor* that can handle events or messages directly sent to it. This choice seems at the moment quite strange or, worst, inappropriate, since it does not capture neither the idea of a `POJO` neither the idea of a `REST`ful device. But this is an example of our goal to be `technology-aware` but not technology-dependent. More precisely, modelling a `LED` as a `QActor` (let us call it `ledcontrol`) gives us the following benefits:

- a `POJO` deployed on the `cleanerrobot` can be (re)used within the `ledcontrol` that acts as a mapper between a message/event and the `POJO` interface;

- a RESTful device can be accessed within ledcontrol that acts as a 'router' between the logical level and the implementation level;
- the ledcontrol should be considered as as part of the cleanerrobot node (Context). But in this case, if the LED is a HUE Lamp, the robot should be connected in Internet, and this is not assured, while it is more probable that the PC (i.e. the Context of the console) has an Internet connection. Since the definition ledcontrol can be easily put in the console-context, we can set our concrete architecture according to the specific Internet configuration.

We note that a ledcontrol actor can be always put on the console-context, by properly extending the cleanerrobot command language to handle a POJO-LED put on a real robot. This case however is excluded at the moment, since the requirements state that the HUE Lamp must be used only in the case of a virtual robot.

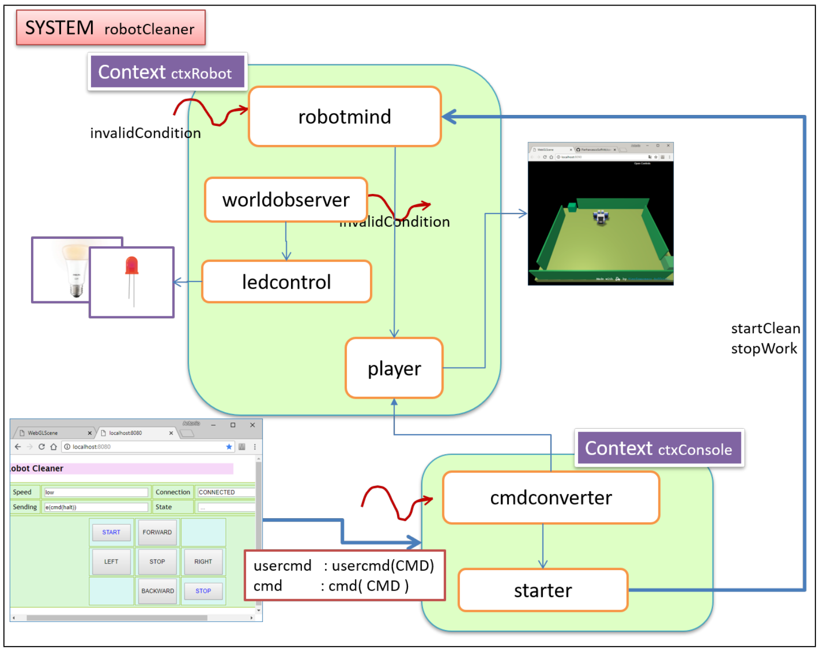
Thus, in order to capture the logic that properly manages the state of the LED according to the robot state, let us introduce the ledcontrol actor in the cctxRobot context.

```
QActor ledcontrol context ctxRobot{
    Plan init normal[  ]
    transition stopAfter 600000
        whenMsg ledBlink -> doBlink
    finally repeatPlan

    Plan doBlink resumeLastPlan[
        println(’LED BLINKING’) //asynch action TODO
    ]
    transition whenTime 500 -> doBlink
        whenMsg ledOff : ledOff do{
            println(’LED (msg) OFF’) // afterwards, resumes init
        }
}
```

Listing 1.25. robot.qa: ledcontrol

7.4 The new system (model1)

At this stage, the system can be represented as in the following, informal picture:



Note that:

1. even within a single node we can organize the software as a set of actors, rather than a set of objects: the *micro-level* mirrors the *macro-level*;
2. the architecture of the system is gradually evolving into a `project architecture` rather than a pure logical architecture. However, the main constrains imposed by the previously defined logical architecture are maintained.

7.5 Executing/testing the system

To execute the distributed system we have to execute a sequence of actions:

1. Activate the `W-Env`
2. Activate the robot, by running the generated program `it.unibo.ctxRobot.MainCtxRobot`;
3. Activate the console, by running the generated program `it.unibo.ctxConsole.MainCtxConsole`.

The same result can be achieved by introducing a proper `Testing activity`. In the `setUp`, we launch the robot:

```
package it.unibo.ft2018.model1;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;
import org.junit.Before;
import org.junit.Test;
import alice.tuprolog.SolveInfo;
import it.unibo.qactors.QActorUtils;
import it.unibo.qactors.akka.QActor;

public class TestModel1 {
    @Before
    public void setUp(){
        try {
            //ASSUMPTION: W-Env is working
            System.out.println(" ***TEST*** setUp START THE ROBOT");
            it.unibo.ctxRobot.MainCtxRobot.initTheContext();
        } catch (Exception e) {
            fail( e.getMessage() ) ;
        }
    }
```

Listing 1.26. `TestModel1.java`: the setup

The console is not launched, since its behavior can be easly simulated.
In the `test`, we:

1. activate the `robotmind` actor by sending to it the message `startClean` with the help of the class `QActorUtils` provided by the `QActor` Software Factory;
2. prepare the generation of a (simulated) temperature value higher than the limit, after `3 secs`;
3. look at the `Knowledge-Base` of the `worldobserver` after `1 sec` and perform our assertions (robot running and led blinking);
4. look at the `Knowledge-Base` of the `worldobserver` after other `4 secs` and perform our assertions (robot stopped and led off).

```
    @Test
    public void testWork(){
        try {
            QActor worldobserver = getQActorRef("worldobserver");
            //SIMULATE the effects of pressing of START button
            System.out.println(" ***TEST*** testWork SIMULATE the effects of pressing of START button");
```

```
            QActorUtils.sendMsg(worldobserver, "robotmind", "startClean", "startClean(init)");
            //SIMULATE the emission of a temperature event OUT the limits after 3 secs
            QActorUtils.emitEventAfterTime(worldobserver, "tester", "temperatureState", "temperatureState(30)", 3000);
            Thread.sleep( 1000 );
            //LOOK AT the worldobserver Knowledge-base (while running)
            SolveInfo solRobot = worldobserver.solveGoal("robotState(V)");
            String robotState = solRobot.getVarValue("V").toString();
            /*
             * robot: running | led: blinking
             */
            System.out.println(" ***TEST*** testWork robotState=" + robotState );
            SolveInfo solLed  = worldobserver.solveGoal("ledState(V)");
            String ledState   = solLed.getVarValue("V").toString();
            System.out.println(" ***TEST*** testWork ledState=" + ledState );
            assertTrue( "testWork", robotState.contains("running") && ledState.contains("blink") );
            /*
             * WAIT for a while ...
             */
            Thread.sleep( 4000 ); //now temperatureState(30)
            //LOOK AT the worldobserver Knowledge-base (while stopped)
            /*
             * robot: stopped | led: off
             */
            solRobot  = worldobserver.solveGoal("robotState(V)");
            robotState = solRobot.getVarValue("V").toString();
            System.out.println(" ***TEST*** testWork robotState=" + robotState );
            solLed   = worldobserver.solveGoal("ledState(V)");
            ledState = solLed.getVarValue("V").toString();
            System.out.println(" ***TEST*** testWork ledState=" + ledState );
            assertTrue( "testWork", robotState.contains("stopped") && ledState.contains("off") );
//          Thread.sleep( 2000 ); //Gives the time to end
        } catch (Exception e) {
            System.out.println(" ***TEST*** testWork ERROR " + e.getMessage() );
            fail( e.getMessage() ) ;
        }
    }
```

Listing 1.27. `TestModel1.java`: the test

The operation `getQActorRef` is introduced to acquire (with the help of the class `QActorUtils`) the reference to the actor named `worldobserver`. The reference to the `worldobserver` allows us to look at its knowledge-base by means of the built-in method `solveGoal`. Of course, this is not the correct way to interact with an actor, since we exploit the fact that it is also a POJO. However, this kind of 'impure' access is acceptable during a testing phase, whose code will be kept outside the deployment.

8 The primary use-case ([R-floorClean])

The principal use-case for our system is represented by the requirement [R-floorClean]. This main system requirement implies that the robot movements cover all the floor, without leaving no free-area dirty. For free-area we intend a zone of the floor not covered by obstacles.

Before any coding activity, a first analysis of the problem is required, in order to detect the different situations to manage and to define a first, logical architecture of the (sub)system that will realize this requirement. This analysis phase is fundamental, but we must foresee that it will be incomplete, since

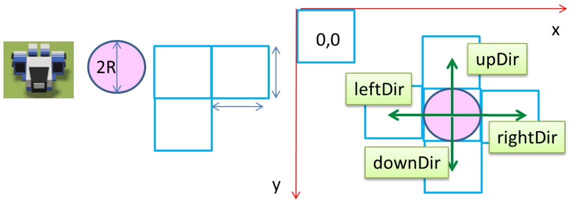
> it can be difficult, if not impossible, to foresee all possibile situations during a problem analysis phase, since these sitatution usually depend not only by the problem but by our solution itself. Thus, we are prepared to follow an incremental analysis, design and development cycle, based on the motto '*design-a-little, code-a-little, test-a-little*'.

8.1 Problem analysis

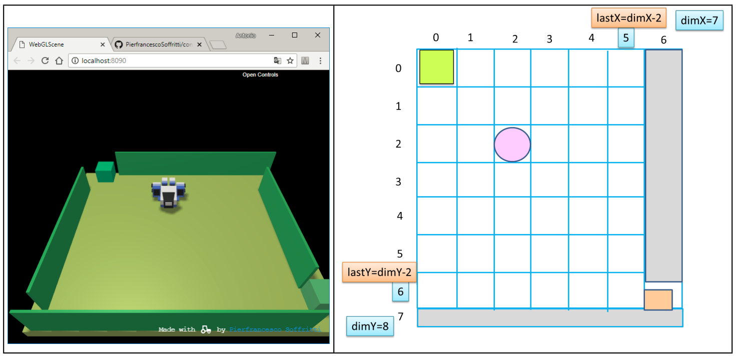
Let us start from the basic assumptions and with the definition of some internal terminology:

1. `Robot Moves`:
   The robot can execute four basic moves: `forward`, `backward`, `turnLeft` and `turnRight`. These moves will be represented by the term `move(M)` with :
   - `M=a` for `turnLeft`.
   - `M=d` for `turnRight`.
   - `M=w` for `forward`.
   - `M=s` for `backward`.

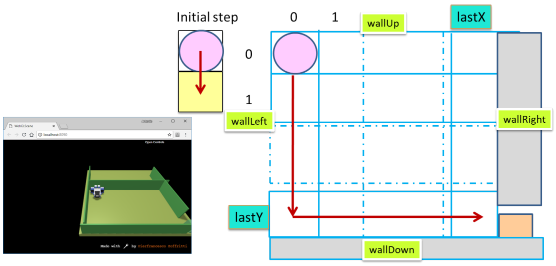
   :
2. `Robot size`:
   A robot can be enveloped in a circle of radius=`R`. A cell is a square of size `2R`.



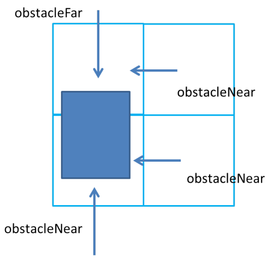
3. `Robot basicStep`:
   The robot should perform a sequence of basic move-steps. A `basicStep` is a `move(w)` with speed `v` and time `t`, so that `2R=vt`. The correct value of `v` is set by a proper `calibration phase`. In this way, after a `basicStep` the robot moves from a cell to an adjacent cell.
4. `Directions`:
   When executing a `basicStep` from a cell `(X,Y)` to an adjacent cell, the robot moves along one of four possible directions (i.e. no 'diagonal' movements are considered) denoted as follows:
   - `upDir`: when the robot executes `move(w)` from `(X,Y)` to `(X,Y-1)`.
   - `downDir`: when the robot executes `move(w)` from `(X,Y)` to `(X,Y+1)`.
   - `leftDir`: when the robot executes `move(w)` from `(X,Y)` to `(X-1,Y)`.
   - `rightDir`: when the robot executes `move(w)` from `(X,Y)` to `(X+1,Y)`.
5. `The grid`:
   The floor can be logically partitioned into a grid of square cells of size `2R`. With reference to the axis shown in the picture above, let us denote as `dimX` the number of cells on the `x` axis and as `dimY` the number of cells on the `y` axis.



6. `Discovery`:
The topology of the floor is not known, i.e. the robot has to 'discover' it while moving, by knowing only that:
   - it starts from cell `(0,0)` moving along `downDir`;
   - it can start moving along the `downDir` direction or along the `rightDir` direction;
   - it is located on the last row of the grid, when (in absence of mobile obstacles) it is detected by `sonar2`.
7. `Walls`:
A conventional room has four walls, that can be named as `wallUp`, `wallRight`, `wallDown`, `wallLeft`.
8. `First intent`:
The first intent (let us call it `intent0`) of the robot could be that of reaching the last row of the floor grid as soon as possible. Thus, it could start by going down, with the goal to detect the value of `lastY=dimY-2` and then turn right with the goal to reach `sonar2` and fix the value of `lastX=dimX-2` and therefore the global topology of the map. For example:



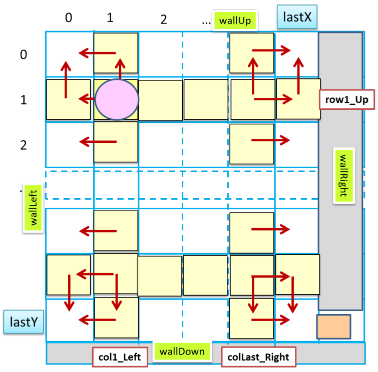
9. `Obstacles (far and near)`:
A `basicStep` (including the first one) cannot be completed if the destination cell is occupied by an obstacle. Since an obstacle usually does not occupies a cell in a compete way, we say that:
   - an obstacle is `far`, if the robot stops its `basicStep` after a prefixed extent of the maximum move time `TM=2R/v`. For example, if the move time `T=TM*90%`, we can say that the obstacle is far.
   - an obstacle is `near`, if it is not far.



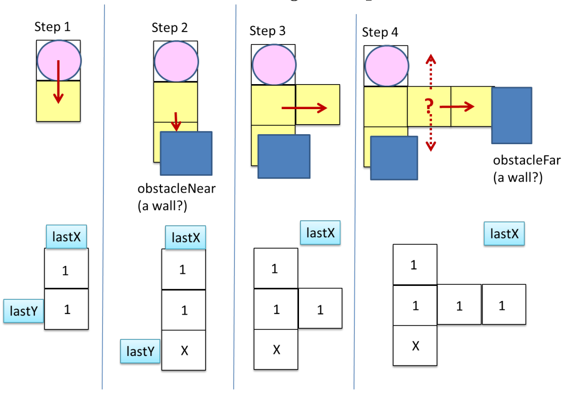
10. `The gridInvariant`:
The `gridInvariant` is a condition that must be preserved during the robot activity. It states that: after `basicStep`, the robot must be always aligned with the grid, even when the `basicStep` cannot be completed for the presence of an obstacle.

11. Critical moves:
A wall of the room is perceived by the robot as an obstacle. When the robot is in a cell near a wall and it moves towards the wall, we identify a criticalMove, since the detection of an obstacle in this case has to be handled in a different way with respect to a conventional obstacle on the floor. In order to capture the critical moves, we introduce the following terms:
   - col1_Left: the robot is on the column (_,1) and it moves in **left** direction.
   - colLast_Right: the robot is on the column (_,lastX) and it moves in **right** direction.
   - row1_Up: the robot is on the row (1,_) and it moves in **up** direction.
   - rowLast_Down: the robot is on the column (_,lastY) and it moves in **down** direction.

   For example, the robot in the cell (1,1) in the picture hereunder is in the col1_Left critical case if it must execute a basic step in leftDir direction. If it has to move in downDir direction, the move is not critical.



12. The robotmind:
Each move of the robot must be decided by proper controller, that will be referred in the following with the term robotmind. The task of the robotmind is to define a sequence of robot moves (Subsection 1) to reach the first dirty cell (with the global goal to cover all the floor).
When the topology of the grid is still unknown, the robotmind can only decide to perform a basicStep (Subsection 3) along the current direction. If such a move can be completed with success, the robot has 'discovered' a new cell. If the basicStep cannot be completed, the robot has found an obstacle. If such an obstacle is permanent, the corresponding cell has to be marked in a proper way.
Let is consider the following example:



   (a) The *step1* is completed with success. Thus, the 'mind' can state that (see Subsection 5): `dimX=1, dimY=2`.
   (b) The *step2* fails. Thus cell (0,2) is marked as 'obstacle' (at the moment we suppose that all the obstacles are permanent, by ignoring the problem of mobile obstacles).
   (c) The *step3* is completed with success. Thus, the 'mind' can state that (see Subsection 5): `dimX=3, dimY=2`.
   (d) The *step4* poses to the mind the problem of the next direction to follow. If it decides to move right, the step fails, but now with the presence of a 'far' obstacle (Subsection 9). The mind could decide to remain in the cell.

8.2 Remarkable points

From the problem analysis we can highlight the following points:

- `Planning`: the mind must take its decision about the next move without loosing the knowledge about the other possible paths (in fact the requirement is to cover all the floor). Thus, the mind should behave as a planner.
- `Obstacle detection`: the robot can detect an obstacle through the sonar put in its front (`sonarRobot`). More precisely, a real robot 'perceives' an obstacle when the distance detected by the `sonarRobot` is less than some prefixed value. Thus, in order to perceive an obstacle, the robot must complete a fraction of the `basicStep`. that should be `compensated` in order to preserve the `gridInvariant` (Subsection 10).
- `Compensation`: the fraction of the `basicStep` completed by a robot before the detecion of an obstacle should be `compensated` in order to preserve the `gridInvariant` (Subsection 10).
- `Rotation`: in the case of a `far-`obstacle, the compensation action required to assure the `gridInvariant` is almost equal to a `basiStep`. However, if the robot remains in the cell of a far-obstacle, its `sonarRobot` will continue to perceive the obstacle. In order to avoid such a situation, we could deactivate the `sonarRobot`. As an alternative, we could rotate the robot of `180` degrees, in order to put a free space in front of the robot.

8.3 A first logical architecture

From our first-level problem analysis, we can say that our system should be composed at least of two main components:

- `player`: an actor that works as an actuator of robot move-commands;
- `robotmind`: an actor that implements the business logic.

The distinction between a 'mind' and a 'body' (the player) in not motivated by philosophical reasons; rather the introduction of a `player` is motivated by the pragmatic reason to encapsulate into a component all the (technological) details related to the interaction with robot, that could be real or virtual and built in very different ways.

8.3.1 The mind and the player (as actors) .

The important point now is to fix the 'interface' between these two components. Logically, these components could be modelled as *functions*, *objects*, *processes*, *tasks*, *agents*, etc. The most probable choice, at the current state of the art in software engineering, is to model each component as an **`object`** (written in `Java` or in `C#/C++`) that interact by means of procedure (method) calls. For example, the `robotmind` could be the 'master' that calls the `player` to move the robot.

With such an approach we envisage two main problems:

- we are induced to use the same programming language to write the code of each component;
- a (non-trivial) extra effort in programming is required if each component must run on a different computational node.

Since we want to design and build an heterogeneous, distributed system, by allowing wide flexibility in the implementation technology of each component, our choice here is to model each component as an `actor` and to base the interaction between the components on *messages* or on *events*. This choice means that the interface of each components has to be defined by defining the messages/events that each component is able to handle, rather than in term of methods.

8.4 Messages and events

According to the `QActor` metamodel, we will introduce the following meaning of terms:

- a `message` is defined as information sent in `asynchronous` way by some source to `some specific destination`. For *asynchronous* transmission we intend that the messages can be '**buffered**' by the infrastructure, while the '**unbuffered**' transmission is said to be `synchronous`;
- an `event` is defined as information emitted by some source without any explicit destination.
  Both messages and events can be *sent/emitted* by a component of our software-system or by entities external to the system.

A (`QActor`) message is syntactically represented - at implementation level - as follows:

```
msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
```

where:

MSGID	Message identifier
MSGTYPE	Message type (e.g.: dispatch,request,event)
SENDER	Identifier of the sender
RECEIVER	Identifier of the receiver
CONTENT	Message payload
SEQNUM	Unique natural number associated to the message

An event implementation is represented as a messages with no destination (`RECEIVER=none`):

8.4.1 Message handling .

In the `QActor` meta-model, a message does not force the execution of code: a message `m` sent from an actor `sender` to an actor `receiver` can trigger a `state-transition` in the `receiver`. If the `receiver` is not 'waiting' for a transition including `m`, the message is enqueued in the `receiver` queue.

The `msg/6` pattern can be used to express `guards` to allow conditional evaluation of actions.

8.4.2 Event handling .

The occurrence of an event can immediately put in execution some code devoted to the management of that event. We qualify this kind of behaviour as `event-driven` behaviour, since the event 'forces' the execution of code.

An event can also trigger state transitions in components, usually working as finite state machines. We qualify this kind of behaviour as `event-based` behaviour, since the event is 'lost' if no actor is in a state waiting for it.

In a `QActor` system, the occurrence of an event activates, in `event-driven` way, all the ***EventHandlers*** declared in actor *Context* for that event.

8.4.3 Messages or Events? .

The choice whether use a *message* or an *event* to model how the information is exchanged among components is an essential and critical aspect of the software design of a distributed systems. Generally speaking, we can say that:

- a `message` is the natural choice when we known that information must flow from a component `A` to a precise destination component `B`. Message exchange is the most natural way to face the transition from object-based systems to actor-based systems. In fact an actor can be viewed as an active object that works by handling the messages it receives in FIFO way. The `QActor` model however, gives to an actor the opportunity to be expressed as a classical Finite State Machine (`FSM`), i.e. to relate the set of messages to handle to its current internal state (and not to the received message sequence);
- an `event` is the natural choice when a component does not known a-priori the components that are interested in the information that it emits. To overcome the problem of event-loss, a possible technique in `QActor` systems, is to write an `EventHandler` that converts an event into a message.

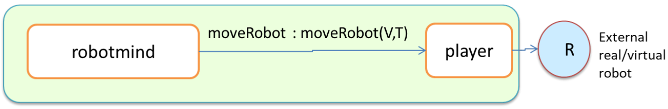
8.5 Mind/player message-based interaction

From the logical point of view, the **robotmind** is a master that must use the **player** to move the robot. This can lead us to base the interaction between these two components on messages rather than on events. Thus, let us introduce the following message declaration:

```
Dispatch moveRobot : moveRobot(V,T) //V= w | s | a | d | h , T = move-time in msec
```

Since we declare `moveRobot` as a *dispatch*, a `fire-and-forget` form of interaction is assumed for the exchange of information between the **robotmind** and the **player**, in which the player is the receiver.

A *request-response* form of interaction between two components of a distributed system is also quite common and often necessary. We will see an example of this case in Subsection 9.15.



For example, if **robotmind** send) to the player the dispatch `moveRobot:moveRobot(a,400)`, the **player** will execute the robot-move **a** (`turnLeft`) for 400 msec.

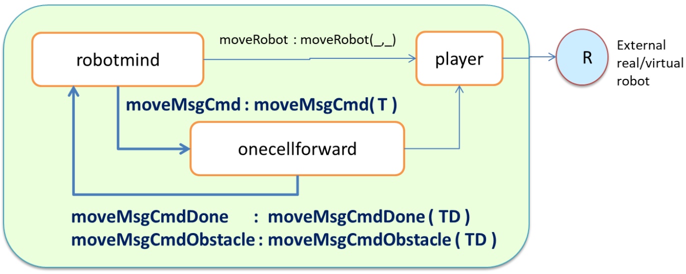
8.5.1 Types of message-based interaction

In general, we distinguish among a set of different forms of message-exchange:

- `dispatch`: ...
- `invitation`: ...
- `request`: ...

8.5.2 The basicStep .

From Subsection 9 of Subsection 8.1, we known that the message `moveRobot:moveRobot(w,_)` is a `basicStep` that could be not completed for the presence of an obstacle. To keep the **robotmind** free from the details of obstacle detection[10], the execution of a `basicStep` could be delegated to another component (actor) that performs the action and tells to the mind the result (a success or an obstacle detection).



The interaction between the **robotmind** and the new component (named `onecellforward`) is now modelled by a set of dispatches defined as follows[11]:

```
Dispatch moveMsgCmd         : moveMsgCmd(TF)         //TF = time to go forward
Dispatch moveMsgCmdDone     : moveMsgCmdDone(TD)     //TD = time of effective execution
Dispatch moveMsgCmdObstacle : moveMsgCmdObstacle(TD) //TD = time after obstacle detection
```

Our assumption now is that the `onecellforward` component/service is also able to detect and avoid the mobile obstacles, so that all the `moveMsgCmdObstacle` answers given to the **robotmind** are related only to fixed obstacles.

[10] An also according to the motto 'divide-et-impera'.

[11] We use here a set of dispatches for the sake of simplicity, but the interaction is logically a `request-response` and the `onecellforward` can be logically conceived as a specialized (micro)service.

8.6 Planning and doing

From Subsection 12 of Subsection 8.1, we know that the main task of the `robotMind` is to plan the sequence of robot moves that:

- allows the discovery of the topology of the floor;
- assures that the robot covers all the free-areas of the floor.

Thus, the `robotMind` logically operates in two phases:

1. first, it detects a proper sequence of moves according to the current state of the world;
2. then, it actuates the sequence, with the caution the the execution of a `basicStep` works also as a '*discovery action*' that could lead to the interruption of the execution of a planned move-sequence for the presence of a fixed obstacle.

Once again, it is wise to encapsulate the planning activity into a specialized component, that can be named as the `planner`. At this level of our analysis, the details of the behaviour of the `planner` can be ignored. The important point is to establish the nature of the `planner` and interaction with the `robotMind` .

8.6.1 The planner .

The `planner` could be modelled as an *object* or as *actor* or even as a remote, specialized `micro-service`, embedding its own representation of the world, obviously consistent and synchronized with the 'real' situation of our room.

For example, a possible representation of the state of the world within the `planner` could be the map of the floor (`curDir` is the current direction of the robot):

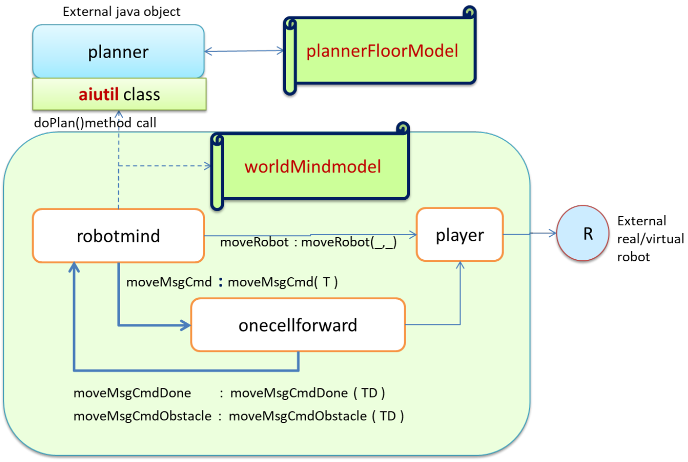
```
   0  1  2  3  4  5
0  1, X, 0, 0, X, 0, X,   r means robot, curdir=downDir
1  1, r, 1, 1, 1, 0, X,
2  1, 1, 1, 1, 1, 1, X,
3  X, 1, 1, 1, 1, 1, X,   X means obstacle
4  0, 1, 0, 0, X, 1, X,   1 means cell cleaned
5  0, 1, 0, 0, 0, 1, X,   0 means cell dirty
6  0, 1, 1, 1, 1, 1, X,
7  X, X, X, X, X, X, X,
```

In any case, we can assume that the main result of the `planner` consists in the specification of a sequence of moves expressed as introduced in Subsection 1. For the example above, the sequence proposed by the planner to move the robot from the current cell `(1,1)` to the cell `(2,0)`, could be:

```
move(a).   //turnLeft, so that curdir=rightDir
move(w).   //forward, to go in cell (2,1)
move(a).   //turnLeft, so that curdir=upDir
move(w).   //forward, to go in cell (2,0)
```

8.6.2 The class aiutil .

Our software factory has already developed a `planner` in Java as a conventional object. Thus, in the following we will reuse such a planner, by exploiting an utility class (named `aiutil`) to adapt the work of this planner to our needs.



The class **aiutil** exposes a set of static methods, including:

```
public static void doPlan( ) throws Exception; //builds a plan
public static void doMove( String move ) throws Exception; //Modifies the planner map according to the result of a move
public static void showMap( ) throws Exception; //Prints the current state of the map
...
```

Other methods will be probably introduced later, since we have to maintain the model internal to the planner consistent with the model of the world of the `robotMind` (that should reflect the real world).

8.6.3 The actuation .

The actuation phase of the `robotMind` must execute the sequence of moves provided by the planner. Since a `basicStep` is delegated to the actor `onecellforward`, the actuation phase must properly handle the answer messages (Subsection 8.5.2) sent by `onecellforward`. More specifically:

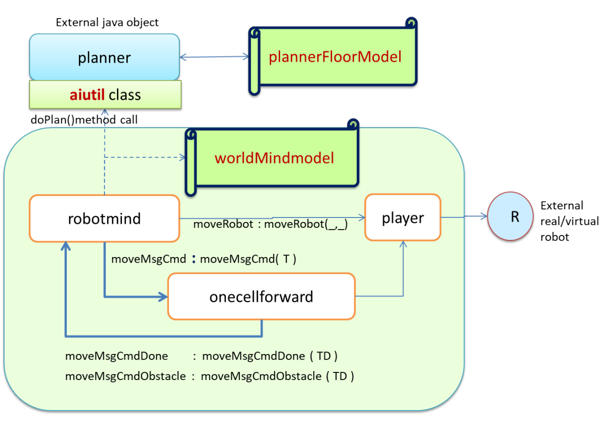
– the answer `moveMsgCmdDone:moveMsgCmdDone(TD)` means that the `basicStep` has been done with success. Thus the actuation phase must inform the planner by calling `aiutil.doMove("w")` and then continue with the next move or with a re-planning, if the sequence is terminated.
– the answer `moveMsgCmdObstacle:moveMsgCmdObstacle(TD)` means that the `basicStep` has found an obstacle after `TD` msecs. Now, the actuation phase must:
  1. check if the obstacle in on the last row of the grid. This case means that the `wallDown` has been detected and the first intent (Subsection 8 of Subsection 8.1) can been achieved. Thus, the strategy now becomes to proceed along the row, until the `wallRight` is found.
  2. check if the `basicStep` was a critical move (Subsection 11 of Subsection 8.1) and, if so, operate in the proper way;
  3. check if the obstacle is a wall (Subsection 7 of Subsection 8.1) or a conventional obstacle;
  4. in case of a conventional obstacle, check if it is *near* or *far* (Subsection 9 of Subsection 8.1) and then operate in the proper way.

8.6.4 The importance of (executable) models .

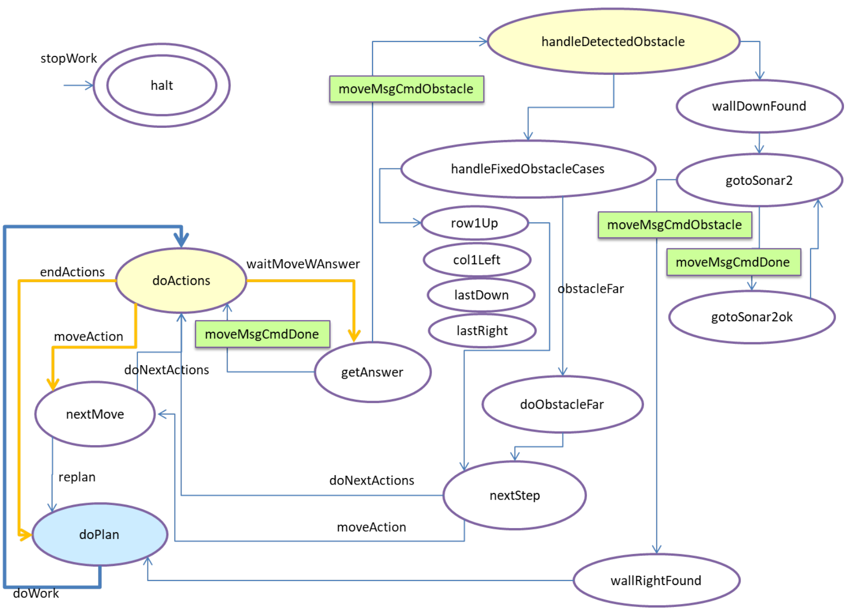
In practice, the actuation phase has the responsibility to implement each one of the situations highlighted in our first problem analysis. Since this phase can be quite complicated, the possibility to express it with a formal, executable model is important, in order to allow the immediate testing of our specification, by starting from very simple cases and then introducing more complex situations in incremental way (according to the motto '*design-a-little, code-a-little, test-a-little*').

The aim is to achieve step-by-step the final complexity of a system covering all the possible situations, and to improve - if it will be the case - our analysis with the experience gained after each step.

We report here a possible result of this work by showing the states of the `robotmind` (without considering messages related to `invalidconditions` and `stop`).



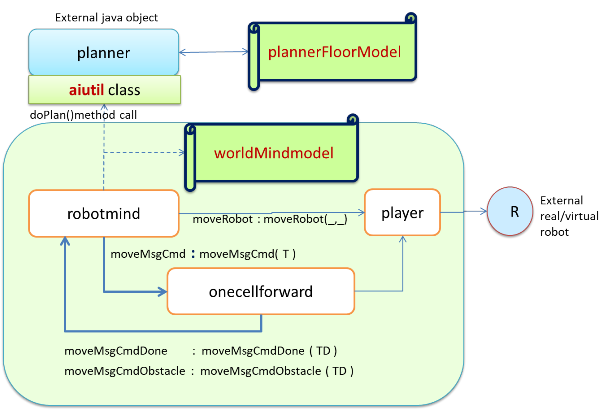

```
System protoCleaner
//Declaration of Messages and Events

Context ctxProto ip [ host="localhost" port=8030]
//Declaration of EventHandlers

QActor onecellforward context ctxProto{
...
}

QActor robotmind context ctxProto {
//Calls the external planner
...
}

QActor player context ctxProto {
...
}
```



The reader should attempt to create his/her step-by-step path for a solution able to cover all the situations described in the analysis Subsection 8.1) and in all the other situations that arise from the solution itself.

9 A first prototype (the main architecture)

Our first prototype is a system composed of three actors: `robotmind`, `player` and `onecellforward` Now it is time to transform the informal picture of Subsection 8.6.2 in formal, executable code.




```
System protoCleaner
//Declaration of Messages and Events

Context ctxProto ip [ host="localhost" port=8030]
//Declaration of EventHandlers

QActor onecellforward context ctxProto{
...
}

QActor robotmind context ctxProto {
//Calls the external planner
...
}

QActor player context ctxProto {
...
}
```

Each actor could run on its own machine; at the moment however, we assume (to make debugging easier) that they work on the same machine, represented by a **Context** named `ctxProto`:

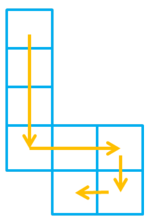
9.1 The (actor) knowledge-base

The external `planner` provides, when called, a sequence of moves, each written in the following form:

```
move(M).            with M= w | a | d
```

An example of moves could be:

move(w).
move(w).
move(w).
move(a).
move(w).
move(w).
move(d).
move(w).
move(d).

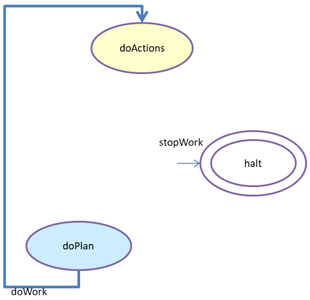


Technically, we assume here that the sequence of moves is represented as a sequence of facts that expressed in `Prolog` syntax. constitute a `knowledge-base`

Each `QActor` owns its private `knowledge-base` (referred as `ActorKB`) that can be accessed by means of declarative `Prolog` rules. As a consequence, some specific part of the behaviour of a `QActor` can be expressed in a *declarative style*, that will help us to make our specifications simpler to understand and to modify.

9.2 The robotmind as a FSM

The behaviour of the `robotmind` can be expressed as a `QActor` that works as `Finite State machine` with two main states: one state (`doPlan`) that defines a sequence of moves)and another state (`doActions`) that performs the actuation phase:



The activity of the `robotmind` starts from `doPlan` and then switches to the state `doActions`. The state `halt` is a final state that can be reached from any state when the message `stopWork` is received by the `robotmind`.

```
Dispatch stopWork : stopWork(V)  //V=<stop details> e.g. stopByuser, nomoves, etc.
```

In the next pictures, the transitions to the `halt` state will not be shown, for the sake of clarity.

9.3 Self-messaging

The state `doPlan` of the `robotmind` defines a plan and then forwards to the `robotmind` itself the dispatch `doWork`.

```
Dispatch doWork : doWork
```

`Self-messaging` is a common pattern in actor-based systems, since the behaviour of an actor is basically determined by the message that the actor receives.

A state could switch to another state with a prefixed instruction (e.g. the `QActor` language provides the `switchTo` `<nextState>` operation). In this case however, the actor remains insensitive to messages. For example, the `robotmind` would not be able to react to a `stopWork` message sent in the meantime. Thus, `self-messaging` is also a useful technique to assure more re-activeness.

9.4 The state doPlan

To be more precise (and, pragmatically, also more operative) let us now introduce an executable specification in the `QActor` language:

```
    Plan doPlan[
        [ !? endOfWork ] selfMsg stopWork : stopWork(nomove);
        else {
            //javaRun it.unibo.exploremap.program.aiutil.doPlan();
            selfMsg doWork : doWork
        }
    ]
    transition whenTime 2000 -> handleError
        whenMsg stopWork -> handleStop , //first to be checked
        whenMsg doWork  -> doActions
```

Note that the state makes us of `guard condition` (`[!? endOfWork]`) with reference to the `ActorKb` (see Subsection 9.1) to terminate its task (e.g. when the `planner` has no move to propose).

As said in Subsection 8.6.1, the planned moves are built by using the utility class `aiutil`; however, the call to this class is commented in the previous specification, since we want to test our software by a *pre-built sequence* of actions. To achieve this goal we have simply to populate the `ActorKB` of the `robotmind` with a sequence of moves like that shown in Subsection 9.1.

9.5 The state doActions

The state doActions 'consumes' the moves in the order of the move-sequence provided by the `planner`. It executes on his own behalf the moves a `(turnLeft)` and d `(turnRight)`, while it delegates the the actor onecellforward the execution of the move w `(basicStep)`. The actor onecellforward returns an answer that denotes:

– the successful completion of the `basicStep` (message moveMsgCmdDone) that gives the effective eecution time of the move. In fact, the effective duration of a `basicStep` is always a little bit higher (randomly) than the prefixed time `2R/v` (Subsection 3).

```
Dispatch moveMsgCmdDone    : moveMsgCmdDone(T)   //T = w-move execution time
```

In this case, the `robotmind` continues with the next move of the sequence;

– the detection of an obstacle (message moveMsgCmdObstacle) that returns an execution time lower than the prefixed time for the `basicStep`.

```
Dispatch moveMsgCmdObstacle : moveMsgCmdObstacle(T) //T = w-move execution time before the obstacle
```

In this case, the `robotmind` must handle the obstacle and discard the previous move-sequence.

```
Dispatch moveMsgCmd        : moveMsgCmd(TF) //TF = execution time of the basicStep
Dispatch waitMoveWAnswer   : waitMoveWAnswer
```

9.6 doActions as a dispatcher

In our `QActor` specification, doActions selects the first move to execute with the guard moveTodo(M) that binds (unifies) the variable `M` with the first current move of the plan.

```
    Plan doActions[ //javaOp "debugStep()";
        [ !? move(M) ] println( robotmind_doActions(M) );
        [ not !? move(M) ] selfMsg endActions : endActions ;
        removeRule moveSelected;
        ReplaceRule moveDuration(_) with moveDuration(moveWDuration(0));

        [ !? moveTodo(a) ] {
            selfMsg moveAction : moveAction(a, false) //moveLeft and continue
        } ;
        [ !? moveTodo(d) ] {
            selfMsg moveAction : moveAction(d, false) //moveRight and continue
        };
        [ !? moveTodo(o) ] {
            [ !? timeForForward(T) ]
                  forward onecellforward -m moveMsgCmd : moveMsgCmd(T);
            selfMsg waitMoveWAnswer : waitMoveWAnswer
        };
        [ !? moveTodo(w) ]{
            [ !? timeForForward(T) ]
                  forward onecellforward -m moveMsgCmd : moveMsgCmd(T);
            selfMsg waitMoveWAnswer : waitMoveWAnswer
        };
        [ !? move(M) ] println( robotmind_doActions( nextMove(M)) )
    ]
    transition stopAfter 60000
        whenMsg stopWork         -> handleStop , //first to be checked
        whenMsg moveAction       -> nextMove,
        whenMsg waitMoveWAnswer  -> waitForwardMoveAnswer,
        whenMsg endActions       -> doPlan //all actions done
```

According to the move given by the guard moveTodo(M), the state `doActions` can prepare the next computational step by sending to itself the message moveAction or by sending to the actor onecellforward the message moveMsgCmd.

The simulated moves are specified with a sequence of facts in TuProlog syntax:

```
move( M ). %% M= w | o
```

The move M=o is introduced to simulate an obstacle.

The transition related to the moveAction message occurs only if no stopWork message is present. If so, the state doActions delegates the work for the moves a (turnLeft) and d (turnRight) to another state (nextMove) that works as a *reusable behaviour* (see Subsection 9.13). The move w (forward) is delegated to the actor onecellforward (as said in Subsection 8.5.2) by sending to it the dispatch moveMsgCmd.

9.7 Declarative code

The rule moveTodo(M) is written as a Prolog rule, directly in the model:

```
QActor robotmind context ctxProto {
Rules{
    eval( eq, X, X ).     //since we have syntax limitations
    //moveSelected is retracted when doActions starts
    moveTodo(M) :- moveSelected,!,fail.
    moveTodo(M) :- //moveSelected does not exist: we can goon
        move(M1), !, //M1 is the first move to do
        eval(eq,M,M1), !,
        assert(moveSelected),
        retract( move(M1) ).
}
```

Since the Rules section in the QActor language supports a quite limited Prolog-like syntax, the declarative specification section of an actor can be written (in full TuProlog syntax) into a file that can be dynamically loaded (*consulted*) in the ActorKb by the actor itself (usually in its initial state).

9.8 waitForwardMoveAnswer

The main task of the state waitForwardMoveAnswer is to handle the message sent as answer by the actor onecellforward:

```
    Plan waitForwardMoveAnswer[ ]
    transition stopAfter 60000
        whenMsg stopWork         -> handleStop , //first to be checked
        whenMsg moveMsgCmdDone   -> actionDoneOk,
        whenMsg moveMsgCmdObstacle -> handleDetectedObstacle
```

If no stopWork message is present, the next state will be related to the completion of the basicStep or to the handling of an obstacle.

9.9 actionDoneOk

The state actionDoneOk updates the map of the room, since the robot has found and cleaned a new cell. Thus, it calls the operation doMove of the class aiutil (Subsection 8.6.2), that updates the map and the current position of the robot in the map.

```
    Plan actionDoneOk[
        onMsg moveMsgCmdDone : moveMsgCmdDone(T) -> ReplaceRule moveDuration(_) with moveDuration(T);
        [ !? moveDuration(T) ] println( actionDoneOk(T) ) ;
        //demo logMove(w);    //(A)
        [ !? timeToClean(T) ] delay T ;
        javaRun it.unibo.exploremap.program.aiutil.doMove("w")
    ]
    switchTo doActions
```

9.10 The current robot position

The current position of the robot in the planner map is also stored in the ActorKb of the robotmind, represented as a fact curPos/3

```
curPos(X,Y,DIRECTION)
```

9.11 Move history

The action `demo logMove(w)` at point `(A)`of the state `nextMove` of Subsection 9.9 show how we can log an history of the moves done by the robot, by simply storing new facts in its `ActorKb`.

```
logMove(plan(M) ):- !, curPos(X,Y,D),storeMove( X,Y,D,plan(M),0 ).
logMove(M):- curPos(X,Y,D),moveDuration(TD),!,storeMove(X,Y,D,M,TD ).
logMove(M):- curPos(X,Y,D), storeMove(X,Y,D,M,unknown).

value(mc,0).  //mc is the move-counter
storeMove(X,Y,D,M,TD) :- inc(mc,1,N), assert( movelog(N,X,Y,D,M,TD) ).
```

In this phase, these action are commented, since we are using the moves as a given input, rather than information to produce.

9.12 World models

After the execution of a successful `basicStep`, our software modifies two different models of the world: the map handled by the planner and the knowledge store in the `ActorKb` of the `robotmind`. Of course, these two models must be kept coherent and synchronized.

9.13 The 'reusable' state nextMove

.

The state `nextMove` is able to actuate the move `a` and the move `d`. It is also able to actuate the move `w` to compensate the movement that the robot has done before bumping into an obstacle. This, in order to maintain the `gridInvariant` (see Subsection 10).

The state is an handler of the dispatch `moveAction` defined as follows:

```
Dispatch moveAction : moveAction(M,REPLAN) //M=a|d|up|down|s, REPLAN=true|false
```

When argument `REPLAN` is true, the next step will be a switch to the state *doPlan*. Otherwise, the next state will be `doActions` again (to continue the execution of the sequence).

```
Plan nextMove[
  onMsg moveAction : moveAction(M,V) -> println(nextMove_moveAction(M,V));
  onMsg moveAction : moveAction( a, _ ) -> { //moveLeft
      //demo logMove(a);        //(A)
      [ !? timeForTurn(T) ] forward player -m moveRobot : moveRobot(a,T);
      [ !? timeForTurn(T) ] delay T;
      javaRun it.unibo.exploremap.program.aiutil.doMove("a")
  };
  onMsg moveAction : moveAction( d, _ ) -> { //moveRight
      //demo logMove(d);        //(B)
      [ !? timeForTurn(T) ] forward player -m moveRobot : moveRobot(d,T);
      [ !? timeForTurn(T) ] delay T;
      javaRun it.unibo.exploremap.program.aiutil.doMove("d")
  };
  onMsg moveAction : moveAction( w, _ ) -> { //move to compensate
      [ !? moveDuration(moveWDuration(T)) ] println( compensate(T));
      [ !? moveDuration(moveWDuration(T)) ]
            forward player -m moveRobot : moveRobot(w,T);
      [ !? moveDuration(moveWDuration(T)) ] delay T
  };
  onMsg moveAction : moveAction( _, true ) -> {
    println("NO retact move/1 IN SIMULATION PROTO");
      selfMsg replan : replan
  } else {
      selfMsg doNextActions : doNextActions
  }
]
transition stopAfter 60000
    whenMsg stopWork     -> handleStop ,//first to be checked
    whenMsg doNextActions -> doActions,
    whenMsg replan       -> doPlan
```

9.14 The actor `oncecellforward` with simulated moves

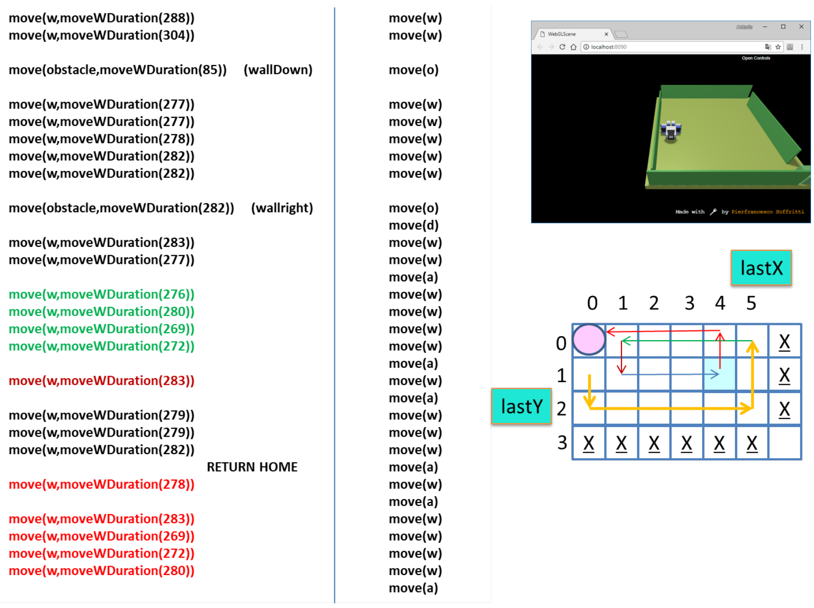
The actor `oncecellforward` logically works as follows: it waits for a dispatch (`moveMsgCmd`) from the `robotMind` and then executes a `w`-move by interacting with the player. Then it sends an answer message to the `robotMind` (see Subsection 9.5).

```
QActor oncecellforward context ctxProto{
    Plan initWork normal [
        //demo consult("protoTraces/logPcOk5ObsA.pl");
        demo consult("protoTraces/logPcNoObsOk.pl");
        demo buildSimulation;
        ReplaceRule moveWDuration(_) with moveWDuration(0)
    ]
    switchTo init

    Plan init[ println("         oncecellforward waits ...")]
    transition stopAfter 6000000
        whenMsg stopWork -> handleStop , //first to be checked
        whenMsg moveMsgCmd -> startWork

    Plan startWork[
        //printCurrentMessage;
        onMsg moveMsgCmd : moveMsgCmd( TF ) ->
                  ReplaceRule timeForForward(_) with timeForForward(TF);
        //SAVE THE CURRENT MESSAGE: it is lost when the state changes
        javaOp "storeCurrentMessageForReply()" //used by replyToCaller
    ]
```

Listing 1.28. `oncecellforward`

At this stage however, we do not want to introduce any robot, neither real or virtual, since our aim at the moment is just to test the logical behavior of our prototype. Thus, we suppose here to *simulate* the result of each move, by exploiting a proper knowledge-base *consulted* in the initial state of the actor.

For example, we could include insert into the `ActorKb` of `oncecellforward` the facts shown in the following picture:



The column on the right shows the sequence of moves statically planned for the `robotMind`, while the column on the left shows the values simulated for the duration of each `w`-move.

Instead of manually writing the simulated moves, we could generate in automatic way the facts above by starting from a proper source of information. In the current version, we use a set of generation rules (see point `(1)`) working on a file of facts (see point `(2)`) produced as the result of tracing the behaviour of our final system.

9.14.1 Execute the simulated move .

The state `doTheMove` 'consumes' the first fact `move/2` by means of the rule `execStep`, finds the duration `TD` of the move, and tells to the player to move the robot for the time `TD`.

```

Plan simulateMove [
    demo doStep;  //add the rule moveWDuration(T) for the first setp
    [ !? stepDuration(T) ] println( onecellforward_stepDuration(T));
    [ !? stepDuration(T) ] forward player -m moveRobot : moveRobot(w,T) ;
    delay 500; //to synch with the move moveWDuration(T) T could be < 0
    [ !? answer(ko) ]{ //rotate
         forward player -m moveRobot : moveRobot(a,400) ;delay 400;
         forward player -m moveRobot : moveRobot(a,400) ;delay 400
    }
]
```

Listing 1.29. `oncecellforward`: simulate move

The rule `execStep` :

- stores in the `ActorKb` a fact `moveWDuration(TD)`, where `TD` is the time of the move at that step;
- prepares also the answer to be given to the `robotmind`, by writing into the `ActorKb` a fact `answer(V)`, with `V=ok | ko`

```
execStep :- retract( move( V,moveWDuration(T) ) ), %% get first
   replaceRule( moveWDuration(_), moveWDuration(T) ),
   defTheAnswer( move( V,T ) ).

defTheAnswer( move( w,_ ) )     :- !, replaceRule( answer(_), answer(ok) ).
defTheAnswer(move( obstacle,_) ) :- replaceRule( answer(_), answer(ko) ).
```

Afterwards, `oncecellforward` evaluates the move duration (rule `stepDuration`) and sends the proper answer message.

The rule `stepDuration(T)` evaluates the time and returns always a positive value in `T`.

```
stepDuration(T) :- moveWDuration(T), T > 0, !.
stepDuration(T) :- moveWDuration(T1), T1 < 0, T is 0-T1.
```

Since the duration of a move can assume a negative value (to denote the detection of a far obstacle), `stepDuration` converts a negative time value into a positive value (that usually is less than the prefixed time for a `basicStep`).

9.14.2 Sending the answer .

The answer is sent by forwarding to the caller a proper message (see Subsection 9.5).

```

Plan simulateAnswer[
    [ ?? answer(ok) ] selfMsg stepAnswerOk : none;
    [ ?? answer(ko) ] selfMsg stepAnswerKo : none
]
transition stopAfter 60000
    whenMsg stepAnswerOk -> endMoveForward,
    whenMsg stepAnswerKo -> sendAnswerAfterCollision

Plan sendAnswerAfterCollision[ //obstacle far has duration negative
    [ !? moveWDuration(T) ]
          println(onecellforward_sendAnswerAfterCollision(T));
    javaOp "ignoreCurrentCaller()"; //set currentMessage = null;
    [ ?? moveWDuration(T) ] //reply to the saved caller
        replyToCaller -m
             moveMsgCmdObstacle : moveMsgCmdObstacle(moveWDuration(T))
]
switchTo init

```

```
    Plan endMoveForward[
        println("        onecellforward endMoveForward ");
        javaOp "ignoreCurrentCaller()"; //set currentMessage = null;
        [ ?? moveWDuration(TD) ]
           replyToCaller -m moveMsgCmdDone : moveMsgCmdDone( TD )
    ]
    switchTo init

    Plan handleStop[
       println("        The robot was STOPPED: no reply to the caller")
    ]
```

Listing 1.30. `oncecellforward`: simulate answer

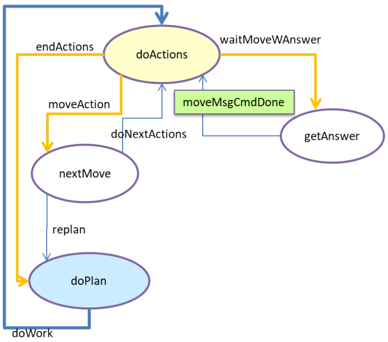
9.15 About request-response

Note that to send the answer to the proper caller, we have to remember the initial caller (by invoking via the `javaOp` operator the built-in operation `storeCurrentMessageForReply` in the state `startWork`. Moreover, we have to call the built-in operation `ignoreCurrentCaller` before using the `replyToCaller` operation in order to ignore (if it exists) the request of any other caller occurred in the meantime.

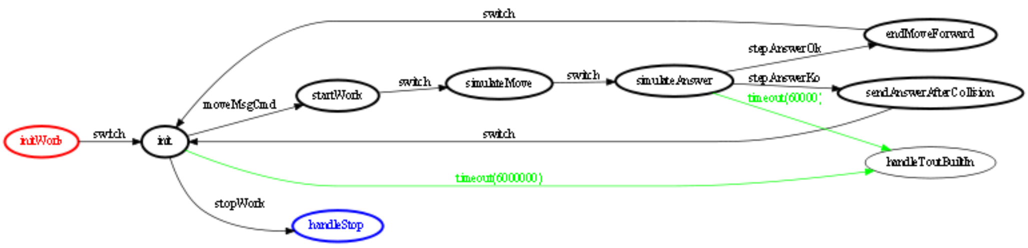
This 'trick' is the symptom that the concept of request-response will requires a proper way to be expressed in the modelling language and a proper implementation, in order to send answers to the correct actor and not to the last actor that has called the current one.

9.16 From code to diagrams

From the textual specification of Subsection 9.4 and Subsection 9.5, we can build a graphical picture of the `robotmind-FSM` under construction:



However, the `QActor` software factory can automatically build this kind of diagrams. For example, the picture hereunder shows the diagram generated form the specification of the `oncecellforward` of Subsection 9.17



In this way, we can overcome one of the main problems in using `UML`: a diagram is now just a visual representation of our executable textual-specification. Any change or re-factoring in our design will be done at textual level (i.e. at 'code level') with a consequent re-generation of support-code and (if needed) of the diagrams.

9.17 The final actor oncecellforward

Our goal now (perhaps a new SPRINT or part of a new SPRINT) is to modify the behavior of the actor oncecellforward introduced in Subsection 9.18, so to move the robot (in a simulated or in a real environment) and to handle the results of the move.

There are several new important points (see Subsection 8.2) to highlight and to discuss:

1. In order to detect an obstacle, we have to handle information generated by the sonarRobot that emits events sonarDetect:sonarDetect(V) with V that depends on the type of the robot. For a virtual robot, V could be bound to the name of the obstacle. For a real robot, V could be the distance of the obstacle, etc.
2. The events can be lost. While a real sonar emits a stream of events, a virtual sonar could emit an event only when the robot 'hits' an obstacle. To avoid the loss of sonarRobot events, we introduce an EventHandler (see Subsection 8.4.2) able to convert a sonarDetect event into a dispatch defined as follows:

```
Dispatch collisionDispatch : sonarDetect(V)
```

   The collisionDispatch has the same payload[12] of sonarDetect, so to be easily forwarded by the handler to an actor, in our case both to the onecellforward and to the robomind.

```
System robotCleaner

Event sonarDetect    : sonarDetect( V )
Dispatch collisionDispatch : sonarDetect(V)
...

Context ctxRobot ip [ host="localhost" port=8030]

EventHandler collisionevh for sonarDetect -print
{
    forwardEvent onecellforward -m collisionDispatch;
    forwardEvent robotmind   -m collisionDispatch
};
```

3. With the conversion of events into messages, information is no more lost, but it can be **replicated**. Thus, we must have the caution to discard repeated (if any) collisionDispatch messages and to limit as much as possible the generation of repetitions. To this end, we will adopt the technique to rotate the robot of 180 after an obstacle detection.
4. An obstacle can be fixed or mobile. The onecellforward component/service must be able (see Subsection 8.5.2) to detect and avoid the mobile obstacles, so that all the moveMsgCmdObstacle answers given to the robotmind are related only to fixed obstacles.

The starting part is quite similar to the previous one, with the difference that, after receiving a moveMsgCmd, the main intention of oncecellforward is to go in the state doMoveForward in order to execute the w-move required by the robotmind. However, our specification must assure that such a state-switch is not done if a stopWork message is arrived in the meantime or if there is some collisionDispatch still pending.

```
}
   Plan init normal [
       println("onecellforward init");
       ReplaceRule moveWDuration(_) with moveWDuration(0)
       //moveWDuration is set by aiutil.getDuration()
   ]
   transition stopAfter 6000000
       whenMsg stopWork -> handleStop , //first to be checked
       whenMsg moveMsgCmd -> startWork

   Plan startWork[
       //printCurrentMessage;
       onMsg moveMsgCmd : moveMsgCmd( TF ) ->
               println( onecellforward_moveMsgCmd( TF ) );
       onMsg moveMsgCmd : moveMsgCmd( TF ) ->
               ReplaceRule timeForForward(_) with timeForForward(TF);
       //SAVE THE CURRENT MESSAGE.
```

[12] This is an assumption in the current implementation of the QActor meta-model.

```
        //Otherwise the currentMessage is lost when the state changes
        javaOp "storeCurrentMessageForReply()" //used by replyToCaller
    ]
    switchTo clearPendingCollisions

    Plan clearPendingCollisions [
        //printCurrentMessage;
        //println("onecellforward_clearPendingCollisions: DO NOTHING")
    ]
```

Listing 1.31. `oncecellforward`: init

9.17.1 Doing the w-move .

The actuation of a `w`-move is done by starting a timer and then sending a `moveRobot` dispatch (without time-limits) to the `player`. The timer is required in order to evaluate the effective duration of the move.

```
    transition whenTime 100    -> doMoveForward
        whenMsg stopWork       -> handleStop , //first to be checked
        whenMsg collisionDispatch -> clearPendingCollisions

    Plan doMoveForward[
        [!? timeForForward( T )] println(doMoveForward_timeForForward(T));
        [ !? timeForForward( T ) ] //just for checking
//      forward player -m robotCmd : robotCmd("w");
```

Listing 1.32. `oncecellforward`: doMoveForward

The state `doMoveForward` starts its transition phase as soon as possible with the following goals:

1. handle a `stopWork` message, if it is present;
2. go to the state `endMoveForward` if the move time indicated by the robotMind in the `moveMsgCmd` message expires;
3. go to the state `checkMobileObstacle` if a message `collisionDispatch` is recived before the completion of the `basicStep`.

9.17.2 endMoveForward .

When in the state `endMoveForward`, the actor `oncecellforward`:

1. stops the robot;
2. evaluates the duration of the move by using they operation *getDuration* defined by the application designer in the class `aiutil`;
3. sends to the (original) caller (i.e. the `robotmind`)) the message `moveMsgCmdDone` with the duration of the move.
4. returns to the state `init`.

```
        forward player -m moveRobot : moveRobot(h,0);
        javaRun it.unibo.exploremap.program.aiutil.getDuration();
//Round the robot to avoid other collision messages
        forward player -m moveRobot : moveRobot(a,400);delay 400;
        forward player -m moveRobot : moveRobot(a,400);delay 400;
        [ !? obstacleBeyondCell ] {
            selfMsg obstacleFar : obstacleFar
        }else{
```

Listing 1.33. `oncecellforward`: endMoveForward

9.17.3 checkMobileObstacle .

The main task of the state `checkMobileObstacle` is to decide whether the obstacle is mobile or fixed. To this end, it:

1. stops the robot;
2. evaluates the duration of the move by using they operation `getDuration` defined by the application designer in the class `aiutil`;
3. undoes the move step and prepares the actor to retry the move after some time, in the hope that the obstacle disappears.

In the transition part, `checkMobileObstacle`

1. reacts to a pending message `stopWork` or a pending `collisionDispatch`;
2. returns to the state `doMoveForward` if no `stopWork` or `collisionDispatch` is present. In this way, it re-executes the state sequence `doMoveForward-checkMobileObstacle`

```
   Plan checkMobileObstacle[
      forward player -m moveRobot : moveRobot(h,0);
      javaRun it.unibo.exploremap.program.aiutil.getDuration();
//UNDO MOVE
       [ !? moveWDuration(T) ] forward player -m moveRobot : moveRobot(s,T);
       [ !? moveWDuration(T) ] delay T;
      delay 1000 //wait for a while...
   ]
   transition whenTime 500   -> doMoveForward  //RETRY
       whenMsg stopWork      -> handleStop
```

9.17.4 probableFixedObstacle .

The state `probableFixedObstacle` handles the detection of an obstacle that probably (no certainty can be assured) is a fixed obstacle in the room. It:

1. stops the robot;
2. evaluates the duration of the move by using they operation `getDuration` defined by the application designer in the class `aiutil`;
3. rotates the robot of `180` degrees, as said in Subsection 9.18
4. evaluates if the obstacle is far; in this case it converts the duration `TD` of the move in its opposite `-TD`, as a form of coding in the answer that the obstacle is far;
5. discards (if any) the pending `collisionDispatch` and then sends the answer to the caller (`robotmind`).

```
        forward player -m moveRobot : moveRobot(w,0)
    ]
    transition [ !? timeForForward( T ) ] whenTime T -> endMoveForward
         whenMsg stopWork        -> handleStop , //first to be checked
//          whenMsg collisionDispatch -> probableFixedObstacle //NO MOBILE
         whenMsg collisionDispatch -> checkMobileObstacle //MOBILE

/*
* Wait for a while and check again if we detect the obstacle
*/
    Plan checkMobileObstacle[
       forward player -m moveRobot : moveRobot(h,0);
       //javaOp "debugStep()";
       javaRun it.unibo.exploremap.program.aiutil.getDuration();
       [!? moveWDuration(T)] println(oncecellforward_checkMobileObstacle(T));
//UNDO MOVE
        [ !? moveWDuration(T) ] forward player -m moveRobot : moveRobot(s,T);
        [ !? moveWDuration(T) ] delay T;
//RETRY
       delay 1000; //moveWDuration will be reset
       ReplaceRule moveWDuration(_) with moveWDuration(0);
       javaRun it.unibo.exploremap.program.aiutil.startTimer();
       forward player -m moveRobot : moveRobot(w,0) //check again(DO NOT STOP)
    ]
```

Listing 1.34. `oncecellforward`: probableFixedObstacle

9.17.5 sendAnswerAfterCollision .

The state sendAnswerAfterCollision sends to the (original) caller (i.e. the **robotmind**)) the message moveMsgCmdDone with the duration of the move, that is a negative value for a far obstacle.

```
    whenMsg stopWork        -> handleStop , //first to be checked
    whenMsg collisionDispatch -> probableFixedObstacle

Plan handleMobileObstacle[
    [!? moveWDuration(T)] println( onecellforward_handleMobileObstacle(T))
]
switchTo doMoveForward //hoping to perform the move
```

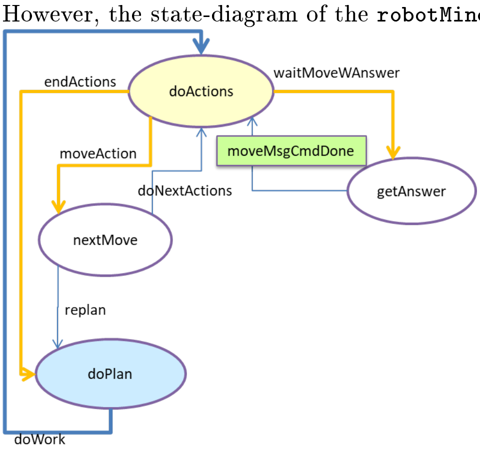
Listing 1.35. `onecellforward`: sendAnswerAfterCollision

9.18 Incremental design and testing

.

The last version of the actor `onecellforward` of Subsection 9.17 should allow us to face all the possible situations found during our analysis in Subsection **??**.

However, the state-diagram of the `robotMind` is now:



In practice, we have defined a part of the system that can work in absence of obstacles and walls. The possibility to test immediately this part is important, in order to find bugs and eliminate them while we still deal with a quite simple situation and a limited functionality. With reference to the `SCRUM` framework, we can say to have completed ('done') a SPRINT. In this view, we should provide a working, deployable prototype to be presented and discussed in the Sprint-Review meeting

The important point is that the system architecture has been defined with reference to an 'holistic' approach[13], in which all the (main) requirements have been considered. The rest of our development should consist in extending the features of the `robotmind` in order to handle the obstacles.

The current system architecture is formally described by a `QActor` model, that can be now reported in a more complete way:

9.18.1 Vocabulary .

The first part of our specification is related to the delclaration of events and messages that will carry information among the components (actors) of the system:

```

System robotCleaner
```

[13] Holistic is defined as: "*relating to the whole of something or to the total system instead of just to its parts*".

```

//Events related to the W-Env
Event sonar          : sonar( NAME, player, DISTANCE )
Event sonarDetect    : sonarDetect( V )

//Application messages
Dispatch startClean   : startClean(V)
                     //V=<clean details>. e.g. accuracy, current time, etc.
Dispatch stopWork     : stopWork(V)  //V=<stop details> e.g. stopByuser, nomoves, etc.
Dispatch doWork       : doWork

//Robot control messages
Dispatch moveRobot    : moveRobot(V,T) //V= w | s | a | d | h

/*
 * Core-Business
 */
Dispatch collisionDispatch : sonarDetect(TARGET)

Dispatch moveMsgCmd       : moveMsgCmd(TF) //TF = execution time of the basicStep
Dispatch waitMoveWAnswer : waitMoveWAnswer
Dispatch moveMsgCmdObstacle : moveMsgCmdObstacle(T) //T = w-move execution time before the obstacle
Dispatch moveMsgCmdDone   : moveMsgCmdDone(T)   //T = w-move execution time

Dispatch moveAction       : moveAction(M,REPLAN) //M=a|d|up|down|s, REPLAN=true|false
```

9.18.2 Structure .

The part related to the specification of the structure of the system does introduce the contexts and the actor working in each context.

```
Context ctxRobot ip [ host="localhost" port=8030]
EventHandler collisionevh for sonarDetect -print
{
    forwardEvent onecellforward -m collisionDispatch;
    forwardEvent robotmind    -m collisionDispatch
};
EventHandler sonarevh for sonarDetect -print ;

QActor onecellforward context ctxRobot{
...
]

QActor robotmind context ctxRobot {
...
}

QActor player context ctxRobot {
...
}
```

Each actor could run on its own machine; at the moment however, we assume (to make debugging easier) that they work on the same machine, represented by a **Context** named ctxRobot:

The business logic of this application is mainly included in the actors `onecellforward` and `robotmind`; the `player` is just an actuator.

9.18.3 Behavior .

The behavior of each component (actor) of the system can be described as a *Finite State Machine* (`FSM`). Each state has a precise responsibility and each requirement is 'covered' by one or more states of the `FSM`. Since the transition between states takes place on behalf of events or messages, each actor can be easily split into a set of different (local or remote) actors, as already done for `onecellforward`.

10 Designing, coding, testing

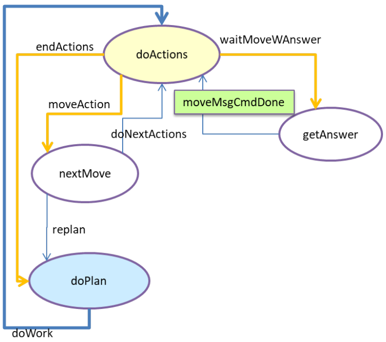
10.0.4 Vocabulary .

The vocabulary of the interaction among the parts is defined in explicit way:

```
Dispatch doWork           : doWork
Dispatch stopWork         : stopWork(V)  //V=<stop details> e.g. stopByuser, nomoves, etc.

Dispatch moveMsgCmd       : moveMsgCmd(TF) //TF = execution time of the basicStep
Dispatch waitMoveWAnswer : waitMoveWAnswer
Dispatch moveMsgCmdObstacle : moveMsgCmdObstacle(T) //T = w-move execution time before the obstacle
Dispatch moveMsgCmdDone   : moveMsgCmdDone(T)   //T = w-move execution time

Dispatch moveAction       : moveAction(M,REPLAN) //M=a|d|up|down|s, REPLAN=true|false
```

10.0.5 Incremental design and testing .

After the introduction of the state nextMove (Subsection 9.13), the state-diagram of the robotMind is now:



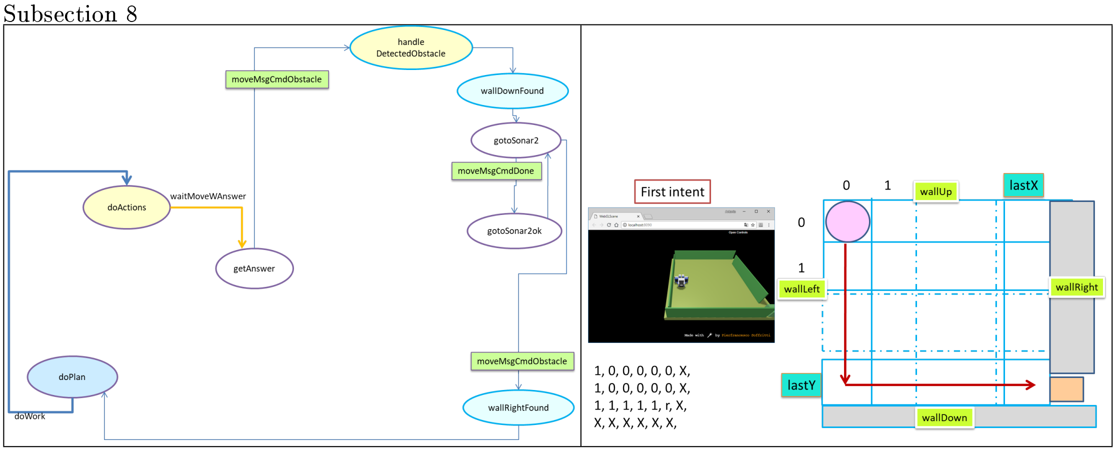
In practice, we have defined a part of the system that can work in absence of obstacles and walls. The possibility to test immediately this part is important, in order to find bugs and eliminate them while we still deal with a quite simple situation and a limited functionality.

The responsibility is split among different states that can easily become independent actors, as already done for the onecellforward actor.

Moreover, with reference to the SCRUM framework, we can say to have completed ('done') a SPRINT. Now we can start another sprint to handle obstacles.

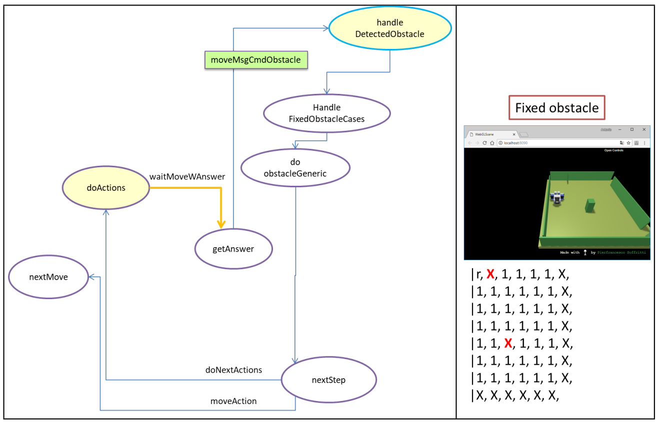
10.0.6 First intent .

Subsection 8

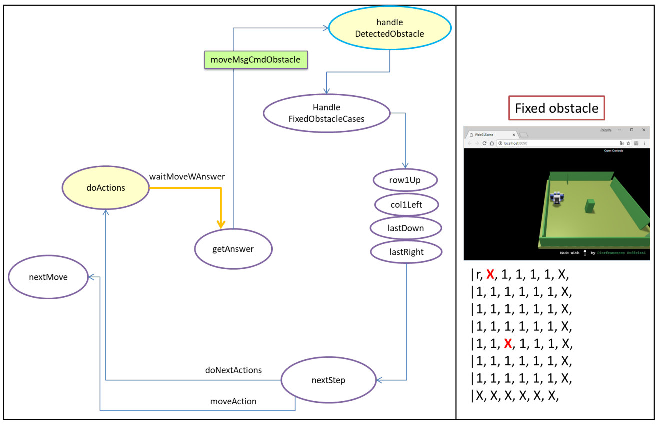


10.0.7 The path without obstacles .

10.0.8 Fixed, generic obstacles .

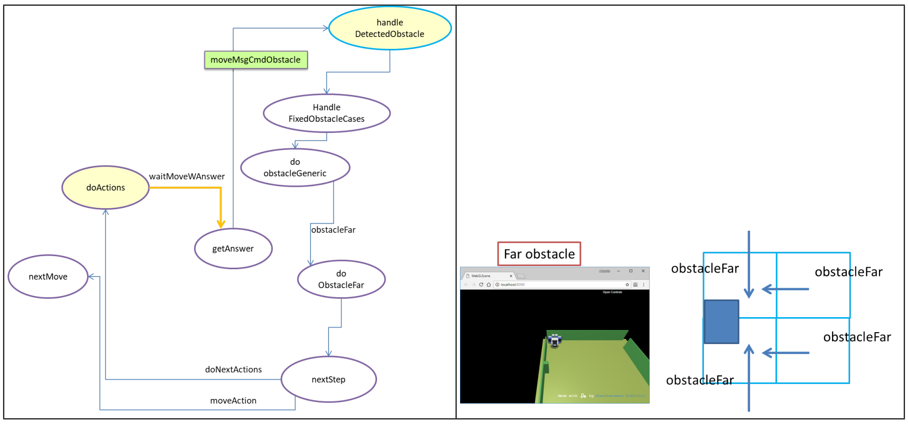


10.0.9 The walls .

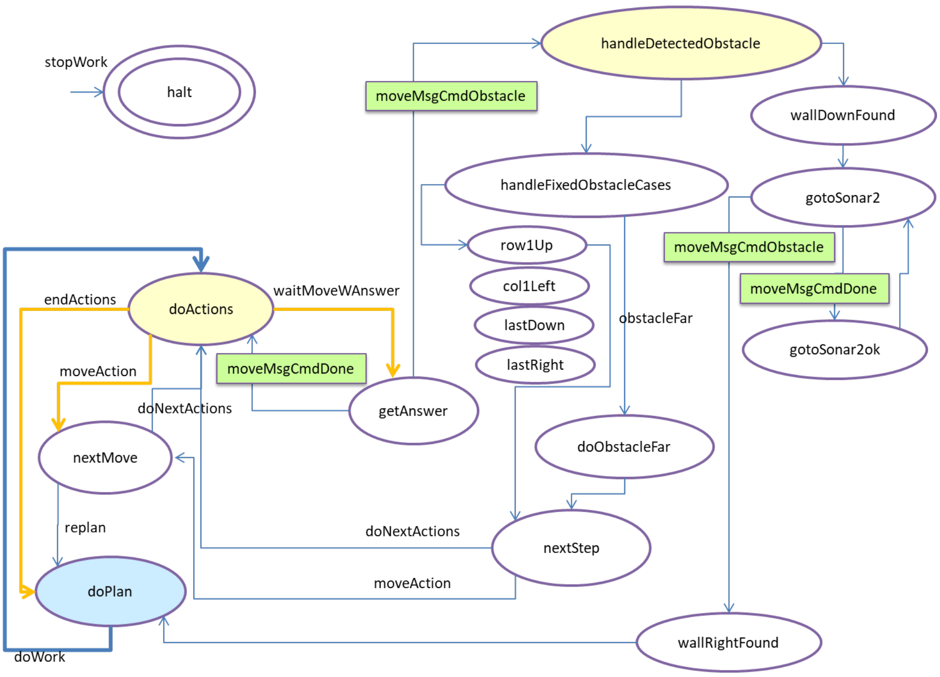


10.0.10 Fixed, far Obstacles .

Optimization (Subsection 9).



10.0.11 The global system .



References