Ingegneria dei Sistemi Software M

Marco Boschi, Marco Rossini

A.A. 2017–2018

1 Requirements Analysis

The DDR robot is a device that can move in an environment through remote commands to move forward, back and turn 90° right or left, in addition to the stop command. The environment in which it will have to move is a square room that can be both physical and virtual. The room is equipped with two sonars: one, called `sonar1`, is placed at the top left and facing down, the other, called `sonar2`, is placed at the bottom right, facing left.

The robot will have to move from an initial position intercepted by `sonar1`, called `start-point`, and move along the room to the final position, called `end-point`, intercepted by `sonar2`, cleaning the largest possible floor area. The initial and final points are not only identified by the sonar, but also by the points close to them.

A *sonar* is a device, real or virtual, that uses some technology to detect when the robot passes in front of it.

The robot is controlled through a graphical interface, called `console`, accessible to a human user authorized by the insertion of credentials recognized by the system. The cleaning activity is triggered by the sending of the `START` command by the authorized user who controls the robot. This happens through any device able to connect to the console, and it will be the latter to connect to the robot. If the robot is not detected (within a certain distance) from `sonar1` when sending the command `START`, the robot will not start with the automatic cleaning task.

The user can only control `START` or `STOP`, and must not have manual controls to drive the robot.

The robot only works if the ambient temperature is not higher than a pre-set threshold and if the current time is within a pre-set interval. The robot is equipped with a temperature sensor and a clock to check these conditions.

During the cleaning activity, the robot must blink a Hue led lamp, that is a light bulb connected to the network, which can be controlled in brightness, power on, off and color through REST APIs.

During the cleaning activity, the robot must also take care to avoid obstacles, fixed and mobile, present in the room. It is supposed that there are no obstacles placed in front (within a certain threshold) of the two sonar.

The cleaning activity must end if:

- the authorized user sends the command `STOP` from the console;
- the ambient temperature exceeds the threshold;
- the current time exits the interval;
- the robot cannot in any way avoid a certain obstacle to reach the `end-point`;
- the robot has finished cleaning the room and has reached the `end-point`.

There are three processing nodes: the robot, the console and hardware devices such as sensors and the Hue lamp.

2 Problem Analysis

By analyzing the projects realized and presented during classes, the console is already available, designed as a frontend server in Node.js technology. It provides the authentication of users through a username-password pair as access credentials and commands to drive the robot in its basic actions. Buttons can easily be added to send the `START` and `STOP` commands.

The system to be implemented is distributed, therefore it is necessary to identify a communication system between the parts, i.e. the frontend, the robot, the sensors and the sonar. This can be identified as a message exchange supported by the MQTT protocol, already integrated within the available console.

When choosing to work in a virtual environment, the ConfigurableThreejsApp project is available, which offers the virtual environment, equipped with sonars, and the virtual robot. It is accessible through a web interface and controllable via a TCP socket.

To create a first prototype of the robot with the application logic that controls the cleaning activity we rely on a language that allows to write executable models to ensure rapid prototyping. This language is identified in QActor, which also inherently allows to integrate with a MQTT message exchange infrastructure and working on a Java VM allows easy access to APIs to manage sockets.

Since the robot does not inherently communicate via messages, it is necessary to create an adapter that intercepts the basic commands sent by the console, or the application logic during cleaning, to drive the robot and translate it into a stream of TCP data that the robot can understand. This adapter will also take care of receiving sonar signals and translating them into messages so that other system components can be aware of them.

To manage the application logic, a dedicated component that receives the `START` and `STOP` commands is required to control the start and end of the cleaning task, to manage it and to check that the conditions specified for operate are satisfied.

To receive information on ambient temperature and current time, two sensors, a thermometer and a clock, must be made to send their respective information. Since these affect only the automatic cleaning activity, they can be intercepted only by the component that takes care of the application logic and use them to update the internal status and end, if necessary, the cleaning activity.

2.1 Hue Lamp

To manage the blinking lamp we need to identify an entity that cyclically alternates between two states by turning the lamp on and off, which is perfectly suited for a QActor actor. In order to not interfere with the normal operations of the application logic it is convenient to model this *blinker* as a separate actor controlled by the application logic and which controls the lamp directly.

2.2 Cleaning and Obstacles

To guide the robot in cleaning the largest possible area of floor, an approach would be to establish a path that the robot follows indiscriminately, but this only works if the morphology of the room is completely known, which is not the case because, although we know the shape of the room, it is not known if and where there are obstacles.

To deal with this problem, an artificial intelligence approach can help by dividing the room into *tiles* (or *cells*) and then using a search algorithm to reach all those to be cleaned and when the task is completed, the `end-point` in front of `sonar2`.

The search algorithm will inherently avoid obstacles and detect if they are unavoidable, preventing the robot from reaching the `end-point` and thus triggering an early stop to cleaning operations.

The detection of obstacles requires further work, detecting when the robot fails to complete one of the planned moves to clean, that is when the on-board sonar signals something. When the robot is moving forward and detects that there is an obstacle in front of it means that it can not reach the tile in front of it. This tile will then be marked as an obstacle, the robot will have to go back to realign with the virtual tiles then cancel all planned moves and make a new plan.

The detection of obstacles also automatically leads to a map of the room as the robot tries to clean. The search algorithm that guides the robot will also have to take care to ignore those parts of the room that are obstructed by obstacles and not clean them because they are unreachable and still go to `end-point`.

2.2.1 Mobile Obstacles

To manage the mobile obstacles it is necessary to check again all the tiles for which an obstacle has been detected. In this regard the search algorithm comes in handy adding to the already used tile states (`0` for

an unexplored and not clean cell, `1` for a free and clean cell and `x` for an obstacle cell) two other states: `t` for detected obstacle and `p` for possible obstruction.

During cleaning when the robot detects an obstacle, according to the process already described, it will no longer mark the tile as `x` but as `t` and proceed to clean the cells `0` avoiding passing on tiles `t`. When there are no other tiles `0` (or those present are not reachable), all the tiles `t` become marked as `p` and the robot will now continue to clean trying to reach the tiles `0` and `p`, on which it can now move. If it manages to go on these tiles, they will be first marked as `1`, if instead it fails to go on a tile `p`, this will finally be marked as `x` (if it can not go on a `0` one, it will be marked as `t` and then repeat the process).

2.3 Mapping

When the robot completes the cleaning naturally stopping at `end-point` it will save the generated map in a file, so where are the obstacles and marking the other cells as `0`. If the robot is stopped during cleaning (manually or because of the sensors) or if it doesn't stop at the `end-point` due to an unavoidable obstacle, the map will not be saved because it could not be completed or an unexpected situation has occurred.

When the robot is started or reconfigured to start again to clean, it will be able to load the generated map and thus already know where the obstacles are located, allowing to simplify the work when it does not have to map them.

2.4 Testing

The most complex part of the project is the QActor `cleaner` which deals with managing the movements of the robot on the basis of a map generated during cleaning and consulted using the A* algorithm (§ 4).

To simplify testing, this actor was then isolated in a dedicated project (see `it.unibo.finaltask.testing`). In addition, the actor has been slightly modified so that the moves are not performed automatically in succession (as happens in the main project) but require the sending of a control message that triggers the execution. This ensures a more granular control to the test suite that can do its job in a simplified way, in a similar way the reset to the state waiting to start is manual and triggered by sending the same message. This is accomplished by adding two states (`waitMove` and `waitCleanKB`) so that it can return to nominal operation by making the transition corresponding to the message no longer dependent on it, but a ε-move.

The testing suite then proceeds to load only the actor (which will not move in an environment neither real nor virtual) and then send appropriate messages of start and end of the automatic cleaning as well as the control messages to allow the execution of the moves and the simulation of the sonar events mounted on the robot. The sending of all the messages will be followed by a check whether the robot is cleaning or not (using an ad hoc fact in the KB) and by a progressive check of the map that the robot knows after every single step (see the `it.unibo.cleaner.TestCleaner` class inside the `test` folder).

3 Project

To maintain the state of the system we can rely on the native support of Prolog by QActor to model a knowledge base that keeps track of temperature and current time, if the cleaning is in progress and any other useful information such as the temperature threshold and the time interval in which it can work.

Not having available a physical or virtual implementation of such sensors, able to interact in any way with the system as it has been identified so far, we have chosen to create them from scratch as virtual sensors directly integrated with the MQTT protocol. A mock was also made for the Hue lamp to test the robot's operation when the physical one is not accessible. This mock interacts directly with the control messages exchanged via MQTT, but to control the physical lamp it is necessary to realize an adapter that intercepts the commands and translates them into appropriate REST APIs.

The interaction between the parts of the system is managed through messages exchanged through the MQTT protocol. These messages are exchanged with a particular structure so that it can interact with the QActor infrastructure, thus being able to have a dispatch behavior, addressed to a particular actor, or event, i.e. directed to all. Not wanting to bind too much to the names of the actors that make up the system and to support multiple actuators at the same time, in particular the physical Hue lamp and its mock,

it is convenient to choose to work with events, limiting the use of dispatch only between actors, in which the QActor language guarantees the control of errors at the semantic level for the names of the recipients. Dispatches will be used in particular by the virtual robot adapter that delegates the management of the received `START` and `STOP` commands to the actor who takes care of the application logic.

The use of dispatches is also preferable because the integration of QActor with MQTT is not complete and in some cases the use of events via MQTT leads to some bugs. The events (via MQTT) will then be used to interact with sensors, actuators and consoles, favoring dispatches (exchanged therefore directly via the QActor architecture) for direct communication between the actors.

From this point of view, sensors can sends new data by emitting a `sensorEvent(ORIGIN, PAYLOAD)` event that specifies the sensor involved and the related information. Similarly, the `ctrlEvent(TARGET, PAYLOAD)` event specifies an actuator and the information for the action to be performed, this will be issued only by the application logic and directed to the Hue lamp to turn it on and off. The control event will then be intercepted directly by the mock to control the virtual lamp and by the already mentioned adapter that will take care of translating the event into REST APIs.

3.1 Hue Lamp

Regarding the status of the lamp, the application logic is not interested in knowing if it is on or off, but whether it is blinking or not. The maintained state will therefore be that of the blinker. The blinker actor will send `ctrlEvent` for the lamp, while the application logic will send a `ctrlMsg` (with the same semantics of `ctrlEvent` but exchanged as dispatch) to the blinker.

4 A* Implementation

The A* algorithm being used has been implemented ad hoc in Prolog (see file `astar.pl` in main or testing project containg also other predicates used to manage the KB). The purpose of this section is not to explain the theory at the base of this algorithm, rather to show how it has been implemented.

The staring point of the algorithm is the `findMove`/1 predicate that first of all decides towards which cell to move by using `establishGoal`/1. This predicate will report the bottom-right cell if the room has been completely cleaned, otherwise a cell marked as `0` then as `p` (in the order they have been saved in the KB). The room is considered clean if the map contains only cells `1` and `x` or if the fact `overrideCleanStatus` is present, this is added by `cleaner` when it is detected that the remaining `0` (or `p`) cells are inaccessible due to obstacles.

Established the goal, `findMove`/1 obtains the current position of the robot (saved in the KB as the fact `curPos(pos(cell(X,Y),D))` where `D` is the direction the robot is facing as `n`, `e`, `s` or `w` respectively for north, east, south and west, and updated by `cleaner` after each successful move) and calls the proper A* algorithm passing:

- the established goal as `cell(X,Y)`;
- the list of open nodes as possible states of the robot (cell and direction), for now only the current one;
- the list of closed states, for now only the current one.

An open node is not only its status, but it is represented as a list of `act(pos(cell(X,Y),D),M)` representing a pair of resulting state and move that caused it (the move for the starting state is `null`), allowing as such the reconstruction of the planned path. The open nodes list is kept sorted by increasing cost function computed by assigning cost 1 to each move and using Manhattan distance as heuristic.

The proper algorithm is represented by the predicate `findMove`/4 and works using an iterative cycle: if the first open node (the most promising one) is the goal then the optimal path has been found, otherwise:

1. The state of the first open node is considered and the successor function is called on this, obtaining in `Succ` a list of possible future states (with the corresponding move): moving forward by a single cell or rotating (left or right) by 90°. The movement forward is available only if the cell in front of the robot exists (the robot is not at the border of the room) and is not marked as obstacle `x` or `t`.

2. The returned states are filtered by `filterVisited`/4 that returns in `SuccFilter` only those states that are not yet visited and in `NewVis` the updated list of closed states. The filtering differentiates depending of which type of move generated the state:
   - with a rotation the comparison with the closed states list considers both the cell and the direction;
   - with a movement forward only the cell is considered, the direction is ignored.
3. Future states are now sorted by `sort`/3 by increasing heuristic (path cost is the same as they come from the same states by adding just one move) which returns the result in `SuccFilterSort`.
4. Futures states are then combined with the path that led to them (also keeping them sorted) by the predicate `multiAppend`/3 which returns the new open nodes in `Paths`.
5. Open nodes not considered and new ones just generated are combined in a single list by using the predicate `mergeSorted`/4 which returns in `Lnext` all open nodes sorted by increasing cost function.
6. The algorithm invokes itself by passing the new lists for open nodes and closed states but the same goal.

The plan generated by A* is then given to `registerMoves`/1 which saves in the KB the moves in the order in which to execute them as fact `move(A,pos(cell(X,Y),D))` where the first argument is the move and the second the resulting state if it is successful.

4.1 Auxiliary Predicates

The algorithm and `cleaner` which uses it both rely on auxiliary predicates to manage the KB. Here's a brief introduction to these, additional details can be found in comments accompanying each predicate in the code:

- `h`/3: calculate the heuristic for a certain cell relative to a goal (another cell), direction is ignored.
- `rotate`/2: supports the successor function `next`/2 by generating only the rotations.
- `jobDone`: determines whether the robot should stop autonomously based on `R-End`.
- `loadStatus`: by using `loadStatus`/1 and `loadCol`/2 creates in the KB an empty map (all cells marked as `0`) saving the cells ordered by columns left to right and top to bottom inside each column. Each cell is saved as a fact `status(cell(X,Y),S)` where `S` is the status such as `0`, `1`, `x`, `p` or `t` and the `X` and `Y` coordinates start at 0 in the top-left corner and `X` increases towards the right and `Y` towards the bottom.
- `loadInitialPosition`: registers the initial current position of the robot in the KB, used when initializing `cleaner` before the start of the cleaning operation.
- `printStatus`: by using `printRow`/1 and `listify`/3 prints to standard output the current map saved in the KB.
- `visit`/1: marks the given cell as `1` only if it was `0` or `p`. `visitCurrent` invokes this predicate passing the current cell.
- `obstacle`/1: marks the given cell as obstacle. A cell marked as `0` becomes `t`, one marked as `p` becomes `x`.
- `recheck`/1: marks the given cell as `p` only if it was marked as `t`. This predicate is called from `cleaner` when there are no more `0` cell (or those present are inaccessible) to clean, triggering a recheck for obstacles to determine if they are fixed or were mobile ones.
- `isWalkable`/1: determines whether if possible to move onto the given cell. This is true only for cells marked as `1`, `0` or `p`, the latter is essential to recheck cells marked as possible obstacles by working together with `establishGoal`/1.
- `registerNext`/1: registers the given state as the future state for the robot if the move being executed will be successful.
- `actualizeNext`: removes the future state which becomes the current one, the corresponding cell is marked as `1`.
- `nextIsObstacle`: removes the future states and the corresponding cell is marked as obstacle by using `obstacle`/1. The current state is not updated as this predicated is called when a move fails mid execution due to a obstacle.

5 Log

For partial projects related to individual sprints, refer to the corresponding tags/releases.

5.1 1st sprint – 23-27/06/2018

We focused on the analysis of the requirements and the problem at a general level to identify the parts that we already have available from projects carried out in class. These parts were assembled together with a mock for the hardware devices (sensors and actuators) required to have a first working prototype, but without application logic, of the system.

- `R-Start`

5.2 2nd sprint – 27/06-04/07/2018

Support of the status of the resources as temperature value and current time, cleaning as movement forward and backward for an indefinite time to control the manual stop or for failure to check the temperature/time conditions.

- `R-TempOk`
- `R-TimeOk`
- `R-BlinkHue`
- `R-Stop`
- `R-TempKo`
- `R-TimeKo`

5.3 3rd sprint – 04-11/07/2018

Cleaning of the entire room and management of fixed obstacles: detected, mapped and avoided.

- `R-FloorClean`
- `R-AvoidFix`
- `R-Obstacle`
- `R-End`
- `R-Map`

5.4 4th sprint – 11-19/07/2018

Management of mobile obstacles and use of the generated map.

- `R-AvoidMobile`

5.5 5th sprint – 20/07/2018

Automatic testing of the QActor `cleaner`.