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 flash 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 RESTful 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) to the two sonar.

The robot must stop only when it has finished cleaning everything.

The cleaning activity must end also if:

- the authorized user sends the command `STOP` from the console;
- the ambient temperature exceeds the prefixed threshold;
- the current time exits the prefixed interval;
- the robot fails, in any way, to avoid a certain obstacle (this insuperable obstacle is a horizontal or vertical stick of the length of the room);
- the robot has finished cleaning, or reaches the final position.

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 in class, the console is already available, designed as a frontend server in Node.js technology. It presents 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, ie the frontend, the robot, the sensors and the sonar. This can be identified in the message exchange supported by the MQTT infrastructure, 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 you to write executable models to ensure rapid prototyping. This language is identified in QActor, which also allows to integrate natively with a MQTT message exchange infrastructure and working on a Java base allows easy access to APIs to manage sockets.

Since the robot does not communicate via messages natively, 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 is required that receives the `START` and `STOP` commands 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 the respective information. Since these affect only the automatic cleaning activity, they can only be intercepted 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 flashing lamp we need to identify an entity that cyclically alternates between two states by turning the lamp on and off, which is perfectly suited to a QActor actor. In order not to 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 the room would be possible to previously establish a path that the robot follows indiscriminately, but this only works if the morphology of the room is completely known, which is not verified because, although we know the shape of the room, it is not known if and where there are obstacles.

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

The search algorithm will natively avoid obstacles and detect if they are unsurpassed by preventing the `end-point` from reaching 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 Moving 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 tile `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 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 one `0` will be marked as `t` by repeating 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 the end of the `end-point` does not end due to unsurpassed obstacles, the map will not be saved because it could not be completed or a situation has occurred sudden.

When the robot is started or reconfigured to start again to clean, it will be able to load the generated map and thus know where the obstacles are already 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 algorithm A* (§ 4).

To simplify testing, this actor was then isolated in a dedicated project (see the project `it.unibo.finaltask.testin` 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 suite of testing 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 intercalated by a robot status check if it is cleaning or not (using an ad hoc fact in the knowledge base) and by a progressive check of the map that knows the robot 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 the 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 realization, 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 infrastructure. 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 one it is necessary to realize an adapter that intercepts the commands and translates them into appropriate RESTful calls.

The interaction between the parts of the system is managed through messages exchanged through the MQTT infrastructure. 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, ie 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. The dispatches will be used in particular of the virtual robot adapter that delegates the management to the actor who takes care of the application logic when it receives the commands of `START` and `STOP`.

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 is due to some bugs. The events (via MQTT) will then be used to interact with sensors, actuators and consoles, favoring instead of dispatches (exchanged therefore directly from the QActor architecture) for direct communication between the actors.

From this point of view, the sending of new data by the sensors occurs through the emission of 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 for shutdown commands and power on. 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 RESTful calls.

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 flashing 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 Implementazione di A*

L'algoritmo A* utilizzato è stato implementato ad hoc in Prolog (si veda il file `astar.pl` nel progetto principale o di testing contenente anche altri predicati di contorno per inizializzare la base di conoscenza). In questa sezione non si pretende di spiegare il funzionamento e la teoria alla base dell'algoritmo, ma di illustrare come questo sia stato implementato.

Il punto di accesso all'algoritmo è il predicato `findMove`/1 che per prima cosa decide verso quale cella muoversi utilizzando `establishGoal`/1. Questo predicato indicherà il tile in basso a destra se la stanza è stata interamente pulita altrimenti (nell'ordine) una cella marcata come `0` o una `p`. La stanza è considerata pulita se la mappa contiene solamente celle `1` e `x` oppure se è presente il fatto `overrideCleanStatus`, aggiunto da `cleaner` quando rileva che le celle `0` rimanenti sono inaccessibili a causa di ostacoli.

Una volta stabilito il goal, `findMove`/1 ricava la posizione corrente nota dal robot e invoca l'algoritmo A* vero e proprio fornendo come lista dei nodi aperti lo stato (cioè cella e orientamento del robot) corrente, questa è anche la lista dei nodi chiusi. Un nodo aperto non è solo lo stato, ma è rappresentato da una lista di coppie mossa-stato risultante (la mossa per lo stato di partenza è nulla) garantendo così la ricostruzione del percorso pianificato. La lista dei nodi aperti viene mantenuta ordinata in ordine crescente secondo la funzione di costo calcolata assegnando costo unitario a ogni azione e utilizzando la distanza di Manhattan come euristica.

L'algoritmo vero e proprio è rappresentato dal predicato `findMove`/4 e funziona secondo un ciclo iterativo: se il primo nodo aperto (e quindi il più promettente) corrisponde al goal allora si è trovato il percorso ottimo, altrimenti:

1. Si considera lo stato del primo nodo aperto e si invoca su questo la funzione successore ottenendo in `Succ` una lista di possibili stati futuri scelti tra movimento in avanti di una cella oppure rotazione a destra o sinistra di 90°. Il movimento in avanti è restituito solamente se la cella di fronte al robot esiste e non è marcata come ostacolo `x` o `t`.

2. Gli stati restituiti sono filtrati usando `filterVisited`/4 ottenendo in `SuccFilter` solo quegli stati che non sono già stati visitati e in `NewVis` la nuova lista di nodi chiusi. Il filtraggio discrimina a seconda di quale mossa lo genera:
   - nel caso di una rotazione il confronto con la lista dei nodi chiusi considera sia la cella che l'orientamento;
   - nel caso di un movimento in avanti il confronto considera solamente la cella, ignorando l'orientamento del robot.
3. Gli stati futuri filtrati sono ora ordinati con `sort`/3 secondo l'euristica crescente (il costo di cammino sarà uguale perché vengono tutti dallo stesso stato) ottenendo il risultato in `SuccFilterSort`.
4. Gli stati futuri sono ora combinati con il cammino che li ha generati (mantenendo anche l'ordinamento) utilizzando il predicato `multiAppend`/3 che restituisce i nuovi nodi in `Paths`.
5. I nodi aperti non considerati e i nuovi nodi aperti appena generati sono unificati in un'unica lista utilizzando il predicato `mergeSorted`/4 che restituisce in `Lnext` tutti i nodi aperti ordinati secondo la funzione di costo crescente.
6. L'algoritmo re-invoca se stesso passando la nuova lista di nodi aperti e chiusi ma lo stesso goal.

Il piano generato da A* viene quindi utilizzato da `registerMoves`/1 che salva nella base di conoscenza le mosse nell'ordine con cui devono essere eseguite.

4.1 Predicati ausiliari

L'algoritmo e `cleaner` che lo usa sfruttano anche dei predicati ausiliari per gestire la base di conoscenza. Di seguito sono introdotti brevemente, maggiori dettagli possono essere trovati direttamente nei commenti a corredo di ogni predicato all'interno del codice:

- `h`/3: calcola l'euristica per una certa cella relativamente a un certo goal.
- `rotate`/2: supporto alla funzione successore `next`/2, si occupa di generare solamente le rotazioni.
- `jobDone`: determina se il robot deve fermarsi autonomamente sulla base di `R-End`.
- `loadStatus`: utilizzando `loadStatus`/1 e `loadCol`/2 popola la base di conoscenza con una mappa vuota (tutte le celle a `0`).
- `loadInitialPosition`: registra la posizione corrente iniziale del robot all'interno della base di conoscenza.
- `printStatus`: utilizzando `printRow`/1 e `listify`/3 stampa su standard output la mappa corrente salvata nella base di conoscenza.
- `visit`/1: segna la cella specificata come `1` (solo se `0` o `p`). `visitCurrent` invoca questo predicato passando la posizione corrente.
- `obstacle`/1: segna la cella specificata come ostacolo. Una cella `0` diventa `t` e una cella `p` diventa `x`.
- `recheck`/1: segna la cella specificata come `p` solamente se era marcata come `t`. Questo predicato viene invocato dal robot su tutte le celle quando non ha più celle `0` da pulire innescando il ricontrollo degli ostacoli rilevati per pulire dove fossero presenti degli ostacoli mobili.
- `isWalkable`/1: se è possibile transitare sulla cella specificata. Questo è vero solo per le celle `1`, `0` e `p`, quest'ultimo caso permette di ricontrollare le celle segnate come possibili ostacoli in combinazione con `establishGoal`/1.
- `registerNext`/1: registra la cella specificata come stato futuro durante l'esecuzione di una mossa.
- `actualizeNext`: distrugge un eventuale stato futuro che diventa lo stato corrente, la cella corrispondente viene marcata come `1`.
- `nextIsObstacle`: distrugge un eventuale stato futuro e la cella corrispondente viene segnata come ostacolo con `obstacle`/1. Il robot viene lasciato nella posizione corrente.

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 bypassed.

- `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`.