Ingegneria dei Sistemi Software M

Marco Boschi, Marco Rossini

A.A. 2017–2018

1 Requirement Analysis

The DDR robot is a device capable of moving in the environment via remote commands for advancing, back up or rotating 90° left and right beside the halt command. The environment where the robot will have to move is a square room that can be both virtual and real. The room is equipped with two sonars: one, called `sonar1`, is located in the top-left pointing downwards, the other one, `sonar2`, is the bottom-right pointing left.

The robot will have to start from an initial position detected by `sonar1`, called `start-point`, and move inside the room to the final position, called `end-point`, detected by `sonar2`, cleaning the largest portion of floor possible. The initial and final positions are not any position detected by the corresponding sonar, but the area immediately in front of each sonar.

A *sonar* is a device, real o virtual, that uses some technology to detected when robot is moving in front of it.

The robot is controlled via a graphical interface, called `console`, accessible to a human user, authorized by using credentials recognized by the system. The cleaning activity is triggered by the user controlling the robot by sending a `START` command. This can be done by using any device capable to connect to the console, that in turn will connect to the robot. If the robot is not detected by `sonar1` (up to a certain distance) when receiving the `START` command, the robot will not start the automatic cleaning activity.

The user can control the robot using only the `START` and `STOP` commands, manual controls for the robot must not be available to him or her.

The robot can work only if the ambient temperature is not higher than a certain threshold and if the current time is inside a certain range. The robot is equipped with a temperature sensor and a clock to check this conditions.

While cleaning the robot must blink a led hue lamp, a lamp connected to a network capable of being controlled in brightness, on/off and color via REST APIs.

While cleaning the robot must take care to avoid obstacles (fixed and mobile) in the room. It can be assumed that no obstacles are present immediately in front of the two sonars in the room.

The robot must stop cleaning when:

- the authorized user sends the `STOP` command from the console;
- the ambient temperature becomes higher than the threshold;
- the current time exits the range;
- the 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`.

2 Problem analysis

Analyzing the projects developed and presented during classes, we found that the console is already available, realized a frontend server based on Node.js. The console provides user authentication using the pair username-password as access credential and command for controlling the robot in its basic movements, commands for `START` e `STOP` can be added very easily.

The system to be created is distributed and as such requires a communication medium between various nodes: the frontend, the robor, sensors and sonars. This can be found in a message exchange architecture offered by the MQTT protocol, already integrated inside the available console.

Choosing to work with a strictly virtual environment, the ConfigurableThreejsApp project offers the virtual room (embedded with sonars) and the virtual robot. This environment is accessible via a web interface and accepts commands via a TCP socket.

In order to create a first robot prototype with application logic that controls the cleaning operation a modeling and executable language is required to guarantee short development times. This language is QActor that inherently supports massage exchange via MQTT and has access to socket APIs as it's based on a Java VM.

As the robot does not inherently communicate via messages, an adapter that intercepts basic commands sent from the console or the application logic is required to translate them into a TCP stream the robot can understand. This adapter will also receives information from the sonars and translates them into messages for other parts of the system.

Per gestire la logica applicativa si rende necessario un componente dedicato che riceva i comandi di `START` e `STOP` per controllare l'avvio e la terminazione dell'attività di pulizia, la gestisca e controlli che le condizioni specificate per operare siano soddisfatte.

Per ricevere informazioni su temperatura ambientale e ora corrente occorre realizzare due sensori, un termometro e un orologio, che possano inviare le rispettive informazioni. Poiché queste interessano solamente l'attività di pulizia automatica, possono essere intercettate solamente dal componente che si occupa della logica applicativa e usarle per aggiornare lo stato interno e terminare, se è il caso, l'attività di pulizia.

2.1 Hue Lamp

Per gestire la lampada che lampeggia occorre identificare un'entità che alterni ciclicamente tra due stati accendendo e spegnendo la lampada, cosa perfettamente adatta a un attore QActor. Per non interferire con le normali operazioni della logica applicativa risulta conveniente modellare questo *blinker* come un attore separato controllato dalla logica applicativa e che a sua volta controlla la lampada direttamente.

2.2 Cleaning and Obstacles

Per guidare il robot nella pulizia della maggiore area possibile della stanza sarebbe possibile stabilire a priori un percorso che il robot segue in maniera indiscriminata, ma questo funziona solo se la morfologia della stanza è completamente nota, cosa non verificata in quanto, sebbene si conosca la forma della stanza, non si sa se e dove sono degli ostacoli.

Per gestire questo problema può venire in aiuto l'intelligenza artificiale dividendo la stanza in tile e sfruttando un algoritmo di ricerca per raggiungere tutti quelli da pulire e quando sono finiti, l'`end-point` davanti al `sonar2`.

L'algoritmo di ricerca porterà nativamente a evitare gli ostacoli e a rilevare se questi sono insuperabili impedendo di raggiungere l'`end-point` portando a uno stop anticipato alle operazioni di pulizia.

La rilevazione degli ostacoli richiede un lavoro ulteriore, rilevando quando il robot non riesce a portare a termine una delle mosse pianificate per pulire, cioè quando il sonar che ha a bordo segnala qualcosa. Quando il robot si sta muovendo in avanti e rileva che ha un ostacolo di fronte vuol dire che non può raggiungere il tile davanti a lui, questo tile sarà quindi marcato come ostacolo, il robot dovrà tornare indietro per riallinearsi con i tile "virtuali" quindi annullare tutte le mosse pianificate e fare un nuovo piano.

La rilevazione degli ostacoli porta automaticamente anche a realizzare una mappa della stanza man mano che il robot prova a pulire. L'algoritmo di ricerca che guida il robot dovrà anche aver cura di ignorare quelle parti della stanza che sono isolate da ostacoli e non pulirle perché irraggiungibili e andare comunque all'`end-point`.

2.2.1 Mobile Obstacles

Per gestire gli ostacoli mobili occorre ricontrollare tutti quei tile per cui è stato rilevato un ostacolo. A questo proposito l'algoritmo di ricerca viene in aiuto aggiungendo ai già usati stati dei tile (`0` per una cella non esplorata e non pulita, `1` per una cella libera e pulita e `x` per una cella con ostacolo) altri due stati: `t` per ostacolo rilevato e `p` per possibile ostacolo.

Durante la pulizia quando il robot rileva un ostacolo, secondo il processo già descritto, non marcherà più il tile come `x` ma come `t` e procederà a pulire le celle `0` evitando di passare su tile `t`. Quando non ci sono altri tile `0` (o quelli presenti non sono raggiungibili), tutti i tile `t` diventano marcati come `p` e il robot ora continuerà a pulire cercando di raggiungere i tile `0` e `p`, su cui ora può muoversi. Se riesce ad andare su questi tile saranno come prima marcati come `1`, se invece non riesce ad andare su un tile `p` questo sarà finalmente marcato come `x` (se non riesce ad andare su uno `0` sarà marcato come `t` ripetendo il processo).

2.3 Mapping

Quando il robot completa la pulizia naturalmente fermandosi all'`end-point` salverà in un file la mappa generata, quindi dove sono gli ostacoli e marcando le altre celle come `0`. Nel caso il robot sia fermato durante la pulizia (manualmente o a causa dei sensori) oppure non termini all'`end-point` a causa di ostacoli insuperabili la mappa non sarà salvata perché non è stato possibile completarla o si è verificata una situazione imprevista.

Quando il robot viene avviato o si riconfigura per ripartire a pulire potrà caricare la mappa generata e quindi sapere già dove si trovano gli ostacoli, permettendo di semplificare il lavoro in quando non deve mapparli.

2.4 Testing

La parte più complessa del progetto è il QActor `cleaner` che si occupa di gestire i movimenti del robot sulla base di una mappa generata durante la pulizia e consultata utilizzato l'algoritmo A* (§ 4).

Per semplificare il testing si è quindi isolato questo attore in un progetto dedicato (si veda il progetto `it.unibo.finaltask.testing`). Inoltre l'attore è stato leggermente modificato affinché le mosse non siano eseguite automaticamente in successione (come accade nel progetto principale) ma richiedono l'invio di un messaggio di controllo che ne scateni l'esecuzione, questo garantisce un controllo più granulare alla suite di testing che può svolgere il suo lavoro in maniera semplificata, in maniera analoga il reset allo stato in attesa di partire è manuale e scatenato dall'invio dello stesso messaggio. Questo è realizzato aggiungendo due stati (`waitMove` e `waitCleanKB`) per cui si può tornare al funzionamento nominale rendendo la transizione corrispondente al messaggio non più dipendente da questo, ma una ε-mossa.

La suite di testing procede quindi a caricare solamente l'attore (che non si muoverà in un ambiente né reale né virtuale) e quindi invierà opportuni messaggi di inizio e fine della pulizia automatica oltre che ai messaggi di controllo per permettere l'esecuzione delle mosse e la simulazione degli eventi del sonar montato sul robot. L'invio di tutti i messaggi sarà intercalato da un controllo dello stato del robot se sta pulendo o meno (usando un fatto ad hoc nella base di conoscenza) e da un check progressivo della mappa che conosce il robot dopo ogni singolo step (si veda la classe `it.unibo.cleaner.TestCleaner` all'interno della cartella `test`).

3 Project

Per mantenere lo stato del sistema ci si può appoggiare al supporto nativo del Prolog da parte di QActor per modellare una base di conoscenza che tenga traccia di temperatura e ora corrente, se si sta facendo la pulizia e qualunque altra informazione utile come la soglia di temperatura e l'intervallo di tempo in cui si può lavorare.

Non avendo a disposizione una realizzazione, fisica o virtuale, in grado di interagire in qualche modo con il sistema per come lo si è individuato finora, si è scelto di realizzarli da zero come sensori virtuali direttamente integrati con l'infrastruttura MQTT. È stato realizzato anche un mock per la lampada Hue per poter testare il funzionamento del robot quando quella fisica non è accessibile. Questo mock interagisce direttamente con i messaggi di controllo scambiati via MQTT, ma per controllare quella fisica occorre realizzare un adapter che intercetti i comandi e li traduca in opportune chiamate RESTful.

L'interazione tra le parti del sistema è gestita attraverso messaggi scambiati attraverso l'infrastruttura MQTT. Questi messaggi sono scambiati con una struttura particolare affinché possa interagire con l'infrastruttura QActor potendo quindi avere un comportamento dispatch, indirizzato ad un attore particolare, o

evento, cioè diretto a tutti. Non volendo vincolarsi troppo ai nomi degli attori che compongono il sistema e per supportare anche contemporaneamente più attuatori, in particolare la lampada Hue fisica e il suo mock, risulta comodo scegliere di lavorare con eventi, limitando l'uso di dispatch solamente tra attori, in cui il linguaggio QActor garantisce il controllo di errori a livello semantico per i nomi dei destinatari. I dispatch verranno impiegati in particolare dell'adapter del robot virtuale che quando riceve i comandi di `START` e `STOP` ne delega la gestione all'attore che si occupa della logica applicativa.

L'uso dei dispatch risulta anche preferibile in quanto l'integrazione di QActor con MQTT non è completa e in alcuni casi l'uso di eventi via MQTT è causa di alcuni bug. Gli eventi (via MQTT) saranno quindi usati per interagire con sensori, attuatori e console, privilegiando invece dei dispatch (scambiati quindi direttamente dall'architettura QActor) per comunicazione diretta tra gli attori.

In quest'ottica, l'invio di nuovi dati da parte dei sensori avviene attraverso l'emissione di un evento `sensorEvent(ORIGIN, PAYLOAD)` che specifica il sensore coinvolto e le informazioni relative. In maniera analoga si ha l'evento `ctrlEvent(TARGET, PAYLOAD)` che specifica un attuatore e le informazioni per l'azione da compiere, questo sarà emesso solamente dalla logica applicativa e indirizzato alla lampada Hue per i comandi di spegnimento e accensione. L'evento di controllo sarà quindi essere intercettato direttamente dal mock per controllare la lampada virtuale e dal già citato adapter che si occuperà di tradurre l'evento in chiamate RESTful.

3.1 Hue Lamp

Per quanto riguarda lo stato della lampada, la logica applicativa non è interessata a sapere se questa è accesa o spenta, ma se questa sta lampeggiando o no. Lo stato mantenuto sarà quindi quello del blinker. Sarà l'attore blinker a inviare `ctrlEvent` per la lampada, mentre la logica applicativa invierà un `ctrlMsg` (con la stessa semantica di `ctrlEvent` ma scambiato come dispatch) al blinker.

4 A* Implementation

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 Auxiliary Predicates

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

Per i progetti parziali relativi ai singoli sprint si faccia riferimento ai corrispondenti tag/release.

5.1 1st sprint – 23-27/06/2018

Ci siamo concentrati sull'analisi dei requisiti e del problema a livello generale per individuare le parti che abbiamo già a disposizione da progetti svolti a lezione. Queste parti sono state assemblate assieme a un mock per i dispositivi hardware (sensori e attuatori) richiesti per avere un primo prototipo funzionante, ma senza logica applicativa, del sistema.

- `R-Start`

5.2 2nd sprint – 27/06-04/07/2018

Supporto dello stato delle risorse come valore di temperatura e ora corrente, pulizia come movimento avanti e indietro per un tempo indeterminato per controllare lo stop manuale o per mancata verifica delle condizioni di temperatura/ora.

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

5.3 3rd sprint – 04-11/07/2018

Pulizia di tutta la stanza e gestione degli ostacoli fissi: rilevati, mappati ed aggirati.

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

5.4 4th sprint – 11-19/07/2018

Gestione degli ostacoli mobili e uso della mappa generata.

- `R-AvoidMobile`

5.5 5th sprint – 20/07/2018

Testing automatico del QActor `cleaner`.