SPRINT2

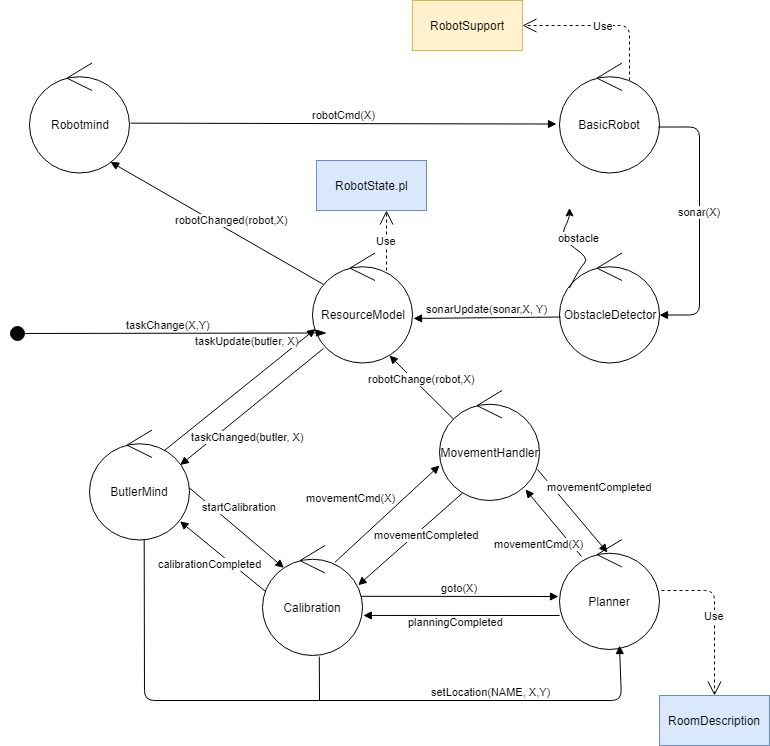
Obiettivi dello sprint:

P1

1. Calibrare la stanza di lavoro
2. Riuscire a raggiungere un punto specifico della stanza data la posizione attuale

P2

1. Implementare il supporto per il maitre
2. Possibilità di mandare i comandi prepare/addFood/clear
3. Possibilità di interrompere/riattivare un task
4. Avere accesso allo stato aggiornato della stanza

Architettura Finale

Spiegazione iniziale

Dall’analisi del problema è emerso come sia conveniente, per il butler, avere a disposizione una rappresentazione interna della stanza in cui opera. La mappa sarà composta da tante celle ognuna della dimensione del robot e per tale motivo ogni spostamento sarà pari ad una cella.

Questa fase di esplorazione e creazione della mappa avrà anche il compito di individuare la posizione e la dimensione del tavolo.

Per la parte di planning è possibile usare un componente già a disposizione nel package *itunibo.planner* il quale contiene al suo interno la mappa in cui operare e tutti i metodi di ausilio sia per trovare il percorso che per mantenere la stanza e la posizione del robot in uno stato consistente.

Il compito dell’esplorazione sarà quello di popolare la mappa del planner delineandone le dimensioni e il tavolo

Con questo sprint inizia la parte di modellazione della parte di business logic del butler. Introduciamo un attore *BUTLERMIND* nel quale verrà inserita tutta la logica dell’applicazione in maniera astratta senza dipendere da alcuna realizzazione particolare.

2.1

Nell’ottica dei microservizi la logica di calibrazione della stanza verrà delegata ad un altro componente, *CALIBRATION.* Visto che lo schema di questo attore è più complesso per semplicità verrà diviso in due parti:

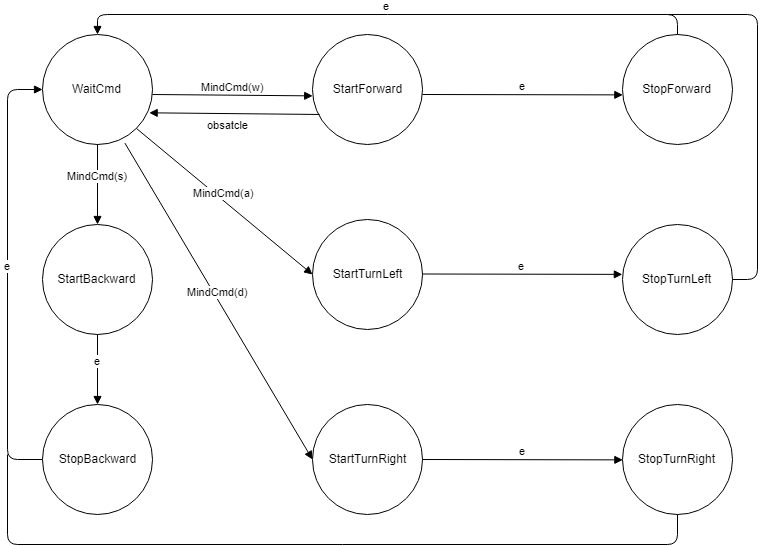
1. Misura dimensioni stanza
2. Individuazione posizione e dimensioni tavolo

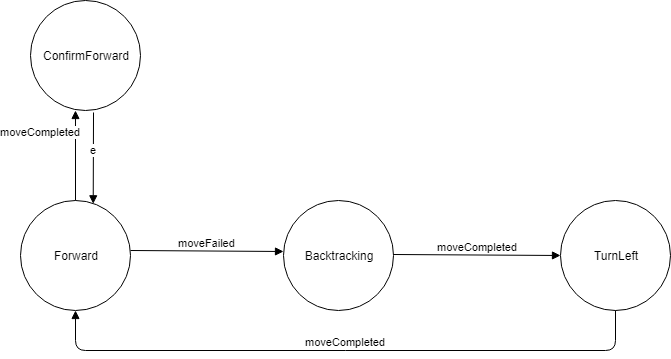
2.1.1 - ExploreRoom

Per calcolare le dimensioni della stanza è sufficiente percorrere i bordi scoprendo da quante celle è composto.

Calibration necessiterà di far muovere il robot, tuttavia sappiamo che opera a livello logico per cui non potrà sapere quando una mossa effettivamente è stata compiuta, informazione di cui necessità per gestire correttamente il comportamento.

Un nuovo attore *MOVEMENTHANDLER*  ha il compito di tradurre quelli che sono i movimenti logici, limitati ad una cella, nei movimenti costitutivi, compiuti dal *basicrobot.*

Nel diagramma di sopra si è assunto che, per il sistema da realizzare, le azioni di girata e di indietreggiamento non possano essere interrotte.

L’informazione di cui ha bisogno *calibration* sarà espressa attraverso i due messaggi *moveCompleted* e *moveFailed.* Il diagramma finale per questa prima parte può essere espresso come:

**Condizione di terminazione**: tramite *itunibo.planner* è possibile sapere la posizione del robot in ogni istante per cui possiamo fermarci quando si torna nella posizione (0,0).

2.1.2 - FindTable

Una volta delimitata la stanza bisogna trovare il tavolo, potenzialmente in una posizione qualsiasi. Di fatto bisogna esplorare l’interno cella per cella: diverse esplorazioni potrebbero avere diversi pro e contro.

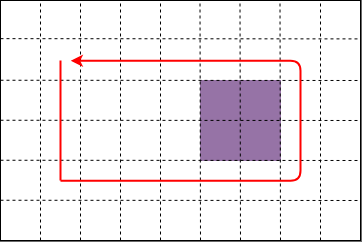
Due ipotesi:

1. Il tavolo non è sui bordi
2. Il tavolo è rettangolare

Dalla seconda ipotesi deriva che è sufficiente esplorare due lati del tavolo e derivare gli altri due.

Strategie di esplorazione

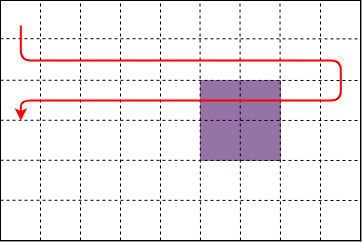
Prima strategia:

Il robot inizia ad esplorare l’interno a mo’ di spirale evitando i bordi

**PRO**: Non esploro zone già esplorate

**CONTRO**: Posso trovare il tavolo da quattro direzioni diverse

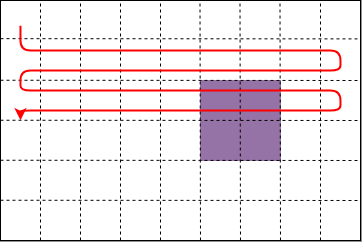
Seconda strategia:

Il robot esplora l’interno con un movimento a “serpente”

**PRO:** Posso trovare il tavolo da due direzioni

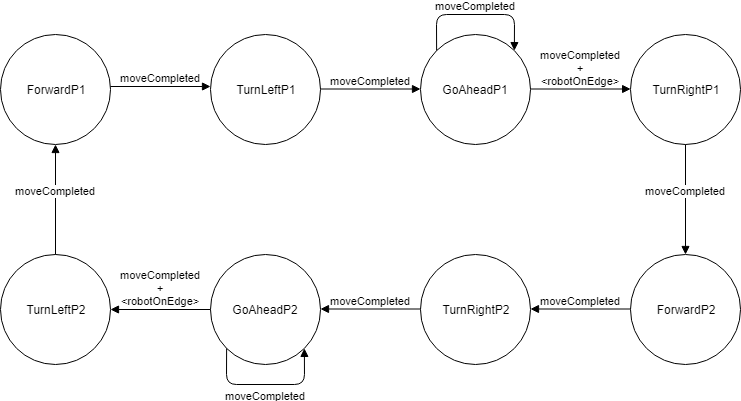
**CONTRO:** Passo di nuovo dai bordi

Terza strategia:

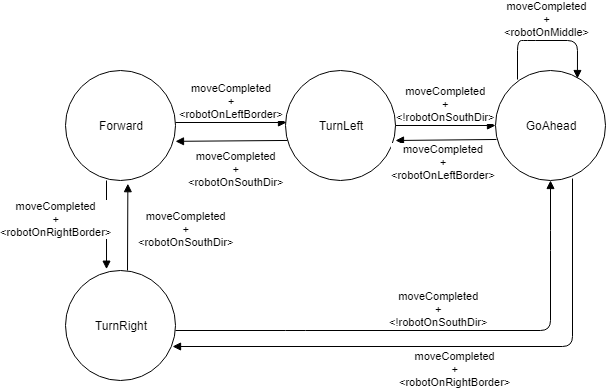
Analogo alla seconda strategia ma questa volta il robot percorre le righe sempre nello stesso verso

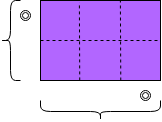
**PRO:** Posso trovare il tavolo solo da una direzione

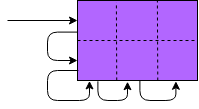
**CONTRO:** Esploro le stesse celle più volte

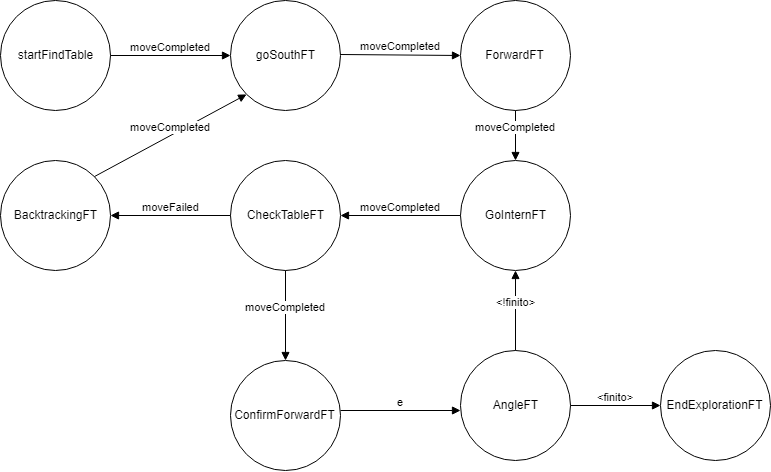
Il giusto compromesso tra “semplicità” e velocità è la seconda opzione che riduce le direzioni in cui è possibile trovare il tavolo senza ripassare troppe volte dalle stesse celle e quindi senza aumentare troppo il tempo di esecuzione della calibrazione. Supponendo di partire dalla cella (0,0) in alto a sinistra e direzione sud, il movimento consiste in due parti, una verso destra ed una verso sinistra.

Il diagramma sopra è sostanzialmente speculare a meno delle direzioni in cui girare che dipendono dal lato da cui si proviene, per questo motivo in realtà l’automa è stato ulteriormente semplificato a 4 stati tramite l’ausilio di guardie.



Per come è stata affrontata l’esplorazione, il tavolo può essere incontrato solo durante lo stato di *GoAhead*. Avendo solamente un sensore, il robot può individuare ostacoli solo se li ha di fronte, per questo motivo è necessario che sia effettivamente di fronte ad ogni cella occupata dal tavolo. Per delineare il diagramma di comportamento sono state utili alcune riflessioni:

l’esplorazione a “serpente” permette di incontrare il tavolo sempre ad un bordo e quindi mai nel mezzo di un suo lato; inoltre, essendo il tavolo rettangolare, l’analisi delle dimensioni è speculare sia che venga incontrato mentre si va a destra che mentre si va a sinistra. Per il calcolo della posizione e delle dimensioni servono la posizione di arrivo e di terminazione (i due pallini) e le lunghezze dei due lati. Alla luce di quanto detto la strategia applicata consiste nel seguente movimento:

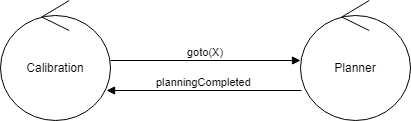
Il diagramma che ne consegue è:

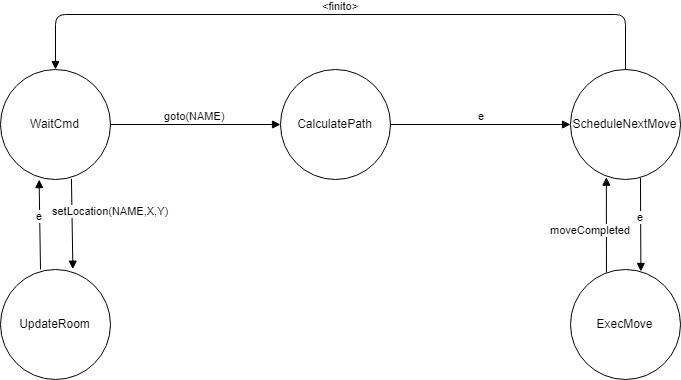
Finita la scoperta del tavolo occorre tornare nella posizione iniziale, questo punto è in stretta relazione con il secondo punto dello sprint ‘*Riuscire a raggiungere un punto specifico della stanza data la posizione attuale’,* per cui occorre analizzare e realizzare la parte di planning.

PLANNER

Lo scopo di questa parte è quello di realizzare un componente capace di schedulare un percorso che parta dalla posizione attuale del robot e termini nella cella obiettivo, tale componente dovrà essere in grado anche di eseguire il percorso dato. Ha senso pensare che una funzionalità del genere non sia utile solo ai fini della calibrazione per riportare il robot nella posizione iniziale ma possa essere utilizzata in svariati ambiti. Per questo motivo è ragionevole delegare queste operazioni ad un nuovo attore che chiameremo *PLANNER* (microservizio) il quale sfrutterà il supporto dato da *itunibo.planner.*

L’interazione con il microservizio potrebbe avvenire in due modi: il primo consiste nel fornire esplicitamente le coordinate da raggiungere, mentre il secondo prevede di interagire con il planner attraverso informazioni logiche, in questo caso con il nome di cosa si vuole raggiungere. La seconda scelta permette quindi di fare in modo che la conoscenza della stanza appena esplorata e quindi dei componenti presenti (RH e tavolo, per ora) sia solamente di un dispositivo e non di tutti che potranno riferirsi a questi in maniera astratta.

Il problema che emerge è quindi come fare in modo che le informazioni relative alle posizioni vengano mantenute dal planner. Una scelta conveniente potrebbe essere sfruttare il supporto per salvare queste informazioni nella base di conoscenza associata al planner con frasi del tipo *location(NAME, X, Y).* Per permettere al planner di estendere dinamicamente la sua conoscenza sulla stanza si potrebbe fare in modo che questo si comporti come una sorta di centrale di **controllo in cui le varie parti del sistema registrano dei nuovi componenti.

L’ultima parte da analizzare consiste nel capire come le mosse individuate dal planner vengano mantenute, anche qui è possibile salvare ogni mossa nella base di conoscenza associata al planner *move(X)* e recuperarle all’occorrenza. Quindi un possibile flow potrebbe prevedere l’estrazione di una mossa e la sua esecuzione e solo nel momento di conferma del movimento eseguito procedere con la mossa successiva.

Fatto questo, far tornare il robot nella sua posizione iniziale rimane semplice in quanto basta far memorizzare al planner la posizione RH e utilizzarlo tramite un messaggio *goto(rh).*

Refactor MVC

Alla luce dell’introduzione di quanto è stato introdotto nel precedente sprint possiamo pensare ad un refactoring model-driven in cui le azioni del robot avvengono come cambiamento del modello stesso. Con queste basi i componenti “mind” hanno il compito di recepire eventuali cambiamenti nel sistema ed innescare i giusti comportamenti.

Tre tipi di messaggi:

* *modelChange* : indica un cambiamento a cui segue una notifica di effettivo cambiamento
* *modelChanged* : è la notifica di effettivo cambiamento
* *modelUpdate*: indica un cambiamento al modello che non necessita di una notifica

Ognuno in tre versioni diverse

* *robotChange/Changed/Update*: per la robotmind, relativa ai movimenti
* *sonarUpdate*: per la notifica dei dati rilevati dal robot
* *taskChange/Changed/Update*: per la butlermind, relativa ai task da eseguire

