Mobile Robotics - Project Report

Contact: Denis.Tognolo@studenti.univr.it, Antonino.Parisi@studenti.univr.it

Denis Tognolo
VR480314
Antonino Parisi
VR468976

1 Introduction

The aim of this project was to implement an high-level controller to keep a mobile robot to follow a wall, likewise the FSM made in laboratory but through the *Behavior Tree* library. The Behavior Tree library is part of the NAV2, and is widely powerful becase of the astraction provided through the nodes and the behaviors that it provides. The tree node is explored in depth from the right to the left side of the tree, there are different type of execution like sequence, fallback, and parallel execution. We've implemented a complex tree of behaviors to handle the main actions and implement newer features, like object collision prevention.

2 Behavior tree introduction

A behavior tree is a data structure, in this case is used as a model of plan execution used and is mostly applied in computer science, robotics, control systems and video games. They describe switchings between a finite set of tasks in a modular fashion. Their strength comes from their ability to create very complex tasks composed of simple tasks, without worrying how the simple tasks are implemented. Behavior tree are based on four kind of nodes:

- Control Nodes that tick a child based on the result of their siblings or/and their own state.
- Decorator Nodes that can alter the result of a children or tick it multiple times.
- Condition Nodes to check conditions without alter the system.
- Action Nodes to perform something that alter the system.

2.1 Control Nodes

The library is provided with both synchronous and asynchronous actions, the difference between sync and async action is the implementation and the idea behind the action. A synchronous action do some computation and it returns SUCCESS or FAILURE, then the tree behavior is defined by the type of sub-tree implemented (sequence, fallback and so on). The asynchronous action return SUCCESS and FAILURE too, but the action has a *"while do"* cycle that every time the action is ticked by the tree and there's no return statement, it means that the computation is running, automatically returns the RUNNING status.

2.1.1 Sequence

A Sequence ticks all its children as long as they return SUCCESS. If any child returns FAILURE, the sequence is aborted. At the moment exists three different types of sequence:

- Sequence
- SequenceStar
- ReactiveSequence

Rules of implementation:

- Before ticking the first child, the node status becomes RUNNING.
- If a child returns SUCCESS, it ticks the next child.
- If the last child returns SUCCESS too, all the children are halted and the sequence returns SUCCESS.

These nodes have different behaviors for sync and async actions, if we consider the SEQUENCE, in synchronous actions the behavior is restart in case of FAILURE and TICK AGAIN in case of RUNNING.

The restart behavior implies the sub-tree restart from the first child, if the return status is *FAILURE*, the TICK AGAIN is the contrary of the the previous one, the action is ticked until it return SUCCESS, all the details can be seen at table 1.

Control Node	FAILURE	RUNNING
Sequence	Restart	tick again
SequenceStar	tick again	tick again
ReactiveSequence	restart	restart

Table 1: Sequence behaviors

2.1.2 Fallback

A Fallback is a node that tries to tackle a problem with different approaches. For example we let's say we want open a door, so we check if is it open otherwise we can look for a key and unlock it, if we don't have a key we can look for the key and after the failure of this action we breach the door to enter. At the moment exists two different types of Fallback:

- Fallback
- ReactiveFallback

rules of implementation

- Before ticking the first child, the node status becomes RUNNING.
- If a child returns FAILURE, the fallback ticks the next child.
- If the last child returns FAILURE too, all the children are halted and the fallback returns FAILURE.
- If a child returns SUCCESS, it stops and returns SUCCESS. All the children are halted.

the details on synchronous and asynchronous nodes can be seen at table 2.

Control Node	RUNNING
Fallback	tick again
ReactiveFallback	restart

Table 2: Sequence behaviors

2.2 Decorator Nodes

A decorator is a node that can have only a single child.

It is up to the Decorator to decide if, when and how many times the child should be ticked.

A decorator node are powerful introducing programming languages behaviors, like boolean negation. Inside this DO WHILE and IF THEN ELSE operator or *RETRY UNTIL CONDITION* node.

Here we report the main decorator nodes:

- InverterNode
- ForceSuccessNode
- ForceFailureNode
- RepeatNode
- RetryNode

Here we report some nodes that we used but there's only the code and not the documentation we provided only the parallel node documentation.

- Do while [Not present inside docs]
- If then else [Not present inside docs]
- Switch n [Not present inside docs]
- Parallel [Not present inside docs]

2.3 Condition Nodes

A condition node is just a class wrapper that implements the if then else flow control, the condition is implemented always in the tick method of the class and return SUCCESS or FAILURE.

2.4 Action Nodes

An action node is a class that represents a single task that is done by the controlled robot , the abstraction of the class is useful to do something inside the tick method, and wen is halted the halt method is also needed to force the robot behave in certain way, i.e. stop the motors if a collision object is detected.

3 Wall follower task

In this work we consider as example of an high level control task, a robot that can find a wall in an environment and than starts to follow it, the following side choice is made by the most near wall, so if the most near wall is to the left then the side choice is left as expected. If the robot reach a point where is in front of a wall can do the routine of alignment and consequently follow again the wall. The most peculiar case is represented when the robot is near a corner (case 3 of figure 1), is the case where the robot follow a wall and at certain point the wall disappear from his left/right, in this case we call the routine of Follow slim wall and turn around the corner following the path imposed by following correctly the wall.

We also implemented as requested the rewind action, that stores all the previous actions and do it in the opposite order with the LIFO metric. The other feature is the collision prevent, where if is detected an object or obstacle under the 90% of the distance threshold it runs this routine, turning 180° and going back to a safe position.

We refer to the figure 1 to let you understand the main cases that we covered in this project, the first is the starting case, here we call the find a wall action, then the case two is the follow wall action routine, the case three is the corner, and last, the fourth case is the align case, here we following a wall and we find a wall, so we do the alignment to the next wall.

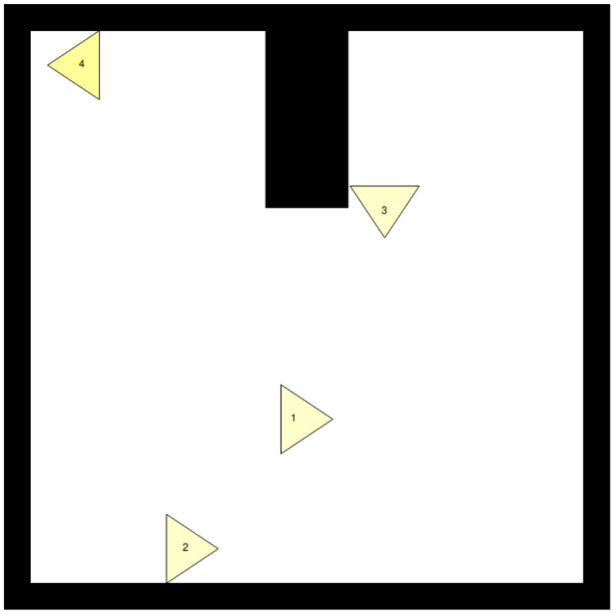

Figure 1: Case of studies

4 Phases of implementation

The high-level controller is visible at figure 2. We firstly focused on how to integrate the behavior tree within the nav2 stack, but after few days we understood that was buggy and this was confirmed by the academic tutor of the course Francesco Trotti, he suggested us to follow the simpler examples that can be found inside the repository and the docs website, and start from those examples. The first way we tried to implement the FSM was in synchronous way; the main problem that we encountered was the syncronous action behavior that not permit to read sensor data and update it, so we switched to asyncronous nodes. The asyncronous action must be single classes that performs a single behavior, like go straight until a wall appears and so on. This time we were able to read sensor data, and publish to ROS2 topic $\backslash cmd\_vel$ the velocity message. With asynchronous actions the main problem was understand how implement it because there were no examples or documentation on it.

So we managed to implement those action with a *DO WHILE* structure, this structure return *SUCCESS* only once the action is completed. With this structure we had some struggles with the parallel node to check the conditions to stop the main execution of high-level control and start the rewind routine, we report that actually the documentation provided is really poor and not well documented or clear to understand; in facts we tried a plethora of combinations to understand the behaviors of each node and implement the FSM correctly.

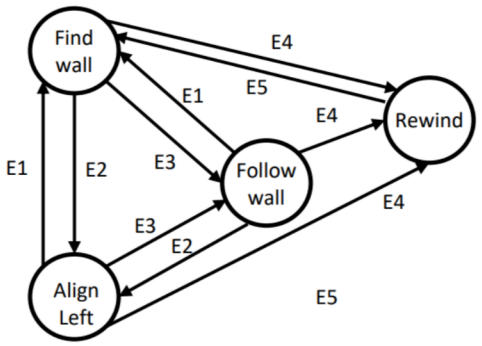

Figure 2: FSM

5 Structure of code

The main program of this project consist in a very few lines of code, in particular we initialize ROS2 and spin the Wall_Follower class that we present you in details in the following subsection. Here also the tree in XML-like structure must be provided.

5.1 Wall_Follower

We implemented a single class called Wall_Follower, extending a ROS2 Node, where we manage with callback functions all the ROS2 topics to control correctly the TurtleBot. This class contains tree methods: a constructor, a laser callback, and a control loop. The goal of the first one is to prepare all the class parameters, create a ROS2 publisher linked to the $\backslash cmd\_vel$ and a subscriber linked to the laser callback, and a timer linked to the control loop. Here we also build the desired behaviors tree, and so we have to specify an action/condition class for each action/condition node of the tree and so instantiate them passing the needed arguments. The only parameters of this class constructor is a string that specify the tree structure in a XML-like format.

In the laser callback, that runs every time something new is written in the $\backslash laser\_scan$, we elaborate the array of laser scans from the robot LIDAR to obtain the presence or not of an object in the frontal, right or left areas, according to a distance threshold. In the control loop, that runs at a certain frequency according to the timer, we simply publish the $\backslash cmd\_vel$ to actually move the robot and tick the tree so that the program can elaborate the next move to perform.

The actions are implemented inside the *action.h* file, each action extends the Asynchronous action of BTree library and implements the constructor, to load the needed arguments (all pointers to some Wall_Follower parameters), and the tick method. The tick method of an action is the method performed when the tree tick this specific action during its execution. In order to understand better the structure of the code we can say, if we want implement another action it must be implemented inside actions.h, and initialized inside wall fol-

lower only if it is present inside the XML tree provided to Wall_Follower constructor.

There is also a package named Direction Controller, it was a first way to control alignment of the robot respect to the wall (LEFT/RIGHT), and execute the rewind routine.

5.2 Actions and Conditions

In this subsection we present you all the leaf nodes of our tree (actions and conditions) used to perform the wall follower task as showed in figure 3, and the idea behind their algorithms.

5.2.1 Find wall

Go straight until a wall in front region is detected, the velocity is computed as quadratic distance from the wall with a maximum limit (1 rad/s).

5.2.2 Side choice

Set the side to follow according to the closest wall, if the nearest side is left the chosen side is left.

5.2.3 Align

Turn, left or right according to the chosen side, until the front region is empty.

5.2.4 Follow wall

Go straight until a wall in the front region is detected or the side to follow region is empty.

5.2.5 Follow corner

Follow a circle path until a wall appears in the side to follow region (and a wall in the front region is detected).

5.2.6 Turn

Do a 180 degree motion over the z axis.

This is the only action we have implemented with ports, in fact every time we put a Turn node on the tree we can set different angles, time to perform it and direction. Turn name="Turn" angle="180" time="10" direction="clockwise"

5.2.7 Rewind

Repeat all action in history.

5.2.8 Collision detector

Return FAILURE if something appear in the frontal region.

5.2.9 Key detector

Return FAILURE when a key is pressed.

5.3 Control Nodes

The idea behind the execution flow that we wanna obtain in order to realize the wall follower task was: IF condition DO secondary_task ELSE primary_task that in our case of study become IF key_detected DO rewind ELSE follow_wall. The condition must be checked even when the primary task is RUNNING and not only at its beginning so a Reactive control node is needed in order to model it.

To obtain this behavior in fact we need a ReactiveFallback with a Sequence of condition and secondary task as first child, and the secondary task as second child. So if the key is not detected the condition return FAILURE, causing the sequence returning FAILURE and so only the secondary task is ticked. On the other hand if the key is detected the condition return SUCCESS, causing the sequence proceed with the secondary task. Here we don't use a ReactiveSequence in fact we don't wanna check the key during the rewind task.

While the secondary task is executing, this node return RUNNING every tick, and so also the sequence return RUNNING. This cause the ReactiveFallback not ticking the primary task until the secondary task ends and so the condition could be ticked and eventually return FAILURE.

With the same pattern we can also model IF collision_detected DO safety_sequence ELSE main_sequence where the main_sequence is the ReactiveFallback just discussed. This also make the collision detection working whatever task is executing (rewind or wall_follower).

It turns out the following tree.

Figure 3: Behavior tree created via groot

6 Execution example

Here we present you an example of execution where a Collision occurs at Tick 219, and a Key is pressed at Tick 423.

It's important to consider that Conditions print something every time are ticked, Actions print something every time a different action starts.

```
—— executeTick() 0 ——
[ Finding a wall ]
Is a collision detected? NO
Is a key detected? NO

—— executeTick() 1 ——
Is a collision detected? NO
Is a key detected? NO
.
.
.
—— executeTick() 153 ——
[ Follow left ]
Is a collision detected? NO
Is a key detected? NO

—— executeTick() 154 ——
[ Aligning ]
Is a collision detected? NO
Is a key detected? NO
.
.
.
—— executeTick() 219 ——
Is a collision detected? YES

—— executeTick() 220 ——
[ Turning ]
.
.
.
—— executeTick() 242 ——
[ Finding a wall ]
Is a collision detected? NO
Is a key detected? NO
.
.
.
—— executeTick() 423 ——
Is a collision detected? NO
Is a key detected? YES

—— executeTick() 424 ——
[ Turning ]
Is a collision detected? NO

—— executeTick() 425 ——
Is a collision detected? NO
.
.
.
—— executeTick() 445 ——
[ Starting Rewind ... ]
[ Aligning ]
Is a collision detected? NO
.
.
.
—— executeTick() 523 ——
[ ... Rewind Complete ]
[ Finding a wall ]
Is a collision detected? NO
Is a key detected? NO
.
.
.
```

When the program starts Find_Wall sequence run and check both condition every thick. At tick 219 a collision is detected so the safety routine (Turning) starts and here system doesn't care about key or collision conditions. At tick 242 the safety routine ends and the Wall_Follower restart with the checking of both conditions. At tick 423 a key is detected so the Rewind sequence (Turning + Rewind) starts and here system doesn't care about key, but still check for collisions. At tick 523 the Safety Sequence ends and the Wall_Follower restart.

7 Experimental parameters

For having a good behavior in both simulation and physics system there are few parameters that must be properly setted and that we found experimentally.

Considering 0° as the front of the robot (and angle growing anti_clockwise) we set the frontal region from -30° to +30°, the right regions from -30° to -90° and the left regions from +30° to +90°.

We set a distant threshold equal to 0.4 to determines if a region is occupied or not and for the collision threshold we consider 80% of it.

Another important parameters is the max linear velocity, to prevent the robot become too fast when the linear velocity is proportional to the square of the distance.

8 Conclusion

Working on this project we are really appreciating the versatility and modularity of behavior trees in fact if we want to add new functionalities it would take only few time. In particular we only need to code a proper new actions or conditions inside the action.h file, add the node in the XML description inside the main.cpp, and bind them in the constructor of the class inside wall_follower.h. The only issue with this choice was the poor and sometime misleading documentation and support, above all considering similar project like the behavior tree in Unity.