Control and Behavior Mechanisms for the UR10e Robotic Arm

Fábio Alves[1] and Marcelo Fraga[2]

***Abstract*— The control of robotic arms, despite the existence of multiple frameworks abstracting the underlying inverse kinematics, can be daunting to the inexperienced programmer. We propose a set of simple to use and mechanisms, based on ROS services and Behavior Trees that allow the user to perform complex tasks on the UR10e, with little effort on the programming side.**

We have successfully tested this mechanisms on the robotics simulator gazebo and performed an object grasp with the UR10e using only visual programming.

I. Introduction

During development of systems comprising robotic arms, the creation of actual control behaviors for the manipulator can sometimes represent a huge chunk of the work needed to be done, since it requires understanding underlying frameworks, like *MoveIt*, or communication protocols.

In that sense, this article aims to present a standard and modular interface that can be used to control robotic arms. For this specific case, the manipulator used to test every single functionality was the Universal Robots 10 e-series.

The main purpose of this system is to essentially provide an easy to understand interface, using *ROS* services, to ultimately allow a user of the manipulator to control it without having to dive deep in the actual robot control methods.

The following sections are intended to give further explanation regarding the core components and interaction points of the system: services and behavior trees. Functional examples and some results are also demonstrated in section V.

II. Architecture

The foundation for the robot interface is based on two already existing packages. First, a package that helps launching the robot driver and other nodes like like *MoveIt* by creating an extra level of abstraction, *iris_ur10e* [1] . It provides a way to launch a simulated environment with the robot. Work related to improve this simulated world was done to allow for grasping routines since they don't work by default given the actual simulation settings (footnote 7). Regarding the second package, this is the actual base of the interface. That is, the core structure and composition of the overall architecture (Figure 1) is defined by this aforementioned package, *iris_sami* [2].

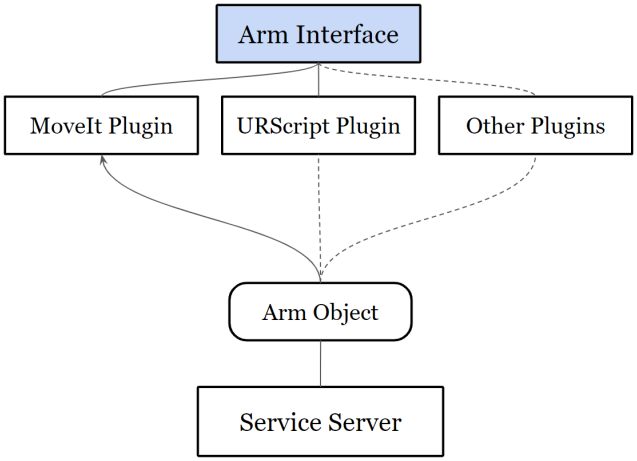

Fig. 1: High level architecture of the system

Figure 1 offers a good representation of the actual abstraction provided by the system. To discuss further on the inner works of it, it's worth noting that the node that ultimately provides all the functionality to the end-user is the service server. This is a *ROS* node that brings up all services needed for the programmer to interact with the robot (section III). It works by interacting with the internal arm representation instantiated here. This arm object has the main purpose of instantiating one of the available plugins depending on the given context. Basically, each plugin serves as a different low-level interacting method and is intended to differentiate the various already mentioned approaches related to the robot control. They must implement the *Arm Interface* base functions and might eventually contain more features depending on the situation. For this specific case, two different plugins are presented. One wraps the *MoveIt* functionality and allows abstraction not only regarding the actual robot interaction but also concerning the underlying environment: real or simulated. The other allows controlling the real robot directly using a TCP/IP connection.

III. Services

The main interface point with the manipulator ends up being a list of *ROS* services that is intended to provide a high level abstraction over the main interaction methods like *MoveIt* or regular communication protocols used by the robot.

In this specific case, and considering that these services, introduced here, represent only a part of what the full service list might eventually be, the services comprise core movement actions as well as status information regarding the

*This work was done for the RMI Course

[1]Fábio Alves is a student in the MSc Computer and Telematics Engeneerin, University of Aveiro, PT fabioalves98@ua.pt

[2]Marcelo Fraga is a student in the MSc Computer and Telematics Engeneering, University of Aveiro, PT fraga16@ua.pt

[1]https://github.com/iris-ua/iris_ur10e

[2]https://github.com/iris-ua/iris_sami

Tool Center Point (TCP) pose or the full joint configuration data.

A. Core Services

Moving the robot around is ultimately the goal to achieve when interacting with a manipulator. The service server provides utilities to move the robot in the simplest way possible. Relatively easy to describe joint configuration movement or even end-effector, absolute or relative, pose goal assignment can be used to move the robot around. More complex routines usually comprise two or more of this simpler movement actions and are discussed further in section IV where behavior trees are used to create more elaborate procedures. A table representation of the core action functionality is presented below in Table I.

Service	Arguments	Description
/joints	float64 [j1, j2, j3, j4, j5, j6]	Sends robot to provided joint configuration.
/pose	float64 [x, y, z, rx, ry, rz]	Sends robot to provided pose.
/move	float64 [x, y, z, rx, ry, rz]	Moves end effector relative to the current pose.

TABLE I: Core movement action services

Besides the actual movement functionality, other core features comprise the ability to set the current speed at which the manipulator is running as well as the capability to get information regarding the robot. The ability to control the gripper by opening or closing it, was also added.

B. Extra Features

While most of the discussed services represent the essential interaction with the a robotic arm, some extra functionality was added to increase even further the abstraction level present here. These features arise as a way to speed up development on projects where arm manipulators were used.

Since these features use the previously mentioned core service functionality and because they are independent of inner robot interaction level, they are implemented directly in the higher level arm class. They are also designed to be completely self-contained and modular. That is, as observed in Figure 2, they are added to the side of the core functionality and other features should never depend on these.

More functionality can - and is encouraged to - be added later, but some examples of already developed features are demonstrated in Figure 2. First, a way to chain multiple movements together as well as a way to define this chain in a text file are included. These are intended as a way to facilitate complex routine making without referring to the more in-depth behavior trees (section IV). To show the modularity present here, it's worth noting that the file parser can actually be changed and attached directly to the arm object, as long as it fulfills the necessary function prototype definitions, allowing for different text file representations. After that, a functionality that allows users to refer to specific joint configurations using an alias is added to improve efficiency when working with the manipulator. This feature consists of

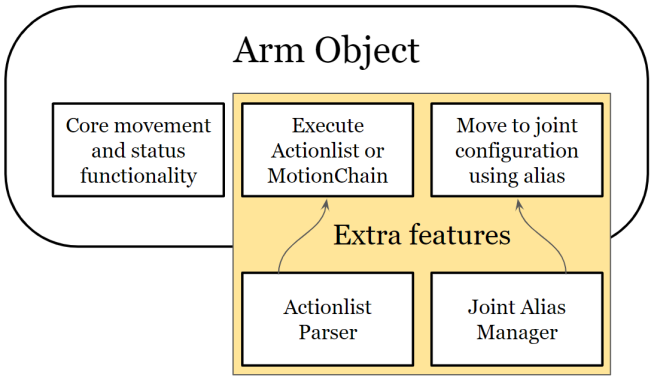

Fig. 2: Arm object modular nature

a simple way to save the current joint configuration in a file, using an alias, so that it can be used later to move the robot to that exact same position, without having to refer directly to the angular information of each joint. The list of services that enable these extra features to be ran is shown below in Table II.

Service	Arguments	Description
/joints_alias	string alias	Sends robot to joint configuration referred by the alias.
/save_joints_alias	string alias	Saves current joing configuration with the provided alias.
/actionlist	string actionlist	Executes the array of actions provided in the actionlist file.

TABLE II: Extra functionality services

IV. Behavior Trees

Behavior Trees are a tool used in computer science, robotics and control systems to create complex behaviors from simple tasks and therefore, increase modularity in the control structures [1]. They can be seen as a tree of hierarchical nodes that controls the flow of decision and the execution of tasks, called actions. Besides this actions, that are also control nodes such as Sequence, Fallback and Decorator which focus on the flow of the actions. To further extend the control interface described before, the BehaviorTree.CPP[3] library was implemented and tested to provide an easier, visual and modular form of robot behavior programming. These trees are described in XML format and for ease of programming and designing, the Groot[4] graphical editor was used. The first aspect to focus, before creating any behavior tree is programming its actions.

A. Core Nodes

Each node that represents a specific task needs to be programmed and structured in a specific way. They are the leaves of the tree, therefore cannot have children. These

[3] https://github.com/BehaviorTree/BehaviorTree.CPP
[4] https://github.com/BehaviorTree/Groot

nodes were implemented according to the services described in section III. For each service was created a class which respected the library's nodes interface, therefore must implement specific functions such as *povidedPorts*, which declares any input/output parameters that a node might have and a *tick* function, in which the code of the action is written.

The list of nodes present in this system are as follows:

- **UR10eOnline -** Simple condition that calls the status service and checks if the robot is operational. Must be used in the beginning of every behavior tree.
- **Velocity -** Sets the robot's velocity by calling the */velocity* service with a float input parameter.
- **Grip -** Calls the */grip* service to close the gripper.
- **Release -** Calls the */release* service to open the gripper.
- **Joints -** Takes as input ports 6 floats, that correspond to joints angles in radians, calls the */joints* service passing those arguments and waits for the completion of the service.
- **JointsAlias -** Takes as input port 1 string, that correspond to a joint configuration previously saved and calls the */joints_alias* forwarding the input ports.
- **Pose -** Takes as input 6 floats, which correspond to the number of parameters needed do describe a pose in 3D space (x ,y ,z , roll, pitch, yaw) and calls the */pose* service forwarding the input ports.
- **Move -** Exactly the same logic as before but calls */move* service to perform a relative move.

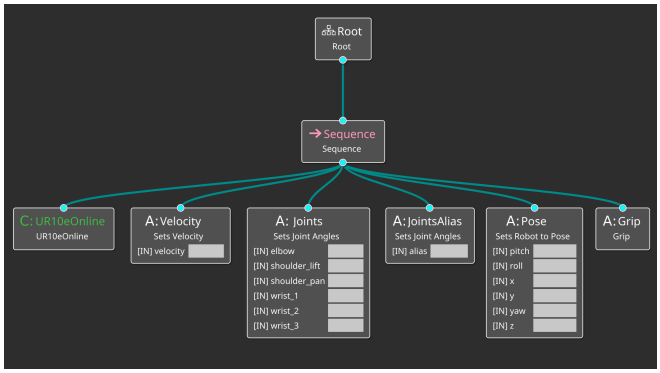

Fig. 3: Behavior Tree showcasing robot specific nodes

All of these nodes, seen in Figure 3 were created to implement all the functionalities provided by the services, but extra behavior tree specific nodes were created.

B. Extra Features

The relevance of behavior trees is that the functionality coded into a node is endless, and in the specific case of control behaviors there are important functionalities that are not robot specific:

- **Pause -** Pauses the execution of the behavior tree by subscribing to a specific topic (*/continue* and wait for a message to be published. Only then it resumes execution. It is very useful for debugging purposes.
- **Sleep -** Takes as input port a float which represents the amount of seconds the node will sleep for.

Examples of use of these nodes can be seen in Figure 4.

In the context of robot behavior, many more nodes that are not robot specific can be created and integrated, such as environment related (sensors, perception) nodes but for the purpose of this project were left to be developed in the future.

C. Example Behavior

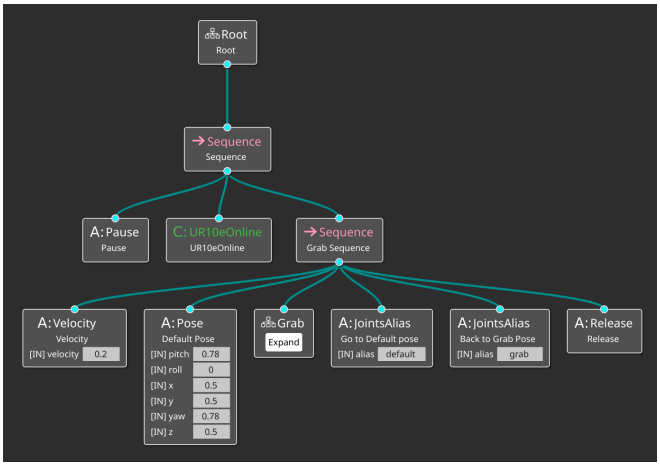

Fig. 4: Complete example of a robot behavior.

To showcase and make use of the implemented nodes, a grabbing behavior was programmed using them. The program starts in the pause node, waiting for the user to initiate the execution. Checks if the robot is online and operational. In any sequence of a behavior tree, if a leaf node returns failure, the sequence stops, and also fails, which means, in order for the next node in a sequence to execute, the previous must execute successfully. Next, the velocity is set and the robot is sent to a default pose, directly coded in the nodes parameters. Then the grabbing sub-tree is implemented.

This is another advantage of the use of behavior trees, in which any behavior described by a tree, can be re-implemented in another behavior tree as a sub-tree, once again increasing the modularity of the control systems. In this case, the gabbing sub-tree, described in Figure 5 uses a predefined joint alias (*grab*) to send the robot to the grabbing position, then, it moves the robot 10cm in the direction of the end-effector in order to place the 2 grip fingers in the right position to grab the object, which is the next action. After grabbing the object, it sleeps for 1 second to give time to the simulator to process the collision that must happen between the gripper fingers and the object (only in a simulated environment). Then, is moves the robot back to the original grabbing position, completing the grabbing sub-tree.

After the grabbing sub-tree is completed, the robot moves to another position to showcase the object manipulation. Since until this point all functionality has been shown, the robot moves back to the grabbing position and releases the object in the spot where it previously grabbed it from.

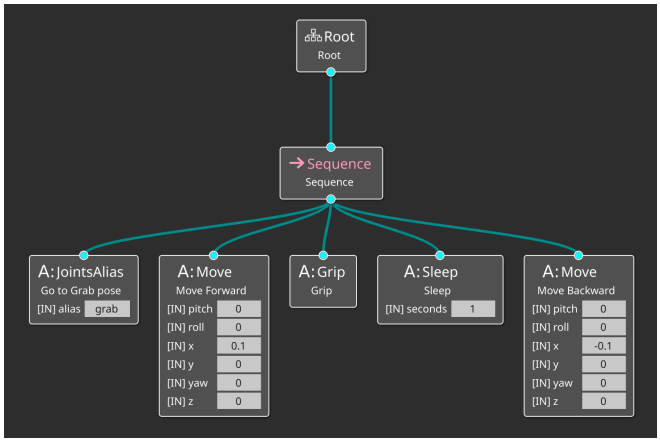

Fig. 5: Specific example of a grabbing behavior.

V. Results

Regarding the testing of the presented functionalities, both a real and simulated environment were used. As for the real ambient, an already existing context allowed for actual testing of the features. That is, since this environment consists of an on-going project using the UR10e, it granted the ability to prove the value of the presented system. Previous control of the robot was time consuming and led to mistakes on moving the manipulator. The presented interface increased not only productivity related to other essential features but also accuracy when dealing with simpler or even more complex movement actions because of the extra features added such as Behavior Trees or the more straight to the point action lists.

When considering the simulated environment, and because it proves its usefulness when experimenting with untested code or unknown behaviors, work to improve the current state of the already provided ambient (footnote 1) was done. The overall working state of this simulation is pretty good but a critical problem regarding the grasping of gazebo models was detected. A solution based on a simulated grasping plugin for gazebo (footnote 7) was included in the original *iris_ur10e* package and the simulation now has full capability regarding object pick and place.

To verify the correctness of the solution, and because one of the main purposes was for it to work independently of the current selected ambient - simulation or real robot - all simpler movements and more complex features were tested in both environments without changing anything. Apart from the *URScript* plugin that is designed to work only on the real robot, the *MoveIt* add-on works perfectly in both environments. Simple grasping routines were defined using behavior trees and ultimately tested in the simulation, to pick up a simple block gazebo model, and in the real environment, to pick random objects.

VI. Steps to Reproduce

In order to implement and replicate the functionalities described in this paper, the user must create a similar ROS workspace and install some dependencies.

- **BehavioralTreeCPP -** This library can be found on the Ubuntu's collection of ROS packages and can be installed uding the apt package management tool. Its dependencies are automatically installed with this package.
- **GRoot -** Dependencies, installation and usage instructions ca be found in the application's GitHub page, referenced in section IV.
- **Catkin Workspace -** The ROS worspace needed to test these functionalities is composed of 4 packages and can be replicated in the following way:
  - $ source /opt/ros/distro/setup.bash
  - $ mkdir -p /catkin_ws/src
  - $ cd /catkin_ws/src
  - $ git clone iris_ur10e repository [5]
  - $ git clone iris_sami repository [6]
  - Follow these instructions to fix the collision of objects in the grasp function [7]
  - $ rosdep install –from-paths src –ignore-src -r -y
  - $ catkin build

The steps above are meant to simply configure and install the necessary workspace. In order to execute the showcased functionalities, the user must execute the following commands:

- After building the workspace and sourcing the environment variables:
- $ roslaunch iris_ur10e sim.launch gui:=true rviz:=false
- $ roslaunch iris_ur10e spawn_box.launch
- $ rosrun iris_sami server.py
- $ rosrun iris_sami ur10e.cpp ur10e.xml
- Optionally start the Groot live tree monitor and connect is to ROS.
- Publish any string message to the /iris_sami/continue topic in order to start the execution of this example behavior tree.

VII. Conclusion

Even experienced users in this context, which are used to directly implement framework methods and deal with all of its complexity, can appreciate the simplicity of an even higher level of abstraction to the control of robotic arms. The way behavior trees work also allow the programmer to design more carefully the robot's control systems. In general, the mechanisms presented proved themselves very useful and are prone to be extended in the future, with perhaps more variety in service parameters and extended functionalities in the way of new behavior tree nodes.

References

[1] M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees," in IEEE Transactions on Robotics, vol. 33, no. 2, pp. 372-389, April 2017, doi: 10.1109/TRO.2016.2633567.

[5] https://github.com/fabioalves98/iris_ur10e
[6] https://github.com/fabioalves98/iris_sami
[7] https://github.com/JenniferBuehler/gazebo-pkgs/wiki/Installation