Task-Level Motion Planning for Multi-Manipulator System

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Rajendra Singh
(111601017)

under the guidance of

Dr. Chandra Shekar

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Task-Level Motion Planning for Multi-Manipulator System**" is a bonafide work of **Rajendra Singh** (**Roll No. 111601017**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my supervision and that it has not been submitted elsewhere for a degree.*

Dr. Chandra Shekar

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

I would like to express my sincere gratitude to my mentor **Dr. Chandra Shekar** and **Mr. Girish Kumar PR**. I would also like to thank my project coordinator **Albert sunny** who gave me the golden opportunity to do this wonderful project at **Gadgeon Smart Systems Pvt Limited, Kochi**. Also, I would also like to thank my office colleague specially Abhinand and Indu for their critical queries, endless help and support throughout this project.

Many thanks and sincere gratitude to my BTP evaluation panelist, **Dr. Deepak Rajendra Prasad** and **Dr. Satyajit Das** for their helpful suggestion and guidance due to which I'm able to improve my work further.

Contents

List of Figures

Chapter 1

Introduction

This project is related to repair any space satellite using multiple robotic manipulator situated on another repair satellite. Currently, it is in feasibility phase and we are doing various simulation, experiment and tests to understand what is possible and what's not. My main work in this project is to coordinate 2 different arms and perform complex manipulation task like repairing various part of satellite. It involves solving kinematics and inverse kinematics for multi-manipulator system. One of the solutions for this is using a particle swarm optimization algorithm which I had used in the BTP phase one. Here, I'm using motion planning framework like Moveit, Move_task_constructor with ROS.

1.1 Problem Statement

Perform complex manipulation task like pick and place, building structures and pouring in multi-manipulator system. **More detailed subtask/problem are as**:

1. Simple Joint space planning(move_group)
2. Simple Cartisian space planning(move_group)
3. Pick & Place Task (move_group)
4. Simple Joint space planning(MTC)

5. Simple Cartisian space planning(MTC)
6. Pick & Place Task(MTC)
7. Multi arm simple Joint space planning
8. Multi arm simple Cartisian space planning
9. Multi arm Simple Pick Place Task(own work)
10. Multi arm Complex Pick Place Task(IIT)
11. Multi arm planning using Serial container
12. Multi arm planning using Parallel container
    12.1 Alternative
    12.2 Fallback
    12.3 Merger
13. Multiple task
14. Single arm pouring task
15. Complex Multi arm pouring
16. Complex Multi arm pouring task with stages intermixing
17. Complex task with orientation constraint imposed

1.2 Goal of Project

Goal of this project is to do motion planning for complex task like pick-place and pouring using multiple-arm which are cooperating with each other to effective do some task. Going further, I've improve the recently released open source motion planning frame MTC. I have analysed time and space complexity of these motion planning algorithms with simulated robots using ROS, thus optimizing the same. I've worked with more complex and constraint environment for motion planning with multiple arms.

1.3 Organization of The Report

chapter 1 : We introduced the problem statement, discussed goal for this semester and organisation of this report.

chapter 2 : Here, I would take you through my prior experience with Robotics Manipulator i.e Uarm Shift Pro Robotics Arm for motion planning using move_group.

chapter 3 : Here, we'll introduce MTC and motivation behind using it.

chapter 4 : Here, We'll have in depth look at MTC planning, container, stages etc.

chapter 5 : Here, We'll port MTC to ROS2 to be able to use it with moveit2 and hence to be able to do real-time motion-planning.

chapter 6 : Here, We'll implement motion planning to perform some task with the help of MTC. All of the codes used in these implementation can be found here on my GitHub repository[3].

chapter 7 : Here, We'll conclude my work and discuss future work.

Chapter 2

Prior Work

2.1 Introduction

During my summer internship(2019) and BTP phase one, I used robotic manipulator and motion planning frame work like Moveit to perform some simple tasks. Thus, this work becomes the basis of more complex manipulation task performed in this project. I had mainly worked on the forward and inverse kinematics of manipulator.

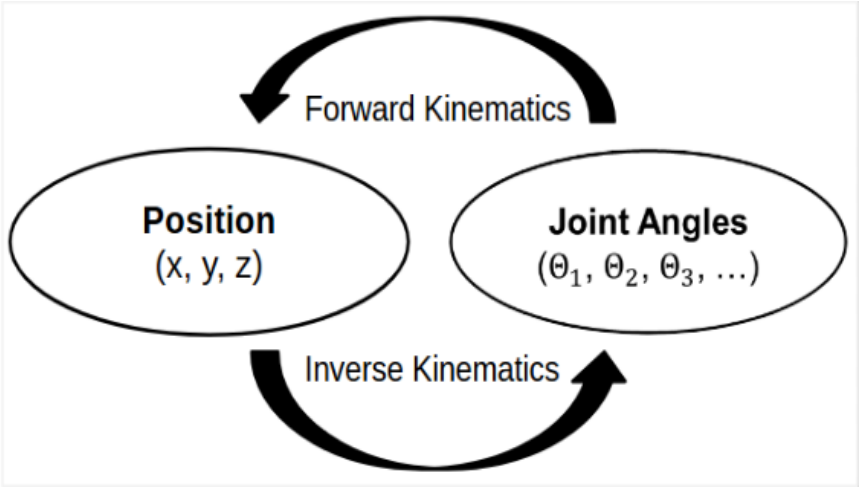


Figure 2.1 Kinematics in robotics arm

2.2 Denavit–Hartenberg(DH) forward kinematics Method

A commonly used convention for selecting frames of reference in robotics applications is the Denavit and Hartenberg (D–H) convention which was introduced by Jacques Denavit and Richard S. Hartenberg. In this convention, coordinate frames are attached to the joints between two links such that one transformation is associated with the joint, [Z], and the second is associated with the link [X].

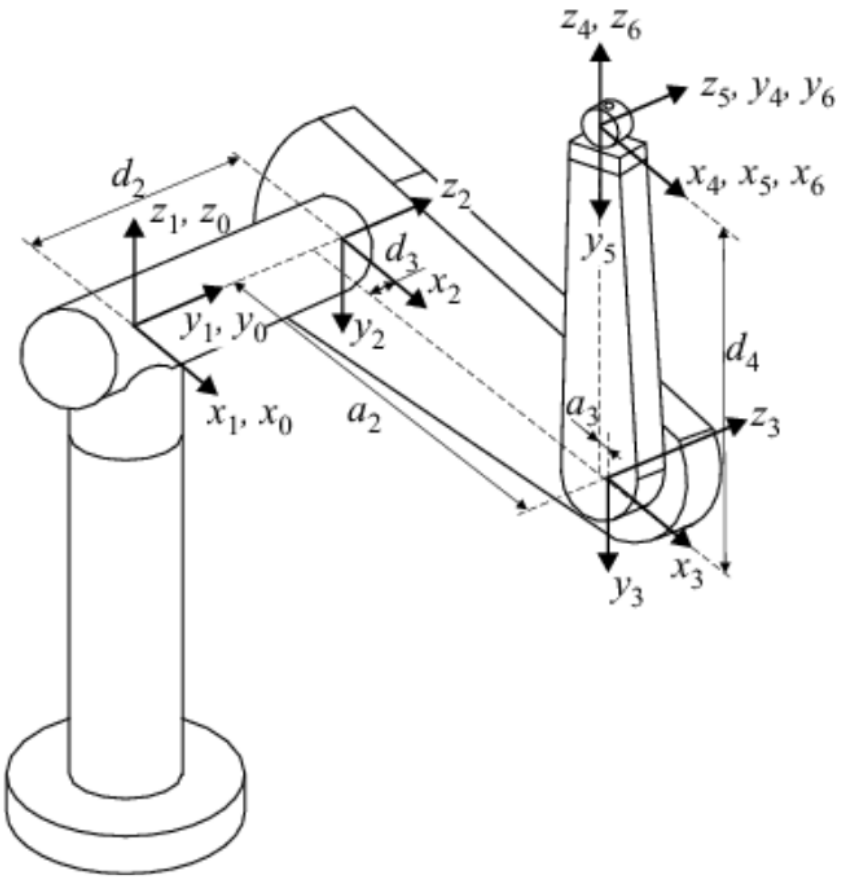


Figure 2.2 Assign frames to each link in arm, Image source

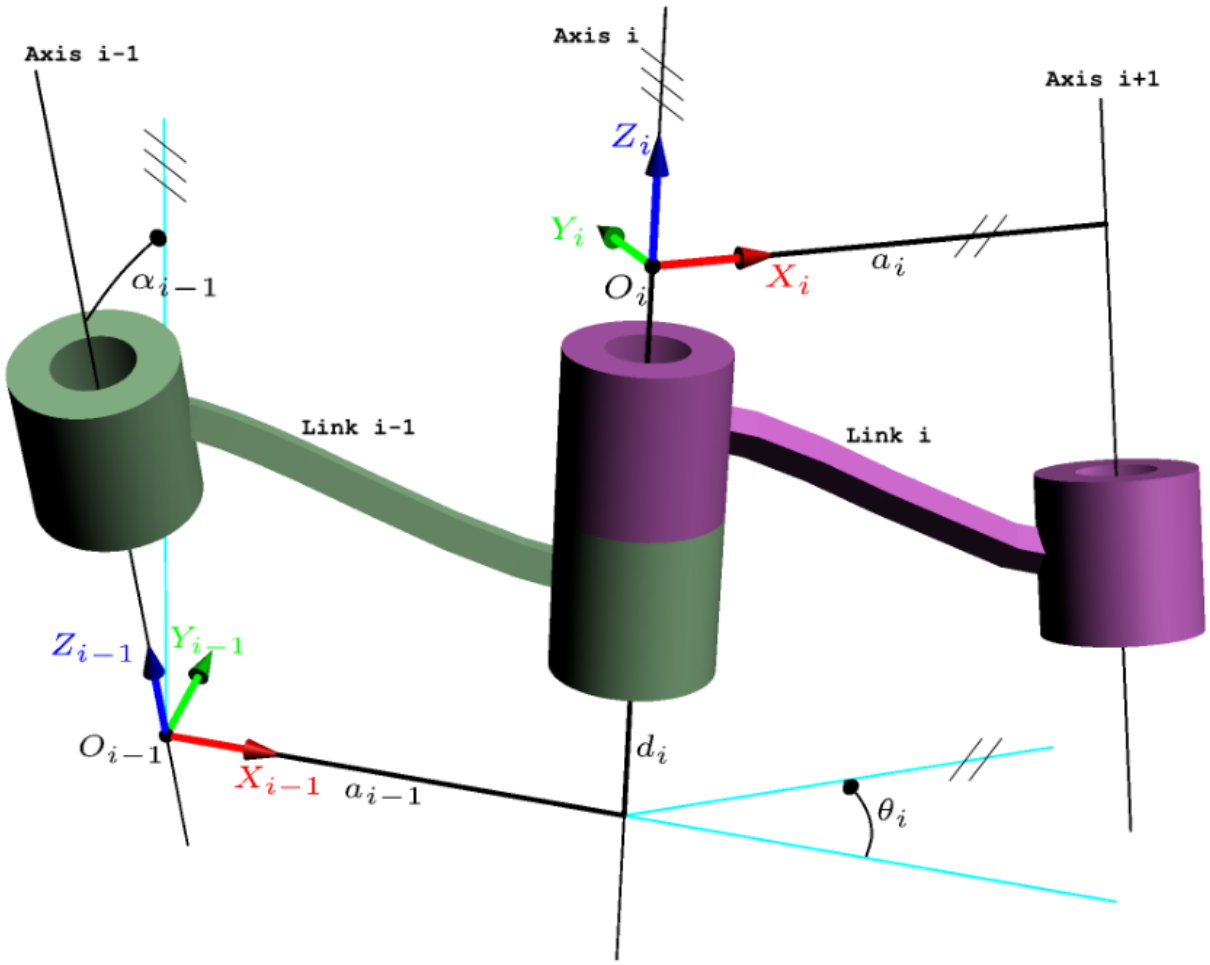


Figure 2.3 Computing the DH transformation between any two links, Image source

The following four transformation parameters are known as D–H parameters:

d: offset along previous z to the common normal

θ: angle about previous z, from old x to new x

r: length of the common normal (aka a, but if using this notation). Assuming a revolute joint, this is the radius about previous z.

α: angle about common normal, from old z axis to new z axis

$$
{}^{n-1}T_n = \left[\begin{array}{cccc}
\cos\theta_n & -\sin\theta_n & 0 & a_{n-1} \\
\sin\theta_n \cos\alpha_{n-1} & \cos\theta_n \cos\alpha_{n-1} & -\sin\alpha_{n-1} & -d_n \sin\alpha_{n-1} \\
\sin\theta_n \sin\alpha_{n-1} & \cos\theta_n \sin\alpha_{n-1} & \cos\alpha_{n-1} & d_n \cos\alpha_{n-1} \\
\hline
0 & 0 & 0 & 1
\end{array}\right]
$$

Figure 2.4 Calculation the transformation matrix

Once we have the transformation matrix between all ad-joint joints, we can use it to easy do coordinate transformation and hence the forward kinematics.

2.3 Solving Inverse Kinematics(IK) using PSO

During my BTP phase, I had used particle swarm optimization(PSO) algorithm to solve the Inverse Kinematics problem in robotic manipulator. In general we use DH method to solve IK.[4] Inverse Kinematics is the inverse algorithm of Forward Kinematics. The Forward Kinematics algorithm takes a target position as the input, and calculates the pose required for the end-effector to reach the target position. Whereas Inverse kinematics find the joint value required for arm to reaches the given coordinate. I have used PSO to solve the IK, and detailed code for the same can be found here.

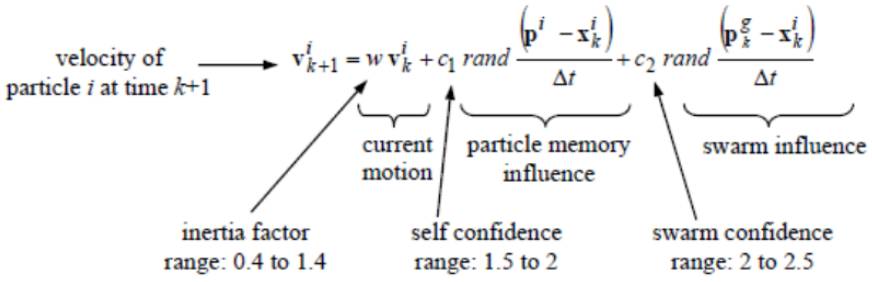


Figure 2.5 PSO equation, image source

In simple words, velocity is the distance achieved/travelled by particle from current position

Simple equation is

v= (pbest-present)+ (gbest-present), random, social , learning factors can also be added

where,

v= velocity / path direction

present= current position of the particle

pbest= best position of specific/ single particle

gbest= best position of all the particles/ swarm

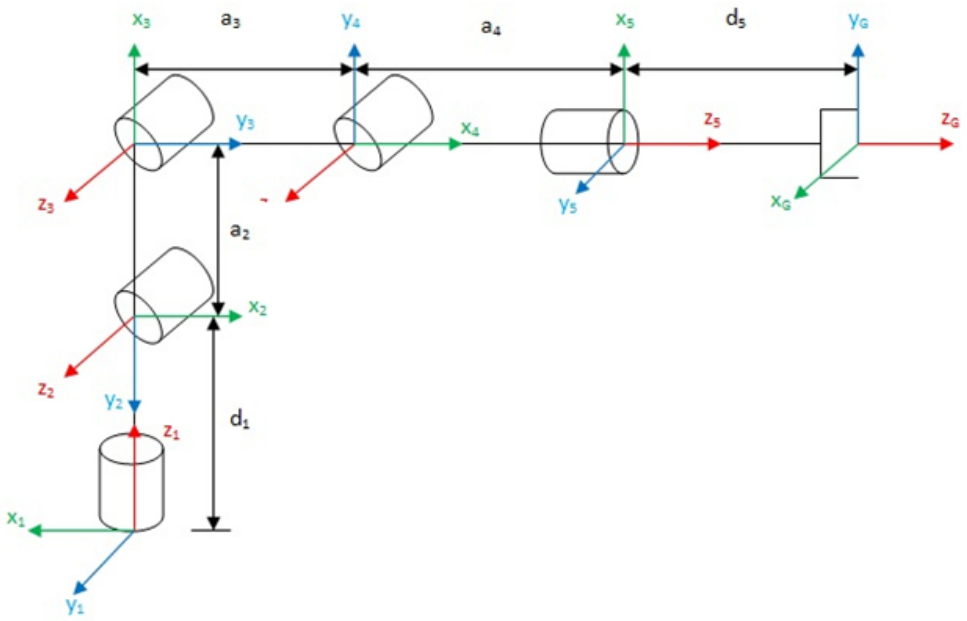


Figure 2.6 IK : Finding inverse kinematics solution for 6 dof robotics manipulator, Image source

Pseudo Code

```
# find forward kinematics pos of end effector
def get_end_tip_position(params):
    # Create the transformation matrices for the respective joints i.e
    find t_00, t_12, t_23, t_34, t_45,t_56


    # Get the overall transformation matrix
```

```
    end_tip_m =
    t_00.dot(t_01).dot(t_12).dot(t_23).dot(t_34).dot(t_45).dot(t_56)
    # The coordinates of the end tip are the 3 upper entries in the 4th
    column
    pos = np.array([end_tip_m[0,3],end_tip_m[1,3],end_tip_m[2,3]])
    return pos

# function to be optimized
def opt_func(X):
    n_particles = X.shape[0]  # number of particles
    target = np.array([-2,2,3])
    dist = [distance(get_end_tip_position(X[i]), target) for i in
    range(n_particles)]
    return np.array(dist)

# Call an instance of PSO
optimizer = ps.single.GlobalBestPSO(n_particles=swarm_size,
                                    dimensions=dim,
                                    options=options,
                                    bounds=constraints)

# Perform optimization
cost, joint_vars = optimizer.optimize(opt_func, iters=1000)
```

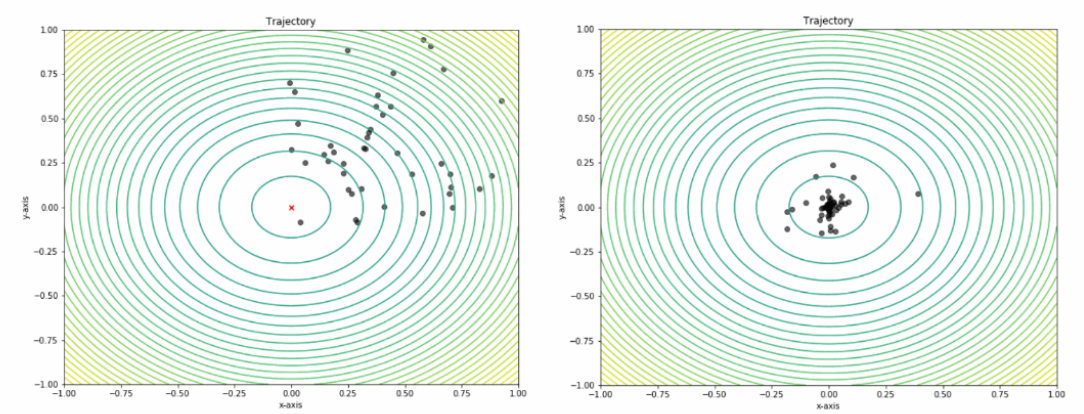


Figure 2.7 PSO gradient descent to IK for robotics arm

2.4 Moveit and Move_group Concept

Here we can see flowchart for any manipulation task utilising the various moveit capabilities.

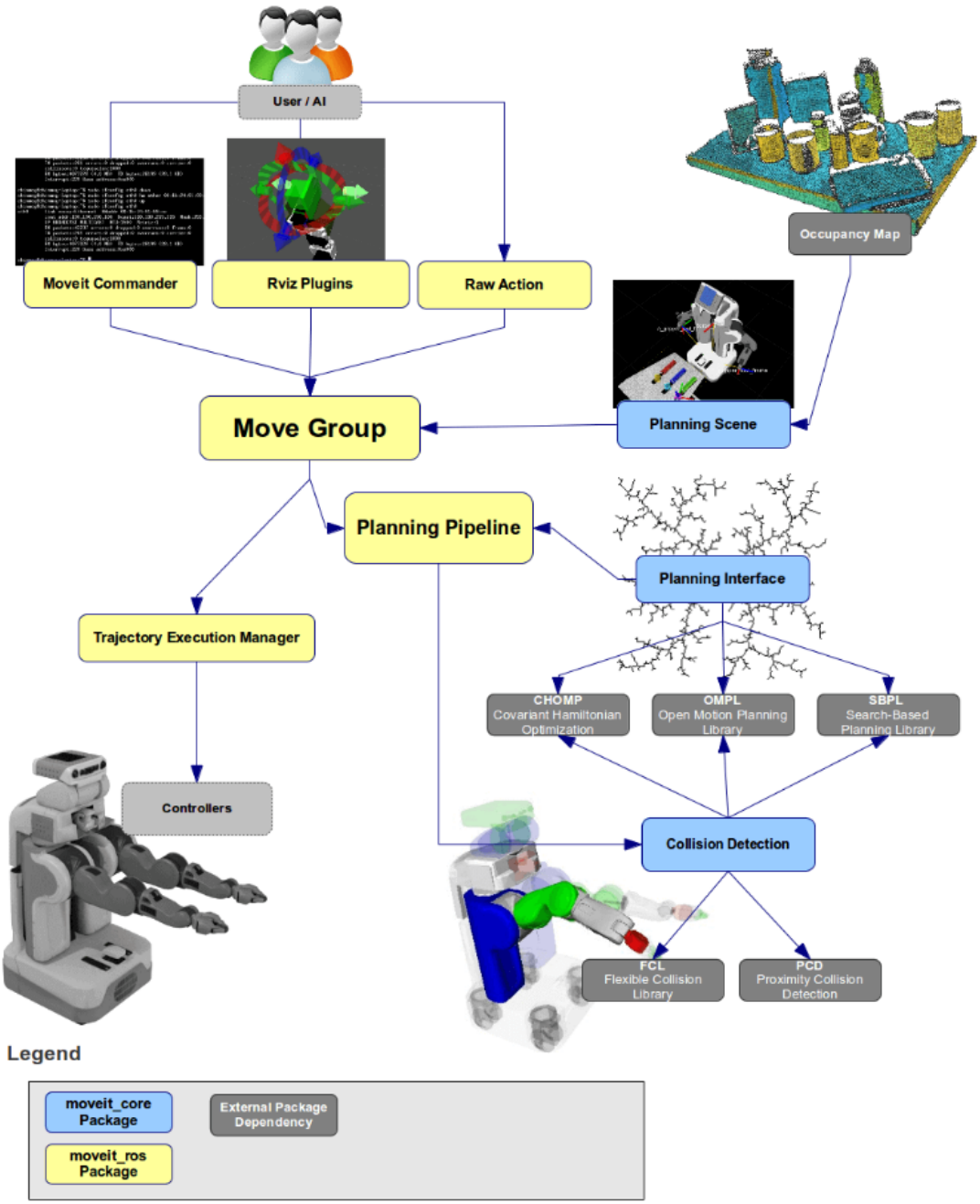


Figure 2.8 Moveit Manipulation Flowchart[1]

Below picture show the various topic that move_group publish and subscribes and its interaction with various other node.

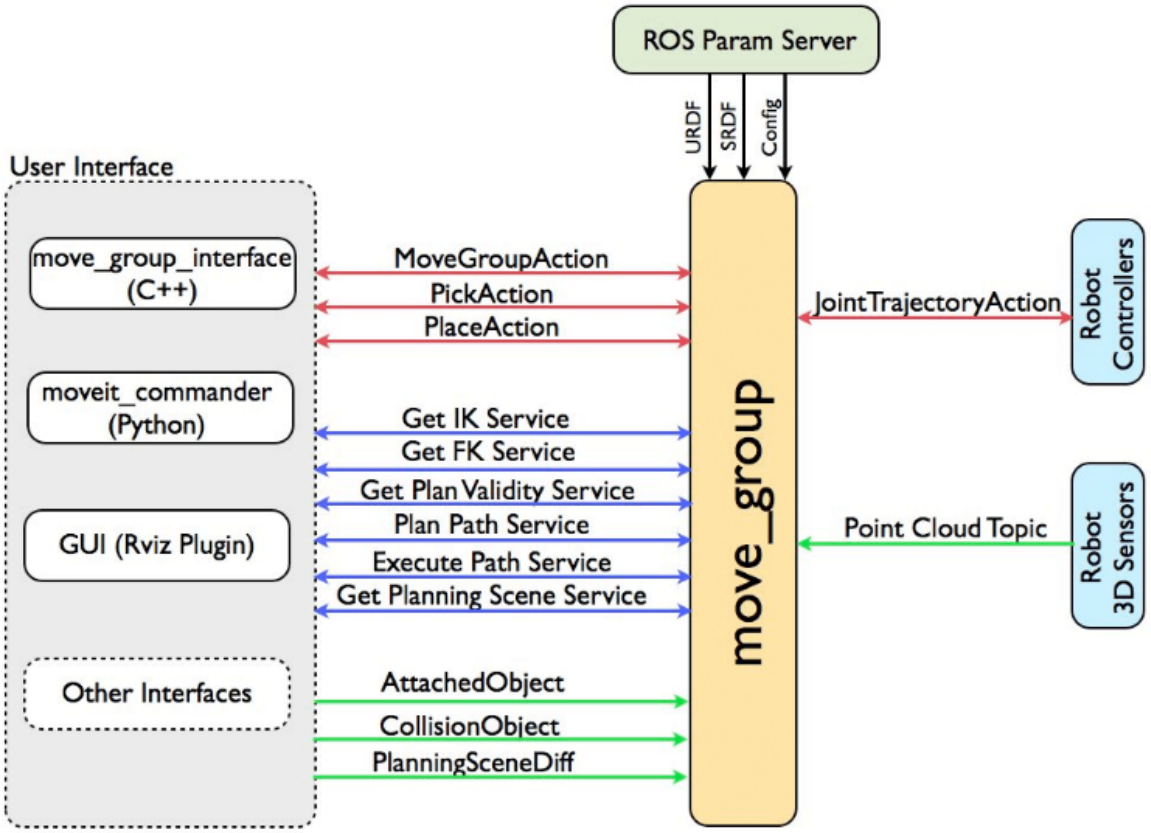


Figure 2.9 Move_group Concept[1]

2.5 Computer Vision for object detection

I had used PCL(Point Cloud Library) to detect and extract Cylinder out of the point cloud in the following steps:

1. Converting pointcloud to pcl:PointXYZRGB
2. PassThroughFilter
3. Compute the point normals
4. Detect and eliminate the plane
5. Extracting plane normals
6. Extract the cylinder

7. Compute cylinder_params

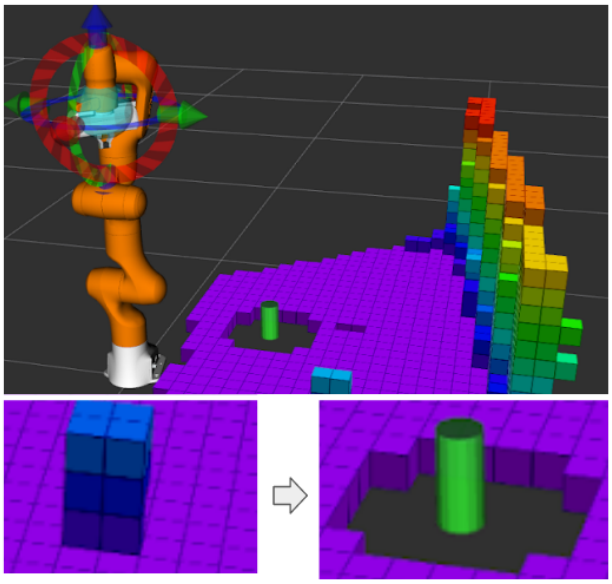


Figure 2.10 Cylinder detection using PCL

2.6 Uarm Shift Pro

I had used UARM with moveit, and since it was 4 dof there were lots of singularity in planning. Also error report was not good enough, which motivated me use MTC in this project.

```
<link name="Link1">
    <inertial>
        <origin xyz="-0.002175 0 0.029097" rpy="0 0 0"/>
        <mass value="0.2141"/>
        <inertia ixx="0.000496945" ixy="-0.000000082" ixz="-0.000002744"
                 iyy="0.000150389" iyz="-0.000000002" izz="0.000522487"/>
    </inertial>
    <visual>
        <geometry>
            <mesh filename="package://uarm/urdf/pro_links/Link1.STL"/>
        </geometry>
        <origin xyz = "0 0 -0.0723" rpy = "0 0 0" />
        <material name = "">
            <color rgba = "0.3 0.3 0.3 1" />
        </material>
    </visual>
</link>

<joint name="Joint1" type="revolute">
    <axis xyz="0 0 1"/>
    <limit effort = "1000.0" lower = "-3" upper = "3" velocity = "0" />
    <parent link="Base"/>
    <child link="Link1"/>
    <origin xyz= "0 0 0.0723" rpy = " 0 0 0" />
</joint>
```

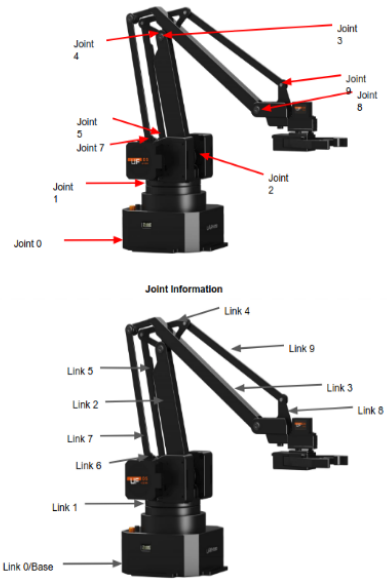


Figure 2.11 Creating the realistic urdf model of UARM

URDF is the universal robot description language, using which we define the relation between the various link in robotic arm and other parameters such as inertia, twist angle and joint angles etc.

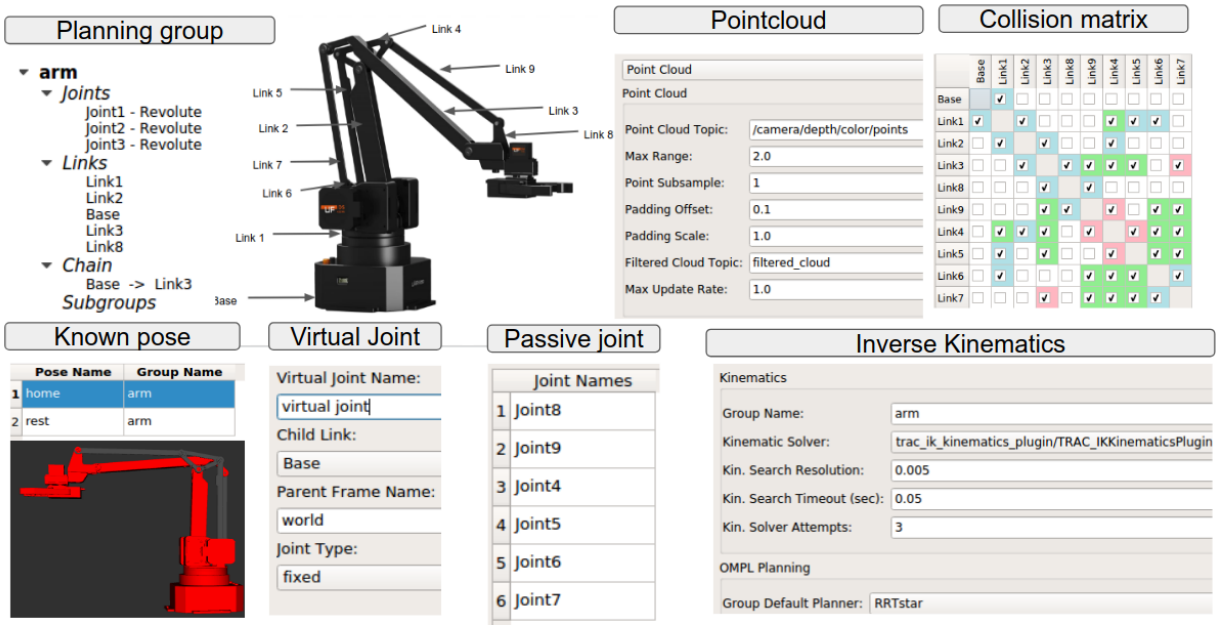


Figure 2.12 UARM Setup Assistance

We use moveit setup assistance to model remaining things collision check etc.

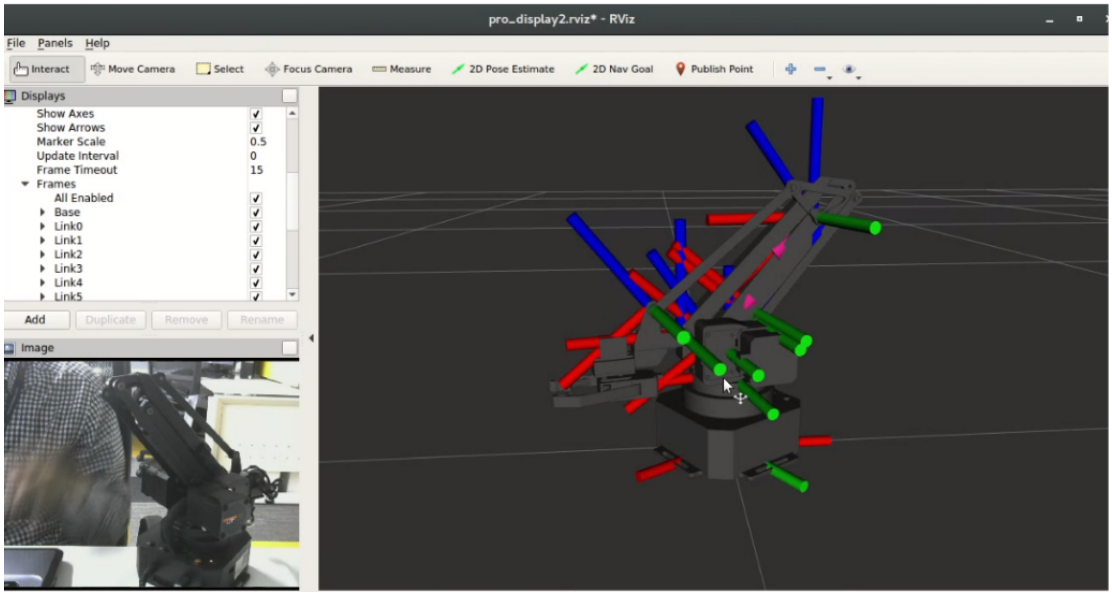


Figure 2.13 UARM JointState, demo available **here**

2.7 Conclusion

We have seen that, for various use cases there are lots of limitation in using the Moveit move_group pipeline for motion planning, thus it motivated me use the MTC framework in the project. Also, We can see that moveit make it impossible to use lower level task planning and also produce bad error reporting. Introspection of stage in task planning is not possible, it became really hard to reason for the failure we get during the planning.

Chapter 3

MoveIt Task Constructor

3.1 Motivation

A lot of motion planning research in robotics focuses on efficient means to find trajectories between individual starting and ending position, but it is challenging to specify and plan robotic manipulation actions which consist of multiple interdependent subtasks. Moveit Task Constructor framework provides a flexible and transparent way to define and plan such actions, enhancing the capabilities of the popular robotic manipulation framework MoveIt!. Subproblems are solved in isolation in black-box planning stages and a common interface is used to pass solution hypotheses between stages. The framework enables the hierarchical organization of basic stages using containers, allowing for sequential as well as parallel compositions. The flexibility of the framework is illustrated in multiple scenarios performed on various robot platforms, including multi-arms.

3.2 Introduction

Motion planning for robot control traditionally considers the problem of finding a feasible trajectory between a start and a goal pose, where both are specified in either joint or Cartesian space. Standard robotic applications, however, are usually composed of multiple,

interdependent sub-stages with varying characteristics and sub-goals. In order to find trajectories that satisfy all constraints, all steps need to be planned in advance to yield feasible, collision-free, and possibly cost-optimized paths.

A typical example are pick-and-place tasks, that require (i) finding a set of feasible grasp and place poses, and (ii) planning a feasible path connecting the initial robot pose to a compatible candidate pose. This in turn involves approaching, lifting, and retracting – performing well-defined Cartesian motions during these critical phases. As there typically exist several grasp and place poses, any combination of them might be valid and should be considered for planning.

Such problems present various challenges: Individual planning stages are often strongly interrelated and cannot be considered independently from each other. For example, turning an object upside-down in a pick-and-place task renders a top grasp infeasible. Whereas some initial joint configuration might be adequate for the first part of a task, it could interfere with a second part due to inconvenient joint limits.

The present work describes the use of MTC framework to describe and plan composite tasks, where the high-level sequence of actions is fixed and known in advance, but the concrete realization needs to be adapted to the environmental context. With this, we aim to fill a gap between high-level symbolic task planning and low-level, manipulation planning, thus contributing to the field of Task and Motion Planning.

Within the framework, tasks are described as hierarchical tree structures providing both sequential and parallel combinations of subtasks. The leaves of a task tree represent primitive stages, which are solved by arbitrary motion planners integrated within MoveIt!, thus providing the full power and flexibility of MoveIt! to model the characteristics of specific subproblems. To account for interdependencies, stages propagate the world state of their sub-solutions within the task tree. Efficient schedulers are proposed to first focus search on critical parts and cheap-to-compute stages of the task and thus retrieve cost-economical solutions as early as possible. Continuing planning can improve the quality of

discovered solutions over time, taking into consideration all generated sub-solutions.

Additionally, the explicit factorization into well-defined stages and world states facilitates error analysis: individual parts of the task can be investigated in isolation and key aspects of individual stages can be visualized easily.

3.3 Conclusion

Here, we have seen how MTC enhance the moveit capability and improve the error reporting, introspection of task etc.

Chapter 4

Task-Level Motion Planning

Here I'll be discussing about how we can think each manipulation task in terms of in sub stages. This stages can be combined using various available containers.

4.1 MTC Task Description

Within this framework, tasks are composed in a hierarchical fashion from primitive planning stages that describe atomic motion planning problems that can be solved by existing motion planning frameworks like OpenRAVE [5], Klamp't [6] or MoveIt! [7]. These frameworks typically allow for motion planning from a single start to a goal configuration, which both are usually fully-specified in configuration space. Often they also permit to specify goal regions, both in configuration and Cartesian space, and appropriate state samplers are employed to yield discrete configurationspace states for planning.

Individual planning stages communicate their results via a common interface using a MoveIt! planning scene to describe the whole state of the environment relevant for motion planning. This comprises the shape and pose of all objects, the collision environment, all robot joint states, and information about objects attached to the robot. This geometric/kinematic state description can be augmented by additional semantic information in terms of typed, named properties, forming the final state representation. Each stage

then attempts to connect states from its start and end interfaces via one or more planned trajectories.

Container stages allow for hierarchical grouping of stages. Depending on the type of the container, solutions found by its children are converted to compound solutions and propagated up the task hierarchy.

4.2 MTC Primitive Stage Types

We distinguish primitive stages based on their interface type.

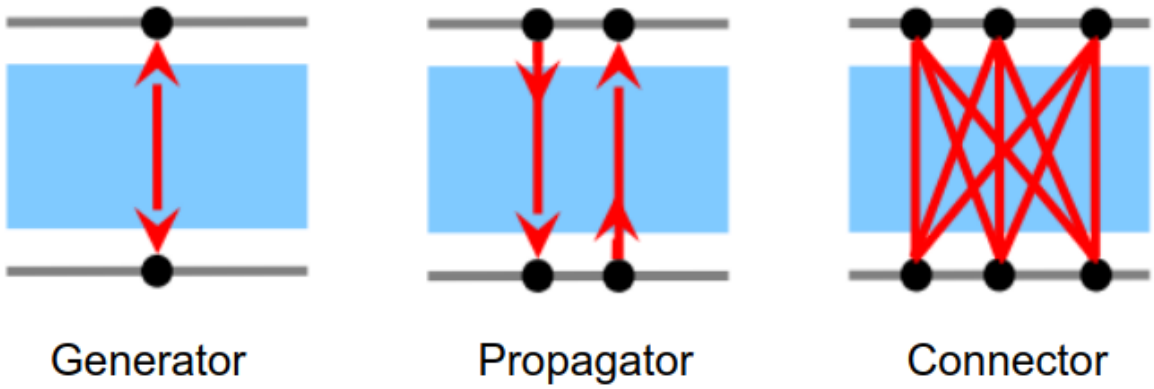


Figure 4.1 Generator, Propagators and Connectors[2]

The classical planning stage is the connecting stage, which takes a pair of start/end states and tries to connect them with a feasible solution trajectory. This type of planning stage often corresponds to transit motions that move the robot between different regions of interest. In this case, any combination of states from the start and end interfaces is considered for planning, realizing an exhaustive search. As such a planning stage will only affect a small set of active joints usually, a pair of start and end states need to match w.r.t. all other aspects of the state representation. Particularly, all other joints as well as the number, pose, and attachment status of collision objects need to match.

The second type, generator stages, populate their start and end interfaces from scratch, without any explicit input from adjacent stages. Usually they define key aspects of an action, for example defining the initial robot state or a fixed goal state, which subsequently can serve as input for adjacent stages. Another example are grasp generators, which provide pairs of pre- and final grasp poses, computing their corresponding robot poses based on inverse kinematics. In this case, generated start and end states usually differ and are connected by a non-trivial joint trajectory (provided by the grasp planner) to accomplish actual grasping.

The most common type of stages are propagators, which read an input state from either its start or end interface, plan to fulfill any predefined goal or action, and finally generate one (or more) new state(s) at the opposite interface together with a solution connecting both states.

Note that propagation can act in both directions, from start to end as well as from end to start. For this reason, it is important to distinguish the temporal from the logical flow. The temporal flow is always from a start to an end interface and defines the temporal evolution of a solution trajectory. However, the logical (program) flow defines the state information flow during planning and is determined by the propagation direction of individual stages. Backwards propagation allows for planning a relative motion to reach a given end state from a yet unknown start state. A common example is the Cartesian approach phase before grasping: Here the final grasp pose is given, and a linear approach motion to the pre-grasp pose needs to be found, whose extent is only coarsely specified within a range of several centimeters. Corresponding solutions are planned in reverse direction, from the end towards the start state. Finally, the solution is reversed to yield a trajectory properly evolving in time from start to end.

4.3 Basic Primitive Stages

The Task Constructor library provides a connecting stage and two basic propagating stages, which all are driven by individual planner instances. While stages specify a subtask, i.e. which robot states to connect, planners perform the actual work to find a feasible trajectory between these two states. Hence, planners can be reused in different stages. Two basic planning instances are provided: (i) MoveIt's planning pipeline offering wrappers for OMPL [8], CHOMP [9], and STOMP [10]; and (ii) a Cartesian path generator based on straight-line Cartesian interpolation and validation.

The two propagating stages allow for (i) absolute and (ii) relative goal pose specification, either in joint or Cartesian space. While in the former case, the goal pose is specified in an absolute fashion w.r.t. a known reference frame, the latter case permits specifying relative motions of a specific endeffector link. In the general case, a twist motion (translation direction and rotation axis) is specified w.r.t. an arbitrary known reference frame and finally applied to the given endeffector link. This makes it possible for example to specify a linear approach or lifting motion relative to the object or a global reference frame.

Generator stages provided are: (i) the current state stage fetching the current planning scene state from MoveIt!'s move_group node, and (ii) the fixed state stage allowing to specify an arbitrary, predefined goal state.[11]

In some cases, the sequential information within the task pipeline is too restrictive to specify a task: Particularly, generator stages might depend on the outcome of another, nonneighboring stage, thus necessitating a short-cut connection within the task pipeline. For example, to place an object after usage at the original pick location, the corresponding place-pose generator needs access to the original pick pose.

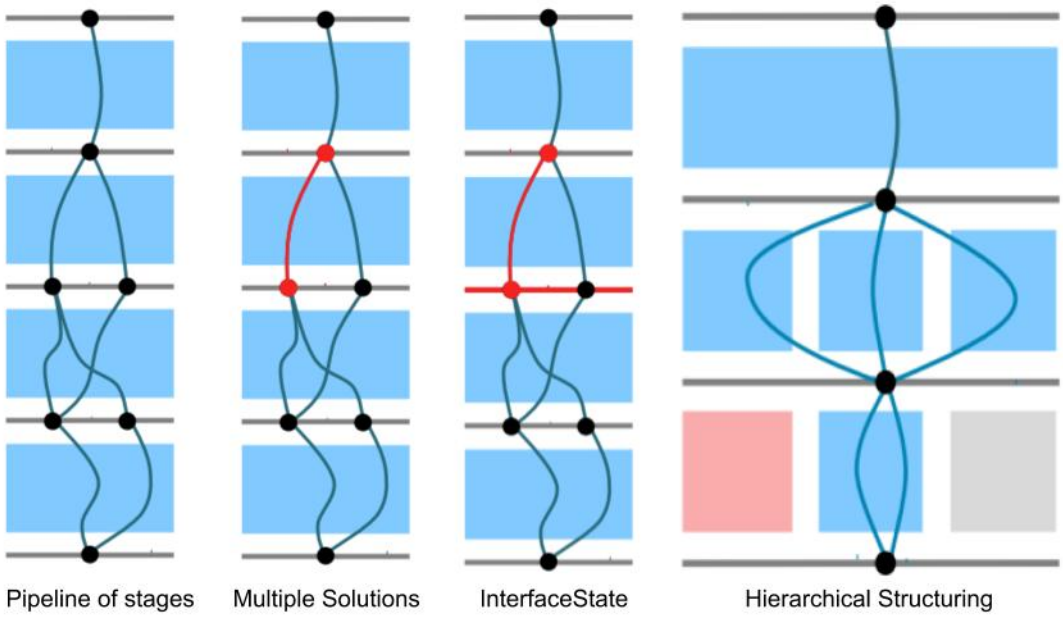


Figure 4.2 Various stages interfaces[2]

4.4 Motion Planning Containers

As mentioned before, container stages are used to hierarchically compose stages into a tree. Each container encapsulates and groups a set of children stages performing some semantically coherent subtask, e.g. grasping or placing. Children stages can easily inherit properties from their parent, thus reducing the configuration overhead. Two main types are distinguished: serial and parallel containers.

4.4.1 Serial containers

Serial containers organize their children into a linear sequence of subtasks which need to be performed in the specified order to accomplish the overall task of the container. Accordingly, a solution of a serial container connects a state from the start interface of the first child stage to the end interface of the last child via a fully-connected, multistage

trajectory path.

In a sequential pipeline, generators play a particularly important role: They generate seed states, which subsequently are extended (in both directions) via propagating stages to form longer partial solution paths. Finally, connecting stages are responsible to link individual partial solution paths to yield a fully-connected solution ranging from the very beginning to the very end of the pipeline.

Obviously, the interface types of stages constrain how they can be sequenced: A stage writing new states along one direction (forward / backward) should be followed / preceded by a stage that reads from the common interface and vice versa. Otherwise, the logical information flow would be broken. Containers provide automatic validation of the connectivity of their children prior to any planning and thus can reject wrongly configured task trees already at configuration time.

Note that in general there can be multiple paths connecting a single pair of start-end states and there can be multiple solutions corresponding to different pairs of start-end states. Hence, it becomes important to rank all found solutions according to a task-specific cost function.

4.4.2 Parallel containers

Parallel containers allow for planning of several alternative solutions, e.g. grasping with left or right arm. Each solution found by its children directly contributes to the common pool of solutions of the container. Different types of parallel containers are distinguished, depending on the planning strategy for children:

(i) *Alternatives*: Consider all children in parallel. All generated solutions become solutions of the container.

(ii) *Fallbacks*: Consider children sequentially, only proceeding to the next child if the previous one has vainly searched its solution space. Only solutions found by the first successful child constitute the solutions of the container.

(iii) *Independent Components*: consider all children in parallel. In contrast to (i), children generate solutions for disjoint sets of robot joints (e.g. arm and hand), which are subsequently merged into a single coherent trajectory performing all sub-solutions in parallel. Obviously, such a merge might fail and explicit constraint checks (including collision checking) are required for final validation. This divide-and-conquer approach is particularly useful, if the planning spaces of individual children are truly independent, as for example in approaching an object for bimanual grasping. In this case, the motion of both arms can be planned independently in lower-dimensional configuration spaces. To enforce independence, one may introduce additional constraints, e.g. a plane separating the Cartesian work spaces of both arms. This task-specific knowledge needs to be provided with the task specification.

4.5 Task Scheduling

The proposed framework exhaustively enumerates all possible solution paths connecting individual interface states, which obviously suffers from combinatorial explosion. Thus, scheduling heuristics are applied to focus the search on promising solution paths first.

To this end, solutions have an associated cost that is computed in a task-specific fashion by user-defined cost functions. Potential functions include, among others, length of trajectory, amount of Cartesian or joint motion, minimum or average clearance. Serial container stages accumulate the costs of all sub-solutions of a full path and only report the minimal-cost path for any pair of start-end states. In a similar fashion, parallel containers only report minimalcost solutions of their children. Each stage, and particularly the root stage of the task tree, can then rank their solutions according to this cost and stop planning when an acceptable overall solution is found.

Each stage ranks all its incoming interface states according to (i) the length and (ii) cost of the associated partial solution. The former criterion biases the search to depth-first (in contrast to breadth-first), which ensures finding full solutions as soon as possible. If a

partial solution fails to extend at either end, this failure is propagated to the other end, and the corresponding interface states are removed from the interfaces of the associated stages as there is no benefit in continuing work on that particular solution.

Additionally, containers handle the scheduling of their children stages. Again the serial container plays the most important role for this. Generators need to be scheduled first in order to generate seed states, which subsequently are extended via propagating stages, and finally connected to full solution paths. Obviously, execution of connecting stages should be postponed as long as possible, because their pairwise combination of start-end states leads to a combinatorial explosion of the search space.

On top of these heuristics, there is room for further optimization. For example, one could try to balance the expected computation time vs. the expected connection success (or reduction in overall trajectory cost) by ranking stages according to the ratio of these values. To yield estimates for them, one could consider heuristic measures (e.g. joint or Cartesian-space distance of states), or maintain statistics over previous stage executions. To yield higher diversity and randomization, actual ranking can be performed based on the Boltzmann distribution of the computed performance rank.

4.6 Task Execution

The main contribution of this work lies in modeling and planning manipulation tasks. Nonetheless, eventually a solution should be executed on the actual robot. Traditionally, planning research simply forwards the final solution trajectory to a low-level controller. To this end, the proposed framework provides utilities to access planned task solutions, such that the user can decide whether to execute, for example, (i) the first valid solution, (ii) the first solution below some cost threshold, or (iii) the best trajectory found within a given amount of time or after exhaustively searching the full solution space. Modifications to the world state performed as part of the task, e.g. attaching or releasing an object, are performed in the same fashion as trajectories are executed, thus ensuring a consistent world

representation throughout task execution.

Given the modularity of the task pipeline, several improvements are possible. Assuming feasible trajectories for the whole task will be found eventually, initial stages (or groups of stages) could commit early to a particular partial solution and forward it for execution before a full solution trajectory is found. As a consequence, this strategy can noticeably reduce the perceived planning time as the robot will start to move early. This is particularly useful when initial stages only yield a single canonical solution, but can also be used to significantly prune the search space, assuming full solutions will be available for most early sub-solutions.

To handle failures during task execution (e.g. due to dynamical changes in the environment, or because an early executed partial solution eventually turns out to be incompatible with later planning stages), a recovery strategy is essential. Again, the modular structure of the task pipeline can be exploited for intelligent recovery, dependent on the failed sub-stage. Potential strategies might replan from the reached stage, or partially revert sub-solutions to continue planning from a well-defined state.

It is also possible to specify different execution controllers (or parameterizations) for individual stages (or groups of stages) to account for different control needs. For example an approach stage might employ visual servoing to account for perception inaccuracies and a grasp stage should use a compliant motion strategy until contact is established and subsequently switch to force-controlled grasp stabilization. As long as the motion of these reactive, sensordriven controllers remains within specified bounds to the planned trajectory, subsequent stages can connect seamlessly.

Finally, solution segments found by individual planning stages can be post-processed to yield a globally smooth solution trajectory. This requires local modifications at the transition between consecutive segments as they might have discontinuous velocity or acceleration profiles. To this end, acceleration-aware trajectory generation [12] can be applied to splice sub-trajectories smoothly within position bounds. The resulting trajectory segments might

only replace the original solutions, if they satisfy collision checks and other constraints.

4.7 Motion Planning Introspection

A key element for the success and acceptance of a software package is its transparency and ease of use. Although MoveIt comes with its own implementation of a manipulation pipeline, its major drawback is its intransparency: the provided pick and place stages are black boxes that do not allow for inspection of their inner workings.

Hence, important elements of the presented software package are pipeline validation, error reporting, and introspection. Stages can publish both successful and failed solution attempts and augment them with arbitrary visual markers or comments, thus providing useful hints for failure analysis. This information, together with the status of the overall planning progress of the pipeline (number of successful and failed solution attempts per stage) is regularly published.

In rviz, the user can monitor the status of the task pipeline and interactively navigate individual solutions of all stages, inspecting their associated markers and comments.

4.8 Conclusion

In this chapter, we have seen how any complex task can be broken down to simple stages. This way we get more control over the planning. These stages can be run serially or paralleling as per the requirements.

Chapter 5

Porting MTC to Moveit2

As we know MoveIt Task Constructor(MTC) framework provides a flexible and transparent way to define and plan actions that consist of multiple interdependent subtasks.
It draws on the planning capabilities of MoveIt to solve individual sub-problems in black-box planning stages. Here we will porting it to MoveIt2 which would also include porting MoveIt 1's MoveGroup interface to MoveIt 2.

Below we will see the summary of the porting process, however port details of the same can be found in this detailed **report**.

5.1 Motivation

I have been working on a multi-manipulator system using MTC for quite some time. I had faced some difficulties in concurrently executing the tasks. I had worked on some solution suggested by Michael Görner. Then I came across a multi-arm demonstration(see **here**) using moveit2 by Acutronic Robotics and I too wanted to do something similar with a panda arm but bit more complex(thus needed MTC). This motivated me to port MTC and the related planning interface to ROS2.

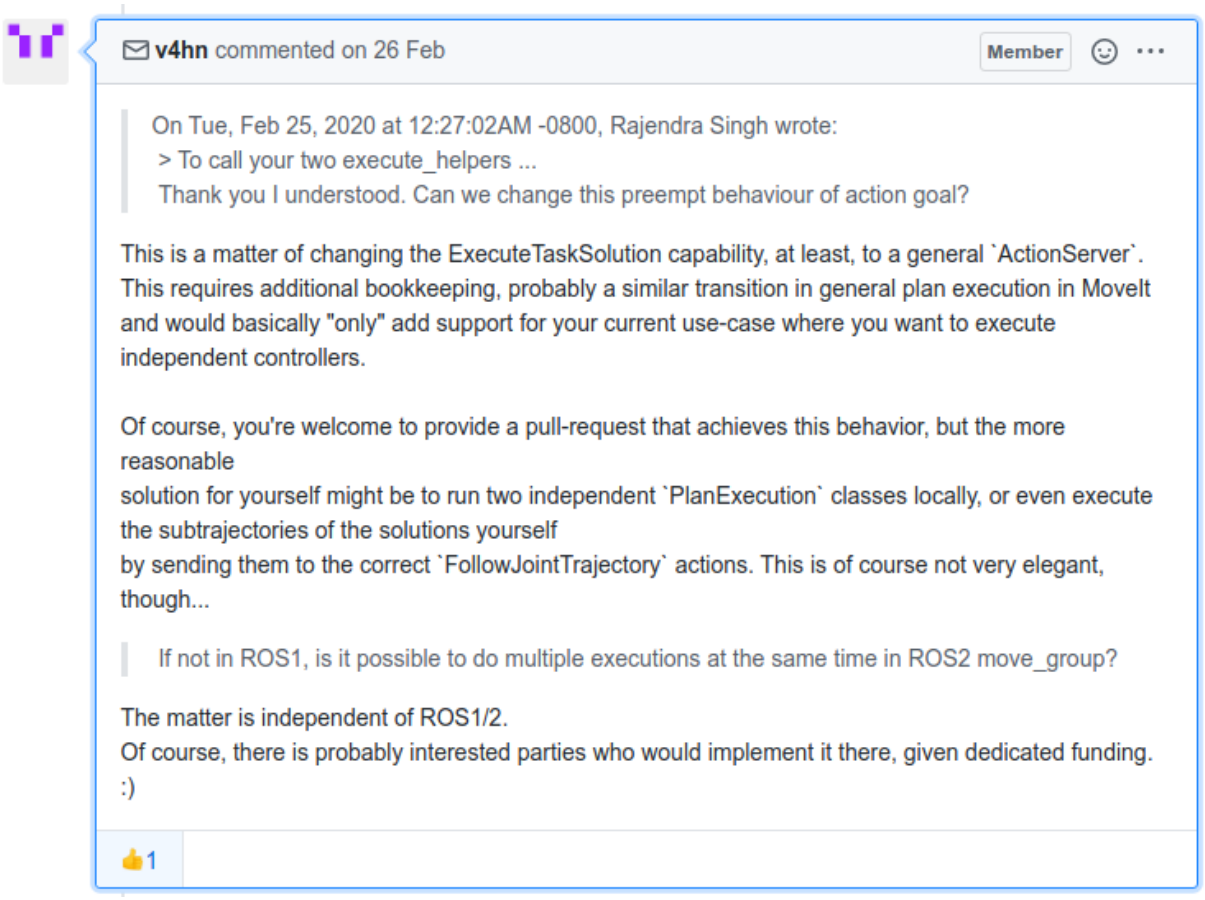


Figure 5.1 Solution by Michael Gorner

5.2 Necessary changes while porting MTC to ROS2

1. **Package manifests**

   (a) Changing package.xml from format 1 of the package specification to newer format wherever applicable.

   (b) Changing the dependencies names if it is named differently in ros2.

2. **Message, service, and action definitions**

   (a) As some primitive types like duration and time which were builtin types in ROS

1 have been replaced with normal message definitions. Thus we should use them from the builtin_interfaces package instead

3. **Build system**

   (a) Modifying the CMakeLists.txt to be used with ament commands instead of a catkin.

   (b) Using rosidl_generate_interfaces to generate msgs, srv and action instead of using generate_messages, add_message_files etc.

   (c) Removing any occurrences of the devel space variables e.g CATKIN_DEVEL_PREFIX

   (d) Adding gtest and linters accordingly (would also need to update package.xml to add corresponding dependencies).

4. **Source code**

   (a) As namespace of ROS 2 messages, services, and actions use a sub-namespace (msg, srv, or action, respectively) after the package name, thus changing the includes and classes accordingly.

   (b) Also changing the included filename from CamelCase to underscore separation and changing include type from *.h to *.hpp

   e.g
   ```
   #include <moveit_msgs/srv/GetPlanningScene.h>
   #include <moveit_msgs/srv/get_planning_scene.hpp>
   ```

   (c) Changing the initialisation of msgs as Shared pointer types are provided as typedefs within the message structs.

5. **Launch files**

   (a) We have to adapt to a lot of changes in the launch file. This would mainly be for MTC/demo package launch files.

5.3 Step-wise porting of MTC to ROS2

Initially, I started with following the full commit history of MTC to build it for ros2 eloquent(see here in the repository) from scratch by making appropriate modifications as and when required. This way I thought I could better understand the reason for a particular design choice of MTC. So far this involved:

- Creating a rough layout of task and subtask class.
- Added first stage i.e CurrentState
- As the gripper required a MoveGroupInterface class, I then ported the move_group package.

ported move_group to moveit2

master

Browse files

iamrajee committed on 18 Mar — 1 parent 047062d — commit fca211349a5ad4f8d737e43cc4c1bea174f68bbe

Showing 73 changed files with 4,335 additions and 597 deletions.

Unified | Split

189 src/moveit2/moveit_ros/move_group/CMakeLists.txt

8 src/moveit2/moveit_ros/move_group/default_capabilities_plugin_description.xml

5 src/moveit2/moveit_ros/move_group/include/moveit/move_group/capability_names.h

20 src/moveit2/moveit_ros/move_group/include/moveit/move_group/move_group_capability.h

```
@@ -34,7 +34,8 @@
34 34
35 35   /* Author: Ioan Sucan */
36 36
37    - #pragma once
      37 + #ifndef MOVEIT_MOVE_GROUP_CAPABILITY_
      38 + #define MOVEIT_MOVE_GROUP_CAPABILITY_
38 39
39 40   #include <moveit/macros/class_forward.h>
40 41   #include <moveit/planning_scene_monitor/planning_scene_monitor.h>
@@ -57,7 +58,7 @@ MOVEIT_CLASS_FORWARD(MoveGroupCapability)
57 58   class MoveGroupCapability
58 59   {
59 60   public:
60    -   MoveGroupCapability(const std::string& capability_name) : node_handle_("~"), capability_name_(capability_name)
      61 +   MoveGroupCapability(const std::string& capability_name) : /*node_("~"),*/ capability_name_(capability_name)
```

- Later ported move_group_inferface package and solved non-matching function errors(#179) by making appropriate modifications in libraries.

Description

I got `No matching function` error for PlanningPipeline() while I was trying to port move_group to eloquent as:

```
--- stderr: moveit_ros_move_group
/home/rajendra/ros2eloquent_moveit_ws/src/moveit2/moveit_ros/move_group/src/move_group_
/home/rajendra/ros2eloquent_moveit_ws/src/moveit2/moveit_ros/move_group/src/move_group_
   planning_pipeline_.reset(new planning_pipeline::PlanningPipeline(planning_scene_monit

In file included from /home/rajendra/ros2eloquent_moveit_ws/src/moveit2/moveit_ros/move_
/home/rajendra/ros2eloquent_moveit_ws/install/moveit_ros_planning/include/moveit/planni
   PlanningPipeline(const robot_model::RobotModelConstPtr& model, const std::shared_ptr
   ^~~~~~~~~~~~~~~~
/home/rajendra/ros2eloquent_moveit_ws/install/moveit_ros_planning/include/moveit/planni
/home/rajendra/ros2eloquent_moveit_ws/install/moveit_ros_planning/include/moveit/planni
   PlanningPipeline(const robot_model::RobotModelConstPtr& model, const std::shared_ptr
```

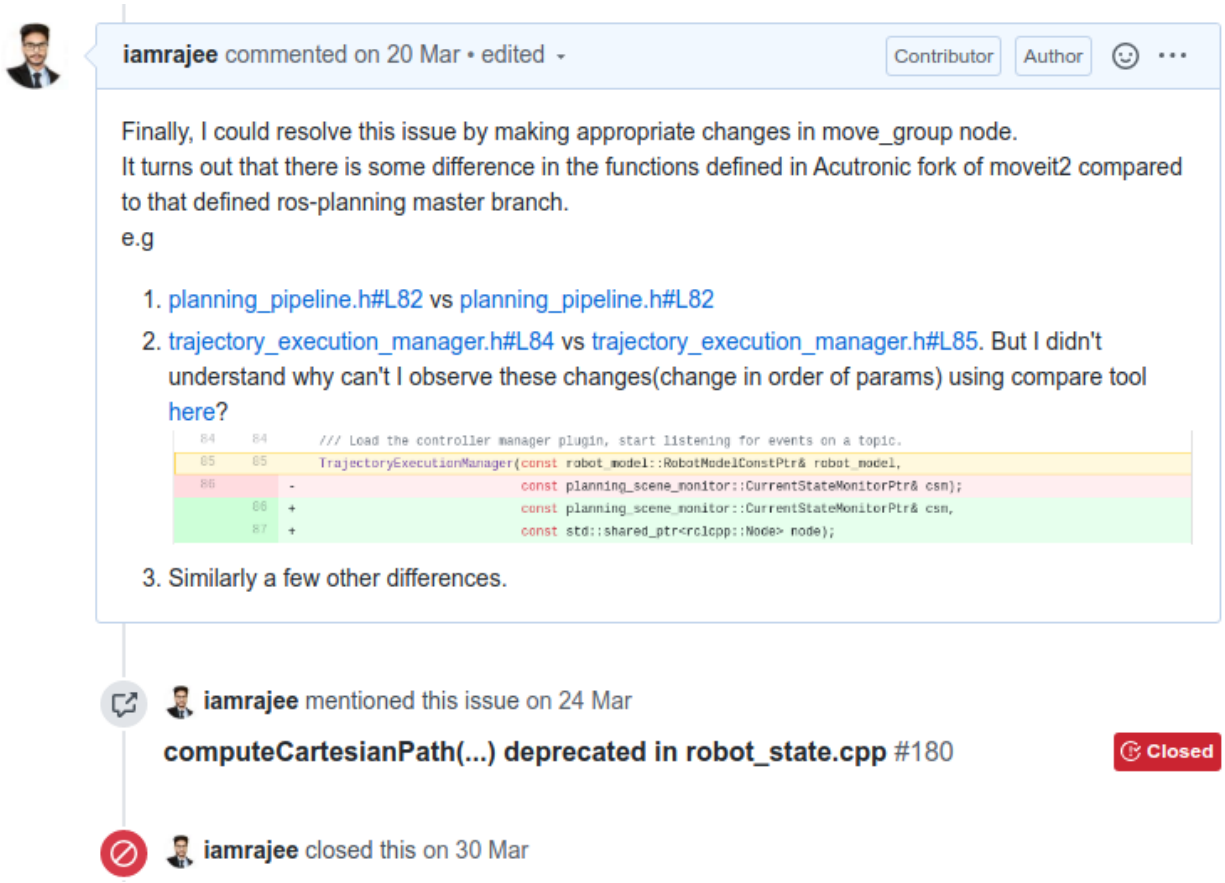
Step to reproduce the error

1. Replace the ros-planning:moveit2:master:**move_group** with AcutronicRobotics:moveit2:master:**move_group**.
2. Build using `colcon build`
3. Encounter `moveit_move_group_capabilities_base` dependency error(see here) so I commented this line to resolve error(can it be the reason for no matching function error?).
4. Then build again.
5. Got `No matching function error` for functions like `PlanningPipeline`, `TrajectoryExecutionManager`, `PlanWithSensing` etc and other type conversion error. See full error **here**.

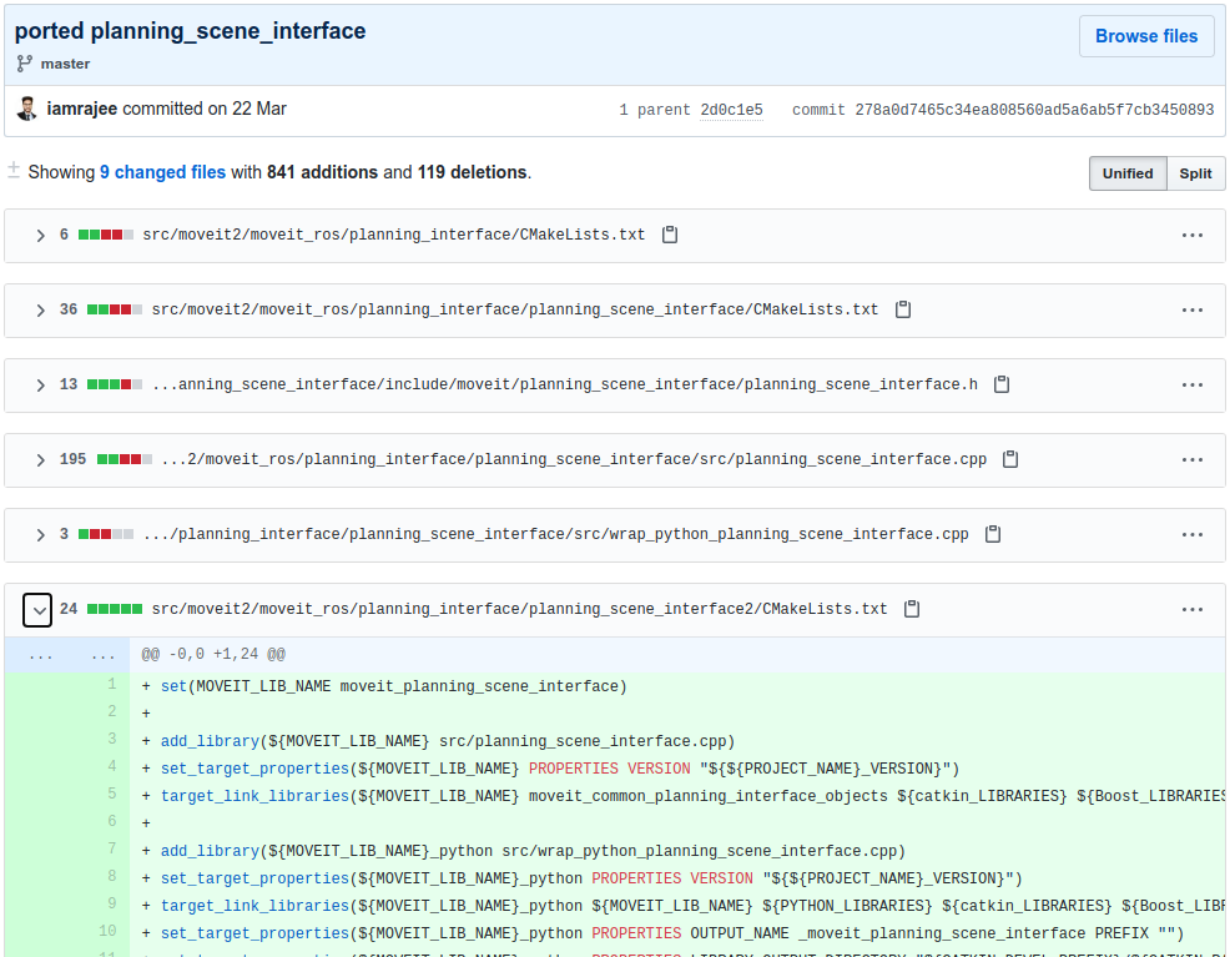
I tried changing the definition of a function with no success. Moreover, I can see that even in moveit1 these functions were defined in the same manner but It doesn't produce error there. I also have ros:melodic:moveit install in a separate workspace along with this ros:eloquent:moveit2 and I can confirm that there is no intermixing in the environment this time as it was there in last time.

Any help would be greatly appreciated.

.



- Added gripper subtask and their relevant function like compute etc.
- Created draft of subtask GenerateGraspPose
- Improved GenerateGraspPose subtask by adding time management, multiple IK solutions, check collisions, angle delta and grasp offset.
- Then ported planning_scene_interface to add objects in the planning scene.



- Creating a test_gen_grasp_pose and tested the working of GenerateGraspPose class.
- Added the first implementation of cartesian_position_motion
- Improved gripper subtask by allowing collisions with grasped objects
- Then tried improving cartesian_position_motion by adding the beginning and end inference but later encountered an issue(#180).

iamrajee commented on 24 Mar • edited ▾ Contributor ☺ ···

Environment

ROS Distro: [eloquent]
OS Version: e.g. Ubuntu 18.04
Binary build
moveit2: cloned from ros-planning:moveit2:master branch at this commit.

Description

I'm porting MTC to ros2 eloquent and currently ported cartesian_position_motion.cpp as this. While building the MTC I got the error stating that the `computeCartesianPath(...)` function is deprecated. I can also see that the same function has been deprecated here at robot_state.h#L1058. Please suggested to me how should I proceed ahead.
Would I be able to solve this if I make appropriate modification in `computeCartesianPath(...)` function similar to what is been done here in Acutronic fork of moveit2?

See my full error report here.

Any help would be greatly appreciated.

.

- See more steps here in this blog.

- [x] Creating a rough layout of task and subtask class.
- [x] Added first subtask i.e subtask::CurrentState
- [x] As the gripper required a MoveGroupInterface class, I then ported the move_group package.
- [x] Later ported move_group_inferface package and solved non-matching function errors(#179) by making appropriate modifications.
- [x] Added gripper subtask and their relevant function like compute etc.
- [x] Created draft of subtask::GenerateGraspPose
- [x] Improved GenerateGraspPose subtask by adding time management, multiple IK solutions, check collisions, angle delta and grasp offset.
- [x] Then ported planning_scene_interface to add objects in the planning scene.
- [x] Creating a test_gen_grasp_pose and tested the working of subtask::GenerateGraspPose class.
- [x] Added first implementation of cartesian_position_motion
- [x] Improved gripper subtask by allowing collisions with grasped objects
- [x] Then tried improving cartesian_position_motion by adding beginning and end inference but later encountered an issue(#180).
- [ ] Now I started porting cartesian_interpolator.h
- split up build into subfolders for tests/demos
  - src/subtask/move.cpp
- Add debug file i.e src/debug.cpp
- Add storage file i.e src/storage.cpp
- Add container file i.e src/container.cpp(only serial at this moment)
- src (Structure so far)
  - task.cpp
  - stage.cpp
  - debug.cpp
  - container.cpp
  - Demo
    - CMakeList.txt
    - plan_pick_ur5.cpp
  - Test
    - CMakeList.txt
    - test_current_state.cpp
  - Subtask
    - CMakeList.txt
    - Current_state.cpp
    - Parallel container
- Add introspection msg & srv
- MTC (Structure so far)
  - msg
    - Stage.msg
    - Task.msg
    - Solution.msg
  - srv
    - GetInterfaceState.srv
    - GetSolution.srv

5.4 Conclusion

As for real time planning and obstacle avoidance we need moveit2, it became extremely important to port MTC port to ros2 so that it can be used with moveit2. But as this is big open-source project and it have lot other dependency which need to be ported first. I'll be continue port MTC as and when possible, so we can do real-time planning in our task.

Chapter 6

Implementations

In the following, we describe few typical manipulation tasks and showcase involved planning stages. For demo we are using various robotics arm such as Kinova jaco j2s6s300, UARM swift pro, panda franka arm etc. Also will demonstrate various task which involve simple pick place, screwdriver handling, single pouring using UR5, creating some structure(here word IIT), multi-arm pouring demo.

6.1 Screwdriver manipulation using a kinova jaco arm

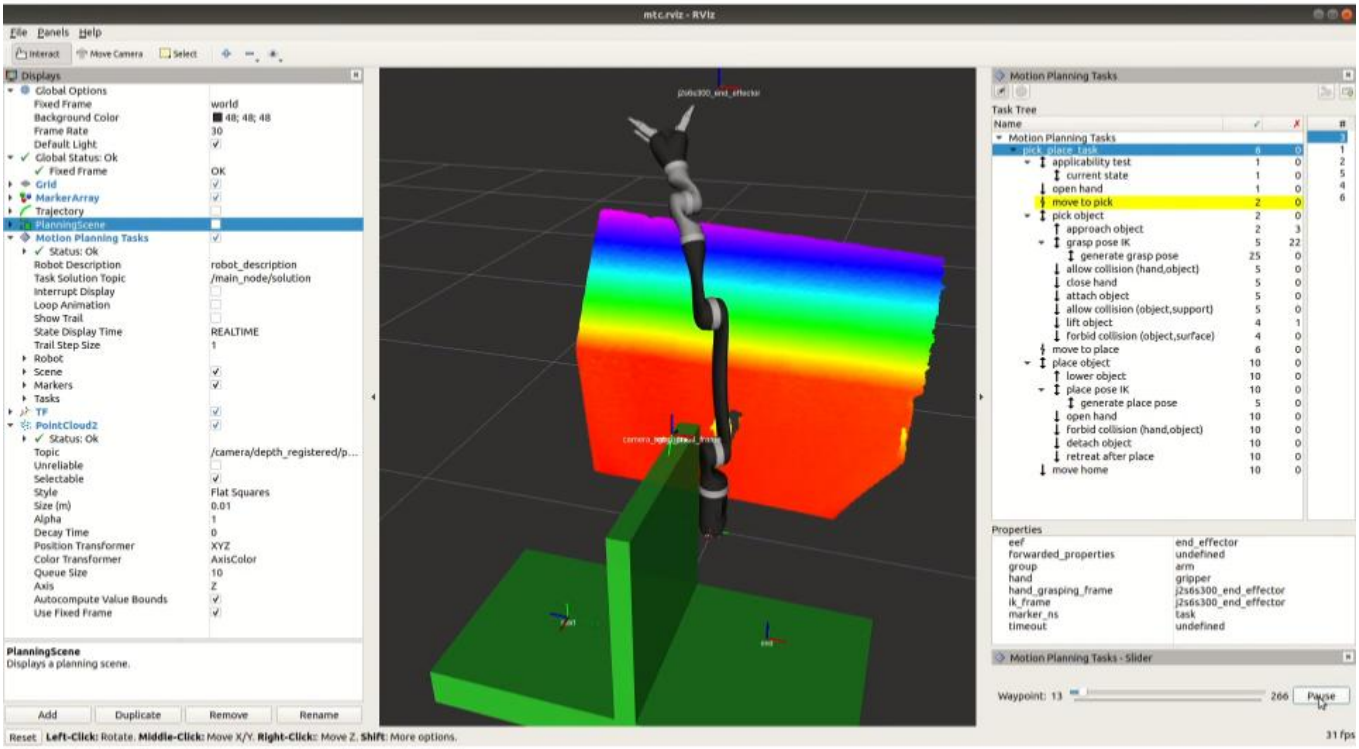


Figure 6.1 Screwdriver manipulation scene, Demo available **here**

Here we are trying to pick the screwdriver in from one position and placing it the another position.

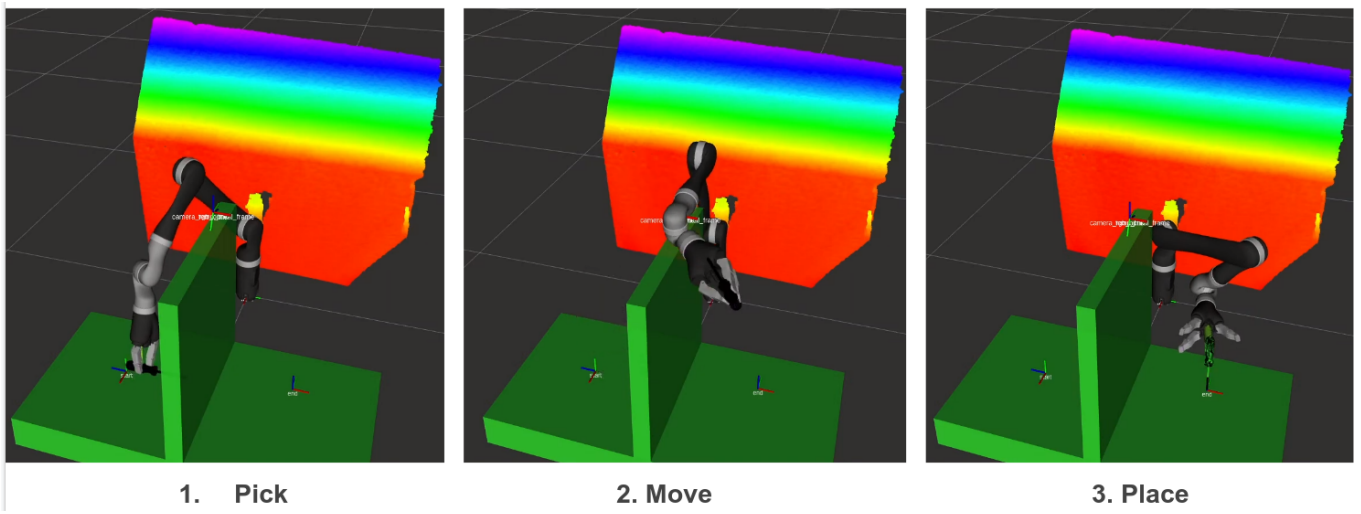


Figure 6.2 Manipulation stages

Note here we have this motion planning such that arm does not collide with the obstacle and pointcloud of real environment.

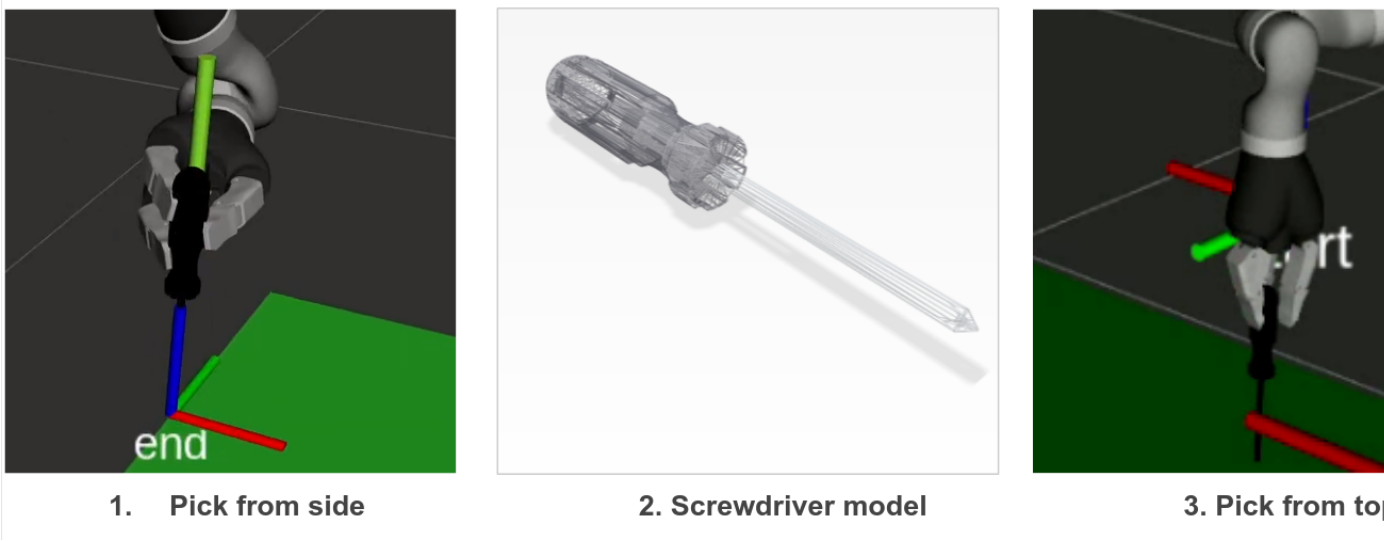


Figure 6.3 Screwdriver pick

Here if you see, we can possibly grasp the screwdriver from both sideways and top, thus we have to explicitly define which one want to do.

6.2 Building a structure using multi arm

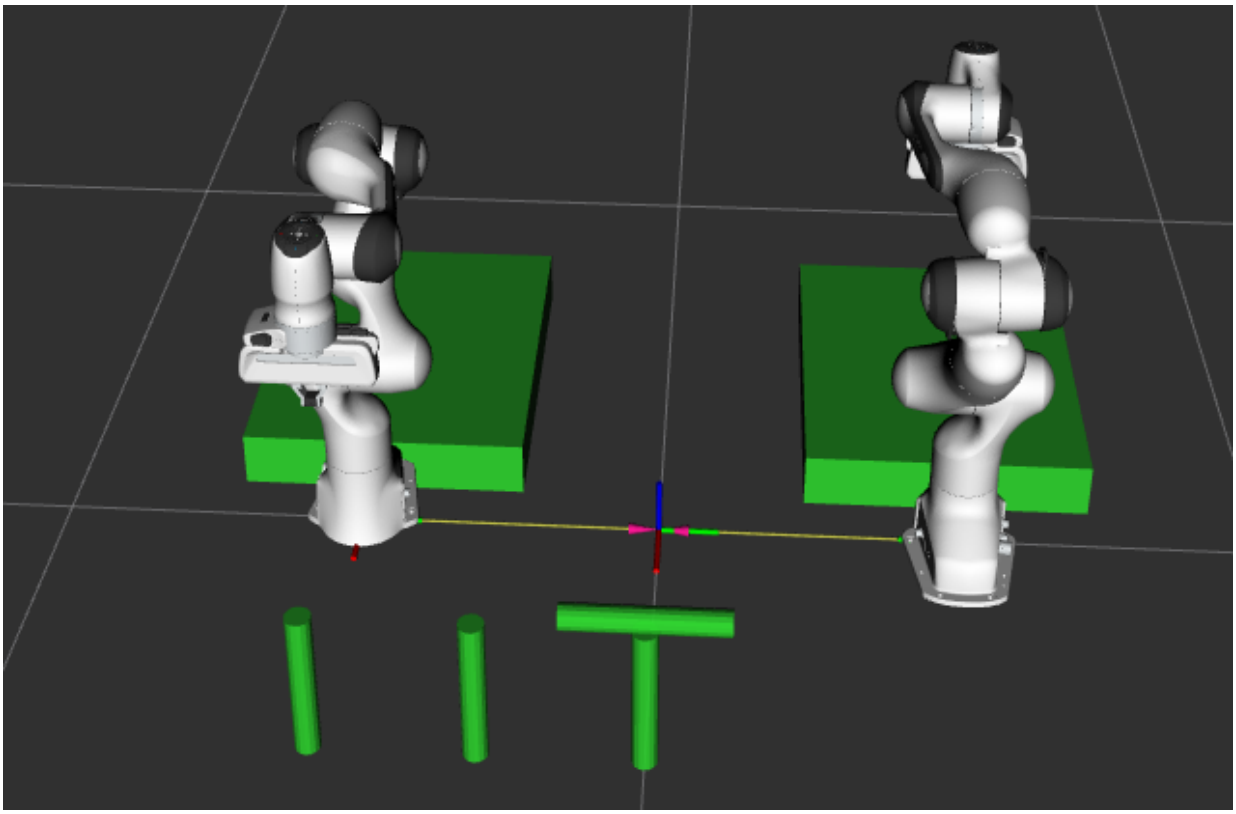


Figure 6.4 PickPlace Scene, Demo available **here**

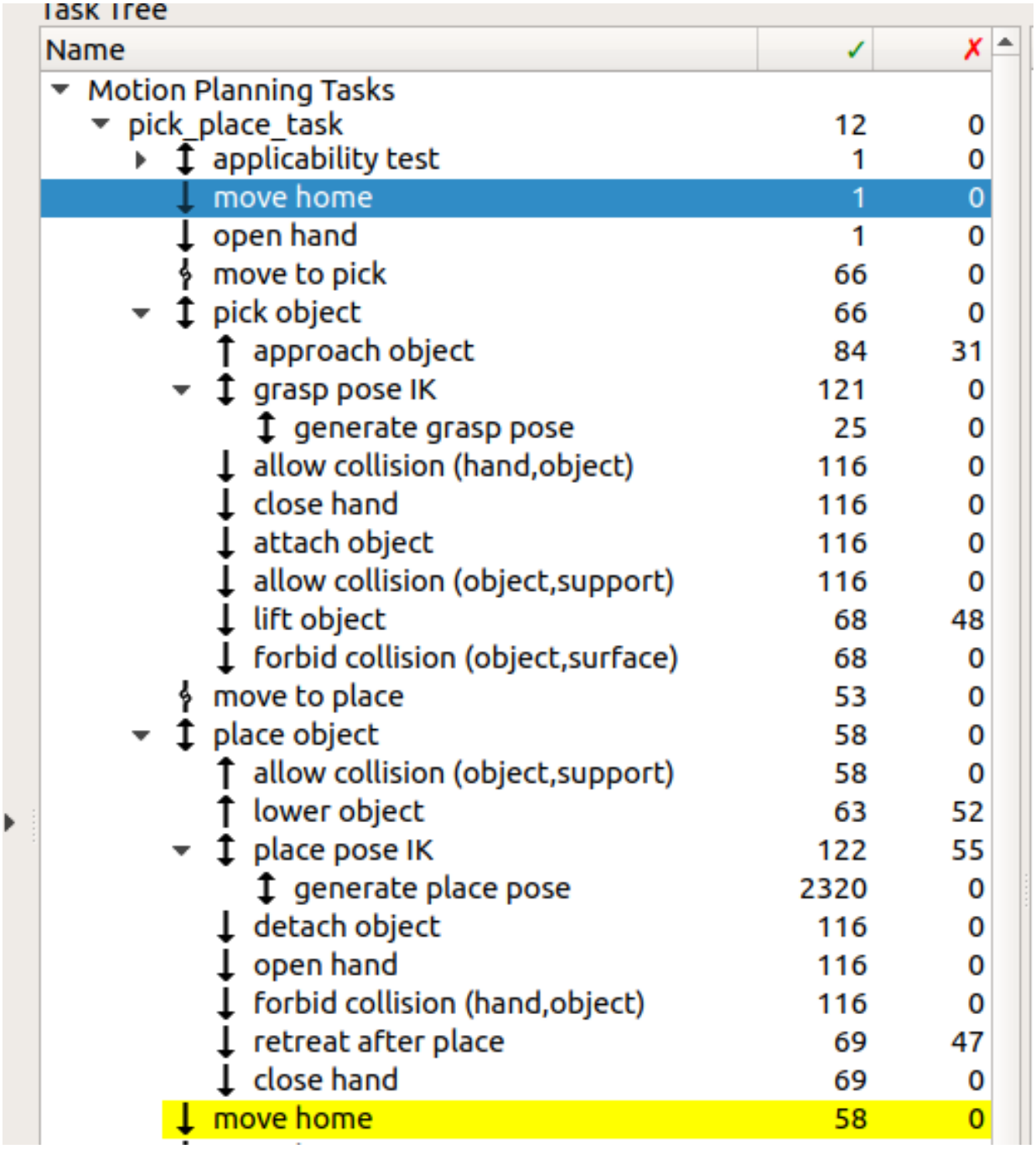


Figure 6.5 PickPlace Stage for Panda arm 1

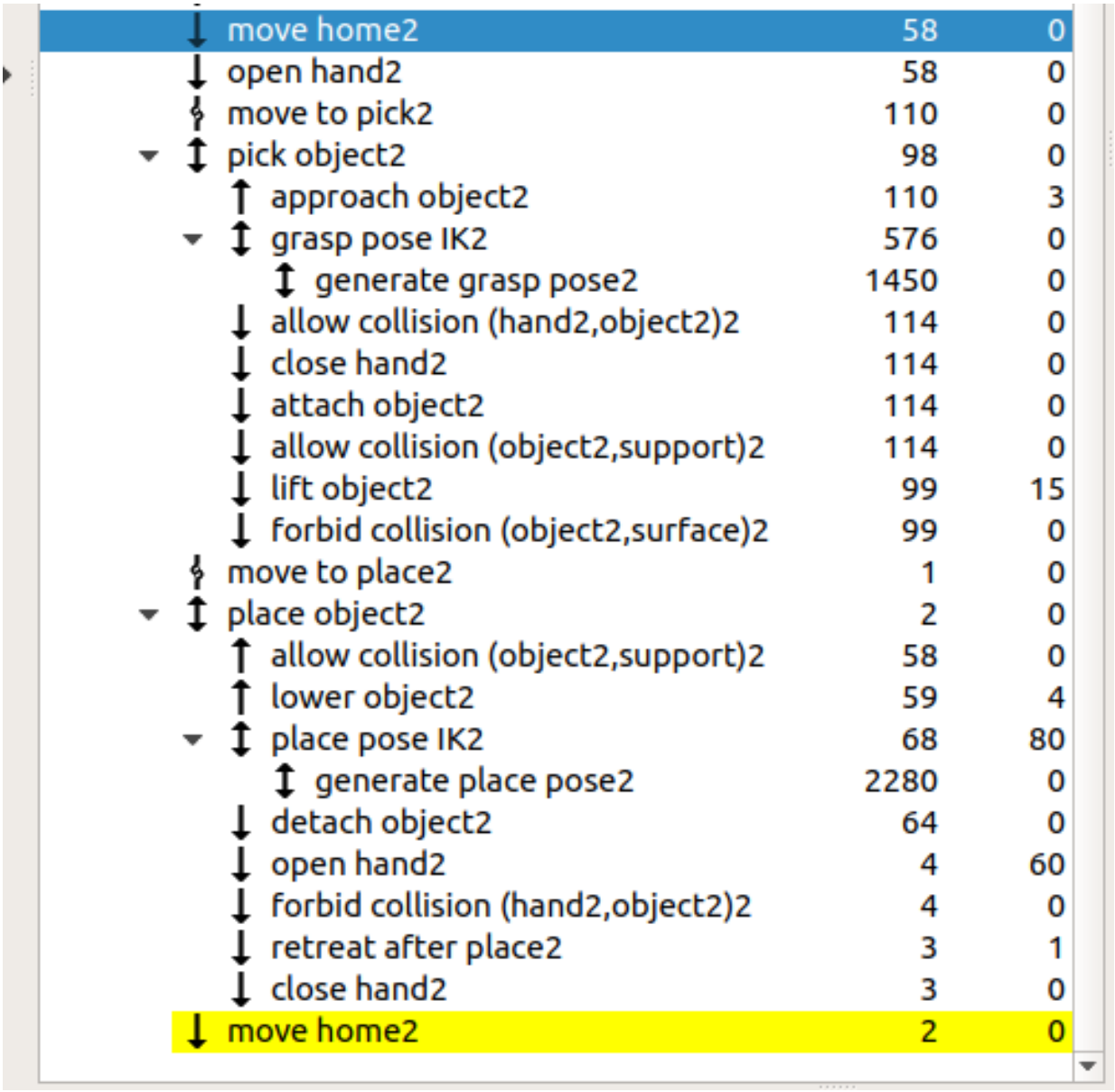


Figure 6.6 PickPlace Stage for Panda arm 2

As picking up an object is a common subtask for many manipulation tasks, a dedicated stage is provided for this. To apply this stage to a specific robot, only some key properties need to be configured, namely the end-effector to use, the name of the object to grasp, and the intended approach and retract directions. The actual grasping is planned by another generic stage, the grasp stage, which is provided as a configurable child stage to the pick

template.

Consequently, the pipeline comprises two alternative pick stages (left and right), configured to use the respective end-effector. The alternatives parallel container follows the current state generator, which fetches the current planning scene state from MoveIt!.

Planning for the pick stages starts with the grasp generator stage and proceeds in both directions: The approach stage realizes a Cartesian, straight-line approach motion, starting from a pre-grasp posture and is planned backwards to find a safe starting pose for grasping. On the opposite side, the lift stage starts from the grasped object state and realizes the Cartesian lifting motion in a forward fashion.

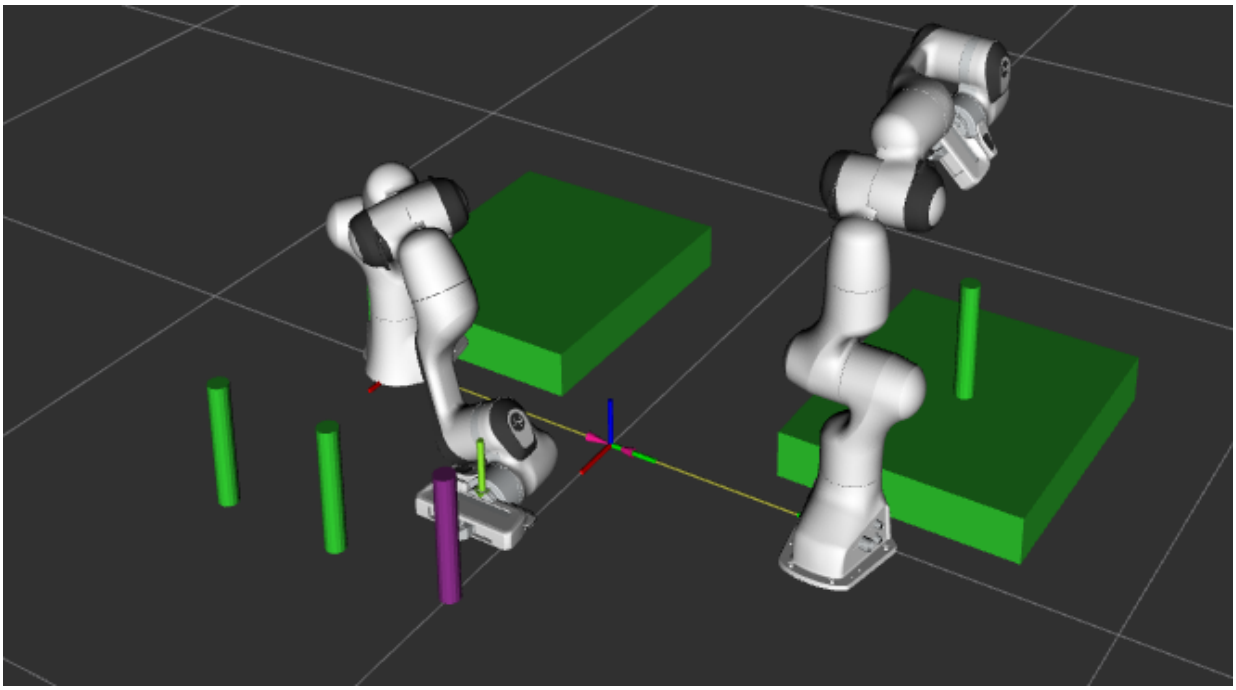


Figure 6.7 Build Structure 1

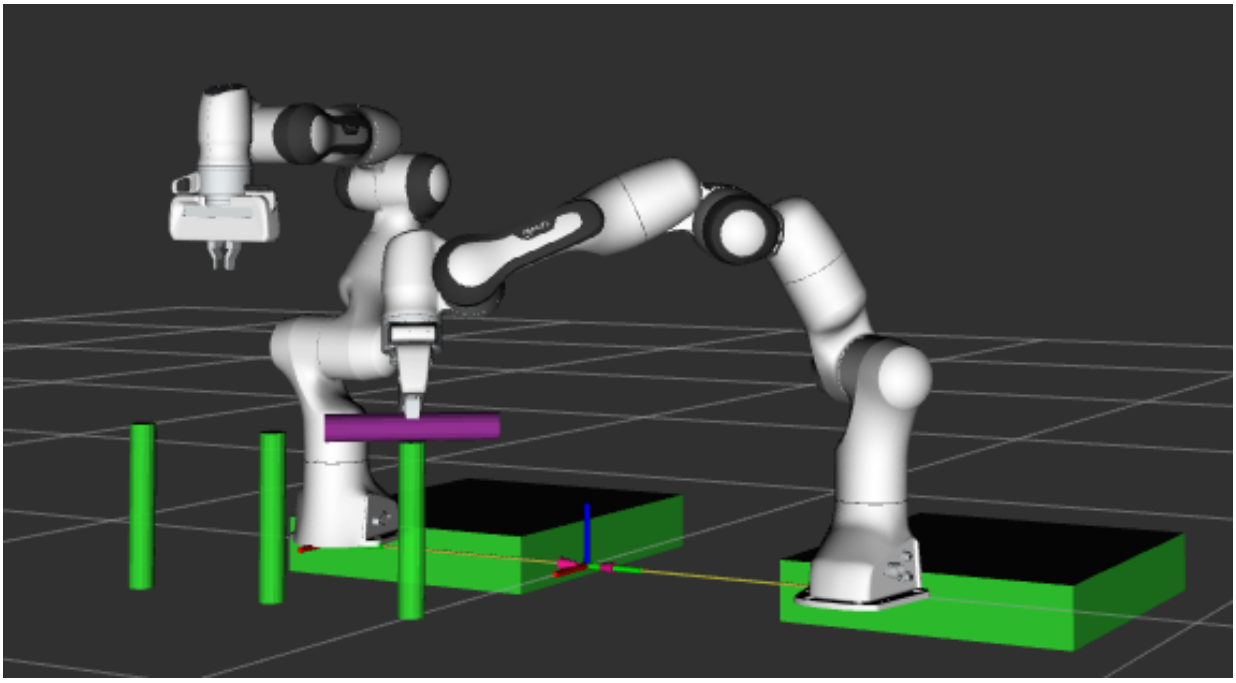


Figure 6.8 Build Structure 2

The grasp stage, in our simple scenarios, samples collision-free pre-grasp poses relative to the object at hand, computes the inverse kinematics to yield a joint-state pose suitable for use in the interface state, and finally performs grasping by closing the gripper. To this end, first collision detection between end-effector and object is turned off to allow the end-effector to contact or penetrate the object when actuating the grasp pose. For real-world execution, the close gripper stage obviously requires a force-controlled or compliant controller to avoid squashing the object. Finally, the object is attached to the end-effector link, such that further planning knows about the grasped-object state. These helper subtasks, which only modify the planning scene state, but do not actually perform any planning, are realized by utility stages, which permit to change allowed collisions as well as to attach and detach collision objects.

Sampling of pre-grasp poses, in our examples, considers a pre-defined open-gripper posture for the end-effector and proposes Cartesian poses of the end-effector relative to

object-specific grasp frames. We sample grasp frames by rotating the object frame about its z-axis in steps of pi/12 radian, resulting in 25(12*2+1 = 25) grasp frame samples. The end-effector is placed relative to these grasp frames by applying the inverse of a fixed tool-to-grasp transform. The resulting transform is used as the target for inverse kinematics. Before applying inverse kinematics sampling, the IK stage validates the feasibility of the targeted pose, i.e. whether placing the end-effector at the target is collisionfree. If not, IK sampling can be skipped and failure is reported immediately. While the first solution on all studied robots is found within a fraction of a second, the planning time for exhaustive search clearly varies between all studied robots and is dominated by the number of sampling-based planning attempts (in stage move to object), which in turn is determined by the number of solutions found by the grasp stage. By repeating this pick and place pipeline for second panda arm, we have created a structure which look similar to word **IIT**. With this we found 12 solution for task execution, and best solution take 78.8 seconds to execute the full task.

1	76.7909
7	77.1628
2	78.6891
8	79.061
3	80.079
4	80.4457
9	80.4509
5	80.5633
6	80.7123
10	80.8176
11	80.9352
12	81.0842

Figure 6.9 Time taken by various solutions

6.3 Single arm pouring using UR5

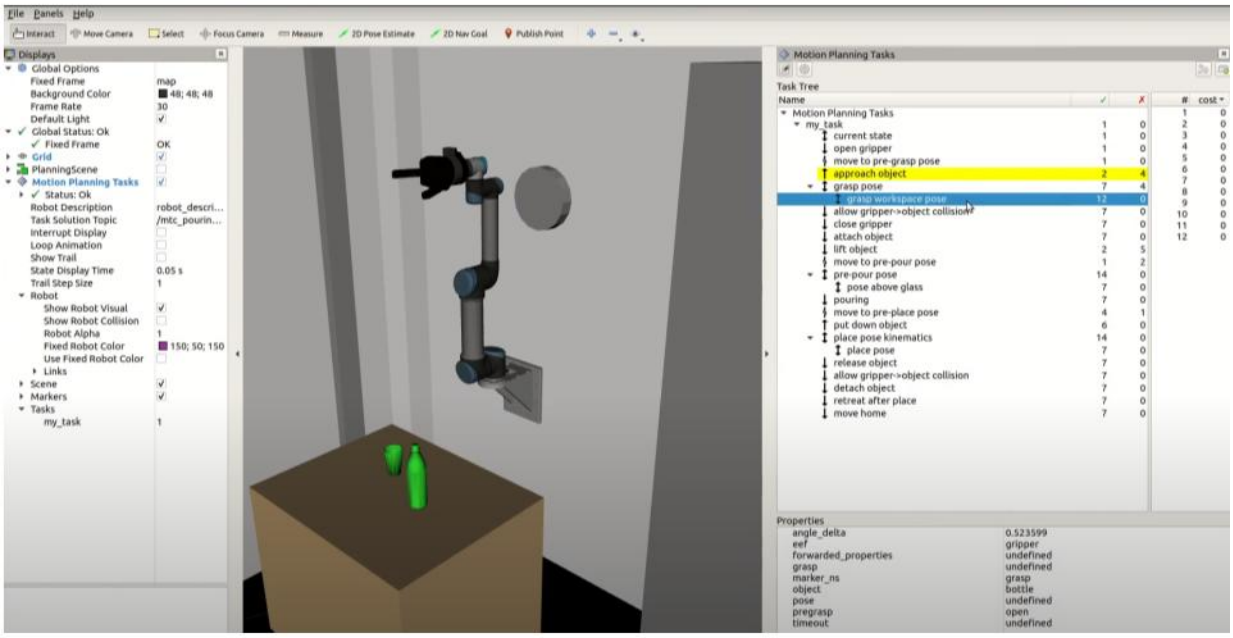


Figure 6.10 Scene consist of UR5 arm attached to wall, glass and bottle are placed on table. Demo available **here**

In above scene, we see UR5 arm will try to pick the bottle from table and pours water in the glass and keep it at some another position.

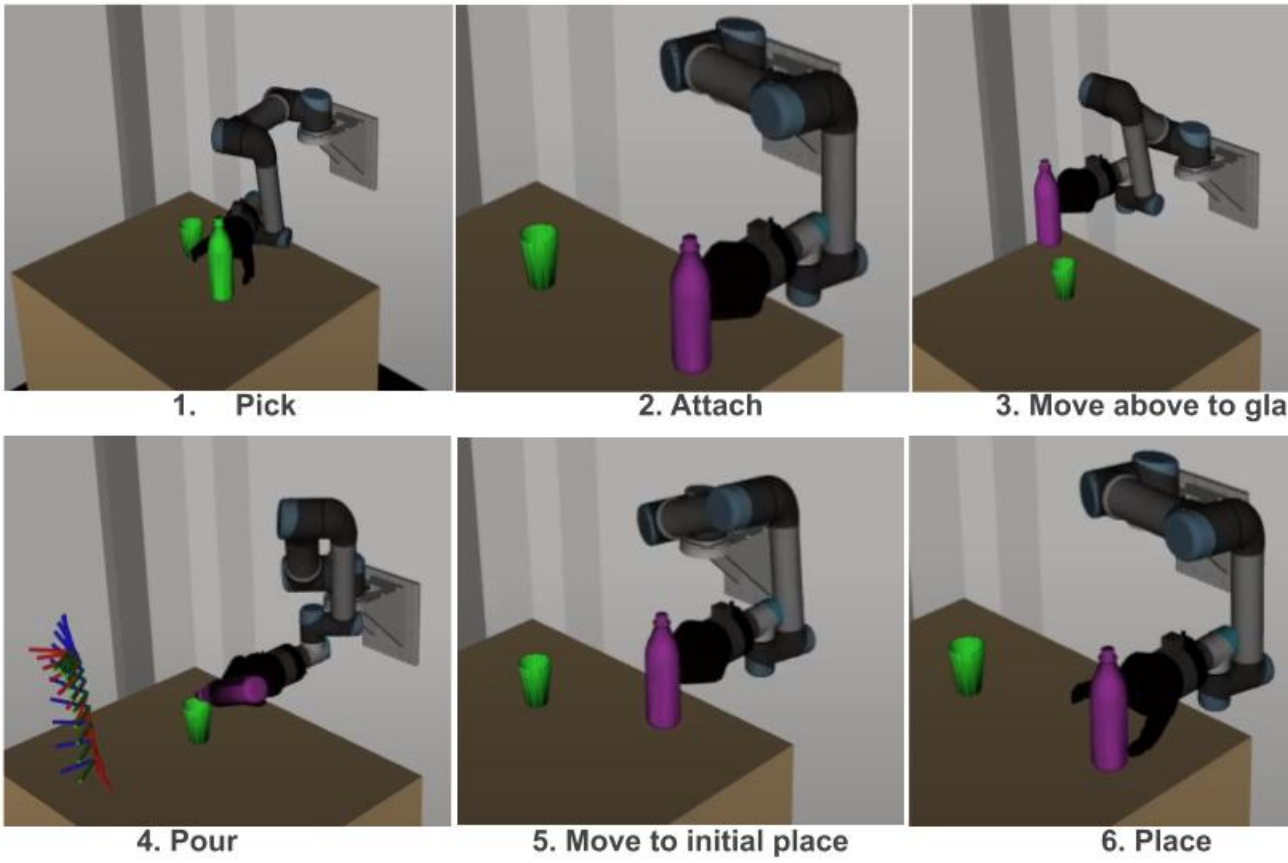
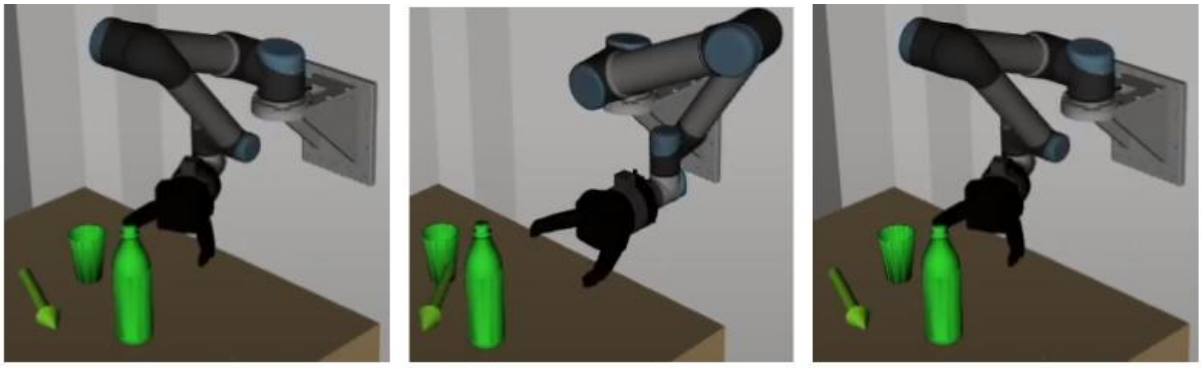


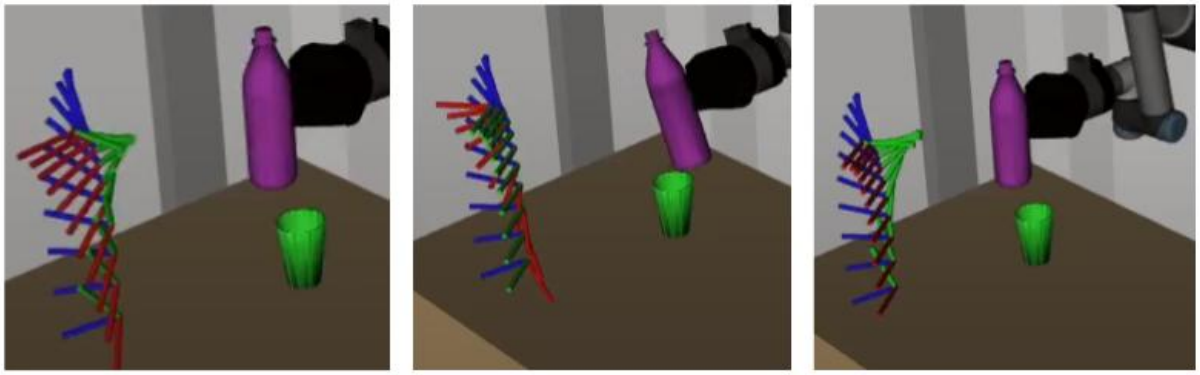
Figure 6.11 Various stages involved in manipulation task

Above figure show the various stages involved in manipulation task. Namely,

1. Pick
2. Attach botthe
3. Move above the glass
4. Pour the liquid
5. Move to the initial place
6. Place the bottle



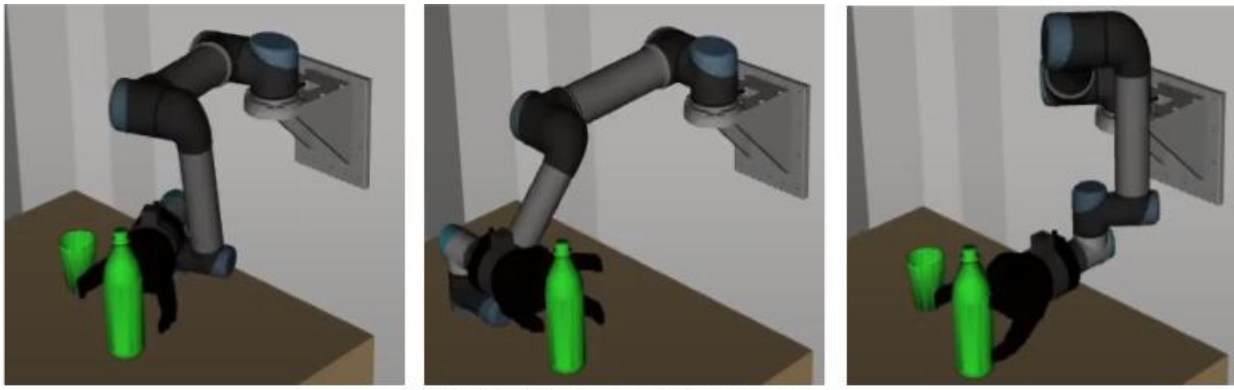
Various approach poses



Various pouring trajectories

Here we see that even for simply approaching to the bottle to grasp it, there are several ways(only three of them shown here), Thus each stage can plan independently.

Also there are several ways to pour the liquid in the glass, i.e there are several trajectories for the mouth of the bottle while pour and the best is the one which takes minimum time and also don't spill the liquid else where.



Various Successful grasp poses

Various Unsuccessful grasp poses

Similarly, grasping of the bottle can be done in several ways. Some of them would be successful, while other won't. This way MTC help us in understanding that what move is feasible and what not.

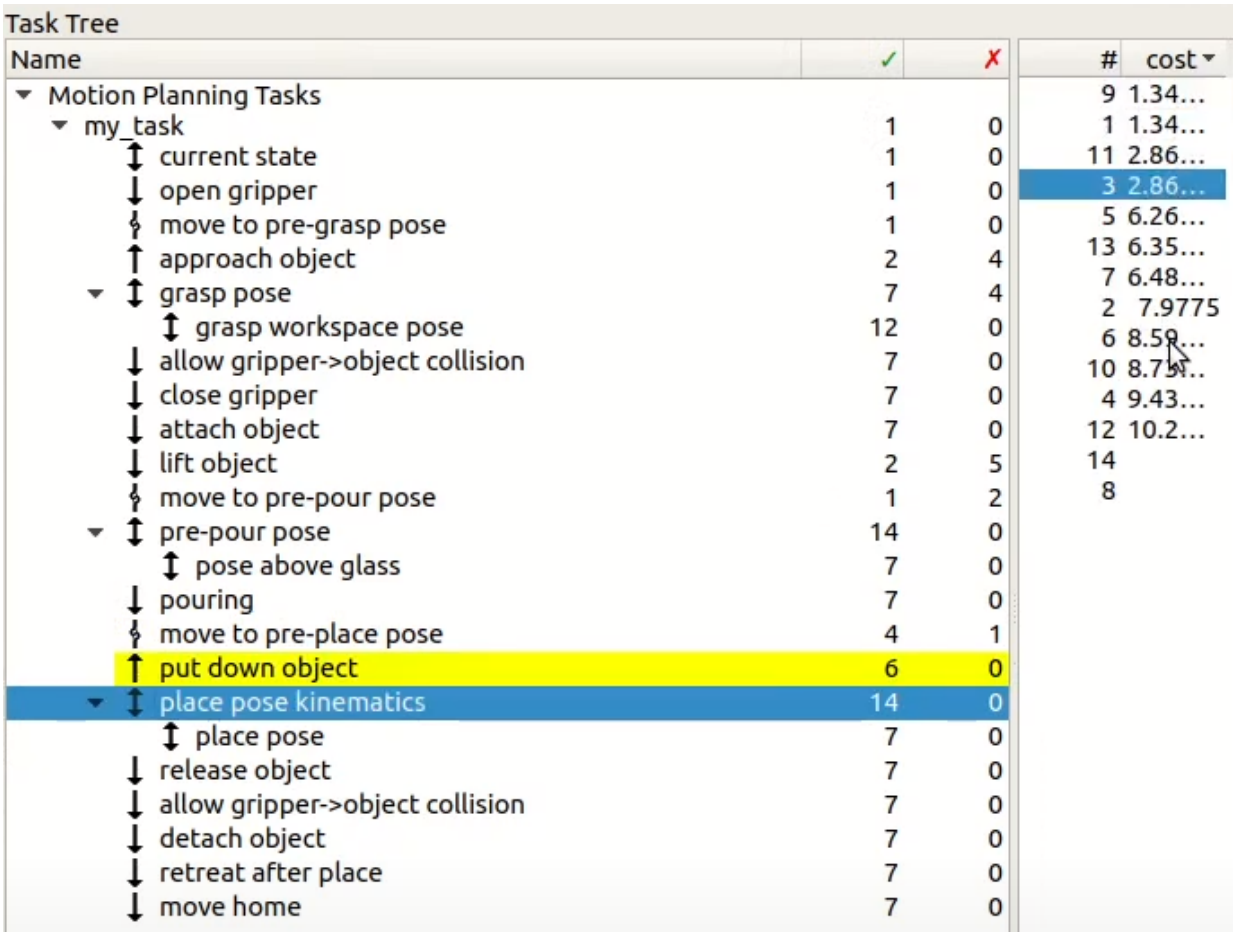


Figure 6.12 Various stage with there times and no. of successful and unsuccessful solution

Here, we can see that each stage have some successful and some unsuccessful solutions. Also, it is possible to rank those solution based on the time it took to execute it. This way we end up getting the best possible for our given task, and ofcourse the best if the one which take the minimum time to complete.

6.4 Multi-arm pouring using panda

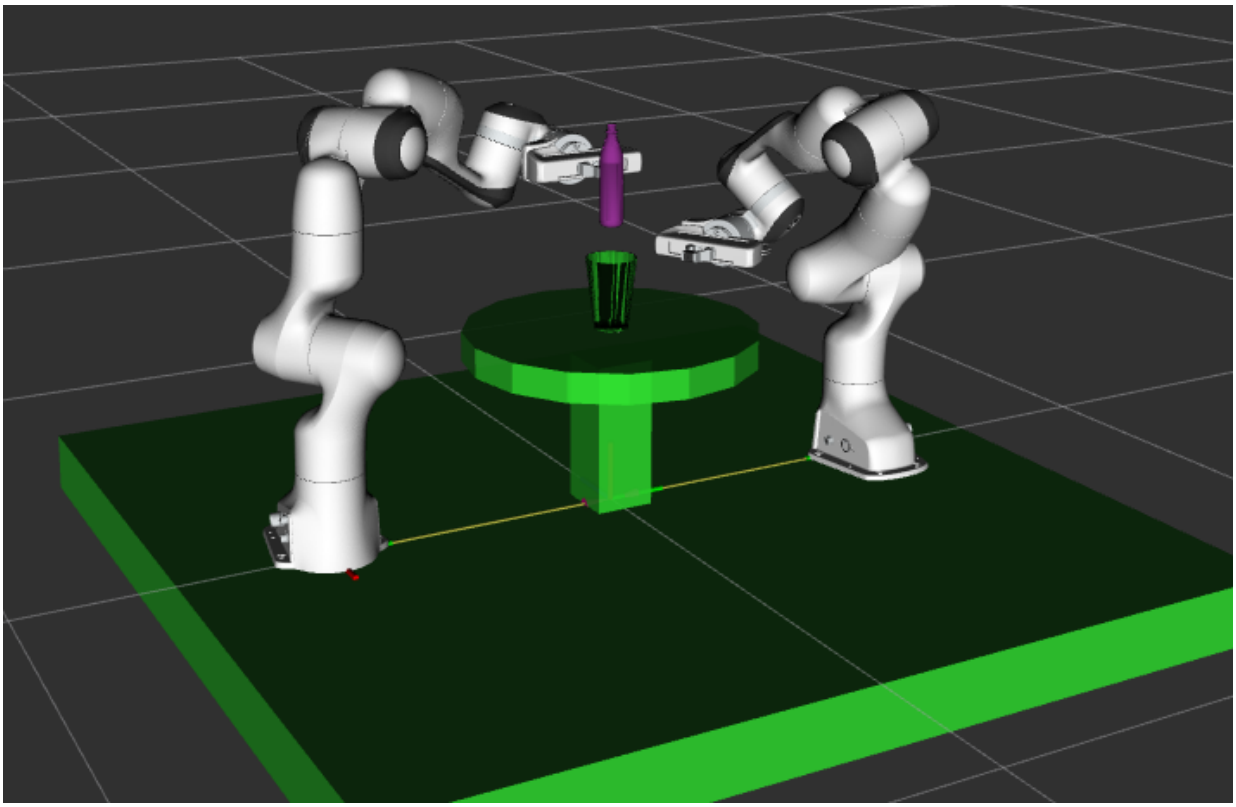


Figure 6.13 Pouring Scene, Demo available **here**

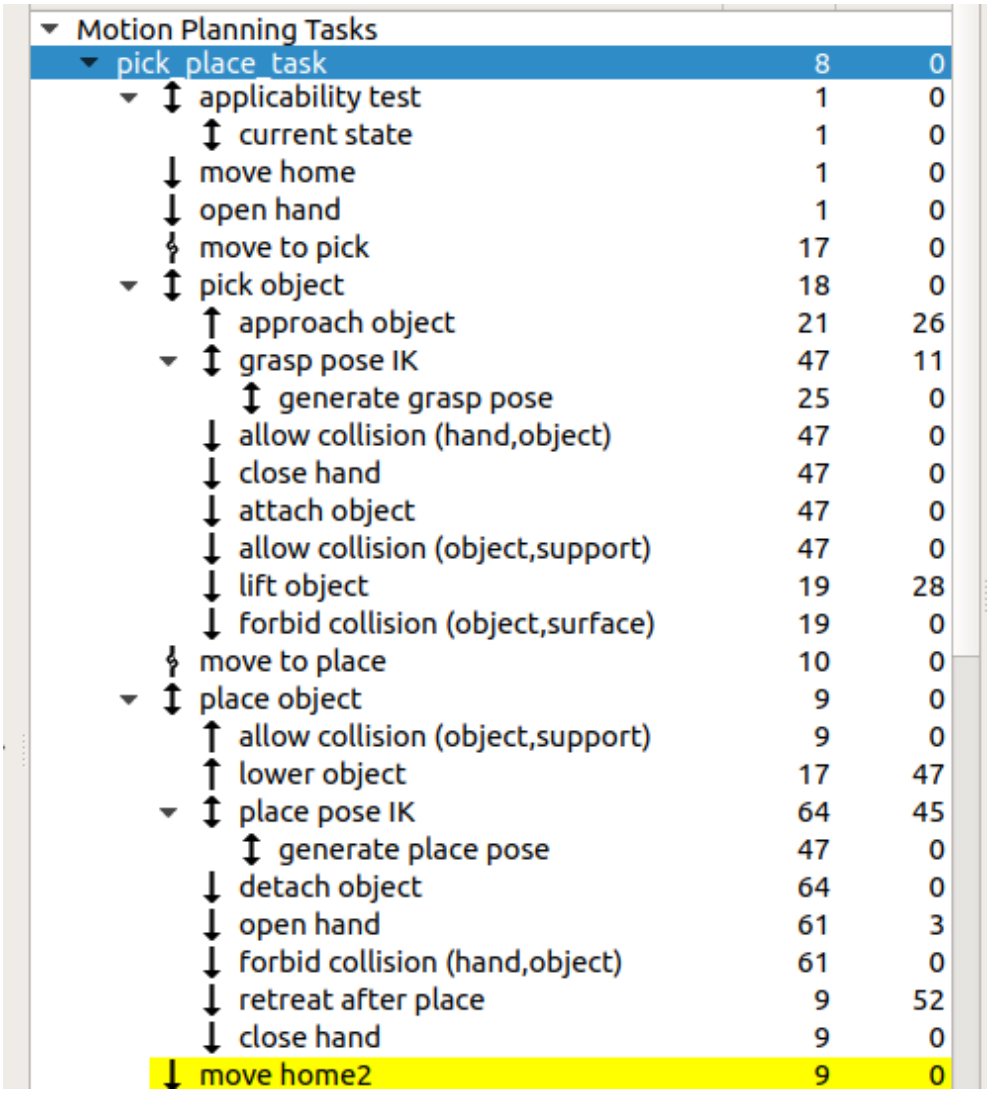


Figure 6.14 Pouring Stage for Panda arm 1

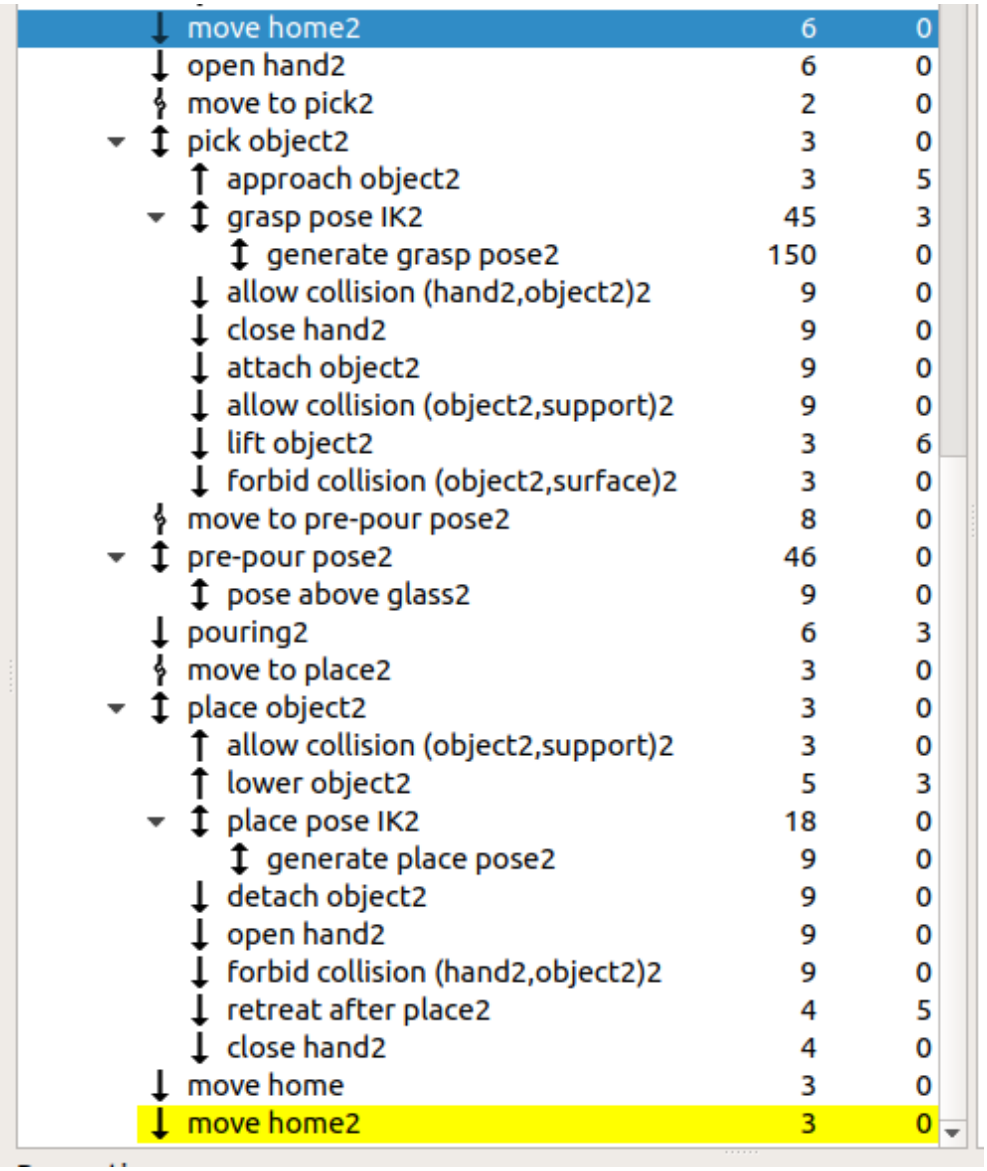


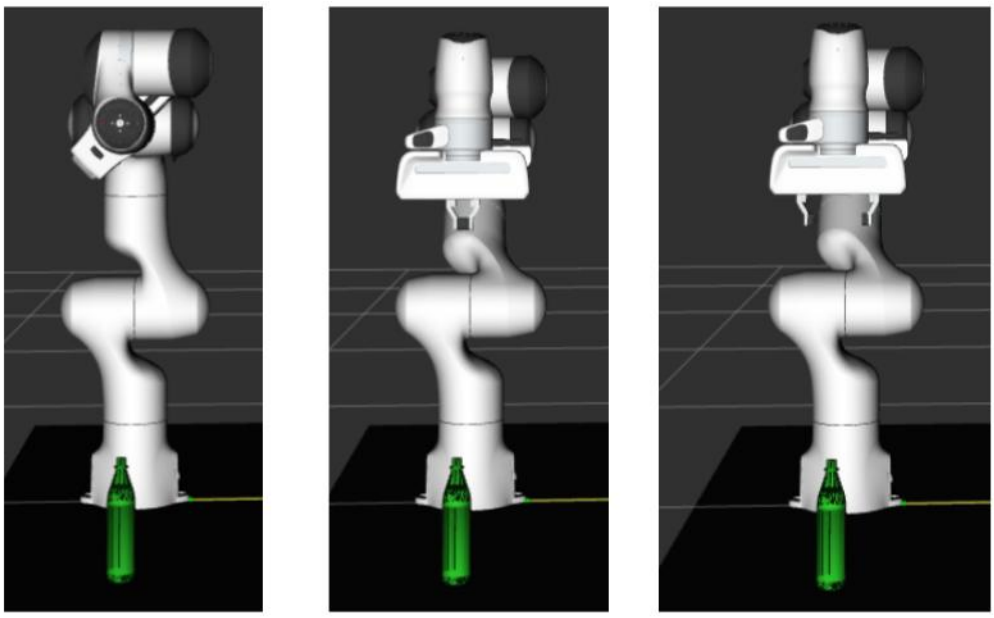
Figure 6.15 Pouring Stage for Panda arm 2

The second described application demonstrates the use of the task pipeline with custom modules, using the example of pouring into a glass. While the scenario requires a custom pouring stage, most other stages are realized with suitably parameterized standard stages to provide a robust context for this central component. The task reuses the previously described pick container to pick up the bottle. A similar container place provides a generic stage to compute place motion sequences, given a generator for feasible place poses.

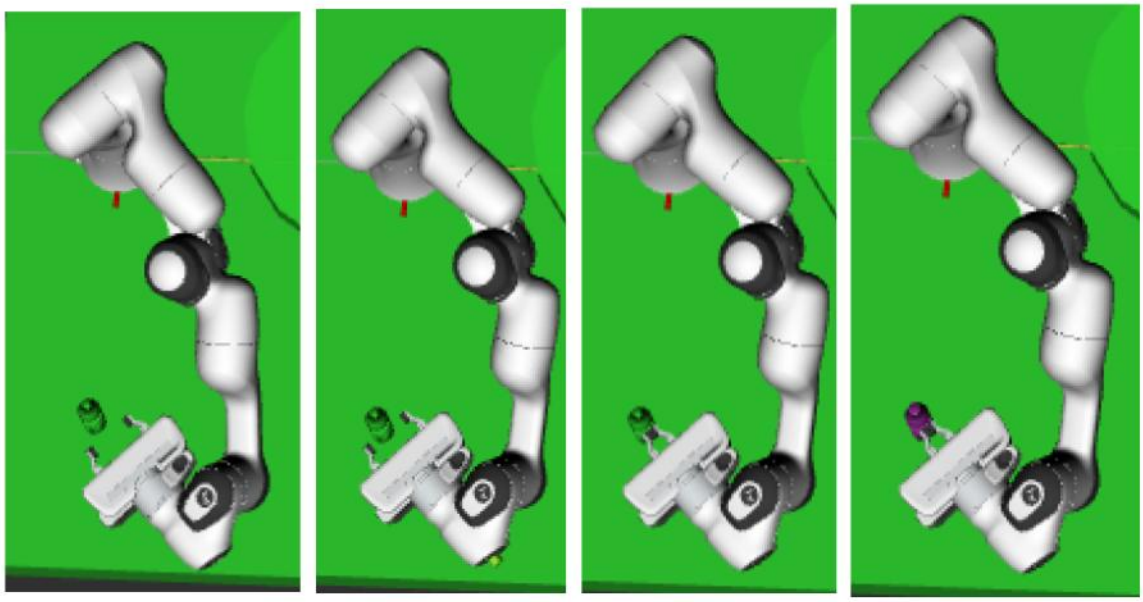
The pouring stage is implemented as tilting the tip of an attached object (the bottle) in a pouring motion over another object (the glass) for a specific period of time. The path is solved by a Cartesian planner along object-centric waypoints.

The four generator stages involved in this task are interrelated: the two last ones, bottle above glass and place location, depend on the grasp pose chosen in the pick stage. To this end, they monitor the solutions generated by the grasp stage and produce matching solutions.

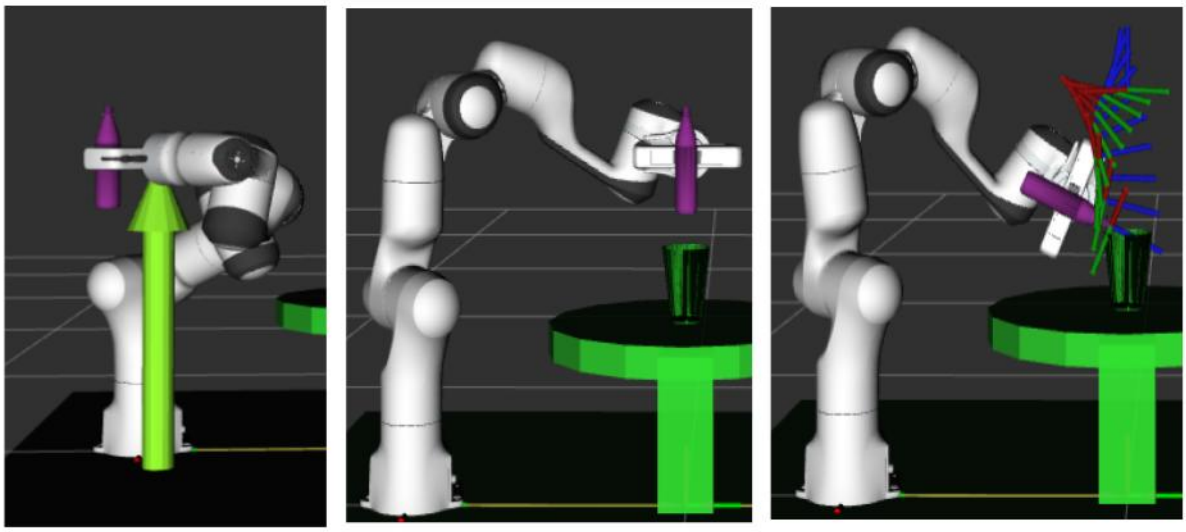
Lastly, moving the bottle over the glass and moving it towards its place location are transit motions that have to account for an additional path constraint, namely keeping the bottle upright to avoid spilling of the liquid. This constraint is specified as part of the stage description and is passed on to the underlying trajectory planner. To accelerate planning with the constraint, we make use of configuration space approximations [13] implemented for OMPL-based solvers. In our experiments, using sequential planning, the task produces its first full solution after 91.38 seconds on average.



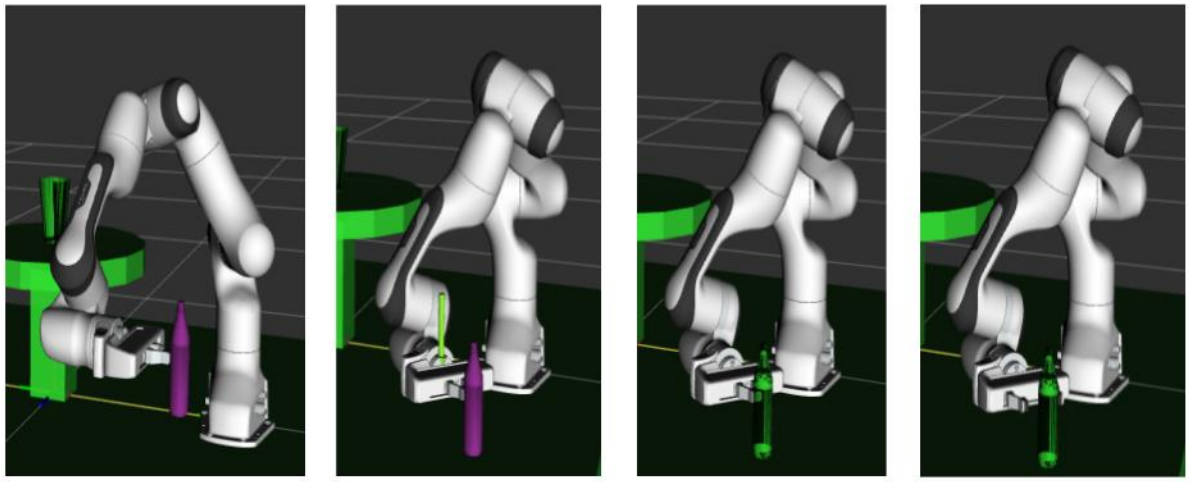
1. Current State 2. MoveTo Home 3. Open Hand



4. MoveTo Pick 5. Approach 6. Grasp 7. Attach



8. Lift 9. MoveTo Pre-Pour 10. Pouring



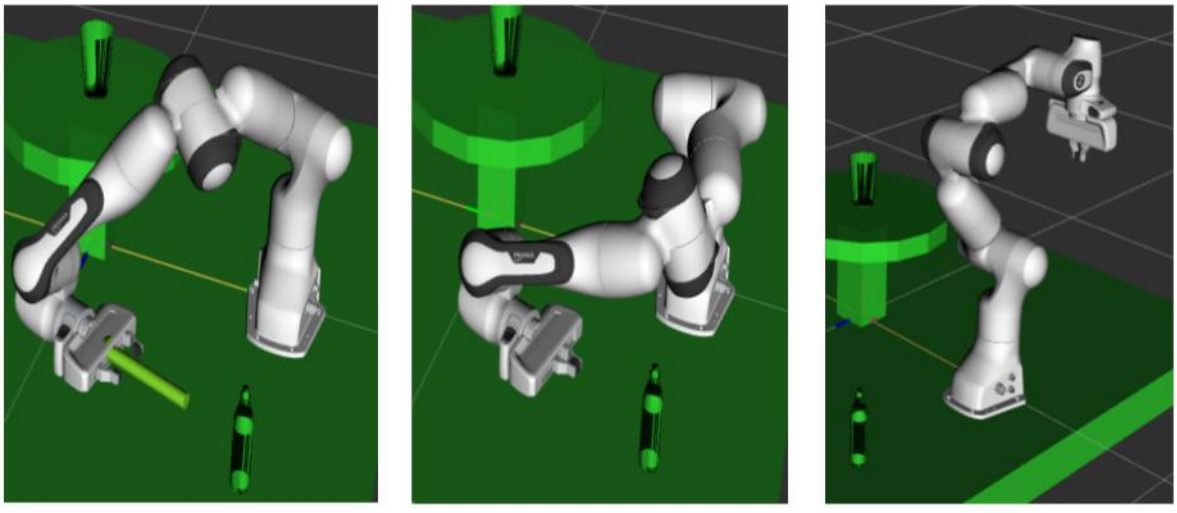


Figure 6.16 Various stages involved in pouring task

Chapter 7

Conclusion and Future Work

We discussed a modular and flexible planning system to fill the gap between high-level, symbolic task planning and low-level motion planning for robotic manipulation. Given a concrete task plan composed of individually characterized sub-stages, we can yield combined trajectories that achieve the whole task. Failures can be readily analyzed by visualization and isolation of problematic stages. The Task Constructor is meant to enhance the functionality of the MoveIt framework and replace its previous, severely limited pick-and-place pipeline. The open-source software library is under continuous development and various extensions were outlined directly within the corresponding sections.

As of now we have used of MTC to perform the multi arm task like building structure and complex pouring task. Going further, I'll continue doing more complex task using MTC.

Bibliography

[1] "Move group concept." [Online]. Available: https://moveit.ros.org/documentation/concepts/

[2] "Mtc roscon presentation." [Online]. Available: https://roscon.ros.org/2018/presentations/ROSCon2018_MoveitTaskPlanning.pdf

[3] "Github repository for btp codes." [Online]. Available: https://github.com/iamrajee/ws_moveit/tree/master/src/moveit_task_constructor/demo

[4] "Denavit–hartenberg parameters." [Online]. Available: https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters

[5] R. Diankov, "Automated construction of robotic manipulation programs, ph.d. dissertation, carnegie mellon university, robotics institute, august 2010." [Online]. Available: http://www.programmingvision.com/rosen_diankov_thesis.pdf

[6] K. Hauser, "Robust contact generation for robot simulation with unstructured meshes, in robotics research. springer, 2016."

[7] S. C. D. Coleman, I. A. Sucan and N. Correll, "Reducing the barrier to entry of complex robotic software: a moveit! case study, journal of software engineering for robotics, may 2014." [Online]. Available: https://moveit.ros.org/

[8] M. M. I. A. Sucan and L. E. Kavraki, "The open motion planning library, ieee robotics automation magazine, december 2012." [Online]. Available: http://ompl.kavrakilab.org/

[9] J. A. B. N. Ratliff, M. Zucker and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning, in robotics and automation, 2009. icra'09. ieee international con- ference on. ieee, 2009."

[10] E. T. P. P. M. Kalakrishnan, S. Chitta and S. Schaal, "Stomp: Stochastic trajectory optimization for motion planning, in robotics and automation (icra), 2011 ieee international conference on. ieee, 2011."

[11] H. R. Michael Gorner, Robert Haschke and J. Zhang, "Moveit! task constructor for task-level motion planning."

[12] T. Kroger, "On-line trajectory generation in robotic systems: Basic concepts for instantaneous reactions to unforeseen (sensor) events. springer, 2010."

[13] I. A. Sucan and S. Chitta, "Motion planning with constraints using configuration space approximations, in intelligent robots and systems (iros), 2012 ieee/rsj international conference on. ieee, 2012."