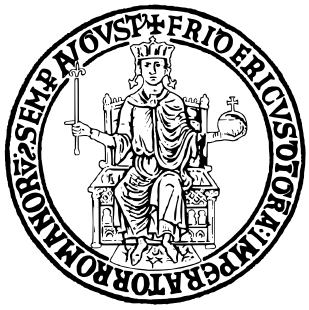
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Planning And Navigation

Pick & Place in a Blocksworld Environment using HTN Planning and MoveIt

Professor:
Riccardo Caccavale

Candidate:
Roberto Rocco P38000274
William Notaro P38000271
Chiara Panagrosso P38000272

GitHub Repositories:
https://github.com/Robertorocco/Pick_Place_Blocksworld_Environment.git
https://github.com/well-iam/Pick_Place_Blocksworld_Environment.git
https://github.com/chiarapanagrosso/Pick_Place_Blocksworld_Environment.git

Contents

Introduction

Overview

The purpose of this document is to provide internal technical insight into the structure and functioning of the packages used within the developed ROS 2-based project. The final objective of the system is to enable a robotic manipulator (specifically the Universal Robots UR5e) to autonomously perform a pick & place task in a blocksworld-like environment.

Given an initial configuration—defined by the pose of each block within a custom Gazebo world—the robot must be capable of reconstructing and executing the correct sequence of physical actions required to achieve a specified symbolic goal state. These actions involve moving individual blocks from their current positions to target locations, potentially stacked on top of one another.

High-Level Execution Pipeline

The complete execution flow can be broken down into the following high-level steps:

Step 1: World Initialization: A custom Gazebo world is initialized where both the robotic arm and the movable blocks are spawned. All blocks must be placed within the physical workspace of the manipulator to ensure reachability.

Step 2: Symbolic State Generation: The initial configuration is extracted by interpreting both the Cartesian coordinates (with respect to the world frame) and joint-space configuration of the robot. From this, a symbolic abstraction of the world state is generated (e.g., which block is on top of which).

Step 3: HTN Planning: Using the symbolic state, an HTN (Hierarchical Task Network) planner computes a plan: a sequence of high-level symbolic actions that, if correctly grounded, will transition the world from the initial to the goal state. An example of such an action is `move(A, B)`, which semantically describes moving block `A` onto block `B`.

Step 4: Symbolic-to-Physical Mapping: Each symbolic action from the planner is translated into a corresponding physical primitive. For instance, executing `unstack(A, B)` requires knowledge of the precise 3D pose (body frame) of the block `B` in the current world.

Step 5: Motion Planning and Execution: For each physical action, a collision-free trajectory for the robot's end effector is computed using a sampling-based motion planner such as RRT (Rapidly-exploring Random Tree). The trajectory is then executed using MoveIt, which handles kinematic planning, collision checking, and trajectory execution on the UR5e manipulator.

The integration between a symbolic plan generated by an HTN planner and its physical execution via MoveIt Task Constructor (MTC) required the design of a robust workflow to receive, interpret, and execute a sequence of pick-and-place actions. The plan, published on a ROS 2 topic as strings in textual format (e.g., `('unstack', 'block_a', 'block_c')`), posed a challenge due to the need to translate it semantically and structurally into parameters interpretable by the execution engine. The first phase consisted of acquiring the complete symbolic plan into an internal buffer of the node, leveraging the fact that the entire plan is published in bulk at startup. A mechanism was then implemented to clean and tokenize the received strings, removing non-significant characters and splitting each action into its logical tokens (e.g., action, object, destination). From this processed plan, a function was designed to split the sequence into consecutive pairs of pick and place actions, based on the assumption that each block is first grasped and then released. For each such pair, a dedicated method was implemented to extract the essential semantic information: the ID of the block to be manipulated and the target for placement (which can either be another block or a predefined pose on the table). This approach allowed a clear separation of concerns between reception, interpretation, and physical planning, improving the modularity of the node and making the system's behavior more transparent and traceable during debugging or future extensions. The entire process was designed to be integrated within an operational loop, where each pick/place pair drives a single instance of the MTC task to be constructed and solved.

To ensure the robustness of repeated task execution, particularly across multiple sequential pick-and-place pairs, several critical refinements were introduced in the planning scene management and internal state resetting mechanisms. A key issue observed during development was the persistent collision between the robot's links (specifically, the forearm) and previously attached objects, which occurred despite successful execution of the initial task. This was traced to the fact that MoveIt Task Constructor (MTC) internally retains a snapshot of the planning scene at the time of task instantiation, and this snapshot can become outdated between successive task executions if not explicitly reset. To address this, the system was updated to explicitly reinitialize the internal 'PlanningScene' used by MTC before each new task, ensuring that it accurately reflects the latest scene state maintained by the 'PlanningSceneMonitor'. Furthermore, the planning scene was systematically cleared of stale collision and attached objects by querying the global 'PlanningSceneInterface' and removing both known and attached collision entries associated with previously manipulated blocks. This ensured that once a block is placed, it no longer remains erroneously attached or duplicated in the collision world, avoiding spurious collisions in subsequent tasks. A short temporal delay was also introduced after scene updates to guarantee propagation and synchronization of the new scene state across all involved MoveIt components. Together, these measures resolved the key failure modes related to stale planning scene data and allowed for stable, repeated execution of symbolic plans over arbitrary sequences of pick-and-place operations.

All the steps discussed were implemented for the Universal Robots manipulator, which in its standard configuration does not include a gripper. Due to the absence of a gripping device, the physical dynamics of the pick-and-place task were simplified: specifically, the simulated world was configured without gravity, while the blocks retained their mass, inertia, and collision properties. The picking action was executed by attaching coordinate frames, rather than physically grasping the objects. Despite this abstraction, the implementation addresses the same core challenges as a traditional pick-and-place system with a gripper. In particular, obstacle avoidance was carefully managed, both for the entire structure of the manipulator and for the object being carried, ensuring collision-free motion within the

workspace. Additionally, the planning scene was dynamically updated to reflect the different stages of the task. The final chapter presents an extended version of the project that includes a simulated gripper, reintroducing the physical complexity of grasping.

Chapter 1

UR5e manipulator

The Universal Robots UR5e is a versatile and widely adopted collaborative robotic arm designed for industrial and research applications. It features six revolute joints, providing six degrees of freedom, which enable it to achieve a high level of dexterity and reach a wide range of poses in three-dimensional space. Each joint allows for precise angular motion, contributing to the arm's smooth and accurate trajectory execution. The UR5e has a payload capacity of up to 5 kg and a reach of approximately 850 mm, making it well-suited for pick-and-place tasks that involve moderate payloads within a confined workspace. One of the key reasons for selecting the UR5e is its ease of integration with ROS 2, thanks to the availability of well-maintained drivers and simulation packages, as well as its high-level control interfaces. Its collaborative nature, combined with the ability to operate safely alongside humans, further enhances its applicability in environments where flexibility, safety, and reliability are critical. In this project, the UR5e was integrated with the Gazebo simulation environment and the MoveIt motion planning framework, allowing for realistic testing and validation of pick-and-place strategies. MoveIt provides a powerful set of tools for inverse kinematics, trajectory planning, and collision checking, while Gazebo offers a physics-based simulation platform that accurately models the robot's interaction with its surroundings. This integration enabled the development of a complete planning pipeline, from symbolic task planning to physical execution in simulation, making the UR5e a particularly suitable platform for research in autonomous manipulation.

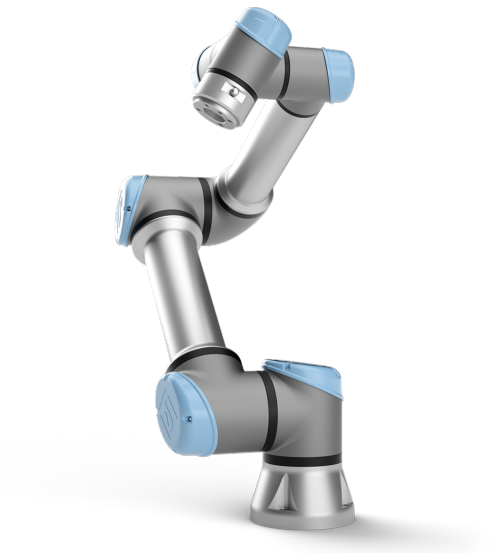


Figure 1.1: Universal Robot 5e manipulator

As a starting point for the project, the official GitHub repositories provided by Universal Robots were consulted, offering two essential ROS 2 packages tailored for the UR5e. The first is `ur_description`, available at `Universal Robots ROS2 Description-humble` repository, which includes the robot's complete description files such as the `URDF` and `xacro` for hardware interfaces, as well as a basic launch file for visualization in RViz. The second is `ur_simulation_gazebo`, accessible at `Universal Robots ROS2 Gazebo Simulation-humble` repository, which provides launch files for simulation in Gazebo along with a set of predefined controllers. These controllers are properly configured through a dedicated YAML file to interface with the UR5e hardware abstraction, facilitating a seamless integration between the robot model and the simulated environment.

Starting from the official packages, several modifications were made to ensure compatibility with the specific requirements of the pick-and-place task. First, a custom Gazebo world named `blocksworld.world` was created, containing the manipulator and six block models to be manipulated during the task. These blocks are spawned within the workspace of the manipulator to guarantee their accessibility and pickability. The URDF of the Universal Robots manipulator was extended by adding a fake link and a fake joint to simulate the presence of a gripper, which will then be subsequently integrated into the projec . To ensure compatibility with the expectations of the MoveIt framework, the robot must include a hand structure, consisting of a link—named `tool0`, visually represented as a green cube at the end of the robotic arm—and a joint—named `flange-tool0`. This joint represents the gripper and, as such, MoveIt expects it to be actuated and to expose a position control interface. Therefore, its type cannot be set to `fixed`; even though it will not physically move, the joint is defined as `revolute`. This workaround enables the creation of a "fake hand" that satisfies MoveIt's structural and interface requirements. Finally, the relevant position control interface for this joint was added within the `control.xacro` file to complete the integration.

A dedicated launch file (`setup.launch.py`) is responsible for loading the Gazebo world and the manipulator, incorporating all the modifications previously discussed. The final set of changes applied to the original packages concerns the controllers. Specifically, as shown in the launch file, the controller used for joint-level actuation is the `JointTrajectoryController`, whose configuration file can be found within the `ur_simulation_gazebo` package. In addition to this controller, the launch file also initializes a custom `HandController`, specifically created for the control of the fake hand. This controller shares the same type as the `JointTrajectoryController`. It is important to emphasize that this controller does not play any functional role in the actual manipulation of the blocks, as the manipulator is not equipped with a real gripper. Its sole purpose is to satisfy MoveIt's structural and control interface requirements. The configuration framework adopted in this scenario highlights the challenges associated with gripper integration into a Gazebo-based environment, providing a setup that is preparatory for full compatibility with MoveIt. In fact, in the later stages of the project development, for the integration of the gripper it is sufficient to replace the placeholder components discussed with the actual hardware and corresponding controller definitions.

Once all components have been correctly configured and the launch file executed, the simulated world is initialized in the desired configuration. Six blocks are spawned within the workspace of the manipulator, each positioned to ensure they are accessible for picking. The manipulator itself is spawned in a "ready" initial pose, carefully chosen to facilitate the execution of the assigned pick-and-place task. This setup ensures a smooth start to the planning and execution phases, minimizing the need for unnecessary joint reconfiguration at the beginning of the operation.

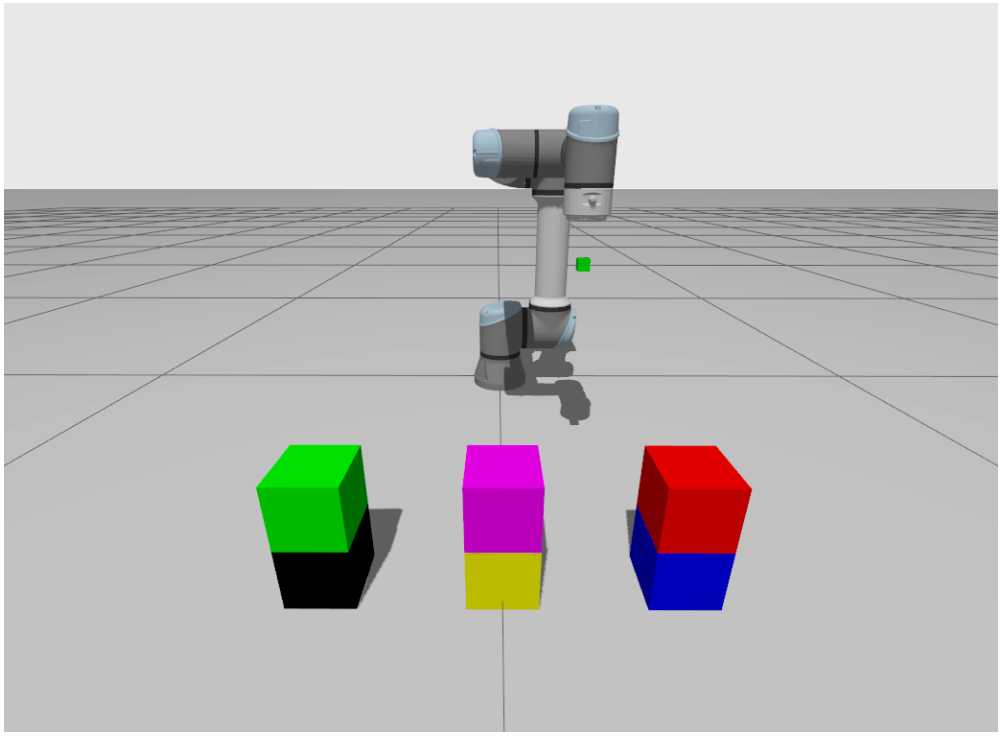


Figure 1.2: World set up

Chapter 2

Symbolic state generation

2.1 Block's Pose Plugin

After successfully loading the manipulator and the blocks into the Gazebo environment in their initial configuration, the system is ready to perform symbolic action planning. The objective is to compute a sequence of actions that transforms the current block configuration from a given initial state to a specified goal state. To enable this process, a mechanism was designed to extract the symbolic pose of the world from its geometric configuration. The symbolic pose represents abstract spatial relations between blocks, such as `on(A,B)` to indicate that block `A` is on block `B`, or `on(A,table)` when a block rests directly on the table.

To begin the symbolic extraction, it is first necessary to retrieve the geometric poses of the blocks within the Gazebo simulation. For this purpose, a plugin named `PoseRelayPlugin` was developed to bridge Gazebo and ROS. This plugin subscribes to the Gazebo topic `/world/blocksworld/pose/info`, which continuously publishes the position and orientation of all entities in the simulated world using the `ignition::msgs::Pose_V` message type—a format not natively supported by ROS. The plugin's role is to convert and republish this data in a custom ROS-compatible format specifically designed to handle block pose information.

The `PoseRelayPlugin` class performs two main functions. First, it subscribes to the `/world/blocksworld/pose/info` topic and, upon receiving each message, executes the `OnPoseMsg` callback, which safely stores the data in a mutex-protected field. Second, every 10 milliseconds, the most recent message is copied and processed: only the poses corresponding to the blocks are extracted and published on a new ROS topic called `/object_pose`. The message type for this topic is `BlockPoseArray`, a custom message consisting of an array of `BlockPose` elements. Each `BlockPose` contains a `string` field named `name`, identifying the block (e.g., `block_a`), and a standard `geometry_msgs/Pose` message. The array is accompanied by a header specifying the reference frame relative to which all poses are expressed. This system enables reliable, real-time communication of the geometric configuration of the environment from Gazebo to ROS, laying the foundation for accurate symbolic reasoning. The use of an internal variable within the `PoseRelayPlugin` class is essential to decouple the subscription to the original Gazebo topic from the publication on the corresponding ROS topic. This design choice allows for precise control over the publishing frequency. In particular, attempting to mirror the high-frequency publication rate of the Gazebo topic directly in ROS led to excessive CPU usage, with the plugin monopolizing computational resources. By storing the latest message in a mutex-protected variable and publishing it at a reduced, fixed rate (e.g., every 10 milliseconds), the system achieves a more efficient balance between responsiveness and resource consumption, ensuring stable operation even

during extended simulation runs.

Finally, the plugin was successfully integrated into the Gazebo world described in the previous section (so, directly into the `ur_description` package. Its inclusion was performed by embedding the plugin directly into the SDF definition of the last block loaded into the environment, which in this case corresponds to the yellow-colored block. This design choice was made to ensure that the plugin is instantiated only after the entire Gazebo world has been initialized. An attempt to load the plugin directly from the launch file proved unreliable, as it sometimes resulted in the plugin being executed before the relevant Gazebo topic was available. This race condition caused subscription errors, preventing the plugin from functioning correctly. Embedding the plugin in the last block's SDF model ensures a consistent and delayed initialization sequence that avoids such issues and guarantees correct operation.

All the modifications discussed in this section can be found within the `my_sim_plugin` folder, which contains both the definition of the custom `BlockPoseArray` message and the full implementation of the `PoseRelayPlugin` bridging class.

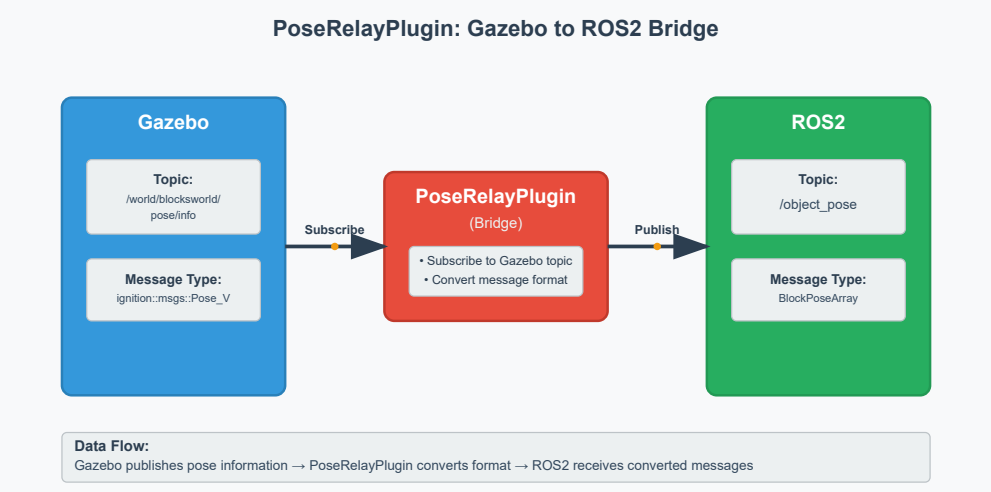


Figure 2.1: Pose Relay Plugin

2.2 From block's coordinates to symbolic state

Once the `/object_pose` topic provides the coordinates of the blocks present in the simulated environment, it becomes possible to reconstruct the symbolic initial state to be supplied to the planner. The implemented algorithm relies on a simple yet effective heuristic based on the known geometry of the blocks. Assuming the blocks are cubes with edge length $h = 0.1$ meters, their body frames are located at the geometric center, resulting in a z-coordinate of approximately 0.05 meters when resting on the ground. Introducing a tolerance margin of 0.025 meters to account for potential noise or numerical inaccuracies, a block A is classified as being on the table, i.e., satisfying the predicate `on(A, table)`, if its z-coordinate is less than 0.075.

If instead the z-coordinate of block A is greater than 0.075, the algorithm attempts to identify another block W such that the predicate `on(A, W)` holds. To do so, it computes the relative spatial differences $\Delta x = x_A - x_W$, $\Delta y = y_A - y_W$, and $\Delta z = z_A - z_W$ for all candidate blocks W. A block W is considered to be directly beneath A if the following conditions are simultaneously satisfied:

$$0 < \Delta z < 0.125 \quad \text{and} \quad |\Delta x| < 0.025 \quad \text{and} \quad |\Delta y| < 0.025$$

The first condition ensures that A is vertically stacked above W without other blocks in between, while the latter two confirm sufficient horizontal alignment on the xy-plane. The margin of 0.025 meters is again used to account for small displacements due to simulation artifacts. This approach effectively transforms continuous geometric information into discrete symbolic predicates suitable for task-level planning.

The main advantage of the implemented logic lies in the fact that the programmer is relieved from manually specifying the initial symbolic state for the planner. Given a properly constructed Gazebo world and the known height of the blocks, the system can autonomously reconstruct the initial symbolic configuration. This approach ensures flexibility and generality, as the only requirement for the user is to define the simulation environment; the symbolic state will be derived automatically based on the geometric poses of the objects. The algorithm responsible for converting geometric coordinates into symbolic predicates is implemented directly within the planner, as part of the `blocksworld_planner` package, which will be discussed in the following chapter.

Chapter 3

Action's Planning

3.1 Hierarchical Task Network

The planner used to generate the sequence of actions required to transition the Gazebo world from the initial to the goal configuration is based on the *Hierarchical Task Network* (HTN) paradigm. The core idea behind HTN planning is to focus on a limited set of abstract actions in order to simplify the derivation of the final execution plan. By appropriately defining these high-level tasks, the planner can more easily infer the required sequence of lower-level actions. Each abstract action is then decomposed into simpler tasks that are directly executable by a motion controller.

Formally, HTN planning operates on the concept of *tasks*, where a task is an expression of the form $t(u_1, u_2, \ldots, u_n)$, with t being a task symbol and u_i the parameters on which the task acts. Tasks are classified as either *primitive*, if the agent knows how to execute them directly, or *compound*, if they require decomposition into subtasks before execution. The decomposition is governed by *methods*, each of which provides a pattern of subtasks invoked to perform a specific compound task.

A method consists of a name, the compound task it implements, a set of preconditions that must be satisfied for the method to be applicable, and a recipe that defines the sequence of subtasks to execute. Unlike classical planning techniques, which reason in terms of world states and goal conditions, HTN planning operates in terms of task progression. In fact, the goal is not specified as a desired world state but as a set of tasks to be completed. The planning process terminates when the task list—composed of both the initial goal tasks and those generated during decomposition—is empty.

3.2 Blocksworld Domain

A planning problem in the Hierarchical Task Network (HTN) formalism is defined by a triple

$$\Pi = (D, s_0, T)$$

where:

- D is the domain, which describes the environment in which the agent operates,
- s_0 is the initial state of the world (automatically reconstructed, as described in the previous section, starting from the Gazebo configuration),
- T is the set of tasks to be achieved.

The domain D consists of:

- a finite set of *constants*, which in the blocksworld correspond to the identifiers of the blocks in the scene (e.g., `block_a`, `block_b`, ...),
- a finite set of *predicates* that describe the state of the world,
- a set of *primitive actions* that the agent can execute directly,
- a set of *methods* used to decompose high-level (compound) tasks into simpler ones.

In a blocksworld-like domain, the predicates used to describe the state are:

- `on(x,y)`: block `x` is placed on entity `y` (which could be another block or the table),
- `clear(x)`: block `x` has no block on top of it and can therefore be picked,
- `holding(x)`: the robot is holding block `x` (or `false` if it is not holding anything).

From the geometric configuration of the blocks in Gazebo, the algorithm described in Section 2.2 reconstructs the initial symbolic state, establishing the value of each predicate.

Primitive Actions In a classical blocksworld setting, there are four primitive actions:

- `pickup(b)`: picks up block `b` from the table, provided `b` is clear and the gripper is free,
- `unstack(b,c)`: picks up block `b` from block `c`, provided `b` is clear and the gripper is free,
- `putdown(b)`: places block `b` onto the table,
- `stack(b,c)`: places block `b` on top of block `c`, provided `c` is clear.

Each of these actions modifies the symbolic state, and their applicability is subject to specific preconditions. They are the only directly executable operations by the agent.

HTN Methods What distinguishes HTN planning from classical approaches (like STRIPS) is the use of *methods*, which define how to decompose a compound task into subtasks. In this implementation, four methods are defined:

1. `get` method: encapsulates the logic required to grasp a block. Whether the block is on the table (requiring `pickup`) or on another block (requiring `unstack`), the `get` method abstracts away from this distinction and delegates to the appropriate primitive action. It solves the `get` task.

2. `put` method: abstracts the placement of a block. Whether the destination is the table (requiring `putdown`) or another block (requiring `stack`), the method handles the decomposition accordingly. It solves the `put` task.

3. `move1` method: defines a complete high-level movement of a block from its current location to its desired position (which may be the table or another block) by composing a `get` and a `put`. It solves the `move_one` task.

4. `move_b` method: the top-level method that recursively generates the sequence of block moves needed to achieve the goal configuration. It relies on helper functions (e.g., `status`, `is_done`) to decide which block to move next. At each recursion step, it schedules a new `move_one` task followed by another call to `move_blocks`, effectively constructing the plan incrementally. It solves the `move_blocks` task.

Execution of the Planner The planner is launched by initializing the SHOP framework with the previously defined methods and primitive operators, specifying the top-level compound task `move_blocks` as the sole planning goal. During execution, SHOP recursively decomposes this task into a sequence of `move_one`, `get` and `put` subtasks, eventually yielding a linear plan composed exclusively of the four primitive actions (`pickup`, `unstack`, `putdown`, `stack`). Once generated, the complete plan is serialized into a series of string tuples—each representing one primitive action—and published on the ROS2 topic `pyhop_plan`. This topic serves as a buffer holding the entire action sequence, which the downstream controller consumes to drive the physical execution of the pick-and-place task. An example message is:

```
('unstack', 'block_a', 'block_c')
```

The entire implementation described in this chapter is encapsulated in the ROS2 package `blocksworld_planner`.

Chapter 4

Moveit Setup

The physical execution of the actions dictated by the planner is entirely delegated to MoveIt 2. The first step toward integrating MoveIt 2 correctly involves generating the necessary configuration files using the *MoveIt Setup Assistant*. This graphical interface facilitates the configuration of any robot for MoveIt by generating a Semantic Robot Description Format (SRDF) file. The SRDF augments the robot's URDF with essential information for planning, such as the definition of planning groups, end-effectors, and various kinematic constraints.

In addition to the SRDF, the Setup Assistant creates all the configuration files required by the MoveIt planning pipeline. To begin the setup process, a URDF description of the robot must be provided. Once imported into the Setup Assistant, the user is guided through a series of steps to configure the robot's kinematic structure, assign joints to planning groups, define end-effectors, and specify settings related to collision checking and planning behavior. Since MoveIt is responsible for the collision-free motion of the manipulator, the SRDF file contains essential information such as the planning groups to be controlled (e.g., the group representing the entire manipulator arm), passive joints, disabled collisions between adjacent links, and the definition of *group states*, which correspond to standard poses used during motion planning stages. In addition to the SRDF, several `xacro` files handle the loading of components required by `ros2_control`, enabling hardware interface integration. Another important configuration generated by the Setup Assistant is the `moveit_controllers.yaml` file. This file specifies the MoveIt controller manager, which acts as an interface between ROS and MoveIt. Once a valid trajectory is planned, the controller manager selects the appropriate ROS controller based on the planning group involved and forwards the trajectory for execution. Therefore, the `moveit_controllers.yaml` must list the controllers already configured and made available in the ROS system, as detailed in Chapter 1.

Once the configuration process is completed through the MoveIt Setup Assistant, a dedicated configuration package is generated containing all the necessary files for MoveIt integration. In our case, this package is named `ur5_new_moveit_config`. It includes the SRDF file enriched with semantic information, the controllers configuration, and other parameters required to enable motion planning and trajectory execution. This package serves as the bridge between the robot description and the motion planning framework, allowing seamless interaction between MoveIt, `ros2_control`, and the simulated environment.

4.1 Fake Hand Configuration

As previously explained, the goal of the first part of this project is to demonstrate a methodology for using MoveIt to plan pick-and-place tasks for a manipulator with no physical gripper. However, MoveIt requires the presence of an end-effector defined in the SRDF to enable planning. To address this, a minimal abstraction of a gripper — referred to as a *fake hand* — was introduced.

This abstraction requires defining a new planning group in the SRDF that represents the hand. The group includes a joint that is considered actuated, even though it is purely fictitious. In our case, a revolute joint was added between the last original link of the manipulator and a newly defined link referred to as the tool frame. Although this joint is not physically meaningful, declaring it as revolute rather than fixed is necessary, since MoveIt only associates actuated joints with end-effectors.

The new link, graphically rendered as a small cube, is positioned slightly forward from the last wrist link and serves as the frame to which objects are virtually attached during pick operations. The associated planning group, named `hand`, contains only this joint and allows MoveIt to treat it as a valid end-effector group. To complete the configuration, the joint representing the fake hand must be registered with a controller in the `ros2_control` configuration. This includes declaring the joint interfaces for position, velocity, and effort, even if these values are never physically used. This trick convinces MoveIt that the joint is controllable and satisfies the internal constraints for planning with end-effectors. It is also essential that the appropriate controller for the fake joint is loaded from the launch file. Without this, MoveIt will not consider the hand group valid for motion planning, and any planning stages requiring an end-effector — such as `Pick` or `Place` — will fail. This modification ensures compatibility with MoveIt's planning pipeline, while maintaining the abstraction of a purely virtual gripping mechanism.

Launch of the simulation environment Once the configuration has been completed as described, the entire simulation environment — including the world, the UR5e manipulator, and the MoveIt pipeline — can be launched through a dedicated launch file named `setup.launch.py`, located in the `ur_description` package.

This launch file starts the Gazebo simulator with the robot and its surroundings, initializes MoveIt with the generated SRDF and the necessary controllers, and opens RViz2 with a preconfigured layout to visualize the robot, its planning groups, and the trajectories. At the same time, it activates the `ros2_control` infrastructure, including the controller for the fake hand. This setup ensures a coherent and synchronized environment for testing and executing pick-and-place tasks within a simulated context.

Chapter 5

MoveIt Task Constructor

The MoveIt Task Constructor (MTC) is a framework designed to decompose complex robotic tasks into a sequence of interdependent subtasks, each of which is solved by the MoveIt planning pipeline. Tasks are represented as a hierarchy of *stages*, where each stage encapsulates a single step in the planning or execution process and exchanges information via an `InterfaceState` object.

Stages are classified into three primary categories based on how they handle data flow: *Generators* initialize solutions (for example, `CurrentState` captures the robot's starting configuration, while a `GeneratePose` stage monitors preceding stages to propose candidate grasp or approach poses); *Propagators* receive an input state from one side, compute successor states (such as moving relative to an object), and pass results onward; and *Connectors* attempt to bridge start and goal states directly by solving for feasible trajectories. Additionally, *Wrappers* may be used to augment existing stages—for instance, by computing inverse kinematics for Cartesian pose proposals.

To organize complex sequences, MTC provides *containers* that allow stages to be composed either in series or in parallel. A default MTC *Task* is implemented as a serial container, that contains a set of stages executed in a strict end-to-end order.

Finally, stages requiring motion planning must be configured with a suitable solver. MTC supports multiple planners: the full MoveIt planning pipeline via `PipelinePlanner`, simple joint-space interpolation through `JointInterpolation`, and straight-line Cartesian motions via `CartesianPath`. Each solver can be assigned to individual stages to match the requirements of the subtask at hand.

5.1 Planning Scene

The first essential step before initiating any Pick–Place task is to update the Planning Scene, which represents the manipulator's internal model of the environment. This is achieved through an instance of the `PlanningSceneInterface`, which provides functionalities for adding or removing collision objects, attaching or detaching them from the robot, and querying the known entities in the scene.

Before each Pick–Place task, the scene is populated with collision objects representing the blocks in the Gazebo world. The block poses are retrieved from the `/object_pose` topic, which continuously publishes their positions. Each block, along with its corresponding dimensions, is then added to the Planning Scene at the appropriate location. This process is carried out in a thread-safe manner by using a mutex that guards the internal class field where the poses are stored and updated through the topic callback. The update of the Planning Scene, based on the actual positions of the blocks observed in the Gazebo world,

lays the groundwork for implementing a replanning strategy in the final chapter of this report. By continuously aligning the internal representation of the environment with its real configuration, it becomes possible to adapt the motion plans to unexpected changes or deviations, thereby improving the robustness and responsiveness of the overall planning system.

5.2 Symbolic-to-Motion Mapping in the MTC Framework

On top of the MTC framework we have defined a composite task comprised of two main containers—*Pick* and *Place*—each containing a carefully ordered sequence of stages tailored to execute the corresponding subtask. A dedicated ROS2 node (`mtc_node.cpp`) orchestrates the task execution. Upon startup, the node subscribes to the `/object_pose` and `/pyhop_plan` topics, thereby remaining aware of the real-time positions of all blocks in the Gazebo world and retrieving the action sequence produced by the HTN planner. As soon as the plan becomes available, each action is received as a single string of the form `('unstack','block_a','block_c')` and is processed by utility functions that split it into a vector of substrings—where the first element denotes the primitive name and the subsequent elements represent its parameters.

Since both the initial and goal states require the manipulator to be holding no block, the total number of primitive actions is guaranteed to be even. Consequently, the sequence can be partitioned into successive pick–place pairs, in which each pick action (`unstack` or `pickup`) is immediately followed by its corresponding place action (`stack` or `putdown`). The node thus constructs a vector of pick–place task pairs, which is then fed into the MoveIt Task Constructor. In this way, each pick–place pair is executed as a coherent subtask within the overall MTC pipeline. Finally, a `for` loop iterates over the number of pick–place pairs and sequentially executes each corresponding task through the MoveIt Task Constructor pipeline. In this way, the overall plan generated by the symbolic planner is carried out as a sequence of physical pick-and-place motions, each of which is fully handled by MoveIt.

5.3 Pick–Place Stages Definition

Starting from the MoveIt `pick_and_place` tutorial with MTC, the task has been adapted to our scenario. The referenced framework provides a general schema for a Pick–Place task by defining the essential stages required for correct execution. The adapted task begins by capturing the robot's current configuration with a `CurrentState` stage, which seeds the pipeline. A `Connect` stage named "move to pick" then plans a collision-free trajectory from that initial state to a pre-grasp configuration using the MoveIt planning pipeline. Within the `pick object` container, a `MoveRelative` stage first approaches the object along the end-effector's forward axis, followed by a `GenerateGraspPose` stage that proposes grasp candidate frames. These Cartesian frames are translated into joint configurations via a wrapped `ComputeIK` stage. A `ModifyPlanningScene` stage temporarily allows collisions between the object and its environment, then another `ModifyPlanningScene` stage attaches the object to the end-effector. A subsequent `MoveRelative` stage lifts the object, and a final `ModifyPlanningScene` stage in the pick container restores collision constraints. The pick operation is consistently executed from above, i.e., by grasping the block on its top face. This behavior is enforced during the generation of grasp poses. In particular, the direction of approach is explicitly defined by constructing a grasp frame whose Z-axis aligns with the normal vector of the block's upper face. This is achieved by computing a quaternion that rotates the end-effector's local axis into the desired approach direction -in this case,

a vector pointing along the positive Z axis in the block's frame. The orientation is then converted into a rotation matrix and applied to the grasp frame. Additionally, the grasp pose is translated outward along this approach direction by a fixed clearance offset, ensuring the manipulator remains at a suitable distance before initiating the final contact. By constraining the sampling space in this way, only top-down grasps are generated, effectively restricting the grasp strategy to a single, well-defined orientation aligned with the vertical.

Next, a second `Connect` stage, "move to place," generates a path from the lifted pose to a target pre-placement configuration. Inside the `place object` container, a `GeneratePlacePose` stage samples valid drop locations—either on the table or atop another block—which are then passed through `ComputeIK`. A brief `MoveRelative` "settle before detach" stage ensures stable placement, followed by a `ModifyPlanningScene` stage that detaches the object, and a final `MoveRelative` "retreat" stage that withdraws the arm. The sequence concludes with a `MoveTo` stage named "return home," which uses joint interpolation to reset the manipulator to its predefined home configuration.

Although the source code lists the stages in a certain sequence, the actual temporal order in which the MoveIt Task Constructor executes them follows the data-flow dependencies rather than their textual arrangement. In particular, all *generator* stages run first to produce the necessary interface states, followed by *propagators* and finally *connectors*. As an example, consider the initial pick phase. The pipeline begins by obtaining the robot's current configuration via the `CurrentState` generator. Next, before planning any approach motion, the `Generate Grasp Pose` generator computes the target grasp frame at which the end-effector must arrive to secure the block, and its IK to get the robot posture at grasp moment. Only once the grasp pose is available can the `Approach Object` propagator determine the direction and distance of the approach movement. Finally, the `Move To Pick` stage links the robot's original pose to the start of that approach trajectory, producing a continuous, collision-free path from the initial state to the pre-grasp configuration.

A critical issue when using MoveIt for collision checking arises from the presence of gravity: Gazebo spawns the blocks in direct contact with one another and with the table, reflecting a physically accurate stacking. Without special handling, MoveIt will detect these initial contacts in the planning scene and refuse to compute any motion due to perceived collisions. To address this, two dedicated stages—`allow block-environment collisions` and `forbid block-environment collisions`—are introduced in the pipeline. The core idea is that, prior to grasping, MoveIt must ignore all collisions involving the blocks (both block-to-block and block-to-ground) in order to avoid false positives arising from the initial stacked configuration. This behavior is governed by the Allowed Collision Matrix, the MoveIt data structure that determines which pairs of entities are evaluated for collisions during planning. By temporarily configuring the matrix to permit these contacts until a block is attached, and then restoring the standard collision constraints once the object is grasped, the planner can operate seamlessly from a physically accurate starting state.

5.3.1 Place Phase

A final analysis concerns the differences between the pick and place phases, specifically when comparing the actions `stack` and `putdown`, and `unstack` and `pickup`. Notably, the pick phase does not require any distinction between grasping a block from another block or from the table. In both cases, the operations involved are essentially the same. Conversely, when placing a block, a distinction between the two actions becomes necessary. If the block is to be placed on top of another, it is essential to compute the coordinates of the supporting block. By adding an appropriate vertical offset to this position, the target place

pose can be determined. If instead the block is to be placed on the table, a valid placement position must be generated. This is achieved through a random sampling procedure that selects candidate poses on the tabletop, varying only in the x and y coordinates. The place poses on the table are sampled within a restricted range: $x \in [0, 1]$ and $y \in [-1, 1]$. This constraint ensures that the candidate positions lie within the frontal quadrants of the robot's workspace, reducing the need for large motions and keeping the blocks spatially close to one another.

The generated poses are subject to two constraints. First, the pose must lie within the robot's reachable workspace, which, without loss of generality, can be validated by confirming the existence of an inverse kinematics solution. Second, the pose must not collide with any blocks already present in the environment. Given the known dimensions of the blocks, the sampling algorithm ensures that the selected pose does not fall within a region considered too close to existing objects, thereby avoiding collisions in the updated planning scene. Finally, in this phase the `SettleBeforeTouch` stage was added to the task construction to ensure maximum precision during the final moments of the *place* phase. This stage is particularly critical because any inaccuracy in the manipulator's motion at this point may result in incorrect positioning of the block. Such an error is especially problematic in stacking scenarios, where a poorly placed block may compromise the stability of the entire stack. To mitigate this risk, the `SettleBeforeTouch` stage introduces a slow and controlled vertical motion, implemented using a dedicated Cartesian interpolator. This controlled descent limits the contact forces exerted during placement and ensures that any residual accelerations of the block are directed only along the vertical axis before it is released. As a result, the manipulator can achieve a stable and precise placement, reducing the likelihood of unintended displacements or stack failures. A careful tuning of the vertical distance covered in the `Settle` stage was carried out. By adjusting this distance, it is possible to control whether the block is released slightly above the target surface—allowing gravity to complete the placement—or directly in contact with it.

Solution Search in MoveIt Task Constructor

The MoveIt Task Constructor provides the functionality to select, via code, the number of solutions to search for the entire task before the pick-and-place motion is actually executed. It also allows the user to visualize in RViz the different solutions found and how the search for solutions is performed at each stage.

Moreover, for each stage, it is possible to define a set of search constraints, such as maximum number of solutions or planning time or the minimum required distance between different solutions. This is useful in particular for stages involving a wider range of motion, that generally require more computational time for solution searching: it is important to properly configure these parameters in order to avoid significantly a limitation of the overall planning performance while also preventing excessive execution times.

Among the candidate solutions found for the entire task, the Task Constructor will select the one with the lowest cost. It is important to note that for certain stages the motion planning is performed using RRT algorithm, which is sampling based. As a result, increasing the number of attempted solutions does not always guarantee a better path, especially considering the random nature of the place pose that may be selected.

Grasping Without a Physical Gripper

In scenarios where the manipulator is not equipped with a physical gripper, performing a pick operation requires simulating the grasp by attaching the block's frame to that of the manipulator's fake hand. As previously introduced, the relevant reference frame for the manipulator is represented by the green cube visible in the simulation.

To this end, when the `attach` stage is invoked within the MTC pipeline, a dedicated node named `BlockFollower` is triggered by the `mtc_node`. This node modifies the Planning Scene to virtually attach the block to the manipulator. Simultaneously, the block's pose in the Gazebo environment is updated via direct communication with the `SetEntityPose` service.

The `BlockFollower` node acts as a client to the aforementioned service, allowing it to overwrite the pose of Gazebo entities. The request includes the current pose of the manipulator's flange, retrieved through a `lookupTransform` call. To ensure realistic rotations of the block during transport, the orientation is derived from one of the wrist joints of the manipulator. Since the block is always grasped from the top—i.e., on its Z face—the `BlockFollower` enforces that the final pose written to Gazebo aligns the block's orientation with the world frame. This guarantees that every pick is performed vertically from above. As will be discussed later, the use of this service becomes unnecessary when simulating the same process in the presence of an actual gripper.

Once Gazebo is launched, the `pick_place.launch.py` file initializes the MoveIt Task Constructor node, which remains idle while waiting for a plan to be received. As soon as a plan is published on the `/pyhop_plan` topic, MoveIt begins the planning phase for the first Pick–Place task.

Chapter 6

Realistic Gripper-Based Grasping

Once the simulations of the manipulator performing Pick–Place operations without a gripper were completed, the integration of a compatible gripper with the Universal Robots manipulator was carried out. It is important to recall that, in the gripper-less configuration, the physical interactions between gripper and object—an essential component of real-world Pick–Place tasks—are entirely bypassed. In this setup, object attachment is simulated by aligning and attaching frames, rather than relying on contact dynamics. Moreover, the simulated world for the gripperless manipulator was launched without gravity. This design choice was made to prevent conflicts between Gazebo's physics engine and the frame-based attachment method. In particular, enabling gravity would cause Gazebo to pull the block away from the attached frame during transport, as it attempted to resolve the physical inconsistency between gravitational forces and the static frame attachment. This led to inaccuracies in the real-time pose updates and degraded simulation reliability.

Strictly speaking, executing a Pick–Place task without gravity is inherently unfeasible. In the absence of gravitational force, once the object is released during the place phase, even the most precise and slow motion is insufficient to ensure stability. If the net force acting on the block is non-zero, the block's pose will diverge after release, leading to unpredictable results. A minimal amount of gravity is therefore required to stabilize the block's position upon release. For this reason, in the gripperless Pick–Place scenario, the settle phase is designed to slightly press the block onto the target surface—whether it is the table or another block—during the final stage of the placement. By carefully tuning the physical parameters in the block's SDF model, such contact pressure enables surface adherence, compensating for the absence of gripping forces. However, this approach must be used with caution. If the applied force is excessive, two undesirable outcomes may occur: in the case of stacking, the underlying structure may become unstable and collapse; alternatively, if the contact force is too high, the resulting interaction may cause the block to settle in an unintended position, introducing unpredictability into the overall task execution.

6.1 Integration of Robotiq 2-Finger 85 Gripper

The chosen gripper to be integrated with the UR5e manipulator is the Robotiq 2F-85, primarily due to the high compatibility between the two systems, which facilitates the development of applications deployable in real-world scenarios. Mechanically, the manipulator and the gripper feature matching mounting surfaces that allow straightforward assembly via screw fixation. Additionally, the UR5e provides a 24V power supply through its flange, which corresponds exactly to the voltage required to operate the gripper.

From a functional standpoint, the gripper supports a payload of up to 5 kg, which

matches the maximum capacity of the selected manipulator, making it an ideal combination for pick-and-place tasks. Software integration was also straightforward thanks to the modular structure of the driver package provided by the manufacturer. Specifically, a custom macro called `eef.xacro` was created within the `ur_description` package, enabling the inclusion of the gripper's macro into the UR5e URDF and its direct attachment to the `flange` link, which represents the manipulator's end-effector mounting interface.

The gripper itself offers a maximum stroke of 85 mm and a gripping force ranging from 20 N to 235 N. The fingers are coated in silicone to ensure high friction forces, enabling a secure static contact without the need to apply excessive force. These characteristics led to a revision of the simulation environment, particularly in the physical modeling of the object interacting with the gripper. First, the size of the blocks had to be reduced to allow for successful grasping with the gripper. Additionally, custom physical properties—such as varying friction coefficients—were assigned to the blocks to highlight the reliability of contact enabled by the gripper's fingers.

Regarding the software functionality provided by the manufacturer' ROS2 package for the gripper, it is important—particularly from a control perspective—to highlight the way the URDF is structured. Among the six internal joints defined in the model, four are passive by design. Among the remaining two finger joints, only one has been equipped with a control interface, and thus is configured to receive commands from an external controller. In practice, the controller acts solely on this single joint, referred to as `robotiq_85_left_knuckle_joint`. All other joints are configured to mimic the behavior of this primary joint by means of appropriate multipliers. These multipliers are tuned so that the resulting motion is physically consistent with the real behavior of the gripper during opening and closing actions.

Figure 6.1: Robotiq 2-Finger Adaptive Gripper

6.2 Gripper Integration in the Pick&Place Pipeline

The integration of a gripper into the work developed in the previous chapters naturally follows the concepts introduced earlier. In Chapter 4.1, the integration of a fake hand into the original UR5 model was discussed in detail. At this stage, all components previously designed to simulate the presence of a hand—required by MoveIt to correctly execute motion planning stages—are now concretely implemented. Starting from the URDF provided in the `robotiq_description`, the SRDF file generated using the MoveIt Setup Assistant also includes the description of the gripper structure and the configuration of its associated controllers.

The entire planning pipeline previously discussed for the gripper-less case remains largely unchanged. In particular, from the planner's perspective, the plan to be generated does not require any structural modification when the manipulator is equipped with a gripper. Similarly, the construction of the task follows the same sequence of stages, with only minor adjustments in some stage parameters to account for the presence of gravity and the slightly smaller size of the blocks. The main difference lies in the fact that, when a physical gripper is present, Gazebo is responsible for handling the interaction dynamics between the gripper and the object during grasping. Moreover, since gravity is now properly simulated, the blocks can be released a few millimetres above the supporting surface during the place phase. This allows the blocks to settle naturally without requiring physical contact in the final stage, preventing deformation or interpenetration, and ensuring that the interaction between the block and the surface is realistic and controlled.

The actuation of the gripper is controlled via a position controller associated with the left finger joint, whose motion is mirrored by the right finger. The actual grasping occurs in the `close` stage, implemented specifically for this version of the code (with the `open`). This stage commands the gripper to move to the `close` configuration, as defined in the SRDF file. This configuration was carefully tuned to ensure a secure grasp on the block while avoiding excessive compression, which could otherwise lead to controller instability or runtime errors due to force limits being exceeded. Once this stage is completed, the block is physically grasped by the manipulator. The `BlockFollower` node, which was previously used to synchronize the block's position by publishing transformations during the gripper-less simulation, is no longer invoked. From this point on, the block's motion is governed by the physics engine in Gazebo, rather than by direct frame overwriting. This shift enables a more realistic and physically consistent interaction between the gripper and the object.

6.3 Experimental Evaluation and Results Comparison

In both the gripper and gripper-less scenarios, the planning and execution of the Pick-and-Place tasks were successfully completed. However, two main differences were observed.

First, in the gripper-less case, due to the near absence of gravity, the block must be forcibly placed in a pose that slightly penetrates the target surface to ensure contact. This constraint may occasionally lead to misplacement or instability, especially during stacking operations, where improper positioning can cause the entire stack to collapse. Conversely, when using the gripper, the presence of gravity allows for more natural and stable placement. Blocks are released a few millimeters above the target surface, relying on gravity to ensure contact without introducing undesired interpenetration or deformation. This approach leads to more reliable stacking behavior.

However, an important drawback was observed during the manipulation phases with the gripper: Gazebo's real-time factor (`real_time_factor`) drops significantly, often reaching values around 0.2 when the gripper is in contact with a block. The `real_time_factor` represents the ratio between the simulation speed and real time; a value of 1.0 indicates real-time simulation, while lower values signal computational slowdowns. A factor of 0.2 indicates that the physics engine is struggling to handle the complexity of the contact interactions. This degradation is attributed to factors such as fine-grained collision computations, low object masses, high friction coefficients, or unstable control loops. The slowdown reflects the increased computational cost of resolving detailed physical interactions within the simulator.

Chapter 7

Replanning

Replanning is a fundamental capability in Pick-and-Place tasks, enabling the system to dynamically adapt in the presence of execution failures, such as missed grasps, unexpected obstacles, or changes in object positions. In real-world environments, uncertainty is a constant factor, and rigid behavior often leads to task failure. Replanning allows the robot to recompute both task-level and motion-level strategies in order to recover from such failures and robustly achieve its objectives.

When designing a replanning strategy, two key questions must be addressed:

- **When** to evaluate whether a task has failed;
- **What** to do in case of failure.

As discussed throughout this report, the planning pipeline from symbolic state to physical execution has been structured to support real-time monitoring of the environment. In particular, the use of the `PoseRelayPlugin`, which publishes the real pose of objects on the `/object_pose` topic, enables feedback on the actual geometric state of the world.

Leveraging this mechanism, a replanning strategy was implemented to handle cases in which the task execution diverges from expectations. Upon detecting such discrepancies, the symbolic state is updated to reflect the real-world conditions, and a new plan is computed accordingly. This ensures that the system can autonomously recover from failures and continue executing the task with resilience and adaptability.

7.1 Replanning Strategy

To implement a replanning strategy, the execution logic of the `mtc_node` — responsible for running the MoveIt Task Constructor — was revised. Specifically, its lifecycle is now managed by a separate node, called `mtc_node_manager`, which handles process supervision and failure recovery. Upon startup, the manager spawns the `mtc_node`, which then executes the Pick-and-Place tasks as described in previous chapters. The main modification lies in the insertion of a `checkState` method, which is invoked before executing each pair of Pick-Place actions in the plan. This method triggers a call to the blocking service `check_states`, exposed by the manager. The request itself is empty and simply signals that it is time to verify whether replanning is necessary. If the service returns `false`, it indicates that the original plan is no longer valid. In this case, the current `mtc_node` is terminated, a new plan is computed by re-invoking the planner, and the `mtc_node` is respawned with the updated plan.

The replanning condition checks whether the symbolic state expected by the planner at the current simulation step matches the actual symbolic state derived from the environment. This check proceeds as follows:

1. The real-time geometric state of the world is retrieved from the `/object_pose` topic and converted into a symbolic state using the same format employed by the planner.
2. The expected symbolic state at the i-th step of the plan is retrieved.
3. A comparison is performed. If the two states match, execution resumes. Otherwise, the replanning procedure is triggered as described.

In particular, with regard to step (2), the expected symbolic state is computed directly within the planner. Specifically, the PyHOP planning algorithm updates the symbolic state during the planning process by applying the transition function $\gamma(s, a)$, which computes the next state resulting from applying action a in state s. As each action is selected during the planning phase, the planner incrementally builds a sequence of expected states by successively applying γ. This sequence is stored in a vector and then published on the topic `/expected_states`, where each entry corresponds to the symbolic state the system is expected to be in at a given step in the plan. This design is general and domain-independent, as it relies on the same γ function used in any domain definition compatible with the SHOP paradigm. Therefore, switching to a different planning domain still provides access to the same sequence of expected symbolic states, represented as strings and available for comparison at execution time.

This evaluation is performed after each Pick-Place task is completed. Notably, during task execution, the system does not monitor failure in real time. For example, if the grasping stage fails, the system will still attempt to perform the place action. Only when execution of the next task is about to begin the manager does assess whether the previous one succeeded, and if not, initiates a new planning cycle.