Task and Motion Planning Framework for Robotic Manipulation Using PDDL and OMPL Integration

Sahil Gajera[1], Ashish Sukumar[2], Anirudh Nallawar[3], Rajdeep Banerjee[4]
RBE550 Motion Planning - Team 5
Robotics Engineering Department
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
[1]sgajera@wpi.edu, [2]asukumar1@wpi.edu, [3]arnallawar@wpi.edu, [4]rbanerjee1@wpi.edu

***Abstract*—This work presents a modular Task and Motion Planning (TAMP) framework that integrates symbolic reasoning with geometric motion planning for robotic manipulation. The system combines predicate-based scene abstraction, PDDL task planning using Pyperplan, and collision-free trajectory generation through OMPL, unified under a closed-loop re-grounding architecture. We implement the framework in a Blocksworld manipulation domain using a Franka Emika Panda robot in the Genesis physics simulator. The proposed pipeline supports robust pick-and-place, stacking, and spatial arrangement tasks through motion primitives grounded in continuous state estimation and validated through integrated collision checking. We demonstrate the system across progressively complex goals, including multi-tower construction, a 5-block tower, dynamic tallest-tower assembly with failure recovery, a two-layer pentagon structure, and a directional-adjacency grid arrangement. Experimental results show that the framework reliably synthesizes symbolic plans, executes feasible motion trajectories, and recovers from disturbances through iterative re-planning. The results highlight the effectiveness of combining discrete planning with continuous control for complex manipulation tasks and provide a foundation for extending TAMP systems toward more physically demanding, contact-sensitive structures.**

***Index Terms*—Task and Motion Planning, PDDL, OMPL, Robotic Manipulation, Symbolic Planning, Motion Primitives**

I. Introduction

Task and Motion Planning (TAMP) addresses one of the fundamental challenges in robotics: bridging the gap between high-level task specifications and low-level motion execution. While symbolic planners excel at reasoning about discrete action sequences, they lack the geometric reasoning necessary for physical manipulation. Conversely, motion planners can generate collision-free trajectories but cannot reason about task-level goals and dependencies.

This work presents a TAMP framework that combines:

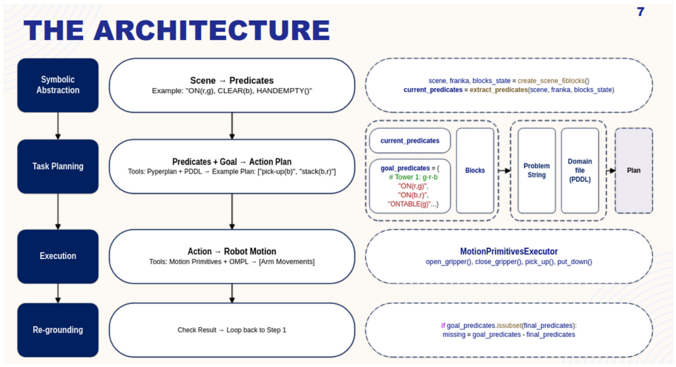
- **Symbolic abstraction** through predicate extraction from scene state
- **Task planning** using PDDL (Planning Domain Definition Language)
- **Motion planning** via OMPL (Open Motion Planning Library)
- **Re-grounding** mechanisms for error detection and recovery

The system is implemented for a blocksworld manipulation domain using a Franka Emika Panda robot in the Genesis physics simulator.

II. System Architecture

A. Overview

The TAMP architecture comprises four interconnected stages operating within a closed-loop control paradigm, as illustrated in Figure 1. This hierarchical decomposition enables seamless integration of high-level symbolic reasoning with low-level geometric motion planning.


THE ARCHITECTURE
7
Symbolic Abstraction
Task Planning
Execution
Re-grounding
Scene → Predicates
Example: "ON(r,g), CLEAR(b), HANDEMPTY()"
Predicates + Goal → Action Plan
Tools: Pyperplan + PDDL → Example Plan: ["pick-up(b)", "stack(b,r)"]
Action → Robot Motion
Tools: Motion Primitives + OMPL → [Arm Movements]
Check Result → Loop back to Step 1
current_predicates
Blocks
Problem String
Domain file (PDDL)
Plan
MotionPrimitivesExecutor

```
scene, franka, blocks_state = create_scene_6blocks()
current_predicates = extract_predicates(scene, franka, blocks_state)
```

```
goal_predicates = {
    # Tower 1: g-r-b
    "ON(r,g)",
    "ON(b,r)",
    "ONTABLE(g)"...}
```

```
open_gripper(), close_gripper(), pick_up(), put_down()
```

```
if goal_predicates.issubset(final_predicates):
missing = goal_predicates - final_predicates
```

Fig. 1: Comprehensive architecture of the TAMP pipeline showing the four-stage workflow.

Stage 1: Symbolic Abstraction - The current scene state is converted into a set of logical predicates describing object relationships and gripper state.

Stage 2: Task Planning - A PDDL problem file is generated from current and goal predicates, then solved using Pyperplan to produce a high-level action sequence.

Stage 3: Execution - Each symbolic action is grounded into concrete motion primitives and executed using OMPL for path planning.

Stage 4: Re-grounding - After execution, predicates are re-extracted to verify goal achievement and detect failures, enabling loop-back to Stage 1 if necessary.

B. Predicate System

The symbolic abstraction layer defines five core predicates for the blocksworld domain:

- `ON(block_a, block_b)`: block_a is stacked on block_b
- `ONTABLE(block)`: block is resting on the table surface
- `CLEAR(block)`: no object is on top of block
- `HOLDING(block)`: gripper is currently grasping block
- `HANDEMPTY()`: gripper is not holding any object

These predicates capture the complete state space necessary for blocksworld task planning while abstracting away continuous geometric details.

C. Predicate Extraction Algorithm

The `extract_predicates()` function implements the grounding from continuous to discrete state:

Algorithm 1 extract_predicates

```
1: function EXTRACT_PREDICATES(scene, franka, blocks)
2:     preds ← ∅
3:     positions ← block → block.pos
4:     hand ← franka.hand.pos
5:     holding ← NONE
6:     for each do (b, p) in positions //block_id and position
7:         if dist(p, hand) < th then
8:             holding ← b; add HOLDING(b) to preds
9:             break
10:        end if
11:    end for
12:    if holding = NONE then
13:        add HANDEMPTY() to preds
14:    end if
15:    for each do (b, p) in positions
16:        if b = holding then
17:            continue
18:        end if
19:        if (p.z − TABLE_Z) < STACK_TOL then
20:            add ONTABLE(b) to preds
21:        end if
22:        for each do (o, q) in positions
23:            if b = o then
24:                continue
25:            end if
26:            if (p.z − (q.z + BLOCK_SIZE)) < STACK_TOL and dist_xy(p,q) < XY_TOL then
27:                add ON(b, o) to preds; break
28:            end if
29:        end for
30:    end for
31:    return preds
32: end function
```

Key parameters include:

- `BLOCK_SIZE = 0.04m`: vertical extent of blocks
- `TABLE_Z = 0.02m`: table surface height
- `STACK_TOLERANCE = 0.015m`: vertical alignment threshold
- `XY_TOLERANCE = 0.02m`: horizontal alignment threshold

III. Task Planning Layer

A. PDDL Problem Generation

The task planning module generates PDDL problem specifications from the extracted predicates:

Algorithm 2 generate_pddl_problem

```
1: function GENERATE_PDDL_PROBLEM(preds, goals, blocks, name)
2:     function FORMAT_PRED(p)
3:         p ← lowercase(p)
4:         if p has no '(' then return p
5:         end if
6:         pred ← text before '('
7:         args ← text inside '(' ')' split by ','
8:         if args not empty then
9:             return (pred args)
10:        else
11:            return (pred)
12:        end if
13:    end function
14:    init ← join(format_pred(p) for p in preds)
15:    goal ← join(format_pred(g) for g in goals)
16:    problem ← PDDL template filled with:
17:        objects = blocks
18:        init = init
19:        goal = goal
20:    return problem
21: end function
```

B. Pyperplan Integration

The system interfaces with Pyperplan, a Python-based PDDL planner, through subprocess calls:

Algorithm 3 call_pyperplan

```
1: function CALL_PYPERPLAN(domain_file, problem_string)
2:     create temporary PDDL file and write problem_string
3:     problem_file ← temp filename
4:     run pyperplan with {domain_file, problem_file}
5:     if planner failed then
6:         return empty list
7:     end if
8:     solution_file ← problem_file + ".soln"
9:     plan ← empty list
10:    for each line in solution_file do
11:        if line is of form "( ... )" then
12:            extract action name and arguments
13:            append (action, args) to plan
14:        end if
15:    end for
16:    delete temporary problem_file
17:    return plan
18: end function
```

The planner returns a sequence of grounded actions such as: `(pick-up r)`, `(stack r g)`, `(unstack b r)`, `(put-down b)`

IV. Motion Planning Layer

A. OMPL Integration

The `PlannerInterface` class provides a bridge to OMPL for generating collision-free trajectories:

Algorithm 4 plan_path

```
1:  function PLAN_PATH(q_goal, q_start, timeout, smooth, N, obj, planner)
2:      if q_start is None then
3:          q_start ← robot.get_qpos()
4:      end if
5:      create OMPL state space of size $n$
6:      set joint limits as bounds
7:      ss ← SimpleSetup(space)
8:      attached_object ← obj
9:      set validity checker to collision function
10:     set planner type (e.g., RRTConnect)
11:     create start and goal states from q_start and q_goal
12:     ss.setStartAndGoalStates(start, goal)
13:     ss.setup()
14:     solved ← ss.solve(timeout)
15:     waypoints ← empty list
16:     if solved then
17:         path ← ss.getSolutionPath()
18:         if smooth then
19:             ss.simplifySolution()
20:         end if
21:         path.interpolate(N)
22:         waypoints ← convert OMPL states to tensors
23:     end if
24:     return waypoints
25: end function
```

B. Collision Checking

The state validity checker ensures collision-free configurations:

Algorithm 5 State Validity Checking

```
1:  function IS_STATE_VALID(state)
2:      q ← convert OMPL state to tensor
3:      robot.set_qpos(q)
4:      collisions ← robot.detect_collision()
5:      if collisions is empty then
6:          return True
7:      end if
8:      if attached_object is None then
9:          return False
10:     end if
11:     return collision_with_attached(collisions)
12: end function
```

Algorithm 6 State Validity Checking with attached object

```
1:  function COLLISION_WITH_ATTACHED(collisions)
2:      finger_links ← {left_finger, right_finger, hand}
3:      for each pair $(a, b)$ in collisions do
4:          name_a ← link name of $a$
5:          name_b ← link name of $b$
6:          if (name_a in finger_links AND $b$ is attached object) then
7:              continue
8:          else if (name_b in finger_links AND $a$ is attached object) then
9:              continue
10:         else
11:             return False
12:         end if
13:     end for
14:     return True
15: end function
```

This implementation allows intentional collisions between the gripper and held objects while rejecting all other collisions.

C. Supported Planners

The system supports multiple OMPL planning algorithms:

- **RRTConnect**: Bidirectional RRT (default)
- **RRT**: Rapidly-exploring Random Tree
- **RRT***: Asymptotically optimal RRT
- **PRM**: Probabilistic Roadmap
- **EST**: Expansive Space Trees
- **FMT**: Fast Marching Tree
- **BITstar**: Batch Informed Trees
- **ABITstar**: Advanced BIT*

V. Motion Primitives

The system implements four core motion primitives that serve as the grounding for symbolic actions:

A. Gripper Control

`open_gripper()` and `close_gripper()` control the parallel jaw gripper by commanding target joint positions for the finger joints.

B. Pick Operation

`pick_up(block, offset)` performs a vertical pick-and-place motion:

1) Move to pre-grasp pose above block
2) Open gripper
3) Descend to grasp height
4) Close gripper
5) Lift to clearance height

C. Place Operation

`put_down(position)` places the held object at a target location through reverse motion sequence.

VI. Diagnostic and Error Handling

A. Bounds Violation Detection

The system provides detailed diagnostics for planning failures:

Algorithm 7 diagnose_bounds_violation

```
 1: function DIAGNOSE_BOUNDS_VIOLATION(si, state)
 2:     violations ← empty list
 3:     for each joint index $i$ do
 4:         val ← state[i]
 5:         (low, high) ← bounds of joint $i$
 6:         if val < low or val > high then
 7:             add $(i, val, low, high)$ to violations
 8:         end if
 9:     end for
10:     log warning about violations
11: end function
```

B. Collision Diagnosis

When states are invalid due to collisions, the system identifies problematic links:

Algorithm 8 diagnose_valid_violation

```
 1: function DIAGNOSE_VALID_VIOLATION(state)
 2:     robot.set_qpos(convert state to tensor)
 3:     collisions ← robot.detect_collision()
 4:     if collisions not empty then
 5:         bad_links ← empty set
 6:         for each pair $(a, b)$ in collisions do
 7:             add link names of $a$ and $b$ to bad_links
 8:         end for
 9:         log warning with sorted(bad_links)
10:     end if
11: end function
```

VII. System Workflow

The complete TAMP loop operates as follows:

1) **Scene Observation**: Extract current state from Genesis simulator
2) **Predicate Extraction**: Convert continuous state to symbolic predicates using geometric thresholds
3) **PDDL Problem Generation**: Create problem file with current state as initial conditions and desired configuration as goal
4) **Task Planning**: Invoke Pyperplan to compute action sequence
5) **Action Grounding**: For each symbolic action:
   - Compute target configuration
   - Call OMPL to plan collision-free path
   - Execute motion in simulator
   - Update attached object state if grasping
6) **Verification**: Re-extract predicates and compare with goal
7) **Re-planning**: If goal not achieved, return to step 3

VIII. Implementation Details

A. Robot Adapter Pattern

The `_ensure_adapter()` function wraps the raw Genesis robot entity to provide a stable interface:

Algorithm 9 ensure_adapter

```
1: function ENSURE_ADAPTER(robot, scene)
2:     if robot is instance of RobotAdapter then
3:         return robot
4:     else
5:         return RobotAdapter(robot, scene)
6:     end if
7: end function
```

This pattern maintains backward compatibility while enabling enhanced functionality through the adapter.

B. Type Conversions

Careful handling of data types ensures compatibility between Python, NumPy, PyTorch, and C++ OMPL:

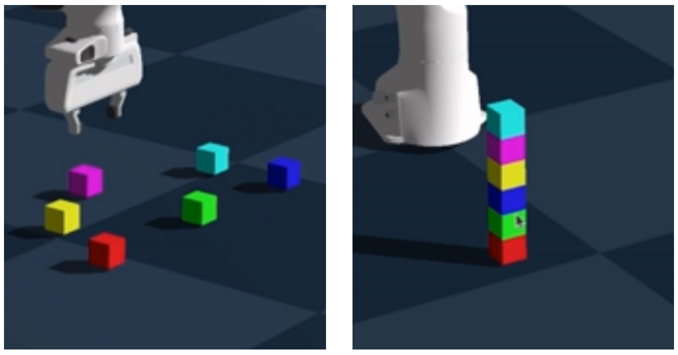
Algorithm 10 Prepare Bounds and Convert State

```
 1: q_lower ← convert robot.q_limit[0] to float array
 2: q_upper ← convert robot.q_limit[1] to float array
 3: for each joint $i$ do
 4:     bounds.setLow(i, q_lower[i])
 5:     bounds.setHigh(i, q_upper[i])
 6: end for
 7: tensor ← empty tensor of size n_joints
 8: for each joint $i$ do
 9:     tensor[i] ← state[i]
10: end for
```

IX. Goal 1 - Two Towers

The goal was achieved in two scenarios (also shown in Figure 2) :

1) Scattered
2) Pre-stacked



(a) Scattered (b) Stacked

Fig. 2: Initial and final block configurations for Goal 1.

A. Objective

The objective was to arrange six blocks into two 3-block towers using the **TAMP pipeline**, starting from an initially scattered configuration. The target towers are:

- **Tower 1:** GREEN → RED → BLUE
- **Tower 2:** MAGENTA → YELLOW → CYAN

The challenge arises from the blocks being randomly scattered on the table, requiring robust task and motion planning to achieve the goal.

B. Execution Details

- **Move to Safe Home** The robot first moves to a predefined safe home configuration to avoid collisions with the environment or blocks. This is performed using linear interpolation in joint space over multiple steps.
- **TAMP Loop** The TAMP loop iteratively executes the following steps until the goal is achieved or a maximum number of iterations is reached:
  1) **Predicate Extraction**
  2) **Task Planning**
  3) **Action Execution** Execute the first action of the plan using motion primitives:
     - `pick-up(block)`
     - `put-down(block)`
     - `stack(block, target)`
     - `unstack(block, target)`

     The `unstack(block, target)` was only used in re-stacked condition, to unstack block first and simultaneously arrange the two towers
  4) **Re-grounding**
  5) **Re-planning**

  The goal predicates were given as:

```
# Tower 1: GREEN → RED → BLUE
ONTABLE(g)
ON(r,g)
ON(b,r)
CLEAR(b)

# Tower 2: MAGENTA → YELLOW → CYAN
ONTABLE(m)
ON(y,m)
ON(c,y)
CLEAR(c)

# Gripper
HANDEMPTY()
```

- **Motion Execution Strategy** To ensure stable block placement, the motion primitive execution follows a **separated movement strategy**:
  1) Move to the target XY position while keeping Z and orientation constant.
  2) Rotate the end-effector to the desired orientation (rotation around Z-axis).
  3) Move along the Z-axis to place the block precisely.

  Gripper control is applied appropriately: closing during pick-up and opening during put-down. Linear interpolation is used for smooth and stable motion.

C. Result:

The TAMP pipeline successfully achieved the goal of constructing two 3-block towers from an initial configurations.

- **Tower 1:** GREEN → RED → BLUE
- **Tower 2:** MAGENTA → YELLOW → CYAN

The result is shown in Figure 3 below:

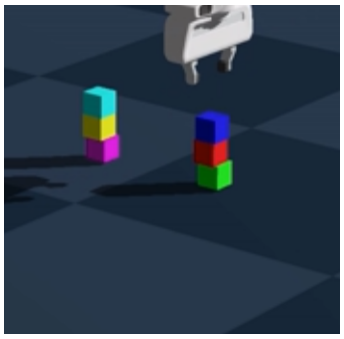


Fig. 3: Goal 1 Result

X. Goal 2 - 5 Block Tower

The goal was achieved in two scenarios (also shown in Figure 2 which is same as goal 1 scenario) :

1) Scattered
2) Pre-stacked

A. Objective

The objective of this task is to build a single 5-block tower from six scattered blocks on the table using the TAMP pipeline. The target tower configuration is:

- **Tower:** GREEN → RED → BLUE → YELLOW → MAGENTA (from bottom to top)
- Cyan (C) remains on the table

B. Execution Details

The execution pipeline is identical to Task 1 (Goal 1: Two 3-block towers). The robot follows the same **TAMP pipeline** with the same motion primitives and iterative re-planning strategy:

1) **Symbolic Abstraction**
2) **Task Planning**
3) **Action Execution**
4) **Re-grounding**
5) **Re-planning**

The only difference from Goal 1 is the **goal predicates**, which now define a single 5-block tower:

```
# Tower : GREEN → RED → BLURE → YELLOW
→ MAGENTA
ONTABLE(g)
ON(r,g)
ON(b,r)
ON(y,b)
ON(m,y)
CLEAR(m)

#Condition for remaining block
ONTABLE(c)
CLEAR(c)

#Gripper
HANDEMPTY()
```

All other aspects, including motion strategy, gripper control, and sequential XY-rotation-Z placement, remain the same as in Goal 1.

C. Motion Strategy

- Blocks are moved in a sequential manner: XY position first, then rotation, and finally Z-axis placement.
- The gripper closes during pick-up and opens during put-down.
- Cyan block is placed at a designated side position since it is not part of the tower.

D. Result:

The final 5-block tower configuration achieved is (also shown in Figure 4 below):

- GREEN → RED → BLUE → YELLOW → MAGENTA
- Cyan remains on the table

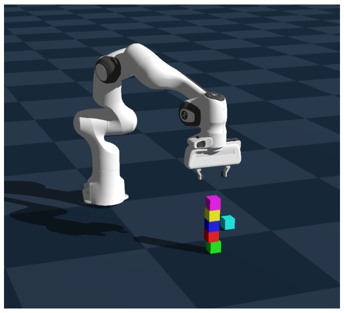


Fig. 4: Goal 2 Result

XI. Goal 3: Tallest Tower - Scattered Scenario

A. Objective

The objective of this task is to build the tallest possible tower using all 10 blocks available in the scene. The blocks start scattered on the table and the robot should stack them into a single tower. The robot must autonomously re-plan if any blocks fall during the stacking process.

Starting Configuration: 10 scattered blocks: r, g, b, y, o, r2, g2, b2, y2, o2

Goal: A single tower containing all 10 blocks stacked vertically.

B. Execution Details

Execution pipeline follows the same TAMP loop described in Section VII:

1) **Symbolic Abstraction**
2) **Task Planning**
3) **Action Execution**
4) **Re-grounding**
5) **Re-planning**

Incremental Strategy:

- Blocks are sorted by proximity to the center of the table (closest first).
- The tower is built incrementally, adding one block at a time.
- Stability is verified after each stack action. If a block falls, the robot re-plans using the updated predicates.

C. Goal Predicates

The incremental goal predicates for the tallest tower are dynamically updated during execution:

```
# Base block on table
ONTABLE(base_block)
HANDEMPTY()

# ON relationships for blocks in current tower
ON(block_i, block_i-1)
CLEAR(top_block)
```

D. Results

After execution, the robot successfully built a tower containing the following blocks in order: r, g, b, y, o, r2, g2, b2, y2, o2 (or the maximum achievable within stability constraints).

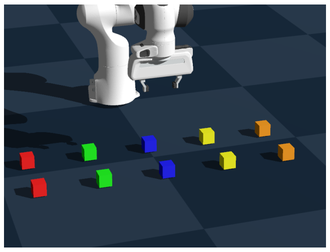
Tower Metrics:

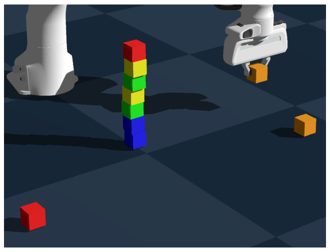
- Base block: `<base_block>`
- Number of blocks stacked: `<tower_height>`
- Physical tower height: `<physical_height> m`

Observations:

- Incremental building ensured higher stability for taller towers.
- The robot autonomously re-planned whenever a block fell, maintaining robustness.
- Target goal (7 blocks stacked) was achieved.
- Attempting to stack the 8th block caused the tower to topple.
- The robot's re-planning mechanism attempted to recover but the tower could not maintain stability for more than 7 blocks.

E. Visualization


(a) Initial scattered scenario


(b) Final tallest tower

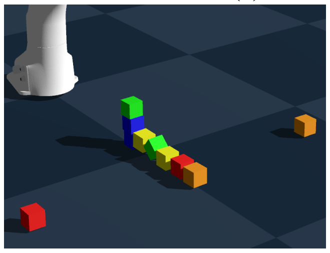

(c) Toppled tallest tower

Fig. 5: Goal 3: Tallest Tower - Before ,Tallest stack and After collapse

XII. Goal 4.1: Pentagon Tower

Objective: Construct a two-layer *pentagon tower* using ten blocks as shown in Figure 6.

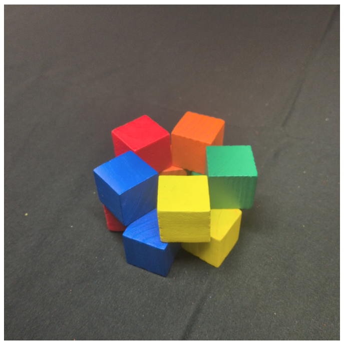


Fig. 6: Special Goal 4.1

A. Approach Overview

The task was completed via a two- phase TAMP pipeline:

- **Base:** In Phase 1, five blocks (`b1--b5`) are arranged into a tight pentagon on the table.
- **Top layer:** Given a stabilized base, in Phase 2, the remaining five blocks (`b6--b10`) are placed as a rotated pentagon that *bridges* the base, such that each top block rests across two neighboring base blocks.

The geometry of each slot is deterministic and precomputed, while the symbolic planner determines the ordering of pick-and-place actions. This cleanly separates *what* to do (task planning) from *how* to do it (motion execution).

Separating the phases reduces the symbolic search space per phase, ensures the base is fully stabilized before stacking, and allows the motion layer to use specialized trajectories for delicate top placements.

B. Deterministic Geometry via Slot Abstraction

The symbolic domain introduces explicit slot predicates that bind discrete planning goals to fixed geometric targets:

- `BASE-LOC(base1..base5)` represent the five vertices of the base pentagon.
- `TOP-LOC(top1..top5)` represent the five bridge locations, obtained by rotating the base pentagon by 36°.
- `AT(block, slot)` binds a block to a specific geometric target.

Each slot is deterministically grounded into a target pose:

$$(x, y, \theta)_{\text{slot}} = \begin{cases} \text{PentagonVertex}(i; R, 0^\circ) & \text{for base slots,} \\ \text{PentagonVertex}(i; R, 36^\circ) & \text{for top slots,} \end{cases}$$

where R is a tuned pentagon radius and θ is the desired wrist yaw for placement.

C. Symbolic Planning Model

The planning domain extends classical Blocksworld with placement actions that explicitly reference slots:

- `pick-up(b)`: grasp a clear block.
- `put-down-base(b, baseSlot)`: place a block at a base pentagon slot.
- `put-down-top(b, topSlot, baseRef)`: place a block at a bridged top slot above the base layer.

Phase-1 goal:

$$\bigwedge_{i=1}^{5} (\text{ONTABLE}(b_i) \wedge \text{AT}(b_i, \text{base}_i) \wedge \text{CLEAR}(b_i)) \wedge \text{HANDEMPTY}()$$

Phase-2 goal:

$$\bigwedge_{i=6}^{10} \big(\text{ON}(b_i, b_{i-5}) \wedge \text{AT}(b_i, \text{top}_{i-5}) \wedge \text{CLEAR}(b_i)\big) \wedge \text{HANDEMPTY}()$$

D. Planning Algorithm - Symbolic

We used **A* search** with the **FF heuristic**. A* improves reliability for cluttered goal conditions (multiple adjacency constraints) by prioritizing states estimated to be closer to the goal, while the FF heuristic provides strong guidance for classical planning problems with interacting predicates.

E. Planning Algorithm - Motion

Each symbolic action is grounded into a continuous trajectory using OMPL, with **RRTConnect** as the default planner for collision-free joint-space paths. When OMPL is unavailable, the system falls back to safe joint interpolation.

F. Execution Strategy for Bridged Placement

Bridged top placements are executed using a conservative, structured motion:

1) Lift the block to a safe height.
2) Translate horizontally to a point above the target slot.
3) Descend vertically toward the placement height.
4) Release the block and retreat vertically.

This strategy minimizes lateral contact with adjacent base blocks. Additionally, each slot encodes a target wrist yaw to maintain consistent gripper alignment during approach and reduce unintended collisions.

G. Observed Failure Modes and Limitations

Despite the structured two-phase formulation and multiple execution-level mitigations, Goal 4.1 remains a **low-margin and failure-prone task**. In practice, the success rate for stacking all five top blocks is limited, and failures are not uniformly distributed across blocks.

Most notably, the **green top block** consistently exhibits a higher failure rate than the others. Across repeated runs, this block frequently slides or falls off shortly after placement, even when symbolic predicates such as `ON` and `AT` are momentarily satisfied. This behavior persists despite deterministic slot geometry, tuned offsets, cautious approach trajectories, and relaxed symbolic tolerances.

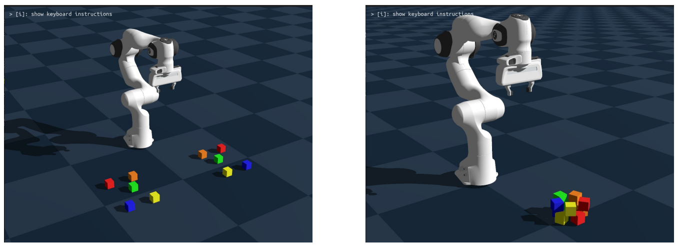
The primary cause is geometric rather than algorithmic. The green block's bridge location lies closest to the *stability boundary* of its two supporting base blocks, placing its center of mass near the edge of the support polygon. As a result, the block is highly sensitive to:

- small lateral impulses during gripper release,
- minor physics integration noise in the simulator,
- and subtle contact forces induced by neighboring blocks during placement.

These effects compound during Phase 2, where stacking occurs in close proximity to already-placed blocks, increasing the likelihood of unintended contact and drift.

H. Result

Overall, Goal 4.1 demonstrates a principled two-phase TAMP formulation for a contact-sensitive structure with tight geometric constraints. While deterministic geometry and symbolic planning enable consistent base construction, the bridged top layer exposes fundamental physical limits of the configuration. The observed reliability issues highlight the inherent difficulty of marginally stable stacking tasks and motivate future extensions such as active stabilization, compliance-aware placement, or closed-loop correction during the final settling phase.



(a) Initial scenario (b) Resulting stack

Fig. 7: Goal 4: Structure Before and After

XIII. Goal 4.2: Directional Adjacency TAMP (3 Red + 3 Green)

Objective: Construct a deterministic 2×2 base grid using *directional* adjacency constraints, then stack the remaining two blocks while preserving the base layout as shown in Figure 8 below:

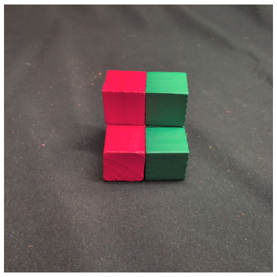


Fig. 8: Special Goal 4.2

A. Approach Overview

The task was completed via a two-phase TAMP pipeline (similar to the approach shown in section XII:

- **Base Construction:** Concretely, Phase 1 forms the grid

  $$[r2][g2] \text{ (back row)}$$
  $$[r1][g1] \text{ (front row)}$$

  using `ADJACENT-X` (left/right) and `ADJACENT-Y` (front/back)
- **Stacking:** Phase 2 stacks `r3` on `r1` and `g3` on `g1` while keeping the base adjacency predicates satisfied.

Goal 4.2 extends the standard Blocksworld formulation by explicitly modeling *directional* adjacency:

- `ADJACENT-X(a,b)`: block a is to the **right** of b (+X)
- `ADJACENT-Y(a,b)`: block a is **in front** of b (+Y)

This is more expressive than a generic `ADJACENT(a,b)` predicate because it removes ambiguity in placement and yields **precise, deterministic** geometric targets for motion execution.

B. Uniqueness of the Two-Phase Strategy

The Phase separation is crucial because base formation is predominantly a **2D spatial arrangement problem**, whereas stacking introduces additional **vertical (3D) constraints** and a higher collision risk near adjacent blocks. By isolating these subproblems, the planner searches a smaller, cleaner state space per phase and the motion layer can use tighter, more reliable placement targets.

C. Directional Predicate Extraction

The symbolic abstraction layer detects adjacency directionally by comparing block centers on the same Z-layer:

- Only block pairs with $|z_a - z_b| < \tau_{layer}$ are considered (**same layer**).
- `ADJACENT-X(a,b)` holds when $x_a > x_b$, $|y_a - y_b|$ is small (same row), and $|x_a - x_b|$ is within a **block-width window**.
- `ADJACENT-Y(a,b)` holds when $y_a > y_b$, $|x_a - x_b|$ is small (same column), and $|y_a - y_b|$ is within a **block-width window**.

This grounding enables the task planner to reason about *where* a block must go (right-of vs front-of), while the motion layer translates that discrete requirement into a concrete (x, y) target offset by one block length.

D. Planning Algorithm Choice - Symbolic

We used the same strategy used in section XII i.e., A* search with the FF heuristic.

1) Planning Algorithm Choice - Motion : We used same strategy used in section XII i.e., RRTConnect.

E. Execution Details: Deterministic Placement + Collision Avoidance

Goal 4.2 implements two practical execution enhancements:

- Deterministic adjacent placement. Directional actions `put-down-adjacent-x` and `put-down-adjacent-y` map to deterministic offsets:

$$(x, y)_{\text{target}} = \begin{cases} (x_{ref} + s,\ y_{ref}) & \text{for } +X \\ (x_{ref} - s,\ y_{ref}) & \text{for } -X \\ (x_{ref},\ y_{ref} + s) & \text{for } +Y \\ (x_{ref},\ y_{ref} - s) & \text{for } -Y \end{cases}$$

  where $s \approx$ `BLOCK_SIZE` (with small tolerance). This makes the base grid formation reproducible across runs.
- Smart wrist rotation for collision avoidance When blocks exist adjacent in the approach direction, the gripper rotates by 90° so the finger opening direction avoids clipping neighboring blocks. Specifically, if blocks are detected in $\pm Y$ near the target, the wrist rotates so the fingers align with X during approach and grasp.

F. Failure Modes Observed and Recovery via Replanning

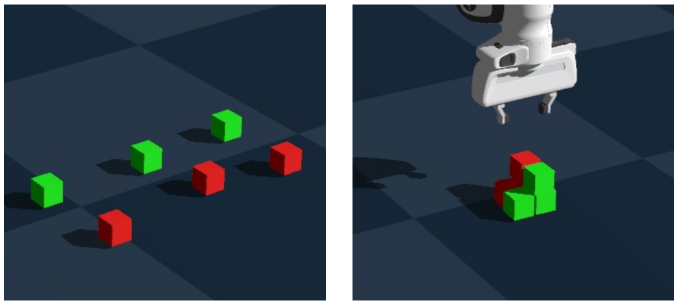
A frequent real-world failure arises during stacking in Phase 2: when a top block is lowered onto its target tower, it may **collide with a neighboring adjacent base block** due to tight clearances and small physics deviations. This can prevent achieving predicates such as:

- `ON(r3,r1)` or `ON(g3,g1)` (stacking fails to settle)
- base adjacency predicates no longer hold after contact-induced drift

To handle this, the system performs **re-grounding** after execution and triggers **re-planning** if the phase goal is not satisfied (up to a fixed number of attempts). Importantly, we execute a *full plan once per phase* (no step-by-step replanning), and only replan when the final goal check fails.

G. Result

Goal 4.2 successfully demonstrates that augmenting Blocksworld with **directional adjacency predicates** enables stable, repeatable base construction, while the two-phase decomposition plus replanning-on-failure provides reliable completion even in the presence of stacking-time collisions and simulator noise.



(a) Initial scenario (b) Resulting stack

Fig. 9: Goal 4: Structure Before and After

XIV. Team Contributions and Timeline

A. Division of Responsibilities

All team members contributed equally to the project's success, with each member taking ownership of specific goals while collaborating on core system components:

- **Sahil Gajera**: Led implementation of Goal 1 (Two Towers) and Goal 4.1 (Pentagon Tower). Developed motion primitives for pick-and-place operations and contributed to the geometric calculation framework for pentagon slot positioning. Assisted in system integration and OMPL wrapper refinement.
- **Rajdeep Banerjee**: Led implementation of Goal 2 (5-Block Tower) and Goal 3 (Tallest Tower Challenge), including task planning strategy and execution sequencing. Contributed to PDDL domain modeling and predicate extraction algorithms. Coordinated system testing and debugging efforts across all goals.
- **Anirudh Nallawar**: Led implementation of Goal 3 (Tallest Tower Challenge) with dynamic failure recovery mechanisms. Played a key role in debugging and troubleshooting throughout the project, particularly in resolving joint limit violations and collision detection issues. Contributed to error handling and diagnostic tools.

- **Ashish Sukumar**: Led implementation of Goal 4.1 (Pentagon Tower) and Goal 4.2 (Directional Adjacency Grid). Designed and implemented the extended PDDL domain with directional predicates. Developed the two-phase planning strategy for spatial arrangement tasks and contributed to Pyperplan integration.

All team members collaborated on:

- Core system architecture and design decisions
- Predicate extraction and symbolic abstraction pipeline
- Motion planning integration with OMPL
- Documentation, video preparation, and final report

B. Project Timeline

The project spanned approximately 6 weeks from November 6 to December 12, following this schedule:

TABLE I: Development Timeline and Milestones

Phase	Timeline
Team formation & Genesis setup	Nov 1-6
Scene initialization & robot control	Nov 7-10
Basic motion primitives (hardcoded)	Nov 11-15
Goal 1, 2, 3 - Initial implementation (hardcoded, no TAMP)	Nov 16-23
Interim Report Submission	*Nov 24*
Post-Interim: Full TAMP Integration	
Predicate extraction & symbolic abstraction	Nov 25-27
PDDL domain design & Pyperplan integration	Nov 28-30
Goals 1-3 re-implementation with proper TAMP	Dec 1-3
Goal 4.1 (Pentagon Tower)	Dec 4-6
Goal 4.2 (Directional Adjacency Grid)	Dec 7-8
System refinement, debugging & testing	Dec 9-11
Final Report & Presentation	*Dec 12*

The project followed a two-phase development approach: initial prototyping with hardcoded solutions (Goals 1-3) for the interim report, followed by complete TAMP integration with symbolic planning, predicate-based abstraction, and closed-loop re-grounding for all goals.

C. Effort Distribution

Each team member contributed approximately **45-50 person-hours** to the project, for a total of approximately **180-200 person-hours**. The effort was distributed across:

- **Planning and design**: 3-4 hours per person (system architecture, domain modeling, component design)
- **Core implementation**: 10-12 hours per person (predicates, PDDL, OMPL, motion primitives, integration)
- **Goal-specific implementation**: 12-14 hours per person (assigned goals, testing scenarios)
- **Debugging and refinement**: 16-18 hours per person (joint limits, stability, collisions, error handling)
- **Documentation**: 5-6 hours per person (report writing, video recording, README)

The collaborative nature of the project ensured all team members gained comprehensive understanding of the complete TAMP pipeline while developing specialized expertise in their assigned goal areas.

XV. Conclusion

This paper presented a complete TAMP framework integrating PDDL-based task planning with OMPL motion planning for robotic manipulation. The system successfully bridges symbolic and geometric reasoning through a layered architecture with explicit grounding and re-grounding mechanisms. The implementation demonstrates robust performance on blocksworld tasks.

XVI. Resources

Videos

https://drive.google.com/drive/folders/1PXzvTZmVpH08VqE7DXWY-XTInKLMhMbe?usp=drive_link

(If the link is not accessible, videos are provided in the zip)

GitHub

https://github.com/AnirudhN-30/RBE550_final_project.git

References

[1] Genesis Team, *Genesis Physics and Simulation Engine*. https://github.com/Genesis-Embodied-AI/Genesis

[2] A. Torralba et al., *Pyperplan: A Python-based Classical Planner*. https://github.com/aibasel/pyperplan

[3] I. A. Şucan, M. Moll, and L. E. Kavraki, “The Open Motion Planning Library,” *IEEE Robotics & Automation Magazine*, 2012. https://ompl.kavrakilab.org