A Physics-Based Simulated Robotics Testbed for Planning and Acting Research

Oscar Lima,[*1] **Martin Günther,**[*1] **Alexander Sung,**[1] **Sebastian Stock,**[1]
Marc Vinci,[1] **Amos Smith,**[1] **Jan Christoph Krause,**[1] **Joachim Hertzberg**[1,2]
[1]German Research Center for Artificial Intelligence (DFKI)
[2]Osnabrück University
{oscar.lima, martin.guenther, alexander.sung, sebastian.stock,
marc.vinci, amos.smith, jan_christoph.krause, joachim.hertzberg}@dfki.de

Abstract

In this paper we introduce the *mobipick_labs* environment, which represents an effort to bring the robotics and planning and acting research communities closer together by releasing a physics-based single robot simulator that closely resembles a real-life setup. This tool is designed for non-robotics experts, providing a user-friendly high-level Python API for easily interacting with a complex robotic system. The framework additionally includes a basic semantic and numeric environment representation that provides real-time knowledge in the form of "facts" that can be used to react to the execution status.

Introduction

Mobile robots and the typical environments in which they operate offer a variety of interesting challenges for AI planning and acting algorithms. These environments are often dynamic, involving other actors such as humans or other robots who may be acting in them. Moreover, they are partially observable and even when an object is within the camera's field of view, it can be occluded by other objects. Additionally, robot action outcomes are temporal, stochastic, with continuous action parametrization, and consume resources when executed. This results in a high uncertainty about the environment state, which is increased by errors inherent to noisy sensor data.

Note that many of those properties violate assumptions made by classical planners (Ghallab, Nau, and Traverso 2004). Dealing with those properties requires to relax some of those assumptions or to use sophisticated plan execution and monitoring algorithms. In recent years, the focus on deliberative acting has received more attention (Ghallab, Nau, and Traverso 2016) and workshops and robotics tracks at ICAPS have put this into focus. Nevertheless, there is still a relatively high entry barrier for planning researchers who find such applications interesting and would like to test their algorithms on real robots.

ROS (Quigley et al. 2009) is the de facto standard middleware in robotics. However, its multiple dependencies and its complexity require developers to spend several months to master its features. Researchers from the planning community typically have a background in AI and rarely have the necessary robotics knowledge to grasp the complexity of a robot acting in a real setup. Therefore, planning research sometimes uses idealized ad hoc simulations that abstract away many features of real robot domains. We argue that these simulators are too abstract, leaving a big gap between the output of a planner and execution on a real robot. To help reduce the gap between the planning and robotics community, we release a software contribution called *mobipick_labs*[1] that is suitable for researchers with little to no experience in robotics or ROS who want to work on planning, acting and monitoring problems in robotic domains.

The Mobipick robot is an indoor mobile manipulator (see Fig. 1) suitable for pick and place tasks with object perception in the loop due to a depth camera attached to the end effector. We have created a physics-based simulation of this robot using Gazebo[2] (Koenig and Howard 2004) and provide this along with other key software components, includ-

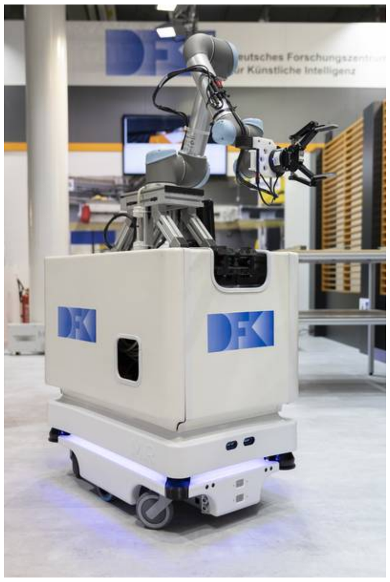

Figure 1: The Mobipick custom robotic mobile manipulator, consisting of a MiR base, UR5 arm and Robotiq gripper.

[*]These authors contributed equally.

[1]https://github.com/DFKI-NI/mobipick_labs
[2]https://gazebosim.org/

ing a custom *Mobipick Robot API* in Python that allows a Linux user to easily control the robot at a high level without the need of expert knowledge in robotics or ROS. Another provided module converts real-time sensor data into facts that can be used to reason about the situation at hand. The installation requires only a few steps before the user can begin using the system. Additionally, we provide a convenient GUI interface which allows the user to interact with the robot before commanding it via code.

While this paper focuses on the simulation aspect, we would like to stress that we use the same software stack in our labs for execution on the real robot. This ensures a close match between the simulated and real robot, and code that runs in the simulator can often run with little or no changes on the real robot.

Therefore, the main contribution of this paper is a simulated robotic mobile manipulator software stack, which is very close to a real robot and enables non-robotics expert users to test their AI planning and acting algorithms with simple Python commands.

Related Work

CraftBots (Nemiro et al. 2021) is a lightweight multi-agent team simulator to evaluate and benchmark integrated planning and execution algorithms. The simulation, while not physics-based, is focused on the high-level aspects suitable for complex planning and execution tasks.

In contrast, the task in our Mobipick simulation environment is much more detailed and close to the real world. Although our environment may at first glance mistakenly seem simple from a pure planning perspective, there exist many complexities in our robotic pick and place task due to partial observability, faulty sensor data, external events that can happen during execution, interaction with humans and coping with action failure.

VirtualHome (Puig et al. 2018) is a household simulator that features interactive objects, humanoid agents, multiple camera views, and concurrent multi-agent simulation. Previous research such as ProgPrompt (Singh et al. 2022) use it for experimentation with decision making using large language models (e.g. GPT-3). To our knowledge, robot simulation is not supported in VirtualHome although the aim of the MIT researchers behind it is to teach chores to robots.

While there are many other robotic simulations available in Gazebo, they rarely provide a high level wrapper to easily access functionalities with simple Python commands. Examples of easily accessible Gazebo simulations can be found on websites like The Construct or Amazon AWS RoboMaker, where one can run them via web browser and without the need of a local installation. Unlike these simulations, ours does require installation on a Linux based computer. However, we do provide automated scripts to ease the process.

The mobipick_labs System

The Mobipick robot can perform the tasks of autonomous navigation, 6DoF object detection and pose estimation, object anchoring (which assigns a persistent unique ID to all perceived objects), arm trajectory motion planning and execution, pick, place and insertion of objects including grasp planning, collision detection, and free space place sampling.

The software components of our system can be roughly grouped into three layers (see Fig. 2): the components that only run on either the real robot or in simulation; the higher-level components that are agnostic of which underlying system (real or sim) they run on as both underlying systems present the same ROS interfaces; and the deliberative layer, which controls the robot via an abstract robot API.

While all software components are in principle reusable for your own ROS system, the ones written for specific hardware components, e.g. the manipulation module, obviously only make sense with the according component being present in your system. A different skill set of your robot might require additions from your side to the robot_api – we

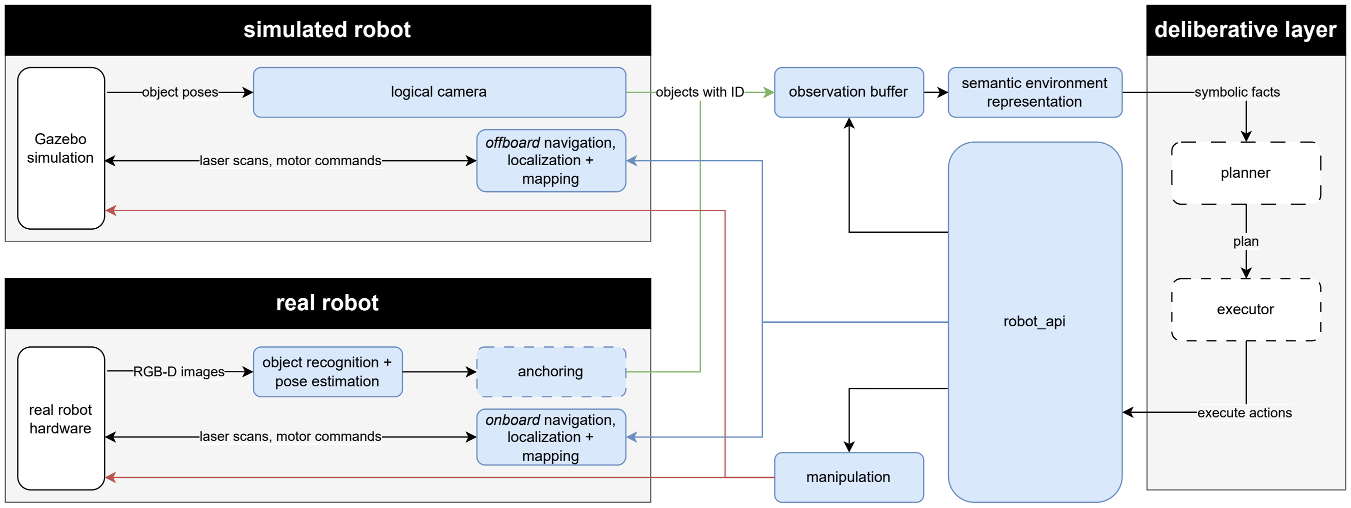

Figure 2: Simplified overview of the software architecture. Blue: Components that are described in the remainder of the paper. Note: Some connections between components have been omitted for clarity of presentation.

will go into more detail about this in its own section. Likewise, where we configured our scenario and robotic skills for the generic planning component, you will need to write such a configuration for your own scenario and robot, potentially by expanding our provided software components.

The following gives a high-level overview of the interplay between the different components, each of which will be described in more detail in the remainder of this paper. The internal onboard PC of the real robot's MiR100 base runs a customized ROS *navigation, localization and mapping* stack that we re-use in our system. In simulation, we replicate this functionality using the standard ROS navigation stack.

On the real robot, images from the robot's camera are processed by the *object recognition and pose estimation* component. The detected objects and their poses are passed to the *anchoring* component, which will track the objects over time and re-identify objects when they re-enter the robot's field of view in order to assign a persistent unique ID to each object. In simulation, the combined function of these two components is performed by the *logical camera*. The last known poses of each unique object are stored in the *observation buffer* and processed (together with other data) by the *semantic environment representation*, which provides symbolic facts about the current world state for use by a planner.

The final set of robot capabilities (pick, place, insert and generic arm movements) is provided by the *manipulation* component.

All robot capabilities are bundled by the *Mobipick Robot API*, which provides a ROS-independent Python library for easy control of the robot.

Finally, the overall control of the robot happens in the deliberative layer, which usually contains a *planner* that processes the symbolic facts about the current world state and generates a plan that is executed by the *executor* by calling functions of the robot_api. While we expect most users of the mobipick_labs system to bring their own planner and executor, we also provide an example implementation of both components.

Gazebo Simulation

Our simulation is based on a custom real robot called Mobipick, which was built in a previous project at DFKI.[3] It consists of several commercially available parts, including a MiR100 base, a UR5 arm, Robotiq 2F-140 gripper and Orbbec Astra Mini S RGB-D camera, as well as two industrial PCs. The real and simulated setups are presented in Fig. 3.

There are five available objects as shown in Fig. 4. The blue box can be used to store and carry multiple objects at once except for the power drill, as this object makes the box too heavy to be carried by the robot in real life.

Navigation

The navigation action allows the robot to autonomously drive from its current location to a destination given by (x, y, θ) with respect to a global reference frame. The behavior automatically avoids dynamic obstacles that might

[3]https://robotik.dfki-bremen.de/de/forschung/projekte/hybr-it/

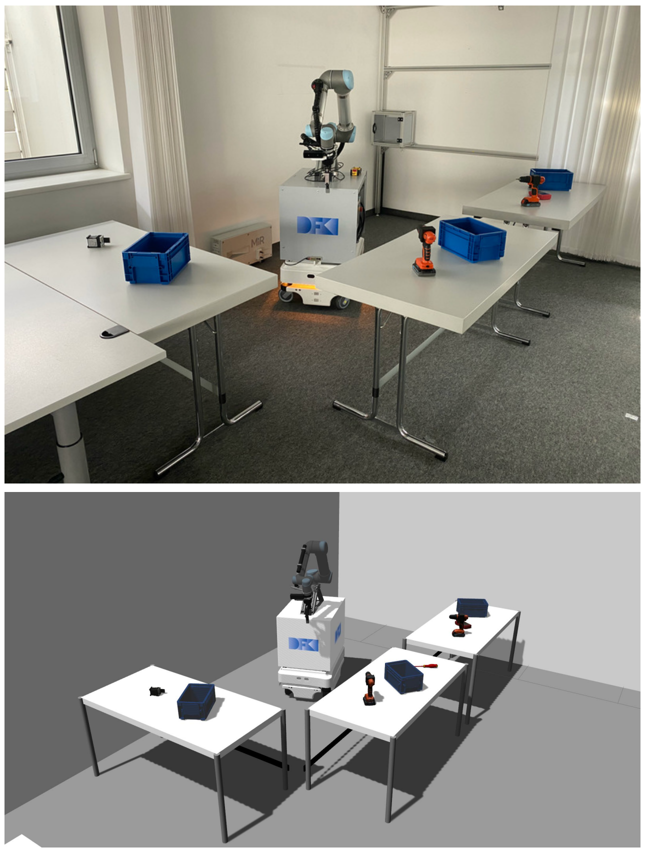
Figure 3: Example of one of the provided pick and place environments. Top: real environment, bottom: simulated environment in Gazebo.

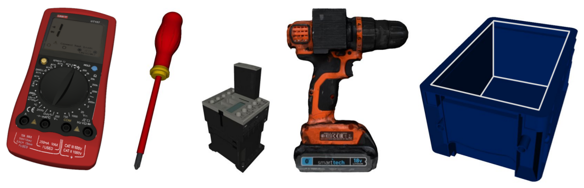
Figure 4: Available objects in mobipick_labs, left to right: multimeter, screwdriver, relay, power drill, blue box. NOTE: objects are not to scale.

appear in the scene at any point in time. For example, a robot serving drinks at a restaurant needs to avoid customers that step into the robot's path. Fortunately, this is all handled by standard software[4] by using the laser scanners of the mobile base. However, they have partial observability of the environment and can only detect obstacles which are at the same height of the LiDAR sensor. This means, e.g., that only the legs of the table in Fig. 5 are visible but not its main surface. To solve this problem, the map of the environment is manually annotated with forbidden zones that the robot cannot enter. Forbidden zones are created for all obstacles that cannot be detected by the laser scanners, specifically all kinds of overhangs (e.g., table surfaces, areas under stairs) and drops (e.g., descending stairways). For the scenario shown in Fig. 3, we additionally attached tape as barriers to the table legs, so the environment perceived by the lasers and the annotated map match more closely.

Perception

Object recognition and pose estimation. The robot is equipped with an RGB-D camera attached to the end effector, allowing the robot to point the camera using its arm to observe different parts of the environment. On the real Mobipick robot, the perception is done via the deep learning approach DOPE (Tremblay et al. 2018), which performs object recognition and 6DoF (degree-of-freedom) pose estimation (i.e., the detection of the object's position and orientation in 3D space).

Anchoring. Anchoring is the process of "creating and maintaining the correspondence between symbols and sensor data that refer to the same physical objects" (Coradeschi and Saffiotti 2003). The anchoring problem refers to the situation where we have multiple objects of the same class, e.g., many blue boxes, but we need to assign persistent unique IDs to each of them, e.g., box_1, box_2, so that a planner can refer to these objects symbolically. The problem is far from trivial and it involves tracking of the objects (as long as the objects are within the camera's field of view), re-identifying individual objects over a longer time horizon when they reappear in the camera's field of view, recognizing unique features on them, and possibly employing domain knowledge to restrict the possible object-symbol assignments. We are currently in the process of integrating the anchoring module from the CoPDA project (Günther et al. 2020; Dittmer et al. 2023) into our system. As a temporary workaround, our demos on the real robot are restricted to only one instance of each object type, which avoids the need for proper anchoring.

Logical camera. A Gazebo "logical camera" sensor is utilized to emulate object recognition, pose estimation, and object anchoring functionalities in simulation. Instead of outputting a 2D image as a typical Gazebo camera sensor would, the logical camera sensor produces the class identification, instance identification, 6D pose, and bounding box dimensions for each object that is inside the camera's frustum (see Fig. 6), using the same output interface as the anchoring module. The class and instance IDs are obtained from the object's name (e.g., box_1), if such a naming structure is adhered to when adding objects to the simulation environment. The bounding box of the object is the axis aligned bounding box (AABB) of the object model. The object's 6D pose is with respect to the logical camera's reference frame but may be transformed into a global reference frame. The plugin[5] for the logical camera allows for adjusting parameters such as the sensor's field of view, clipping planes, and aspect ratio.

One benefit of using the logical camera sensor to simulate the object recognition and pose estimation components is that a GPU is not needed, whereas the deep learning pose estimation approach used on the real robot requires a GPU. The logical camera therefore allows for a fast simulation of ground truth object data on a wider range of hardware, enabling one to focus on evaluating other framework components. However, if desired, one can apply object recognition and pose estimation approaches (e.g., the aforementioned

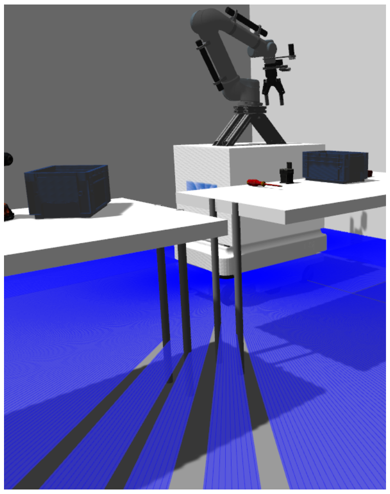
Figure 5: A partially observable table where only the legs are visible to the laser scanner.

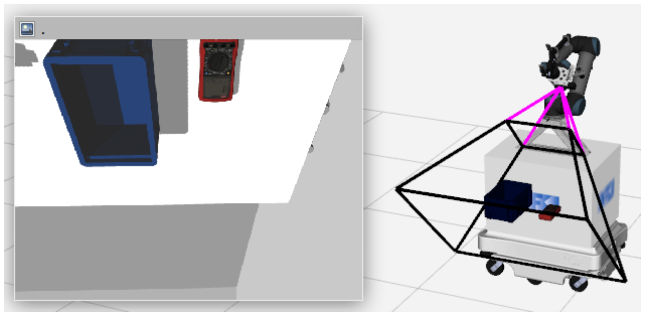
Figure 6: Left: Simulated camera image, Right: Mobipick frustum, field of view of the camera.

[4]https://github.com/DFKI-NI/mir_robot

[5]https://github.com/DFKI-NI/gazebo_logical_camera_plugin_ros

DOPE) directly to the simulated images instead of using the logical camera.

Objects within the logical camera's frustum are fully observable, even if an object may be visually occluded by the environment or another object. In addition, if an object's bounding box partially intersects the camera frustum, the object will be reported as being in the camera's field of view. In such cases, the object's geometry leads to this false positive output as the object is not visually in the field of view. One should be aware of both the lack of occlusion and false positives in the logical camera, as these behaviors differ greatly from the real-world scenario.

Observation Buffer. The poses of perceived objects are stored in an observation buffer called the "pose selector"[6]. The object information is provided by the logical camera or by perception components (e.g., DOPE and object anchoring). In addition, other system components may update or delete the information in the observation buffer as the robot interacts with and manipulates the objects. Each object in the observation buffer has its object class name, unique instance ID, bounding box dimensions, and globally-referenced 6D pose stored. The saved poses are able to be queried by unique class and instance ID (e.g., to retrieve the pose of a specific instance of an object), or by class alone (e.g., to retrieve all poses of a specific object class, which, for example, could be used to find the closest object of a specific type). The observation buffer ROS node allows for service calls to execute the query and update processes.

While the user might not need to know this level of detail, it might be beneficial to discuss some specific details: The observation buffer stores the last known positions of all perceived objects. Due to the partial observability that the robot has about the environment, this information can be partly outdated if objects were moved while the robot was not observing this part of the scene. The task of tracking and clearing information about a specific object falls to a component accessing the observation buffer itself. In the general case, this is performed by object anchoring on the real robot or by the logical camera in simulation.

Semantic Environment Representation

In order for the robot to plan and solve tasks, it needs the current state of the environment as symbolic facts. These facts can be automatically generated by the provided symbolic fact generator[7] module. The generated facts include information about the current arm pose of the robot, the current position of the robot as well as the location of objects on tables, if they have been perceived before. The predefined symbolic arm poses are used and compared to the current state of the arm joints within a given threshold to generate the fact that the robot is in a specific arm pose. Similarly for the robot location and heading, the predefined waypoints are used to generate the fact of where the robot currently is and in which direction it is facing. For the object locations, the information perceived and saved in the aforementioned observation buffer is used to determine if objects are on top of the tables present in the environment. To generate these "on" facts, the bounding boxes and poses of the objects and the bounding boxes and poses of the tables, which are specified in a configuration file, are used to perform a collision check between them using the Separating Axis Theorem. If a collision is detected, a simple comparison of the z-coordinates gives the answer whether an object is on the table or not.

The symbolic fact generator module provides a ROS node, which publishes these facts on a specified topic. Alternatively, the fact generating Python module can be imported directly to create these facts on demand.

Manipulation

All robotic manipulation within mobipick_labs is based on MoveIt (Coleman et al. 2014). In a nutshell, MoveIt is a robotics manipulation framework that integrates motion planning, collision checking, path parametrization, trajectory execution and monitoring, inverse kinematics, Cartesian control, pick and place, and more. Here is a description of the functionality that we expose to the planning community:

Move Arm in Joint Space. Mobipick has a UR5 arm from Universal Robots with 6 degrees of freedom and 5 kg payload. Each joint can be controlled separately by commanding it to a desired target angle that is constrained by custom joint limits.

We can also control all joints at once by sending a list of target angles. This is called a joint space goal configuration. In Fig. 7 we show some pre-recorded arm configurations that are used in some of our demos.

Our API allows to easily send the arm to a named goal configuration, which maps to 6 numeric target joint angles. The use of existing arm configurations or the recording of new ones is a choice to be made by the end user. Under the hood, the MoveIt framework will perform several computations, including collision checking with the environment as well as with the robot itself, plan a path from the start configuration to the desired one, parameterize the previous path to obtain a trajectory and execute-monitor in real-time. As input, MoveIt requires a spatial description of the environment, which is shown in Fig. 8.

Such hand-coded collision boxes are provided and transparent to the end user, however it is important to know that every movement of the arm via MoveIt relies on simplified collision models where also objects need to be included.

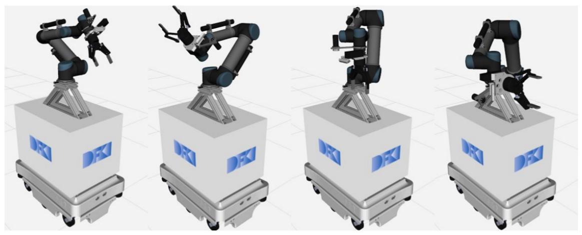

Figure 7: Some pre-recorded named arm configurations: home, observe100cm_right, transport, handover; e.g., transport = [-0.89, -1.87, 2.13, -1.82, -1.57, 3.80]

[6]https://github.com/DFKI-NI/pose_selector

[7]https://github.com/DFKI-NI/symbolic_fact_generator

Object Grasping. In order for the robot to be able to grasp an object, a grasp planner is required. Given the 6DoF pose estimate of an object, the job of the grasp planner is to generate possible end effector poses such that when the gripper is closed, the object is firmly attached to the arm. While generic grasp planning for unknown objects is an open research problem in robotics, we can manually define multiple grasp configurations specific to a gripper and object (see Fig. 9).

From the available grasp configuration set, MoveIt iterates over each of them and queries inverse kinematics to find a feasible motion plan that moves the robotic arm from the current configuration to the target one without colliding with the environment. Such a planning process is time consuming; the main bottleneck is the collision checking step. From a high-level perspective, the parametrization of the behavior is relevant: assuming the 6DoF pose estimate of the object is available in the observation buffer and the object is within reach, we can pick an object from a known class and ID, within a deadline and with the option of ignoring specified objects from collision checking. This is necessary when picking a box with objects inside, e.g., pick blue_box_1 from table_1 with a timeout of 50 sec, ignoring collisions with relay_1 (assuming that relay_1 is inside the box).

Object Placing. Assuming the robot is holding an object, using the object place action will place the object on a desired target table. The algorithm for placing looks for free space on the table and samples multiple random configurations until a collision-free motion plan is found. An example of this action is shown in Fig. 10.

Object Insertion. One or more objects can be placed inside the blue box by calling this action, allowing the robot to carry several items at once and therefore save time. The behavior is almost identical to the place action, but the object is dropped above the blue box instead of being placed on a table.

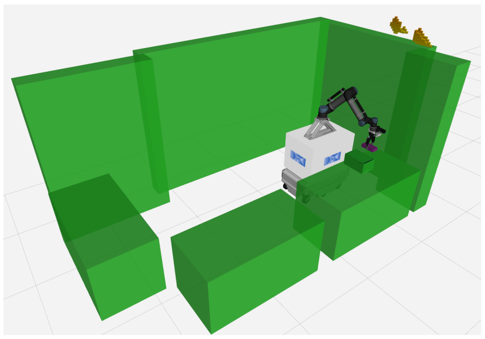
Figure 8: Planning scene: a spatial description of the environment, used for collision avoidance.

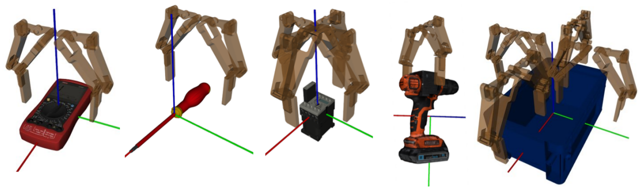
Figure 9: Recorded grasp configurations per object.

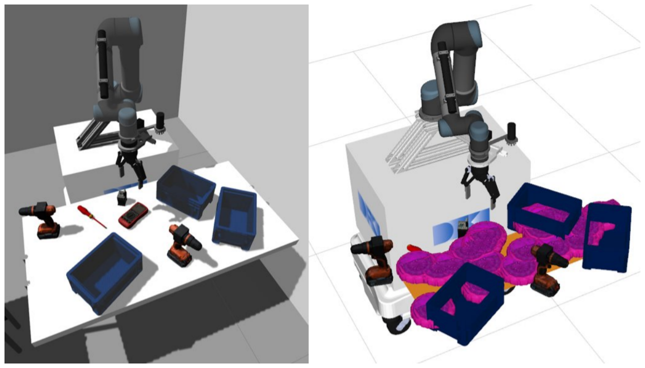
Figure 10: Mobipick placing a multimeter on a cluttered target table by finding free space. Right: sampling possible placements of the multimeter. Left: Result after placing the multimeter.

Mobipick Robot API

For the actions of the Mobipick robot, we have created a Python library called robot_api[8] that allows the user to control all capabilities of the robot via simple Python function calls. It uses ROS underneath and supports ROS message types at its interface but does not require them. This way the user doesn't have to be knowledgeable about ROS to control the Mobipick robot. The small example in Listing 1, taken from the robot_api documentation, demonstrates its overall idea. This code can be run as a Python script or via an interactive Python console. After importing the robot_api module, a `Robot` representation from robot_api is obtained through its namespace.

```
mobipick = robot_api.Robot("mobipick")
```

Listing 1: Robot API basic functionality

```
import robot_api
# Get a Robot object using the robot's ROS
↪   topic namespace.
mobipick = robot_api.Robot("mobipick")

# Get the robot's 2D pose using
↪   localization.
robot_pose = mobipick.base.get_2d_pose()

# Move the robot's arm using MoveIt.
mobipick.arm.move("transport")

# Move the robot's base using move_base.
mobipick.base.move(21.0, 7.0, 3.141592)
```

[8]https://github.com/DFKI-NI/robot_api

Components supported by the robot_api can then be accessed on this `Robot` instance `mobipick` via simple function calls. The following three calls in Listing 1 all use ROS components at the back-end without exposing this to the user.

While the idea of creating a high-level abstract robot API is not new (Diprose et al. 2017; Angerer et al. 2010), we believe it could be helpful to the planning community to encapsulate the complexity of a mobile manipulator behind simple Python commands. In comparison to related work, we intentionally make use of Python 3's type annotation system at all of the robot_api's interfaces with the intent of flexible yet well-defined interfaces. In our opinion, this abstraction helps to keep robotic functionalities and their implementation details hidden as simple actions usable for planning.

The robot_api's design aims at being generic but tries to support many optional robot components. In the previous example we showed that it can make use of the move_base and MoveIt ROS components but it does not have a dependency on these ROS packages, only on their message definitions. By using discovery mechanisms at runtime on the available ROS topics and services, the robot_api can make use of existing ROS components of a system dynamically. For example, calling `robot_api.find_namespace()` will return the detected robot namespaces and thus the supported robot representation obtained from `robot_api.Robot()`. In our Mobipick environment, `["/mobipick"]` would be returned. Where one would typically set up a ROS subscriber or ROS service client first to establish the connection and communicate with other ROS components, the robot_api performs this in the background. It also calls `rospy.init_node()` on demand, i.e., if there is no ROS node already running in the current process, a new node will be initialized the first time it is needed.

The robot_api is still in development and supports only basic functionality assuming you have a mobile robot. By default, it thus only supports localization through `tf` and navigation using `move_base` by providing methods like `get_2d_pose()` and `move()` or similar variants. Its `extensions.py` module assumes a robot arm to be present, which can be controlled using MoveIt.

The concept of the robot_api is not limited to these few functions we implemented here, though. Along with it we provide the mobipick_api package as a robot specific extension, which may serve you as a template for your own robots. Our Mobipick robot has a parallel gripper attached to its robot arm and a depth camera next to it. To make use of them we implemented the grasplan module for manipulation actions and the pose_selector and symbolic_fact_generation modules for storing perceived objects and generating symbolic facts for planning. The mobipick_api imports robot_api, thus extending the few generic methods by more specific ones related to our Mobipick robot.

As a teaser, Listing 2 demonstrates how to navigate to a location, perceive the objects with the camera mounted on the robot arm, pick an object from the table and then place it again on the same table.

Listing 2: Mobipick API usage example

```python
import mobipick_api
mobipick = mobipick_api.Robot('mobipick')

# autonomous navigation
mobipick.base.move(21.0, 7.0, 3.141592)

# estimate 6D pose estimate of an object
mobipick.arm_cam.perceive()

# pick a previously perceived object
mobipick.arm.pick_object('relay_1',
↪ 'table_3',
↪ planning_scene_ignore_list=[],
↪ timeout=50.0)

# assuming that mobipick has an object in
# its gripper, place it on a surface
mobipick.arm.place_object('table_3',
↪ observe_before_place=False,
↪ timeout=50.0)
```

Making use of the robot_api for your own robot is a two-stage process. As a developer you first need to make sure that the assumptions made in the robot_api with regard to namespaces, ROS services and actions, etc. are fulfilled in your ROS system. While these aspects have been developed with genericity in mind, our templates might just not fit to your individual system architecture. In this case, and also in the general case if you intend to expand the API to specific skills of your robot, you would need to write your own API extension first, just as the mobipick_api is built on top of the robot_api. Once this is ensured, any user of your robot can as a second step access the robot_api or your custom API extension just as easily as shown in the code examples of Listings 1 and 2.

Planning and Acting Interfaces

We aim to make mobipick_labs suitable for task planning and provide implementation examples in which we use task planning ourselves. Planning requires very descriptive and precise state and action definitions, which potentially differ from the requirements for execution. In the action interface of

```python
def move_base(self, _: Pose, pose: Pose) ->
↪ bool:
```

for example, we see two pose parameters – but the first one is actually not used within the method. The robot and thus its move_base component does not need to be informed about its current pose to move somewhere else. It has to and will rely on its own localization and navigation components to find a path to a given new pose. Task planning in the general case, however, will possibly calculate long sequences of different actions into the future when the robot is at an arbitrary pose from which it should move. Therefore, an action definition which is suitable for both planning and execution needs to accept a current pose parameter together with the target pose parameter.

It is out of scope of this contribution to describe our work on task planning in detail. On this topic we just like to emphasize that we in fact use such action interface definitions in the robotics application domain to automatically generate the problem description in the planning domain.

Example Planner and Executor

While we expect many potential users of our mobipick_labs testbed to implement their own planner or executor and integrate it using the provided interfaces, we include an example implementation of the planner and executor that runs a simple demo scenario in which the robot has to find a certain object (a multimeter) and a blue box, place the multimeter into the box and bring the box with the multimeter inside to a given target table. The scenario is simple but flexible enough to demonstrate the use of a task planner in a robotics application (as opposed to, e.g., a state machine or behavior tree, which are popular alternatives in robotics). For example, the robot could first bring the box to the table where the multimeter is, set it down, insert the multimeter, and transport the now filled box to the target table; or, it could bring the empty box to the target table, get the multimeter, and insert it directly at the target table. Also, there are a several interesting cases of execution failures that have to be dealt with, especially if the demo takes place in a dynamic environment where humans move some of the objects while the robot is currently not observing the scene. The Mobipick robot is able to detect changes in the scene, react to them, and to some degree adapt to the situation to still fulfill the given goal.

Our example planner and executor builds on the following interfaces and capabilities that we have presented so far, mainly: the semantic environment representation for providing the current state of the world as symbolic facts for the planner; and the robot_api and its specialization mobipick_api to execute the planned actions on the robot. For planning, we use the Unified Planning[9] (UP) library together with the Embedded Systems Bridge (ESB) (Hastam Sathiya Satchi Sadanandam et al. 2023) which are both developed in the project AIPlan4EU. Unified Planning provides a Python interface for creating planning problems and using different integrated planners of various kinds. Using this allows to easily switch between a wide variety of planners; currently, we use Fast Downward (Helmert 2006). The ESB connects our robotic system with Unified Planning in two ways: First, it provides means for connecting the existing Python functions for calculating the symbolic facts that represent the world state and the executable actions of the robot_api with their counterparts in the problem specification of Unified Planning. Second, it executes and monitors the resulting plans. More details about the ESB as well as code examples are given in the referenced publication.

Planning experts can use and integrate their own planners in different ways. The first option is to integrate the planner with the Unified Planning library. It is out of the scope of this paper to describe how this can be done in practice as it depends on the individual planner's programming language and domain format. For example, if the planner already supports the domain description languages PDDL (Ghallab et al. 1998), ANML (Smith, Frank, and Cushing 2008) or HDDL (Höller et al. 2020) the integration into UP could be achieved with a small Python wrapper that uses UP's functions for exporting the problem in that format, execute the planner and give the results back to UP. For details we refer to the documentation of the Unified Planning library. This approach has the advantage that the Mobipick system including the domain representation and execution system does not need to be adapted and the planner can be directly compared to the other planners that are already available in UP by just telling it in one line of code to use another planner. The second option for using another planner in our system is to replace the use of the Unified Planning library completely and replace it with the new planner. It requires the user to generate the planning problem on its own. This can be based on the semantic environment representation or even by processing the robot's sensor data directly. Furthermore, the user needs to dispatch the plan's actions by calling the functions of the robot_api at the appropriate time. The third option lies between between the former two options. The user could re-use the example planning problem formulation that we provide with UP and use UP's export functionalities to export it into one of the aforementioned domain description languages. Afterwards, the new planner can be triggered, e.g., with a system call, to plan the problem given in the exported representation.

In addition to the task planners, the execution algorithms can be changed as well. For this the user can take the resulting plan from UP and execute it with its own execution system and dispatch actions by calling the respective robot_api functions.

ROSPlan Integration

An example of how to use the Mobipick labs system with ROSPlan is available under rosplan demos Github repository[10]. We created a task planning model using temporal PDDL and solved a sample problem using the popf planner in combination with the standard esterel plan dispatcher. The integration details are out of scope of this paper, for more information please refer to the online documentation of the demo.

Tools

To easily interact with the robot capabilities we provide a GUI interface. It is based on Qt5 and can access the high-level functionality described in this paper (see Fig. 11).

The GUI is ideal for a first interaction with the system to get an intuition on how the robot behaves in the scenario and understanding how actions are parameterized.

Another important introspection tool that is available in the robotics community is RViz.[11] We provide a configuration that allows for the visualization of various aspects of the

[9]https://github.com/aiplan4eu/unified-planning

[10]https://github.com/kcl-planning/rosplan_demos/rosplan_mobipick_labs_demo

[11]http://wiki.ros.org/rviz

demo, such as candidate grasp poses, place sampling alternatives, the navigation map and more. By using this tool, real-time visual feedback about specific execution details from the robotics perspective can be easily obtained.

Discussion

Robotic actions require sub-symbolic numeric parametrization, e.g., navigate to coordinates (x, y, θ), or move the arm to a 6D float tuple. However, in planning (and with humans in general), it is often preferred to work with symbolic parameters instead, e.g., navigate to *kitchen* or move the arm to *home* pose, where the symbols *kitchen, home* need to be mapped to their correspondent tensors to generate a complete execution specification. Our API supports this mapping as the user can specify custom waypoints in a configuration file and call the actions in a symbolic or sub-symbolic way as preferred.

Regarding extensibility, the Robot API is not specific to the Mobipick robotic platform and can be used on different mobile manipulators as long as they provide the standard ROS interfaces (e.g., move_base for navigation and MoveIt for manipulation). Also, the software framework is modular in the sense that the separate components (simulation, logical camera, object recognition and pose estimation, anchoring, observation buffer, and semantic environment representation) are separate ROS packages. They can be used independently of each other and be integrated into different robotic platforms through their ROS interfaces. The only parts that are specific to the Mobipick robotic platform are the configurations of the hardware drivers (on the real

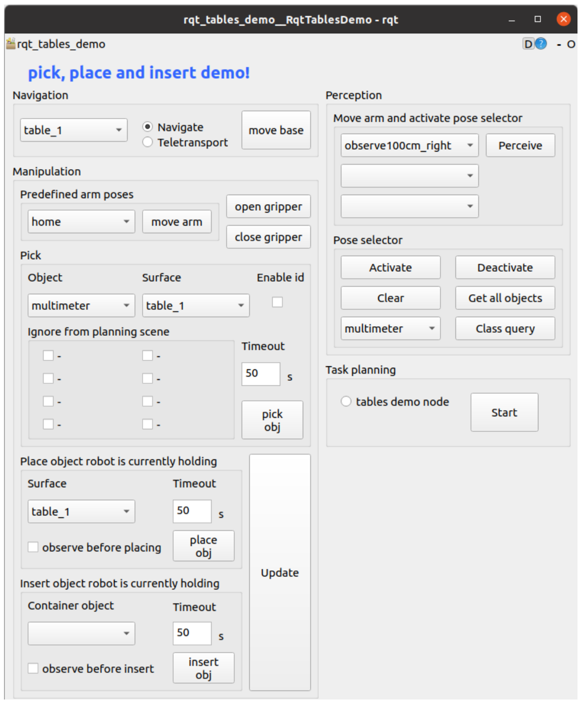

Figure 11: Provided GUI interface to interact with the high-level robot actions.

robot), the navigation and manipulation configurations and the URDF description of the robot.

The simulator can support multi-agent acting scenarios; currently, only multiple instances of the Mobipick robot are possible. Additional custom robots could be used provided they are fully integrated in ROS and provide the standard ROS interfaces for sensors, actors, navigation and manipulation.

Limitations

The mobipick_labs simulation is unfortunately not suitable to simulate human-robot interaction, e.g., on the real robot we have a demo where a power drill is handed over to a person. Such behavior is not possible to replicate in our simulation accurately. In the simulation we provide, only rigid body human simulation is available, so as far as we can go is to simulate one or multiple humans moving around the environment.

Another limitation of our approach lies in the fact that it is specific to one particular scenario: an indoor single mobile manipulator with one arm. While multiple robots are already supported and other extensions such as adding different types of robots are possible, such changes require a certain level of ROS knowledge.

Physics interactions are difficult to simulate accurately, in particular friction. One needs to provide accurate information about the weight, moment of inertia, and friction coefficients along with visual and collision models. Unfortunately, our simulation cannot handle flexible or deformable materials, and is sometimes not very realistic when it comes to force interaction, e.g., a robot pushing objects into the table, or multiple objects being put inside a box.

Conclusions

We present a software contribution called "mobipick_labs" which essentially is a Gazebo[12] based robot simulation of an indoor mobile manipulator for a simple pick and place task of industrial-like objects which are to be transported between tables. The approach is suitable for researchers in the intersection between planning and acting that have little knowledge in robotics but are interested in developing algorithms to control their behavior at the high-level, addressing relevant real world challenges such as partial observability, exogenous events, coping with action failure, dealing with multiple object instances, and knowledge acquisition from faulty sensor data.

VirtualHome seems to be a good tool for simulating human behavior and their interactions with the environment, whereas our work falls short in this aspect with virtually no human-robot interaction capability available at present and no future plans to improve it.

Our contribution is well suited for labs that do not have a complex robot alongside the personnel and the resources required to keep it up to date and eliminates the need for prior expert knowledge in robotics or ROS, enabling researchers to focus on decision making and execution aspects.

[12]https://gazebosim.org/home

Acknowledgments

This work is supported by the CoPDA project through a grant of the German Federal Ministry of Education and Research (BMBF) with Grant Number 01IW19003, and it is supported by the InCoRAP project through a grant of the German Federal Ministry of Education and Research (BMBF) with Grant Number 0IW19002. It is also supported by the AIPlan4EU project which has received funding from the European Union's Horizon 2020 research and innovation programme under GA no. 101016442. Additionally we acknowledge the support from the APRIL EU project via the grant from the European Commission 870142. The DFKI Niedersachsen (DFKI NI) is sponsored by the Ministry of Science and Culture of Lower Saxony and the Volkswagen-Stiftung.

References

Angerer, A.; Hoffmann, A.; Schierl, A.; Vistein, M.; and Reif, W. 2010. The Robotics API: An object-oriented framework for modeling industrial robotics applications. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 4036–4041. IEEE.

Coleman, D.; Sucan, I. A.; Chitta, S.; and Correll, N. 2014. Reducing the Barrier to Entry of Complex Robotic Software: A MoveIt! Case Study. *J. Softw. Eng. Rob.*, 5(1): 3–16.

Coradeschi, S.; and Saffiotti, A. 2003. An Introduction to the Anchoring Problem. *Robot. Auton. Syst.*, 43(2-3): 85–96.

Diprose, J.; MacDonald, B.; Hosking, J.; and Plimmer, B. 2017. Designing an API at an appropriate abstraction level for programming social robot applications. *Journal of Visual Languages & Computing*, 39: 22–40.

Dittmer, A.; Stolzmann, T.; Kammler, F.; Günther, M.; Ferdinand, O.; Thomas, O.; Hertzberg, J.; and Zielinski, O. 2023. Der Dynamic Anchoring Agent: Wissensrepräsentation und Reasoning zur automatischen Wiedererkennung von individuellen Objekten. In D'Onofrio, S.; and Meinhardt, S., eds., *Robotik in der Wirtschaftsinformatik*. Wiesbaden: Springer Vieweg, HMD edition. (in press).

Ghallab, M.; Knoblock, C.; Wilkins, D.; Barrett, A.; Christianson, D.; Friedman, M.; Kwok, C.; Golden, K.; Penberthy, S.; Smith, D.; Sun, Y.; and Weld, D. 1998. PDDL - The Planning Domain Definition Language. Technical report, AIPS.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated planning and acting*. Cambridge University Press.

Günther, M.; Kammler, F.; Ferdinand, O.; Hertzberg, J.; Thomas, O.; and Zielinski, O. 2020. Automatische Wiedererkennung von individuellen Objekten mit dem Dynamic Anchoring Agent. *HMD Prax. Wirtsch.*, 57(6): 1173–1186.

Hastam Sathiya Satchi Sadanandam, S.; Stock, S.; Sung, A.; Ingrand, F.; Lima, O.; Vinci, M.; and Hertzberg, J. 2023. A Closed-Loop Framework-Independent Bridge from AIPlan4EU's Unified Planning Platform to Embedded Systems. In *ICAPS Workshop on Planning and Robotics (PlanRob 2023)*.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26: 191–246.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9883–9891.

Koenig, N.; and Howard, A. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, 2149–2154. IEEE.

Nemiro, L.; Canal, G.; Lima, O.; Cashmore, M.; and Roberts, M. 2021. Designing an adaptable benchmark and competition simulation for integrated planning and execution. In *Workshop on the International Planning Competition (WIPC)*.

Puig, X.; Ra, K.; Boben, M.; Li, J.; Wang, T.; Fidler, S.; and Torralba, A. 2018. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 8494–8502.

Quigley, M.; Conley, K.; Gerkey, B. P.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. Y. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*. Kobe, Japan.

Singh, I.; Blukis, V.; Mousavian, A.; Goyal, A.; Xu, D.; Tremblay, J.; Fox, D.; Thomason, J.; and Garg, A. 2022. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML Language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Tremblay, J.; To, T.; Sundaralingam, B.; Xiang, Y.; Fox, D.; and Birchfield, S. 2018. Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects. In *Conference on Robot Learning (CoRL)*.