Real-time Perception-Guided Motion Planning for a Personal Robot

Radu Bogdan Rusu[†], Ioan Alexandru Şucan[*], Brian Gerkey[‡], Sachin Chitta[‡], Michael Beetz[†], Lydia E. Kavraki[*]

[†] Intelligent Autonomous Systems, Department of Computer Science, Technische Universität München, Germany

`{rusu, beetz}@cs.tum.edu`

[*] Department of Computer Science, Rice University, USA

`{isucan, kavraki}@rice.edu`

[‡] Willow Garage, USA

`{gerkey, sachinc}@willowgarage.com`

Abstract— **This paper presents significant steps towards the online integration of 3D perception and manipulation for personal robotics applications. We propose a modular and distributed architecture, which seamlessly integrates the creation of 3D maps for collision detection and semantic annotations, with a real-time motion replanning framework. To validate our system, we present results obtained during a comprehensive mobile manipulation scenario, which includes the fusion of the above components with a higher level executive.**

I. INTRODUCTION

In this paper we report on our investigations into the realization of a basic manipulation layer for a personal robot performing pick-and-place tasks in environments where humans live and work. The overall goal of our research effort is developing a framework that enables a mobile manipulation system to enter a previously unknown room and clean a table together with a person. The objects to be picked up are easy to grasp, rigid everyday objects such as cups, soda cans, tea boxes, bowls, etc., but they are not known to the robot beforehand.

This seemingly trivial manipulation task poses serious challenges for the state of the art in autonomous mobile manipulation. These challenges include the perception of object constellations on the table, the geometric reconstruction of the unknown objects to be picked up from partial sensor views, the treatment of the rest of the scene as dynamic obstacles – in particular the person reaching into the operating space of the robot, the planning and seamless execution of planned motions when reaching for objects in the presence of moving obstacles, and the determination of appropriate grasps based on partial and possibly inaccurate geometric reconstruction of the objects to be picked up.

Manipulation systems that aim to solve tasks such as the one presented above require the successful operation of many complex components, such as 3D perception, motion planning and more. Furthermore, these components must function robustly and reliably over extended periods of time on real hardware. These constraints may explain why relatively few systems are intended to address the domain that we study [1]–[6]. Due to their complexity, complete systems have not been fully realized yet, and the research problem remains open. Our work describes a manipulation system

Fig. 1. Snapshot of the PR2 robot used during the experiments.

with emphasis on robustness and software re-usability, in addition to safety and functionality.

Our long term goal is to provide robots with safe and goal-directed real-time manipulation behavior. The robots should be capable of performing well while using realistic sensors and making few assumptions about the environment setup. We aim to produce a system that is safe and robust, can run autonomously, can handle failures and situations it has not been presented before. Reliability and flexibility is not only achieved through the capabilities of the individual components (perception, planning, etc.) but also through the task management that synchronizes and parameterizes the activities in dynamic, context-specific ways and detects and locally recovers from failures. We use no scripting for any of the experiments, as all our code runs online and all decisions the robot makes are in real-time. To this end, we report on the experiments carried out using our perception and planning approach on the mobile manipulation system shown in Figure 1, based on the Personal Robot 2 (PR2).

Our research contributes to the state-of-the-art in autonomous mobile manipulation in the following ways:

- *3D scene perception for manipulation:* We propose a 3D scene perception component that computes:
  - A dynamic obstacle map, which is represented by a voxel-based collision map for 3D motion planning.
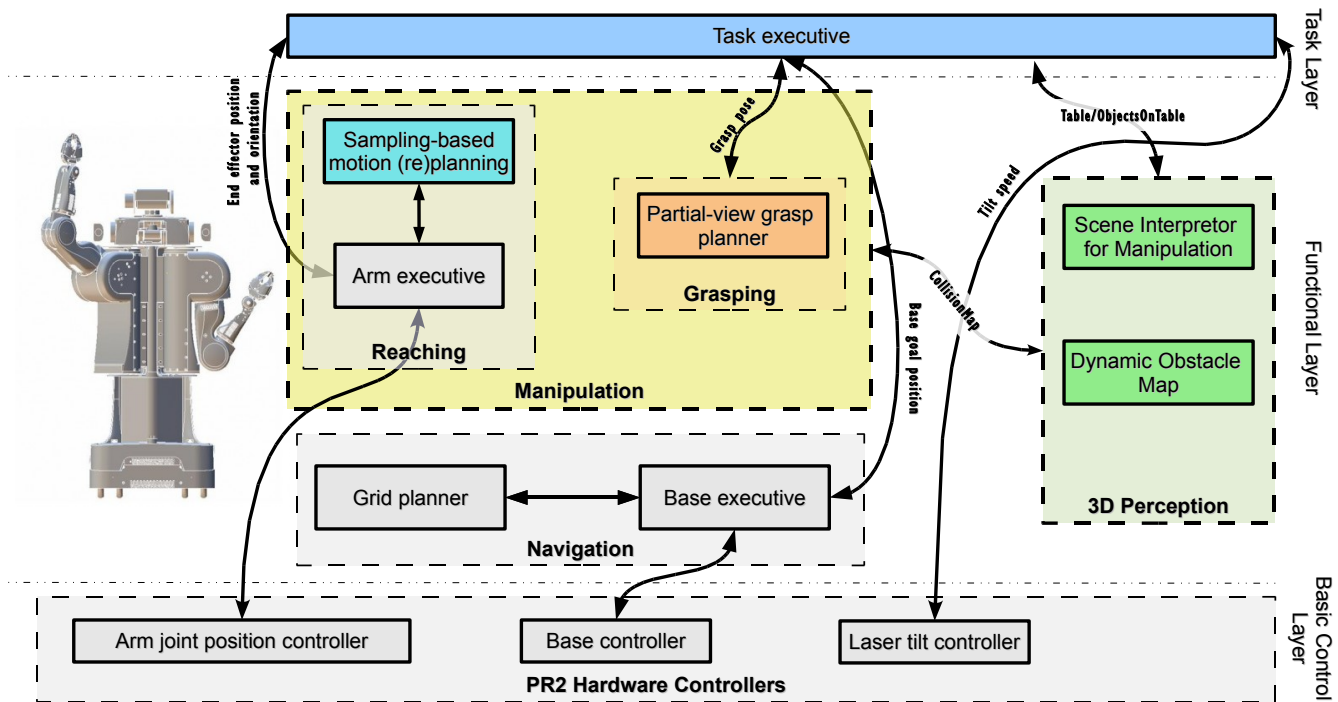
Fig. 2. The overall architecture of our system using ROS. Boxes with continuous lines define nodes. Boxes with dashed lines delimit groups of nodes with related functionality. Communication between nodes is represented by arrows.

– A labeled manipulation scene, which provides a classification of 3D points into various semantic categories such as walls and tables.

- *Sampling-based motion (re-)planning for reaching:* We propose a sampling-based motion (re-)planning module that operates on abstract robot specifications and uses the obstacle map provided by the 3D scene perception for collision detection. The motion planning component uses the Open Motion Planning Library (OMPL), which provides leading-edge motion planning algorithms and tools for optimizing and smoothing motion trajectories (more information in Section IV).

All software components necessary to perform the tasks discussed in this work are modular so they can be reused individually, and are available Open Source.[1] Because of their modularity, they can be used for robots other than the PR2 as long as a respective robot description is available.

II. SYSTEM ARCHITECTURE

A. Hardware

Our experimental system is an alpha prototype of the PR2 mobile manipulation platform (Figure 1). The PR2 comprises an omni-directional wheeled base, telescoping spine, two force-controlled 7-DOF arms and an actuated sensor head. Each arm has a 1-DOF gripper attached to it. The robot can negotiate ADA-compliant[2] wheelchair-accessible environments, and its manipulation workspace is similar to that of an average-height adult.

The sensor head comprises a Hokuyo UTM-30 planar laser range-finder on a tilt stage, and a Videre stereo camera on a pan-tilt stage. The laser is tilted up and down continuously, providing a 3D view of the area in front of the robot. The resulting point clouds are the input to our perception system, which in turns drives our manipulation system. We can, but do not currently make use of data from the stereo camera. A second Hokuyo UTM-30 laser sensor, attached to the base, is used for navigation.

The PR2 carries multiple computers, connected by a gigabit LAN. The current computing configuration is four dual-core 2.6GHz machines running Linux. One of the four machines is modified to run a real-time kernel, providing a guaranteed 1KHz control loop, via EtherCAT,[3] over the robot's motors and encoders.

B. Software

A system as complex as the PR2 is driven by a number of subsystems that need to easily be able to communicate with each other: hardware drivers, controllers, visual perception, motion planning, higher level control, etc. The computational needs of these components virtually necessitate the use of a distributed computing environment. To accommodate these needs, we use ROS,[4] which provides a communication framework that hides the complexities of transferring data between processes, regardless whether the processes run on the same machine or not.

Processes using ROS are called *nodes*. Nodes communicate primarily in a publish/subscribe fashion, using common

data formats on channels called *topics*. Nodes can also call *services* provided by other nodes, in a manner akin to remote procedure calls. Abstracting communication between running processes in this way, through topics and services, allows for the implementation of a modular system. Modules can be written in a variety of languages and can execute anywhere in the network. In addition to the basic communication system, ROS offers an extensive suite of tools for process control, system inspection, and data visualization.

The software architecture of our mobile manipulation framework is illustrated in Figure 2. The figure shows the complete system, but the focus of this paper is on the components depicted in color: 3D perception and motion planning. The architecture can be viewed as a 3-tiered one. The lowest layer provides the PR2 hardware controllers. The middle layer contains the functional modules including the ones discussed in this paper. The top layer is represented by the task executive that manages the processes at the lower layers. This is where the tasks that the robot performs are specified. In particular, the task executive parameterizes and synchronizes processes such as perception, navigation, reaching and grasping. Its purpose is to also provide context-specific control, responsiveness to sensory events, and failure detection, analysis, and recovery.

The 3D perception system consists of two main nodes which build two separate maps: a dynamic obstacle map used for collision detection in 3D while performing arm movements, and a scene interpretor which aggregates the acquired point cloud dataset with semantic annotations (e.g., class labels such as: floor, walls, tables, objects on table [7]).

Motion planning requiring 3D collision avoidance (for example when moving one of the arms) is done using sampling-based algorithms [8], [9]. From an architectural standpoint, sampling-based motion planning relies on a set of libraries, following a modular design: actual motion planning, collision checking and robot modeling are kept separate. These libraries are used together in a node that monitors the current robot state and the dynamic obstacle map, and provides the interface (service calls, topics) to perform motion (re-)planning.

The modular structure of our framework allows seamlessly swapping nodes that provide the same interface. This is an option we often make use of in the development process and it enables the advancement of the platform.

III. 3D PERCEPTION

To be able to interact with its environment, a robotic system must first be able to perceive it, and it must do so accurately and detect pertinent changes as they occur. This perception component is fundamental to both localization and motion planning.

The input to the 3D perception module is a set of 2D laser scan messages, acquired from the tilting Hokuyo laser sensor installed on the head of the PR2 robot. Figure 3 presents a detailed version of the 3D perception pipeline used for the experiments in this paper.
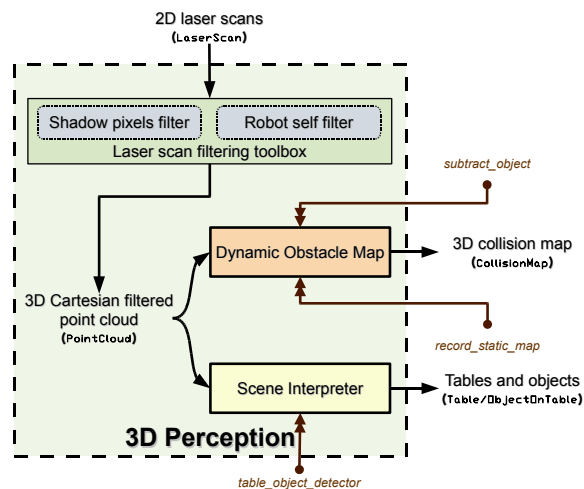

Fig. 3. Detailed diagram of the 3D perception pipeline presented in Figure 2. The double arrow messages represent service requests served by the nodes.

Before projecting each 2D laser scan into a 3D point cloud Cartesian frame, a set of preprocessing filters are applied, in order to clean the data of spurious measurements, as well as to remove points from the scan which are sampled from the robot body. The latter is achieved by treating the robot's body parts as convex hulls and checking the intersection of the laser scans with the hulls, while the shadow pixels filter removes erroneous points by comparing the angles formed between the lines created by subsequent scan measurements with the viewpoint, against a statistical threshold. However these filters are part of a more general laser scan filtering toolbox[5]. After the data is partially cleaned, we project all the points left into a global 3D coordinate frame. This constitutes the input of the two main nodes: the Dynamic Obstacle Map, and the Scene Interpreter.

To formulate the perception computational models, we introduce the following notations:

- p_i is a 3D point with $\{x_i, y_i, z_i\}$ geometric coordinates, p_i^x, p_i^y, and p_i^z refer to the individual x, y, respectively z dimensions of the point;
- n_i is a surface normal estimate at point p_i having a $\{nx_i, ny_i, nz_i\}$ direction;
- $\mathcal{P}_n = \{p_1, p_2, \cdots\}$ is a set of nD points (also represented as $\mathcal{P}$ for simplicity).

A. Dynamic Obstacle Map

The Dynamic Obstacle Map is comprised of a set of geometric primitives created from the point cloud $\mathcal{P}$. This representation is updated in real-time for every new set of points, and its main usage is to provide support for collision avoidance (i.e., reactive motion execution) for motion planning.

The node implementing the Dynamic Obstacle Map takes any $\mathcal{P}$ set of points that is available from the sensor and inserts it into the map, which can generally be represented in multiple formats, using: spheres, boxes, points, or triangle

---

[5]The toolbox constitutes a separate component that we work on as part of the ROS project, and falls outside the scope of this paper.

meshes. The latter is not recommended for situations where the data is extremely sparse, as incremental model updates become more difficult. In general, the node discretizes the space into smaller, uniform parts (called *leaves* for the remaining of this article), with a certain user-defined resolution and publishes them in one of the above mentioned formats on the ROS network to the collision checker (see Section IV).

If the robot is to be allowed to operate and manipulate at high speeds, the laser tilt unit must be actuated as fast as possible, to account for the environment dynamics. This however has the undesired effect that the total number of points per sweep is diminished considerably, and small or fast moving obstacles will not be present in the resultant map. To compensate for these types of situations, the node keeps a temporal sliding window (i.e., data queue) for all data received, and computes the final obstacle map by integrating all the data together.

A deciding factor on the average computation time per map is represented by the number of data leaves that are requested from the motion planner, that is, the resolution of the world. Because there is no good fixed resolution that best accounts for all possible scenarios, the Dynamic Obstacle Map architecture is designed to be parameterizable on the fly. Furthermore, the task executive can specify what the space of interest around the robot is – the space that needs to be accounted for in the map. Taken together, these two constraints lead to significant computational decreases in the creation of the map. Algorithm 1 outlines the basic steps that are used to construct the map in real-time.

Algorithm 1 Dynamic Obstacle Map creation

$\mathsf{b}_r, \mathsf{r}_r$ *// robot bounds and resolution*
$\mathcal{P} = \{\boldsymbol{p}_1 \cdots \boldsymbol{p}_n\}$ *// set of 3D points*
$t_{\min} \stackrel{+}{=} 1, t_{\max} \stackrel{+}{=} 1$ *// increment sliding window time*
if $\neg(\exists \mathcal{L})$ *// no spatial decomposition exists (first $\mathcal{P}$)*
   $\mathcal{L} \leftarrow \mathsf{F}(\mathsf{b}_r, \mathsf{r}_r)$ *// create an empty list $\mathcal{L}$ of 3D leaves*
for all $\boldsymbol{p}_i \in \mathcal{P}$ **do**
  if $(-\mathsf{b}_r \le \boldsymbol{p}_i \le \mathsf{b}_r)$ *// check if inside bounds*
    estimate $(\mathsf{l}_{xyz} \in \mathcal{L}, \boldsymbol{p}_i \subset \mathsf{l}_{xyz})$ *// find the right leaf for $\boldsymbol{p}_i$*
    add $\boldsymbol{p}_i$ to l_{xyz}
for all $\mathsf{l}_{xyz} \in \mathcal{L}$ **do**
  if $(\boldsymbol{p}_j^t \le t_{\min} \lor \boldsymbol{p}_j^t \ge t_{\max}, \boldsymbol{p}_j^t \subset \mathsf{l}_{xyz})$ *// check if outside sliding window*
    remove $\boldsymbol{p}_j$ from l_{xyz}
$\mathcal{M} \leftarrow \mathsf{F}(\mathcal{L})$ *// create the final map from the set of leaves*

In addition to the periodical map creation, the node supports two important service calls, underlined with a double arrow in Figure 3. The *record_static_map* service call triggers a special mode in the node which records a complete sweep map as a static fixed reference map, that all recorded data will be merged with until the next service call. This translates into: $\mathcal{M}_f = \mathcal{M} \cup \mathcal{M}_r$, where $\mathcal{M}$ represents the map created from the current set of leaves, $\mathcal{M}_r$ is the static fixed reference map recorded at the service call, and $\mathcal{M}_f$ is the final 3D dynamic obstacle map. Having a static reference map is beneficial during longer manipulation scenarios, where the estimated collision map may contain holes due to occlusions caused by the robot's arms at certain positions in time. This may in turn negatively affect the motion planning algorithms, which in the absence of pertinent solutions will attempt to use this occluded space, even though it has a

certain probability of still being occupied. By recording reference maps and combining them, this problem can be alleviated. Furthermore, the Scene Interpreter can decompose regions of interest in the reference map such as the table for example, and remove everything else (such as the objects sitting on top of it), as shown in Section III-B. Thus, the reference maps can be updated only with those parts of the environment which have a higher probability of being fixed over subsequent pick and place operations.
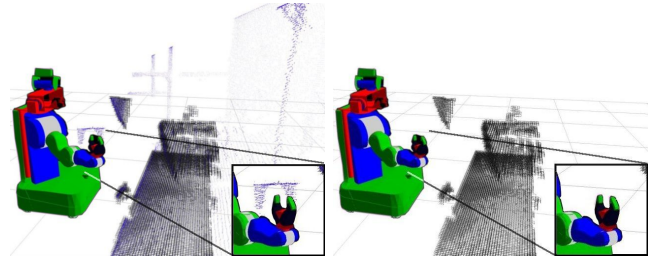

Fig. 4. Left: a dynamic obstacle map representation $\mathcal{M}_f$ over-imposed on the point cloud $\mathcal{P}$; right: the dynamic obstacle map $\mathcal{M}_f$ alone. Please notice that the object held in the gripper is part of the point cloud $\mathcal{P}$, but has been removed using the *subtract_object* service call from the $\mathcal{M}_f$ map.

The second service call, *subtract_object*, introduces a negative collision map created from a given object that can be continuously subtracted from the computed maps, that is: $\mathcal{M}_f = \mathcal{M} \setminus \mathcal{M}_o$, where $\mathcal{M}_o$ represents the negative map created from the given object. The object is given as either a list of leaves or an oriented 3D bounding box. A special use of this service call is to subtract objects from the collision map currently being manipulated with the gripper (see Figure 4). To achieve this, the node subscribes to the ROS network and enquires for the current position of the end-effector, and uses the given oriented object bounds to subtract the data. Please note that the object present in the gripper from the left part of Figure 4 is incomplete in $\mathcal{P}$, because parts of it are removed by the robot self filter at the laser scan level, due to the convex hulls of the gripper being much larger than the actual gripper itself.

B. Scene Interpreter

The second actor of the 3D perception pipeline is the Scene Interpreter node, which continuously segments and classifies surfaces by fitting planar models to them, in order to abstract sensor data and make it more informative.

In general, the planar classification can be performed in two different ways: i) using a set of heuristic rules, which make use of the fact that the robot's Z_r axis is parallel to the Z_w axis of the world, and label points based on the geometrical properties of the planar surfaces they belong to; ii) using machine learning classifiers which can be trained on simple sets of features extracted from segmented planar regions, and the resultant models are used to annotate the planes with the desired classes. For the purpose of the experiments in this paper, we are mostly interested in pick and place operations, and thus the most important areas that we need to detect are tables and other horizontal planes which can support objects. Therefore there is no need to create a feature set and train a model, as it suffices to

construct a single heuristic rule for determining all major horizontal planar areas in the world that the robot could operate on. The vertical bounds of the search are given by the physical constraints of the robot arms, i.e., what is the lowest and highest place where the robot could pick an object from. Figure 5 presents a more general scene interpretation for planar areas segmented from a $\mathcal{P}$ point cloud dataset.
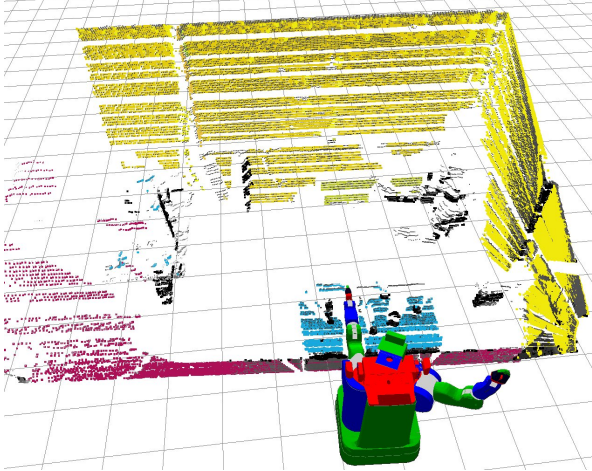

Fig. 5. A scene with point annotations labeled with the Scene Interpreter: floor (dark red), ceiling (dark blue), yellow (walls), and light blue (tables).

The node has two operation modes: i) a continuous mode where for each acquired $\mathcal{P}$ set of points, it labels all major planar areas; and ii) a service mode, where the task executive requests a list of tables and objects sitting on them in view. Algorithm 2 presents a brief description of the computational steps invoked for finding tables and objects on them when the node is running in the service mode, and partial results are presented in Figure 6.

Algorithm 2 Find tables and objects

b_{min}^z, b_{max}^z *// min/max reachable arm positions on z, i.e., table bounds*
$\mathcal{P} = \{p_1 \cdots p_n\}$ *// set of 3D points*
$\mathcal{P}_b = \{p_i, b_{min}^z \leq p_i^z \leq b_{max}^z\}$ *// subset of all 3D points within table bounds*
for all $p_i \in \mathcal{P}_b$ **do**
　estimate $(n_i$ from $\mathcal{P}^k)$ *// estimate surface normal from nearest neighbors*
　if $(\alpha = n_i \times \mathbb{Z} \approx 0)$ *// check if the normal is parallel to the Z axis*
　　$\mathcal{P}_z \leftarrow p_i$ *// add p_i to the $\mathcal{P}_z$ set*
estimate $(\mathcal{C} = \{\mathcal{P}_z^1 \cdots \mathcal{P}_z^n\}, \mathcal{P}_z^i \subset \mathcal{P}_z)$ *// break $\mathcal{P}_z$ into Euclidean clusters*
for all $c_i = \mathcal{P}_z^i \in \mathcal{C}$ **do**
　// find the best plane fit using sample consensus
　estimate $(\{a, b, c, d\}, a \cdot p_i^x + b \cdot p_i^y + c \cdot p_i^z + d = 0, p_i \in c_i)$
　estimate (a_{min}, a_{max}) *// find the min/max bounds of the planar area*
　$\mathcal{M} \leftarrow F(c_i)$ *// add the table parameters to the final map*
　for all $p_i \in \mathcal{P}$ **do**
　　if $(a_{min}^x \leq n_i^x \leq a_{max}^x, a_{min}^y \leq n_i^y \leq a_{max}^y)$ *// within bounds?*
　　　$\mathcal{P}_o \leftarrow p_i$ *// add p_i to the $\mathcal{P}_o$ set*
　estimate $(\mathcal{O} = \{\mathcal{P}_o^1 \cdots \mathcal{P}_o^n\}, \mathcal{P}_o^i \subset \mathcal{P}_o)$ *// break $\mathcal{P}_o$ into Euclidean clusters*
　for all $o_i \in \mathcal{O}$ **do**
　　$\mathcal{M} \leftarrow F(o_i)$ *// add the object parameters to the final map*

The resultant map $\mathcal{M}$ contains a set of tables with their planar equations and bounds, and a set of object models $\mathcal{O} = \{o_1 \cdots o_n\}$. These object models represent compact partial-views representations for the objects supported by tables in the real world. Their usage is twofold: i) on one hand they provide the exact goal positions for the gripper; and ii) they support the *subtract_object* service call from the Dynamic Obstacle Map, once the object is picked up.
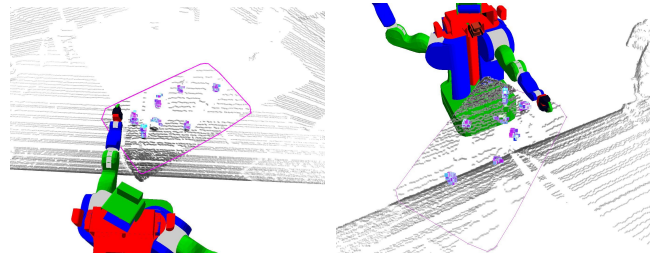

Fig. 6. The extraction of a table surface and the individual object clusters supported by it, from a partial view.

IV. MOTION PLANNING

Computing safe motions for a robotic system is the basic component for achieving any useful task. The motion planning component is what computes these motions. To perform planning that is fast enough for the PR2, different planners are used depending on the task.

A. Grid-based motion planning for navigation

When moving the robot base, without moving the arms, we employ a grid-based global planner and a local controller designed for efficient motion in tight spaces. Given a goal by the task-level executive, the planner computes the gradient of the global navigation function [10], determining an optimal X-Y (ignoring yaw) kinematic path. This kinematic path is passed to a trajectory rollout controller [11], which accounts for the robot's exact shape and dynamic constraints in computing desired X, Y, and yaw velocities.

Both the planner and the controller consult a 2D obstacle map constructed by projecting obstacle data from all available sources (in our case the horizontal laser and tilting laser) onto the X-Y plane. This projection produces conservative navigation behavior: the robot will never choose an unsafe path, at the cost of excluding some safe paths, such as moving the base partially underneath a table. In future work, we will relax the latter constraint by planning for the base and arm together, possibly by employing the sampling-based techniques described in the next section.

B. Sampling-based motion planning for reaching

When the robot is supposed to move its arm in order to reach an object, a motion planner capable of planning in a 7-dimensional state space is needed. The dynamic obstacle map received from the 3D perception pipeline is to be used for collision avoidance. The task executive specifies a goal position / goal region as input. The goal can be specified in one of two ways:

- *explicit*: the goal positions are specified for each of the joints in the arm
- *implicit*: constraints on the goal position are imposed. The motion planner is to find a path that takes the robot to a state that satisfies these constraints.

The output of the motion planner is a kinematic path that takes the arm from the its current position to the goal while avoiding collisions.

In addition to constraints specified for implicit goals, it is also possible that constraints for all states along the solution

path are needed. As an example, this would be useful when planning for the arm while holding a glass of water: we want to make sure the water will not be spilled, so the task executive can impose a constraint on the orientation of the gripper. In general, a constraint is a function $f : \mathcal{X} \rightarrow \{true, false\}$, meaning that for any state $\boldsymbol{x} \in \mathcal{X}$, it can be decided whether it satisfies a constraint or not. For the purposes of our experiments, constraints that restrict the position of certain robot parts (the gripper) at the goal state were all that was necessary.

In order to address these needs, sampling-based motion planners, which have been shown to be able to handle such problems [8], [9], are used. In particular, Kinematic Motion Planning by Interior-Exterior Cell Exploration (KPIECE) is used – a kinematic version of the work in [12].

The main idea behind sampling-based motion planning [8], [9] is to sample the state space $\mathcal{X}$ of the robotic system and maintain a data-structure of the samples that can be used to guide the robot to the goal [13]. $\mathcal{X}$ is assumed to be a differentiable manifold of dimension m such that a single element $\boldsymbol{x} \in \mathcal{X}$, $\boldsymbol{x} = (v_1, ..., v_\mathsf{m})$ completely describes the state of a robotic system. This means that each component v_i defines a parameter of the system (joint angles, positions in space, etc.). The notion of $\mathcal{X}_{valid}$ is defined to represent the states in $\mathcal{X}$ the robot is allowed to be in. These states are usually collision free states, potentially respecting additional constraints. An exact representation of $\mathcal{X}_{valid}$ is not needed, and would be hard to compute; a process that decides whether a state $\boldsymbol{x} \in \mathcal{X}$ satisfies $\boldsymbol{x} \in \mathcal{X}_{valid}$ is sufficient.

A solution to a motion planning problem is defined to be the set $\mathcal{S} = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_k\}$, where $\boldsymbol{x}_0$ is the initial state and $\boldsymbol{x}_k$ is in the goal region. Each path segment $[\boldsymbol{x}_i, \boldsymbol{x}_{i+1}]$, obtained through linear interpolation, $i \in \{0, ..., k-1\}$, is assumed to be contained in $\mathcal{X}_{valid}$.

As shown in Algorithm 3, KPIECE proceeds by iteratively building a tree of motions in the state space of the robot, starting from the initial state. At every iteration, a state in the tree is selected to continue the expansion from. This state is chosen such that it is part of a less-explored region of the state space. The less explored regions are detected by imposing a grid on a projection $\mathcal{E}(\mathcal{X})$ of the state space $\mathcal{X}$, as shown in Figure 7.

A value of importance is associated to grid cells so that cells that would benefit the exploration most are more important. The most important cell is deterministically selected at every iteration and a state from that cell is chosen for tree expansion. Grid cells are considered more important if they are created later in the run, states from them have been selected few times and the number of states in the tree of motions covered by the cell is low. Cells are separated into interior and exterior. A cell is interior if it has $2n$ non-diagonal neighbors, where n is the dimension of the projection $\mathcal{E}(\mathcal{X})$. Giving preference to exterior cells over interior ones pushes the tree to cover more space faster.

The previously presented KPIECE algorithm is part of a larger library project in which we work on, named the

---

Algorithm 3 KPIECE ($\mathbf{x}_{start}$, $allowed\_time$)

$time\_limit \leftarrow \text{NOW}() + allowed\_time$
Create an empty grid G
$G.\text{ADDSTATE}(\mathbf{x}_{start})$
while $\text{NOW}() < time\_limit$ **do**
  Select a cell c from G, with a bias on exterior cells (80% - 90%)
  Select a state $\mathbf{s}$ from c according to a half normal distribution
  Sample a state $\mathbf{x} \in \mathcal{X}$
  Extend from $\mathbf{s}$ towards $\mathbf{x}$ until some state $\mathbf{x}'$ (linear interpolation)
  if path segment $[\mathbf{s}, \mathbf{x}'] \in \mathcal{X}_{valid}$
    $G.\text{ADDSTATE}(\mathbf{x}')$
    $\mathsf{c}.score = \mathsf{c}.score \cdot \alpha$ *// penalize the score of the cell by α,*
                             // $0 < \alpha < 1$ (usually close to 1)
    If $\mathbf{x}'$ in goal region, **return** path to $\mathbf{x}'$
  else
    $\mathsf{c}.score = \mathsf{c}.score \cdot \beta$ *// penalize the score of the cell by β, $0 < \beta < \alpha$*
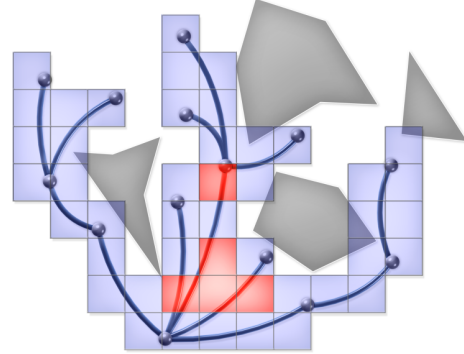

Fig. 7. Representation of a tree of motions and its corresponding discretization. Cells are distinguished into interior (red) and exterior (blue).

Open Motion Planning Library (OMPL). Currently the library contains implementations of RRT (Rapidly-exploring Random Trees) [14], [15], EST (Expansive Space Trees) [16], [17], SBL (Single-query Bi-directional probabilistic roadmap planner with Lazy collision checking) [18] and the version of KPIECE presented in Algorithm 3. In addition to planning algorithms, OMPL contains code that can do inverse kinematics using a genetic algorithm: GAIK (Genetic Algorithm Inverse Kinematics) [19]. While this can in general be orders of magnitude slower than actually solving for an arm's angle positions, it is significantly more general: any set of constraints defining the goal position representable as code can be used and the provided solution is a valid one (according to collision detection). Tools for smoothing solution paths exist as well. For the purposes of this work, in addition to KPIECE, we also used GAIK, to provide a goal state when the goal is specified implicitly, and path smoothing.

The visualization and collision checking tools are kept separate from OMPL. An abstract interface for a collision detector is available in ROS, so that different libraries can be used as a back-end. The representation of the robot inside the collision detector is based on URDF[6]. This description is loaded at runtime from the ROS network and a representation of the robot's state space $\mathcal{X}$ is automatically constructed. The robot's parts are used for collision checking. Having a state space representation automatically built allows for more general code, in the sense that the code is not specific to the PR2, but can be applied to any robot described using URDF.

---

[6]Universal Robot Description Format, http://pr.willowgarage.com/wiki/RobotDescriptionFormat

When a robot holds an object in its gripper, collisions between the grasped object and the other objects in the environment must be avoided as well. To achieve this, the collision detector allows attaching objects to various robot parts. At the same time, the perception pipeline subtracts the grasped object as if it were part of the robot, as shown in Figure 4. This allows motion planning to compute plans that correctly account for the changes in the environment after an object is grasped.
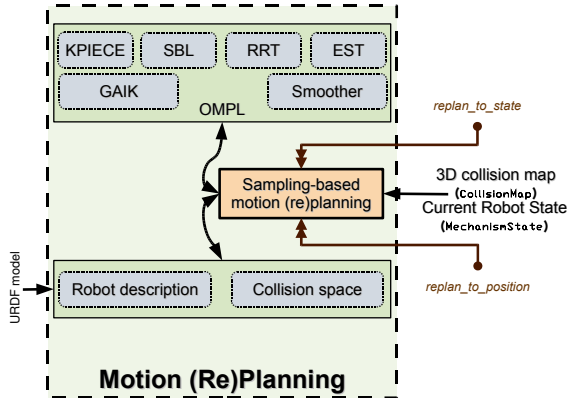

Fig. 8. Detailed diagram of the sampling-based motion planning architecture presented in Figure 2. The double arrow messages represent service requests served by the nodes.

A node monitors the current state of the robot and maintains a collision space based on the dynamic obstacle map received from the perception pipeline (see Figure 2). Once a request for planning is received, a motion plan that starts at the current state of the robot and takes it to the goal position is computed using `OMPL` (the detailed architecture is represented in Figure 8). The computation has a time limit, and if no motion plan is found within the allotted time, failure is reported. On success, the computed motion plan is published to the `ROS` network. As long as the robot has not reached its goal position and a new goal has not been received, the node continues to monitor the current state of the robot and the perceived map. If at any point in time the currently executed plan becomes invalid (something is obstructing the robot's path), or the computation of the motion plan previously failed, this information is reported and a new motion plan is computed (hence the term "replanning"), to address the changes observed in the environment. This logic is represented in Algorithm 4. Previous work [20] advocates the benefits of maintaining information between planning attempts and not necessarily planning until the goal is in fact reached. While this is indeed an ingenious approach to replanning, we chose to employ a simpler version of replanning for the initial version of our setup: we use faster planning algorithms and always plan to a state as close as possible to the goal.

C. Grasping

The long term goal of our setup is to use a partial-view grasp planner for determining the grasp (end-effector position and orientation). The idea would be to attempt to compute reasonable grasps based on laser data that does not

---

Algorithm 4 Motion (Re-)Planning

$\mathcal{G}$ *// goal region specification*
$\mathsf{s} = \boldsymbol{x}_0$ *// current/starting state (s)*
repeat
  repeat
    get $(\mathcal{M} = \mathcal{M}_i, \mathsf{s} = \boldsymbol{x}_i)$ *// get the map $\mathcal{M}_i$ & robot state $\boldsymbol{x}_i$*
    estimate $(\mathcal{S} = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_k\} = \mathsf{F}(\mathcal{M}, \mathsf{s}, \mathcal{G}))$ *// compute a motion plan*
  until $(\mathcal{S} \subset \mathcal{X}_{valid}, \boldsymbol{x}_k \in \mathcal{G})$ *// found a plan*
  repeat
    get $(\mathcal{M} = \mathcal{M}_i, \mathsf{s} = \boldsymbol{x}_i)$ *// get the map $\mathcal{M}_i$ & robot state $\boldsymbol{x}_i$*
  until $(\mathcal{S} \not\subset \mathcal{X}_{valid}) \vee (\mathsf{s} = \boldsymbol{x}_k)$ *// current plan not valid or we are done*
until $(\mathsf{s} = \boldsymbol{x}_k)$ *// goal is reached*

---

necessarily fully cover the object we want to grasp. However, until this part of our system is complete, we simplify the task and assume that a direct approach with the gripper in horizontal orientation will properly grasp the object. This limits the set of objects that we can grasp to those that can be reasonably approximated using the bounding boxes generated by our perception approach. Such objects include small box shaped objects and soda cans. Our approach, however, is not restricted to these kinds of objects and can easily be extended to more general grasping situations.

V. EXPERIMENTAL RESULTS

We have conducted multiple experiments on the `PR2` robot to validate the efficiency of our proposed architecture. The experiments consisted in the tight integration of the aforementioned system components[7], in the following scenario:

1) the task executive triggers a slow laser scan and calls the Scene Interpreter's service request for a list of table candidates with objects on them;
2) if the list returned is not within reach, the executive plans a trajectory for the base, and moves the robot in the vicinity of the closest table;
3) the task executive triggers fast laser scans and starts the Dynamic Obstacle Map;
4) from the list of objects on table, one is randomly chosen, and its goal pose is given to the motion planner, which computes a safe trajectory for the arm;
5) while the arm is moving, the information provided by the Dynamic Obstacle Map is used to replan the trajectory in case it's deemed as no longer safe;
6) the scenario ends when the end effector reaches the goal position and the object is in a graspable state.

An example of a path planned for the arm is shown in Figure 9. Figures 10, 11, and 12 present the time spent for computing the collision map, the arm inverse kinematics, and the trajectory to follow respectively, for one such experiment. The four plots shown in each figure represent laser sweeps of: 1 second in a static scene (1s_static), 1 second in a dynamic scene (1s_dynamic), 2 seconds in a static scene (2s_static), and 2 seconds in a dynamic scene (2s_dynamic) respectively.

As shown, both for laser sweeps with a period of 1 or 2 seconds, all components manage to finish their computations in real-time, that is before the next state update. The perception system reliably detected a candidate table

---

[7]http://www.willowgarage.com/iros2009-tabletop-manipulation contains demonstration videos taken during the experiments.

and the object clusters sitting on it every time, the only exception being environments where multiple tables were located approximatively at the same distance from the robot. The planning component had a success rate of 89%. The failures we observed were due to inaccuracies in modeling our obstacles which sometimes caused the planner to consider the initial state of the robot was in collision. We are working towards avoiding this issue in the future.
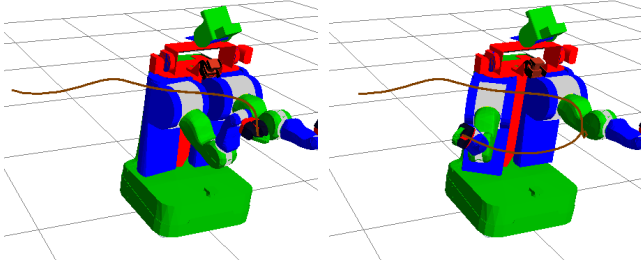

Fig. 9. Two screen-shots of a path plan being executed. The trail of the end-effector is displayed as well.
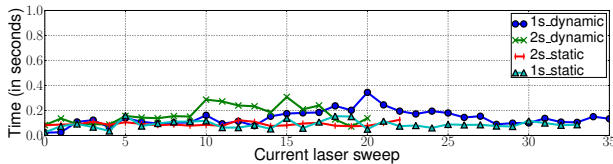
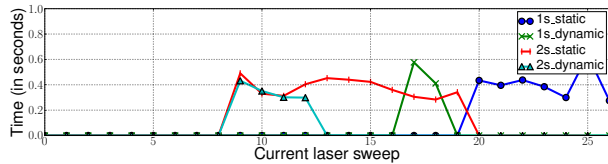Fig. 10. Dynamic Obstacle Map computation plot (queue of 5 seconds)

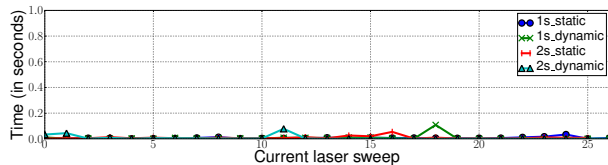Fig. 11. Inverse Kinematics (GAIK) computation plot

Fig. 12. KPIECE Motion Planning computation plot

VI. CONCLUSIONS AND FUTURE WORK

We presented a modular and distributed framework for a system that integrates online 3D perception with mobile manipulation for personal robotics applications. To validate its efficiency, we showed the integration of 3D collision maps and semantic annotations created in real-time from sensed laser data with a motion replanning framework as building blocks towards efficient and robust pick and place scenarios.

While this is still work in progress, as we plan to deploy and integrate our architecture with grasping planners to close the sensing-action loop, we already make our software available as an off-the-shelf component of the ROS project.

REFERENCES

[1] S. Srinivasa, D. Ferguson, M. V. Weghe, R. Diankov, D. Berenson, C. Helfrich, and H. Strasdat, "The Robotic Busboy: Steps Towards Developing a Mobile Robotic Home Assistant," in *Intl. Conference on Intelligent Autonomous Systems (IAS-10)*, July 2008.

[2] C. Borst, C. Ott, T. Wimbock, B. Brunner, F. Zacharias, B. Baeum, U. Hillenbrand, S. Haddadin, A. Albu-Schaeffer, and G. Hirzinger, "A humanoid upper body system for two-handed manipulation," in *IEEE Intl. Conference on Robotics and Automation (ICRA)*, 2007.

[3] T. Asfour, K. Regenstein, P. Azad, J. Schroeder, A. Bierbaum, N. Vahrenkamp, and R. Dillmann, "ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control," in *IEEE-RAS Intl. Conference on Humanoid Robots (Humanoids 2006), Genoa, Italy, December*, 2006.

[4] D. Katz, E. Horrell, Y. Yang, B. Burns, T. Buckley, A. Grishkan, V. Zhylkovskyy, O. Brock, and E. Learned-Miller, "The UMass Mobile Manipulator UMan: An Experimental Platform for Autonomous Mobile Manipulation," in *IEEE Workshop on Manipulation for Human Environments, Philadelphia, USA, August*, 2006.

[5] M. Quigley, E. Berger, and A. Y. Ng, "STAIR: Hardware and Software Architecture," in *AAAI 2007 Robotics Workshop, Vancouver, B.C, August*, 2007.

[6] H. Nguyen, A. Jain, C. Anderson, and C. C. Kemp, "A Clickable World: Behavior Selection Through Pointing and Context for Mobile Manipulation," in *IEEE/RJS Intl. Conference on Intelligent Robots and Systems (IROS)*, 2008.

[7] R. B. Rusu, B. Gerkey, and M. Beetz, "Robots in the kitchen: Exploiting ubiquitous sensing and actuation," *Robotics and Autonomous Systems Journal (Special Issue on Network Robot Systems)*, 2008.

[8] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, June 2005.

[9] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.

[10] K. Konolige, "A gradient method for realtime robot control," in *IEEE/RJS Intl. Conference on Intelligent Robots and Systems*, 2000.

[11] B. P. Gerkey and K. Konolige, "Planning and control in unstructured terrain," in *ICRA Workshop on Path Planning on Costmaps*, 2008.

[12] I. A. Şucan and L. E. Kavraki, "Kinodynamic Motion Planning by Interior-Exterior Cell Exploration," in *Intl. Workshop on the Algorithmic Foundations of Robotics*, Guanajuato, Mexico, December 2008.

[13] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, August 1996.

[14] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *IEEE Intl. Conference on Robotics and Automation*, April 2000, pp. 995–1001.

[15] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *Intl. Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[16] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *IEEE Intl. Conference on Robotics and Automation*, vol. 3, April 1997, pp. 2719–2726.

[17] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *Intl. Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, March 2002.

[18] G. Sánchez and J.-C. Latombe, "A single-query bi-directional probabilistic roadmap planner with lazy collision checking," *Intl. Journal of Robotics Research*, vol. 6, pp. 403–417, 2003.

[19] S. Tabandeh, C. Clark, and W. Melek, "A genetic algorithm approach to solve for multiple solutions of inverse kinematics using adaptive niching and clustering," *IEEE Congress on Evolutionary Computation*, pp. 1815–1822, July 2006.

[20] K. I. Tsianos and L. E. Kavraki, "Replanning: A powerful planning strategy for hard kinodynamic problems," in *IEEE/RJS Intl. Conference on Intelligent Robots and Systems*, 2008, pp. 1667–1672.