Generating and Executing Hierarchical Mobile Manipulation Plans

Sebastian Stock, Martin Günther
Osnabrück University, Germany

Joachim Hertzberg
Osnabrück University and DFKI-RIC Osnabrück Branch, Germany

Abstract

To execute complex tasks with a mobile robot in challenging environments, task planning and execution monitoring play a decisive role. This paper presents the integration of an off-the-shelf HTN planner into a robotic system which is aimed at enabling a service robot to learn from experiences. For plan execution, a state-machine-based approach is employed. The system has successfully been demonstrated on a PR2 robot in a restaurant scenario.

1 Introduction

With the increasing capabilities of mobile service robots, planning and execution are crucial for the performance of complex tasks. However, the integration of planning and execution is not straightforward: the resulting plans and their execution should be robust and safe for the robot and people in the environment, the plans should be easily understandable for the user, and the planner should create plans for arbitrary starting configurations (if a plan exists). Furthermore, in the ongoing FP7 project RACE,we aim at enabling the robot to learn from experiences and increase its robustness based on that. For that purpose an ontology-based architecture and knowledge representation framework has been developed in RACE [1]. In our evaluation scenario, a PR2 robot operates as a waiter in a restaurant environment. Typical tasks include serving a cup of coffee to a guest or clearing the table after the guest has left.

In contrast to classical planning, where we have a flat representation of the possible actions in the planning domain, Hierarchical Task Network (HTN) planning distinguishes non-primitive and primitive tasks [2]. Primitive tasks are like actions in classical planning that can directly be executed. Non-primitive tasks are decomposed by methods into subtasks. The planner gets a goal task and uses its methods to decompose it into subtasks until only primitive tasks are left that can directly be executed.

We have chosen the HTN planning approach because of several advantages it has in our domain: the plan generation is fast, so the user does not have to wait long until the robot starts acting; because of their hierarchical structure, the resulting plans can easily be inspected by the user; and it is possible to learn HTN methods based on previous execution traces. Therefore, we have integrated the HTN planner SHOP2 [3] into the RACE architecture.

The remainder of this paper has the following structure: after a discussion of related work, the RACE architecture, the integration of SHOP2, and state machine based plan execution will be described. This is followed by sections about modeling our restaurant environment and an evaluation of the system for the tasks of serving a mug of coffee to a guest and clearing all mugs from a table. The paper concludes with a summary and outlook.

2 Related Work

The CRAM [4] system provides a framework for reasoning and high level robot control for autonomous mobile robots in household environments. Complex robot control programs can be created and executed with it. Instead of using a dedicated planner, the robot's plans are already specified as action recipes in the CRAM Plan Language (CPL) and it needs to reason about the plans in order to execute them. Like HTNs the plans have a hierarchical structure. Part of CRAM is the KnowRob [5] knowledge processing system which uses a knowledge representation based on OWL-DL and provides tools to reason about that knowledge. This can be used to infer which objects to use for an action, where to find that object and positions where to execute an action.

Awaad et al. [6] aim at using functional affordances to make robot task planning and execution more robust. They present this idea for the task of watering plants, which would normally fail, if no watering can is available. They show that functional affordances could be used to modify the plan in a way that it uses a tea pot instead of the watering can. In order to do this, they propose to use a modified HTN planning algorithm that reuses the procedural knowledge of the planning methods and finds object substitutes. Therefore, they extend the JSHOP2 planner.

Another possible way to connect HTN planning and an OWL ontology is described in [7]. It deals with the prob-

lem that some of the contents of planning domains are irrelevant for the planning process, but will increase the time needed for plan generation. To deal with this, all necessary information is stored in OWL-DL and a reasoner generates the problem description for the planner by filtering only the relevant information. This way, even in large domains the problem description and therefore also the planning time remains small.
The HTN planner SHOP2 has been used in very different applications of which an overview is presented in [8]. These include evacuation planning, material selection for manufacturing, project planning and automated composition of web services.

3 HTN planning and plan execution in the RACE architecture

3.1 The RACE architecture

In the RACE project we have developed a robot architecture that aims at enabling the robot to learn from experiences and thereby improve its performance [1]. **Figure 1** shows the RACE architecture. It addresses several purposes: it needs to be suitable to allow planning and execution of complex tasks in dynamic environments and therefore allow the integration of different kinds of reasoners; and it also needs to provide means to record experiences of execution traces, learn from them and use this learned knowledge in the future. In this paper we focus on the former. The latter and the overall goals of RACE are described in more detail in [1]. For communication of the modules we use ROS [9] as a robot framework.

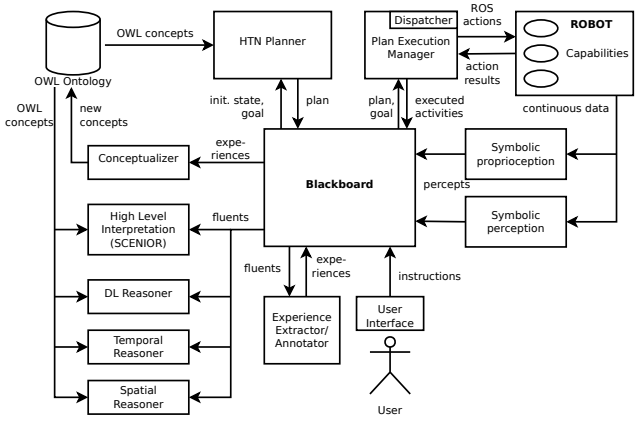

Figure 1: The RACE control software architecture

The central piece in this architecture is the Blackboard. It is implemented on top of an RDF triple store and can be thought of as the A-Box of an OWL-DL ontology, where each individual additionally has a temporal interval where it is valid. The ontology provides a common representation format on which nearly all modules are based. This way it also contains all the information that is relevant for planning: the different areas in the room and their connections, the current state of the robot including the position of the torso, the arms and the position in the room, the position of guests, mugs and other objects and even the tasks and activities themselves.
The symbolic proprioception and symbolic perception modules update the robot's internal state by sending fluents to the Blackboard. A fluent is an instance of a concept represented in our OWL ontology together with a start and finish time to indicate when the this instance is true. The start and finish time are both divided into an earliest and latest time point. Fluents are used by all modules to communicate with the Blackboard. This way, the OWL ontology serves as a common representation formalism for the different modules. An example of a fluent is given in **Listing 1**. Here, the fluent is represented in YAML format in which we store fluents in order to record experiences of execution traces. For the communication between the modules ROS messages are used. The field `Class_Instance` contains the OWL concept of that fluent and the name of an instance of that concept. The properties are represented in a list of triples with name of the property, the type of the filler and the instance of the filler. The fluent in Listing 1 represents an instance of the concept `RobotAt` indicating that the robot `trixi1` is currently at the ManipulationArea `mae3`. The value `0.0` of the finish time represents that the fluent is active. When the robot leaves the area the fluent will be closed by publishing it again with an updated `FinishTime`. Symbolic proprioception contains modules for publishing the position of the robot's torso, the position of the robot's arms and grippers and a symbolic representation of the robot in the room, i.e., the area it currently is in.

```
!Fluent
Class_Instance: [RobotAt, robotAt7]
StartTime: [139.744, 139.744]
FinishTime: [0.0, 0.0]
Properties:
   - [hasRobot, Robot, trixi1]
   - [hasArea, Area, mae3]
```

Listing 1: Example of a fluent representing the robot's current position.

For plan generation, the user inserts the goal task as an instruction via a user interface into the Blackboard. This triggers the HTN planner which creates a *problem description* by querying all the information that is potentially needed for planning from the Blackboard. The planner's problem description contains changing information like the robot's current state published by the proprioception modules, the positions of objects and guest but also static knowledge like the fragmentation of the room and the tables into smaller areas and their connection. The planner needs to convert the OWL ontology format into its own format, i.e., ground atoms in SHOP2 syntax. Furthermore, the planner needs a domain description which is currently predefined but should later in the project be learned by the conceptualizer and extracted from the OWL ontology. Based on this information the planner creates a plan as a sequence

of ground actions and writes the plan into the Blackboard. This triggers the plan execution manager, which has to dispatch the planned tasks to the robot.

3.2 HTN planning

In HTN planning [10], the planner gets as input a planning domain, the current state of the world and a task that has to be accomplished. The domain consists of *methods* and *operators*. There are two different kinds of *tasks*: *primitive* and *compound* tasks. Primitive tasks can directly be executed with operators. The operators consist of the name of the task to which they can be applied, a list of preconditions and a list of effects. When the planner plans for a primitive task, it checks the preconditions of the operator that is relevant for that task if the operator is applicable in the current state. When applying the operator, its effects are added (positive effects) to or deleted (negative effects) from the planner's internal state. Compound tasks are handled in a different way. They need to be decomposed into smaller subtasks. This is done using methods. A method consists of the name of the compound task to which it can be applied, a list of preconditions and a list of tasks to which the original task will be decomposed. An example of a method for moving an object to a given area is presented in **Listing 2**. The name of the task is `move_object`. Variables begin with a question mark. This method checks with its preconditions if the object is on some area and binds the variables `?on` and `?obj` accordingly. If this is the case it decomposes the compound task `move_object` into the compound tasks `get_object_w_arm` and `put_object`.

```
(:method (move_object ?obj ?toArea)
 ((Instance On ?on)
  (HasPhysicalEntity ?on ?obj)
  (HasArea ?on ?fromArea))       ; precond.
 ((get_object_w_arm ?obj leftArm1)
  (put_object ?obj ?toArea)))    ; subtasks
```

Listing 2: Definition of a method for moving an object to an area.

We use the well known planner SHOP2 [3]. It decomposes the tasks in the same order as they will be executed later. This makes the planning process very fast. The planner works by starting with the goal task and decomposing it into subtasks which are then likewise decomposed with methods or, in case of primitive tasks, operators are applied and the planner's internal state is updated. A plan is found when only actions are left leading to a decomposition tree of tasks with ground actions as its leaves. The resulting plan is then a sequence of ground actions which accomplishes the goal task when successfully executed. SHOP2 can also provide the grounded preconditions and effects of used operators and the hierarchy how tasks have been decomposed. We extract this information from SHOP2 and record it as part of the robot's experience to be able to use it for learning new methods.

3.3 Plan Execution: A state machine based approach

For plan execution, we use a state machine based approach. Therefore, we transform the plan into a SMACH state machine. SMACH [11] is a Python library for creating and executing hierarchical state machines. We modeled the operators and SMACH states in a way that the created states directly correspond to the planned ground actions. Furthermore, they correspond to the basic robot capabilities. This makes implementing the states straightforward: for most states, e.g., tucking both arms, a ROS action that accomplishes the desired behavior on the robot can directly be called, or only few additional queries to other modules, e.g., to get the pose of an object, are required. The successful or failed execution and the start and finish time of an action are published to the Blackboard. This is relevant for recording experiences. Based on these experiences, new HTN methods or modifications of them will be learned to improve robustness and performance of the system.

Encapsulating each action into its own SMACH state has a number of advantages. First, it decouples the abstract task planning from the details of the execution and creates a clear interface between these two levels. This reduces the complexity of the domain both for the planning domain modeler as well as for the learning algorithm that has to work with this domain. Second, it leaves the decision whether an action has failed (which is very specific to the action) to the implementer of the action.

If an action fails, it is most often caused by a change in the environment that has not been foreseen during plan generation. For example, a guest might have removed an object. In this case, the execution of the current plan is marked as failed and the overall task is again added to the Blackboard as a goal task. This triggers replanning resulting in a plan that is executable in the updated state of the environment and might involve using another object.

4 Modeling the domain

The demonstration scenario in the RACE project contains a counter and two tables. Enabling the robot to generate plans requires modeling the planning domain. Our planning operators directly correspond to basic capabilities of the PR2 robot. We have modeled the following eight operators. All operator names in this paper begin with an exclamation mark.

`!move_arm_to_side ?arm` Moves one arm to the side.

`!move_arms_to_carryposture` Moves the arms in front of the torso so that it can safely carry mugs.

`!move_base ?area` Drives with standard ROS navigation capabilities.

!move_base_blind ?area When in front of the table, the robot drives closer without obstacle avoidance.

!move_torso ?torsoposture Moves the torso to a low, middle or high position.

!pick_up_object ?object ?arm Grasps a given object when it is reachable.

!place_object ?object ?arm ?placingArea Puts an object that the robot is holding onto a specified area on the table.

!tuck_arms ?left_goal ?right_goal Tucks or untucks the arms.

The PR2 platform itself imposes several constraints that have to be addressed when modeling the HTN methods and operators: The torso should be up for manipulating objects to avoid restricting the possible arm motions too much because of possible collisions with the table top; for driving, on the other hand, the torso should be down to lower the robot's center of gravity. However, if we want to carry objects, the torso should be at a middle position to be able to move its arms in front of the robot without hitting its base. The arms should be tucked or at a carry posture while driving to prevent collision with obstacles and an arm should be at the side before manipulation. Furthermore, to manipulate objects on the counter or table, the robot has to drive closely to it. This is not possible with standard path planning because of its obstacle avoidance. Therefore, two different operators are necessary for driving, i.e., `move_base` which calls the corresponding action that already exists in ROS and `move_base_blind` which only moves straight.

This leads to the representation of the restaurant environment given in **Figure 2**. Different kinds of areas have been modeled in the OWL ontology and they are connected via properties. We have divided the top of the tables and counter into smaller *PlacingAreas* on which the robot can manipulate objects from corresponding *ManipulationAreas*. Adjacent to the ManipulationAreas are the *PreManipulationAreas* to which driving with standard navigation capabilities is possible. Furthermore, at the tables there are *SittingAreas* where the guests can sit and which are in turn connected to a left and a right PlacingArea. The bounding boxes of the areas and the connection between them are calculated by a module for spatial reasoning. The connections of areas can then directly be used by the planner, e.g., to decide which ManipulationArea to drive to in order to serve a coffee to a guest at a given SittingArea, without having to call the spatial reasoner as an external module during planning, which would increase the time needed for plan generation.

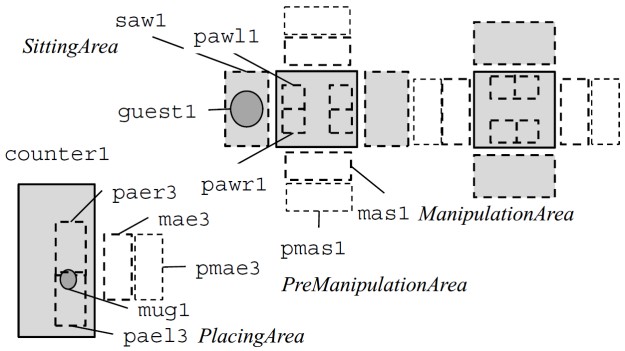

Figure 2: Representation of the restaurant environment with a counter and two tables.

The HTN methods have been modeled in a way to keep them general and reusable for other tasks as well. The top level method for the task of serving a coffee just selects a coffee mug and a PlacingArea that is suitable to place the coffee in front of the guest and then decomposes its task into the compound task `move_object`. The corresponding method is given in Listing 2. This method is also reused by other tasks, e.g., for the task of clearing the table. To comprehend the parts of a plan it is useful to have a look at the tree of decompositions from tasks into subtasks. This is can be very useful while modeling the domain or for the user to understand what the robot is doing. An example of the upper level of such a decomposition hierarchy for the task of serving a coffee is shown in **Figure 3**. The upper part of this hierarchy, i.e., the decompositions of `serve_coffee_to_guest` and `move_object` have already been discussed. In the same way, the task `get_object_w_arm` is decomposed to the compound tasks of driving to the PreManipulationArea and grasping `mug1` using the right arm. `put_object` is defined recursively. It is decomposed into driving to the PreManipulationArea of the table and the task `put_object`, again.

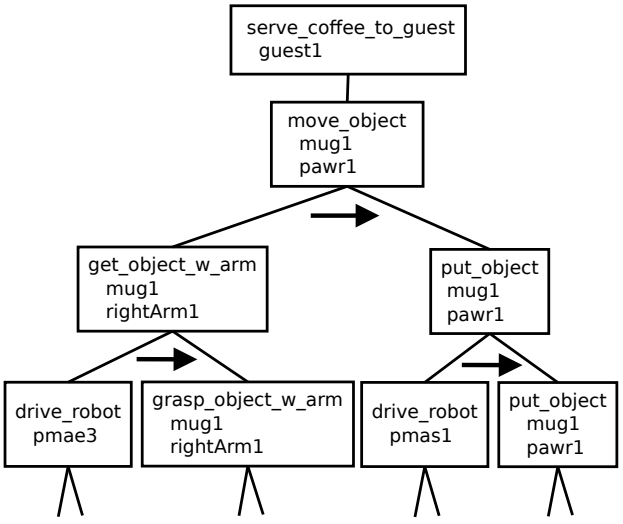

Figure 3: Example of a decomposition hierarchy for serving a coffee.

For grasping an object, the robot first has to assume a pose that is suitable for object manipulation before it can pick the object up and leave the manipulation pose again. An example of the whole decomposition hierarchy for grasping a mug is presented in **Figure 4**. Primitive tasks are marked green. The robot first has to move its torso up and untuck the right arm in order to be able to move the arm to the side afterwards. In this posture it drives more closely to the table. Now, it can pick up the mug and drive back again.

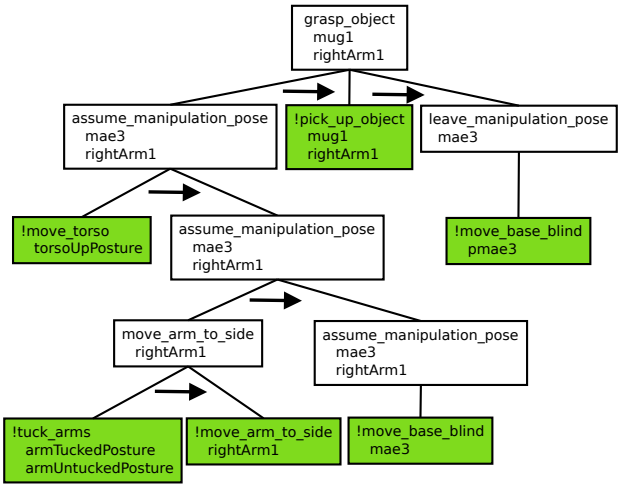

Figure 4: Example of a decomposition hierarchy for grasping a mug.

To be able to clear the table, only few additional methods had to added to the planning domain because they can reuse many of the methods and operators that have already been modeled for serving a coffee. Besides a method for the goal task `clear_table`, methods for the tasks `get_objects` and `put_objects` had to be added in order to enable the robot to carry objects with both arms at the same time if there are multiple mugs on the table.

5 Experiments

The planning and execution system has successfully been tested in simulation and on a real PR2 robot for the tasks of serving a coffee to a guest and clearing the table. For serving a coffee, three different starting configurations have been tested. First, the starting situation was the one displayed in Figure 2 with `mug1` on the left PlacingArea of `counter1` and the guest sitting at the west side of the left table. The plan given in **Listing 3** of ground actions has been created. The plan was successfully transformed to a state machine and executed. An image of the PR2 serving the mug in front of the guest can be seen in **Figure 5**. The robot was also able to generate a plan and successfully execute it in different starting configurations when the guest was sitting at the opposite side of the left table or the southern side of the right table.

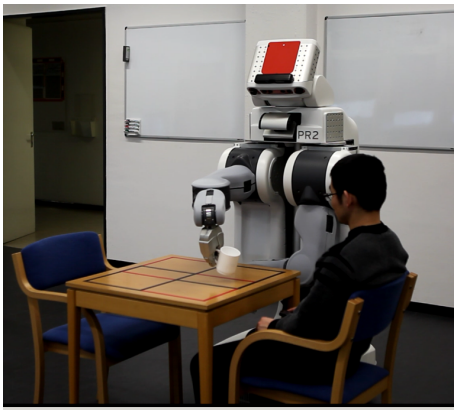

Figure 5: PR2 serving a coffee.

```
!move_torso torsoDownPosture
!tuck_arms armTuckedPosture
           armTuckedPosture
!move_base pmae3
!move_torso torsoUpPosture
!tuck_arms armTuckedPosture
           armUntuckedPosture
!move_arm_to_side rightArm1
!move_base_blind mae3
!pick_up_object mug1 rightArm1
!move_base_blind pmae3
!move_torso torsoMiddlePosture
!move_arms_to_carryposture
!move_base pmas1
!move_torso torsoUpPosture
!move_arm_to_side rightArm1
!move_base_blind mas1
!place_object mug1 rightArm1 pawr1
!move_base_blind pmas1
```

Listing 3: Plan for serving a coffee.

The second demonstration scenario was about clearing all mugs from a table. Two mugs were placed on different areas on the left table and the robot had to bring them to the counter. The plan contained 27 operators. Pictures from the successful execution of this task are shown in **Figure 6**. Because of the additional methods that guide the search in HTN planning, the time needed for plan generation is very low compared to the time the robot needs to execute its plans. For both tasks in the different starting configurations, planning itself takes less than one second.

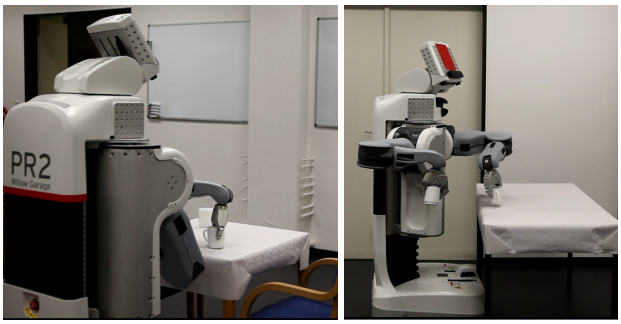

Figure 6: PR2 clearing a table. On the left side, the robot grasps a mug from the table and on the right side, it places two mugs on the counter.

Before that, additional time is needed to generate the problem description based on the robots current state. Therefore, several queries have to be sent to the Blackboard whose time duration depends on the overall load of the system. The complete duration for generating a plan has always been less than ten seconds. Thus, planning has no big influence on the execution time of our demonstration scenarios.

6 Conclusions and Outlook

In this paper we presented the integration of an off-the-shelf HTN planner together with a state-machine based approach for plan execution into an architecture that is aimed at learning from experiences. The approach was demonstrated on a PR2 robot in a mobile manipulation domain. Benefits of HTN planning are the low runtime for plan generation, because of the hierarchical structure the plans are easily inspectable by the user, and it allows to learn or modify further methods to achieve increased robustness. The use of an OWL ontology as a common representation format makes the planning domain and the robot's capabilities transparent to the different modules. It is the basis for the Blackboard which stores the robot's internal knowledge about occurrences in the environment. In future work, we will integrate spatial and temporal reasoning into the planning process and connect the planner and the plan executor more closely to easily switch between the two and allow plan repair.

Acknowledgment

This work is supported by the EC Seventh Framework Program under Project RACE (Robustness by Autonomous Competence Enhancement, grant agreement no. 287752).

References

[1] S. Rockel, B. Neumann, J. Zhang, K. S. R. Dubba, A. G. Cohn, Š. Konečný, M. Mansouri, F. Pecora, A. Saffiotti, M. Günther, S. Stock, J. Hertzberg, A. M. Tomé, A. J. Pinho, L. S. Lopes, S. von Riegen, and L. Hotz, "An ontology-based multi-level robot architecture for learning from experiences," in *Designing Intelligent Robots: Reintegrating AI II, AAAI Spring Symposium*, 2013.

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

[3] D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *Journal of Artificial Intelligence Research*, vol. 20, pp. 379–404, 2003.

[4] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments," in *Proc IROS*, Taipei, Taiwan, 2010, pp. 1012–1017.

[5] M. Tenorth and M. Beetz, "KnowRob – knowledge processing for autonomous personal robots," in *Proc IROS*, 2009, pp. 4261–4266.

[6] I. Awaad, G. K. Kraetzschmar, and J. Hertzberg, "Affordance-Based Reasoning in Robot Task Planning," in *Planning and Robotics (PlanRob) Workshop ICAPS-2013*, Rome, Italy, 2013.

[7] R. Hartanto and J. Hertzberg, "Fusing DL reasoning with HTN planning," in *KI 2008: Advances in Artificial Intelligence*. Springer, 2008, pp. 62–69.

[8] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, H. Munoz-Avila, J. W. Murdock, D. Wu, and F. Yaman, "Applications of SHOP and SHOP2," *IEEE Intelligent Systems*, vol. 20, no. 2, pp. 34–41, Mar. 2005.

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

[10] K. Erol, J. Hendler, and D. Nau, "HTN Planning: Complexity and expressivity," in *In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. AAAI Press, 1994, pp. 1123–1128.

[11] J. Bohren, R. Rusu, E. Gil Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer, "Towards autonomous robotic butlers: Lessons learned with the PR2," in *Proc ICRA*, 2011, pp. 5568–5575.