Albert retail store simulation

Storing, sorting, and discarding

Ayush Kulshreshtha
MSc Robotics TU Delft
5695821

Anish Diwan
MSc Robotics TU Delft
5706580

Leonoor Verbaan
MSc Robotics TU Delft
5415721

I. Introduction

While robots are increasingly adopted in the retail industry, automating in-store logistics processes still remains a challenge. Some of the main challenges lie in handling individual products, dynamic environments, and challenging conditions while making a robot follow some set of basic rules. Data driven or function approximation style methods fail to be effective in such environments because of the sheer number of possible state-action combinations and the lack of any symbolic knowledge. Data driven methods can identify objects but it is still quite challenging for them to make meaningful task specific connections between them. One way to solve such a problem would be symbolic reasoning using a knowledge base curated by task-experts. In this report we provide a knowledge driven retail store simulation where a robot is tasked to handle objects while adhering to some basic rules. We use *Prolog* to define propositional logic rules for the store, and the *Planning Domain Definition Language (PDDL)* to then convert these rules to actionable outputs through predicates and actions. This report is divided into individual sections discussing each of these aspects in detail.

A. Albert's Tasks

In today's retail industry, employees often face tedious and time-consuming tasks such as stocking shelves, fulfilling orders, and keeping track of product shelf lives while also managing other job responsibilities. To alleviate employees from some of these tasks, we propose a solution by making the Albert robot carry out some of these repetitive tasks using a store specific knowledge base and symbolic reasoning. The tasks that Albert can perform are storing, sorting, and discarding. Albert's software stack is developed, tested and tweaked in a simulation environment which consists of empty tables, and shelves scattered with items. Some of the items have an expiration date. At present, this is modelled as a "expiry priority level" predicate ranging from levels 1 (highest priority) to level 5 (least priority) instead of an actual calendar date. The environment also contains a waste basket where Albert is supposed to discard any items that are expired or are empty. At the start of the simulation, all items are placed on the shelves. Albert is supposed to reason about each item and infer its place on one of the tables as per the defined store rules and following tasks:

Sorting items based on their product category
(Eg: milk $\Rightarrow$ beverage; hagelslag $\Rightarrow$ Snack) to be placed on a relevant table (Refered to as the drink-table & snack-table), if the table has room.

Sorting items based on their expiry priority levels
Within a certain product category, items will be sorted based on their expiry date and the relevant table would be stocked in a *First in First out (FIFO)* manner. This means that the items that expire the earliest will be processed first. For now, the items will be placed in a line on the table, with waypoints as per their assigned priority level. Future versions could look into stocking items towards the back/front of shelves.

Gathering items for online orders
In the event of an online order, Albert would pick the relevant items from the relevant shelf, and place it on their dedicated table. For the current scope of the project, this task has not been differentiated from the sorting tasks since all picking and placing is being done from the shelves to the tables.

Discarding empty or expired items
In all the above mentioned tasks, empty / expired items would be discarded by Albert in the basket, if the basket has room.

Failure Recovery
(Future scope) Albert would be able to recover from any failure in the task pipeline, and reset to its start position to attempt the task again.

Figure 1 illustrates the retail store simulation environment. The following sections elaborate on the robot's knowledge base, reasoning with PDDL, simulation and are followed by results and a discussion. Further, the appendix provides details on the implementation related aspects of the simulation environment and the PDDL planner, and discusses some technical specifics that do not find a suitable spot in the body of this report.

II. Methods

The simulation method can be described as comprising the following main components: *(A) task modeling* and, *(B) planning and Prolog*, and *(C) Planning in PDDL*. In the subsequent

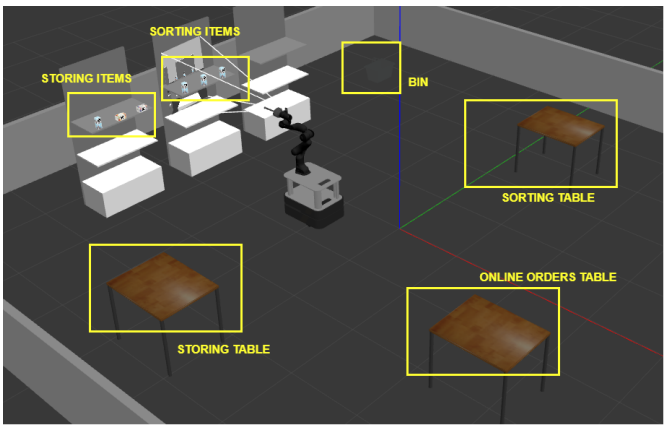

Fig. 1. Retail store simulation environment with labels for shelves, items, and tables

sections, these components will be discussed in greater detail, highlighting their respective roles and contributions to the simulation methodology.

A. Task Modelling & Rules in Propositional/First-Order Logic

Modelling of the tasks and store rules is done in propositional logic with some of these propositions being translated to first order logic predicates to carry out actions. We chose to use two different logic based approaches upon receiving feedback during the project presentation. Our initial approach used multiple very similar predicates to carry out the same action in different contexts. For instance, three place actions were defined based on where the robot was required to place an item. This was done because of a lack of PDDL feature support in the available solvers and is discussed in detail in the appendix. The current method uses distinct actions and offloads the task-specific reasoning to propositional logic done via Prolog. This also decouples task-specific reasoning (Prolog) from the general purpose planning (PDDL) and improves the re-usability of our code. Ultimately, the rule-based outputs from Prolog are translated to predicates and actions in PDDL which are executed with ROSPLAN[1]. We omit the propositional logic rules for brevity and to avoid re-writing the same basic ideas from the previous section. The store rules are written in first order logic and natural language as follows.

Store rules

- `drink(tea), snack(hagelslag)`
  Tea is a drink, Hagelslag is a snack.
- `drinktable(table1), snacktable(table2), bin(basket)`
  Table1 and Table2 are dedicated as drink-table and snack-table, while the basket is dedicated as the bin.
- $\forall x\ drink(x) \vee snack(x) \Rightarrow item(x)$
  Both drinks and snacks are items.
- $\forall x\ item(x) \wedge (\text{empty}(x) \vee \text{expired}(x) \Rightarrow \text{bad}(x)$
  If an item is expired or an item is empty, then it is bad.
- $\forall x \quad \text{bad}(x) \wedge \text{has-room}(\text{basket}) \wedge bin(basket) \Rightarrow \text{should-be-discarded}(x, basket)$
  If an item is bad and there is room in the bin, then it goes in the bin.
- $\forall s\ storage(s) \wedge \text{has-room(s)} \Rightarrow \text{is-stockable}(s)$
  If there is room on a storage area, then it is stockable.
- $\forall x \forall s \quad \neg bad(x) \wedge \text{drink}(x) \wedge drinktable(s) \wedge \text{is-stockable}(s) \Rightarrow \text{store-in-shelf}(x, s)$
  If a drink is not bad, and there is room on a drink-table, then the drink should be stored on the drink-table.
- $\forall x \forall s \quad \neg bad(x) \wedge \text{snack}(x) \wedge snacktable(s) \wedge \text{is-stockable}(s) \Rightarrow \text{store-in-shelf}(x, s)$
  If a snack is not bad, and there is room on a snack-table, then the snack should be stored on the snack-table.

The above-specified store rules and facts are used to model the knowledge base in Prolog as illustrated in *Figure 2*. We utilise predicates to only model the symbolic names and waypoints of the items, with the actual waypoints information being stored within yaml files, that are parsed by *ROSPLAN*. This prevents any waypoints from being hardcoded into the Prolog file and this formalisation allows abstraction of the actual waypoint data from the symbolic planning process.

```
%Facts
drink('AH_thee_mango_tag_11_00030'). %Tea is a drink
drink('AH_thee_bosvruchten_tag36_11_00049').
snack('AH_hagelslag_mel_tag36_11_00000'). %Hagelslag is a snack
snack('AH_hagelslag_mel_tag36_11_00001').
snack('AH_hagelslag_mel_tag36_11_00002').
snack('AH_hagelslag_mel_tag36_11_00003').
drinktable(wp_table_1).  %drink-table
snacktable(wp_table_2).  %snack-table
bin(basket). %bin

at_location('AH_thee_mango_tag_11_00030', cab_2_shelves).
at_location('AH_thee_bosvruchten_tag36_11_00049', cab_2_shelves).
at_location('AH_hagelslag_mel_tag36_11_00000', cab_2_shelves).
at_location('AH_hagelslag_mel_tag36_11_00001', cab_2_shelves_0).
at_location('AH_hagelslag_mel_tag36_11_00002', cab_2_shelves_0).
at_location('AH_hagelslag_mel_tag36_11_00003', cab_2_shelves_0).

expired('AH_hagelslag_mel_tag36_11_00000'). %This Hagelslag is expired
empty('Random_hagelslag').
has_room(wp_table_1,3).
has_room(wp_table_2,3).
has_room(basket,3).
```

Fig. 2. Facts in the prolog knowledge base

B. Planning in Prolog

Sorting based on product category: One major inference task in Prolog is for Albert to allocate the correct storage location for any item, whether one of the tables or the bin, based on the store rules. This can be achieved within the domain of First Order Logic, with the Modus Ponens inference rule.

```
%Rules
%drinks and snacks are items
item(X):-drink(X);snack(X).

%If an item is expired or an item is empty, then it is bad.
bad(X) :- item(X), (expired(X);empty(X)).

good(X) :- item(X), \+(expired(X);empty(X)).

%If item is bad and there is room in a bin, then the item goes in the bin.
%bin_loc(X) :- bad(X).
final_location(X,Y):- bad(X), bin(Y), stockable(Y).

%If location X has room greater than 0, than it is stockable
stockable(X):-has_room(X, Y), Y>0.

%If a drink is not bad and there is a drinktable that is stockable,
%then the item goes on the drinktable.
%drink_loc(X):- good(X), drink(X), stockable(wp_table_1).
final_location(X,Y):- good(X), drink(X), drinktable(Y), stockable(Y).

%If a snack is not bad and there is a snacktable that is stockable,
%then the item goes on the snacktable.
%snack_loc(X):- good(X), snack(X), stockable(wp_table_2).
final_location(X,Y):- good(X), snack(X), snacktable(Y), stockable(Y).
```

Fig. 3. Rules in the prolog knowledge base

Sorting based on expiry priority levels: In our knowledge base, we currently manually assign a priority level to each item, modelled as the predicate `priority_expiry/2`. We also store a separate list of these priority level integers in the knowledge base. The idea is that as more and more products with expiry levels keep getting added or the expiry levels get changed, it would become increasingly difficult to manually enter a sorted priority list in the knowledge base. Our solution is to sort this list using a built-in Prolog arithmetic function, and then recursively call the first element of the list to retrieve the most urgent priority level / date.

```
%1 is highest priority, 5 is least
priority_expiry('AH_hagelslag_mel_tag36_11_00001', 1).
priority_expiry('AH_hagelslag_mel_tag36_11_00002', 5).
priority_expiry('AH_hagelslag_mel_tag36_11_00003', 2).

sortlist([1, 5, 2]).
sorted(L):- sortlist(A), sort(A, L).
head(X):- sorted(L), L = [X | T], length(L, Len), Len>0.
priority(Y):-head(Y).
sortlist(T):- sorted(L), L = [X | T], length(L, Len), Len>1.
```

Fig. 4. Facts & Rules for sorting in the prolog knowledge base

The Knowledge base can be queried to retrieve all items with the specified priority level / date, and these would be stored at a dedicated waypoint on table3, corresponding to that priority level. A known bug in our implementation is that the current recursive calls in Prolog do not terminate even after reaching an empty list, which eventually leads to the stack limit being exceeded. This is mitigated through exception handling in the python file querying the knowledge base.

Query the knowledge base using pyswip: To query this knowledge base, we use the pyswip python library for Prolog. This allows us to directly obtain query outputs as variables in python. For instance `prolog.query("priority(X), priority_expiry(Y,X)")` returns a list with pairs X, Y containing the priority number and the item from reasoning on sorting.
`prolog.query("final_location(X,Y)")` gives the required place of locations of each item.

C. Planning in PDDL

As indicated above, task-specific reasoning is carried out in Prolog while general planning is carried out in PDDL. This section discusses the PDDL planner, explaining the predicates, actions, and the problem. To combine both PDDL and Prolog, we use the ROSPlan knowledge base and its associated services. The technicalities of these are explained in the appendix. The PDDL domain only includes predicates that are general in nature and can apply to any pick-place problem irrespective of the environment. The types only include the robot, the object, the gripper, and waypoints. Further, predicates only specify where the robot is, what it is holding, and where an object is placed. The reasoning on the specific objects and their place locations is handled in Prolog. The predicates to do this are as follows.

```
(define (domain albert-does-stuff)

    (:requirements
        :typing
        :durative-actions
        :negative-preconditions
        :disjunctive-preconditions
        )

    (:types
        waypoint
        robot
        object
        gripper
    )

    (:predicates
        (visited ?wp - waypoint) ; To define which states the robot has already visited
        (robot-at ?v - robot ?wp - waypoint) ; To define which robot is at which waypoint
        (object-at ?obj - object ?wp - waypoint) ; To define which object is at which waypoint
        (is_holding ?g - gripper ?obj - object) ; To define which gripper is holding which object

        (free ?g - gripper) ; To define which gripper is free. This is needed to simplify if the gripper
                            ;is engaged or not. Without this, you would have to loop through all objects

        (object_placed ?obj - object ?wp - waypoint) ; To define if an object is placed at its destination
    )
```

Fig. 5. PDDL domain setup

The robot's actions are defined to be durative as usual. In terms of actions we find the following points noteworthy.

1) We modify the "move" action to have an additional condition that requires the gripper to be free. Hence, the robot only takes the "move" action when its gripper is free. Additionally, we add an action called "carry" to get called when the robot's gripper is holding an item. Having two different actions enables us to differentiate their outcomes. We need one action to change the location (defined by the predicates robot-at and object-at) of both the robot and the object (done via carry) and another action to just change the location of the robot (done with move). This seemed to solve the issue where the planner would route the robot to an intermediate waypoint. With the addition of "carry", the planner outputs an optimal plan.
2) The addition of "carry" also enables us to be more specific in the conditions of our "place" action. An item can only be placed if both the item and the robot are

at the place waypoint. Before the addition of carry, this could not have been done without ambiguity.

3) The carry action also uses the moveBase interface and is handled by a node that is very similar to the ”move” action.

A plan from this domain definition is illustrated in 7. This plan is generated using the defauly `popf` planner as per [2]. Finally, the problem file initializes the robot and all objects according to the object and waypoint names mentioned in the objects.yaml and waypoints.yaml files. The exception here is that the problem file is initialized in an empty state and does not have any goals at the start. Goals are added via the ROSPlan knowledge base service `rosservice call /rosplan_knowledge_base/update`. A brief explanation of the integration between Prolog and PDDL is mentioned in the appendix.

```
Number of literals: 36
Constructing lookup tables: [10%] [20%] [30%] [40%] [50%] [60%] [70%] [80%] [90%] [100%] [110%]
Post filtering unreachable actions:  [10%] [20%] [30%] [40%] [50%] [60%] [70%] [80%] [90%] [100%] [110%]
No analytic limits found, not considering limit effects of goal-only operators
72% of the ground temporal actions in this problem are compression-safe
Initial heuristic = 18.000
b (17.000 | 2.000)b (16.000 | 6.003)
Resorting to best-first search
b (17.000 | 2.000)b (16.000 | 4.001)b (14.000 | 6.002)b (12.000 | 8.003)b (11.000 | 10.004)b (10.000 | 12.005)b (8.
; States evaluated: 118
; Cost: 24.011
; Time 0.02
0.000: (move tiago wp0 cab_2_shelves rightgrip)  [2.000]
2.001: (pick tiago cab_2_shelves ah_hagelslag_mel_tag36_11_00000 rightgrip)  [2.000]
4.002: (carry tiago cab_2_shelves basket ah_hagelslag_mel_tag36_11_00000 rightgrip)  [2.000]
6.003: (place tiago basket ah_hagelslag_mel_tag36_11_00000 rightgrip)  [2.000]
8.004: (move tiago basket cab_2_shelves rightgrip)  [2.000]
10.005: (pick tiago cab_2_shelves ah_thee_bosvruchten_tag36_11_00049 rightgrip)  [2.000]
12.006: (carry tiago cab_2_shelves wp_table_2 ah_thee_bosvruchten_tag36_11_00049 rightgrip)  [2.000]
14.007: (place tiago wp_table_2 ah_thee_bosvruchten_tag36_11_00049 rightgrip)  [2.000]
16.008: (move tiago wp_table_2 cab_2_shelves rightgrip)  [2.000]
18.009: (pick tiago cab_2_shelves ah_thee_mango_tag_11_00030 rightgrip)  [2.000]
20.010: (carry tiago cab_2_shelves wp_table_2 ah_thee_mango_tag_11_00030 rightgrip)  [2.000]
22.011: (place tiago wp_table_2 ah_thee_mango_tag_11_00030 rightgrip)  [2.000]
```

Fig. 6. Plan with ”carry”

III. Results & Discussion

Video demonstrations of this project can be found on the following link: https://drive.google.com/drive/folders/1TJoJHYEcYsMU5gFstFI1KYBQUuZQK1xn?usp=sharing

The Gitlab repositories related to this project are:

1) https://gitlab.tudelft.nl/cor/ro47014/2023_course_projects/group_02/retail_store_simulation
2) https://gitlab.tudelft.nl/cor/ro47014/2023_course_projects/group_02/rosplan
3) https://gitlab.tudelft.nl/cor/ro47014/2023_course_projects/group_02/retail_store_skills

The results of the prolog queries can be seen in the generated `new_executor.bash` script and as printed outputs while executing the PROLOG.py script. We skip these here for brevity. They are shown in the demo videos above. The PDDL plan is very similar to the plan shown above. This too is hence not shown here for brevity. In our experience, the simulation was still a significant hurdle. Significant tuning of waypoints is needed along with proper intermediate poses, to get albert to perform adequetly. However, from a knowledge representation and symbolic reasoning perspective, the results are quite promising. Prolog queries provide a simple solution and are capable of fulfilling item placement, sorting items, and assigning place waypoints without any hard-coding. Similarly, the PDDL plans are optimal (without any redundant intermediate moves) and there seems to be seamless integration between Prolog and PDDL. We discuss possible future additions in the next section and touch on implementation related improvements – wherever apparent to us – in the appendix.

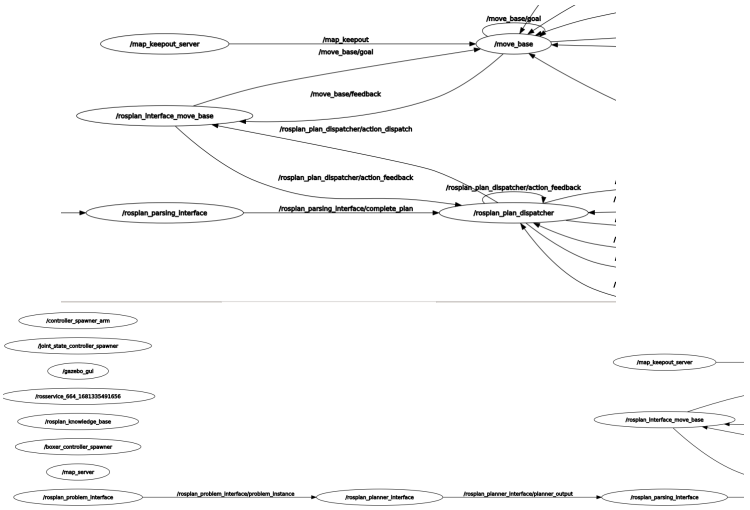

Fig. 7. RQT Graph, ROSPLAN

A. Further extensions

Possible extensions could be:

ROS parameter server and date parser
The items with an expiry date are now modelled with a ”expiry priority level” predicate ranging from levels 1 (highest priority) to level 5 (least priority). In order to optimize on filling items in FIFO, actual calendar dates could be implemented. Future versions of the simulation can use the ROS parameter server and a date parser script to obtain this date to expiration for each item.

Numeric fluents in PDDL and a better solver
The process of mapping between states in situation calculus can be achieved through the implementation of *numeric-fluents*, which facilitate integer valued predicates. This was a part of our initial planning scheme but was dropped due to issues related to conditional-effects not being supported. Together with numeric fluents and conditional-effects, we can incorporate a notion of quantity in the simulator. At present, quantities are handled via propositions (as full or empty or has_room). However, this could be changed to integer valued quantities and logic based sorting/reasoning in PDDL.

Storage space
One of our design choices is to model the room within the table or the bin as an integer. This was done so that after an item has been successfully placed onto its location (as conveyed by the place server), we can make use of the arithmetic capabilities in Prolog to decrease this integer by one. This has currently not been implemented in our solution and we model the tables and the bin to have an available

storage space greater than that required within the scope of the simulation. However, this is a potential extension to the planning that can be explored in the future.

Integrating PDDL with KnowRob.
To make a more sophisticated system that can reason in complex environments we can separate the knowledge representation framework (KnowRob) and the modeling of planning problems. By using KnowRob with RosProlog, prolog query objects can be created and stored in variables to organize information about the environment and tasks in re-usable knowledge chunks. This extension could make adding new items, rules or tasks more efficiently because then no whole file structures need to be changed.

Integrating a Semantic Realization and Refreshment Module (SRRM).
This module has the ability to discover and select entities in the robot's environment that are semantically equivalent or similar to entities related to the robot's plan by measuring the degree of similarity between entities. SRRM supports robotic task planning by generating approximate plans to solve tasks when no exact plan can be generated using the initial and goal states. This enables direct interactions between the task planner, and the knowledge base through creating a planning domain or extended planning domain with predicates that specify domain features, including semantically equivalent or similar predicates. By enabling the Albert to discover and use semantically equivalent or similar entities, the module can extend the system's knowledge and provide more options to solve tasks. This can be particularly useful in scenarios where the robot faces unexpected conditions or errors that require it to adapt its plan to complete its tasks successfully. Therefore SRRM can improve our program's flexibility and adaptability. [3]

REFERENCES

[1] Michael Cashmore et al. "Rosplan: Planning in the robot operating system". In: *Proceedings of the international conference on automated planning and scheduling*. Vol. 25. 2015, pp. 333–341.

[2] Amanda Coles et al. "Forward-chaining partial-order planning". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 20. 2010, pp. 42–49.

[3] Al-Moadhen et al. "Improving the efficiency of robot task planning by automatically integrating its planner and common-sense knowledge base." In: *Knowledge-Based Information Systems in Practice* (2015), pp. 185–199.

IV. APPENDIX

A. Reproducing This Work

All changes have been pushed to the respective gitlab repositories. To reproduce first clone all repositories and ensure that you have the 2022 and 2023 singularity versions that were made available during the course. Then follow these instructions.

1) In v23 singularity: `roslaunch albert_gazebo albert_gazebo_navigation.launch`
2) In v22 singularity: `roslaunch retail_store_planning rosplan_carry.launch`
3) In v23 singularity: `roslaunch retail_store_skills load_skills.launch`
4) Outside singularity in the retail_store_planning directory (ensure that pyswip and swipl are installed): `python3 PROLOG.py`. NOTE: You can change the to_do variable inside the python script to switch between sorting items or fulfilling orders.
5) Outside singularity (after new executor is created): `chmod +x new_executor.bash`
6) `./new_executor.bash`

B. Retail Store Simulation

The simulation for our Albert robot is based upon the retail store simulation repository provided in the KRR course. The simulation world was slightly edited to replace the multi-shelf cabinets with two-shelf cabinets to prevent the robot from colliding with shelves and to avoid picking/placing inconsistencies. Another reason for creating a new world was to get more control over waypoints in the simulation environment. With the provided simulation, we were restricted in terms of the known positions and had to resort to a lot of trial and error to obtain waypoints where the pick/place actions were feasible. With a custom world, we were aware of the exact positions of all shelves, tables, and baskets. The robot's spawn location was also modified accordingly.

This change required us to create new maps for the new world. This was done by first switching to `gmapping` for localization, then teleoperating the robot via pose goals in RViz, and finally saving the newly created map using the ROS map server `$ rosrun map_server map_saver -f 'file name'`. The map server creates both the map .yaml and .pgm files. These files were modified in GIMP (GNU image manipulation program) to create a keepout map. Further, the new maps were cropped for better alignment (this required changing the origin in the map .yaml files). The localization method was switched to `amcl` after creating new maps. The maps and the RViz sensor visualizations are shown in *Figure 8*.

C. Prolog + PDDL via ROSPlan

We use the Python Prolog library cally PySwip (https://github.com/yuce/pyswip) to interface SWI-Prolog through Python. This returns a set of locations for each object in the store. These locations are then stored as object - location pairs. We then automatically create a ROSPlan KnowledgeUpdateService (which internally contains the KnowledgeItem message) message and

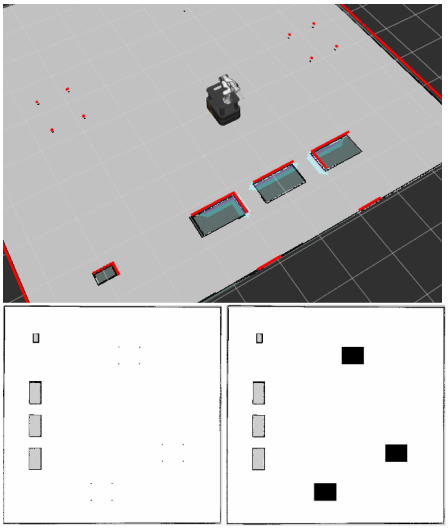

Fig. 8. Maps and visualization of the modified world

write those outputs as a service call to a new bash script using Python. The service call is a call to `/rosplan_knowledge_base/update`. Once the planner launch file is executed, ROSPlan makes a copy of the currently goal-less PDDL problem file and publishes that to the `/rosplan_problem_interface/problem_instance` topic. The first thing that the new rosplan executor does is to make the service call to add the goal (as shown in *Figure 9*). This automatically updates the problem instance and the executor continues to plan, parse, and dispatch. *Figure 10* shows a flowchart of this.

```
echo "Adding goals to the problem file"
rosservice call /rosplan_knowledge_base/update "update_type: 1
knowledge:
  knowledge_type: 1
  attribute_name: 'object_placed'
  values: [{key: 'obj', value: 'AH_thee_mango_tag_11_00030'}, {key: 'wp', value: 'wp_table_2'}]"
```

Fig. 9. Command to update goals through a service call

We realise that although this works well for the current case, it is not the best solution in terms of scalability. A better approach would have been to make these service calls though a rosprolog node or via KnowRob. This would allow much more dynamic goal updation. However, due to a shortage of time at present, this would have to be done as future work.

D. Albert's Place Skill

In our experience, Albert's place skill was quite unreliable. To improve Albert's placing, we first redefined our tasks such that products get picked from shelves and placed on tables. However, the Panda arm has a reach of just 855mm and place waypoints and 3D place poses need to be specifically curated such that the 3D place pose is within reach of the arm at the given place waypoint – this is because the two poses are decoupled and robot does not adjust its base location to enable reaching a waypoint. In future versions we would like to couple the two positions this so that the robot automatically

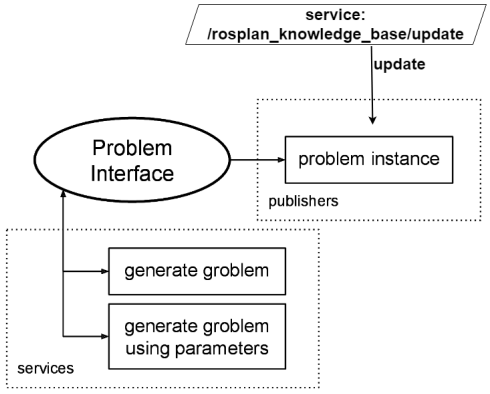

Fig. 10. Dynamic ROSPlan knowledge base updation

updates its base waypoint such that the ordered place pose is within reach of the Panda arm. This could be done in several clever ways such as calculating the closest 3D base pose using the L2 norm or pre-evaluating place poses using the keepout map. At present, however, we have solved this issue by pre-moving the panda arm to a designated place location and just dropping the object above from that pose.